

Py2HWSW

Python framework for embedded hardware/software codesign

January 7, 2025







Document Version History

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





Contents

1	Introduction	1
1.1	What Is Py2HWSW?	1
1.2	What Is Py2HWSW For?	1
1.3	What Problem Does Py2HWSW Solve?	1
1.4	What Design Principles Underlie Py2HWSW?	2
1.5	How Does Py2HWSW Accomplish Its Goals?	2
2	Getting Started	3
2.1	Setup Directory	3
2.2	Create An AND Gate Core: <code>iob_and</code>	4
2.3	Setup And Build	5
3	How It Works	6
3.1	Setup Flow Chart	6
3.2	Block hierarchy	8
3.3	Main launch script: <code>py2hws.py</code>	8
3.4	Main class for core representation: <code>iob_core.py</code>	9
3.5	Configuration class: <code>iob_conf.py</code>	10
3.6	Signal class: <code>iob_signal.py</code>	13
3.7	Wire class: <code>iob_wire.py</code>	14
3.8	Port class: <code>iob_port.py</code>	15



List of Tables



List of Figures

1	High-Level Flow Chart of Py2HWSW Setup Procedure	7
2	Block Hierarchy of a Py2HWSW Project	8





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 py2hws, a powerful tool designed to streamline the process of generating Verilog cores from high-level descriptions provided in Python or JSON dictionaries. With py2hws, 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?

Py2hws 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. Py2hws 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. Py2hws 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. Py2hws 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 compatibility 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:

```
1 # SPDX-FileCopyrightText: 2024 IObundle
2 #
3 # SPDX-License-Identifier: MIT
4
5
6 def setup(py_params_dict):
7     attributes_dict = {
8         "version": "0.1",
9         "confs": [
10             {
11                 "name": "general",
12                 "descr": "General group of confs",
13                 "confs": [
14                     {
15                         "name": "W",
16                         "type": "P",
17                         "val": "21",
18                         "min": "1",
19                         "max": "32",
20                         "descr": "IO width",
21                     },
22                 ],
23             },
24         ],
25         "ports": [
26             {
27                 "name": "a_i",
28                 "descr": "Input port",
29                 "signals": [
30                     {"name": "a_i", "width": "W"},
31                 ],
32             },
33             {
34                 "name": "b_i",
35                 "descr": "Input port",
36                 "signals": [
37                     {"name": "b_i", "width": "W"},
38                 ],
39             },
40             {
```

```

41         "name": "y_o",
42         "descr": "Output port",
43         "signals": [
44             {"name": "y_o", "width": "W"},
45         ],
46     },
47 ],
48 "superblocks": [
49     {
50         "name": "simulation",
51         "descr": "Blocks for simulation",
52         "blocks": [
53             # Simulation wrapper
54             {
55                 "core_name": "iob_sim",
56                 "instance_name": "iob_sim",
57                 "dest_dir": "hardware/simulation/src",
58             },
59         ],
60     },
61 ],
62 "snippets": [{"verilog_code": "    assign y_o = a_i & b_i;"}],
63 }
64
65 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:

```

1 $ git clone --recursive git@github.com:IObundle/py2hwsw.git
2 $ cd py2hwsw/
3 $ nix-shell py2hwsw/lib/ # Optional step to install environment with
   necessary dependencies
4 $ py2hwsw iob_and setup --no_verilog_lint

```

To do a clean setup:

```

1 $ py2hwsw iob_and clean
2 $ 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:

```

1 $ 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.

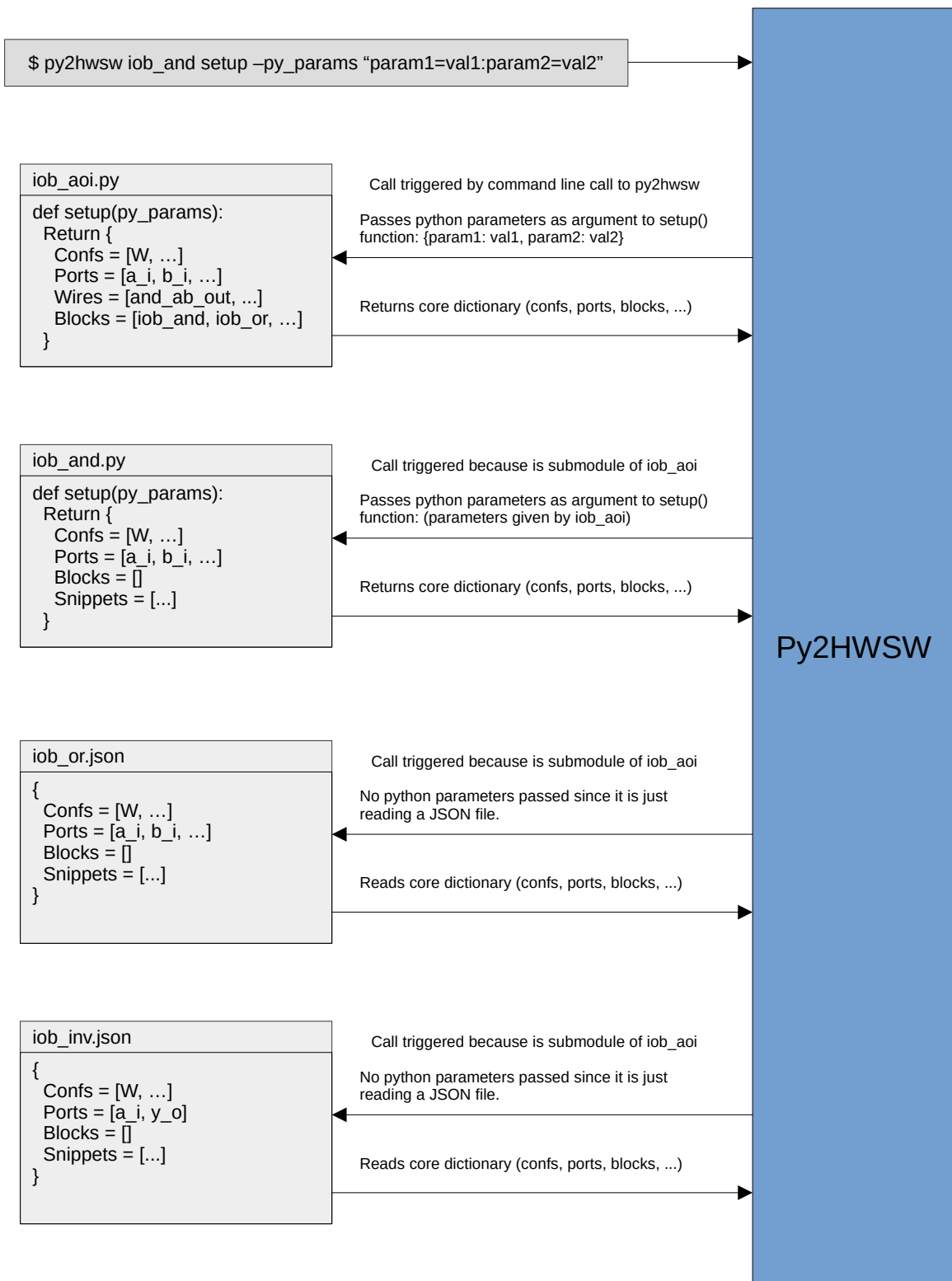


Figure 1: High-Level Flow Chart of Py2HWSW Setup Procedure

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.

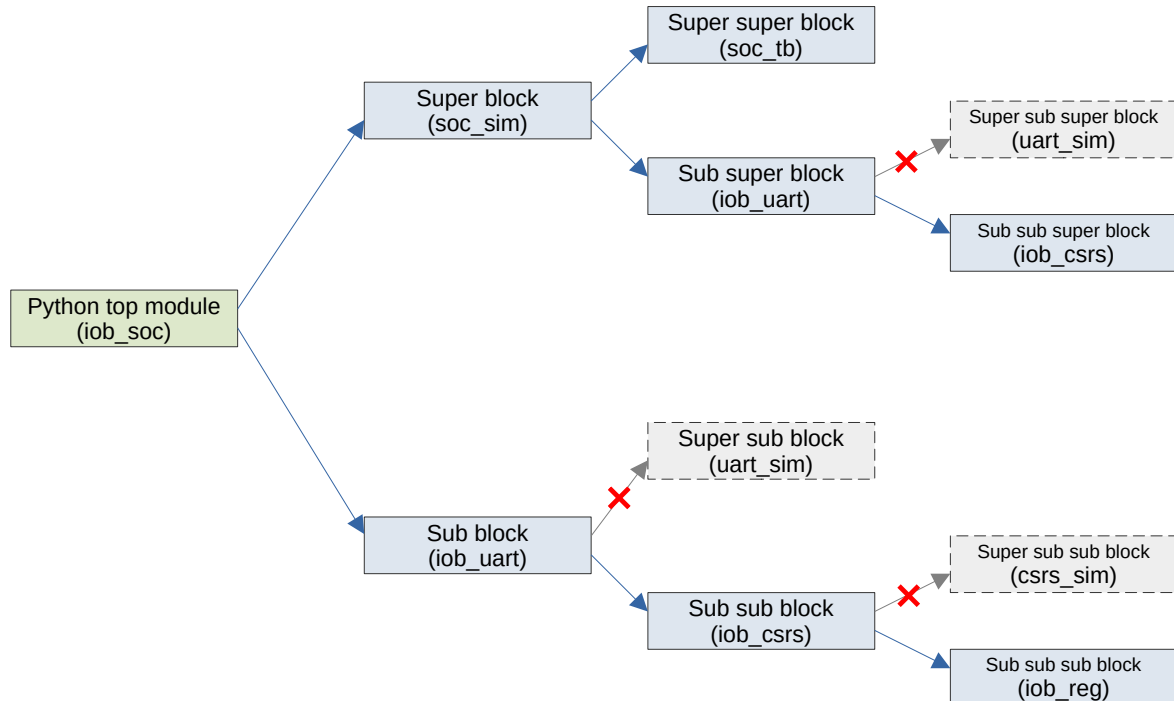


Figure 2: Block Hierarchy of a Py2HWSW Project

3.3 Main launch script: py2hws.py

The main launch script for the Py2HWSW program is the 'py2hws.py' script.

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

```

1  iob_core.global_build_dir = args.build_dir
2  iob_core.global_project_root = args.project_root
3  iob_core.global_project_vformat = args.verilog_format
4  iob_core.global_project_vlint = args.verilog_lint
5  iob_core.global_clang_format_rules_filepath = args.clang_rules
6  iob_base.debug_level = args.debug_level
7
8  if args.py2hws_docs:
9      iob_core.setup_py2_docs(PY2HWSW_VERSION)

```



```

10     exit(0)
11
12     if not args.core_name:
13         parser.print_usage(sys.stderr)
14         fail_with_msg("Core name is required.")
15
16     py_params = {}
17     if args.py_params:
18         for param in args.py_params.split(":"):
19             k, v = param.split("=")
20             py_params[k] = v
21
22     if args.target == "setup":
23         iob_core.get_core_obj(args.core_name, **py_params)
24     elif args.target == "clean":
25         iob_core.clean_build_dir(args.core_name)
26     elif args.target == "print_build_dir":
27         iob_core.print_build_dir(args.core_name, **py_params)
28     elif args.target == "print_core_dict":
29         iob_core.print_core_dict(args.core_name, **py_params)
30     elif args.target == "print_py2hwsw_attributes":
31         iob_core.print_py2hwsw_attributes(args.core_name, **py_params)
32     else:
33         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`.

```

1  """Generic class to describe how to generate a base IOB IP core"""

```

[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.

```

1      )
2      os.system(f"make -C {module.build_dir} clean")
3      shutil.rmtree(module.build_dir)
4      print(
5          f"{iob_colors.INFO}Cleaning complete. Removed: {module.build_dir}"
6          f"{iob_colors.ENDC}"
7      )
8
9      @staticmethod
10     def print_build_dir(core_name, **kwargs):
11         """Print build directory."""
12         # Set project wide special target (will prevent normal setup)

```

```

12     __class__.global_special_target = "print_build_dir"
13     # Build a new module instance, to obtain its attributes
14     module = __class__.get_core_obj(core_name, **kwargs)
15     print(module.build_dir)
16
17     @staticmethod
18     def print_core_dict(core_name, **kwargs):
19         """Print core attributes dictionary."""
20         # Set project wide special target (will prevent normal setup)
21         __class__.global_special_target = "print_core_dict"
22         # Build a new module instance, to obtain its attributes
23         module = __class__.get_core_obj(core_name, **kwargs)
24         print(json.dumps(module.attributes_dict, indent=4))
25
26     @staticmethod
27     def print_py2hwsw_attributes(core_name, **kwargs):
28         """Print the supported py2hw attributes of this core.
29         The attributes listed can be used in the 'attributes' dictionary of
30         the
31         constructor. This defines the information supported by the py2hw
32         interface.
33         """
34         # Set project wide special target (will prevent normal setup)
35         __class__.global_special_target = "print_attributes"
36         # Build a new module instance, to obtain its attributes
37         module = __class__.get_core_obj(core_name, **kwargs)
38         print(
39             f"Attributes supported by the '{module.name}' core's 'py2hwsw'
40             interface:"
41         )
42         for name in module.ATTRIBUTE_PROPERTIES.keys():
43             datatype = module.ATTRIBUTE_PROPERTIES[name].datatype
44             descr = module.ATTRIBUTE_PROPERTIES[name].descr
45             align_spaces = " " * (20 - len(name))
46             align_spaces2 = " " * (18 - len(str(datatype)))
47             print(f"- {name}:{align_spaces}{datatype}{align_spaces2}{descr}"
48                 )
49
50     @staticmethod
51     def get_core_obj(core_name, **kwargs):
52         """Generate an instance of a core based on given core_name and
53         python parameters
54         This method will search for the .py and .json files of the core, and
55         generate a

```

[View Source](#)

3.5 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.

```

1 class iob_conf:

```

```

2  """Class to represent a configuration option."""
3
4  # Identifier name for the configuration option.
5  name: str = ""
6  # Type of configuration option, either M (Verilog macro), P (Verilog
7  # parameter) or F (Verilog false-parameter).
8  # False-parameters are the same as verilog parameters except that the
9  # its value must not be overridden.
10 type: str = ""
11 # Value of the configuration option.
12 val: str | int | bool = ""
13 # Minimum value supported by the configuration option (NA if not
14 # applicable).
15 min: str | int = "NA"
16 # Maximum value supported by the configuration option (NA if not
17 # applicable).
18 max: str | int = "NA"
19 # Description of the configuration option.
20 descr: str = "Default description"
21 # Only applicable to Verilog macros: Conditionally enable this
22 # configuration if the specified Verilog macro is defined/undefined.
23 if_defined: str = ""
24 if_not_defined: str = ""
25 # If enabled, configuration option will only appear in documentation.
26 # Not in the verilog code.
27 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.

```

1 class iob_conf_group:
2     """Class to represent a group of configurations."""
3
4     # Identifier name for the group of configurations.
5     name: str = ""
6     # Description of the configuration group.
7     descr: str = "Default description"
8     # List of configuration objects.
9     confs: list = field(default_factory=list)
10    # If enabled, configuration group will only appear in documentation. Not
11    # in the verilog code.
12    doc_only: bool = False
13    # If enabled, the documentation table for this group will be terminated
14    # by a TeX '\clearpage' command.
15    doc_clearpage: bool = False

```

[View Source](#)

The py2hws 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.

```
1  for group in macros:
2      # If group has 'doc_only' attribute set to True, skip it
3      if group.doc_only:
4          continue
5      for macro in group.confes:
6          # If macro has 'doc_only' attribute set to True, skip it
7          if macro.doc_only:
8              continue
9          if macro.if_defined:
10             file2create.write(f"{'ifdef {macro.if_defined}}\n")
11             if macro.if_not_defined:
12                 file2create.write(f"{'ifndef {macro.if_not_defined}}\n")
13             # Only insert macro if its is not a bool define, and if so only
14             # insert it if it is true
15             if type(macro.val) is not bool:
16                 m_name = macro.name.upper()
17                 m_default_val = macro.val
18                 file2create.write(f"{'define {core_prefix}{m_name} {m_default_val}}\n")
19             elif macro.val:
20                 m_name = macro.name.upper()
21                 file2create.write(f"{'define {core_prefix}{m_name} 1}\n")
22             if macro.if_defined or macro.if_not_defined:
23                 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 'job_conf' class.

```
1  lines = []
2  core_prefix = f"{'{core.name}_'.upper()}"
3  for idx, parameter in enumerate(core_parameters):
4      # If parameter has 'doc_only' attribute set to True, skip it
5      if parameter.doc_only:
6          continue
7
8      p_name = parameter.name.upper()
9      p_comment = ""
10     if parameter.type == "F":
11         p_comment = " // Don't change this parameter value!"
12     lines.append(f"{'parameter {p_name} = '{core_prefix}{p_name},{p_comment}}\n")
13
14     # Remove comma from last line
15     if lines:
16         lines[-1] = lines[-1].replace(", ", "", 1)
```

[View Source](#)

3.6 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.

```

1 class iob_signal:
2     """Class that represents a wire/port signal"""
3
4     # Identifier name for the signal.
5     name: str = ""
6     # Number of bits in the signal.
7     width: str or int = 1
8     # Description of the signal.
9     descr: str = "Default description"
10    # If enabled, signal will be generated with type 'reg' in Verilog.
11    isvar: bool = False
12
13    # Used for 'iob_comb': If enabled, iob_comb will infer a register for
14    # this signal.
15    isreg: bool = False
16    # Used for 'iob_comb': List of signals associated to the inferred
17    # register.
18    reg_signals: list[str] = field(default_factory=list)
19
20    # Logic value for future simulation effort using global signals list.
21    # See 'TODO' in iob_core.py for more info: https://github.com/IObundle/
22    # py2hwsw/blob/a1e2e2ee12ca6e6ad81cc2f8f0f1c1d585aaee73/py2hwsw/scripts
23    # /iob_core.py#L251-L259
24    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.

```

1 def get_verilog_wire(self):
2     """Generate a verilog wire string from this signal"""
3     wire_type = "reg" if self.isvar or self.isreg else "wire"
4     width_str = "" if self.get_width_int() == 1 else f"[{self.width
5     }-1:0] "
6     return f"{wire_type} {width_str}{self.name};\n"
7
8 def get_verilog_port(self, comma=True):
9     """Generate a verilog port string from this signal"""
10    self.assert_direction()
11    comma_char = "," if comma else ""
12    port_type = " reg" if self.isvar or self.isreg else ""
13    width_str = "" if self.get_width_int() == 1 else f"[{self.width
14    }-1:0] "
15    return f"{self.direction}{port_type} {width_str}{self.name}{
16    comma_char}\n"

```

[View Source](#)

3.7 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.

```
1 class iob_wire:
2     """Class to represent a wire in an iob module"""
3
4     # Identifier name for the wire.
5     name: str = ""
6     # Name of the standard interface to auto-generate with 'if_gen.py'
7     script.
8     interface: if_gen.interface = None
9     # Description of the wire.
10    descr: str = "Default description"
11    # Conditionally define this wire if the specified Verilog macro is
12    defined/undefined.
13    if_defined: str = ""
14    if_not_defined: str = ""
15    # List of signals belonging to this wire
16    # (each signal represents a hardware Verilog wire).
17    signals: List = field(default_factory=list)
```

[View Source](#)

The 'signals' attribute stores a list of signal objects, represented by the 'iob_signal' class ??.

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.

```
1 for wire in core.wires:
2     # Open ifdef if conditional interface
3     if wire.if_defined:
4         code += f"ifdef {wire.if_defined}\n"
5     if wire.if_not_defined:
6         code += f"ifndef {wire.if_not_defined}\n"
7
8     signals_code = ""
9     for signal in wire.signals:
10        if isinstance(signal, iob_signal):
11            signals_code += "    " + signal.get_verilog_wire()
12    if signals_code:
13        code += f"    // {wire.name}\n"
14        code += signals_code
15
16    # Close ifdef if conditional interface
17    if wire.if_defined or wire.if_not_defined:
18        code += "\nendif\n"
```

[View Source](#)

3.8 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. 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.

```

1 class iob_port(iob_wire):
2     """Describes an IO port."""
3
4     # External wire that connects this port
5     e_connect: iob_wire | None = None
6     # Dictionary of bit slices for external connections. Name: signal name;
7     # Value: bit slice
8     e_connect_bit_slices: list = field(default_factory=list)
9     # If enabled, port will only appear in documentation. Not in the verilog
10    # code.
11    doc_only: bool = False
12    # If enabled, the documentation table for this port will be terminated
13    # by a TeX '\clearpage' command.
14    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 ??.

The py2hws 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.

```

1 for port_idx, port in enumerate(core.ports):
2     # If port has 'doc_only' attribute set to True, skip it
3     if port.doc_only:
4         continue
5
6     # Open ifdef if conditional interface
7     if port.if_defined:
8         lines.append(f"{'ifdef {port.if_defined}}\n")
9     if port.if_not_defined:
10        lines.append(f"{'ifndef {port.if_not_defined}}\n")
11
12    lines.append(f"    // {port.name}\n")
13
14    for signal_idx, signal in enumerate(port.signals):
15        lines.append("        " + signal.get_verilog_port())
16
17    # Close ifdef if conditional interface
18    if port.if_defined or port.if_not_defined:
19        lines.append("{'endif\n")
20
21    # Remove comma from last port line
22    if lines:
23        i = -1
24        while lines[i].startswith("{'endif"):
25            i -= 1

```



```
26 lines[i] = lines[i].replace(", ", "", 1)
```

[View Source](#)