Antmicro

Topwrap

2026-11-09

DOCUMENTATION

1 Introduction to Topwrap **1**

2 Installing Topwrap **2**
2.1 1. Install required system packages . . . 2
2.2 2. Install the Topwrap user package . . . 2
2.3 3. Verify the installation . . . 3

3 Getting started **4**
3.1 Design overview . . . 4
3.2 Packaging Verilog sources . . . 5
3.3 Building designs with Topwrap . . . 5
3.4 Command-line flow . . . 7
3.5 Using libraries . . . 10

4 Advanced options **11**
4.1 Creating block designs in the GUI . . . 11

5 Sample projects **15**
5.1 Embedded GUI . . . 15
5.2 Constant . . . 15
5.3 Inout . . . 16
5.4 Hierarchy . . . 19
5.5 PWM . . . 20
5.6 HDMI . . . 21
5.7 SoC . . . 22
5.8 Memory Map . . . 24
5.9 AXI interconnect . . . 25

6 Creating a design **27**
6.1 Design description . . . 27
6.2 IP description files . . . 31
6.3 Interface definition files . . . 40
6.4 Resource path syntax . . . 42
6.5 Positions YAML . . . 44

7 Interface inference **47**

8 Configuration **49**
8.1 Configuration file location . . . 49
8.2 Configuration precedence . . . 49

8.3 Available config options . . . 50

9 Libraries **52**
9.1 Registering a library . . . 52
9.2 Inspecting libraries . . . 52
9.3 Describing library contents . . . 53
9.4 Referencing a library module . . . 53
9.5 Declaring libraries in a design . . . 54
9.6 Creating a library core . . . 54

10 Constructing, configuring and loading repositories **56**
10.1 Supported resource types . . . 56
10.2 CLI . . . 57
10.3 Using the open source IP cores library with Topwrap . . . 58

11 Interconnect generation **59**
11.1 Format . . . 60
11.2 Supported interconnect types . . . 61

12 Using FuseSoC for automation **63**
12.1 Default tool for synthesis, bitstream generation and programming the FPGA . . . 63
12.2 Additional build options . . . 63
12.3 Synthesis . . . 64

13 Plugin Support **65**
13.1 Ways to get plugins . . . 65
13.2 Plugin configurations . . . 65

14 Setup **66**

15 Code style **67**
15.1 Lint with pre-commit . . . 67
15.2 Tools . . . 67

16 Tests **68**
16.1 Test execution . . . 68
16.2 Test coverage . . . 69
16.3 Updating kpm test data . . . 69

17 Internal Representation **70**
17.1 Class diagram . . . 70
17.2 Frontend & Backend . . . 70
17.3 Module . . . 73
17.4 Design . . . 74
17.5 Interface . . . 76
17.6 Interface inference . . . 79
17.7 Connections . . . 81
17.8 HDL Types . . . 83
17.9 Interconnects . . . 86
17.10 Miscellaneous . . . 88
17.11 A note on “sliced” vs. “independent” signals . . . 90

18 Frontend **92**

18.1 SystemVerilogFrontend . . . 92
18.2 KpmFrontend . . . 92
18.3 YamlFrontend . . . 92
18.4 IpXactFrontend . . . 93

19 Backend **94**
19.1 SystemVerilogBackend . . . 94
19.2 KpmBackend . . . 94
19.3 YAML backends . . . 94
19.4 IPXACTBackend . . . 94

20 FuseSocBuilder **96**
20.1 Customizations . . . 97

21 Interface Definition **100**

22 Config **101**

23 Validation of design **102**
23.1 Tests for validation checks . . . 104

24 Future planned enhancements in Topwrap **112**
24.1 Library of open-source cores . . . 112
24.2 Support for hierarchical block designs in Topwrap's GUI . . . 112
24.3 Support for parsing SystemVerilog sources . . . 112

25 Other possible improvements **113**
25.1 Ability to produce top-level wrappers in VHDL . . . 113
25.2 Bus management . . . 113
25.3 Improve the process of recreating a design from a YAML file . . . 113
25.4 Deeper integration with other tools . . . 114
25.5 Provide a way to parse HDL sources from the GUI level . . . 114

26 Using KPM iframes inside docs **115**
26.1 Usage . . . 115
26.2 Tests . . . 116

27 Examples for Internal Representation **118**
27.1 Simple . . . 118
27.2 Interface . . . 119
27.3 Hierarchical . . . 119
27.4 Interconnect . . . 120
27.5 Advanced . . . 121
27.6 SoC . . . 122
27.7 SoC_AXI . . . 122
27.8 Inverted . . . 123
27.9 Clocks . . . 124
27.10 Complex hierarchy parsing . . . 124
27.11 Other . . . 125

28 IP-XACT format **126**
28.1 General observations . . . 126
28.2 Simple example . . . 130

28.3 Interface example . . . 132
28.4 Hierarchical example . . . 136
28.5 Interconnect example . . . 136
28.6 Other features . . . 138
28.7 Conclusion . . . 138

29 Generator **139**
29.1 How to implement a Generator . . . 140
29.2 Lookup maps . . . 140
29.3 API Reference . . . 140

30 Interconnect **143**

31 Repository **144**
31.1 API reference . . . 144

32 Logging **147**
32.1 Using a Custom Logging Configuration . . . 147
32.2 Creating a New Logger . . . 148

33 Plugins **150**
33.1 Examples . . . 150
33.2 Plugin Base . . . 150
33.3 Build Context . . . 151

34 Caliptra UART **153**

35 VeeR EL2 **154**

36 HMAC **155**

37 SHA256 **156**

38 SHA512 **157**

39 SHA3 **158**

Python Module Index **159**

Index **160**

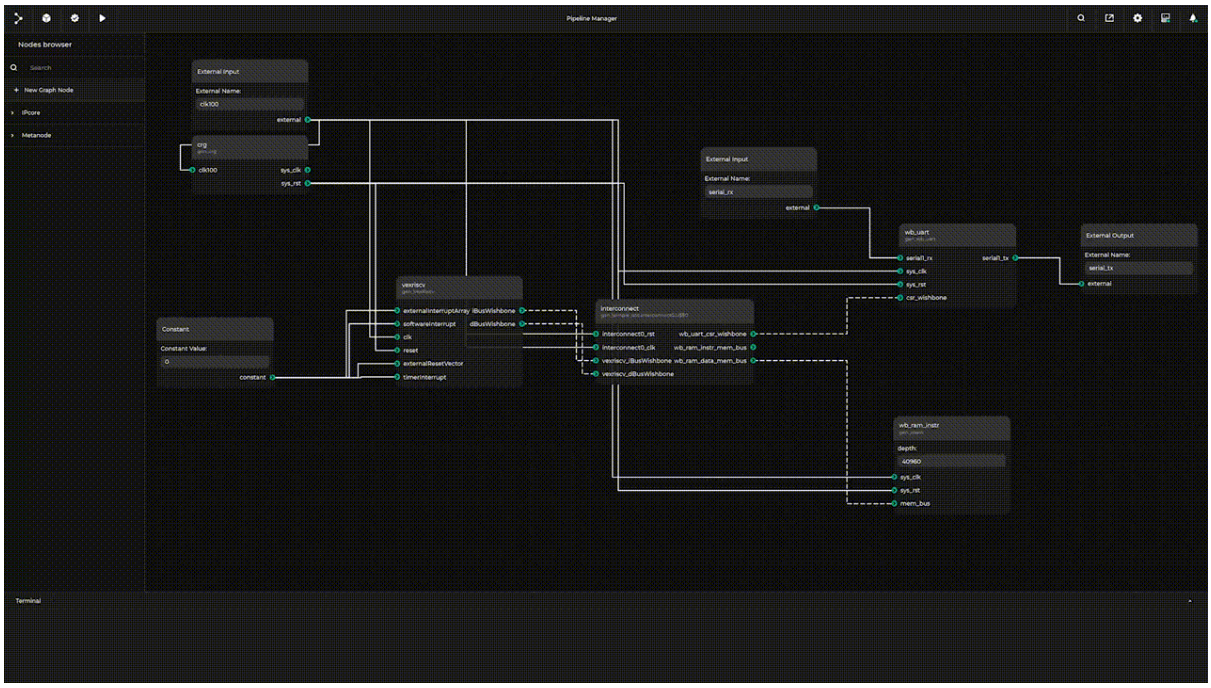
CHAPTER

ONE

INTRODUCTION TO TOPWRAP

Topwrap leverages modularity to enable the reuse of design blocks across different projects, facilitating the transition to automated logic design. It provides a standardized approach for organizing blocks into various configurations, making top-level designs easier to parse and process automatically.

As a tool, Topwrap makes it *straightforward to build* complex and *synthesizable designs* by generating a design file in formats such as SystemVerilog or *IP-XACT 2022*. The combination of *GUI and CLI-based* configuration options provides for fine-tuning possibilities. Packaging multiple files is accomplished by including them in a *custom user repository*, and an internal API enables repository creation using Python.



CHAPTER

TWO

INSTALLING TOPWRAP

2.1 1. Install required system packages

> **Caution:** Topwrap has been tested on Debian Bullseye and Bookworm. While other distributions may also be compatible, Bullseye and Bookworm reflect environments where functionality has been verified.

On Debian and Debian-based distributions, follow these steps to install the required dependencies:

Install required dependencies

```
apt install -y python3 python3-pip yosys npm
```

And uv if you want to use it to install Topwrap:

```
curl -LsSf https://astral.sh/uv/install.sh | sh
```

For more details see uv installation page

2.2 2. Install the Topwrap user package

Recommended: Use uv to directly install Topwrap as a user package:

```
uv tool install "topwrap @ git+https://github.com/antmicro/topwrap"
```

If you can't use uv, you can use regular pip instead. It may be necessary to do it in a Python virtual environment, such as venv:

```
python3 -m venv venv
source venv/bin/activate
pip install "topwrap@git+https://github.com/antmicro/topwrap"
```

2.3 3. Verify the installation

Make sure that Topwrap was installed correctly and is available in your shell:

```
topwrap --help
```

This should print out the help string with Topwrap subcommands listed out.

See also:

If you want to contribute to the project, check the *Developer's setup guide* for more information.

CHAPTER

THREE

GETTING STARTED

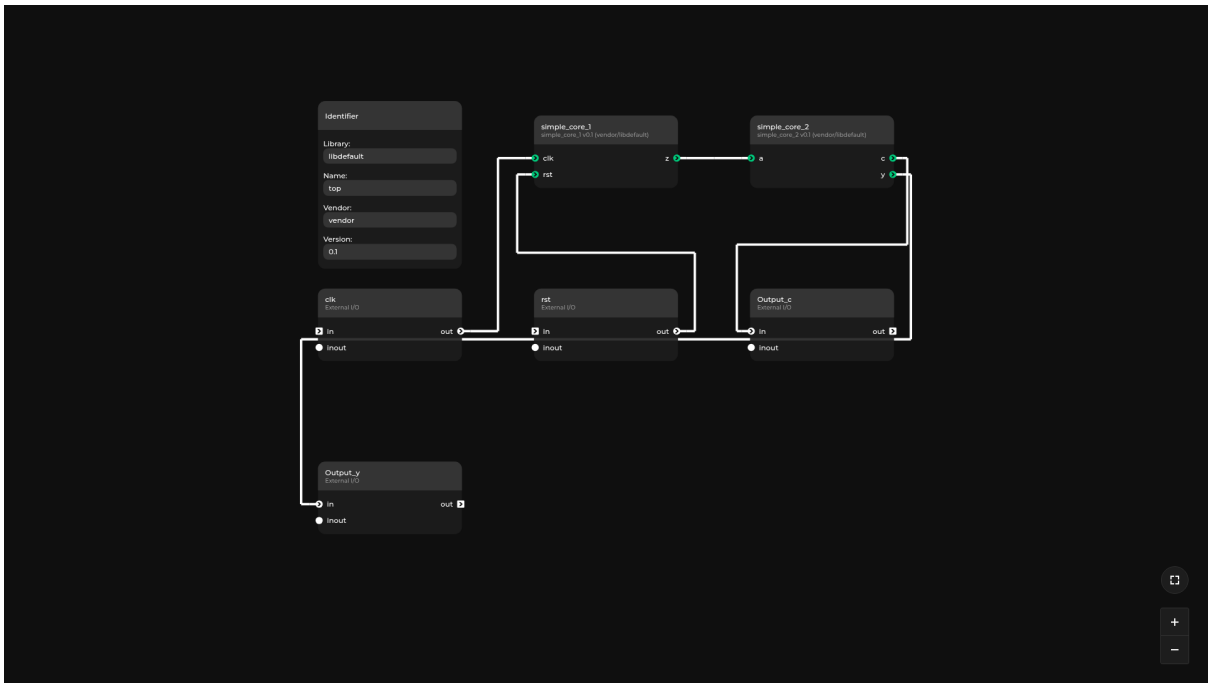
The purpose of this chapter is to provide a step by step guide on how to create a simple design with Topwrap. All the necessary files needed to follow this guide are in the examples/getting_started_demo directory.

Important

If you haven't installed Topwrap yet, go to the *Installation chapter*.

3.1 Design overview

We are going to create a design that will be visually represented in an *interactive GUI*, as seen below.



It consists of two cores: `simple_core_1` and `simple_core_2` that connect to each other and to an external input/output.

Note: Metanodes are always utilized in designs to represent external input/output ports,

module hierarchy ports or constant values. They can be found in the “Metanode” section.

3.2 Packaging Verilog sources

The `cores` directory contains two Verilog files, `simple_core_1.v` and `simple_core_2.v`. Topwrap can automatically scan them and create the necessary IP description files using `topwrap package`:

```
topwrap package cores/simple_core_1.v --output cores
topwrap package cores/simple_core_2.v --output cores
```

This is the most basic way of using topwrap, and we can now start building systems.

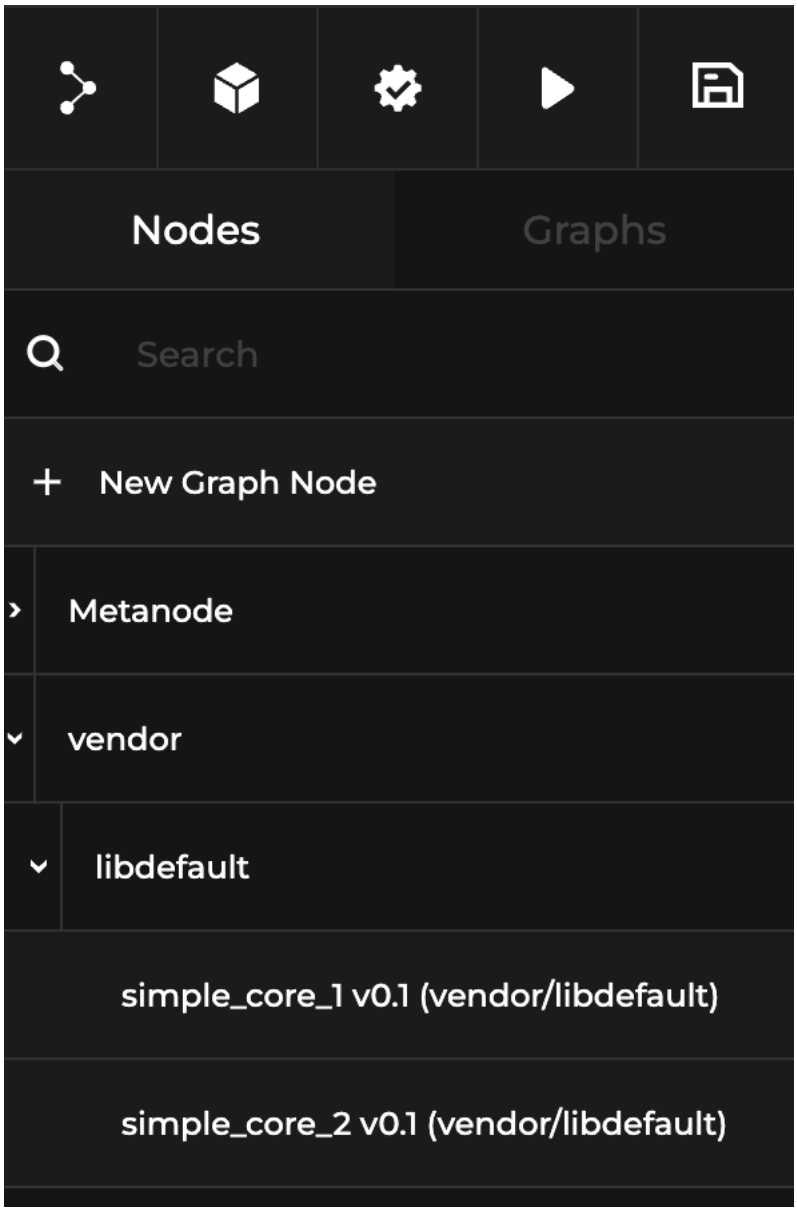
3.3 Building designs with Topwrap

3.3.1 Creating the design

The previously parsed source files can be loaded into the GUI with:

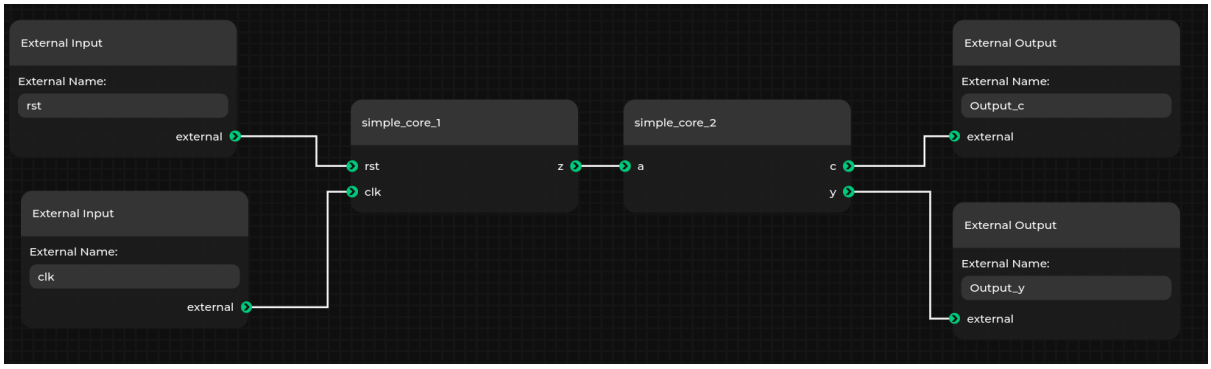
```
topwrap gui cores/simple_core_1.yaml cores/simple_core_2.yaml
```

The loaded IP cores can be found in the IPcore section:



With these IP cores and default metanodes, you can easily create designs by dragging and connecting cores.

Let's make the design from the demo in the *introduction*.



Note: You can change the name of an individual node by right clicking on it and selecting

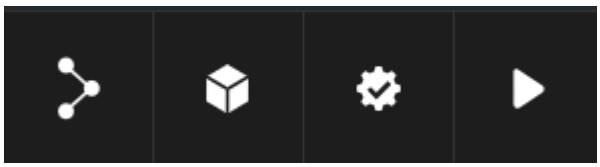
`rename`. This is useful when creating multiple instances of the same IP core.

You can save the design using the `Save graph file` option from the menu. This will create a graph JSON format file used by the GUI. This is essentially a snapshot of everything that you created in the editor and can be loaded back later.

Alternatively, you can save the design into a *design description* YAML file by using the `Save file` option from the menu. When using this option, in addition to downloading the design YAML file, Topwrap will also create an additional *positions YAML* file describing the node positions. This file, along with a copy of the design YAML, will be saved in a new subdirectory in the directory `topwrap gui` was run from. The positions YAML file will be referenced from the saved design YAML file (via the `positions` property), and Topwrap will automatically attempt to load it next time you open the design.

3.3.2 Generating Verilog in the GUI

You can generate Verilog from the design created in the previous section if you have the example running as described in the previous section. On the top bar, these four buttons are visible:



1. Save/Load designs.
2. Toggle the node browser.
3. Validate the design.
4. Build the design. If it does not contain errors, a top module will be created in the directory where `topwrap gui` was run.

3.4 Command-line flow

3.4.1 Creating designs

The manual creation of designs requires familiarity with the *Design Description* format.

First, include all the IP core files needed in the `ips` section.

```
ips:
  simple_core_1:
    file: cores/simple_core_1.yaml
  simple_core_2:
    file: cores/simple_core_2.yaml
```

Now we can start creating the design by defining connections in the `connections.ports` section. In the demo example, there is only one connection - between `simple_core_1` and `simple_core_2`.

In our design, it is represented like this:

```
connections:
  ports:
    simple_core_2:
      a: [simple_core_1, z]
```

All that is left to do is to declare external ports, like this:

```
external:
  ports:
    in:
      - rst
      - clk
    out:
      - Output_y
      - Output_c
```

Now connect them to IP cores.

```
connections:
  ports:
    simple_core_1:
      clk: clk
      rst: rst
    simple_core_2:
      a: [simple_core_1, z]
      c: Output_c
      y: Output_y
```

The final design:

```
ips:
  simple_core_1:
    file: cores/simple_core_1.yaml
  simple_core_2:
    file: cores/simple_core_2.yaml
connections:
  ports:
    simple_core_1:
      clk: clk
      rst: rst
    simple_core_2:
      a: [simple_core_1, z]
      c: Output_c
      y: Output_y
external:
  ports:
    in:
      - rst
      - clk
    out:
      - Output_y
```

(continues on next page)

(continued from previous page)

```
    - Output_c
```

3.4.2 Generating Verilog top files

To generate the top file, use `topwrap generate` and provide the design. To do this, ensure you are in the `examples/getting_started_demo` directory and run:

```
topwrap generate {design_name.yaml}
```

Where `{design_name.yaml}` is the design saved at the end of the previous section. This generates a `top.sv` SystemVerilog top wrapper in the target directory (`./build/src` by default).

3.4.3 Generating IPXACT 2022 files

To generate IP-XACT 2022 files from a design, use `topwrap generate --ipxact`:

```
topwrap generate --ipxact {design_name.yaml}
```

By default the generated files are written to the `build/src` directory. Change the parent directory with the `--target-dir` argument. Use `--iface-compliance` to force interface compliance checking.

See *Backends* for details on the backend driving this command.

3.4.4 Synthesis & FuseSoC

You can additionally generate a *FuseSoC core* file during `topwrap generate` to automate further synthesis and implementation by adding `--fusesoc`. The command also accepts `--part` for the FPGA part and repeatable `--sources` directories containing additional HDL or constraint sources.

3.4.5 Logging

Topwrap uses Python's built-in `logging` module. By default, logging is configured with the `INFO` level. The default can be changed using the `--log-level` option:

```
topwrap --log-level DEBUG generate {design_name.yaml}
```

3.5 Using libraries

So far, the IP cores in this guide were referenced directly by their YAML files with `file:`. Once you have cores you want to reuse across multiple designs or projects, it is more convenient to register them as a library and reference them by name with `core:` instead.

A library is a directory (or git repository) of CAPI2 `.core` files, as we are leveraging the FuseSoC package management infrastructure. To also add those extra files, just add `--lib` to `topwrap package`:

```
topwrap package --lib cores/simple_core_1.v --output cores
topwrap package --lib cores/simple_core_2.v --output cores
```

This writes `simple_core_1.core` and `simple_core_2.core` next to the YAML files, which is all that's needed to register the directory as a library:

```
topwrap library add mycores cores/
```

You can check what the registered library provides with:

```
topwrap library list all
```

With the library registered, a design can reference a module by name instead of a file path:

```
ips:
  simple_core_1:
    core: simple_core_1
  simple_core_2:
    core: simple_core_2
```

The main advantage of libraries is sharing: since a library can be a git repository, a set of cores can be published once and reused across as many projects as needed, simply by registering the same URL wherever it is needed, instead of copying files around. In our example you would push the library into a repository and then use it:

```
topwrap library add antmicro https://github.com/antmicro.com/topwrap-cores
```

See the *Libraries* chapter for the full picture, including versioning and interface definitions.

CHAPTER

FOUR

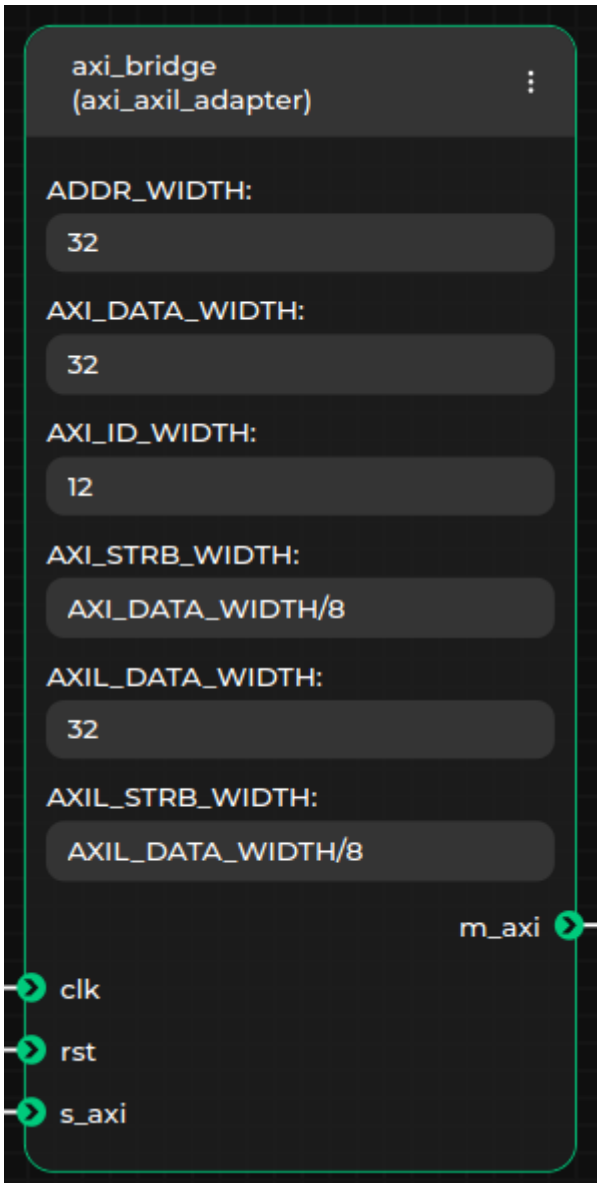
ADVANCED OPTIONS

This chapter builds upon the content covered in the *Getting started* chapter. If you have not yet reviewed it, we recommend doing so before proceeding.

4.1 Creating block designs in the GUI

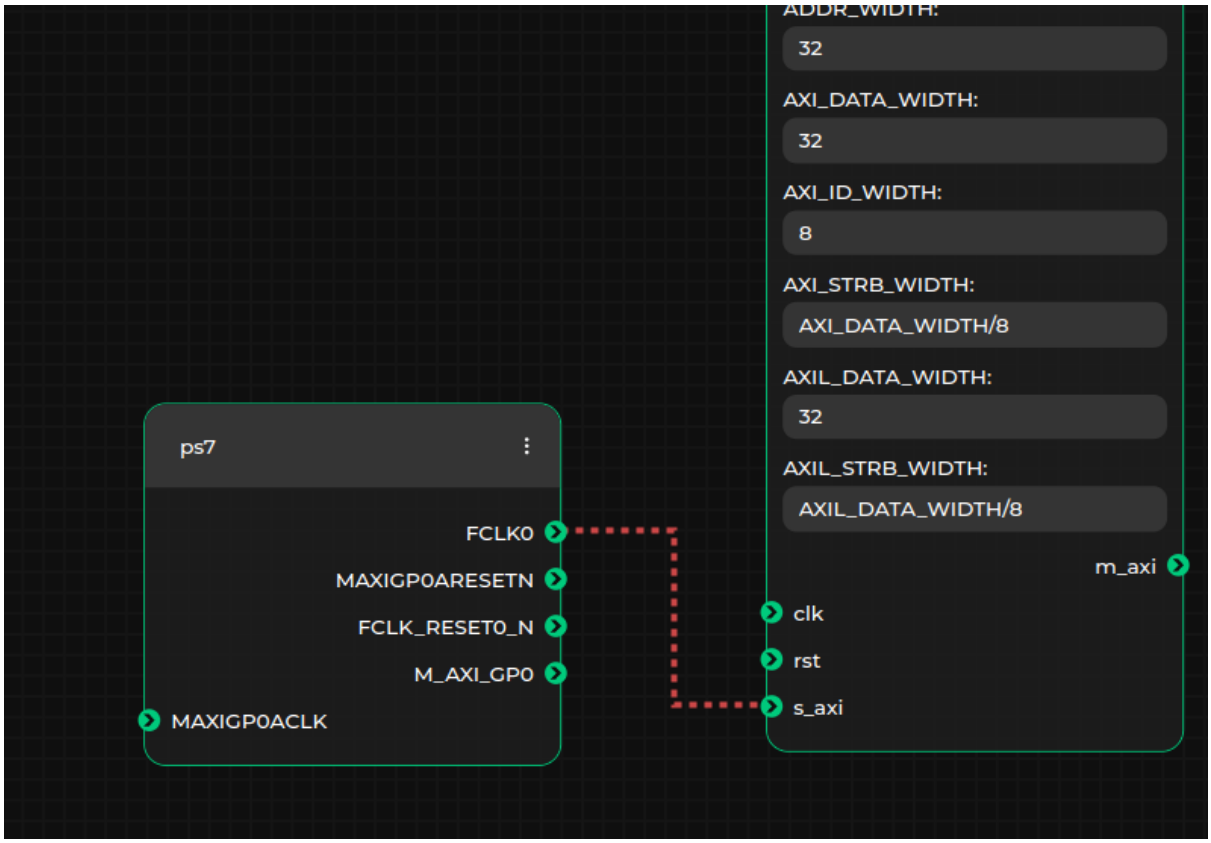
Upon successfully connecting to the server, Topwrap will generate and transmit a specification describing the structure of the selected IP cores. If the `-d` option is specified, the design will be displayed in the GUI. The following content is based on the `PWM` example located in `examples/pwm`. From this point, you can create or modify designs by:

- adjusting the parameter values of IP cores. Each node includes input fields where you can specify parameter values (default values are automatically assigned when an IP core is added to the block design):

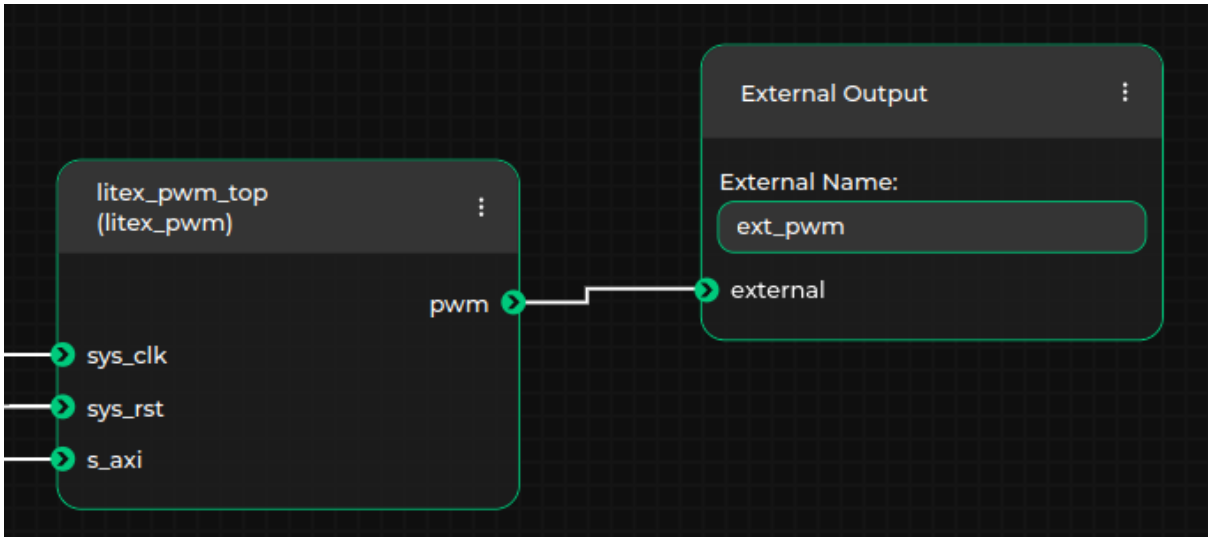


Parameter values can be specified as integers in various bases (e.g., `0x28`, `40`, or `0b101000`) or as arithmetic expressions, which will be evaluated later (e.g., `(AXI_DATA_WIDTH + 1) / 4` is a valid expression, provided a parameter named `AXI_DATA_WIDTH` exists in the same IP core). Additionally, parameter values can be written in Verilog format (e.g., `8'b00011111` or `8'h1F`), in which case they will be interpreted as fixed-width bit vectors

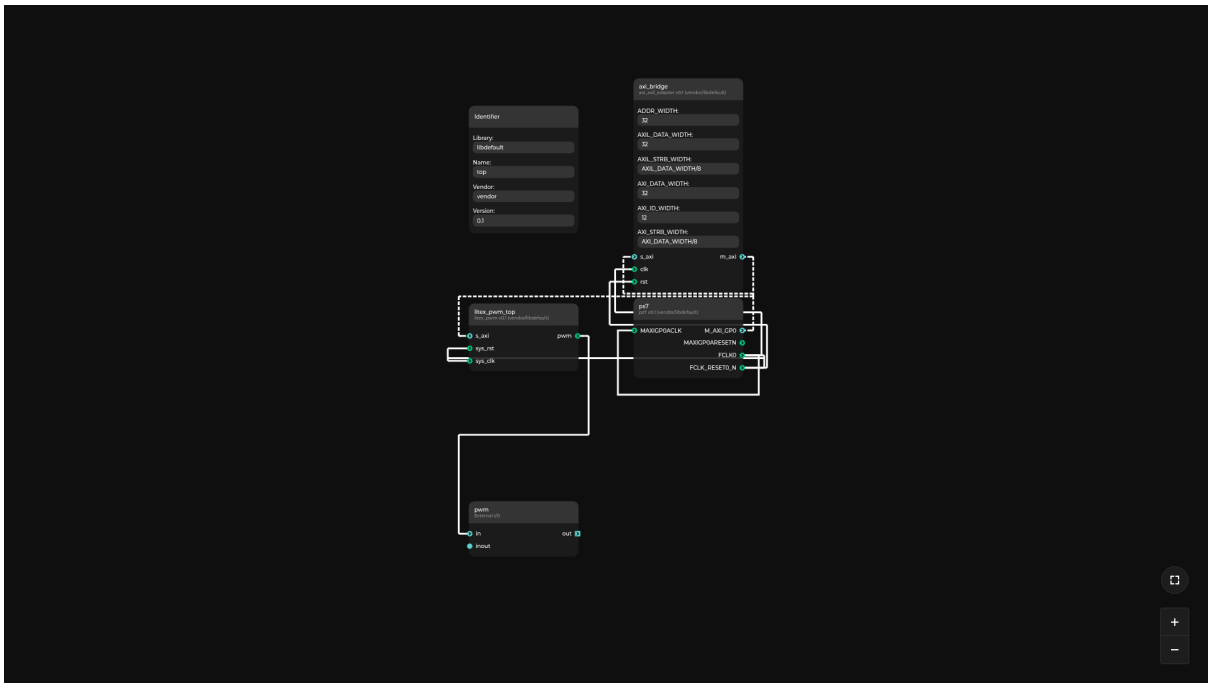
- connecting the ports and interfaces of IP cores. Only connections between ports or interfaces of matching types are allowed. This is automatically validated by the GUI, which uses the type information from the loaded specification. As a result, the GUI will prevent users from making invalid connections (e.g., connecting `AXI4` with `AXI4Lite`, or connecting a port to an interface). A green line will indicate a valid connection, while a red line will indicate an invalid one:



- specifying external ports or interfaces in the top module. To do this, add the appropriate `External Input`, `External Output`, or `External Inout` metanodes, and establish connections between these metanodes and the desired ports or interfaces. Ensure that the name of the external port or interface is updated in the textbox within the selected metanode. For example, in the case where the output pwm port of the `litex_pwm_top` IP core is external to the generated top module, the external port name should be set to `ext_pwm`, as shown below:



An example block design in the Topwrap GUI for the PWM project may look like this:



Important: With each graph change, Topwrap will save the current dataflow to ensure it's not lost, e.g. during an accidental page refresh. The file is located at `$XDG_DATA_HOME/topwrap/dataflow_latest_save.json`.

More information about this example can be found here

CHAPTER

FIVE

SAMPLE PROJECTS

These projects demonstrate how to use Topwrap on a practical level, with examples based on a variety of useful designs.

5.1 Embedded GUI

This section extensively uses an embedded version of Topwrap's GUI to visualize the design of all the examples.

You can use it to explore designs, while adding new blocks, connections, nodes and hierarchies.

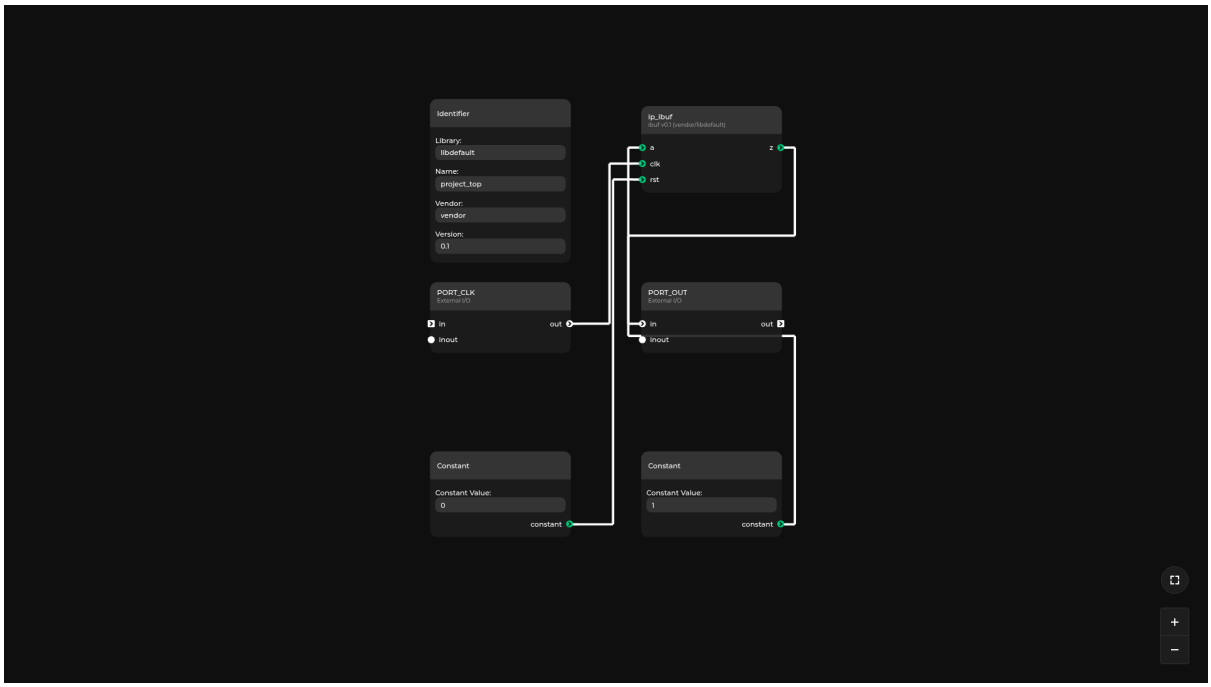
The features that require direct connection with Topwrap's backend are not implemented in this demo version, including:

- saving and loading data in `.yaml` files
- building designs
- verifying designs

Tip: Don't forget to use the "Enable fullscreen" button if the viewport is too small.

5.2 Constant

Link to source



This example shows how to assign a constant value to a port in an IP core. You can see it in the GUI by using the interactive preview functionality. It is also visible in the description file (`project.yaml`).

Tip: You can find the constant node blueprint in the nodes browser within the `Metanode` section.

5.2.1 Usage

Switch to the subdirectory with the example:

```
cd examples/constant
```

Generate the HDL source:

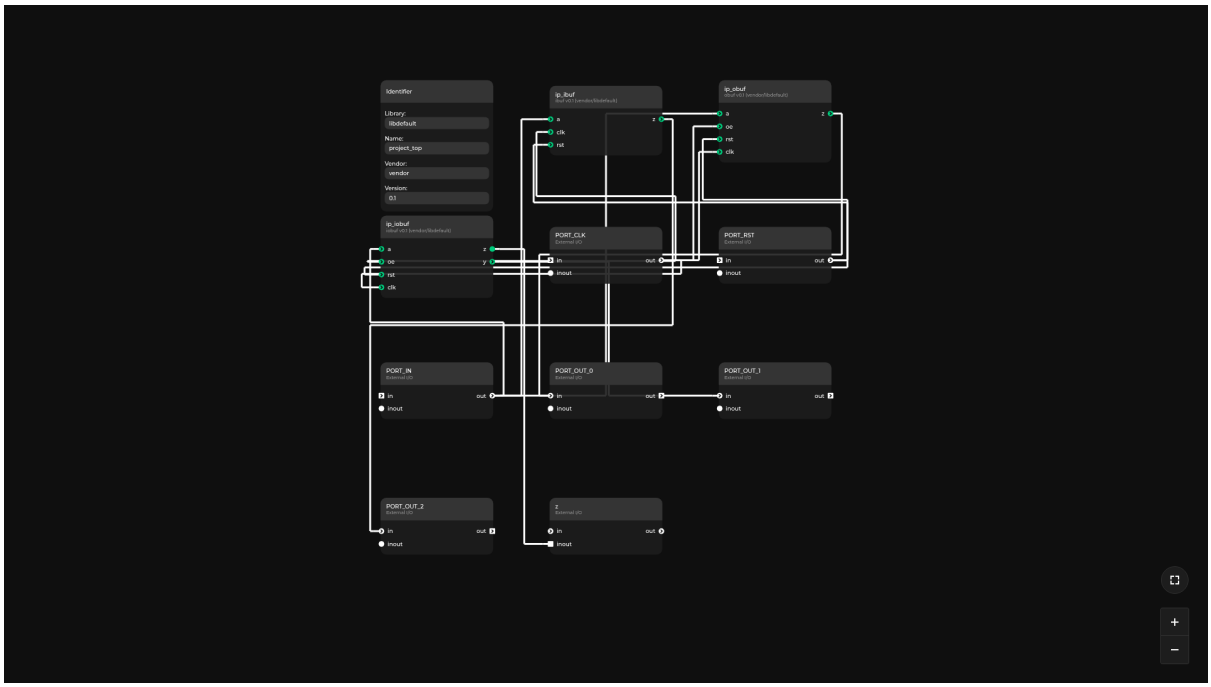
```
make generate
```

Extract an *IP description file* describing the design's top module:

```
make extract
```

5.3 Inout

Link to source



This example showcases the usage of an inout port and its representation in the GUI.

Tip: An inout port is marked in the GUI by a green circle without a directional arrow inside.

The design consists of 3 modules: input buffer `ibuf`, output buffer `obuf`, and bidirectional buffer `iobuf`. Their operation can be described as:

- the input buffer is a synchronous D-type flip flop with an asynchronous reset
- the output buffer is a synchronous D-type flip flop with an asynchronous reset and an `output enable`, which sets the output to a high impedance state (Hi-Z)
- the inout buffer instantiates 1 input and 1 output buffer. The input of the `ibuf` and output of the `obuf` are connected with an inout wire (port).

5.3.1 Usage

Switch to the subdirectory with the example:

```
cd examples/inout
```

Install the required dependencies:

```
pip install -r requirements.txt
```

Or using uv:

```
uv pip install -r requirements.txt
```

To generate the bitstream for Zynq, use:

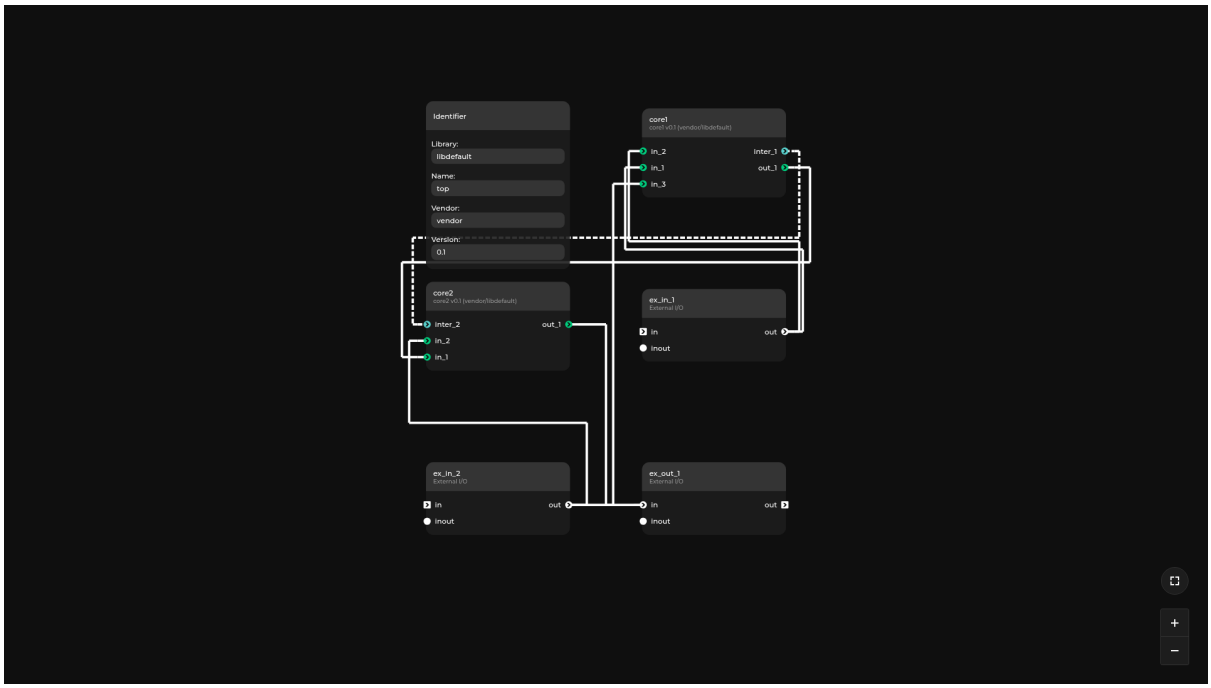
```
make
```

To generate only the HDL sources use:

```
make generate
```

User repository

Link to source



This example presents the structure of a user repository containing prepackaged IP cores with sources and custom interface definitions.

Elements of the `repo` directory can be easily reused in different designs by linking to them from the config file or in the CLI.

See also:

For more information about user repositories see *this chapter*.

Tip: As other components of the design are automatically imported from the repository, it's possible to load the entire example by specifying the design file:

```
topwrap gui -d project.yml
```

5.3.2 Usage

Navigate to the `/examples/user_repository/` directory and run:

```
topwrap gui -d project.yml
```

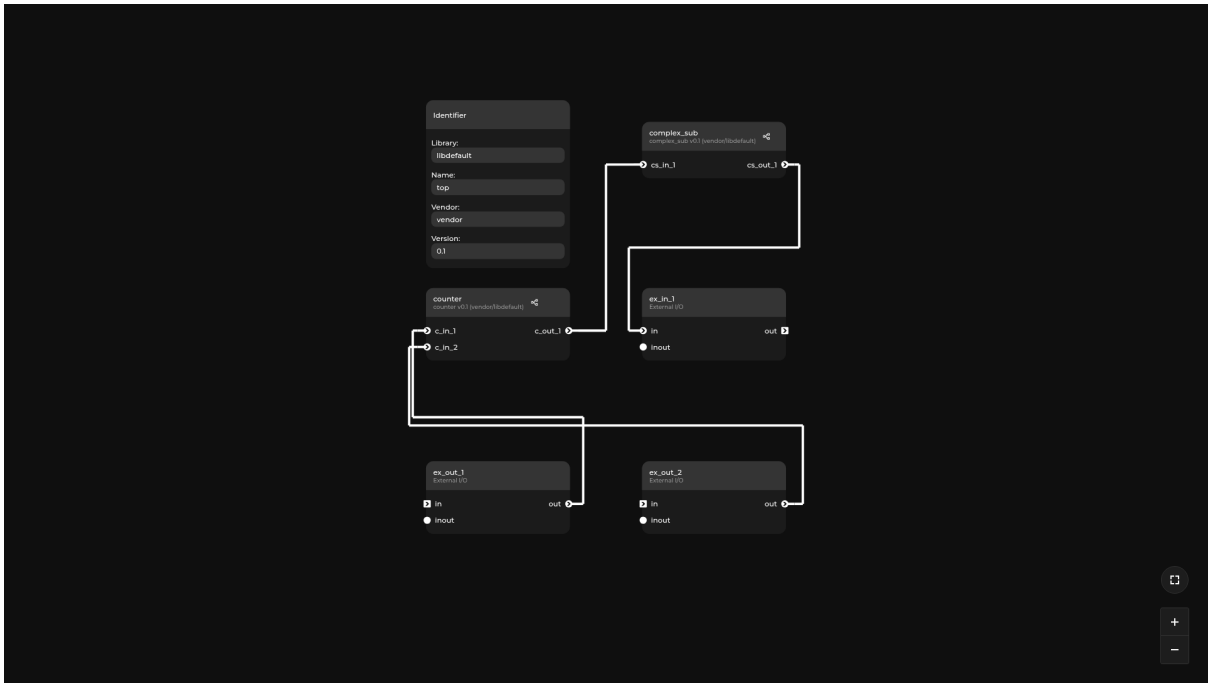
Expected result

Topwrap will load two cores from the `cores` directory, using the interface from the `interfaces` directory.

In the Nodes browser under `IPcore`, two loaded cores: `core1` and `core2`, should be visible.

5.4 Hierarchy

Link to source



This example shows how to create a hierarchical design in Topwrap, including a hierarchy that contains IP cores as well as other nested hierarchies.

Check out `project.yaml` to learn how the above design translates to a *design description file*

See also:

For more information, see the section on *Hierarchies*.

Tip: Hierarchies are represented in the GUI by nodes with a green header. To display inner designs, click the `Edit subgraph` option from the context menu.

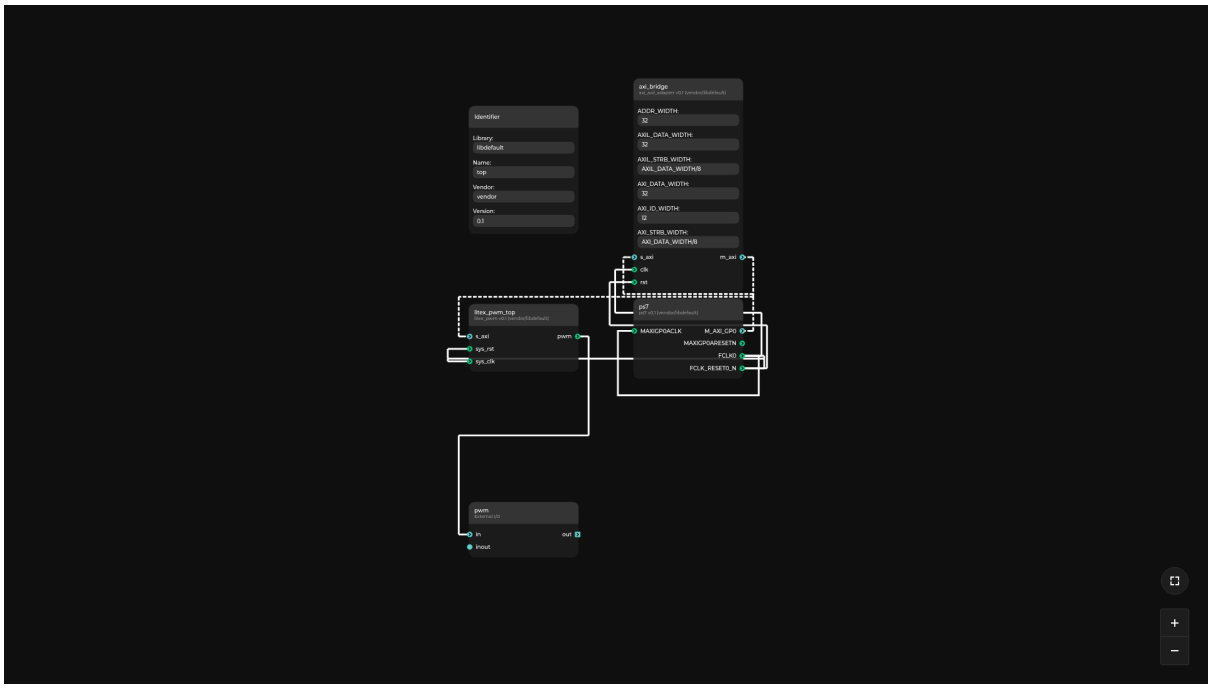
To exit from the hierarchy subgraph, use the back arrow button on the top left. To add a new hierarchy node, use the `New Graph Node` option in the node browser.

5.4.1 Usage

```
topwrap gui -d project.yaml
```

5.5 PWM

Link to source



Tip: The IP core in the center of the design (`axi_axil_adapter`) showcases how IP cores with overridable parameters are represented in the GUI.

This is an example of an AXI-mapped PWM IP core that can be generated with LiteX, connected to the ZYNQ Processing System. The core uses the AXILite interface, so a `AXI -> AXILite` converter is needed. You can access its registers starting from address `0x4000000` (the base address of `AXI_GP0` on ZYNQ). The generated signal can be used in a FPGA or connected to a physical port on a board.

Note: To connect I/O signals to specific FPGA pins, you must use mappings in a constraints file. See `zynq.xdc` used in the setup and modify it accordingly.

5.5.1 Usage

Switch to the subdirectory with the example:

```
cd examples/pwm
```

Install the required dependencies:

```
pip install -r requirements.txt
```

Or using recommended uv:

```
uv pip install -r requirements.txt
```

Note: In order to generate a bitstream, install Vivado and add it to the PATH.

To generate bitstream for Zynq, use:

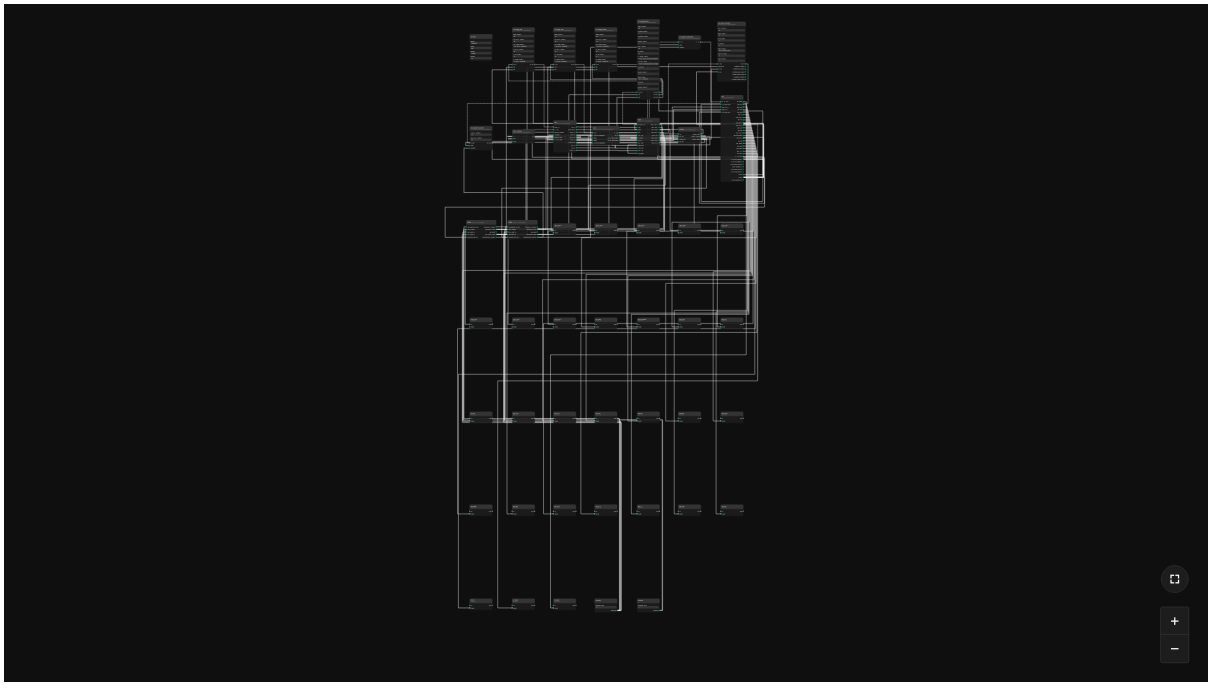
```
make
```

To generate HDL sources without running Vivado, use:

```
make generate
```

5.6 HDMI

Link to source



This is an example of how to use Topwrap to build a complex and synthesizable design.

5.6.1 Usage

Switch to the subdirectory with the example:

```
cd examples/hdmi
```

Install the required dependencies:

```
pip install -r requirements.txt
```

Or using recommended uv:

```
uv pip install -r requirements.txt
```

Note: In order to generate a bitstream, install Vivado and add it to the PATH.

Generate bitstream for desired target

Snickerdoodle Black:

```
make snickerdoodle
```

Zynq Video Board:

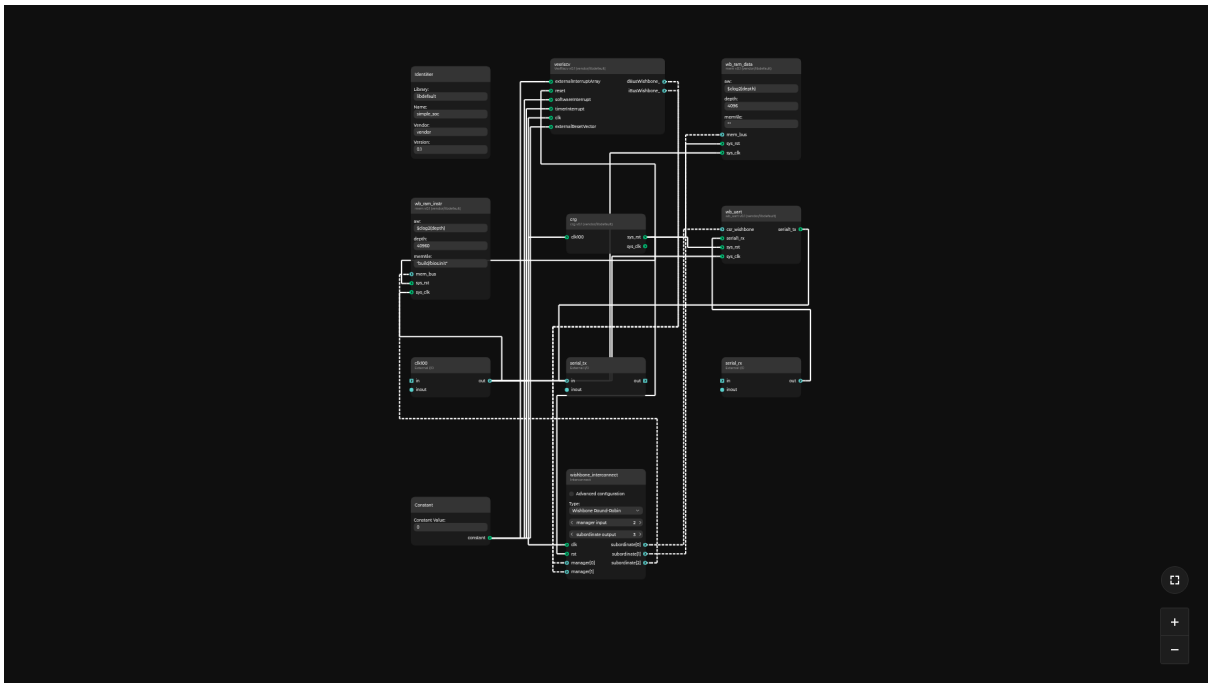
```
make zvb
```

To generate HDL sources without running Vivado, use:

```
make generate
```

5.7 SoC

Link to source



This example is SoC that has an RISC-V processor, memory, and a UART connected by Wishbone interconnect. Wishbone interconnect has source or can be generated by Topwrap, it enables to compare generated interconnect with existing source code. This example can be run in a simulator such as Verilator. To run this example, the RISC-V 64-bit toolchain and Verilator simulator need to be installed.

5.7.1 Usage

Switch to the subdirectory with the example:

```
cd examples/soc
```

Install the required dependencies:

```
sudo apt install git make g++ ninja-build gcc-riscv64-unknown-elf bsdextrautils
```

Note: To run the simulation, you need:

- verilator

To create and load the bitstream, use:

- Vivado (preferably version 2020.2).
- openFPGALoader (branch)

Generate HDL sources:

```
make generate
```

Build and run the simulation:

```
make sim
```

The expected waveform generated by the simulation is shown in `expected-waveform.svg`. Generated waveforms by simulation are present in `build/inter-topwrap` for interconnect generated by topwrap, and `build/inter-source` for interconnect from source.

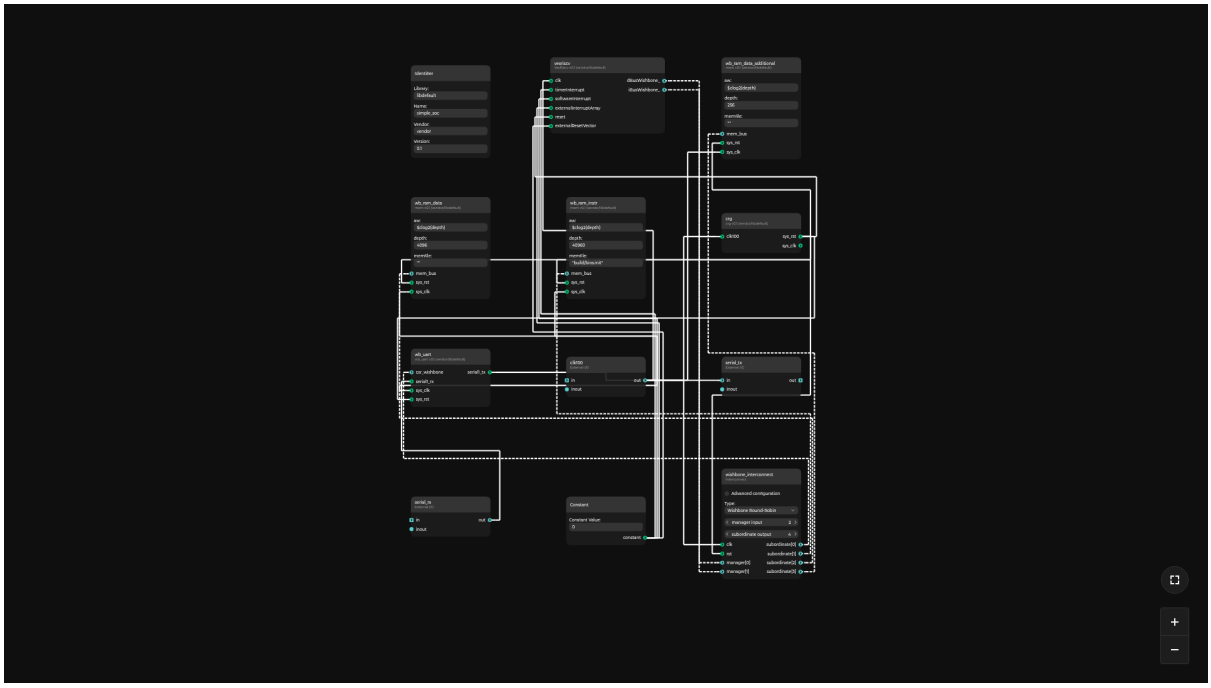
Generate the bitstream:

```
make bitstream
```

Note: For `generate`, `sim` and `bitstream` targets there also targets with suffix `-inter-topwrap` and `-inter-source`, they can be used to generate HDL sources, simulate or generate the bitstream using topwrap generated or from source interconnect.

5.8 Memory Map

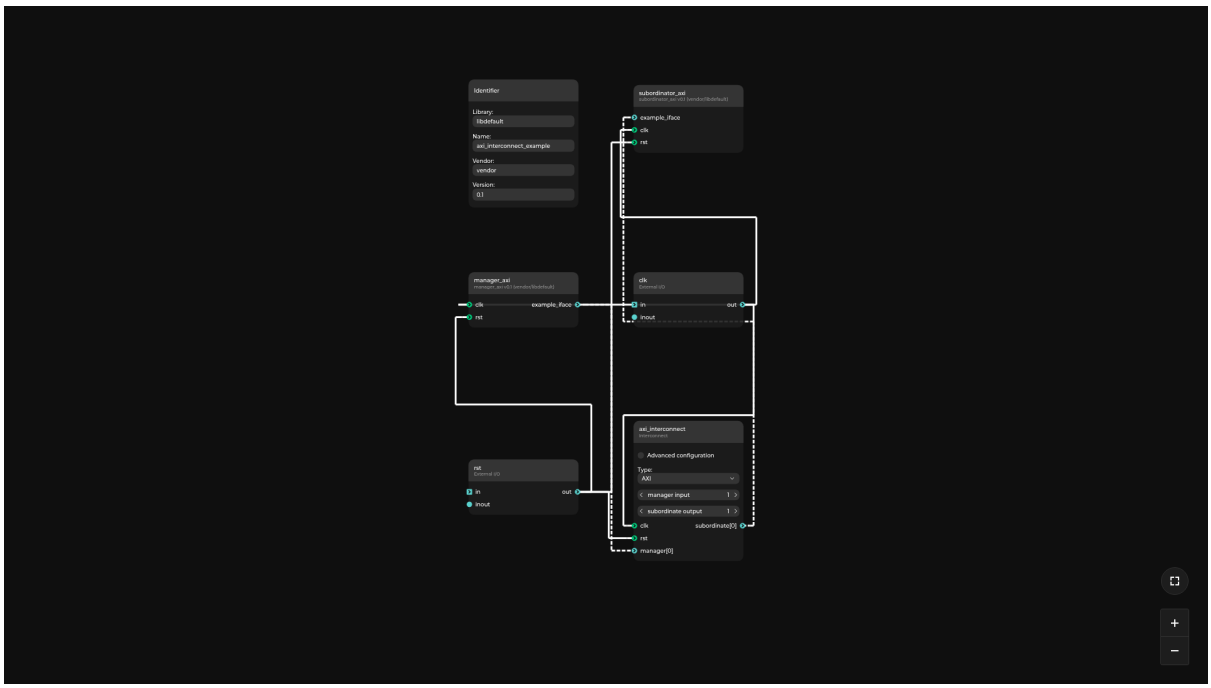
Link to source



This example is based on the SoC example and it uses memory maps instead of the `subordinates` section in the interconnect. The repository contains hand-edited yaml files with the interface size specified for the `wb_uart` module. Refer to the *Description Files* and the *Interconnect Generation* for more information about memory maps.

5.9 AXI interconnect

Link to source



This example has one subordinator and one manager connected to each other using AXI4 interconnect generated by Topwrap. It is mainly used for checking if AXI4 interconnect generated by Topwrap is working in simulation. Similar examples that focus more on interconnect representation in Topwrap can be found in *ir_examples*

5.9.1 Usage

Switch to the subdirectory with the example:

```
cd examples/axi_interconnect
```

To generate HDL sources, use:

```
make generate
```

To run simulation using Verilator and generate `wave.vcd` use:

```
make sim
```

5.9.2 IPXACT

Link to source

This example contains IP-XACT 2022 files as sources. It shows how to use *IpXactFrontend* to parse IP-XACT sources into a Topwrap repo.

5.9.3 Usage

```
cd examples/ipxact
```

To generate the HDL top file, use:

```
make generate
```

5.9.4 Existing interface definition

Link to source

This example showcases how Topwrap behaves when there is existing interface definition present in HDL files.

5.9.5 Usage

```
cd examples/existing_iface
```

To generate the HDL top file, use:

```
make generate
```

CHAPTER

SIX

CREATING A DESIGN

This chapter explains how to create a design in Topwrap, including a detailed overview of how Topwrap design files are structured.

6.1 Design description

To create a complete and fully synthesizable design, a design file is needed. It is used for:

- specifying IP cores and their parameters
- specifying interconnects
- describing hierarchies for the project
- connecting the IPs and hierarchies
- defining design interfaces and ports

You can see example design files in the `examples` directory. The structure of the design file is shown below:

```
name: {design_name} # optional name of the toplevel

ips:
  # specify relations between IPs instance names in the
  # design yaml and IP cores description YAMLs
  {ip1_instance_name}:
    # exactly one of `file` or `core` identifies the IP core description
    file: {resource_path} # see "Resource path syntax" section for more␣
↪information
    core: {vlnv}          # a module from a registered library, see "Libraries"
    parameters:  # specify IP parameter values to be overridden
      {parameters_name} : {parameters_value}
      ...
    clocks:
      {module_clock_name}: {domain_name}
    resets:
      {module_reset_name}: {domain_name}
  ...

connections:
```

(continues on next page)

(continued from previous page)

```
  ports:
    # specify the incoming ports connections of an IP named `ip1_name`
    {ip1_name}:
      {port1_name} : [{ip2_name}, {port2_name}]
      # connections between modules can be (bit-wise) inverted
      {port3_name} : ~[{ip2_name}, {port4_name}]
      ...
    # specify the incoming ports connections of a hierarchy named `hier_name`
    {hier_name}:
      {port1_name} : [{ip_name}, {port2_name}]
      ...
    # specify the external port connections
    {ip_instance_name}:
      {port_name} : ext_port_name
      # connections to external ports can be (bit-wise) inverted
      {other_port_name} : ~other_ext_port_name
    ...

  interfaces:
    # specify the incoming interface connections of the `ip1_name` IP
    {ip1_name}:
      {interface1_name} : [{ip2_name}, {interface2_name}]
      ...
    # specify the incoming interface connections of the `hier_name` hierarchy
    {hier_name}:
      {interface1_name} : [{ip_name}, {interface2_name}]
      ...
    # specify the external interface connections
    {ip_instance_name}:
      {interface_name} : ext_interface_name
    ...

interconnects:
  # see the "Interconnect generation" page for a detailed description of the␣
↪format
  ...

clock_domains:
  default:
    signal: [{ip_name}, {port_name}]
  {other_domain_name}:
    signal: [{ip_name}, {port_name}]
  {another_domain_name}:
    signal: {external_port_name}
  ...

reset_domains:
  default:
    signal: [{ip_name}, {port_name}]
```

(continues on next page)

(continued from previous page)

```
    polarity: active high
  {other_domain_name}:
    signal: [{ip_name}, {port_name}]
    polarity: active low
  {another_domain_name}:
    signal: {external_port_name}
    polarity: active low
    synchronous_to: default
  ...

external: # specify the names of external ports and interfaces of the top module
  ports:
    out:
      - name: {ext_port_name}
        bound: [[dim0_high, dim1_low], ..., [dimN_high, dimN, low]]
    inout:
      - [{ip_name/hierarchy_name, port_name}]
  interfaces:
    in:
      - {ext_interface_name}
    # note that `inout:` is invalid in the interfaces section

hierarchies:
   # see "Hierarchies" below for a detailed description of the format
      ...

memory_maps:
  {memory_map_name}: # this name is referenced in an interconnect
    {instance_name}: # component instance reference
      {interface_name1}: # interface of component
        address: {address1}
        {additional_parameter_name}: {value_accepted_by_an_interconnect_
↪implementation} # additional parameters for specific interconnects can be␣
↪specified
        ...
      {interface_name2}:
        address: {address2}
        size: {size2} # a default size can be overwritten
    ...
    {instance_name}: # when an instance has only one interface, it is not␣
↪required to specify the interface name
      address: {address1}

config: # override global configuration settings (optional)
  force_interface_compliance: {true/false}
  repositories:
    {repo_name}: {repo_source}

positions: path/to/positions.yaml
```

(continues on next page)

(continued from previous page)

```
extensions: # plugin-specific settings
    {plugin_setting_scope}: ...
```

`inout` ports are handled differently than the `in` and `out` ports. When an IP has an inout port or when a hierarchy has an inout port specified in its `external.ports.inout` section, it must be included in the `external.ports.inout` section of the parent design. It is required to specify the name of the IP/hierarchy and the port name that contains it. The name of the external port is identical to the one in the IP core. In case of duplicate names, a suffix `$n` is added (where `n` is a natural number) to the name of the second and subsequent duplicate names. `inout` ports cannot be connected to each other.

The design description YAML format allows for creating hierarchical designs. In order to create a hierarchy, add its name as a key in the `hierarchies` section and describe the hierarchy design "recursively" by using the same keys and values (`ports`, `parameters` etc.) as in the top-level design (see above). Hierarchies can be nested recursively, which means that you can create a hierarchy inside another one.

Note that IPs and hierarchies names cannot be duplicated on the same hierarchy level, as each entry in `hierarchies` is treated as equal with entries in `ips`.

Additionally, designs can contain clock and reset domains. These domains allow for automatic wiring of clock and reset inputs of IP cores, and performing sanity checks for interface connections. Domains reference a port that acts as the source. This may be either a module port (for example from a PLL instance), or an external port.

Additionally, reset domains contain information about the reset signal's polarity and whether it's synchronous to some domain or asynchronous. The polarity may be either `active low` or `active high`. The synchronicity is specified via the `synchronous_to` key, which can be `null`/omitted if asynchronous, or otherwise specifies the name of a clock domain.

When instantiating IP cores, their clocks and resets can be assigned to domains via the `clocks` and `resets` keys. If a domain is not assigned explicitly, the clock/reset signal is implicitly assigned to the corresponding `default` domain.

For reset connections, Topwrap will automatically invert connections between resets of different polarities. For resets that don't have the same synchronicity, an error is raised describing the problem.

The `config` option allows the design to override *configuration settings* using the same syntax as in the configuration files. This includes the list of repositories to load. Topwrap pre-parses the design document before fetching repositories, so the settings placed are picked up before the design is evaluated.

The optional `positions` property allows specifying a path to the positions YAML file for this design (see below). The specified path, if relative, will be checked both in the directory Topwrap was launched from, and in the directory the design YAML is in. If the positions YAML file is present, it will automatically be loaded alongside the design YAML file.

The `extensions` property allows the design to optionally configure settings for any *plugins* that are loaded. These settings are scoped by a label `{plugin_setting_scope}`. Every plugin supports zero or more of these setting-scopes. How these settings are intrepreted depends on the plugin and should be specified as part of the plugin's own documentation. For example, an *imaginary* plugin that checks the spelling of port names could possibly require these settings:

```
extensions:
  spelling:
    dictionary: "./path/to/dict.txt"
```

Here, `spelling` is the setting-scope and the key-value pair `{ dictionary: ... }` is the setting-scope's value.

6.1.1 Hierarchies

Hierarchies allow for creating designs with subgraphs in them. The subgraphs can contain multiple IP cores and other subgraphs, allowing for the creation of nested designs in Topwrap.

6.1.2 Format

Hierarchies are specified in the *design description*. The `hierarchies` key must be a direct descendant of the `design` key.

The format is as follows:

```
hierarchies:
  {hierarchy_name_1}:
     # Design description, see above
     ...
```

A more complex example of a hierarchy can be found in the examples/hierarchy directory.

6.2 IP description files

The primary purpose of the IP core YAML is to provide the tool with essential metadata and structural information about a specific IP core to facilitate automated SoC assembly. It contains information about signals, clock domains, parameterization and interfaces. Interfaces are defined as a list of signals and parameters such as `mode` and `type`.

Such a file can be handwritten, or generated from an existing *design description* using the `topwrap extract` command, which describes the design's top module as a reusable IP core:

```
topwrap extract --design design.yaml
```

See the *Constant example* for a runnable usage of this command.

Warning: `topwrap extract` only captures the top module's external signals, interfaces, clocks, resets and parameters - it does not preserve internal hierarchy. If the design's top module is itself composed of sub-instances (via `hierarchies/ips`), that internal composition (instances, sub-hierarchies, internal connections) is not included in the generated IP description; the design is reduced to a single black-box IP core described only by what's externally visible. If you need to preserve the internal structure, use the *design description* format instead (the output of `topwrap build`), which does capture instances and connections.

The example below presents an example IP YAML core description file.

```yaml
id:
  library: libdefault
  name: axi4_to_ahb
  vendor: vendor

parameters:
  RESET_ADDRESS: 32'h80000000

signals:
  in:
    - sys_clk
    - sys_rst
    - name: core_id
      bound: [3, 0]
      default: 0
    - name: wb_req
      type: wb_req_t
  out:
    - core_halted
    - name: wb_resp
      type: wb_resp_t

interfaces:
  ifu_axi:
    type:
      name: AXI4
      vendor: vendor
      library: libdefault
    mode: manager
    # the size field in manager mode is unused by topwrap
    parameters:
      DATA_WIDTH: 64
      ID_WIDTH: 2
    signals:
      out:
        bready: ifu_bready
        awaddr: ifu_awaddr
        # ... remainder of AXI signals ...
      in:
        awready: ifu_awready
        rdata: ifu_rdata
        # ... remainder of AXI signals ...
  iface1_wb:
    type: wishbone
    mode: subordinate
    parameters:
      DATA_WIDTH: 32
    size: 0xFFFF # the size field in subordinate mode is optional
    signals:
```

(continues on next page)

(continued from previous page)

```
      # wishbone signals
      addr:
        path: wb_req.addr

types:
  wb_req_t:
    members:
      - name: cyc
        type: [0, 0]
      - name: addr
        type: ...
      ...
  wb_resp_t:
    members:
      ...
existing_iface_definitions:
- library: libdefault
  name: AXI4Lite
  vendor: vendor
  version: '0.1'

extensions: ...
```

6.2.1 File format explanation

- `id` - identification of the Core. An instance of *Identifier IR class*
  - `library` - by default `libdefault`
  - `vendor` - by default `vendor`
  - `name` - same as the name in the HDL file
- `parameters` - a dictionary with IP core parameters
- `signals` - contains the names of signals used in the module ports
  - `out` - a list of signals that are in the output direction
  - `in` - a list of signals that are in the input direction
  - `inout` - a list of signals that can be both input and output, similar to an `inout` port in Verilog
- `interfaces` - a list of interfaces
  - `type` - the *ID* of the interface definition that is used for that interface. See the *Interface Definition YAML*.
  - `mode` - `manager`, `subordinate` or `unspecified`. When generating ports, the mode is used to determine the direction of the port. `unspecified` behaves the same as `manager`. When doing inference, all newly created interfaces have their `type` set to `unspecified`.

  - `size` - the optional field that is only used when `mode` is `subordinate`, it is used when there is an instance of this module specified in `address_maps` in the design YAML.
  - `signals` - a list of signals mapped to ports. Signals from the interface definition need to be mapped to ports in the HDL module.
  - `clock` - (optional) name of the clock input this interface is synchronous to.
  - `reset` - (optional) name of the reset input for this interface.
  - `parameters` - (optional) values for parameters (described in *Interface Definition YAML*) for this interface instance
- `types` - structure type definitions used by signals (described below in more detail)
  - `members` - member fields of a struct type.
- `existing_iface_definitions` - a list of interface definitions that exist in parsed HDL code in form of *ID*s.
- `extensions` - plugin specific settings, has the same meaning as the `extensions`-property in *design files*.

Interfaces can also specify which clock and reset input of the IP core they use. This is done via the optional `clock` and `reset` keys, which specify the name of the input (see below).

When connecting interfaces that have clock inputs assigned on both sides of the connection, Topwrap will automatically perform sanity checks, and report errors for connections between interfaces that belong to different clock domains.

6.2.2 Signals

Each signal (for both port and interface definitions) can be specified in one of two formats.

New signal format

The signal is defined by a YAML object, with the following properties:

- `name` (optional) - the name of the signal.
- `bound` (optional) - the bounds of the signal. The signal can be multidimensional.
- `slice` (optional) - the bounds of the slice bit range (only applicable to interface definitions).
- `default` (optional) - default value to assign to this port if nothing is connected to it, applicable only to input ports.
- `type` (optional) - the name of the type (defined in the `types` section) used for this signal
- `path` (optional) - the path to a field to use as the source for this signal

A signal must have either a `name` or a `path` specified, but not both at once.

Additionally, `path` may only be specified for interface signals. For example, a path of `some_port.some_field` describes a signal that uses a subfield of a port. Paths can also describe slices, e.g. `some_port[31:0]` is equivalent to specifying a `name` of `some_port` and a `slice` of `[31, 0]`. Paths can use any combination of slicing and field selection operations, e.g. `some_port[3].some_field[1].some_other_field[31:0]`.

A signal may specify `bound` or `type` (or neither for a single bit port), but not both at the same time.

Example port definitions using this format:

```
signals:
    in:
        - name: foo
          bound: [31, 0]
          default: 32'hDEADBEEF
        - name: bar
          type: some_struct_t
```

Example interface signal definitions using this format:

```
interfaces:
    foo:
        type: AXIStream
        mode: subordinate
        signals:
            in:
                BAR:
                    name: bar
                    slice: [47, 16]
                ...
                BAZ:
                    name: BAZ
                QUX:
                    path: some_struct_port.qux
```

Old signal format

The signal is defined by either:

- just a name (implied one-bit signal):

  ```
  signals:
      in:
          - foo
  ```

- an array consisting of the name and the bounds of the bit range:

  ```
  signals:
      in:
          - [foo, 31, 0]
  ```

- an array consisting of the name, the bounds of the bit range, and the bounds of the slice bit range (only applicable to interface definitions):

  ```
  signals:
      in:
          TDATA: [foo, 63, 0, 47, 16]
  ```

As an example, this is the description of ports in the `Clock Crossing` IP:

```
# file: clock_crossing.yaml
name: cdc_flag
signals:
    in:
        - clkA
        - A
        - clkB
    out:
        - B
```

Types

IP core YAML files may additionally describe types used by the module's ports in a dedicated section. This is primarily intended to support modules using structure ports.

Each named type is defined in the `types` section:

```
types:
  my_struct_type_t:
    ...
  my_other_type_t:
    ...
```

Each type is either:

- a two element array describing a bit vector (incl. single bits), e.g. `[31, 0]` or `[0, 0]`
- an object with a `members` describing a struct, described below

The members of a struct are defined in an array of field objects, each of which consists of the following properties:

- `name` - the name of the field
- `type` - the type definition for the field

Example struct definition describing AXI output signals:

```
types:
  my_axi_req_t:
    members:
      - name: aw_valid
        type: [0, 0]
      - name: aw
        type:
          members:
            - name: addr
              type: [31, 0]
            ...
      ...
```

6.2.3 Differences between the IP and design formats

The following information is present in the design format, but not in the IP format:

- `ips`, `hierarchies`, `connections` and `interconnects` and `memory_maps`
- `config`
- `positions`

The following information is present in the IP format, but not in the design format:

- `parameters`
- `types`

The mapping between other sections is as follows:

- `external/ports` corresponds to `signals`
- `external/interfaes` corresponds to `interfaces`
- `clock_domains/reset_domains` corresponds to `clocks/resets`
- VLNV fields (`vendor`, `library`, `name`, `versions`) are packaged into `id`

Note that in the IP format, the name of the clock/reset signal is specified inside the `clocks` field and not as a separate port. The same applies to reset specification.

6.2.4 Interface definitions

The purpose of the interface YAML file is to store information about interface definitions. These definitions are used for inferring ports to interface instances. It contains the id of the interface, which is used to reference the definition from the IP core YAML file. The YAML file separates `signals` into those that are required for a list of ports to be interpreted as this interface and optional signals that, if present, are inferred into a single interface instance.

Regular expressions are used to find ports in the HDL module that correspond to the signals in the interface definition.

```
id
  library: libdefault
  name: wishbone
  vendor: vendor
signals:
    required:
        out:
            cyc: cyc
            stb: stb
        in:
            ack: ack
    optional:
        out:
            dat_w: dat_(w|mosi)|mosi
            adr: adr
            tgd_w: tgd_w
```

(continues on next page)

(continued from previous page)

```
            lock: lock
            sel: sel
            tga: tga
            tgc: tgc
            we: we
            cti: cti
            bte: bte
        in:
            dat_r: dat_(r|miso)|miso
            tgd_r: tgd_r
            stall: stall
            err: err
            rty: rty
```

File format explanation:

- `id` - identification of the interface. An instance of *Identifier IR class*
  - `library` - by default `libdefault`
  - `vendor` - by default `vendor`
  - `name` - same as the name in the HDL file
- `signals` - contains the names of signals with regular expressions that are used in the ports of Modules.
  - `required` - contains signals that must be present in the list of ports in the module
  - `optional` - contains signals that are connected to the interface instance when present in the list of ports in the module
    - `out` - a list of signals that are in the output direction
    - `in` - a list of signals that are in the input direction
    - `inout` - a list of signals that can be both the input and the output, similar to an `inout` port in Verilog

Interfaces can also specify which clock and reset input of the IP core they use. This is done via the optional `clock` and `reset` keys, which specify the name of the input (see below).

When connecting interfaces that have clock inputs assigned on both sides of the connection, Topwrap will automatically perform sanity checks, and report errors for connections between interfaces that belong to different clock domains.

6.2.5 Parameterization

Port widths don't have to be hardcoded, as parameters can describe an IP core in a generic way, and values specified in IP core YAMLs can be overridden in a design description file (see *Design description*).

```
parameters:
    DATA_WIDTH: 8
    KEEP_WIDTH: (DATA_WIDTH+7)/8
    ID_WIDTH: 8
    DEST_WIDTH: 8
    USER_WIDTH: 1

interfaces:
    s_axis:
        type: AXI4Stream
        mode: subordinate
        parameters:
          TID_WIDTH: ID_WIDTH
          TDATA_WIDTH: DATA_WIDTH
          TDEST_WIDTH: DEST_WIDTH
          TUSER_WIDTH: USER_WIDTH
        signals:
            in:
                TDATA: s_axis_tdata
                TKEEP: s_axis_tkeep
                ...
                TID: s_axis_tid
                TDEST: s_axis_tdest
                TUSER: s_axis_tuser
signals:
  out:
    - name: other_dat_out
      bound: [DATA_WIDTH-1, 0]
      default: 0
```

The parameter values can be integers or math expressions.

6.2.6 Clock and reset inputs

IPs can define clock and reset inputs, which allow for automatic connections to clock/reset domains within a design. Each clock/reset specifies a port that is used for it. Additionally, reset inputs specify polarity and synchronicity similarly to reset domains in the design description.

Example clock and reset definition:

```
# Two independent clocks
clocks:
  some_clock:
    signal: some_clock_port
  some_other_clock:
```

(continues on next page)

(continued from previous page)

```
    signal: some_other_clock_port

# Two resets, one synchronous, one asynchronous
resets:
  some_reset:
    signal: some_reset_port
    polarity: active high
    synchronous_to: some_other_clock
  some_other_reset:
    signal: some_other_reset_port
    polarity: active low
    synchronous_to: null
```

In both cases, the `signal` key defines which of the module's ports belongs to this clock/reset. The `polarity` and `synchronous_to` keys are the same as the ones in the design description, with one difference: in IP cores `synchronous_to` references a clock input, not a clock domain.

6.3 Interface definition files

The purpose of the interface definition YAML file is to store information about an interface: it's signals and their directions, whether they're optional, default values, etc. The signals also contain regex patterns used for matching them up to module ports during interface inference.

Example of a (simplified) definition of the Wishbone interface:

```
id:
  name: wishbone
  library: libdefault
  vendor: vendor

parameters:
  GRANULARITY: 8
  ADDR_WIDTH: 32
  DATA_WIDTH: 64
  TGA_WIDTH: 32
  TGD_WIDTH: 32
  TGC_WIDTH: 32

signals:
  adr:
    pattern: adr
    modes: {manager: [out, required]}
    bound: ['ADDR_WIDTH-1', '$clog2(DATA_WIDTH/GRANULARITY)']
  dat_w:
    pattern: mosi|dat_(w|mosi)
    modes: {manager: [out, required]}
    bound: ['DATA_WIDTH-1', '0']
  dat_r:
    pattern: miso|dat_(r|miso)
```

(continues on next page)

(continued from previous page)

```
    modes: {manager: [in, required]}
    bound: ['DATA_WIDTH-1', '0']
  sel:
    pattern: sel
    modes: {manager: [out, required]}
    bound: ['DATA_WIDTH/8-1', '0']
  cyc:
    pattern: cyc
    modes: {manager: [out, required]}
  stb:
    pattern: stb
    modes: {manager: [out, required]}
  we:
    pattern: we
    modes: {manager: [out, required]}
  ack:
    pattern: ack
    modes: {manager: [in, required]}
  err:
    pattern: err
    modes: {manager: [in, optional]}
  rty:
    pattern: rty
    modes: {manager: [in, optional]}
  stall:
    pattern: stall
    modes: {manager: [in, optional]}
  lock:
    pattern: lock
    modes: {manager: [out, optional]}

  ...
```

File format explanation:

- `id` - identifier of the interface
- `parameters` - dictionary of parameter names and their default values. Those can be used to allow Interfaces to have e.g. adjustable size data signals
- `signals` - dictionary of signals that this interface is comprised of; each item's key is the signal name, and the value is an object with the following keys:
  - `pattern` - the regex pattern used for matching signals to ports during interface inference
  - `manager` - (optional) the signal direction and whether it's optional or required in the manager mode of the interface
  - `subordinate` - (optional) the signal direction and whether it's optional or required in the subordinate mode of the interface
  - `unspecified` - (optional) the signal direction and whether it's optional or required in

the unspecified mode of the interface

- `default` - (optional) the value assigned to the signal if the other side does not contain the signal
- `bound` - (optional) bounds of the signal (can be parameterized)

For each mode specification, the first value is one of `in`, `out` or `inout`, and the second value is one of `required`, `optional`.

At least one of `manager`, `subordinate` or `unspecified` must be specified for each signal.

If only one of `manager` or `subordinate` is specified, the other is inferred as having an opposite direction and the same optionality. Explicitly specifying both `manager` and `subordinate` allows for specifying the same direction and/or different optionality. If not defined, `unspecified` is inferred to be the same as `manager`.

6.3.1 Interface compliance

During the *build process*, an optional verification of whether the interface instances used in IP cores are compliant with their respective descriptions can be enabled. The verification consists of checking in the instance if:

- all signals designated as required in the description are included.
- no additional signals beyond those defined in the description are included.

This feature is controlled by the `--iface-compliance` CLI flag or the `force_interface_compliance` key in the *configuration file* and is turned off by default.

6.4 Resource path syntax

Fields specified in the YAML file as a "resource path" support extended functionality and have their own specific syntax.

This field type is used for example in the *Design Description* for specifying an IP Core description location:

```
ips:
  ip_inst_name:
    file: {resource path}
...
```

A value with no scheme is a filesystem path, resolved relative to the YAML file it appears in:

```
ips:
  ip_inst_name:
    file: ./my_directory/core.yaml
```

To reach anything else, prefix the value with a scheme:

```
SCHEME[ARG1|ARG2...]:SCHEME_PATH
```

- `SCHEME` is the scheme of this path (e.g. `get` for remote resources)

- `ARGS` are `|`-separated positional arguments for the specific scheme (e.g. the user repo name for the `repo` scheme)
  - If there are no arguments to supply you can omit the square brackets entirely
- `SCHEME_PATH` is the path to the resource interpreted by the specific scheme (e.g. the URL for the `get` scheme)

6.4.1 Available schemes

- `file`
  - `SCHEME_ARGS`: None
  - `SCHEME_PATH`: A filesystem path relative from the currently edited YAML file to the resource
  - Deprecated since version 1.0.0: Write the path directly, without the `file:` prefix.
- `repo`
  - `SCHEME_ARGS`: Repository name
  - `SCHEME_PATH`: A name of resource
  - Deprecated since version 1.0.0: The user repository mechanism is being removed. Register the sources as a *library* and reference them with `core:` instead.
- `get`
  - `SCHEME_ARGS`: None
  - `SCHEME_PATH`: The URL address of the remote resource. Only `http(s)://` URLs are currently supported.
- `git`
  - `SCHEME_ARGS`:
    1. An optional git ref (branch, tag, or commit SHA) to check out. If omitted, the repository's default branch is used.
    2. An optional subdirectory inside the repository to use as the resolved path.
  - `SCHEME_PATH`: The URL of a git repository to clone.

Warning: If a repository was injected to the config with the `--repo` CLI option, the name of such repository will be the name of the directory containing it.

6.4.2 Examples

```
./my_directory/file.txt
```

A path to the file on the filesystem, relative to the YAML file it is written in.

```
repo[my_repo]:res.txt
```

This loads the `res.txt` file inside the `my_repo` loaded user repository.

```
get:https://raw.githubusercontent.com/antmicro/topwrap/refs/heads/main/pyproject.
↪toml
```

This loads the remote resource. When necessary, it's automatically downloaded into a temporary directory.

```
git[main]:https://github.com/antmicro/topwrap.git
```

This clones the `main` branch of the given git repository into a persistent cache directory and uses it as the resource location. Omitting the `[main]` argument checks out the repository's default branch instead. A commit SHA can be used in place of a branch or tag name.

```
git[main|topwrap/builtin]:https://github.com/antmicro/topwrap.git
```

This clones the `main` branch as above, but resolves to the `topwrap/builtin` subdirectory of the repository instead of its root.

Cloned repositories are cached persistently between runs. To remove all locally cached clones, run:

```
topwrap clean-cache --target git
```

Omitting `--target` removes every cache topwrap maintains.

6.5 Positions YAML

Positions YAML files are auxiliary files that save the positions of elements set in the GUI when saving the design to the YAML format. These files are not meant to be edited or inspected by hand, and ought to be treated as ephemeral.

The structure of the positions file is shown below:

```yaml
modules:
- id:
    vendor: vendor
    library: libdefault
    name: my_module
    version: 0.1
  identifier: [-1000.0, -1000.0]
  components:
  - name: foo
```

(continues on next page)

(continued from previous page)

```
    position: [-100.0, 200.0]
  - ...
  externals:
  - name: clk
    position: [-200.0, 200.0]
  - ...
  constants:
  - name: 32
    position: [-200.0, 200.0]
  - ...
  clock_domains:
  - ...
  reset_domains:
  - ...
  interconnects:
  - ...
  inverters:
  - source: clk
    target: [foo, clk]
    position: [-150.0, 200.0]
  - ...
```

6.5.1 File format explanation

- `modules` - list of objects describing positions of elements within modules
  - `id` - the identifier of the module this description is for
  - `identifier` - the position of the identifier node
  - `components` - list of objects describing positions of components
    - `name` - the name of the component
    - `position` the component's position
  - `externals` - list of objects describing positions of external I/O nodes
    - entries use the same format as `components`
  - `constants` - list of objects describing positions of constant value nodes
    - entries use the same format as `components`
  - `clock_domains` - list of objects describing positions of clock domain nodes
    - entries use the same format as `components`
  - `reset_domains` - list of objects describing positions of reset domain nodes
    - entries use the same format as `components`
  - `interconnects` - list of objects describing positions of interconnect nodes
    - entries use the same format as `components`
  - `inverters` - list of objects describing positions of inverter nodes

* `source` - the source signal of the connection with an inverter
* `target` - the target signal of the connection with an inverter
* `position` - the inverter's position

CHAPTER

SEVEN

INTERFACE INFERENCE

Warning: SystemVerilog modules with parameter types are not fully supported by inference. There is no way to override default parameter values, and as such, you must create a wrapper module that has concrete parameter values specified. Ports with non-parameterized types are unaffected by this.

This is particularly important to keep in mind when using modules that use ports with struct types, since the types are likely realized using parameter types.

In addition to explicit interface definitions, Topwrap also supports automatically inferring interfaces from module ports based on the interface and module definitions. This can be done when parsing SystemVerilog modules using `topwrap repo parse` by specifying the `--inference` flag.

Inference can be performed on groups of module ports, or fields of ports with struct types, and in the latter case, it also supports ports that are one-dimensional arrays of structs, creating separate interfaces for each array element.

The inference process is done in two steps. First, Topwrap finds all groups which might form an interface. For module ports, this is done by finding common prefixes in port names. For structures, all struct members (recursively including members of members) of a port are considered to be within one group, named after the port.

Additionally, the inference logic accepts grouping hints, which describe which groups should be merged together, and what the resulting group should be called. This is particularly useful in conjunction with the struct-based inference, as usually there are separate ports for the input and output signal structs, which need to be considered together to form a full interface. For example, modules found in `pulp-platform/axi` have two ports for each AXI interface: `req_i/o` and `resp_o/i`, each being a struct which contains all the signals.

When specified on the command line via the `--grouping-hint` argument (which can be specified multiple times to add multiple hints), grouping hints use the following syntax: `old1,old2,...,oldN=new`. Whitespace between names, commas, and the equal sign is ignored, and all group names must be non-empty. For example, grouping hints arguments for `axi_cdc` would look like this: `--grouping-hint=src_req_i,src_resp_o=src` and `--grouping-hint=dst_req_o,dst_resp_i=dst`.

The second step is checking which interfaces fit which groups, and ranking how good of a fit they are. Then, interface modes are determined based on port directions, and finally, candidate assignments are applied, going from best to worst. If a port is used by two or more interfaces, the higher scoring one takes precedence, and the lower scoring interfaces are ignored.

Candidates are ranked by the number of required signals of the interface, then by the number of optional signals matched, and finally by how much of its own definition the ports fill. The last rule makes ports without burst signals become `AXI4Lite` rather than `AXI4` or `AXI3`, which also match them. Candidates that still tie are tried in the order of their combined `vendor_library_name_version` identifiers, regardless of the order the definitions were loaded in.

You can limit which interface definitions are considered during inference using the `--inference-interface` argument. The argument takes an interface definition name, and can be specified multiple times. During inference, only interfaces with names matching the specified ones will be considered. For example, to only attempt to infer `AXI4` and `AHB` interfaces, the arguments would look like this: `--inference-interface AXI4` and `--inference-interface AHB`.

CHAPTER

EIGHT

CONFIGURATION

8.1 Configuration file location

The configuration file must be located in one of the following locations:

```
topwrap.yaml
~/.config/topwrap/topwrap.yaml
~/.config/topwrap/config.yaml
```

Additionally, *some* configuration options can be specified in the *design YAML*.

8.2 Configuration precedence

When multiple configuration files are present, the options are evaluated and overridden based on the location of the configuration file. The precedence, from highest to lowest, is as follows:

- Design YAML when evaluating the design
- `topwrap.yaml` in the current working directory
- `~/.config/topwrap/topwrap.yaml` (user-specific configuration)
- `~/.config/topwrap/config.yaml` (fallback configuration)

For example, if `force_interface_compliance` is set to `true` in `~/.config/topwrap/config.yaml` but overridden to `false` in `topwrap.yaml`, the latter value will take precedence when running Topwrap in the directory containing `topwrap.yaml`.

8.2.1 Merging strategies for configuration options

Different configuration options use different merging strategies when multiple configuration files are combined:

- **Override** (e.g. `force_interface_compliance`): The value from the higher-precedence file completely replaces the value in lower-precedence files.
- **Merge** (e.g. `repositories`): Values from all configuration files are merged. For example, repositories defined in `~/.config/topwrap/topwrap.yaml` are combined with repositories defined in `topwrap.yaml`. Note however that if a repository name is defined in two places, the higher priority setting still **overrides** the lower.

8.3 Available config options

The configuration file for Topwrap provides the following options:

- `force_interface_compliance`
  - Type: Boolean
  - Default: `false`
  - Merging strategy: Override

  This option enforces compliance with interface definitions when parsing HDLs.

  For more details, refer to *Interface compliance*.

- `repositories`
  - Type: Dictionary of `name: path`
  - Merging strategy: Merge
  - Specifies repositories to load, with each repository defined as an entry in which:
    - The key is the name of the repository.
    - The value is the *resource path* to the repository.
  - Example of specifying multiple repositories:

    ```
    repositories:
      name_of_repo: file:path_to_repo
      another_repo: file:/absolute/path/to/repo
      remote_repo: git[main]:https://github.com/org/some-ip-repo.git
    ```

  Repositories are used to package and load multiple IP cores and custom interfaces.

  For more information, refer to *User repositories*.

  Deprecated since version 1.0.0: Use `libraries` instead.

- `libraries`
  - Type: Dictionary of `name: uri`
  - Merging strategy: Merge
  - Specifies the libraries to load, with each library defined as an entry in which:
    - The key is the name the library is referred to by.
    - The value is a local directory or a git URL, optionally suffixed with `@<ref>` to pin a branch, tag, or commit.
  - Relative local paths are resolved against the directory of the file declaring them, so a library keeps resolving regardless of the working directory.
  - Example:

    ```
    libraries:
      my_cores: ./cores
    ```

(continues on next page)

(continued from previous page)

```
  remote_cores: https://github.com/org/some-ip-repo.git
  pinned_cores: https://github.com/org/some-ip-repo.git@v1.2.0
```

Entries are normally added with `topwrap library add` rather than by hand.

For more information, refer to *Libraries*.

8.3.1 Example configuration file

Here is a sample configuration file used in the *hierarchy example*

```
force_interface_compliance: true
repositories:
  my_repo: file:./repo
```

CHAPTER

NINE

LIBRARIES

A library is a directory of CAPI2 `.core` files that describe IP cores and interface definitions. Topwrap resolves them with FuseSoC, so a library can be a plain local directory or a git repository.

A library is not laid out by convention: each `.core` file states which of its files Topwrap should read, and what they are.

9.1 Registering a library

```
topwrap library add my_cores ./path/to/cores
topwrap library add remote_cores https://github.com/example/cores.git
```

The name is what the library is referred to by; the URI is a local directory or a git URL. A git URL may carry a `@<ref>` suffix to pin a branch, tag, or commit:

```
topwrap library add pinned https://github.com/example/cores.git@v1.2.0
```

Registering writes the entry to the `libraries:` section of `topwrap.yaml` in the current directory, and checks a git library out into a project-local `.fusesoc-cores/` directory. Because the checkout is project-local, two projects never contend over the same clone.

To fetch the latest upstream state of the registered git libraries:

```
topwrap library update            # all of them
topwrap library update my_cores   # or just one
```

9.2 Inspecting libraries

```
topwrap library list              # registered libraries and their checkout state
topwrap library list modules      # IP modules each one provides
topwrap library list interfaces   # interface definitions each one provides
topwrap library list all          # both
```

Each of the listing commands accepts `--design <file>`, which also takes into account the libraries declared in that design's `config:` section.

9.3 Describing library contents

A `.core` file marks each file it contributes with a `file_type`, which is what tells Topwrap how to read it:

- `topwrapModule` — an *IP description*
- `topwrapInterface` — an *interface definition*

```
CAPI=2:

name : example.com:my-cores:fifo

filesets:
  topwrap:
    files:
        - module.yaml : { file_type : topwrapModule }
```

A single `.core` may contribute any number of modules and interfaces. Files without one of these types are ignored by Topwrap.

The identifier a module or interface is known by comes from the `id:` field of its YAML file. The `.core` name identifies the owning FuseSoC package and may differ, for example when one package provides several Topwrap modules. Generated FuseSoC dependencies use that owning `.core` name.

9.4 Referencing a library module

In a design, `core:` takes the VLNV of a module from any registered library:

```
ips:
  adapter:
    core: vendor:libdefault:axi_axil_adapter:0.1
```

The reference may be partial, and is matched right-aligned onto `vendor:library:name`, so all of these select the same module when unambiguous:

```
core: axi_axil_adapter                # just the name
core: libdefault:axi_axil_adapter     # library and name
core: :libdefault:axi_axil_adapter    # an empty field is a wildcard
```

Spelling out all four fields also constrains the version. A reference that matches nothing, or more than one module, is an error that names the candidates.

`core:` and `file:` are mutually exclusive — each IP instance uses exactly one.

9.5 Declaring libraries in a design

A design can register libraries for its own parse, without touching `topwrap.yaml`:

```
config:
  libraries:
    my_cores: ./cores
```

Relative paths are resolved against the design file. This is how a design stays self-contained for anyone who checks it out. The declaration is scoped to that design operation and does not affect later designs parsed in the same process.

9.6 Creating a library core

`topwrap package --lib` writes a CAPI2 `.core` file next to the generated IP description:

```
cores/widget/
└── rtl/
    ├── include/
    │   └── config.svh
    └── widget.sv
```

```
topwrap package cores/widget/rtl/widget.sv \
  +incdir+cores/widget/rtl/include +define+WIDTH=8 \
  --vlnv example.com:widgets:widget:1.0 --lib --output cores/widget
```

This creates `widget.yaml` and `widget.core` in `cores/widget`.

Interface definitions declared by the input HDL are written as separate YAML files and, with `--lib`, listed in the generated core as `topwrapInterface`. Topwrap saves each full interface VLNV once. Definitions already supplied by a built-in, registered library, or user repository are referenced rather than duplicated in the new package.

Filelists supplied with `--flist` may contain source paths, `+incdir+`, `+define+`, and nested `-f` or `-F` entries. Relative paths inside a filelist are resolved from that filelist's directory.

The generated core records source files in its RTL fileset. Files found under `+incdir+` directories are marked as include files with their include path, while `+define+NAME` and `+define+NAME=VALUE` entries become FuseSoC `vlogdefine` parameters enabled on the default target.

With `--lib`, every source and include directory must be inside the output directory. If they are elsewhere, pass `--copy` to copy them into it while packaging. Source files are copied by filename, and include directories by directory name, so the generated `.core` refers to the copies. For example:

```
topwrap package ../rtl/widget.sv +incdir+../rtl/include \
  --lib --copy --output cores/widget
```

Without `--copy`, Topwrap reports an error without writing the package when sources are outside output directory. You can also choose a common parent as `--output` or move the sources

beneath the package directory. Keeping all files alongside the `.core` avoids references to files outside its directory, which FuseSoC is deprecating.

CHAPTER

TEN

CONSTRUCTING, CONFIGURING AND LOADING REPOSITORIES

Deprecated since version 1.0.0: User repositories and the `topwrap repo` commands are being removed. Use *libraries* instead, and reference their cores with `core:` in a design.

By using Topwrap repositories, you can package and load different resources and use them in your designs.

You can specify the repositories to be loaded each time Topwrap runs, by listing them in a *configuration file* or by supplying a path to one or more repository directories using the `--repo` CLI argument.

A single repository can store multiple resource types. Each supported type is associated with a subdirectory in the top-level repository directory.

A sample repository can be found in examples/user_repository.

10.1 Supported resource types

10.1.1 IP cores

- Path: `repo_dir/cores/`

The "Core" resource represents a self-contained IP core/module that can then be used as a component in a custom design created by the user. It corresponds to *Internal Representation* Module object.

A resource is stored in the `module.yaml` file, it follows *IP description format*. It contains information about the HDL module and should be generated by `topwrap repo parse`, but it can be done manually. The `module.yaml` should be edited by a user to fill in information's that cannot be deduced from the HDL file or when there is a custom interface that does not have an Interface Definition.

10.1.2 Interface definitions

- Path: `repo_dir/interfaces/`

This resource represents a custom definition of an interface that can be used to make connections between IP cores.

Each such definition is stored as a separate *interface description file* under the `interfaces` subdirectory.

10.2 CLI

10.2.1 `topwrap repo init`

```
topwrap repo init [OPTIONS] REPO_NAME REPO_PATH
```

This command creates and initializes a new repository named `REPO_NAME` in a chosen `REPO_PATH`.

By default, it also adds the entry with the new repository into the local Topwrap *configuration file*. If the file does not exist, it gets created. This behavior can be disabled using the `--no-config-update` CLI flag.

10.2.2 `topwrap repo list`

```
topwrap repo list
```

This will print out names and paths of all repositories from which resources are loaded and can be used in other Topwrap commands.

10.2.3 `topwrap repo parse`

```
topwrap repo parse [OPTIONS] REPOSITORY SOURCES...
```

This command will parse all supported source files given in the `SOURCES` argument, extract IP cores from them, and save them in a repository given by the `REPOSITORY` argument.

Note: When a directory path is given among the `SOURCES` argument, it gets recursively expanded and all nested files are extracted from it.

To see the listing of all supported `OPTIONS`, see `topwrap repo parse --help`

10.3 Using the open source IP cores library with Topwrap

Topwrap comes with built-in support for an extensive library of open source IP cores available through the FuseSoC package manager, which also serves as a build system. This library offers a wide range of reusable IP cores for various applications, enabling easy integration into Topwrap projects. Topwrap simplifies the process of accessing, downloading, and packaging these IP cores, making them readily available for local use in your designs.

To include an IP core from the open source library, there are two methods:

1. **Select the Desired Core**: Browse the available cores (cores_export artifact).
2. **Download and build all available cores**: Use Topwrap's package management command:

```
just package
```

This will download and parse all the cores from Fusesoc into `build/fusesoc_workspace/build/export/cores/`, making them accessible from within Topwrap.

You can learn more about Topwrap integration with FuseSoC *here*

CHAPTER

ELEVEN

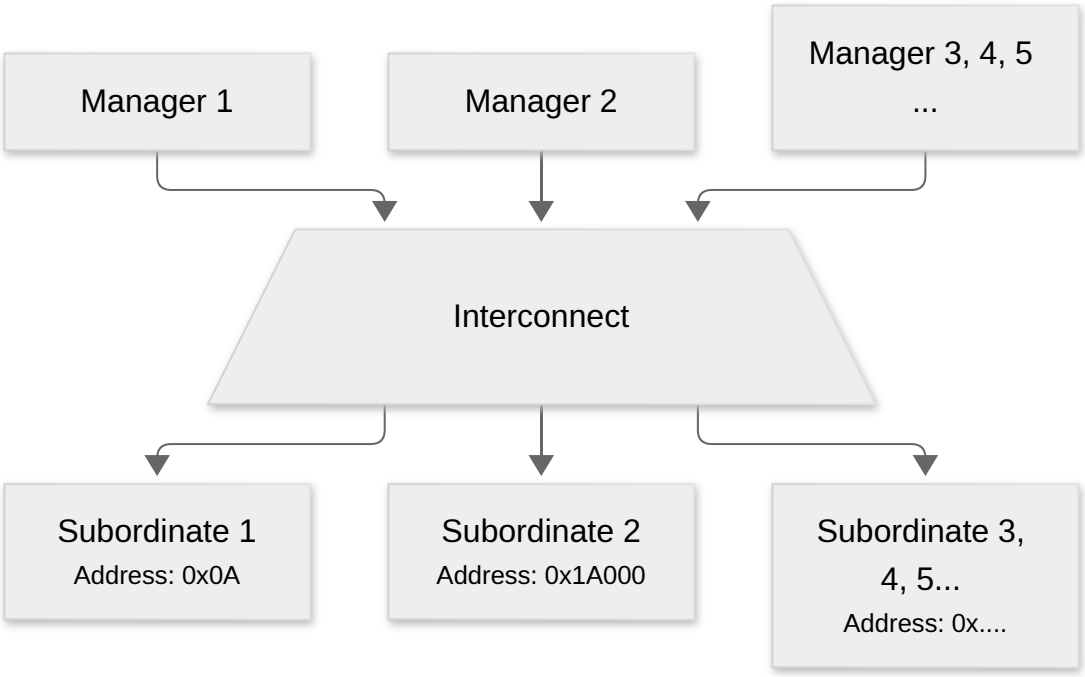
INTERCONNECT GENERATION

Interconnects enable the connection of multiple interfaces in a many-to-many topology, as opposed to the traditional one-to-one manager-subordinate connection. This approach facilitates data transmission between multiple IP cores over a single interface, with the interconnect serving as an middle-man.

> **Warning:** Interconnect generation is an experimental feature.
>
> Currently, creating and showing them is not possible in the Topwrap GUI.

Each manager can communicate with any subordinate connected to the interconnect. Every connected subordinate must be assigned a predefined address range, allowing the interconnect to route data based on the address specified by the manager.

A typical interconnect topology diagram is shown below.



In order to generate an interconnect, you have to describe its configuration in the *Design description* under the `interconnects` key in the following format, as specified below:

11.1 Format

The `interconnects` key must be a direct descendant of the `design` key in the *Design description*.

```
interconnects:
  {interconnect1_name}:
    # Specify clock and reset to drive the interconnect with
    clock: [{ip_name, clk_port_name}]
    reset: [{ip_name, rst_port_name}]
    # Alternatively you can specify a connection to an external port of this␣
↪design:
    # clock: ext_clk_port_name
    # reset: ext_rst_port_name

    # Specify the interconnect type.
    # See the "Supported interconnect types" section below for available types
    # and their characteristics
    type: {interconnect_type}

    # custom parameter values for the specific interconnect type
    parameters:
      {parameters_name1}: parameters_value1
      ...

    # specify managers and their interfaces connected to the bus
    managers:
      {manager1_name}:
        - {manager1_interface1_name}
        ...
      ...

    # specify subordinates, their interfaces connected to the bus and their bus␣
↪parameters
    subordinates:
      {subordinate1_name}:
        {subordinate1_interface1_name}:
          # requests in address range [address, address+size) will be routed to␣
↪this interface
          address: {start_address}
          size: {range_size}
        ...
      ...

    # additionally, in addition to the subordinates section, a map from the␣
↪memory_maps section can be specified
    # the subordinates is updated with data present in the selected memory map
    memory_map: {map_name}
  ...
```

- `clock` - a port that has the clock signal
- `reset` - a port that has the reset signal

- `type` - a type of interconnect, listed in [Supported interconnect types][#supported-interconnect-types]
- `parameters` - a list of parameters specific to the interconnect
- `managers` - a list of interfaces used as managers, each ip module can have multiple interfaces connected
- `subordinates` - a list of interfaces used as subordinates, each ip module can have multiple interfaces connected. Address and size is specified in bytes.
- `memory_map` - the memory map used to update subordinates

11.2 Supported interconnect types

11.2.1 `Wishbone Round-Robin`

This interconnect only supports Wishbone interfaces for managers and subordinates. It supports multiple managers, but only one of them can drive the bus at a time (i.e. only one transaction can be happening on the bus at any given moment). A round-robin arbiter decides which manager can currently drive the bus.

Parameters

Subordinate

- `addr_width` - bit width of the address line (addresses access `data_width`-sized chunks)
- `data_width` - bit width of the data line
- `granularity` - access granularity - the smallest unit of data transfer that the interconnect can transfer. Must be: 8, 16, 32, 64
- `features` - optional, list of optional wishbone signals, can contain: `err`, `rty`, `stall`, `lock`, `cti`, `bte`

Known limitations

The currently known limitations are:

- only word-sized addressing is supported (in other words - consecutive addresses can only access word-sized chunks of data)
- crossing clock domains, down-converting (initiating multiple transactions on a narrow bus per one transaction on a wider bus) and up-converting are not supported

11.2.2 AXI

This interconnect only supports AXI4 interfaces for managers and subordinates. Uses pulp-platform/axi as dependency.

> **Caution:** `Generator` for SystemVerilog for this interconnect needs plugin mechanism to work, it isn't currently implemented. As workaround Topwrap downloads scripts to `$XDG_CACHE_HOME/topwrap/pulp_axi_scripts` or `~/.local/cache/topwrap/pulp_axi_scripts`.

Parameters

- `atop` - `True` or `False`, enables or disables atomic operations

Manager

- `id_width` - bit width of ID signals

CHAPTER

TWELVE

USING FUSESOC FOR AUTOMATION

Topwrap uses the FuseSoC package manager and build tools for HDL code to automate project generation and the build process. When `topwrap generate` is used with the `--fusesoc` option, it generates a FuseSoC `.core` file along with the top-level wrapper.

12.1 Default tool for synthesis, bitstream generation and programming the FPGA

Topwrap assumes that you're using Vivado. You can change the default tool to something other than Vivado by modifying the generated `.core` file.

12.2 Additional build options

To enable `.core` file generation, supply `--fusesoc` to `topwrap generate`:

```
topwrap generate design.yaml --fusesoc
```

If you have any additional directories with HDL sources or constraint files required for synthesis, you can specify them using the repeatable `--sources` option. Sources from these directories get appended to the `filesets.rtl.files` entry in the generated FuseSoC `.core` file.

```
topwrap generate design.yaml --fusesoc --sources ./srcs_v --sources ./srcs_vhd
```

If you're targeting a specific FPGA chip, you can additionally specify its number using the `--part` option.

The supplied part number is passed to the FuseSoC `.core` file. It is included in the `targets.default.tools.vivado.part` entry, which is then supplied to Vivado when you run FuseSoC and use the default target. This can be any part number available to your local Vivado installation.

```
topwrap generate design.yaml --fusesoc --part 'xc7z020clg400-3'
```

Library-backed IPs used by the design are emitted as dependencies of the generated core. FuseSoC therefore includes their own RTL filesets when it builds the design.

Use `--target-dir` to change the output root. The wrapper and `.core` file are written to its `src` subdirectory.

The generated core currently defines a default Vivado target. Edit the generated `.core` file if another tool, additional targets or hooks are required.

12.3 Synthesis

After generating the `.core` file, you can run FuseSoC to generate the bitstream and program the FPGA:

```
fusesoc --cores-root build/src run {design_name}
```

This requires having a suitable backend tool that is specified in the `.core` file under `targets.default.tools` available in your `PATH` (e.g. Vivado).

CHAPTER

THIRTEEN

PLUGIN SUPPORT

Topwrap uses a plugin mechanism to allow external software to customize or extend the capabilities of Topwrap. Plugins take the form of Python modules.

13.1 Ways to get plugins

There are two ways to get plugins, install and configure pre-existing plugins or build your own.

13.1.1 Plugin repositories

Antmicro hosts plugins in https://github.com/antmicro/topwrap-utils.

13.1.2 Building your own

Building plugins on your own requires insight into how Topwrap works. You are encouraged to review the Developer's guide on *plugins* and to use an existing plugin as a start-of point.

13.2 Plugin configurations

Plugins are configured both at the IP-core level and the design level. Both files provide a configurable property `extensions` that is used to pass options to the plugin. See the chapter on *creating designs* for more info about the format of configuration options. Also review the documentation for each plugin, which will describe plugin-specific settings.

CHAPTER

FOURTEEN

SETUP

It is required for developers to keep the current code style and it is recommended to frequently run tests.

In order to set up the development environment, install all the optional dependency groups as specified in `pyproject.toml`, which also includes `pre-commit`:

```
uv sync --extra all
source .venv/bin/activate
```

The `--extra` option is for installing in editable mode - meaning changes in the code under development will be immediately visible when using the package.

CHAPTER

FIFTEEN

CODE STYLE

`pre-commit` performs automatic formatting and linting of the code.

15.1 Lint with pre-commit

Alternatively, use `pre-commit` to perform the same job. `pre-commit` hooks need to be installed:

```
pre-commit install
```

Now, each use of `git commit` in the shell will trigger actions defined in the `.pre-commit-config.yaml` file. `pre-commit` is easily deactivated with a similar command:

```
pre-commit uninstall
```

If you wish to run `pre-commit` asynchronously, use:

```
pre-commit run --all-files
```

Note: `pre-commit` by default also runs `ruff` and `codespell` sessions.

15.2 Tools

Tools used in the Topwrap project for maintaining the code style:

- `pre-commit` is a framework for managing and maintaining multi-language pre-commit hooks.
- `ruff` is a Python linter and code formatter.
- `codespell` is a Python tool to fix common spelling mistakes in text files

CHAPTER

SIXTEEN

TESTS

Topwrap functionality is validated with tests that leverage the `pytest` library.

16.1 Test execution

The tests are located in the `tests` directory. All tests can be run with `just` by specifying the `test` task:

```
just test
```

To force a specific Python version and avoid running tests for all listed versions, pass the version to the `test` task:

```
just test 3.11
```

Tests can also be launched without `just` by executing:

```
python -m pytest
```

> **Warning:** When running tests by invoking `pytest` directly, tests are ran only on the locally selected Python interpreter. As the CI runs on all supported Python versions, it's recommended to run tests with `just` on all versions before pushing.

Ignoring a particular test can be performed with `--ignore=test_path`, e.g:

```
python -m pytest --ignore=tests/tests_build/test_interconnect.py
```

To run a specific test, use the `-k test_name` flag, e.g:

```
python -m pytest -k TestKpmSpecificationBackend
```

For debugging purposes, Pytest captures all output from the test and displays it when all tests are completed. To see the output immediately, pass the `-s` flag to pytest:

```
python -m pytest -s
```

16.2 Test coverage

Test coverage is automatically generated when running tests with `just`. When invoking `pytest` directly, it can be generated with the `--cov=topwrap` flag. This will generate a summary of coverage, displayed in the CLI.

```
python -m pytest --cov=topwrap
```

Additionally, the summary can be generated in HTML with the flags `--cov=topwrap --cov-report html`, where lines that were not covered by tests can be browsed:

```
python -m pytest --cov=topwrap --cov-report html
```

The generated report is available at `htmlcov/index.html`

16.3 Updating kpm test data

All `kpm` data from examples can be generated using `just`. This is useful when changing Topwrap functionality relating to kpm, as it avoids manually changing test data in every sample. Users can either update of example data such as the specification or update everything (dataflows, specifications).

To update everything run:

```
just update-testdata --dataflow --specification
```

To update only specifications run:

```
just update-testdata --specification
```

Valid options for the `update-testdata` task, are:

- `specification`
- `dataflow`

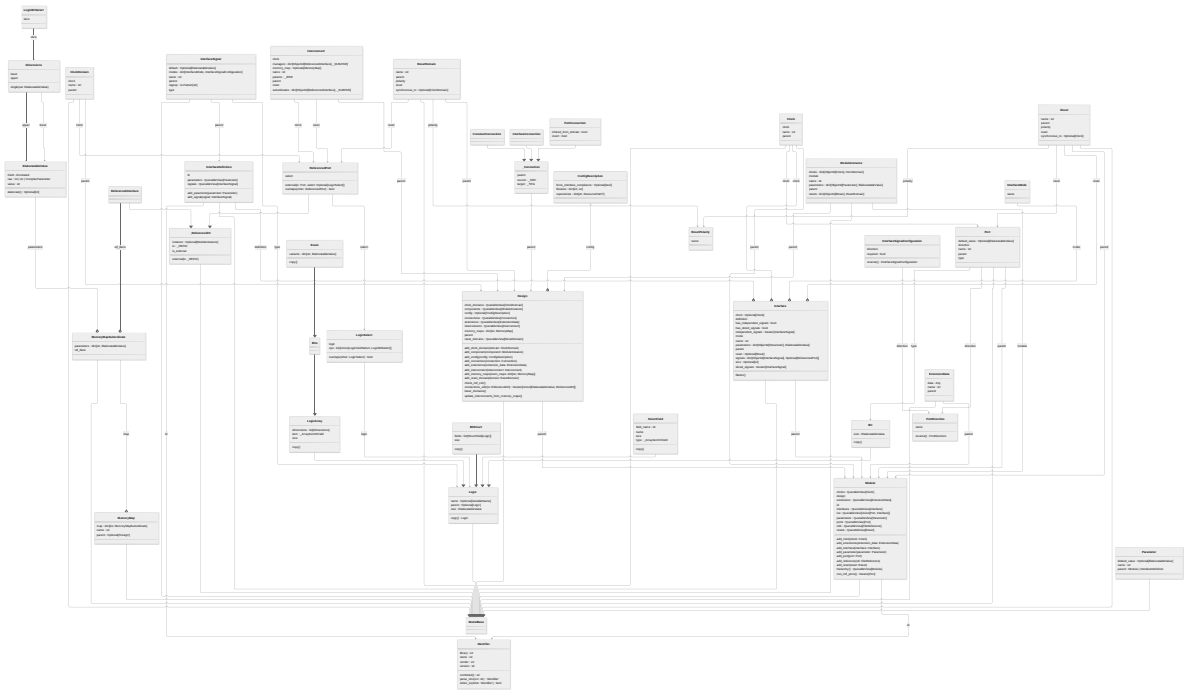
CHAPTER

SEVENTEEN

INTERNAL REPRESENTATION

Topwrap uses a custom object hierarchy, further called “internal representation” or “IR”, in order to store block design and related data in memory and operate on it.

17.1 Class diagram



17.2 Frontend & Backend

The `Frontend` based classes converts external formats, such as SystemVerilog, VHDL, KPM or IP-XACT into the IR. Complementarily, the `Backend` based classes convert our IR into external formats, such as SystemVerilog, YAML, KPM or IP-XACT.

The reason for separating the logic like this is to be able to easily add support for multiple fronteds and backends formats and make them interchangeable.

17.2.1 API interface

class FrontendParseStrInput(name: *str*, content: *str*)

Bases: object

name : str

content : str

class FrontendMetadata(name: str, file_association: Iterable[str] = <factory>)

Bases: object

name : str

A short name for this frontend used in contexts when a specific frontend needs to be identified for example in a configuration file or CLI

file_association : Iterable[str]

File extensions associated with this frontend in the form of a set of extensions (e.g. {*".yaml", ".yml"*})

class FrontendParseOutput(modules: list[~topwrap.model.module.Module] = <factory>, interfaces: list[~topwrap.model.interface.InterfaceDefinition] = <factory>, positions: dict[~topwrap.model.misc.Identifier, ~topwrap.backend.kpm.common.Positions] = <factory>)

Bases: object

Variables

modules
Modules parsed from the input sources. A parsed module is included even if its identifier matches a module provided to the frontend during initialization.

interfaces
Interface definitions parsed from the input sources.

positions
Mapping from module identifiers to the positions of elements in their corresponding KPM graphs.

modules : list[*Module*]

interfaces : list[*InterfaceDefinition*]

positions : dict[*Identifier*, Positions]

exception FrontendParseException

Bases: TranslationError

Exception occurred during parsing sources by the frontend

class Frontend(modules: *Iterable[*Module*]* = (), interfaces: *Iterable[*InterfaceDefinition*]* = ())

Bases: ABC

parse_str(sources: *Iterable[str | FrontendParseStrInput]*) → *FrontendParseOutput*

Parse a collection of string sources into IR modules

Parameters

sources: Iterable[str | *FrontendParseStrInput*]
Iterable of string sources. Items can either be a plain *str* with the content or the *FrontendParseStrInput* dataclass where additional parameters related to the source can be specified.

abstract property metadata : *FrontendMetadata*

Return metadata about this frontend such as its file associations

abstract parse_files(sources: *Iterable[Path]*, *, include_dirs: *Iterable[Path]* = ()) → *FrontendParseOutput*

Parse a collection of source files into IR modules

Parameters

sources: Iterable[Path]
Collection of paths to sources

exception BackendParseException

Bases: `TranslationError`

Exception occurred during dumping internal representation by the backend

class BackendOutputInfo(filename: *str*, content: *str*)

Bases: `object`

Output of the backend's serialization process

filename : str

The filename for this output as suggested by the specific backend E.g. `dataflow.kpm.json` or `axibridge.sv`

content : str

The content of the file generated by the backend

save(path: *Path*)

Save this output as a file on the filesystem under a given path

Parameters

path: Path
The path where to save the file. If it points only to a directory, then the file is saved with the `self.filename` name.

class Backend(existing_interfaces: *set[Identifier] | None* = **None**)

Bases: `ABC`, `Generic[_MODFORMAT]`

The base class for backend implementations used to convert our IR represented by the `Module` class into various external formats, represented by the generic parameter `_MODFORMAT`.

existing_interfaces : Set[*Identifier*]

List of InterfaceDefinition's that already exists in source files or has been serialized, it is a hint for backend implementations

abstract represent(module: Module, /) → _MODFORMAT

Convert the IR into an arbitrary custom external format.

abstract serialize(repr: *_MODFORMAT*, /) → Iterator[*BackendOutputInfo*]

Serialize the custom format object into one or more text files, represented by their content, alongside with additional information about them, like the suggested filename.

17.3 Module

class Module(*, id: Identifier, refs: *Iterable[*FileReference*]* = (), ports: *Iterable[*Port*]* = (), parameters: *Iterable[*Parameter*]* = (), interfaces: *Iterable[*Interface*]* = (), clocks: *Iterable[*Clock*]* = (), resets: *Iterable[*Reset*]* = (), design: Design | *None* = **None**, extensions: *Iterable[*ExtensionData*]* = ())

Bases: ModelBase

The top-level class of the IR. It fully represents the public interface of a HDL module, holding definitions of all of its structured ports and interfaces, parameters, and optionally its inner block design if available.

id : *Identifier*

property design

Returns the optional inner block design of this module

property ports : *QuerableView*[*Port*]

property parameters : *QuerableView*[*Parameter*]

property interfaces : *QuerableView*[*Interface*]

property ios : *QuerableView*[*Port* | *Interface*]

Returns a combined view on both ports and interfaces

property refs : *QuerableView*[*FileReference*]

Returns references to external files that define this module, if any. This information is generally added by the respective frontend used to parse this module.

property clocks : *QuerableView*[*Clock*]

property resets : *QuerableView*[*Reset*]

property extensions : *QuerableView*[*ExtensionData*]

add_port(port: Port)

add_parameter(parameter: Parameter)

add_interface(interface: Interface)

add_reference(ref: FileReference)

add_clock(clock: Clock)

add_reset(reset: Reset)

add_extensions(extension_data: ExtensionData)

non_intf_ports() → Iterator[*Port*]

Yield ports that don't realise signals of any interface

hierarchy() → *QuerableView*[*Module*]

Traverses the entire hierarchy tree of this module in order, using a BFS algorithm. Returns every unique module encountered on the way. The result also includes the current module.

17.4 Design

class ModuleInstance(*, name: *VariableName*, module: Module, parameters: *Mapping*[ObjectId[Parameter], ElaboratableValue] = {}, clocks: *Mapping*[ObjectId[Clock], ClockDomain] = {}, resets: *Mapping*[ObjectId[Reset], ResetDomain] = {})

Bases: ModelBase

Represents an instantiated module with values supplied for its appropriate respective parameters. This class is necessary to differentiate multiple instances of the same module in a design.

parent : *Design*

Reference to the design that contains this component

name : VariableName

The name of this instance. It corresponds to "instance_name" in this exemplary Verilog construct: `MODULE #(.WIDTH(32)) instance_name (.clk(clk));`

module : *Module*

The module that this is an instance of. Corresponds to "MODULE" in the Verilog construct defined above.

parameters : dict[*ObjectId*[*Parameter*], *ElaboratableValue*]

Concrete parameter values for the module that this is an instance of. Corresponds to "#(.WIDTH(32))" in the above Verilog construct.

clocks : dict[*ObjectId*[*Clock*], *ClockDomain*]

Assignments between this instance's clock inputs and clock domains.

resets : dict[*ObjectId*[*Reset*], *ResetDomain*]

Assignments between this instance's reset inputs and reset domains.

class ClockDomain(*, name: *str*, clock: ReferencedPort)

Bases: ModelBase

This class represents a clock domain defined within a design.

parent : *Design*

Reference to the design that contains this clock domain.

name : VariableName

clock : *ReferencedPort*

Reference to a port acting as the source for this clock domain.

class ResetDomain(*, name: *str*, reset: ReferencedPort, polarity: ResetPolarity, synchronous_to: ClockDomain | *None* = **None**)

Bases: ModelBase

This class represents a reset domain defined within a design.

parent : *Design*

Reference to the design that contains this reset domain.

name : VariableName

reset : *ReferencedPort*

Reference to a port acting as the source for this reset domain.

polarity : *ResetPolarity*

Polarity of the reset signal.

synchronous_to : *ClockDomain* | None

Clock domain to which this reset domain is synchronous to. None if the reset signal is asynchronous.

exception DesignDomainException

Bases: Exception

Error relating to clock and reset domains within a design.

class Design(*, components: *Iterable[ModuleInstance]* = (), interconnects: *Iterable[Interconnect]* = (), connections: *Iterable[ConstantConnection | PortConnection | InterfaceConnection]* = (), clock_domains: *Iterable[ClockDomain]* = (), reset_domains: *Iterable[ResetDomain]* = (), memory_maps: *dict[str, MemoryMap]* | *None* = **None**, extensions: *Iterable[ExtensionData]* = (), config: *ConfigDescription* | *None* = **None**)

Bases: ModelBase

This class represents the inner block design of a specific Module. It consists of instances of other modules (components) and connections between them, each other, and external ports of the module that this design represents.

parent : *Module*

memory_maps : dict[str, MemoryMap]

config : ConfigDescription | None

property components : *QuerableView*[*ModuleInstance*]

property interconnects : *QuerableView*[*Interconnect*]

property connections : *QuerableView*[*ConstantConnection* | *PortConnection* | *InterfaceConnection*]

property clock_domains : *QuerableView*[*ClockDomain*]

property reset_domains : *QuerableView*[*ResetDomain*]

property extensions : *QuerableView*[*ExtensionData*]

add_component(component: ModuleInstance)

add_interconnect(interconnect: Interconnect)

add_connection(connection: ConstantConnection | PortConnection | InterfaceConnection)

add_clock_domain(domain: ClockDomain)

add_reset_domain(domain: ResetDomain)

add_extensions(extension_data: ExtensionData)

add_memory_maps(mem_maps: *dict[str, MemoryMap]*)

add_config(config: *ConfigDescription*)

connections_with(io: ReferencedPort | ReferencedInterface) → Iterator[*ElaboratableValue* | *ReferencedPort* | *ReferencedInterface*]

Yields everything that is connected to a given IO.

Parameters

io: ***ReferencedPort* | *ReferencedInterface***
The IO of which connections to yield.

lower_domains()

Lower clock/reset domain assignments into regular port connections.

check_intf_cdc()

update_interconnects_from_memory_maps()

17.5 Interface

class InterfaceMode(value, names=**None**, *, module=**None**, qualname=**None**, type=**None**, start=1, boundary=**None**)

Bases: Enum

MANAGER = 'manager'

SUBORDINATE = 'subordinate'

UNSPECIFIED = 'unspecified'

class InterfaceSignalConfiguration(direction: PortDirection, required: *bool*)

Bases: object

Holds the mode-specific configuration of an interface signal

direction : *PortDirection*

The direction of this signal in a given `InterfaceMode`

required : bool

Whether this signal is required or optional

reverse() → *InterfaceSignalConfiguration*

Returns a configuration with the opposite direction.

class InterfaceSignal(*, name: *str*, regexp: *Pattern[str]*, type: Logic, default: ElaboratableValue | *None* = **None**, modes: *Mapping[*InterfaceMode, InterfaceSignalConfiguration*]* = {})

Bases: `ModelBase`

This class represents a signal in an interface definition. E.g.: The `awaddr` signal in the AXI interface etc.

parent : *InterfaceDefinition*

The definition that this signal belongs to

name : VariableName

regexp : re.Pattern[str]

While automatically deducing interfaces, if an arbitrary signal's name matches this regular expression then that signal is considered as a candidate for realizing this signal definition

type : *Logic*

The logical type of this signal. Fulfills the same function as `Port.type`

default : *ElaboratableValue* | None = **None**

The default value for this signal, if any

modes : dict[*InterfaceMode*, *InterfaceSignalConfiguration*]

A dictionary of modes for this signal and their specific configurations E.g.: A signal that only has an `InterfaceMode.MANAGER` entry in this dictionary means that it's valid only on the manager's side.

class InterfaceDefinition(*, id: Identifier, signals: *Iterable[*InterfaceSignal*]* = (), parameters: *Iterable[*Parameter*]* = ())

Bases: `ModelBase`

This represents a definition of an entire interface/bus. E.g. AXI, AHB, Wishbone, etc.

id : *Identifier*

property signals : *QuerableView*[*InterfaceSignal*]

A list of signal definitions that make up this interface E.g. awaddr, araddr, wdata, rdata, etc.... in AXI

property parameters : *QuerableView*[*Parameter*]

add_signal(signal: InterfaceSignal)

add_parameter(parameter: Parameter)

class Interface(name: *str*, mode: InterfaceMode, definition: InterfaceDefinition, signals: *dict[*ObjectId*[*InterfaceSignal*]*, ReferencedPort | *None]*, clock: Clock | *None* = **None**, reset: Reset | *None* = **None**, size: *int* | *None* = **None**, parameters: *Mapping[*ObjectId*[*Parameter*]*, ElaboratableValue*]* = {})

Bases: ModelBase

A realised instance of an interface that can be connected with other interface instances through InterfaceConnection.

The relationship between this class and InterfaceDefinition is similar to the relationship between ModuleInstance and Module classes.

parent : *Module*

The module definition that contains this interface instance

name : VariableName

Name for this interface instance

mode : *InterfaceMode*

The mode of this instance (e.g. manager/subordinate)

definition : *InterfaceDefinition*

The definition of the interface that this is an instance of

signals : dict[*ObjectId*[*InterfaceSignal*], *ReferencedPort* | None]

Realization of signals defined in this interface. A signal can be realized either by:

- Slicing an already existing external port of the module that this instance belongs to (self.parent), in that case the value in this dictionary is the aforementioned port reference.
- Independently, meaning that whenever necessary, e.g. during output generation by a Backend that does not support interfaces, an arbitrarily generated port based on the InterfaceSignal.type should be generated to represent it. In that case the value in this dictionary is None.

For more information about sliced and independent signals, see *A note on "sliced" vs. "independent" signals*

If an entry for a specific signal that exists in the definition is not present in this dictionary, then that signal will not be realized at all. E.g. when it was configured as optional or was given a default value in InterfaceSignalConfiguration.

clock : *Clock* | None

Clock signal used by this interface.

reset : *Reset* | None

Reset signal used by this interface.

size : int | None

parameters : dict[*ObjectId*[*Parameter*], *ElaboratableValue*]

Concrete parameter values for the interface that this is an instance of.

property independent_signals : Iterator[*InterfaceSignal*]

Yields signals that are not realized by any external ports of the Module

For more information about sliced and independent signals, see *A note on "sliced" vs. "independent" signals*

property sliced_signals : Iterator[*InterfaceSignal*]

Yields signals that are realized by external ports of the Module

For more information about sliced and independent signals, see *A note on "sliced" vs. "independent" signals*

property has_independent_signals : bool

property has_sliced_signals : bool

flatten()

Transforms all independently realized signals of this interface into sliced signals, creating the necessary ports and adding them to the parent module.

17.6 Interface inference

17.6.1 Port selector

class PortSelectorField(**, load_default: ~typing.Any = <marshmallow.missing>, missing: ~typing.Any = <marshmallow.missing>, dump_default: ~typing.Any = <marshmallow.missing>, default: ~typing.Any = <marshmallow.missing>, data_key: str | None = None, attribute: str | None = None, validate: ~typing.Callable[[~typing.Any], ~typing.Any] | ~typing.Iterable[~typing.Callable[[~typing.Any], ~typing.Any]] | None = None, required: bool = False, allow_none: bool | None = None, load_only: bool = False, dump_only: bool = False, error_messages: dict[str, str] | None = None, metadata: ~typing.Mapping[str, ~typing.Any] | None = None, **additional_metadata*)

Bases: `Field`

class PortSelectorOp(value, names=**None**, *, module=**None**, qualname=**None**, type=**None**, start=1, boundary=**None**)

Bases: `Enum`

FIELD = 1

SLICE = 2

class PortSelector(*port: str, ops: tuple[tuple[<PortSelectorOp.FIELD: 1>, str] | tuple[<PortSelectorOp.SLICE: 2>, tuple[int, int]], ...]*)

Bases: `object`

A selector of (potentially a smaller part of) a port.

The selector starts with an external port name, and is followed by one or more of the following operations:

- field selection (e.g. `.some_field`),
- array indexing/slicing (e.g. `[1]` or `[3:0]`).

For example, accessing a part one of the manager port fields of an instance of `axi_demux` from pulp-platform/axi would look like this: `mst_reqs_o[2].ar.addr[3:0]`.

port : str

Name of the module port this selector targets.

ops : tuple[tuple[<PortSelectorOp.FIELD: 1>, str] | tuple[<PortSelectorOp.SLICE: 2>, tuple[int, int]], ...]

Tuple of operations to be performed on the port.

classmethod from_str(sel: *str*) → *PortSelector*

Parse a port selector string into an instance of `PortSelector`.

classmethod from_referenced_port(ref: ReferencedPort) → *PortSelector*

Turn a `ReferencedPort` instance into an instance of `PortSelector`.

make_referenced_port(module: Module, mode: InterfaceMode, signal: InterfaceSignal) → *ReferencedPort*

Construct a `ReferencedPort`, potentially with an instance of `LogicSelect` based on the information contained in this selector.

17.6.2 Inference

parse_grouping_hints(grouping_hints: *Iterable[str]*) → dict[str, str]

Parse user-facing grouping hints into a dictionary for `infer_interfaces_from_module()`.

The incoming hints are stored as they are specified on the command line, in the form of: `"old1,old2,...,oldN=new"`, and are parsed into dictionaries with entries like this: `{ "old1": "new", "old2": "new", ..., "oldN": "new" }`

class InterfaceInferenceOptions(min_group_size: int = 2, prefix_split_tokens: list[str] = <factory>, prefix_consider_camel_case: bool = True)

Bases: `object`

Configuration options for `infer_interfaces_from_module()`.

min_group_size : int = 2

Minimum number of ports that must be in a group for it to be considered.

prefix_split_tokens : list[str]

Tokens on which prefixes are split on.

prefix_consider_camel_case : bool = **True**

Should camel case prefixes be considered.

infer_interfaces_from_module(module: Module, intf_defs: *Collection[InterfaceDefinition]*, grouping_hints: *dict[str, str] | None* = **None**, options: InterfaceInferenceOptions | *None* = **None**)

Perform interface inference. Attempts to infer interfaces that the given module contains, and adds them to the module.

Parameters

module: *Module*
Module to perform inference on.

intf_defs: Collection[*InterfaceDefinition*]
Interface definitions to consider.

grouping_hints: dict[str, str] | None = None
Hints for merging discovered groups into one.

options: *InterfaceInferenceOptions* | None = None
Configuration options for inference.

17.7 Connections

class PortDirection(value, names=None, *, module=None, qualname=None, type=None, start=1, boundary=None)

Bases: Enum

IN = 'in'

OUT = 'out'

INOUT = 'inout'

reverse() → *PortDirection*

class Port(*, name: *str*, direction: PortDirection, type: Logic | *None* = None, default_value: ElaboratableValue | *None* = None)

Bases: ModelBase

This represents an external port of a HDL module that can be connected to other ports or constant values in a design.

parent : *Module*
The module definition that exposes this port

name : VariableName

direction : *PortDirection*

type : *Logic*
The type of this port. (Bit, BitStruct, LogicArray etc.)

default_value : *ElaboratableValue* | None
The default value to assign to this port if left unconnected.

class ReferencedPort(*, instance: ModuleInstance | *None* = None, io: Port, select: LogicSelect | *None* = None)

Bases: _ReferencedIO[Port]

Represents a correctly typed reference to a port

select : *LogicSelect*

classmethod external(io: Port, select: LogicSelect | *None* = None)
A shortcut constructor for a reference to a top-level IO of the current module

overlaps(other: ReferencedPort) → bool

Checks for overlap between two port references.

See docs for `overlaps()`

class ReferencedInterface(*, instance: ModuleInstance | *None* = **None**, io: *_REFIO*)

Bases: `_ReferencedIO[Interface]`

Represents a correctly typed reference to an interface

class ConstantConnection(source: *_SRC*, target: *_TRG*)

Bases: `_Connection[ElaboratableValue, ReferencedPort]`

Represents a connection between a constant value and a port of a component

class PortConnection(source: *_SRC*, target: *_TRG*, invert: *bool* = **False**, infered_from_domain: *bool* = **False**)

Bases: `_Connection[ReferencedPort, ReferencedPort]`

Represents a connection between two ports of some components

invert : bool = **False**

Specifies whether the connection should be bitwise inverted.

infered_from_domain : bool = **False**

Specifies whether the connection was created by lowering domains.

class InterfaceConnection(source: *_SRC*, target: *_TRG*)

Bases: `_Connection[ReferencedInterface, ReferencedInterface]`

Represents a connection between two interface references.

The `source` and `target` fields identify the endpoints of the connection, but do not determine signal direction. The direction of each interface signal is defined by the interface's mode and its `InterfaceSignalConfiguration`.

class Clock(*, name: *str*, clock: Port)

Bases: `ModelBase`

Represents a clock input

parent : *Module*

The module definition that this clock belongs to.

name : str

clock : *Port*

class ResetPolarity(value, names=**None**, *, module=**None**, qualname=**None**, type=**None**, start=1, boundary=**None**)

Bases: `Enum`

ACTIVE_LOW = `'active low'`

ACTIVE_HIGH = `'active high'`

class Reset(*, name: *str*, reset: Port, polarity: ResetPolarity, synchronous_to: Clock | *None* = **None**)

Bases: ModelBase

Represents a reset input

parent : *Module*

The module definition that this reset belongs to.

name : str

reset : *Port*

polarity : *ResetPolarity*

Polarity of the reset signal.

synchronous_to : *Clock* | None

Clock to which this reset is synchronous to. None if the reset is asynchronous.

17.8 HDL Types

class Dimensions(upper: ~topwrap.model.misc.ElaboratableValue = <factory>, lower: ~topwrap.model.misc.ElaboratableValue = <factory>)

Bases: object

A pair of values representing bounds for a single dimension of a Logic type. Verilog examples: - logic -> upper == lower == 0 - logic[31:0] -> upper == 31, lower == 0 - logic[0:64] -> upper == 0, lower == 64

upper : *ElaboratableValue*

lower : *ElaboratableValue*

classmethod single(val: ElaboratableValue)

class Logic(name: *str* | *None* = **None**)

Bases: ModelBase, ABC

An abstract class representing anything that can be used as a logical type for a port or a signal in a module.

parent : *Logic* | None

The parent of this type. Used to create a reference tree between complex type definitions, for example between LogicArray and the inner type it contains or between a BitStruct and its fields. Is None when this represents a top-level type

name : VariableName | None

abstract copy() → *Logic*

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

abstract property size : *ElaboratableValue*

All Logic subclasses should have an elaboratable size (the number of bits)

has_symbolic_dimensions(net_type: Logic | *None*) → bool

class Bit(name: *str* | *None* = **None**)

Bases: Logic

A single bit type. Equivalent to `logic` or `logic[0:0]` type in Verilog.

property size : *ElaboratableValue*

All `Logic` subclasses should have an elaboratable size (the number of bits)

copy()

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

class LogicArray(*, name: *str* | *None* = **None**, dimensions: *Iterable[*Dimensions*]*, item: *_ArrayItemOrField*)

Bases: Logic, Generic[_ArrayItemOrField]

A type representing a multidimensional array of logical elements.

property size

All `Logic` subclasses should have an elaboratable size (the number of bits)

copy()

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

dimensions : list[*Dimensions*]

item : _ArrayItemOrField

class Bits(*, name: *str* | *None* = **None**, dimensions: *Iterable[*Dimensions*]*)

Bases: LogicArray[Bit]

A multidimensional array of bits

class Enum(*, name: *str* | *None* = **None**, dimensions: *Collection[*Dimensions*]* = (), variants: *Mapping[str,* ElaboratableValue*]*)

Bases: Bits

A bit vector limited to a range of predefined variants, each one with an explicit name.

copy()

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

variants : dict[str, *ElaboratableValue*]

class StructField(*, name: *str*, type: *_ArrayItemOrField*)

Bases: Logic, Generic[_ArrayItemOrField]

A field in a `BitStruct`

copy()

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

property size

All `Logic` subclasses should have an elaboratable size (the number of bits)

field_name : VariableName

type : _ArrayItemOrField

class BitStruct(*, name: *str* | *None* = **None**, fields: *Iterable[*StructField*[*Logic*]]*)

Bases: `Logic`

A complex structural type equivalent to a `struct {...}` construct in SystemVerilog.

copy()

Clones the Logic type in a way to create two separate object trees, so that a deeper modification to one tree is not be reflected in the other.

property size

All `Logic` subclasses should have an elaboratable size (the number of bits)

fields : list[*StructField*[*Logic*]]

class LogicSelect(logic: ~topwrap.model.hdl_types.Logic, ops: list[~topwrap.model.hdl_types.LogicFieldSelect | ~topwrap.model.hdl_types.LogicBitSelect] = <factory>)

Bases: `object`

Represents an arbitrary selection of a part of a logical type. For example if `self.logic` references a multidimensional `LogicArray`, then you could have multiple `LogicBitSelect` operations in `self.ops` to select an arbitrary bit from that slice. Similarly, if there's a structure somewhere in the path you can add `LogicFieldSelect` to the operations to subscribe a specific logical field.

logic : *Logic*

ops : list[*LogicFieldSelect* | *LogicBitSelect*]

overlaps(other: LogicSelect) → bool

Checks if this selection of a logic type overlaps with another selection. An overlap occurs if two selections of types flatten to a packed bit-vector would target a range of the same bits. E.g.:

logic [127:0] a; wire [63:0] b = a[63:0]; wire [63:0] c = a[127:64]; wire [51:0] d = a[100:50];

assert not b.overlaps(c) and not c.overlaps(b) assert b.overlaps(d) assert c.overlaps(d) assert d.overlaps(c) and d.overlaps(b)

class LogicFieldSelect(field: StructField*[*Logic*]*)

Bases: `object`

Represents access operation to a field of a structure

field : *StructField*[*Logic*]

class LogicBitSelect(slice: Dimensions)

Bases: `object`

Represents a logic array indexing operation

slice : *Dimensions*

class ElabvalueEvaluator

Bases: object

sv_sysfunc_pfx = '_9263cec0fd53ad433112bdf3883bbd7216015109_'

funcs = {'_9263cec0fd53ad433112bdf3883bbd7216015109_clog2': <function ElabvalueEvaluator.<**lambda**>>}

static substitute_intf_params_for_module(intf: InterfaceDefinition, mod: Module, params: *dict[*ObjectId*[*Parameter*],* ElaboratableValue*]*) → dict[*ObjectId*[*InterfaceSignal*], *Logic*]

static resolve_intf_values(intf: Interface, instance: *dict[*ObjectId*[*Parameter*],* ElaboratableValue*]*) → tuple[Mapping[*ObjectId*[*Parameter*], *ElaboratableValue*], Mapping[*ObjectId*[*InterfaceSignal*], *Logic*]]

Resolve interface parameters to their integer values and Resolve interface signals types to their final types

static just_eval(val: ElaboratableValue, env: *dict[str, int]*) → int

static get_elab_value_deps(val: ElaboratableValue, params: *Iterable[*Parameter*]*) → Iterator[*Parameter*]

For a given ElaboratableValue and a set of parameters find out which parameters it uses

17.9 Interconnects

class InterconnectParams

Bases: MarshmallowDataclassExtensions

Base class for parameters/settings specific to a concrete interconnect type

Schema

alias of InterconnectParams

class InterconnectManagerParams

Bases: MarshmallowDataclassExtensions

Base class for manager parameters specific to a concrete interconnect type

Schema

alias of InterconnectManagerParams

class InterconnectSubordinateParams(address: ~topwrap.model.misc.ElaboratableValue = <factory>, size: ~topwrap.model.misc.ElaboratableValue = <factory>)

Bases: MarshmallowDataclassExtensions

Base class for subordinate parameters specific to a concrete interconnect type.

Transactions to addresses in range [self.address; self.address + self.size) will be routed to this subordinate.

address : ElaboratableValue.Field

The start address of this subordinate in the memory map

size : ElaboratableValue.Field

The size in bytes of this subordinate's address space

Schema

alias of `InterconnectSubordinateParams`

class Interconnect(*, name: *str*, clock: ReferencedPort, reset: ReferencedPort, params: *_IPAR*, managers: *Mapping[*ObjectId*[*ReferencedInterface*]*, *_MANPAR]* = {}, subordinates: *Mapping[*ObjectId*[*ReferencedInterface*]*, *_SUBPAR]* = {}, memory_map: *MemoryMap* | *None* = **None**)

Bases: `ABC`, `Generic[_IPAR, _MANPAR, _SUBPAR]`

Base class for multiple interconnect generator implementations.

Interconnects connect multiple interface instances together in a many-to-many topology, combining multiple subordinates into a unified address space so that one or multiple managers can access them.

parent : *Design*

The design containing this interconnect

name : VariableName

clock : *ReferencedPort*

The clock signal for this interconnect

reset : *ReferencedPort*

The reset signal for this interconnect

params : _IPAR

Interconnect-wide type-specific parameters

managers : dict[*ObjectId*[*ReferencedInterface*], _MANPAR]

Manager interfaces controlling this interconnect described as a mapping between a referenced interface in a design and the type-specific manager configuration

subordinates : dict[*ObjectId*[*ReferencedInterface*], _SUBPAR]

Subordinate interfaces subject to this interconnect described in the same format as managers

memory_map : MemoryMap | None

17.10 Miscellaneous

exception TranslationError

Bases: Exception

Fatal error while translating between IR and other formats

exception RelationshipError

Bases: Exception

Logic error of an IR hierarchy, like trying to double-assign a parent to an object

exception NotElaboratedException

Bases: Exception

Elaboratable value accessed before it could be properly elaborated

set_parent(child: *Any*, parent: *Any*)

VariableName

A placeholder for a future, possibly bounded type for IR object names. For example we may want to reduce possible names to only alphanumerical strings in the future.

class QuerableView(*parts: *Sequence[_E]*)

Bases: Sequence[_E]

A lightweight proxy for exploring sequences of elements or concatenations of multiple sequences of elements in our IR, e.g. *Design.components*, *Module.ios*, etc. that has convenient methods for finding specific entries, for example by their name, which can replace such cumbersome constructs:

comp = next((c for c in design.components if c.name == “name”), None)

with:

comp = design.components.find_by_name(“name”)

find_by(filter: *Callable[[_E], bool]*) → _E | None

find_by_name(name: *str*) → _E | None

find_by_name_or_error(name: *str*) → _E

find_by_id_or_error(id: Identifier) → _E

class ModelBase

Bases: ABC

This is a base class for all IR objects implementing common behavior that should be shared by all of them. Currently it only assigns them unique ObjectId s.

class ObjectId(obj: *_T*)

Bases: Generic[_T]

Represents a runtime-unique id for an IR object that can always be resolved to that object and can be used as a dictionary key/set value.

resolve() → _T

Resolve this id to a concrete object instance

class ElaboratableValue(expr: *int* | *str* | *ComplexParameter*)

Bases: `object`

A WIP class aiming to represent any generic value that can be resolved to a concrete constant during elaboration.

It should be able to e.g. reference multiple `Parameter` s by name and perform arbitrary arithmetic operations on them.

raw : int | str | ComplexParameter

value : str

elaborate() → int | None

class DataclassRepr(*, *load_default: ~typing.Any = <marshmallow.missing>, missing: ~typing.Any = <marshmallow.missing>, dump_default: ~typing.Any = <marshmallow.missing>, default: ~typing.Any = <marshmallow.missing>, data_key: str | None = None, attribute: str | None = None, validate: ~typing.Callable[[~typing.Any], ~typing.Any] | ~typing.Iterable[~typing.Callable[[~typing.Any], ~typing.Any]] | None = None, required: bool = False, allow_none: bool | None = None, load_only: bool = False, dump_only: bool = False, error_messages: dict[str, str] | None = None, metadata: ~typing.Mapping[str, ~typing.Any] | None = None, **additional_metadata*)

Bases: `Field`

Field

alias of `ElaboratableValue`[`ElaboratableValue`]

class Identifier(name: *str*, vendor: *str* = `'vendor'`, library: *str* = `'libdefault'`, version: *str* = `'0.1'`)

Bases: `object`

An advanced identifier of some IR objects that can benefit from storing more information than just their name. Based on the VLNV convention.

name : str

vendor : str = `'vendor'`

library : str = `'libdefault'`

version : str = `'0.1'`

static parse_vlnv(src: *str*) → *Identifier*

refers_to(other: Identifier) → bool

Whether this identifier - given as just a name, or as `vendor:library:name` - refers to `other`.

Mirrors how FuseSoC resolves a core name (`eq_vln` in the resolver, the bare-name lookup in `fusesoc.main._get_core`): `name` is compared case-insensitively, `vendor` /

`library` are only checked when this identifier specifies them (i.e. is not left at the empty string or the field default), and `version` is ignored. Uniqueness / ambiguity is the caller's concern.

combined() → str

class FileReference(file: *Path*, line: *int* = 0, column: *int* = 0)

Bases: `object`

A reference to a particular location in a text file on the filesystem

file : Path

line : int = 0

column : int = 0

class Parameter(*, name: *str*, default_value: ElaboratableValue | *None* = **None**)

Bases: `ModelBase`

Represents a parameter definition for a HDL module

parent : *Module* | *InterfaceDefinition*

name : VariableName

default_value : *ElaboratableValue* | None

If a value for this parameter was not provided during elaboration, this default will be used.

class ExtensionData(*, name: *str*, data: *Any*)

Bases: `ModelBase`

Represents generic module data

parent : *Module*

name : VariableName

data : Any

17.11 A note on "sliced" vs. "independent" signals

There is specific terminology used in Topwrap in the context of interface signals. An interface is a collection of signals, each one being either `sliced` or `independent`. A `sliced` signal exists as a plain external port in `Module.ports`, either the entire port, or some arbitrary slice of it and is just a mapping for Topwrap to know that such a plain port actually realizes a specific signal of some defined interface.

For example, if the user had an IP core written in pure Verilog that used the AHB interface for data exchange and control, it would most likely have all AHB signals (`haddr`, `htrans`, `hresp`, etc.) present as individual external ports on it:

```
module custom_core (
    input  wire [31:0] haddr,
    input  wire [1:0]  htrans,
    output wire        hresp,
    ...
);

endmodule
```

If such a core was parsed into Topwrap's internal representation, each such port would end up in `Module.ports`. Clearly though, all of these ports are actually realizing specific signals of the AHB interface. To store that information clearly in the IR, an `Interface` instance should be created and added to `Module.interfaces`. During the creation of this interface instance, the `Interface.signals` field needs to be filled with the mapping of each `InterfaceSignal` from the AHB's `InterfaceDefinition` to a `ReferencedPort` referencing the plain port from `Module.ports`. Now each backend will know that despite there being an interface instance on this module, all of its signals are in reality just plain ports and when such a module is used in a design, appropriate code will be generated to interact with these ports. This situation, where an `InterfaceSignal` is mapped to a concrete plain port in the module, is called a "sliced" signal.

Now there is another type of a signal realization called an "independent" signal. Independent signals are not, and cannot be mapped to plain ports. They are transparent and get automatically connected to each other just by the action of connecting two interface instances together. This situation naturally corresponds to using the `interface` construct in SystemVerilog:

```
interface AHB_intf;
    logic [31:0] haddr,
    logic [1:0]  htrans,
    logic        hresp,
    ...
endinterface

module custom_core (
    AHB_intf ahb_sub,
    ...
);

endmodule
```

In the above example, the AHB interface is now explicit in the module definition itself. The individual signals still exist, but are enclosed in the interface instance and will get automatically connected in case two such interface instances are connected. There is still a need to store that interface in `Module.interfaces`, but now there doesn't exist any plain port that the interface signals would map onto, thus instead of an `InterfaceSignal` mapping to a `ReferencedPort` in the IR, it's mapped to `None` instead. With that, the backends will know that this signal is represented by the interface instance itself and generate appropriate code.

Generally, in a single IR `Interface` all signals are either "sliced" or "independent", but a situation where majority of the signals are independent and a few are sliced or vice-versa is entirely possible and should appropriately handled by a specific backend.

CHAPTER

EIGHTEEN

FRONTEND

Frontend is a class that transforms a specific format into IR. There are multiple frontends:

- SystemVerilogFrontend class supports SystemVerilog and Verilog files
- KpmFrontend class supports Pipeline Manager files
- YamlFrontend class supports YAML files
- IpXactFrontend class supports IP-XACT 2022 files

Frontends may be initialized with predefined modules and interfaces, which are used to resolve references while parsing. Parsed modules are returned in `FrontendParseOutput`, including modules whose identifiers match predefined modules. The handling of interfaces in `FrontendParseOutput` differs between frontends: `SystemVerilogFrontend` returns parsed or reused interface definitions, `IpXactFrontend` returns interface definitions parsed from `abstractionDefinition` files, while `YamlFrontend` and `KpmFrontend` only use predefined interfaces during parsing and do not return them. `AutomaticFrontend` preserves the behavior of the frontend it delegates to.

18.1 SystemVerilogFrontend

SystemVerilogFrontend class is used to parse System Verilog and Verilog files. It is mainly used for adding modules to a repo, connected by Topwrap in a design.yaml.

18.2 KpmFrontend

KpmFrontend class is used to parse KPM-specific files. It is mainly used for getting a design from a GUI.

18.3 YamlFrontend

YamlFrontend class is used to parse design.yaml and module.yaml files. It follows a format specified in *design description*.

18.4 IpXactFrontend

IpXactFrontend class is used to parse existing IP-XACT files the same way as SystemVerilogFrontend. Not all tags defined in the 2022 specification are supported. This frontend supports parsing `component`, `design` and `abstractionDefinition` root tags. Supported features are:

- `VLNV`, `parameters`, `ports` and `busInterfaces` in `component` root tag. `views` tag is unused and only one `designConfigurationInstantiation` is supported. Only one `componentInstantiation` tag is supported.
- `componentInstances`, `adHocConnections` and `interconnections` in `design` root tag.
- `ports`, with support for only `logicalName`, `onTarget` and `onInitiator` in `port` in `abstractionDefinition` root tag.

CHAPTER

NINETEEN

BACKEND

Backend is a class that transforms IR into a specific output format. There are multiple backends:

- SystemVerilogBackend class generates SystemVerilog top files
- KpmBackend class generates Pipeline Manager files
- IpCoreDescriptionBackend and DesignDescriptionBackend classes generate YAML files
- IPXACTBackend class generates IP-XACT 2022 files

19.1 SystemVerilogBackend

SystemVerilogBackend class is used to generate a SystemVerilog top wrapper from a design. It is the backend used by `topwrap generate`, see *Generating Verilog top files*.

19.2 KpmBackend

KpmBackend class is used to generate Pipeline Manager specification and dataflow files. It is mainly used to export a design to the GUI.

19.3 YAML backends

IpCoreDescriptionBackend and DesignDescriptionBackend classes are used to generate IP core description and design description YAML files, respectively. They follow the format specified in *design description*.

19.4 IPXACTBackend

IPXACTBackend class is used to generate IP-XACT 2022 files from a design. It produces `component`, `design` and `abstractionDefinition/busDefinition` files and relies on the `topwrap-ipxact-parser` library. For interconnects it reuses the SystemVerilog generators, as different HDL backends may yield different IR modules.

This is the backend used by `topwrap generate --ipxact`, see *Generating IPXACT 2022 files*. For details on the IP-XACT format and the Topwrap to IP-XACT conversion, see the *IP-XACT format* document.

19.4.1 UNSPECIFIED interface type

Interfaces defined in module can have MANAGER, SUBORDINATE or UNSPECIFIED types. This types are used to select direction of port. IPXACT 2022 don't have ability to represent UNSPECIFIED type. When IPXACTBackend encounters UNSPECIFIED type it will log warning and proceed as if it was MANAGER type.

19.4.2 Elaborating parameters

Elaboration of parameters that are present in widths of ports and interfaces is not supported. Generated IPXACT files can contain illegal widths, it is recommended to use sed to change them to selected values.

CHAPTER

TWENTY

FUSESOCBUILDER

Topwrap supports generating FuseSoC `.core` files with `FuseSocBuilder`.The `.core` file contains information about source files and synthesis tools.

Generation of FuseSoC `.core` files is based on a Jinja template that defaults to `topwrap/templates/core.yaml.j2`, but can be overridden.

Here's an example of how to generate a simple project:

```
from topwrap.fuse_helper import FuseSocBuilder
fuse = FuseSocBuilder()

# add source of the IPs used in the project
fuse.add_source('DMATop.v', 'verilogSource')

# add source of the top file
fuse.add_source('top.v', 'verilogSource')

# specify the names of the core file and the directory where sources are stored
# generate the project
fuse.build('build/top.core', 'sources')
```

Warning: Default template in `topwrap/templates/core.yaml.j2` does not make use of resources added with `add_external_ip()`, i.e. they won't be present in the generated core file.

class FuseSocBuilder(part: *str* | *None*)

Use this class to generate a FuseSoc .core file

add_dependency(dependency: *str* | Identifier)

Adds a dependency to the list of dependencies in the core file

add_external_ip(vlnv: *str*, name: *str*)

Store information about IP Cores from Vivado library to generate hooks that will add the IPs in a TCL script.

add_fileset(name: *str*, files: *list[SourceFile]*, depends: *list[*Identifier*]*)

Add a complete fileset to the core file

add_script(name: *str*, args: *list[str]*)

Add a named fusesoc script

add_source(filename: *Path*, type: *str*)

Adds an HDL source to the list of sources in the core file

add_sources_dir(sources_dir: *Collection[Path]*, core_path: *Path*)

Given a name of a directory, add all files found inside it. Recognize VHDL, Verilog, and XDC files.

add_target(target: *FuseSocTarget*)

Add a named fusesoc target

build(top_name: *str*, core_path: *Path*, sources_dir: *Collection[Path]* = `[]`)

Generate the final create .core file

Parameters

sources_dir: Collection[Path] = []
additional directory with source files to add

template_name
name of jinja2 template to be used, either in working directory, or bundled with the package. defaults to a bundled template

20.1 Customizations

By default, Topwrap will generate a core file with a default target of running Vivado on the top-level defined by the Topwrap project. This can be overridden in several ways. To disable the generation of a default Vivado target, you would call

```
from topwrap.fuse_helper import FuseSocBuilder
fuse = FuseSocBuilder(None)

fuse.set_generate_vivado(False) # disables the default vivado target
```

You can add custom targets by constructing `FuseSocTarget`. Here's an example to create a Verilator flow-target that builds and runs a testbench:

```
from pathlib import Path

from topwrap.fuse_helper import (
    FuseSocBuilder,
    FuseSocFlowApi,
    FuseSocTarget,
    SourceFile,
)

fuse = FuseSocBuilder(None)

fuse.set_generate_vivado(False)

# Add a source to the default `rtl` fileset
fuse.add_source(Path("top.v"), "verilogSource")
toplevel = "top"  # toplevel, as defined in top.v
```

(continues on next page)

(continued from previous page)

```
# Add a second fileset, `verilator_tb`, with testbench
fuse.add_fileset(
    "verilator_tb",
    files=[SourceFile(Path("top_tb.sv"), "systemVerilogSource")],
    depends=[],
)

# Add a testbench target `tb`, that references both the default fileset and the␣
↪custom
fuse.add_target(
    FuseSocTarget(
        "tb",
        toplevel,
        FuseSocFlowApi(
            type="sim",
            options={
                "tool": "verilator",
                "mode": "cc",
                "verilator_options": ["-Wno-fatal", "--main", ["--timing"]],
            },
            make_options=["-s"],
        ),
        filesets=["rtl", "verilator_tb"],
        hooks=[],
    )
)

fuse.build(toplevel, Path("top.core"))
```

This will generate the following:

```
CAPI=2:
name: ::top
description: Core file generated by TopWrap
filesets:
    verilator_tb:
        files:
        -   top_tb.sv:
                file_type: systemVerilogSource
    rtl:
        files:
        -   top.v:
                file_type: verilogSource
targets:
    tb:
        filesets:
        - rtl
        - verilator_tb
```

(continues on next page)

(continued from previous page)

```
        toplevel: top
        flow: sim
        flow_options:
            tool: verilator
            mode: cc
            verilator_options:
            - -Wno-fatal
            - --main
            -   - --timing
            flow_make_options:
            - -s
```

CHAPTER

TWENTYONE

INTERFACE DEFINITION

Topwrap uses *interface definition files* for its parsing functionality.

These are used to match a given set of signals that appear in the HDL source with signals in the interface definition.

`InterfaceDefinition` is defined as a `marshmallow_dataclass.dataclass` - this enables loading the YAML structure into Python objects and performs validation (that the YAML is in the correct format) and typechecking (that the loaded values are of the correct types).

CHAPTER

TWENTYTWO

CONFIG

The `Config` object stores configuration values. The global `topwrap.config.Config` object is used throughout the codebase to access the Topwrap configuration.

It is created by `ConfigManager` that reads the config files as defined in `topwrap.config.ConfigManager.DEFAULT_SEARCH_PATHS`, with local files taking precedence. For flows consuming a design YAML, the configuration is additionally overridden in the pre-parse stage.

class Config(force_interface_compliance: bool | None = False, repositories: dict[str, ~topwrap.resource_field.ResourceReferenceHandler] = <factory>, libraries: dict[str, str] = <factory>)

Global topwrap configuration

Schema

alias of `Config`

class ConfigManager(search_paths: *Sequence[Path]* | *None* = **None**)

Manager used to load topwrap's configuration from files.

The configuration files are loaded in a specific order, which also determines the priority of settings that are defined differently in the files. The list of default search paths is defined in the *DEFAULT_SEARCH_PATH* class variable. Configuration files that are specified earlier in the list have higher priority and can overwrite the settings from the files that follow. The default list of search paths can be changed by passing a different list to the ConfigManager constructor.

CHAPTER

TWENTYTHREE

VALIDATION OF DESIGN

One of topwrap features is to run validation on user's design which consists of series of checks for errors user may do while creating a design.

class DataflowValidator(dataflow: *dict[str, Any]*)

The main class that contains all the validation checks. The purpose of validation is to check for common errors the user may make while creating the design and make sure the design can be saved in topwrap yaml format. These functions are called in two cases:

1) When there is a call from KPM for dataflow_validate (the user has clicked Validate in GUI)
2) When a user tries to save the design

check_connection_to_subgraph_metanodes() → *CheckResult*

Check for any connections to exposed subgraph metanode ports.

In this context:

- **Exposed port: A port on a subgraph metanode that represents the interface of the**
  subgraph to the external graph. It is visible and accessible from outside the subgraph.
- **Unexposed port: An internal port on a subgraph metanode that is used for internal**
  connections within the subgraph but is not accessible from the external graph.

These metanodes are meant to represent ports of subgraph nodes. Connections to these metanodes should only occur via the unexposed ports. Any connection to an exposed port is considered an error because such connections cannot be represented in the design.

check_duplicate_metanode_names() → *CheckResult*

Check for duplicate names of external metanodes. The name of metanode is in "External Name" property. In design, these external metanodes are referenced by this name so if there are multiple metanodes with the same name it will not be possible to represent them, hence the error.

check_duplicate_node_names() → *CheckResult*

Check for any duplicate IP instance names in the graph (graph represents a hierarchy level). This check prevents from creating multiple nodes with the same "instanceName" in a given graph, since this is invalid in design. There can be multiple nodes with the same "instanceName" in the whole design (on various hierarchy levels).

check_external_in_to_external_out_connections() → *CheckResult*

Check for connections between two external metanodes. In our design format (YAML), connections to external nodes are always represented as *port: external*, regardless of whether the *external* node is an input or output. Therefore, connections directly between two external metanodes cannot be represented within this format and are invalid by design.

check_external_io_exposed_ifaces()

Check whether subgraph interfaces are the same as interfaces opposite to connected External I/O node interfaces inside the subgraph.

check_parameters_values() → *CheckResult*

Check if parameters in IP nodes are valid.

This check ensures users are informed of any errors in parameter definitions. While it is possible to save the design with invalid parameters (e.g. save to YAML), such a design cannot be successfully built into Verilog because integer widths of all parameters must be determined during the build process.

A parameter is considered valid if it meets any of the following conditions:

- It is an instance of `int`.
- It has a correct value format, e.g.: `16'h5A5A`.
- **It can be evaluated based on other parameters, e.g.:**
  `ADDR_WIDTH: 32`

  `DATA_WIDTH: ADDR_WIDTH/4` (evaluates to 8, which is valid).

check_port_to_multiple_external_metanodes() → *CheckResult*

Check for ports that have connections to multiple external metanodes. Design schema allows only one connection between an IPcore/hierarchy port and an external metanode. The connection between the port and external metanode is a single entry, not a list that's why we can't add more connections.

check_unconnected_ports_interfaces() → *CheckResult*

Check for unconnected ports or interfaces. This check helps identify any unconnected elements, warning the user about potential oversights or missed connections.

check_unnamed_external_metanodes_with_multiple_conn() → *CheckResult*

Check for external metanodes that are connected to more than one port and don't have a user-specified name. This is important to check because it is an undefined behavior when saving a design. Currently, when there is a connection to an unnamed metanode in design this metanode will have the name of the port it's connected to.

validate_kpm_design() → dict[str, list[str]]

Run checks to validate the user-created design in KPM. Checks are designed to inform the user about errors present in his design that make it impossible to save and display warnings about potential issues in the design.

Each check returns the following class:

class CheckResult(check_name: *str*, status: *MessageType*, error_count: *int* = 0, message: *str* | *None* = **None**)

Return type of each validation check

Parameters

check_name : str
Name of the check

status : MessageType

Check can be return one of three MessageTypes (OK, ERROR, WARNING)

- OK - this status is set when check was successful
- WARNING - check have failed but it is possible to represent the graph in design yaml
- ERROR - it is not possible to represent graph in design yaml

error_count : int
Number of errors if status is not OK

message : str | None
Message describing errors

23.1 Tests for validation checks

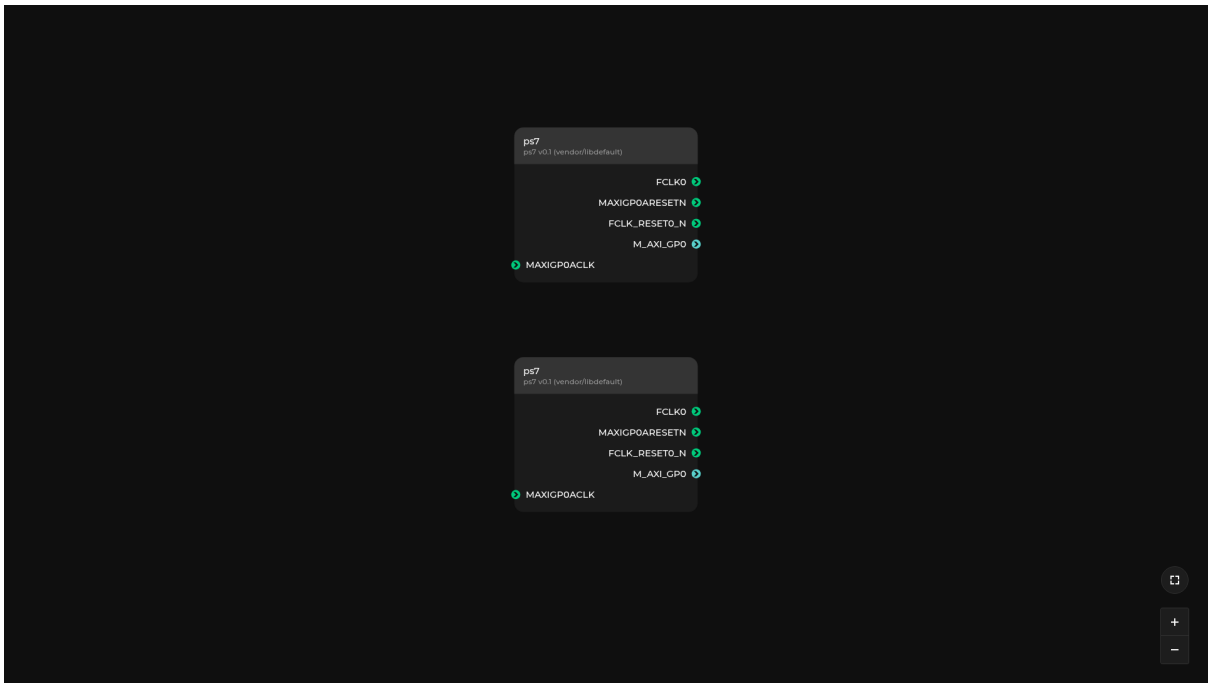
Tests for the `DataflowValidator` class are done using various designs that are valid or have some errors in them with the goal to check everything validation functions can do.

Below are all the graphs that are used for testing.

23.1.1 Duplicate IP names

dataflow_duplicate_ip_names()

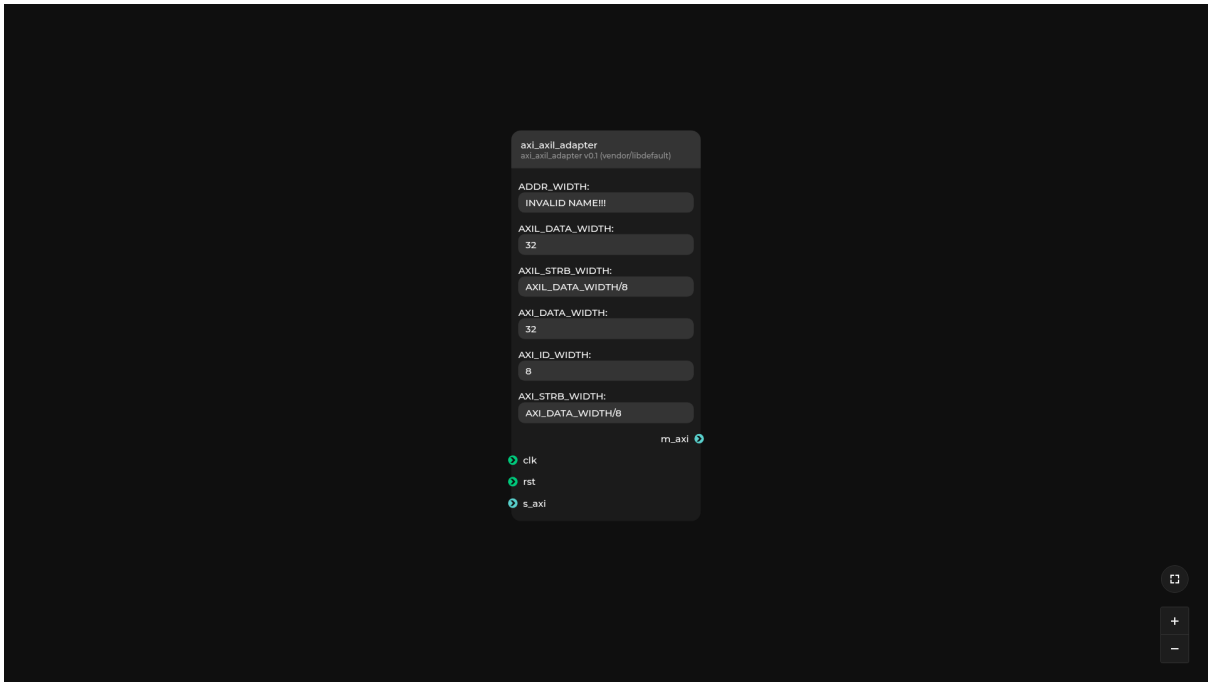
Dataflow containing two IP cores with the same instance name. This is considered as not possible to represent in design yaml since we can't distinguish them.



23.1.2 Invalid parameters' values

`dataflow_invalid_parameters_values()`

Dataflow containing an IP core with multiple parameters, but it's impossible to resolve the *INVALID NAME!!!*.



23.1.3 Connection between external Metanodes

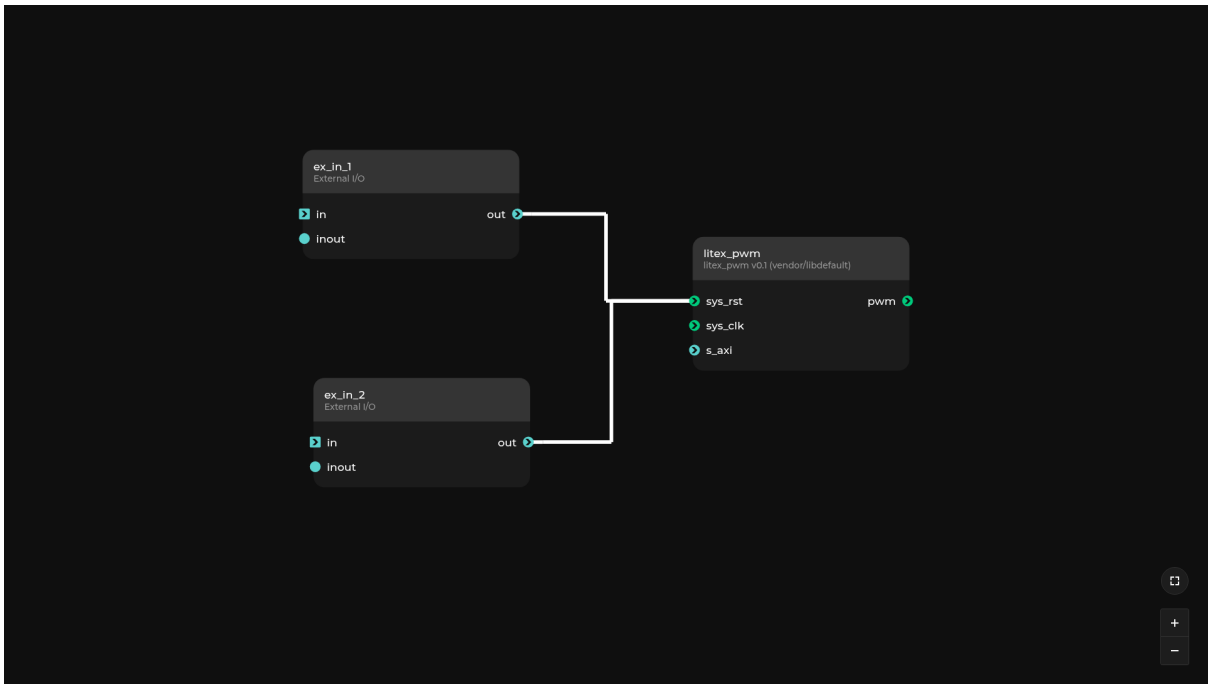
`dataflow_ext_in_to_ext_out_connections()`

Dataflow containing Metanode<->Metanode connection.

23.1.4 Ports connected to multiple external Metanodes

`dataflow_ports_multiple_external_metanodes()`

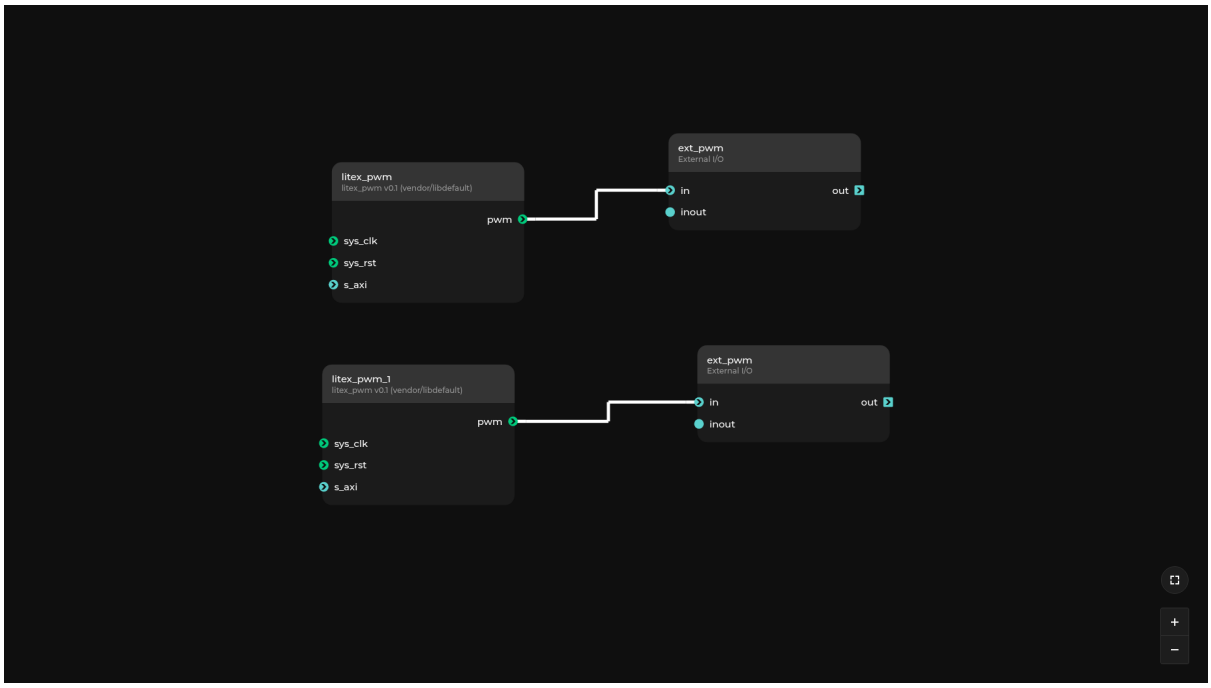
Dataflow containing a port connected to two External Metanodes.



23.1.5 Duplicate Metanode names

`dataflow_duplicate_metanode_names()`

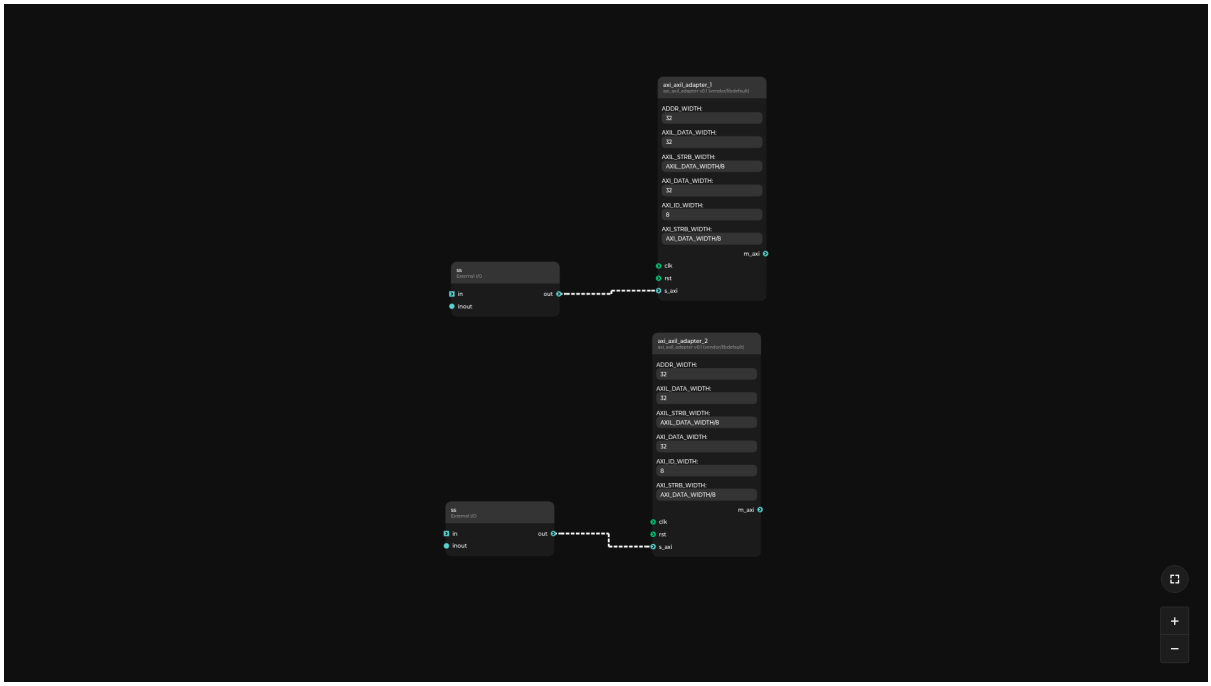
Dataflow containing two External Output Metanodes with the same “External Name” value.



23.1.6 Duplicate Metanode connected to interface

`dataflow_duplicate_external_input_interfaces()`

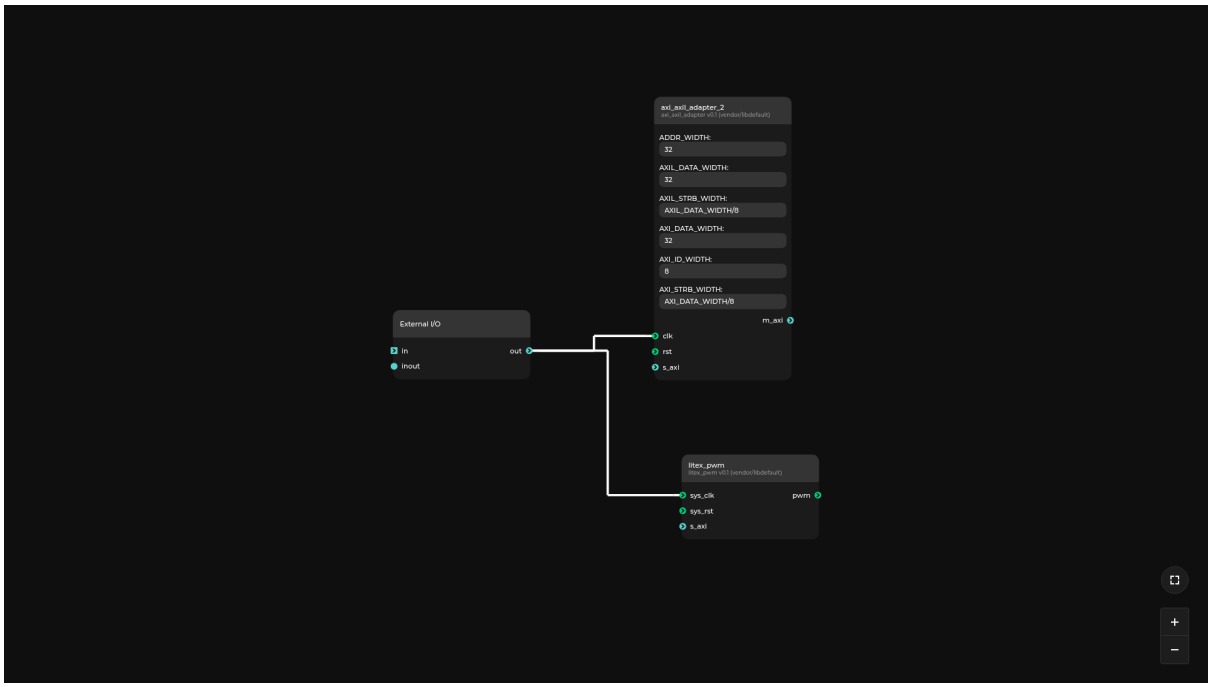
Dataflow containing two External Input Metanodes with the same name. Here connection is to interface instead of port as in the example above.



23.1.7 Unnamed Metanodes

`dataflow_unnamed_metanodes()`

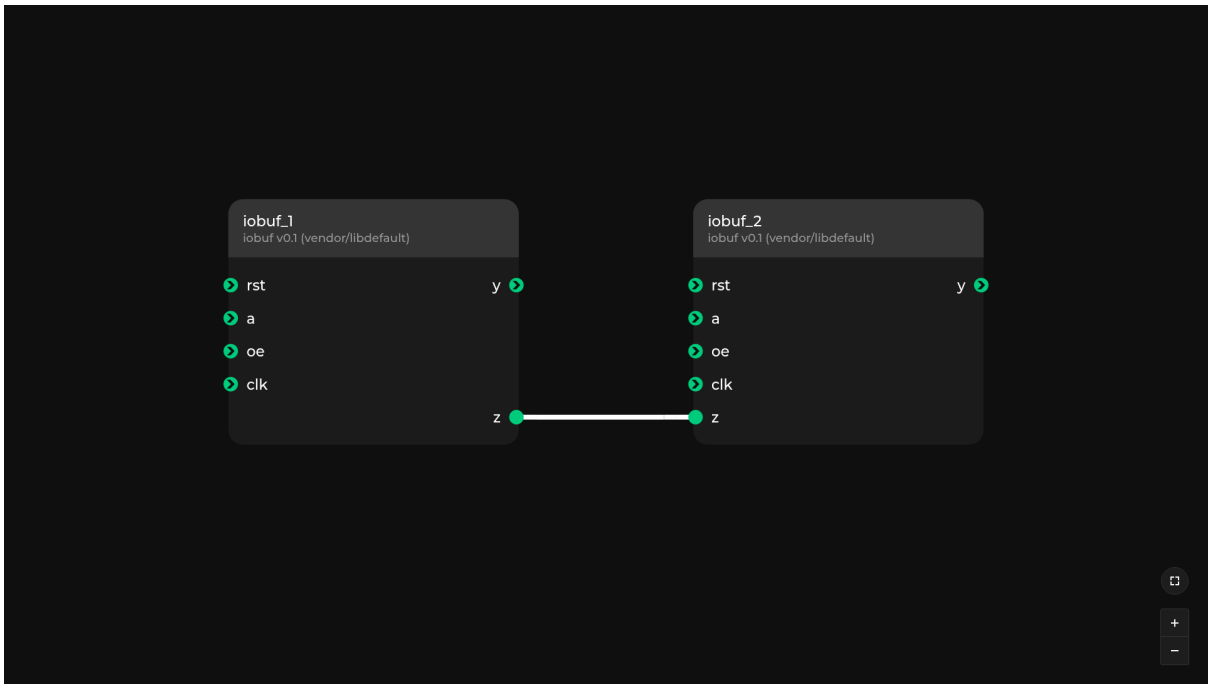
Dataflow containing unnamed External Input Metanode with multiple connections to it.



23.1.8 Connection between two inout ports

`dataflow_inouts_connections()`

Dataflow containing a connection between two inout ports.



23.1.9 Unconnected ports in subgraph node

dataflow_unconn_hierarchy()

Dataflow containing subgraph node with two unconnected interfaces.

23.1.10 Connection of subgraph node to multiple External Metanodes

dataflow_subgraph_multiple_external_metanodes()

Dataflow containing subgraph node with connection to two External Output Metanodes.

23.1.11 Connection to subgraph Metanode

dataflow_conn_subgraph_metanode()

Dataflow containing subgraph metanode with connection to exposed interface. It can be seen by selecting the “Edit Subgraph” on subgraph node.

23.1.12 Complex hierarchy graph

dataflow_complex_hierarchy()

Dataflow containing many edge cases such as duplicate subgraph node names, stressing out the capabilities of saving a design.

23.1.13 Duplicate IP cores in subgraph node

dataflow_hier_duplicate_names()

Dataflow containing subgraph node inside which are duplicate IP's.

CHAPTER

TWENTYFOUR

FUTURE PLANNED ENHANCEMENTS IN TOPWRAP

24.1 Library of open-source cores

Currently, users have to manually or semi-manually (e.g. through FuseSoC) supply all of the cores used in the design. In future, a repository of open-source cores that can be easily reused will be provided, allowing users to quickly put together designs from premade hardware blocks.

24.2 Support for hierarchical block designs in Topwrap's GUI

Topwrap supports creating hierarchical designs by manually writing the hierarchy in the design description YAML, while in future, this feature will be additionally supported in the GUI for visually organizing complex designs.

24.3 Support for parsing SystemVerilog sources

Information about IP cores is stored in *IP description files*. These files can be generated automatically from HDL source files - currently Verilog and VHDL are supported. In a future release, Topwrap will also provide the possibility of generating YAMLs from SystemVerilog.

CHAPTER

TWENTYFIVE

OTHER POSSIBLE IMPROVEMENTS

25.1 Ability to produce top-level wrappers in VHDL

Topwrap currently has a SystemVerilog backend and an *IP-XACT backend* for generating top-level designs. We would also like to add the ability to produce such designs in VHDL.

25.2 Bus management

Another feature that could be introduced is allowing users to create full-featured designs with processors by providing proper support for bus management.

This should include features such as:

- the ability to specify the address of a peripheral device on the bus
- support for the most popular buses (AXI, TileLink, Wishbone)

This will require writing or creating bus arbiters (`round-robin`, `crossbar`) and providing a mechanism for connecting manager(s) and subordinate(s) together. As a result, the user would be able to create complex SoCs directly in Topwrap.

Currently, only experimental support for `Wishbone with a round-robin arbiter` is available.

25.3 Improve the process of recreating a design from a YAML file

One of the main features supported by Topwrap and the GUI is exporting and importing user-created designs, both to or from a *design description* YAML. However, during these conversions, information about the position of user-added nodes is not preserved. This is cumbersome in the case of complicated designs since the imported nodes are not retained in the most optimal positions.

Therefore, one of our objectives is to provide a convenient way of creating and restoring user-created designs in the GUI, so that the node positions are retained.

25.4 Deeper integration with other tools

Topwrap can build designs, but testing and synthesis rely on the user - they have to automate this process themselves (e.g. with makefiles). To improve the usability of Topwrap, a potential area of improvement is to integrate tools for synthesis, simulation and co-simulation (with e.g. Renode with Topwrap, accessible through scripts. Some could be pre-packaged with Topwrap (e.g. simulation with Verilator, synthesis with Vivado).

It could also be possible to invoke these from the GUI by adding custom buttons or through the integrated terminal.

25.5 Provide a way to parse HDL sources from the GUI level

Another issue related to HDL parsing is that the user has to manually parse HDL sources to obtain the IP core description YAMLs. These files then need to be provided as command-line parameters when launching the Topwrap GUI client application. Therefore, we aim to provide a way of parsing HDL files directly from the GUI.

CHAPTER

TWENTYSIX

USING KPM IFRAMES INSIDE DOCS

It is possible to use the `pipeline_manager` Sphinx directive to embed KPM directly inside a doc.

26.1 Usage

```
```{pipeline_manager}
:spec: <KPM specification .json file URI>
:graph: <KPM dataflow .json file URI>
:preview: <a Boolean value specifying whether this KPM should be started in␣
↪preview mode>
:height: <a string CSS height property that sets the `height` of the iframe>
:alt: <a custom alternative text used in the PDF documentation instead of the␣
↪default one>
```
```

`URI` represents either a local file from sources that are copied into the build directory, or a remote resource.

All parameters in this directive are optional.

26.2 Tests

26.2.1 Use local files

26.2.2 Open in preview mode

26.2.3 Use a custom alt text

Note: The alternative text is visible instead of the iframe in the PDF version of this documentation.

CHAPTER

TWENTYSEVEN

EXAMPLES FOR INTERNAL REPRESENTATION

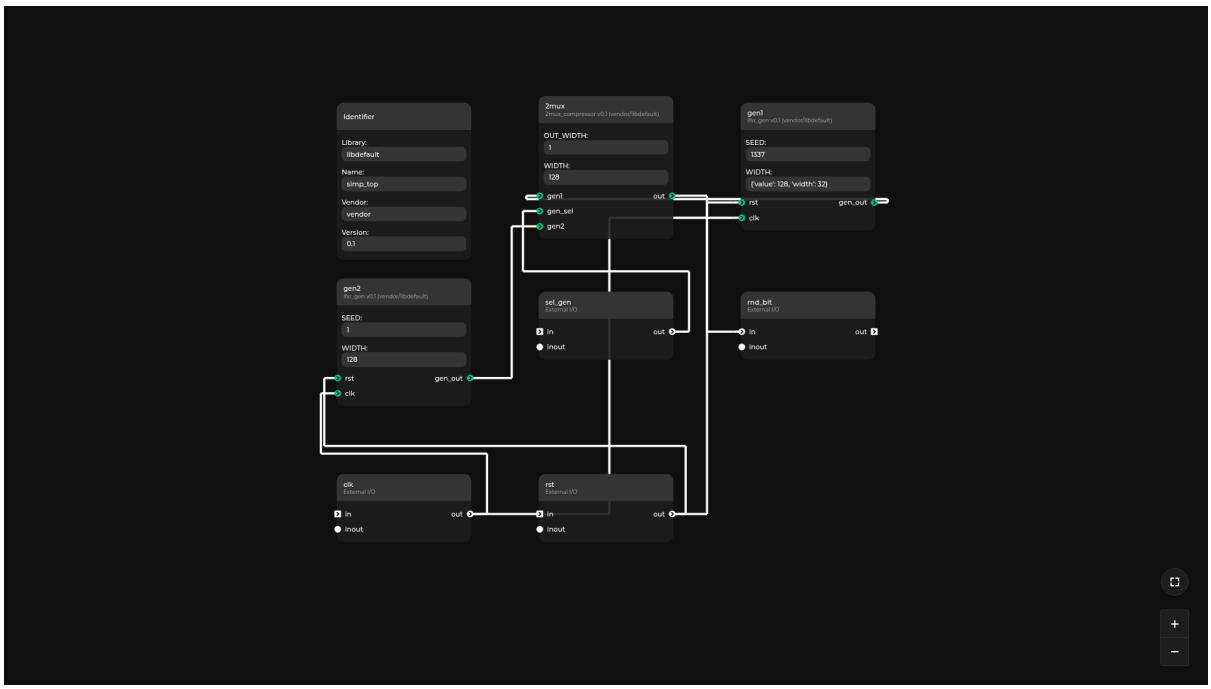
There are four examples in `examples/ir_examples` showcasing specific features of Topwrap which we want to take into consideration while creating the new internal representation.

27.1 Simple

This is a simple non-hierarchical example that uses two IPs. Inside, there are two LFSR RNGs constantly generating pseudorandom numbers on their outputs. They are both connected to a multiplexer that selects which generator's output should be passed to the `rnd_bit` external output port. The specific generator is selected using the `sel_gen` input port.

This example features:

- IP core parameters
- variable width ports

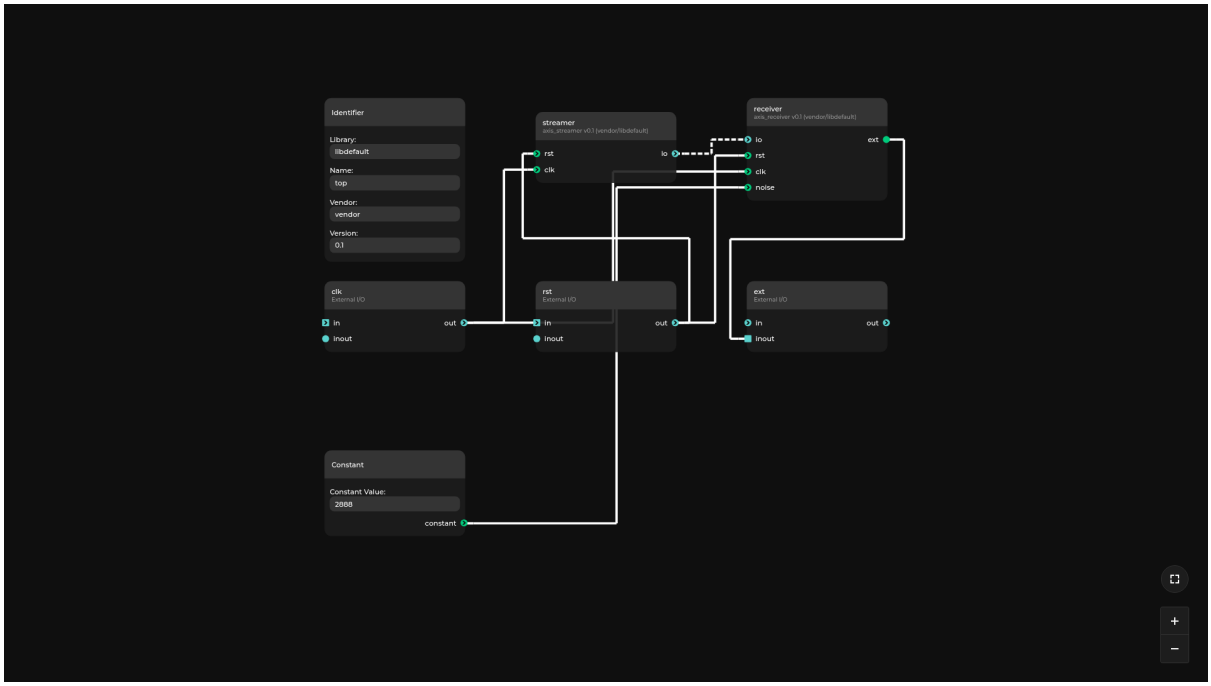


27.2 Interface

This is another simple example using two IPs, this time with an interface. The design consists of a streamer IP and a receiver IP. They both are connected using the AXI4Stream interface. The receiver then passes the data to an external inout port.

This example features:

- usage of interface ports
- port slicing
- constant value connected to a port
- an Inout port

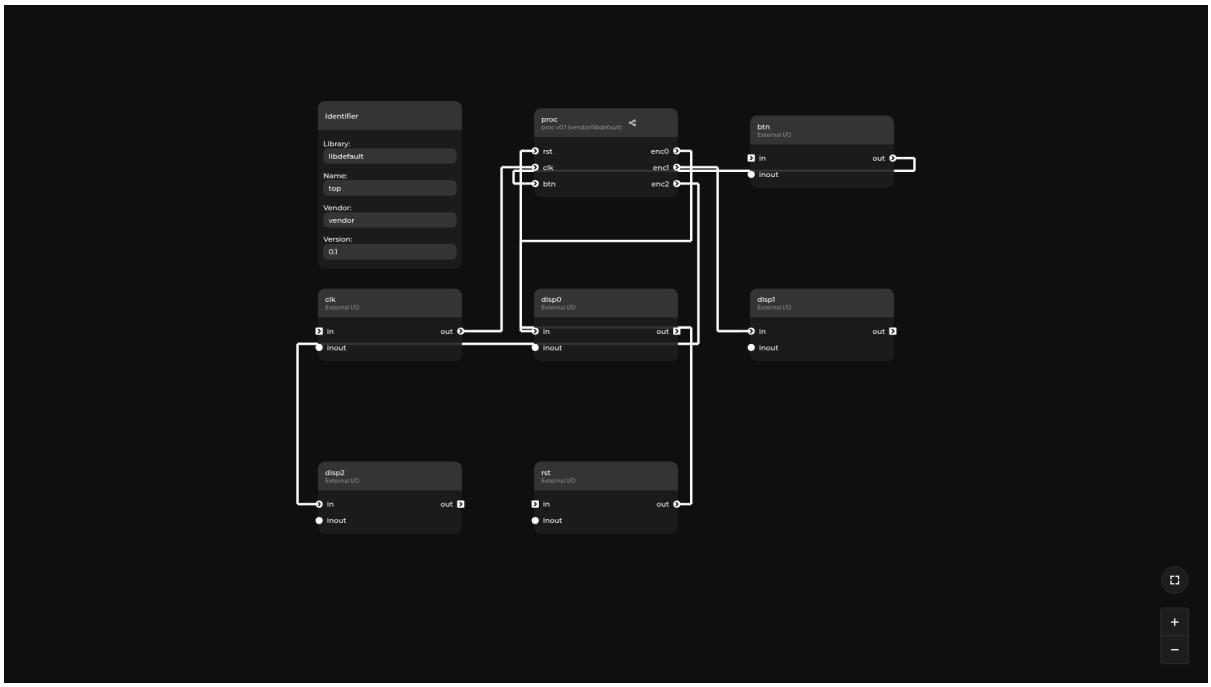


27.3 Hierarchical

This is an example of a hierarchical design. The top-level features standard external ports `clk` and `rst`, a `btn` input that represents an input from a physical button, and `disp0..2` outputs that go to an imaginary 3-wire-controlled display. All these ports are connected to a processing hierarchy `proc`. Inside this hierarchy we can see the `btn` input going into a "debouncer" IP, its output going into a 4-bit counter, the counter's sum arriving into an encoder as the input number, and the display outputs from the encoder further lifted to the parent level. The encoder itself is a hierarchy, though an empty one with only the ports defined. The 4-bit counter is also a hierarchy that can be further explored. It consists of a variable width adder IP and a flip-flop register IP.

This example features:

- hierarchies of more than one depth



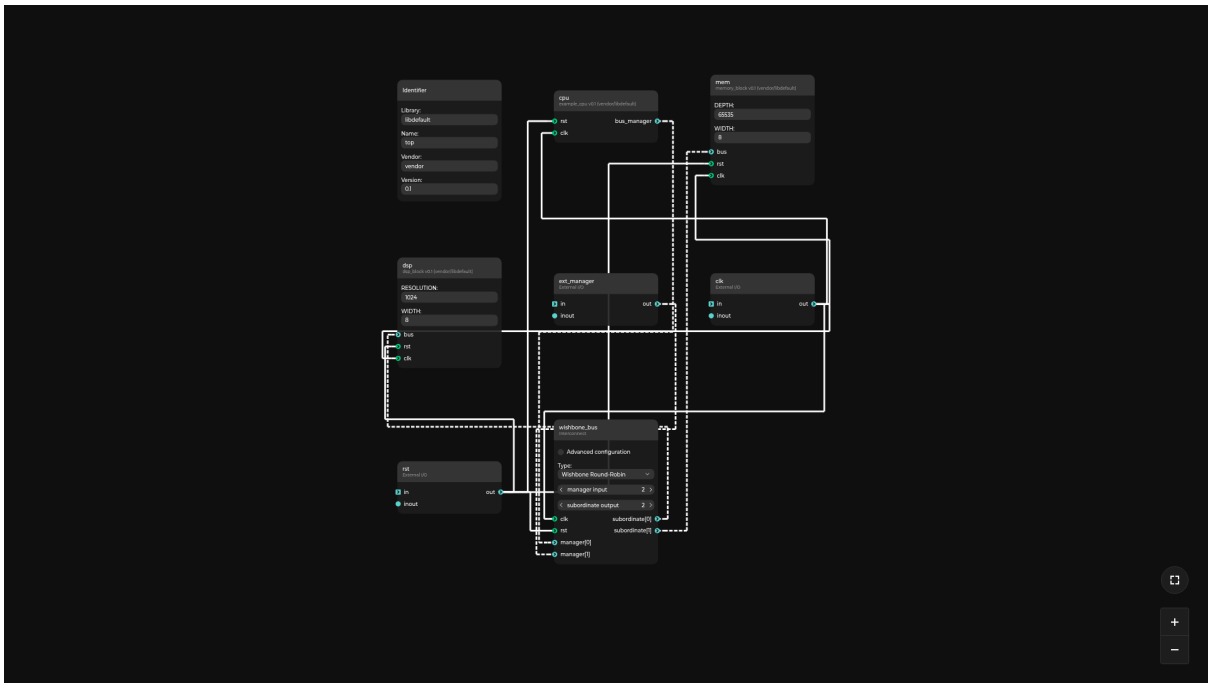
27.4 Interconnect

This is an example of our interconnect generation feature. The design features 3 IP cores: a memory core (`ips/mem.yaml`), a digital signal processor (`ips/dsp.yaml`) and a CPU (`ips/cpu.yaml`). All of them are connected to a wishbone interconnect where both the CPU and an external interface `ext_manager` act as managers and drive the bus. DSP and MEM are subordinates, one available at address 0x0, the other at 0x10000.

Note that while this specific example uses a "Wishbone Round-Robin" interconnect, we still aim to support other types of them in the future. Each one will have its own schema for the "params" section so make sure not to hardcode the parameters' keys or values.

This example features:

- usage of interface ports
- interconnect usage



27.5 Advanced

This example was created some time after the previous ones and it uses features that the IR didn't have previously. It's main purpose is to demonstrate and test:

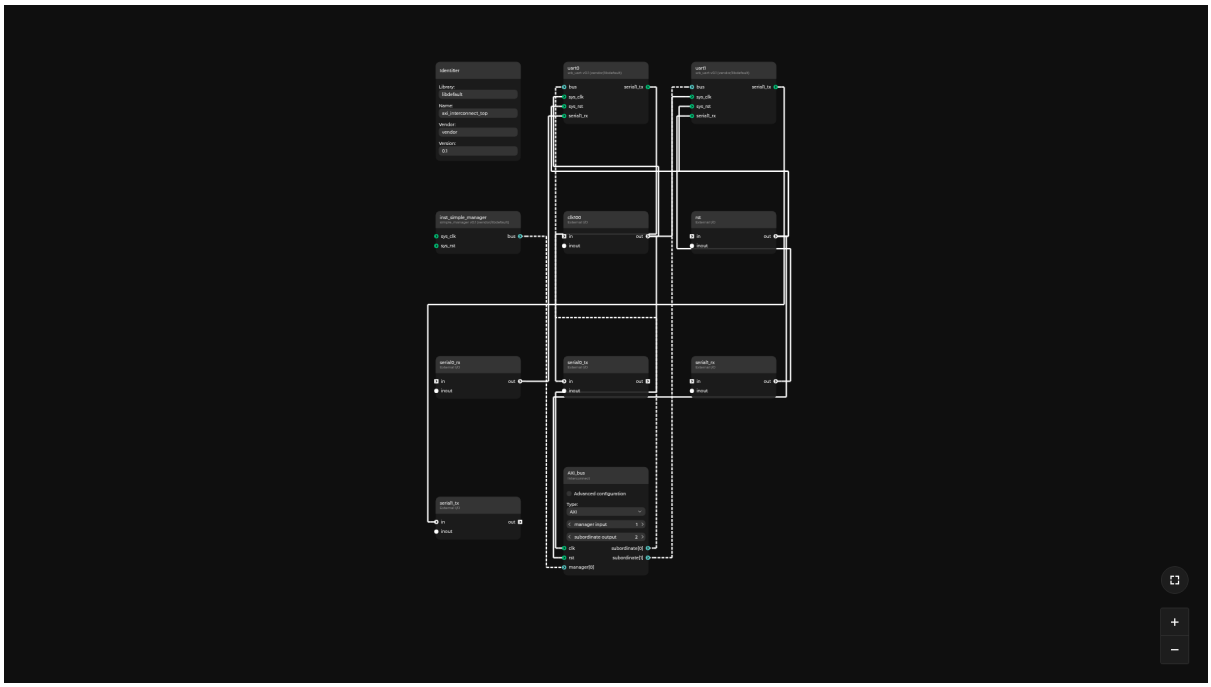
- Multidimensional bit arrays
- Named vs anonymous `BitStructs`
- Arbitrarily sliced port connections (more complex than just a bit-vector slice)
- Partially inferred interfaces (some of their signals are realized using sliced external ports while some signals come from a "real" interface IO)
- Interface signals having types just like ports
- Interface signals having default values
- Complex interface mode configurations:
  - Signals optional for the subordinates or masters
  - Signals legal only for one side of the interface

27.6 SoC

This example has an RISC-V processor, memory, and a UART connected by Wishbone interconnect. Interconnect is not present as a Verilog file, but is generated by Topwrap. Unlike other ir-examples it is localized in `examples/soc`. It has both IR modules defined in python files and System Verilog files that are parsed by `topwrap parse`. It can be run in a simulator such as Verilator. To run this example, the RISC-V 64-bit toolchain and Verilator simulator need to be installed. To generate interconnect, use `make generate`, to build Verilator sim use `make build-sim`, and to simulate the design use `make sim`.

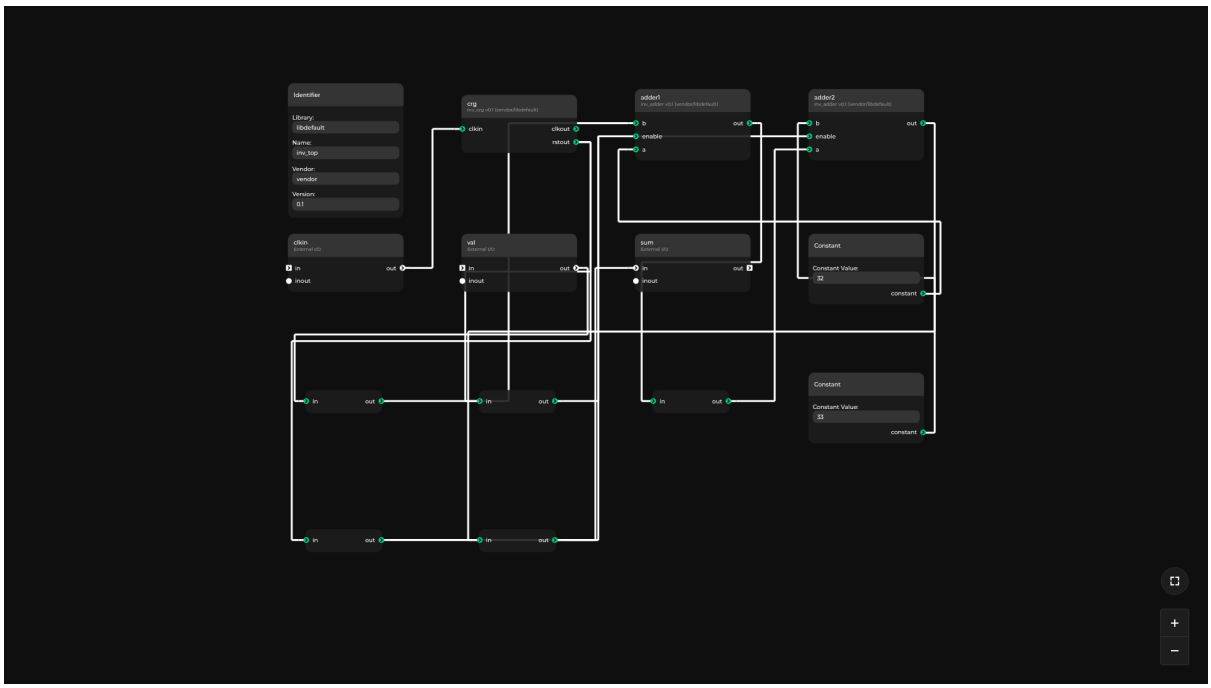
27.7 SoC_AXI

This example has connected manager and two UART subordinates to AXI interconect. Interconnect is generated by Topwrap. UART0 subordinate has address 0xF0000000 and size 0x1000. UART1 subordinate has address 0xF0001000 and size 0x1000.



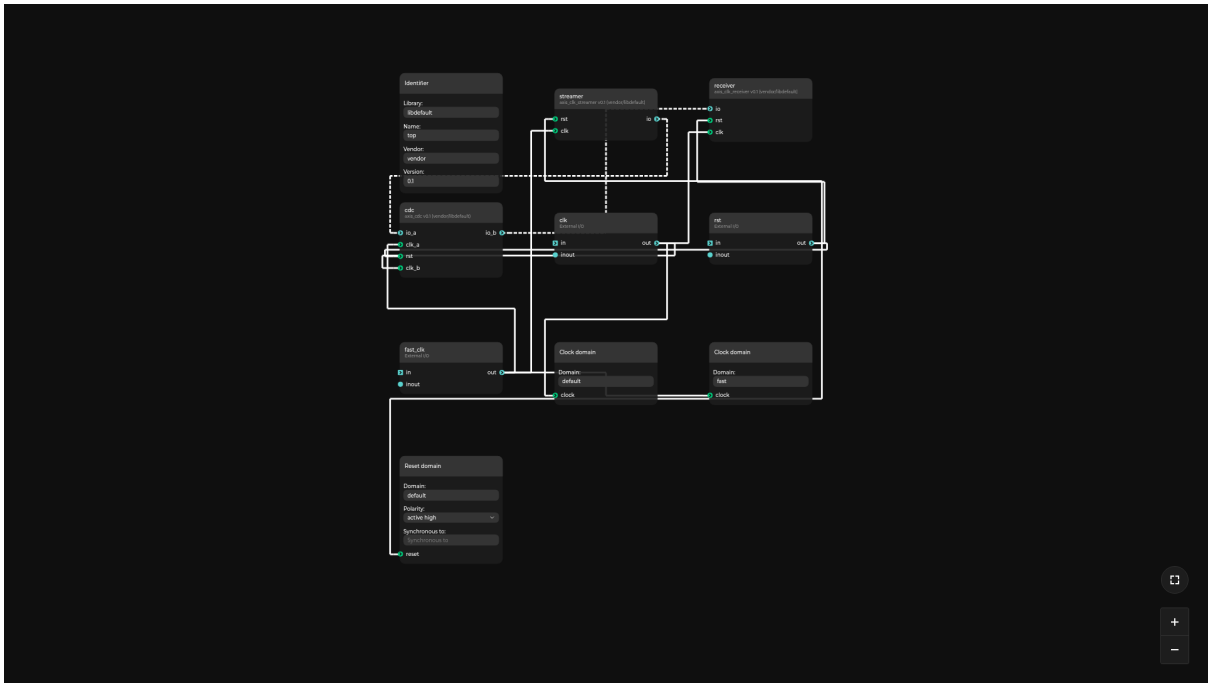
27.8 Inverted

This is an example showcasing the ability to invert (negate) connections between ports of modules and external ports.



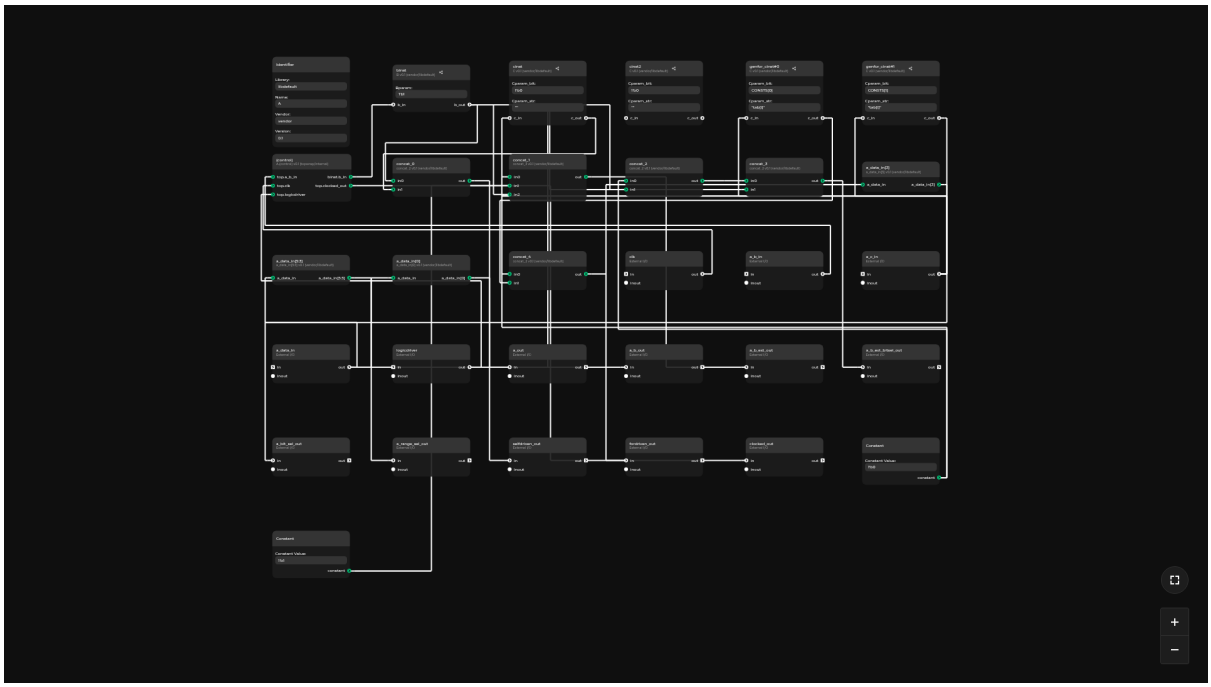
27.9 Clocks

This is an example showcasing the clock and reset domain functionality.



27.10 Complex hierarchy parsing

This example shows a complex hierarchy with subgraphs parsed from SystemVerilog files



27.11 Other

Something that was not taken into account previously, because we don't support it yet, and it's impossible to represent in either format, is a feature/syntax that would allow us to dynamically change the collection of ports/interfaces an IP/hierarchy has. Similarly to how we can control the width of a port using a parameter (like in the "simple" example).

CHAPTER

TWENTYEIGHT

IP-XACT FORMAT

This document is an exploration of the IP-XACT format.

All IP-XACT elements generated for the IR examples are located under `examples/ir_examples/[example]/ipxact/antmicro.com/[example]` where `antmicro.com/[example]` represents the `vendor/library`. They all conform to the 2022 version. These files can be regenerated with the `topwrap ipxact_gen` command (see *Generating IPXACT 2022 files*), which is driven by the *IPXACTBackend*.

28.1 General observations

28.1.1 VLNV

The IP-XACT format enforces the usage of VLNV (vendor, library, name, version) for every single design and component.

```
<ipxact:vendor>antmicro.com</ipxact:vendor>
<ipxact:library>simple</ipxact:library>
<ipxact:name>lfsr_gen</ipxact:name>
<ipxact:version>1.2</ipxact:version>
```

For now, Topwrap can only reliably handle the `name` value, while `vendor` and `version` are not used anywhere and their concept is unrecognised in the codebase. Arguably, `library` could be represented by the name of a user repository.

Special consideration needs to be taken for these values, as the XML schema defines specific allowed characters for some fields, while Topwrap doesn't sanity-check any fields that accept custom names.

Warning: Later in this document this group of four tags will be represented by `<VLNV... />` to avoid repetition.

28.1.2 Multiple versions

There are many versions of the IP-XACT schema, as visible here, on the official page of Accellera - developers of the format.

Version before 2014 and after 2014 use two different XML namespaces for the tags, respectively: `spirit:` and `ipxact:`.

Vivado seemingly only supports the 2009(!) specification version.

This means the discrepancies between different versions and incompatibilities between tools must be taken into account.

There are official XSLT templates (bottom of the page) available that can convert any IP-XACT .xml file one version up, using an xslt tool like `xsltproc`.

28.1.3 Design structure

The IP-XACT format revolves mainly around "components". This is something that is closest to our `IPCoreDescription` class and its respective YAML schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:component>
    <VLNV... />
        <ipxact:model>
                <ipxact:instantiations>
                        <ipxact:componentInstantiation>
                                <ipxact:moduleParameters>
                                        ...
                                </ipxact:moduleParameters>
                        </ipxact:componentInstantiation>
                </ipxact:instantiations>
                <ipxact:ports>
            ...
                </ipxact:ports>
        </ipxact:model>
        <ipxact:parameters>
                ...
        </ipxact:parameters>
</ipxact:component>
```

A singular component represents a black-box, with the outside world seeing only its ports, buses and parameters. In order to represent its inner design there needs to be a separate design XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<ipxact:design>
        <VLNV...  />
        <ipxact:componentInstances>
        ...
        </ipxact:componentInstances>
        <ipxact:adHocConnections>
```

(continues on next page)

(continued from previous page)

```
                <ipxact:adHocConnection>
                        <ipxact:name>gen2_gen_out_to_two_mux_gen2</ipxact:name>
                        <ipxact:portReferences>
                                <ipxact:internalPortReference␣
↪componentInstanceRef="ip1" portRef="port1"/>
                                <ipxact:internalPortReference␣
↪componentInstanceRef="ip2" portRef="port1"/>
                        </ipxact:portReferences>
                </ipxact:adHocConnection>
                ...
        </ipxact:adHocConnections>
</ipxact:design>
```

which later *is attached* to the component description under the instantiations section, thus making the design an optional property of a module/component.

To describe a top-level wrapper you need both its description as a component, where the external IO is defined, and its design file that describes what other IPs are incorporated by this wrapper.

28.1.4 Parameter passing

IP-XACT introduces a distinction between parameters of a component, and module parameters of the component's instantiation.

This allows most IP-XACT objects to accept parameters that are only internal to them and are unrelated to the potentially generated RTL. In order to define RTL module parameters, you need to specify them under two separate sections.

Below is an example of defining a `paramWIDTH` parameter with default value of 64 in a component that gets realised in Verilog as `parameter WIDTH = 64;`:

Take note of the top-level `<ipxact:parameters>` tag and the `<ipxact:moduleParameters>` tag of the component instantiation.

```
<ipxact:instantiations>
    <ipxact:componentInstantiation>
        <ipxact:name>rtl</ipxact:name>
        <ipxact:displayName>rtl</ipxact:displayName>
        <ipxact:language>Verilog</ipxact:language>
        <ipxact:moduleParameters>
            <ipxact:moduleParameter>
                <ipxact:name>WIDTH</ipxact:name>
                <ipxact:displayName>WIDTH</ipxact:displayName>
                <ipxact:value>paramWIDTH</ipxact:value>
            </ipxact:moduleParameter>
        </ipxact:moduleParameters>
    </ipxact:componentInstantiation>
</ipxact:instantiations>
<ipxact:parameters>
    <ipxact:parameter parameterId="paramWIDTH" resolve="user" type="longint">
```

(continues on next page)

(continued from previous page)

```
        <ipxact:name>paramWIDTH</ipxact:name>
        <ipxact:displayName>paramWIDTH</ipxact:displayName>
        <ipxact:value>64</ipxact:value>
    </ipxact:parameter>
</ipxact:parameters>
```

In Topwrap, all IP parameters do get realised in the generated Verilog and there is no notion of internal parameters.

28.1.5 File sets

Each component in IP-XACT can contain an <ipxact:fileSets> section. This is a very exhaustive section about one or more groups of *files* that this component depends on. The type and purpose of every such file is marked, e.g: `verilogSource`.

```
<ipxact:fileSets>
    <ipxact:fileSet>
        <ipxact:name>fs-rtl</ipxact:name>
        <ipxact:file>
            <ipxact:name>../RTL/transmitter.v</ipxact:name>
            <ipxact:fileType>verilogSource</ipxact:fileType>
            <ipxact:logicalName>transmitter_lib</ipxact:logicalName>
        </ipxact:file>
    </ipxact:fileSet>
</ipxact:fileSets>
```

This concept currently only exists as a `--sources` CLI option for `topwrap generate --fusesoc`, where all recognized HDL and constraint sources are forwarded to the FuseSoC `.core`. There is no notion of other file dependencies inside IP Core description YAMLs.

28.1.6 Vendor extensions

The IP-XACT format allows storing completely custom data inside most of the tags using the `<ipxact:vendorExtensions>` group. Topwrap could use them to store additional data about the IPs or designs.

Example theoretical vendor extensions:

```
<ipxact:vendorExtensions>
    <topwrap:interconnectType>wishbone</topwrap:interconnectType>
    <topwrap:kpm_position x="600" y="180" />
    <topwrap:repo>builtin</topwrap:repo>
</ipxact:vendorExtensions>
```

28.1.7 Catalogs

Catalogs describe the location and the VLNV identifier of other IP-XACT elements such as components, designs, buses etc. in order to manage and allow access to collections of IP-XACT files. In most cases defining a catalog is not required as all necessary files are automatically located by the used tool.

```
<ipxact:catalog>
    <VLNV... />
    <ipxact:components>
        <ipxact:ipxactFile>
            <ipxact:vlnv vendor="antmicro.com" library="simple" name="lfsr"␣
↪version="1.0" />
            <ipxact:name>./antmicro.com/simple/lfsr/lfsr.1.0.xml</ipxact:name>
        </ipxact:ipxactFile>
    </ipxact:components>
    <ipxact:busDefinitions>
        ...
    </ipxact:busDefinitions>
    ...
</ipxact:catalog>
```

28.2 Simple example

This is the simplest IP-XACT example as it contains only plain IP cores with standalone ports, and parameters.

28.2.1 Instance names

Since Topwrap doesn't verify any user-defined names, an accidental creation of a `2mux.yaml` IP Core named `2mux_compressor` instantiated with a `2mux` name, was possible in the YAML format. Many environments, IP-XACT included, don't actually allow users to start custom names with a number. The instance name of `2mux` had to be changed to `two_mux` for this purpose.

28.2.2 Parameters

The special syntax of IP-XACT parameters is mostly explained in the *Parameter passing* section.

Variable widths

If you look at either `ips/2mux.yaml` or `ips/lfsr_gen.yaml` you'll see that there are ports with widths defined by the parameters inside an arithmetic expression:

```
# ips/2mux.yaml
out:
    - [out, OUT_WIDTH-1, 0]
```

This is easily realisable in IP-XACT because just like our port widths, they also accept arbitrary arithmetic expressions that can reference other parameters inside them:

```
<ipxact:port>
    <ipxact:name>out</ipxact:name>
    <ipxact:wire>
        <ipxact:direction>out</ipxact:direction>
        <ipxact:vectors>
            <ipxact:vector>
                <ipxact:left>paramOUT_WIDTH - 1</ipxact:left>
                <ipxact:right>0</ipxact:right>
            </ipxact:vector>
        </ipxact:vectors>
    </ipxact:wire>
</ipxact:port>
```

28.2.3 Duality of the design description

The design of the *Simple* example is defined (from the Topwrap's perspective) purely in the `design.yaml` file. This is not so simple in IP-XACT, see *Design structure*.

Mostly this means that the "external" section of our design YAML lands in its own component/IP file and the connections and module instances in a separate one that is attached to the component file as a "design instantiation".

The generated top-level component for this example and its design (`top.design.1.0.xml`) are located inside the `top` directory in the IP-XACT library.

Additionally a "design configuration" file is generated that contains additional configuration information for the main design file. Not much is specified there for this example though.

So finally the original `design.yaml` ends up becoming 3 interconnected .xml files in IP-XACT.

28.2.4 Connections

Port connections between IP cores, and IP cores and externals are all specified in the XML design file. There isn't much special about them, they are represented very similarly to our design description yaml connections:

```
<ipxact:adHocConnection>
    <ipxact:name>gen2_gen_out_to_two_mux_gen2</ipxact:name>
    <ipxact:portReferences>
        <ipxact:internalPortReference componentInstanceRef="gen2" portRef="gen_out
↪"/>
        <ipxact:internalPortReference componentInstanceRef="two_mux" portRef="gen2
↪"/>
    </ipxact:portReferences>
</ipxact:adHocConnection>
```

28.3 Interface example

The key thing about this example is that it uses an interface connection (AXI 4 Stream) between two IPs, an inout port, a constant value supplied to a port and *port slicing*.

Info

An interface is a named, predefined collection of logical signals used to transfer information between different IPs or other building blocks. Common interface types include: Wishbone, AXI, AHB, and more.

Topwrap, like SystemVerilog, refers to this concept as an "interface".

IP-XACT refers to the same concept as a "bus".

28.3.1 Bus definitions

Custom interfaces in Topwrap are defined using *interface definition files*.

Custom interfaces are well recognized and supported in IP-XACT. They are represented by two files, a "bus definition" that defines the existence of the interface/bus itself, its name and configurable parameters; and an "abstraction definition" that defines the logical signals of the interface.

It's possible to have more than one abstraction definition for a given bus definition.

Often times the necessary definitions for a given interface are already publicly available. For example, the IP-XACT bus definitions of all ARM AMBA interfaces are available here in the 2009 version of IP-XACT. For this document, they were up-converted to the 2022 version with the help of *XSLT templates*.

Format

If not, a custom definition has to be created. Starting with the bus definition:

```
<ipxact:busDefinition>
  <VLNV... />
  <ipxact:description>This is the AXI4Stream stream bus definition.</
↪ipxact:description>
  <ipxact:directConnection>true</ipxact:directConnection>
  <ipxact:isAddressable>false</ipxact:isAddressable>
</ipxact:busDefinition>
```

VLNV entries and description are both present at the start, like in all other IP-XACT definitions. Then there are two configuration bools:

- `<ipxact:directConnection>` decides if this bus allows direct connection between a manager/initiator and subordinate/targets. Important for "asymmetric buses such as AHB".
- `<ipxact:isAddressable>` decides if this bus is addressable using the address space of the manager side of the bus. e.g. `true` for AXI4, `false` for AXI4Stream.

Then to specify the logical signals of the interface, an abstraction definition has to be created:

```
<ipxact:abstractionDefinition>
        <VLNV... />
        <ipxact:description>This is an RTL Abstraction of the AMBA4/AXI4Stream</
↪ipxact:description>
        <ipxact:busType vendor="amba.com" library="AMBA4" name="AXI4Stream"␣
↪version="r0p0_1"/>
        <ipxact:ports>
                <ipxact:port>
                        <ipxact:logicalName>TREADY</ipxact:logicalName>
                        <ipxact:description>indicates that the Receiver can␣
↪accept a transfer in the current cycle.</ipxact:description>
                        <ipxact:wire>
                                <ipxact:onInitiator>
                                        <ipxact:presence>optional</
↪ipxact:presence>
                                        <ipxact:width>1</ipxact:width>
                                        <ipxact:direction>in</ipxact:direction>
                                </ipxact:onInitiator>
                                <ipxact:onTarget>
                                        <ipxact:presence>optional</
↪ipxact:presence>
                                        <ipxact:width>1</ipxact:width>
                                        <ipxact:direction>out</ipxact:direction>
                                </ipxact:onTarget>
                                <ipxact:defaultValue>1</ipxact:defaultValue>
                        </ipxact:wire>
                </ipxact:port>
    </ipxact:ports>
</ipxact:abstractionDefinition>
```

This is a fragment of the `TREADY` signal definition of the AXI 4 Stream interface.

There's the classic VLNV + Description combo at the start, then the associated bus definition is referenced and lastly the signals of the interface are defined.

In IP-XACT, unlike in Topwrap, you can specify different options for signals on both the manager and the subordinate separately, importantly a signal can be required on one side of the bus while being optional on the other. This is currently impossible to represent in Topwrap. The width specification and the default value are not supported either by Topwrap.

Moreover, unlike in Topwrap, in IP-XACT the clock and reset signals are also specified in the definition alongside other signals. They are however marked with special qualifiers that distinguish their roles and enforce certain behaviours.

Example qualifiers:

```
<ipxact:wire>
    <ipxact:qualifier>
        <ipxact:isClock>true</ipxact:isClock>
        <ipxact:isReset>true</ipxact:isReset>
    </ipxact:qualifier>
</ipxact:wire>
```

Info

While Topwrap uses the `manager` and `subordinate` terms to refer to the roles an IP can assume in the bus connection, IP-XACT pre-2022 uses `master`, `slave` and IP-XACT 2022-onwards uses `initiator` and `target` respectively.

Interface deduction

Topwrap supports specifying both a regex for each signal and the port prefix for the entire interface in order to *automatically group raw ports* from HDL sources into interfaces. None of that is possible to represent in IP-XACT, though this information can be stored anyways using *Vendor extensions*.

28.3.2 Bus instantiation

To use the bus inside a component definition you have to:

- Add all the physical ports that will get used as the bus signals just like regular *ad-hoc ports*
- Map these physical ports to logical ports of the interface

The portMap format

```
interfaces:
    io:
        type: AXI4Stream
        mode: subordinate
        signals:
            in:
                TDATA: [dat_i, 31, 0]
```

This fragment of *Design description* would translate to the below IP-XACT description, assuming the `dat_i` signal was previously defined in the ad-hoc ports section.

```
<ipxact:busInterfaces>
    <ipxact:busInterface>
        <ipxact:name>io</ipxact:name>
        <ipxact:busType vendor="amba.com" library="AMBA4" name="AXI4Stream"␣
↪version="r0p0_1"/>
        <ipxact:abstractionTypes>
            <ipxact:abstractionType>
                <ipxact:abstractionRef vendor="amba.com" library="AMBA4" name=
↪"AXI4Stream_rtl" version="r0p0_1"/>
                <ipxact:portMaps>
                    <ipxact:portMap>
                        <ipxact:logicalPort>
                            <ipxact:name>TDATA</ipxact:name>
                        </ipxact:logicalPort>
```

(continues on next page)

(continued from previous page)

```
                        <ipxact:physicalPort>
                            <ipxact:name>dat_i</ipxact:name>
                        </ipxact:physicalPort>
                    </ipxact:portMap>
                </ipxact:portMaps>
            </ipxact:abstractionType>
        <ipxact:abstractionTypes>
        <ipxact:target/>
    </ipxact:busInterface>
</ipxact:busInterfaces>
```

The `<ipxact:busInterfaces>` tag is a direct child of the top-level `<ipxact:component>` tag.

Port slicing is supported as well:

```
<ipxact:physicalPort>
    <ipxact:name>ctrl_i</ipxact:name>
    <ipxact:partSelect>
        <ipxact:range>
            <ipxact:left>4</ipxact:left>
            <ipxact:right>4</ipxact:right>
        </ipxact:range>
    </ipxact:partSelect>
</ipxact:physicalPort>
```

28.3.3 Inout ports

This example contains an external inout port raised from one of the IPs. While the *Topwrap syntax* for specifying inout ports in a design is a bit awkward, in IP-XACT inout ports are represented just like ports with other directions.

28.3.4 Constant assignments

This example also features a constant value (2888) assigned to the `noise` port of the `receiver` IP instead of any wire. In IP-XACT this is done similarly to *Connections*:

```
<ipxact:adHocConnection>
    <ipxact:name>receiver_0_noise_to_tiedValue</ipxact:name>
    <ipxact:tiedValue>2888</ipxact:tiedValue>
    <ipxact:portReferences>
        <ipxact:internalPortReference componentInstanceRef="receiver_0" portRef=
↪"noise"/>
    </ipxact:portReferences>
</ipxact:adHocConnection>
```

Additionally, the `tiedValue` can be given by an arithmetic expression that resolves to a constant value.

28.4 Hierarchical example

The hierarchical example features deeply nested hierarchies. The purpose of a hierarchical design is to group together into separate levels/modules, connections that could just as well be realised flatly in the top-level.

In Topwrap, all hierarchies are specified in the respective *design description file* YAML using a special syntax that allows multiple design descriptions to be nested together in a single file.

IP-XACT has no notion of any special syntax for hierarchies, because it doesn't need to. Due to the *architecture of design XMLs* being extensions to component XMLs, it's possible to just generate a component+design pair for every hierarchy and connect them just as if they were regular IPs that happen to have a design available alongside them. This is exactly what was done to represent this example.

28.5 Interconnect example

This example features the *Interconnect generation* functionality of Topwrap.

Specifying interconnects in the Topwrap design description implies dynamic generation of necessary arbiters and bus components during build-time using parameters defined under the interconnect instance key.

IP-XACT doesn't support such functionality because it's just a file format and it doesn't necessarily have any dynamic code associated with it.

Conversion from Topwrap -> IP-XACT should probably just generate the interconnect bus component with the required amount of manager and subordinate ports and package it alongside the generated RTL implementation of routers and arbiters.

Reverse conversion (from the concrete generated IP-XACT interconnect to Topwrap's interconnect entry) is probably impossible, we can't know the interconnect specifics to know which type to pick after it's already generated. However, all this necessary information could be stored in a vendor extension.

28.5.1 The interconnect component

The generated interconnect is located in `./antmicro.com/interconnect/interconnect/wishbone_interconnect1.xml`. As mentioned, it has just enough interface ports to connect the two specified managers and two subordinates.

The Wishbone interface definition from `opencores.org` was used.

The main difference that differentiates the interconnect component from raw interface connections like in the *Interface example* is the explicit definition and mapping of the address space with the `<ipxact:addressSpaces>` tag and assignment of each manager port to one or more subordinates.

The extensions used in the bus instance element in the component definition. Focus on the `ipxact:addressSpaceRef` tag where the base address of this subordinate is specified:

```
<ipxact:busInterface>
    <ipxact:name>target_1</ipxact:name>
    <ipxact:busType vendor="opencores.org" library="interface" name="wishbone"␣
↪version="b4"/>
    <ipxact:abstractionTypes>
        ...
    </ipxact:abstractionTypes>
    <ipxact:initiator>
        <ipxact:addressSpaceRef addressSpaceRef="address">
            <ipxact:baseAddress>'h10000</ipxact:baseAddress>
        </ipxact:addressSpaceRef>
    </ipxact:initiator>
</ipxact:busInterface>
```

The extension used at the top-level in the component definition to map the address space:

```
<ipxact:addressSpaces>
    <ipxact:addressSpace>
        <ipxact:name>address</ipxact:name>
        <ipxact:range>2**32/8-1</ipxact:range>
        <ipxact:width>8</ipxact:width>
        <ipxact:segments>
            <ipxact:segment>
                <ipxact:name>mem</ipxact:name>
                <ipxact:addressOffset>'h0</ipxact:addressOffset>
                <ipxact:range>'hFFFF+1</ipxact:range>
            </ipxact:segment>
            <ipxact:segment>
                <ipxact:name>dsp</ipxact:name>
                <ipxact:addressOffset>'h10000</ipxact:addressOffset>
                <ipxact:range>'hFF+1</ipxact:range>
            </ipxact:segment>
        </ipxact:segments>
        <ipxact:addressUnitBits>8</ipxact:addressUnitBits>
    </ipxact:addressSpace>
</ipxact:addressSpaces>
```

The assignment of a manager port to specified subordinates(targets):

```
<ipxact:busInterface>
    <ipxact:name>manager0</ipxact:name>
    <ipxact:busType vendor="opencores.org" library="interface" name="wishbone"␣
↪version="b4"/>

    <ipxact:target>
        <ipxact:transparentBridge initiatorRef="target_0"/>
        <ipxact:transparentBridge initiatorRef="target_1"/>
    </ipxact:target>
</ipxact:busInterface>
```

28.5.2 External interface

In the Topwrap definition of this example, a `wishbone_passthrough` IP core is used in order to allow the external interface to be connected as a manager to the interconnect. This is due to limitations of the schema and the fact that under the `managers` key Topwrap expects the IP instance name with the specified manager port, completely disregarding the possibility of it being external.

28.6 Other features

28.6.1 Dynamic number of ports/interfaces based on a parameter

This is not possible in IP-XACT. All ports/interfaces and connections need to be explicitly defined. While the amount of bits in a port can vary based on a parameter value, as was presented in *Variable widths*, higher level concepts such as the number of ports cannot.

28.7 Conclusion

In most aspects IP-XACT is a superset of what's possible to describe in Topwrap, making the Topwrap -> IP-XACT conversion pretty trivial.

Syntax impossible to represent natively in IP-XACT such as:

- Abstract interconnects without concrete implementation
- Interface signal name regexes and port prefixes (see *Interface inference*)

can even if not implemented, be at least preserved using *Vendor extensions*.

Other visible issue for this conversion are:

- *VLNV* being mandatory for IP-XACT files, but Topwrap containing only the name information
- Lack of input sanitization of string fields on Topwrap's side

On the other hand, the conversion from a generic IP-XACT file to Topwrap's internal representation may prove more tricky and definitely suffer from information loss as the IP-XACT format is packed with more features and elements that are not exactly useful for our purposes and were not even mentioned in this document at all.

CHAPTER

TWENTYNINE

GENERATOR

Generator is used for generating ModuleInstance and HDL code. There can be a Generator for each Interconnect and for each Backend.

Each Generator needs to implement the generate() method that will generate interconnect code specific for used backend.

```
generate(self, interconnect: _IT, module_instance: ModuleInstance) -> _T:
  pass
```

_IT is bound to Interconnect and is set by interconnect-specific implementation (e.g. WishboneRRSystemVerilogGenerator).
_T isn't bound and is set by backend-specific implementation (e.g. SystemVerilogGenerator).

Note: Backend-specific generators inheritance from Generator (e.g. SystemVerilogGenerator).
Interconnect-specific generators inheritance from backend-specific (e.g. WishboneRRSystemVerilogGenerator).

Generated HDL code need to follow naming convention, for ports it is set by get_name(), and for Module by default it is interconnect_{interconnect.name}. add_module_instance_to_design() is used to create ModuleInstance and add it to Design, it can be overridden when generated HDL code needs it.

Important: clk and rst ports are always generated by default, if generate() method don't create these ports it is needed to override add_module_instance_to_design() to change that behavior.

Backend-specific implementation can introduce new methods that the interconnect-specific class can implement or use. Each backends need to have its own lookup list that contains Interconnect and is mapped to Generator

29.1 How to implement a Generator

Based on example of `WishboneRRSystemVerilogGenerator`, a new SystemVerilog-based generator can be implemented as follows. Create subclass of `SystemVerilogGenerator` and implement `generate()`. It needs to return generated System Verilog code, the code needs to have same ports as generated `ModuleInstance`. To make name generation less prone to errors use `get_name()` in implementation of `generate()` or override `get_name()` with your naming convention.

29.2 Lookup maps

`SystemVerilogBackend` uses `verilog_generators_map` as lookup map. `IPXACTBackend` reuses the same `verilog_generators_map`, since different HDL backends may yield different IR modules for interconnects.

29.3 API Reference

class Generator

add_module_instance_to_design(interconnect: *_INTERCONNECT*) → *ModuleInstance*

Returns generated *ModuleInstance* based on *Interconnect*, generated *ModuleInsance* always has *rst* and *clk*, *ModuleInstance* also has additional ports and interfaces based on what bus is used with managers and subordinates. *ModuleInstance `s of subordinates and managers are connected to generated `ModuleInsatnce*

Parameters

interconnect: _INTERCONNECT
Interconnect to represent as *ModuleInstance*

abstract generate(interconnect: *_INTERCONNECT*, module_instance: ModuleInstance) → _HDL

Returns generated HDL code wrapped in class specific for backend

Parameters

interconnect: _INTERCONNECT
HDL code is generated based on this *Interconnect*

module_instance: *ModuleInstance*
generated based on *Interconnect*, it don't need to be used for generation, but can be helpful

Raises

InterconnectGenerationError – Raised when there is problem with generating HDL code

get_name(referenced_interface: ReferencedInterface, signal: InterfaceSignal) → str

Returns name for *InterfaceSignal*, generated backend specific code need to have same naming convention

Parameters

referenced_interface: *ReferencedInterface*
InterfaceInstance containing this *InterfaceSignal*

signal: *InterfaceSignal*
Signal to give name

class SystemVerilogGenerator

It is System Verilog specific generator, it's empty and need subclass for each Interconnect that SV backend need to support

class WishboneRRSystemVerilogGenerator

add_module_instance_to_design(interconnect: *WishboneInterconnect*) → *ModuleInstance*

Returns generated *ModuleInstance* based on *Interconnect*, generated *ModuleInstance* always has *rst* and *clk*, *ModuleInstance* also has additional ports and interfaces based on what bus is used with managers and subordinates. *ModuleInstance `s of subordinates and managers are connected to generated `ModuleInstance*

Parameters

interconnect: WishboneInterconnect
Interconnect to represent as *ModuleInstance*

generate(interconnect: *WishboneInterconnect*, module_instance: ModuleInstance) → SVFile

Returns generated HDL code wrapped in class specific for backend

Parameters

interconnect: WishboneInterconnect
HDL code is generated based on this *Interconnect*

module_instance: *ModuleInstance*
generated based on *Interconnect*, it don't need to be used for generation, but can be helpful

Raises
InterconnectGenerationError – Raised when there is problem with generating HDL code

get_name(referenced_interface: ReferencedInterface, signal: InterfaceSignal) → str

Returns name for *InterfaceSignal*, generated backend specific code need to have same naming convention

Parameters

referenced_interface: *ReferencedInterface*
InterfaceInstance containing this *InterfaceSignal*

signal: *InterfaceSignal*
Signal to give name

class AXIVerilogGenerator

add_module_instance_to_design(interconnect: *AXIInterconnect*) → *ModuleInstance*

Returns *ModuleInstance* with ports interfaces generated based on *Interconnect ModuleInstance* and connections are added to design

Parameters

interconnect: AXIInterconnect
Interconnect to represent as *ModuleInstance*

generate(interconnect: *AXIInterconnect*, module_instance: ModuleInstance) → SVFile

Returns generated HDL code wrapped in class specific for backend

Parameters

interconnect: AXIInterconnect
HDL code is generated based on this *Interconnect*

module_instance: ***ModuleInstance***
generated based on *Interconnect*, it don't need to be used for generation, but can be helpful

Raises

InterconnectGenerationError – Raised when there is problem with generating HDL code

get_name(referenced_interface: ReferencedInterface, signal: InterfaceSignal) → str

Returns name for *InterfaceSignal*, generated backend specific code need to have same naming convention

Parameters

referenced_interface: ***ReferencedInterface***
InterfaceInstance containing this *InterfaceSignal*

signal: ***InterfaceSignal***
Signal to give name

verilog_generators_map : dict[type[*Interconnect*], type[*SystemVerilogGenerator*[Any]]]

Used by SV backend to get correct generator. All implementations of *Generator* for SV backend need to be present in this map.

CHAPTER

THIRTY

INTERCONNECT

This document is about implementing new `Interconnect`, check *Interconnect generation* to read about `Interconnect` concept in Topwrap.

`Interconnect` is base class for all interconnects, it has 3 generic classes that are used to represent params: `InterconnectParams`, `InterconnectManagerParams` and `InterconnectSubordinateParams`. All new implementations of interconnect need to be placed in in `topwrap/interconnects/` and added to `INTERCONNECT_TYPES`. Each interconnect needs to have implemented `Generator`, refer to *Generator* to check how to implement one.

Base class for multiple interconnect generator implementations.

Interconnects connect multiple interface instances together in a many-to-many topology, combining multiple subordinates into a unified address space so that one or multiple managers can access them.

Base class for parameters/settings specific to a concrete interconnect type

Base class for subordinate parameters specific to a concrete interconnect type.

Transactions to addresses in range [self.address; self.address + self.size) will be routed to this subordinate.

Base class for manager parameters specific to a concrete interconnect type
InterconnectTypeInfo(intercon: Type[topwrap.model.interconnect.Interconnect], params: Type[topwrap.model.interconnect.InterconnectParams], man_params: Type[topwrap.model.interconnect.InterconnectManagerParams], sub_params: Type[topwrap.model.interconnect.InterconnectSubordinateParams])

INTERCONNECT_TYPES : dict[str, InterconnectTypeInfo]

Maps name to specific interconnect implementation. Used by YAML frontend.

CHAPTER

THIRTYONE

REPOSITORY

Repositories are used for storing, packaging and loading different types of resources that can be used by Topwrap.

An example of the implementation of a concrete repository is the `topwrap.repo.user_repo.UserRepo` class. Its user interface is documented in *Constructing, configuring and loading repositories*.

31.1 API reference

class Resource(name: *str*)

Base class for representing a resource in a repository. Each derived resource should define its own structure.

name : str

Name of the resource

class ResourceHandler

Base for classes that can perform various operations on resources of a given type.

The main responsibilities of a resource handler are saving and loading resources of a given type in a repository.

add_resource(resources: *dict[str, ResourceType]*, res: *ResourceType*, exists_strategy: *ExistsStrategy* = `ExistsStrategy.RAISE`)

Custom behavior for adding a resource to a loaded repository :raises ResourceExistsException: Resource already exists in the repository

abstract load(repo_path: *Path*) → Iterator[ResourceType]

Loads list of resources from the repo_path repository

remove_resource(resources: *dict[str, ResourceType]*, res: *ResourceType*)

Custom behavior for removing a resource from a loaded repository :raises ResourceNotFoundException: Could not find that resource

resource_type : Type[ResourceType]

For which resource type this handler is responsible

abstract save(res: *ResourceType*, repo_path: *Path*) → None

Saves a resource in the repo_path repository

class Repo(resource_handlers: *list[*ResourceHandler*[*Resource*]]*, name: *str*)

Base class for implementing repositories. A repository is a container for resources of various types.

Derived classes should be associated with a set of supported resource handlers that define what types of resources can be stored in the repo. An example of such derived repo is `UserRepo`

add_files(handler: FileHandler, exist_strategy: *ExistsStrategy* = `ExistsStrategy.RAISE`) → None

Parses resources available in files and adds them to the repository

Parameters

handler: *FileHandler*
Handler that contains sources

exist_strategy: ExistsStrategy = ExistsStrategy.RAISE
What to do if resource exists in repo already

Raises

- **ResourceExistsException** – Raised when exist_strategy is set to RAISE
- **ResourceNotSupportedException** – Raised when handler returns resources not supported by repo

add_resource(resource: Resource, exist_strategy: *ExistsStrategy* = `ExistsStrategy.RAISE`) → None

Adds a single resource to the repository

Parameters

resource: *Resource*
Resource to add to repo

exist_strategy: ExistsStrategy = ExistsStrategy.RAISE
What to do if resource exists

Raise
ResourceExistsException: Raised if exist_strategy is set to RAISE

Raise
ResourceNotSupportedException: Raised if the repository doesn't have a handler for this resource type

get_resource(resource_type: *type[ResourceType]*, name: *str*) → ResourceType

Searches for resource with given type in repository

Raises

- **ResourceNotFoundException** – Raised when resource with given name isn't present
- **ResourceNotSupportedException** – Raised when resource is not supported by repo implementation

get_resources(type: *type[ResourceType]*) → list[ResourceType]

Implements the same operation as self.resources[type] but gives correct hints to the typechecker

load(repo_path: *Path*, **kwargs: *Any*) → None

Loads repository from repo_path

name : str

Name of the repository

remove_resource(resource: Resource) → None

Removes a single resource from repository

Parameters

resource: ***Resource***
Resource to remove from repo

Raises

- **ResourceNotFoundException** – Raised when resource is not present in repository
- **ResourceNotSupportedException** – Raised if repo don't have handler for this resource type

save(dest: *Path*, **kwargs: *Any*) → None

Saves repository to dest :raises ResourceNotSupportedException:

class FileHandler(files: *Iterable[File]*)

Base class for file handlers. A file handler is used to extract repository resources from a set of given files.

abstract parse() → list[*Resource*]

Parses a file to extract resources

CHAPTER

THIRTYTWO

LOGGING

32.1 Using a Custom Logging Configuration

A custom logging configuration file can be provided with the `--log-cfg` option:

```
topwrap --log-cfg logger.cfg build --design {design_name.yaml}
```

Example configuration:

```
[loggers]
keys=root,frontend,backend

[handlers]
keys=rootStreamHandler,rootFileHandler,frontendStreamHandler,frontendFileHandler,
↪backendStreamHandler,backendFileHandler

[formatters]
keys=rootFormatter,frontendFormatter,backendFormatter

[logger_root]
level=DEBUG
handlers=rootStreamHandler,rootFileHandler

[logger_frontend]
level=DEBUG
handlers=frontendFileHandler,frontendStreamHandler
qualname=topwrap.frontend.yaml.frontend
propagate=0

[logger_backend]
level=DEBUG
handlers=backendFileHandler,backendStreamHandler
qualname=topwrap.backend.sv.backend
propagate=0

[handler_rootStreamHandler]
class=StreamHandler
formatter=rootFormatter
args=(sys.stdout,)
```

(continues on next page)

(continued from previous page)

```
[handler_rootFileHandler]
class=FileHandler
formatter=rootFormatter
args=("topwrap.log",)

[handler_frontendStreamHandler]
class=StreamHandler
formatter=frontendFormatter
args=(sys.stdout,)

[handler_frontendFileHandler]
class=FileHandler
formatter=frontendFormatter
args=("topwrap.log","w")

[handler_backendStreamHandler]
class=StreamHandler
formatter=backendFormatter
args=(sys.stdout,)

[handler_backendFileHandler]
class=FileHandler
formatter=backendFormatter
args=("topwrap.log","w")

[formatter_rootFormatter]
format=%(levelname)s:%(message)s

[formatter_frontendFormatter]
format=%(levelname)s:frontend:%(message)s

[formatter_backendFormatter]
format=%(levelname)s:backend:%(message)s
```

If both `--log-level` and `--log-cfg` are provided, the logging configuration file is loaded first, and the log level specified by `--log-level` overrides the configured levels.

32.2 Creating a New Logger

Since Topwrap relies on Python's standard `logging` module, users can easily add custom loggers.

Each logger should correspond to a Python module and use the module name as its `qualname`.

The module name can be obtained by:

- printing `__name__` inside the module, or
- converting the file path into dotted module notation.

For example:

```
topwrap/frontend/frontend.py
```

corresponds to:

```
topwrap.frontend.frontend
```

CHAPTER

THIRTYTHREE

PLUGINS

Plugins are Python modules that are installed in the same environment as Topwrap. They must define an entry point in the `topwrap.plugins` group. The entry point is a class that inherits from `BasePlugin`. `pyproject.toml` must contain the following:

```
[project.entry-points."topwrap.plugins"]
<plugin-name> = "<plugin-name>.<module-path>:<PluginBase-subclass>"
```

Important: Plugin support is experimental and both the mechanism and API may change partially or completely between Topwrap versions.

33.1 Examples

Use an existing plugin as a base for developing your own: github.com/antmicro/topwrap-utils.

33.2 Plugin Base

By defining implementations for the various hooks defined by `BasePlugin`, the plugin can modify or extend Topwrap's capabilities. All methods are passed the current `BuildContext` which contains the *internal representation* of the design and other metadata.

class BasePlugin

Bases: `object`

priority : int = `0`

Priority of the plugin. Hooks are invoked in order of decreasing priority.

pre_ir_generation(ctx*:* BuildContext)

Hook invoked before parsing sources.

post_ir_generation(ctx*:* BuildContext)

Hook invoked after parsing sources.

pre_transform(ctx*:* BuildContext)

Hook invoked before applying transformations.

post_transform(ctx: BuildContext)

Hook invoked after applying transformations.

pre_validate(ctx: BuildContext)

Hook invoked before performing validation.

post_validate(ctx: BuildContext)

Hook invoked after performing validation.

pre_output_generation(ctx: BuildContext)

Hook invoked before generating outputs.

post_output_generation(ctx: BuildContext)

Hook invoked after generating outputs (but before writing outputs to files).

pre_output_writing(ctx: BuildContext, target_dir: *OutputDir*)

Hook invoked before writing outputs to files.

post_output_writing(ctx: BuildContext, target_dir: *OutputDir*)

Hook invoked after writing outputs to files.

33.3 Build Context

The `BuildContext` exposes runtime metadata including the internal representation of the design. It is **shared** between Topwrap and any loaded plugins. Use the documentation on the *internal representation* for insight into what the metadata represents.

class BuildContext(*repo_modules: list[topwrap.model.module.Module], repo_interfaces: list[topwrap.model.interface.InterfaceDefinition], design_source: Union[topwrap.plugin.base.FileSource, topwrap.plugin.base.StringSource, NoneType], extra_sources: list[typing.Union[topwrap.plugin.base.FileSource, topwrap.plugin.base.StringSource]], top_module: Optional[topwrap.model.module.Module], loaded_modules: list[topwrap.model.module.Module], existing_interfaces: set[topwrap.model.misc.Identifier], outputs: dict[str, typing.Any] = <factory>, positions: dict[topwrap.model.misc.Identifier, topwrap.backend.kpm.common.Positions] = <factory>*)

Bases: `object`

repo_modules : list[*Module*]

Modules loaded from repositories.

repo_interfaces : list[*InterfaceDefinition*]

Interface definitions loaded from repositories.

design_source : FileSource | StringSource | None

Path to the source file containing the design.

extra_sources : list[FileSource | StringSource]

Paths to additional source files to be parsed.

top_module : *Module* | None

Top-level module containing the design.

loaded_modules : list[*Module*]

All loaded modules.

existing_interfaces : set[*Identifier*]

List of identifiers of existing definitions in HDL code

outputs : dict[str, Any]

Map of generated outputs, keyed on the output stage name.

positions : dict[*Identifier*, Positions]

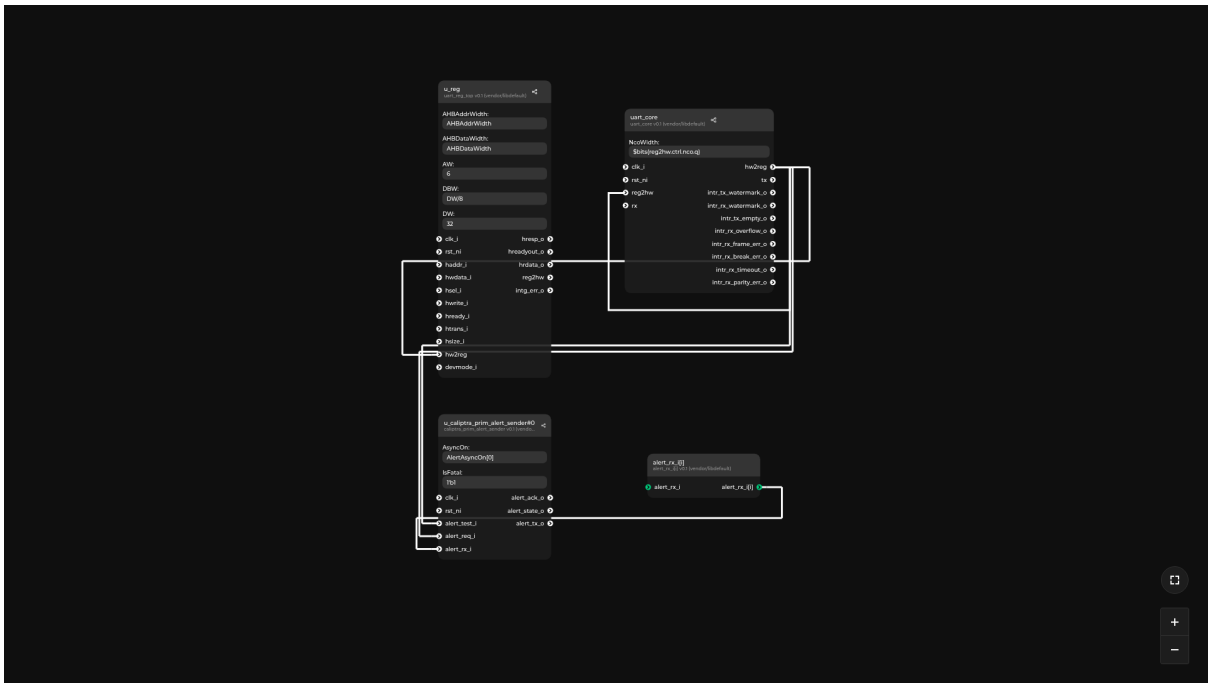
Map of module identifiers to the KPM position definitions.

property all_modules : Iterable[*Module*]

CHAPTER

THIRTYFOUR

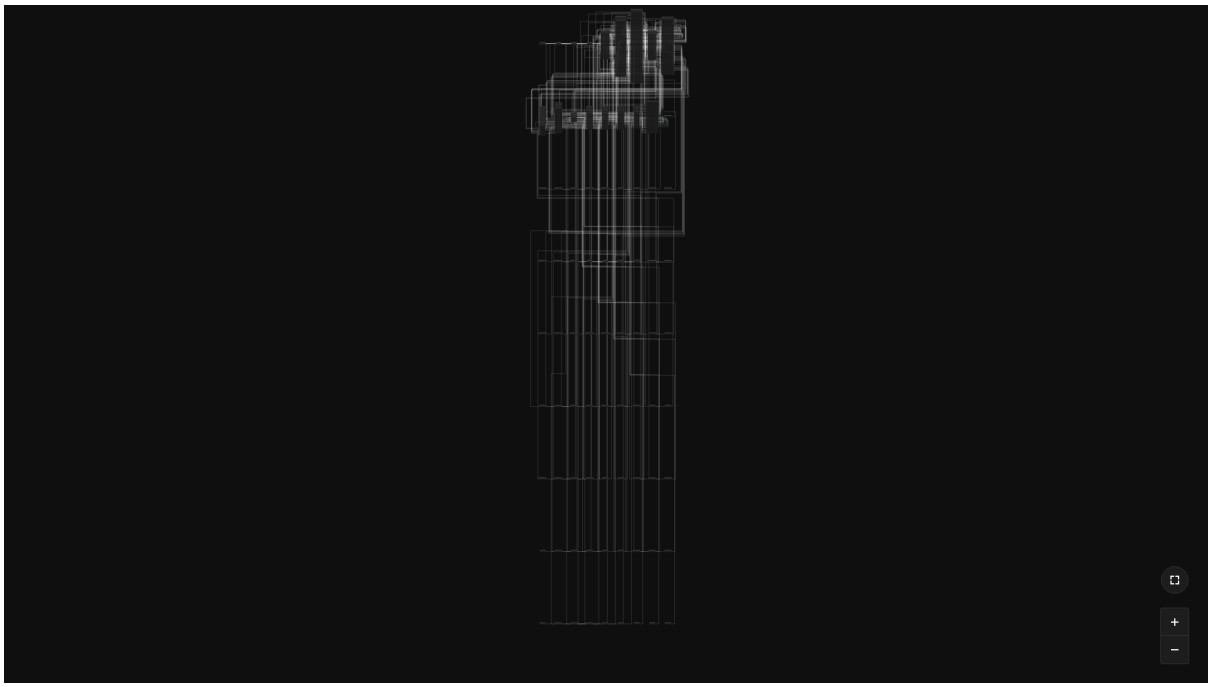
CALIPTRA UART



CHAPTER

THIRTYFIVE

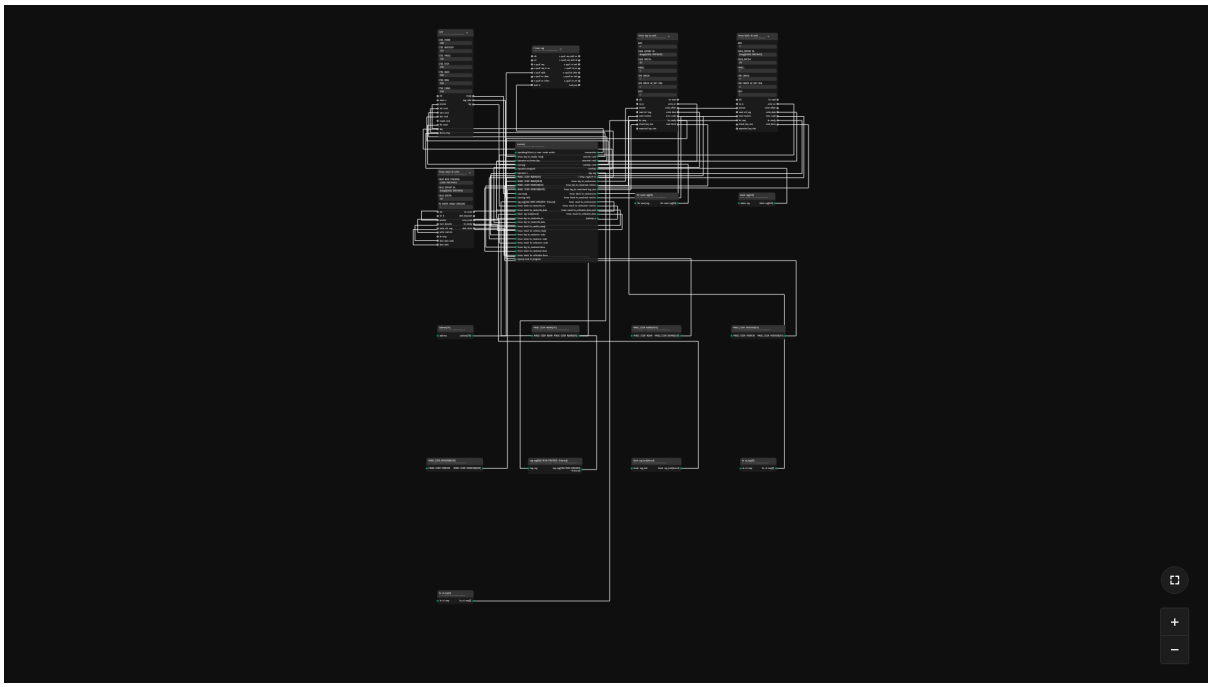
VEER EL2



CHAPTER

THIRTYSIX

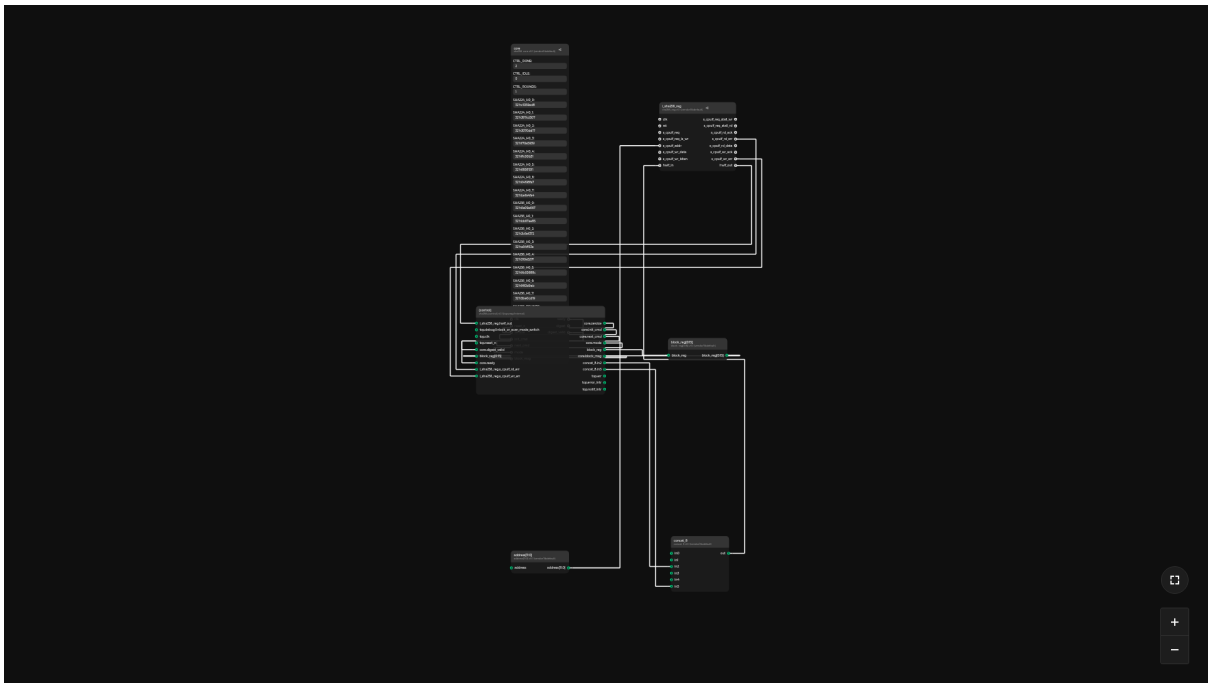
HMAC



CHAPTER

THIRTYSEVEN

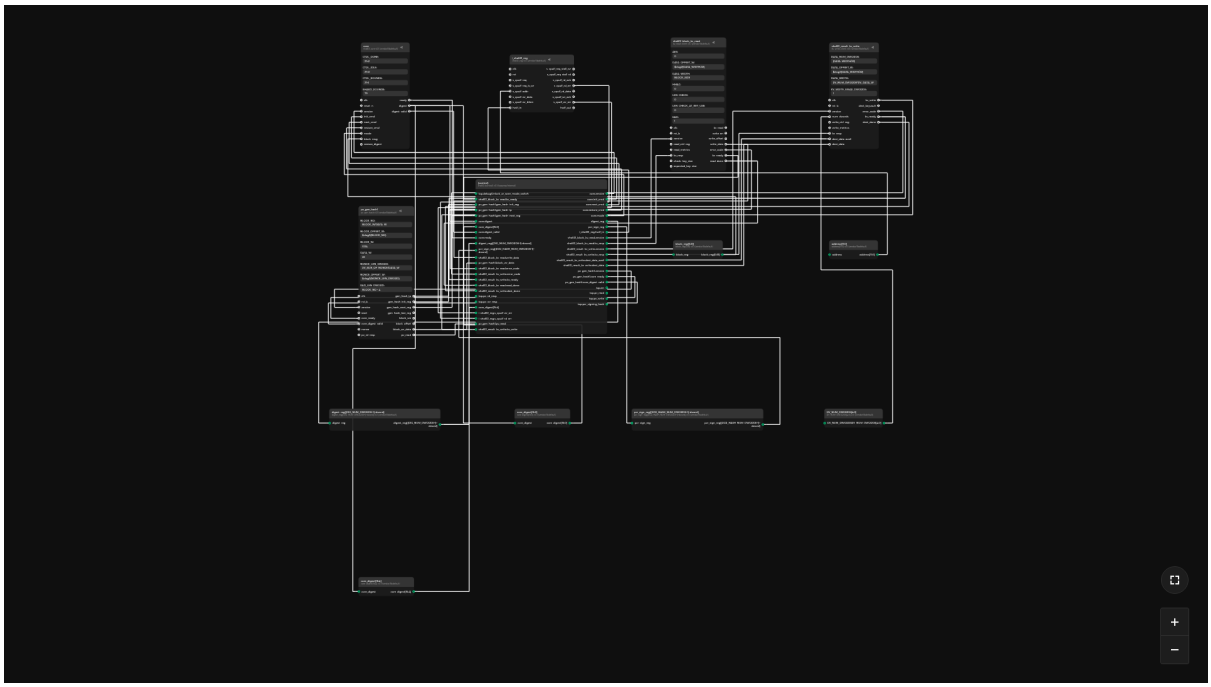
SHA256



CHAPTER

THIRTYEIGHT

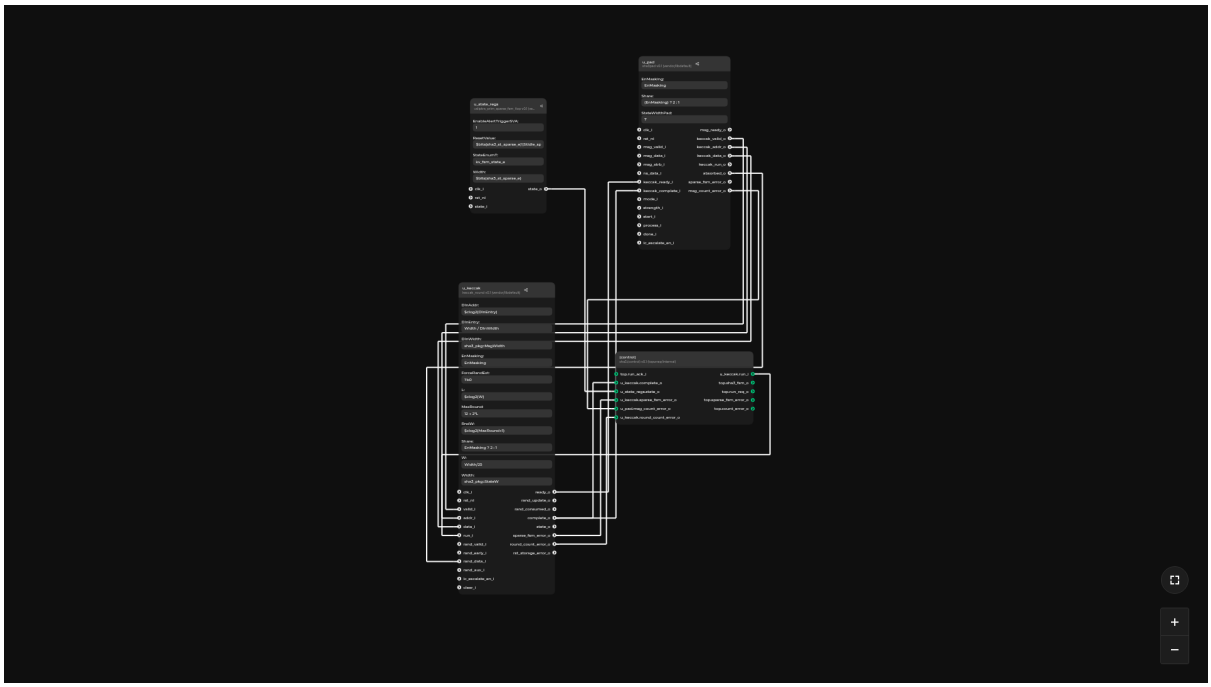
SHA512



CHAPTER

THIRTYNINE

SHA3



PYTHON MODULE INDEX

t

topwrap.backend.backend, 72
topwrap.frontend.frontend, 71
topwrap.interconnects.types.InterconnectTypeInfo, 143
topwrap.model.connections, 81
topwrap.model.design, 74
topwrap.model.hdl_types, 83
topwrap.model.inference.inference, 80
topwrap.model.inference.port, 79
topwrap.model.interconnect, 86
topwrap.model.interface, 76
topwrap.model.misc, 88
topwrap.model.module, 73
topwrap.plugin.base, 151

INDEX

A

ACTIVE_HIGH (*ResetPolarity attribute*), 82
ACTIVE_LOW (*ResetPolarity attribute*), 82
add_clock() (*Module method*), 73
add_clock_domain() (*Design method*), 76
add_component() (*Design method*), 76
add_config() (*Design method*), 76
add_connection() (*Design method*), 76
add_dependency() (*FuseSocBuilder method*), 96
add_extensions() (*Design method*), 76
add_extensions() (*Module method*), 74
add_external_ip() (*FuseSocBuilder method*), 96
add_files() (*Repo method*), 145
add_fileset() (*FuseSocBuilder method*), 96
add_interconnect() (*Design method*), 76
add_interface() (*Module method*), 73
add_memory_maps() (*Design method*), 76
add_module_instance_to_design() (*AXIVerilogGenerator method*), 141
add_module_instance_to_design() (*Generator method*), 140
add_module_instance_to_design() (*WishboneRRSystemVerilogGenerator method*), 141
add_parameter() (*InterfaceDefinition method*), 77
add_parameter() (*Module method*), 73
add_port() (*Module method*), 73
add_reference() (*Module method*), 73
add_reset() (*Module method*), 73
add_reset_domain() (*Design method*), 76
add_resource() (*Repo method*), 145
add_resource() (*ResourceHandler method*), 144
add_script() (*FuseSocBuilder method*), 96
add_signal() (*InterfaceDefinition method*), 77
add_source() (*FuseSocBuilder method*), 96
add_sources_dir() (*FuseSocBuilder method*), 97
add_target() (*FuseSocBuilder method*), 97
address (*InterconnectSubordinateParams attribute*), 87
all_modules (*BuildContext property*), 152
AXIVerilogGenerator (*class in topwrap.backend.sv.generators*), 141

B

Backend (*class in topwrap.backend.backend*), 72
BackendOutputInfo (*class in topwrap.backend.backend*), 72
BackendParseException, 72
BasePlugin (*class in topwrap.plugin.base*), 150
Bit (*class in topwrap.model.hdl_types*), 84
Bits (*class in topwrap.model.hdl_types*), 84
BitStruct (*class in topwrap.model.hdl_types*), 85
build() (*FuseSocBuilder method*), 97
BuildContext (*class in topwrap.plugin.base*), 151

C

check_connection_to_subgraph_metanodes() (*DataflowValidator method*), 102
check_duplicate_metanode_names() (*DataflowValidator method*), 102
check_duplicate_node_names() (*DataflowValidator method*), 102
check_external_in_to_external_out_connections() (*DataflowValidator method*), 102
check_external_io_exposed_ifaces() (*DataflowValidator method*), 103
check_intf_cdc() (*Design method*), 76
check_parameters_values() (*DataflowValidator method*), 103
check_port_to_multiple_external_metanodes() (*DataflowValidator method*), 103
check_unconnected_ports_interfaces() (*DataflowValidator method*), 103

check_unnamed_external_metanodes_with_multiple_conn() (*DataflowValidator method*), 103
CheckResult (*class in topwrap.kpm_dataflow_validator*), 103
Clock (*class in topwrap.model.connections*), 82
clock (*Clock attribute*), 82
clock (*ClockDomain attribute*), 75
clock (*Interconnect attribute*), 87
clock (*Interface attribute*), 78
clock_domains (*Design property*), 75
ClockDomain (*class in topwrap.model.design*), 74
clocks (*Module property*), 73
clocks (*ModuleInstance attribute*), 74
column (*FileReference attribute*), 90
combined() (*Identifier method*), 90
components (*Design property*), 75
Config (*class in topwrap.config*), 101
config (*Design attribute*), 75
ConfigManager (*class in topwrap.config*), 101
connections (*Design property*), 75
connections_with() (*Design method*), 76
ConstantConnection (*class in topwrap.model.connections*), 82
content (*BackendOutputInfo attribute*), 72
content (*FrontendParseStrInput attribute*), 71
copy() (*Bit method*), 84
copy() (*BitStruct method*), 85
copy() (*Enum method*), 84
copy() (*Logic method*), 83
copy() (*LogicArray method*), 84
copy() (*StructField method*), 84

D

data (*ExtensionData attribute*), 90
dataflow_complex_hierarchy() (*in module tests_kpm.test_kpm_validation*), 110
dataflow_conn_subgraph_metanode() (*in module tests_kpm.test_kpm_validation*), 110
dataflow_duplicate_external_input_interfaces() (*in module tests_kpm.test_kpm_validation*), 107
dataflow_duplicate_ip_names() (*in module tests_kpm.test_kpm_validation*), 104
dataflow_duplicate_metanode_names() (*in module tests_kpm.test_kpm_validation*), 107
dataflow_ext_in_to_ext_out_connections() (*in module tests_kpm.test_kpm_validation*), 106
dataflow_hier_duplicate_names() (*in module tests_kpm.test_kpm_validation*), 111
dataflow_inouts_connections() (*in module tests_kpm.test_kpm_validation*), 108
dataflow_invalid_parameters_values() (*in module tests_kpm.test_kpm_validation*), 105
dataflow_ports_multiple_external_metanodes() (*in module tests_kpm.test_kpm_validation*), 106
dataflow_subgraph_multiple_external_metanodes() (*in module tests_kpm.test_kpm_validation*), 109
dataflow_unconn_hierarchy() (*in module tests_kpm.test_kpm_validation*), 109
dataflow_unnamed_metanodes() (*in module tests_kpm.test_kpm_validation*), 108
DataflowValidator (*class in topwrap.kpm_dataflow_validator*), 102
default (*InterfaceSignal attribute*), 77
default_value (*Parameter attribute*), 90
default_value (*Port attribute*), 81
definition (*Interface attribute*), 78
Design (*class in topwrap.model.design*), 75
design (*Module property*), 73
design_source (*BuildContext attribute*), 151
DesignDomainException, 75
Dimensions (*class in topwrap.model.hdl_types*), 83
dimensions (*LogicArray attribute*), 84
direction (*InterfaceSignalConfiguration attribute*), 76
direction (*Port attribute*), 81

E

ElaboratableValue (*class in topwrap.model.misc*), 89
ElaboratableValue.DataclassRepr (*class in topwrap.model.misc*), 89
elaborate() (*ElaboratableValue method*), 89
ElabvalueEvaluator (*class in topwrap.model.hdl_types*), 86
Enum (*class in topwrap.model.hdl_types*), 84

existing_interfaces (*Backend attribute*), 72
existing_interfaces (*BuildContext attribute*), 152
ExtensionData (*class in topwrap.model.misc*), 90
extensions (*Design property*), 76
extensions (*Module property*), 73
external() (*ReferencedPort class method*), 81
extra_sources (*BuildContext attribute*), 151

F

Field (*ElaboratableValue attribute*), 89
field (*LogicFieldSelect attribute*), 85
FIELD (*PortSelectorOp attribute*), 79
field_name (*StructField attribute*), 85
fields (*BitStruct attribute*), 85
file (*FileReference attribute*), 90
file_association (*FrontendMetadata attribute*), 71
FileHandler (*class in topwrap.repo.resource*), 146
filename (*BackendOutputInfo attribute*), 72
FileReference (*class in topwrap.model.misc*), 90
find_by() (*QuerableView method*), 88
find_by_id_or_error() (*QuerableView method*), 88
find_by_name() (*QuerableView method*), 88
find_by_name_or_error() (*QuerableView method*), 88
flatten() (*Interface method*), 79
from_referenced_port() (*PortSelector class method*), 80
from_str() (*PortSelector class method*), 80
Frontend (*class in topwrap.frontend.frontend*), 71
FrontendMetadata (*class in topwrap.frontend.frontend*), 71
FrontendParseException, 71
FrontendParseOutput (*class in topwrap.frontend.frontend*), 71
FrontendParseStrInput (*class in topwrap.frontend.frontend*), 71
funcs (*ElabvalueEvaluator attribute*), 86
FuseSocBuilder (*class in topwrap.fuse_helper*), 96

G

generate() (*AXIVerilogGenerator method*), 142
generate() (*Generator method*), 140
generate() (*WishboneRRSystemVerilogGenerator method*), 141
Generator (*class in topwrap.backend.generator*), 140
get_elab_value_deps() (*ElabvalueEvaluator static method*), 86
get_name() (*AXIVerilogGenerator method*), 142
get_name() (*Generator method*), 140
get_name() (*WishboneRRSystemVerilogGenerator method*), 141
get_resource() (*Repo method*), 145
get_resources() (*Repo method*), 145

H

has_independent_signals (*Interface property*), 79
has_sliced_signals (*Interface property*), 79
has_symbolic_dimensions() (*in module topwrap.model.hdl_types*), 83
hierarchy() (*Module method*), 74

I

id (*InterfaceDefinition attribute*), 77
id (*Module attribute*), 73
Identifier (*class in topwrap.model.misc*), 89
IN (*PortDirection attribute*), 81
independent_signals (*Interface property*), 78
infer_interfaces_from_module() (*in module topwrap.model.inference.inference*), 80
infered_from_domain (*PortConnection attribute*), 82
INOUT (*PortDirection attribute*), 81
Interconnect (*class in topwrap.model.interconnect*), 87
INTERCONNECT_TYPES (*in module topwrap.interconnects.types*), 143
InterconnectManagerParams (*class in topwrap.model.interconnect*), 86
InterconnectParams (*class in topwrap.model.interconnect*), 86
interconnects (*Design property*), 75
InterconnectSubordinateParams (*class in topwrap.model.interconnect*), 86
Interface (*class in topwrap.model.interface*), 77
InterfaceConnection (*class in topwrap.model.connections*), 82
InterfaceDefinition (*class in topwrap.model.interface*), 77
InterfaceInferenceOptions (*class in topwrap.model.inference.inference*), 80
InterfaceMode (*class in topwrap.model.interface*), 76

interfaces (*FrontendParseOutput attribute*), 71
interfaces (*Module property*), 73
InterfaceSignal (*class in topwrap.model.interface*), 77
InterfaceSignalConfiguration (*class in topwrap.model.interface*), 76
invert (*PortConnection attribute*), 82
ios (*Module property*), 73
item (*LogicArray attribute*), 84

J

just_eval() (*ElabvalueEvaluator static method*), 86

L

library (*Identifier attribute*), 89
line (*FileReference attribute*), 90
load() (*Repo method*), 146
load() (*ResourceHandler method*), 144
loaded_modules (*BuildContext attribute*), 152
Logic (*class in topwrap.model.hdl_types*), 83
logic (*LogicSelect attribute*), 85
LogicArray (*class in topwrap.model.hdl_types*), 84
LogicBitSelect (*class in topwrap.model.hdl_types*), 85
LogicFieldSelect (*class in topwrap.model.hdl_types*), 85
LogicSelect (*class in topwrap.model.hdl_types*), 85
lower (*Dimensions attribute*), 83
lower_domains() (*Design method*), 76

M

make_referenced_port() (*PortSelector method*), 80
MANAGER (*InterfaceMode attribute*), 76
managers (*Interconnect attribute*), 87
memory_map (*Interconnect attribute*), 87
memory_maps (*Design attribute*), 75
metadata (*Frontend property*), 72
min_group_size (*InterfaceInferenceOptions attribute*), 80
mode (*Interface attribute*), 78
ModelBase (*class in topwrap.model.misc*), 88
modes (*InterfaceSignal attribute*), 77
module
    topwrap.backend.backend, 72
    topwrap.frontend.frontend, 71
    topwrap.interconnects.types.InterconnectTypeInfo, 143
    topwrap.model.connections, 81
    topwrap.model.design, 74
    topwrap.model.hdl_types, 83
    topwrap.model.inference.inference, 80
    topwrap.model.inference.port, 79
    topwrap.model.interconnect, 86
    topwrap.model.interface, 76
    topwrap.model.misc, 88
    topwrap.model.module, 73
    topwrap.plugin.base, 150, 151
Module (*class in topwrap.model.module*), 73
module (*ModuleInstance attribute*), 74
ModuleInstance (*class in topwrap.model.design*), 74
modules (*FrontendParseOutput attribute*), 71

N

name (*Clock attribute*), 82
name (*ClockDomain attribute*), 74
name (*ExtensionData attribute*), 90
name (*FrontendMetadata attribute*), 71
name (*FrontendParseStrInput attribute*), 71
name (*Identifier attribute*), 89
name (*Interconnect attribute*), 87
name (*Interface attribute*), 78
name (*InterfaceSignal attribute*), 77
name (*Logic attribute*), 83
name (*ModuleInstance attribute*), 74
name (*Parameter attribute*), 90
name (*Port attribute*), 81
name (*Repo attribute*), 146
name (*Reset attribute*), 83
name (*ResetDomain attribute*), 75
name (*Resource attribute*), 144
non_intf_ports() (*Module method*), 74
NotElaboratedException, 88

O

ObjectId (*class in topwrap.model.misc*), 88
ops (*LogicSelect attribute*), 85
ops (*PortSelector attribute*), 80
OUT (*PortDirection attribute*), 81
outputs (*BuildContext attribute*), 152
overlaps() (*LogicSelect method*), 85
overlaps() (*ReferencedPort method*), 81

P

Parameter (*class in topwrap.model.misc*), 90
parameters (*Interface attribute*), 78
parameters (*InterfaceDefinition property*), 77
parameters (*Module property*), 73
parameters (*ModuleInstance attribute*), 74

params (*Interconnect attribute*), 87
parent (*Clock attribute*), 82
parent (*ClockDomain attribute*), 74
parent (*Design attribute*), 75
parent (*ExtensionData attribute*), 90
parent (*Interconnect attribute*), 87
parent (*Interface attribute*), 78
parent (*InterfaceSignal attribute*), 77
parent (*Logic attribute*), 83
parent (*ModuleInstance attribute*), 74
parent (*Parameter attribute*), 90
parent (*Port attribute*), 81
parent (*Reset attribute*), 83
parent (*ResetDomain attribute*), 75
parse() (*FileHandler method*), 146
parse_files() (*Frontend method*), 72
parse_grouping_hints() (*in module topwrap.model.inference.inference*), 80
parse_str() (*Frontend method*), 71
parse_vlnv() (*Identifier static method*), 89
polarity (*Reset attribute*), 83
polarity (*ResetDomain attribute*), 75
Port (*class in topwrap.model.connections*), 81
port (*PortSelector attribute*), 80
PortConnection (*class in topwrap.model.connections*), 82
PortDirection (*class in topwrap.model.connections*), 81
ports (*Module property*), 73
PortSelector (*class in topwrap.model.inference.port*), 79
PortSelectorField (*class in topwrap.model.inference.port*), 79
PortSelectorOp (*class in topwrap.model.inference.port*), 79
positions (*BuildContext attribute*), 152
positions (*FrontendParseOutput attribute*), 71
post_ir_generation() (*BasePlugin method*), 150
post_output_generation() (*BasePlugin method*), 151
post_output_writing() (*BasePlugin method*), 151
post_transform() (*BasePlugin method*), 150
post_validate() (*BasePlugin method*), 151
pre_ir_generation() (*BasePlugin method*), 150
pre_output_generation() (*BasePlugin method*), 151
pre_output_writing() (*BasePlugin method*), 151
pre_transform() (*BasePlugin method*), 150
pre_validate() (*BasePlugin method*), 151
prefix_consider_camel_case (*InterfaceInferenceOptions attribute*), 80
prefix_split_tokens (*InterfaceInferenceOptions attribute*), 80
priority (*BasePlugin attribute*), 150

Q

QuerableView (*class in topwrap.model.misc*), 88

R

raw (*ElaboratableValue attribute*), 89
ReferencedInterface (*class in topwrap.model.connections*), 82
ReferencedPort (*class in topwrap.model.connections*), 81
refers_to() (*Identifier method*), 89
refs (*Module property*), 73
regexp (*InterfaceSignal attribute*), 77
RelationshipError, 88
remove_resource() (*Repo method*), 146
remove_resource() (*ResourceHandler method*), 144
Repo (*class in topwrap.repo.repo*), 144
repo_interfaces (*BuildContext attribute*), 151
repo_modules (*BuildContext attribute*), 151
represent() (*Backend method*), 72
required (*InterfaceSignalConfiguration attribute*), 77
Reset (*class in topwrap.model.connections*), 82
reset (*Interconnect attribute*), 87
reset (*Interface attribute*), 78
reset (*Reset attribute*), 83
reset (*ResetDomain attribute*), 75
reset_domains (*Design property*), 76
ResetDomain (*class in topwrap.model.design*), 75
ResetPolarity (*class in topwrap.model.connections*), 82
resets (*Module property*), 73
resets (*ModuleInstance attribute*), 74
resolve() (*ObjectId method*), 88
resolve_intf_values() (*ElabvalueEvaluator static method*), 86
Resource (*class in topwrap.repo.resource*), 144
resource_type (*ResourceHandler attribute*), 144
ResourceHandler (*class in topwrap.repo.resource*), 144

reverse() (*InterfaceSignalConfiguration method*), 77
reverse() (*PortDirection method*), 81

S

save() (*BackendOutputInfo method*), 72
save() (*Repo method*), 146
save() (*ResourceHandler method*), 144
Schema (*Config attribute*), 101
Schema (*InterconnectManagerParams attribute*), 86
Schema (*InterconnectParams attribute*), 86
Schema (*InterconnectSubordinateParams attribute*), 87
select (*ReferencedPort attribute*), 81
serialize() (*Backend method*), 73
set_parent() (*in module topwrap.model.misc*), 88
signals (*Interface attribute*), 78
signals (*InterfaceDefinition property*), 77
single() (*Dimensions class method*), 83
size (*Bit property*), 84
size (*BitStruct property*), 85
size (*InterconnectSubordinateParams attribute*), 87
size (*Interface attribute*), 78
size (*Logic property*), 83
size (*LogicArray property*), 84
size (*StructField property*), 84
slice (*LogicBitSelect attribute*), 85
SLICE (*PortSelectorOp attribute*), 79
sliced_signals (*Interface property*), 79
StructField (*class in topwrap.model.hdl_types*), 84
SUBORDINATE (*InterfaceMode attribute*), 76
subordinates (*Interconnect attribute*), 87
substitute_intf_params_for_module() (*ElabvalueEvaluator static method*), 86
sv_sysfunc_pfx (*ElabvalueEvaluator attribute*), 86
synchronous_to (*Reset attribute*), 83
synchronous_to (*ResetDomain attribute*), 75
SystemVerilogGenerator (*class in topwrap.backend.sv.generators*), 141

T

top_module (*BuildContext attribute*), 151
topwrap.backend.backend
    module, 72
topwrap.frontend.frontend
    module, 71
topwrap.interconnects.types.InterconnectTypeInfo
    module, 143
topwrap.model.connections
    module, 81
topwrap.model.design
    module, 74
topwrap.model.hdl_types
    module, 83
topwrap.model.inference.inference
    module, 80
topwrap.model.inference.port
    module, 79
topwrap.model.interconnect
    module, 86
topwrap.model.interface
    module, 76
topwrap.model.misc
    module, 88
topwrap.model.module
    module, 73
topwrap.plugin.base
    module, 150, 151
TranslationError, 88
type (*InterfaceSignal attribute*), 77
type (*Port attribute*), 81
type (*StructField attribute*), 85

U

UNSPECIFIED (*InterfaceMode attribute*), 76
update_interconnects_from_memory_maps() (*Design method*), 76
upper (*Dimensions attribute*), 83

V

validate_kpm_design() (*DataflowValidator method*), 103
value (*ElaboratableValue attribute*), 89
VariableName (*in module topwrap.model.misc*), 88
variants (*Enum attribute*), 84
vendor (*Identifier attribute*), 89
verilog_generators_map (*in module topwrap.backend.sv.generators*), 142
version (*Identifier attribute*), 89

W

WishboneRRSystemVerilogGenerator (*class in topwrap.backend.sv.generators*), 141