Antmicro

Topwrap

2025-10-30

DOCUMENTATION

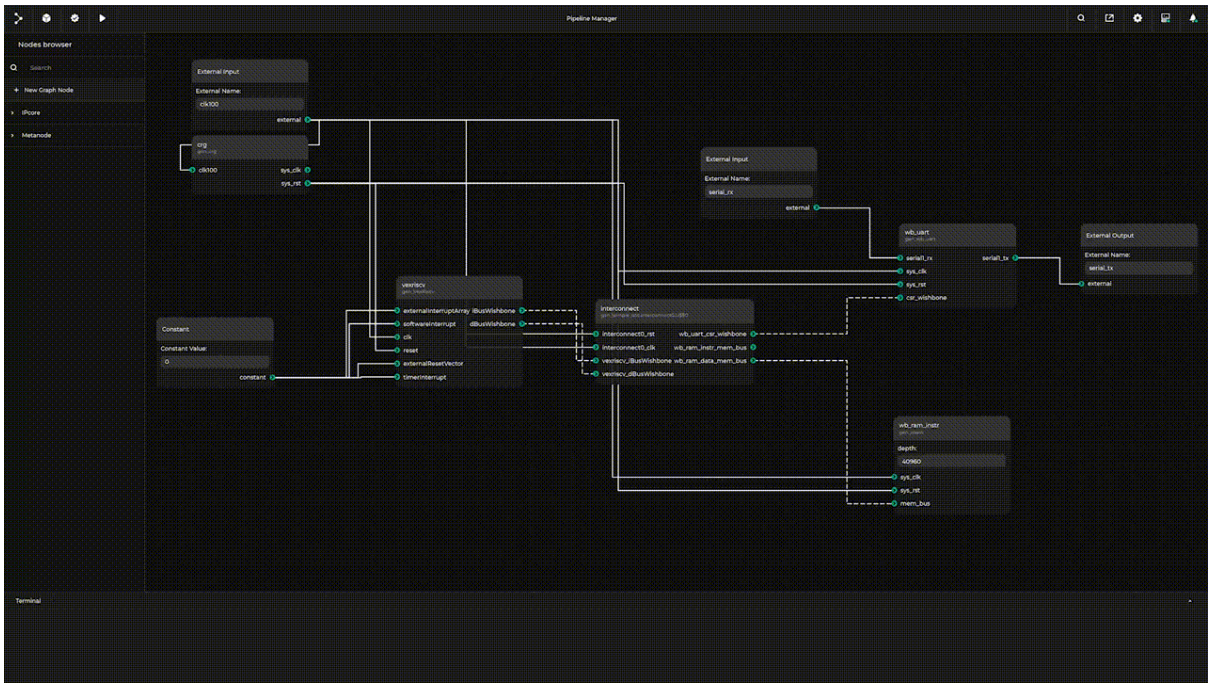
CHAPTER

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. 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:

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

2.2 2. Install the Topwrap user package

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

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

If you can't use pipx, 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.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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 Adding Verilog sources to repository

The `verilogs` directory contains two Verilog files, `simple_core_1.v` and `simple_core_2.v`.

In order to use external IP cores in a Topwrap design, they need to be added to a *repository* first.

The repository could either be created manually, or in an easier way, through CLI commands:

```
# To initialize an empty repository `my_repo_name` in `my_repo` directory
topwrap repo init my_repo_name my_repo
```

(continues on next page)

(continued from previous page)

```
# To automatically parse all supported HDL sources from the `verilogs` directory
# and copy them to the previously created `my_repo` repository
topwrap repo parse my_repo verilogs
```

The `topwrap repo init` command will add the newly created repository to the project *configuration file* or create one if it's missing. Thanks to this, the repository will be automatically loaded by Topwrap in further commands.

Additionally, Topwrap supports adding interfaces to modules based on mapping files, and automatically generating mapping files when parsing sources. For more details, see the *Interface mapping and inference* page.

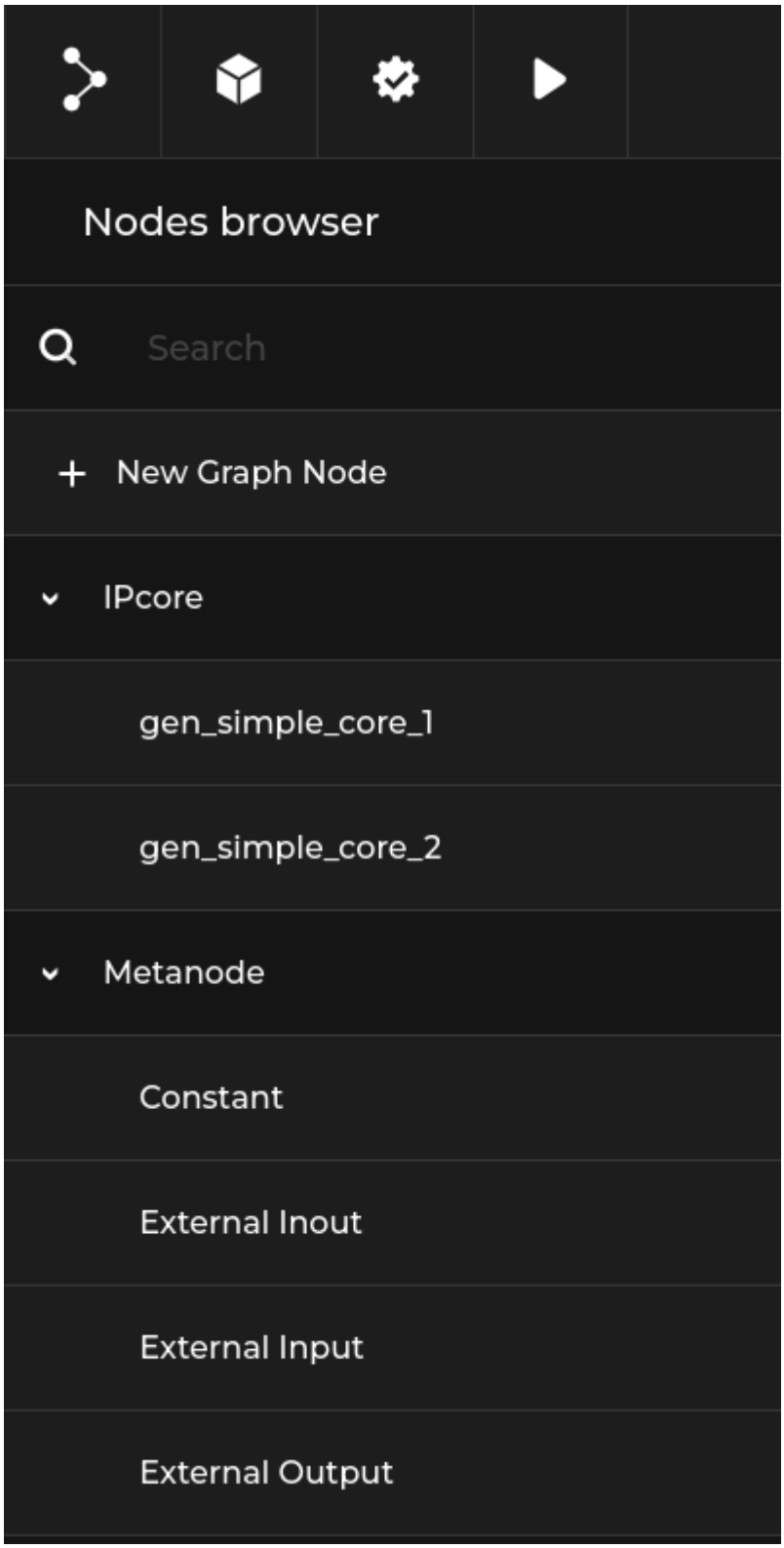
3.3 Building designs with Topwrap

3.3.1 Creating the design

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

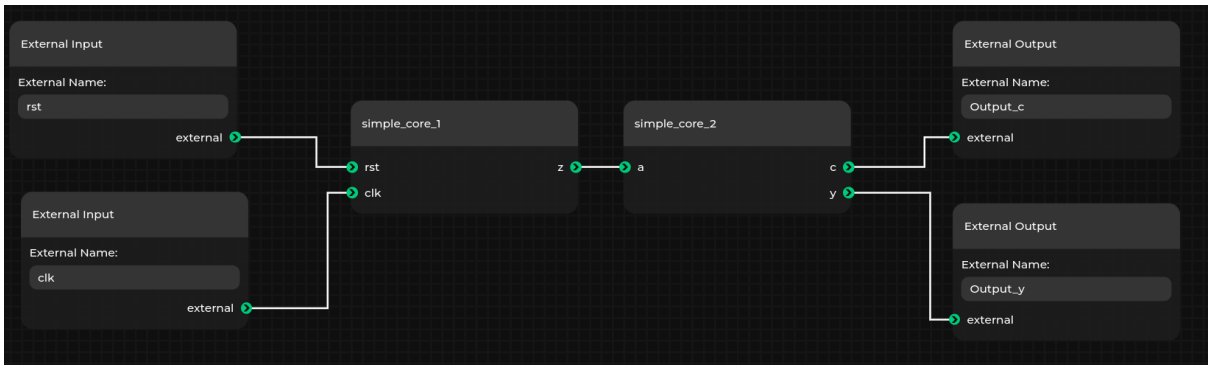
```
topwrap gui
```

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*.

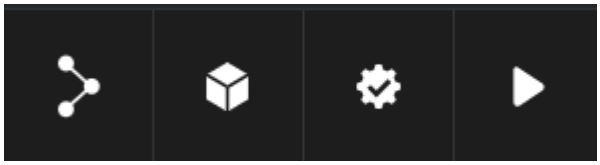


Note: You can change the name of an individual node by right clicking on it and selecting `rename`. This is useful when creating multiple instances of the same IP core.

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.

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: repo[my_repo]:simple_core_1
  simple_core_2:
    file: repo[my_repo]:simple_core_2
```

Here, `simple_core_1` and `simple_core_2` are given instance names for IP cores loaded from the `my_repo` repository.

Now we can start creating the design under the `design` section. The design doesn't have any parameters set, so we can skip this part and go straight into the `ports` section. In there, the connections between IP cores are defined. 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:

```
design:
  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.

```
design:
  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: repo[my_repo]:simple_core_1
  simple_core_2:
    file: repo[my_repo]:simple_core_2
design:
  ports:
    simple_core_1:
      clk: clk
      rst: rst
    simple_core_2:
      a: [simple_core_1, z]
```

(continues on next page)

(continued from previous page)

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

3.4.2 Generating Verilog top files

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

```
topwrap build --design {design_name.yaml}
```

Where `{design_name.yaml}` is the design saved at the end of the previous section. This will generate a `top.v` Verilog top wrapper in the specified build directory (`./build` by default).

3.4.3 Synthesis & FuseSoC

You can additionally generate a *FuseSoC core* file during `topwrap build` to automate further synthesis and implementation by simply adding the `-f` (`--fuse`) option.

CHAPTER

FOUR

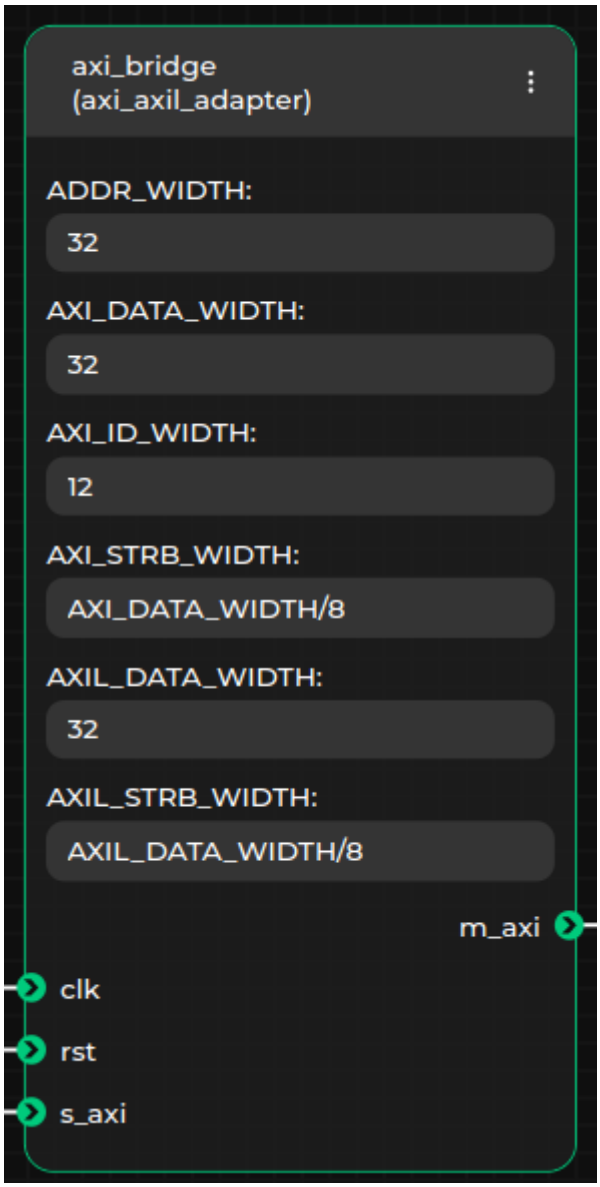
ADVANCED OPTIONS

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

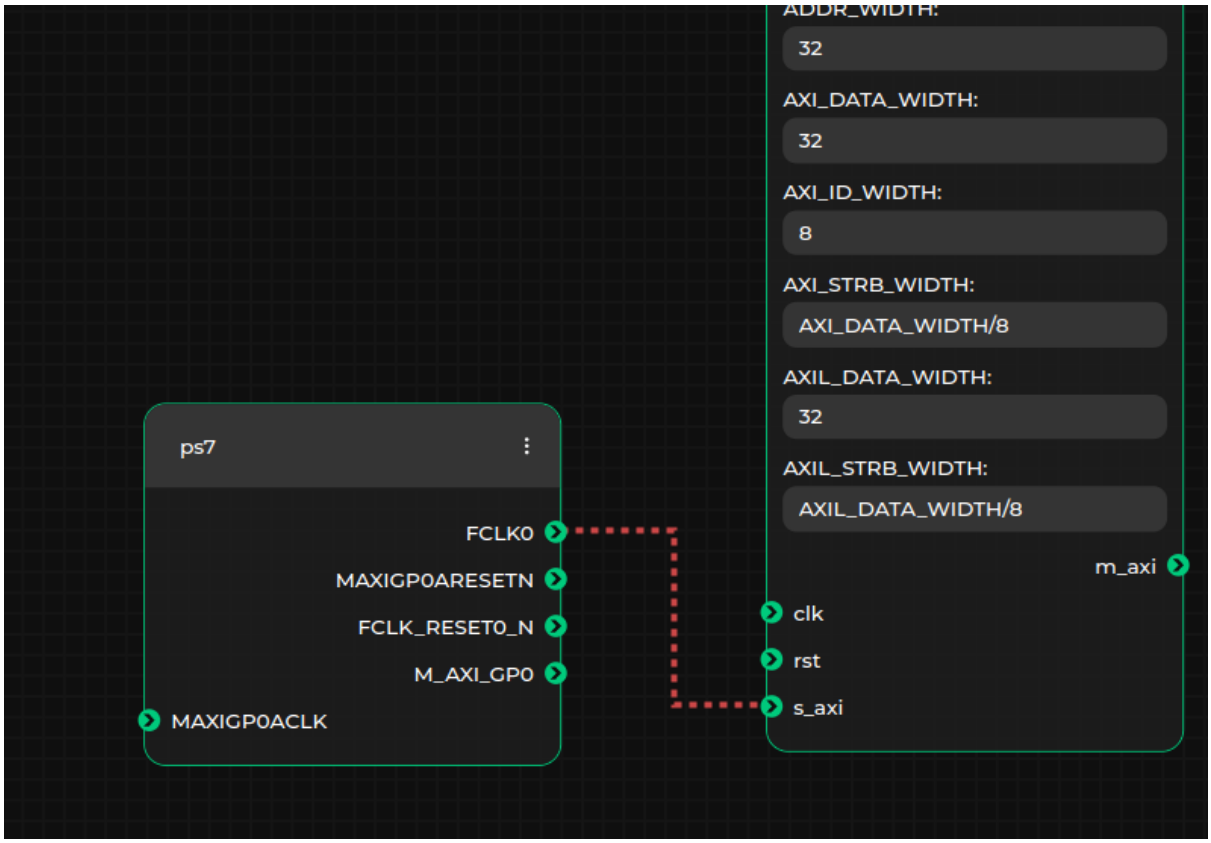
Upon successfully connecting to the server, Topwrap will generate and transmit a specification describing the structure of the selected IP cores. If the `-d` option is specified, the design will be displayed in the GUI. The following content is based on the `PWM` example located in `examples/pwm`. From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

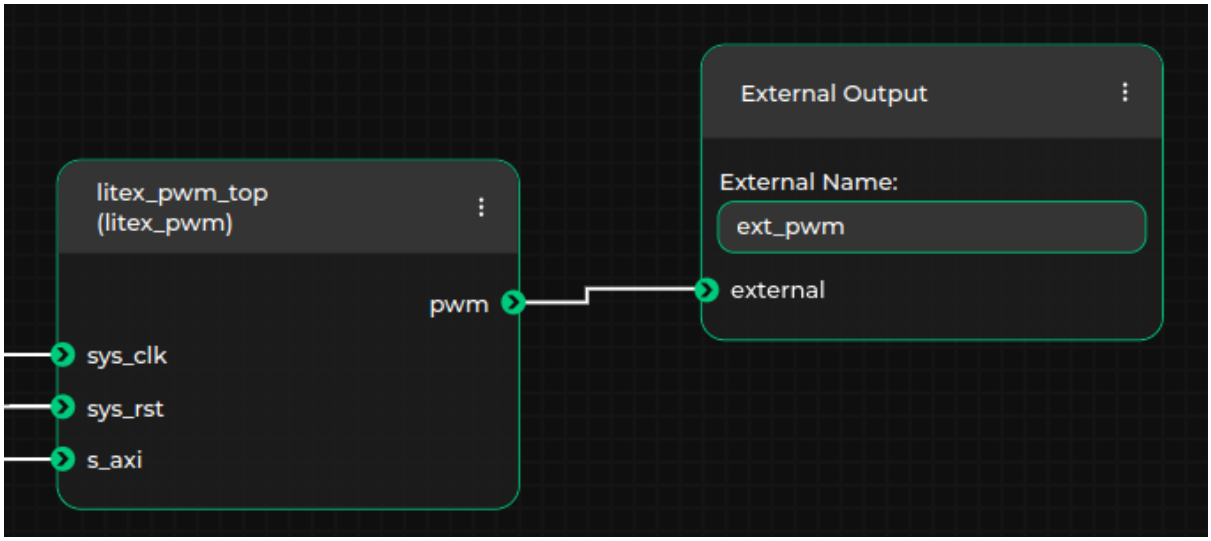


Parameter values can be specified as integers in various bases (e.g., `0x28`, `40`, or `0b101000`) or as arithmetic expressions, which will be evaluated later (e.g., `(AXI_DATA_WIDTH + 1) / 4` is a valid expression, provided a parameter named `AXI_DATA_WIDTH` exists in the same IP core). Additionally, 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:

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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
```

5.3 Inout

Link to source

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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
```

To generate the bitstream for Zynq, use:

```
make
```

To generate only the HDL sources use:

```
make generate
```

User repository

Link to source

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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
```

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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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
```

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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

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.

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 interconnects and IP cores
- setting parameter values and describing hierarchies for the project
- connecting the IPs and hierarchies
- picking external ports (those which will be connected to the physical I/O).

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

```
ips:
  # specify relations between IPs instance names in the
  # design yaml and IP cores description YAMLs
  {ip1_instance_name}:
    file: {resource_path} # see "Resource path syntax" section for more␣
↪information
  ...

design:
  name: {design_name} # optional name of the toplevel
  hierarchies:
      # see "Hierarchies" below for a detailed description of the format
      ...
  parameters: # specify IP parameter values to be overridden
    {ip_instance_name}:
      {parameters_name} : {parameters_value}
      ...
  ports:
    # specify the incoming ports connections of an IP named `ip1_name`
    {ip1_name}:
      {port1_name} : [{ip2_name}, {port2_name}]
      ...
```

(continues on next page)

(continued from previous page)

```
    # 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
    ...

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

`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 `design` 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. For example, the `design` section cannot contain two identical keys, but it is possible to have `ip_name` key in this section and `ip_name` in the `design` section of a separate hierarchy.

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}:
      ips: # ips that are used on this hierarchy level
        {ip_name}:
          ...

      design:
        parameters:
          ...
        ports: # ports connections internal to this hierarchy.
          # note that also you have to connect port to it's external port␣
↪equivalent (if exists).
          {ip1_name}:
              {port1_name} : [{ip2_name}, {port2_name}]
              {port2_name} : {port2_external_equivalent} # connection to external␣
↪port equivalent. Note that it has to be the parent port.
              ...
        hierarchies:
          {nested_hierarchy_name}:
            # structure here will be the same as for {hierarchy_name_1}
            ...
      external:
        # external ports and/or interfaces of this hierarchy; these can be
        # referenced in the upper-level `ports`, `interfaces` or `external`␣
↪section
        ports:
            in:
              - {port2_external_equivalent}
        ...
    {hierarchy_name_2}:
      ...
```

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

6.2 IP description files

The ports of an IP should be placed in the global `signals` key, followed by the direction - `in`, `out` or `inout`. The module name of an IP should be placed in the global `name` key, and it should be consistent with the definition in the HDL file.

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
```

The previous example can be used with any IP. However, in order to benefit from connecting entire interfaces simultaneously, the ports must belong to a named interface as in this example:

```
#file: axis_width_converter.yaml
name: axis_width_converter
interfaces:
    s_axis:
        type: AXIStream
        mode: subordinate
        signals:
            in:
                TDATA: [s_axis_tdata, 63, 0]
                TKEEP: [s_axis_tkeep, 7, 0]
                TVALID: s_axis_tvalid
                TLAST: s_axis_tlast
                TID: [s_axis_tid, 7, 0]
                TDEST: [s_axis_tdest, 7, 0]
                TUSER: s_axis_tuser
            out:
                TREADY: s_axis_tready
    m_axis:
        type: AXIStream
        mode: manager
        signals:
            in:
                TREADY: m_axis_tready
            out:
                TDATA: [m_axis_tdata, 31, 0]
                TKEEP: [m_axis_tkeep, 3, 0]
                TVALID: m_axis_tvalid
                TLAST: m_axis_tlast
                TID: [m_axis_tid, 7, 0]
                TDEST: [m_axis_tdest, 7, 0]
```

(continues on next page)

(continued from previous page)

```
                TUSER: m_axis_tuser
signals: # These ports do not belong to an interface
    in:
        - clk
        - rst
```

The names `s_axis` and `m_axis` will be used to group the selected ports. Each signal in an interface has a name which must match with the signal that it is connected to, for example `TDATA: port_name` must connect to `TDATA: other_port_name`.

6.2.1 Port widths

You can specify the port width in the following format:

```
signals:
    in:
        - [port_name, upper_limit, lower_limit]
```

- `port_name` - name of the port.
- `upper_limit` and `lower_limit` define the bit range, where `[upper_limit, lower_limit]` determines the number of bits for the port (e.g. `[63, 0]` for 64 bits).

As an example:

```
signals:
    in:
        - [gpio_io_i, 31, 0] # 32 bits
```

If the bit range is omitted, as in the example below, then the default width of `port_name` is 1 bit.

```
signals:
    in:
      - port_name
```

You can also specify the signal width within interfaces.

```
interfaces:
    s_axis:
        type: AXIStream
        mode: subordinate
        signals:
            in:
                TDATA: [s_axis_tdata, 63, 0] # 64 bits
                ...
                TVALID: s_axis_tvalid # defaults to 1 bit
```

- `TDATA` is assigned to `s_axis_tdata` and is 64 bits wide, defined by `[63, 0]`.
- `TVALID` is assigned to `s_axis_tvalid` and, without a specified range, defaults to `1 bit`.

6.2.2 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
        signals:
            in:
                TDATA: [s_axis_tdata, DATA_WIDTH-1, 0]
                TKEEP: [s_axis_tkeep, KEEP_WIDTH-1, 0]
                ...
                TID: [s_axis_tid, ID_WIDTH-1, 0]
                TDEST: [s_axis_tdest, DEST_WIDTH-1, 0]
                TUSER: [s_axis_tuser, USER_WIDTH-1, 0]
```

The parameter values can be integers or math expressions.

6.2.3 Port slicing

Ports can be sliced for using some parts of the port as a signal that belongs to a defined interface.

As an example: Port `m_axi_bid` of the IP core is 36 bits wide. Use bits `23..12` as the `BID` signal of the `m_axi_1` AXI manager

```
m_axi_1:
    type: AXI
    mode: manager
    signals:
        in:
            BID: [m_axi_bid, 35, 0, 23, 12]
```

6.3 Interface description files

Topwrap can use predefined interfaces, as illustrated in YAML files that come packaged with the tool. The currently supported interfaces are AXI3, AXI4, AXI Lite, AXI Stream and Wishbone.

An example file looks as follows:

```yaml
name: AXI4Stream
port_prefix: AXIS
signals:
    # The convention assumes the AXI Stream transmitter (manager) perspective
    required:
        out:
            TVALID: tvalid
            TDATA: tdata
            TLAST: tlast
        in:
            TREADY: tready
    optional:
        out:
            TID: tid
            TDEST: tdest
            TKEEP: tkeep
            TSTRB: tstrb
            TUSER: tuser
            TWAKEUP: twakeup
```

The `name` of an interface must be unique.

Signals are either required or optional, and their direction is described from the perspective of the manager (i.e. the direction of signals in the subordinate are flipped). Note that `clock` and `reset` are not included as these are usually inputs to both the manager and subordinate, so they are not supported in the interface specification. Every signal is a key-value pair, where the key is a generic signal name (normally taken from the interface specification) and used to identify it in other parts of Topwrap (i.e. IP core description files), and the value is a regex used to deduce which port defined in the HDL sources represents this signal.

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}
...
```

The syntax is as follows:

```
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
- `repo`
  - `SCHEME_ARGS`: Repository name
  - `SCHEME_PATH`: A name of resource
- `get`
  - `SCHEME_ARGS`: None
  - `SCHEME_PATH`: The URL address of the remote resource. Only `http(s)://` URLs are currently supported.

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

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

A path to the file on the filesystem.

```
repo[builtin]:axi_protocol_converter
```

This loads the `axi_protocol_converter` core located in the builtin user repository.

```
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.

CHAPTER

SEVEN

INTERFACE MAPPING AND INFERENCE

Warning: SystemVerilog modules with parameter types are not fully supported by mapping and inference. There is no way to override default parameter values in mapping files, 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.

7.1 Interface mapping

Topwrap supports adding additional interfaces to modules which didn't originally have them. To facilitate this, mapping files, which describe what interfaces to add to what modules are used. These mappings describe which module ports (or smaller parts thereof) correspond to what interface signals, and what the resulting interfaces should be named.

Mapping files are stored within the `mappings` subdirectory within repositories, and are applied automatically discovered and applied during `topwrap build`.

7.1.1 Mapping files

An example mapping file, adding an AXI4 interface to a module, looks like this:

```
modules:
  - id:
      name: some_ip_core
      vendor: some_vendor
      library: some_library
    interfaces:
      M_AXI:
        interface:
          name: AXI4
        mode: MANAGER
        signals:
          awaddr: M_AXI_awaddr
          awlen: M_AXI_awlen
          # ...
```

(continues on next page)

(continued from previous page)

```
        clock: MCLK
        reset: MRSTN
```

At the top level, there is a single key, `modules`, which contains a list of module mapping definitions.

Each module has two keys:

- `id`, which describes the identifier (name, vendor, and library), used to match this definition to a module. The vendor and library fields may be omitted and will use Topwrap's default values instead (`vendor` and `libdefault` respectively).
- `interfaces`, which is a list of interfaces to be added to this module.

Each interface is described by an object with the following keys:

- `interface`, which is the identifier of the interface definition, and behaves in the same way as the module's `id` key.
- `mode`, which describes the mode of the interface, and may be either `MANAGER` or `SUBORDINATE`.
- `signals`, which is a list of signal assignments, each in the following format: `signal-name: port-selector`. In addition to selecting module ports, the port selector also supports selecting fields within structs, and bit slicing, with a SystemVerilog-like syntax, e.g. `some_port.some_field[1:0]`.
- `clock`, which is a port selector for a clock signal for this interface. This key is optional, and the same clock signal might be shared between multiple interfaces within a module.
- `reset`, which is a port selector for a reset signal for this interface. This key is optional, and the same reset signal might be shared between multiple interfaces within a module.

7.2 Interface inference

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.

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`.

Finally, after the inference is done, a mapping is produced. This mapping is then saved into a mapping file in the destination repository, in the `mappings` subdirectory, for use with future `topwrap build` invocations.

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
```

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:

- `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`.

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
    ```

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

  For more information, refer to *User repositories*.

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

CONSTRUCTING, CONFIGURING AND LOADING REPOSITORIES

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.

9.1 Supported resource types

9.1.1 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.

Each core should be stored under its own subdirectory within the IP core resources directory and contain at least a description file named `.core.yaml`. Other than that, the end user is free to store necessary RTL sources for the IP core in an arbitrary layout, even outside the repository directory itself, since all of them have to be described in the `.core.yaml` file anyways.

`.core.yaml` description file reference

```
# The user-facing name for the IP core resource
# that is also used to identify it amongst other cores
name: antmicro_axilib_axireflector

# The name of the concrete top-level module that represents
# this core as it appears in the referenced sources.
top_level_name: axi_reflector

# Source files that have to be parsed by Topwrap in order
# to automatically extract information about the IP core.
# The definition of `top_level_name` module must be included
```

(continues on next page)

(continued from previous page)

```
# somewhere in the sources as well.
sources:
  # The format of paths in this array follows the "Resource path syntax"␣
↪specification:
  # https://antmicro.github.io/topwrap/description_files.html#resource-path-syntax
  - file:./srcs/axi_reflector.sv

  # Different types of sources are supported and recognised
  # automatically according to their extension
  - get:https://file.antmicro.com/abc123/axi_regs.yaml

  # (Recommended) The type of the resource can also be given explicitly
  - resource: file:./srcs/axi_pkg.svp
    # This identifies the frontend that will be used to parse this file
    type: systemverilog
```

9.1.2 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.

9.1.3 Interface port mappings

- Path: `repo_dir/mappings/`

This resource represents a custom mapping from a module's ports to an interface.

Mappings are stored in *interface mapping files* under the `mappings` subdirectory, with each file containing one or more mapping.

9.2 CLI

9.2.1 `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.

9.2.2 `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.

9.2.3 `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`

9.3 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:

```
nox -s package_cores
```

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

TEN

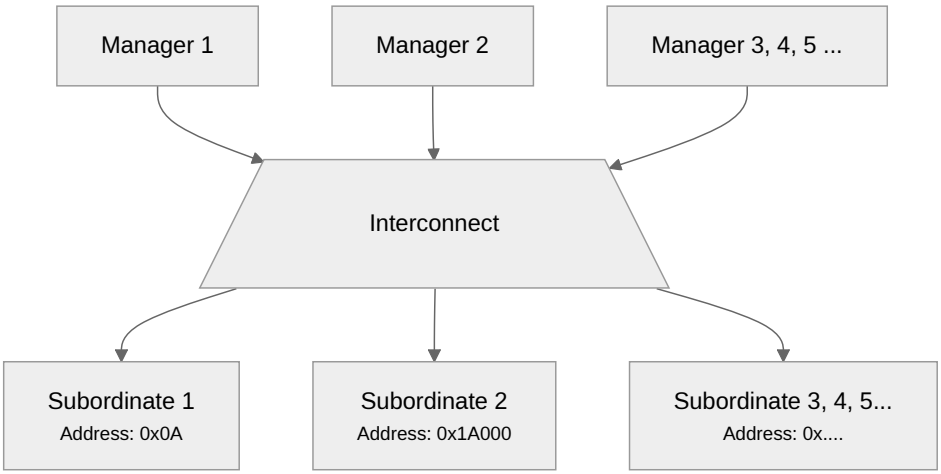
INTERCONNECT GENERATION

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:

10.1 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}
        ...
      ...
  ...
```

10.2 Supported interconnect types

10.2.1 `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

- `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

CHAPTER

ELEVEN

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 build` is used with the `--fuse` option, it generates a FuseSoC `.core` file along with the top-level wrapper.

11.1 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.

11.2 Additional build options

To enable `.core` file generation, supply the `--fuse/-f` flag to `Topwrap build`:

```
topwrap build -d design.yaml --fuse
```

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

```
topwrap build -d design.yaml -f --sources ./srcs_v -s ./srcs_vhd
```

If you're targeting a specific FPGA chip, you can additionally specify its number using the `--part/-p` 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 build -d design.yaml -f --part 'xc7z020clg400-3'
```

11.3 .core file template

A template for the .core file is bundled with Topwrap (templates/core.yaml.j2).

By default, topwrap.fuse_helper.FuseSocBuilder searches for the template file in working directory, meaning you must copy the template file into the project location. You may also need to edit the file to change the backend tool, add more targets, set additional Hooks and edit other parameters.

11.4 Synthesis

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

```
fusesoc --cores-root build 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

TWELVE

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 `nox` and `pre-commit`:

```
python -m venv venv
source venv/bin/activate
pip install -e ".[all]"
```

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

CHAPTER

THIRTEEN

CODE STYLE

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

13.1 Lint with nox

After successful setup, Nox sessions can be executed to perform lint checks:

```
nox -s lint
```

This runs ruff and codespell and fixes almost all formatting and linting problems automatically.

Note: To reuse the current virtual environment and avoid lengthy installation processes, use the -R flag:

```
nox -R -s lint
```

Note: pre-commit can also be run from nox:

```
nox -s pre_commit
```

13.2 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 `nox` with `ruff` and `codespell` sessions.

13.3 Tools

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

- `nox` is a tool, which simplifies management of Python testing.
- `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

FOURTEEN

TESTS

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

14.1 Test execution

The tests are located in the `tests` directory. All tests can be run with `nox` by specifying the `tests` session:

```
nox -s tests
```

This runs tests on the Python interpreter versions that are available locally. There is also a session `tests_in_env` that will automatically install all required Python versions, provided you have pyenv installed:

```
nox -s tests_in_env
```

Note: To reuse an existing virtual environment and avoid lengthy installation times, use the `-R` flag:

```
nox -R -s tests_in_env
```

To force a specific Python version and avoid running tests for all listed versions, use `-p VERSION` flag:

```
nox -p 3.10 -s tests_in_env
```

Tests can also be launched without `nox` 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 `nox` 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
```

14.2 Test coverage

Test coverage is automatically generated when running tests with `nox`. 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`

14.3 Updating kpm test data

All `kpm` data from examples can be generated using `nox`. 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:

```
nox -s update_test_data
```

To update only specifications run:

```
nox -s update_test_data -- specification
```

Valid options for `update_test_data` sessions, are:

- `specification`
- `dataflow`

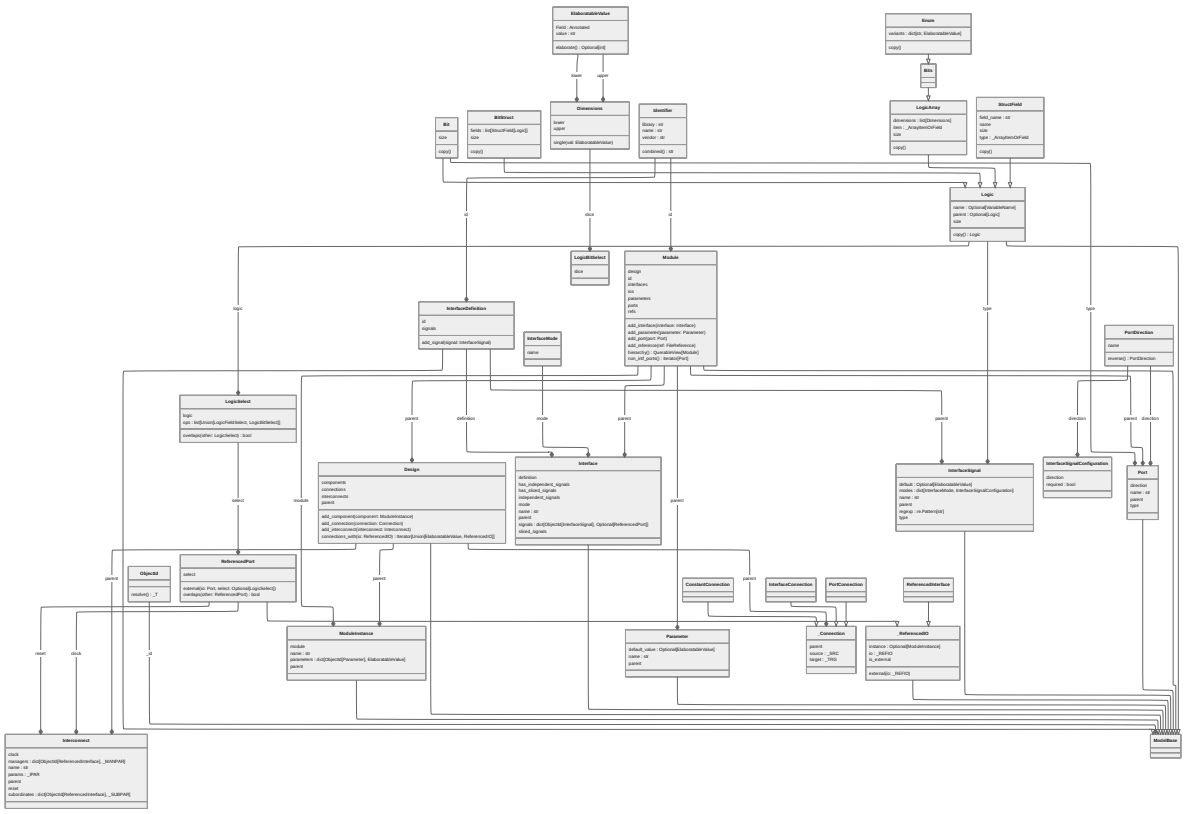
CHAPTER

FIFTEEN

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.

15.1 Class diagram



15.2 Frontend & Backend

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

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.

15.2.1 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"*})

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]*) → Iterator[*Module*]

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]*) → Iterator[*Module*]

Parse a collection of source files into IR modules

Parameters

sources: Iterable[Path]
Collection of paths to sources

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

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

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 `_T`.

abstract represent(module: Module, /) → _T

Convert the IR into an arbitrary custom external format.

abstract serialize(repr: *_T*, /) → 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.

15.3 Module

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

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.

add_port(port: Port)

add_parameter(parameter: Parameter)

add_interface(interface: Interface)

add_reference(ref: FileReference)

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.

15.4 Design

class ModuleInstance(*, name: *VariableName*, module: Module, parameters: *Mapping*[ObjectId[Parameter], ElaboratableValue] = {})

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.

class Design(*, components: *Iterable[*ModuleInstance*]* = (), interconnects: *Iterable[*Interconnect*]* = (), connections: *Iterable[*ConstantConnection | PortConnection | InterfaceConnection*]* = ())

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*

property components : *QuerableView*[*ModuleInstance*]

property interconnects : *QuerableView*[*Interconnect*]

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

add_component(component: ModuleInstance)

add_interconnect(interconnect: Interconnect)

add_connection(connection: ConstantConnection | PortConnection | InterfaceConnection)

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.

15.5 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

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*]* = ())

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

add_signal(signal: InterfaceSignal)

class Interface(name: *str*, mode: InterfaceMode, definition: InterfaceDefinition, signals: *dict[*ObjectId*[*InterfaceSignal*]*, ReferencedPort | *None]*)

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`.

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

15.6 Interface mapping and inference

15.6.1 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`.

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.

15.6.2 Mapping

class **InterfacePortGrouping**(interface: Identifier, mode: *str*, signals: *dict[str,* PortSelector*]*, clock: PortSelector | *None* = **None**, reset: PortSelector | *None* = **None**)

Bases: `MarshmallowDataclassExtensions`

A grouping of ports that are to be put into one interface.

interface : *Identifier*

Identifier of the interface that this grouping uses.

mode : str

The interface mode (manager, subordinate) for this grouping.

signals : dict[str, *PortSelector*]

A signal name to port selector mapping for this grouping.

clock : *PortSelector* | None = **None**

Clock signal for this interface. Might be shared with other interfaces in a module.

reset : *PortSelector* | None = **None**

Reset signal for this interface. Might be shared with other interfaces in a module.

Schema

alias of `InterfacePortGrouping`

class **InterfacePortMapping**(id: ~topwrap.model.misc.Identifier, interfaces: dict[str, ~topwrap.model.inference.mapping.InterfacePortGrouping] = <factory>)

Bases: `MarshmallowDataclassExtensions`

A mapping between module ports and interface signals.

id : *Identifier*

The identifier of the module this mapping applies to.

interfaces : dict[str, *InterfacePortGrouping*]

Named groupings of ports into interfaces. See `PortSelector` for a description of the format.

Schema

alias of `InterfacePortMapping`

class **InterfacePortMappingDefinition**(modules: *list[*InterfacePortMapping*]*)

Bases: `MarshmallowDataclassExtensions`

Top level port mapping definition. Can contain mappings for multiple modules.

modules : list[*InterfacePortMapping*]

Schema

alias of `InterfacePortMappingDefinition`

exception InterfaceMappingError

Bases: RuntimeError

Error raised for problems that occurred during interface mapping.

map_interfaces_to_module(mappings: *Sequence[*InterfacePortMapping*]*, intf_defs: *Sequence[*InterfaceDefinition*]*, module: Module)

Apply mappings to the given module. Potentially modifies the module by adding new interfaces, as described in the given mappings.

Parameters

mappings: Sequence[*InterfacePortMapping*]
The mappings to use.

intf_defs: Sequence[*InterfaceDefinition*]
Known interface definitions for mappings.

module: *Module*
Module to apply mappings to.

15.6.3 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, min_signal_count: int = 2, prefix_split_tokens: list[str] = <factory>, prefix_consider_camel_case: bool = True, prefix_length_score: int = 2, required_match_score: int = 15, optional_match_score: int = 10, required_missing_score: int = -100, optional_missing_score: int = -2, unmatched_port_penalty_leniency: float = 7.5, score_lower_limit: int = 0)

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.

min_signal_count : int = 2
Minimum number of signals an interface must have 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.

prefix_length_score : int = 2

Points awarded to each character of the prefix.

required_match_score : int = 15

Points awarded for matching all required signals (applied proportionally).

optional_match_score : int = 10

Points awarded for matching all optional signals (applied proportionally).

required_missing_score : int = -100

Points awarded for missing all required signals (applied proportionally).

optional_missing_score : int = -2

Points awarded for missing all optional signals (applied proportionally).

unmatched_port_penalty_leniency : float = 7.5

Leniency in penalty for unmatched ports in a group. Penalty is computed as `-(exp(n/leniency) - 1)`, such that matching all ports yields 0 points.

score_lower_limit : int = 0

Candidate interfaces with score below or equal to this limit will be ignored.

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

Perform interface inference. Yields a mapping that can be applied using `map_interfaces_to_module()`.

Parameters

module: *Module*
Module to perform inference on.

intf_defs: Iterable[*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.

15.7 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**)

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.)

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*)

Bases: _Connection[ReferencedPort, ReferencedPort]

Represents a connection between two ports of some components

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

Bases: _Connection[ReferencedInterface, ReferencedInterface]

Represents a connection between two interfaces of some components

15.8 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)

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*

15.9 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]* = {})

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

15.10 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

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*)

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.

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'`)

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'`

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*

name : VariableName

default_value : *ElaboratableValue* | None

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

15.11 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
```

(continues on next page)

(continued from previous page)

```
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

SIXTEEN

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_dependency()` or `add_external_ip()`, i.e. they won't be present in the generated core file.

class FuseSocBuilder(part)

Use this class to generate a FuseSoC .core file

add_dependency(dependency: *str*)

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_source(filename, type)

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.

build(top_name: *str*, core_path: *Path*, sources_dir: *Collection[Path]* = `[]`, template_name: *str* | *None* = `None`)

Generate the final create .core file

Parameters

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

template_name: str | None = `None`
name of jinja2 template to be used, either in working directory, or bundled with the package. defaults to a bundled template

CHAPTER

SEVENTEEN

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).

class **InterfaceDefinition**(name: str, port_prefix: str, signals: ~topwrap.interface.InterfaceDefinitionSignals = <factory>)

Interface described in YAML interface definition file

Schema

alias of `InterfaceDefinition`

static **get_builtins**() → dict[str, *InterfaceDefinition*]

Loads all builtin internal interfaces

Returns

a dict where keys are the interface names and values are the InterfaceDefinition objects

get_interface_by_name(name: *str*) → *InterfaceDefinition* | None

Retrieve interface definition by its name

Returns

InterfaceDefinition object, or *None* if there's no such interface

CHAPTER

EIGHTEEN

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.

class Config(force_interface_compliance: bool | None = False, repositories: dict[str, ~topwrap.resource_field.ResourceReferenceHandler] = <factory>, kpm_build_location: str = '/github/home/.local/cache/topwrap/kpm_build')

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.

BUILTIN_DIR = <contextlib._GeneratorContextManager object>

CHAPTER

NINETEEN

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_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

19.1 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.

19.1.1 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.

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.2 Invalid parameters' values

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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.3 Connection between external Metanodes

`dataflow_ext_in_to_ext_out_connections`()

Dataflow containing Metanode<->Metanode connection.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.4 Ports connected to multiple external Metanodes

`dataflow_ports_multiple_external_metanodes`()

Dataflow containing a port connected to two External Metanodes.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.5 Duplicate Metanode names

`dataflow_duplicate_metanode_names`()

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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.6 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.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.7 Unnamed Metanodes

dataflow_unnamed_metanodes()

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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.8 Connection between two inout ports

dataflow_inouts_connections()

Dataflow containing a connection between two inout ports.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.9 Unconnected ports in subgraph node

dataflow_unconn_hierarchy()

Dataflow containing subgraph node with two unconnected interfaces.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.10 Connection of subgraph node to multiple External Metanodes

dataflow_subgraph_multiple_external_metanodes()

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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.11 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.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.12 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.

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

19.1.13 Duplicate IP cores in subgraph node

dataflow_hier_duplicate_names()

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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

CHAPTER

TWENTY

FUTURE PLANNED ENHANCEMENTS IN TOPWRAP

20.1 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.

20.2 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.

20.3 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

TWENTYONE

OTHER POSSIBLE IMPROVEMENTS

21.1 Ability to produce top-level wrappers in VHDL

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

21.2 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.

21.3 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.

21.4 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.

21.5 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

TWENTYTWO

USING KPM IFRAMES INSIDE DOCS

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

22.1 Usage

```
```{kpm_iframe}
:spec: <KPM specification .json file URI>
:dataflow: <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.

22.2 Tests

22.2.1 Use remote specification

Note: The graph below is supposed to be empty.

It doesn't load a dataflow, only a specification that provides IP-cores to the Nodes browser on the sidebar.

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

22.2.2 Use local files

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

22.2.3 Open in preview mode

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

22.2.4 Use a custom alt text

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

Note: This diagram showcases the block design of the "hierarchy" example

CHAPTER

TWENTYTHREE

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.

23.1 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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

23.2 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

> **Note:** An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

23.3 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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

23.4 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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

23.5 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

Note: An interactive KPM frame, where you can explore the block design for this section, is available here in the HTML version of this documentation.

23.6 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`.

23.7 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

TWENTYFOUR

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.

24.1 General observations

24.1.1 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.

24.1.2 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`.

24.1.3 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.

24.1.4 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.

24.1.5 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 flag for `topwrap build` where all HDL sources are plainly forwarded to the FuseSoC .core. There is no notion of other file dependencies inside IP Core description YAMLs.

24.1.6 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>
```

24.1.7 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>
```

24.2 Simple example

This is the simplest IP-XACT example as it contains only plain IP cores with standalone ports, and parameters.

24.2.1 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.

24.2.2 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>
```

24.2.3 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.

24.2.4 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>
```

24.3 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".

24.3.1 Bus definitions

Custom interfaces in Topwrap are defined using *Interface description 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*.

24.3.2 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>
```

24.3.3 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.

24.3.4 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.

24.4 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.

24.5 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.

24.5.1 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>
```

24.5.2 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.

24.6 Other features

24.6.1 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.

24.7 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 mapping and 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

TWENTYFIVE

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

25.1 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.

25.2 Lookup maps

`SystemVerilogBackend` uses `verilog_generators_map` as lookup map.

25.3 API Reference

class Generator

add_module_instance_to_design(interconnect: *_IT*) → *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: _IT
Interconnect to represent as *ModuleInstance*

abstract generate(interconnect: *_IT*, module_instance: ModuleInstance) → _T

Returns generated HDL code wrapped in class specific for backend

Parameters

interconnect: _IT
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

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

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

TWENTYSIX

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

TWENTYSEVEN

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*.

27.1 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

PYTHON MODULE INDEX

t

INDEX

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W