



Antmicro

Topwrap

2024-08-26

DOCUMENTATION

1	Introduction	1
2	Getting started	2
2.1	Installation	2
2.2	Example usage	3
3	Description files	7
3.1	Design Description	7
3.2	IP description files	9
3.3	Interface Description files	11
4	Interconnect generation	13
4.1	Format	13
4.2	Interconnect params	14
4.3	Limitations	14
5	Command-line Interface	15
5.1	Building design	15
5.2	Connect Topwrap to Pipeline Manager	15
5.3	Generating IP core description YAMLs	16
6	User repositories	17
7	Hierarchies	19
7.1	Format	19
8	Kenning Pipeline Manager	21
8.1	Run Topwrap with Pipeline Manager	21
8.2	Pipeline Manager features	24
9	FuseSoC	26
10	Setup	27
11	Code style	28
11.1	Lint with nox	28
11.2	Lint with pre-commit	28
11.3	Tools	29
12	Tests	30
12.1	Test execution	30

12.2 Test coverage	31
13 Wrapper	32
14 IPWrapper class	33
15 IPConnect class	35
16 ElaboratableWrapper class	40
17 Wrapper Port	41
18 FuseSocBuilder	43
19 Interface definition	45
20 Config	46
21 Deducing interfaces	47
21.1 Step 1. - splitting ports into subsets	47
21.2 Step 2. - matching ports with interface signal names	48
21.3 Step 3. - inferring interface direction	48
21.4 Step 4. - computing interface matching score	48
22 Examples	50
23 Future enhancements	51
23.1 Support for hierarchical block design in Pipeline Manager	51
23.2 Bus management	51
23.3 Improve the process of recreating a design from a YAML file	51
23.4 Support for parsing SystemVerilog sources	52
23.5 Provide a way to parse HDL sources from the Pipeline Manager level	52
23.6 Ability to produce top-level wrappers in VHDL	52
23.7 Library of open-source cores	52
23.8 Integrating with other tools	52
Index	53

INTRODUCTION

ASIC and FPGA designs consist of distinct blocks of logic bound together by a top-level design. To take advantage of this modularity and enable reuse of blocks across designs and so facilitate the shift towards automation in logic design, it is necessary to derive a generic way to aggregate the blocks in various configurations and make the top-level design easy to parse and process automatically.

Topwrap is an open source command line toolkit for connecting individual HDL modules into full designs of varying complexity. The toolkit is designed to take advantage of the ever-growing availability of open source digital logic designs and offers a user-friendly graphical interface which lets you mix-and-match GUI-driven design with CLI-based adjustments and present designs in a diagram form thanks to the integration with Antmicro's [Pipeline Manager](#).

Topwrap's most notable features are:

- Parsing HDL design files with automatic recognition of common interfaces
- Simple YAML-based description for command-line use
- Capability to create a custom libraries for reuse across projects
- User-friendly GUI powered by [Kenning Pipeline Manager](#).

GETTING STARTED

2.1 Installation

1. Install required system packages:

Debian:

```
apt install -y git g++ make python3 python3-pip antlr4 libantlr4-runtime-dev  
↪ yosys npm
```

Arch:

```
pacman -Syu git gcc make python3 python-pip antlr4 antlr4-runtime yosys npm
```

Fedora:

```
dnf install git g++ make python3 python3-pip python3-devel antlr4 antlr4-cpp-  
↪ runtime-devel yosys npm
```

2. Install the Topwrap package (It is highly recommended to run this step in a Python virtual environment, e.g. **venv**):

```
pip install .
```

Note: To use topwrap parse command you also need to install optional dependencies:

```
pip install ".[topwrap-parse]"
```

On Arch-based distributions a symlink to antlr4 runtime library needs to be created and an environment variable set:

```
ln -s /usr/share/java/antlr-complete.jar antlr4-complete.jar  
ANTLR_COMPLETE_PATH=`pwd` pip install ".[topwrap-parse]"
```

On Fedora-based distributions symlinks need to be made inside /usr/share/java directory itself:

```
sudo ln -s /usr/share/java/stringtemplate4/ST4.jar /usr/share/java/  
↪ stringtemplate4.jar  
sudo ln -s /usr/share/java/antlr4/antlr4.jar /usr/share/java/antlr4.jar
```

(continues on next page)

(continued from previous page)

```
sudo ln -s /usr/share/java/antlr4/antlr4-runtime.jar /usr/share/java/antlr4-  
↳runtime.jar  
sudo ln -s /usr/share/java/treelayout/org.abego.treelayout.core.jar /usr/share/  
↳java/treelayout.jar  
pip install ".[topwrap-parse]"
```

2.2 Example usage

This section demonstrates the basic usage of Topwrap to generate IP wrappers and a top module.

1. Create *IP core description* file for every IP Core you want to use or let *topwrap parse* handle this for you. This file describes the ports, parameters and interfaces of an IP core.

As an example, Verilog module such as:

```
module ibuf (  
    input wire clk,  
    input wire rst,  
    input wire a,  
    output reg z  
);  
    // ...  
endmodule
```

Needs this corresponding description:

```
signals:  
  in:  
    - clk  
    - rst  
    - a  
  out:  
    - z
```

2. Create a *design description* file where you can specify all instances of IP cores and connections between them (project.yaml in this example)
 - Create instances of IP cores:

```
ips:  
  vexriscv_instance:  
    file: ipcores/gen_VexRiscv.yaml  
    module: VexRiscv  
  wb_ram_data_instance:  
    file: ipcores/gen_mem.yaml  
    module: mem  
  wb_ram_instr_instance:  
    file: ipcores/gen_mem.yaml  
    module: mem
```

file and module are mandatory fields providing the IP description file and the name of the HDL module as it appears in the source.

- (Optional) Set parameters for IP core instances:

```
parameters:
  wb_ram_data_instance:
    depth: 0x1000
    memfile: "top_sram.init"
  wb_ram_instr_instance:
    depth: 0xA000
    memfile: "bios.init"
```

- Connect desired ports of the IP cores:

```
ports:
  wb_ram_data_instance:
    clk: [some_other_ip, clk_out]
    rst: [reset_core, reset0]
  wb_ram_instr_instance:
    clk: [some_other_ip, clk_out]
    rst: [reset_core, reset0]
  vexriscv_instance:
    softwareInterrupt: [some_other_ip, sw_interrupt]
  ...
```

Connections only need to be written once, i.e. if A connects to B, then only a connection A: B has to be specified (B: A is redundant).

- Connect desired interfaces of the IP cores:

```
interfaces:
  vexriscv_instance:
    iBusWishbone: [wb_ram_instr_instance, mem_bus]
    dBusWishbone: [wb_ram_data_instance, mem_bus]
```

- Specify external ports or interfaces of the top module and connect them with chosen IP cores' ports or interfaces:

```
ports:
  vexriscv_instance:
    timerInterrupt: ext_timer_interrupt
  ...

external:
  ports:
    out:
      - ext_timer_interrupt
  interfaces:
    ...
```

3. Create a Core file template for FuseSoC, or use a default one bundled with Topwrap.

You may want to modify the file to add dependencies, source files, or change the target board. The file should be named `core.yaml.j2`. You can find an example template in `examples/hdmi` directory of the project. If you don't create any template a default template bundled with Topwrap will be used (stored in `templates` directory).

4. Place any additional source files in a directory (sources in this example).
5. Run Topwrap:

```
python -m topwrap build --design project.yaml --sources sources
```

2.2.1 Example PWM design

There's an example setup in `examples/pwm`.

In order to generate the top module, run:

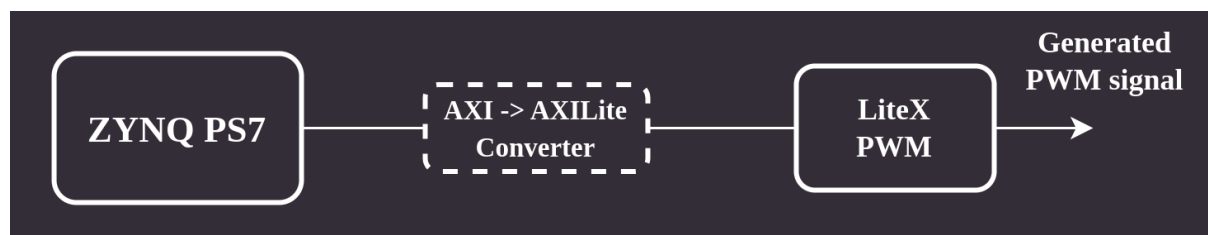
```
make generate
```

In order to generate bitstream, add Vivado to your path and run:

```
make
```

The FPGA design utilizes an AXI-mapped PWM IP Core. You can access its registers starting from address `0x4000000` (that's the base address of `AXI_GP0` on ZYNQ). Each IP Core used in the project is declared in `ips` section in `project.yaml` file. `ports` section allows to connect individual ports of IP Cores, and `interfaces` is used analogously for connecting whole interfaces. Finally, you can specify which ports will be used as external I/O in `external` section.

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



2.2.2 Example HDMI design

There's an example setup stored in `examples/hdmi`.

You can generate bitstream for desired target:

- Snickerdoodle Black:

```
make snickerdoodle
```

- Zynq Video Board:

```
make zvb
```

If you wish to generate HDL sources without running Vivado, you can use:


```
make generate
```

You can find more information in README of the example setup.

DESCRIPTION FILES

3.1 Design Description

To create a complete, fully synthesizable design, a proper design file is needed. It's used to specify interconnects, IP cores, set their parameters' values, describe hierarchies for the project, connect the IPs and hierarchies, and pick external ports (those which will be connected to physical I/O).

You can see example design files in examples directory. The structure is as below:

```
ips:
  # specify relations between IPs names in the design yaml
  # and IP cores description yamls and modules
  {ip_instance_name}:
    file: {path_to_yaml_file_of_the_ip}
    module: {name_of_the_HDL_module}
  ...

design:
  name: {design_name} # optional name of the toplevel
  hierarchies:
    # see "Hierarchies" page for a detailed description of the format
    ...
  parameters: # specify IPs parameter values to be overridden
    {ip_instance_name}:
      {param_name} : {param_value}
    ...

ports:
  # specify incoming ports connections of an IP named `ip1_name`
  {ip1_name}:
    {port1_name} : [{ip2_name}, {port2_name}]
    ...
  # specify incoming ports connections of a hierarchy named `hier_name`
  {hier_name}:
    {port1_name} : [{ip_name}, {port2_name}]
    ...
  # specify external ports connections
  {ip_instance_name}:
    {port_name} : ext_port_name
```

(continues on next page)

(continued from previous page)

```
...

interfaces:
  # specify incoming interfaces connections of `ip1_name` IP
  {ip1_name}:
    {iface1_name} : [{ip2_name}, {iface2_name}]
    ...
  # specify incoming interfaces connections of `hier_name` hierarchy
  {hier_name}:
    {iface1_name} : [{ip_name}, {iface2_name}]
    ...
  # specify external interfaces connections
  {ip_instance_name}:
    {iface_name} : ext_iface_name
  ...

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

external: # specify 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_iface_name}
  # note that `inout:` is invalid in the interfaces section
```

inout ports are handled differently than in and out ports. When any 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 `external.ports.inout` section of the parent design by specifying the name of the IP/hierarchy and port name that contains it. Name of the external port will be 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 creating hierarchical designs. In order to create a hierarchy, it suffices to 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’s correct to have `ip_name` key in this section and `ip_name` in the design section of some hierarchy.

3.2 IP description files

Every IP wrapped by Topwrap needs a description file in YAML format.

The ports of an IP should be placed in global signals node, followed by the direction of in, out or inout. Here's an example description of ports of Clock Crossing IP:

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

The previous example is enough to make use of any IP. However, in order to benefit from connecting whole interfaces at once, ports must belong to a named interface like in this example:

```
#file: axis_width_converter.yaml
interfaces:
  s_axis:
    type: AXIStream
    mode: slave
    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: master
    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]
        TUSER: m_axis_tuser
signals: # These ports don't belong to any interface
```

(continues on next page)

(continued from previous page)

```
in:
  - clk
  - rst
```

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 it's supposed to be connected to, for example `TDATA: port_name` will be connected to `TDATA: other_port_name`.

Note that you don't have to write IP core description yamls by hand. You can use Topwrap's `parse` command (see [Generating IP core description YAMLS](#)) in order to generate yamls from HDL source files and then adjust the yaml to your needs.

3.2.1 Port widths

The width of every port defaults to 1. You can specify the width using this notation:

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

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

3.2.2 Parameterization

Port widths don't have to be hardcoded - you can use parameters to describe an IP core in a generic way. 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: slave
    signals:
```

(continues on next page)

(continued from previous page)

```

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]

```

Parameters values can be integers or math expressions, which are evaluated using `numexpr.evaluate()`.

3.2.3 Port slicing

You can also slice a port, to use some bits of the port as a signal that belongs to an interface. The example below means:

Port `m_axi_bid` of the IP core is 36 bits wide. Use bits 23..12 as the BID signal of AXI master named `m_axi_1`

```

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

```

3.3 Interface Description files

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

You can see an example file below:

```

name: AXI4Stream
port_prefix: AXIS
signals:
  # convention assumes the AXI Stream transmitter (master) perspective
  required:
    out:
      TVALID: tvalid
      TDATA: tdata
      TLAST: tlast
    in:
      TREADY: tready
  optional:
    out:
      TID: tid
      TDEST: tdest

```

(continues on next page)

(continued from previous page)

TKEEP: tkeep
TSTRB: tstrb
TUSER: tuser
TWAKEUP: twakeup

The name of an interface has to be unique. We also specify a prefix which will be used as a shortened identifier. Signals are either required or optional. Their direction is described from the perspective of master (i.e. directionality of signals in the slave is flipped) - note that clock and reset are not included as these are usually inputs in both master and slave so they're not supported in interface specification. These distinctions are used when an option to check if all mandatory signals are present is enabled and when parsing an IP core with `topwrap parse` (not all required signals must necessarily be present but it's taken into account). Every signal is a key-value pair, where the key is a generic signal name (usually from interface specification) and value is a regex that is used to pair the generic name with a concrete signal name in the RTL source when using `topwrap parse`. This pairing is performed on signal names that are transformed to lowercase and have a common prefix of an interface they belong to removed. If a regexp occurs in such transformed signal name anywhere, that name is paired with the generic name. Since this occurs on names that have all characters in lowercase, regex must be written in lowercase as well.

INTERCONNECT GENERATION

Generating interconnects is an experimental feature of topwrap. With a specification of which interfaces are masters or slaves and their address ranges, topwrap is able to automatically generate an interconnect conforming to this description. Currently supported interconnect types are:

- Wishbone round-robin

4.1 Format

The format for describing interconnects is specified below. 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 interface:
    # clock: ext_clk_port_name
    # reset: ext_rst_port_name

    # specify interconnect type
    type: {interconnect_type}

    # specify interconnect parameters - interconnect-type-dependent (see
    ↪ "Interconnect params" section):
    params:
      {param_name1}: param_value1
      ...

    # specify masters and their interfaces connected to the bus
    masters:
      {master1_name}:
        - {master1_iface1_name}
        ...
      ...

    # specify slaves, their interfaces connected to the bus and their bus_
```

(continues on next page)

(continued from previous page)

```
→parameters
  slaves:
    {slave1_name}:
      {slave1_interface1_name}:
        # requests in address range [address, address+size) will be routed to this_
→interface
  address: {start_address}
  size: {range_size}
  ...
  ...
```

4.2 Interconnect params

Different interconnect types may provide different configuration options. This section lists parameter names for available interconnects for use in the params section of interconnect specification.

4.2.1 Wishbone round-robin

Corresponds to type: wishbone_roundrobin

- `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 - smallest unit of data transfer that the interconnect is capable of transferring. Must be one of: 8, 16, 32, 64
- `features` - optional, list of optional wishbone signals, can contain: `err`, `rty`, `stall`, `lock`, `cti`, `bte`

4.3 Limitations

Known limitations currently are:

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

COMMAND-LINE INTERFACE

5.1 Building design

To run Topwrap, use:

```
python -m topwrap build --design project.yml
```

Where `project.yml` should be your file with description of the top module.

You can specify a directory to be scanned for additional sources:

```
python -m topwrap build --sources src --design project.yml
```

To implement the design for a specific FPGA chip, provide the part name:

```
python -m topwrap build --sources src --design project.yml --part 'xc7z020c1g400-3'
↪
```

5.2 Connect Topwrap to Pipeline Manager

If you want to use Pipeline Manager as a UI for creating block design, you need to:

1. Build and run Pipeline Manager server application.

```
python -m topwrap kpm_build_server
python -m topwrap kpm_run_server
```

2. Run Topwrap's client application, that will connect to a running Pipeline Manager server app.

```
python -m topwrap kpm_client [-h ip_addr] [-p port] [-d FILE] FILES
```

Topwrap will then try to connect to the server running on `ip_addr:port` and send a specification generated from `FILES`, which should be IP core description yamls.

If `-h` or `-p` options are not specified, ip address `127.0.0.1` and port `9000` will be chosen by default.

If `-d` option is specified, kpm will start with specified design file loaded.

5.3 Generating IP core description YAMLs

You can also use Topwrap to generate ip core description yamls from HDL sources, that can be later used in your `project.yml`:

```
python -m topwrap parse HDL_FILES
```

In HDL source files, ports that belong to the same interface (e.g. wishbone or AXI), have often a common prefix, which corresponds to the interface name. If such naming convention is followed in the HDL sources, Topwrap can also divide ports into user-specified interfaces, or automatically deduce interfaces names when generating yaml file:

```
python -m topwrap parse --iface wishbone --iface s_axi HDL_FILES
```

```
python -m topwrap parse --iface-deduce HDL_FILES
```

To get help, use:

```
python -m topwrap [build|kpm_client|parse] --help
```

USER REPOSITORIES

Repositories allow for easy loading packages with IP-cores.

You can add repositories to be loaded each time topwrap is ran by specifying them in configuration file.

It has to be located in one of the following locations:

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

Example contents of user config:

```
force_interface_compliance: true
repositories:
  - name: name of repo
    path: ~/path_to_repo/repo
```

Topwrap provides internal API for constructing repositories in python code which can be [found here](#)

Structure of repository has to be as follows:

```
path_to_repository/
|—cores
|   |—someCore1
|   |   |—srcs
|   |   |   | file1.v
|   |   |   | design.yaml
|   |
|   |—someCore1
|       |—srcs
|           | file1.v
|           | design.yaml
|
|—interfaces(Optional)
|   interface1.yaml
|   interface2.yaml
```

Repository has two main catalogs: cores and interfaces. Inside cores each core has it's own catalog with it's design file and srcs where are stored verilog/VHDL files.

There is optional interfaces catalog where can be stored interfaces for cores.

Example User Repo can be found in [examples/user_repository](#).

HIERARCHIES

Hierarchies allow for creating designs with subgraphs in them. Subgraphs can contain multiple IP-cores and other subgraphs. This allows creating nested designs in topwrap.

7.1 Format

All information about hierarchies is specified in *design description*. hierarchies key must be a direct descendant of the design key. 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 to 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}
        ...
```

(continues on next page)

(continued from previous page)

```
{hierarchy_name_2}:  
...
```

More complex hierarchy example can be found in [examples/hierarchy](#).

KENNING PIPELINE MANAGER

Topwrap can make use of **Kenning Pipeline Manager** to visualize the process of creating block design.

8.1 Run Topwrap with Pipeline Manager

1. Build and run Pipeline Manager server

In order to start creating block design in Pipeline Manager, you need to first build and run a server application - here is a brief instruction on how to achieve this (the process of building and installation of Pipeline Manager is described in detail in its [documentation](#)):

```
python -m topwrap kpm_build_server
python -m topwrap kpm_run_server
```

After executing the above-mentioned commands, the Pipeline Manager server is waiting for an external application (i.e. Topwrap) to connect on 127.0.0.1:9000 and you can connect to the web GUI frontend in your browser on <http://127.0.0.1:5000>.

2. Establish connection with Topwrap

Once the Pipeline Manager server is running, you can now launch Topwrap's client application in order to connect to the server. You need to specify:

- IP address (127.0.0.1 is default)
- listening port (9000 is default)
- yamls describing IP cores, that will be used in the block design
- design to load initially (None by default)

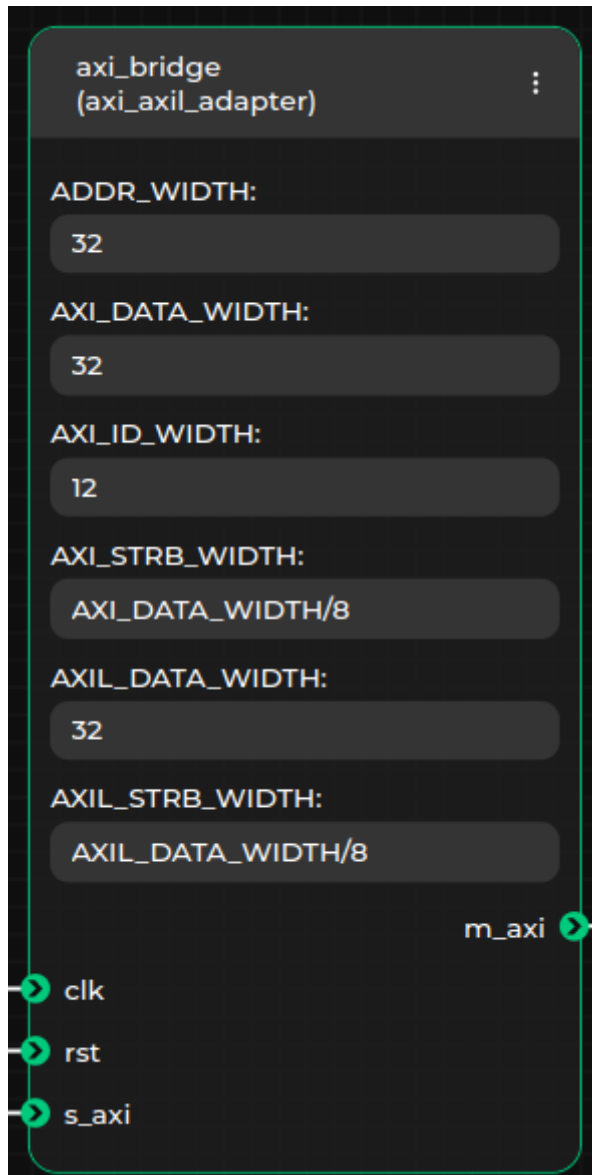
An example command, that runs Topwrap's client, may look like this:

```
python -m topwrap kpm_client -h 127.0.0.1 -p 9000 \
    topwrap/ips/axi/axil_adapter.yaml \
    examples/pwm/ipcores/{litex_pwm.yaml,ps7.yaml} -d examples/pwm/project.yaml
```

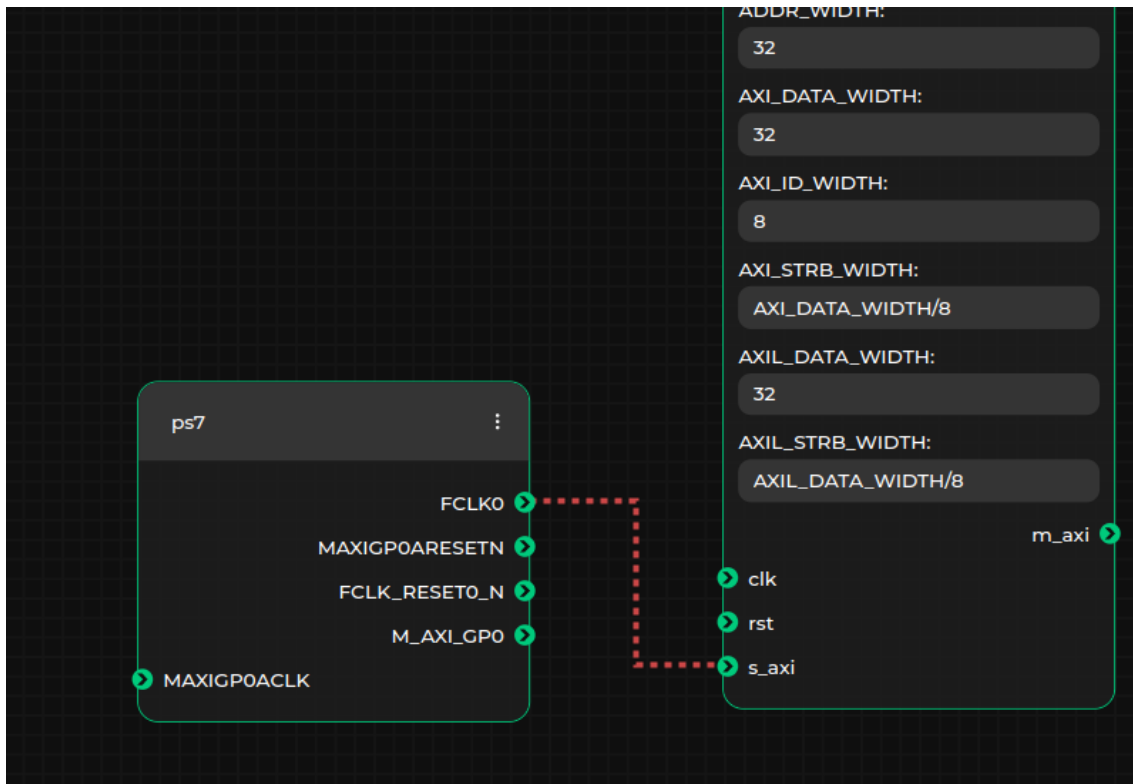
3. Create block design in Pipeline Manager

Upon successful connection to a Pipeline Manager server, Topwrap will generate and send to the server a specification describing the structure of previously selected IP cores. After that, you are free to create a custom block design by means of:

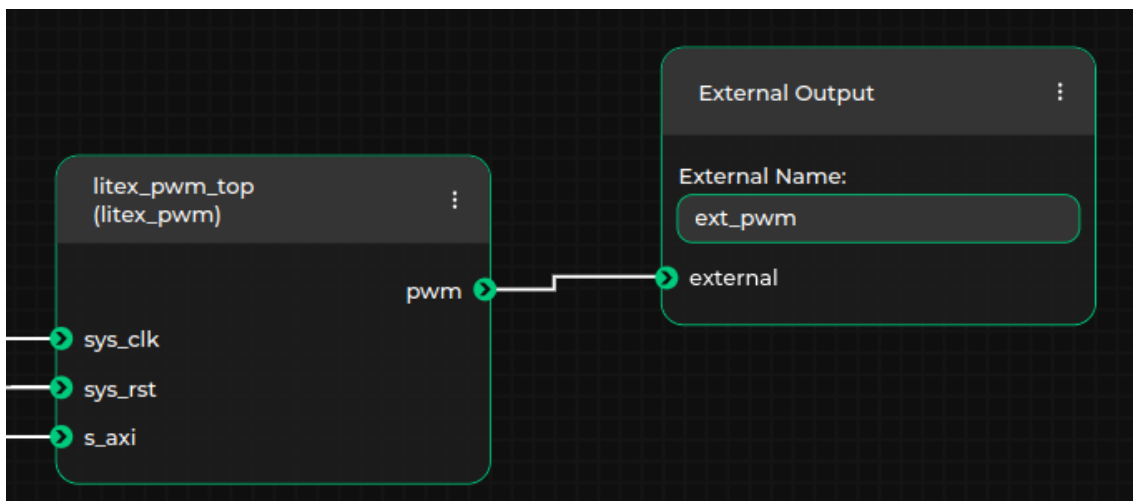
- adding IP core instances to the block design. Each Pipeline Manager's node has delete and rename options, which make it possible to remove the selected node and change its name respectively. This means that you can create multiple instances of the same IP core.
- adjusting IP cores' parameters values. Each node may have input boxes in which you can enter parameters' values (default parameter values are added while adding an IP core to the block design):



- connecting IP cores' ports and interfaces. Only connections between ports or interfaces of matching types are allowed. This is automatically checked by Pipeline Manager, as the types of nodes' ports and interfaces are contained in the loaded specification, so Pipeline Manager will prevent you from connecting non-matching interfaces (e.g. *AXI4* with *AXI4Lite* or a port with an interface). A green line will be displayed if a connection is possible to create, or a red line elsewhere:

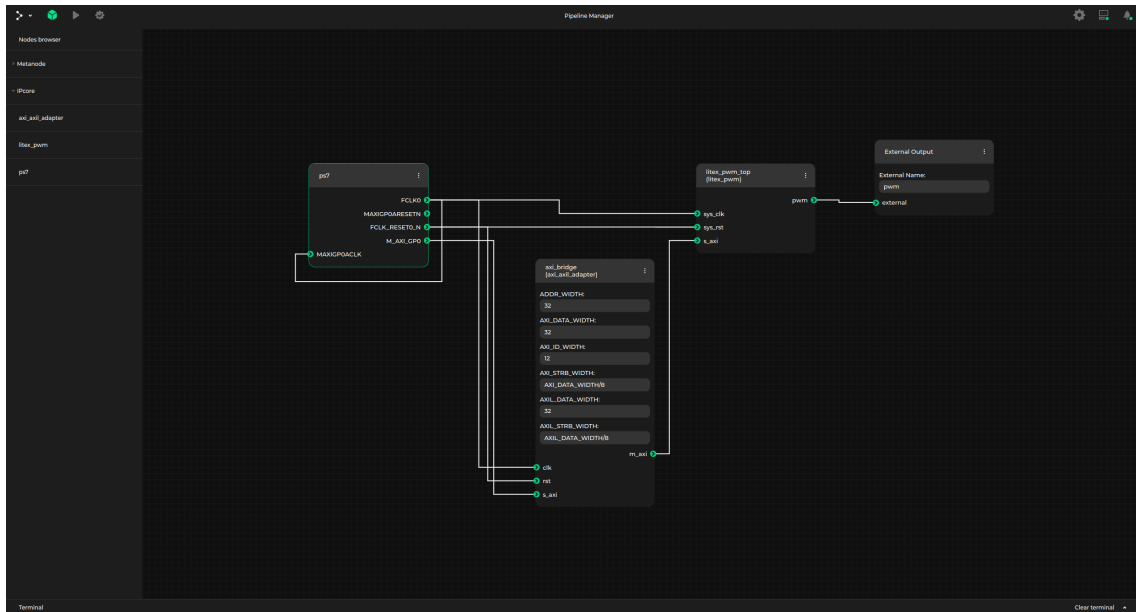


- specifying external ports or interfaces in the top module. This can be done by adding External Input, External Output or External Inout metanodes and creating connections between them and chosen ports or interfaces. Note that you should adjust the name of the external port or interface in a textbox inside selected metanode. In the example below, output port `pwm` of `litex_pwm_top` IP core will be made external in the generated top module and the external port name will be set to `ext_pwm`:



Note, that you don't always have to create a new block design by hand - you can use a *design import* feature to load an existing block design from a description in Topwrap's yaml format.

An example block design in Pipeline Manager for the PWM project may look like this:



8.2 Pipeline Manager features

While creating a custom block design, you can make use of the following Pipeline Manager's features:

- export (save) design to a file
- import (load) design from a file
- validate design
- build design

8.2.1 Export design to yaml description file

Created block design can be saved to a *design description file* in yaml format, using Pipeline Manager's Save file option. Target location on the filesystem can then be browsed in a filesystem dialog window.

8.2.2 Import design from yaml description file

Topwrap also supports conversion in the opposite way - block design in Pipeline Manager can be generated from a yaml design description file using Load file feature.

8.2.3 Design validation

Pipeline Manager is capable of performing some basic checks at runtime such as interface type checking while creating a connection. However you can also run more complex tests by using Pipeline Manager's `Validate` option. Topwrap will then respond with a validity confirmation or error messages. The rules you need to follow in order to keep your block design valid are:

- multiple IP cores with the same name are not allowed (except from external metanodes).
- parameters values can be integers of different bases (e.g. `0x28`, `40` or `0b101000`) or arithmetic expressions, that are later evaluated using `numexpr.evaluate()` function (e.g. $(AXI_DATA_WIDTH+1)/4$ is a valid parameter value assuming that a parameter named `AXI_DATA_WIDTH` exists in the same IP core). You can also write a parameter value in a Verilog format (e.g. `8'b00011111` or `8'h1F`) - in such case it will be interpreted as a fixed-width bit vector.
- a single port or interface cannot be external and connected to another IP core at the same time.
- connections between two external metanodes are not allowed.
- all the created external output or inout ports must have unique names. Only multiple input ports of IP cores can be driven by the same external signal.

Topwrap can also generate warnings if:

- some ports or interfaces remain unconnected.
- multiple ports are connected to an External Input metanode with an empty External Name property.
- inout ports of two modules are connected together (all inout ports are required to be directly connected to External Inout metanodes)

If a block design validation returns a warning, it means that the block design can be successfully built, but it is recommended to follow the suggestion and resolve a particular issue.

8.2.4 Building design

Once the design has been created and tested for validity, you can build design using Run button. If the design does not contain any errors, this will result in creating a top module in a directory where topwrap kpm_client was ran, similarly when using Topwrap's `topwrap build` command.

FUSESOC

Topwrap uses FuseSoC to automate project generation and build process. When `topwrap build` is invoked it generates a FuseSoC core file along with the top-level wrapper.

A template for the core file is bundled with Topwrap (`templates/core.yaml.j2`). You may need to edit the file to change the backend tool, set additional Hooks and change the FPGA part name or other parameters. By default, `topwrap.fuse_helper.FuseSocBuilder` searches for the template file in the directory you work in, so you should first copy the template into the project's location.

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

```
fusesoc --cores-root build run project_1
```

This requires having the suitable backend tool in your PATH (Vivado, for example).

SETUP

It is required for developers to keep code style and recommended to frequently run tests. In order to setup the developer's environment install optional dependency groups `topwrap-parse`, `tests` and `lint` specified in `pyproject.toml` which include `nox` and `pre-commit`:

```
python -m venv venv
source venv/bin/activate
pip install -e ".[topwrap-parse,tests,lint]"
```

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

CODE STYLE

Automatic formatting and linting of the code can be performed with either `nox` or `pre-commit`.

11.1 Lint with nox

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

```
nox -s lint
```

This runs `isort`, `black`, `flake8` and `codespell` and fixes almost all formatting and linting problems automatically, but a small minority has to be fixed by hand (e.g. unused imports).

Note: To reuse current virtual environment and avoid long installation time use `-R` option:

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

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

```
nox -s pre_commit
```

11.2 Lint with pre-commit

Alternatively, you can 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` can be easily disabled with a similar command:

```
pre-commit uninstall
```

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

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

Note: pre-commit by default also runs nox with isort,flake8, black and codespell sessions

11.3 Tools

Tools used in project for maintaining code style:

- Nox is a tool, which simplifies management of Python testing. [Visit nox website](#)
- Pre-commit is a framework for managing and maintaining multi-language pre-commit hooks. [Visit pre-commit website](#)
- Black is a code formatter. [Visit black website](#)
- Flake8 is a tool capable of linting, styling fixes and complexity analysis. [Visit flake8 website](#)
- Isort is a Python utility to sort imports alphabetically. [Visit isort website](#)
- Codespell is a Python tool to fix common spelling mistakes in text files [Visit codespell repository](#)

Topwrap functionality is validated with tests, leveraging the pytest library.

12.1 Test execution

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

```
nox -s tests
```

This only runs tests on 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 existing virtual environment and avoid long installation time use `-R` option:

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

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

```
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 CI runs them on all supported Python versions it's recommended to run tests with `nox` on all versions before pushing.

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

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

Sometimes it's useful to see what's being printed by the test for debugging purposes. Pytest captures all output from the test and displays it when all tests finish. To see the output immediately, pass `-s` option to pytest:

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

12.2 Test coverage

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

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

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

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

Generated report is available at `htmlcov/index.html`

WRAPPER

`Wrapper` is an abstraction over entities that have ports - examples include IP cores written in Verilog/VHDL, cores written in Amaranth and hierarchical collections for these that expose some external ports. Subclasses of this class have to supply implementation of property `get_ports()` that has to return a list of all ports of the entity.

class `Wrapper(*args, src_loc_at=0, **kwargs)`

Base class for modules that want to connect to each other.

Derived classes must implement `get_ports` method that returns a list of `WrapperPort`'s - external ports of a class that can be used as endpoints for connections.

`__init__(name: str)`

`get_port_by_name(name: str) → WrapperPort`

Given port's name, return the port as `WrapperPort` object.

Raises

ValueError – If such port doesn't exist.

`get_ports() → List[WrapperPort]`

Return a list of external ports.

`get_ports_of_interface(iface_name: str) → List[WrapperPort]`

Return a list of ports of specific interface.

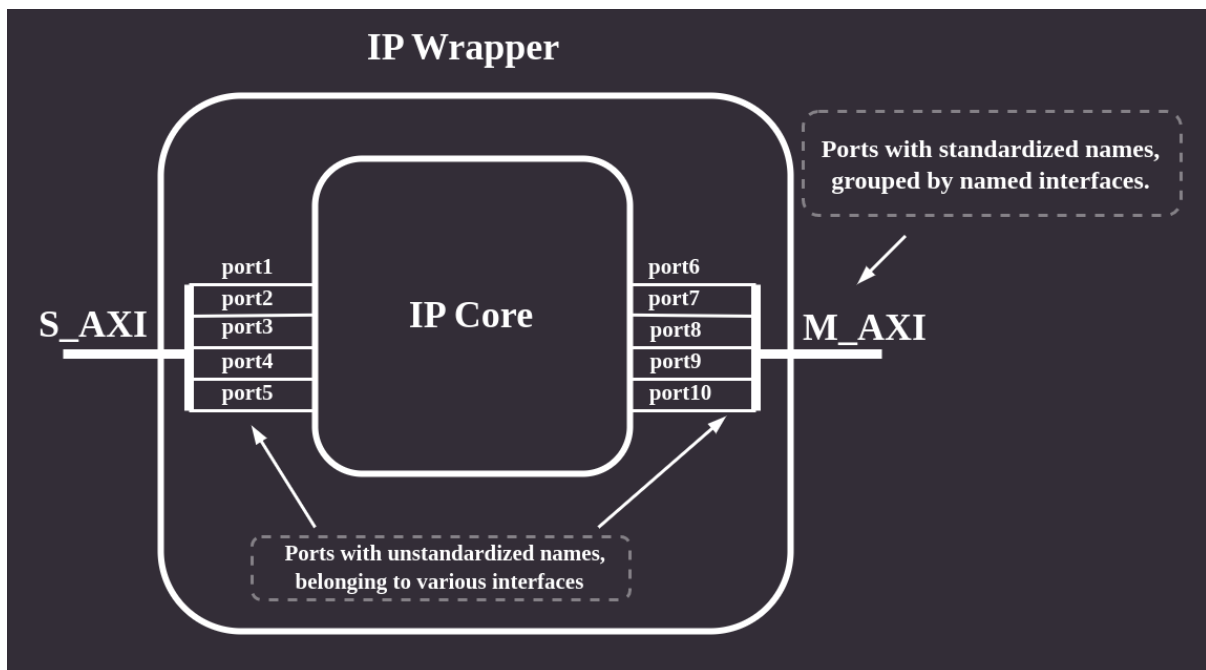
Raises

ValueError – if such interface doesn't exist.

IPWRAPPER CLASS

IPWrapper provides an abstraction over a raw HDL source file. Instances of this class can be created from a loaded YAML IP-core description.

Under the hood it will create Amaranth's Instance object during elaboration, referencing a particular HDL module and it will appear as a module instantiation in the generated toplevel. Ports and interfaces (lists of ports) can be retrieved via standard methods of **Wrapper**. These are instances of **WrapperPorts**.



```
class IPWrapper(*args, src_loc_at=0, **kwargs)
```

This class instantiates an IP in a wrapper to use its individual ports or grouped ports as interfaces.

```
__init__(yaml_path: Path, ip_name: str, instance_name: str, params={})
```

Parameters

yaml_path: Path

path to the IP Core description yaml file

ip_name: str

name of the module to wrap

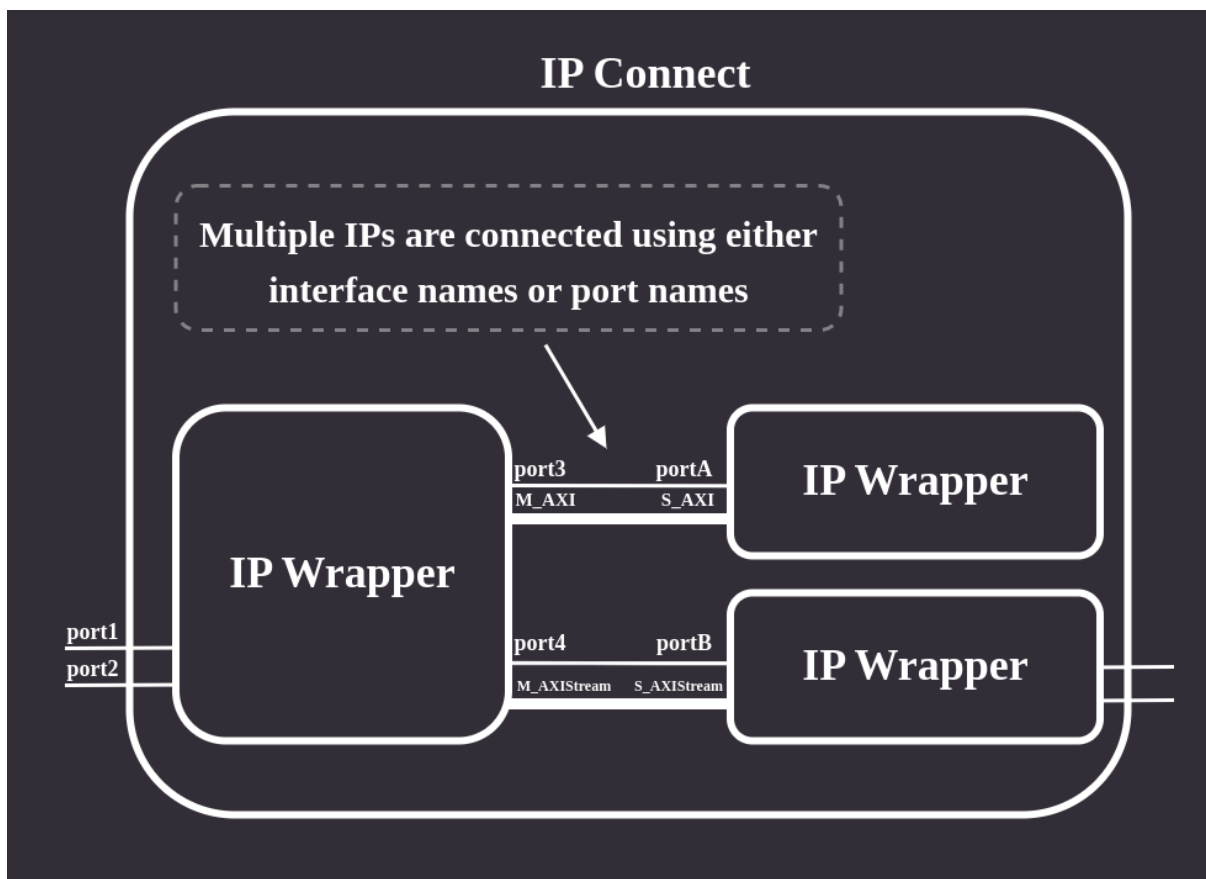
instance_name: str
name of this instance

params={}
optional, HDL parameters of this instance

get_ports() → List[*WrapperPort*]
Return a list of all ports that belong to this IP.

IPCONNECT CLASS

`IPConnect` provides means of connecting ports and interfaces of objects that are subclasses of `Wrapper`. Since `IPConnect` is a subclass of `Wrapper` itself, this means that it also has IO - ports and interfaces, and that multiple `IPConnects` can have their ports and interfaces connected to each other (or other objects that subclass `Wrapper`).



Instances of `Wrapper` objects can be added to an `IPConnect` using `add_component()` method:

```
# create a wrapper for an IP
dma = IPWrapper('DMATop.yaml', ip_name='DMATop', instance_name='DMATop0')
ipc = IPConnect()
ipc.add_component("dma", dma)
```

Connections between cores can then be made with `connect_ports()` and `connect_interfaces()` based on names of the components and names of ports/interfaces:

```
ipc.connect_ports("comp1_port_name", "comp1_name", "comp2_port_name", "comp2_name"
↪)
ipc.connect_interfaces("comp1_interface_name", "comp1_name", "comp2_interface_name"
↪, "comp2_name")
```

Setting ports or interfaces of a module added to `IPConnect` as external with `_set_port()` and `_set_interface()` and allows these ports/interfaces to be connected to other `Wrapper` instances.

```
ipc._set_port("comp1_name", "comp1_port_name", "external_port_name")
ipc._set_interface("comp1_name", "comp1_interface_name", "external_interface_name"
↪)
```

This is done automatically in `make_connections()` method when the design is built based on the data from the YAML design description.

class IPConnect(*args, src_loc_at=0, **kwargs)

Connector for multiple IPs, capable of connecting their interfaces as well as individual ports.

__init__(name: str = 'ip_connector')

_connect_external_ports(internal: WrapperPort, external: WrapperPort)

Makes a pass-through connection - port of an internal module in IPConnect is connected to an external IPConnect port.

Parameters

internal: WrapperPort

port of an internal module of IPConnect

external: WrapperPort

external IPConnect port

_connect_internal_ports(port1: WrapperPort, port2: WrapperPort)

Connects two ports with matching directionality. Disallowed configurations are: - input to input - output to output - inout to inout All other configurations are allowed.

Parameters

port1: WrapperPort

1st port to connect

port2: WrapperPort

2nd port to connect

_connect_to_external_port(internal_port: str, internal_comp: str, external_port: str, external: DesignExternalPorts) → None

Connect internal port of a component to an external port

Parameters

internal_port: str

internal port name in internal_component to connect to external_port

internal_comp: str

internal component name

external_port: str
external port name

external: DesignExternalPorts
dictionary in the form of {"in": list, "out": list, "inout": list} containing port names specified as external in each of the three categories. All keys are optional and lack of a category implies an empty list

_set_interface(comp_name: str, iface_name: str, external_iface_name: str) → None
Set interface specified by name as an external interface

Parameters

comp_name: str
name of the component - hierarchy or IP core

iface_name: str
interface name in the component

external_iface_name: str
external name of the interface specified in "external" section

Raises

ValueError – if such interface doesn't exist

_set_port(comp_name: str, port_name: str, external_name: str) → None
Set port specified by name as an external port

Parameters

comp_name: str
name of the component - hierarchy or IP core

port_name: str
port name in the component

external_name: str
external name of the port specified in "external" section

Raises

ValueError – if such port doesn't exist

add_component(name: str, component: Wrapper) → None
Add a new component to this IPConnect, allowing to make connections with it

Parameters

name: str
name of the component

component: Wrapper
Wrapper object

connect_interfaces(iface1: str, comp1_name: str, iface2: str, comp2_name: str) → None

Make connections between all matching ports of the interfaces

Parameters

iface1: str
name of the 1st interface

comp1_name: str
name of the 1st IP

iface2: str
name of the 2nd interface

comp2_name: str
name of the 2nd IP

Raises

ValueError – if any of the IPs doesn't exist

connect_ports(**port1_name: str**, **comp1_name: str**, **port2_name: str**, **comp2_name: str**)
→ None

Connect ports of IPs previously added to this Connector

Parameters

port1_name: str
name of the port of the 1st IP

comp1_name: str
name of the 1st IP

port2_name: str
name of the port of the 2nd IP

comp2_name: str
name of the 2nd IP

Raises

ValueError – if such IP doesn't exist

get_ports() → list
Return a list of external ports of this module

make_connections(**ports: DS_PortsT**, **interfaces: DS_InterfacesT**, **external: DesignExternalSection**) → None

Use names of port and names of ips to make connections

Parameters

ports: DS_PortsT
“ports” section in the YAML design specification

interfaces: DS_InterfacesT
“interfaces” section in the YAML design specification

external: DesignExternalSection
“external” section in the YAML design specification

make_interconnect_connections(**interconnects: Dict[str, DesignSectionInterconnect]**,
external: DesignExternalSection)

Connect slaves and masters to their respective interfaces in the interconnect

Parameters

interconnects: Dict[str, DesignSectionInterconnect]
“interconnects” section in the YAML design specification

external: DesignExternalSection
“external” section in the YAML design specification

set_constant(**comp_name:** str, **comp_port:** str, **target:** int) → None

Set a constant value on a port of an IP

Parameters

comp_name: str
name of the IP or hierarchy

comp_port: str
name of the port of the IP or hierarchy

target: int
int value to be assigned

Raises

ValueError – if such IP doesn’t exist

validate_inout_connections(**inouts:** Collection[Tuple[str, str]])

Checks that all inout ports of any IP or hierarchy in the design are explicitly listed in the ‘external’ section.

Parameters

inouts: Collection[Tuple[str, str]]
external.ports.inout section of the design description YAML

ELABORATABLEWRAPPER CLASS

`ElaboratableWrapper` encapsulates an Amaranth's `Elaboratable` and exposes an interface compatible with other wrappers which allows making connections with them. Supplied `elaboratable` must contain a `signature` property and a conforming interface as specified by [Amaranth docs](#). Ports' directionality, their names and widths are inferred from it.

class `ElaboratableWrapper`(*args, src_loc_at=0, **kwargs)

Allows connecting an Amaranth's `Elaboratable` with other classes derived from `Wrapper`.

__init__(name: str, elaboratable: `Elaboratable`)

Parameters

name: str

name of this wrapper

elaboratable: `Elaboratable`

Amaranth's `Elaboratable` object to wrap

get_ports() → List[`WrapperPort`]

Return a list of external ports.

get_ports_hier() → Mapping[str, Signal | Mapping[str, Signal | SignalMapping]]

Maps `elaboratable`'s `Signature` to a nested dictionary of `WrapperPorts`. See `_gather_signature_ports` for more details.

WRAPPER PORT

Class `WrapperPort` is an extension to Amaranth's `Signal`. It wraps a port, adding a new name and optionally slicing the signal. It adds these attributes:

```
WrapperPort.internal_name    # name of the port in internal source to be wrapped
WrapperPort.direction        # DIR_FANIN, DIR_FANOUT or DIR_NONE
WrapperPort.interface_name   # name of the group of ports (interface)
WrapperPort.bounds           # range of bits that belong to the port
                             # and range which is sliced from the port
```

See [Port slicing](#) to know more about bounds.

This is used in `IPWrapper` class implementation and there should be no need to use `WrapperPort` individually.

Warning: `WrapperPort` is scheduled to be replaced in favor of plain Amaranth's `Signal` so it should not be used in any new functionality.

```
class WrapperPort(shape=None, src_loc_at=0, **kwargs)
```

```
    __init__(shape=None, src_loc_at=0, *, bounds: List[int], name: str, internal_name: str,
              interface_name: str | None = None, direction: PortDirection)
```

Wraps a port, adding a new name and optionally slicing the signal

Parameters

bounds: `List[int]`

4-element list where: [0:1] - upper and lower bounds of reference signal, [2:3] - upper and lower bounds of internal port, which are either the same as reference port, or a slice of the reference port

name: `str`

a new name for the signal

internal_name: `str`

name of the port to be wrapped/sliced

interface_name: `str | None = None`

name of the interface the port belongs to

direction: `PortDirection`

one of `PortDirection`, e.g. `DIR_OUT`

static **like**(**other**, ****kwargs**)

Creates a WrapperPort object with identical parameters as other object

Parameters

other

object to clone data from

****kwargs**

optional constructor parameters to override

FUSESOCBUILDER

Topwrap has support for generating FuseSoC's core files with `FuseSocBuilder`. Such core file contains information about source files and synthesis tools. Generation 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

`__init__(part)`

`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*)

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

build(*core_filename*, *sources_dir*=[], *template_name*=None)

Generate the final create .core file

Parameters

sources_dir=[]

additional directory with source files to add

template_name=None

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

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 YAML structure into Python objects and performs validation (that the YAML has the correct format) and typechecking (that the loaded values are of correct types).

class *InterfaceDefinition*(*name: str, port_prefix: str, signals: InterfaceDefinitionSignals*)

Interface described in YAML interface definition file

__init__(*name: str, port_prefix: str, signals: InterfaceDefinitionSignals*)

Schema

alias of *InterfaceDefinition*

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

CONFIG

A **Config** object stores configuration values. A global `topwrap.config.config` object is used throughout the codebase to access topwrap's configuration. This is created by **ConfigManager** that reads config files defined in `topwrap.config.ConfigManager.DEFAULT_SEARCH_PATHS`, with files most local to the project taking precedence.

```
class Config(force_interface_compliance: bool | None = False, repositories:
    ~typing.List[~topwrap.config.RepositoryEntry] | None = <factory>)
```

Global topwrap configuration

```
__init__(force_interface_compliance: bool | None = False, repositories:
    ~typing.List[~topwrap.config.RepositoryEntry] | None = <factory>)
```

Schema

alias of Config

```
class ConfigManager(search_paths: List[PathLike] | 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.

```
__init__(search_paths: List[PathLike] | None = None)
```

DEDUCING INTERFACES

This section describes how inferring interfaces works when using `topwrap parse` with `--iface-deduce`, `--iface` or `--use-yosys` options.

The problem can be described as follows: given a set of signals, infer what interfaces are present in this set and assign signals to appropriate interfaces. Interface names and types (AXI4, AXI Stream, Wishbone, etc.) are, in the general case, not given in advance. Algorithm implemented in `topwrap` works roughly as follows:

1. Split the given signal set into disjoint subsets of signals based on common prefixes in their names
2. For a given subset, try to pair each signal name (as it appears in the RTL) with the name of an interface signal (as it is defined in the specification of a particular interface). This pairing is called “a matching”. Matching with signals from all defined interfaces is tried.
3. For a given subset and matched interface, infer the interface direction (master/slave) based on the direction of some signal in this set.
4. Compute score for each matching, e.g. if signal names contain `cyc`, `stb` and `ack` (and possibly more) it’s likely that this set is a Wishbone interface. Among all interfaces, interface that has the highest matching score is selected.

21.1 Step 1. - splitting ports into subsets

First, all ports of a module are grouped into disjoint subsets. Execution of this step differs based on the options supplied to `topwrap parse`:

- with `--iface` the user supplies `topwrap` with interface names - ports with names starting with a given interface name will be put in the same subset.
- with `--use-yosys` grouping is done by parsing the RTL source with `yosys`, where ports have attributes in the form of `(* interface="interface_name" *)`. Ports with the same `interface_name` will be put in the same subset.
- with `--iface-deduce` grouping is done by computing longest common prefixes among all ports. This is done with the help of a [trie](#) and only allows prefixes that would split the port name on an underscore (e.g. in `under_score` valid prefixes are an empty string, `under` and `under_score`) or a camel-case word boundary (e.g. in `wordBoundary` valid prefixes are an empty string, `word` and `wordBoundary`). As with user-supplied prefixes, ports with names starting with a given prefix will be put in the same subset.

21.2 Step 2. - matching ports with interface signal names

Given a subset of ports from a previous step, this step tries to match a regexp from an interface definition YAML for a given interface signal to one of the port names and returns a collection of pairs: RTL port + interface port. For example, when matching against AXI4, a port named `axi_a_arvalid` should match to an interface port named `ARVALID` in the interface definition YAML.

This operation is performed for all defined interfaces per a given subset of ports so the overall result of this step is a collection of matchings. For most interfaces these matching will be poor - e.g. `axi_a_arvalid` or other AXI4 signals won't match to most Wishbone interface signals, but an interface that a human would usually assign to a given set of signals will have the most signals matched.

21.3 Step 3. - inferring interface direction

This step picks a representative RTL signal from a single signal matching from the previous step and checks its direction against direction of the corresponding interface signal in interface definition YAML - if it's the same then it's a master interface (since the convention in interface description files is to describe signals from the master's perspective), otherwise it's a slave.

21.4 Step 4. - computing interface matching score

This step computes a score for each matching returned by step 2. This score is based on the number of matched/unmatched optional/required signals in each matching.

Not matching some signals in a given group (from step 1.) is heavily penalized to encourage selecting interface that "fits" a given group best. For example, AXI Lite is a subset of AXI4, so a set of signals that should be assigned AXI4 interface could very well fit the description of AXI Lite, but this mechanism discourages selecting such matching in favor of selecting the other.

Not matching some signals of a given interface (from interface description YAML) is also penalized. Inverting the previous example, a set of signals that should be assigned AXI Lite interface could very well fit the description of AXI4, but because it's missing a few AXI4 signals so selecting this matching is discouraged in favor of selecting the other.

21.4.1 Good scoring function

A well-behaving scoring function should satisfy some properties to ensure that the best "fitting" interface is selected. To describe these we introduce the following terminology:

- `>/>=/==` should be read as "must have a greater/greater or equal/equal score than".
- Partial matching means matching where some rtl signals haven't been matched to interface signals, full matching means matching where all have been matched.

Current implementation when used with default config values satisfies these properties:

1. full matching with $N+1$ signals matched (same type) `==` full matching with N signals matched (same type)

2. full matching with N signals matched (same type) $>$ partial matching with N signals matched (same type)
3. partial matching with $N+1$ signals matched (same type) $>$ partial matching with N signals matched (same type)
4. full matching with $N+1$ required, $M+1$ optional signals \geq full matching with $N+1$ optional, M optional signals \geq full matching with N required, $M+1$ optional signals \geq full matching with N required, M optional signals

Properties 2-4 generally ensure that interfaces with more signals matched are favored more over those with less signals matched. Property 1. follows from the current implementation and is not needed in all implementations.

Full details can be found in the implementation itself.

EXAMPLES

Note: Basic usage of examples is explained in the *Getting started* section.

Examples provided with this project should cover from very simple designs to complex fully synthesizable cores. They should be sorted by increasing complexity and number of used features, e.g:

- 101: minimal base design
- 102: introduce user to parameters
- 103: introduce user to slicing
- 104: introduce user to interfaces
- 105: etc.

Developers are encouraged to create/add new examples in the same spirit. Simple examples are used to teach how to use this tool and demonstrate its features. Real-world use cases are also welcome to prove that the implementation is mature enough to handle practical designs.

FUTURE ENHANCEMENTS

23.1 Support for hierarchical block design in Pipeline Manager

Currently topwrap supports creating hierarchical designs only by manually writing the hierarchy in the design description YAML. Supporting such feature in the Pipeline Manager via its subgraphs would be a huge improvement in terms of organizing complex designs.

23.2 Bus management

Another goal we'd like to achieve is to enable users to create full-featured designs with processors by providing proper support for bus management. This should include features such as:

- ability to specify the address of a peripheral device on the bus
- support for the most popular buses or the ones that we use (AXI, wishbone, Tile-link)

This will require writing or creating bus arbiters (round-robin, crossbar) and providing a mechanism for connecting master(s) and slave(s) together. As a result, the user should be able to create complex SoC with Topwrap.

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

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

One of the main features that are supported by Topwrap and Pipeline Manager is exporting and importing user-created design to or from a design description YAML. However, during these conversions, information about the positions of user-added nodes is not preserved. This is cumbersome in the case of complicated designs since the imported nodes cannot be placed in the optimal positions.

Therefore, one of our objectives is to provide a convenient way of creating and restoring user-created designs in Pipeline Manager, so that the user doesn't have to worry about node positions when importing a design to Pipeline Manager.

23.4 Support for parsing SystemVerilog sources

Information about IP cores is stored in *IP core description YAMLS*. These files can be generated automatically from HDL source files - currently Verilog and VHDL are supported. Our goal is to provide the possibility of generating such YAMLS from SystemVerilog too.

23.5 Provide a way to parse HDL sources from the Pipeline Manager level

Another issue related to HDL parsing is that the user has to manually parse HDL sources to obtain the IP core description YAMLS. Then the files need to be provided as command-line parameters when launching the Topwrap Pipeline Manager client application. Therefore, we aim to provide a way of parsing HDL files and including them in the editor directly from the Pipeline Manager level.

23.6 Ability to produce top-level wrappers in VHDL

Topwrap now uses Amaranth to generate top-level design in Verilog. We would also like to add the ability to produce such designs in VHDL.

23.7 Library of open-source cores

Currently user has to supply all of the cores used in the design manually or semi-manually (e.g. through FuseSoC). A repository of open-source cores that could be easily reused in topwrap would improve convenience and allow quickly putting together a design from premade hardware blocks.

23.8 Integrating with other tools

Topwrap can build the design but testing and synthesis rely on the user - they have to automate this process themselves (e.g. with makefiles). Ideally the user should be able to write scripts that integrate tools for synthesis, simulation and co-simulation (e.g. with Renode) with topwrap. Some would come pre-packaged with topwrap (e.g. simulation with verilator, synthesis with vivado). It should also be possible to invoke these from the Pipeline Manager GUI by using its ability to add custom buttons and integrated terminal.

Symbols

__init__() (*Config method*), 46
 __init__() (*ConfigManager method*), 46
 __init__() (*ElaboratableWrapper method*), 40
 __init__() (*FuseSocBuilder method*), 43
 __init__() (*IPConnect method*), 36
 __init__() (*IPWrapper method*), 33
 __init__() (*InterfaceDefinition method*), 45
 __init__() (*Wrapper method*), 32
 __init__() (*WrapperPort method*), 41
 _connect_external_ports() (*IPConnect method*), 36
 _connect_internal_ports() (*IPConnect method*), 36
 _connect_to_external_port() (*IPConnect method*), 36
 _set_interface() (*IPConnect method*), 37
 _set_port() (*IPConnect method*), 37

A

add_component() (*IPConnect method*), 37
 add_dependency() (*FuseSocBuilder method*), 43
 add_external_ip() (*FuseSocBuilder method*), 43
 add_source() (*FuseSocBuilder method*), 43
 add_sources_dir() (*FuseSocBuilder method*), 43

B

build() (*FuseSocBuilder method*), 44

C

Config (*class in topwrap.config*), 46
 ConfigManager (*class in topwrap.config*), 46
 connect_interfaces() (*IPConnect method*), 37
 connect_ports() (*IPConnect method*), 38

E

ElaboratableWrapper (*class in topwrap.elaboratable_wrapper*), 40

F

FuseSocBuilder (*class in topwrap.fuse_helper*), 43

G

get_interface_by_name() (*in module topwrap.interface*), 45
 get_port_by_name() (*Wrapper method*), 32
 get_ports() (*ElaboratableWrapper method*), 40
 get_ports() (*IPConnect method*), 38
 get_ports() (*IPWrapper method*), 34
 get_ports() (*Wrapper method*), 32
 get_ports_hier() (*ElaboratableWrapper method*), 40
 get_ports_of_interface() (*Wrapper method*), 32

I

InterfaceDefinition (*class in topwrap.interface*), 45
 IPConnect (*class in topwrap.ipconnect*), 36
 IPWrapper (*class in topwrap.ipwrapper*), 33

L

like() (*WrapperPort static method*), 41

M

make_connections() (*IPConnect method*), 38
 make_interconnect_connections() (*IPConnect method*), 38

S

Schema (*Config attribute*), 46
 Schema (*InterfaceDefinition attribute*), 45
 set_constant() (*IPConnect method*), 39

V

validate_inout_connections() (*IPConnect method*), 39

W

Wrapper (*class in topwrap.wrapper*), [32](#)

WrapperPort (*class in topwrap.amaranth_helpers*), [41](#)