Antmicro

Topwrap

2024-11-04

DOCUMENTATION

CHAPTER

ONE

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:

- User-friendly GUI
- Parsing HDL design files with automatic recognition of common interfaces
- Simple YAML-based description for command-line use
- Capability to create custom libraries for reuse across projects

CHAPTER

TWO

INSTALLATION

1. Install required system packages:

   Debian:

   ```
   apt install -y git g++ make python3 python3-pip yosys npm
   ```

   Arch:

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

   Fedora:

   ```
   dnf install git g++ make python3 python3-devel yosys npm
   ```

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

   ```
   python3 -m venv venv
   source venv/bin/activate
   pip install .
   ```

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

```
apt install -y antlr4 libantlr4-runtime-dev
pip install ".[topwrap-parse]"
```

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

```
pacman -Syu antlr4 antlr4-runtime
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:

```
dnf install antlr4 antlr4-cpp-runtime-devel
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]"
```

If you want to contribute to the project please see the *Developer's setup guide*.

CHAPTER

THREE

GETTING STARTED

Goal of this chapter is to show step by step how to create simple design with topwrap.

All necessary files to follow this guide are in `examples/getting_started_demo` directory.

Important

This is just an example, if you haven't installed the topwrap yet go to the *Installation chapter* and make sure to install additional dependencies for `topwrap parse`.

3.1 Design overview

The design we are going to create is visually represented 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 are connected to each other and to some external metanodes.

3.2 Parsing verilog files

First step in creating own designs is to parse verilog files into ip core description yamls that are understood by topwrap.

In the `verilogs` directory you can find two verilog files which describe `simple_core_1` and `simple_core_2`.

To generate ip core descriptions from these verilogs run:

```
topwrap parse verilogs/{simple_core_1.v,simple_core_2.v}
```

Topwrap will generate two files `gen_simple_core_1.yaml` and `gen_simple_core_2.yaml` that represent corresponding verilogs.

3.3 Building design with topwrap

3.3.1 Creating the design

Generated ip core yamls can be loaded into GUI.

1. Build and run gui server

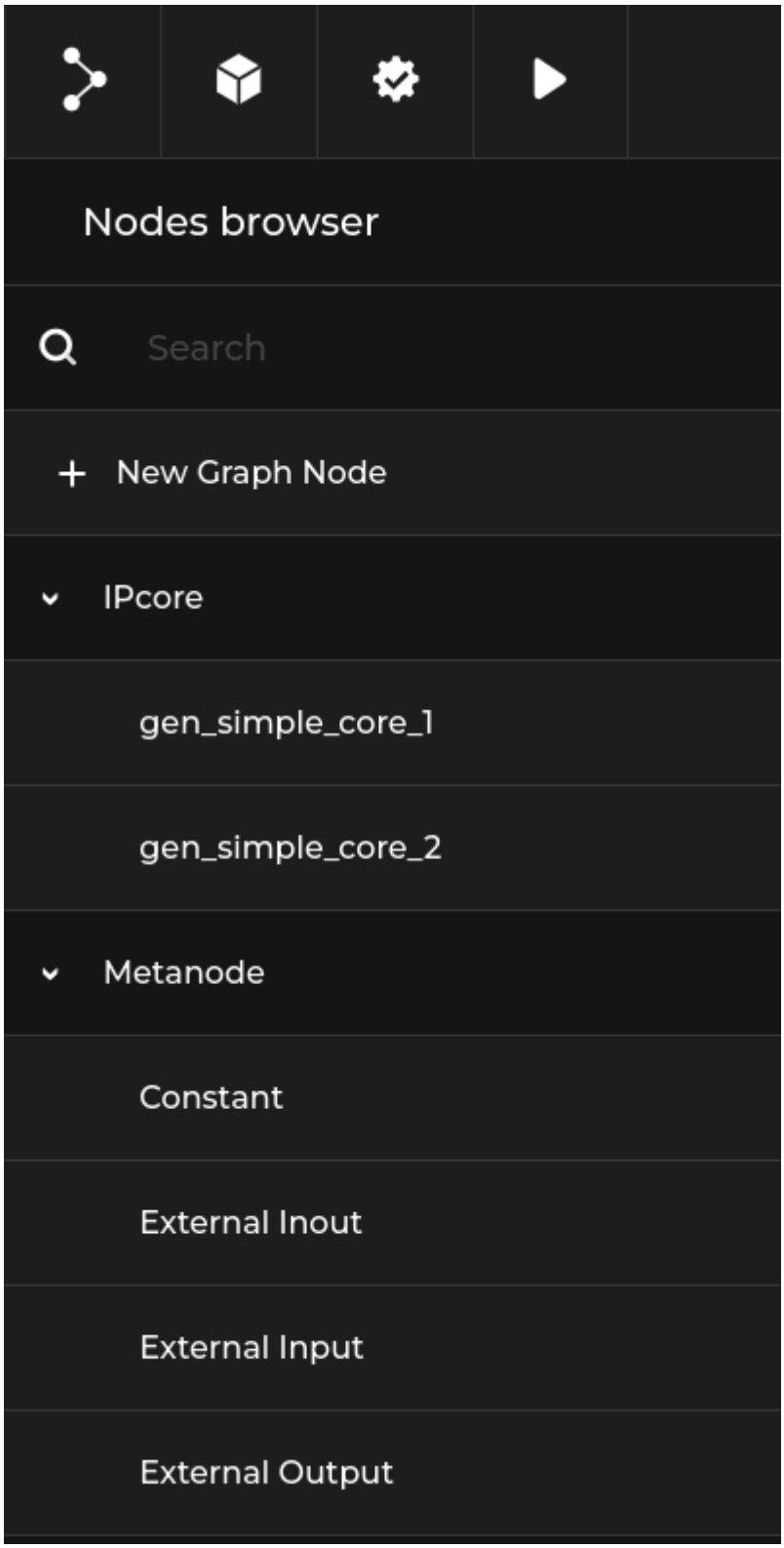
```
topwrap kpm_build_server && topwrap kpm_run_server &
```

2. Run gui client with the generated ip core yamls

```
topwrap kpm_client gen_simple_core_1.yaml gen_simple_core_2.yaml
```

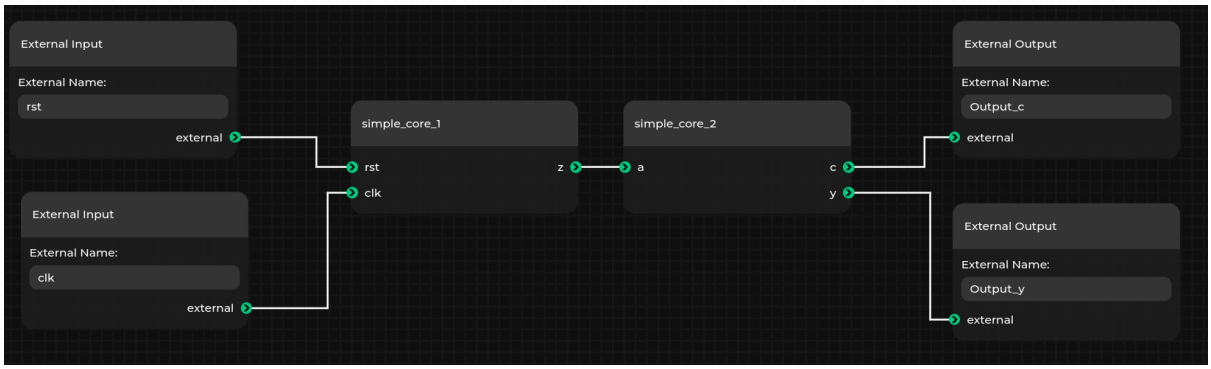
Now when you connect to http://127.0.0.1:5000 there should be kpm gui.

Loaded ip cores can be found under IPcore section:



With these IPcores and default metanodes you can easily create designs by dragging cores and connecting them.

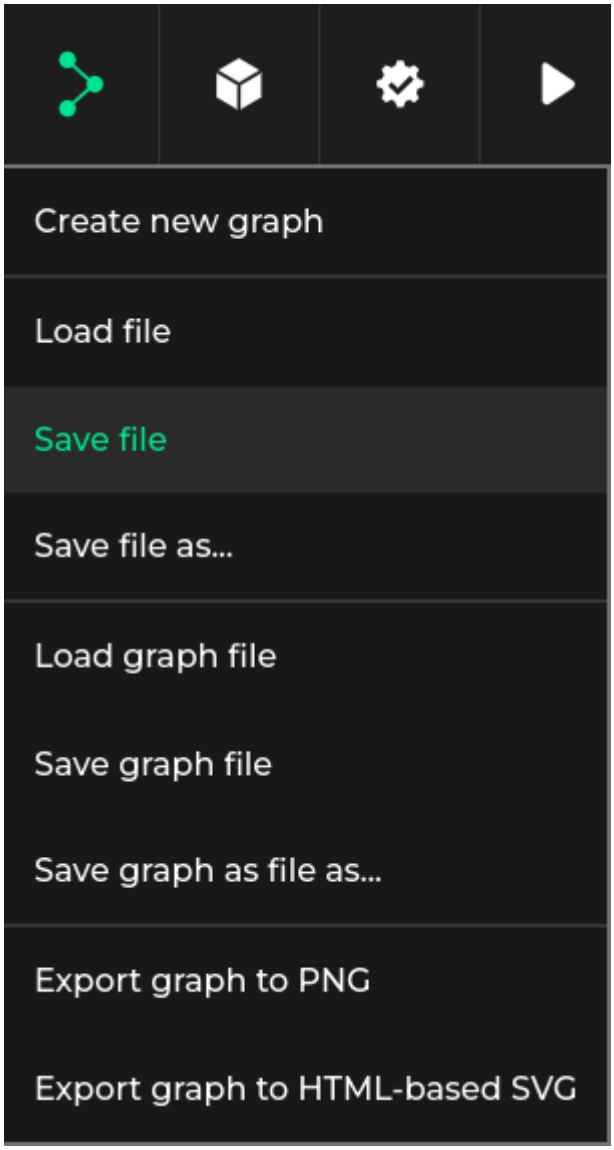
Let's make the design from the demo that was shown at the beginning of this guide.



Note: You can change name of node by right clicking on it and selecting `rename`.

You can save the project to the *Design Description* format, which is used by topwrap to represent the created design.

To do this select the graph button and select `Save file`.



Note: The difference between `Save file` and `Save graph file` lays in which format will be used for saving.

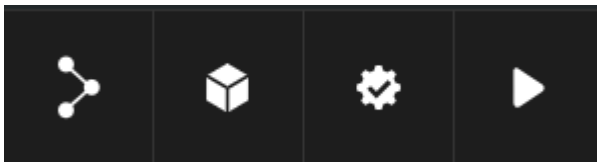
`Save file` will save the design description in yaml format which topwrap uses.

`Save graph file` will save the design in graph json format which kpm uses. You should only choose this one if you have some specific custom layout of nodes in design and you want to save it.

3.3.2 Generating verilog

You can generate verilog from design created in previous section.

If you have the example running as described in previous section then on the top bar you should see these 4 buttons:



First one is for loading or saving designs. Second is for toggling node browser. Third one is for validating the design. And the fourth is for building the design which will generate the verilog file in `/build` directory of the current example and run the design.

3.4 Appendix: Command-line flow

3.4.1 Creating the design

Manual creation of designs requires familiarity with *Design Description* format.

First let's include all the ip core files we will need in the `ips` section.

```
ips:
  simple_core_1:
    file: gen_simple_core_1.yaml
  simple_core_2:
    file: gen_simple_core_2.yaml
```

Notice that here we also declare how to node will be named. Ip core `gen_simple_core_1.yaml` will be named `simple_core_1` in gui. Now we can start creating the design under the `design` section. Our design doesn't have any parameters so we can skip this part and go straight into `ports` section. There we define connections between ip cores. In demo example there is only one connection - between `gen_simple_core_1` and `gen_simple_core_2`. In our design it will look like below:

```
design:
  ports:
    simple_core_2:
```

(continues on next page)

(continued from previous page)

```
      a:
      - simple_core_1
      - z
```

Notice that we connect `input` to `output`. All left to do are external connections to metanodes. We declare them 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
```

Final design:

```
ips:
  simple_core_1:
    file: gen_simple_core_1.yaml
  simple_core_2:
    file: gen_simple_core_2.yaml
design:
  ports:
    simple_core_1:
      clk: clk
      rst: rst
    simple_core_2:
      a:
      - simple_core_1
      - z
      c: Output_c
      y: Output_y
external:
  ports:
```

(continues on next page)

(continued from previous page)

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

3.4.2 Generating verilog

Info

Topwrap uses Amaranth for generating verilog top file.

To generate top file use `topwrap build` and provide the design.

Ensure you are in the `examples/getting_started_demo` directory and run:

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

Where the `{design_name.yaml}` is the design saved at the end of previous section. The generated verilog file can be found in `/build` directory.

Notice that you will get warning:

```
WARNING:root:You did not specify part number. 'None' will be used and thus your␣
↪implementation may fail.
```

It's because we didn't specify any part with `--part` flag since it's just a dummy example that is not for any specific FPGA chip. For building your designs we recommend specifying the `--part`.

CHAPTER

FOUR

EXAMPLE PROJECTS

These example projects show some useful ways in which Topwrap can be used by the end-user.

Information about embedded GUI

This section extensively uses an embedded version of Topwrap's GUI, *Kenning Pipeline Manager*, to visualize the design of all the examples.

You can use it to freely explore the entire design, add new blocks, connections, nodes and hierarchies. You cannot however use features that require direct connection with the Topwrap's backend. These features include, among others:

- Saving and loading data from/to `.yaml` files
- Verifying designs
- Building designs

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

4.1 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.

There is often a need to pass constant values to input ports of some IP Cores. This example shows how easy expressing that is in the GUI and correspondingly, in the design description file (`project.yml`).

Tip: You can find the constant node blueprint in the Nodes browser under the `Metanode` section.

4.1.1 Usage

Enter the example's directory

```
cd examples/constant
```

Generate HDL source

```
make generate
```

4.2 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 the way it's represented in the GUI.

> **Tip:** An inout port is denoted 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:

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

4.2.1 Usage

Enter the example's directory

```
cd examples/inout
```

Install required dependencies

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

Generate bitstream for Zynq

```
make
```

Generate HDL sources without implementation

```
make generate
```

4.3 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 a structure of a user repository containing prepackaged IP cores with sources and custom interface definitions, the design file and the config file. Elements of the `repo` directory can be easily reused in different designs as long as you point to it either in the config file or in the CLI.

See also:

For more information about user repositories see *Packaging multiple files*.

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

```
topwrap kpm_client -d project.yml
```

4.3.1 Usage

Build and run Pipeline Manager server

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

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

```
python -m topwrap kpm_client -d project.yml
```

Connect to the web GUI frontend in your browser on `http://127.0.0.1:5000`.

Expected result

Topwrap will load two cores from the `cores` directory that use an interface from the `interfaces` directory.

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

4.4 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. It includes a hierarchy containing some IP cores and other nested hierarchies.

Check out `project.yml` to learn how does the above design translate to a *design description file*

See also:

For more information about hierarchies see *hierarchies docs*.

Tip: Hierarchies are represented in the GUI by nodes with a green header.

You can display their inner designs by clicking the `Edit subgraph` option from the right click menu.

To exit from the hierarchy subgraph, find the back arrow button in the top left.

To add a new hierarchy node use the `New Graph Node` option in the node browser!

4.4.1 Usage

This example contains user repo (`repo` directory) and a configuration file for topwrap (`topwrap.yaml`) so it can be loaded by running

```
python -m topwrap kpm_client -d project.yml
```

in this example's directory.

4.5 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 being connected to the ZYNQ Processing System. The Core uses AXILite interface, so a proper `AXI -> AXILite` converter is needed. You can access its registers starting from address `0x4000000`

(that's the base address of `AXI_GP0` on ZYNQ). The generated signal can be used in FPGA or connected to a physical port on a board.

Note: 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.

4.5.1 Usage

Enter the example's directory

```
cd examples/pwm
```

Install required dependencies

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

In order to be able to generate a bitstream you also need to install Vivado and add it to your `PATH`.

Generate bitstream for Zynq

```
make
```

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

```
make generate
```

4.6 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 on how to use Topwrap to build a complex, synthesizable design.

4.6.1 Usage

Enter the example's directory

```
cd examples/hdmi
```

Install required dependencies

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

In order to be able to generate a bitstream you also need to install Vivado and add it to your PATH.

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
```

4.7 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 is an example on how to use Topwrap to build a synthesizable SoC design. The SoC contains a VexRiscv core, data and instruction memory, UART and interconnect that ties all components together.

4.7.1 Usage

Enter the example's directory

```
cd examples/soc
```

Install required dependencies

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

To run the simulation you also need:

- verilator

To create and load bitstream you also need:

- vivado (preferably version 2020.2)
- openFPGALoader (this branch)

Generate HDL sources

```
make generate
```

Build and run simulation

```
make sim
```

Expected waveform generated by the simulation is shown in `expected-waveform.svg`.

Generate bitstream

```
make bitstream
```

CHAPTER

FIVE

CREATING A DESIGN

5.1 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 instance names in the
  # design yaml and IP cores description yamls
  {ip_instance_name}:
    file: {path_to_ip_description}
  ...

design:
  name: {design_name} # optional name of the toplevel
  hierarchies:
      # see "Hierarchies" below 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.

5.1.1 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.

5.1.2 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}
        ...
    {hierarchy_name_2}:
      ...
```

More complex hierarchy example can be found in examples/hierarchy.

5.2 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`. The module name of an IP should be placed in global `name` node, it should be consistent with how it is defined in HDL file. Here's an example description of ports of Clock Crossing IP:

```yaml
# file: clock_crossing.yaml
name: cdc_flag
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:

```yaml
#file: axis_width_converter.yaml
name: axis_width_converter
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
```

(continues on next page)

(continued from previous page)

```
                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
    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.

5.2.1 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
```

5.2.2 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:
```

(continues on next page)

(continued from previous page)

```
    s_axis:
        type: AXI4Stream
        mode: slave
        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]
```

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

5.2.3 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]
```

5.3 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
```

(continues on next page)

(continued from previous page)

```
    optional:
        out:
            TID: tid
            TDEST: tdest
            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 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.

CHAPTER

SIX

USING TOPWRAP

6.1 GUI

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

6.1.1 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:

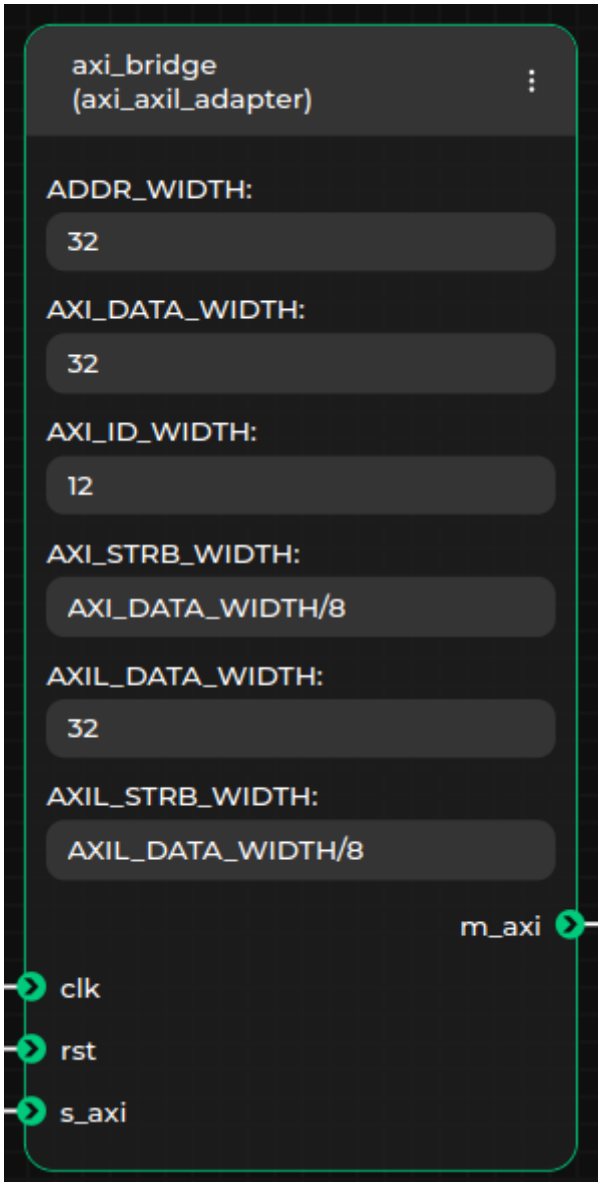
   ```
   python -m topwrap kpm_client -h 127.0.0.1 -p 9000 \
       topwrap/ips/axi/axi_axil_adapter.yaml \
       examples/pwm/ipcores/{litex_pwm.yml,ps7.yaml} -d examples/pwm/project.yml
   ```

3. Create block design in Pipeline Manager

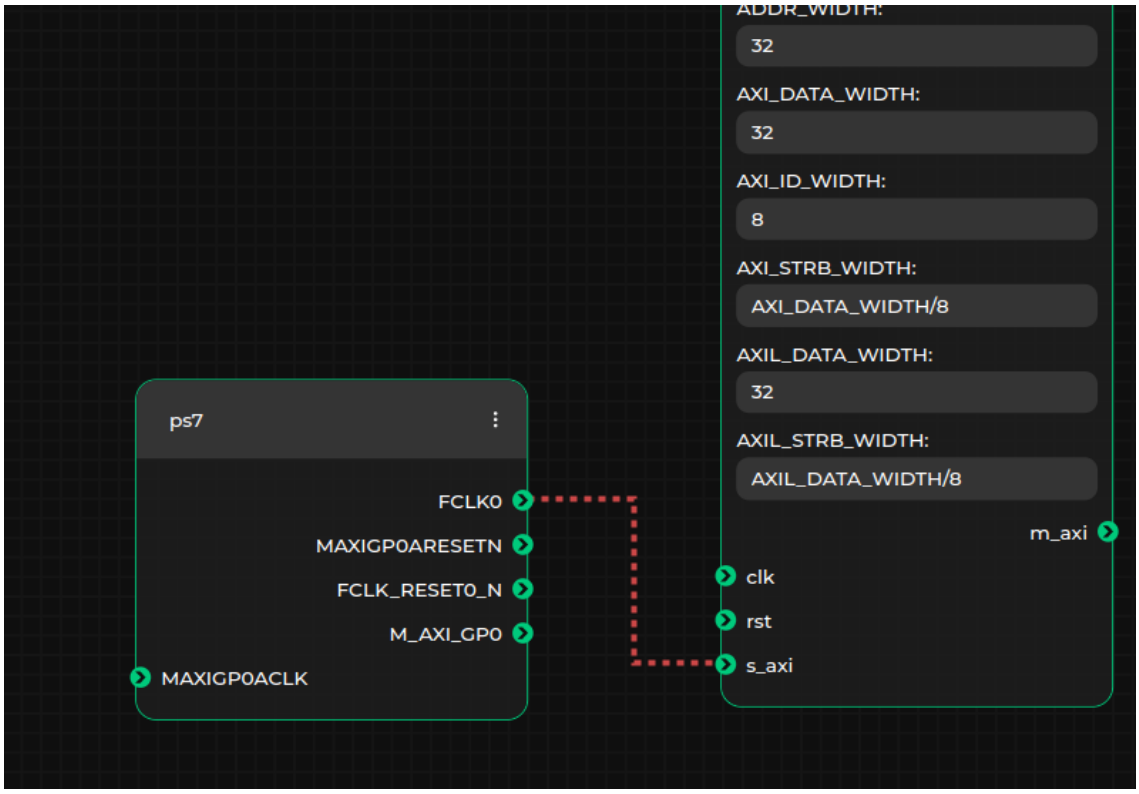
   Upon successful connection to a Pipeline Manager server, Topwrap will generate and send to the server a specification describing the structure of previously selected IP cores. If the

`-d` option was used a design will be shown in gui. From there you can create or modify designs by:

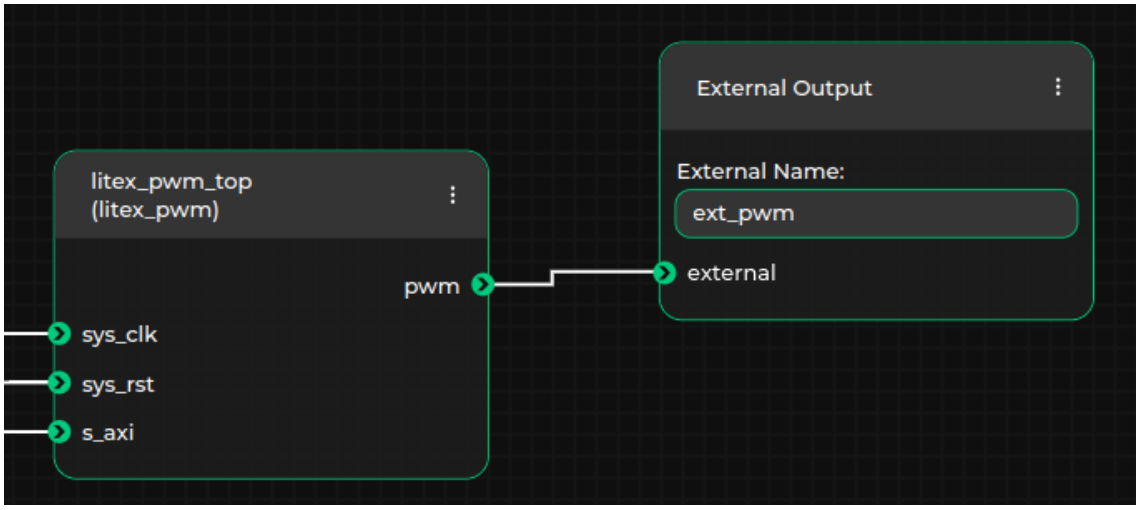
- adding IP core instances to the block design. Each Pipeline Manager's node has `delete` and `rename` options, which make it possible to remove the selected node and change its name respectively. This means that you can create multiple instances of the same IP core.
- adjusting IP cores' parameters 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:

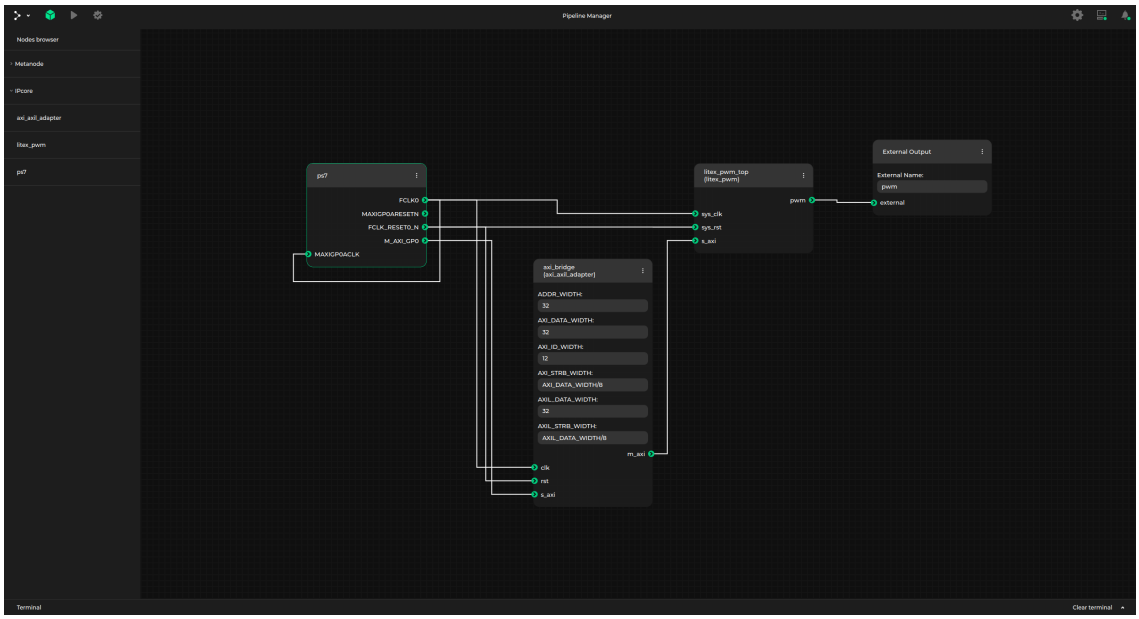


- specifying external ports or interfaces in the top module. This can be done by adding `External Input`, `External Output` or `External Inout` metanodes and creating connections between them and chosen ports or interfaces. Note that you should adjust the name of the external port or interface in a textbox 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:



6.1.2 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

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.

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.

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 be 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.

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.

6.2 CLI

Topwrap has a couple CLI only functions that expand gui functionality.

6.2.1 Generating IP core description YAMLs

You can use Topwrap to generate ip core description yamls from HDL sources to use them in your `project.yml`. To learn how project and core yamls work check *design description* and *ip description*

```
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
```

6.2.2 Building design

Topwrap can build a synthesizable design from source files connected in a way described by a design file, to do this run:

```
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 'xc7z020clg400-3
↪'
```

CHAPTER

SEVEN

PACKAGING MULTIPLE FILES

Repositories allow for easy packaging and loading multiple IP-cores and custom interfaces.

You can specify repositories to be loaded each time topwrap is ran by listing them in a configuration file that should 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 directories: `cores` and `interfaces`.

Inside `cores` each core has it's own directory with it's description file and `srcs` where the verilog/VHDL files are stored.

The `interfaces` directory is optional, and contains interface description files.

Example User Repo can be found in examples/user_repository.

CHAPTER

EIGHT

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

8.1 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}
      ...
    ...
```

8.2 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.

8.2.1 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`

8.3 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

CHAPTER

NINE

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).

CHAPTER

TEN

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.

CHAPTER

ELEVEN

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

CHAPTER

TWELVE

TESTS

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`

12.3 Updating kpm test data

All kpm data from examples can be generated using nox. This is useful when changing topwrap functionality related to kpm in order to avoid manually changing every example's test data. You can either update only one part of examples data like specification or update everything (dataflows, specifications, designs).

To update everything run:

```
nox -s update_test_data
```

To update only specifications run:

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

Possible options for `update_test_data` session:

- `specification` - updates specifications
- `dataflow` - updates dataflows
- `design` - updates designs

CHAPTER

THIRTEEN

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.

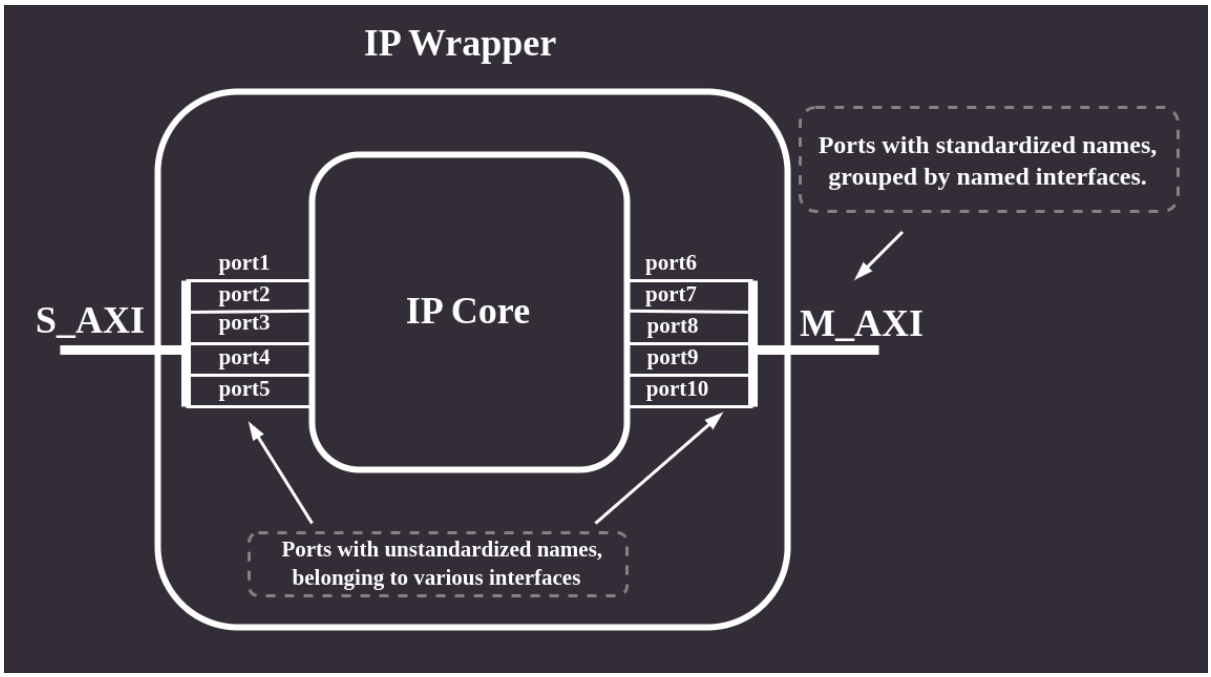
CHAPTER

FOURTEEN

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 `WrapperPort`s.



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*]

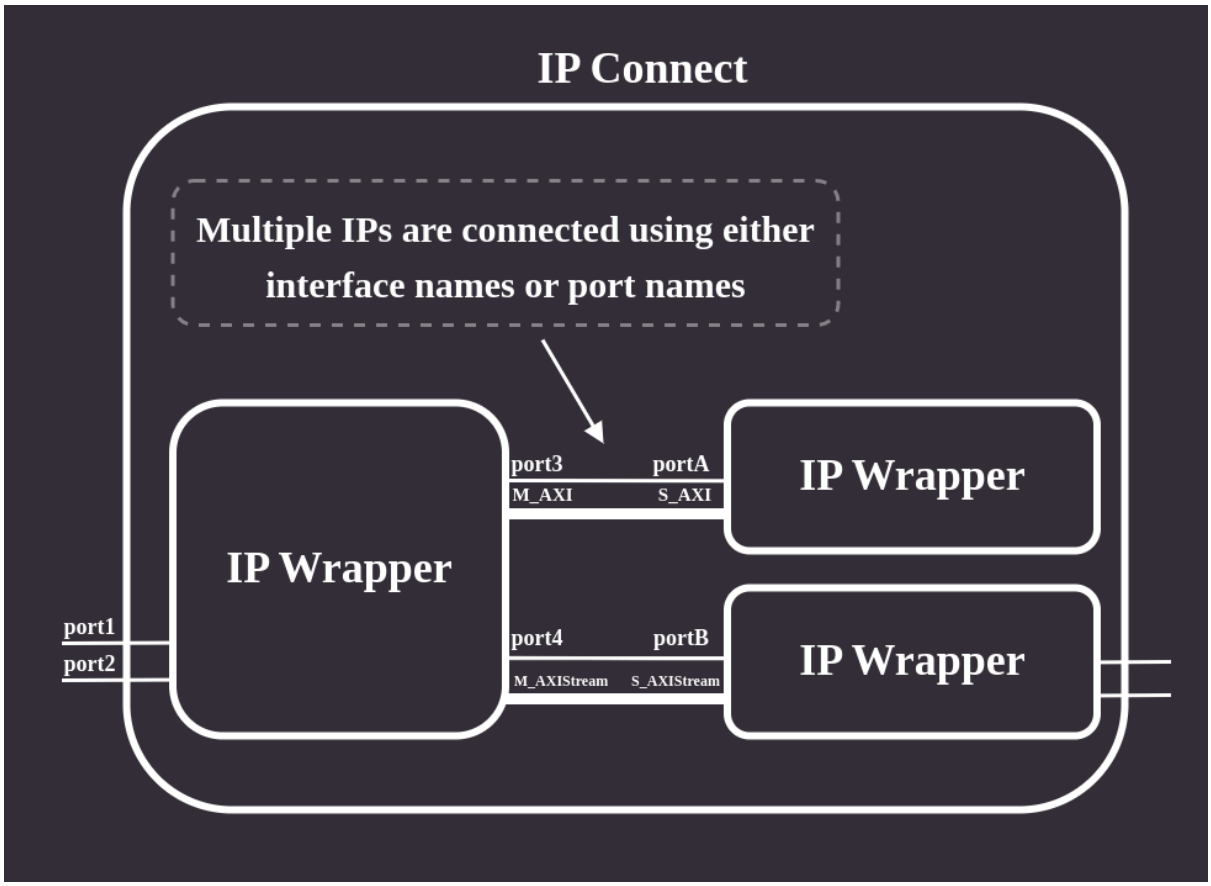
Return a list of all ports that belong to this IP.

CHAPTER

FIFTEEN

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 `IPConnect`s 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

CHAPTER

SIXTEEN

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.

CHAPTER

SEVENTEEN

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

CHAPTER

EIGHTEEN

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

CHAPTER

NINETEEN

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: ~topwrap.interface.InterfaceDefinitionSignals = <factory>*)

Interface described in YAML interface definition file

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

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

CHAPTER

TWENTY

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*)

CHAPTER

TWENTYONE

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 >= full matching with N+1 optional, M optional signals >= full matching with N required, M+1 optional signals >= 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.

CHAPTER

TWENTYTWO

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.

CHAPTER

TWENTYTHREE

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.

CHAPTER

TWENTYFOUR

USING KPM IFRAMES INSIDE DOCS

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

24.1 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 iframe>
:alt: <a custom alternative text used in the PDF documentation instead of the␣
↪default one>
```
```

`URI` can represent either a local file from sources that gets copied into the build directory, or a remote resource.

All parameters of this directive are optional.

24.2 Tests

24.2.1 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.

24.2.2 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.

24.2.3 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.

24.2.4 Use 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

INDEX

Symbols

A

B

C

E

F

G

I

L

M

S

V

W