

Antmicro

Omnibus

2026-11-13

CONTENTS

1 Overview **1**
- 1.1 Introduction . . . 1
- 1.2 Simple Example . . . 1

2 Installation **2**
- 2.1 System Dependencies . . . 2
- 2.2 Package . . . 2

3 User Guide **3**
- 3.1 Specification . . . 3
- 3.2 Visualization . . . 4
- 3.3 Generation . . . 4
- 3.4 Compilation with Verilator . . . 5

4 Examples **6**
- 4.1 Two crossbars with memory and APB devices . . . 6
- 4.2 ASoCa . . . 8
- 4.3 Butterfly . . . 10

5 Interconnect Specification **17**
- 5.1 YAML Syntax Specification . . . 17

6 Tool Organization **23**
- 6.1 Overview . . . 23
- 6.2 Frontends . . . 23
- 6.3 Internal Processing . . . 23
- 6.4 Backends . . . 24

7 Simulation Tests **26**
- 7.1 Quick start . . . 26
- 7.2 Layout . . . 27
- 7.3 How a test run works . . . 27
- 7.4 The DUT's interface . . . 30
- 7.5 Inside the test bench . . . 31
- 7.6 Tiers . . . 33
- 7.7 Known failures and guards . . . 33
- 7.8 Working with the suite . . . 34
- 7.9 Notes for reviewers . . . 34
- 7.10 Converter tests . . . 34

8 Planned Extensions **36**
8.1 General 36
8.2 Frontend 36
8.3 Backend 37
8.4 Analysis 37

CHAPTER

ONE

OVERVIEW

1.1 Introduction

Omnibus lets you generate interconnects from specifications. It populates the interconnect with all necessary components, like CDC or protocol converters. Currently, it is used as supplementary to Topwrap - it generates all wrappers and creates a Topwrap project file. You can view the generated interconnect with Graphviz. In the future, Omnibus will support a variety of backends, including direct Verilog generation.

1.2 Simple Example

Below is a simple example of an interconnect with 5 ports:

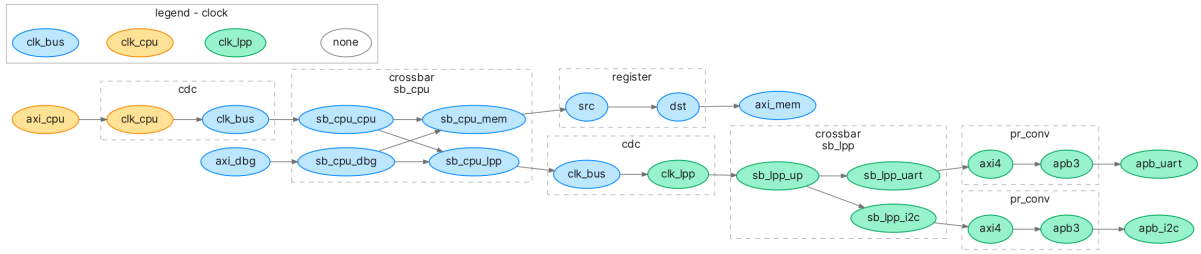
- CPU
- debug
- memory
- uart
- i2c

and two crossbars:

- cpu/debug to memory/io
- io

Omnibus automatically inserts clock and protocol converters between ports. The output is colored according to the clock domain of each port.

For more examples, see *Examples*.



CHAPTER

TWO

INSTALLATION

2.1 System Dependencies

Omnibus always depends on `python` and uv.
The Graphviz (graph visualization) backend also uses `graphviz`.

2.1.1 Debian based systems

```
apt install python3 graphviz
```

2.2 Package

Omnibus is installed with uv, either as a command line tool or into a virtual environment.

2.2.1 Command line tool

```
git clone --recursive https://github.com/antmicro/omnibus
uv tool install ./omnibus/
```

2.2.2 Virtual environment

```
git clone --recursive https://github.com/antmicro/omnibus
uv venv
uv pip install ./omnibus/
```

CHAPTER

THREE

USER GUIDE

Before running any commands, check *installation*.

3.1 Specification

You describe the interconnect to generate in a `yaml` file, according to the *specification*. There are no requirements on the file location or name.

A very simple specification of interconnect:

```yaml
desc_version: 1.0

name: simple
version: 1.0

default:
  data_width: 64
  clock: clk
  frequency: 10MHz
  protocol: AXI4

ports:
  pin:
    role: "subordinate"
  pout1:
    role: "manager"
    protocol: AXI4Lite
  pout2:
    role: "manager"

address_map:
  pout1:
    to: pout1
    base: 0x0000_0000
    size: 0x8000_0000
  pout2:
    to: pout2
    base: 0x8000_0000
    size: 0x8000_0000
```

The file can also be found in examples/simple.yml.

Most of the fields are automatically inferred by Omnibus or are taken from the `default`, so you don't need to specify them explicitly. The file specifies an interconnect with one input port and two output ports, where the lower half of address leads to the first output, and the upper half to the second output.

The ports are described from the perspective of the interconnect, so connect the manager device to the subordinate port.

The interconnect has the data width of 64, automatically generated clock `clk` with frequency of 10MHz assigned to all ports and components. The interconnect mostly uses AXI4 protocol, but `pout1` uses AXI4Lite, Omnibus notices that and inserts protocol converters.

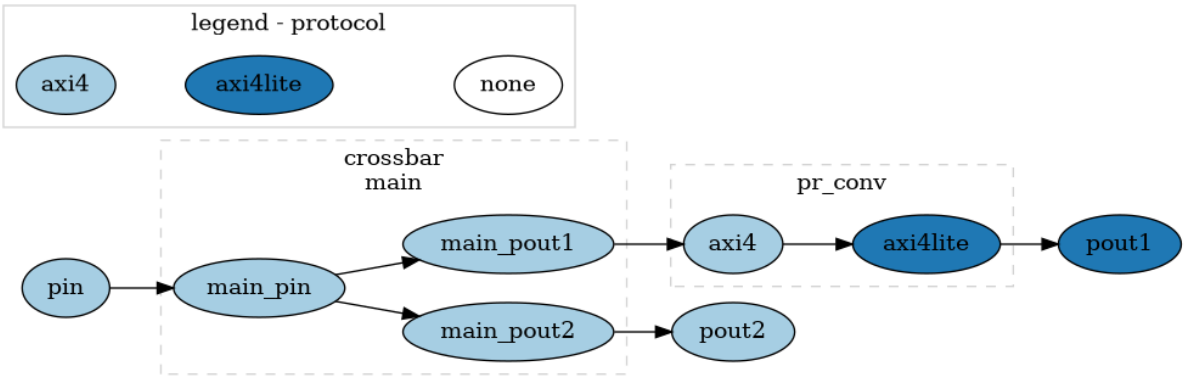
Currently, you can use Omnibus in two ways - visualization and generation.

3.2 Visualization

By running:

```
omnibus examples/simple.yml -g --colorby protocol
```

The window opens displaying the following interconnect graph:



The colors are assigned based on the protocol assigned to each port. Notice that `pr_conv` (protocol converter) has two ports, each of them uses a different protocol. This visualization is very useful for debugging problems with interconnect structure as it abstracts implementation details.

3.3 Generation

Omnibus generates the final interconnect with Topwrap.

Run Omnibus with `--verilog`:

```
omnibus examples/simple.yml --verilog
```

This generates the necessary files in `out/`, then runs Topwrap to parse them (`topwrap repo parse`) and generate the top level (`topwrap generate --verilog --flatten-interfaces`).

You'll find the interconnect in `build/src/`; use `-b` to choose a different build directory. It contains instantiation of all used components, in this case that is just crossbar.

3.4 Compilation with Verilator

To fully use this file, clone its dependencies (these and only these dependencies are needed, regardless of interconnect complexity).

```
git clone https://github.com/pulp-platform/axi
git clone https://github.com/pulp-platform/common_cells
git clone https://github.com/pulp-platform/tech_cells_generic
```

And then verify the output with `verilator`:

```
verilator --lint-only -Wall \
    -Iaxi/include/ -Icommon_cells/include/ \
    out/verilator/general_config.vlt \
    out/verilator/pulp_platform_config.vlt \
    -f out/verilator/pulp_platform_filelist.vc \
    out/src/*sv build/src/*sv  \
    --top-module simple
```

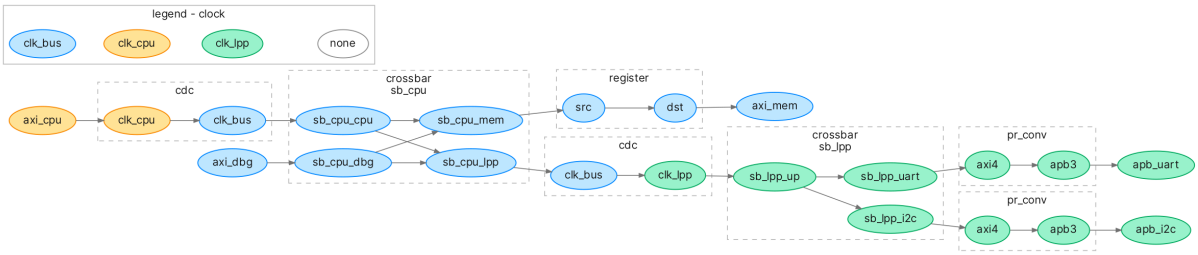
CHAPTER

FOUR

EXAMPLES

4.1 Two crossbars with memory and APB devices

This example demonstrates the usage of Omnibus to generate an interconnect with two crossbars. End ports are assigned to two different interfaces and three different clocks. Omnibus automatically generates appropriate clock domain crossings and protocol conversions.



4.1.1 Specification

```
desc_version: 1.0

name: MyInterconnect
version: 0.85a

default:
    data_width: 32
    protocol: AXI4
    clock: clk_bus

clocks:
  clk_bus:
    frequency: 100e6
  clk_cpu:
    frequency: 1GHz
  clk_lpp:
    frequency: 25M

ports:
  axi_cpu:
    clock: clk_cpu
    role:  "subordinate"
```

(continues on next page)

(continued from previous page)

```
  axi_dbg:
    role:  "subordinate"
  axi_mem:
    role:  "manager"
  apb_uart:
    protocol:  APB
    clock: clk_lpp
    role:  "manager"
  apb_i2c:
    protocol:  APB
    clock: clk_lpp
    role:  "manager"

crossbars:
  sb_cpu:
    ports:
      cpu:
        role: "subordinate"
      dbg:
        role: "subordinate"
      mem:
        role: "manager"
      lpp:
        role: "manager"
  sb_lpp:
    clock: clk_lpp
    ports:
      up:
        role: "subordinate"
      uart:
        role: "manager"
      i2c:
        role: "manager"

connections:
  "axi_cpu -> sb_cpu.cpu":
  "axi_dbg -> sb_cpu.dbg":
  "sb_cpu.mem -> axi_mem":
    "registers": 1 # Insert a skid buffer to help timing closure
  "sb_cpu.lpp -> sb_lpp.up":
  "sb_lpp.uart -> apb_uart":
  "sb_lpp.i2c -> apb_i2c":

address_map:
  "memory": # All ports can access axi_mem
    to:    axi_mem
    base:  0x00000000
    size:  0x20000000
  "uart":  # Only CPU can access UART
```

(continues on next page)

(continued from previous page)

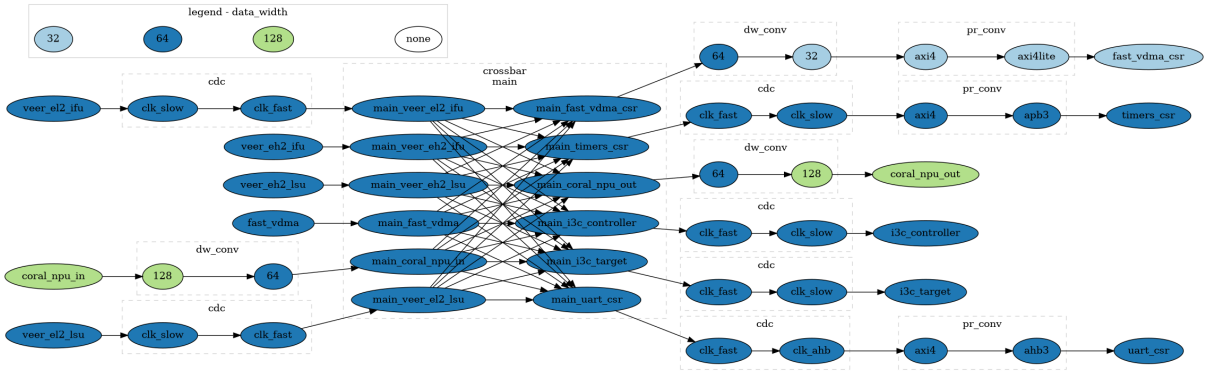
```
    from:  axi_cpu
    to:    apb_uart
    base:  0x40000000
    size:  0x00001000
  "i2c":   # Explicitly axi_cpu and axi_dbg can access apb_i2c
    from:  ["axi_cpu", "axi_dbg"]
    to:    apb_i2c
    base:  0x40001000
    size:  0x00001000
```

4.1.2 Command

```
omnibus examples/spec.yml -g --colorby clock
```

4.2 ASoCa

This example demonstrates the usage of Omnibus to generate ASoCa's interconnect. It does not have an explicit crossbar specified, but instead automatically generates an all-to-all one.



4.2.1 Specification

```
desc_version: 1.0

name: ASoCaInt
version: 0

default:
  clock:
    clk_fast
  protocol:
    axi4
  data_width: 64

clocks:
  clk_slow:
    frequency: 300MHz
  clk_fast:
    frequency: 800MHz
```

(continues on next page)

(continued from previous page)

```
  clk_ahb:
    frequency: 100MHz

ports:
  veer_el2_ifu:
    role: "subordinate"
    clock: clk_slow
  veer_el2_lsu:
    role: "subordinate"
    clock: clk_slow
  veer_eh2_ifu:
    role: "subordinate"
  veer_eh2_lsu:
    role: "subordinate"
  fast_vdma:
    role: "subordinate"
  fast_vdma_csr:
    protocol: axi4 lite
    role: "manager"
    data_width: 32
  timers_csr:
    protocol: apb3
    role: "manager"
    clock: clk_slow
  coral_npu_out:
    role: "manager"
    data_width: 128
  coral_npu_in:
    role: "subordinate"
    data_width: 128
  i3c_controller:
    role: "manager"
    clock: clk_slow
  i3c_target:
    role: "manager"
    clock: clk_slow
  uart_csr:
    protocol: ahb3
    role: "manager"
    data_width: 64
    clock: clk_ahb

address_map:
  "i3c_target_csr":
    to: i3c_target
    base: 0x4000_0000
    size: 0x0000_FFFF
  "i3c_controller_csr":
    to: i3c_controller
```

(continues on next page)

(continued from previous page)

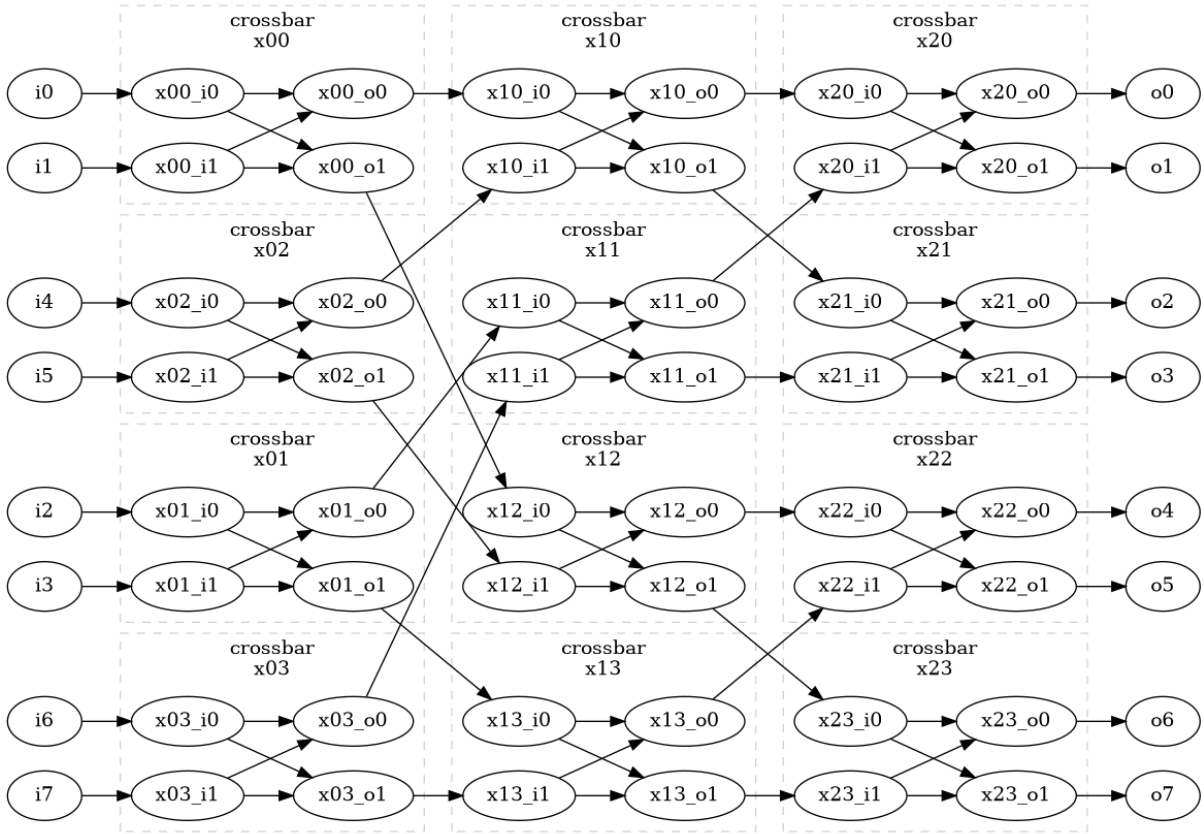
```
    base: 0x4001_0000
    size: 0x0000_FFFF
  "uart_csr":
    to: uart_csr
    base: 0x4002_0000
    size: 0x0000_FFFF
  "timers_csr":
    to: timers_csr
    base: 0x4003_0000
    size: 0x0000_FFFF
  "fastvdma_csr":
    to: fast_vdma_csr
    base: 0x5000_0000
    size: 0x0000_FFFF
  "coral_npu_out":
    to: coral_npu_out
    base: 0x6000_0000
    size: 0x0FFF_FFFF
```

4.2.2 Command

```
omnibus examples/asoca_implicit_crossbar.yml -g --colorby data_width
```

4.3 Butterfly

This example demonstrates a more complex network of crossbars - a butterfly configuration.



4.3.1 Specification

```yaml
desc_version: 1.0

name: butterfly

global:
  addr_width: 16

default:
  protocol: AXI4
  clock: clk
  data_width: 32

ports:
  i0:
    role:  "subordinate"
  i1:
    role:  "subordinate"
  i2:
    role:  "subordinate"
  i3:
    role:  "subordinate"
  i4:
    role:  "subordinate"
  i5:
```

(continues on next page)

(continued from previous page)

```
    role:  "subordinate"
  i6:
    role:  "subordinate"
  i7:
    role:  "subordinate"
  o0:
    role:  "manager"
  o1:
    role:  "manager"
  o2:
    role:  "manager"
  o3:
    role:  "manager"
  o4:
    role:  "manager"
  o5:
    role:  "manager"
  o6:
    role:  "manager"
  o7:
    role:  "manager"

crossbars:
  x00:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x01:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x02:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
```

(continues on next page)

(continued from previous page)

```
    o0:
      role: "manager"
    o1:
      role: "manager"
x03:
  ports:
    i0:
      role: "subordinate"
    i1:
      role: "subordinate"
    o0:
      role: "manager"
    o1:
      role: "manager"
x10:
  ports:
    i0:
      role: "subordinate"
    i1:
      role: "subordinate"
    o0:
      role: "manager"
    o1:
      role: "manager"
x11:
  ports:
    i0:
      role: "subordinate"
    i1:
      role: "subordinate"
    o0:
      role: "manager"
    o1:
      role: "manager"
x12:
  ports:
    i0:
      role: "subordinate"
    i1:
      role: "subordinate"
    o0:
      role: "manager"
    o1:
      role: "manager"
x13:
  ports:
    i0:
      role: "subordinate"
    i1:
```

(continues on next page)

(continued from previous page)

```
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x20:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x21:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x22:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"
  x23:
    ports:
      i0:
        role: "subordinate"
      i1:
        role: "subordinate"
      o0:
        role: "manager"
      o1:
        role: "manager"

connections:
#port -> interconnect
  "i0 -> x00.i0":
```

(continues on next page)

(continued from previous page)

```
  "i1 -> x00.i1":
  "i2 -> x01.i0":
  "i3 -> x01.i1":
  "i4 -> x02.i0":
  "i5 -> x02.i1":
  "i6 -> x03.i0":
  "i7 -> x03.i1":

#1st stage
  "x00.o0 -> x10.i0":
  "x00.o1 -> x12.i0":
  "x01.o0 -> x11.i0":
  "x01.o1 -> x13.i0":
  "x02.o0 -> x10.i1":
  "x02.o1 -> x12.i1":
  "x03.o0 -> x11.i1":
  "x03.o1 -> x13.i1":

#2nd stage
  "x10.o0 -> x20.i0":
  "x10.o1 -> x21.i0":
  "x11.o0 -> x20.i1":
  "x11.o1 -> x21.i1":
  "x12.o0 -> x22.i0":
  "x12.o1 -> x23.i0":
  "x13.o0 -> x22.i1":
  "x13.o1 -> x23.i1":

#interconnect -> port
  "x20.o0 -> o0":
  "x20.o1 -> o1":
  "x21.o0 -> o2":
  "x21.o1 -> o3":
  "x22.o0 -> o4":
  "x22.o1 -> o5":
  "x23.o0 -> o6":
  "x23.o1 -> o7":

address_map:
  "o0":
    to:    o0
    base:  0x0000
    size:  0x1000
  "o1":
    to:    o1
    base:  0x1000
    size:  0x1000
  "o2":
    to:    o2
```

(continues on next page)

(continued from previous page)

```
    base:  0x2000
    size:  0x1000
  "o3":
    to:    o3
    base:  0x3000
    size:  0x1000
  "o4":
    to:    o4
    base:  0x4000
    size:  0x1000
  "o5":
    to:    o5
    base:  0x5000
    size:  0x1000
  "o6":
    to:    o6
    base:  0x6000
    size:  0x1000
  "o7":
    to:    o7
    base:  0x7000
    size:  0x1000
```

4.3.2 Command

```
omnibus examples/butterfly.yml -g
```

CHAPTER

FIVE

INTERCONNECT SPECIFICATION

The input depends on frontend used. Below are the specifications for available frontends.

5.1 YAML Syntax Specification

Full examples are provided at the bottom of this file and in *examples*.

The current version of the specification is `1.0`.

5.1.1 Top Level Fields

Field Name	Required	Description	Default
`desc_version`	yes	version of specification used	-
`name`	no	interconnect name	`interconnect`
`version`	no	interconnect version	`0`

5.1.2 Top Level Sections

These sections are explained in detail in their own subsections of this document.

Field Name	Required	Description
`global`	no	interconnect configuration options
`default`	no	default options for parameters
`clocks`	no	clocks used in the interconnect
`ports`	yes	interconnect ports
`crossbars`	no*	crossbars used
`connections`	no*	connections between ports and crossbars
`address_map`	yes	determines which addresses lead to what port

*`crossbars` and `connections` must either both be defined or both be left for inference

5.1.3 Global

Global parameters for configuring global interconnect behavior.

None of the fields are required. If they are unspecified, they are set to default value.
If the `global` field is empty, it is valid to omit it completely.

Field Name	Description	Allowed Values	Default
addr_width	address width, used in address map and by the components	any positive integer	32

5.1.4 Default

Default parameters specify the default value for various fields used throughout the specification.

None of the fields are required; if they are unspecified, they are unset.
If the `default` field is empty, it is valid to omit it completely. For the allowed set of values, see the corresponding fields in other sections.

Field Name	Description
data_width	data width of interconnect ports
clock	clock name to be used by components and ports
frequency	frequency to be used by clocks
protocol	bus protocol of interconnect ports

Backend Specific

Pulp Platform Axi

Accepted only with the `topwrap_pulpplatform` backend (the default), which defines them in `omnibus/backend/topwrap_pulpplatform.schema.yml`. Other backends report them as unknown with a warning.

Field Name	Description
axi_max_reads	max number of outstanding reads of data width and ID width converters

5.1.5 Clocks

Specifies clocks and their frequencies. Each clock implies a reset domain.

Each clock is defined as:

```
name:
    frequency: ...
```

where `name` has to be a unique clock identifier. `frequency` can be omitted, in which case it is taken from *Defaults*, or set to `n/a` if not available. Valid values for `frequency` are positive numbers in Hz (e.g. `100e6`), or strings made of an optional positive integer and an optional suffix `Hz`, `k`, `kHz`, `M`, `MHz`, `G` or `GHz`, which scales the value accordingly (e.g. `"100MHz"`, `"25k"`), or `n/a`. Strings can't hold fractions or exponents: write `"2500kHz"` or `2.5e6` rather than `"2.5MHz"`.

When omitted, one clock with the name `clk` and a default frequency is created.

5.1.6 Ports

Interconnect ports.

Each port has to have a unique name.

Field Name	Optional	Allowed Values	Description
`role`	no	'subordinate' or 'manager'	determines port role from the perspective of interconnect
`data_widt`	yes*	any positive integer	-
`clock`	yes*	any valid clock name, see *Clocks*	clock domain of the port
`protocol`	yes*	one of `axi`, `axilite`, `apb`, `ahb`	bus protocol to be used
`group`	yes	any valid name	group for muxed buses, NOT SUPPORTED YET

* optional only if default exists, see *Defaults*

5.1.7 Crossbars

Interconnect crossbars.

Each crossbar has to have a unique name. If a crossbar's clock is not specified and no default exists, then the clock will be inferred from the design.

Field Name	Optional	Allowed Values	Description
`clock`	yes	any valid clock name, see *Clocks*	clock domain of the crossbar
`ports`	no	see below	ports of the crossbar

Each port is defined as:

```
name:
    role: ...
```

where `name` has to be unique to the crossbar, while `role` takes either `subordinate` or `manager` as arguments. They specify whether the port is a subordinate or a manager with respect to the crossbar.

The crossbars field can be omitted. In such case:

- `connections` field must also be omitted
- one all-to-all crossbar is generated

5.1.8 Connections

Interconnect connections between ports and crossbars. If the `crossbars` field is omitted, this field must be omitted too; otherwise, it must be specified.

Each connection is specified as `porta -> portb` in the direction of interconnect subordinate ports to interconnect manager ports (device manager to device subordinate). If a port is a top

level port (from `ports` field), its name is taken as is, e.g. `cpu`. If a port is a crossbar port, its name is "crossbar.port", e.g. `main.cpu`.

Field Name	Optiona	Allowed Values	Default	Description
`regist`	yes	any non-negative integer	`0`	number of skid buffers NOT SUPPORTED YET
`cdc`	yes	`True` or `False`	`False`	whether to treat this connection as clock barier
`data_w`	yes	any positive integer	`False`	sets data width of this connection
`protoc`	yes	`True` or `False`	`True` on connections to interconnect manager ports, otherwise `False`	whether to treat this connection as protocol propagation barrier

Backend specific parameters

Pulp Platform Axi

Accepted only with the `topwrap_pulppplatform` backend (the default), which defines them in `omnibus/backend/topwrap_pulppplatform.schema.yml`. Other backends report them as unknown, with a warning.

Field Name	Optiona	Allowed Values	Default	Description
`axi_max`	yes	any positive integer	value from *Defaults* if exists, otherwise 1	max number of outstanding reads of data width and ID width converters on this connection

5.1.9 Address Map

Address map inside an interconnect. It is specified as a set of entries. Each entry is identified by a unique name.

Field Name	Optional	Allowed Values	Default	Description
`from`	yes	single port or list of ports	all ports	source ports for which address map applies NOT SUPPORTED YET
`to`	no	single port	-	destination port for which address map applies
`base`	no	any valid nonnegative integer	-	base address of address map
`size`	no	any valid nonnegative integer	-	size of address map entry

5.1.10 Full File

```
desc_version: 1.0

name: Interconnect
version: 0.1

#addr width defaults to 32

default:
    data_width: 64
    clock: clka
    frequency: 10MHz
    protocol: AXI4
    axi_max_reads: 1

clocks:
    clka:
        frequency: 10GHz
    clkb:
        frequency: 10e6
    clkc:
        #frequency taken from default

ports:
    porta:
        role: "subordinate"
        #other parameters are taken from their default value
    portb:
        protocol: AXI4Lite
        role: "manager"
        clock: clkb
        data_width: 32

crossbars:
    main:
        #clock taken from default
        porta:
            role: "subordinate"
        portb:
            role: "manager"

connections:
    "porta -> main.porta":
        registers: 1
        axi_max_reads: 2 #overwrites default
        #does NOT have "protocol: True"
    "main.portb -> portb":
        cdc: True
        #also has "protocol: True", can be overwritten with "protocol: False"
```

(continues on next page)

(continued from previous page)

```
address_map:
    "portb":
        to: portb
        base: 0x0000
        size: 0xFFFF
```

CHAPTER

SIX

TOOL ORGANIZATION

This page describes the general high level overview of the design structure. You'll find low level details in the comments and the source code.

6.1 Overview

Omnibus has a simple CLI interface specified in `main.py`.

The main file is `interconnect.py`, which provides `InterconnectBuilder` that takes optional frontend and backend as arguments. To provide illusion of a single `Interconnect` object with swappable frontend and backend, these arguments specify parents of `Interconnect`, which in turn make the class inherit all their methods.
This design provides an easy way to add a new frontend or backend - add a new file with a proper interface and specify it as an argument to `InterconnectBuilder`.

Internal processing is implemented directly in the `Interconnect` class, as there is never a reason to swap it.

6.2 Frontends

All frontends are put in `frontends/` and must have `Frontend` the class. They must populate the internal list of components. They may add new members to `Interconnect`. In particular, they are allowed, and even expected, to add new methods.

6.2.1 YAML

Argument name: "yaml".

This fronted reads the specification from a YAML file specified as an argument and populates the internal representation with error checking.
It is done with a single `read_yaml(filename)` call. No other methods are exposed.

6.3 Internal Processing

The `process()` method is implemented as shorthand for all the steps described below.

1. All paths from sources to destinations are found and encoded as crossbar configuration.
2. These paths are then picked depending on the address map needs; the rest are discarded.
3. Paths are coalesced into minimal representations for each crossbar - consecutive addresses are grouped together.

4. Parameters are propagated from each port that has them to all neighbors, but not over barriers. This process is repeated until there is nothing more to propagate.
5. ID widths are found, increasing width on each crossbar.
6. New components are inserted on each connection that has two different parameters on its end ports.

6.4 Backends

All backends are put in `backends/` and must have the `Backend` class. They read the internal list of components and create an appropriate output.

Generally, they should expose a simple `generate(outdir)` interface, but they are allowed to use different one.

A backend can accept specification parameters of its own, in `default` and in connections. It declares them in three places:

- `parse_default` and `parse_connections`: parameter name to parse function, set in its `__init__`. `Interconnect` adds the common parameters to them, and frontends warn about parameters found in neither.
- `SCHEMA_EXTENSION`: file name of a JSON Schema fragment next to the backend, or `None`. Properties under its `$defs/default` extend `default`, those under `$defs/connection` extend connection parameters. The YAML frontend validates specifications against `frontend/spec.schema.yml` with the fragment of the selected backend.
- `port_connection_params`: connection parameters that internal processing copies onto the port each connection starts at, for components that read them from ports.

See `topwrap_pulpplatform` and its `topwrap_pulpplatform.schema.yml` for an example.

6.4.1 Graphviz

Argument name: “graphviz”.

This backend converts an internal representation into a graphviz graph. It generates a single `.gv` file and can be configured to display it in browser. It allows for coloring according to some parameters.

This backend’s primary purpose is debugging and visualization.

6.4.2 Topwrap Pulp Platform

Argument name: “topwrap_pulpplatform”.

This backend converts an internal representation into a Topwrap file and SystemVerilog wrappers. It expects Pulp Platform modules for operation.
It requires external commands to be executed to produce a proper output.
Verilator files are also provided to the set list of files and remove unnecessary warnings.

New components can be added to it by:

1. creating a new component class deriving from `MappedComponent`, which:

- forwards constructor arguments to `MappedComponents` and specifies the `ip` name,

- creates the `generate_verilog(self, outdir)` method which creates verilog wrappers/modules,
- creates `generate_hierarchy(self, ports, params)`, which populates Topwrap's hierarchy - ports and parameters.

2. populating the registry in `map_components`.
3. possibly adjusting the choice procedure in `map_components`.

In short:

1. It goes through all components that were generated in the internal processing stage, and using some choice function, they are mapped to Verilog components with the help of `registry`.
2. Verilog files are generated.
3. Verilator filelist and waivers are generated.
4. Topwrap hierarchy is generated.

CHAPTER

SEVEN

SIMULATION TESTS

`tests/sim` holds simulation tests of the RTL that Omnibus generates. Each specification in `tests/sim/specs/` is turned into SystemVerilog, compiled with Verilator, and exercised by cocotb 2.0 tests using cocotbext-axi managers and memories, and a cocotb-ahb memory for AHB3 ports.

The bridge `axi_prconv_axi_ahb` also has UVM tests, with the AXI4 and AHB AVIPs on Verilator; see uvmtests.md.

This page describes how the suite is organized, what it generates while running, and how a single test runs. It is intended for reviewers of the suite and for anyone extending it.

7.1 Quick start

Requirements:

- Verilator on `PATH`; the suite was run with v5.052, 5.053 and a newer commit of Verilator's master.
- The Python packages in the `sim` extra: `pip install ".[sim]"` (cocotb, cocotbext-axi, cocotb-ahb, pytest).
- The pulp submodules in `omnibus/third_party` (`axi`, `common_cells`, `tech_cells_generic`): `git submodule update --init`.

The tests skip themselves when Verilator, cocotb, cocotbext-axi or Topwrap are missing. The bridge tests (see *Converter tests*) also need cocotb-ahb.

```
pytest tests/sim                                    # everything
pytest tests/sim -m tier1_smoke                     # one tier
pytest tests/sim -k dw_mix                          # one specification
pytest tests/sim -k "dw_mix and tier1_smoke"        # one specification and tier
COCOTB_TEST_FILTER=byte_lanes pytest tests/sim -k "dw_mix and tier1_smoke"
                                                    # one cocotb test
pytest tests/sim --sim-waves                        # also dump waveforms
pytest tests/sim/test_bridges.py --sim-seed 42   # repeat the random stimulus of␣
↪the bridge tests
```

7.2 Layout

```
tests/sim/
|-- conftest.py          pytest options, tier markers, skip without Verilator
|-- test_sim.py          pytest entry point: specifications x tiers
|-- specs/*.yml          the specifications under test, one feature each
|-- harness/
|   |-- __init__.py      paths and the list of tiers
|   |-- generate.py      specification -> RTL -> Verilator build, cached
|   |-- refmodel.py      what a specification says the interconnect must do
|   |-- env.py           the test bench: clocks, resets, managers, memories
|   |-- ahb.py           AHB memory (cocotb-ahb) with the same backdoor access as␣
↪AxiRam
|   |-- axichecker.py    passive AXI protocol checker
|   |-- apbchecker.py    passive APB protocol checker
|   `-- ahbchecker.py    passive AHB protocol checker
|-- tiers/
|   |-- tier0_build.py   cocotb tests of tier 0
|   |-- tier1_smoke.py   cocotb tests of tier 1
|   `-- tier2_negative.py cocotb tests of tier 2
|-- test_bridges.py      pytest entry point of the converter tests
|-- bridges/axi_ahb.py   cocotb tests of the AXI4 -> AHB3 bridge module alone
`-- .build/              generated, git-ignored: RTL, simulator builds, logs
```

Two kinds of tests are involved, and it helps to keep them apart:

- **pytest tests**, in `test_sim.py`, which is what `pytest` collects and reports:
  - `test_build[<spec>]` generates and builds one specification;
  - `test_tier[<spec>-<tier>]` runs one tier module against one build.
- **cocotb tests**, the `@cocotb.test()` functions in `tiers/*.py`. They run inside the simulator. One `test_tier` runs all cocotb tests of its tier module in a single simulator process, and fails if any of them fails.

7.3 How a test run works

The whole path of one `pytest tests/sim` run, for one specification:

```
pytest                                  simulator process (Verilator + cocotb)
------                                  --------------------------------------
collect specs/*.yml x TIERS
test_build[spec]
  builds fixture -> generate.build()
    refmodel.load(spec)
    cached build? -- yes ---------+
    _check_generation(spec)       |
    omnibus <spec>                |
    topwrap repo parse ...        |
    topwrap generate ...          |
    Verilator build (runner)      |
```

(continues on next page)

(continued from previous page)

```
  check EXPECTED_MODULES  <-------+
test_tier[spec-tier]  (per tier)
  runner.test(test_module=tier,
    TOPWRAP_SIM_SPEC=spec)  ----------> import tiers/<tier>.py
                                        for each @cocotb.test():
                                          IcEnv(dut): load $TOPWRAP_SIM_SPEC,
                                            models and checkers per port
                                          env.start(): clocks, reset
                                          accesses, checks, env.finish()
                                        write run/<tier>/results.xml
  get_results(results.xml)  <----------
  assert no cocotb test failed
```

7.3.1 Collection

`test_sim.py` parametrizes both pytest tests over every `specs/*.yml` and, for `test_tier`, over `TIERS` from `harness/__init__.py`. Each tier gets a pytest marker of the same name (registered in `conftest.py`), which is what `-m tier1_smoke` selects.

The session-scoped `builds` fixture builds each specification once and hands the same build to `test_build` and every `test_tier` of that specification. If the build fails, the fixture remembers the exception; `test_build` reports it, and every `test_tier` of that specification is skipped with a pointer to `test_build`, instead of failing with the same error.

7.3.2 Generating the DUT

`generate.build()` (in `harness/generate.py`) runs the same steps as `tests/generate.sh`, in a build directory of its own:

1. `_check_generation()` runs `read_yaml()` and `process()` in-process and fails on any recorded error. The `omnibus` command line tool exits 0 even when `process()` records errors, so without this, a broken specification would still produce RTL.
2. `omnibus <spec>` generates the interconnect RTL and a Topwrap design of it into `out/`. It also writes `topwrap.yaml`, a Topwrap configuration that registers `out/modules` as a repository named `repo`, and `out/interfaces`, which holds the AHB and APB interface definitions, as a repository named `interfaces`.
3. `topwrap repo parse repo out/src -I` infers Topwrap IP core descriptions for the generated modules and stores them in that repository.
4. `topwrap generate --verilog out/<name>.yml --flatten-interfaces` writes the top-level module `build/src/<name>.sv`.
5. cocotb's runner builds that top level with Verilator (see *Building the simulator*).

All commands and their output go to `generate.log`, so a failed generation can be read and re-run by hand from there.

7.3.3 What is generated

Everything below `tests/sim/.build/` is generated during the run. For a specification `specs/<spec>.yml` with `name: <name>`:

```
tests/sim/.build/<spec>-<hash>[-waves]/
|-- generate.log                    commands of steps 2-4 and their output
|-- topwrap.yaml                    step 2: Topwrap config, repositories `repo`␣
↪and `interfaces`
|-- out/                            step 2, omnibus:
|   |-- src/*.sv, *.vh              RTL: crossbars (from axi_intercon_gen),
|   |                               converter and register slice wrappers
|   |-- <name>.yml                  Topwrap design of the interconnect
|   |-- <name>.f                    filelist of the RTL in src/ to compile
|   |-- *.core                      FuseSoC cores written by axi_intercon_gen
|   |-- modules/                    the `repo` repository:
|   |   `-- cores/<m>/module.yaml   step 3: inferred IP core descriptions
|   `-- interfaces/interfaces/      the `interfaces` repository, step 2:
|                                   AHB/APB interfaces, when used
|-- build/src/<name>.sv             step 4, topwrap generate: the top level (DUT)
|-- sim_build/                      step 5: Verilator model, simulator binary
|-- build.log                       Verilator's output
|-- built                           marker: the build completed
`-- run/<tier>/                     one per tier, written while the tier runs:
    |-- sim.log                     cocotb's log
    |-- results.xml                 cocotb's JUnit results
    `-- dump.fst                    waveform, with --sim-waves
```

Nothing is generated outside `.build/`. The harness runs every command inside the build directory, because the Omnibus backend writes `topwrap.yaml` into the working directory.

7.3.4 Building the simulator

The Verilator build compiles:

1. the pulp sources listed in `tests/verilator/pulp_platform_filelist.f`, from the `omnibus/third_party` submodules;
2. the generated RTL listed in `out/<name>.f`;
3. the top level `build/src/<name>.sv`.

The order matters, as the generated wrappers use the `axi/typedef.svh` macros without including the file and get them from the pulp files compiled before them. That is also why the generated filelist is passed as sources and not with `-F`: cocotb's runner puts build arguments before the sources.

Other options:

- The `+incdir+` and `+define+` lines of the pulp filelist, including `COMMON_CELLS_NO_DEPRECATED_WARNINGS`, which silences a deprecation `$warning` in `common_cells`.
- `-Wno-fatal -Wno-lint -Wno-style`: warnings don't fail the build. Linting is the job of `tests/generate.sh`.

- `--x-assign unique --x-initial unique`: Verilator has no X state. These randomize what would be X, so an undriven signal shows up as a wrong value instead of a quiet 0.
- A 1 ns / 1 ps timescale. The top-level module is the specification's `name`.

7.3.5 The build cache

A build is reused as long as its inputs are unchanged. The directory name carries a hash (`_cache_key()`) of:

- the specification;
- the installed `topwrap` package (`*.py`, `*.yaml`, `*.j2`);
- the `omnibus` package (`*.py`, `*.sv`, `*.yml`, including the `.sv` files of the pulp submodules);
- `harness/generate.py` and the pulp filelist.

A directory is reused only if its `built` marker exists, so an interrupted build starts over. Starting a new build of a specification deletes its older builds.

Changes to the test bench (`env.py`, the checkers, `refmodel.py`) or to `tiers/` don't need a rebuild: they're Python, loaded when a tier runs. To force a rebuild, delete the build directory.

7.3.6 Running a tier

`test_tier` calls cocotb's `runner.test()`, which starts the simulator binary once per specification and tier. It runs with `run/<tier>/` as its working directory, and with:

- `TOPWRAP_SIM_SPEC`: the specification's path. The test bench reads it to know what it's testing.
- The pytest process's `sys.path` as `PYTHONPATH`, which cocotb's runner passes on. `conftest.py` puts `tests/sim` and `tests/sim/tiers` on it, so the simulator's Python can import the tier module and `harness`.
- The rest of the environment. That's why `COCOTB_TEST_FILTER` and `COCOTB_RANDOM_SEED` work from the command line.

cocotb runs the tier module's tests one after another in the same simulation. Each test builds its own test bench (`IcEnv`) and resets the DUT, and cocotb stops the tasks a test started (clocks, models, checkers) when it ends.

When the simulator exits, `test_tier` reads `results.xml` and fails if any cocotb test failed, naming the log to look at.

7.4 The DUT's interface

The top-level module exposes flat ports:

- `<port>_<signal>` for each top-level port of the specification, e.g. `cpu_awaddr`;
- one clock and one active-low reset per clock of the specification: `clk_<name>_i` / `rst_<name>_ni`, where a leading `clk_` or `clk` is removed from the clock name, so `clk` gives `clk_i` / `rst_ni`.

The test bench attaches to them by name:

Specification port	Driven by the test bench as
`role: subordinate` (an external manager connects to it)	`AxiMaster`
`role: manager`, AXI4	`AxiRam`
`role: manager`, AXI4-Lite	`AxiLiteRam`
`role: manager`, AHB3	`AhbRam`: a cocotb-ahb `DUTManager` on the port pins, a `SimMem1PSubordinate` with random wait states behind it
`role: manager`, APB3	`ApbRam`

7.5 Inside the test bench

7.5.1 Reference model

`harness/refmodel.py` turns a specification into `SpecModel`: its ports, clocks, address windows, and which ports the test bench can't model.

Port attributes and clock frequencies are read through Omnibus's own YAML frontend, because they're inputs to the generator. The address map, however, is read from the file and decoded by the model itself (`decode()`, `testable_windows()`), without Omnibus's routing code, so that the expected results don't inherit the generator's bugs. A window with no `from` is reachable from every manager.

7.5.2 Test bench

`IcEnv` in `harness/env.py`, created at the start of each cocotb test:

- **Clocks:** one per specification clock, from its frequency (100 MHz if `n/a`), with the period rounded to an even number of picoseconds as cocotb's `Clock` requires.
- **Resets:** `start()` holds every reset low for `RESET_CYCLES` cycles of the slowest clock, then releases them in random order.
- **Managers and memories:** one `AxiMaster` per manager port, one memory per memory port (see *The DUT's interface*).
  - Memories have size `2**addr_width` and are sparse, so they answer at the absolute addresses the interconnect passes through at no cost. cocotbext-axi's default size of 2**64 doesn't work: it overflows Python's `len()`.
- **Protocol checker:** one per port: `AxiChecker`, `ApbChecker` or `AhbChecker`, by the port's protocol.
- **Logging:** cocotbext-axi logs every transaction at INFO, so its loggers are raised to WARNING.
- **Timeouts:** every bus operation goes through `op()`, which fails the test after `OP_TIMEOUT_CYCLES` cycles of the slowest clock instead of hanging on a deadlock.

A memory accepts any address, so writing a value and reading it back over the same path can't tell a misrouted access from a correct one. Two checks close that gap:

- `prime(address, length)` fills that range in *every* memory with bytes unique to the memory (`tag()`). A read that lands in the wrong memory, or at the wrong offset, returns the wrong bytes.

- `check_write()` inspects the memories directly after a write, without going through the bus: the target must hold the data, and every other memory must still hold its tags.

`finish()`, called at the end of a cocotb test, fails the test on any protocol violation any checker has seen.

7.5.3 Protocol checkers

`AxiChecker` in `harness/axichecker.py` samples one port on every rising edge of its clock, in both directions, so it checks the DUT's outputs and the test bench's own stimulus alike:

- a VALID that wasn't accepted stays high, with its payload unchanged;
- W bursts have AWLEN+1 beats, with WLAST on the last (AXI4);
- R bursts have ARLEN+1 beats, with RLAST on the last, per ID (AXI4);
- every B and R response has an outstanding request with its ID.

Its state is cleared while the port's reset is asserted. The first 20 violations of a port are kept, with their simulation time, for `finish()` to report.

`AhbChecker` in `harness/ahbchecker.py` samples one AHB manager port on every rising edge of its clock. It checks the DUT's address phase and the responses it gets:

- an address aligned to HSIZE, with the transfer within the data bus;
- an address phase that stays unchanged while HREADY is low (an ERROR response may cancel it);
- SEQ and BUSY only inside a burst, at the address the burst continues with;
- HBURST, HSIZE and HWRITE constant inside a burst, and INCR bursts not crossing a 1KB boundary;
- fixed-length bursts completed (unless an ERROR cancelled them), and no burst ending with BUSY;
- no transfer after the second cycle of an ERROR response.

It also records the accepted NONSEQ and SEQ address phases (`xfers`) and the BUSY cycles (`busy`), for tests that look at how a burst was split. Its state is cleared while the port's reset is asserted. The first 20 violations of a port are kept, with their simulation time, for `finish()` to report.

7.6 Tiers

cocotb test	What it does
`tier0_build.port_widths`	every modeled port exists, with the specification's data width and address width
`tier0_build.reset_idle`	no VALID driven by the DUT rises during reset or in the 100 cycles after it
`tier1_smoke.rw_every_window`	from every manager, the first, middle and last word of every window it may access
`tier1_smoke.address_bit_walk`	from every manager, `base + 2**k` for every address bit inside each window: catches decoding and aliasing bugs
`tier1_smoke.byte_lanes`	narrow writes of every size below the bus width, on every byte lane, change only their bytes
`tier1_smoke.shared_visibilit`	what one manager writes to a window, every other manager allowed there reads back; returns right away with a single manager
`tier2_negative.unmapped_access`	from every manager, reads and writes (one beat and 16 beats) to the first, the last and a random burst position of every gap of the address map: DECERR, and no memory changes
`tier2_negative.mixed_legal_and_u`	every manager reads and writes at the same time, each access to a random legal or unmapped address: OKAY with the right data or DECERR, whatever order the responses come back in

Each Tier 1 access is checked three ways: the response code, the data read back, and, through `prime()` and `check_write()`, that it landed in the right memory at the right address.

Tier 2 collects accesses the interconnect has to refuse. The gaps come from `SpecModel.gaps()`, the address ranges no window covers; the access never reaches a target, so a leaked write shows as changed tags and a leaked read as an OKAY response. A gap smaller than a bus word is accessed with narrower beats, so no gap is skipped; `specs/small_gaps.yml` has gaps of 4 bytes and 1 byte for this. A specification has to leave a gap, and in `mixed_legal_and_unmapped` every manager needs a legal range to read and to write as well: the test fails otherwise, instead of testing nothing. Given that, every reader and writer issues both kinds of access. In `mixed_legal_and_unmapped`, reads use the lower half of a window, which nothing writes, and each manager writes only to its own part of the upper half, so the data checks hold with all managers running at once.

7.7 Known failures and guards

Two tables in `test_sim.py` keep the suite honest:

- **`EXPECTED_MODULES`** lists the modules a specification's generated RTL must contain, e.g. `axi_cdc_wrapper` for `cdc`. `test_build` fails if one is missing, so a specification can't silently stop exercising the component it's in the corpus for.
- **`XFAIL`** lists known failures by (specification, tier) or (specification, `"build"`), with the reason. They're marked `xfail(strict=True)`: pytest reports them as `XFAIL` without failing the run, and if one starts passing it's reported as `XPASS(strict)` and fails, so the entry gets removed together with the fix.

Expected failures and skipped tests don't fail the run, but `pytest -rxXs` lists them in its summary. An expected failure accepts any failure of its test, not only the one its reason describes.

7.8 Working with the suite

7.8.1 Adding a specification

Add a `.yml` to `tests/sim/specs/`; `pytest` picks it up. If it's there to exercise a specific component, add the component's module name to `EXPECTED_MODULES`.

7.8.2 Adding a cocotb test or a tier

A new cocotb test is a `@cocotb.test()` function in the tier module it belongs to. It typically starts with `env = IcEnv(dut)` and `await env.start()`, uses `env.managers`, `env.memories` and `env.model`, and ends with `env.finish()`.

A new tier is a module in `tiers/`, plus its name in `TIERS` in `harness/__init__.py`, which also gives it a pytest marker.

7.8.3 Debugging a failure

- **Logs.** The pytest failure names the log: `run/<tier>/sim.log` in the specification's build directory. Build problems are in `generate.log` (generation) and `build.log` (Verilator).
- **Narrowing down.** Select one specification and tier with `-k "<spec> and <tier>"`, and one cocotb test with `COCOTB_TEST_FILTER=<test name>`.
- **Reproducing random data.** `sim.log` starts with `Seeding Python random module with <seed>`; `COCOTB_RANDOM_SEED=<seed>` repeats that run. The bridge tests take one seed per pytest run: it is printed in the pytest header (`sim seed: <seed>`), stored as the `seed` property of every test in the JUnit report and repeated in the message of a failure, and `--sim-seed <seed>` reuses it.
- **Waveforms.** `--sim-waves` builds a separate simulator with tracing (a `-waves` build directory) and writes `run/<tier>/dump.fst`.
- **Generation by hand.** The commands in `generate.log` can be re-run in the build directory.

7.9 Notes for reviewers

- **Deprecation warnings.** cocotbext-axi uses APIs that cocotb 2.0 deprecates; the `py.warnings` lines in `sim.log` come from there.
- **Known issue: builds with and without `--sim-waves` evict each other.** Removing a specification's older builds also removes its build of the other kind, so alternating between the two rebuilds every time.
- **Not covered yet.** The rest of tier 2 (window edges, exclusive accesses, protocol violations), bursts, concurrent traffic and performance, planned as further tiers.

7.10 Converter tests

`test_bridges.py` tests single converter modules without an interconnect around them: the module under `omnibus/modules` is the Verilator top level, and `bridges/<name>.py` holds its cocotb tests. Currently that is `axi_prconv_axi_ahb`, built for 32 and 64 bit data and for `FifoDepth` 1 to 4 (the FIFO depth parameter):

```
pytest tests/sim/test_bridges.py
pytest tests/sim/test_bridges.py -k "64-3"      # 64 bit data, FifoDepth 3
```

The AXI side is driven with cocotbext-axi channel sources and sinks, and the AHB side is answered by an `AhbRam`. A Python model of the AXI address rules says which bytes each beat of a burst carries, and the memory is compared with a byte-for-byte model after every write. The tests cover:

- full-width, narrow and unaligned transfers, INCR, WRAP and FIXED bursts, sparse and empty `WSTRB`, IDs, read/write arbitration, and random traffic with backpressure;
- which AHB bursts an AXI burst becomes (`ahb_bursts`): HBURST=INCR restarted at every 1KB, WRAP4/8/16 for reads, single transfers for FIXED, WRAP2, WRAP writes and beats that have to be split;
- BUSY while the R channel is blocked (`busy`), and the cycle counts of 16-beat bursts with no wait states (`throughput`; reads need `FifoDepth` 3 for one beat per cycle);
- AHB errors (`ahb_error`): no further AHB transfers after the error, SLVERR on the remaining read beats and on the B of a write.

The AHB side is watched by an `AhbChecker` (see *Protocol checkers*). `ahb_bursts` and `busy` also look at the address phases and BUSY cycles it records, and `Bench.finish()` fails a test on its violations.

CHAPTER

EIGHT

PLANNED EXTENSIONS

This page is accurate as of: 2026-08-18.
It might become out of date, especially regarding WIP features. It will be periodically updated. Always reference the main documentation for up-to-date information.

8.1 General

8.1.1 Muxed Buses

Status: Not started

Buses like AHB and APB are muxed, they do not necessarily need a dedicated crossbar. Right now, this is not supported; individual protocol converters are inserted at each such port.

8.2 Frontend

8.2.1 Python API

Status: Planning

This frontend will make the usage of the Python interface more convenient and user-friendly. Omnibus will be importable from Python. Through methods like `add_port()` or `verify()`, it will provide a way to construct an interconnect. The primary purpose of this interface is to act as one of Topwrap's interconnect backends.

8.2.2 KPM Frontend

Status: Planning

This frontend will allow the GUI to specify the interconnect structure in similar way to Topwrap.

8.2.3 YAML Verification

Status: Partially implemented

Not all cases are properly verified; Omnibus does not always abort on erroneous input, but it does check for most problems.

8.3 Backend

8.3.1 Verilog backend

Status: Not started

This backend will allow Omnibus to be fully standalone without relying on Topwrap.

8.3.2 Pulp Platform

Status: WIP

The backend is mostly complete - however, it does rely on certain AXI scripts, which limit capabilities of the address map. Eventually, these scripts will be replaced.

8.4 Analysis

8.4.1 Latency Analysis

Status: Planning

This feature will allow to estimate the latency between ports in an interconnect.

8.4.2 Throughput analysis

Status: Planning

This feature will allow to estimate the throughput of an interconnect.