# Towards a CERN-wide common cores library

LHCb

EP R&D

Alberto Perro

FPGA Developers' Forum 2024 - CERN

# Overview

Current situation

Motivation

Library Structure and Philosophy

Library Content

Testbenches

Formal Verification

Coding Guidelines

Case studies

# Current situation in LHCb: Gateware

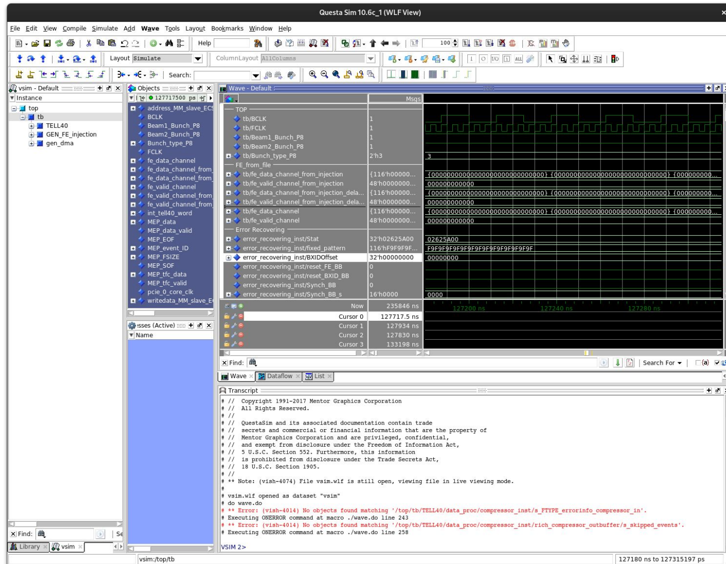
LHCb uses a **common FPGA hardware platform** (PCIe40) shared across all sub-detectors:

- A **common framework** is shared between sub-detectors to reduce development overhead
- Sub-detectors have to develop their own **custom gateware** on top to process the frontend data formats

This results in more than **30 different gateware variants** to test and maintain.

As most FPGA designs, **gateware is not made to be portable** between devices of the same vendor, let alone different vendors.

# Current situation in LHCb: Validation




**Test coverage is limited** and testbenches can cover only a small amount of cases.

Each test uses **custom Tcl** scripting and it can only run on one simulator (and sometimes a specific version).

Simulation also takes a long time: almost 3 hours for a single gateware flavor.

Covering all flavors requires lots of instances in parallel, each requiring a license.

# Motivation

We started an effort to use **modern validation** and **verification** methodologies in our gateware development. This showed us:

- We need better and more exhaustive integrated testing
- Most modules use **proprietary code**, which limits our selections of test tools and portability to new platforms
- A bunch of entities have similar functionality, but are written by hand by different developers, hiding **corner case bugs**

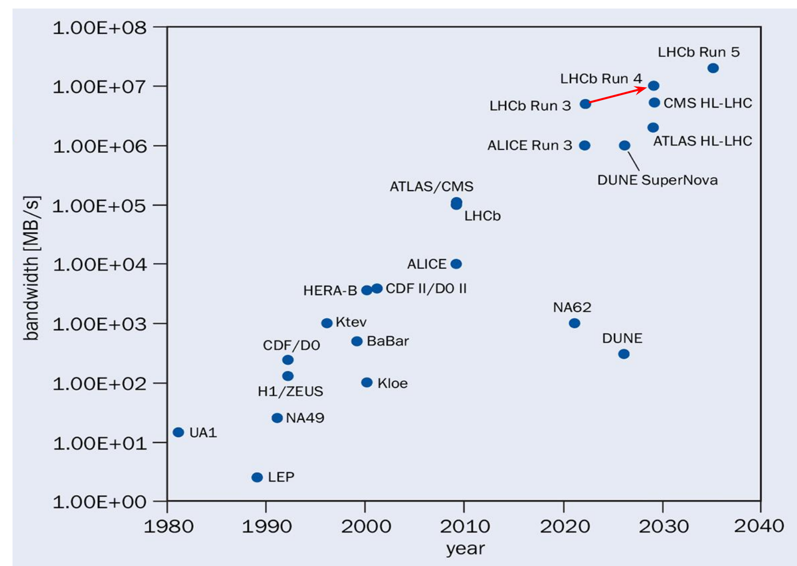
*Reinventing the Square Wheel*

# DAQ Upgrade in LHCb

Higher DAQ throughput requires LHCb to upgrade its hardware platform.

The PCIe40 successor - *PCIe400* - will require a **major porting effort** of the current gateware, even though its FPGA is made by the same vendor (Arria 10 to Agilex 7).

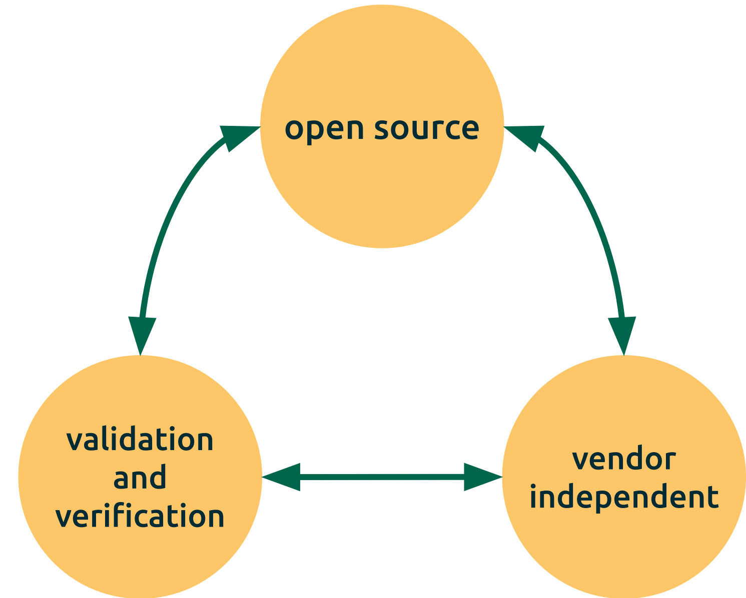
We decided that a **common core library** - *colibri* - was necessary to both speed up development, improve code reuse, and step away from vendor lock-in.



# Library Structure and Philosophy

colibri has these foundation principles in mind:

- The library needs to be **open source** (GNU GPLv3) to build a strong community around it
- It has to be completely **vendor-independent** to ensure portability and avoid vendor lock-in
- All components must be **validated and verified**, to ensure their correct functionality



# Library Structure and Philosophy

These principles, translated into practice, become:

- The whole library is written in a subset of **VHDL-2008**, which is compatible with all toolchains tested so far.
- The choice of a single language enables the use of **open source simulators** such as GHDL and NVC.
- Open source simulators and **modern testing frameworks** (UVVM, OSVVM, VUnit) can run in CI/CD without requiring licensing.
- Tcl scripting is discouraged, use of **python-based scripting** frameworks is highly recommended.

*Toolchains tested:*

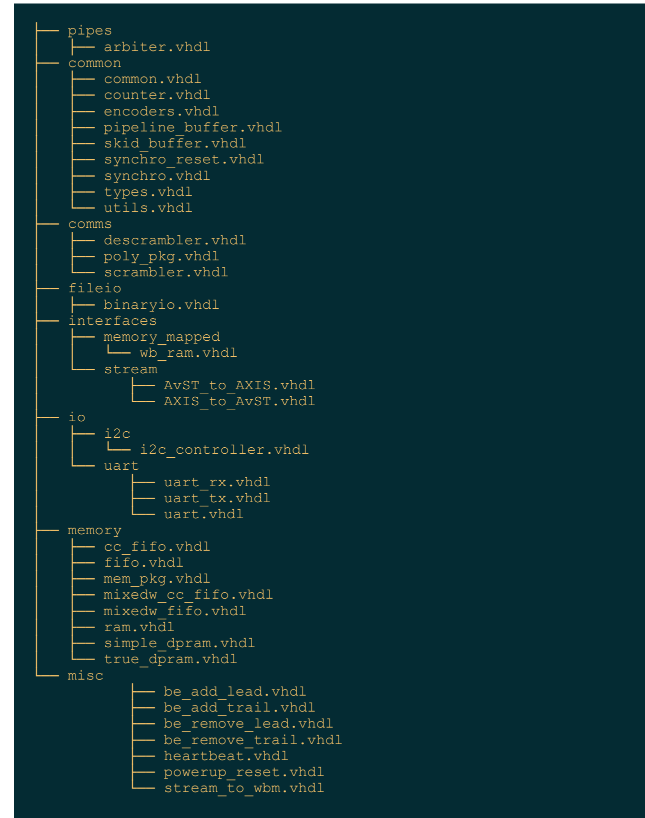
- AMD Vivado
- Intel Quartus
- Lattice Radiant
- Microsemi Libero
- Efinix Efinity
- Gowin EDA
- Aldec Riviera Pro
- Mentor Modelsim
- NVC
- GHDL

# Library Content

The library is a collection of the **most common cores** we encountered during design and simulation of gateware. Each core has been written from scratch, new cores are continually added.

Here are the main categories present:

- Common Functions
- Pipes entities
- File Operations
- Memory Entities (FIFOs, RAMs)
- Utilities for Communications (scrambler, PRBS, …)
- Input/Output (I2C, UART, …)
- Adapters for interfaces (AXI, Avalon, …)
- Miscellaneous entities yet to classify



# Validation and Verification: Testbenches

UVVM

OSVVM

**Self-checking testbenches** are present for each component available in colibri.

Unit tests and coverage tests are organized under the **VUnit** testing framework, which enhances portability between simulators.

**UVVM** and **OSVVM** verification frameworks are also used to improve test functionality (random stimuli, scoreboards, BFMs).

All testbenches are **run in CI** using the open source NVC simulator, which offers great simulation speed and stability.

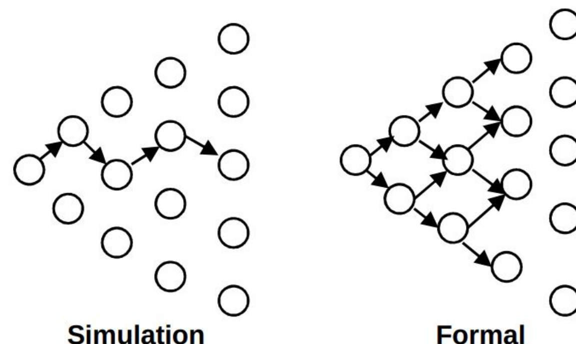
# Validation and Verification: Formal

Formal Verification (FV) methodologies are a set of techniques use static analysis based on mathematical transformations to determine the correctness of gateware behaviour.

**FV can guarantee the absence of bugs,** since it can prove mathematically that the model satisfies the requirements.

Library components which functionality can be formally proved are equipped with FV tests.

The full workflow in CI uses **GHDL** as VHDL interpreter, **Yosys** as a synthesis engine and **SymbiYosys** as the FV front-end.




GHDL

YosysHQ

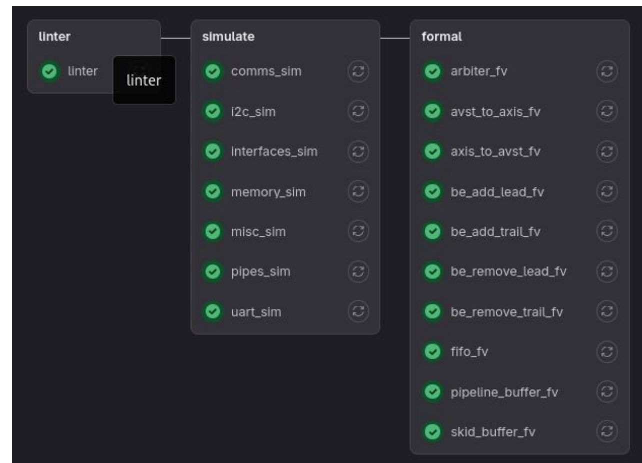
# Continuous Integration

Continuous Integration is a crucial tool to enforce and guarantee the **functionality** of the library.

**Containers** on GitLab runners are used for running self-checking testbenches, style-checking, and formal verification.

Both testbenches and style checking are integrated to the automated pipeline report using the **JUnit XML reporting** framework.

The reports help significantly the developers in spotting bugs without going through all test artifacts and logs.



# Coding Guidelines

**Coding style guidelines** are included in the project. These guidelines help to maintain uniformity throughout the codebase.

Guidelines are **enforced in CI** by using the open source linter and style checker VSG.

Well commented code, markdown documentation, and diagrams are also crucial for the wide adoption of colibri.

# Case Studies

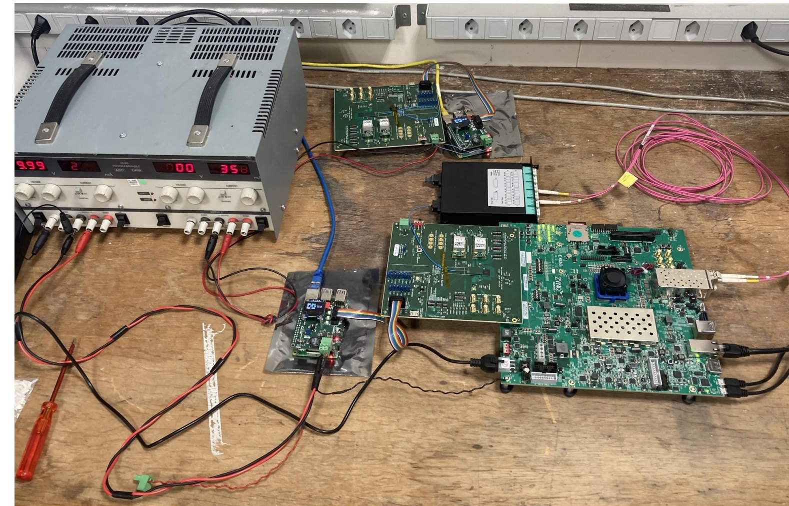
Using colibri on the field

# Case 1: Aurora Protocol

**Aurora** is an open protocol supported by AMD Xilinx devices. It will be used by the **FastRICH** Frontend ASIC over an **lpGBT** link to encode and transmit data.

The decoding of the protocol, which will run on the backend FPGA, has been **developed entirely using colibri**.

The FPGA gateware has been tested and ported with success on both **AMD Xilinx** boards and **Altera** boards. The porting effort only required setting the right timing and physical constraints.



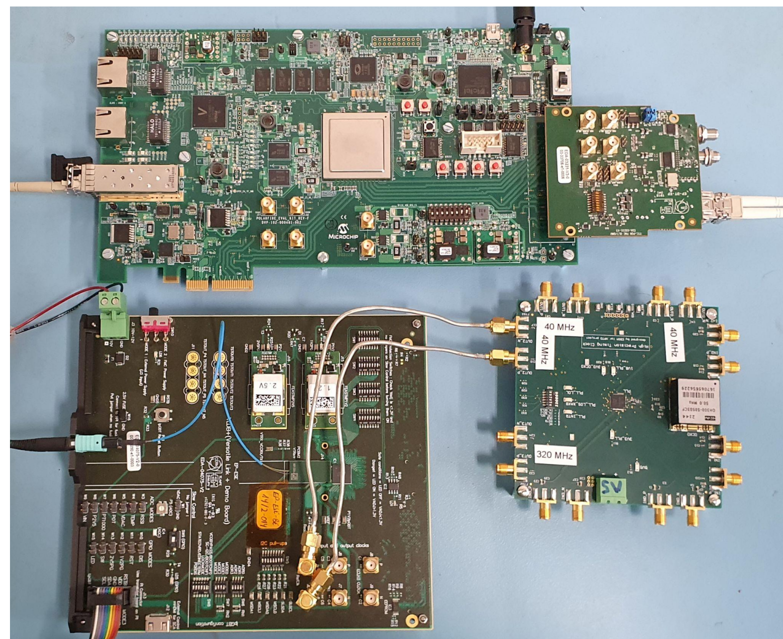
# Case 2: 10G UDP/IP transmitter

As part of the EP R&D WP9.3 and ECFA DRD7.5, **Ethernet** is being evaluated as a frontend data link.

A proof-of-concept for a media converter between **lpGBT** and **Ethernet** was developed.

The full UDP/IP tx pipeline has been designed **using colibri components**.

The code was ported and tested successfully on **AMD Xilinx** and **Microsemi** FPGAs.



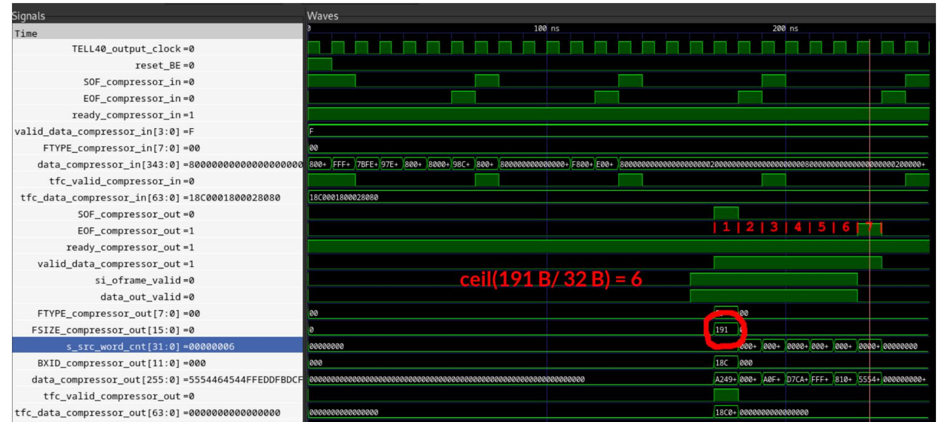

*Credits to Valentin Stumpert of EP-ESE*

# Case 3: Verification of LHCb Gateware

Formal verification was applied to the LHCb gateware in different situations to find corner cases that showed up during commissioning.

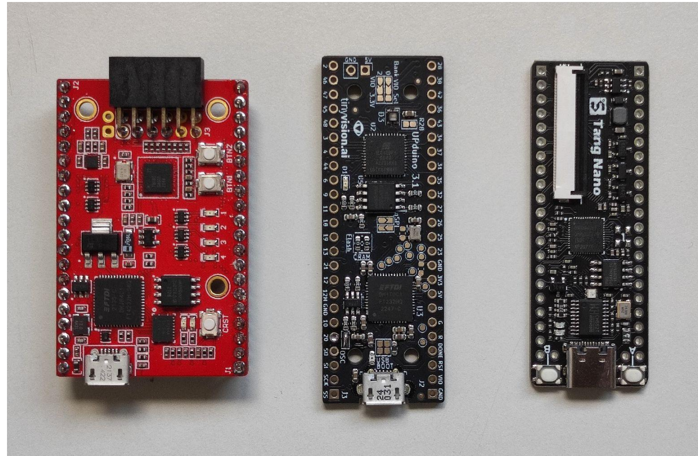
Components from colibri were used to **replace proprietary components** that are not compatible with the FV toolchains.

This also allowed the use of **open source simulators** to extend the test coverage.




*A corner case bug in the LHCb gateware where there is a mismatch between the number of words in the packet and the packet size. FV was used to find it.*

# Vendor Agnosticism




| Efinix<br>Trion T8F81C2 | Lattice<br>iCE40UP5k | Gowin<br>GW1N-1 |
|---|---|---|

The portability of the library between vendors was tested with a simple gateware consisting of:

- External sensor serial readout
- Mixed width FIFOs
- FIFO data buffers
- UART to read raw data
- UART for control and status registers

The gateware design aims to simulate a very simple frontend readout design.

The gateware was **ported with success** to Lattice, Efinix, and Gowin.

# Future Prospects

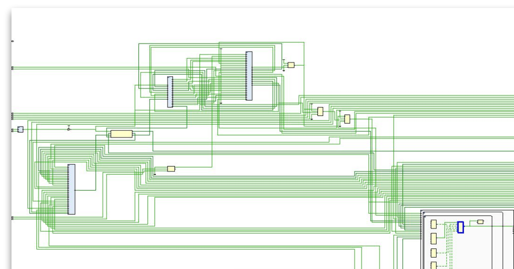
The work presented wants to be a step towards a **CERN-wide common core library**.

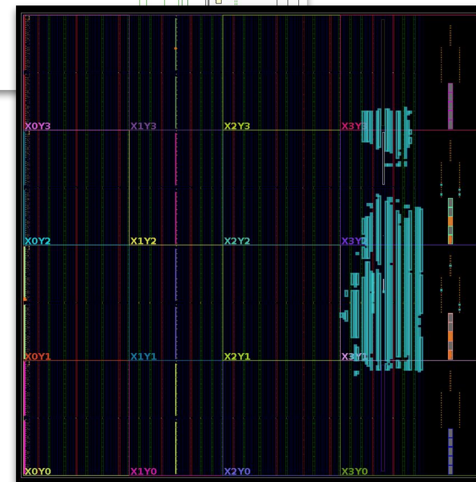
The structure and the philosophy behind the project allows easy **adoption**, **portability**, and **integration** in current and future gateware designs.

The LHCb developers will widely use colibri to develop the gatewares for the next upgrade.

The library is also being included in the EP R&D working package 9.3.1: Firmware Portability Framework.

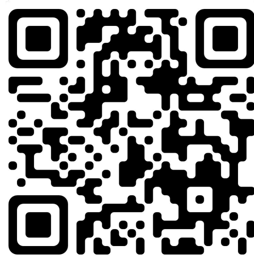
We hope that more experiments and groups will join us and adopt colibri for their developments.





# Thanks for the attention!

The library can be found at:
https://gitlab.cern.ch/colibri/colibri



*Special thanks to:*
LHCb Collaboration - Online Project
EP department - R&D WP9.3

# References: Tools

- https://vunit.github.io/
- https://www.uvvm.org/
- https://osvvm.org/
- https://yosyshq.readthedocs.io/
- https://github.com/ghdl/ghdl
- https://github.com/nickg/nvc
- https://github.com/jeremiah-c-leary/vhdl-style-guide

# Backup

# VUnit Configuration Script

VUnit is a Python Package. The main configuration script is a python script where the source files and libraries are declared.

Custom tests, simulator-specific options, and test configuration is also done in this script.

```
from vunit import VUnit

vu = VUnit.from_argv()
vu.add_vhdl_builtins()

work_lib = vu.add_library(‘work_lib’)
work_lib.add_source_files("<path_to_sources>/*.vhd")

vu.main()
```

# VUnit Testbenches

VUnit testbenches are defined in the HDL. The testbenches require specific libraries and generics to interface with VUnit.

VUnit automatically recognizes tests when scanning the source files.

```
library vunit_lib;
context vunit_lib.vunit_context;

entity test_tb is
  generic (
    runner_cfg : string
  );
end entity test_tb;

architecture sim of test_tb is
begin

  seq: process
   test_runner_setup(runner, runner_cfg);
    if run("test_case_1") then
    end if;
   test_runner_cleanup(runner);
  end process;
end architecture;
```

# Formal Verification tools for VHDL

A combination of open source tools is necessary to perform FV on VHDL code.

A full workflow uses GHDL as VHDL interpreter, Yosys as a synthesis engine and SymbiYosys as the FV front-end.

While already capable, the tools are still in development and can require a bit of tinkering to get the most out of them.

Using Docker is suggested: **docker pull hdlc/formal**

# Avalon ST specification

Avalon ST interfaces have 5 signals:

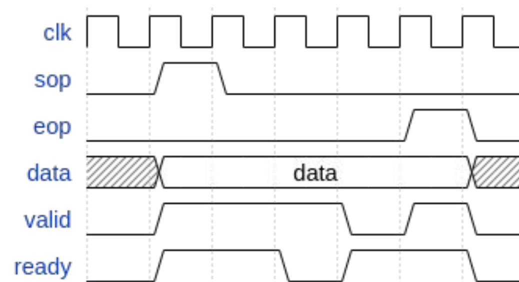
- `StartOfPacket` (source->sink)
- `EndOfPacket` (source->sink)
- `Data` (source->sink)
- `Valid` (source->sink)
- `Ready` (sink->source)

A packet transmission is defined by a `SOP` asserted for 1 cycle and it's closed by an `EOP` asserted for 1 cycle.

If the packet is only one word long, `SOP` and `EOP` are asserted at the same time.

The signals `SOP`, `EOP`, `Data` are valid only if `valid` is asserted. They must be ignored otherwise.

The `ready` signal is driven low by the DUT when it is not ready to process incoming data and should block the source, without corrupting the input.



# PSL: Avalon ST

VHDL processes can be put inside PSL files to model external signals used for assumptions and assertions.

```
vunit test_name(entity_name(rtl))
{

default clock is rising_edge(clk);

assume {rst};

type T_PKT is (IDLE, SINGLE, MULTI);
signal s_snk_pkt : T_PKT := IDLE;

.
.
.
.
}
```

Toolchain supports only clocked operation, the clock has to be declared.

First assumption:
Assume the DUT is reset.

This means that `rst` is being triggered during the first cycle.

Additional types and signal can be added to expand and simplify the verification.

# PSL: Avalon ST

PSL statements can be specified in the VHDL directly.

For an easier integration, it's preferred to declare them in a separate .psl file, specifying a *vunit* test, targeting a specific entity.

```
p_snk_pkt : process (clk)
begin
   if rising_edge(clk) then
      if rst = '1' then
         s_snk_pkt <= IDLE;
      else
         if (snk_valid = '1' and snk_ready = '1') then
            if(snk_sop = '1' and snk_eop = '0') then
               s_snk_pkt <= MULTI;
            elsif(snk_sop = '0' and snk_eop = '1') then
               s_snk_pkt <= IDLE;
            elsif(snk_sop = '1' and snk_eop = '1') then
               s_snk_pkt <= SINGLE;
            end if;
         end if;
      end if;
   end if;
end process;
```

# PSL: Avalon ST

Assumptions model the floating signals (inputs). Contradictory assumptions are detected by the solver (PRE-UNSAT).

Assertions should be named. The solver will try to prove them wrong.

```
-- packet format assumptions for input driver
-- if the packet is a multi word packet, no sop during the packet
assume always (s_snk_pkt = MULTI) -> (not snk_sop);
-- no valid or ready if there's no sop and it's not a multi word packet
assume always (not snk_sop and s_snk_pkt /= MULTI) -> (not snk_valid and
not snk_ready);



-- TESTS on output packets
-- if the out is IDLE and src is ready and valid, a sop must be present
t_valid_out_sop : assert always (not rst and s_src_pkt = IDLE and
src_ready and src_valid) -> (src_sop);
-- if the out is a MULTI word and src is ready and valid, a sop must NOT
be present
t_valid_out_multi : assert always (not rst  and s_src_pkt = MULTI and
src_ready and src_valid) -> (not src_sop);
-- if there is sop eop ready and valid, the out must be a single packet
(on the next clock)
t_valid_out_single : assert always (not rst and src_ready and src_valid
and src_sop and src_eop) |=> (s_src_pkt = SINGLE);
```

# SBY

The .sby file configures SymbiYosys to run the FV.

Run the FV by:

sby --yosys "yosys -m ghdl" \ -f /path/to/file.sby

```
[tasks]
bmc

[options]
bmc: mode bmc
bmc: expect pass
bmc: append 1
bmc: depth 10

[engines]
smtbmc

[script]
ghdl --std=08 VHDL_FILES PSL_FILE -e entity_name
prep -top entity_name

[files]
PSL_FILES
VHDL_FILES_IN_COLUMN
```

BMC stands for Bounded Model Checking

Number to cycles to append to the counter-example

Number to cycles to run the solver for

# Interpreting the results

If the job is successful, the tool will report that the test has passed.

If the tool has found any **counterexample** to the specifications we gave, it will:

- create a **waveform file** with all the cycles that led to the failure.
- create a **SystemVerilog testbench** to with the stimuli used to reach the failure state.