LHCb-PUB-2025-009
May 8, 2025

Guidelines for FPGA Gateware Development in LHCb

Alberto Perro[1,2], Mitja Vodnik[1], Paolo Durante[1], Guillaume Vouters[3], Federico Alessio[1]

[1] *CERN, Geneva, Switzerland*
[2] *Aix Marseille Univ, CNRS/IN2P3, CPPM, Marseille, France*
[3] *Laboratoire d'Annecy de Physique des Particules, Annecy, France*

Abstract

This technical note outlines best practices and methodologies for FPGA development, with a focus on coding standards, verification techniques, and reusable design components tailored for LHCb gateware community. It introduces coding guidelines to ensure consistency, readability, and maintainability in FPGA designs, followed by an exploration of simulation methods and formal verification techniques to guarantee functional correctness and comprehensive coverage. The *colibri* library is presented as a standardized collection of reusable components, enabling portability and reliability across diverse applications. Additionally, the integration of testbenches into Continuous Integration pipelines is discussed, providing automated testing and feedback to maintain code quality throughout the development lifecycle. By combining these approaches, the technical note aims to establish a robust framework for efficient and reliable FPGA development, fostering collaboration and innovation within the LHCb community.

1 Introduction

This technical note provides a comprehensive guide to FPGA development practices intended to support and standardize future developments, focusing on coding standards, verification methodologies, and reusable design components tailored for LHCb gateware development. It begins with coding guidelines for FPGA designs in Section 2, emphasizing standardized practices to ensure readability, maintainability, and collaboration. Section 3 explores simulation techniques, particularly Register Transfer Level (RTL) simulations, as a traditional method for verifying gateware functionality. Formal Verification (FV) is introduced in Section 4, highlighting its advantages and optimization techniques. Section 5 presents the *colibri* library, a standardized collection of reusable components designed to enhance portability and reliability in FPGA designs. Finally, Section 6 discusses the integration of testbenches into Continuous Integration (CI) pipelines, ensuring automated testing and feedback to maintain code quality throughout the development process. Together, these sections provide a structured framework aimed at enabling efficient and reliable FPGA development for future projects in LHCb.

2 Codebase

This section outlines the coding guidelines for FPGA designs in LHCb. Section 2.1 details file structure, formatting, and naming conventions. Architectural design methods aimed at improving readability and debugging are discussed in Section 2.2. Finally, Section 2.3 emphasizes the importance of documentation for ensuring clarity and maintainability.

2.1 VHDL Guidelines

Adhering to standardized coding guidelines is crucial for maintaining readability, consistency, and efficiency in software development. By following conventions for indentation, naming, and commenting, developers can ensure that their code is easier to understand and collaborate on, especially in team environments like LHCb, where multiple programmers contribute to the same project. Consistent coding practices also reduce the likelihood of errors and improve maintainability, enabling future developers to debug or modify the code more effectively. Ultimately, well-structured and standardized code results in higher-quality software that is easier to develop, test, and maintain.

The VHDL Style Guide [1] tool can automatically enforce the following coding style rules using a YAML configuration file, which is available in the GitLab Repository.

File Rules

1. **File Naming** (mandatory): The name of the file must match the name of the design unit contained in the file followed by the `.vhd` or `.vhdl` extension.

2. **File Extension** (recommended): The file extension should be `.vhdl`. All tools and OSs support more than 3 letters extension and `.vhd` is already used for virtual hard disk.

3. **File Content** (recommended): A VHDL File should contain either:

- An entity and its architecture, the entity must not have multiple architectures.
- A package declaration, if no body is required.
- A package declaration and package body.

4. **File Header** (mandatory): Each source file has to start with a comment to describe the content of the file and the license. Doxygen syntax should be embraced to automate the generation of documentation. Header template:

```
--! @title Round-Robin Arbiter
--! @file arbiter.vhdl
--! @author Alberto Perro (alberto.perro at cern.ch)
--! @date 09-01-2024
--! @version 0.1
--! @copyright CERN 2024

--! This module arbitrates a bus connected to a resource.
--! Inspired by https://github.com/chclau/arbiter_rr
--! Release log:
--! - 0.1 first release
--!
--! **License**: GPLv3
```

   The `!` character at the start of a comment highlights the comment for the automated documenter (TerosHDL).

5. **Line Length** (Mandatory): The recommended line length is 80 characters, the maximum is 120 characters.

6. **End of Line** (Mandatory): A single newline `\n` character should be used following the Unix convention.

7. **Language** (Mandatory): Always use the English language in your comments and identifiers.

8. **Avoid Special Characters** (Mandatory): Use always the plain 7-bit ASCII encoding. If special chars are necessary, use their correspondent hex encoding.

9. **No tabulation** (Mandatory): Horizontal tabulations are not allowed. Indentation must be done with spaces.

10. **Last line in a File** (Mandatory): The last line must finish with an end of file and must not be empty.

11. **Trailing Spaces** (Mandatory): No trailing spaces are allowed.

Format Rules

1. **Comment style** (Mandatory): Comments should be placed on dedicated lines, although short comments may be added at the end of a line to provide hints. Comments must start with `--` followed by a white space, except when a `!` immediately follows the `--`, which serves as an identifier for the documenter.

2. **Indentation** (Mandatory): Indentation must be done using two white spaces and applied to:
   - Declarative items,
   - Nested concurrent or sequential statements, and
   - `use` clauses of a library.
3. **Spaces** (Mandatory): There must be at least one white space:
   - Before and after `:` in declarations,
   - Before and after `:=` and `<=` in assignments and default values,
   - Before and after comparison operators, and
   - Before and after `=>` in named associations.

   The number of white spaces can be more than one for alignment purposes. However, there must be **no** white space:
   - Before `,` and `;`,
   - Before `(` when used for conversion, function calls, indexing, or slicing, and
   - Between `process` and `(`.
4. **Context clauses** (Mandatory): Context clauses must be organized into groups, starting with a library clause followed by `use` clauses and a blank line. The library clause should be omitted for `std` and `work`. Each `library` clause must contain only one library name, while each `use` clause must contain only one selected name. A `use` clause should make an entire package visible, written as the library name followed by the package name and `all`. The `std.textio` clause must be the last `use` clause in the `ieee` group, if present. The first group should be for the `ieee` library, followed by vendor libraries, project libraries, and finally `use` clauses for the `work` library.
5. **Use of context keyword** (Mandatory): Due to incompatibility with Xilinx Vivado, the use of the `context` keyword is not allowed in files used for synthesis. However, simulation-only files such as testbenches may use it.
6. **Entity declaration layout** (Mandatory): An entity declaration must follow this layout: `entity`, white space, entity name, white space, `is`, and a new line. If there are generic interfaces, they must be declared one per line, aligned to the group if separated by comments. The `(` should be on the same line as the `generic` keyword, followed by a new line, and the `)` should be on a new line with correct indentation, followed by `;` and a new line. These rules also apply to port declarations.
7. **Structures** (Mandatory): Each structure requiring an `end` keyword (e.g., `if`, `for`, `while`, `entity`, `procedure`) must be closed with the `end` keyword followed by the name of the structure and, if present, the label of the structure. Loops are an exception, as they are closed with the `loop` label. For example:

```
for i in val range loop
...
end loop;
```

```
    procedure pack
    (signal arg : std_logic) is
    begin
    ...
    end procedure pack;
```

8. **Instantiation** (Mandatory): For component or entity instantiation, generics and ports must be associated by name, following the order of declaration, one per line, with arrows (`=>`) aligned. The label and instantiated unit must appear on the first line, followed by generics (if present) on a separate line, and then ports on another separate line.

9. **Process label** (Mandatory): Each process must have a label that clearly explains its purpose, with a brief description provided in a comment just before the declaration.

10. **Parenthesis** (Mandatory): Parentheses must be used in expressions to make the evaluation order explicit, and conditions in `if` and `while` statements must be enclosed in parentheses.

Identifiers

1. **Keywords case** (Mandatory): Keywords case (VHDL reserved identifiers) must be written in lower case.

2. **Identifiers case** (Mandatory): Identifiers should be written in lower case. The only exceptions are generics and constants, which should be written in upper case. Identifiers should use underscores for multi-word naming (e.g., `this_is_multi_word`); no `camelCase` or `PascalCase` is allowed.

3. **References** (Mandatory): References must use the same case as the object they are referring to.

4. **Architecture names** (Mandatory): Only two architecture names are allowed: `rtl` should be used for synthesizable code, while `sim` must be used for simulation-only code.

5. **Constants names** (Mandatory): Constants declarations should be in UPPER case with a `c_` prefix.

6. **Generics names** (Mandatory): Generic identifiers should also be in UPPER case, with the `g_` prefix.

7. **Ports names** (Mandatory): Port names must be in lower case and should be followed by a suffix specifying the direction of the port:

    - `_i` for inputs,
    - `_o` for outputs, and
    - `_io` for bidirectional ports.

8. **Variables names** (Mandatory): Variables must have the `v_` prefix.

9. **Types names** (Mandatory): User-declared types and subtypes must have the `_t` suffix.

10. **Signals name** (Mandatory): Clock signals should have the `clk_` prefix and the direction suffix. In case of multiple clock domains, the domain name and the frequency (if known) should be specified. Reset signals should have the `reset_` prefix and the direction suffix. In case of multiple clocks, there should be one reset per clock, and the reset name should report the corresponding domain name. List of suffixes:

    - Active-low signals should have a `_n` suffix,
    - Asynchronous signals should have a `_a` suffix,
    - Delayed signals should have a `_d` suffix. If delays are concatenated, use an index (e.g., `_d1`, `_d2`, ...) or a vector.

    The order of suffixes should follow alphabetical order, except for delays or port directions, which should be the last ones.

11. **Process labels** (Mandatory): Process labels should have the `proc_` prefix.

12. **Generate label** (Mandatory): Generate labels should have the `gen_` prefix.

Language Rules

1. **VHDL Version** (Mandatory): Synthesizable units must follow the VHDL-2008 standard. The only exception is rule Format.5.

2. **IEEE Packages** (Mandatory): The only IEEE packages allowed are:

    - `std_logic_1164`,
    - `numeric_std`,
    - `numeric_bit`,
    - `math_real`,
    - `math_complex`,
    - `std_logic_misc` (only the reduce functions), and
    - `std_logic_textio`.

    In particular, `std_logic_unsigned`, `std_logic_signed`, and `std_logic_arith` are **not** allowed. Use `numeric_std` instead.

3. **Attribute declarations** (Mandatory): It is not allowed to declare custom attributes, except for tool-specific attributes.

4. **Enumerations** (Mandatory): All user-defined enumerations must use names for literals. Characters are not allowed.

5. **Buffer, linkage, guarded signals** (Mandatory): `bus`, `register`, `buffer`, and `linkage` are not allowed.

6. **Block statements** (Mandatory): Block statements can be used to group concurrent statements. No ports, generics, and implicit guards are allowed.

7. **Configurations** (Mandatory): Configurations are not allowed.

Synthesis Rules

1. **Type of top-level ports** (Mandatory): The type of the ports in a top-level entity must be `std_logic`, `std_logic_vector`, `unsigned`, or a user-defined bounded composite type (array or record) composed of these types.

2. **Type of top-level generics** (Mandatory): The type of generics in a top-level entity must be `string`, `integer`, `boolean`, `std_logic`, `std_logic_vector`, an enumerated type, or a user-defined bounded composite type of these types.

3. **Wrapper of top-level units** (Recommended): Bus signals should be grouped into records to reduce the number of connections. There can be two versions of the top-level unit: a wrapped one and a non-wrapped one. The name of the wrapped unit is the name of the normal one, but with the `_x` prefix. Each bus should have two records: one for the input signals and one for the output signals. The records should be declared in a package. The default top-level entity should be the wrapped version, while the unwrapped version should only unwrap the signals.

4. **Process for a register** (Recommended): Use only registers triggered on the rising edge of the clock and with a synchronous reset.

```
proc_reg : process (clk) is
begin
    if rising_edge(clk) then
        if (reset = '1') then
            q <= '0';
        else
            q <= d;
        end if;
    end if;
end process proc_reg;
```

5. **Asynchronous reset** (Mandatory): If reset is asynchronous, it must be synchronously deasserted.

6. **Register reset** (Mandatory): All registers **must** be initialized during reset.

7. **Signal attributes** (Mandatory): Do not use time-related signal attributes for synthesis (`event`, `active`, `delayed`). Use the function `rising_edge(clk)` or `falling_edge(clk)`, if necessary.

8. **Index direction** (Mandatory): Subtypes of `std_logic_vector`, `signed`, and `unsigned` must always use the `downto` direction. `integer` and array indexes can use the `to` direction.

9. **Array size** (Mandatory): Array length should be 2 or more if the size is not computed.

10. **Clock and reset ports** (Mandatory): The main clock should be the first port of the list, and the main reset must be the second port.

11. **Usage of clocks** (Mandatory): Logic on clocks is not allowed.

12. **FSM code style** (Recommended): If all outputs are a function of the current state, the FSM can be written using only one process (Moore). In other cases, the FSM can be written with two processes (Mealy): one handling the register, and the second computing states and outputs.

13. **Unused Ports map** (Mandatory): Unused ports should be assigned to `open` signals.

2.2 Architecture

This section is based on the work of Jiri Gaisler [2].

The *dataflow* style is a traditional approach for synthesizable VHDL models, relying on numerous concurrent statements and small processes connected through signals. While effective, this style can be difficult to read and understand, as execution depends on signal changes rather than the order of written statements. Engineers often need to create block diagrams to understand dependencies, making maintenance challenging, especially in large designs like memory controllers. Additionally, dataflow VHDL has a low abstraction level, relying on basic constructs like multiplexers and bitwise operators, which makes recognizing complex algorithms difficult. Simulation performance is also inefficient, as signal assignments are slower than variable assignments due to event scheduling overhead.

The *two-process* coding method is proposed as an alternative for synchronous single-clock VHDL designs. This method improves readability, abstraction, debugging, and simulation efficiency by:

- Using record types for ports and signals.
- Limiting each entity to two processes: one for combinational logic and one for sequential logic.
- Employing high-level sequential statements for algorithm encoding.

Unlike traditional VHDL, where concurrent statements complicate readability as designs grow, the two-process method structures logic sequentially, similar to standard programming languages like C. This approach simplifies analysis, debugging, and maintenance while maintaining a single model for both synthesis and simulation.

Listing 1 demonstrates the Run Length Encoder from *colibri* written using this architectural style.

Listing 1: Run Length Encoder

```
  architecture rtl of rle_decode is

  constant c_WORD_MAX : positive := 2 ** g_COUNT_WIDTH - 1;

  type sig_t is record
    word_cnt : natural range 0 to c_WORD_MAX;
    buf      : snk_data_i'subtype;
    odata    : src_data_o'subtype;
    valid    : std_logic;
    ready    : std_logic;
  end record sig_t;

  signal rreg, rcmb : sig_t;

begin

  rreg <= rcmb when rising_edge(clk_i);

  proc_encode : process (all) is

    variable v_int : sig_t;

  begin

    v_int := rreg;

    -- acknowledge a read
    if (src_ready_i = '1' and rreg.valid = '1') then
      v_int.valid := '0';
    end if;

    if (v_int.valid = '0' and rreg.word_cnt /= 0) then
      v_int.valid    := '1';
      v_int.odata    := rreg.buf(g_WORD_WIDTH - 1 downto 0);
      v_int.word_cnt := rreg.word_cnt - 1;
    end if;

    v_int.ready := not v_int.valid;

    -- if input is valid and no word is buffered
    if (snk_valid_i = '1' and v_int.valid = '0') then
      v_int.buf      := snk_data_i;
      v_int.word_cnt := to_integer(unsigned(snk_data_i(snk_data_i
         'left downto g_WORD_WIDTH)));
      -- forward the current word
      v_int.valid    := '1';
      v_int.odata    := snk_data_i(g_WORD_WIDTH - 1 downto 0);
      v_int.word_cnt := v_int.word_cnt - 1;
    end if;
```

```
    if (reset_i) then
      v_int.word_cnt := 0;
      v_int.buf      := (others => '0');
      v_int.valid    := '0';
      v_int.ready    := '0';
    end if;

    rcmb <= v_int;

  end process proc_encode;

  src_data_o  <= rreg.odata;
  src_valid_o <= rreg.valid;
  snk_ready_o <= rcmb.ready;

  end architecture rtl;
```

While not mandatory, adopting this architectural style is highly recommended, particularly for large designs where debugging can become complex.

2.3 Documentation

Clear and comprehensive documentation is vital for improving code readability, maintainability, and collaboration among developers. Most documentation can be included directly in comments, ensuring that explanations remain close to the relevant code sections. Structured comments help future developers understand the design and functionality, simplifying debugging and modifications.

In VHDL, the `--!` comment keyword can be used to generate automatic documentation with tools like Doxygen [3], providing a systematic way to describe modules, ports, and functionality. Additionally, tools like WaveDrom [4] enable easy waveform visualization, clarifying timing and signal behavior. Integrated development environments such as TerosHDL [5] can further automate documentation generation.

3 Simulations

Simulation is a widely used and traditional method for verifying gateware functionality. This section focuses on Register Transfer Level (RTL) simulations, which provide cycle-accurate behavioral information while remaining independent of implementation details. A typical testbench, illustrated in Figure 1, consists of three main components: the Device Under Test (DUT), which is the component being verified; a stimuli generator, which provides inputs to the DUT; and a checker, which compares the DUT's outputs against expected values. The designer is responsible for developing test cases and providing a quantitative measure of test coverage.

To ensure effective verification, the following generic guidelines are recommended when designing a verification plan:

- **Test Minimal Components:** Focus on verifying the smallest functional units possible. Testing isolated sub-components simplifies test case development and

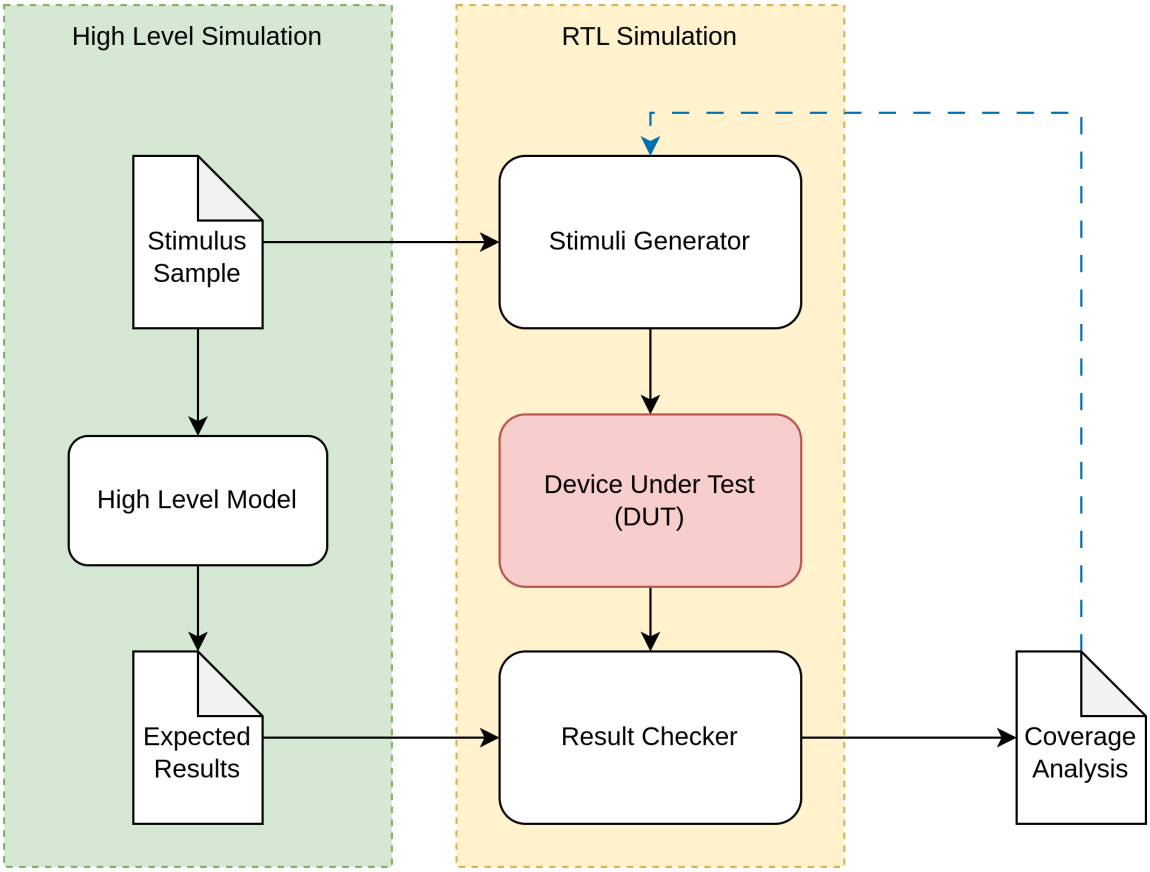


Figure 1: An example diagram of a functional verification testbench. The primary components of the simulation are highlighted in yellow. Stimuli generators and result checkers are often implemented in HDL code, while high-level simulation is typically implemented using languages like SystemC or C.

reduces complexity. Rigorous verification of these smaller units facilitates the subsequent verification of the complete design. Additionally, organizing tests based on specifications improves bug tracking and collaboration by making the tests easier to understand.

- **Cover Normal Operational Modes:** Ensure that the design performs as expected in all operational modes under normal conditions, as specified in the functional requirements. For larger or parameterizable designs, test all combinations of normal modes to verify the correct interaction between modules. The system should always return to an idle state after the tests.

- **Test Exception Conditions:** Validate the system's behavior under abnormal or exceptional conditions. This includes testing the system's recovery capabilities when operating outside normal modes. Tests should cover illegal conditions and protocol violations, ensuring the system can gracefully handle unexpected scenarios.

Functional Coverage Coverage is a quantitative metric used to measure the completeness of the verification effort. It is typically expressed as the percentage of verification objectives met. Two primary types of coverage metrics are used:

- **Code Coverage:** This metric, derived directly from the code, indicates whether and how frequently specific lines of code are exercised during testing. While useful, it does not provide insights into the context or purpose of the exercised code.

- **Functional Coverage:** Defined by the designer, this metric ties verification to the design's intended functionality. It measures whether specific scenarios, corner cases, design requirements, and other critical conditions of the DUT have been observed, validated, and tested.

To design an effective functional coverage testbench, methodologies such as *Constrained Random Testing* can be employed. Instead of manually crafting each test case, pseudo-random stimuli generators can automatically create a set of test cases based on general specifications. The testbench is equipped with checkers to verify that coverage goals are met. A high-level diagram of this methodology is shown in Figure 2. Verification frameworks, discussed in Section 3.1, can simplify this process by providing integrated utilities and dedicated components. An example of this methodology can be found in the FIFO testbench available in the *colibri* library [6].

Constrained random stimuli can also be produced outside of the VHDL testbench. Stimuli data can be provided from Monte-Carlo simulations and external synthetic data generators. For example, Python frameworks such as *Hypothesis* [7] can be used to generate constrained random data based on specification constraints. If this approach is used, the developers should also provide an external checker and detail the data formats for both the stimuli input and the DUT output.

Transaction Level Modeling FPGA designs often involve interactions between components over interfaces, with signals divided into data and control paths. These interactions, defined by a *protocol*, consist of specific *transactions* that dictate component responses based on control signal combinations.

Abstracting these interactions into interfaces enables higher-level modeling, known as Transaction Level Modeling (TLM). TLM separates communication details from the implementation specifics of functional units, simplifying the representation of complex systems.

Most FPGA designs use two primary categories of data communication mechanisms:

- **Streaming Interfaces:** These are point-to-point links where the transmitter is referred to as the source (or master) and the receiver as the sink (or slave). Transactions on a streaming interface are defined by a basic *handshake* mechanism: the source drives the data along with a *valid* signal, indicating that the data is ready to be read. The sink responds with a *ready* signal to indicate its readiness to receive the data. Additional control signals can transform the communication from a cycle-based approach to a packet-based one, where a single packet is transmitted over multiple clock cycles.

  Standard streaming interfaces include AXI-Stream (used by AMD Xilinx) and Avalon Stream (used by Altera). An example of a streaming transaction is shown in Figure 3a.

- **Memory-Mapped Interfaces:** These interfaces are used for bus communication and consist of a bidirectional data path, an address path, and control signals. A host

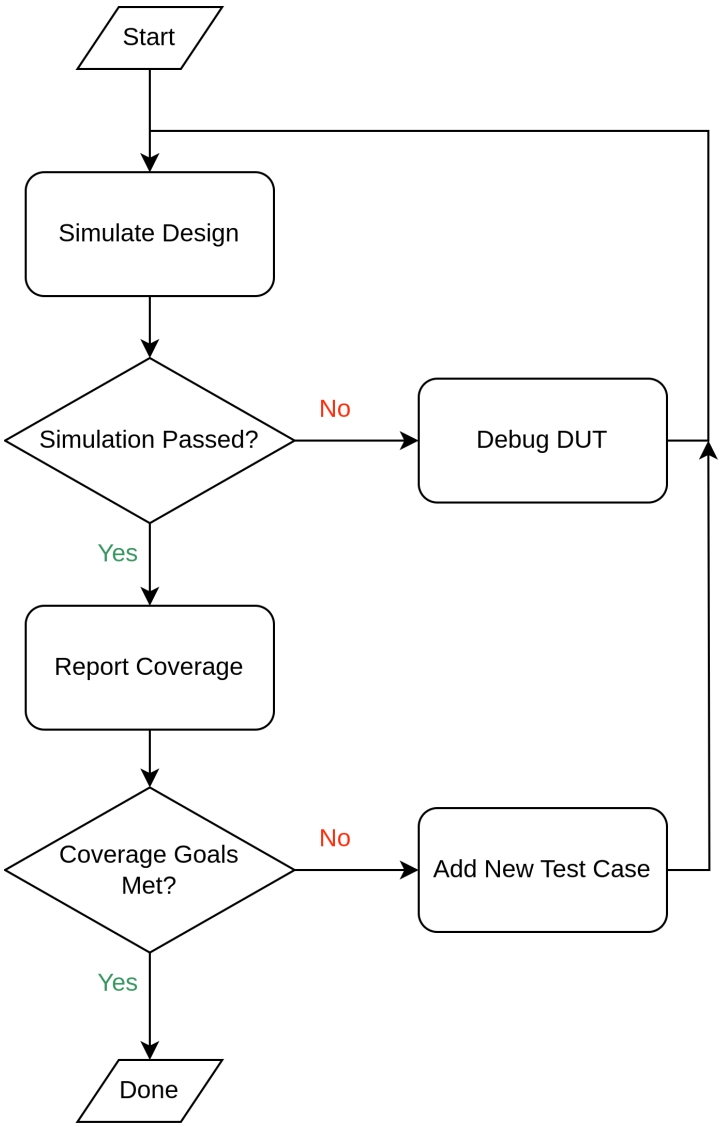


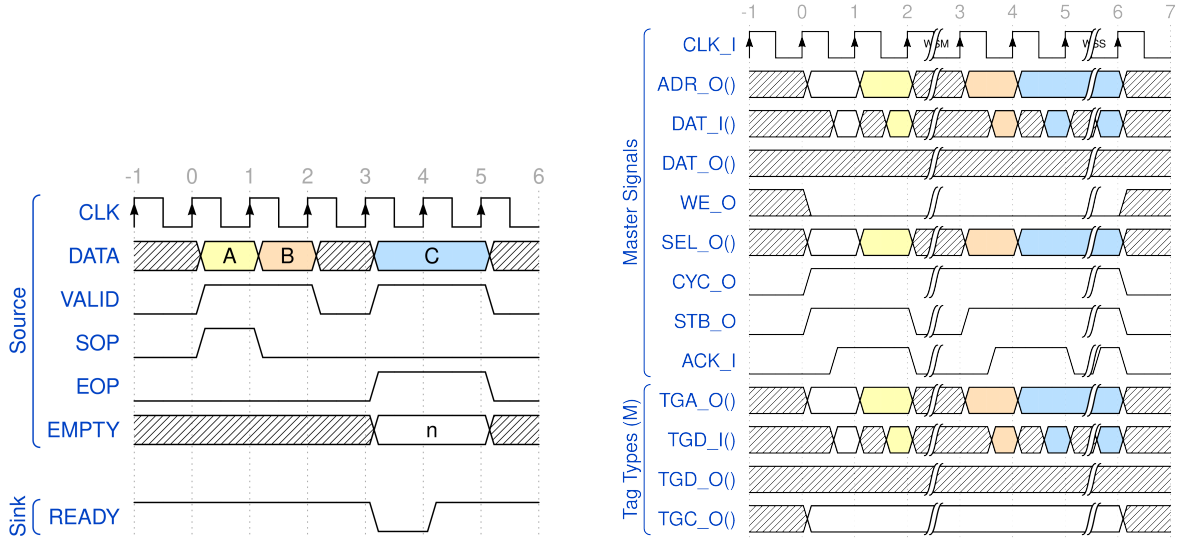
Figure 2: Flow chart of the constrained random test methodology.

orchestrates transactions across multiple peripherals. Unlike streaming interfaces, memory-mapped interfaces include an address path to specify the physical location of a register, memory, or resource. The use of a bus allows multiple peripherals to share the same signals, saving resources.

Basic memory-mapped interfaces support read and write operations, where each operation specifies the address and control signals. Extended features, such as pipelined burst transfers or variable latency support, can add complexity.

Common memory-mapped protocols include AXI4, AXI4-Lite, Avalon Memory-Mapped, and Wishbone Bus. An example of a memory-mapped transaction is shown in Figure 3b.

Verification frameworks often provide functional models for these standards, allowing developers to import models into testbenches and connect them to the appropriate interfaces. These frameworks integrate seamlessly with constrained random testing,



(a) Example of an Avalon Streaming packet transaction. SOP and EOP signal the start and the end of the packet, respectively. EMPTY indicates how many symbols are valid in the last data word.

(b) Example of a Wishbone Block Read transaction. CYC_O high signals the transaction is happening. The operation spans over five clock cycles.

Figure 3: Example waveforms of interfaces.

further simplifying and enhancing the verification process.

3.1 Verification Frameworks

VHDL-based designs rely on several open-source verification frameworks. While these frameworks offer similar functionalities, they differ in implementation focus and methodology. Designers can choose the most suitable framework for their application or combine multiple frameworks within the same testbench.

UVVM [8] is a free and open-source framework for developing structured, reusable testbenches. Supported by industry and research institutions like the European Space Agency, UVVM is fully implemented in VHDL-2008, making it compatible with a wide range of simulators. Its standout feature is the VHDL Verification Components (VVCs), which extend the TLM concept by autonomously handling transaction scheduling and component implementation. VVCs support complex protocols, including packet-based streams and out-of-order communication. However, their complexity results in a steeper learning curve, particularly for customization.

OSVVM [9] is another free and open-source framework. It offers features such as verification components, functional coverage, and constrained random tools. Its strengths include intuitive pseudo-random generators and powerful scoreboards. However, its Tcl-based scripting interface may feel outdated compared to modern alternatives.

VUnit [10] provides a modern approach to HDL verification, with a Python-based scripting interface. Its test runner automates test discovery, file scanning, and compilation

order management, simplifying test design and organization. Python scripting enhances portability and testbench development.

A comparison of these frameworks is summarized in Table 1.

	UVVM	**OSVVM**	**VUnit**
Checkers and Loggers	yes	yes	yes
Verification Components	excellent	good	basic
Randomized Coverage	basic	excellent	not available
Scoreboards	good	excellent	not available
Scripting	not available	poor (Tcl)	excellent (Python)

Table 1: Comparison of key features among the three major open-source VHDL verification frameworks.

3.2 Testbench Design

Testbenches should be all designed using the VUnit python framework which allows portability between simulators and simplifies dependencies management. For simple components and VHDL functions, a single testbench file is sufficient. These testbenches generate randomized stimuli and verify results against expected values, leveraging utility and random functions provided by OSVVM and UVVM.

For more complex components, a modular testbench structure is recommended, following UVVM's guidelines. This structure consists of three core elements:

- **Test Package:** Contains common constants, functions, and procedures used across the testbench. It also defines types and subtypes for custom interfaces, such as clock periods and data widths.
- **Test Harness:** A non-synthesizable entity responsible for instantiating and interconnecting all components used in the tests, including the DUT and auxiliary entities. The harness entity is designed without ports, ensuring all signal sources and sinks are contained within the entity.
- **Test Bench:** Hosts the test procedure itself. It instantiates the test harness and controls its behavior through a single sequencer process. Commands are dispatched via the framework's internal communication network to the various TLM entities in the test harness.

This modular architecture, shown in Figure 4, ensures a clear separation of implementation details from the test procedure. High-level TLM commands are managed in the sequencer process, while low-level signal handling resides in a separate file. This approach enhances test clarity and facilitates reusability.

4 Formal Verification

Formal Verification (FV) is a mathematical approach used to analyze the entire space of possible behaviors of a design. It can be applied to both software and hardware problems and is useful at various stages of the development process.

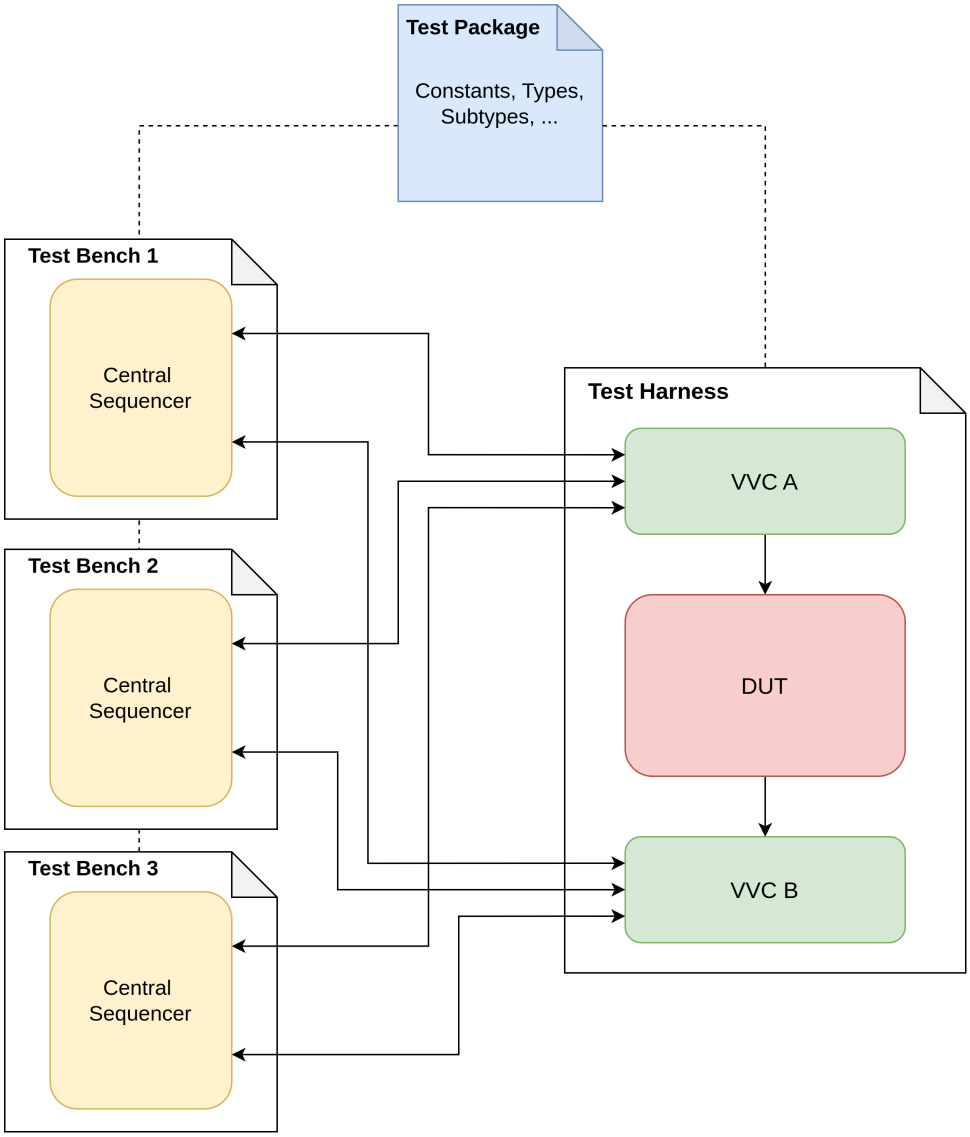


Figure 4: Modular testbench structure. This implementation allows the reuse of the test package and the harness across multiple testbenches. For instance, VVC A stimulates the DUT by generating the low-level signals from the sequencer commands. VVC B receives the output signals of the DUT and transmits them to the sequencer for checking.

Like simulation, FV employs tools to ensure that a design satisfies the properties and requirements of its specification. However, the methodology differs fundamentally: while simulation evaluates individual test cases, FV explores the entire design space at once. This does not mean that FV executes all possible simulations; instead, it uses mathematical techniques to efficiently compute and simplify the behavior of the design.

FPGA designs are particularly well-suited to FV because they are deterministic digital systems defined by Boolean logic. Although Boolean Satisfiability Problems (SAT) are NP-complete—meaning the time required to solve them grows exponentially with input size—clever strategies can significantly reduce problem complexity, making FV a practical approach. Figure 5 illustrates the difference in coverage between simulation and FV.

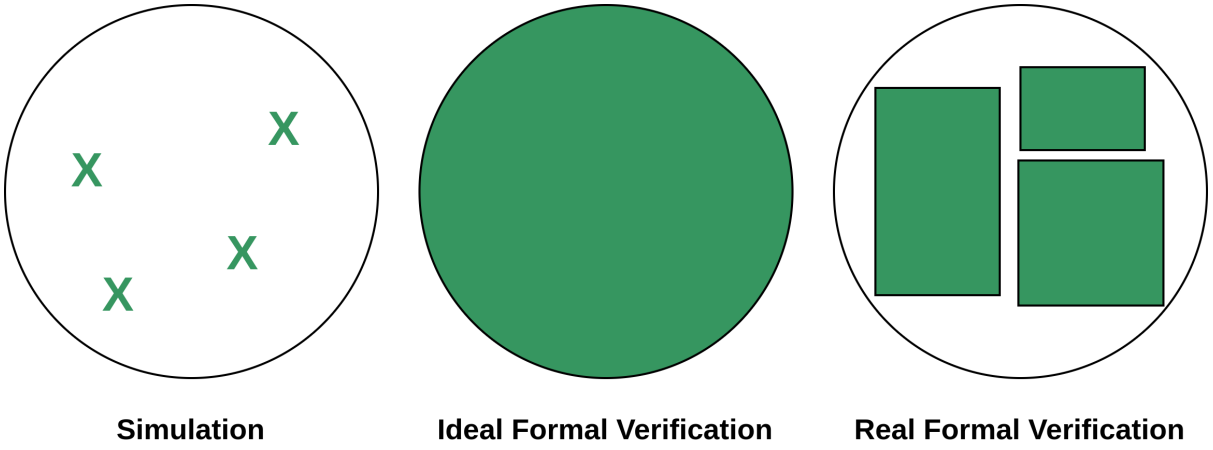


Figure 5: Coverage level of formal verification compared to simulation: simulation covers only specific points, while formal verification aims to cover the entire design space. In practice, full coverage is achievable only in certain areas.

Bounded Model Checking Bounded Model Checking (BMC) is the most widely used FV technique. In BMC, the design is represented as a finite-state system, and its properties are expressed in temporal logic. The tool translates the design into a logical expression that models all possible state transitions from the initial state. A finite bound of steps is specified by the user, limiting the scope of verification to a manageable range.

A counterexample is found when a combination of variables falsifies the logical expression. If no counterexample exists within the specified bound, the tool guarantees that the design satisfies the property for all states within that range. It is the user's responsibility to determine whether the chosen bound is sufficient for verification closure.

Property Specification Languages To perform FV, the properties of the design must be specified using a Property Specification Language (PSL). PSL enables precise definitions of expected behavior and verification methodology. Properties are categorized as:

- **Assertions:** Define the allowed behavior of the DUT.
- **Assumptions:** Specify constraints on the inputs to the DUT.

PSL assertions are more expressive than those used in simulation, as they can incorporate temporal logic to describe how conditions evolve over time. IEEE PSL [11] is a generic property specification language that supports multiple HDL languages, including VHDL. For example, a complete FV testbench for the FIFO in the *colibri* library can be implemented in fewer than 20 lines of PSL, whereas an equivalent constrained random simulation requires at least ten times as much code without guaranteeing comprehensive coverage.

Optimization Techniques In computational theory, FV is an *NP-hard* problem, meaning that the time and memory required to compute FV tests grow exponentially with the size of the design. However, several techniques can reduce problem complexity and improve tool efficiency:

- **Reducing Problem Size:** FPGA designs often involve wide data paths and large memories. If no constraints are provided, the tool will attempt to model all possible combinations of these bits, leading to an explosion of states. Reducing data widths and memory sizes to a few bits typically preserves the design's functionality while allowing the tool to catch most bugs. If reducing data widths is not feasible, limiting the search space to specific data patterns (e.g., most bits being 0) can also help.

- **Decomposing Tests:** Complex properties can be broken down into simpler, independent ones. Similarly, independent behaviors of the DUT can be tested separately. This decomposition reduces the complexity of individual tasks, enabling tools to manage them more efficiently and significantly reducing computation time and memory usage.

Formal Verification Tools An open-source toolkit is available for performing FV, consisting of the following components:

- **Yosys:** An HDL synthesizer.
- **GHDL:** A VHDL interpreter.
- **SymbiYosys (SBY):** A formal verification frontend.

These tools are often packaged together in pre-configured Docker images, such as hdlc/formal, simplifying setup and usage.

An FV testbench typically consists of three essential files:

- **DUT:** The design under test.
- **PSL Specification:** A file (usually with a `.psl` extension) containing the verification unit, where assertions and assumptions are defined. Additional glue HDL code can be included to encode more complex behaviors.
- **SBY Configuration:** A file specifying the configuration of the FV toolkit and listing the files required to synthesize the design.

Many examples of FV testbenches can be found in the *colibri* library, demonstrating how to apply these tools effectively.

5 Common Components Library

Gateware components, often referred to as Intellectual Property (IP), are essential for developing FPGA designs. Many components used in designs are generic and implement reusable functionalities. These generic components are often provided directly by vendors under licensing agreements. However, relying on vendor-provided components can lock developers into a specific ecosystem, limiting flexibility and choice.

To avoid vendor lock-in, many designers choose to develop custom implementations tailored to their specific needs. While this approach provides independence from proprietary solutions, it often lacks thorough verification and proper documentation, leading to challenges such as reduced portability and collaboration.

The *colibri* library [6] was developed to address these limitations, providing a standardized components library for CERN developers. The library ensures functionality and reliability through rigorous verification and validation, employing the techniques discussed in previous sections.

Library Content The *colibri* library consolidates commonly used components for DAQ applications, drawing inspiration from the selection used in the current PCIe40 gateware. The components are designed to be generic and parameterizable, accommodating a wide range of user applications. As of this writing, *colibri* contains 57 components, organized into the following categories:

- **Common:** Entities and packages implementing frequently used functions, such as synchronizers, buffers, and debouncers.
- **Communication:** Entities designed for communication tasks, including scramblers and gearboxes.
- **Encoders:** Components for standard data encoding and decoding, such as Run-Length Encoding.
- **File I/O:** Functions and packages that simplify file operations during simulation.
- **Interfaces:** Components for translating and converting between different interfaces, such as Avalon and AXI.
- **Input/Output:** Basic low-level protocols (I^2C, SPI, UART) for interacting with external sensors and microcontrollers.
- **Memory:** Implementations of FIFOs for single and dual clocks, along with RAM entities.
- **Packet:** Common operations for packet-based streaming interfaces.
- **Pipes:** Components for managing streaming interfaces, such as arbiters, broadcasters, and routers.
- **Protocols:** High-level protocols like Ethernet and Aurora.

The *colibri* library is hosted and maintained on GitLab, provided by the CERN IT department. Developers are encouraged to provide feedback through issues and contribute via merge requests.

6 Continuous Integration

Testbenches developed using the methods described above can be seamlessly integrated into the Continuous Integration (CI) system in GitLab. This integration provides immediate feedback after each commit, ensuring code quality and reliability. Open-source VHDL simulators such as GHDL [12] and NVC [13] are recommended, as they do not require licenses.

A suggested layout for the CI pipeline is the one used by *colibri*. For both simulation and FV, GitLab pipelines capture detailed test artifacts, such as logs and waveforms, in the event of test failures. These artifacts can be downloaded and analyzed for debugging without the need to rerun the tests.

The pipelines run on a distributed computing cluster provided by CERN IT, which is accessible to all FPGA developers at CERN. Each pipeline stage executes within a Docker container preloaded with the necessary tools and frameworks. Pipelines are triggered by new commits to the `devel` and `master` branches, as well as merge requests. To optimize resource usage, tests are executed selectively, only when relevant source or simulation files have been modified.

References

[1] J. Leary, *jeremiah-c-leary/vhdl-style-guide*, https://github.com/jeremiah-c-leary/vhdl-style-guide.

[2] Jiri Gaisler, *A structured VHDL design method*, in *Fault-tolerant Microprocessors for Space Applications*.

[3] *Doxygen*, https://www.doxygen.nl/.

[4] A. Chapyzhenka and J. Probell, *Rendering Beautiful Waveforms from Plain Text*, 2016.

[5] Teros Technology, *terosHDL*, https://github.com/TerosTechnology/vscode-terosHDL.

[6] *colibri*, https://gitlab.cern.ch/colibri/colibri.

[7] D. MacIver, Z. Hatfield-Dodds, and M. Contributors, *Hypothesis: A new approach to property-based testing*, Journal of Open Source Software **4** (2019) 1891.

[8] UVVM, *UVVM/UVVM*, https://github.com/UVVM/UVVM.

[9] OSVVM, *Open Source VHDL Verification Methodology*, https://osvvm.org/.

[10] *VUnit/vunit*, https://github.com/VUnit/vunit.

[11] *IEEE Standard for Property Specification Language (PSL)*, IEEE Std 1850-2005 (2005) 1, Conference Name: IEEE Std 1850-2005.

[12] *ghdl/ghdl*, https://github.com/ghdl/ghdl.

[13] N. Gasson, *nickg/nvc*, https://github.com/nickg/nvc.