# POLITECNICO DI TORINO

Master's Degree in Cybersecurity Engineering


Master's Degree Thesis

# Reproducible Design and Verification Workflow for TRNG Integration in a RISC-V SoC

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Candidate

Giovanni NICOSIA

March 2026

**Abstract**

Open-source hardware ecosystems have enabled complete processor development flows, from HDL design to silicon tape-out. The VE-HEP project demonstrated an end-to-end open ASIC implementation based on VexRiscv, an open-source RISC-V processor written in SpinalHDL, with cryptographic accelerators and side-channel hardening. However, the project explicitly excluded a hardware TRNG due to the lack of available open-source components, leaving entropy generation unaddressed.

This thesis contributes the integration of OpenTRNG's physical TRNG module into a VexRiscv-based System-on-Chip. The Verilog TRNG IP is wrapped as a memory-mapped peripheral within the existing SpinalHDL-based SoC architecture. Bare-metal firmware drivers implement initialization, status monitoring, polling, and entropy acquisition routines. System-level verification exercises complete CPU-to-peripheral transactions, validating both functional correctness and bus integration.

In parallel, the thesis formalizes the hardware design and verification process using Nix, a purely functional system in which builds are described as derivations — pure functions from declared inputs to immutable outputs. Firmware compilation, HDL generation, and simulation are expressed as Nix derivations forming a build-time dependency graph, explicitly capturing tool versions, sources, and build dependencies. This approach ensures hermetic builds and deterministic reconstruction of artifacts across systems and over time.

Together, these contributions provide both a reference implementation for TRNG integration in open RISC-V SoCs and a reproducible, declarative build infrastructure whose applicability extends beyond hardware projects to software development in general. The complete implementation is released as open source to enable independent validation and further development.

# Table of Contents

# Acronyms

**ASIC**
Application-Specific Integrated Circuit

**FPGA**
Field-Programmable Gate Array

**SoC**
System-on-Chip

**HDL**
Hardware Description Language

**RTL**
Register-Transfer Level

**TRNG**
True Random Number Generator

**PTRNG**
Physical True Random Number Generator

**HSM**
Hardware Security Module

**APB3**
Advanced Peripheral Bus, version 3

**RISC-V**
Reduced Instruction Set Computer - Five

**PDK**

Process Design Kit

**DRC**

Design Rule Check

**LVS**

Layout Versus Schematic

**GDSII**

Graphic Design System II

# Chapter 1

# Introduction

**Motivation** Open-source hardware security projects have made end-to-end hardware design flows increasingly accessible, from HDL to RTL simulation and even silicon. A notable example is the VE-HEP project (section 3.1), which demonstrates a fully open-source hardware security module (HSM) flow. However, in that reference design a concrete, physical TRNG is not integrated in the open-source SoC, and randomness is effectively provided by non-physical sources. This leaves a practical gap for systems that must rely on true physical entropy, especially in security-critical settings.

**Contributions** This thesis addresses that gap by integrating a real physical TRNG design into a RISC-V SoC based on VexRiscv. The work uses OpenTRNG's PTRNG (section 3.2) as a black-box component and focuses on system-level integration: bus wrapping, register mapping, reset/clock alignment, firmware control, and end-to-end functional verification. While the entropy source itself ultimately requires hardware characterization, the SoC integration must already ensure correct control, data flow, and alarm propagation.

A second objective is reproducibility. Environment reproducibility is a pervasive problem in software development—the same source can produce different artifacts on different machines—and hardware design flows are no exception. This thesis therefore adopts Nix not merely as a shell environment, but as a full derivation-based build system. The resulting workflow captures toolchain versions, build steps, and dependencies in a reproducible, hermetic way.

The contributions of this work are:

i. SoC integration of OpenTRNG PTRNG into a VexRiscv-based system with a clean APB3 interface, C firmware driver, and functional verification of the full firmware → bus → PTRNG path.

ii. Nix-based build system: every stage (shell environment, firmware, RTL generation, simulation) expressed as a Nix derivation and exposed via Nix Flakes, ensuring hermetic reproducibility.

**Thesis Organization** This thesis follows a parallel structure across Background (chapter 2), Methodology (chapter 4), and Implementation (chapter 5), organized around four recurring technical areas:

- **Build Systems:** Nix package management, Flakes, reproducible workflows (section 2.1, section 4.1, section 5.1)
- **Hardware Design:** HDLs (SpinalHDL), RISC-V (VexRiscv), bus protocols (APB3), ASIC flow (section 2.2, section 4.2, section 5.2)
- **Hardware Security:** TRNG primitives, OpenTRNG PTRNG integration (section 2.3, section 4.3, section 5.3)
- **Simulation & Verification:** Verilator, Cocotb, firmware-driven testing (section 2.4, section 4.4, section 5.4)

For each area, the reader encounters: *what exists* (Background), *why these choices* (Methodology), and *how implemented* (Implementation).
Related Work (chapter 3) interrupts this pattern to provide critical analysis of existing projects (VE-HEP, OpenTRNG) and identify gaps, motivating the thesis contributions before Methodology resumes the parallel flow.

# Chapter 2

# Background

**ASIC Design Flow and Implementation Targets** An Application-Specific Integrated Circuit (ASIC) is a custom-designed silicon chip optimized for a specific computational task, as opposed to general-purpose processors. ASICs achieve superior power efficiency, performance, and area utilization compared to programmable alternatives by implementing only the necessary logic in dedicated silicon. Common applications include cryptographic accelerators, digital signal processors, sensor fusion engines, and hardware security modules. Recent open-source Process Design Kits (PDKs)—such as Google's SkyWater 130nm and IHP's SG13G2 130nm—have democratized ASIC development, enabling academic research projects to fabricate functional silicon.

The ASIC design flow transforms high-level hardware specifications into physical layouts ready for fabrication. This process consists of six main stages:

1. **RTL Design:** Hardware behavior is specified using Register-Transfer Level (RTL) descriptions in Hardware Description Languages (HDLs). RTL abstracts hardware as synchronous state machines with registers, combinational logic, and dataflow operations.

2. **Simulation:** Functional verification validates design correctness through testbenches that exercise the RTL model. Simulation catches logic errors, protocol violations, and corner cases before committing to physical implementation.

3. **Synthesis:** Logic synthesis tools translate RTL behavioral descriptions into gate-level netlists composed of standard cells from a technology library (e.g., AND gates, flip-flops, multiplexers). The synthesis process optimizes for area, timing, and power constraints while mapping abstract RTL operations to physical gates.

4. **Place & Route:** Physical design tools determine the spatial placement of standard cells on the silicon die and route metal interconnects between them.

This stage accounts for physical constraints such as wire delays, signal integrity, power distribution, and clock tree synthesis.

5. **Physical Verification:** Design Rule Check (DRC) validates that the layout satisfies foundry manufacturing constraints (minimum spacing, width, overlap rules). Layout Versus Schematic (LVS) verification ensures the physical layout matches the intended netlist connectivity.

6. **Tape-out:** The final layout is exported as GDSII files (Graphic Design System) containing the complete geometric description of each mask layer. These files are submitted to a semiconductor foundry for fabrication, initiating a 3–6 month production cycle.

ASICs require significant upfront investment—mask set costs range from tens of thousands to millions of dollars depending on process node—and any post-fabrication bug requires a complete redesign and re-fabrication cycle.

Field-Programmable Gate Arrays (FPGAs) mitigate this risk by providing reconfigurable hardware prototyping platforms. The same RTL design written for ASIC tape-out can be synthesized onto FPGA fabric and deployed in hours, enabling rapid design iteration and hardware validation. This validation step is particularly critical for components whose behavior depends on physical hardware characteristics—such as analog entropy sources or precise timing relationships—that cannot be faithfully modeled in RTL simulation alone. FPGA prototyping allows designers to verify functionality on real hardware before committing to expensive and irreversible ASIC fabrication.

This thesis focuses on **RTL design and functional verification** (stages 1–2), producing hardware descriptions suitable for either FPGA prototyping or ASIC fabrication workflows. The methodology establishes a reproducible foundation for future hardware validation and potential silicon implementation.

This chapter provides the technical foundation for understanding the methodological contributions of this thesis:

- **Nix** (section 2.1): Since a reproducible toolchain is central to the work, and existing documentation is broad but fragmented, we provide a practical guide to Nix's core concepts.
- **Digital Hardware Design** (section 2.2): Fundamental concepts in digital hardware design, SoC architecture, and simulation.
- **Hardware Security Primitives** (section 2.3): Overview of hardware security primitives with focus on True Random Number Generators.
- **Simulation & Verification** (section 2.4): Simulation tools (Verilator, Cocotb), testbench frameworks (block-level and system-level testing), and waveform-based debugging.

## 2.1 Nix: Reproducible Package Management

Nix [1] is a functional package manager that addresses fundamental reproducibility challenges in software development through immutable package storage, cryptographic hashing, and declarative configuration.

**On Nix's Maturity and Documentation** Nix—first released in 2003—has comprehensive but *fragmented* documentation distributed across multiple official sources. This complexity is compounded by *flakes* (an experimental feature since 2020, discussed in subsection 2.1.6)—a debated approach championed by Determinate Nix but not referenced in classic Nix tutorials or nixpkgs documentation. This can confuse newcomers about which patterns to adopt. The official documentation is distributed across multiple sources:

- **Nix Reference Manual** (`https://nix.dev/manual/nix/`):
  Low-level documentation for the Nix language, built-in functions, and command-line tools.
- **Nixpkgs Manual** (`https://nixos.org/manual/nixpkgs/`):
  Documentation for `nixpkgs`, i.e. the nix package collection, including packaging guidelines and language-specific builders.
- **NixOS Wiki** (`https://wiki.nixos.org/`):
  Official wiki covering both NixOS (the Linux distribution) and some aspects of Nix (the package manager).
- **NixOS Manual** (`https://nixos.org/manual/nixos/`):
  System configuration reference for the NixOS operating system (not directly relevant for this thesis, but related).
- **Nix Flakes** (`https://nix.dev/concepts/flakes.html`):
  Conceptual overview of flakes, including adoption context and design considerations.

This fragmentation, combined with the coexistence of pre-flakes and flakes-based workflows, can make it challenging to determine which practices are current and which are deprecated. Community resources like blog posts and third-party guides often reflect specific historical moments or personal preferences, adding further complexity.

This section therefore acts as a *practical guide* covering Nix's core concepts—sufficient for readers to understand the thesis's methodology and to independently explore topics in greater depth—with the aim of lowering the adoption barrier for researchers and software developers encountering Nix for the first time.

**Organization** This section is organized thematically:

1. **The Reproducibility Problem** (subsection 2.1.1): Motivation for Nix and comparison with traditional approaches.
2. **The Nix Model** (subsection 2.1.2): Foundational functional package management model.
3. **Nix Terminology and Files** (subsection 2.1.3): Key terminology and the basic structure of Nix files.
4. **Managing Inputs** (subsection 2.1.4): Challenge of dependency pinning and reproducible inputs.
5. **Derivations and Outputs** (subsection 2.1.5): How Nix specifies and produces reproducible build outputs.
6. **Nix Flakes** (subsection 2.1.6): Unified project schema for explicit inputs, outputs, and deterministic builds.

### 2.1.1 The Reproducibility Problem

Traditional package managers (e.g., `apt`, `yum`, `pacman`) follow an *imperative* model: packages are installed globally, mutating the system state. This approach suffers from several fundamental problems, in relation to software development:

- **Dependency Hell:** Different projects may require incompatible versions of the same library. Installing version $A$ for project $X$ breaks project $Y$ that requires version $B$.
- **Non-Reproducibility:** The phrase "it works on my machine" epitomizes the problem. Two developers with ostensibly identical setups may have different package versions due to installation timing or system history.
- **Irreversibility:** System upgrades can break existing projects, and rolling back is non-trivial or impossible.
- **Implicit Dependencies:** Build processes may inadvertently depend on globally installed tools, creating hidden dependencies that fail on clean systems.

These issues affect software development broadly, and are particularly critical in *academic research*, where reproducibility is a prerequisite for independent validation.

For this reason, there is a growing trend to *ship the toolchain and dependencies alongside the project* rather than relying on the host system. Solutions offering containerization, such as Docker, have become widespread because they provide

isolated, self-contained environments. However, containers follow an imperative model rather than functional builds and environments. The comparison between Nix and Docker is discussed in the Methodology chapter (section 4.1.1), which justifies the choice of Nix for this thesis.

### 2.1.2 The Nix Model: Functional Package Management

Nix [2] addresses these problems through a *purely functional* approach inspired by functional programming languages. The key insight is to treat package management as a *pure function*: given the same inputs (source code, dependencies, build instructions), the build process must produce the same output, with no side effects.

#### Core Principles

1. **Immutability:** Packages are stored in the *Nix store* (`/nix/store`), an append-only content-addressed store where every package, library, or tool resides. Packages are stored with cryptographic hashes encoding their dependencies: `/nix/store/<hash>-<name>-<version>`. Once built, a package never changes.

2. **Isolation:** Each package is installed in a unique directory identified by `<hash>`, which is computed from all inputs (source, dependencies, build script, compiler flags, etc.). Multiple versions coexist without conflict. This enforces safe concurrent installation and rollback of different versions.

3. **Explicit Dependencies:** A package's hash includes hashes of all its dependencies. Changing a dependency (e.g., upgrading `glibc`) produces a different hash, and thus a different store path. This makes the dependency graph explicit and verifiable.

4. **Declarative Specification:** Packages are described by *expressions* in the Nix language, a purely functional domain-specific language for package management. The expression declares *what* to build, not *how* the system should mutate.

5. **Reproducibility:** The hash-based addressing ensures that identical inputs produce identical store paths. If two developers build the same expression, they get byte-for-byte identical results (modulo timestamps and other controlled variations).

6. **Cross-Distribution:** The Nix package manager works uniformly across Linux distributions (Ubuntu, Arch, Fedora, NixOS, etc.) and macOS. A Nix expression written on Ubuntu will produce the same result on Arch Linux

or macOS, because Nix builds everything from source (or uses binary caches with verified hashes) without relying on distribution-provided libraries.

### 2.1.3 Nix Terminology and Files

Before discussing how Nix manages dependencies (subsection 2.1.4) and produces outputs (subsection 2.1.5), it is useful to clarify the overloaded term *Nix* and introduce the basic structure of Nix files.

#### Terminology

The name *Nix* refers to several overlapping but distinct concepts [3]:

- **Nix (package manager):** tooling for managing packages and build processes using the Nix language.
- **Nix (language):** a purely functional, lazy, declarative language used for writing expressions evaluated by the Nix tools.
- **Nix (command):** the experimental unified CLI (`nix build`, `nix develop`, `nix flake`...) that supersedes older per-command tools such as `nix-build` and `nix-shell`.
- **Nix store:** the content-addressed filesystem at `/nix/store` where all built packages and their dependencies reside immutably.
- **The Nix ecosystem:** the broader set of tools and projects built on the Nix language.

Related concepts that frequently appear alongside Nix:

- **Nixpkgs:** the community-maintained package collection—a GitHub repository of over 130,000 derivations [4], the largest and most actively maintained across all package managers. It also provides the *standard library* of Nix helper functions (`stdenv`, `mkShell`, language-specific builders, etc.). The derivation model makes automated package maintenance and updates straightforward. In practice, most packages are downloaded from `cache.nixos.org`, the official binary cache, which serves pre-built binaries identified and verified by hash.
- **NixOS:** the Linux distribution built on top of the Nix package manager and Nixpkgs, where the entire OS configuration is a Nix expression.

### A Minimal Nix File

Nix files are *expressions* that evaluate to values—attribute sets, functions, derivations, or other Nix values. Probably the first file a new user encounters is a `shell.nix`, which creates a reproducible development environment [5]:

```
{
  pkgs ? import <nixpkgs> { },
}:

pkgs.mkShell {
  packages = [
    pkgs.git
    pkgs.nodejs
  ];

  MY_VAR = "example nix shell";

  shellHook = ''
    echo "MY_VAR is: $MY_VAR"
  '';
}
```

**Listing 2.1:** Example shell.nix

This file demonstrates some of the fundamental language constructs:

- **`{ pkgs ? ...}:`** A function with a named argument and a default value (the `?` operator). If no `pkgs` is passed by the caller, Nix imports nixpkgs from `NIX_PATH`.
- **`import <nixpkgs> {}:`** Evaluates the nixpkgs collection, returning an attribute set of all available packages and helper functions. `<nixpkgs>` resolves via the host's `NIX_PATH`—convenient for local use, but not reproducible (discussed in subsection 2.1.4).
- **`pkgs.mkShell:`** A Nixpkgs helper that creates a development shell derivation. `packages` lists tools to put in `$PATH`; arbitrary attributes (e.g. `MY_VAR`) become shell environment variables.
- **`shellHook:`** A bash snippet executed each time the shell starts—useful for printing status, setting up aliases, or running project-specific initialisation.

Running `nix-shell` drops into a Bash session where `git` and `nodejs` are available in `$PATH` and `MY_VAR` is set, regardless of what is installed system-wide. A development shell is itself a *derivation*—the same abstraction used for packages (subsection 2.1.5).

### 2.1.4 Managing Inputs: The Pinning Problem

The Nix model described in subsection 2.1.2 guarantees reproducible builds when all inputs are known. However, the common pattern of inheriting nixpkgs from system state creates a critical reproducibility gap. This subsection explains the problem and its evolution from manual workarounds to modern solutions.

#### The Function Pattern and the `NIX_PATH` Problem

Many Nix files are written as *functions* to allow overriding inputs. A now discouraged pattern relies on `<nixpkgs>` inheriting from `NIX_PATH`:

```
{
  pkgs ? import <nixpkgs> { },
}:
{
  # File contents using pkgs
  # ...
}
```

**Listing 2.2:** Nix file inheriting system state

This defines a function that accepts an argument `pkgs` with a *default value* via the `?` operator. The `<nixpkgs>` syntax uses *angle-bracket notation*, where `<variable>` looks up the value of `variable` in the `NIX_PATH` environment variable. For example, `<nixpkgs>` resolves to the path stored in `NIX_PATH` under the key `nixpkgs`.

This pattern provides flexibility—callers can override `pkgs` for reproducibility, or rely on the default for convenience. However, since `NIX_PATH` is mutable system state (typically managed via `nix-channel`), relying on the default breaks reproducibility. Two developers might use different channels (`nixos-25.11` vs `nixos-unstable`), or the same developer at different times (before and after `nix-channel -update`), resulting in different nixpkgs versions and thus different builds. For reproducibility, nixpkgs must be *explicitly pinned* within the project, not inherited from external state.

**Native Pinning** Nix provides native mechanisms to pin nixpkgs to a specific commit. For fetching nixpkgs tarballs, `fetchTarball` is the appropriate built-in function:

```nix
{
  pkgs ?
    import
      (fetchTarball
      ↪  "https://github.com/NixOS/nixpkgs/archive/06278c77....tar.gz")
      { },
}:
{
  # File contents using pkgs
  # ...
}
```

**Listing 2.3:** Nix file with explicit pinning

This approach replaces `<nixpkgs>` (which resolves via `NIX_PATH`) with an explicit `fetchTarball` call that downloads nixpkgs from a specific Git commit (`06278c77...`).

The tarball URL is deterministic: given the same commit SHA, it always points to the same snapshot of nixpkgs. This solves the reproducibility problem—anyone using this file gets exactly the same nixpkgs version, regardless of their `NIX_PATH` configuration (`fetchTarball` remains deterministic because nixpkgs has internal mechanisms to verify integrity). Other fetcher functions exist for different use cases (`fetchurl` for arbitrary files with explicit hashes, `fetchFromGitHub` for GitHub repositories, etc.).

**Ergonomic Pinning with Lock Files** While native pinning works correctly, updating dependencies requires manual work: finding the desired commit, constructing the correct fetcher call, and editing the Nix file. Tools like npins [6, 5] and Niv [7] simplify this workflow by providing dedicated commands that generate and manage lock files, recording both the commit hash and the appropriate fetcher function to use.

For example, npins can be initialized in a project and used to pin nixpkgs:

```
$ npins init --bare
$ npins add github nixos nixpkgs --branch nixos-unstable
```

This generates `npins/sources.json` (the lock file) and `npins/default.nix` (which exposes dependencies as an attribute set):

```
{
  "nixpkgs": {
    "type": "Git",
    "repository": {
      "type": "GitHub",
      "owner": "NixOS",
      "repo": "nixpkgs"
    },
    "branch": "nixos-25.11",
    "revision": "06278c77b5d162e62df170fec307e83f1812d94b",
    "url":
     ↪  "https://github.com/NixOS/nixpkgs/archive/06278c77b5d162e62df170fec307e83f18
    "hash": "0sjjj9z1dhilhpc8pq4154czrb79z9cm044jvn75kxcjv6v5l2m5"
  }
}
```

**Listing 2.4:** npins/sources.json (generated by npins)

The project's `default.nix` then imports these pinned sources as default arguments:

### Advantages of Lock File Workflows with npins

- **Automated Updates:** Commands like `npins update nixpkgs` fetch the latest commit from the tracked branch and update the lock file, eliminating manual fetcher construction.
- **Human-Readable Metadata:** Lock files (`sources.json`) record the branch, repository, revision, and URL—making it clear what `06278c77...` represents and where it comes from.
- **Separation of Concerns:** Lock files are stored in a dedicated file, keeping versioning information separate from the main Nix expressions.
- **No Runtime Dependency:** Once the lock file is generated, users cloning the project don't need npins installed to *use* the environment—npins is only required for *updating* dependencies. The lock file could even be manually edited, though the tool provides better ergonomics.

```nix
{
  sources ? import ./npins,
  system ? builtins.currentSystem,
  pkgs ? import sources.nixpkgs {
    inherit system;
    config = { };
    overlays = [ ];
  },
}:
{
  # File contents using pkgs
  # ...
}
```

**Listing 2.5:** default.nix using npins

It's important to note that these tools are not solving technical limitations—they simply provide a more convenient user experience. The underlying pinning mechanism remains the same; these tools automate the bookkeeping and provide dedicated update commands.

Another approach is *Nix Flakes* (subsection 2.1.6), which address input pinning, output declaration, and project structure in a single standardized schema.

**Note on Examples** While `<nixpkgs>` is discouraged for reproducibility, the examples that follow use this syntax for simplicity; fully pinning nixpkgs would obscure the core concepts being illustrated.

### 2.1.5 Derivations and Outputs

Having discussed how Nix manages *inputs* (subsection 2.1.4), we now turn to what Nix produces. This subsection focuses on *derivations*—the specifications for building artifacts—and their *outputs*, the concrete store objects produced when derivations are evaluated.

#### What is a Derivation?

The most important built-in function is `derivation`, which describes a single derivation [8, 9, 10]: a specification for running an executable on precisely defined

input files to repeatably produce output files at uniquely determined file system paths.

A derivation takes as input an attribute set specifying the build process, and produces a *store derivation* (a `.drv` file) as a side effect. When this derivation is built, it produces one or more *outputs*—store objects containing the built artifacts. Formally:

$$\text{derivation} : \text{Inputs} \rightarrow \text{Store Derivation} \rightarrow \text{Outputs} \tag{2.1}$$

where *Inputs* include source code, dependencies, build tools, environment variables, and build instructions; *Store Derivation* is the `.drv` file describing the build; and *Outputs* are the resulting store objects (e.g., `/nix/store/<hash>-hello`).

Under the hood, a derivation is represented as a `.drv` file—a low-level specification that Nix uses to execute the build. This file contains output paths (computed from input hashes), input derivations (dependencies), build script and environment variables, and system architecture (`x86_64-linux`, `aarch64-darwin`, etc.).

The power of this abstraction is its universality: every package in `nixpkgs` (GCC, Python, LLVM), every NixOS system configuration (producing `/run/current-system`), and even development shells (`mkShell`) are all derivations. This unification means that the same reproducibility guarantees apply everywhere—whether you're building a kernel, compiling a Rust application, or entering a development shell, the underlying mechanism guarantees deterministic outputs.

### Low-Level Derivations

Derivations can be written manually using the built-in `derivation` function. All higher-level utilities ultimately compile down to this primitive. While rarely used directly in practice (it's quite verbose and low-level), understanding it clarifies what the ecosystem's convenience functions are doing for you.

Here's a minimal example that creates a script displaying "Hello, world!":

Key observations:

- **Explicit Everything:** You must manually specify the builder (`bash`), all required utilities (`coreutils`), and construct `$PATH` yourself.
- **`$out` Variable:** Points to the derivation's unique store path (e.g., `/nix/store/<hash>-hello-world`). Anything placed in `$out` becomes the derivation's result.
- **System Architecture:** `builtins.currentSystem` ensures the derivation is built for the current platform (`x86_64-linux`, `aarch64-darwin`, etc.).

As you can see, this approach quickly becomes unwieldy. For this reason, the Nix ecosystem provides higher-level utility functions that abstract common patterns.

```nix
let
  pkgs = import <nixpkgs> { };
in
derivation {
  name = "hello-world";
  system = builtins.currentSystem;
  builder = "${pkgs.bash}/bin/bash";
  args = [
    "-c"
    ''
      export PATH="$PATH:${pkgs.coreutils}/bin"
      echo '#!/${pkgs.bash}/bin/bash' > $out
      echo 'echo "Hello, World!"' >> $out
      chmod +x $out
    ''
  ];
}
```

**Listing 2.6:** Low-level derivation example

### Utility Functions for Creating Derivations

In practice, derivations are almost always created using utility functions from `nixpkgs` [11] that handle the tedious low-level details. The following examples illustrate common patterns; note that these are not rigid categories—a single derivation can freely combine them (e.g. `stdenv.mkDerivation` paired with `fetchurl`, a Fixed-Output Derivation, to fetch a remote source tarball).

**Standard Derivations** The **standard environment** (`stdenv`) provides the most common utility function for building packages: `stdenv.mkDerivation`. It ensures you have all the typical binaries and libraries expected in a Linux build environment (compiler, coreutils, Make, patch, etc.), abstracting away the low-level details shown in section 2.1.5.

Compared to the low-level example, `stdenv.mkDerivation` can be written much more concisely:

A minimal derivation for building GNU Hello:

Notable features:

- **Explicit dependency:** `stdenv` provides the complete standard build environment (compiler, coreutils, Make, etc.).

```nix
{ stdenv }:

stdenv.mkDerivation {
  pname = "hello";
  version = "2.12";

  src = ./.;

  buildPhase = ''
    ./configure --prefix=$out
    make
  '';

  installPhase = ''
    make install
  '';
}
```

**Listing 2.7:** default.nix for GNU Hello

- **Local source:** `src = ./.` points to the current directory; Nix copies it into the store, making the build hermetic and its hash derived from the source tree.
- **Standard output layout:** Files are installed under `$out/` following a conventional subdirectory layout (`bin/`, `lib/`, `include/`, etc.), mirroring the `/usr` prefix convention. Unlike FHS, each Nix package has its own isolated store prefix with no shared directories.
- **Build Phases:** The `buildPhase` and `installPhase` are part of a standard sequence (`unpackPhase`, `patchPhase`, `configurePhase`, `buildPhase`, `checkPhase`, `installPhase`, `fixupPhase`) that mirrors typical software builds. If omitted, `stdenv` provides sensible defaults (e.g. `./configure && make && make install`).
- **Deterministic:** Given these inputs, the build always produces the same binary (modulo known sources of non-determinism like timestamps, which `stdenv` neutralizes).

By convention, package derivations are placed in a file named `default.nix`—the file `nix-build` looks for by default when invoked without arguments.

**Shell Derivations** `pkgs.mkShell` [12] is a convenience wrapper around `stdenv.mkDerivation` specifically designed for development environments. A `shell.nix` uses it to declare which packages should be in `$PATH`, optional environment variables, and `shellHook` scripts executed on entry:

```
{
  pkgs ? import <nixpkgs> { },
}:
pkgs.mkShell {
  packages = [
    pkgs.yosys
    pkgs.verilator
  ];
}
```

**Listing 2.8:** shell.nix for hardware development

Running `nix-shell` drops into a Bash session where the declared tools are available in `$PATH`, regardless of what is installed system-wide—providing a self-contained development environment for each project.

A common pattern is to commit a `shell.nix` to each project repository: any developer can then run `nix-shell` to enter the project's environment without touching the host system. By default, `nix-shell` looks for `shell.nix` in the current directory; a different file can be specified with `nix-shell path/to/file.nix`. However, `mkShell` is intended for interactive development, not for distribution. When the goal is to *build and package* a project, a proper derivation using `stdenv.mkDerivation` or a language-specific builder is required. The two roles can coexist in the same repository: a `shell.nix` for day-to-day development and a `default.nix` for reproducible builds.

**Fixed-Output Derivations** As explained above, Nix builds run in a sandbox that restricts network access by default to ensure reproducibility. However, many build systems need to download sources or dependencies from the internet. A *Fixed-Output Derivation* (FOD) is the **only** mechanism that allows network access during a build. The key idea: the output hash is *specified in advance*, allowing Nix to verify that the downloaded content is exactly what was expected.

A common FOD is `fetchurl`—the standard way to fetch a remote tarball with hash verification:

Under the hood, `fetchurl` is a wrapper around a FOD that:

```
  src = fetchurl {
    url = "mirror://gnu/hello/hello-2.12.tar.gz";
    sha256 = "1ayhp9v4m4rdhjmnl2bq3cibrbqqkgjbl3s7yk2nhlh8vj3ay16g";
  };
```

**Listing 2.9:** fetchurl is an FOD

1. Enables network access for the build
2. Downloads the file from the specified URL
3. Computes `hash($out)` and compares it to the provided `sha256`
4. Fails if the hashes don't match

This is why every `fetchurl`, `fetchFromGitHub`, or `fetchgit` call requires a hash: without it, the network-accessing build would be non-reproducible.

For languages with their own package managers (Maven, npm, Cargo), FODs allow downloading dependencies during the build. The FOD contract is simple: Nix enables network access, then verifies that `hash($out)` matches the declared `outputHash`. If they match, the build succeeds; otherwise it fails. This ensures that even though the build process is non-deterministic (network timing, mirror selection), the *output* remains deterministic and cryptographically verified.

**Language-Specific Derivations** For most programming languages and frameworks, `nixpkgs` provides specialized builder functions that understand language-specific conventions and package managers [11]. These include:

- **Python:** `buildPythonPackage`, `buildPythonApplication` (handles `setup.py`, `pyproject.toml`, dependencies from PyPI)
- **Rust:** `buildRustPackage` (integrates Cargo, verifies `Cargo.lock`)
- **Go:** `buildGoModule` (handles Go modules and `go.mod`)
- **Node.js:** `buildNpmPackage`, `buildYarnPackage` (manages npm/Yarn dependencies)
- **Java/Maven:** `maven.buildMavenPackage` (records Maven dependencies as a FOD)
- **Java/Gradle:** `gradle.fetchDeps` (uses a MITM proxy to intercept and hash all Gradle dependency downloads)

These builders abstract language-specific patterns while maintaining reproducibility through lock files and cryptographic verification. For example, this thesis uses Gradle for building Scala-based hardware tools; `nixpkgs` provides `gradle.fetchDeps`, which uses a MITM proxy cache to intercept and record all Maven dependency downloads in a lockfile (`deps.json`) containing SHA-256 hashes.

**System Derivations** At the highest level of abstraction, entire system configurations are also derivations. For example:

- **NixOS System:** The full OS configuration (kernel, services, users, packages) is a single derivation producing `/run/current-system`. This enables atomic upgrades and rollbacks—each system generation is an immutable store path.
- **Home Manager:** A complement to NixOS (but also usable standalone, e.g. on macOS) that manages user-space configuration through declarative Nix modules: dotfiles, installed programs, and user services are all expressed as derivations, producing a versioned user profile.
- **Docker Images:** Nix can build Docker images declaratively using `pkgs.dockerTools.buildo` producing minimal, reproducible containers.

This hierarchy—from low-level `derivation` primitives to system-wide configurations—demonstrates the uniformity of Nix's approach: everything is a derivation, ensuring consistent reproducibility guarantees across all abstraction levels.

### Cross-Compilation

Nixpkgs offers strong cross-compilation support. It follows the GNU autoconf convention of three platform roles [11, 13, 14, 15]:

- **build**: the machine performing the compilation; corresponds to `localSystem`, inferred from `builtins.currentSystem` if omitted.
- **host**: the platform where the resulting binaries will run; equals build by default (native build).
- **target**: relevant only for compilers whose build produces binaries for a single platform (GCC, Binutils, GHC); irrelevant for all other packages.

Nixpkgs exposes two ways to cross-compile, with different scopes:

- **`crossSystem`:** a configuration option passed when evaluating nixpkgs. It affects the *entire* resulting package set: every package in `pkgs` is compiled for the specified host platform.

- **`pkgsCross.<target>`:** an attribute of the *native* `pkgs`, exposing a pre-configured cross package set for a specific target. The native `pkgs` is unchanged; only packages accessed via `pkgs.pkgsCross.<target>` are cross-compiled. This is the more granular approach, preferred when only a subset of the build needs to target a different platform.

For a single package, the two forms produce the same derivation:

```nix
# explicit crossSystem
let
  pkgs = import <nixpkgs> {
    localSystem = "x86_64-linux"; # inferred if omitted
    crossSystem = {
      config = "riscv32-none-elf"; # bare-metal RISC-V
    };
  };
in
pkgs.hello
```

**Listing 2.10:** Cross-compilation via `crossSystem`

```nix
# equivalent via pkgsCross
let
  pkgs = import <nixpkgs> { };
in
pkgs.pkgsCross.riscv32-embedded.hello
```

**Listing 2.11:** Equivalent via `pkgsCross`

This build/host distinction maps directly onto `stdenv.mkDerivation` dependency attributes [11]:

- `nativeBuildInputs` (`depsBuildHost`): dependencies *executed during the build*—tools that must run on the build machine (e.g. the cross-compiler, `cmake`, `pkg-config`, setup hooks). Placed in `$PATH` during the build, not embedded in the output.
- `buildInputs` (`depsHostTarget`): dependencies that will be *copied or linked*

*into the final output*, or otherwise used at runtime—typically libraries compiled for the host platform.

When `crossSystem` is unset (the default), build and host are the same platform, so both package sets coincide and packages end up in the same place regardless of which attribute they are listed under. When it is set, they diverge: placing a tool in `buildInputs` instead of `nativeBuildInputs` would produce a binary compiled for the host platform, making it unavailable in `$PATH` and unusable during the build. This separation allows nixpkgs to support cross-compilation transparently for the vast majority of packages without per-package changes.

For a concrete application of `pkgsCross` in this thesis, see subsection 5.1.2. Flakes impose an additional ergonomic constraint on cross-compilation workflows, discussed in section 2.1.6.

### 2.1.6 Nix Flakes

#### Introduction: A Debated Solution

The previous subsections presented Nix's foundations (subsection 2.1.2), the challenges of managing inputs (subsection 2.1.4), and how Nix specifies and produces outputs through derivations (subsection 2.1.5). However, these concerns were historically managed through separate, uncoordinated mechanisms: native pinning or helper tools (see subsection 2.1.4) for inputs, and ad-hoc conventions (`shell.nix`, `default.nix`) for outputs—with no standardized schema for a project to declare and expose multiple artifacts.

Flakes [16, 17, 18] provide a *debated solution* to unify this fragmented landscape. While still experimental (requiring the `experimental-features = nix-command flakes` flag as of early 2026), flakes have gained widespread adoption in the Nix community.

A *flake* is a simply a directory containing a `flake.nix` file with a standardized structure that defines four top-level attributes:

1. **`description`:** A human-readable string describing the project.
2. **`inputs`:** Dependencies declared with URL-like syntax (Git repos, GitHub/GitLab, tarballs, local paths).
3. **`outputs`:** A function taking all resolved inputs and returning the artifacts this flake provides (packages, development shells, NixOS modules, etc.), following a standardized schema.
4. **`nixConfig`:** Optional Nix configuration (e.g., binary caches) scoped to this flake.

An example minimal flake:

```nix
{
  description = "A simple flake";

  inputs = {
    # Dependencies (other flakes, Git repos, tarballs)
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-25.11";
  };

  outputs =
    { self, nixpkgs }:
    {
      # What this flake provides
      packages.x86_64-linux.hello = {
        # ...
      };
      devShells.x86_64-linux.default = {
        # ...
      };
    };
}
```

**Listing 2.12:** Minimal flake.nix

**Adoption and Controversies** Despite widespread adoption, flakes remain experimental in upstream Nix [19], while *Determinate Nix* [16, 20] and community forks like Lix have declared them stable in their distributions. This split explains the documentation gap noted above: official Nix sources remain cautious about flakes, so most flake documentation lives in community wikis and third-party guides rather than the official manuals. This is also the reason this section introduces flakes as a *debated solution*: they are widely adopted in practice but remain not yet stabilized upstream.

### Schema

**Inputs** The `inputs` attribute declares all external dependencies using URL-like syntax. Common input types include:

- **GitHub/GitLab:** `github:NixOS/nixpkgs/nixos-unstable` or `github:NixOS/nixpkgs/n` (latest stable release branch at the time of writing)

- **Git repositories:** `git+https://example.com/repo.git?ref=main`
- **Tarballs:** `https://example.com/archive.tar.gz`
- **Local paths:** `path:./subproject`
- **Indirect references:** `nixpkgs` (resolved via the flake registry)

Inputs can be either flakes themselves (exposing packages via the standardized schema) or plain repositories marked with `flake = false`. An example:

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-unstable";

    # Flake input: another project that exposes some outputs via its own flake.nix
    some-flake.url = "github:user/some-project";

    # Non-flake input: plain repository without flake.nix
    vendor-lib = {
      url = "github:vendor/library";
      flake = false; # Just source files, stored in /nix/store
    };
  };

  outputs =
    {
      self,
      nixpkgs,
      some-flake,
      vendor-lib,
    }:
    # ... outputs implementation ...
    { };
}
```

**Listing 2.13:** Flake inputs example

Non-flake inputs (marked `flake = false`) are fetched and stored in `/nix/store` (read-only), then referenced in derivations as file paths. This is useful for vendoring source code without requiring the upstream project to adopt flakes.

All inputs are automatically resolved to specific commits and recorded in `flake.lock` on first evaluation. The lock file is JSON and should be committed to version control to guarantee all developers use identical dependency versions. It is never manually edited: `nix flake lock` generates or refreshes it from `flake.nix` without building anything; `nix flake update` updates all inputs to their latest versions; `nix flake update <input-name>` updates a single input.

**Outputs** The `outputs` attribute is a function receiving all resolved inputs (plus `self`, the current flake) and returning an attribute set following a standardized schema. Common output types include (non-exhaustive):

- `packages.<system>.<name>`: Derivations buildable with `nix build .#<name>`. Also used by `nix shell` to add packages to `$PATH` temporarily, and by `nix run` to execute binaries in `$out/bin/`.
- `apps.<system>.<name>`: Custom executable applications for `nix run .#<name>`. Unlike `packages`, `apps` allow specifying custom execution logic (wrapper scripts, environment setup, predefined arguments) via an explicit `program` attribute.
- `devShells.<system>.<name>`: Development environments for `nix develop .#<name>`, typically created with `mkShell`.
- `formatter.<system>`: Code formatter invoked by `nix fmt`
- `checks.<system>.<name>`: Test derivations run by `nix flake check`
- `overlays.<name>`: Functions extending nixpkgs with custom packages, consumable by other flakes
- `nixosConfigurations.<hostname>`: Complete NixOS system configurations, deployed with `nixos-rebuild`

The schema enforces system-specific outputs (e.g., `packages.x86_64-linux.*`, `packages.aarch64-linux.*`). A representative example (Listing 2.14):

Some observations:

- **Import pattern:** The actual derivation and shell logic live in separate `.nix` files (`packages/hello.nix`, `shell.nix`) and are brought in via Nix's built-in `import` function. This is not a requirement—derivations can be written inline inside `flake.nix`—but it is the idiomatic approach: `flake.nix` acts as an entry point that wires inputs to outputs, while the build logic lives in plain `.nix` files that can be tested or reused independently of the flake infrastructure [21].

- **legacyPackages:** A substitute for `packages` used by nixpkgs to avoid making `nix flake show nixpkgs` unusably slow. The Nix CLI evaluates everything under `packages.<system>.*` when listing outputs; with `legacyPackages` it displays `omitted` instead, skipping evaluation of all 100,000+ packages [22]. The name does not imply the packages themselves are legacy.

This is the *recipe list* model: `flake.nix` declares all artifacts the project provides under a standardized schema, making them instantly discoverable and accessible to consumers—whether a developer running `nix build`, a CI system invoking `nix flake check`, or another flake importing an overlay.

### Flake Commands

Flakes are designed to work with the unified `nix` CLI, enabled by the `nix-command` experimental feature (the same flag that enables flakes: `experimental-features = nix-command flakes`). This modern CLI replaces the older per-purpose tools (`nix-build`, `nix-shell`, `nix-env`, `nix-store`) with a single `nix` command and consistent sub-commands.

**Commands** The main `nix` sub-commands [23] and their relationship to flake outputs are:

- `nix build .#<name>` — builds `packages.<system>.<name>` and links the result to `./result`
- `nix develop .#<name>` — enters the `devShells.<system>.<name>` environment (drops into a Bash shell with the declared packages in `$PATH` and hooks executed)
- `nix run .#<name>` — runs `apps.<system>.<name>` or, if no explicit app is defined, the default binary of the corresponding package
- `nix shell .#<name>` — temporarily adds `packages.<system>.<name>` to `$PATH`
- `nix flake show` — lists all outputs declared in `flake.nix` without building anything
- `nix flake check` — builds and runs all `checks.<system>.*` derivations
- `nix fmt` — runs the `formatter.<system>` derivation on the project source

**Flake URIs** All commands above default to the local `.` directory, but accept any flake URI in its place—allowing direct access to remote flakes without cloning:

- `nix run github:nixvim/nixvim` — run a package from GitHub
- `nix shell github:NixOS/nixpkgs/nixos-unstable#librelane` — temporary shell with LibreLane
- `nix develop github:user/project` — enter project's development shell

Nix automatically fetches the repository, evaluates the flake, and executes the specified output. This provides instant access to any artifact exposed in a flake's outputs, enabling workflows like testing unreleased packages or sharing reproducible development environments with a single command.

### Benefits and Trade-offs

Flakes provide significant advantages but also introduce limitations that merit consideration [19, 21].

#### Benefits

- **Pure evaluation and enforced pinning:** Flakes disable `NIX_PATH` and system channels, forcing all dependencies to be declared in `inputs` and pinned in `flake.lock` with cryptographic hashes. Unlike traditional workflows where `<nixpkgs>` inherits mutable system state, flakes use the *flake registry* for convenience references (e.g., `nix run nixpkgs#hello`) but always resolve to `flake.lock` entries within the project—dependencies only change through explicit `nix flake update` calls.
- **Standardized schema:** The `flake.nix` structure (inputs/outputs) provides a uniform interface across projects. `nix flake show` lists outputs, `nix flake update` refreshes dependencies (see section 2.1.6 for the full command set).
- **Evaluation caching:** Flakes cache evaluation results, accelerating builds when inputs remain unchanged.

#### Limitations and Controversies

- **Experimental status and documentation gap:** As discussed in section 2.1.6, flakes remain experimental in upstream Nix while distributions like Determinate Nix consider them stable. This fragmentation means that documentation relies on community wikis rather than official manuals, increasing the learning curve and leaving their stabilization as an official Nix feature formally unresolved.

- **Lack of parameterization:** `flake.nix` cannot accept arbitrary arguments [21]; `-arg` and `-argstr` are explicitly incompatible with flakes. Build configuration such as feature flags or variants must be anticipated ahead of time, with workarounds such as `-override-input` or delegation to non-flake primitives.

- **Limited cross-compilation ergonomics:** Because `flake.nix` outputs are a static attribute set, target architectures must be enumerated ahead of time as separate outputs (e.g. `packages.x86_64-linux.firmware-riscv32`) [21]. The traditional `nix-build` workflow allows passing `crossSystem` dynamically at invocation time, which is more flexible. For the cross-compilation mechanism itself, see section 2.1.5.

- **No version resolution:** Flakes use the exact versions from dependencies' `flake.lock` files. Without explicit `follows` declarations, transitive dependencies may pull multiple versions of the same library [19]. There is no SAT solver or version constraint system like in traditional package managers.

- **Tight Git coupling:** When a flake resides in a Git repository, only files tracked by Git (`git add`) are visible to builds. Untracked files are silently invisible to Nix with no warning, requiring workarounds; improved diagnostics are expected in future releases.

- **Constrained input syntax:** The `inputs` attribute uses a restricted subset of Nix, preventing programmatic generation or DRY patterns [21].

## 2.2 Hardware Design

Hardware description languages (HDLs) are specialized languages for modeling digital circuits, enabling designers to describe hardware at multiple abstraction levels. This section introduces traditional HDLs (Verilog and VHDL), modern high-level approaches (Chisel, SpinalHDL), and the interoperability patterns that enable mixed-language designs.

### 2.2.1 Hardware Description Languages

#### Traditional HDLs: Verilog and VHDL

Traditional hardware description languages, Verilog and VHDL, were developed in the 1980s to model digital circuits for simulation and synthesis.

**Verilog** was created in 1984 and standardized as IEEE 1364-1995 [24]. It features C-like syntax with constructs such as `always` blocks, `wire`/`reg` variable types, and blocking (`=`) versus non-blocking (`<=`) assignments. Verilog uses a 4-valued logic system (0, 1, Z for high-impedance, X for unknown).

**VHDL** (VHSIC Hardware Description Language) was developed in 1983 by the U.S. Department of Defense for documenting ASICs and standardized as IEEE 1076-1987 [25]. Inspired by Ada, VHDL features strongly typed constructs, `process` blocks, and verbose syntax. It uses the IEEE 1164 multi-valued logic standard (`std_logic`) with nine states (0, 1, X, Z, W, H, L, U, -).

**Event-driven semantics.** Both languages are fundamentally *event-driven*: designers describe *how to simulate* hardware behavior using procedural code. Synthesis tools then *infer* the actual hardware structures (registers, latches, state machines) from this simulation-oriented description. This indirect approach requires designers to understand both simulation semantics and synthesizable coding patterns.

**Limitations.** The key limitations of traditional HDLs include:

- **Verbosity:** Extensive boilerplate for wiring, module instantiation, and bus declarations.
- **Limited type safety:** Verilog provides weak typing; errors often appear only during simulation or synthesis.
- **Manual parametrization:** Verilog `parameter` and VHDL `generic` provide limited abstraction; complex meta-programming patterns are not supported.

### Modern High-Level HDLs

Modern hardware description languages address these limitations by embedding HDL constructs within general-purpose programming languages as *domain-specific languages* (DSLs).

**Structural abstraction and meta-hardware generation.** Unlike event-driven HDLs where synthesis tools *infer* registers and state machines from simulation-oriented code, modern HDLs provide hardware constructs as *first-class types*: `Reg`, `Latch`, `FSM`, and `Bus` are directly instantiated. By leveraging host language features (OOP, functional programming, type systems), they act as *hardware generators*: parametric designs that produce optimized RTL from configuration parameters.

**Examples.** Prominent modern HDLs include:

- **Chisel** [26] (Berkeley, 2012): A Scala-based DSL with functional programming emphasis. Chisel generates Verilog through the Firrtl intermediate representation and has been used in production designs such as the RISC-V Rocket chip and SiFive's commercial RISC-V processors.
- **SpinalHDL** [27] (2014): A Scala-based DSL combining object-oriented and functional paradigms. SpinalHDL generates Verilog and VHDL directly and provides compile-time design checks (combinatorial loops, clock domain crossings).
- **MyHDL, Amaranth**: Python-based DSLs suitable for prototyping and verification, though less widely adopted for ASIC production.

**Interoperability.** Modern HDLs generate standard hardware description output compatible with established EDA toolchains for simulation, synthesis, and physical implementation.

### 2.2.2 RISC-V ISA and VexRiscv Core

#### RISC-V Instruction Set Architecture

RISC-V is an open-source instruction set architecture (ISA) developed at UC Berkeley [28]. Unlike proprietary architectures (ARM, x86), RISC-V specifications are freely available and can be implemented without licensing fees. The ISA follows a modular design:

**Base integer ISA.** RV32I defines the core 32-bit integer instruction set with 40 instructions covering integer ALU operations, byte/halfword/word memory access, conditional branches, and jump-and-link calls.

**Standard extensions.** Optional extensions add functionality:

- **M**: Integer multiplication and division (`MUL`, `DIV`, `REM`)
- **A**: Atomic operations for multicore synchronization
- **F/D**: Single/double-precision floating-point
- **C**: Compressed 16-bit instructions for code density

**Privilege levels.** RISC-V defines three privilege modes:

- **Machine (M-mode)**: Highest privilege, mandatory, used for bare-metal firmware
- **Supervisor (S-mode)**: OS kernel level, optional, manages virtual memory
- **User (U-mode)**: Application level, optional, isolated processes

Control and Status Registers (CSRs) provide system configuration, exception handling, and performance monitoring capabilities.

### VexRiscv Soft-Core Processor

VexRiscv [29] is a RISC-V soft-core implementation written in SpinalHDL, designed for FPGA and ASIC deployment. Its key characteristics include:

**Plugin architecture.** VexRiscv uses a modular plugin system where features are added through composable plugins:

- **Instruction fetch**: `IBusSimplePlugin` (no cache) or `IBusCachedPlugin` (with instruction cache)
- **Data access**: `DBusSimplePlugin` (no cache) or `DBusCachedPlugin` (cached, pipelined)
- **Arithmetic**: `IntAluPlugin` (integer ALU), `MulPlugin` (fast multiplication), `DivPlugin` (iterative division)
- **Branch prediction**: Static, dynamic (GShare), return address stack (RAS)

**Bus interface flexibility.** VexRiscv supports multiple memory bus protocols:

- **Simple bus (PipelinedMemoryBus)**: SpinalHDL-specific request-response bus used internally; bridges adapt it to standard protocols
- **AXI4**: High-bandwidth, burst transfers
- **AHB-Lite3**: Pipelined system bus, single-master subset of AHB
- **Wishbone**: Open-source bus standard used in many open-hardware SoC designs
- **Avalon**: Intel/Altera FPGA-native interface (Quartus/Platform Designer)

The `iBus` (instruction bus) and `dBus` (data bus) are separate interfaces, enabling Harvard architecture with independent instruction and data memory paths.

**ASIC-proven implementation.** VexRiscv has been used in production ASIC tape-outs, including the VE-HEP project [30] at IHP's SG13G2 130nm process, validating its suitability for ASIC implementation beyond FPGA prototyping.

### 2.2.3 Bus Protocols and Memory-Mapped I/O

SoC designs require bus protocols to connect processors to memory and peripherals. This section introduces the AMBA protocol family and memory-mapped I/O fundamentals.

#### AMBA Protocol Family

The Advanced Microcontroller Bus Architecture (AMBA) is the de-facto industry-standard family of on-chip interconnect specifications developed by ARM. AMBA specifications are publicly available and widely adopted in commercial SoC designs (ARM Cortex-M/A systems, mobile processors, automotive chips). Since its introduction in 1996, AMBA has evolved through five major revisions (AMBA 1 through AMBA 5, published 1996–2013), each adding new protocols and features for system-level integration.

**Protocol hierarchy.** The AMBA family defines multiple protocols optimized for different interconnect requirements:

- **Advanced eXtensible Interface (AXI)** [31] targets high-performance, high-frequency designs. It features five independent channels (AR: read address, AW: write address, R: read data, W: write data, B: write response), burst transfers, out-of-order responses, and multiple outstanding transactions. Typically used for main memory controllers, DMA engines, and high-bandwidth peripherals.
- **Advanced High-performance Bus (AHB)** is a pipelined system bus with burst support and centralized arbitration. AHB-Lite (AMBA 3) is a single-master subset commonly used for cache controllers, on-chip memory, and intermediate interconnects.
- **Advanced Peripheral Bus (APB)** is designed for low-bandwidth control accesses, such as register interfaces on system peripherals. This bus has an address and data phase similar to AHB, but a much reduced, low complexity signal list (no bursts, no pipelining). APB is optimized for low-power operation and reduced area, making it suitable for UART, GPIO, timers, and similar control-oriented peripherals.

Systems commonly combine multiple AMBA protocols in a hierarchical structure: AXI for memory subsystems and high-throughput devices, AHB for intermediate

buses, and APB bridges for low-bandwidth peripheral clusters. This layered approach balances performance, power, and design complexity.

### APB3 Protocol

The SoC presented in this thesis uses APB3 [32] (from the AMBA 3 specification) for peripheral interconnect. The rationale for selecting APB3 over alternative bus protocols is discussed in Section 4.2.3. The APB3 specification defines a simple two-phase handshake protocol with the following signals:

- **Control signals:**
  - `PSEL`: Peripheral select (one bit per peripheral, asserted by bus bridge)
  - `PENABLE`: Enable (strobes during access phase)
  - `PREADY`: Ready (peripheral indicates transaction completion)
  - `PWRITE`: Write enable (1 = write, 0 = read)
- **Data signals:**
  - `PADDR`: Address bus (byte-aligned register offset)
  - `PWDATA`: Write data (valid when `PWRITE` = 1)
  - `PRDATA`: Read data (valid when `PWRITE` = 0 and `PREADY` = 1)

**Transaction phases.** An APB3 transaction consists of:

1. **Setup phase** (1 cycle): Bus bridge asserts `PSEL`, `PADDR`, `PWRITE`, and `PWDATA` (if write). `PENABLE` remains low.
2. **Access phase** ($\geq$ 1 cycle): Bridge asserts `PENABLE`. Peripheral processes the request and asserts `PREADY` when complete. If `PREADY` is low, the bridge extends the access phase (wait states).

Minimum transaction latency is 2 clock cycles (setup + access). Peripherals can insert wait states by deferring `PREADY` assertion, enabling slow peripherals to interface with fast CPUs.

### Memory-Mapped I/O

CPU architectures use two primary approaches for peripheral communication [33]:

- **Port-mapped I/O (PMIO)**: Peripheral registers occupy a separate I/O address space, accessed via dedicated instructions (`IN`/`OUT` on x86). Preserves memory address space but requires distinct instruction types; used mainly on x86.

- **Memory-mapped I/O (MMIO)**: Peripheral registers occupy address space alongside RAM. The CPU uses standard load/store instructions (`LW`, `SW`) for both memory and I/O access, with a single unified address space. This simplifies programming and instruction set design, though it consumes memory address range. Modern RISC architectures (ARM, RISC-V, MIPS, PowerPC) exclusively use memory-mapped I/O.

Since the SoC presented in this thesis is based on the RISC-V ISA, it uses memory-mapped I/O for all peripheral access—the standard approach for modern RISC processors.

For example, addresses starting with `0x8` target RAM, while `0xF` targets peripheral space. Within the peripheral region, a secondary decoder inspects lower address bits to generate peripheral-specific `PSEL` signals.

## 2.3 Hardware Security Primitives

Cryptographic systems require high-quality random numbers for key generation, initialization vectors, nonces, and session keys. Hardware Security Modules (HSMs) rely on True Random Number Generators (TRNGs) to provide unpredictable entropy that cannot be reproduced mathematically, unlike Pseudo-Random Number Generators (PRNGs) which are deterministic algorithms. This section establishes the theoretical foundation for TRNG integration in embedded systems, covering HSM requirements (subsection 2.3.1), the distinction between PRNGs and TRNGs (subsection 2.3.2), TRNG architecture and quality assurance mechanisms (subsection 2.3.3), and integration patterns in SoC designs (subsection 2.3.4).

### 2.3.1 Hardware Security Modules and Randomness Requirements

A Hardware Security Module (HSM) is a tamper-resistant cryptographic device that performs sensitive operations such as key generation, encryption/decryption, digital signing, and secure key storage [34]. HSMs are physical computing devices that safeguard and manage cryptographic keys, providing a hardened, tamper-evident environment for processing cryptographic operations. Modern HSMs are typically validated against FIPS 140-3 (Federal Information Processing Standard), which defines security requirements for cryptographic modules across four progressive security levels [35].

Standard HSM components include:

- **Hardware cryptographic accelerators**: Dedicated circuits for AES, RSA, ECC, and hash functions (SHA-2/SHA-3), providing orders-of-magnitude speedup over software implementations while reducing side-channel leakage.

- **Secure key storage**: Non-volatile memory (EEPROM, Flash, battery-backed RAM) with access controls, storing wrapped keys that never leave the device in plaintext.
- **True Random Number Generator (TRNG)**: Physical entropy source for generating unpredictable cryptographic keys, nonces, and initialization vectors. FIPS 140-3 mandates validated entropy sources compliant with NIST SP 800-90B.
- **Tamper detection and response**: Physical sensors (mesh grids, voltage monitors, temperature sensors, light detectors) that trigger zeroization (immediate key erasure) upon intrusion attempts.
- **Secure boot and firmware authentication**: Cryptographically signed firmware prevents unauthorized code execution. Boot ROM verifies signatures before transferring control.
- **Memory protection**: Hardware-enforced isolation between trusted execution environments and untrusted code, using MMUs or MPUs to prevent buffer overflows from leaking keys.
- **Side-channel countermeasures**: Constant-time implementations, masking (randomizing intermediate values), noise injection, and power/clock randomization to resist differential power analysis (DPA) and timing attacks.

HSMs are designed to resist both logical attacks (software exploits, side-channel analysis) and physical attacks (probing, decapsulation, fault injection). FIPS 140-3 Level 3 and Level 4 certifications require progressively stronger tamper resistance, including epoxy coating over chips, active tamper detection circuits, and environmental failure protection.

As discussed in Related Work (subsection 3.1.6), the VE-HEP open-source HSM **lacked a True Random Number Generator**, forcing reliance on external entropy sources for cryptographic key generation. An HSM without on-chip randomness cannot generate keys autonomously: it must rely on external entropy inputs, increasing the attack surface. Integrating a hardware TRNG into the SoC is therefore a primary objective of this thesis.

High-quality randomness is essential for multiple cryptographic operations:

- **Key generation**: RSA and elliptic curve cryptography (ECC) require large random prime numbers or random field elements. Predictable key generation allows attackers to enumerate possible keys, compromising confidentiality.
- **Initialization Vectors (IVs)**: Symmetric encryption modes such as AES-CBC and AES-CTR require unique IVs for each encryption operation. Reusing IVs allows known-plaintext attacks that recover the encryption key.

- **Nonces and session keys**: Protocols like TLS and SSH generate per-session ephemeral keys using random nonces. Predictable nonces enable replay attacks and session key recovery.
- **Challenge-response authentication**: Random challenges prevent attackers from pre-computing valid responses.

Historical cryptographic failures illustrate the catastrophic consequences of weak random number generation:

- **Debian OpenSSL (2008)**: A code change accidentally reduced OpenSSH key generation entropy, enabling brute-force attacks against millions of deployed SSH servers [36].
- **Java SecureRandom nonce collision (2013)**: Bugs in Java's `SecureRandom` class generated colliding ECDSA nonce values ($k$) in Bitcoin wallet implementations on Android. When two signatures reused the same nonce, the private key could be recovered algebraically, enabling theft of Bitcoins [37].
- **Sony PlayStation 3 (2010)**: Fixed ECDSA signing nonce allowed private key recovery from two signed messages, completely breaking the console's security model and enabling unsigned code execution [38].

These incidents demonstrate that even mathematically strong cryptographic algorithms collapse when built on weak randomness. HSMs must therefore include TRNGs as a foundational security primitive, generating entropy from physical processes that cannot be predicted or reproduced by attackers.

### 2.3.2 Random Number Generation: PRNG vs TRNG

Random number generators fall into two fundamentally different categories: Pseudo-Random Number Generators (PRNGs), which use deterministic algorithms, and True Random Number Generators (TRNGs), which extract entropy from physical processes.

#### Pseudo-Random Number Generators

PRNGs are deterministic algorithms that generate sequences from an initial seed value. Given the same seed, a PRNG always produces the same output sequence. Common PRNG algorithms include:

- **Linear Congruential Generators (LCG)**: Fast but cryptographically weak (predictable from partial output).

- **Mersenne Twister (MT19937)**: High-quality statistical properties, widely used in simulations, but not cryptographically secure.
- **Xorshift family**: Fast CPU-friendly algorithms with good statistical properties, unsuitable for cryptography.
- **Cryptographically Secure PRNGs (CSPRNGs)**: ChaCha20, AES-CTR mode, designed to resist prediction even when partial state is known.

PRNGs offer significant practical advantages:

- **High throughput**: Pure computation with no hardware dependencies, generating gigabytes per second on modern CPUs.
- **Reproducibility**: Fixed seed produces identical sequence, essential for debugging simulations and reproducible testing.
- **No hardware required**: Software-only implementation, portable across platforms.

However, PRNGs have a fundamental limitation for cryptographic applications: if an attacker learns the internal state (or the seed), the entire past and future output sequence becomes predictable. This is an issue for key generation; an attacker who compromises the seed can regenerate all keys, breaking confidentiality retroactively. CSPRNGs, designed to resist state-recovery attacks, cannot provide true unpredictability: their security relies on the seed remaining secret and having sufficient entropy.

### True Random Number Generators

TRNGs extract entropy from physical processes that are fundamentally unpredictable. Unlike PRNGs, TRNG output cannot be reproduced even with complete knowledge of the system's past state, because the randomness originates from quantum mechanical phenomena or chaotic macroscopic processes.

Common entropy sources include [39]:

- **Thermal noise (Johnson-Nyquist noise)**: Random voltage fluctuations in resistors caused by thermal agitation of charge carriers. Magnitude increases with temperature and resistance, typically amplified and digitized using comparators.
- **Shot noise**: Discrete nature of electric current (quantized electron flow) produces random fluctuations in semiconductor diodes. Requires reverse-biased PN junctions and high-gain amplifiers.

- **Radioactive decay**: Alpha particle detection from isotopes like Americium-241 provides quantum-mechanical randomness. Requires specialized sensors and regulatory compliance for radioactive materials.
- **Quantum effects**: Photon arrival times at beam splitters, quantum tunneling in Zener diodes. Provides provable randomness but requires precise analog circuits or optical components.
- **Oscillator jitter**: Timing uncertainty in electronic oscillators caused by thermal noise in transistors, power supply fluctuations, and electromagnetic interference. Most practical entropy source for integrated circuits—requires only digital logic (ring oscillators) without analog amplifiers or external components.
- **Metastability**: When a flip-flop's setup/hold time is violated, the output enters an unpredictable metastable state before resolving to 0 or 1. Resolution time and final state depend on sub-threshold transistor physics. Requires careful circuit design to avoid stability issues.

Among these sources, **oscillator jitter** is most widely used in FPGA and ASIC TRNGs because:

- Digital-only implementation (no analog components, no external sensors)
- Portable across semiconductor processes
- Low area overhead (dozens of logic gates)
- Self-contained (no external dependencies)

Ring oscillator TRNGs typically use chains of inverters (odd number for oscillation) whose frequency varies randomly due to transistor-level thermal noise. Multiple oscillators running at slightly different frequencies produce unpredictable phase drift, sampled as random bits.

TRNGs provide **true unpredictability**—output cannot be predicted even with complete knowledge of past states—making them essential for cryptographic key generation. However, TRNGs have practical limitations:

- **Lower throughput**: Entropy generation limited by physical process rates.
- **Hardware dependency**: Requires dedicated circuits, increasing chip area and power consumption.
- **Environmental sensitivity**: Entropy quality degraded by extreme temperatures, voltage fluctuations, electromagnetic interference.
- **Validation complexity**: Statistical testing required to ensure sufficient randomness (min-entropy, bias, correlation analysis).

#### Hybrid Approach

A common approach combines TRNGs and CSPRNGs in a **hybrid architecture**: a TRNG generates a high-entropy seed (256-512 bits), which initializes a CSPRNG that produces high-throughput random output. The TRNG periodically reseeds the CSPRNG (e.g., every hour or after generating a threshold amount of data) to maintain long-term unpredictability. This approach appears in Intel RDSEED/RDRAND instructions, Linux `/dev/random`, and commercial HSMs.

### 2.3.3 TRNG Architecture and Quality Assurance

Raw entropy from physical sources exhibits statistical imperfections: bias (unequal probability of 0s and 1s), temporal correlation (adjacent bits not independent), and non-uniform distribution. TRNG systems employ post-processing algorithms to condition raw entropy and health monitoring to detect failures. The following subsections describe a typical TRNG system architecture before detailing these post-processing stages.

#### TRNG System Architecture

A typical TRNG implementation follows a multi-stage pipeline architecture:

1. **Entropy source**: Physical process generating random bits (ring oscillators, noise amplifiers, etc.).
2. **Digitization**: Analog signals converted to digital bits via comparators or sampling flip-flops.
3. **Conditioner**: Post-processing to remove bias and correlation (Von Neumann, XOR, hash).
4. **Health tests**: Continuous monitoring for failures (RCT, APT, alarm logic).
5. **Output buffer**: FIFO or register to decouple entropy generation rate from consumption rate.
6. **Bus interface**: Memory-mapped registers for firmware access (control, status, data).

This modular pipeline separates concerns: entropy generation (physics), conditioning (signal processing), monitoring (security), and system integration (hardware/software interface).

### Conditioning Algorithms

Conditioning (also called randomness extraction or debiasing) transforms imperfect raw entropy into uniformly distributed output:

- **Von Neumann corrector**: Examines consecutive bit pairs from the raw entropy stream. Discards correlated pairs (00 and 11), outputs 0 for 01 and 1 for 10. This removes first-order bias but suffers worst-case 4:1 compression (if raw source is already unbiased, half of all pairs are discarded). Simple hardware implementation (shift register and combinational logic).
- **XOR decorrelation**: Reduces temporal correlation by XORing consecutive bits or outputs from multiple parallel entropy sources. If sources are independent, output entropy is the sum of individual entropies. Common in ring oscillator TRNGs, where multiple oscillators are XORed together.
- **Hash-based extraction**: Applies cryptographic hash functions (SHA-256, SHA-3/SHAKE) to blocks of raw entropy. Extractors with provable security bounds exist (Leftover Hash Lemma), guaranteeing output is statistically close to uniform if input has sufficient min-entropy. Higher hardware cost (hash engine) but maximum entropy extraction efficiency.
- **Whitening**: Additional cryptographic transformation (AES in counter mode, stream ciphers) spreads entropy uniformly across output bits, decorrelating bit positions and masking statistical structure.

Conditioner choice depends on hardware resources, throughput requirements, and security target. Simple TRNGs use Von Neumann correctors; high-assurance systems use hash-based extraction with provable security.

### Health Monitoring and NIST SP 800-90B

Entropy sources can degrade over time due to component aging, temperature extremes, voltage droops, or deliberate attacks (e.g., clock glitching to force oscillator synchronization). NIST Special Publication 800-90B [40] establishes requirements for entropy sources in cryptographic applications, mandating continuous health testing:

- **Startup tests**: Execute during system initialization to detect manufacturing defects or catastrophic failures before generating cryptographic keys. Tests verify minimum entropy rate and distribution properties.
- **Continuous health tests**: Run in real-time during operation to detect gradual degradation or active attacks. Two critical tests:

  - **Repetition Count Test (RCT)**: Counts consecutive identical bits. If the same bit value repeats more than a threshold $C$ times (typically 33-65 depending on entropy rate), an alarm is raised. Detects stuck-at faults, oscillator shutdown, or clock glitching attacks.
  - **Adaptive Proportion Test (APT)**: Counts the number of ones in a sliding window of $W$ samples (e.g., 1024 bits). If the count falls outside the range $[W/2-C, W/2+C]$ (where $C$ is derived from expected min-entropy), an alarm is raised. Detects statistical bias drift, correlated outputs, or frequency injection attacks.

- **Alarm generation**: When a health test fails, the TRNG must halt operation immediately and signal the failure to the system (via status register or interrupt). Cryptographic operations must not proceed with potentially weak entropy. Hardware or software intervention is required to diagnose and resolve the fault before re-enabling the TRNG.

NIST SP 800-90B also requires **entropy estimation**: measuring the min-entropy (worst-case entropy per bit) of the raw source using statistical tests over large sample sets. This validation is typically performed during design characterization, not in real-time hardware.

### 2.3.4 TRNG Integration in SoC Architectures

Integrating a TRNG into a SoC requires design decisions about hardware placement, interface protocols, and software access patterns. A TRNG can connect to the processor through several integration approaches, with distinct trade-offs in complexity, latency, and portability.

#### TRNG Integration Approaches Overview

Possible approaches for integrating a TRNG into a SoC include:

1. **Direct on-chip integration**: The TRNG is synthesized directly into the FPGA fabric or ASIC alongside other system components. Provides minimum latency and maximum physical security (no external interfaces vulnerable to probing). This is the form in which OpenTRNG provides its PTRNG IP core: HDL source code (Verilog/VHDL) ready for synthesis, but *without* infrastructure for bus integration or processor access—only the bare entropy generation pipeline (digitalnoise → conditioner → health tests → bitpacker).

2. **Memory-mapped peripheral on standard bus**: The TRNG connects to the processor via a standard bus protocol (APB3, AXI4-Lite, Wishbone,

Avalon-MM) and appears as memory-mapped registers. Software reads random data as if accessing hardware registers at fixed addresses (e.g., 0xF0030000). This is the most common approach in SoC designs and the method adopted in this thesis, adding the bus integration infrastructure that OpenTRNG does not provide natively.

3. **Dedicated hardware interface to other IP**: The TRNG exposes a simple interface (valid/ready handshake, FIFO ports) directly to another hardware module without processor involvement. Common when the entropy consumer is a cryptographic accelerator (AES, RSA engine) rather than firmware. Eliminates bus protocol overhead but couples TRNG to specific consumer IP.

4. **External module on serial interface**: The TRNG resides in a dedicated external chip communicating via serial protocol (SPI, I2C, UART). Advantages: Specialized analog design, certified hardware security modules available off-the-shelf. Disadvantages: External dependency (supply chain, BOM cost), vulnerable to bus snooping, higher latency.

**Positioning of this work.** OpenTRNG provides approach 1 (on-chip HDL synthesis) but lacks SoC integration infrastructure. This thesis bridges the gap by implementing approach 2 (memory-mapped peripheral with bus protocol). The choice of memory-mapped integration over alternatives (coprocessor instructions, DMA, external chip) is discussed in Methodology section 4.3.

### Memory-Mapped Peripheral Interface

This section details the memory-mapped peripheral approach (approach 2 above), which this thesis adopts for OpenTRNG integration. As introduced in Background subsection 2.2.3, memory-mapped peripherals connect via standard bus protocols (APB3 for low-speed register access, AXI4-Lite for higher throughput, or Avalon-MM in Intel FPGA systems).

A typical TRNG peripheral exposes three categories of memory-mapped registers:

- **CONTROL register** (write): Enable/disable entropy generation, configure conditioning parameters (oscillator mask, frequency dividers), reset internal state.
- **STATUS register** (read-only): Data ready flag (indicates valid output available), alarm flags (health test failures), FIFO depth indicators (for burst reads).
- **DATA register** (read-only): Random output word (typically 32 bits). Reading may auto-clear the data ready flag (read-once semantics to prevent duplicate values).

Firmware accesses these registers via memory-mapped I/O (MMIO), dereferencing pointers to the peripheral's base address (e.g., 0xF0030000).

### Firmware Access Patterns

Two primary patterns exist for firmware to retrieve random numbers:

**Polling-based access.** Firmware busy-waits in a loop checking the STATUS register until the data ready flag is set, then reads the DATA register.

- **Advantages**: Simple implementation (no interrupt handler, no concurrency issues), easy debugging, deterministic execution flow.
- **Disadvantages**: CPU cycles wasted during entropy generation latency (typically 1-10 ms per 32-bit word depending on conditioning).
- **Use case**: Low-frequency operations (key generation at boot, periodic reseeding), single-threaded bare-metal firmware.

**Interrupt-driven access.** TRNG asserts an interrupt when data becomes available. CPU services the interrupt asynchronously, allowing other tasks to execute during entropy generation.

- **Advantages**: CPU executes other tasks while waiting for entropy, higher system throughput, better resource utilization in multi-tasking environments.
- **Disadvantages**: Interrupt handler complexity, priority management if multiple peripherals share interrupts, non-deterministic execution timing (complicates debugging).
- **Use case**: High-throughput applications, RTOS environments, concurrent firmware tasks.

**The `volatile` qualifier.** Register accesses must use the `volatile` qualifier (ISO C99/C11 Section 6.7.3) to prevent the compiler from caching values in CPU registers. Without it, an optimising compiler may legally eliminate the polling loop entirely, reasoning that STATUS never changes between iterations. Marking the struct fields `volatile` forces a fresh memory load on every access, which is the only correct behaviour for memory-mapped hardware.

## 2.4 Simulation & Verification

Digital hardware verification requires simulating RTL designs (Verilog/VHDL) before synthesis to validate functional correctness, protocol compliance, and integration behavior. This section surveys HDL simulation approaches and testbench methodologies relevant to SoC development.

### 2.4.1 HDL Simulation Approaches

HDL simulators execute register-transfer level (RTL) designs, modeling hardware behavior at clock-cycle granularity. Two primary approaches exist: **event-driven simulation** (traditional, 4-state logic) and **cycle-accurate compilation** (modern, 2-state logic).

#### Event-Driven Simulators

Traditional HDL simulators (ModelSim, Icarus Verilog, GHDL) use event-driven kernels that process signal changes as discrete events in a priority queue. These simulators support **4-state logic** (`0`, `1`, `X` (unknown), `Z` (high-impedance)), enabling detection of uninitialized signals, bus contention, and metastability. The IEEE 1364 (Verilog) and IEEE 1076 (VHDL) standards define simulation semantics (delta cycles, non-blocking assignments, process scheduling).

**GHDL.** GHDL [41] is an open-source VHDL simulator that compiles VHDL code to native machine code (via GCC or LLVM), executed directly on the host CPU. GHDL implements the full IEEE 1076 standard (VHDL-93, VHDL-2002, VHDL-2008), supporting advanced features like protected types, shared variables, and sensitivity lists. Waveform output formats include VCD (Value Change Dump) and FST (Fast Signal Trace, compressed binary format).

Event-driven simulation is **flexible** (supports analog behavior, tri-state buses, arbitrary timing) but **slow**—typical throughput is 1-10 kHz for complex SoCs, making firmware-driven testing (millions of cycles) impractical.

#### Cycle-Accurate Compilers

Cycle-accurate simulators (Verilator, Synopsys VCS with `-cc`) **compile Verilog to C++**, generating an executable model that updates all signals on each clock edge. This approach trades flexibility for **speed** (10-100× faster than event-driven), sacrificing 4-state logic (`X`/`Z` not modeled) to enable efficient native code execution.

**Verilator.** Verilator [42] is an open-source Verilog simulator that compiles RTL to a cycle-accurate C++ model. The generated code is compiled with GCC/Clang and linked with a C++ testbench, or driven via Python through frameworks such as Cocotb, that drives the clock and reset signals. Because execution runs as native code rather than through an event-driven kernel, Verilator is substantially faster than traditional simulators—a property that matters for system-level tests where the CPU must execute thousands of firmware instructions. Verilator operates on a synthesizable Verilog subset (no full four-state simulation, no behavioral delays) and is widely used in open-source SoC projects including OpenTitan, LiteX, and VexRiscv.

### 2.4.2 Testbench Frameworks

Testbenches provide stimulus to the design-under-test (DUT) and verify outputs against expected behavior. Testing is commonly organized by scope: **block-level testing** targets a single component in isolation, while **system-level testing** validates the complete SoC with a CPU executing real firmware.

#### Block-Level Testing

Block-level testing verifies a single peripheral independently of the CPU and memory subsystem. A software driver—often called a Bus Functional Model (BFM)—generates bus transactions (APB3, AXI4) directly, without a processor. This isolates the component under test from system complexity, making it possible to exercise corner cases (protocol violations, backpressure, alarm conditions) early in the design cycle, before full SoC integration.

#### System-Level Testing

System-level testing instantiates the complete SoC—CPU, bus arbiter, and all peripherals—and executes bare-metal firmware compiled for the target instruction set. The CPU performs MMIO transactions to peripheral registers, and results are observed via UART output or memory inspection. This approach validates end-to-end integration and firmware correctness, at the cost of higher simulation time (millions of clock cycles per test).

Debugging combines two complementary techniques. The more immediate one is observing UART output: firmware prints status messages to a UART peripheral, which the testbench decodes from TX pin transitions. This is a lightweight, direct approach—analogous to inspecting program output rather than stepping through a debugger. For failures that require signal-level visibility, simulators can emit waveform dumps (VCD or FST format) that record every signal transition. GTKWave [43] opens these dumps and exposes the full design hierarchy through

a Signal Search Tree (SST), allowing any internal signal to be inspected across time.

```nix
{
  inputs = {
    nixpkgs.url = "github:NixOS/nixpkgs/nixos-25.11";
  };

  outputs =
    { self, nixpkgs, ... }:
    let
      system = "x86_64-linux";
      # legacyPackages: lazy access to the full nixpkgs set (see text)
      pkgs = nixpkgs.legacyPackages.${system};
    in
    {
      # nix build .#hello  /  nix build .
      packages.${system}.hello = import ./packages/hello.nix { inherit pkgs; };
      packages.${system}.default = self.packages.${system}.hello;

      # nix run .  (custom execution logic, e.g. wrapper scripts)
      apps.${system}.default = {
        type = "app";
        program = "${self.packages.${system}.hello}/bin/hello";
      };

      # nix develop
      devShells.${system}.default = import ./shell.nix { inherit pkgs; };

      # nix flake check
      checks.${system}.hello-builds = self.packages.${system}.hello;

      # nix fmt
      formatter.${system} = pkgs.nixfmt-tree;

      # consumed by other flakes: overlays.default
      overlays.default = final: prev: {
        hello = self.packages.${system}.hello;
      };

      # nixos-rebuild switch --flake .#myhost
      nixosConfigurations.myhost = nixpkgs.lib.nixosSystem {
        inherit system;
        modules = [ ./hosts/myhost/configuration.nix ];
      };
    };
}
```

**Listing 2.14:** Flake outputs example with imported derivations

# Chapter 3

# Related Work

This chapter surveys existing open-source hardware projects relevant to TRNG integration and reproducible toolchain management. We examine two key projects: VE-HEP (section 3.1), demonstrating a complete open-source ASIC tape-out workflow but *without* a hardware TRNG, and OpenTRNG (section 3.2), providing reference PTRNG IP for integration. The gap analysis (section 3.3) identifies a build-pipeline limitation in VE-HEP and the absence of a worked SoC integration example for OpenTRNG, both of which motivate this thesis's contributions.

## 3.1 VE-HEP: Open-Source Hardware Security Module

### 3.1.1 Project Overview

Project HEP (Hardening the value chain through open-source EDA tools and processors) is a German BMBF-funded initiative demonstrating ASIC production using exclusively open-source tools and designs [44]. The project produced a hardware security module (HSM) prototype from SpinalHDL to GDS using OpenROAD/OpenLane, successfully taping out at IHP's MPW service in 2023. The design flow established: SpinalHDL → Verilog → Yosys → OpenROAD → GDS, integrated with formal verification (SymbiYosys) and automated masking (EASIMask tool). Results published at DATE 2024 and code released on GitHub (HEP-Alliance/VE-HEP-HW-SW), packaged in a Nyx container for reproducibility.

### 3.1.2 Architecture

The HEP chip integrates a VexRiscv RISC-V CPU (RV32IM ISA, excluding hardware division) implemented in SpinalHDL [27, 29], with 256 KB on-chip SRAM

and peripherals (UART, SPI, GPIO; JTAG added in second tape-out). Security features include an AES-128 accelerator with automated masking (EASIMask tool) and a BigNum accelerator for large-integer arithmetic. The chip occupies 11.5 $mm^2$ (IHP SG13G2 130nm), with logic area 1.3 $mm^2$ and SRAM 10.2 $mm^2$, running at 25 MHz. Notably, a hardware TRNG was **not integrated**: the paper explicitly states that "non-volatile memory and a trustworthy random number generator have not been integrated because no suitable open-source components are available" [30].

### 3.1.3 Toolchain and Verification

The project demonstrates a complete open-source flow: SpinalHDL → Verilog → Yosys → OpenROAD/OpenLane → GDSII, fabricated via IHP MPW service (SG13G2 PDK, 130nm). However, the flow required commercial tools for scan insertion and proprietary DRC/LVS checkers where open-source alternatives were incomplete. Verification uses SymbiYosys for formal ISA correctness, and EASIMask tool for automated side-channel masking of the AES accelerator, validated on FPGA before tape-out.

### 3.1.4 Reproducibility Approach

VE-HEP uses Nix to provide a reproducible development environment: tool dependencies (Yosys, OpenROAD, Verilator, SpinalHDL/Gradle, RISC-V GCC) are declared in `shell.nix` via `pkgs.mkShell`, with nixpkgs inputs pinned to fixed versions. The project additionally packages the toolchain into a Nyx container [30] for offline distribution.

However, Nix derivations are used only to **instantiate the development shell**, not to produce build outputs. Firmware compilation, RTL generation, and simulation all execute as imperative scripts (Gradle, OpenLane) inside that shell. Consequently, build outputs reside in the source tree rather than the content-addressed Nix store, there is no Nix-level dependency graph between build stages, and long-term reproducibility of outputs is not guaranteed.

### 3.1.5 Results and Future Work

The project successfully taped out a functional open-source RISC-V ASIC (IHP MPW 2023), demonstrating a working CPU core, AES accelerator with side-channel masking, and peripherals. This validated the SpinalHDL → OpenROAD → GDS flow with formal verification and automated masking integration. The paper notes the design is "not optimized for area nor speed" due to OpenLane resizer limitations [30]. Future work identified includes developing an open European

SRAM generator, completing open-source DRC/LVS checkers, and automated metal filling to eliminate remaining proprietary dependencies.

### 3.1.6 Identified Gaps

Two key gaps relevant to this thesis:

1. **Build-level reproducibility gap**: Nix provides a reproducible development environment, but build stages (firmware, RTL, simulation) run as imperative scripts inside that environment—not as Nix derivations. Build outputs reside in the source tree with no content-addressed guarantee and no declarative dependency graph.

2. **No TRNG**: The design explicitly excludes a hardware random number generator due to the absence of suitable open-source components at the time, leaving entropy generation unaddressed.

## 3.2 OpenTRNG: Open-Source TRNG IP Core

### 3.2.1 Project Overview

OpenTRNG [45, 46] provides reference PTRNG implementations based on ring oscillators, maintained by CEA-Leti under MIT license. The project focuses on FPGA validation (Xilinx Artix-7), entropy analysis, and NIST compliance evaluation, providing Verilog/VHDL sources with emulation and testing infrastructure. The README explicitly disclaims production readiness: "reference implementations... for academic and research purposes only... must not be used as is in products." This positions OpenTRNG as an *educational reference* for researchers building custom TRNG solutions.

### 3.2.2 PTRNG Architecture

The PTRNG implements a modular pipeline: raw entropy (`digitalnoise.v`) → conditioner (`conditioner.v`, extended Von Neumann encoder, optional) → bitpacker (`bitpacker.v`, 32-bit word assembly) → `data`/`valid` output. Two health monitors run in parallel on the raw entropy stream: `alarm.v` (run-length counter, raises `alarm_detected`) and `onlinetest.v` (cumulative-sum test, raises `onlinetest_valid`). Neither gates the data path. Output is 32-bit words with a `valid` strobe.

The PTRNG module exposes a **bare-metal interface** with 15 input ports (3 control + 12 configuration: `ring_en`, `freqdivider_value`, `alarm_threshold`, conditioning controls, online test parameters) and 7 output ports (`data`, `valid`,

`alarm_detected`, and frequency counter/online-test status signals). The module does not provide a standard bus interface (APB3, AXI4, Avalon)—users must implement their own bus wrapper.

Firmware operates the TRNG by configuring entropy sources (ring oscillator enable masks), conditioning parameters (frequency dividers, Von Neumann enable), health test thresholds, and polling the `valid` strobe before reading `data` outputs. Alarm flags indicate health test failures requiring system intervention.

### 3.2.3 Verification and Status

The repository provides Cocotb testbenches (Python) for GHDL and Verilator, verifying module-level functionality. FPGA validation targets the Xilinx Artix-7 (Arty A7). Consistent with its standalone-IP scope, the repository does not include SoC integration examples, firmware drivers, or system-level testbenches—each of these is left to adopters. Statistical entropy evaluation relies on external NIST SP 800-22 tools (not automated). Build scripts are ad-hoc (shell, Python) without version pinning or reproducible workflows. OpenTRNG provides true ring oscillator implementations in `digitalnoise.v` (ERO/MURO/COSO digitalizer topologies), but Cocotb testbenches use `digitalnoise_stub.v`—an empty module—and inject deterministic values directly into `raw_random_number`/`raw_random_valid`, testing only the downstream post-processing modules (`conditioner.v`, `onlinetest.v`, `bitpacker.v`) in isolation.

## 3.3 Gap Analysis and Thesis Positioning

### 3.3.1 Summary of Identified Gaps

Analysis of VE-HEP and OpenTRNG reveals two key gaps that motivate this thesis:

1. **Non-hermetic build pipeline**: VE-HEP uses Nix to provide a reproducible development environment—a good foundation—but build stages (firmware, RTL generation, simulation) execute imperatively inside that shell, not as Nix derivations. Build outputs go to the source tree, not the content-addressed Nix store. No published HDL project exposes all build stages as proper Nix derivations with a declarative, hermetic dependency graph.

2. **No published TRNG SoC integration**: OpenTRNG intentionally scopes itself to standalone IP; VE-HEP explicitly excluded a hardware TRNG due to the absence of suitable open-source components at the time.

### 3.3.2 Thesis Contributions

This thesis addresses the identified gaps through two primary contributions:

1. **OpenTRNG SoC integration** (Chapter 5): First reference design integrating OpenTRNG PTRNG into a VexRiscv RISC-V SoC. Demonstrates SpinalHDL BlackBox wrapping of Verilog IP with APB3 bridge, C firmware drivers for TRNG control (initialization, polling, data reading), and Cocotb testbenches for system-level verification (CPU → APB3 → TRNG transaction sequences). Addresses Gap 2.

2. **Derivation-based HDL build pipeline** (Chapter 5): Where VE-HEP uses Nix derivations only to instantiate a development shell, this thesis expresses every build stage—firmware cross-compilation, RTL generation, and simulation—as a proper Nix `mkDerivation`, so all outputs enter the content-addressed Nix store. The pipeline is exposed as a Nix Flakes project (`flake.lock` pins all inputs cryptographically, outputs are addressable as flake attributes). Establishes a reusable Gradle FOD pattern for JVM-based HDL tools (SpinalHDL, Chisel) capturing 426 Maven dependencies. Addresses Gap 1.

Both contributions are published with complete source code and documentation, enabling replication and extension by future researchers.

# Chapter 4

# Methodology

This chapter justifies the technological choices and design decisions for integrating a True Random Number Generator (TRNG) into a RISC-V SoC using a fully reproducible Nix-based workflow. The parallel structure mirrors Chapter 2, addressing design rationale across four technical areas:

- **Build System Choice** (section 4.1): Why Nix for reproducible builds, comparisons with Docker and Bazel, and the rationale for adopting Flakes.
- **Hardware Design Choices** (section 4.2): Justification for SpinalHDL, VexRiscv, APB3, and BlackBox integration patterns.
- **Security Primitive Choice** (section 4.3): Why OpenTRNG PTRNG, integration approach, and firmware-based testing strategy.
- **Simulation Tool Choices** (section 4.4): Rationale for Verilator, Cocotb, and the adopted testing strategy.

## 4.1 Build System Choice

### 4.1.1 Why Nix?

This section justifies the choice of Nix as the build system for the thesis, addressing the gaps identified in Chapter 3. VE-HEP [44] uses Nix to pin tool dependencies in a development shell, but build stages execute imperatively inside that environment rather than as proper Nix derivations—the key reproducibility gap identified in subsection 3.1.6. OpenTitan [47] employs Bazel, a build system suited to monorepo scale but requiring significant `BUILD` file overhead and a migration of the project's build logic (discussed in section 4.1.1).

As discussed in Chapter 2.1, Nix's functional package management model provides immutability and isolation (Nix store), explicit dependencies (hash-based addressing), declarative specification (functional language), cross-distribution portability, and reproducibility (identical inputs produce identical outputs). Among functional package managers, the only meaningful point of comparison is GNU Guix [48], which was directly inspired by Nix and shares the same core model. However, Nix benefits from a considerably larger package collection (nixpkgs, 130,000+ packages [4]), a broader community, and a more mature ecosystem of tooling—making it the practical choice for a project with heterogeneous dependencies. The following properties of Nix are particularly relevant here—and, more broadly, for any academic hardware project where reproducibility is a first-class concern [1] (see subsection 2.1.1):

- **Hermetic evaluation**: no implicit system dependencies; pure evaluation without `NIX_PATH` means the build is self-contained and portable across machines.
- **Long-term reproducibility**: the combination of content-addressed store and pinned inputs means a future researcher can check out this repository and rebuild the exact toolchain used today.
- **Multi-stage declarative pipelines**: the derivation model naturally expresses build graphs where one output feeds the next (RTL generation → firmware → simulation), with each stage hermetically defined.
- **Multi-language**: nixpkgs spans all languages and build systems—JVM/Gradle, C/GCC, C++, Python, and more—under a single coherent model, eliminating per-language toolchain managers. Being cross-distribution, it works identically on any Linux (or macOS) host without relying on system-provided libraries.
- **Cross-compilation**: unlike most package managers that ship pre-built cross-toolchains as individual packages, nixpkgs treats cross-compilation as a first-class property of the package set: any package can be cross-compiled by specifying the target, with no manual toolchain assembly required [11] (see section 2.1.5).

No other single tool satisfies all of these jointly. Despite a steep learning curve and documentation spread across multiple sources (discussed in section 2.1), the investment is justified: the reproducibility and hermeticity guarantees are permanent properties of the build artifacts.

### Nix vs Containers

One of the most common questions, given that Nix is associated with reproducible environments, is how it compares to containerization tools such as Docker [49].

The comparison is technically imprecise: Docker is a deployment tool, not a package manager, and offers a reproducible run-time environment rather than a reproducible build. What people usually mean is the contrast between two approaches to managing dependencies:

- **Container workflow**: a Dockerfile starts from a base image (e.g., Debian) and runs imperative steps (`RUN apt-get update && apt-get install ...`). The resulting image is a snapshot of the system at build time, not guaranteed to be identical across rebuilds or hosts.

- **Nix**: builds are pure functions of declared inputs, with outputs stored immutably in the Nix store. The same derivation produces the same result on any machine, regardless of the host distribution.

The distinction that matters is therefore **imperative snapshots versus functional builds**. Since they serve different purposes, they can be combined: Nix can be installed inside containers (e.g., Docker, Podman) when users lack root privileges on the host—a common pattern in HPC and academic environments.

### Nix vs Bazel

Bazel [50] is a build system developed at Google, adopted by large-scale hardware projects such as OpenTitan. Both Nix and Bazel aim for reproducible, hermetic builds—but they differ on several axes:

- **Hermeticity**: Nix enforces full isolation for every derivation; all dependencies must be explicitly declared and the build cannot access anything outside them. Bazel also sandboxes builds, but is less hermetic by default.

- **Scope**: Nix is a package manager that wraps existing build tools (Make, CMake, Gradle, etc. as native build systems inside hermetic derivations), without replacing them. Bazel is a build system that typically replaces those tools, requiring the project's build logic to be expressed in its own rule language (Starlark) via `BUILD` files.

- **Integration effort**: introducing Nix into an existing project does not require restructuring its build logic. Adopting Bazel typically amounts to a migration of the build system itself.

- **Scalability**: Bazel's strengths—fine-grained incremental builds, aggressive caching, parallel execution—are most valuable at monorepo scale with thousands of targets.

They can be combined [51, 52, 53]: Nix provides pinned toolchains and a hermetic environment, while Bazel handles incremental builds within it. For this thesis, with a linear RTL → firmware → simulation pipeline and acceptable build times (∼3 minutes), pure Nix suffices. OpenTitan's adoption of Bazel reflects its monorepo scale; this project's pipeline does not justify that overhead.

### 4.1.2 Why Flakes?

Flakes are not strictly required for reproducible Nix builds—native pinning and tools like npins are valid alternatives (see subsection 2.1.4). Two properties make them the right choice for this project specifically:

- **Pipeline composability**: this project has a linear RTL → firmware → simulation pipeline in which each stage consumes a previous stage's output (e.g., the RAM-initialized RTL variant embeds `${firmware}/firmware.hex`). Flakes' outputs schema maps each stage to a named, independently buildable artifact (`nix build .#rtl`, `.#firmware`, `.#simulate`), with the dependency wiring explicit in `flake.nix`.

- **Self-describing project interface**: `nix flake show` lists every artifact the project exposes—packages, development shells, checks, formatter—without reading any documentation. Combined with `flake.lock`, the repository is fully self-contained: a user cloning it sees what can be built and can reproduce any stage exactly.

The trade-offs of flakes' experimental status are acknowledged in section 2.1.6.

## 4.2 Hardware Design Choices

### 4.2.1 Why SpinalHDL?

The choice of SpinalHDL as the hardware description language for the Taurus SoC is based on the VE-HEP project (Related Work section 3.1), which successfully demonstrated SpinalHDL and VexRiscv viability for open-source ASIC tape-out. This thesis builds upon that validated toolchain, extending the design with a standardized TRNG peripheral.

Since VexRiscv is itself written in SpinalHDL, using the same language for the surrounding SoC allows the CPU to be instantiated as a plain SpinalHDL component alongside the peripherals, keeping the entire design in a single language and build flow.

#### SpinalHDL Language Features

As SpinalHDL is based on a high-level language, it provides several advantages over traditional HDLs [54]:

- **No more endless wiring:** Create and connect complex buses like AXI in one single line. The `Apb3SlaveFactory` provides a declarative API for peripheral register maps.
- **Type conversions:** Bidirectional translation between any data type and bits, with complex data structures loadable from memory-mapped registers with minimal boilerplate.
- **Design checks:** Early stage lints to check that there are no combinatorial loops or latches. Signal width mismatches are caught at compile time.
- **Clock domain safety:** Early stage lints to ensure there are no unintentional clock domain crossings; any crossing must be explicitly declared.
- **Generic design:** No restrictions to the genericity of hardware descriptions by using Scala constructs (parametric generators, conditional instantiation).

**BlackBox for IP integration.** The BlackBox construct allows wrapping external Verilog or VHDL modules within SpinalHDL designs. This is essential for integrating the OpenTRNG PTRNG (Section 5.3.1), which is provided as Verilog source code. SpinalHDL's Verilog output is then used downstream with Verilator for simulation.

### 4.2.2 Why RISC-V and VexRiscv?

#### Open-Source ISA: RISC-V

RISC-V [28] was selected for its royalty-free licensing, modular ISA design, and mature open-source ecosystem (GCC, LLVM, OpenOCD)—properties detailed in Section 2.2.2 and aligned with the open-source design philosophy of the VE-HEP project [30].

#### VexRiscv as RISC-V Implementation

Among available RISC-V soft-cores, VexRiscv [29] was selected based on the VE-HEP reference design [30]:

**Proven and native.** VexRiscv was successfully taped out in the VE-HEP project (IHP SG13G2 130nm) [30], validating its ASIC suitability. Being written in SpinalHDL, the CPU core, bus infrastructure, and peripheral wrappers all share a single language and build system—the entire design, from CPU plugins to TRNG register map, is expressed in one toolchain with no cross-language integration overhead.

**Murax as reference architecture.** The SoC is based on Murax [29], a minimal VexRiscv demo bundling CPU, RAM, UART, GPIO, and timer. Starting from a working SoC concentrates effort on TRNG integration and verification rather than bus infrastructure.

**Plugin-based configurability.** VexRiscv's plugin system (Section 2.2.2) allows precisely scoping the CPU to the target workload. The VE-HEP project [30] already demonstrated that a minimal no-cache, no-multiply configuration is sufficient for ASIC tape-out; this thesis adopts the same baseline. Peripheral polling and register access require only integer ALU, hazard interlocking, and a single CSR set—hardware that would be added by branch prediction, caches, or a multiplier would serve no purpose and increase area.

**APB3 peripheral support.** Since VexRiscv is implemented in SpinalHDL, it lives in the same library ecosystem: `PipelinedMemoryBusToApbBridge` from `spinal.lib.bus.simple` connects the arbiter-merged `PipelinedMemoryBus` (produced by `MuraxMasterArbiter` from VexRiscv's native bus interfaces) to the APB3 domain, enabling straightforward memory-mapped I/O integration (GPIO, UART, Timer, TRNG) without custom adapter logic.

### 4.2.3 Why APB3?

The Taurus SoC inherits the APB3 peripheral bus from the Murax reference architecture, which already uses `PipelinedMemoryBusToApbBridge` and provides its peripherals (GPIO, UART, Timer) as APB3 components. Evaluating the available alternatives confirms that APB3 is the appropriate choice for this use case—not merely an inherited default.

#### SpinalHDL Bus Library Options

SpinalHDL provides native support for multiple industry-standard bus protocols in its `spinal.lib.bus` package:

- **AHB-Lite3**: Simplified AHB variant for single-master systems with burst support

- **Apb3**: APB3 protocol for low-bandwidth peripheral access
- **Axi4**: Full AXI4 protocol with five independent channels for high-performance interconnects
- **AvalonMM**: Intel/Altera's Avalon Memory-Mapped interface for Quartus/Platform Designer integration
- **Tilelink**: Berkeley's open-source cache-coherent interconnect protocol from the Rocket Chip project

### VexRiscv Ecosystem Support

The `IBusSimplePlugin`/`DBusSimplePlugin` outputs are fused by `MuraxMasterArbiter` into a single `PipelinedMemoryBus`. Within `spinal.lib.bus.simple`, `PipelinedMemoryBusToAp` is the only bridge defined for this type—there is no equivalent to AXI4 or AHB-Lite3. Conversion methods such as `.toAxi4()` exist on the raw plugin output types (`IBusSimple`/`DBusSimple`), but using them would mean bypassing `MuraxMasterArbiter` entirely and exposing separate master ports per bus—discarding the Murax bus infrastructure. APB3 is therefore the most convenient bridge target: it reuses existing Murax infrastructure without requiring any additional adapter logic. On the peripheral side, `Apb3SlaveFactory` provides a declarative register-map API, with address decoding, `PRDATA` muxing, and `PREADY` assertion generated automatically.

### Low-Bandwidth Peripheral Requirements

All four Taurus peripherals (UART at 115200 baud, TRNG, GPIO, Timer) are control-register oriented with no burst access needs. APB3's non-burst single-transfer protocol with optional PREADY wait states provides sufficient throughput; bus bandwidth is never a bottleneck.

### Simplicity Over Performance

APB3 offers significant advantages for verification and debugging:

- **Simple handshake**: Three-signal protocol (PSEL, PENABLE, PREADY) compared to AXI4's five channels
- **Non-burst transactions**: Single-transfer protocol with no burst support simplifies peripheral state machines
- **Synchronous design**: All signals sampled at clock edge, no asynchronous handshakes

- **Waveform inspection**: Easy to trace register reads/writes in simulation

These properties reduce testbench complexity and accelerate design iteration.

#### Trade-offs

APB3 lacks burst transfers, supports only a single master, and has a minimum 2-cycle per-transaction overhead. For Taurus—low-bandwidth peripheral control with a single CPU—these constraints are irrelevant.

## 4.3 TRNG Peripheral Choice

### 4.3.1 Why OpenTRNG PTRNG?

OpenTRNG (Related Work section 3.2) provides reference TRNG implementations maintained by CEA-Leti under MIT license. The PTRNG IP core was selected because it is open-source (MIT licensed), built around a modular Verilog pipeline, ships with Cocotb testbenches that serve as verification templates, includes built-in health monitoring, and has been validated on Artix-7 FPGAs. Custom TRNG design was ruled out due to the depth of statistical validation it would require; vendor-supplied IP (Xilinx XADC, Intel TRNG) conflicts with the open-source goals of the project and does not carry over to an ASIC flow.

### 4.3.2 Simulation Entropy Source Strategy

Ring oscillator entropy sources cannot be simulated: Verilog simulators model deterministic gate-level logic, not transistor-level thermal noise, so the `digitalnoise` module requires a substitute for simulation. OpenTRNG's own testbenches sidestep this by injecting values directly from Python (`dut.raw_random_number.value = value`), bypassing `digitalnoise` altogether. This thesis additionally implements a Verilog LFSR stub (`digitalnoise_stub.v`) at the `digitalnoise` interface, generating a continuous pseudo-random bit stream that exercises the full downstream pipeline (`conditioner.v`, `alarm.v`, `onlinetest.v`, `bitpacker.v`) with more realistic input than a Cocotb-injected constant. Direct value injection at the `digitalnoise` output port from the testbench remains possible; the stub was chosen to observe a more realistic stream of bits.

In the Cocotb testbench only `digitalnoise` is swapped out, leaving conditioner, health tests, and bitpacker running with pseudo-random input; the goal is to verify that the PTRNG's own pipeline handles the bit stream correctly end-to-end. The concrete stub implementation is in Implementation subsection 5.3.1.

### 4.3.3 Verification Approach and Limitations

For this integration, polling is preferred over interrupt-driven access (see Background section 2.3.4): the expected usage is infrequent key generation on single-threaded bare-metal firmware, where the simplicity of a busy-wait loop outweighs the cost of stalling the CPU for the few milliseconds the entropy pipeline needs.

#### System-Level Testing

The verification relies on firmware-driven testing: the bare-metal RISC-V binary drives the Cocotb testbench, exercising the CTRL, STATUS, and DATA registers through a sequence of writes, polls, and reads that together trace the full CPU-to-peripheral transaction path. The concrete test sequence is described in Implementation subsection 5.4.2.

## 4.4 Simulation Tool Choices

Selecting simulation tools for SoC verification involves trade-offs among execution speed, language support, and ecosystem compatibility. This section justifies the choice of Verilator (cycle-accurate Verilog simulator) and Cocotb (Python testbench framework) over alternatives, and defines the testing strategy adopted for this thesis.

### 4.4.1 Why Verilator

Among HDL simulators (Background subsection 2.4.1), Verilator was selected for **firmware-driven testing** (complete SoC with CPU executing bare-metal code) based on speed, reproducibility, and ecosystem maturity.

#### Selection Rationale

**Execution speed.** System-level firmware tests require millions of clock cycles (boot sequence, UART echo, TRNG polling loop). Verilator compiles Verilog to optimized C++ and is significantly faster than event-driven simulators such as GHDL or Icarus Verilog, making it the practical choice for this class of tests. The VexRiscv community standardizes on Verilator for SpinalHDL-based SoC testing [29], providing validated build patterns.

**Open-source and Nix compatibility.** Verilator is MIT-licensed and packaged in `nixpkgs`, enabling hermetic builds with pinned versions. Nix flakes can declare `verilator` as a build input, ensuring reproducible simulation environments (2030

researcher can reproduce 2026 test results by evaluating the flake). This aligns with the thesis goal of demonstrating reproducible open-source methodologies.

**Ecosystem alignment.** Verilator is the simulator of choice for several prominent open-source SoC projects: OpenTitan uses it for block-level and system-level testing [55], LiteX defaults to it for simulation [56], and VexRiscv's own documentation demonstrates Verilator-based firmware testing [29]. Adopting the same tool reduces friction when cross-referencing those projects' build patterns.

#### Alternatives Considered

**GHDL.** SpinalHDL can emit both Verilog and VHDL, and OpenTRNG's main branch provides working VHDL sources, so GHDL would be a technically viable path. The choice of Verilator was a deliberate preference: given the speed advantage for system-level tests and the ecosystem alignment described above, the Verilog RTL backend was selected to match.

### 4.4.2 Why Cocotb

Cocotb [57] was selected as the testbench framework for Taurus SoC simulation, using Verilator as its simulator backend via VPI. Running `make sim-verilator` invokes Cocotb with the Verilog RTL and the real OpenTRNG pipeline, consistent with the Verilog-primary toolchain.

#### Selection Rationale

OpenTRNG's own simulation infrastructure uses Cocotb for PTRNG component verification (Related Work subsection 3.2.3), which made it the natural choice for this thesis as well. The same direct-injection approach remains available; this thesis additionally implements an LFSR stub at the `digitalnoise` interface so the downstream pipeline (conditioning, health checks, bit packing) receives a continuous pseudo-random bit stream—giving more realistic behavior to observe in simulation. Python's `async/await` syntax maps naturally to sequential test scenarios (boot sequence, UART echo, TRNG polling loop), and Cocotb's per-test pass/fail reporting in JUnit XML format makes it straightforward to isolate and rerun individual scenarios without modifying the simulator invocation.

### 4.4.3 Testing Strategy

The strategy combines three complementary techniques:

**Firmware-driven verification.** The most pragmatic check is firmware-driven: the complete Taurus SoC boots, the firmware enables the TRNG, polls the STATUS register, and reads DATA, confirming via UART output that random values propagate end-to-end through the real APB3 interface. This validates the full path from CPU instruction to peripheral response.

**Cocotb test harness.** Cocotb structures the simulation as discrete named test cases with per-test pass/fail reporting, making it possible to isolate and rerun individual scenarios (UART echo, TRNG read, boot sequence) without modifying the simulator invocation.

**Waveform inspection.** Verilator captures the full pipeline state into a `.fst` dump at each simulation run. The Signal Search Tree in GTKWave allows manual inspection of any internal signal—APB3 handshakes, PTRNG conditioner output, state machine registers—providing visibility that UART logging alone cannot offer.

# Chapter 5

# Implementation

This chapter presents the concrete implementation of the Taurus SoC, a RISC-V system-on-chip integrating the OpenTRNG PTRNG as an APB3 peripheral. Following the parallel structure, we detail: Nix Flake setup (section 5.1), RTL generation with SpinalHDL (section 5.2), TRNG integration (section 5.3), and testbench implementation with end-to-end test sequence (section 5.4).

## 5.1 Build Setup

This section describes the Nix Flakes implementation for reproducible RTL generation. The build system addresses three challenges: cross-compilation (firmware), JVM dependency management (SpinalHDL/Gradle), and BlackBox integration (VexRiscv, OpenTRNG).

### 5.1.1 Flake Infrastructure

The flake (`flake/flake.nix`) declares three inputs:

- **nixpkgs** (unstable channel): provides compilers, build tools, and standard libraries
- **vexriscv** (`github:SpinalHDL/VexRiscv`, `flake = false`): CPU core as SpinalHDL BlackBox
- **opentrng** (`github:opentrng/ptrng/port_verilog`, `flake = false`): TRNG peripheral Verilog sources

The `flake = false` attribute treats upstream repositories as raw source trees (since they lack `flake.nix`). All inputs are pinned in `flake.lock` with `rev` (Git commit) and `narHash` (content verification). The flake inputs resolve to immutable

Nix store paths at evaluation time; these paths are forwarded to both build derivations and the development shell, as arguments of the Nix files:

```
export VEXRISCV_SRC="${vexriscv}"
export OPENTRNG_SRC="${opentrng}"
```

Outputs are organized in three categories:

- **packages**: cross-compiled RISC-V firmware; Verilog RTL and VHDL RTL, each in a plain variant and a RAM-initialized variant with firmware embedded. Built with `nix build .#name`.
- **checks**: CI/CD validation—`nix flake check` builds all package outputs in parallel; any build failure returns a non-zero exit. The testbench already emits `results.xml` in JUnit format, making it straightforward to add a simulation check suitable for git commit hooks or a CI/CD pipeline.
- **devShells**: development environment with Gradle, JDK, Verilator, and RISC-V GCC in `$PATH`. Activated with `nix develop`, or automatically via direnv if a `.envrc` file invoking `use flake` is present in the directory.

### 5.1.2 Building Firmware

Firmware targets RISC-V 32-bit bare-metal (RV32I ISA). As described in section 2.1.5, `pkgs.pkgsCross.<target>` provides a pre-configured cross package set without altering the native `pkgs`; here `pkgsCross.riscv32-embedded` provides GCC cross-compiler, binutils, and newlib (C library for bare-metal).

The Makefile defines per-tool variables (`RISCV_CC`, `RISCV_OBJCOPY`, `RISCV_OBJDUMP`, `RISCV_SIZE`) with `riscv32-none-elf-*` defaults. Because `riscv-toolchain.gcc` and `riscv-toolchain.binutils` are listed in `nativeBuildInputs`, Nix automatically adds their `bin/` directories to `$PATH` during the build.

The build produces `firmware.hex` (Intel HEX for memory initialization), `firmware.elf` (with symbols for debugging), and `firmware.bin/.asm/.map`.

### 5.1.3 Generating RTL

**Gradle Dependency Management** SpinalHDL uses Gradle, which downloads Maven dependencies at build time. Nix's sandbox disables network access, requiring a Fixed-Output Derivation (FOD) with `gradle.fetchDeps` [11].

As described in the Nixpkgs manual, `gradle.fetchDeps` uses a MITM proxy that intercepts HTTP/HTTPS requests during Gradle builds, recording downloaded

files and their SHA-256 hashes in `deps.json`. On subsequent builds, Nix verifies each dependency's hash against the lock file.

The RTL derivation declares:

```
mitmCache = pkgs.gradle.fetchDeps {
  pkg = finalAttrs.finalPackage;
  data = ../../step1-rtl/deps.json;
};
```

The `deps.json` file records all Maven dependencies (SpinalHDL, Scala compiler, transitive dependencies), each with SHA-256 hash. To update dependencies after changing `build.gradle`:

```
$(nix-build -A rtl-verilog.mitmCache.updateScript)
```

**BlackBox Integration** SpinalHDL's BlackBox mechanism wraps external Verilog modules; the source paths are forwarded from flake inputs as described in Section 5.1.1.

OpenTRNG requires compile-time configuration via `settings.v`. A dedicated derivation (`opentrng-with-settings.nix`) generates this file from a template and packages it with OpenTRNG sources. The RTL generation derivation uses this preprocessed version.

**Multi-Stage Dependency** Two RTL variants (`genVerilogWithRamInit`, `genVhdlWithRamIni` embed firmware in the output. SpinalHDL reads `firmware.hex` and generates `.bin` files for memory initialization.

The derivation creates a symbolic link to match SpinalHDL's expected path:

```
mkdir -p ../step2-simulation/firmware/build
ln -s ${firmware}/firmware.hex
↪  ../step2-simulation/firmware/build/firmware.hex
```

Nix ensures firmware builds before RTL generation, providing the output path automatically.

**RTL Variants** The derivation is parameterized by `gradleBuildTask`, producing four outputs:

- `genVerilog`: Verilog without RAM initialization

- `genVerilogWithRamInit`: Verilog + `.bin` files (firmware embedded)
- `genVhdl`: VHDL without RAM initialization
- `genVhdlWithRamInit`: VHDL + `.bin` files

The Gradle build script defines these tasks; the `installPhase` adapts based on the task.

## 5.2 RTL Generation

This section describes the RTL generation pipeline for the Taurus SoC. It covers the VexRiscv CPU configuration, the SoC peripheral and memory map composition, and the APB3 integration point for the TRNG peripheral.

Figure 5.1 shows the complete path from CPU to peripherals:

- **CPU** (`iBus` + `dBus`) → **MuraxMasterArbiter**: merges instruction and data buses into a single `PipelinedMemoryBus`
- **MuraxPipelinedMemoryBusDecoder**: routes by address prefix—`0x8...` to RAM, `0xF...` to the APB3 bridge
- **PipelinedMemoryBusToApbBridge**: converts the internal bus protocol to APB3 (PSEL/PENABLE/PREADY/PADDR/PWDATA/PRDATA)
- **Apb3Decoder**: selects one of four peripherals based on address bits `[19:16]`: GPIO (`0xF0000000`), UART (`0xF0010000`), Timer (`0xF0020000`), TRNG (`0xF0030000`)

### 5.2.1 VexRiscv Configuration

This thesis implements a minimal RISC-V System-on-Chip for TRNG peripheral verification. The SoC is named **Taurus**—a reference to the symbol of Turin, the bull (*toro* in Italian, *taurus* in Latin). As is typical in engineering, creativity in naming is not our strongest suit.

The design is based on **Murax**, a minimal reference SoC provided as a demo in the VexRiscv repository [29]. Murax bundles a working VexRiscv CPU with RAM, UART, GPIO, and timer peripherals, providing a proven foundation to build on. This thesis extends Murax by adding the TRNG peripheral, keeping the focus on peripheral integration and verification rather than SoC architecture.

The configuration is defined in `Taurus.scala` using SpinalHDL's plugin system.

### Plugin Selection

The VexRiscv plugin system allows fine-grained ISA and microarchitecture configuration. Taurus runs RV32I without extensions at 12 MHz in M-mode only. The configuration (`Taurus.scala`) selects:

- **IBusSimplePlugin**: No-cache instruction fetch, boot from `0x80000000`, no branch prediction
- **DBusSimplePlugin**: No-cache load/store, no address-misaligned or access-fault traps
- **CsrPlugin**: Minimal CSR set (`CsrPluginConfig.smallest`), M-mode only, `mtvec = 0x80000020`
- **IntAluPlugin**: Integer arithmetic and logical operations
- **HazardSimplePlugin**: Hazard resolution via interlocking (stalls); no forwarding paths, smaller area
- **BranchPlugin**: Branch resolution in Execute stage (`earlyBranch = false`)
- **DebugPlugin**: JTAG support, software breakpoints only, separate clock domain

## 5.2.2 Bus Structure and Memory Map

The Taurus SoC uses a hierarchical bus architecture connecting the VexRiscv CPU to on-chip RAM and memory-mapped peripherals.

### Bus Architecture

Figure 5.1 shows the complete hierarchy. The four bus components are:

- **MuraxMasterArbiter**: Merges `iBus` (instruction fetch) and `dBus` (load/store) into a single `PipelinedMemoryBus`; `dBus` has priority on conflict.
- **MuraxPipelinedMemoryBusDecoder**: Routes by address prefix—`0x8...` to RAM, `0xF...` to the APB3 bridge.
- **PipelinedMemoryBusToApbBridge**: Converts `PipelinedMemoryBus` handshake to APB3 PSEL/PENABLE/PREADY timing (`Apb3Config(addressWidth=20, dataWidth=32)`).
- **Apb3Decoder**: Generates per-peripheral PSEL signals from address bits `[19:16]`, selecting GPIO (`0x0`), UART (`0x1`), Timer (`0x2`), or TRNG (`0x3`).

#### Memory Map

The complete memory map is shown in Table 5.1.

| Address Range | Size | Function |
|---|---|---|
| 0x80000000–0x80001FFF | 8 KB | On-chip RAM (code + data) |
| 0xF0000000–0xF0000FFF | 4 KB | GPIO (32 pins, read/write/direction) |
| 0xF0010000–0xF0010FFF | 4 KB | UART (TX/RX FIFOs, status, baud config) |
| 0xF0020000–0xF0020FFF | 4 KB | Timer (prescaler, compare, interrupt) |
| 0xF0030000–0xF0030FFF | 4 KB | TRNG (control, status, data output) |

**Table 5.1:** Taurus SoC memory map

**RAM allocation.** The linker script places `.text` at `0x80000000` (reset vector), `.data`/`.bss` after code, and the stack at the top of RAM (`0x80002000`), growing downward.

### 5.2.3 APB3 Peripheral Integration

The Taurus SoC integrates four memory-mapped peripherals on the APB3 bus.

#### Apb3SlaveFactory Pattern

GPIO, UART, and Timer are SpinalHDL library components that handle APB3 protocol timing internally. The TRNG peripheral (`TrngCoreWithPtrng`) is the new component written for this thesis, and directly instantiates `Apb3SlaveFactory` to define its register map. Peripheral logic is expressed as declarative register bindings (`readAndWrite`, `read`), with address decoding, `PRDATA` muxing, and `PREADY` assertion generated automatically.

#### Peripheral Instances

GPIO, UART, and Timer are standard library components from the Murax/SpinalHDL ecosystem, included as-is to provide a functional SoC for firmware testing. The TRNG is the new peripheral contributed by this thesis:

- **GPIO** (`Apb3Gpio`, `0xF0000000`): 32 bidirectional pins.
- **UART** (`Apb3UartCtrl`, `0xF0010000`): 115200 baud, 8N1, 16-byte TX/RX FIFOs. Used by firmware to output test results.
- **Timer** (`MuraxApb3Timer`, `0xF0020000`): 32-bit counter with prescaler and compare-interrupt.

- **TRNG** (`TrngCoreWithPtrng`, `0xF0030000`): OpenTRNG PTRNG wrapped as APB3 peripheral—described in detail in Section 5.3 (full source in Appendix section A.2).

## 5.3 TRNG Integration

This section describes the integration of OpenTRNG's PTRNG IP core (Related Work section 3.2) into the Taurus SoC, directly addressing the TRNG gap identified in VE-HEP (Related Work subsection 3.1.6). It covers the full stack from SpinalHDL BlackBox wrapping down to the bare-metal firmware driver.

Figure 5.2 shows the complete integration stack: the SpinalHDL layers (left) wrap the external OpenTRNG Verilog IP (right), from the APB3 bus interface down to the `ptrng.v` pipeline.

### 5.3.1 OpenTRNG BlackBox Integration

The OpenTRNG project provides a Verilog implementation of the PTRNG core with a bare-metal interface (no standard bus protocol). This integration uses the `port_verilog` branch of OpenTRNG, which exposes the HDL in Verilog rather than VHDL. Integrating this IP into the SpinalHDL-based Taurus SoC involves three layers, described top-down: the APB3 peripheral wrapper (`TrngCoreWithPtrng`, Section 5.2.3) interfaces the SoC bus; below it, a simplified convenience wrapper reduces the PTRNG's 15 inputs to four signals; and at the base, a BlackBox provides Verilog interoperability.

#### PTRNG Verilog IP Interface

The OpenTRNG PTRNG module (`ptrng.v`) exposes 15 input ports (3 basic + 12 configuration) and 7 output ports; the abbreviated interface below shows the key signals:

```
module ptrng (
  input         clk, reset, clear,

  // Configuration inputs (6 key parameters shown;
  // also: freqcount_{en,select[4:0],start},
  // onlinetest_{clear,average[15:0],deviation[15:0]})
  input  [31:0] ring_en,              // Ring oscillator enable mask
  input  [31:0] freqdivider_value,    // Sampling clock divider
  input         freqdivider_en,       // Enable frequency divider
  input  [31:0] alarm_threshold,      // Health test threshold
```

```
  input         conditioning,         // Von Neumann conditioner enable
  input         nopacking,            // Output bit packing control

  // Output ports (key signals shown;
  // also: onlinetest_valid, freqcount_{done,overflow,value[31:0]})
  output [31:0] data,                 // Random data words
  output        valid,                // Data ready strobe
  output        alarm_detected        // Total failure alarm
);
```

The module does not implement any standard bus interface, requiring a custom peripheral wrapper for SoC integration. Its internal pipeline (data path, health monitors) is described in Related Work subsection 3.2.2.

**Simulation entropy source.** In simulation, the `digitalnoise.v` module (ring oscillator chains and digitalizer logic) is replaced with `digitalnoise_stub.v`. The stub matches the `digitalnoise.v` interface port-for-port, so the PTRNG's internal hierarchy is preserved and only the leaf module is swapped at simulation compile time. It produces a deterministic pseudo-random stream—for simulation purposes only—giving the downstream pipeline more realistic input than a direct Cocotb-injected constant, but direct value injection from the testbench remains possible.

### Simplified Wrapper

The `PtrngSimple` wrapper reduces the PTRNG's 15 inputs to four signals (`enable`, `data`, `valid`, `alarm`), simplifying integration at call sites (full listing in Appendix section A.1).

The wrapper serves two roles. First, it ties control signals that must remain deasserted in this integration to safe values: `clear`, frequency counter start/select, and `onlinetest_clear` are all held constant, preventing accidental misconfiguration. Second, it pre-configures the operating parameters: all ring oscillators enabled, sampling rate and conditioning active. `nopacking` is set to bypass the LSB-packing stage: the PTRNG passes the full 32-bit conditioned word directly to `data` instead of accumulating 32 individual LSBs from successive conditioned pairs. This increases output throughput and was chosen to exercise the full downstream register path during the verification stage.

The `PtrngSimple` component is instantiated by the `TrngCoreWithPtrng` APB3 peripheral (Section 5.2.3), which adds the memory-mapped register interface and bus protocol handling.

#### BlackBox Wrapper

The `PtrngBlackBox` class wraps the Verilog module using SpinalHDL's BlackBox pattern, providing the Verilog interoperability layer. The BlackBox must declare every port of `ptrng.v` with matching names, directions, and bit widths (full listing in Appendix section A.1).

The key mechanism is `addRTLPath()`, which tells SpinalHDL where to find the external Verilog files to include during simulation and synthesis. The path is resolved via the `OPENTRNG_SRC` environment variable, exported by both the Nix development shell and the build derivation.

### 5.3.2 Firmware Driver

The driver maps the peripheral register layout to a C struct at base address `0xF0030000` (matching the SoC memory map in Section 5.2.3; full listing in Appendix section A.3). Three design choices follow the pattern described in Background subsection 2.3.4:

- **Volatile register fields**: each field is declared `volatile`, preventing the compiler from caching register values across accesses.
- **Caller-managed polling**: the read accessor returns `DATA` directly; callers check the ready flag before reading, keeping the polling strategy out of the low-level accessor.
- **Read-clear semantics**: reading `DATA` automatically clears the `data_ready` flag (Section 5.2.3), preventing accidental re-reads of the same value.

## 5.4 Testbench Implementation

### 5.4.1 Cocotb Testbench

The Cocotb testbench provides end-to-end coverage of the OpenTRNG integration: only `digitalnoise` is replaced by the LFSR stub (Methodology subsection 4.3.2), leaving `ptrng.v`, `conditioner.v`, `alarm.v`, `onlinetest.v`, and `bitpacker.v` as real OpenTRNG Verilog. The stub generates an autonomous pseudo-random bit stream; Python-side value injection remains available on top for targeted test cases. Each test case boots the complete SoC and observes behaviour through the UART TX pin; Verilator captures the full pipeline state in `taurus_sim.fst` at each run.

### 5.4.2 Test Sequence

The integration test validates the full register interface by running the firmware test against the simulated SoC and inspecting the resulting `taurus_sim.fst` waveform in GTKWave. The Signal Search Tree exposes every internal signal, allowing each step of the CPU-to-peripheral path to be confirmed:

1. **CTRL write**: firmware writes `0x1` to `CTRL` (address `0x00`); the APB3 signals `PSEL`, `PENABLE`, `PWRITE`, and `PWDATA` are visible in the waveform, confirming the write transaction reaches the peripheral and `enable` transitions to 1.

2. **PTRNG pipeline activation**: with `enable` asserted, the LFSR stub pulses `digitalnoise.valid` every 16 clock cycles. Each pulse propagates through the real OpenTRNG stages with one-cycle latency per stage: `conditioned_valid` (conditioner output), `intermediate_random_valid` (internal pipeline register), and finally `ptrng_wrapper.opentrng.valid` (PTRNG output)—a total of approximately 3–4 clock cycles from stub output to `dataReg` capture. At that point `dataReady` asserts.

3. **STATUS polling**: firmware reads `STATUS` (address `0x04`) in a loop; repeated APB read transactions with `PWRITE=0` are visible until `PRDATA[0]` reflects `dataReady=1`.

4. **DATA retrieval**: firmware reads `DATA` (address `0x08`); `PRDATA` carries the value of `dataReg`. Consecutive reads return distinct values, confirming the pipeline continues producing output.

5. **Read-clear handshake**: each DATA read automatically clears `dataReady`—visible as a brief pulse to 0 on `statusReg[0]`—before the next valid word reasserts it.

6. **Alarm deasserted**: `alarm_detected` remains low throughout the test, confirming that the online test and alarm health monitors execute against the LFSR input without raising false positives.

The UART output provides a second, independent confirmation channel alongside the waveform.

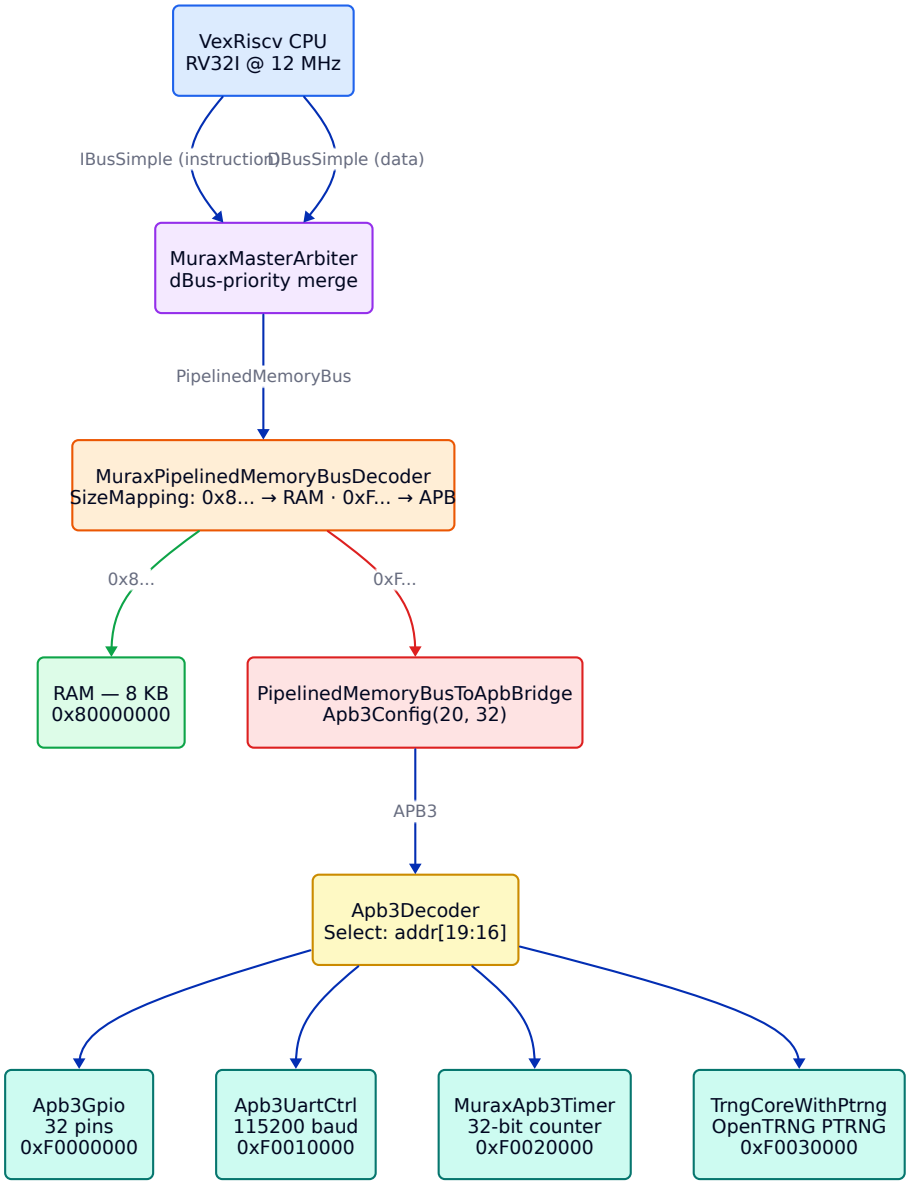


**Figure 5.1:** Taurus SoC bus hierarchy and address decoding. The diagram shows the complete path from CPU instruction/data buses through arbitration, address decoding, and protocol conversion to APB3 peripherals. VexRiscv's proprietary `IBusSimple` and `DBusSimple` interfaces merge at the arbiter, then route through a decoder (RAM vs peripherals based on address prefix) and a bridge that translates `PipelinedMemoryBus` protocol to industry-standard APB3. The `Apb3Decoder` generates peripheral-specific `PSEL` signals based on upper address bits. Key signals are annotated on bus interfaces.

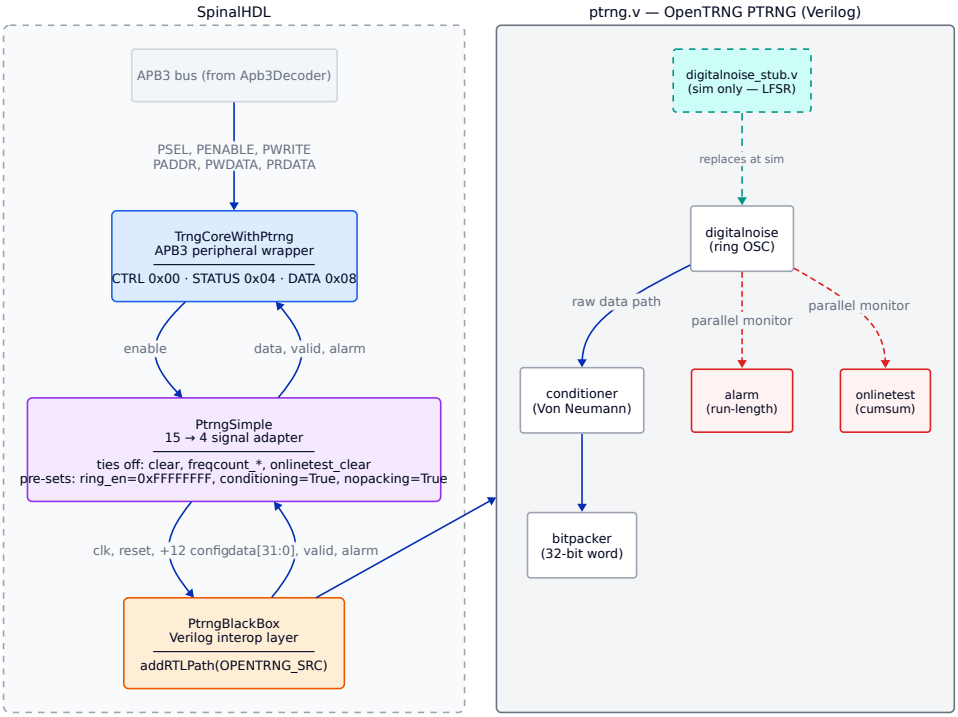


**Figure 5.2:** TRNG integration stack. Three SpinalHDL layers (dashed box) wrap the external OpenTRNG Verilog IP: `TrngCoreWithPtrng` exposes a memory-mapped APB3 interface with three registers; `PtrngSimple` reduces the PTRNG's 15-port interface to four signals by pre-setting configuration inputs; `PtrngBlackBox` provides the SpinalHDL/Verilog boundary via `addRTLPath()`. The `ptrng.v` data path is: ring oscillators → conditioner → bit packer → `data`/`valid`. `alarm` and `onlinetest` run as parallel health monitors on the raw entropy stream and do not gate the data path. In simulation, `digitalnoise.v` is replaced by an LFSR stub at compile time.

# Chapter 6

# Testing and Results

# Chapter 7

# Conclusions and Future Work

This thesis addressed two gaps in the open-source ASIC landscape. The Taurus SoC integrates OpenTRNG's PTRNG as an APB3 peripheral into a VexRiscv RISC-V system, providing the first reference design for that IP with a standard bus interface and a C firmware driver. End-to-end functional verification was conducted via Cocotb driven by bare-metal firmware, confirmed through UART output and GTKWave waveform inspection. Every build stage—firmware cross-compilation, RTL generation, and simulation—is expressed as a Nix derivation, establishing a hermetic, fully reproducible pipeline.

The work covers ASIC stages 1–2 (RTL and simulation). Natural next steps are:

- **Formal verification.** SymbiYosys (`sby`) [58] could exhaustively verify APB3 protocol compliance and read-clear register semantics of `TrngCoreWithPtrng`—following the pattern already used by VE-HEP for the VexRiscv CPU.
- **Physical implementation.** The natural next step is FPGA prototyping, which would replace the LFSR stub with real ring oscillators for the first time. LibreLane [59] (successor to OpenLane) then provides the complete RTL-to-GDSII path should an ASIC flow be pursued; LibreLane uses Nix internally and was packaged for `nixpkgs` during the writing of this thesis.

# Appendix A

# TRNG Integration Code Listings

The listings below are simplified versions of the actual source files, edited for clarity. Comments, diagnostic fields, and non-essential configuration have been omitted. The full sources are available in the project repository.

## A.1 SpinalHDL BlackBox Wrappers

### PtrngBlackBox

```
class PtrngBlackBox(
  regWidth: Int = 32,
  randWidth: Int = 32
) extends BlackBox {

  addGeneric("REG_WIDTH", regWidth)
  addGeneric("RAND_WIDTH", randWidth)

  val io = new Bundle {
    val clk = in Bool()
    val reset = in Bool()
    val clear = in Bool()
    val ring_en = in Bits(regWidth bits)
    val freqdivider_value = in Bits(regWidth bits)
    // ... 10 further inputs (freqdivider_en, alarm_threshold,
    //     onlinetest_{clear,average,deviation}, conditioning,
    //     nopacking, freqcount_{en,select,start}) ...
```

```
    val data = out Bits(randWidth bits)
    val valid = out Bool()
    val alarm_detected = out Bool()
  }.setName("")  // Remove io_ prefix to match Verilog port names

  addRTLPath(s"$opentrngSrc/hardware/hdl/verilog/ptrng/ptrng.v")

  ↪  addRTLPath(s"$opentrngSrc/hardware/hdl/verilog/ptrng/digitalnoise.v")
  addRTLPath(s"$opentrngSrc/hardware/hdl/verilog/ptrng/conditioner.v")
  // ... include all Verilog source files ...

  setDefinitionName("ptrng")
}
```

## PtrngSimple

```
class PtrngSimple extends Component {
  val io = new Bundle {
    val enable = in Bool()
    val data = out Bits(32 bits)
    val valid = out Bool()
    val alarm = out Bool()
  }

  val opentrng = new PtrngBlackBox(regWidth = 32, randWidth = 32)

  opentrng.io.clk := clockDomain.readClockWire
  opentrng.io.reset := clockDomain.readResetWire

  // Safe defaults for PTRNG configuration
  opentrng.io.ring_en := Mux(io.enable, B"32'hFFFFFFFF",
  ↪  B"32'h00000000")
  opentrng.io.freqdivider_value := B(100, 32 bits)
  opentrng.io.freqdivider_en := True
  opentrng.io.conditioning := True
  opentrng.io.nopacking := True
  opentrng.io.alarm_threshold := B(1000, 32 bits)
  // ... configure remaining parameters ...

  io.data := opentrng.io.data
```

```
  io.valid := opentrng.io.valid
  io.alarm := opentrng.io.alarm_detected
}
```

## A.2 APB3 Peripheral Wrapper

`TrngCoreWithPtrng` instantiates `PtrngSimple` and exposes the CTRL, STATUS, and DATA registers on an APB3 slave interface. The read-clear logic on DATA is the key hardware handshake: a read of address `0x08` deasserts `dataReady` in the same cycle.

```
class TrngCoreWithPtrng extends Component {
  val io = new Bundle {
    val apb = slave(Apb3(Apb3Config(addressWidth = 4, dataWidth =
    ↪  32)))
  }

  val controlReg = Reg(UInt(32 bits)) init(0)
  val enable     = controlReg(0)

  val ptrng = new PtrngSimple()
  ptrng.io.enable := enable

  val dataReg   = Reg(UInt(32 bits)) init(0)
  val dataReady = Reg(Bool())        init(False)

  when(ptrng.io.valid) {
    dataReg   := ptrng.io.data.asUInt
    dataReady := True
  }

  val statusReg = Cat(B(0, 30 bits), ptrng.io.alarm.asBits,
                      dataReady.asBits).asUInt

  val apbCtrl = Apb3SlaveFactory(io.apb)
  apbCtrl.readAndWrite(controlReg, address = 0x00)
  apbCtrl.read(statusReg,          address = 0x04)
  apbCtrl.read(dataReg,            address = 0x08)

  // Read-clear: reading DATA clears dataReady
```

```
  when(io.apb.PSEL(0) && io.apb.PENABLE && !io.apb.PWRITE
                       && io.apb.PADDR === 0x08) {
    dataReady := False
  }
}
```

## A.3 Firmware Driver

```
// trng.h
typedef struct {
    volatile uint32_t CTRL;   // 0x00 RW: bit 0 = enable
    volatile uint32_t STATUS; // 0x04 RO: bit 0 = data_ready, bit 1 =
    ↪ alarm
    volatile uint32_t DATA;   // 0x08 RO: random word (read clears
    ↪ data_ready)
} Trng_Reg;

#define TRNG ((Trng_Reg*)(0xF0030000))

static inline uint32_t trng_read(void) { return TRNG->DATA; }

// trng.c
uint8_t trng_ready(void) { return TRNG->STATUS & 0x1; }
```

# Bibliography


[1] Eelco Dolstra. «The Purely Functional Software Deployment Model». PhD thesis. Utrecht University, 2006. URL: https://edolstra.github.io/pubs/phd-thesis.pdf (cit. on pp. 5, 53).

[2] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. «Nix: A Safe and Policy-Free System for Software Deployment». In: *Proceedings of the 18th USENIX Large Installation System Administration Conference (LISA '04)*. USENIX Association, 2004. URL: https://edolstra.github.io/pubs/nspfssd-lisa2004-final.pdf (cit. on p. 7).

[3] NixOS Wiki contributors. *Nix*. Disambiguation page for the overloaded term "Nix". URL: https://wiki.nixos.org/wiki/Nix (cit. on p. 8).

[4] Repology. *Repository: nixpkgs unstable*. URL: https://repology.org/repository/nix_unstable (cit. on pp. 8, 53).

[5] Nix Documentation Team. *Dependency management*. URL: https://nix.dev/guides/recipes/dependency-management.html (cit. on pp. 9, 11).

[6] Andreas Rammhold. *npins: Simple and convenient dependency pinning for Nix*. URL: https://github.com/andir/npins (cit. on p. 11).

[7] Nicolas Mattia. *Niv: Easy dependency management for Nix projects*. URL: https://github.com/nmattia/niv (cit. on p. 11).

[8] Nix Development Team. *Nix Reference Manual: Derivations*. URL: https://nix.dev/manual/nix/2.25/language/derivations (cit. on p. 13).

[9] NixOS Wiki contributors. *NixOS Wiki: Derivations*. URL: https://wiki.nixos.org/wiki/Derivations (cit. on p. 13).

[10] Determinate Systems. *Zero to Nix: Derivations*. URL: https://zero-to-nix.com/concepts/derivations/ (cit. on p. 13).

[11] Nix Community. *Nixpkgs Reference Manual*. URL: https://nixos.org/manual/nixpkgs/stable/ (cit. on pp. 15, 18–20, 53, 64).

[12] NixOS Community. *NixOS Wiki: Development environment with nix-shell*. URL: https://wiki.nixos.org/wiki/Development_environment_with_nix-shell (cit. on p. 17).

[13] Nix Community. *Nixpkgs Reference Manual: Cross-compilation*. URL: https://nixos.org/manual/nixpkgs/stable/#chap-cross (cit. on p. 19).

[14] NixOS Wiki contributors. *Cross Compiling*. URL: https://wiki.nixos.org/wiki/Cross_Compiling (cit. on p. 19).

[15] Nix Documentation Team. *Cross compilation*. URL: https://nix.dev/tutorials/cross-compilation.html (cit. on p. 19).

[16] Determinate Systems. *Nix Flakes Explained*. 2024. URL: https://determinate.systems/blog/nix-flakes-explained/ (cit. on pp. 21, 22).

[17] NixOS Wiki contributors. *Nix Flakes*. URL: https://wiki.nixos.org/wiki/Flakes (cit. on p. 21).

[18] Ryan Yin. *NixOS & Flakes Book*. Unofficial but widely referenced practical guide to NixOS and Nix flakes. URL: https://nixos-and-flakes.thiscute.world/ (cit. on p. 21).

[19] Nix Documentation Team. *Flakes*. URL: https://nix.dev/concepts/flakes.html (cit. on pp. 22, 26, 27).

[20] Graham Christensen. *Experimental does not mean unstable*. Determinate Systems. Sept. 2023. URL: https://determinate.systems/posts/experimental-does-not-mean-unstable (cit. on p. 22).

[21] Jade Lovelace. *Flakes aren't real and cannot hurt you: a guide to using Nix flakes the non-flake way*. Jan. 2024. URL: https://jade.fyi/blog/flakes-arent-real/ (cit. on pp. 24, 26, 27).

[22] NixOS Contributors. ***flake.nix*** — *NixOS/nixpkgs*. URL: https://github.com/NixOS/nixpkgs/blob/nixos-25.11/flake.nix (cit. on p. 25).

[23] *Nix Reference Manual: New CLI Commands*. URL: https://nix.dev/manual/nix/stable/command-ref/new-cli/nix (cit. on p. 25).

[24] IEEE. *IEEE Standard for Verilog Hardware Description Language*. IEEE, 2006. DOI: 10.1109/IEEESTD.2006.99495 (cit. on p. 28).

[25] IEEE. *IEEE Standard VHDL Language Reference Manual*. IEEE, 2009. DOI: 10.1109/IEEESTD.2009.4772740 (cit. on p. 28).

[26] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. «Chisel: Constructing Hardware in a Scala Embedded Language». In: *Proceedings of the 49th Annual Design Automation Conference (DAC '12)*. San Francisco, California: Association for Computing Machinery, 2012, pp. 1216–1225. ISBN: 9781450311991. DOI: 10.1145/2228360.2228584 (cit. on p. 29).

[27] SpinalHDL Community. *SpinalHDL Documentation*. 2024. URL: https://spinalhdl.github.io/SpinalDoc-RTD/ (cit. on pp. 29, 47).

[28] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, 2017. ISBN: 978-0999249109 (cit. on pp. 29, 56).

[29] Charles Papon. *VexRiscv: A FPGA Friendly RISC-V CPU Implementation*. 2024. URL: https://github.com/SpinalHDL/VexRiscv (cit. on pp. 30, 47, 56, 57, 60, 61, 66).

[30] Tim Henkes et al. «Evaluating an Open-Source Hardware Approach from HDL to GDS for a Security Chip Design — a Review of the Final Stage of Project HEP». In: *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2024, pp. 1–6. DOI: 10.23919/DATE58400.2024.10546500 (cit. on pp. 31, 48, 56, 57).

[31] ARM Limited. *AMBA AXI and ACE Protocol Specification*. Version E. 2013. URL: https://developer.arm.com/documentation/ihi0022/ (cit. on p. 31).

[32] ARM Limited. *AMBA APB Protocol Specification*. Version 2.0. 2010. URL: https://developer.arm.com/documentation/ihi0024/ (cit. on p. 32).

[33] Wikipedia contributors. *Memory-mapped I/O and port-mapped I/O*. Describes memory-mapped I/O (MMIO) and port-mapped I/O (PMIO) approaches for peripheral communication. 2026. URL: https://en.wikipedia.org/wiki/Memory-mapped_I/O_and_port-mapped_I/O (cit. on p. 32).

[34] Wikipedia contributors. *Hardware security module*. 2026. URL: https://en.wikipedia.org/wiki/Hardware_security_module (cit. on p. 33).

[35] National Institute of Standards and Technology. *Security Requirements for Cryptographic Modules*. Federal Information Processing Standards Publication FIPS 140-3. Supersedes FIPS 140-2. U.S. Department of Commerce, Mar. 2019. URL: https://csrc.nist.gov/pubs/fips/140-3/final (cit. on p. 33).

[36] Debian Security Team. *Debian OpenSSL Predictable PRNG (CVE-2008-0166)*. 2008. URL: https://www.debian.org/security/2008/dsa-1571 (cit. on p. 35).

[37] Dan Goodin. *Android bug batters Bitcoin wallets*. The Register. 2013. URL: https://www.theregister.co.uk/2013/08/12/android_bug_batters_bitcoin_wallets/ (cit. on p. 35).

[38] Bushing, Marcan, Segher, and Sven. *Console Hacking 2010: PS3 Epic Fail*. 27th Chaos Communication Congress (27C3). Chaos Computer Club, 2010. URL: https://media.ccc.de/v/27c3-4087-en-console_hacking_2010 (cit. on p. 35).

[39] Wikipedia contributors. *Hardware random number generator*. 2026. URL: https://en.wikipedia.org/wiki/Hardware_random_number_generator (cit. on p. 36).

[40] Elaine Barker and John Kelsey. *Recommendation for the Entropy Sources Used for Random Bit Generation*. NIST Special Publication 800-90B. National Institute of Standards and Technology, Jan. 2018. DOI: 10.6028/NIST.SP.800-90B. URL: https://doi.org/10.6028/NIST.SP.800-90B (cit. on p. 39).

[41] Tristan Gingold and contributors. *GHDL: Open Source Analyzer, Compiler, Simulator and Synthesizer for VHDL*. 2024. URL: https://github.com/ghdl/ghdl (cit. on p. 43).

[42] Wilson Snyder, Wim Lavrijsen, et al. *Verilator Manual*. 2024. URL: https://verilator.org/guide/latest/ (cit. on p. 44).

[43] Tony Bybell. *GTKWave: Electronic Waveform Viewer*. 2024. URL: https://gtkwave.sourceforge.net/ (cit. on p. 44).

[44] HEP Alliance. *VE-HEP: Hardware Security Module - HEP Validation Environment*. 2024. URL: https://github.com/HEP-Alliance/VE-HEP-HW-SW (cit. on pp. 47, 52).

[45] Florian Pebay-Peyroula, Licinius-Pompiliu Benea, Mikael Carmona, and Romain Wacquez. «OpenTRNG: an Open-Source Initiative for Ring-Oscillator Based TRNGs». In: *2024 IEEE International Conference on Design, Test and Technology of Integrated Systems (DTTIS)*. 2024, pp. 1–6. DOI: 10.1109/DTTIS62212.2024.10780212 (cit. on p. 49).

[46] CEA-Leti and Contributors. *OpenTRNG: Open-Source Framework for True Random Number Generators*. 2024. URL: https://github.com/opentrng/ptrng (cit. on p. 49).

[47] lowRISC CIC. *OpenTitan: Open Source Silicon Root of Trust*. 2026. URL: https://opentitan.org/ (cit. on p. 52).

[48] GNU Guix Contributors. *GNU Guix Reference Manual*. URL: https://guix.gnu.org/manual/en/guix.html (cit. on p. 53).

[49] Numtide. *Nix, Docker, or both?* 2023. URL: https://numtide.com/blog/nix-docker-or-both (cit. on p. 53).

[50] Bazel Team. *Bazel: Fast, Correct Builds at Scale*. URL: https://bazel.build/ (cit. on p. 54).

[51] Ben Radford. *Bazel and Nix: A Migration Experience*. Case study: migrating a C++ game (15,000 lines) from Makefiles to Bazel with Nix for dependency management. Discusses complementarity: "Nix as a package manager and Bazel as a build system". Dec. 2022. URL: https://www.tweag.io/blog/2022-12-15-bazel-nix-migration-experience/ (cit. on p. 55).

[52] Mathieu Boespflug. *Bazel and Nix: A Migration Story*. Technical comparison of Bazel and Nix approaches to hermetic builds and content-addressable storage. Mar. 2018. URL: https://www.tweag.io/blog/2018-03-15-bazel-nix/ (cit. on p. 55).

[53] Tweag. *Nix + Bazel: Fully Reproducible, Incremental Builds*. Official landing page for rules_nixpkgs. Compares Bazel sandbox (weak, accesses /) vs Nix sandbox (strict). Positions Nix+Bazel hybrid approach. URL: https://nix-bazel.build/ (cit. on p. 55).

[54] SpinalHDL Community. *About SpinalHDL*. Introduction to SpinalHDL language features and advantages. 2024. URL: https://spinalhdl.github.io/SpinalDoc-RTD/master/SpinalHDL/Introduction/SpinalHDL.html (cit. on p. 56).

[55] lowRISC Contributors. *OpenTitan Documentation*. 2024. URL: https://opentitan.org/book/ (cit. on p. 61).

[56] Florent Kermarrec et al. *LiteX: A Python-based SoC Builder and Framework*. 2024. URL: https://github.com/enjoy-digital/litex (cit. on p. 61).

[57] cocotb Contributors. *cocotb Documentation*. 2024. URL: https://docs.cocotb.org/en/stable/ (cit. on p. 61).

[58] YosysHQ GmbH. *SymbiYosys: Front-end driver program for Yosys-based formal hardware verification flows*. 2024. URL: https://symbiyosys.readthedocs.io/ (cit. on p. 76).

[59] LibreLane Contributors. *LibreLane: Automated RTL to GDSII Flow*. Successor to OpenLane. 2026. URL: https://github.com/librelane/librelane (cit. on p. 76).