SKALP: An Intent-Preserving Hardware Description Language with Compile-Time Clock Domain Safety and GPU-Accelerated Fault Simulation

Shankar Giri V
https://github.com/girivs82/skalp
https://mikaana.com

***Abstract*—Hardware description languages face a fundamental tension: register-transfer level (RTL) languages like SystemVerilog and VHDL give designers precise control but require tedious manual specification, while high-level synthesis (HLS) tools promise abstraction but produce unpredictable results and discard designer intent during compilation. We present SKALP, a hardware description language that resolves this tension through a four-stage intermediate representation (IR) pipeline that preserves designer intent—including safety annotations, clock domain assignments, and power intent—from source through synthesis and simulation. SKALP introduces three key innovations: (1) a multi-IR compilation architecture (HIR, MIR, LIR, SIR) where each representation serves a distinct purpose and intent propagates through all stages; (2) compile-time clock domain crossing (CDC) safety enforced through the type system, making CDC violations compiler errors rather than post-synthesis lint warnings; and (3) a GPU-accelerated fault simulation backend targeting Apple Metal that achieves approximately 11 million fault-cycle simulations per second on M4 Max hardware. The language additionally provides first-class ISO 26262 functional safety support with automated FMEDA generation, Null Convention Logic (NCL) asynchronous circuit synthesis, and AIG-based logic optimization. SKALP is implemented in Rust across approximately 290,000 lines of code organized into 23 crates and includes a complete toolchain with package manager, LSP server, and native iCE40 FPGA place-and-route.**

I. Introduction

The design of digital hardware has long been characterized by a gap between abstraction and control. At one extreme, RTL languages such as SystemVerilog [1] and VHDL [2] require designers to manually specify every flip-flop, multiplexer, and wire—a laborious process that, while offering precise control over the resulting hardware, makes designs error-prone and difficult to maintain. At the other extreme, high-level synthesis (HLS) tools [3], [4] promise to raise the abstraction level by compiling algorithmic descriptions (typically in C or C++) into hardware, but they frequently produce circuits whose timing, area, and power characteristics are difficult to predict or control.

Both approaches share a more insidious problem: *intent is lost during compilation*. When a designer specifies that a signal belongs to a particular clock domain, that a module implements a safety mechanism, or that a datapath should be pipelined with a specific number of stages, this information exists only as comments or naming conventions in traditional HDLs. Synthesis tools are free to ignore, misinterpret, or discard it. The consequences of this loss of intent are severe in safety-critical applications: automotive ICs designed to ISO 26262 [5] require traceability from safety goals through gate-level implementations, but current tools force this traceability to be maintained manually through side-channel documentation.

SKALP addresses these problems through four contributions:

1) **Intent-preserving multi-IR compilation.** A four-stage intermediate representation pipeline (HIR, MIR, LIR, SIR) where each stage serves a distinct purpose—from polymorphic high-level design through cycle-accurate RTL, word-level synthesis primitives, and GPU-optimized simulation—with designer intent annotations propagating through all stages.
2) **Compile-time clock domain safety.** Clock domains are tracked as part of the type system, analogous to lifetime tracking in Rust [6]. Unsynchronized clock domain crossings are compiler errors, not post-synthesis lint warnings. The compiler can automatically insert appropriate synchronization primitives (2-FF, Gray code, pulse, handshake, or async FIFO).
3) **GPU-accelerated fault simulation.** A Simulation IR (SIR) designed from the ground up for parallel execution, enabling embarrassingly parallel fault injection on GPU hardware. Each GPU thread independently simulates the full design with one injected fault, achieving approximately 11 million fault-cycle simulations per second on Apple M4 Max.
4) **Integrated safety analysis.** ISO 26262 safety annotations are first-class language constructs that propagate through all four IRs, enabling automated Failure Mode, Effects, and Diagnostic Analysis (FMEDA) generation directly from the compiler's gate-level representation.

SKALP uses a Rust-inspired syntax with entities (modules), traits, generics, const generics, and pattern matching. It is expression-based, meaning every construct evaluates to a value, and it provides built-in verification constructs (assertions, assumptions, formal properties) alongside power intent attributes and memory configuration annotations.

The remainder of this paper is organized as follows. Section II surveys related work across HDLs, HLS tools, and

hardware-oriented functional languages. Section III describes the language design. Section IV details the four-IR compilation pipeline. Section V presents the simulation architecture. Section VI covers the safety integration flow. Section VII discusses implementation details. Section VIII provides evaluation results. Section IX discusses limitations and future work. Section X concludes.

II. Related Work

A. Traditional Hardware Description Languages

SystemVerilog [1] is the dominant HDL for ASIC and FPGA design. It provides a mature ecosystem of simulation, synthesis, and verification tools, and its SystemVerilog Assertions (SVA) enable sophisticated property-based verification. However, SystemVerilog's C-like syntax and implicit semantics (blocking vs. non-blocking assignments, sensitivity list inference, implicit wire declarations) are frequent sources of design errors. Clock domain crossings are handled through lint tools (e.g., Spyglass CDC [7]) that operate post-synthesis and produce advisory warnings rather than compiler errors. Safety annotations are entirely absent from the language.

VHDL [2] offers stronger typing than SystemVerilog and explicit signal assignment semantics, but its verbosity and limited support for modern language features (generics are constrained, no pattern matching, no algebraic data types) make it cumbersome for complex designs. Like SystemVerilog, it provides no native support for clock domain safety or functional safety annotations.

B. Modern HDLs and Hardware Construction Languages

Chisel/FIRRTL [8] introduced the concept of hardware construction in a general-purpose language (Scala). Chisel's use of Scala's type system enables powerful parameterization and code generation, and the FIRRTL intermediate representation provides a clean compilation target. However, Chisel inherits Scala's compilation times and runtime overhead, and its approach of generating Verilog as a final output means that designer intent expressed in Chisel is not preserved in the output artifacts consumed by downstream tools. FIRRTL provides a single IR with multiple levels of lowering, whereas SKALP uses four distinct IRs optimized for different purposes.

Amaranth (formerly nMigen) [9] takes a similar approach in Python, constructing hardware descriptions as Python data structures. Amaranth provides an elegant abstraction for combinational and synchronous logic and handles clock domains more explicitly than Chisel. However, being embedded in Python means that the "language" is really a library, limiting the ability to provide true compile-time guarantees about clock domains or safety properties.

Spade [10] is a hardware description language that, like SKALP, draws inspiration from Rust's type system for hardware design. Spade provides pipeline-aware types and prevents common hardware bugs through its type system. SKALP differs from Spade in its four-IR architecture, its integration of functional safety annotations, its GPU-accelerated simulation backend, and its support for NCL asynchronous circuits.

Clash [11] compiles Haskell to hardware, leveraging Haskell's strong type system and functional programming model. Clash's approach produces correct-by-construction circuits for designs that fit within its functional paradigm, but designs requiring explicit state machines or low-level timing control can be difficult to express. Clash also does not address safety annotations or GPU-accelerated simulation.

Bluespec SystemVerilog (BSV) [12] uses guarded atomic actions to describe hardware, providing a higher-level abstraction than RTL while maintaining cycle-accurate semantics. BSV's rule-based model is elegant for many designs but can be difficult to reason about when rules interact in complex ways. BSV does not provide clock domain safety through its type system or functional safety integration.

C. High-Level Synthesis

Vivado HLS (now Vitis HLS) [3] and **Catapult HLS** [4] compile C/C++ (or SystemC) to RTL. These tools can dramatically reduce design time for datapath-oriented designs, but they share common limitations: unpredictable quality of results (QoR), loss of designer intent during the C-to-RTL transformation, limited control over the resulting microarchitecture, and difficulty in verifying that the generated RTL matches the designer's expectations. Pragma-based optimization (e.g., `#pragma HLS PIPELINE II=1`) provides some control but operates at a different semantic level than the source specification. SKALP occupies a middle ground: it gives designers RTL-level control while providing higher-level abstractions (pipeline annotations, safety attributes) that are preserved through compilation rather than interpreted as optimization hints.

D. Synthesis and Verification Tools

ABC [13] provides state-of-the-art And-Inverter Graph (AIG) based logic optimization and technology mapping. SKALP's synthesis engine is inspired by ABC's approach but integrates safety-aware optimization that preserves FIT rate annotations and ISO 26262 traceability through the optimization pipeline. **Yosys** [14] provides an open-source synthesis framework with its own internal representation (RTLIL); SKALP's LIR serves a similar purpose but is designed to also feed into simulation and safety analysis, not just synthesis.

For formal verification, **Z3** [15] and SAT solvers are widely used for bounded model checking and equivalence checking. SKALP integrates SMT-based bounded model checking and combinational equivalence checking (CEC) directly into its toolchain, enabling verification of RTL-to-gate-level equivalence without external tools.

III. Language Design

A. Syntax Overview

SKALP uses a syntax inspired by Rust, with hardware-specific extensions. The fundamental design unit is the **entity**, analogous to a Verilog module or VHDL entity-architecture pair:

```
entity Counter {
    in clk: clock
    in rst: reset
    out count: nat[8]
}

impl Counter {
    signal counter: nat[8] = 0

    on(clk.rise) {
        if rst {
            counter = 0
        } else {
            counter = counter + 1
        }
    }

    count = counter
}
```

Several design decisions are visible in this example. The `entity` declaration defines the interface (ports and their types), while `impl` provides the behavior—separating interface from implementation as in Rust's trait system. The `on(clk.rise)` block is the equivalent of a SystemVerilog `always_ff @(posedge clk)` block but is syntactically scoped to make the clock domain explicit. Signals have explicit types (`nat[8]` for an 8-bit unsigned natural number) and optional initial values. Continuous assignments (e.g., `count = counter`) exist outside procedural blocks.

B. Type System

SKALP provides a rich type system designed for hardware:

- **Bit vectors:** `bit[N]` for 2-state N-bit values, `logic[N]` for simulation-compatible vectors (maps to SystemVerilog `logic`; X/Z are modeled as deterministic values rather than true 4-state simulation).
- **Numeric types:** `nat[N]` for unsigned naturals, `int[N]` for signed integers, with width inference.
- **Fixed-point:** First-class support via `fixed<I, F>` specifying integer and fractional bit widths.
- **Clock and reset:** `clock` and `reset(active_high)` / `reset(active_low)` are distinct types with domain tracking.
- **Compound types:** Structs, enums with variants (algebraic data types), tuples, and arrays.
- **Streams:** `stream<T>` for pipelined data, carrying backpressure semantics.
- **Protocols:** Named protocol types for bus interfaces (e.g., AXI4-Lite).

Width inference allows the compiler to determine signal widths from context, reducing verbosity while maintaining the explicitness that hardware designers require. The type system is parametric: entities accept generic type parameters and const generics:

```
entity FIFO<const WIDTH: nat = 8,
            const DEPTH: nat = 16> {
    in clk: clock
    in rst: reset(active_high)
    in wr_en: bit
    in wr_data: bit[WIDTH]
    out full: bit
    in rd_en: bit
    out rd_data: bit[WIDTH]
    out empty: bit
}
```

Const generics are resolved during monomorphization at the HIR level, producing fully concrete MIR modules for each instantiation—similar to Rust's monomorphization of generic functions.

C. Clock Domain Safety

Clock domains in SKALP are part of the type system. Each clock signal carries a domain identifier, and the compiler tracks which signals belong to which domain. A signal driven in an `on(clk_a.rise)` block belongs to `clk_a`'s domain; reading that signal in an `on(clk_b.rise)` block constitutes a clock domain crossing.

The CDC analysis operates at the MIR level, where clock domains, processes, and signal assignments are fully resolved. The analyzer classifies crossings into several categories:

- **Direct crossing:** A signal sampled in a different clock domain without synchronization.
- **Combinational mixing:** Signals from different domains combined through combinational logic before being registered.
- **Async reset crossing:** An asynchronous reset from one domain used in another.
- **Arithmetic mixing:** Arithmetic operations on operands from different clock domains, which can produce glitchy intermediate values.

Unsynchronized crossings are reported as compiler errors with source locations and suggested fixes. The compiler can automatically generate appropriate synchronization structures—two-flip-flop synchronizers for single-bit signals, Gray code synchronizers for multi-bit counters, pulse synchronizers for edge-triggered events, handshake synchronizers for multi-word transfers, and asynchronous FIFOs for streaming data.

D. Intent Annotations

SKALP provides a set of attributes that declare designer intent and propagate through the compilation pipeline:

Pipeline annotations declare intended pipeline structure:

```
#[pipeline(stages=5,
           target_frequency="200MHz")]
entity ImageProcessor { ... }
```

Safety annotations declare functional safety properties (detailed in Section VI):

```
#[implements(SG001::TmrVoting)]
#[safety_mechanism(type=tmr)]
entity TmrVoter {
    in a: bit[8]
```

```
    in b: bit[8]
    in c: bit[8]
    out voted: bit[8]
    #[detection_signal]
    out fault_detected: bit
}
```

Power intent attributes declare power domain assignments and control strategies:

```
#[power_domain("vdd_core")]
#[retention]
signal critical_state: bit[32]

#[isolation(clamp=0)]
out data_out: bit[16]
```

Implementation style hints guide synthesis without overriding it:

```
#[intent(style=parallel)]
signal sum: nat[32] = a + b
```

These annotations are not comments or pragmas that synthesis tools may ignore. They are part of the IR at every stage and are verified for consistency as the design is lowered.

E. Expression-Based Design

SKALP is expression-based: every construct evaluates to a value. This includes conditional expressions, which can be used directly in signal assignments:

```
voted = if ab_match { a }
        else if bc_match { b }
        else if ac_match { a }
        else { a }
```

Pattern matching on enums is supported:

```
match opcode {
    Op::Add => result = a + b,
    Op::Sub => result = a - b,
    Op::Xor => result = a ^ b,
    _ => result = 0,
}
```

This eliminates the distinction between procedural and continuous assignments for many common patterns, reducing the frequency of errors that arise from mixing blocking and non-blocking assignment semantics in SystemVerilog.

F. Built-in Verification

Verification constructs are part of the language rather than a separate verification layer:

```
assert!(count <= DEPTH, "FIFO count
    overflow")
assume!(wr_en -> !full, "Write when not
    full")
cover!(count == DEPTH, "FIFO reaches full")
```

These constructs generate both simulation checks and formal verification properties. Temporal properties use a linear temporal logic (LTL) inspired syntax for specifying multi-cycle behaviors, and the toolchain integrates SMT-based bounded model checking for property verification.

IV. Compilation Pipeline

A. Four-IR Architecture

SKALP uses four distinct intermediate representations, each optimized for a specific phase of the design flow:

```
Source (.sk) --> HIR --> MIR --+--> LIR
                               |     |
                               |     +-->
    SIR (gate)
                               |
                               +--> SIR (
    behavioral)
```

HIR (High-level IR) preserves the full richness of the source language: generics, pipeline annotations, safety attributes, clock domains, power intent, and design hierarchy. HIR remains polymorphic—generic entities are not yet specialized. The purpose of HIR is to serve as a faithful representation of designer intent that can be queried, analyzed, and transformed before committing to concrete hardware decisions. HIR is produced by the frontend (`skalp-frontend` crate) from the parsed AST.

MIR (Mid-level IR) is a cycle-accurate RTL representation with concrete ports, signals, processes with sensitivity lists, module hierarchy, and clock domain information. Generics have been monomorphized, types have been resolved to concrete bit widths, and the design is ready for SystemVerilog generation. MIR retains safety context (`SafetyContext` on modules, signals, and instances) and pipeline configuration. The MIR is where CDC analysis is performed, as clock domains and their interactions are fully resolved at this level. MIR also carries formal verification assertions, generate blocks, and vendor IP configurations.

LIR (Low-level IR) represents the design as word-level primitives: standard logic gates, sequential elements, arithmetic operators, FPGA LUT primitives, power infrastructure cells, and NCL threshold gates. Critically, LIR preserves multi-bit operations (e.g., an 8-bit adder is a single `Add` node, not 8 individual full-adders) rather than eagerly decomposing to individual gates. This design choice is deliberate: technology libraries frequently contain compound cells (ADDER8, MUX4, AOI22), and premature decomposition destroys information needed for optimal technology mapping. Each LIR node carries optional `LirSafetyInfo` that tracks the safety classification and FIT rate contribution of the primitive, maintaining traceability from source-level safety annotations to individual gates.

SIR (Simulation IR) is a flat, GPU-optimized representation with explicit separation of combinational and sequential logic, pre-computed topological ordering, and flattened module hierarchy. SIR is designed to minimize control flow divergence during parallel simulation: all combinational nodes are evaluated in topological order, then all sequential nodes

are updated, producing a two-phase simulation cycle that maps naturally to GPU compute kernels. SIR carries implementation style hints from the intent system, type information including signed and floating-point types, and a name registry for mapping hierarchical signal paths to internal identifiers.

B. Lowering Transformations

Each lowering step is a well-defined transformation:

HIR to MIR (`skalp-mir` crate) performs monomorphization, type resolution, clock domain inference, and safety context propagation. Generic entities are specialized for each unique set of type/const parameters. This step also performs CDC analysis and reports violations.

MIR to LIR (`skalp-lir` crate) decomposes RTL processes into primitive operations while preserving word-level semantics. Conditional assignments become multiplexers, sequential blocks become registers with enable logic, and arithmetic operations become word-level primitives. Safety annotations from MIR's `SafetyContext` are mapped to LIR's per-primitive `LirSafetyInfo`. The LIR module provides separate lowering functions for different use cases: `lower_mir_module_to_lir` for full gate-level decomposition, `lower_mir_module_to_word_lir` for word-level preservation, and `lower_mir_hierarchical` for hierarchical multi-module designs.

MIR to SIR occurs along two distinct paths (see Section IV-C). The behavioral path (`skalp-sir` crate) compiles MIR directly to SIR, preserving multi-bit semantics for efficient simulation. The gate-level path goes through LIR first: MIR to LIR to gate netlist to SIR, producing a gate-level SIR where each node corresponds to a single primitive for fault injection.

C. Path Divergence

A key architectural decision is that the compilation pipeline **diverges at MIR**:

- **Synthesis path:** HIR → MIR → LIR → AIG optimization → Technology mapping → GateNetlist → SystemVerilog or bitstream.
- **Behavioral simulation path:** HIR → MIR → SIR → Metal shaders or C++ compiled kernels.
- **Fault simulation path:** HIR → MIR → LIR → GateNetlist → SIR (gate-level).
- **Formal verification path:** HIR → MIR → LIR → AIG → SAT/SMT solvers.

This divergence is not merely an implementation convenience but a deliberate design choice: behavioral simulation needs multi-bit semantics for speed, fault simulation needs gate-level decomposition for accuracy, and synthesis needs word-level optimization opportunities. A single IR cannot serve all three purposes without compromising one.

V. Simulation Architecture

A. Behavioral Simulation

SKALP provides two behavioral simulation backends:

GPU backend (Metal). On macOS systems with Apple Silicon, behavioral simulation compiles SIR modules to Metal compute shaders (`skalp-sim/gpu_runtime.rs`). The SIR's separation of combinational and sequential nodes maps directly to GPU kernel structure: a combinational kernel evaluates all nodes in topological order, then a sequential kernel updates all registers. Double-buffered register storage (main and shadow buffers) ensures that sequential updates are atomic—the sequential kernel reads from one buffer and writes to the other, then buffers are swapped.

CPU backend (Compiled C++). For portability and debugging, SIR modules are compiled to C++ code via a code generation backend (`skalp-sim/cpp_compiler.rs`), which is then compiled to a shared library (`.so` on Linux, `.dylib` on macOS) and loaded at runtime. This approach provides near-native simulation speed without requiring GPU hardware.

Both backends use a `UnifiedRuntime` dispatch layer that selects the appropriate backend based on platform capabilities and user configuration. On macOS with Apple Silicon, the GPU backend is used by default; the `SKALP_SIM_MODE=cpu` environment variable forces CPU simulation for debugging or compatibility.

B. Fault Simulation

Fault simulation is the core driver for SKALP's GPU architecture. The problem is embarrassingly parallel: given a gate-level netlist and a set of test vectors, each possible stuck-at fault can be simulated independently. SKALP exploits this parallelism through its GPU fault simulator (`skalp-sim/gpu_fault_simulator.rs`):

1) The design is compiled to a gate-level SIR through the LIR path.
2) A fault campaign is generated: for each gate output, stuck-at-0 and stuck-at-1 faults are enumerated.
3) The gate-level SIR, test vectors, and fault list are uploaded to GPU memory.
4) A Metal compute shader is dispatched with one thread per fault. Each thread simulates the complete design for all test vectors with its assigned fault injected, producing a detected/not-detected result.
5) Results are collected to compute fault coverage metrics.

This architecture achieves approximately 11 million fault-cycle simulations per second on M4 Max (40 GPU cores). For a 1,470-gate FIFO design with complete fault enumeration (5,880 faults across stuck-at-0, stuck-at-1, bit-flip, and transient types), a 10,000-cycle campaign (58.8 million fault-cycle simulations) completes in 5.4 seconds.

C. SIR Design for Parallelism

Several SIR design decisions specifically target parallel execution:

- **Pre-computed topological order.** The `sorted_combinational_node_ids` field stores the evaluation order, eliminating the need for each thread to perform graph traversal.

- **Flat hierarchy.** Module instances are inlined, producing a single flat list of nodes. This eliminates indirect function calls during simulation.
- **Explicit signal types.** The `SirType` enum preserves width and signedness information, enabling the simulator to select appropriate operations without runtime type dispatch. Supported types include arbitrary-width bit vectors (`Bits(N)`, `SignedBits(N)`), IEEE 754 floating-point formats (`Float16`, `Float32`, `Float64`), vectors (`Vec2`, `Vec3`, `Vec4`), and arrays.
- **Two-phase evaluation.** The separation into `combinational_nodes` and `sequential_nodes` vectors means that the combinational evaluation kernel and sequential update kernel can be dispatched independently, with an implicit barrier between them.

VI. Safety Integration

A. ISO 26262 Workflow

ISO 26262 [5] requires that safety-critical automotive ICs demonstrate traceability from safety goals through hardware architectural metrics. The key metrics are the Single Point Fault Metric (SPFM), Latent Fault Metric (LFM), and Probabilistic Metric for random Hardware Failures (PMHF). Computing these metrics requires knowing, for each gate in the design: (a) its failure mode distribution, (b) which safety mechanisms detect its failures, and (c) the diagnostic coverage of those mechanisms.

Traditionally, this information is assembled manually: a safety engineer reviews the gate-level netlist, classifies each cell, maps safety mechanisms to cells, and computes metrics in a spreadsheet. This process is error-prone, does not scale, and is disconnected from the design flow.

B. Annotation Propagation

SKALP makes safety annotations first-class language constructs that propagate through all four IRs:

HIR level. The designer declares safety intent using attributes. `DetectionConfig` on ports and signals specifies how fault detection operates (continuous, boot-time, periodic, on-demand). `SeoocConfig` (Safety Element out of Context, per ISO 26262-10:9) declares assumptions about external safety mechanisms. `PowerDomainConfig` assigns power domains for Common Cause Failure (CCF) analysis.

```
#[safety_mechanism(type=tmr)]
#[implements(SG001::TmrVoting)]
entity TmrVoter {
    #[detection_signal(mode=continuous)]
    out fault_detected: bit
}
```

MIR level. `SafetyContext` aggregates module-level safety information: which safety goal an element implements, whether it is part of a safety mechanism, its mechanism name, and optional diagnostic/latent coverage overrides. The MIR carries `ModuleSafetyDefinitions` that define safety goals, mechanisms, and the Hardware-Software Interface (HSI).

LIR level. Each primitive node carries `LirSafetyInfo` that tags individual gates with their safety classification, base FIT rate, and the safety mechanism (if any) that covers them.

SIR level. Detection signal net IDs are tracked so that fault simulation can determine, for each injected fault, whether the detection signal was asserted—directly computing diagnostic coverage from simulation rather than relying on structural analysis alone.

C. Automated FMEDA Generation

SKALP's `skalp-safety` crate generates FMEDA tables directly from the gate-level netlist. For each cell, the system:

1) Looks up the cell type (AND, OR, DFF, MUX, etc.) in a failure mode library that provides standard distributions per IEC 61508 Annex C [16].
2) Classifies the cell as safe, dangerous-detected, or dangerous-undetected based on the safety mechanism annotations.
3) Computes base FIT rates and applies diagnostic coverage from either the annotation or simulation results.
4) Generates a per-cell FMEDA entry with hierarchical path, safety classification, failure distribution, and effective FIT rates.
5) Aggregates results to compute SPFM, LFM, and PMHF at the module and design level.

The FMEDA output is exported as CSV for assessor review, maintaining compatibility with existing safety assessment workflows while eliminating the manual netlist classification step.

D. Safety-Aware Synthesis

The synthesis engine in SKALP is safety-aware: AIG optimization passes preserve safety barrier annotations. Certain gates may be marked as safety boundaries that the optimizer must not merge or restructure across. This ensures that a safety mechanism's detection path is not inadvertently optimized away or combined with the function it is protecting—a common and dangerous failure mode in traditional synthesis flows where the tool has no knowledge of safety intent.

VII. Implementation

A. Rust Implementation

SKALP is implemented in Rust, which provides several advantages for a compiler toolchain: memory safety without garbage collection (critical for long-running synthesis jobs), a strong type system that catches many compiler implementation bugs at compile time, zero-cost abstractions that enable high-performance data structures, and Cargo's ecosystem for dependency management and testing.

The implementation comprises approximately 290,000 lines of Rust code organized into 23 crates (22 workspace crates plus the root binary crate), as shown in Table I.

TABLE I
SKALP CRATE ORGANIZATION.

Crate	Purpose
`skalp-frontend`	Lexer, parser, AST, HIR, type system
`skalp-mir`	MIR, HIR-to-MIR lowering, CDC analysis
`skalp-lir`	LIR, MIR-to-LIR, AIG synthesis, tech map
`skalp-sir`	Behavioral SIR, MIR-to-SIR lowering
`skalp-sim`	Simulation runtimes (GPU, CPU), faults
`skalp-codegen`	SystemVerilog code generation
`skalp-backends`	Backend dispatch, target configuration
`skalp-safety`	FMEDA generation, ASIL metrics
`skalp-formal`	Equivalence checking, BMC, SAT/SMT
`skalp-verify`	Assertions, properties, coverage
`skalp-place-route`	iCE40 packing, placement, routing
`skalp-asic`	ASIC timing, reset tree synthesis
`skalp-ml`	Synthesis heuristics
`skalp-lsp`	Language Server Protocol
`skalp-stdlib`	Standard library
`skalp-package`	Package manager
`skalp-manifest`	Project manifest parsing
`skalp-resolve`	Dependency resolution
`skalp-lint`	Design rule checking
`skalp-testing`	Test infrastructure, testbench runtime
`skalp-parallel`	Parallel compilation
`skalp-incremental`	Incremental compilation

B. Synthesis Engine

The synthesis engine operates on And-Inverter Graphs (AIGs) [13], a canonical representation for Boolean functions. The pipeline is:

```
GateNetlist -> AIG Builder -> AIG
  -> [Optimization Passes]
  -> AIG Writer -> GateNetlist
```

Optimization passes include structural hashing (`strash`), rewriting, balancing, resubstitution (`resub`), refactoring, retiming, don't-care-based optimization (`dc2`), sequential equivalence reduction (`scorr`), functionally-reduced AIGs (`fraig`), constant propagation, dead code elimination, and buffer optimization. These passes can be composed in configurable sequences, with preset configurations (e.g., speed-optimized, area-optimized) available.

C. Place-and-Route

SKALP includes native place-and-route for the Lattice iCE40 FPGA family. The flow includes:

- **Packing:** Logic cells are packed into iCE40 PLBs (Programmable Logic Blocks).
- **Placement:** Simulated annealing and analytical placement engines position packed blocks on the device grid, with constraint support for pin assignments and region constraints.
- **Routing:** A PathFinder-based [17] negotiated-congestion router assigns signals to the FPGA's routing fabric, with special handling for global clock networks, carry chains, and LUT input permutation.
- **Bitstream generation:** The routed design is serialized to iCE40 bitstream format (compatible with IceStorm [18] tooling) for device programming.

Timing analysis is integrated into the place-and-route flow, with delay models derived from the iCE40 chip database.

D. Toolchain

SKALP provides a unified command-line interface:

- `skalp build` — Compile design to SystemVerilog, bitstream, or simulation artifacts
- `skalp sim` — Run behavioral or gate-level simulation
- `skalp synth` — Run logic synthesis and optimization
- `skalp pnr` — Place and route for FPGA targets
- `skalp program` — Program FPGA devices
- `skalp fmt` — Format source code
- `skalp lint` — Run design rule checks
- `skalp test` — Run testbenches
- `skalp ec` — Run equivalence checking
- `skalp safety` — Run safety analysis and generate FMEDA
- `skalp verify` — Run formal verification
- `skalp new` — Create new project from template
- `skalp add/remove/update/search` — Package management

An LSP server (`skalp-lsp` crate) provides IDE integration including go-to-definition, find references, context-aware completion, hover documentation, document and workspace symbols, semantic token highlighting, and real-time diagnostics. The accompanying VSCode extension (`vscode-skalp`) adds an integrated waveform viewer for `.skw` files with zoom, pan, multi-radix signal display, and breakpoint markers; a Debug Adapter Protocol (DAP) implementation supporting cycle-level and half-cycle stepping, conditional breakpoints, and variable inspection across input, output, register, and signal scopes; testbench integration through VSCode's test explorer with per-test waveform association; and schematic and expression viewers.

E. Standard Library

The standard library provides reusable components and type definitions:

- **Bit operations:** Shift, rotate, population count, leading/-trailing zeros
- **Math operations:** Arithmetic, comparison, reduction
- **Fixed-point arithmetic:** Configurable integer/fractional widths
- **Floating-point types:** IEEE 754 half, single, and double precision
- **Vector operations:** 2/3/4-component vectors for graphics and DSP
- **Trigonometric functions:** CORDIC-based implementations
- **Hardware components:** Counters, FIFOs, shift registers, adders, multipliers, UART, SPI, AXI4-Lite interfaces
- **Safety primitives:** TMR voters, error detection components

TABLE II
FAULT SIMULATION THROUGHPUT ON APPLE M4 MAX (40 GPU CORES).

Design	Gates	Faults	Cycles	Time	Throughput
ALU	1,162	4,648	10,000	4.01 s	11.6M/s
FIFO	1,470	5,880	10,000	5.42 s	10.8M/s

VIII. EVALUATION

A. Fault Simulation Performance

We measured fault simulation throughput on Apple Silicon hardware using the GPU-accelerated fault simulator. Throughput is measured in fault-cycle simulations per second, where each fault-cycle represents evaluating the complete design for one simulation cycle with one injected fault.

Throughput stabilizes at approximately **11M fault-cycle simulations per second** once simulation dominates over fixed overhead (parsing, lowering, SIR conversion). For a design with 1,470 gate-level primitives and complete stuck-at-0/stuck-at-1/bit-flip/transient fault enumeration (5,880 faults), a 10,000-cycle fault campaign completes in under 6 seconds on commodity hardware.

B. Compilation Pipeline

The four-IR pipeline introduces compilation overhead compared to single-pass approaches, but each IR stage enables optimizations that would be difficult or impossible in a monolithic compiler:

- HIR enables polymorphic analysis and intent verification before committing to concrete hardware.
- MIR enables CDC analysis with full clock domain resolution.
- LIR enables word-level synthesis optimizations that exploit multi-bit structure.
- SIR enables simulation-specific optimizations (topological sorting, hierarchy flattening) without affecting synthesis.

The path divergence at MIR means that behavioral simulation does not pay the cost of LIR lowering, while fault simulation and synthesis benefit from the detailed gate-level representation.

C. Supported Targets

SKALP currently supports:

- **Simulation:** Metal GPU (macOS), compiled C++ (cross-platform)
- **RTL output:** SystemVerilog
- **FPGA:** Lattice iCE40 family (native place-and-route and bitstream generation)
- **ASIC:** Timing analysis and reset tree synthesis (gate-level netlist export)
- **Asynchronous:** NCL dual-rail circuits with threshold gate synthesis

D. NCL Synthesis Results

The NCL synthesis flow converts synchronous Boolean logic to dual-rail NCL circuits. The optimize-first approach—mapping to single-rail Boolean gates, applying standard optimization, then converting to dual-rail NCL—typically reduces gate count by 50–80% compared to direct NCL expansion, as the optimizer can eliminate redundant logic before the dual-rail transformation doubles the gate count.

IX. LIMITATIONS AND FUTURE WORK

We are candid about SKALP's current limitations:

Early-stage tooling. SKALP is under active development. While the compilation pipeline, simulation backends, and safety analysis are functional, the toolchain has not yet been validated on large industrial designs. The synthesis quality of results (QoR) has not been benchmarked against mature commercial tools.

Limited FPGA target support. Native place-and-route currently supports only the Lattice iCE40 family. Targeting other FPGA families (Xilinx 7-series, Intel Cyclone/Stratix) requires generating vendor-compatible SystemVerilog and using vendor tools for implementation. Expanding native PAR support is a priority.

GPU backend portability. The GPU simulation backend currently requires Apple Metal, limiting it to macOS. A Vulkan or WebGPU backend would enable GPU-accelerated simulation on Linux and Windows. The CPU backend (compiled C++) is available on all platforms as a fallback.

Formal verification scalability. The integrated BMC and CEC engines handle moderately sized designs but have not been tested on designs with millions of gates. Integration with industrial-strength solvers and abstraction techniques is planned.

Standard library coverage. While the standard library includes common components, coverage of complex IP blocks (DDR controllers, PCIe interfaces, Ethernet MACs) is limited. The package manager infrastructure is in place to support community-contributed libraries.

Language evolution. Some language features (e.g., trait bound enforcement at instantiation time, comprehensive pattern matching exhaustiveness checking) are partially implemented and continue to evolve.

Future work priorities include: expanding FPGA target support, porting the GPU backend to Vulkan/WebGPU, scaling formal verification, expanding the standard library, publishing benchmark comparisons against established tools, and building a community package ecosystem.

X. CONCLUSION

SKALP demonstrates that hardware description languages can preserve designer intent through compilation without sacrificing the control that hardware designers require. By structuring the compiler around four intermediate representations—each optimized for a specific purpose—SKALP enables capabilities that are difficult or impossible in single-IR architectures: compile-time clock domain safety through the

type system, automated safety analysis with traceability from source annotations to individual gates, and GPU-accelerated fault simulation on the same IR pipeline used for synthesis.

The key insight is that the information needed for safety analysis, simulation, and synthesis overlaps but does not coincide. Rather than forcing a single representation to serve all purposes (and compromising each), SKALP's path divergence at MIR allows each downstream consumer to receive the representation it needs. Safety annotations flow through all paths, ensuring consistency.

SKALP is open source and available at https://github.com/girivs82/skalp.

REFERENCES

[1] IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800-2017, 2018.

[2] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-2019, 2019.

[3] Xilinx, Inc., "Vivado Design Suite User Guide: High-Level Synthesis," UG902, 2020.

[4] Siemens EDA, "Catapult High-Level Synthesis Reference Manual," 2021.

[5] International Organization for Standardization, "ISO 26262: Road vehicles—Functional safety," 2018.

[6] The Rust Programming Language. [Online]. Available: https://doc.rust-lang.org/book/

[7] Synopsys, Inc., "SpyGlass CDC User Guide," 2023.

[8] J. Bachrach *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. Design Automation Conf. (DAC)*, 2012, pp. 1216–1225.

[9] Amaranth HDL. [Online]. Available: https://amaranth-lang.org/

[10] Spade HDL. [Online]. Available: https://spade-lang.org/

[11] C. Baaij *et al.*, "Clash: Structural descriptions of synchronous hardware using Haskell," in *Proc. Euromicro Conf. Digital System Design (DSD)*, 2010, pp. 714–721.

[12] R. Nikhil, "Bluespec System Verilog: Efficient, correct RTL from high-level specifications," in *Proc. Formal Methods and Models for Co-Design (MEMOCODE)*, 2004, pp. 69–70.

[13] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification (CAV)*, LNCS 6174, 2010, pp. 24–40.

[14] C. Wolf, "Yosys Open SYnthesis Suite." [Online]. Available: https://yosyshq.net/yosys/

[15] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, LNCS 4963, 2008, pp. 337–340.

[16] International Electrotechnical Commission, "IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems," 2010.

[17] L. McMurchie and C. Ebeling, "PathFinder: A negotiation-based performance-driven router for FPGAs," in *Proc. Field-Programmable Gate Arrays (FPGA)*, 1995, pp. 111–117.

[18] C. Wolf, "Project IceStorm." [Online]. Available: https://github.com/YosysHQ/icestorm