Defining Safe Hardware Design

Rachit Nigam
Massachusetts Institute of Technology

Abstract

Type systems have been remarkably successful within the software engineering community as a lightweight mechanism for formal verification: sophisticated type systems, such as Rust's, can prove properties such as memory safety without imposing any overheads. However, type-based approaches have seen little development for hardware description languages (HDLs): most state of the art languages provide simple guarantees that do not dramatically reduce the verification burden. We define more powerful criteria for *safe hardware design*, discuss how language-based approaches enforce these properties in recent HDLs, and make a case for the development of sophisticated type systems for HDLs that provide order of magnitude improvements in the hardware verification process.

1 Type Safety

Legacy HDLs [6] provided minimal static checking; a common source of bugs was *bitwidth mismatches* where an input port on a module might expect 8-bit signals but the user was allowed to connect a 16-bit value which silently lost information. Modern HDLs [2, 3, 8, 9] remedy this problem by performing *type checking* and generating a compile-time error message; if the user wants to connect a 16-bit wire to an 8-bit port, they have to explicit *truncate* the signal.

Type safety is a *design-time* abstraction: at the netlist-level, the bits on a wire do not know if they are an IP packet headers or the mantissa of a floating-point number. However, at design time, users can use a type system to mark some 8-bit values as mantissas and others as IP headers. Furthermore, because type safety is checked statically, it imposes no overhead on the final circuit.

However, type safety alone is not enough to guarantee correctness. For example, in software languages like C, a program can be type safe but still exhibit bugs because of memory management bugs such as double-frees and use-after-free. To address these challenges, the categorized bugs that pointer-manipulating programs suffer from and called them *memory safety* violations. By defining a target, the community focused its efforts towards a clear and shared goal and developed a wide arsenal of techniques to attack the problem; run-time mechanisms like automatic memory management as well as compile-time such as Rust's type systems. We argue that hardware design similarly rich notions of *safety* which categorizes a large class of bugs and gives our community a clear goal to achieve.

2 Eliminating Structural Hazards

In order to provide this guarantee, we develop clearer definition of resource constraint violation based on Filament's formalism [10]. From a producer-consumer model of a pipeline, there are two causes of a structural hazard: (1) if a consumer reads a value that the producer did not intend for it to, and (2) if a producer assumes that the consumer accepted a value from it when it did not.

Latency safety. At the netlist-level, circuits continuously read and write values to wires. At the design-level, the programmer has to define which signals are *meaningful* and should be used to perform a computation. This is usually done through a *protocol* which defines how physical signals should be used over time. For example, a producer might provide an explicit 1-bit `valid` signal that the consumer must check before using the value on the data signal. Similarly, a producer might declare its latency is four cycles and the consumer is responsible for tracking the passage of time and read the output on the correct cycle.

However, like data types, protocols are a purely semantic concepts; they do not exist within the netlist. Therefore, failing to follow such protocols leads to silent data corruption which is hard to debug. **A latency-safe HDL guarantees that whenever a signal value is *used*, it is *semantically meaningful***, as defined by a protocol.

Resource safety. On the consumer side, a module uses a protocol to define when it can accept new inputs, i.e., its *reuse constraint*. For example, it might use a 1-bit `ready` signal to indicate that it can accept new inputs or statically declare that it can accept new inputs every two cycles. Once again, these guidelines are purely semantic but failing to follow them creates logical errors in the design which are hard to debug. **A resource-safe HDL guarantees that all reuse constraints on resources are respected.**

3 Analysis

Like memory safety, latency and resource safety can be enforced using language abstractions, dynamic mechanisms, and static reasoning. Table 1 overviews existing safe HDLs and their enforcement mechanisms. Traditional and modern HDLs do not provide safety guarantees so we elide their discussion.

Bluespec. Bluespec modules are organized as *rules*—single-cycle computations with a combinational guard that determines if the rule can execute in a given cycle. Rules can

HDLs	Abstraction	Type	Latency	Resource
Modern [1, 7], Embedded [2, 3, 8]	RTL	✓	✗	✗
Rules-based [4, 12]	Atomic Actions	✓	**Programmatic**	**Dynamic**
Synchronous [5, 13, 14]	Pipelines	✓	✓	✗
Filament [10, 11]	Pipelines	✓	✓	✓
Anvil [15]	Message-passing	✓	✓	**Programmatic**

Table 1. *Type safety* ensures that bits on a wire represent semantic data structures, *latency safety* ensures that values are read when they are meaningful, and *resource safety* ensures that resources are used when they are available. Properties are enforced at compile time (✓), through extra circuitry (**Dynamic**), or through language abstractions (**Programmatic**).

use overlapping sets of resources which creates opportunities for resource safety violations. However, Bluespec's compiler analyzes all the rules and synthesizes a *scheduler* which dynamically detects and eliminates conflicts by aborting the execution of conflicting rules.

Next, methods provide a structured way for Bluespec modules to communicate and enforce latency safety. For example, performing a push or peek on a FIFO is done by calling the respective method on an instance. Bluespec's compiler then generates a `enable-ready` interface to activate the method and ensure that it receives meaningful data, ensuring latency safety. The downside is all modules in Bluespec must use the same interface.

While Bluespec is arguably the first safe HDL, its enforcement mechanisms limit both expressivity, by requiring designs to use methods and rules, and are expensive, requiring synthesis of a scheduling circuit.

Synchronous HDLs. Synchronous HDLs use a type system to enforce latency safety. Pipelines and registers are first-class constructs in such languages and allow the type system to track the cycle in which a signal is produced using a *latency tag*. If a combinational computation attempts to use signals with different latency tags, the user gets a compile-time error message indicating that the pipeline may be imbalanced. Some embedded HDLs [2, 8] also provide this reasoning capability. However, this approach only works with *statically-known* latencies and does not allow for variable latency computations. Most HDLs in this category provide escape hatches to latency safety to allow for general purpose design and therefore do not guarantee that all programs are latency safe. Synchronous HDLs do not provide resource safety. For example, Spade [13] and Sus [14] do not support reasoning about partially pipelined modules.

Filament. Filament [10] introduced the concept of latency safety and resource safety and demonstrated that they can be enforced using purely static reasoning. Module signatures in Filament use *events*—which model `valid` signals—and availability intervals—which capture when signals are required and provided—to track the timing behavior of a module. Events additionally provide a *delay* which describes how often the valid signal can be toggled and therefore captures the reuse constraint for a module. Filament's type system statically guarantees that all signals are used when they are available (as defined by their *availability interval*) and modules are sent inputs at a rate they can accept them.

Filament's approach has a couple of benefits. First, it does not need to "bake-in" the concept of registers or pipelines; its type system is powerful enough to characterize registers as just another module with a type signature. Second, Filament's multi-cycle availability intervals allow modules to express that a producer needs to hold a value stable for multiple cycles allowing optimizations such as multi-cycle paths. Finally, Filament's reasoning has been recently extended to parameterized and generator-produced designs enable it to statically guarantee that *all possible parameterizations* of a design are latency and resource safe. However, like synchronous HDLs, Filament is limited to static pipelines.

Anvil. Anvil uses the message passing abstraction to describe hardware designs: modules use the `send` and `recv` operators to pass signals through channels and Anvil's compiler synthesizes a ready-valid interface for each channel. Anvil defines a type system that uses Filament's abstractions of events and availability intervals to guarantee *timing safety* which combines latency safety with the guarantee registers are not mutated if a future computation may use their value. Finally, Anvil programmatically enforces resource safety by disallowing multiple senders to provide a value on the same channel. Anvil, like Bluespec, enables general-purpose and safe hardware design but suffers from the abstraction overhead: designs *must use* the message passing abstraction to receive its guarantees and are limited to single producer-consumer relationships.

4 Conclusion

New abstractions for safe hardware design can bring the same revolutionary reusability and reliability that languages like Rust brought to the software ecosystem. However, we should not let our ambitions run amok: safe HDLs should not aim to provide the same guarantees as heavyweight formal tools and instead focus on automatically eliminating a broad class of bugs that are obviously beneficial for every hardware design.

References

[1] IEEE Standards Association. 2018. IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. doi:10.1109/IEEESTD.2018.8299595

[2] C. Baaij, M. Kooijman, J. Kuper, A. Boeijink, and M. Gerards. 2010. CλaSH: Structural Descriptions of Synchronous Hardware Using Haskell. In *Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. doi:10.1109/DSD.2010.21

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. doi:10.1145/2228360.2228584

[4] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. 2020. The essence of Bluespec: a core language for rule-based hardware design. doi:10.1145/3385412.3385965

[5] Steven F Hoover. 2017. Timing-abstract circuit design in transaction-level Verilog. In *IEEE International Conference on Computer Design (ICCD)*.

[6] IEEE. 2006. IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)* (2006).

[7] IEEE. 2009. IEEE Standard VHDL Language Reference Manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (Jan 2009).

[8] Jane Street. 2022. *HardCaml: Register Transfer Level Hardware Design in OCaml*. Retrieved October 15, 2022 from https://github.com/janestreet/hardcaml

[9] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. doi:10.1109/MICRO.2014.50

[10] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular Hardware Design with Timeline Types. doi:10.1145/3591234

[11] Rachit Nigam, Ethan Gabizon, Edmund Lam, Carolyn Zech, Jonathan Balkind, and Adrian Sampson. 2026. Parameterized Hardware Design with Latency-Abstract Interfaces.

[12] Rishiyur Nikhil. 2004. Bluespec System Verilog: Efficient, correct RTL from high level specifications. In *Conference on Formal Methods and Models for Co-Design (MEMOCODE)*. doi:10.1109/MEMCOD.2004.1459818

[13] Frans Skarman and Oscar Gustafsson. 2023. Spade: An Expression-Based HDL With Pipelines. doi:10.48550/arXiv.2304.03079

[14] Lennart Van Hirtum and Christian Plessl. 2024. Latency counting in the SUS language. In *Workshop Languages, Tools, and Techniques for Accelerator Design*.

[15] Jason Zhijingcheng Yu, Aditya Ranjan Jha, Umang Mathur, Trevor E. Carlson, and Prateek Saxena. 2025. Anvil: A General-Purpose Timing-Safe Hardware Description Language. arXiv:2503.19447 [cs.AR] https://arxiv.org/abs/2503.19447