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 [5] 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 [1, 2, 6, 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* property: at the netlist-level, the circuit does not distinguish between bits that represent packet headers from those that represent the mantissa of a floating-point number. However, by statically specifying this information, an HDL can automatically check if there are any logical bugs that treat an packet header as a mantissa. Furthermore, because this enforcement is completely static, it has no overhead in the final circuit.

However, type safety alone is not enough to guarantee correctness. For example, in software languages like C, a type-safe program can still be wrong due memory management bugs such as double-frees and use-after-free. Such bugs, called *memory safety* violations, are *ubiquitous*: they can arise in any program, regardless of the specific application logic. Because of this ubiquity and the challenge of debugging them, the software community categorized and formally defined *memory safety* as an important design criterion for new languages.

This shared definition enabled development of many different static and dynamic approaches to enforcing memory safety as well as a clear understanding of the trade-offs.[1]

[1]This statement is broadly true but the definition of memory-safety is still being debated in the parallel program setting [7, 8].

We argue that hardware design should similarly define richer notions of *safety* which categorizes a ubiquitous class of bugs that should be eliminated through language design. We identify *structural hazards* as a possible candidate for such a definition of safety, give a precise definition, and use it to characterize existing HDLs and their capabilities.

2 Eliminating Structural Hazards

Pipeline hazards, commonly taught in the context of processor design, fall into three categories: *structural hazards*, which relate to resource use violations, *data hazards*, which correspond to data races, and *control hazards*, which arise from speculative execution of unresolved branches. Pipeline hazards are important because they are a prerequisite for functional correctness: a design that has a hazard cannot implement a higher-level specification. Of these, data and control hazards are application-specific: they arise because a processor has to appear as if it is executing instructions sequentially. On the other hand, structural hazards can *ubiquitously* appear in any pipelined hardware design and is, therefore, a candidate for a richer definition of safety.

We provide a definition of structural hazards independent of their appearance in a processor pipeline. An abstract pipeline consists of stages, each of which acts as either a producer or a consumer. Structural hazards occur when there is a violation in how producers and consumers interact.

Specifically, building on Filament's formalism [10], there are two causes of a structural hazard: (1) if a consumer reads an invalid value (i.e., a value not marked as valid by the producer), and (2) if a producer assumes that the consumer accepted a value from it when it did not. The two definitions of safety correspond to these errors.

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**

HDLs	Abstraction	Type	Latency	Resource	Fully Expressive
Embedded [1, 2, 6]	RTL	✓	✗	✗	✓
Rules-based [3, 12]	Atomic Actions	✓	**Programmatic**	**Dynamic**	✓
Synchronous [4, 13, 14]	Pipelines	✓	✓	✗	✗
Filament [10, 11]	Pipelines	✓	✓	✓	✗
Anvil [15]	Message-passing	✓	✓	**Programmatic**	✓

Table 1. Safety properties in HDLs. *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. Safety guarantees are enforced at compile time (✓), through extra circuitry (**Dynamic**), or through language abstractions (**Programmatic**). *Fully expressive* means that the language can express arbitrary circuits.

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

To demonstrate the utility of our two definitions of safety, we taxonomize the capabilities of existing HDLs. Concretely, latency and resource safety can be enforced in three ways:

1. *Language abstractions*: By making unsafe programs unrepresentable through structured abstractions.
2. *Dynamically*: By generating circuitry that dynamically ensures absence of safety violations.
3. *Statically*: Through the use of static mechanisms such as type systems.

Table 1 overviews existing HDLs and their safety 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 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.

By our definitions, Bluespec is arguably the first safe HDL. However, its safety guarantees are tied to its abstractions: a programmer must reorganize their programs to use methods and rules and must pay the cost of scheduling circuit.[2]

Synchronous HDLs. Synchronous HDLs such as Spade [13], SUS [14], and TL-Verilog [4], 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 [1, 6] 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 and Sus 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 is expressive and zero-cost: latency and resource safety are enforced purely through type-level

[2]Advanced users can eliminate much of the scheduling circuitry but need to carefully reason about correctness.

reasoning and impose no overhead in the final circuit. Furthermore, the general abstractions of events and intervals can provide a type for every circuit with input-independent timing behavior. These ideas have recently been extended to parameterized designs enabling Filament to provide a first of its kind guarantee that *all possible parameterizations* of a design are safe. However, like synchronous HDLs, Filament is limited to static pipelines and cannot express circuits with input-dependent timing behaviors.

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 along with a guarantee that registers are not mutated before a future computation needs to use their value. Finally, it guarantees resource safety by programmatically disallowing reuse of channels which limits expressivity. Anvil, like Bluespec, enables general-purpose and safe hardware design but suffers from the abstraction overhead: designs *must use* the message passing abstraction.

4 Alternative Properties

Our proposed definitions of latency and resource safety represents a ubiquitous class of bugs and taxonomize existing approaches and provide a unified framework to understand the guarantees of different HDLs. We overview some alternative properties proposed in the community.

Transactional updates. Like all concurrent programming models, hardware design needs to contend with interleaved state updates. Bluespec's programming model allows designs to group together logically related state updates into an atomic action and ensures that such a group appears to execute as a transaction. The property is cross-cutting and hierarchical: a rule might affect the state of other modules by calling methods, which might themselves call rules, creating a cascade of changes that execute atomically. While elegant and powerful, this property is expensive to enforce dynamically. To our knowledge, no static approaches for enforcing transactional updates have been proposed for HDLs. We hypothesize any such approach would either reject too many useful programs or run into expressivity limitations due to over-approximation needed for sound enforcement.

Timing safety. Anvil defines timing safety through three requirements on its messaging passing primitives (`send` and `recv`): (1) VALID USE: values are used when they are valid, (2) VALID REGISTER MUTATION: registers are not mutated while they are loaned (i.e., some future computation depends on their value), and (3) VALID MESSAGE SEND: values are provided for the duration and messages do not overlap. These properties are equivalent to latency and resource safety: VALID USE and the first part of VALID MESSAGE SEND correspond to the consumer and producer side requirements for latency safety while the absence of overlapping messages is required by resource safety. Finally, VALID REGISTER MUTATION is also required for resource safety since Anvil treats registers as a built-in primitive; within Filament, the same property is automatically enforced as a consequence of general constraints on availability intervals.

5 Conclusion

New abstractions for safe hardware design can transform the discipline and bring the reusability and reliability that are widely available in the software community. Achieving this vision requires the community to crisply define and unite behind the goal and tackle it with tools, techniques, and linguistic abstractions!

References

[1] 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

[2] 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

[3] 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

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

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

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

[7] Ralph Jung. 2025. *There is no memory safety without thread safety*. https://www.ralfj.de/blog/2025/07/24/memory-safety.html

[8] Alex Kladov. 2025. *Memory safety is …* https://matklad.github.io/2025/12/30/memory-safety-is.html

[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