Spade – A Modern Hardware Description Language

FRANS SKARMAN, GUSTAV SÖRNÄS, and OSCAR GUSTAFSSON, Linköping University, Sweden

The need for custom hardware to meet compute demands is ever increasing. However, the hardware description languages that these accelerators are primarily built with were designed in the 1980s which means that they are missing out on over 35 years of development in programming language design.

In this paper, we present Spade — a new hardware description language that takes inspiration from modern software language design and aims to be more productive than traditional HDLs without sacrificing performance. This is achieved through a mix of building abstractions for common hardware constructs such as pipelines, and outright borrowing ideas from software, such as a type system that comes close in power to that of Rust or Haskell.

Compared to contemporary hardware description languages such as Chisel, which are embedded in a host language, Spade is a standalone language with a type system that is available in hardware, not just at elaboration-time. Its abstractions also build *on top* of the RTL abstraction instead of replacing it as is done in languages like BlueSpec and in high-level synthesis.

CCS Concepts: • **Hardware → Hardware description languages and compilation**.

Additional Key Words and Phrases: Hardware Description Language

ACM Reference Format:
Frans Skarman, Gustav Sörnäs, and Oscar Gustafsson. 2026. Spade – A Modern Hardware Description Language. *ACM Trans. Reconfig. Technol. Syst.* 1, 1 (January 2026), 28 pages. https://doi.org/10.1145/3793550

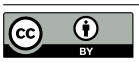
1 Introduction

The software world has seen a steady stream of new programming languages over the past three decades, with 14 out of the 18 most popular languages in common use today being created after 1990 [53]. In the hardware world, however, the vast majority of developers still use VHDL and Verilog which were introduced in the mid 1980s [17]. Modern software development benefits greatly from this continuous innovation, with modern languages significantly reducing the number of bugs in projects [55], and providing significant boosts to developer productivity [9].

In this paper, we present Spade – a Hardware Description Language (HDL) which attempts to bring some of these innovations from software languages into hardware description. Some features are borrowed outright, for example, the strong static type system that many languages employ, and the tooling that boosts developer productivity through helpful error messages, streamlined dependency management, and effective debug tools.

Hardware is of course different from software in many crucial ways: It is inherently parallel, development cycles — especially in ASIC designs — are much less agile, and "performance" is always a tradeoff between area and runtime. In addition, while it is often reasonable to trade some raw performance for developer productivity in software, this is not the case nearly as often in hardware, since maximizing the system performance is the reason for building custom hardware in the first place. For these reasons, many of the features that make modern software languages more

Authors' Contact Information: Frans Skarman, mail@fransskarman.com; Gustav Sörnäs, gustav@sornas.net; Oscar Gustafsson, oscar.gustafsson@liu.se, Linköping University, Linköping, Sweden.



ACM 1936-7414/2026/1-ART
https://doi.org/10.1145/3793550

productive or correct cannot be directly ported to hardware. Despite these mismatches, Turing Award winners Hennessy and Patterson have argued that hardware development must become more like software development to meet the growing need for custom hardware accelerators [22].

When outright borrowing features from software does not work, Spade instead borrows design philosophy. In particular the overarching goal of the language is building abstractions that allow developers to reason about their design at a higher level, while still giving full control over the resulting hardware. Some examples of this include pipelining as a language feature, and a type system that is expressive enough to build new hardware-specific abstractions.

For a language to be truly productive it must also come with appropriate tools. The Spade compiler is built to provide helpful and clear error messages, its build system provides integrated dependency management, and it has editor integration through the language server protocol. For debugging, it provides integration with both cocotb and Verilator, and the Surfer waveform viewer [48] was originally built for the language to enable productive debugging of Spade designs.

The primary goal in the design of Spade is not to invent completely novel features, it is to build a language that is complete and productive for hardware designers today. This is done primarily through incorporating ideas and features both from contemporary software langauges, and hardware language research in a package that works well for day-to-day RTL hardware design.

The rest of the paper is structured as follows. Section 2 goes into more detail about the benefits that modern languages provide, what features of those languages give the observed benefits, and discusses how some of those benefits can be achieved in hardware. The basics of the language are introduced in Section 3, and the native pipelining construct is described in Section 4. The type system is described in detail in Section 5. The tools built for and around the language are discussed in Section 6. The implementation of the compiler is briefly discussed in Section 7, and a comparison with other contemporary HDLs is made in Section 8. Finaly, the target audience and goals of the language are discussed in Section 9 before briefly discussing future work in Section 10.

2 Motivation for a Software-Inspired HDL

Over the past 35 years, the software world has seen a consistent stream of new programming languages providing techniques and ideas to make software developers more productive. In the 2023 StackOverflow developer survey [53], 18 languages are used by more than 5% of respondents and 78% of those languages first appeared after 1990. The situation in the hardware world is very different. The 2022 Wilson Research Group Functional Verification Study [17] included questions on which languages are used for design in FPGAs and ASICs, five languages are named explicitly: VHDL, Verilog, SystemC, SystemVerilog, and C/C++. All of these originate before 1990. Though SystemVerilog and SystemC are languages which appeared in the 2000s, they are supersets of Verilog and C++ and as such retain many fundamental design issues of those languages. The survey mentions one more category, “other”, but that category is used by less than 5% of respondents. This means that the vast majority of hardware developers are missing out on 35 years of innovation in programming language design, apart from incremental improvements to the languages which are typically adopted very slowly by tool vendors.

In software, these new languages have a massive impact on developer productivity and program correctness. For example, a study [9] conducted by Google showed that developers using Rust or Go are twice as productive as developers using C++. In addition, 85% of surveyed developers are more confident in the correctness of their Rust code compared to other languages. Similarly, [18] found that 90% of interviewed developers mentioned that Rust makes them more confident that their production code is bug-free. Another study, [55], found that “first time contributors to Rust projects are about 70 times less likely to introduce vulnerabilities than first-time contributors to C++ projects”.

These massive gains in both productivity and correctness make it very attractive to explore if similar gains can also be achieved in hardware, but to do so we need to understand where the gains come from. Fulton et al. [18] identified several reported benefits of Rust that likely contribute to these results. First there are of course technical benefits: Rust provides memory safety, concurrency safety, immutability by default and no null pointers all of which over 80% of survey respondents reported as benefits. Most survey respondents also reported performance as an important benefit. There are also non-technical reasons listed as important benefits, most importantly that the tools included with the language are good or excellent compared to other languages. A vast majority also found that the compiler's helpful error messages provide a major problem-solving benefit.

2.1 Technical Benefits and Abstraction

The technical benefits of software languages are the hardest to replicate, as many of them come from abstractions designed specifically for the software domain. A prominent example is the ownership and borrowing system that is central to Rust, and is an abstraction around a processor's memory. This abstraction cannot be applied in hardware as processors and memory as viewed through a processor are at a much higher level of abstraction than hardware design. Taking a step back, however, the idea of the ownership and borrowing system is to eliminate a common and costly class of problems by adding an abstraction around the area where the problem arises, namely the memory. This is certainly not exclusive to Rust, all high level languages use abstraction to hide unimportant details and highlight or reframe other details in a way that makes them easier for both the compiler and programmer to reason about. However, which details are important and unimportant in software are often very different from hardware, which means that simply copying the abstraction outright is doomed to fail. Instead, the abstractions in an HDL must be designed specifically for the hardware domain with design inspiration from software.

Static type systems are a key component of most modern languages and many studies have attempted to examine their utility with varying conclusions. There are several studies that attempt to compare developer productivity between languages with dynamic and static typing, e.g. [21, 15] though those studies are plagued by small sample sizes and/or analyzing the effects of types in small exercises rather than large scale projects.

On the other hand, several studies have found benefits to using static types in the maintainability and code quality of open source projects [26, 43] and correctness of projects in statically typed languages. Additionally, it has been shown that 15% of bugs in public JavaScript [19] and Python projects [25] would have been caught by type systems at compile time if the projects used TypeScript in the JavaScript case, or MyPy type signatures in Python. While 15% may appear low at first sight, it is worth remembering that these are the bugs that slipped through all code review and testing getting far enough to be reported as bugs.

2.2 Performance

There are several ways to achieve the safety benefits that Rust provides, but many of them come with a significant runtime cost. The unique benefit that made Rust stand out is that it relies heavily on using *zero cost abstractions*, i.e. abstractions that have no runtime performance overhead. This point is perhaps even more important in hardware than it is in software as the reason hardware is built in the first place is to achieve better performance than software can provide. Again, the exact zero cost abstractions that a language like Rust uses is in general not possible to port to hardware, instead, this design philosophy should be a guiding principle in the design of new hardware focused abstractions.

2.3 Tooling

Unlike the previous areas, the tooling that makes modern software languages successful can pretty easily be ported to hardware. Modern software build tools provide everything needed to build and run a project. They allow developers to specify dependencies to include in the build, to download and manage those dependencies, and run the tools required to build and run the project. A hardware build tool can do the same things, the only difference being which build tools it calls. Similarly, the rust compiler's error messages are often praised as being very helpful, and this property can also be replicated in an HDL compiler.

2.4 Hardware is Not Software

While borrowing features from software in order to make hardware design more productive is an underlying design philosophy in Spade it is of course worth noting that hardware and software are vastly different domains, and that *blindly* copying design decisions from software is most likely going to end poorly.

First, the resulting physical chip is vastly different from a sequence of processor instructions. Second, "performance" in software is generally easier to reason about. The most important metric is almost always runtime with memory usage being a secondary, usually less important, concern followed further by code size. There is also usually either zero or positive interplay between these metrics; a reduction in memory usage means less allocations, which in turn results in a faster program. Hardware, on the other hand, has a fundamental tradeoff between space and time. A hardware designer can often trade an increase in chip area for a reduction in runtime. In addition, the third metric of power consumption is often as important as runtime and area. For this reason, simply having a "fast" language is not sufficient for efficient hardware design, the language must enable convenient design space exploration in order to allow designers to make appropriate tradeoffs between the metrics.

3 The Spade Language

With the background out of the way, we will start introducing the Spade language, starting with an overview of the basic syntax and semantics. Listing 1 shows Spade code that describes a circuit which blinks an LED at a configurable interval. The first line defines the interface of the *unit*. It is an `entity` called `blink` which takes three inputs: `clock`, `rst` and `max`, and it returns a `bool`. Units are the basic building blocks of circuits in Spade and come in three flavors: entity, function, and pipeline. Functions, denoted `fn`, are the most restrictive, as they only allow combinational logic[1]. Pipelines, denoted `pipeline` can have registers and are used for describing circuits with a pipeline-like structure. They will be discussed in more detail in Section 4. Finally, entities can contain sequential logic and do not enforce a specific structure, making them the most general unit. For this `blink` example, we need a register to store a counter value, and we do not have a need for pipelining, which is why it is defined as an entity. Functions, pipelines and entities are instantiated differently which shows, at the call site, whether an instance is sequential or just combinatinoanl. This is analogous to the distinction between functions and methods in BlueSpec [39].

Registers in Spade are a dedicated language construct, rather than something that is inferred by synthesis tool. This makes them more compact, clearly communicates design intent, and eliminates pitfalls[2]. On Line 2, a register called `counter` is defined. It is clocked on the positive flank of the `clk` signal and reset back to `0` synchronously if the `rst` input is `true`. The value of the register is

[1]They are pure software terms
[2]https://blog.award-winning.me/2017/11/resetting-reset-handling.html

```
1 entity blink(clk: clock, rst: bool, max: uint<20>) -> bool {
2     reg(clk) counter reset(rst: 0) =
3         if counter == max {
4             0
5         } else {
6             trunc(counter + 1)
7         };
8
9     counter > (max / 2)
10 }
```

Listing 1. Spade code which blinks an LED.

specified on Lines 3–7 and showcases some important properties of the language. First, Spade is expression based and has only immutable variables. Rather than conditionally assigning the value of `counter` in each branch of an `if`-*statement*, the `if`-*expression* returns a value which is assigned to counter. As `if`-conditionals get compiled to multiplexers which select among several values, the expression based approach is closer to the resulting hardware. In addition, this style makes it much harder to accidentally create latches as it is a compilation error to forget to specify an output value in each branch.

Spade is a statically typed language with type inference. This means that all values in the language have a type that is known at compile time, but that the compiler can infer the type of most values. For example, the value of `counter` is never specified by the user, but because it is being compared with `max` whose type is `uint<20>`, i.e. a 20-bit unsigned integer, the compiler can infer that `counter` must also be a 20-bit unsigned integer. Spade also prevents throwing away bits implicitly. For example, since `counter + 1` can overflow and require one more bit than `counter`, its type is `uint<21>`. However, in this case, the result of the addition must be assigned to `counter` which throws away information. This truncation must be done explicitly using the `trunc` function. Finally, unlike languages like Haskell, Spade does not do whole-program type inference which is why all the types of the unit are specified in its head. The type system is discussed in more detail in Section 5.

The final expression in any block is the output of that block. This is true both for the output (return value) of the unit specified on Line 9, as well as the result of each if branch on Line 4 and Line 6.

One final thing to note in this example is that Spade code is more “linear” than conventional HDLs. Units take a list of inputs and produce a single output (though that output may consist of multiple values). The code is also read top-to-bottom with variables only being visible below their definition, which is closer to modern software languages. In some cases, it is of course desirable to have registers depend on each other, but this requires an explicit pre-declaration of the variable using the `decl` keyword. For whole units which do not fit very well with this linear flow, such as memories, Spade has special types called ports which are discussed in more detail in Section 5.7.

4 Pipelines

Pipelining is an important construct in most hardware designs, it allows designs to maintain a high clock frequency and throughput at the cost of latency. However, despite their importance, most HDLs require the user to manually build their pipelines, a process that is both tedious and error-prone as one must make sure that computations are performed on values corresponding to

```
1 pipeline(2) X(clk: clock, a: int<32>, b: int<32>) -> int<33> {
2     let x = inst(1) g(clk, a);
3     let product = a*b;
4 reg;
5     let sum = x + f(a, product)
6 reg;
7     sum
8 }
9
10 pipeline(1) g(..) -> int<32> {..}
11 fn f(...) -> int<32> {...}
```

Listing 2. Example of the pipelining construct in Spade.

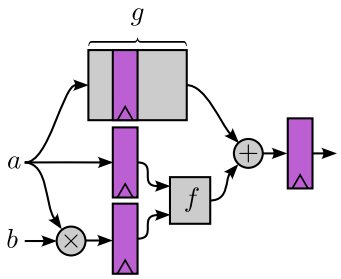


Fig. 1. Hardware described by pipeline X by the code in Listing 2.

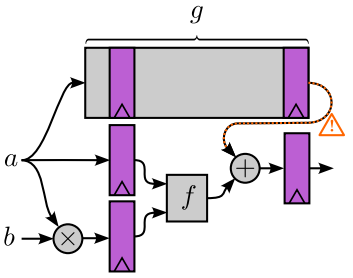


Fig. 2. Hardware described by pipeline X by the code in Listing 2 after changing the depth of g to 2.

the correct time step. In some cases, designers use patterns on their variable names, and ad-hoc static checking tools to verify that the pipelining is correct [31]. In Spade, the language natively supports describing pipelines, and the compiler ensures that those pipelines are used correctly.

To exemplify this, Listing 2 contains Spade code for describing the pipeline shown in Fig. 1. On Line 1, the external interface of the pipeline is defined. On pipelines, this interface includes its depth which is effectively the latency between the pipeline inputs and outputs. In this case, the depth is 2. Including this depth in the external interface means that both a user and the compiler can understand the timing behavior of pipelines without reading the body; they know that outputs will arrive `depth` cycles after the inputs. Inside the pipeline, the `reg` statements tell the compiler where to insert pipeline stages, registering all the variables visible above it. For example, after the `reg` statements on Line 4, any use of the variables above it (in this case `a`, `b`, `x`, and `product`) will refer to the registered copies. Currently, pipeline registers do not support resets and the clock used for the pipeline registers is the first argument to the pipeline.

Since the compiler is aware of the latency of pipelines, it correctly handles nested pipelines. For example, the variable `x` is not registered in the first stage, as it is the output of another pipeline with a latency of 1. The compiler also ensures that pipelined results are not used before they are ready. For example, if the user modifies the inner `g` pipeline to contain an extra stage as is shown in Fig. 2, then the outer pipeline, `X`, will no longer behave as expected. This issue is caught by the compiler which will emit the error message shown in Listing 3. The fix is simple, the user must

```
error: Use of x before it is ready
  ┌─ src/main.spade:5:15
  │
2 │     let x = inst(3) g(clk, a);
  │         - x is defined here at stage 0 with a latency of 3
  .
5 │     let sum = x + f(a, product);
  │               ^
  │               │
  │               x is unavailable for another 2 stages
  │               This is stage 1
  │
= note: Requesting x at stage 1
= note: But it will not be available until stage 3
= help: Consider adding more reg; statements between the definition and use of x
```

Listing 3. Error emitted by the Spade compiler when describing the hardware in Fig. 2 without explicitly referring to another stage.

```
1     ...
2     let x = inst(2) g(clk, a) ;
3     let product = trunc(a*b);
4   reg;
5     let sum = stage(x_output).x + f(a, product);
6   reg;
7     'x_output
8     ...
```

Listing 4. Example of a stage reference used to access the original x signal without going through pipeline registers.

either revert the change to g, or insert an additional stage on Line 4 and change the depth of X to 3. Regardless of what fix is applied, once the code compiles again, the behavior of the pipeline will be the same as it was originally (apart from latency) which enables purely mechanical retiming of circuits without worrying about the behavior of the circuit changing.

4.1 Stage References

In most cases, the previous example prevented a logic bug in the circuit by requiring that outputs from pipelines are only accessed in the stage they belong. However, in some cases it is desirable to describe hardware which is not a pure pipeline. The user may in fact want to describe the hardware shown in Fig. 2 or, perhaps more likely, they want to describe a pipeline with internal state such as an accumulator or a processor pipeline register file with data forwarding. In these cases, Spade supports *stage references*, an example of which is shown in Listing 4, where the code in Listing 2 has been modified to describe the hardware in Fig. 2. On Line 7, the stage at which the output of g is availble is given the name x_output, then the value of x is fetched from that stage on Line 5. Stages can also be referenced by their offset from the current stage. For example, the data forwarding logic in a pipelined processor can be written as Listing 5 where `stage(+x)` refers to variables in the stage x cycles *ahead* of the current stage.

```
1   let reg_out = inst regfile(srca, srcb, stage(writeback).writeback)
2 reg;
3   let opa = if stage(+1).dest == srca {stage(+1).alu_out}
4        else if stage(+2).dest == srca {stage(+2).alu_out}
5        else                           {reg_out.a};
6
7   let opb = if stage(+1).dest == srcb {stage(+1).alu_out}
8        else if stage(+2).dest == srcb {stage(+2).alu_out}
9        else                           {reg_out.b};
10
11  let alu_out = alu(insn, opa, opb);
12 reg;
13  // Memory reads are done here
14 reg;
15  'writeback
16  let writeback = (alu_out, insn.dest)
```

Listing 5. Example of how the data path of a processor with data forwarding can be written in Spade.

4.2 Dynamic Behavior

As designs grow large, it is very common to require latency insensitive interfaces, for example, to allow reading or writing data to external memories. The pipelines discussed so far are ill-suited for this task as they will always consume and produce values. For this reason, Spade allows specifying a condition for a stage to accept inputs as `reg[condition]`. Every clock cycle where `condition` is false, the stage is *disabled* and the registers will retain their previous value rather than updating to the value from the preceding stage. This effectively stalls the pipeline. The condition propagates *upwards*, so any stage before a disabled stage will also be stalled.

For interactions with the outside world, it is important to know if a stage is ready to accept inputs or if the contents of a stage is valid. This information can be read from the special expressions "`stage.ready`" and "`stage.valid`".

To exemplify this, Listing 6 shows how this can be used to perform stalls in a pipelined CPU. As there is cross dependencies between stages, the code is easiest to understand by first discussing the `reg` statement on Line 11. It disables the stage if the instructions in the decode- or execute stages are jumps, or if the decode stage contains a load instruction. When the pipeline is stalled, the stateful parts of the processor must be modified to accommodate this, which is done in two places in this example. The program counter logic on Line 1 uses the `stage.ready` expression is used to keep the previous program counter value during clock cycles where the pipeline is stalled. Downstream of the stall, on Lines 16–17, `stage.valid` is used to disable writes to the register file and prevent jumps during clock cycles where the upstream pipeline was stalled.

It is worth noting that the current implementation does not automatically propagate stall conditions to sub-pipelines, which means that developers must manually ensure that nested pipelines with stalls are correct[3].

5 Type System

A type system is a central component of any programming language and can serve many purposes. The most basic purpose is to dictate the storage requirements for values. This is of course *required*

[3]This issue is being tracked in https://gitlab.com/spade-lang/spade/-/issues/279

```
  reg(clk) pc = match (stage.ready, stage(mem).jump_destination) {
    (false, _) => pc
    (true, Some(dest)) => dest,
    (true, None) => trunc(pc+1)
  }
reg;
  ...
  let stall = stage(decode).is_jmp
            || stage(execute).is_jmp
            || stage(decode).is_load;
reg[rst || !stall];
  'decode
  ...
reg;
  'execute
  let regfile_we = stage.valid && should_write_insn_result(op);
  let jump_destination = if stage.valid && is_jmp && jump_taken {
      Some(int_to_uint(jump_target))
    } else {
      None()
    };
reg;
  'mem
  ...
reg[memory.valid];
```

Listing 6. Excerpt from a pipelined RISC-V implementation which showcases the use of register conditions, as well as `stage.ready` and `stage.valid`.

in an HDL in order to allocate bit widths to operators and registers. The type system in Verilog pre-SystemVerilog does only this and nothing more. Type systems can also be used to limit which operations are possible to perform on values. For example, VHDL does not allow arithmetic on arbitrary bit vectors (`std_logic_vector`), for that the values must be converted to a signed or unsigned type. Type systems can also group related values together, often in records or structs and sometimes as unions. Grouping and restricting the available operations on types prevents bugs at compile time by disallowing misinterpretation of values, and disallowing creation of invalid values. Type systems can also be used to facilitate code re-use. Generic types and modules allow code to be written once and reused for any compatible type, and type systems which support higher order functions allows the use of techniques like combinators [20] to re-use code for working with streams of data. Finally, type systems can be used to facilitate correct-by-construction interoperability between modules, in the simplest case by the techniques already discussed to prevent incorrect values from being created and passed between modules, but also by more advanced techniques such as Filaments timeline types [35] which encode signal timing requirements of modules in the type system.

As mentioned earlier, there is clear evidence that type systems prevent bugs [19, 25] but it of course also comes with a cost. A type system that is strict about its interpretation of values will be more verbose since it requires explicit conversions when a less strict language may have performed implicit conversions instead. Spade takes the stance that in hardware, where the cost of bugs is high, a powerful type system can be of great help for preventing issues. Therefore, Spade

has a type system that is more powerful than most contemporary HDLs. In order to address the verbosity concerns, the language uses type inference to infer most types which gives some of the advantages of dynamic types, namely not having to state every type, while providing all the correctness guarantees that come with static types.

Like almost all type systems it supports primitive types such as booleans, signed and unsigned integers, and compound types such as tuples, arrays, and structs. In addition to this, it also supports *sum types*[4] and the pattern matching that makes these ergonomic to work with. All these types can be generic for code re-use, and unlike generic types in VHDL, the generic parameters can be other types in addition to compile-time integers. In addition to types which control the interpretation of values, there is also support for higher order functions to facilitate, among other things, transformation of values wrapped in other types. For composition and encapsulation, the language supports implementing methods on types. These methods can also be parts of traits which allows generic units to place requirements on the types that are accepted by the generic unit. All these features put the power of the Spade type system close to that of Rust and above most contemporary HDLs[5]. There are also a few HDLs with similarly powerful type systems, e.g., Clash [3] and Bluespec [39].

In addition to these heavily software-inspired type system features, the language has some hardware specific type features as well. First, it models *ports* as a collection of directional wires which allows, for example, passing the interface ports of a memory or an AXI bus as a whole between units. A linear type system ensures that this is done correctly and every wire gets driven exactly once. Finally, the language includes type level arithmetic which is used, among other things, to prevent implicit data loss due to truncation, and tracking things like fixed point numbers. These type level integers can also be used in pipelines to build generic pipelines whose latency depends on the number of bits in their inputs or outputs.

The rest of this section goes into more details on the Spade type system.

5.1 Sum Types and Pattern Matching

The language supports *sum types* inspired by languages like Rust, Haskell and ML and which in Spade are called `enum`. Unlike their C or VHDL namesake, enums in Spade have data associated with them in addition to being one of a set of variants.

A common use for sum types in hardware is to represent states in state machines, which is exemplified in Listing 7. It contains an implementation of an addressable configuration register with a 16-bit address and 8-bit value. The external interface of the register is defined on Line 9: the first input, `self_addr`, is the address of this register, while the second input, `data`, is a tuple `(bool, uint<8>)` which represents a stream of bytes that control the register. When the `bool` is true, the byte is valid. A command consists of a sequence of three bytes, the first two containing the address to write data to, and the third byte containing the data to write. To decode this, a Finite State Machine (FSM) is used, and the `enum` type that represents the state is defined on Line 1. It can take on one of three values: `Idle`, `WaitAddr1` and `WaitData`. As mentioned, enums in Spade can have data associated with each variant, in this case `WaitAddr1` has the first byte of the address that has been received and `WaitData` has both the first and second addresses.

The main state machine itself is defined on Lines 10–16. This makes heavy use of the `match` expression and its built in pattern matching. The first branch is a catch-all for when the byte-valid bit is not set in which case the FSM should simply retain its current state. The second branch handles the idle state in which the next byte to arrive is the first byte of a target address. This target address is stored in the payload of the next state, `WaitAddr1`. The next branch which handles the `WaitAddr1`

[4]Sometimes called tagged unions

[5]Of course excluding the ownership system which is an abstraction around processor memory.

```
1  enum State {
2    Idle,
3    WaitAddr1{addr0: uint<8>},
4    WaitData{addr0: uint<8>, addr1: uint<8>},
5  }
6
7  use State as S;
8
9  entity creg(self_addr: uint<16>, data: (bool, uint<8>)) -> uint<8>  {
10   reg(clk) state reset(rst: S::Idle) =>
11     match (data, state) {
12       (_,                   (false, _   )) => state
13       (S::Idle,             (_    , byte)) => S::WaitAddr1(byte),
14       (S::WaitAddr1(addr0), (_    , byte)) => S::WaitData(addr0, byte),
15       (S::WaitData(_, _),   (_    , _   )) => S::Idle,
16     };
17
18   reg(clk) value reset(rst: 0) =
19     match (state, data) {
20       (State::WaitData(addr0, addr1), (true, data)) => {
21         if addr0 `concat` addr1 == self_addr {
22           data
23         } else {
24           value
25         }
26       }
27       _ => value
28     };
29   value
30 }
```

Listing 7. Implementation of an addressable configuration register implemented using sum types. The clock and reset inputs have been omitted in the interest of space.

state is similar, but it also reads the `addr0` payload of the current state when transitioning to the `WaitData` state. Finally, the last state simply jumps back to idle when the data has been read.

The `match` block is the only way to access `enum` members, as it ensures that value has the correct variant before accessing the underlying fields. Match blocks are not only used to access enum variants, however. The match block here matches on a tuple consisting of the state and the input, which in turn is another tuple. In addition, on Line 12, it is used to match on a boolean value to prevent the state machine from progressing when the input is not valid. This also makes use of wildcards (_) to ignore the rest of the matched value. The first branch of a `match` block which matches a value is prioritized, which is why wildcards are used for the valid signal after the first branch. Finally, the compiler ensures that patterns are exhaustive, i.e. every value matches at least one branch. This makes refactoring easier as the compiler forces developers to address all match blocks that are affected when new enum variants are added.

Another match block is used on Lines 18–28 to update the value of the register to the incoming data if the current state is `Waitdata` *and* the incoming data is valid.

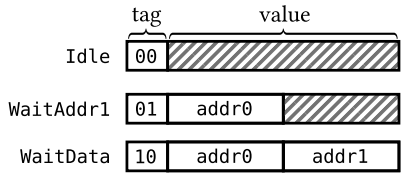


Fig. 3. Representation of the State enum defined in Listing 7.

Using sum types and pattern matching for this application has several advantages. First, associating the payload with individual states means that there is no way to read "invalid payload", the compiler ensures that the data is only accessed when valid. The compiler also ensures that all cases are covered by the match blocks which reduces the risk of errors. The strict type system also ensures that the second match block, which updates the `value` register, is correct-by-construction; the only state in which a valid 16-bit address exists is the final state, in any other state there is no value available for `addr0` *and* `addr1`.

There are several potential ways to represent enums in hardware. Spade does not specify an exact representation in order to allow optimizations down the line, but the representation that is currently used is shown in Fig. 3. The value consists of a tag which discriminates the variants, and the "payload" of each variant is packed from left to right. There are several avenues for optimization here, a one-hot encoding of the tag can be beneficial for decoding but requires additional bits for storage. In software, unused bytes of some variants is sometimes used to encode parts of the tag which can reduce storage requirements, but results in more complex decode logic in hardware. Field ordering can also have an impact on performance [32].

5.2 Generics

Both types and units in Spade can be generic over other types which is denoted by `<T>` where `T` is the name of the generic parameter. This is useful for creating complex "container types" such as a ready-valid interface where the inner type can be any other type. Generics can also be integers denoted by `<#uint N>` or `<#int N>` which is used frequently for things like integer widths, array sizes, or pipeline depths that depend on the type of the pipeline input or output. The implementation of generics in the compiler is a mix C++ style templates where final type checking is only done once all types are fully known, and true generics where type errors are caught through traits, which will be discussed later. Where possible, full generics give early feedback while developing and prevents unexpected issues that only occur with some combinations of types while templates allow more complex type expressions to be evaluated non-symbolically for difficult cases such as type level arithmetic.

5.3 The Option Type

A special case of the `enums` and generic types discussed so far is the `Option` type which has wide-reaching utility in hardware. Intuitively, the Option type is best viewed as a (valid, data)-pair where the language and compiler help prevent misuse of the underlying data. To motivate it, consider the `data` input in the entity from Listing 7 which is also a (valid, data)-pair. In this case, the `bool` field represents the validity of the `uint<8>` value, but this relationship between the fields is not captured in the type system, which means that it is possible to read the invalid data if one forgets to check the valid flag. In addition, there are other possible interpretations of a `(bool, uint<8>)`-tuple which could be passed to the unit accidentally.

```
enum Option<T> {
  None,
  Some{val: T}
}
```

Listing 8. The definition of the `Option` type.

The `Option` type solves both of these problems and its definition in the Spade standard library is shown in Listing 8. First, it being an `enum` means that it is not possible to read the contained value without a `match` block, preventing invalid values from being accessed accidentally. While the bit representation of `(bool, uint<8>)` and `Option<uint<8>>` are the same, the semantics of the `Option` type are clear and enforced by the compiler, reducing the risk of passing an incorrect value to a unit which expects data with a valid tag. Since it is defined with a generic parameter `T` it can wrap any other type, and other types which build on top of valid data such as a ready/valid can re-use the option type for correctness.

Defining a canonical representation for combined data and valid signals in the standard library comes with additional advantages. First, it is possible for the compiler to perform optimizations that rely on the known relationship between the data and its valid signal. One such optimization is automated fine-grained clock gating, where switching power can be reduced by disabling registers during clock cycles where the contained data is invalid. Another advantage is in composability, the `creg` unit defined previously when rewritten to use an `Option` type can be connected to *any* source of `Option` type values whether it is data coming from a serial bus like UART, a network controller, an AXI bus, or anything else which produces a stream of validated bytes.

5.4 Methods

Almost all modern software languages include features akin to *methods* – functions that are associated with a type and can be *called* on an instance of that type instead of as freestanding functions. Methods are a central component of object-oriented languages such as Java and C++ but are also present and used extensively in languages based on other paradigms such as Rust, JavaScript, and OCaml.

Methods have several advantages over pure freestanding functions, some of which are demonstrated by Listing 9 which showcases how a stream of bytes in the CSI2 protocol can be transformed into a stream of pixel data. Listing 9a shows the definition of two types of data to be streamed, Listing 9b shows how a byte stream is converted to pixels using methods, and Listing 9c shows the corresponding implementation using free-standing functions. Using methods in this case makes the code readable from top-to-bottom which is well suited for a gradual transformation of the streams, whereas the function-based implementation must be read inside-out and requires more effort to find where the stream starts, and which arguments belong to which function. Another advantage of methods is that similar methods can exist on multiple types, making refactoring to change between different underlying types much easier. It is also worth noting that even the function based approach here is significantly more compact than the equivalent code would be in Verilog or VHDL where the output of one module or entity cannot be passed directly to another, instead requiring definition of intermediate signals which pass the values between units.

The `inst` keyword differentiates the instantiation of function-methods from entity-methods. This warns readers of the code that while the code looks like a pure transformation, there is stateful logic behind some of the methods. In this case, that stateful logic looks at things like packet headers in the stream of bytes and transforms it into a sparse stream of packets.

```
struct ByteStream {
  inner: Option<uint<8>>
}

struct PixelDataStream {
  rgb: Option<uint<8>>
}
```

(a) Definition of two stream types.

```
let pixels = byte_stream
  .inst into_packets(clk)
  .long_packets()
  .filter_header(0x2A)
  .inst into_pixels(
    clk, byte_stream
  );
```

(b) Transformation from `ByteStream` to `PixelDataStream` using methods.

```
use lib::packet::bytes_to_packets;
use lib::packet::to_pixel_stream;
use lib::packet::filter_header;
use lib::packet::long_packets;

let pixels = inst to_pixel_stream(
    clk,
    filter_header(
      long_packets(
        inst bytes_to_packets(
          clk,
          byte_stream
        )
      ),
      0x2A
    )
  );
```

(c) Transformation from `ByteStream` to `PixelDataStream` using free-standing functions.

Listing 9. Example of using free-standing functions and methods.

5.5 Traits and Requirements

Generic types and units must sometimes enforce requirements on the types they are generic over. This is accomplished using `traits` and `where` clauses. For example, one can write a generic unit which computes the sum of a stream of values that not only works for a specific type, but any type which can be "added". An example of this is shown in Listing 10. On Line 2 a trait called `Addable` is defined. In order to satisfy the `Addable` trait, a type must implement every method in the trait body, in this case just `add`. The trait is implemented for integers of any size `N` on Lines 7–11. Finally, an entity called `sum` is defined starting on Line 15. This entity is generic over a type T which, via the where clause on Line 16, is restricted to only be a type which implements the `Addable` trait.

5.6 Higher Order Circuits

Functional programming has been shown on several occasions to be a good fit for hardware design [20, 3], and a central enabling feature of functional programming is being able to pass functions as values, which Spade also supports. Dynamic dispatch is of course difficult or impossible to do without overhead in hardware, so only static dispatch is supported. Units which accept higher order circuits are generic over an `Fn`, or `Entity`, or `Pipeline` trait which has a `call` method that receives the relevant arguments and produces the corresponding output.

Higher order circuits are especially useful on containers such as arrays and option types as they can be used to transform the contents of the container in a concise and correct manner. An example of this is shown in Listing 11 which implements an FIR filter on a stream of samples represented by an `Option<Sample>` type. The sliding window function retains the previous 16 values and produces a new value of type `Option<[Sample; 16]>` for every input sample. The outer `map` function transforms this `window` of values into an output sample by pairing the values (`zip`) with

```
// Definition of a trait for types whose values can be added
trait Addable {
  fn add(self, other: Self) -> Self;
}

// Implementation of the trait for signed integers of any width
impl<#uint N> Addable for int<N> {
  fn add(self, other: int<N>) -> int<N> {
    trunc(self + other)
  }
}

// An entity which can sum values of any type which implements
// the Addable trait.
entity sum<T>(clk: clock, rst: bool, reset_value: T, in: T) -> T
  where T: Addable
{
  reg(clk) sum reset(rst: reset_value) = sum.add(in);
  sum
}
```

Listing 10. Definition, implementation, and use of an `Addable` trait that indicates that values of the types which implement the trait can be added.

```
samples
  .inst sliding_window::<16>()
  .map(fn (window) {
    window
      .zip(coefficients)
      .map(fn (x, c) { x * c })
      .sum()
  })
```

Listing 11. FIR filter operating on a stream of samples encapsulated in `Option<T>`.

the corresponding coefficients producing a `[(Sample, Coefficient); 16]` array, multiplying each pair, and finally summing the products.

5.7 Linear Types and Ports

So far, most of the features described have largely focused around describing computations; pipelines and combinators describe sequences of computation, and enums and match statements allow describing state machines. However, of equal importance when designing hardware is the interconnection between different compute modules as well as the internal and external memories they need to perform their tasks. The description of interconnections using the language described so far is more awkward than traditional HDLs as units take a set of inputs and produce a set of outputs, rather than operating on a set input and output wires. To exemplify this problem, consider the following definition of a compute unit which requires access to a memory for its computation:

```
entity compute(clk: clock, mem_in: MemData) -> (Out, uint<16>)
```

```
1 struct port MemReadPort<T> {
2   addr: &inv uint<16>
3   value: &T
4 }
5
6 entity compute(clk: clock, mem: MemReadPort<MemData>) -> Out {...}
```

Listing 12. A compute unit which accepts a single memory port

```
1 entity dp_mem<T>(clk: clock) -> (MemPort<T>, MemPort<T>) {...}
2
3 entity top() {
4   let (p1, p2) = inst dp_mem();
5
6   (inst compute(p1), inst compute(p1))
7 }
```

Listing 13. Definition of a dual port memory, along with a top module that instantiates the memory and two compute units. The first memory port (`p1`) is accidentally passed to both compute units.

Its output value is a tuple of the output of the module (`Out`), and the address it reads from. It has a single input, `mem_in`, which is the data that was read from memory. This has a number of issues: first, the address is now mixed in with the computed result, and second, there is no logical connection between the address and the returned value, a problem which is compounded if the compute unit needs access to more than one memory or when the unit needs both read and write access. Finally, if the compute unit is a pipeline, the read address and the memory output value will be pipelined, which is generally not desired.

In order to work around all these issues Spade includes a family of types called *port*. The simplest ports are the wire (`&T`) and inverted wire (`&inv T`). A wire is simply a value which is not pipelined between pipeline stages, while an inverted wire reverses the usual direction of data flow. A unit which accepts an inverted wire as an input can set the value of that wire using a `set` statement, whereas a unit which returns an inverted wire can read its value when set by another unit. Using these, the compute unit can be rewritten as

```
entity compute(clk: clock, addr: &inv uint<16>, mem_in: &MemData) -> Out
```

which resolves the problem of memory addresses being mixed with the output *value*, and since wires are not pipelined, will not cause issues with additional delay introduced by pipelines. However, there is still no clear grouping of all the wires belonging to a memory, a problem which may seem small here, but is made larger when dealing with complex buses such as AXI which has more than 10 related wires per bus. For this reason, Spade supports compound ports either as tuples of wires , `(&inv uint<16>, &MemData)`, or as structs as shown in Listing 12.

An important property of ports that must be upheld is that inverted wires are written to exactly once. If no writers are connected, then the circuit will have undefined values which leads to bugs. Similarly, if multiple drivers are connected to the same wire, the value on the wire becomes undefined. Even in a simple case, such as the one shown in Listing 13, it is easy to accidentally violate this requirement. Here, the user instantiated a dual port memory on Line 4, intending to hand out the two ports to separate compute units. However, when instantiating the compute units on Line 6, a mistake was made and `p1` was passed to both compute units. This means that the

```
struct port MemPort<T> {
  addr: inv& uint<16>,
  value: inv& T
}
```

```
let mem: MemPort = memory();
let addr = mem.addr;
set addr = value;
consume(mem);
```

Listing 14. Struct definition and example code for Fig. 4

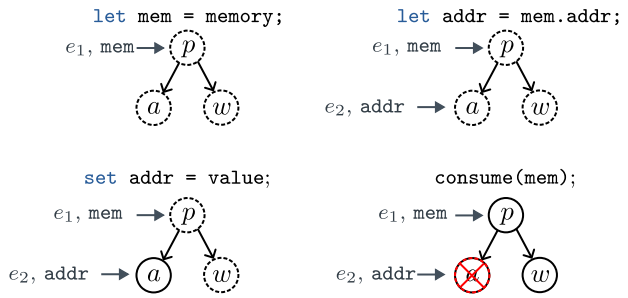


Fig. 4. The linear type checking tree when run on Listing 14

address line on `p1` has multiple drivers while `p2` has none. Spade uses linear types [58] to ensure at compile time that all inverted wires are consumed exactly once. An inverted wire is consumed either when it is given a value, or when it is passed along to another unit who then receives the responsibility of assigning a value. Reading from an inverted wire does not consume it.

Linear type checking happens after normal type checking, meaning that the compiler knows which resources are of linear type and must be checked.

To exemplify the linear type checking algorithm, Fig. 4 shows it in action as it processes each line in Listing 14. In the figure, p represents the whole `MemoryPort` struct, a represents the address field, and w the value field. The e_x variables represent anonymous names given to sub-expressions before they are bound to variables. A dashed node represents it not being consumed, a solid node means it is consumed, and a crossed node indicates double consumption.

For each expression which produces a resource of linear type, a tree is created where leaf nodes represent primitive linear types, and non-leaves represent compound linear types such as tuples or structs. Any statement that aliases a resource, such as a `let`-binding of an expression, or field access on a struct, creates a reference from the alias to the corresponding tree and node.

When a resource is consumed for example by being passed to another module or being **`set`**, the nodes corresponding to the consumed object are marked as consumed. If the node or its child nodes are already marked as consumed, the resource is used more than once and an error is thrown.

This ensures that nodes are not used more than once, but does not guarantee that they are used exactly once. Therefore, at the end of the process, a final pass goes over all the trees to ensure that each node is consumed. If the traversal finds an unconsumed leaf, it represents a resource of linear type which was not set, and an error is reported.

The use of substructural type systems such as linear types to model hardware is not entirely new. In particular, Dahlia [38] uses affine types to model memory ports and users in a high level synthesis context.

```
struct port Rv<T> {
  data: &Option<T>,
  ready: inv &bool,
}

impl<T1> Rv<T1> {
  fn map<T2, F>(self, f: F) -> Rv<T2>
    where F: Fn((T1), T2)
  {
    Rv$(
      ready: self.ready,
      data: match self.data {
        Some(inner) => Some(f.call((inner, ))),
        None => None
      },
    )
  }
}
```

Listing 15. Definition of a ready-valid interface along with an implementation of a `map` combinator for transforming the inner data.

5.8 Latency-Insensitive Combinators

ShakeFlow [20] is a functional HDL centered around combinators for describing latency insensitive interfaces. The authors show that this can be used to describe realistic circuits without performance or resource overhead, and with significant reduction in lines of code. However, they also note that existing HDLs are not able to support latency insensitive combinators as that requires combinators that operate on bidirectional data types where data and valid signals flow "forward" and ready signals flow "backward". ShakeFlow does support this, but does not support other abstraction making its use as a general purpose HDL difficult.

The type system features discussed so far are sufficient to reimplement ShakeFlow inside Spade which enables taking advantage of the benefits of latency-insensitive combinators and mixing them with other abstractions such as pure RTL or pipelines. As an example of this, Listing 15 contains the definition of a ready-valid (`Rv`) type which corresponds to the `VrCh` type in ShakeFlow. The type has two fields: the `data` which is an `Option` type representing the `valid` signal and the validated payload, and the `ready` field which is the ready signal that is propagated "backwards". The `impl` block defines a `map` function, which transforms the inner data according to a function `F` while retaining validity and propagating the `ready` signal.

6 Tooling

As discussed in Section 2, a big reason that modern software languages lead to a productivity boost is through their tooling. Therefore, Spade also includes several tools including a build tool, editor integration and a custom built waveform viewer.

6.1 Compiler Errors

The primary method for interacting with a language is through the compiler, which broadly has two jobs. First, it must of course transform the input language to its output, a process which is largely transparent to the user, and second, it must catch and report any errors that arise. The

```
error: Cannot apply `#[no_mangle(all)]`
   ┌─ error test.spade:2:32
   │
2  │ entity external(clk: clock) -> uint<8> {
   │                                ^^^^^^^ Output types are always mangled
   │
   = Consider replacing the output with an inverted input
   │
2  │ entity external(clk: clock, out: inv &uint<8>) -> uint<8> {
   │                             ++++++++++++++++++  ----------
   = ...and `set`ing the inverted input to the return value
   │
3  │     set out = 5;
   │     ++++++++++ +
```

Listing 16. Example of a Spade error message.

Spade compiler, like many modern compilers, is built to not only *report* errors, but to do so in a way that accurately explains what the problem is, and where possible, guides the user to a solution for the problem. This includes providing code suggestions where possible. Listing 3 showed an example of an error message used to indicate an error in a pipeline, and Listing 16 shows an error which makes heavy use of suggestions to guide the user to a solution. Code suggestions are primarily useful to developers who are new to the language, as well as developers who are new to hardware. With editor integration, they can also serve as the basis for "code actions", allowing the user to automatically apply suggestions in their editor. A survey of questions asked on question and answers forums found that questions about error messages are over-represented in HDLs compared to general purpose programming languages, indicating a real-world need for improved error messages [61].

6.2 Swim - The Spade Build Tool

The most important tool apart from the compiler is the build tool which orchestrates the building of Spade projects. A project consists of a number of Spade files, along with a project configuration file written in `TOML`, an example of which is shown in Listing 17a. This configuration specifies among other things the project dependencies, the backend tool to use, and which raw Verilog files should be included in the build. Swim then manages the namespacing of project files, downloading and versioning of dependencies, and running the backend tools allowing users to build a project from scratch and upload it to an FPGA board with just a single `swim upload` command.

The namespacing and modules system in Spade ensures that there are no name collisions between any dependencies. As an example, consider the project configuration in Listing 17a and project structure in Listing 17b. Since the name of the project is `example` everything defined in the project will be included in the `example` namespace. In that namespace, each file in `src` will be included in a sub-namespace, for example, a function `func` defined in `display.spade` will be reachable as `example::display::func`. Similarly, subdirectories in `src` are put in an additional namespace, so `example::stubs::ecp5::stub` refers to `stub` defined in `src/stubs/ecp5.spade`. Finally, dependencies are included in separate namespaces at the root so for example, `hdmi_driver` defined in a `lib.spade` file in the `hdmi` dependency will be reachable via `hdmi::lib::hdmi_driver`.

```
name = "example"
[libraries]
hdmi = {
  git = "https://gitlab.com/TheZoq2/hdmi"
}
[synthesis]
command = "synth_ecp5"
top = "example::main::main"
extra_verilogs = [ "src/stubs/ecp5.sv" ]
[pnr]
...
```

(a) Swim project configuration.

```
example_project
├─ src
│  ├─ stubs
│  │  ├─ ecp5.spade
│  │  └─ ecp5.sv
│  ├─ display.spade
│  └─ main.spade
├─ test
│  ├─ tests.py
│  └─ visualizer.cpp
└─ swim.toml
```

(b) Swim project structure.

Listing 17. Example of a Swim project.

```
#top = lib::add_mul

async def check_out(dut, i, o):
  s = SpadeExt(dut)
  clk = # Cocotb clock setup
  s.i.op = i
  await FallingEdge(clk)
  s.o.out.assert_eq(o)

@cocotb.test()
async def mult(dut):
  await check_out(
    "Op::Mul(5, 6)",
    "Some(30)"
  )

@cocotb.test()
async def add(dut):
  await check_out(
    "Op::Add(5, 6)",
    "Some(11)"
  )
```

(a) Cocotb test bench.

```
// top = main::main
#define TOP main
#include <verilator_util.hpp>

TEST_CASE(smoke_test, {
  s.i.op = "Op::Mul(1234, 6789)"
  TICK
  s.o.assert_eq("Some(8377626)")
  ...
})
TEST_CASE(smoke_test, { ... }
TEST_CASE(random_inputs, { ... }
```

(b) Verilator test bench.

```
FAIL   test/tests.py 1/2 failed
 ┗ mult FAILED [path_to_vcd1.vcd]
 ┗ test ok [path_to_vcd.vcd]

ok     test/visualizer.cpp 0/3 failed
 ┗ smoke_test ok [cpp_vcd1.vcd]
 ┗ long_running ok [cpp_vcd2.vcd]
 ┗ random_inputs ok [cpp_vcd3.vcd]
```

(c) Test result report.

Listing 18. Example of a test setup with tests written in both cocotb and Verilator.

6.3 Testing

Swim supports test benches for Spade projects using cocotb [14] or Verilator [49], depending on the user's preference. Swim automatically discovers files containing tests in the `test` directory, and further discovers individual test cases in each file. These tests are then executed in parallel leading to significantly faster test running than the Verilator and cocotb default of running each test sequentially.

Since Spade puts heavy emphasis on the type system, it is rare for the user to know the bit patterns of most values, they only know them as the Spade representation. Therefore, the language includes wrappers around cocotb and Verilator values which allows writing Spade expressions as strings which get translated to bit patterns that in turn are fed to the design under test. Listing 18 shows an example of how the tests for the project in Listing 17 can be written. The file `tests.py` shown in Listing 18a uses cocotb to define two test cases, `mult` and `add`, which are using the helper function `check_out` defined on Lines 3–8 to supply inputs and check the outputs one clock cycle later. Similarly, Listing 18b contains a Verilator test bench which defines three test cases using the `TEST_CASE` macro which is bundled with Swim. Finally, Listing 18c shows the resulting output of running `swim test` which lists the success or failure of each individual test case after running them concurrently.

6.4 Waveform Analysis Tools

As mentioned in the previous section, it is rare for users to know the bit patterns of most values, which makes debugging failures in a traditional waveform viewer difficult. To remedy this, a new waveform viewer — Surfer — was created specifically for Spade [48]. The key feature of this waveform viewer from the perspective of Spade is that it is extensible to support translation from bit vectors into hierarchical values. This gives the user a human-readable representation of the value, and allows expansion of things like structs into individual fields.

While Surfer was originally created for Spade, it has evolved into a standalone project with significant community adoption, including integration with other modern HDLs, for example Chisel via the Tywaves project [33].

For higher level analysis, Spade is also integrated with the Waveform Analysis Language (WAL) [27] as described in [47].

6.5 Surrounding infrastructure

For IDE functionality, the Language Server Protocol (LSP) [34] serves as a middle-man between text editors and compilers in order to avoid each text editor having to build support for every language. Spade has a work-in-progress LSP server, which supports inline diagnostics, hover hints, as well as navigation to references and definitions of variables and units. For syntax highlighting, there is a tree-sitter grammar available for editors which support it, primarily Vim and Helix. The language also has a few community-contributed tools including a documentation renderer, a work-in-progress auto-formatter, and editor plugins for several editors.

Finally, to simplify onboarding for new users, there is an online playground[6] which uses WebAssembly to run the compiler and simulation directly in the user's browser without requiring any installation.

7 Implementation

The Spade compiler is a standalone compiler written in Rust. It is released as open source[7] under the EUPL-1.2 license. The rest of the tooling around the language are licensed similarly, and the standard library is permissively licensed in order to allow the language itself to be used in most projects. Like most modern compilers, it is a multi-stage compiler with several intermediate representations that gradually lower the input to the output. The compiler currently emits a small subset of Verilog, since Verilog or VHDL are the only formats that can be reliably consumed by backend tools. However,

[6] https://play.spade-lang.org
[7] https://gitlab.com/spade-lang/spade/

the compiler is designed to be backend agnostic and only requires changing the code generation to target other backends such as CIRCT [13], Calyx [36], or RTLIL [45].

8 Related Work

There is no shortage of new HDLs being developed; the authors are aware of over 50 languages at various stages of completion and use[8]. Naturally, there is not enough space to summarize all of them here, instead, we will highlight some languages of particular interest. For a more in-depth discussion on the HDL ecosystem, see Chapter 3 of [46].

Many modern HDLs, and most of the ones that have gained some ground in industry, are Hardware Construction Languages (HCLs) — RTL-level languages embedded in a software programming language [23]. In HCLs, developers describe hardware by using libraries in the host language to construct synthesizeable hardware components. Having the full power of a software language means that these languages come with extremely powerful metaprogramming facilities. A common use case for this is processor generators like VexRiscv [41] and SoC generators like RocketChip [2]. These generators can be used to generate efficient hardware that can be highly tailored to a specific use case simply by changing parameters. Perhaps the two most well known language in this category are Chisel [5] and SpinalHDL [52] which are both embedded in Scala. Another well-used languages in this category is Amaranth [1] which is embedded in Python and is inspired by Migen.

HCLs and other embedded languages provide very powerful meta-programming facilities, but this means that many abstractions they use are somewhat ad-hoc, potentially limiting interoperability between libraries. They also tend to have somewhat weak type systems at the hardware level, typically only supporting primitives and records at the hardware level even if the host language has a powerful type system. In addition, the embedded language can often feel like a "second class citizen" since the standard keywords are reserved for the host language, and error messages end up being runtime errors in the host language. In contrast, Spade being a standalone language does not benefit from the ability to use a powerful software language for metaprogramming. Instead, metaprogramming is done primarily with the type system, for example using higher order functions to configure functionality. While this limits the possibility of building very powerful generators such as VexRiscV and RocketChip in Spade, it makes the experience of building more conventional hardware designs better.

The pipelining feature in Spade is very similar to that of TLVerilog [44] and ROHD [28], though neither of those handle nested pipelines out of the box. Broadly, pipelining systems give some ability to explicitly reason about timing of signals in the language. Filament [35] and its Parafill extension [37] provide even more guarantees about timing with a timeline type system that allows reasoning not only about latency, but also initiation intervals, ensuring correct-by-construction composition of latency sensitive modules with resource sharing. However, because of these strict guarantees, working with designs that are not latency sensitive or otherwise do not fit the Filament model becomes difficult. In addition, the Filament language is primarily built to demonstrate the timeline types which means it is lacking other language features to make it broadly usable. The Spade pipelining system in contrast provide far fewer guarantees but provides a subset of the guarantees that Filament does in a langauge that is usable today. SUS [56] uses a related technique called latency counting which tracks the latency of signals individually instead of whole pipelines at once. As far as the authors are aware, SUS was built in part after seeing the pipelining system in Spade, and wanting to address some of its perceived shortcomings.

The type system in Spade is more powerful than many contemporary HDLs, but a few other languages have similarly or more powerful type systems. Clash [3] has the full power of the Haskell

[8] https://gitlab.com/TheZoq2/list_of_hdls/

type system available in hardware, and Bluespec [39] has a very similar type system to that of Haskell. RHDL [7, 6] and PyGears [57] include support for sum types but not many of the other functional programming-inspired type system features which Spade supports. In addition, many HCLs have powerful type systems available in the host language, but do not expose these at the hardware level, making some features, particularly pattern matching, difficult or impossible to implement.

The goal of Spade is to build abstractions *on top* of RTL, giving designers full control over the generated hardware when necessary. Many of the languages discussed so far fall in this category, but some abandon the RTL abstraction and operate purely at higher levels of abstraction. Bluespec [39] is a prominent example which uses actions and rules as the base building block. This model has been reused by several other HDLs including Kami [12], Kôika [10], and cmt2 [60, 59]. Other languages with non-RTL abstraction include DFiant HDL [40], Silice [30] and PipelineC [24]. As the abstraction level is raised even further, languages tend to not fall in the HDL category, instead being Accelerator Design Languages (ADLs) or just high-level synthesis. Many of these languages are surveyed in [50].

In summary, many of the key features Spade provides exist in some way in other languages, whether it be software languages or modern HDLs. The key contribution of Spade is to take these existing ideas and adapting them to fit in a package that works as a productive language for developers who want a standlone language with an RTL level description at its core but with powerful abstractions on top. In addition, the type system in particular is powerful enough to allow developers to build further abstractions on top of the base language. As an example, in particular, the type system is powerful enough to reimplement ShakeFlow [20] as discussed in Section 5.8.

9 Target Audience, Scope, and Goals

Finally, having discussed all the details of the language, it is worth taking a step to discuss the overarching goals the target audience of Spade.

The programming model of Spade is quite different from that of Verilog and VHDL, which are event-based imperative languages. We argue that the RTL level description with immutable variables in Spade is a much more appropriate fit for describing synthesizeable hardware as it has a much more obvious mapping to hardware. An obvious consequense of this is decision that Spade is limited to writing synthesizeable hardware: simulation or higher level modelling are not possible. However, the emergence of projects like Cocotb [14] and Verilator [49] shows that even when it is possible to do verification in the synthesis language, many developers prefer to use a convenitional software language for verification.

In addition, since simulation and synthesis are vastly different activities, designing a language that does both requires sacrifices in both areas. In Verilog and VHDL, this manifests in many ways, but an obvious consequence is that neither langauge specifies which parts of the language are synthesizeable, nor how they should be synthesized. Instead, this is left to vendors which results in obvious problems when moving code between vendors and tools. In fact, even in between tools from the same vendor, the supported subset of synthesizeable Verilog differ [54]. While these problems are certainly possible to work around, doing so for experienced developers requires concious effort. For students, the situation is worse. In our experience, students, especially those coming from a software background struggle with VHDL because the mapping between VHDL and hardware is often unclear and because it is far too easy to write complex logic that works in simulation but fails to produce good results in synthesis. In addition, the feedback cycle between design entry and synthesis is often long, and ends with unhelpful warnings buried in the logs of synthesis tools. Switching away from VHDL or Verilog in teaching has been shown to significantly improve student satisfaction and course results [42] and student project outcomes [8].

The non-synthesizeable parts of Verilog and VHDL are sometimes also used for initial high level modelling which is then gradually refined and lowered to synthesizeable code. This possibility is of course lost in a language with only synthesizeable constructs, but we argue that this is a reasonable tradeoff in order to make a language that is more well suited for describing synthesizeable hardware. In addition, we argue that the high-level features that the language adds on top of the base RTL description reduce the necessity of doing non-synthesizeable high level design in the RTL language.

It is also worth noting that the Spade programming model is limited to synchronous designs. As the vast majority of digital designs today are synchronous [51], this is not particularly limiting. In addition, there are dedicated HDLs for asynchronous logic, for example, Loom [11].

The very long term goal of Spade is to be a serious contender as a replacement for Verilog and VHDL, though the road there is long. As a software-inspired HDL, Spade has the potential to address the needs both of current hardware developers and of software developers who are interested in building hardware. Hardware developers benefit from the improved correctness and productivity that the language provides, while software developers benefit from a language that is closer to what they are used to in modern programming languages. In addition, with abstractions that clearly separate combinational logic, sequential logic, and memories, includes abstractions for common hardware constructs like pipelines, and gives a quick feedback cycle thanks to helpful error messages, it is likely that software developers will find it easier to build up the intuition required for good hardware design. Similar effects have hinted at for example when switching from VHDL to Chisel [42] in teaching. On the other hand, there is a risk of the language failing to address the needs of either audience, with the higher level concepts and abstractions being unapproachable to hardware developers who are used to working at a lower level, and with software developers having difficulties adapting to the differences between hardware and software. This is best mitigated by well written documentation and teaching resources that address both audiences by introducing hardware concepts to software developers, and software concepts to hardware developers. Spade has some documentation that follows these ideas already at https://docs.spade-lang.org/, but this is an area where further work is required before the language can be widely adopted.

There is a small but healthy community of people interested in the language, with around 300 users in the official chat group[9] at the time of writing. The community consists of both hardware developers and software developers, though our impression is that it is slighyly skewed towards software developers interested in hardware development who want a language that is closer to what they are used to from software.

10 Future Work

Programming languages are rarely truly “done”, there are always new features and improvements that can be made. Spade is of course no exception and has plenty of big and small improvements that can be made. Finite state machines are ubiquitous in hardware design, and while the enums and match expressions in Spade make it easier to express FSMs than in Verilog and VHDL, it is still a manual and possibly error-prone process that often involves manual translation from a higher level description. Some HDLs address this already: YieldFSM [32] and CoHDL [16] do so by drawing inspiration from “generators” in software, while other languages, such as RubyRTL [29], still requires developers to be explicit about states but provide a a more structured means of describing state transitions. Adopting some of these ideas in Spade will further improve developer productivity.

Crossing clock domains is a potential source of errors that are often hard to debug because they only trigger issues spurriously, which makes them a prime candidate for abstraction. Native clock

[9] https://discord.gg/gxNGKsFPxf

domains also allows propagating enable signals for pipelines, making them far more powerful and addressing some shortcomings with the current stall system, which is why it is a prime candidate for including support for them in Spade. Several existing HDLs support clock domains including Chisel [5], SpinalHDL [52], and Clash [4].

11 Conclusion

Spade is a standalone HDL which aims to make hardware development more productive through hardware specific abstractions such as pipelining and ports. The language has a powerful type system with similar power to that of Haskell and Rust. The abstractions and type system allow easier and safer composition of modules, which in turn enables easier code re-use. Unlike many contemporary HDLs, Spade is a standalone language, and the abstractions it includes are built on top of RTL, rather than replacing it. This ensures that the user is in control over important details where necessary, but can reason at a higher level of abstraction when convenient. For a language to be productive it not only needs good language design but also good tooling. Spade comes with a compiler that emits high quality error messages, a build system which manages dependencies to enable easy re-use of high quality libraries, editor integration through the language server protocol and tree-sitter, and a purpose built waveform viewer.

12 Acknowledgements

The authors are grateful to all external contributors who have helped improving Spade, its surrounding tools, and documentation. An updated list can be found in the repositories, but currently these are:

Alex Pichler	Ethan Uppal	Paul Young
Andrew Nichols	Fabian Bleck	Rachit Nigam
Astrid Lauenstein	Florian Gilcher	Rene Wimmer
Bryce Berger	Francesco Urbani	Riley
Colton Pawielski	José Miguel Sánchez García	Scott Mansell
DasLixou	lajka	Uri Shaked
Edvard Thörnros	Lucas Klemmer	xd009642
Emil Segerbäck	martell	XeroOl

We are also very grateful to the annonymous reviewers who all provided very thorough and detailed reviews that undoubtebly significantly improved the paper.

References

[1] Amaranth contributors. *Amaranth HDL*. 2022. URL: https://github.com/amaranth-lang/amaranth.

[2] Krste Asanović et al. *The Rocket Chip generator*. Tech. rep. UCB/EECS-2016-17. Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[3] C.P.R. Baaij. "Digital circuits in ClaSH". PhD. Thesis. University of Twente, Jan. 2015. DOI: 10.3990/1.9789036538039.

[4] Christiaan Baaij et al. "ClaSH: Structural descriptions of synchronous hardware using Haskell". In: *Proc. Euromicro Conf. on Digit. System Design: Architectures, Methods and Tools*. IEEE, Sept. 2010, pp. 714–721. DOI: 10.1109/dsd.2010.21.

[5] J. Bachrach et al. "Chisel: Constructing hardware in a Scala embedded language". In: *Proc. Des. Automat. Conf.* June 2012, pp. 1212–1221. DOI: 10.1145/2228360.2228584.

[6] Samit Basu. "RHDL: Rust as a hardware description language". In: *Workshop Lang. Tools Tech. Accelerator Design*. Apr. 2025.
[7] Samit Basu. "Rust as a hardware description language". In: *Workshop Lang. Tools Tech. Accelerator Design*. 2024.
[8] Scott Beamer. "Teaching Agile Hardware Design with Chisel". In: *2024 27th Euromicro Conference on Digital System Design (DSD)*. IEEE, Aug. 2024, pp. 161–167. DOI: 10.1109/dsd64264.2024.00030.
[9] Lars Bergstrom. *Beyond safety and speed: how Rust fuels team productivity*. Apr. 2024. URL: https://www.youtube.com/watch?v=QrrH2lcl9ew.
[10] Thomas Bourgeat et al. "The essence of Bluespec: a core language for rule-based hardware design". In: *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*. PLDI '20. ACM, June 2020, pp. 243–257. DOI: 10.1145/3385412.3385965.
[11] Broccoli LLC. *Loom*. https://github.com/broccolimicro/loom. Sept. 2025.
[12] Joonwon Choi et al. "Kami: a platform for high-level parametric hardware specification and its modular verification". In: *Proc. ACM Program. Lang. (PLDI)* 1.ICFP (Aug. 2017), pp. 1–30. ISSN: 2475-1421. DOI: 10.1145/3110268.
[13] CIRCT Project. *CIRCT*. 2022. URL: https://circt.llvm.org/.
[14] Cocotb maintainers. *cocotb*. URL: https://www.cocotb.org/.
[15] Mark T. Daly, Vibha Sazawal, and Jeffrey S. Foster. "Work in progress: An empirical study of static typing in Ruby". In: *Workshop Evaluation Usability Programm. Lang. Tools*. 2009. URL: https://www.cs.umd.edu/projects/PL/druby/papers/druby-pilot-plateau09.pdf.
[16] Alexander Forster. *CoHDL*. URL: https://github.com/alexander-forster/cohdl.
[17] Harry Foster. *Part 6: the 2022 Wilson research group functional verification study*. Nov. 2022. URL: https://blogs.sw.siemens.com/verificationhorizons/2022/11/21/part-6-the-2022-wilson-research-group-functional-verification-study/.
[18] Kelsey R. Fulton et al. "Benefits and drawbacks of adopting a secure programming language: Rust as a case study". In: *Symp. Usable Privacy Security (SOUPS)*. USENIX Association, Aug. 2021, pp. 597–616. ISBN: 978-1-939133-25-0. URL: https://www.usenix.org/conference/soups2021/presentation/fulton.
[19] Zheng Gao, Christian Bird, and Earl T. Barr. "To type or not to type: quantifying detectable bugs in JavaScript". In: *IEEE/ACM Int. Conf. Software Engineering (ICSE)*. IEEE, May 2017. DOI: 10.1109/icse.2017.75.
[20] Sungsoo Han, Minseong Jang, and Jeehoon Kang. "ShakeFlow: functional hardware description with latency-insensitive interface combinators". In: *Proc. ACM Int. Conf. Arch. Support Programming Lang. Operating Syst.* Vol. 2. ASPLOS '23. ACM, Jan. 2023, pp. 702–717. DOI: 10.1145/3575693.3575701.
[21] Stefan Hanenberg. "An experiment about static and dynamic type systems: doubts about the positive impact of static type systems on development time". In: *ACM SIGPLAN Notices* 45.10 (Oct. 2010), pp. 22–35. ISSN: 1558-1160. DOI: 10.1145/1932682.1869462.
[22] John Hennessy and David Patterson. "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development". In: *ACM/IEEE Int. Symp. Computer Architecture (ISCA)*. 2018, pp. 27–29. DOI: 10.1109/ISCA.2018.00011.
[23] Adam Izraelevitz et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Nov. 2017, pp. 209–216. DOI: 10.1109/iccad.2017.8203780.

[24] Julian Kemmerer. "PipelineC: Easy open-source hardware description between RTL and HLS". In: *Proc. Workshop Open-Source EDA Technol.* 2022. URL: https://woset-workshop.github.io/PDFs/2022/17-Kemmerer-poster.pdf.

[25] Faizan Khan et al. "An empirical study of type-related defects in Python projects". In: *IEEE Trans. Softw. Eng.* 48.8 (Aug. 2022), pp. 3145–3158. ISSN: 2326-3881. DOI: 10.1109/tse.2021.3082068.

[26] Sebastian Kleinschmager et al. "Do static type systems improve the maintainability of software systems? An empirical study". In: *IEEE Int. Conf. Program Comprehension (ICPC)*. IEEE, June 2012. DOI: 10.1109/icpc.2012.6240483.

[27] Lucas Klemmer and Daniel Große. "WAVING goodbye to manual waveform analysis in HDL design with WAL". In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 43.10 (Oct. 2024), pp. 3198–3211. ISSN: 1937-4151. DOI: 10.1109/tcad.2024.3387312.

[28] Max Korbel. "Rapid open hardware development framework". In: *Proc. Workshop Open-Source EDA Technol.* 2022.

[29] Jean-Christophe Le Lann, Hannah Badier, and Florent Kermarrec. "Towards a hardware DSL ecosystem: RubyRTL and friends". In: *Proc. Des. Automat. Test in Eur. Conf. – Open Source Des. Automat. Workshop*. Mar. 2020.

[30] Sylvain Lefebvre. *Silice*. Nov. 2022. URL: https://github.com/sylefeb/Silice/tree/5003ec72.

[31] Dan Luu. *Writing safe Verilog*. URL: https://danluu.com/pl-troll/.

[32] Marek Materzok. "Generating circuits with generators". In: *Proc. ACM Program. Lang. (PLDI)* 6.ICFP (Aug. 2022), pp. 52–79. ISSN: 2475-1421. DOI: 10.1145/3549821.

[33] Raffaele Meloni, H. Peter Hofstee, and Zaid Al-Ars. "Tywaves: a typed waveform viewer for Chisel". In: *Proc. Nordic Circuits Syst. Conf.* IEEE, Oct. 2024, pp. 1–6. DOI: 10.1109/norcas64408.2024.10752465.

[34] Microsoft. *Language server protocol*. URL: https://microsoft.github.io/language-server-protocol/.

[35] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. "Modular hardware design with timeline types". In: *Proc. ACM Program. Lang. (PLDI)* 7 (June 2023), pp. 343–367. DOI: 10.1145/3591234.

[36] Rachit Nigam et al. "A compiler infrastructure for accelerator generators". In: *Proc. ACM Int. Conf. Arch. Support Programming Lang. Operating Syst.* ACM, Apr. 2021, pp. 804–817. DOI: 10.1145/3445814.3446712.

[37] Rachit Nigam et al. *Correct and compositional hardware generators*. 2024. DOI: 10.48550/ARXIV.2401.02570.

[38] Rachit Nigam et al. "Predictable accelerator design with time-sensitive affine types". In: *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*. ACM, June 2020. DOI: 10.1145/3385412.3385974.

[39] R. Nikhil. "Bluespec SystemVerilog: efficient, correct RTL from high-level specifications". In: *Proc ACM/IEEE Int. Conf. Formal Methods Models Co-Design*. IEEE, 2004, pp. 69–70. DOI: 10.1109/memcod.2004.1459818.

[40] Yoav Etsion Oron Port. "Registerless hardware description". In: *Workshop Lang. Tools Tech. Accelerator Design*. 2021.

[41] Charles Papon. *VexiiRiscv : A Debian demonstration*. Sept. 2024. URL: https://www.youtube.com/watch?v=dR_jqS13D2c&t=112s.

[42] Luca Pezzarossa and Martin Schoeberl. "Transitioning to Chisel in University Education: Experiences and Lessons Learned". In: *2023 IEEE Nordic Circuits and Systems Conference (NorCAS)*. IEEE, Oct. 2023, pp. 1–7. DOI: 10.1109/norcas58970.2023.10305476.

[43] Baishakhi Ray et al. “A large-scale study of programming languages and code quality in GitHub”. In: *Commun. ACM* 60.10 (Sept. 2017), pp. 91–100. ISSN: 1557-7317. DOI: 10.1145/3126905.

[44] Redwood EDA. *TL-Verilog*. Oct. 2022. URL: https://web.archive.org/web/20221006080731/https://www.redwoodeda.com/tl-verilog.

[45] David Shah et al. “Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs”. In: *Proc. Int. Symp. Field-Programmable Custom Comput. Machines*. 2019, pp. 1–4. DOI: 10.1109/FCCM.2019.00010.

[46] Frans Skarman. *Improved tooling for digital hardware development: Spade, Surfer, and more*. Linköping University Electronic Press, Aug. 2025. DOI: 10.3384/9789181181777.

[47] Frans Skarman et al. “Enhancing compiler-driven HDL design with automatic waveform analysis”. In: *Forum on Specification & Design Languages (FDL)*. IEEE, Sept. 2023, pp. 1–8. DOI: 10.1109/fdl59689.2023.10272204.

[48] Frans Skarman et al. “Surfer – an extensible waveform viewer”. In: *Int. Conf Comput. Aided Verif.* July 2025, pp. 392–404. DOI: 10.1007/978-3-031-98685-7_19.

[49] Wilson Snyder. *Verilator*. URL: https://veripool.org/verilator/.

[50] Emanuele Del Sozzo et al. “Pushing the level of abstraction of digital system design: a survey on how to program FPGAs”. In: *ACM Comput. Surveys* 55.5 (Dec. 2022), pp. 1–48. ISSN: 1557-7341. DOI: 10.1145/3532989.

[51] Jens Sparsø. *Introduction to Asynchronous Circuit Design*. English. Paperback edition available here: https://www.amazon.com/dp/B08BF2PFLN. DTU Compute, Technical University of Denmark, 2020. ISBN: 979-86-550-5385-4.

[52] SpinalHDL contributors. *SpinalHDL*. 2022. URL: https://github.com/SpinalHDL/SpinalHDL.

[53] Stack Overflow. *2023 developer survey*. 2023. URL: https://survey.stackoverflow.co/2023/.

[54] Stuart Sutherland. “Synthesizing SystemVerilog. Busting the myth that SystemVerilog is only for verification”. In: *Synopsys User Group*. 2013.

[55] Justin Tracey and Ian Goldberg. “Grading on a curve: how Rust can facilitate new contributors while decreasing vulnerabilities”. In: *IEEE Secure Development Conf. (SecDev)*. IEEE, Oct. 2023. DOI: 10.1109/secdev56634.2023.00016.

[56] Lennart Van Hirtum and Christian Plessl. “Latency counting in the SUS language”. In: *Workshop Lang. Tools Tech. Accelerator Design*. 2024. URL: https://capra.cs.cornell.edu/latte24/paper/4.pdf.

[57] Bogdan Vukobratović, Andrea Erdeljan, and Damjan Rakanović. “PyGears: a functional approach to hardware design”. In: *Workshop Open-Source Des. Automat.* 2019.

[58] Philip Wadler. “Linear types can change the world!” In: *Programming Concepts and Methods*. 1990.

[59] Youwei Xiao, Zizhang Luo, and Yun Liang. “cmt2: Rule-based hardware description in Rust with temporal semantics”. In: *Workshop Lang. Tools Tech. Accelerator Design*. 2025.

[60] Youwei Xiao et al. “Cement: streamlining FPGA hardware design with cycle-deterministic eHDL and synthesis”. In: *Proc. ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*. FPGA ’24. ACM, Apr. 2024, pp. 211–222. DOI: 10.1145/3626202.3637561.

[61] Fatemeh Yousefifeshki, Heng Li, and Foutse Khomh. “Studying the challenges of developing hardware description language programs”. In: *Inform. Softw. Tech.* 159 (July 2023), p. 107196. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2023.107196.

Received 30 May 2025; revised 19 August 2025; accepted 15 December 2025