Spade: An Expression-Based HDL With Pipelines

Frans Skarman and Oscar Gustafsson
Department of Electrical Engineering, Linköping University
SE-581 83 Linköping, Sweden
Email: {frans.skarman, oscar.gustafsson}@liu.se

***Abstract*—Spade is a new open source hardware description language (HDL) designed to increase developer productivity without sacrificing the low-level control offered by HDLs. It is a standalone language which takes inspiration from modern software languages, and adds useful abstractions for common hardware constructs. It also comes with a convenient set of tooling, such as a helpful compiler, a build system with dependency management, tools for debugging, and editor integration.**

***Index Terms*—Hardware description languages, languages and compilers, Design automation**

I. Introduction

Developing digital hardware is traditionally done using Verilog or VHDL, both languages originating in the 1980s. While those languages have evolved since their inception, they are still lacking many subsequent advancements in programming language design.

Spade[1] is a new open source Hardware Description Language (HDL) which aims to reduce the development effort of digital hardware by taking inspiration from software languages, and adding language-level mechanisms for common hardware structures. This is while still retaining the low-level control provided by HDLs.

Being inspired by Rust and functional programming languages, Spade is expression-based and has a rich type system. It supports product-types like structs and tuples, and sum-types in the form of enums. The language also supports linear type-checking which can be used to ensure that hardware resources such as memory ports are used exactly once.

Spade has built in constructs and abstractions for common hardware structures such as pipelines, memories, and registers. Pipelining allows the user to specify a computation to be performed, with explicit statements for separating stages of the pipeline, but without the need of separate variables for each pipeline stage. Retiming such a pipeline does not require changing any variables, only moving the staging statement. Additionally, the delays of pipelines are explicit in the language and checked by the compiler to ensure that changes to a pipeline do not affect the computation results. In order to more accurately reflect the hardware being described, all logic is combinatorial by default with the only sequential elements being registers and memories which are instantiated explicitly.

The rest of the paper is structured as follows. Related works are discussed in Section II, including highlighting where Spade differs. The basic semantics are introduced in Section III, while in Section IV the use of linear types to model input and output ports is described. The software provided for Spade is discussed in Section V, with concluding remarks in Section VI.

[1]https://gitlab.com/spade-lang/spade/

II. Related Work

In recent years, several new HDLs have been developed. A common approach is to embed the language as a Domain Specific Language (DSL) inside a conventional software programming language such as Scala, Python, or Ruby. There, the HDL consists of a library of hardware constructs which the user instantiates in the host language in order to describe their hardware. This allows the user to take advantage of the power of the host language in their hardware description, for example, by using software control flow structures to generate parameterized hardware and using object-oriented or functional programming approaches in the hardware description. Additionally, this approach exposes all the tooling available for the host language to the hardware designer, such as dependency managers, build tools, and IDEs. Examples of this include Chisel [1], SpinalHDL [2] and DFiant [3] embedded in Scala, Amaranth [4] embedded in Python, ROHD [5] embedded in Dart, and RubyRTL [6] embedded in Ruby.

Some languages take subsets of conventional programming languages and compile them to hardware. An example of this is Clash [7] which compiles a large subset of Haskell to hardware by taking advantage of the natural mapping between a pure functional language and hardware. Another example is PipelineC [8] which is a C-like HDL with automatic pipelining. While not regular C, it is close enough to it to allow many PipelineC programs to be compiled and "simulated" by a standard C-compiler.

TL-Verilog [9] is a SystemVerilog extension which supports timing abstract modeling, where behavior and timing are separated. It provides language features for common hardware constructs such as pipelines and FIFOs.

There are also several languages which are completely independent of existing languages. One such example is Bluespec [10] in which hardware is described by guarded atomic actions. Another example is Silice [11] which contains abstractions for common hardware constructs without losing control over the generated hardware. In addition, it provides a higher-level description style where the design is expressed as sequences of operations with software-like control flow.

Finally, a common alternative for describing hardware is High-Level Synthesis (HLS), in which higher-level languages, typically designed for software, are compiled to hardware.

This design methodology is quite different from the HDLs described previously. Specifically, HLS tools generally provide limited control over the hardware being generated. In particular, the exact timing of any circuit is generally abstracted away, and synchronization between HLS generated modules is done at runtime via synchronization signals.

Now, a natural question to ask is whether there is a need for another HDL, and what Spade offers that the existing work does not? First, it is not an embedded DSL in another language, which separates it from the likes of Chisel and Amaranth. This gives more freedom in the design of the language, as it is not restricted to the expressiveness of a DSL. For example, most embedded HDLs are forced to invent new "keywords" in order to not clash with the host language, a typical example being Chisel using `when`, and `otherwise` instead of the more familiar `if`, and `else`. This problem runs deeper than surface level keywords, however. As an example, Scala has pattern matching, but in Chisel and SpinalHDL, that feature can only be used at "build time" on Scala values, not in hardware on "runtime" values. Finally, a potential user is not required to learn both the host language and DSL when initially picking up the language.

Unlike Clash and PipelineC, Spade is not restricted to conforming to the execution model of the host language which means that the language can be designed directly for hardware description without any restrictions.

Spade describes the behavior of circuits in a cycle-to-cycle manner, which gives more control than Bluespec, where the fundamental abstraction is atomic rules, and PipelineC where the compiler automatically performs pipelining.

Finally, Spade is similar in spirit to TL-Verilog. However, by designing a new language instead of building on SystemVerilog, there is more freedom to explore new language constructs.

III. Basic Semantics

In order to describe the programming model and semantics of Spade, an example is used. Listing 1 shows Spade code which blinks an LED at a configurable interval. Lines 1–2 define an `entity` called blink which takes a clock signal of type `clock`, a reset signal of type `bool`, and a 20-bit integer, `int<20>`, specifying the maximum value for the counter. The return value is a single `bool` value which drives the led. The `entity` keyword specifies that this *unit* can contain sequential logic. One can also write a function, `fn`, which only allows combinatorial logic, and `pipeline` which defines a pipeline. The details of pipelines are discussed later. When instantiating a unit, the syntax differs between functions, entities and pipelines, giving a reader of the code an indication of what can happen behind an instantiation.

Line 4 defines a register called `counter` which is clocked by the `clk` signal and reset to 0 by the `rst` signal. Registers in Spade are explicit constructs rather than being inferred from the use of something like `rising_edge`. Registers are, together with writes to memories, the only sequential constructs in the language, everything else describes combinatorial circuits.

Listing 1. Spade code which blinks an LED

```
1  entity blink(clk: clock, rst: bool, max: int<20>)
2      -> bool
3  {
4      reg(clk) counter reset (rst: 0) =
5          if counter == max {
6              0
7          }
8          else {
9              trunc(counter + 1)
10         };
11     counter > (max >> 1)
12 }
```

In Spade, the behavior of a circuit is described in a cycle-to-cycle manner. The new value of a register in the design is given as an expression of the register values in the current clock cycle. All variables in Spade are immutable: they can only be assigned once. Single assignment is possible because Spade is an expression-oriented language, meaning that most statements are expressions and produce values [12]. For example, the `counter` register is set to the value of the expression following the equals sign on line 4, in this case an if-expression spanning lines 5–10. If the counter has reached the max value, the value *returned* by the if expression is 0, otherwise it is the next value of the counter. Compared to traditional imperative HDLs, the use of immutable variables and expression-based control flow is closer to the resulting hardware where variables correspond to wires and "control flow" is implemented with multiplexers. The language also does not have an explicit return keyword, the last expression in a unit body is the output of the entity, here on line 11.

The call to the `trunc` function on line 9 truncates the value of `(counter + 1)` from 21 bits down to 20 bits. This is needed as Spade prevents accidental overflow by using the largest possible values for most arithmetic operations. More details on the type system is given in Section III-B

Spade has similar scoping rules to most modern software languages: variables are only visible below their definition which makes it more difficult to accidentally create combinatorial loops. Additionally, these scoping rules make it easier for a developer to find the definition of a variable. The definition will always be above where it is used, and generally be grouped with its assignment. Sometimes it is still useful to create a loop of dependencies between variables, however, this is done explicitly using the `decl` keyword.

A. 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 tedious and error-prone as one must make sure that computations are performed on values corresponding to 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 [13].

Spade on the other hand has language-level support for describing pipelines where the user describes which computa-

Listing 2. Example of the pipelining construct in Spade.

```
1  pipeline(4) X(clk: clock, a: int<32>, b: int<32>)
2      -> int<33>
3  {
4          'initial
5          let x = inst(3) subpipe(clk, a);
6          let p = a * b;
7      reg * 3;
8          let s = x + f(a, p);
9      reg;
10         s + stage(initial).a
11 }
12
13 pipeline(3) subpipe(...) -> int<32> {...}
```

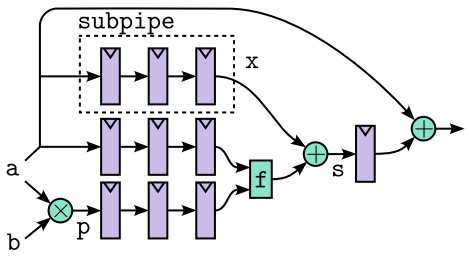

Fig. 1. Hardware described in pipeline `X` by the code in Listing 2.

tions are performed in each stage, and the compiler manages the insertion of registers between stages. Listing 2 shows an example of the pipelining construct in use to describe the hardware in Fig. 1. Like the `entity` discussed previously, the first two lines describe the external interface of the pipeline: name, inputs, and outputs. It also specifies depth, in this case 4, which is the number of registers in the pipeline and therefore the delay between input and output. While the compiler can infer the depth, it has to be specified manually because it is part of the public API. This allows a user to see the interface of the pipeline without reading the body.

Line 4 names the current pipeline stage `initial`, allowing references to it later in the code. On line 5, another pipeline called `subpipe` is instantiated using the `inst` keyword. When instantiating pipelines, the depth of the pipeline has to be specified, and the compiler checks this against the depth specified in the head of the instantiated pipeline. If the depth of the instantiated pipeline is changed later, the compiler gives an error, forcing the developer to consider whether that change has an effect in the instantiating code. The compiler also checks the availability of variables from sub-pipelines, giving an error if the result is used before it is ready. As an example, if `x` is used in the stage right after its definition, it will emit the error message shown in Listing 3.

Line 6 defines a new variable called `p`, which is the product of `a` and `b`, and, from type inference, an `int<64>`. Following that is a `reg`-statement, which behaves differently from the `reg` statement in an entity that was described previously. In a pipeline, the `reg`-statement tells the compiler to insert a pipeline stage, registering all the values visible above it. After the `reg`-statement on line 7, any references to `p` will refer to a delayed version of the value. The same is true for the `x` value computed by `subpipe`, however, because it is computed by a pipeline of depth three, the first three registers are present

Listing 3. Example of pipeline error message

```
error: Use of x before it is ready
   --> src/main.spade:10:19
   |
10 | let sum = x + f(a, product);
   |           ^ Is unavailable for another 2 stages
   = Requesting x from stage 1
   = But it will not be available until stage 3
```

in the sub-pipe and will not be inserted in the instantiating pipeline. The times three operator on the register, `reg * 3`, specifies that three stages should be inserted without any new computations. Finally, on line 10, `s` is added to the value of `a` from the first stage. The `stage` keyword is used to bypass some pipeline registers and to refer directly to a signal as it appears in another stage, in this case the stage named `initial`. Stage references can refer to both "future" and "past" stages, and can do so by name, as is done here, or as relative offsets, for example `stage(-2)`.

The pipelining feature decouples the description of the computation from the description of the pipeline itself. In a pipeline without feedback a developer can easily add and remove pipeline stages as needed without altering the output value of the pipeline. If such changes potentially affect the outcome of other parts of the project, the inclusion of the depth at the call site ensures that the user is made aware of such potential issues via compiler error messages. In pipelines with feedback the structured description of the stages still helps during design iteration, even if some manual care is required to ensure correct computation.

B. Types and Pattern Matching

Spade is a statically typed language with type inference. This means that types can be omitted in most code since the compiler will infer the appropriate types from context, and report errors if types can not be inferred.

Like most languages, Spade has primitive types such as integers and booleans, and compound types like arrays, tuples, and structs. In addition, the language supports enum types inspired by software languages like Rust, Haskell, and ML. Unlike their C or VHDL namesake, these enums have data associated with them in addition to being one of a set of variants. A common use case for this construct is the `Option` type which is defined in the standard library as shown in Listing 4. It is generic over a contained type `T` and takes on one of two values: `Some` in which case a value, `val`, of type `T` is present, or `None` in which case no such value is present. One can view the `Option` type as having a valid/invalid signal bundled with the data it validates.

The main way to interact with enums in Spade is the `match`-expression, which allows pattern matching on values. As an example, Listing 5 shows the `match`-expression in use on a tuple of `Option`-values: `a` and `b`. The resulting hardware is shown in Fig 2. If `a` is `Some`, its inner value is bound to `val` and returned from the match expression. If `a` is `None` but `b` is `Some`, its inner value is returned. Finally, if both `a` and `b` are `None`, 0 is returned. Enums are encoded as bits that specify the currently active variant, the discriminant, followed by bits containing the

Listing 4. Definition of the Option type.

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

Listing 5. Example of pattern matching on a tuple of `Option` values.

```
let a: Option<T> = ...;
let b: Option<T> = ...;
let result = match (a, b) {
    (Some(val), _) => val,
    (_,  Some(val)) => val,
    _ => 0
}
```

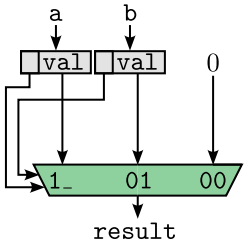

Fig. 2. Hardware generated by Listing 5.

payload. In this case, since the enum consists of two variants, the discriminant is a single bit while the remaining bits contain the `val` when present and are undefined otherwise.

C. Memories

As discussed earlier, Spade requires register definitions to include an expression for the new value of the whole register as a function of the values from the previous clock cycle. However, this abstraction becomes problematic when working with memory-like structures in which only a small part of the total state is updated in each cycle. In order to mitigate this, Spade has an explicit construct for memories, currently implemented as entities defined in the standard library which the compiler handles separately. Instantiation of a memory is done using the `clocked_memory` entity which creates a memory with a fixed set of write ports. Reading from said memory is done via the similarly defined `read_memory` entity. This means that unlike VHDL and Verilog, memories are explicitly instantiated as memories, rather than inferred from the structure of the code. An example of using memories is included in the next Section.

IV. Ports

The discussion so far has been centered around computations on values. A unit receives a set of values as inputs, and produces a set of values as output. Internally, values are used in computation and are pipelined by pipelines. This becomes inconvenient when working with something like a memory which is external to the current unit. The unit must return an address, a write-enable signal, and a value to write to the memory. Then the value read from the memory must be passed as an input to the unit. The memory in turn, produces an output value which must be fed back into the controlling unit. This approach has a few issues: the control signals and output are delayed by the pipeline mechanism. This pipelining introduces additional delays unless manually mitigated, for example by stage references. Additionally, there is no clear link between memory control-signals and the corresponding output.

In order to mitigate this issue, the Spade type system contains the concept of ports and wires. Wires come in two forms: mutable and immutable denoted by `&mut` and `&` respectively. An immutable wire can be used to pass values via units without them being delayed in pipelines. Mutable wires are similarly not pipelined but allow setting the value

Listing 6. Ports being used to share access to a memory between modules.

```
1  struct port RPort {
2    addr: &mut int<16>, read_val: &int<32>
3  }
4  struct port WPort {
5    inner: &mut (int<16>, Option<int<32>>)
6  }
7
8  entity dp_mem(clk: clock) -> (RPort, WPort) {
9    let (r_addr, w) = ...;
10   let w_ports = [inst read_wire(w.inner)];
11   let mem = inst clocked_memory(clk, w_ports);
12   let r_val = inst read_memory(
13       mem, inst read_wire(r_addr)
14   );
15   (RPort(r_addr, &r_val), w)
16 }
17
18 pipeline(10) reader(clk: clock, r_port: RPort) {
19   reg;
20     set r_port.addr = address;
21     let mem_out = *r_port.read_val;
22   reg; // ...
23 }
24
25 pipeline(5) writer(clk: clock, w_port: WPort) {
26   reg;
27     set w_port.inner = (address, Some(value))
28   reg; // ...
29 }
30
31 entity top(clk: clock) -> ... {
32   let (r, w) = inst dp_mem(clk);
33   let reader_out = inst(10) reader(clk, r);
34   let writer_out = inst(5) writer(clk, w);
35   // ...
36 }
```

of the wire in a module which takes the wire as an input. Finally, ports are collections of wires and other ports which are bundled together.

Listing 6 contains an example of how the `port` feature can be used to share a memory between two pipelines. The first lines define `port`-types containing read, write, and address wires. Line 8 defines an `entity` where the actual memory is instantiated. It returns a read-port, and a write-port. Line 9 has been trimmed for space but instantiates the `mut`-wire `r_addr` and a `WPort`. Lines 10–11 instantiate a memory with a single write-port using the `clocked_memory` entity, and lines 12–14 asynchronously reads one value from the memory. At the end of the `entity`, the read-port is assembled from the mutable address, and the result wire, and returned along with the write-port.

The pipelines `reader` and `writer` are two pipelines which use the read-port and write-port respectively. On line 20, the reader pipeline sets the address it wants to read from, and reads the resulting value on line 21. Similarly, the writer sets the target address and write-value on line 27.

Finally, the three units are instantiated on lines 32–34. The ports returned by the memory are passed to the reader and writer.

There are some pitfalls when working with mutable wires: A unit returning a mutable wire expects a value to be set for that wire, otherwise the value of the wire may be undefined, or if it is set conditionally, a latch could be inferred. Similarly,

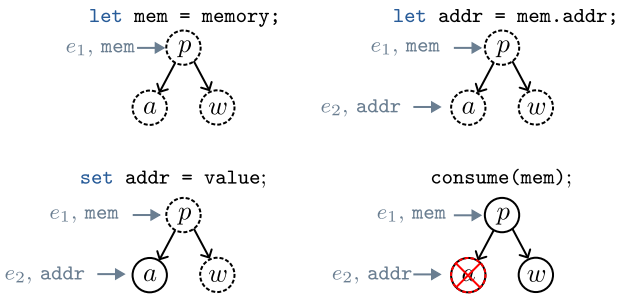

Fig. 3. The linear type checking procedure for the code in Listing 7.

Listing 7. Example code for Fig. 3.

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

if a wire is driven by multiple drivers, conflicting values may cause issues. While it is possible to catch this in simulation, it is better to let the compiler catch such errors.

The solution to this is inspired by [14] which uses affine types to ensure correct memory access patterns in an HLS tool. Affine types can guarantee that a value is used at most once, resolving the multiple driver problem, however, to ensure that all mutable wires are set exactly once, the stronger notion of linear types [15] is required, and implemented in Spade.

Because Spade describes behavior in a cycle-to-cycle manner, the implementation of a linear type system is easier than in the general case. Each resource of linear type must be *consumed* exactly once each clock cycle. A value is consumed when it is set using the `set` statement, or passed to another unit, which delegates the consumption requirement to that unit.

Linear type checking happens after normal type checking, meaning that the compiler knows which resources are of linear type and must be checked. 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 pointer from the alias to the corresponding tree and node.

When expressions or statements which consume resources, such as the `set`-statement, or a resource being passed to a unit are encountered, 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.

To exemplify the linear type checking algorithm, Fig. 3 shows this process in action for the code in Listing 7. In the figure, e_x 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.

V. Spade Software

A good set of tools for a language, both for working with the language itself, and for integration with existing tools can be of huge help for driving language adoption. First and foremost, the Spade compiler and the language is built from the start to produce useful and easy to read error messages such as the one shown in Listing 3, and unhelpful diagnostics from the compiler are considered bugs. The language is also designed to prefer emitting errors over potentially surprising behavior, for example, by requiring explicit truncation after potentially overflowing arithmetic.

A. Compiler Architecture

The Spade compiler is a multi-stage compiler written in Rust, which compiles the input Spade code to a target language, currently a slim subset of SystemVerilog. The compilation process starts with lexing and parsing to generate an Abstract Syntax Tree (AST). The AST is then traversed thrice, first to collect all types in the program, then to collect all units, and finally to be lowered into a High-Level Intermediate Representation (HIR). The AST to HIR lowering process retains the tree structure of the AST but resolves names and scoping rules, and performs initial semantic analysis. Once the HIR is generated, type inference is performed, followed by linear type checking as discussed earlier. The HIR along with the type information is used to generate a Mid-Level Intermediate Representation (MIR). In this step, more semantic analysis is performed, and the tree structure is flattened to a list of simple statements. Finally, SystemVerilog is generated from the MIR. SystemVerilog is chosen as the target language as it is well-supported by both open source and proprietary simulation and synthesis tools, but the compiler is written in a target independent way to enable experimenting with, or changing to a different backend with ease. Especially interesting backend are the CIRCT [16] dialects, such as LLHD [17] or Calyx [18], which can offer language independent optimization as well as code generation of other output languages than SystemVerilog.

B. Tooling and Ecosystem

Spade comes with the build tool Swim[2] which manages project files, backend build tools, and dependencies. A Swim project consists of Spade files and a configuration file written in TOML. This file contains, among other things: build tool parameters, raw Verilog files to be included in the project, and external dependencies. Swim then manages namespacing of project files, downloading and versioning of dependencies as well as calling the synthesis and simulation tools with a convenient interface. It also has a plugin system for extending the build flow, for example by running commands to generate Spade or Verilog code, loading additional Yosys plugins or

[2]https://gitlab.com/spade-lang/swim

Listing 8. Example of a test bench for Spade written using cocotb.

```
1  # top=peripherals::timer::timer_test_harness
2  @cocotb.test()
3  async def timer_works(dut):
4    s = SpadeExt(dut)
5
6    clk = dut.clk_i
7    await start_clock(clk)
8
9    s.i.mem_range = "(1024, 2048)"
10   s.i.addr = "1024 + 0"
11   s.i.memory_command = "Command::Write(10)"
12   await FallingEdge(clk)
13   s.o.assert_eq("10")
```

bundling the output bitstream into an executable for a microcontroller which in turn programs a target FPGA.

To facilitate integration of spade with existing projects, units can be annotated to prevent name mangling. Spade units can also be marked as external, in order to allow use of existing IP blocks within Spade projects.

A tree-sitter grammar, and a rudimentary Language Server Protocol (LSP) [19] server is available, enabling an IDE-like experience in any text editor supporting LSP and/or tree-sitter.

There is limited effort to generate Verilog similar in structure to the input Spade code. However, the compiler does attempt to keep names readable, and has functions for mapping names and expressions back to their source location to aid debugging. Swim automatically translates values in VCD files into their high-level Spade values, and the compiler includes a source mapping in the output Verilog which makes things like timing reports readable without looking at the output Verilog.

C. Test Benches and Simulation

The Spade language itself is designed primarily for hardware description and synthesis, rather than simulation. However, in order to verify correctness of the resulting hardware, simulation and test benches are essential. Spade tests are written using cocotb [20], a Python based co-simulation test bench environment for verification. The cocotb library is extended with features for writing Spade values as inputs and outputs to the unit under test, while Verilog generated by the compiler is simulated with off-the-shelf Verilog simulators.

As an example, Listing 8 shows part of a test bench for a memory mapped timer peripheral. The first line specifies the module being tested, and the next two lines are a standard cocotb test case definition. Line 4 wraps the cocotb design under test in a Spade class which extends the cocotb interface to add Spade-specific features. Lines 6–7 start a task to drive the clock of the Design Under Test (DUT). On lines 9–11, the inputs to the module are set to values which are compiled and evaluated by the Spade compiler. Finally, line 13 asserts that the output is as expected, again passing a string containing a Spade expression as the expected value.

In order to achieve this, the Python test bench must be able to compile Spade code, and in order to allow the use of types and units defined in the project inside a test bench, the state of the compiler must be made available to the test bench. For this reason, the state of the compiler after building a project is serialized and stored on disk. Additionally, parts of the Spade compiler are exported as a Python module which reads the stored state, then compiles and evaluates the expressions used in the test bench. The resulting bit vectors are used to drive the inputs of the DUT, or compared to the expected output.

VI. Conclusions

Spade is an HDL which attempts to ease the development of FPGAs or ASICs. To do so, it makes common hardware constructs like pipelines, registers, and memories explicit and part of the language. It is heavily inspired by modern software languages, e.g., by integrating a powerful type system and pattern matching. Unlike most current alternative HDLs, Spade is its own standalone language with a custom compiler, and does not abstract away the underlying hardware. Finally, it comes with useful tooling, such as a compiler designed to emit detailed error messages, a build system with dependency management, and an LSP implementation for an IDE-like experience.

References

[1] J. Bachrach *et al.*, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. Des. Automat. Conf.*, Jun. 2012, pp. 1212–1221.

[2] SpinalHDL contributors, "SpinalHDL," https://github.com/SpinalHDL/SpinalHDL, 2022.

[3] O. Port and Y. Etsion, "DFiant: A dataflow hardware description language," in *Proc. Int. Conf. Field Programmable Logic Appl.* IEEE, Sep. 2017.

[4] Amaranth contributors, "Amaranth HDL," https://github.com/amaranth-lang/amaranth, 2022.

[5] M. Korbel, "Rapid open hardware development framework," in *Proc. Workshop Open-Source EDA Technol.*, 2022.

[6] J.-C. Le Lann, H. Badier, and F. Kermarrec, "Towards a hardware DSL ecosystem: RubyRTL and friends," in *Proc. Des. Automati. Test Europe Conf. – Open Source Des. Automat. Workshop.*, Mar. 2020.

[7] C. Baaij, "Digital circuits in cλaSH," PhD. Thesis, University of Twente, Jan. 2015.

[8] J. Kemmerer, "PipelineC: Easy open-source hardware description between RTL and HLS," in *Proc. Workshop Open-Source EDA Technol.*, 2022.

[9] S. F. Hoover, "Timing-abstract circuit design in transaction-level Verilog," in *Proc. IEEE Int. Conf. Comput. Des.* IEEE, Nov. 2017.

[10] R. Nikhil, "Bluespec SystemVerilog: efficient, correct RTL from high-level specifications," in *Proc ACM/IEEE Int. Conf. Formal Methods and Models for Co-Design*. IEEE, 2004.

[11] S. Lefebvre, "Silice," https://github.com/sylefeb/Silice/tree/5003ec72, Nov. 2022.

[12] S. Klabnik and C. Nichols, *The Rust programming language*, 1st ed. No Starch Press, Jun. 2018, ch. Glossary.

[13] D. Luu, "Writing safe Verilog," https://danluu.com/pl-troll/.

[14] R. Nigam *et al.*, "Predictable accelerator design with time-sensitive affine types," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*. ACM, Jun. 2020.

[15] P. Wadler, "Linear types can change the world!" in *Programming Concepts and Methods*, 1990.

[16] CIRCT Project, "CIRCT," https://circt.llvm.org/, 2022.

[17] F. Schuiki, A. Kurth, T. Grosser, and L. Benini, "LLHD: a multi-level intermediate representation for hardware description languages," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*, Jun. 2020.

[18] R. Nigam, S. Thomas, Z. Li, and A. Sampson, "A compiler infrastructure for accelerator generators," in *Proc. ACM Int. Conf. Arch. Support for Programming Lang. Operating Syst.*, Apr. 2021.

[19] Microsoft, "Language server protocol," https://microsoft.github.io/language-server-protocol/.

[20] cocotb contributors, "cocotb," https://www.cocotb.org/, Nov. 2022.