# Spade: An HDL Inspired by Modern Software Languages

Frans Skarman, 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 hardware description language which aims to make hardware description easier and less error prone. It does this by taking lessons from software programming languages, and adding language level support for common hardware constructs, all without compromising the low level control over what hardware gets generated.**

***Index Terms*—Hardware Description Languages, Languages and Compilers**

## I. Introduction

Programming Field-Programmable Gate Arrays (FPGAs) is traditionally done using Verilog or VHDL, both of which are languages originating in the 1980s. While newer versions of the languages incorporate more features, they are still missing out on the developments that have been made since their inception in the software programming language world. High-Level Synthesis (HLS) has seen a steady increase in usage in recent years and while it improves the ease of programming FPGAs, it abstracts away a lot of the low level details of the FPGA.

Spade[1] is a new Hardware Description Language (HDL) which attempts to improve the programmability of FPGAs by taking lessons from software languages, and adding language level mechanisms for common hardware structures, while still retaining the low level control provided by HDLs.

The main features of Spade are:

- Type inference
- Sum and product types inspired by functional programming
- Pipeline constructs
- Immutable variables
- Combinatorial logic by default with explicit registers

Unlike most modern HDLs like Spinal HDL [1], Chisel [2] which are embedded in Scala, Armanath [3] which is embedded in Python, and Clash [4] which compiles a subset of Haskell to hardware, Spade is a stand-alone language designed from the ground up as an HDL.

The rest of the paper describes some important concepts of the Spade language and briefly touches on the architecture of the compiler. Throughout the discussion, some parallels are drawn to other HDLs in order to point out differences and similarities.

## II. The Spade Language

In Spade, variables are immutable, that is, their value is only assigned at one point in the program. In order to facilitate this, Spade is expression-based meaning that most language constructs are expressions which produce a value. For example, one can assign the "result" of an if-expression to a variable instead of conditionally assigning values in the separate branches. In addition, Spade has similar visibility rules to those found in most software languages: variables are only visible after their declaration unless explicitly predeclared. These properties together make the code easier to reason about, as one does not need to look through the whole program to find where a variable is assigned.

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

All expressions in Spade describe combinatorial circuits. Sequential logic is implemented using the `reg` keyword where the user specifies a clock, a name for the register and optionally a reset signal and reset value, as well as an expression which gives the next value of the register. As an example, the following code describes a counter `x` with a max value, `max`, using a clock, `clk`, and a reset signal, `rst`, resetting the value to 0:

```
reg(clk) x reset (rst: 0) =
    if x == max {0} else {trunc(x + 1)};
```

Spade describes hardware in a cycle to cycle manner. This is unlike some modern HDLs like DFiant which uses a data flow abstraction [5], and PipelineC which automatically pipelines pure functions and generates state machines from functions with state [6],

### A. Pipelines

Spade has language level support for describing pipelines similar to what is provided by TL-verilog [7]. A pipeline consists of a series of manually separated stages where the compiler inserts registers to pipeline intermediate signals to ensure that variables stay in sync between stages. For feedback between stages, one can refer to values in previous or future stages, both using relative references, like "the current stage + 1", and named stages, like "the stage named `writeback`". The pipeline depth is visible in the head of a pipeline, as well as at the instantiating site which means that the compiler can notify the user when code which instantiates a pipeline must be modified to accommodate the new depth. The compiler also ensures that results coming out of pipelines instantiated in other pipelines are not read before they are ready. For example, the result of a five-stage pipeline instantiated in stage $x$ cannot be read until stage $x + 5$ and attempting to do so results in a compilation error.

The pipelining feature enables much easier writing, and especially modification of existing pipelines, as adding or removing additional variables for signals in the intermediate stages is not required.

Listing 1 shows how the pipeline construct can be used to build a circuit which computes $f(a, a \times b)$ where the multiplication is given three cycles to complete and $f$ is some combinatorial circuit. The resulting hardware is shown in Fig. 1. The registers added after the multipliers labeled by the dotted box can be retimed into a DSP block by the synthesis tool.

Listing 1. A four-stage pipeline computing $f(a, a \times b)$.

```
pipeline(4) X(clk: clk, a: int<32>, b: int<32>)
    -> int<64> {
        let product = a * b;
    reg * 3;
        let result = f(a, product);
    reg;
        result
}
```

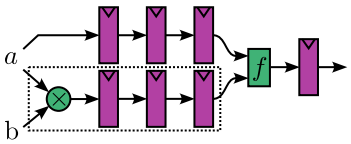


Fig. 1. Hardware generated by the code in Listing 1.

### B. Types and Match Expressions

Spade is a statically typed programming language with a type system inspired by Rust and functional programming languages like Haskell and ML. This means that Spade has primitive types, structs and tuples, and tagged unions called `enum`s. The `enum`s, unlike their C or VHDL namesake have data associated with them in addition to being one of a set of variants. A very common use of this construct is the `Option` type which is written as

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

The type is generic over some type `T`, and can be one of two variants: `Some(val)` where a value is present, and `None` where no value is present. Before accessing the associated `val`, one must check the variant to make sure that a value exists. The `Option`-type can be seen as a valid signal together with the value it validates.

The main way to use `enum`s is together with the `match`-statement which is similar to but more powerful than a `switch-case`-block in C or Verilog. It requires listing each of the potential values an expression can hold can hold as patterns, and produces a result depending on which pattern matches the current value. As an example, the following code shows the use of patterns for tuples, wild card patterns (`_`) and binding of variables to sub-patterns in a block of code which sets `result` to the first of two `Option` values which is `Some`, defaulting to 0 if both are `None`:

```
let result = match (a, b) {
    (Some(x), _) => x,
    (_,  Some(x)) => x,
    _ => 0
}
```

### C. Type Inference

While Spade is statically typed, the variable-types in function bodies can usually be omitted as the compiler can infer them from the context in which they are used, as can be seen by the lack of explicit types in the previous code samples. This provides the correctness benefits of a static type-system but alleviates the need for the user to manually write out any types, which has benefits to productivity as well as allowing more complex types to be used ergonomically. While the type inference can technically be implemented for function arguments as well, Spade requires explicit types there in order to make the types part of the contract of the module.

### D. Memories

Memories are modeled by built in functions which accept an array of write and read ports. This is done in part to require the user to be explicit when memories are instantiated rather than relying on the synthesis inferring it from the structure of the code. Another reason for this design choice is that expressions in Spade always describe whole values, and variables are always updated in full on every clock cycle. This is fundamentally different from memories where only parts of a value are changed.

At present, parts of memory ports must be passed between units as normal arguments and return values, which is unergonomic when multiple units interact with the memory. For this reason, it is interesting to add the concept of ports to the language: a unit which uses a memory accepts a port as an argument, through which it can interact with the memory. Memory instantiating then produces a set of ports which can be distributed to units which need them. Finally, affine types can be used to ensure that each memory port is only used once. This can also be used to model other finite resources, such as shared hardware. A similar scheme is implemented in the Dhalia language [8].

## III. Compiler Architecture

The Spade compiler is a stand-alone program written in the Rust programming language. It is a multi-stage compiler which performs in order: lexing and parsing to an AST, semantic analysis which results in a High-Level Intermediate Representation (HIR), type inference, more semantic analysis and finally lowering to a Medium-Level Intermediate Representation (MIR) which can be compiled to a target language, currently System Verilog. At present, optimizations are delegated to the synthesis tool during Verilog synthesis, but there is most likely room for optimizations before that, at the Spade level.

As future work, it is also interesting to explore the use of existing compiler infrastructure for hardware generation, like one of the immediate representations in CIRCT [9] such as LLHD [10] which can offer language independent optimization as well as code generation to languages other than System Verilog.

## IV. Conclusions

Spade is a HDL which takes inspiration from modern software programming languages by including a rich type system with type inference, pattern matching constructs and immutable variables by default. When coupled with abstractions for common hardware constructs such as registers, memories and pipelines, the language becomes more ergonomic than the traditional HDL alternatives, without sacrificing the low level control which is lost when using HLS or higher level HDLs.

## References

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

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a Scala embedded language," in *Proc. Des. Automat. Conf.*, Jun. 2012, pp. 1212–1221.

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

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

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

[6] J. Kemmerer, "PipelineC," Mar. 2022. [Online]. Available: https://github.com/JulianKemmerer/PipelineC/wiki

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

[8] R. Nigam, S. Atapattu, S. Thomas, Z. Li, T. Bauer, Y. Ye, A. Koti, A. Sampson, and Z. Zhang, "Predictable accelerator design with time-sensitive affine types," in *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation*. ACM, Jun. 2020.

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

[10] 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.