Abstraction in the Spade Hardware Description Language

Frans Skarman
Linköping University
Sweden
frans.skarman@liu.se

Oscar Gustafsson
Linköping University
Sweden
oscar.gustafsson@liu.se

ABSTRACT

Spade is an HDL that enhances the productivity of HDL designers by adding useful abstractions for hardware design. These abstractions are zero- or low-cost, meaning that the designer still has full control over what hardware gets generated.

1 INTRODUCTION

Spade[1] is an open source HDL which aims to be more productive than traditional HDLs without sacrificing the low-level control usually provided by HDLs. A primary way to achieve this is via abstractions for common hardware constructs. Such abstractions make design intent clearer, which in turn allows the compiler to help find and report potential issues with the design. They also make code re-use easier, by encoding more of the properties of the design in the language. The focus of this paper is on the abstractions built into the Spade language, for a more thorough introduction to the syntax and semantics the reader is referred to a previous paper[10]. For now, it is enough to know that Spade describes the behavior of a circuit in a cycle-to-cycle manner, and that the basic building block of a Spade design is a *unit*, which corresponds to a *module* in Verilog or *entity* in VHDL. Units can be *functions* which are purely combinatorial, *entities* which allow sequential logic, and *pipelines* which are discussed in detail in the next section. A lot of the syntax of the language is inspired by the Rust language, meaning that those who are familiar with it should be comfortable with the syntax of Spade.

Unlike most HDLs, which are often imperative, Spade is an expression based language, with immutable variables. Additionally, while many modern HDLs [1, 2, 4, 5, 9, 11], are embedded DSLs in a software language, Spade is a standalone language built from the ground up for hardware description.

The rest of the paper focuses on some of the abstractions which Spade provides to enable more productive hardware design.

2 PIPELINES

Pipelines are a common construct in hardware designs, but traditional HDLs lack structured methods for describing and reasoning about them. Spade on the other hand has language-level support for describing pipelines. The user describes which computations are performed in each stage, and the compiler manages the insertion of registers between stages. Listing 1 shows an example of the `pipeline` construct in use.

Each pipeline has a depth as part of its public interface, in this case 4, which must be specified when instantiating the pipeline, as is done in the instantiation of `subpipe` on line 4. This allows the compiler to notify designers if the timing of a pipeline has changed, potentially invalidating assumptions made during instantiation.

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

Listing 1: Example of the pipeline construct in Spade.

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

Additionally, when instantiating pipelines inside other pipelines, the compiler ensures that values are not read before they are ready.

The `reg` statement separates stages of the pipeline. Whenever the compiler encounters this statement, it inserts registers for all variables declared above it, and aliases the name of those variables to the corresponding register. This makes retiming and re-pipelining the design significantly easier, as one can add, move, or remove `reg` statements and computations without changing any references.

Finally, values in previous or future stages can be referenced by relative index or by stage name in order to create feedback or bypass pipeline stages, as shown on line 9.

3 TYPES

Spade is a statically typed language with type inference, which allows the user to omit the type of most values in unit bodies. The language contains compound types such as structs, tuples, and arrays. In addition, it contains "sum types" in the form of enums. These allow the programmer to reason about values which take on one of several distinct types. Pattern matching is a first class feature of the language, enabling convenient use of the enum values.

The type system also prevents accidental overflow by extending the result of arithmetic to the appropriate bit widths, and requiring explicit truncation to add or discard bits. Using type inference, this extension or truncation is generally non-intrusive as the compiler can infer the desired bit width.

Simply using the bit widths of intermediate calculations results in a pessimistic bit width, however. It is therefore of interest to explore inferring the widths of values based on the range of the values they represent, rather than directly from the bit widths.

In order to facilitate code re-use, the type system makes use of generics, which allows code to be written to accept any type that satisfies some constraints. These constraints are modeled as *traits* which, like their rust namesake, specify what functionality must be implemented for the type in order to satisfy the trait. For example, a type representing values which can be added together, such as a fixed-point or a complex number, can implement an `Add` trait. A unit can then be generic over any type which implements the `Add` trait, enabling its re-use with any such type. An example is shown

Listing 2: Generic accumulator with an implementation for complex numbers

```
trait Add<Rhs> {
    type Output;
    fn add(self, rhs: Rhs) -> Output;
}

entity accumulate<T, O>(clk: clock, new: T) -> O
where T: Add, O: T::Output {
    reg(clk) x = x + new;
    x
}

impl<#N> Add<Complex<N>> for Complex<N> {
    type Output = Complex<N+1>;
    fn add(self, rhs: Complex<N>) -> Complex<N+1> {
        Complex(self.r + rhs.r, self.i + rhs.i)
    }
}
```

in Listing 2 where an accumulator which supports any addable type is described, and the `Add` trait is implemented for a `Complex` struct.

It is worth noting that at the time of writing, support for generics in the compiler is implemented, but the trait system is not. However, the design is similar enough to the existing rust trait system that there should be no issues blocking the implementation.

As Spade makes heavy use of integers in generics, for example to track the size of values, checking constraints on integers becomes a necessity. However, type checking arbitrary constraints on integers for generics is difficult. For that reason, Spade currently falls back on template-like type checking for units which are generic over integers. Final type checking of such units happens during instantiation when all concrete types are known, rather than during type checking of the generic unit.

4 LINEAR TYPES AND PORTS

Apart from types which represent values to be computed on, Spade also contains *ports*. Ports consist of immutable and mutable *wires* which can be read from or written to inside units. The main use for these types is for interaction between units where the interaction is more complex than one unit passing its output to another unit's input. A good example of this is a compute-unit interacting with a memory, where it produces an address and receives data at a later time. Ports allow such input and output signals to be grouped together, and prevents them from being delayed in pipelines.

Units with control ports, such as memories or buses, return values of port type which in turn can be passed to modules which use them, such as accelerators or processors. The controlling units then set the control signals, and read the results from the port.

Spade uses linear types to ensure mutable wires are driven exactly once. This prevents bugs caused by signals with missing or multiple drivers.

5 FUTURE WORK

Spade is currently usable for building non-generic designs[2], but work remains to make it useful generally. A big part of this is finalizing the generic type system as discussed in Section 3, though further additions will also help.

[2]Some example projects are listed at https://spade-lang.org/showcase

Compile time code generation is a useful feature of any language, especially HDLs. Several approaches for this exist. Verilog and VHDL have generate-statements, Silice [6] uses embedded Lua to generate code, hardware construction languages like Chisel use their native language for code generation, and Clash [2] uses recursion and iteration functions such as map, filter and reduce. As Spade has no host language, the hardware construction language approach is impossible. Generate-like statements are an option, as is embedding another language such as Lua as a preprocessor. However, as Spade makes heavy use of types, the expansion should preferably be done after type inference, *and* generate code with valid types. For this reason, the Clash approach of using recursion and dedicated iteration functions may be preferable, though it is harder to use for those more comfortable with traditional iteration. In addition, a lot of languages have a macro feature which expands things earlier in the compilation process. This is also desirable in Spade, and here, more options such as embedded Lua, or Rust inspired macros are possible.

Pipelines currently consume and produce one value per clock cycle, and the time between input and output is the same for all inputs and outputs. Adopting timeline types introduced by Filament [7] may improve the expressiveness of pipelines in addition to allowing more structured reasoning about signal timing and hardware resource sharing. Another potential extension to the pipeline system is native support for more complex pipelines which support stalling when execution is blocked, when inputs are unavailable, or when there is back-pressure from downstream pipelines.

Clock- and reset-domain handling is another area where improvements are possible. Currently, it is left as an exercise to the user, but there is a lot of room for the compiler and language to provide assistance. Like types, it is desirable for the domain of signals and registers to be inferred when possible, e.g. in a unit with only one domain. When more domains are involved, the language should require explicit annotation of domains, and the compiler should produce an error when domains are accidentally mixed. Several modern HDLs have features like this, so the main challenge here is finding a design that works with the rest of the language. Taking inspiration from Rust again, the syntax for lifetimes in Rust can potentially be adapted to domains in Spade.

Finally, the compiler currently emits Verilog code which is fed into simulators and synthesis tools, but it is structured to be retargetable to other intermediate languages or representations. Integration of the compiler with a modern hardware IR such as Calyx [8] or CIRCT [3] will enable the use of their included optimization passes and backends.

6 CONCLUSIONS

Spade is a standalone HDL which uses zero- and low-cost abstractions to simplify hardware description. These abstractions include a system for reasoning about pipelines, a static type system for less error-prone code where more assumptions about the structure of data can be encoded, and linear types for ports between units. Future extensions of the language include a trait system for enabling generic re-usable code, macro and code generation systems, as well as clock-domain inference and checking.

REFERENCES

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

[2] C.P.R. Baaij. 2015. *Digital circuits in CλaSH.* PhD. Thesis. University of Twente. https://doi.org/10.3990/1.9789036538039

[3] CIRCT Project. 2022. CIRCT. https://circt.llvm.org/.

[4] Julian Kemmerer. 2022. PipelineC. https://github.com/JulianKemmerer/PipelineC/wiki

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

[6] Sylvain Lefebvre. 2022. Silice. https://github.com/sylefeb/Silice/tree/5003ec72.

[7] Rachit Nigam, Pedro Henrique Azevedo de Amorim, and Adrian Sampson. 2023. Modular hardware design with timeline types. In *Proc. ACM SIGPLAN Conf. Program. Language Des. Implementation.*

[8] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A compiler infrastructure for accelerator generators. In *Proc. ACM Int. Conf. Arch. Support for Programming Lang. Operating Syst.* https://doi.org/10.1145/3445814.3446712

[9] Oron Port and Yoav Etsion. 2017. DFiant: A dataflow hardware description language. In *Proc. Int. Conf. Field Programmable Logic Appl.* IEEE. https://doi.org/10.23919/fpl.2017.8056858

[10] Frans Skarman and Oscar Gustafsson. 2023. Spade: an expression-based HDL with pipelines. In *Proc. Workshop Open-Source Des. Automat.*

[11] SpinalHDL contributors. 2022. SpinalHDL. https://github.com/SpinalHDL/SpinalHDL.