Frost HDL `struct`s and `interface`s

Andrew Clark

March 17, 2019

Overview

`struct`s and `interface`s are two major features being borrowed from SystemVerilog. `struct`s in Frost HDL will have more features than those in SystemVerilog. However, `interface`s in Frost HDL will have fewer features than `interface`s in SystemVerilog.

`structs`

Notably, `struct`s in Frost HDL can be `parameter`ized. `struct`s in SystemVerilog cannot be `parameter`ized.

2.1 The Three Forms

`struct`s come in three forms in Frost HDL: `packed`, `unpacked`, and `splitvar`.

`packed` and `unpacked` `struct`s are similar to the `struct`s of SystemVerilog, but `unpacked` is a required keyword if that type of `struct` is desired in Frost HDL.

`splitvar` `struct`s in Frost HDL are like an inbetween of `packed` and `unpacked` `struct`s.

Additionally, `struct`s in general can be used *as* `parameter`s of other entities.

2.1.1 `packed structs`

`packed` `struct`s are the most restrictive type of `struct` in Frost HDL in terms of what their elements can be.

They can placed on `module` ports and can be placed inside of `interface`s.

They cannot store arrays of any type, but they can store scalars of built-in types and scalars of other `packed` `struct`s.

Also, they compile into plain old Verilog vectors (of type `wire` or `reg`), with different (non-overlapping) slices of the vector indicating the element of the `packed struct`.

They can be used as vectors within Frost HDL source code, as well, but this will require manual casting.

2.1.2 `unpacked structs`

`unpacked structs` can not be placed on `module` ports or inside of `interfaces`.

However, other than this restriction, they are the most flexible type of `struct` in Frost HDL. They can contain arrays, even arrays of other other `structs`.

An array of `unpacked structs` will be compiled into a `structure` of arrays. This is the primary method by which one can use multi-dimensional arrays in Frost HDL. Simply use an array of `unpacked structs` where each `struct` contains another array. This will produce a multi-dimensional array in the generated Verilog. This is admittedly a little inconvenient, so direct support for multi-dimensional arrays *may* be added to the language later.

In some simple cases cases, an `unpacked struct` *may* be compiled into a vector in the generated Verilog.

2.1.3 `splitvar structs`

`splitvar structs` are inbetween `packed` and `unpacked structs` in terms of functionality.

They can be placed on ports.

They cannot be treated as vectors, but they can contain arrays of built-in types and arrays of `packed structs`. They can contain non-array elements of these as well.

The elements of a `splitvar struct` are compiled into separate variables in the generated Verilog, including if placed on `module` ports or inside of an `interface`. These variables will share the same port direction if their `splitvar struct` is used as `module` ports.

`interfaces`

These are similar to basic `interface`s in SystemVerilog. Unlike in SystemVerilog, however, they can not have their own ports.

`modport` must be used if an `interface` is used for a `module` port. A `modport` will be "flattened" into Verilog `module` ports.

In the `module` that an `interface` is instantiated, the component variables of that `interface` will be implemented, in the generated Verilog, as separate variables, similar to what *may* happen with `unpacked struct`s and what *always* happens with `splitvar struct`s.

`interface`s cannot be instantiated inside of other `interface`s, and `unpacked struct`s may not be instantiated inside of `interface`s.

All other types (built-in; `enum`s; `packed struct`s; `splitvar struct`s) are fair game, and they may be instantiated inside of `interface`s.

`interface`s may be `parameter`ized, and they may contain `function`s and `task`s.

Note that any Frost HDL language constructs that are not mentioned here are probably *not* going to be allowed inside of an `interface`.

What I'm not sure about is if I want to support polymorphism via `interface`s and the name of a `modport`. This seems like a desirable feature, and it would be feasible to implement because the compiler can see which `interface`s are used where.

Perhaps some form of implementation inheritance for `interface`s could be

supported as well.