Program Blocks as a Procedural Context

Jay Lawrence
02/05/2003

1 Purpose

The purpose of this proposal is to redefine the program block as a procedural context rather than the hierarchical context it is defined as today. It allows program blocks to occur anywhere an initial block can occur in SystemVerilog, but preserves the simulation semantic that the program block is run during the proposed "verification phase".

The program block is not left in the verification phase because I believe the verification phase belongs in SystemVerilog, but because a new working group has been formed to discuss the simulation cycle semantics. If, in time, the simulation semantics are resolved such that the verification phase is removed, then a follow-on proposal will be made to eliminate the program block all together as it will be identical to an initial block.

2 Problem Statement

Section 15.1 of the draft 2 LRM provides the introduction to the program block. It makes a series of important points:

- "The module is the basic building block in Verilog."
- "The testbench … [is]… modeling the large environment in which a device needs to be verified."
- "A lot of effort is spent in"
    - "getting the environment initialized and synchronized"
    - "avoiding races between the hardware and the test-bench"
    - "automating the generation of input stimuli"
    - "in reusing existing models and other infrastructure"

The introduction goes on to explain that a module and initial block would be an obvious choice for modeling a testbench except for the following distinctions:

- "The communication between the test-bench and the design takes place via special *ports* that … specify a clocking scheme"
- "It provides for race-free cycle and transaction-level abstractions as well as event abstractions"
- "It indicates the special nature of the *test-bench module*, thus enabling specialized execution semantics for all elements within the program"

The remainder of this proposal will show that the goals outlined above are better met by isolating the special semantics of a program block to only the sequential code executing in the program block, and allowing the natural power of Verilog to model the test environment. The specific goals met by moving the program block to a purely procedural context are:

- The testbench is a modeling a large (often hardware) environment and the power of modules to represent this environment hierarchically should be exploited.
- The initialization of a testbench is exactly the same as initializing a design. A module containing both initial and program blocks can easily accomplish this.
- The extension of clocking domains could well be useful for not-only test benches, but system-level models so although it is used by program blocks. It too is a general extension which should be available in any module.

The other goals given in the introduction are specific to the simulation semantics of the sequential code inside a program block or completely unrelated to the program block. This proposal does not attempt to address:

- Avoiding races between the hardware and the test-bench
- Generating input stimuli
- Transaction and event-level abstractions
- The specialized execution semantics of the program

3 Analysis

Chapter 15 of the draft LRM goes into great detail to specify a lot of information that is a complete duplication of content in modules. This section goes through each of these details and shows why the overlap with modules is unnecessary and would be more clearly replaced with a direct re-use of modules.

3.1 The program construct (Section 15.2)

The syntax of a program is given as:

```
program program_name ( list_of_ports )
      program_declarations
      program_code
endprogram
```

Examining the elements of this syntax one at a time, the `list_of_ports` is exactly the same as the list of ports on a module.

The `program_declarations`[1] syntactic element is never defined anywhere in the draft LRM so it is unclear if this is different than what can be declared in a program vs. a module. However, it is clear that at least parameters and all kinds of SystemVerilog variables (including classes) should be permitted. Declaration of nets in a testbench would potentially disrupt the proposed simulation semantics, but would also make modeling environments simpler so I propose that the items that can be declared in a program be identical to those in a module.

The last syntactic element is the `program_code`. The syntax for `program_code` is also not given anywhere in the donation, but I assume that it is any sequential code. I propose the syntax:

```
program_construct ::= program statement
```

The statement could of course be **begin … end**, including a named block which could contain declarations local to the block (or declarations in an unnamed block as allowed now by SystemVerilog).

Section 15.2 goes on to explain that a **program** can be seen as a special type of Verilog module (a module with a testbench attribute), and that it can be instantiated at the proper hierarchical level and connected in the same manner as any other module. This use of a **program** as a module instance is very powerful, but loses some of this power when module instances are not allowed as children of the program. As stated in the original goals above, the testbench is modeling a complex environment and attempt to reuse verification IP and infrastructure. Creating modules that represent the various verification components and using the natural hierarchy and interconnect mechanisms of Verilog to reuse them and connect them to the design is extremely powerful and intuitive to the Verilog user.

The current lack of a static hierarchical interconnect in the program requires that all data be automatic, even if it is constructed at initialization time, and never destroyed. This object that has a lifetime of the entire simulation is best modeled as a hardware component, not a dynamically allocated object. This also creates a natural hierarchy which precisely mimics the hierarchy in which the DUT will eventually be used.

[1] Editorial note: In the draft LRM this is misspelled as ‘program_declarartions’

Lastly, section 15.2 discusses the restrictions on "test-bench constructs and data types" indicating that they must be restricted to the "procedural context". It is important to clarify that this contexts includes always, initial, task, function, and program contexts. The power of data types added in SystemVerilog is useful not only for testbenches but for system-level models. By making the program a procedural context, the intent of this usage is preserved, the overlap with module is eliminated, and the power of hierarchy can be exploited in testbenches.

3.2 Static Data Initialization (Section 15.3)

The special initialization semantics required for SystemVerilog 3.1 objects would be unnecessary if programs were just another procedural context. To my knowledge, all Verilog simulators execute all **initial** blocks before all **always** blocks at time zero of simulation. This is necessary for backward compatibility with Verilog-XL. As long as **initial** blocks were executed before **program** blocks as well, then static data could be initialized in **initial** blocks in the same way it is initialized in the design. The special case initialization semantics could be eliminated including the complications added by dynamically allocating memory as a part of static initialization (because it would now be done at time 0 of runtime).

3.3 Scope and Lifetime (Section 15.4)

This entire section goes to great pain to express that **program** and **module** have identical rules for scope and lifetime. If this is the case, then why does **program** exist as a hierarchical context?

3.4 Multiple Programs (Section 15.5)

This section allows for multiple programs and multiple instances of the same program. By redefining the program as a sequential construct, this becomes even simpler. If the first thing a program would do is bring multiple cooperative threads into existence using fork/join, they could instead be declared as multiple program blocks inside the same module, cooperatively sharing the static data in that module. Multiple instances of a program could be accomplished by instantiating the module in which the program is declared.

3.5 Other sections 15.6-15.9

These sections deal with the simulation semantics of programs which are beyond the scope of this proposal. However, the simulation semantics are specific to code that executes sequentially within a program. This proposal preserves this sequential execution and thus allows these special semantics to be preserved if the working group decides that should happen.

4 Conclusion

The document proposes that declarations, ports, parameters, and instantiation of programs is in no way unique from the same constructs in modules and thus the program should be eliminated as a hierarchical construct. The program should become instead, a procedural context which executes at the same simulation time as **always** blocks with whatever simulation semantics is agreed on by the new working group.

The following is a list of specific changes which would need to be made if this proposal is adopted:

- Replace all references to instances of programs and the phrase '**module** or **program**' with modules.
- Remove all references to `program_declarations`
- Add `program_construct` to `module_or_generate_item`
- Define `program_construct` as:

```
program_construct ::= program statement
```