An analysis of the "logic" data type by Cliff Cummings - 20021209

The logic data type (and now other new data types) have been the subject of much debate and confusion. Most of the debate has surrounded the introduction of (a) new keywords, (b) non-hardware behavior and (c) the lack of multi-driver capability. The confusion has mostly surrounded how logic, bit, real, structs, enums, and ints really behave and when they should be used in place of existing Verilog data types.

When logic was first proposed for SystemVerilog, most of the early committee thought we had our universal data type and we could finally abandon the separate wire and reg data types. We soon learned this was not the case.

This document proposes the following:

- allow logic to have multiple drivers (same with bit, real, struct and int).
- do not permit last-assignment wins behavior outside of the current scope.
- (assuming the above two conditions) make logic the default type for SystemVerilog
- add a new type called ulogic (unresolved logic) that does the exact same thing as logic but prohibits multiple drivers - OR - a new option: `default_nettype unresolved (the second alternative means that we would not need new keywords: ulogic, ubit, ureal, ustruct, uint).
- Define resolution tables for each of the new types.
- Add a new compiler directive: `default_resolution bit 0 (or 1)
- Possibly add user-defined default resolution capability for real, struct and int, with permitted legal values

Details and descriptions follow:

(1) As currently defined in the SystemVerilog 3.0 Standard, logic is not, and cannot be the default data type in SystemVerilog (at one point, logic was going to be the default data type in SystemVerilog, until we realized that many existing designs would break due to replacing wire with logic as the default type).

The logic type has two distinct behaviors that would break existing designs: (a) the logic type does not permit multiple drivers - this breaks any existing design that uses an implicit net variable with multiple drivers, and (b) the logic type has last-assignment-wins behavior at higher levels of hierarchy if the same logic variable is procedurally assigned from two different procedural blocks in two lower-level modules. Existing Verilog models resolve procedural assignments outside of the module where the procedural assignment was made - again breaking existing Verilog models.

(2) logic does not behave like any known real hardware

The last-assignment-wins behavior for multiple lower-level procedural assignments is behavior that does not match any known hardware.

Consider the case of two instantiated flip-flop models as shown below:

```
module top;
  logic q, d1, d2, clk, rst_n;

  dff u1 (.q(q), .d(d1), .clk(clk), .rst_n(rst_n));
  dff u2 (.q(q), .d(d2), .clk(clk), .rst_n(rst_n));
endmodule

module dff (
  output logic q,
  input logic d, clk, rst_n);

  if (!rst_n) q <= 0;
  else        q <= d;
endmodule
```

In this design, an engineer mistakenly connected the q-outputs of two flip-flops together in the top module. This design has a race condition and last q-assignment wins, which is not anything close to how the real hardware works. In Verilog, the outputs would resolve on a net and conflicting values would show up in a simulation as an unknown (X). The X shows the engineer there is a problem in the design that will not be found using SystemVerilog logic variables until either a lint-tool shows the problem or until after synthesis when a gate-level simulation is performed.

I can think of no real hardware that would behave like this. I can think of no verification strategy that would want to take advantage of this "feature." If a design really needs to set a variable from two or more modules (again, I consider this to be generally a very bad design practice) cross-module-references can be used to change the variable from another module.

Some Co-Design engineers say that all regs should be replaced with logic. Peter Flake said no RTL models should use logic. There is real confusion surrounding the capabilities and pitfalls of the logic variable.

(3) Multi-driver prohibition - `default_nettype unresolved

For better or for worse, the default Verilog data type (wire) allows multi-driver resolution. Had Verilog enforced single-driver restrictions on "wire" and permitted multi-driver resolution on "tri," we could have easily changed this now, but that is all ancient history.

Verilog engineers know that wires can have multiple drivers. Changing this mindset because somebody wants to restrict multi-driver variable usage seems somewhat arbitrary. Nevertheless, if we add a `default_nettype unresolved capability, we could satisfy both sides of this argument rather easily.

We would now have:
`default_nettype logic // this would be the default and permit multi-driver assignments
`default_nettype unresolved // to prohibit multi-driver assignments
`default_nettype none // for those misguided souls who want to declare everything due to a strong software or VHDL background
Of course, we would still have all of the other existing `default_nettype capabilities for other types of design.

(4) logic variables are generally the required data type for interfaces

One unique feature of the logic type is that a logic-type variable can be assigned by either a procedural assignment, or be driven by a continuous assignment or any other driver type.

Since an interface is used to make connections between two or more ends of a variable, the input-side needs to see a wire, while the output-side is frequently driven from a procedural assignment.

A wire type can only be used in an interface (and must be used in an interface) if there are multiple drivers, such as a crossbar design or a bi-directional bus.

The reg type can only be used in an interface if the reg-variable does not go to an interface port (if the reg-variable is used for intra-interface assignments.

(5) logic variables must be declared

Because logic cannot be the default data type for Verilog, all logic variables must be declared. For those of us who are looking for fewer declarations in a design, the logic type does not achieve this goal. It seems rather silly that all of my output, intput and inout declarations carry the extra characters "logic" so I can ignore how the assignment was made to the variables.

(6) logic was added to SystemVerilog to add an orthogonal 4-state variable, along with real and bit

This was a rude discovery. The real, bit, struct, int and enum data types also have the non-hardware last-assignment-wins behavior at upper level modules. If an engineer accidentally ties together the outputs of two D-to-A converters, the last conversion will win. This again does not represent any known real hardware behavior.

Othogonality is important if you want to switch between logic and bit data types for simulation speed or alternate verification strategies. If the compiler permits an upper level logic variable, and if the logic variable was really a typedef that can be changed to a bit type, it is very upsetting to allow a design to compile with a logic type but cause a syntax error to occur when compiled with a bit type. This is why orthogonal data types are important.

(7) `default_nettype ulogic

Again, whether or not Verilog ever should have allowed multi-driver resolution on the default data type is conceptually not that important to me, except that the default data type in Verilog (wire) has always permitted multi-driver resolution. An argument could be made that the default data type, wire, never should have permitted multi-driver resolution and that the "tri" data type (which is the exact same thing as wire) should have been required to allow multi-driver resolution. Unfortunately, that is now ancient history. To change the default behavior of the default data type in Verilog would break countless existing Verilog designs.

Perhaps a reasonable alternative would be to invent a new data type, perhaps called ulogic (unresolved logic - similar to the VHDL std_ulogic type) which does not permit multi-driver resolution. Then any design team that wanted to prohibit multi-driver assignments could set the compiler directive: `default_nettype ulogic. The directive could be set just once in the first-compiled module to enforce the no-multi-driver rule. With this directive set, SystemVerilog users would have to declare all multi-driver variables as type logic.

This technique would give the extra checking desired by design teams that want to ensure that no variable is driven by more than one source, plus it would make it very easy to find all of the multi-driver variables in the design by enabling `default_nettype ulogic and then compiling the design. The compiler would show syntax errors for all multiple-driven variables. A clever vendor would create a +ulogic_warn command line switch to do the warn of multi-driver variables without requiring the `default_nettype ulogic compiler directive.

I actually like the suggestion under bullet #3 above, to just add a `default_nettype unresolved

(8) Should "logic" be removed from SystemVerilog and change reg to permit continuous assignments and remove the requirement to declare 1-bit regs (as suggested by some of the Cadence committee members?)

The logic type has been part of the Superlog language for a couple of years now and there are a number of Superlog customers that have written a significant amount of code using "logic." I know this is not necessarily an argument to add logic, but maybe Cadence should speak to some of the Intel engineers I have talked to and see if his company ever sells another Cadence Verilog tool to Intel ever again!

Besides, I like the idea of a clean break from the old "reg" variable that was not always a real register, and now we are proposing allowing continuous assignments to "reg" that NEVER infers register logic. Despite the probability that "logic" has been used in a number of older Verilog designs, I still favor using this keyword to describe the unified capabilities of "reg" and "wire."

(9) Dave Rich claims that "we could have just extended the functionality of reg to behave like other SV variables and still have been 100% backward compatible. However people ... thought this was too

> dramatic a change; people would get lost if they didn't get their dosage of "illegal left-hand-side assignment" error messages. So logic was created to distance itself from the legacy "reg" behavior.

I believe this claim is unfounded. Extending reg to allow wire-like assignments is very similar to a proposal I made in my HDLCON 2000 paper, except that I proposed modifying wire to permit reg-assignments (in order to make most declarations optional).

If the earlier proposal were to change reg to behave like logic, there certainly would have been objections because this change would break existing designs, not because we still wanted to see "illegal LHS assignment errors," which is exactly what we are trying to remove.

Regards - Cliff