# Logic vs Wire in SV – some misconceptions

- Mitu Raj, chip@chipmunklogic.com, Feb-2024

## Background Story

Often when I browse across articles, blogs, and discussion forums in professional platforms, I come across this comparison between *wire* and *logic* in SV. Many of them got me baffling (but thanks to Dave Rich, for correcting people at many places; the living LRM!). Some of the claims used to be true, but not anymore. This white paper's objective is to break down the popular misconceptions and outdated facts about this comparison. The internet may have misguided many of us. So, let's try to settle this, once and for all!

## Logic and Wire – the origin of comparison

The *wire* and *logic* have been the subject of much debates, confusion, and comparisons ever since SV introduced *logic* in **SV3.0** LRM in 2002. The most popular statement surrounding the comparison is:

*"wire can have multiple drivers, but logic cannot."*

The *logic* cannot have multiple drivers. This used to be true, but not anymore. This is in fact an outdated argument.

Cliff Cummings in one of the white papers in 2002 [1] analyzed and pointed out the flaws of *logic* in **SV3.0 (2002)**. The *logic* being not having multiple drivers was the first flaw he argued about *logic*, which was odd for what is supposed to be the universal data type in the SV standard.

The argument remained true for the updated standard, **SV3.1 (2004)** as well. The *logic* was introduced in the standard as:

*"Verilog-2001 has net data types, which can have 0, 1, X or Z, plus 7 strengths, giving 120 values. It also has variable data types such as reg, which have 4 values 0, 1, X, Z. These are not just different data types, they are used differently. SystemVerilog adds another 4-value data type, called logic."*

The *wire* is a **net data type**, *logic* is a **variable data type**. Both are data types. Hence, the use cases and comparisons between them made sense. But this made *logic* look vaguer in the use cases, adding only more confusion along with the companions *wire* and *reg*. In conclusion, *logic* was not so universal in application as analyzed by Cliff [1].

## Where does the comparison stand today?

Fast forwarding to the latest SV standard: **SV IEEE Std 1800-2017**, *logic* has evolved. SV now has distinction between **data types** and **data objects**. The *sec 6.2* says:

*"SystemVerilog makes a distinction between an object and its data type. A data type is a set of values and a set of operations that can be performed on those values. Data types can be used to declare data objects or to define user-defined data types that are constructed from other data types. A data object is a named entity that has a data value and a data type associated with it, such as a parameter, a variable, or a net."*

Data objects can be divided into two groups: *variables* and *nets*. Both *variables* and *nets* should be declared with their associated data type. The *logic, int, bit* etc are data types. The *wire* is a *net* type data object. Everything which is declared as *var,* is *variable* type data object. The differences and use cases of both are in detail explained in the LRM [2]).

For the context of this paper, the above information is enough to summarize our conclusion on *wire* and *logic*:

*"The wire is a data object, while logic is a data type which can be used to declare data objects like wire or var."*

For e.g., you can declare:

```
  wire logic [3:0] data ;  // data is a net of type logic
  var  logic [3:0] data ;  // data is a variable of type logic
```

The *wire* can have multiple drivers and associated resolution function, just like how the legacy Verilog has always supported. Hence, leading to the most important conclusion:

*"logic can have multiple drivers, if declared as wire."*

So much for the multiple driver arguments between *logic* and *wire*!

The SV IEEE Std 1800-2017 standard was published quite long ago. Still, the misconception that *logic* doesn't support multiple drivers exists around in the forums. Let me present a fallacious code snippet which seems to back this argument.

Consider the following code snippet:

```
// Case 1
output logic [3:0] data ;
……
assign data = 1'z ;
assign data = a   ;
```

The compiler will throw multiple driver error on *data* if compiled in SV-2017!

The code snippet is now changed to:

```
// Case 2
output wire [3:0] data ;
……
assign data = 1'z ;
```

```
assign data = a   ;
```

The compiler will throw no error in this case. And *data* is synthesisable to a tristate bus.

This seems to contradict our earlier conclusions on *logic* from the LRM. But it doesn't really contradict if you observe deeper. To understand the reason behind the compiler error, we have to understand SV's implicit behavior on signal declarations.

## Implicit behaviors in SV – be cautious!

In SV, an internal signal or port can be declared implicitly or explicitly. Implicit declaration happens when you decide to omit some information on signal/port to avoid being too verbose. Explicit declaration happens when you add complete information on the signal/port. Typical explicit declaration looks like:

```
// Port declaration
<port direction> <net type or variable type> <data type> <port name>

// Internal signal declaration
<net type or variable type> <data type> <signal name>
```

SV allows to skip one or more of the declaration parameters. This leads to implicit behaviors as described in detail in the SV standard LRM. Let's consider only what's required in the context of this paper.

1. If *port direction* is omitted, it defaults to *inout* port.
2. If *net/variable* type is omitted for an input port, it defaults to *net* type: *wire*
3. If *net/variable* type is omitted for an output port, it depends on the data type.
    a. If data type is omitted as well, it defaults to *net* type: *wire*
    b. If data type is declared explicitly, it defaults to *variable* type
4. If data type is omitted for any signal or port, it defaults to *logic* type. (Universal data type!)
5. If *net/variable* type is omitted for an internal signal and is not used in port connections/port mapping, it defaults to *variable* type.

Let's go back to the earlier example (Case 1).

```
output logic [3:0] data ;  // This is equivalent to output var logic [3:0] data
```

SV IEEE Std 1800-2017 *sec 6.5* says:

*"Variables can be written by one continuous assignment or one port."*

The *data* is of *variable* type. Therefore, it doesn't support multiple drivers through more than one continuous assignments. Hence, the compiler would flag error.

Case 2:

```
output wire [3:0] data ;  // This is equivalent to output wire logic [3:0] data
```

The *data* is of *net* type. It supports multiple drivers. Hence, the compiler flags no errors.

Therefore, we can conclude that whether a signal/port supports multiple driver, depends on whether it is declared as *net* or *variable*. It has nothing to with the data type.

## Conclusion

Comparing *logic* and *wire* is like comparing apples and oranges by today's SV standards. In fact, it's not even apples and oranges. The *logic* is a data type, while *wire* is a data object of *net* type, which is described by the data type. If *wire* is assumed to be a car, *logic* tells if it's a hatchback or SUV!

The *logic* tells nothing about the multiple driver capability of a signal. It depends solely on whether the signal is *net* or *variable*.

It is always good to stay updated with the latest SV LRM to not fall into misrepresentations and misconceptions. When describing RTL, it is good to be wary of implicit declarations and assignments in SV and be explicit whenever in doubt. And whenever in doubt: refer to and trust only the LRM!

## Appendix

The *variable* type has some confusing use cases. It supports having multiple drivers when driven by two or more procedural blocks like *always@()* and *always_ff()* blocks. Except for *always_comb* blocks, where it is not waived by SV compiler by default.

```
always_ff @(posedge clk) b <= ~a ;
always_ff @(posedge clk) b <= a & c ;  // Designer added this accidentally
```

This piece of code leads to race conditions and the compiler may not even complain because this is legal in SV. The simulator's behavior could be unpredictable here. This vague behavior is even mentioned in the standard *sec 6.5*:

*"Variables can be written by one or more procedural statements, including procedural continuous assignments. The last write determines the value."*

I personally never liked this feature of *logic* because this behavior doesn't match with any known synthesisable hardware!

## Disclaimer

It's possible that SV will update *logic* in future to address its flaws or update the use cases. So, I want to re-state that all the content presented this white paper is in adherence to the latest SV IEEE Std 1800-2017 standard, and may get outdated in future with newer standards.

## References

[1] An analysis of the "logic" data type by Cliff Cummings – in 2002:
https://www.accellera.org/images/eda/sv-ec/att-0319/01-Logic_20021209.PDF
[2] SV IEEE Std 1800-2017: https://fpga.mit.edu/6205/_static/F23/documentation/1800-2017.pdf