A Proposal To Remove Those Ugly Register Data Types From Verilog

Clifford E. Cummings
Sunburst Design, Inc.
15870 SW Breccia Drive
Beaverton, OR 97007
cliffc@sunburst-design.com / www.sunburst-design.com

Abstract

One of the most confusing concepts in the Verilog language is, when is a variable a "reg" and when is it a "wire?" Although the rules for declaring registers and wires are really very simple, most new and self-taught Verilog users don't understand when and why one type of declaration is required over another.

This paper will detail the differences between register and net data types and propose an enhancement to the Verilog language that would eliminate the need to declare register data types altogether.

The proposal

Proposal:

- Remove the requirement to declare scalar register data types and replace vector register data types with vector net declarations.
- Report a syntax error whenever a procedural assignment is made to a variable that is also being driven to a value by a continuous assignment or instance port.

Reasons:

- To remove an annoying and confusing declaration requirement of the Verilog language.
- To reduce and simplify the required number of Verilog declarations.

Introduction

The concept of register and net variables in Verilog is largely misunderstood.

A VHDL process is roughly equivalent to a Verilog always block and a VHDL concurrent signal assignment is roughly equivalent to a Verilog continuous assignment, but VHDL does not require different data type declarations for process and concurrent signal assignments. In VHDL, "signals" are commonly used in place of both Verilog register and net data types.

Why are Verilog users burdened with these two distinct data types?

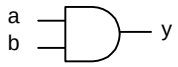
Register & net declarations - simple rule

In Verilog, the register data types include: reg, integer, time, real and realtime.

In Verilog, the net data types include: wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1 and trireg.

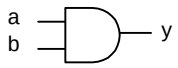
Let's look at two simple Verilog examples to help understand the declarations of register and net data types.

```
module and2a (y, a, b);
  output y;
  input  a, b;

  assign y = a & b;
endmodule
```



Example 1 - Valid continuous assignment with no wire declaration

```
module and2b (y, a, b);
  output y;
  input  a, b;
  wire   y;

  assign y = a & b;
endmodule
```



Example 2 - Valid continuous assignment with wire declaration

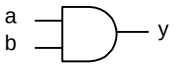
In Example 1, a 2-input and gate is modeled using a continuous assignment statement. The y-output does not have to be declared because it is a 1-bit wire. Example 2

is the exact same 2-input and gate with optional "wire y;" declaration.

In Example 3, we decide to replace the continuous assignment with an always block, but when this code is compiled, Verilog compilers report a syntax error of the form "illegal left-hand-side assignment" because we forgot to change the "wire y;" declaration to "reg y;"

If the problem-declaration is changed to "reg y;" the model compiles and simulates correctly.

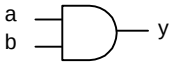
```
module and2c (y, a, b);
  output y;
  input  a, b;
  wire   y;

  always @(a or b) y = a & b;
endmodule
```



Example 3 - "illegal left-hand-side assignment" add the declaration: `reg y`

Now if the always block from the 2-input and gate of Example 3 is changed back to a continuous assignment as shown in Example 4, the Verilog compiler will again report a syntax error, but this time the message will be of the form "illegal assignment to net" because we forgot to change the "reg y;" declaration to "wire y;" Very annoying!

```
module and2d (y, a, b);
  output y;
  input  a, b;
  reg    y;

  assign y = a & b;
endmodule
```



Example 4 - "illegal assignment to net" either remove the "`reg y`" declaration or change it to "`wire y`"

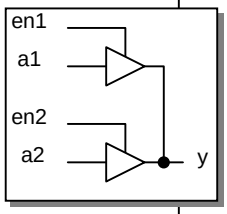
Simple rule: In Verilog, anything on the left hand side (LHS) of a procedural assignment must be declared as a register data type. Everything else in Verilog is a net data type. No exceptions!

Why differentiate nets & registers?

Why differentiate between net and register data types in Verilog? The answer to this question seems to be, data type checking is an easy way to recognize the erroneous assignment of the same variable from both continuous and procedural assignments.

Continuous assignments setup drivers on a net. Multiple drivers can drive the same net as shown in Example 5.

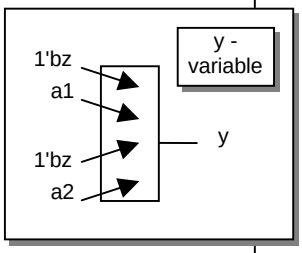
```
module drivers1 (y, a1, en1, a2, en2);
  output y;
  input  a1, en1, a2, en2;

  assign y = en1 ? a1 : 1'bz;

  assign y = en2 ? a2 : 1'bz;
endmodule
```



Example 5 - Multiple drivers on a common net using continuous assignments

Procedural assignments, such as always block assignments, cause changes to a single behavioral variable. The multiple always block assignments of Example 6 are simply assignments to the same behavioral variable and do not setup multiple drivers. In this example, last assignment wins.

```
module drivers2 (y, a1, en1, a2, en2);
  output y;
  input  a1, en1, a2, en2;
  reg    y;

  always @(a1 or en1)
    if (en1) y = a1;
    else     y = 1'bz;

  always @(a2 or en2)
    if (en2) y = a2;
    else     y = 1'bz;
endmodule
```



Example 6 - Multiple assignments to a behavioral variable using always-block assignments

If one tries to setup a driver and behavioral assignment to the same variable, the driver requires a net declaration while the always block assignment requires a reg declaration, both to the same variable, which is a syntax error. This syntax error is one method of keeping Verilog designers from trying to make two fundamentally different types of assignments to the same variable.

```
module drivers3 (y, a1, en1, a2, en2);
  output     y;
  input      a1, en1, a2, en2;
  wire?/reg? y;

  always @(a1 or en1)
    if (en1) y = a1;
    else     y = 1'bz;

  assign y = en2 ? a2 : 1'bz;
endmodule
```

Example 7 - Illegal driver and behavioral assignment to the same variable

The code in Example 7 cannot legally declare the y-variable to be either a net type or a register type. The diagram in Figure 1 shows conceptually that the code of Example 7 is trying to both change a behavioral variable and drive the same variable with a continuous assignment.

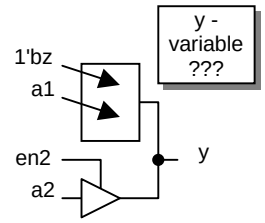


Figure 1 - Illegal variable/driver combination

If a designer really wanted to make a procedural assignment to the same variable as a net-driven variable, one could declare the LHS of the always block to be what is frequently referred to as a "shadow" register, which is a temporary register that is then driven onto a net by a continuous assignment as shown in Example 8 and Figure 2. But if you are going to do this, you might as well skip the always block assignment altogether and just make the assignment using a second continuous assignment statement.

```
module drivers4 (y, a1, en1, a2, en2);
  output y;
  input  a1, en1, a2, en2;
  wire   y;
  reg    y_tmp;

  always @(a1 or en1)
    if (en1) y_tmp = a1;
    else     y_tmp = 1'bz;

  assign y = y_tmp;

  assign y = en2 ? a2 : 1'bz;
endmodule
```

Example 8 - Multiple drivers on a common net - one shadow register assignment

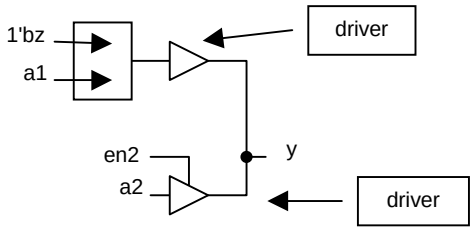


Figure 2 - Variable assigned to a driver

Conditional compilation

What if a designer wants to include conditional compilation, selecting either an always block, or a continuous assignment as shown in Example 9. The conditionally compiled 1-bit continuous assignment requires no data type declaration or can include an optional wire declaration.

The other conditionally compiled branch, the 1-bit always block assignment, requires a reg data type declaration.

Some companies have coding guidelines that require all data type declarations be placed at the top of a module, immediately after all of the I/O declarations. The conditionally compiled always-block code will violate this guideline, unless a separate conditionally compiled declaration section is added to the grouped declarations near the top of the module code (not shown).

```
module inva (y, a);
  output y;
  input  a;

  `ifdef ASSIGN
    assign #(1:2:3,4:5:6) y = ~a;
  `else
    // mid-code reg declaration
    reg    y;
    always @(a) #(1:2:3) y = ~a;
  `endif
endmodule
```

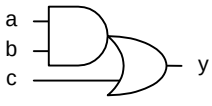
Example 9 - Conditional compilation with mid-code reg declaration

Verilog-2000 port enhancements

In Verilog-1995 [1], all register-type output ports must be declared three times:

(1) in the module header
(2) with a port declaration, and
(3) as a separate register data type.

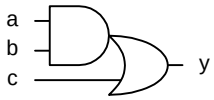
```
module and2ora (y, a, b, c);
  output y;
  input  a, b, c;
  reg    y;
  reg    tmp;

  always @(a or b)
    tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



Example 10 - Verilog-1995 and-or gate with required triple-declared port and "`reg tmp`"

Another requirement of Verilog-1995 is that any net-variable on the LHS of a continuous assignment, that does not connect to a port, must also be declared, including 1-bit nets. This inconsistent requirement of Verilog-1995 is fixed in Verilog-2000 [2]. Until Verilog-2000 is widely implemented, if the always block assignment of Example 10 is replaced with an equivalent continuous assignment as shown in Example 11, the net declaration for tmp is required.

```
module and2orb (y, a, b, c);
  output y;
  input  a, b, c;
  reg    y;
  wire   tmp;

  assign tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



Example 11 - Verilog-1995 and-or gate with required triple-declared port and "`wire tmp`"

Starting in Verilog-2000, port declaration simplification enhancements will become available.

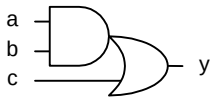
For the purposes of this paper, the following port declaration style definitions are used:

- Style #1 port declarations declare both the port direction and data type, including all of the optional data type declarations.
- Style #2 port declarations declare all port directions but only the required data types. All optional data types are omitted.

It is also possible to do a mixture of style #1 and style #2, but none of the examples in this paper show this combination.

The first port declaration enhancement in Verilog-2000 includes the ability to combine port and type declarations. Example 12 shows all of the ports declared with data types (referred to above as style #1). A separate register declaration for the y-output is not required.

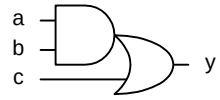
```
module and2orc (y, a, b, c);
  output reg  y;
  input  wire a, b, c;
  reg    tmp;

  always @(a or b)
    tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



Example 12 - Verilog-2000 and-or gate with double-declared port (style #1) and "`reg tmp`"

Example 13 also shows legal Verilog-2000 declarations where only the register-type port declarations include data types while all of the net-type port declarations omit the data types (referred to earlier as style #2).

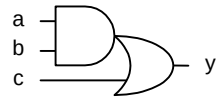
```
module and2ord (y, a, b, c);
  output reg  y;
  input       a, b, c;
  reg    tmp;

  always @(a or b)
    tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



Example 13 - Verilog-2000 and-or gate with double-declared port (style #2) and "reg tmp"

Another port-enhancement coming to Verilog-2000 is that port directions and data types will be permitted in the module header itself, making it possible to declare all of the ports just once. The anticipated way of making module-header port declarations is to code the module header with open-parenthesis followed by each port declared on a separate, subsequent line and ending with a close-parenthesis and semi-colon on a stand-alone line as shown in Example 14.

```
module and2ore (
  output reg  y;
  input       a, b, c;
  );
  reg    tmp;

  always @(a or b)
    tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



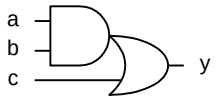
Example 14 - Verilog-2000 and-or gate with single-declared port (style #2) and "`reg tmp`"

Making all of the port declarations in the module header will guarantee that all ports will be declared at the top of the module, which the Verilog Standards Group (VSG) anticipates will permit enhanced optimization and acceleration during Verilog compilation. In Verilog-1995, port declarations can appear anywhere in a module, which means a compiler cannot recognize and report a missing port until the endmodule statement is read.

In the single-declared, enhanced-port coding styles shown in Example 14 and Example 15, the tmp variable still needs to be declared as a reg, if assigned in an always

block (Example 14), or the tmp variable can be omitted or declared as a wire, if assigned from a continuous assignment (Example 15). The y-output also requires a reg declaration in both examples.

```
module and2orf (
  output reg  y;
  input       a, b, c;
  );
  wire   tmp;

  assign tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```

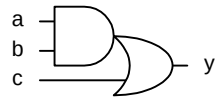


Example 15 - Verilog-2000 and-or gate with single-declared port (style #2) and "`wire tmp`"

The problem that still exists with all of the Verilog-2000 port declaration enhancements is that changing an output port or internal variable assignment from a continuous assignment to an always block still requires the enhanced port declarations to be changed to reflect the data type of the variable being modified, the same as with Verilog-1995 data type declarations.

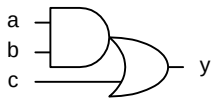
If separate register and net data type requirements are eliminated, the same enhanced port declarations as shown in Example 16 and Example 17 will be both abbreviated and legal. Note that in both examples, the declarations are identical and no net or register declarations are required.

```
module and2org (
  output y;
  input  a, b, c;
  );

  always @(a or b)
    tmp = a & b;

  always @(tmp or a)
    y = tmp | c;
endmodule
```



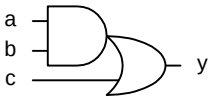
Example 16 - Verilog-2005(?) and-or gate with single-declared port (always y-output)

```
module and2orh (
  output y;
  input  a, b, c;
  );

  assign tmp = a & b;

  assign y = tmp | c;
endmodule
```



Example 17 - Verilog-2005(?) and-or gate with single-declared port (assign y-output)

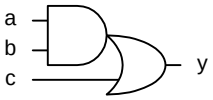
```
module and2ori (
  output y;
  input  a, b, c;
  );

  assign tmp = a & b;

  always @(tmp or c)
    y = tmp | c;
endmodule
```



Example 18 - Verilog-2005(?) and-or gate with single-declared port (always y-output)

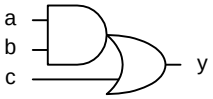
Of course, the and-or model can also be simplified in Verilog-2005(?) by combining the separate assignments seen in earlier examples into either a single continuous assignment as shown in Example 19 or into a single always block as shown in Example 20. Example 20 also shows the combinational sensitivity list operator "@*" that is used to gather all RHS variables, if-expression variables (not in this example) and case-expression variables (not in this example) into the sensitivity list. The "@*" operator is new with Verilog-2000.

```
module and2orj (
  output y;
  input  a, b, c;
  );

  assign y = (a & b) | c;
endmodule
```



Example 19 - Verilog-2005(?) and-or gate with continuous assignment

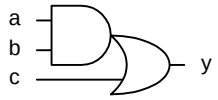
```
module and2ork (
  output y;
  input  a, b, c;
  );

  always @*
    y = (a & b) | c;
endmodule
```



Example 20 - Verilog-2005(?) and-or gate with procedural assignment

In both Example 19 and Example 20, the code has been simplified over the equivalent Verilog-1995 module, shown in Example 21.

```
module and2or1 (y, a, b, c);
  output y;
  input  a, b, c;
  reg    y;

  always @(a or b or c)
    y = (a & b) | c;
endmodule
```



Example 21 - Verilog-1995 and-or gate with required triple-declared port and one always block

Pros & cons of declaring wires

Some Verilog designers believe it is a good practice to declare all wires, including 1-bit wires, in every module were the wires exist. The apparent reasons for making all declarations is to (1) document the existence of all wires and (2) the mistaken notion that Verilog does comprehensive size checking on all declared variables.

Declaring all wires is also a habit that is developed by some engineers that have previously designed using VHDL, a language where all signal declarations are required. In VHDL, the compiler does checking between declared signals, signal sizes, and the sizes of the signals used in the actual VHDL models.

In Verilog, the same rigorous size-checking does not exist. Although some size-checking does occur, unless a bit-range is included on variables in the body of the code (not just in the declaration), much of the size-checking can be easily missed. Many variables declared as 1-bit wires and then used as buses, without referencing the bus-range in the Verilog code, will be translated into 1-bit wires where the assignment of all leading bits-positions are padded with 0's.

```
module invbad1 (y, a);
  output [7:0] y;
  input  [7:0] a;
  wire         tmp;

  assign tmp = ~a;
  assign y   = tmp;
endmodule
```

Example 22 - Undetected bad 1-bit wire declaration

The model in Example 22 is a contrived, but simple, example of an 8-bit inverter, where the internal tmp variable is erroneously declared to be a 1-bit wire. When compiled there is no syntax error or warning, and when simulated using the testbench in Example 23, the 1-bit tmp is padded with leading zeros causing the upper seven bits of the module output to always be zero.

```
module tb;
  reg  [7:0] a;
  wire [7:0] y;

  inv_module u1 (.y(y), .a(a));

  initial begin
    $monitor ("y=%h  a=%h", y, a);
        a = 8'h00;
    #10 a = 8'h55;
    #10 a = 8'hCC;
    #10 $finish;
  end
endmodule
```

Example 23 - Testbench for inverter modules

The same model using a 1-bit reg variable as shown in Example 24 suffers from the exact same problem as the 1-bit wire code of Example 22. In neither case did the presence of a wire or reg declaration assist in locating a coding mistake; indeed, it could be argued that the presence of the declarations might have masked the fact that the variables had been improperly declared.

```
module invbad2 (y, a);
  output [7:0] y;
  input  [7:0] a;
  reg          tmp;

  always @(a) tmp = ~a;

  assign      y   = tmp;
endmodule
```

Example 24 - Undetected bad 1-bit reg declaration

As a side note, even though VHDL performs all of the previously mentioned size checking, on a very large VHDL design that was completed by the author, the author noticed that he spent almost as much time debugging the pages of required signal declarations at the top-level of the design as he spent debugging actual design problems.

It is the authors opinion that limiting declarations to just bus declarations helps to concisely show which identifiers should be multi-bit in width while eliminating unneeded and verbose 1-bit declarations that tend to fill space and mask the existence of internal buses. The author believes that linting tools are best suited to examine sizes and report potential problems. The author acknowledges that other skilled designers hold the opposite opinion, that all variables should be declared.

In Example 25 and Example 26, the appropriate internal bus declarations have been made and both models simulate correctly.

```
module invgood1 (y, a);
  output [7:0] y;
  input  [7:0] a;
  wire   [7:0] tmp;

  assign tmp = ~a;
  assign y   = tmp;
endmodule
```

Example 25 - Good 8-bit wire reg declaration

```
module invgood2 (y, a);
  output [7:0] y;
  input  [7:0] a;
  reg    [7:0] tmp;

  always @(a) tmp = ~a;

  assign      y   = tmp;
endmodule
```

Example 26 - Good 8-bit wire reg declaration

Net and register differences

There are some significant differences between Verilog net and register data types. Figure 3 shows a table that lists important differences between net and register data types.

	Net types	Register types
Verilog strengths	Yes	No
Uninitialized value	HiZ	X (unknown)
Multiple assignments	Combination of all driven values	Last assignment wins
Types allowed to be declared with a range	wire, tri, wor, trior, wand, triand, tri0, tri1, supply0, supply1, trireg	reg
Types with implied range (excluding 1-bit values)	none	integer (32-bits), time (64-bits), real, realtime

Figure 3 - Net and register differences

Handling existing register types

In order to implement the reg-removal enhancement, a plan must be put in place to make this enhancement backward-compatible with existing register-type declarations. All existing models with all Verilog-1995 and Verilog-2000 register type declarations need to be properly read and simulated.

To show how Verilog compilers might treat the different register data types to be backward compatible, Figure 4 shows a table of possible implementations.

Register type	Net type implementation?
reg	wire
reg [msb:lsb]	wire [msb:lsb]
integer	wire signed [31:0]
time	wire [63:0]
real*	* only assigned in a procedural block
realtime*	

Figure 4 - Net implementations of existing register types

The above integer, time, real and realtime keywords could still be used to imply certain data types with certain ranges. The integer declaration could also infer the signed net data type that was added to Verilog-2000. Real and realtime data types might not have meaning when rendered as wires so assignments to these variables might continue to be restricted to procedural blocks, with the possible exception of Verilog-AMS.

Internally, Verilog compilers could continue to treat all data types the same way it does now. The difference is that Verilog should be able to infer the appropriate internal data type from the context of the code. 1-bit wire variables assigned inside of an always block can still go unknown at the beginning of a simulation if not assigned, and wire variables assigned inside of a procedural block can still be implemented without Verilog strengths.

There might be certain minor problems with the above implementation proposals, but the author believes that these are minor details that can worked out by the IEEE Verilog Standards Group.

Existing register data type declarations could for the most part be ignored except as they pertain to whether or not a variable is a signed variable (integer), implied bus widths (integer - 32 bits wide / time - 64 bits wide) or explicit bus widths (reg [msb:lsb]).

A new type of syntax check

Instead of forcing users to make distinctions between data types used in procedural blocks and data types used outside of procedural blocks, why not define a syntax error whenever the same variable is assigned both inside and outside a procedural block.

One potential problem that exists with the proposed reg-removal enhancement is that the elimination of required net and register data types would allow designers to make both driver-type assignments and behavioral-type

assignments to the same variable. This should not be permitted.

Implementation of the reg-removal proposal should be accompanied by a new type of syntax check, one that determines which variables are assigned from a procedural block and which are not. It shall be illegal to make assignments to the same variable from both a procedural assignment and a non-procedural assignment. This is really what Verilog compilers enforce with current net and register declaration requirements.

Further assignment restrictions might include:

- It shall not be permitted to make procedural assignments to an inout port.
- It shall not be permitted to make procedural assignments to input ports of the enclosing module.
- It shall not be permitted to make procedural assignments to nets that are driven by instantiated-module output ports.

Conclusions

In conclusion, the current net and register data type requirements are both confusing and annoying. The only apparent reason to enforce these declaration rules is to keep engineers from making procedural assignments to variables that are driven from a non-procedural assignment source.

To eliminate the above problems and simplify Verilog modeling, the author makes the following proposals:

- Remove the requirement to declare register data types for procedural assignments.
- Permit assignments to net data types within procedural blocks.
- Permit optional register-type declarations for backward compatibility and for short-hand declarations
- Require compliant simulators to flag a syntax error if assignments are made to the same variable from both inside and outside of a procedural block.

References

[1] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995

[2] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Std P1364-Y2K (Draft 4)

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 19 years of ASIC, FPGA and system design experience and nine years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.

Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site:
www.sunburst-design.com/papers

(Data accurate as of March 7th, 2001)