SystemVerilog Ports & Data Types For Simple, Efficient and Enhanced HDL Modeling

Clifford E. Cummings
Sunburst Design, Inc.
cliffc@sunburst-design.com

Abstract

Verilog-2001 introduced an enhanced and abbreviated method to declare module headers, ports and data types. The Accellera SystemVerilog effort will further enhance Verilog design by abbreviating the capability to instantiate modules with implicit port connections and interface types. These capabilities and additional complimentary enhancements are detailed in this paper.

1. Introduction

To declare, or not to declare, that is the question!

Verilog-1995[1] had verbose and redundant port declaration requirements. Verilog-2001[2] introduced the "ANSI-C"-style enhancement to remove port declaration redundancy from the Verilog language. Accellera SystemVerilog proposals will further enhance port declarations with the introduction of interface declarations. The evolution of and enhancements to Verilog port declarations are detailed in this paper.

For those who prefer a requirement that all variables be declared before they are used, Verilog-2001 introduced a new "**none**" option for the **`default_nettype** compiler directive. The usage and disadvantages of this "enhancement" are also discussed in this paper.

Another Accellera SystemVerilog proposed enhancement is to permit instantiation of modules with implicit connections. This proposed enhancement is also detailed and promoted in this paper.

This paper concludes with guidelines to increase Verilog and SystemVerilog design productivity.

2. Verilog-1995: verbose module headers

Verilog-1995 had the annoying requirement that all module ports had to be declared two or three times.

The Verilog-1995 code for the muxff block diagram of Figure 1 is shown in Example 1.

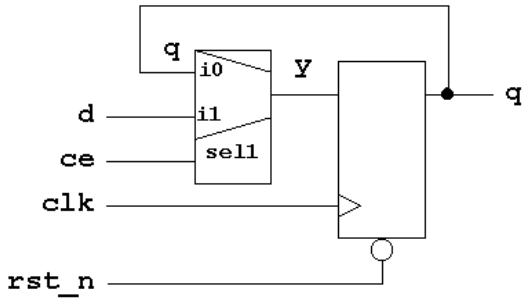

Figure 1 - muxff Block Diagram

```
module muxff1 (q, d, clk, ce, rst_n);
  output q;
  input  d, clk, ce, rst_n;
  reg    q;
  wire   y;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= y;

  assign y = ce ? d : q;
endmodule
```

Example 1 - Verilog-1995 version of the muxff module

A Verilog-1995 version of this model requires that the **q**-port be declared three times: once in the module header, once as an **output** port and once as a **reg**-variable data type. The **d**, **clk**, **ce** and **rst_n** ports must all be declared twice: once in the module header and once as **input** data ports (the port-**wire** data type declaration is not required).

Verilog-1995 requires that an *internal* 1-bit **wire** driven by a continuous assignment must be declared. The **y-wire** declaration in Example 1 - Verilog-1995 version of the muxff module

is required.

To avoid extra **wire** and **wire**-bus declarations that can exist in a large Verilog design, some engineers have adopted the strategy to make the continuous assignment in the **wire** declaration itself, as shown in Example 2[3].

```
module muxff2 (q, d, clk, ce, rst_n);
  output q;
  input  d, clk, ce, rst_n;
  reg    q;

  // Embedded wire declaration to
  // eliminate the need for separate wire
  // declaration and assign statement
  wire y = ce ? d : q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= y;
endmodule
```

Example 2 - Verilog-1995 version of the muxff module with combined wire declaration & assignment

The only disadvantage to making net-assignment declarations in a design is that the declarations tend to be dispersed throughout the RTL code, which seems somewhat strange. My own preference is to make all declarations at the top of the module and then make assignments where appropriate throughout the RTL code; however, making the dispersed net-assignment declarations throughout the RTL code is a reasonable means to reduce all of the extra net declarations in a model, a goal that I support.

The only other problem associated with making net-assignment declarations is when a right-hand-side variable is required in an assignment before it is declared. In those cases, the separate net or net-bus declaration is required.

The requirement to make all of the extra port and extra net declarations seemed to be both verbose and redundant.

3. Verilog-2001: "ANSI-C" style ports

Verilog-2001 introduced an abbreviated module port declaration enhancement, often referred to as "ANSI-C" style port declarations, where each module port could be declared just once and include the port position, port direction and port data type all in a single declaration.

The Verilog-2001 code for the muxff block diagram of Figure 1 is shown in Example 3.

```
module muxff3 (
  output reg q,
  input      d, clk, ce, rst_n);

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 0;
    else        q <= y;

  assign y = ce ? d : q;
endmodule
```

Example 3 - Verilog-2001 version of the muxff module

The Verilog-2001 version of this model has combined the **q** port-header, port-direction and data type into a single declaration. The other ports have similarly been combined into a single port-header, port-direction declaration (the port-**wire** data type declaration is again not required).

Also note that the **y-wire** declaration is not required in the Verilog-2001 version of this model. Verilog-2001 internal **wire** declaration requirements are discussed in the next section.

4. Verilog-2001: `default_nettype none

As noted in section 2, 1-bit internal wires driven from continuous assignments had to be explicitly declared in Verilog-1995. Verilog-2001 removed this inconsistency from the Verilog language.

Verilog-2001 also introduced the **`default_nettype** compiler directive option "**none**." This option forces Verilog-2001-compliant compilers to require that all **wire** declarations be explicitly declared. Some engineers prefer the requirement that all **wire** declarations be explicitly made while others would prefer to do away with the additional verbiage and potential source of misspelled identifiers and keywords.

By default, the implied net type of a module is **wire**. The Verilog-1995 Standard allowed a different default net type to be specified by adding the compiler directive **`default_nettype** and then selecting an argument from one of the following list: **wire**, **tri**, **tri0**, **wand**, **triand**, **tri1**, **wor**, **trior** and **trireg**.

Verilog-2001 added a new **`default_nettype** option, "**none**." Specifying a **none** option tells the compiler that all net variables must be declared and that no default type is inferred.

The debatable intent of this option is to assist engineers in finding typos more easily since every identifier in a design must have a corresponding declaration.

Consider the circuit of Figure 2. This is the intended logic for a simple design.

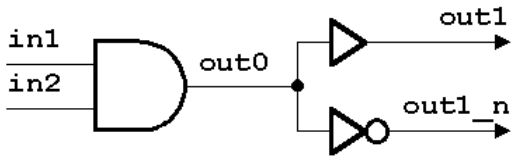

Figure 2 - Test-module intended logic

The Verilog code of Example 4 was intended to implement the circuit of Figure 2, but two common mistakes were made: (1) the output of the buffer was labeled **`out1`** (letter "**`l`**") instead of **`out1`** (digit **`1`**), and (2) the input of the inverter was labeled **`out0`** (letter "**`O`**") instead of **`out0`** (digit **`0`**). With implicit **`wire`** data types, this circuit compiles without error but will simulate incorrectly due to two disconnects in the circuit as shown in Figure 3.

```
module testmod1(
  output out1, out1_n,
  input  in1, in2);

  and u1 (out0, in1, in2);

  // Typo: should be out1 (1 not l)
  buf u2 (outl, out0);

  // Typo: should be out0 (0 not O)
  not u3 (out1_n, outO);
endmodule
```

Example 4 - Verilog testmod1code with no `default_nettype compiler directive

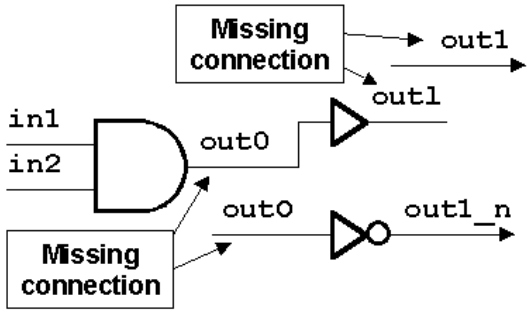

Figure 3 - Verilog testmod1 actual logic

The Verilog code of Example 5 includes the new Verilog-2001 option "**`none`**" for the compiler directive, **`` `default_nettype ``**.

This design will fail compilation with two syntax errors indicating that the identifiers "**`outl`**" and "**`outO`**" have not been declared (see Figure 4). Because the errors were caught during compilation instead of simulation, the theory is that this design is easier to debug.

```
`default_nettype none
module testmod2(
  output wire out1, out1_n,
  input  wire in1, in2);
  wire out0;

  and u1 (out0, in1, in2);

  // Typo: should be out1 (1 not l)
  buf u2 (outl, out0);

  // Typo: should be out0 (0 not O)
  not u3 (out1_n, outO);
endmodule
```

Example 5 - Verilog testmod2 code with `default_nettype none compiler directive

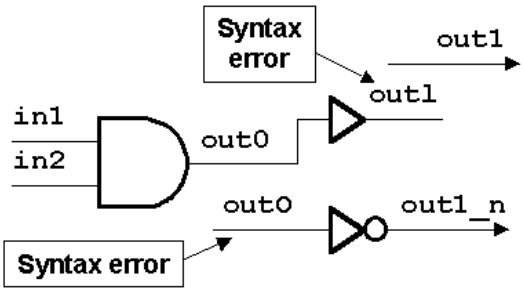

Figure 4 - Verilog testmod2 syntax-error logic

Unfortunately, additional declarations provide additional opportunities to introduce errors. A problem with the Example 6 Verilog code is that the model is actually correct but in the process of adding the additional **`wire`** declarations to satisfy the **`` `default_nettype none ``** option, three typos actually occurred in the wire declarations (**`outl`**, **`ou1_n`** and **`outO`**).

This design will fail compilation with three syntax errors indicating that the identifiers "**`outl`**," "**`out1_n`**" and "**`outO`**" have not been declared (see Figure 5). These three identifiers are actually correct but the declarations for these identifiers are wrong. The error messages have actually hidden the real errors in this design, which occurred in the declarations.

```
`default_nettype none
module testmod3(out1, out1_n,
                in1, in2);
  output out1, out1_n;
  input  in1, in2;
  wire outl, ou1_n; // Typos
  wire in1, in2;
```

```
  wire out0;         // Typo

  and u1 (out0, in1, in2);
  buf u2 (out1, out0);
  not u3 (out1_n, out0);
endmodule
```

Example 6 - Verilog testmod3 code with declaration errors

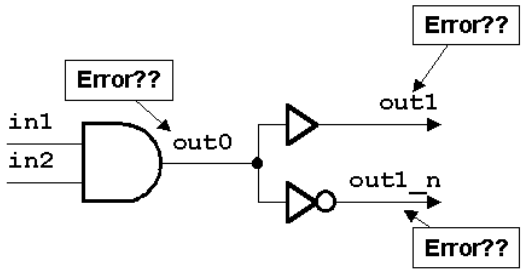

Figure 5 - Verilog testmod3 correct logic (but does not compile)

VHDL has a similar requirement that all identifiers, including 1-bit signals, must be declared. My experience in doing top-level VHDL ASIC designs has proven to be frustrating because of this requirement.

While doing large VHDL designs, I discovered that I spent almost as much time in the top-level blocks debugging pages of signal declarations as I did debugging actual RTL bugs.

My experience has been that adding pages of declarations is almost as error prone as doing RTL design, that adding all of the declarations can be very verbose and that error messages can be very confusing (indicating a syntax error in the error-free RTL code while hiding the fact that the error is in the declaration).

Although finding typos before simulation is desirable, a more reasonable approach to the problem would be a compiler or lint tool that could scan RTL source code to insure that each identifier has at least one driving source and one receiver. Adding pages of declarations just to find typos in the RTL code (and introduce more typos in the declarations) is tedious, verbose and frustrating.

Adding the **`default_nettype none** directive to a working Verilog model is asking for trouble. Adding the directive will require all identifiers to be declared. Since the RTL code is already working, more mistakes could be introduced by adding declarations.

My experience has also generally been, the more concise the code, the fewer the errors and the greater the clarity of the design.

Guideline: do not add **`default_nettype none** to a working Verilog design.

Guideline: do not use **`default_nettype none**, ever!

5. SystemVerilog: Implicit port connections

Two exciting proposals that were passed in April 2002 by the Accellera SystemVerilog Committee provide enhanced methods for instantiating modules with implicitly connected ports to reduce the verbose overhead of listing all named ports in an instantiation.

5.1. Verilog positional port connections

Both Verilog-1995 and Verilog-2001 permit module instantiation using positional port connections.

Instantiation using positional ports simply requires listing the port connections in the correct order to match the port order of the instantiated module.

Advantage: instantiation of a module is fast and easy.

Disadvantages:

- if the module is instantiated with an incorrect port order, the simulation will fail and the task of debugging this type of bug can be time consuming.
- if the port order of the instantiated module is changed by a third party, the instantiation will also have to change.
- minor port order changes are often the most difficult to find and debug. Swapping two control signal inputs will still compile without syntax error and there will be no compilation warning. Finding this type of bug can be very time consuming and the motive for some homicides!

Guideline: when instantiating a module from another design team member, from a third party or vendor, or if you do not have direct control over the module being instantiated, do not use positional port connections.

Reason: If the author of the module decides that the port order should be changed to “make the design more readable,” all instantiations of the module will have to be updated.

This is a common guideline at many companies (companies who have experienced problems and delays when an instantiated module was changed but the change went unannounced and initially unnoticed).

5.2. Verilog named port connections

Both Verilog-1995 and Verilog-2001 permit module instantiation using named port connections.

Instantiation using named ports simply requires listing both the explicit port names of the module being instantiated and the accompanying net names that connect to the explicit ports.

Disadvantage: instantiations are twice as verbose as using positional port connections and take longer to build.

Advantages:

- the ordering of the instance ports does not cause design failures.
- if the port order of the instantiated module is changed by a third party, no instantiation changes are required.
- minor port order changes do not impact a design.
- if the port names change, the compiler gives immediate feedback, in the form of a syntax error, indicating that the specified ports do not exist.

Guideline: in general, use named port connections when instantiating modules into a design.

Guideline: show all unconnected module ports using empty named connections.

Reason: some Verilog compilers give warnings for missing ports from a named port connection list. Unnecessary warnings should be avoided.

These are common guidelines at many companies.

5.3. SystemVerilog: Implicit .name port connections (passed April 2002)

SystemVerilog will permit an abbreviated method to instantiate modules using the implicit **`.name`** port connection enhancement.

In SystemVerilog, if the net connected to a port of an instantiated model matches the name of the port on the instantiated model, the size of the port on the instantiated model and has a compatible data type to the port on the instantiated model, the matching port name can be listed just once with a leading period. This is the same syntax as named port connections only without the redundant name inside of parentheses.

Implicit port connections reduce the redundant nature of listing a name twice for each named port connection when the port name matches the variable name that is connected to the port.

With a careful naming convention, instantiating large logic blocks into a higher level module will be less tedious by using SystemVerilog **`.name`** implicit port connections.

When using the **`.name`** implicit port connection technique, any sub-block port that does not match in size or name to the module net or bus connected to the port, must be connected using a named-port connection. Mixing implicit **`.name`** port connections with positional port connections is not permitted.

Implicit port connections must follow these rules:

- Implicit **`.name`** ports shall not be used in an instantiated sub-block with positional port connections.
- Implicit **`.name`** ports may be used in an instantiated sub-block with named port connections.
- It is permitted to instantiate instances with positional port connections, instances with named port connections, instances with implicit **`.name`** port connections and instances with implicit **`.*`** port connections, all in the same upper-level module.
- If implicit **`.name`** port connections are used to instantiate a sub-block with named port connections, the **`.name`** ports may be placed anywhere in the port list.
- For an implicit **`.name`** connection to be made, the port name on an instantiated sub-block must match the net or bus name of the connecting module.
- For an implicit **`.name`** connection to be made, the port size on an instantiated sub-block must match the net or bus size of the connecting module.
- For an implicit **`.name`** connection to be made, the port data type on an instantiated sub-block must be compatible to the data type of the variable connecting to the module. Any port that would connect to a variable in the upper module without causing a Verilog error or warning is considered to be compatible.
- Any individual port in an implicitly instantiated module that does not match size and name and is not data-type compatible with the net or bus of the upper-level module, must be instantiated by name.
- If a port on an instantiated sub-block is unconnected in the upper-level module, the port can be explicitly listed as a named port with empty parentheses, or omitted from the instantiation port list (**Guideline**: add unconnected ports to the instantiation port list with empty connection parentheses).
- All nets or busses in the upper-level module that connect to implicit ports must either be explicitly declared as a scalar-net, vector-net, or as a port on the upper-level module.

5.4. SystemVerilog: Implicit .* port connections (passed April 2002)

SystemVerilog will also permit an enhanced and abbreviated method to instantiate modules using the implicit `.*` port connection token.

In SystemVerilog, if the net connected to a port of an instantiated model matches the name of the port on the instantiated model, the size of the port on the instantiated model and has a compatible data type to the port on the instantiated model, the matching port name can be omitted and an implicit `.*` port connection can be made.

Many design teams use the proven methodology of using the same name for a top-level net and all of the ports that connect to that net. These same design teams generally use the safe method of making named port connections when instantiating the top-level blocks.

Engineers who have coded large top-level ASIC designs, have experienced the pain of connecting hundreds and thousands of named ports to tens and hundreds of top-level modules.

SystemVerilog introduces the capability to instantiate modules with highly abbreviated and efficient `.*` implicit port connections. Implicit `.*` port connections are intended to facilitate the process of instantiating large sub-blocks into upper-level modules without having to type multiple lines of named port connections where the sub-blocks are instantiated.

Implicit `.*` port connections reduce the verbose nature of most higher-level modules by limiting the number of named ports that actually have to be listed when a module is instantiated. At the same time, since only those nets or busses that do not match must be listed in the module instantiation, they are emphasized and not hidden in a sea of named port connections.

With a careful naming convention, instantiating large logic blocks into a higher level module can now be greatly facilitated by using SystemVerilog implicit `.*` port connections.

When using the `.*` implicit port connection token, any sub-block port that does not match in size or name to the module net or bus connected to the port, must be connected using a named-port connection. Mixing implicit port connections (`.*` connections) with positional port connections is not permitted.

Implicit port connections must follow these rules:

- Implicit `.*` ports shall not be used in an instantiated sub-block with positional ports.
- Implicit `.*` ports may be used in an instantiated sub-block with named ports.
- It is permitted to instantiate instances with positional port connections, instances with named port connections, instances with implicit `.name` port connections and instances with implicit `.*` port connections, all in the same upper-level module.
- If implicit port connections are used to instantiate a sub-block, the `.*` token may be placed anywhere in the instantiated port list with other named port connections, if any are listed.
- For an implicit `.*` connection to be made, the port name on an instantiated sub-block must match the net or bus name of the connecting module.
- For an implicit `.*` connection to be made, the port size on an instantiated sub-block must match the net or bus size of the connecting module.
- For an implicit `.*` connection to be made, the port data type on an instantiated sub-block must be compatible to the data type of the variable connecting to the module. Any port that would connect to a variable in the upper module without causing a Verilog error or warning is considered to be compatible.
- Any individual port in an implicitly instantiated module that does not match both size and name of the net or bus of the upper-level module, or that does not have a compatible data type to the net or bus of the upper-level module, must be instantiated by name.
- If a sub-block is instantiated using implicit `.*` port connections and if a port on the instantiated sub-block is unconnected in the upper-level module, the port shall be explicitly listed as a named port with empty parentheses, showing there is no connection to the port.
- All nets or busses in the upper-level module that connect to implicit `.*` ports must either be explicitly declared as a scalar-net, vector-net, or as a port on the upper-level module.

5.5. Instantiation Example

Consider the example of a Complex Arithmetic Logic Unit[4] (CALU) as shown in Figure 6. This CALU design has nine instantiated sub-blocks. The Verilog-2001 module headers for the nine sub-blocks are shown in Example 7 through Example 15.

If the sub-block module port lists are carefully named so that the port names match the names of the top-level CALU module nets or busses that are connected to the instantiated ports, implicit port connections can be used when the sub-blocks are instantiated into the CALU.

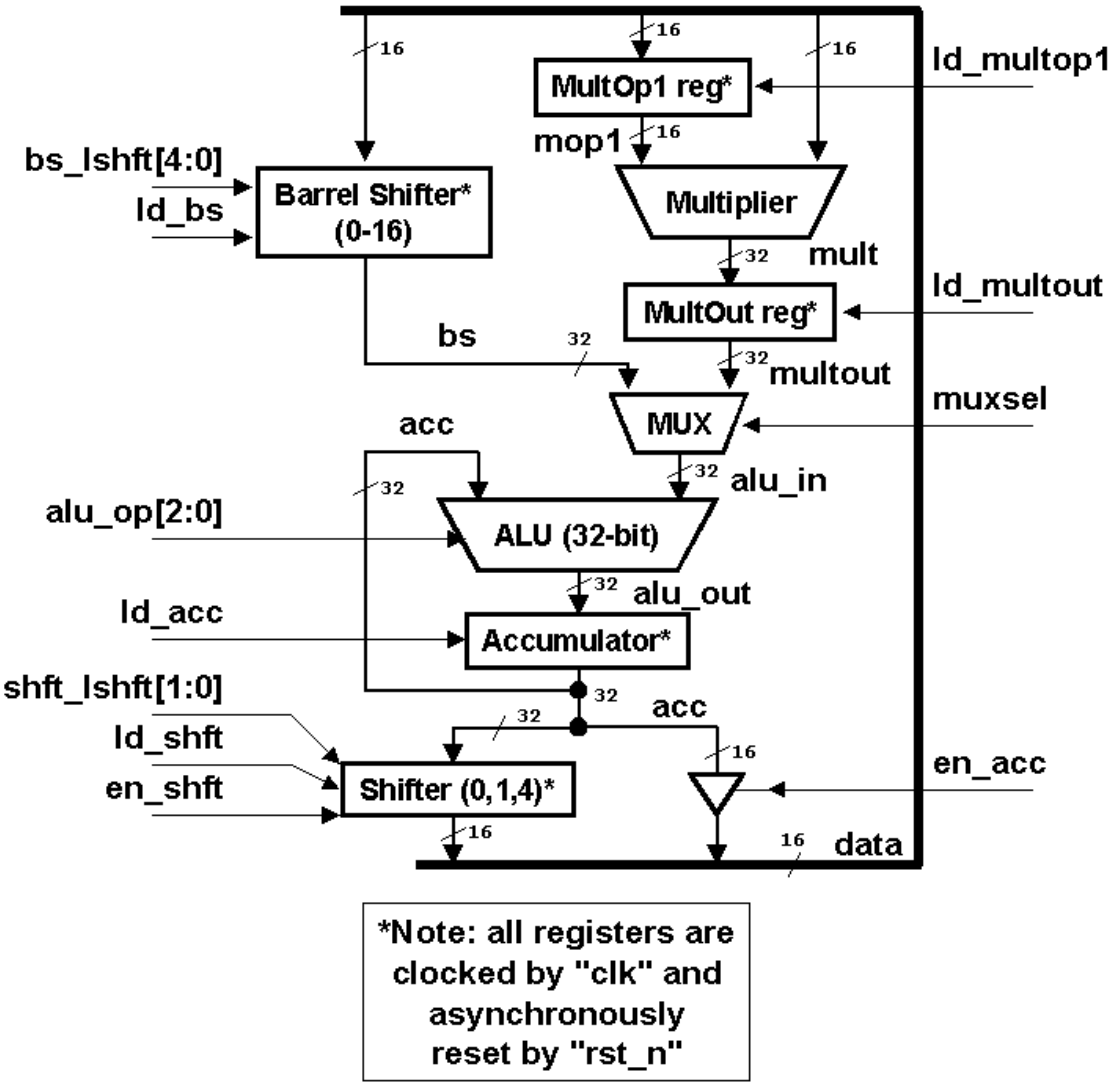

Figure 6 - Complex Arithmetic Logi Unit (CALU) block diagram

```
module multop1 (
  output [15:0] mop1,
  input  [15:0] data,
  input         ld_multop1, clk, rst_n);
  // RTL code for multiplier operand1 reg
endmodule
```

Example 7 - multop1 module header - Verilog-2001 version

```
module multiplier (
  output [31:0] mult,
  input  [15:0] mop1, data);
  // RTL code for the multiplier
endmodule
```

Example 8 - multiplier module header - Verilog-2001 version

```
module multoutreg (
  output [31:0] multout,
  input  [31:0] mult,
  input         ld_multout, clk, rst_n);
  // RTL code for he multiplier output reg
endmodule
```

Example 9 - multoutreg module header - Verilog-2001 version

```
module barrel_shifter (
  output [31:0] bs,
  input  [15:0] data,
  input  [ 4:0] bs_lshft,
  input         ld_bs, clk, rst_n);
  // RTL code for the barrel shifter
endmodule
```

Example 10 - barrel_shifter module header - Verilog-2001 version

```
module mux2 (
  output [31:0] y,
  input  [31:0] i1, i0,
  input         sel1);
  // RTL code for a 2-to-1 mux
endmodule
```

Example 11 - mux2 module header - Verilog-2001 version

```
module alu (
  output [31:0] alu_out,
  output        zero, neg,
  input  [31:0] alu_in, acc,
  input  [ 2:0] alu_op);
  // RTL code for the ALU
endmodule
```

Example 12 - alu module header - Verilog-2001 version

```
module accumulator (
  output [31:0] acc,
  input  [31:0] alu_out,
  input         ld_acc, clk, rst_n);
  // RTL code for the accumulator register
endmodule
```

Example 13 - accumulator module header - Verilog-2001 version

```
module shifter (
  output [15:0] data,
  input  [31:0] acc,
  input  [ 1:0] shft_lshft,
  input         ld_shft, en_shft, clk,
rst_n);
  // RTL code for the shifter
endmodule
```

Example 14 - shifter module header - Verilog-2001 version

```
module tribuf #(parameter SIZE=16)
  (output [SIZE-1:0] data,
   input  [SIZE-1:0] acc,
   input             en_acc);
  // RTL code for the tristate buffer
endmodule
```

Example 15 - tribuf module header - Verilog-2001 version

```
module calu1 (data, bs_lshft, alu_op, shft_lshft, calu_muxsel,
              en_shft, ld_acc, ld_bs, ld_multop1, ld_multout,
              ld_shft, en_acc, clk, rst_n);
  inout  [15:0] data;
  input  [ 4:0] bs_lshft;
  input  [ 2:0] alu_op;
  input  [ 1:0] shft_lshft;
  input         calu_muxsel, en_shft, ld_acc, ld_bs;
        input         ld_multop1, ld_multout, ld_shft, en_acc;
  input         clk, rst_n;
  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.mop1(mop1), .data(data),
                                 .ld_multop1(ld_multop1),
                                 .clk(clk), .rst_n(rst_n));
  multiplier     multiplier     (.mult(mult), .mop1(mop1),
                                 .data(data));
  multoutreg     multoutreg     (.multout(multout),
                                 .mult(mult),
                                 .ld_multout(ld_multout),
                                 .clk(clk), .rst_n(rst_n));
  barrel_shifter barrel_shifter (.bs(bs), .data(data),
                                 .bs_lshft(bs_lshft),
                                 .ld_bs(ld_bs),
                                 .clk(clk), .rst_n(rst_n));
  mux2           mux            (.y(alu_in),
                                 .i0(multout),
                                 .i1(acc),
                                 .sel1(calu_muxsel));
  alu            alu            (.alu_out(alu_out),
                                 .zero(), .neg(), .alu_in(alu_in),
                                 .acc(acc), .alu_op(alu_op));
  accumulator    accumulator    (.acc(acc), .alu_out(alu_out),
                                 .ld_acc(ld_acc), .clk(clk),
                                 .rst_n(rst_n));
  shifter        shifter        (.data(data), .acc(acc),
                                 .shft_lshft(shft_lshft),
                                 .ld_shft(ld_shft),
                                 .en_shft(en_shft),
                                 .clk(clk), .rst_n(rst_n));
  tribuf         tribuf         (.data(data), .acc(acc[15:0]),
                                 .en_acc(en_acc));
endmodule
```

Example 16 - Verilog-2001 calu1 module with all sub-modules instantiated using named port connections

When these modules are instantiated into a Verilog-1995-style or Verilog-2001-style CALU module with named port connections, the correct CALU module code is shown in Example 16.

Although using named port connections is the recommended way to instantiate sub-blocks into an upper-level module, inspection of the code in Example 16 shows that most of the connections have matching port and connecting-net names.

As shown in this example, named port connections can be both tedious to create and verbose.

```
module calu2 (data, bs_lshft, alu_op, shft_lshft, calu_muxsel,
              en_shft, ld_acc, ld_bs, ld_multop1, ld_multout,
              ld_shft, en_acc, clk, rst_n);
  inout  [15:0] data;
  input  [ 4:0] bs_lshft;
  input  [ 2:0] alu_op;
  input  [ 1:0] shft_lshft;
  input         calu_muxsel, en_shft, ld_acc, ld_bs;
        input         ld_multop1, ld_multout, ld_shft, en_acc;
  input         clk, rst_n;
  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.mop1, .data, .ld_multop1,
                                 .clk, .rst_n);
  multiplier     multiplier     (.mult, .mop1, .data);
  multoutreg     multoutreg     (.multout, .mult, .ld_multout,
                                 .clk, .rst_n);
  barrel_shifter barrel_shifter (.bs, .data, .bs_lshft, .ld_bs,
                                 .clk, .rst_n);
  mux2           mux            (.y(alu_in),
                                 .i0(multout),
                                 .i1(acc),
                                 .sel1(calu_muxsel));
  alu            alu            (.alu_out, .zero(), .neg(), .alu_in,
                                 .acc, .alu_op);
  accumulator    accumulator    (.acc, .alu_out, .ld_acc, .clk,
                                 .rst_n);
  shifter        shifter        (.data, .acc, .shft_lshft,
                                 .ld_shft, .en_shft, .clk, .rst_n);
  tribuf         tribuf         (.data, .acc(acc[15:0]), .en_acc);
endmodule
```

Example 17 - Verilog-2001 calu2 module with all sub-modules instantiated using implicit **.name** port connections

When these modules are instantiated into a SystemVerilog-style CALU module with **.name** implicit port connections, the correct CALU module code is shown in Example 17.

In the example code for the **calu2** module, note that all of the busses and nets that are connected to the ports of both the **multop1** and **multiplier** modules have names and sizes that match the port names and sizes on the instantiated modules. There is a 16-bit bus named **mop1** that is driven by the **multop1** register into the **multiplier** module. Since this bus is not a declared port on the **calu2** module, it must be explicitly declared in the **calu2** module in order to take advantage of the **.name** implicit port connection capability. Similarly, the 32-bit **mult** bus, driven by the multiplier module is also an internal bus and must be explicitly declared in the **calu2** module. All of the other ports that are connected to the **multop1** register and the **multiplier** instance are connected to busses and nets that are explicitly declared as ports on the **calu2** module and therefore they do not require separate explicit net declarations.

In the Example 17 code for the **calu2** module, note that a generic 32-bit-wide, 2-to-1 **mux** has been instantiated. Since none of the ports on this sub-block match the net or bus names of the **calu2** module, all of the ports must either be connected by name or by position (in this example, they are connected by name).

In the example code for the **calu2** module, note that the **alu** has two unused outputs, **zero** and **neg**. The unused ports may be listed with empty connections as shown, or omitted from the instantiation port list when using the implicit **.name** port connection enhancement.

In the example code for the **calu2** module, note that the **tribuf** module has a 16-bit input port named **acc** but it is connected to a 32-bit bus also named **acc**. Since the port and bus sizes do not match, a named connection is required to show which bits of the 32-bit **acc** bus are connected to the 16-bit **acc** port. SystemVerilog does not assume that the low-order bits of a same-named port and bus are connected. That information must be provided in the named port connection.

For all of the other sub-blocks in the example code for the **calu2** module, any instance port that is connected to a **calu2** module port requires no additional explicit net declaration, while all of the

instance ports of sub-blocks that are connected to **calu2** internal busses require explicit net declarations within the **calu2** module.

It can be seen from this example that the implicit .name port connects save a significant amount of typing.

```
module calu3 (
  inout  [15:0] data,
  input  [ 4:0] bs_lshft,
  input  [ 2:0] alu_op,
  input  [ 1:0] shft_lshft,
  input         calu_muxsel, en_shft, ld_acc, ld_bs,
  input         ld_multop1, ld_multout, ld_shft, en_acc,
  input         clk, rst_n);

  wire   [31:0] acc, alu_in, alu_out, bs, mult, multout;
  wire   [15:0] mop1;

  multop1        multop1        (.*);
  multiplier     multiplier     (.*);
  multoutreg     multoutreg     (.*);
  barrel_shifter barrel_shifter (.*);
  mux2           mux            (.y(alu_in), .i0(multout),
                                 .i1(acc), .sel1(calu_muxsel));
  alu            alu            (.*, .zero(), .neg());
  accumulator    accumulator    (.*);
  shifter        shifter        (.*);
  tribuf         tribuf         (.*, .acc(acc[15:0]));
endmodule
```

Example 18 - SystemVerilog calu3 module with sub-modules instantiated using implicit **.*** port connections

When these modules are instantiated into a SystemVerilog-style CALU module with **.*** implicit port connections, the correct CALU module code is shown in Example 18.

In the example code for the **calu3** module, note that all of the busses and nets that are connected to the ports of both the **multop1** and **multiplier** modules have names and sizes that match the port names and sizes on the instantiated modules. There is a 16-bit bus named **mop1** that is driven by the **multop1** register into the **multiplier** module. Since this bus is not a declared port on the **calu3** module, it must be explicitly declared in the **calu3** module in order to take advantage of the **.name** implicit port connection capability. Similarly, the 32-bit **mult** bus, driven by the multiplier module is also an internal bus and must be explicitly declared in the **calu3** module. All of the other ports that are connected to the **multop1** register and the **multiplier** instance are connected to busses and nets that are explicitly declared as ports on the **calu3** module and therefore they do not require separate explicit net declarations.

In the Example 18 code for the **calu3** module, note that a generic 32-bit-wide, 2-to-1 **mux** has been instantiated. Since none of the ports on this sub-block match the net or bus names of the **calu3** module, all of the ports must either be connected by name or by position (in this example, they are connected by name).

In the example code for the **calu3** module, note that the **alu** has two unused outputs, **zero** and **neg**. The unused ports must be listed with empty connections when using the implicit port connection token (**.***) to make the rest of the connections.

In the example code for the **calu3** module, note that the **tribuf** module has a 16-bit input port named **acc** but it is connected to a 32-bit bus also named **acc**. Since the port and bus sizes do not match, a named connection is required to show which bits of the 32-bit **acc** bus are connected to the 16-bit **acc** port. SystemVerilog does not assume that the low-order bits of a same-named port and bus are connected. That information must be provided in the named port connection.

For all of the other sub-blocks in the example code for the **calu3** module, any instance port that is connected to a **calu3** module port requires no additional explicit net declaration, while all of the instance ports of sub-blocks that are connected to **calu3** internal busses require explicit net declarations within the **calu3** module.

6. SystemVerilog: interface port types

Pending approval, SystemVerilog will add an enhanced and powerful method to declare reusable complex module port bundles called an "interface."

An interface is declared between the keywords **`interface`**/**`endinterface`** and is typically declared outside of module boundaries.

6.1. Interface ports for IP development

Pending approval of SystemVerilog interface ports, one of the primary benefactors of this capability will be intellectual property (IP) developers.

Verilog-2001 IP is frequently developed as an RTL block that is provided for instantiation into a customer design.

Similarly, SystemVerilog IP will also be an RTL model that is provided to a customer through an **`interface`** block. The advantage of the **`interface`** block is that the IP provider will code how to connect the IP into an **`interface`** that can simply be instantiated into a customer module.

Because both sides of the IP connection can now be specified and coded by the IP developer, the customer can more easily instantiate the IP block and the developer can insure that the **`interface`** between the commercial IP and the customer module is correctly specified.

Using **`modports`**, an IP developer can even indicate the direction of the customer ports, giving further guidance to the customer on how to connect to the IP design.

7. SystemVerilog Proposals

The SystemVerilog Standard is currently scheduled for first release around June of 2002, and input, support or requests for removal of the enhancements outlined in this paper will still be considered.

8. Summary and conclusions

Verilog-2001 offers significant simplification over Verilog-1995 when declaring module ports, directions ad data types.

Although the new compiler directive option **`` `default_nettype ``** none forces declaration checking that may help find typos in the body of the Verilog code, the added declarations themselves can be tedious to code, add verbosity to a design and be the source of additional unnecessary typos. The author believes the **`none`** option should be avoided.

SystemVerilog will offer greater abstraction and design brevity through the use of implicit port connections and interfaces.

The following is a summary of the guidelines that were given in this paper:

Guideline: do not add **`` `default_nettype none ``** to a working Verilog design.

Guideline: do not use **`` `default_nettype none ``**, ever!

Guideline: when instantiating a module from another design team member, from a third party or vendor, or if you do not have direct control over the module being instantiated, do not use positional port connections.

Guideline: in general, use named port connections when instantiating modules into a design.

Guideline: show all unconnected module ports using empty named connections.

References

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

[2] IEEE Standard Verilog Hardware Description Language, IEEE Computer Society, IEEE, New York, NY, IEEE Std 1364-2001,

[3] Adam Krolnik, personal communication.

[4] First-Generation TMS320 User's Guide, Texas Instruments, 1989, pg. 3-17.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten 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 and the Accellera SystemVerilog Standardization 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 April 18th, 2002)