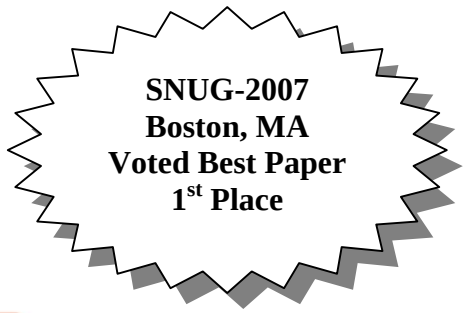

World Class Verilog & SystemVerilog Training

SystemVerilog Implicit Port Enhancements Accelerate System Design & Verification

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

ABSTRACT

The IEEE Std 1800-2005 SystemVerilog Standard added new implicit port instantiation enhancements that help accelerate top-level composition of large ASIC & FPGA Designs.

This paper details the new **.*** and **.name** implicit port instantiation capabilities, the rules related to the use of these new enhancements, how these enhancements offer concise RTL coding styles while enforcing stronger port-type checking and will include statistics from actual ASIC design projects that incorporated these enhanced capabilities.

Table of Contents

1 Introduction 4
1.1 VCS version 4
2 Sample design 4
3 Different port connection styles 5
3.1 Verilog positional port connections 5
3.2 Verilog named port connections 6
3.3 The .* implicit port connection enhancement 7
3.4 The .name implicit port connection enhancement 8
4 .* and .name details 9
4.1 .* and .name advantages 9
4.2 .* usage 9
4.3 .* and .name rules 9
4.4 Comparing Verilog-2001 to SystemVerilog implicit ports 12
4.5 Packages 12
4.6 Mixing .name and .* 13
4.7 Abstraction and fear 14
5 Verilog EMACs mode port expansion assistance 15
5.1 EMACS basic setup 15
5.2 EMACS mode requirements for SystemVerilog 15
5.3 EMACS mode file-trailer comments 17
5.4 .* Adoption 17
5.5 EMACS -vs- vi 17
6 Converting existing designs 18
6.1 Large ASIC design #1 18
6.2 Signal naming conventions 20
7 Arrays of instances & generate statements 20
7.1 Pipeline register 21
7.2 Pipelined design - named port connections 21
7.3 Pipelined design - named port connections with .* 21
7.4 Pipelined design - arrays of instance 22
7.5 Pipelined design - generate instances 23
7.6 Array Of Instance - Arrayed Instance Bus Matching 23
8 Interfaces 24
8.1 SystemVerilog Interface Usage with .* Implicit Ports 24
9 Other Issues 24
9.1 Limitation - gate-level netlists 24
9.2 EMACS Availability 25
9.3 Regular Expression Matching 25
10 Recommended Warnings & Errors 25
11 Submit Your Design Experiences! 26
12 Conclusions 27
13 Acknowledgements 27
14 References 27
15 Author & Contact Information 28

Table of Examples

Example 1 - alu_accum model built using positional port connections 5
Example 2 - alu_accum model built using named port connections 6
Example 3 - - alu_accum model built using .* implicit port connections 7
Example 4 - alu_accum model built using .name implicit port connections 8
Example 5 - Mixed .name and .* ports in alu instance 13
Example 6 - Explicit package variable with .* instantiation 14
Example 7 - alu_accum3 design with EMACS expanded ports 16
Example 8 - alu_accum3 design with ports collapsed by EMACS 16
Example 9 - 16-bit register - to be instantiated into 4-stage pipeline register designs 21
Example 10 - 4-stage pipeline register with Verilog named port connections 21
Example 11 - 4-stage pipeline register with SystemVerilog .* implicit port connections 22
Example 12 - 4-stage pipeline register using an array of instance and SystemVerilog .* implicit ports 22
Example 13 - 4-stage pipeline register using a generate statement and SystemVerilog .* implicit ports 23
Example 14 - 1-bit input buffer 23
Example 15 - ibuf-top-module with ports that match the arrayed ibuf instance ports 24

Table of Figures

Figure 1 - alu_accum block diagram 4

1 Introduction

For large ASIC and FPGA designs, the top-level design module or modules are 10's of pages of 100's of instantiations using 10,000's of named port connections.

The top-level design has very little value to design engineers. There are so many port and signal names in a top-level module that it is nearly impossible to follow the intended design structure simply by looking at all the text. The only value of this top-level design is to establish connectivity for EDA tools such as simulation and synthesis. Since there is so little value in this top-level module, why should design engineers spend so much time piecing together a top-level design? Why can't we ask the tools to make the connection for us? That is the idea behind **.*** implicit port connections.

1.1 VCS version

All VCS capabilities described in this paper were tested using VCS version Y-2006.06-SP1.

2 Sample design

The IEEE Std 1800-2005, section 18.11, includes multiple instantiation examples of an **`alu_accum`** design modeled using Verilog and SystemVerilog instantiation methods. Below is a slightly modified version of that same design. In this design, the **`alu`** block has unconnected **`zero`** and **`ones`** output pins. The **`alu_accum`** model also has an unconnected **`zero`** output pin that is not connected to the unconnected **`alu zero`** output. This was done just to add an interesting scenario to the design.

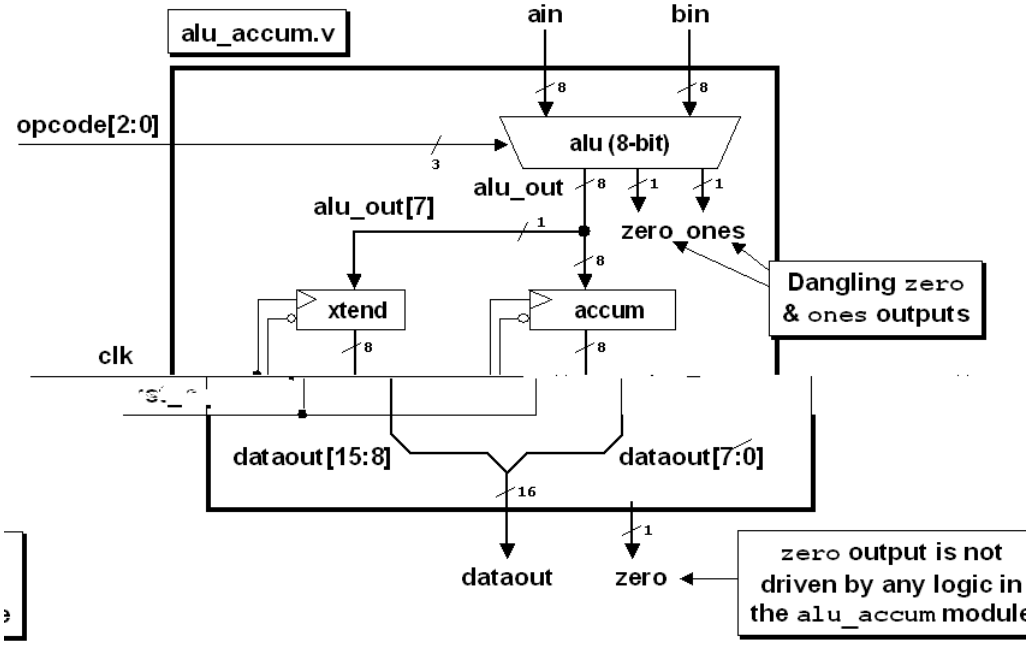

Figure 1 - alu_accum block diagram

3 Different port connection styles

In this section, the **alu_accum** model will be coded four different ways: (1) using positional port connections, (2) using named port connections, (3) using SystemVerilog **.*** implicit port connections, and (4) using SystemVerilog **.name** implicit port connections.

The styles are compared for advantages, disadvantages, coding effort and efficiency.

3.1 Verilog positional port connections

Verilog has always permitted positional port connections. The Verilog code for positional port connection instantiation of the sub-modules in the **alu_accum** block diagram is shown in Example 1. The model requires 15 lines of code and 249 characters.

```
module alu_accum1 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (alu_out, , , ain, bin, opcode);

  accum accum (dataout[7:0], alu_out, clk, rst_n);

  xtend xtend (dataout[15:8], alu_out[7], clk, rst_n);
endmodule
```

Example 1 - alu_accum model built using positional port connections

Advantage: The port signal names only have to be listed once so instantiation is relatively easy to complete.

Disadvantages: (1) A module instantiated with an incorrect port order can be time consuming to detect and debug. (2) If the port order of the instantiated module is changed by a third party, the instantiation will also have to change. (3) Minor port order changes are often the most difficult to find and debug.

3.2 Verilog named port connections

Verilog has always permitted named port connections (also called explicit port connections). Any engineer who has ever assembled a top-level netlist for a large ASIC or FPGA is familiar with the tedious pattern of instantiating ports of the form:

```
mymodule u1 (.data(data), .address(address),  ...  .BORING(BORING));
```

The top-level module description for a large ASIC or FPGA design may be 10-20 pages of tediously instantiated modules forming a collection of port names and net names that offer little value to the author or reviewer of the code. With net names potentially dispersed onto multiple pages of code, it is difficult for any engineer to comprehend the structure of such a design.

Most engineers agree that large top-level ASIC or FPGA netlists offer very little value aside from connecting modules together to simulate or synthesize. They are painful to assemble, painful to debug and sometimes painful to maintain when lower-level module port lists are modified, requiring top-level netlist modifications.

The problem with large top-level netlists is that there is too much information captured and the information is spread out over too many pages to allow easy visualization of the design structure. For all practical purposes, the top-level design becomes a sea of names and gates. The information is all there but it is in a largely unusable form!

The named port connections version of the Verilog code for the **`alu_accum`** block diagram is shown in Example 2.

```
module alu_accum2 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (.alu_out(alu_out), .zero(), .ones(),
               .ain(ain), .bin(bin), .opcode(opcode));

  accum accum (.dataout(dataout[7:0]), .datain(alu_out),
               .clk(clk), .rst_n(rst_n));

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]),
               .clk(clk), .rst_n(rst_n));
endmodule
```

Example 2 - alu_accum model built using named port connections

Advantages: (1) Mis-ordering of instance ports does not cause design failures. (2) If the port order of the instantiated module is changed by a third party, no instantiation changes are required. (3) Minor port order changes do not impact the design. (4) If a port name changes, the compiler gives immediate feedback that the specified port does not exist.

Disadvantages: Instantiations are twice as verbose as instantiation using positional port connections and take longer to type.

3.3 The .* implicit port connection enhancement

SystemVerilog introduces the ability to do **.*** implicit port connections. Whenever the port name and size matches the connecting net or bus name and size, the port name can be omitted and **.*** will make the connection automatically as shown in Example 3. The model requires 15 lines of code and 261 characters.

```
module alu_accum3 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (.zero(), .ones(), .*);

  accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*);
endmodule
```

Example 3 - - alu_accum model built using .* implicit port connections

3.4 The .name implicit port connection enhancement

SystemVerilog also introduces the ability to do **.name** implicit port connections. Just like the **.*** implicit port connection style, whenever the port name and size matches the connecting net or bus name and size, the port name can be listed just once with a leading period as shown in Example 4. The model requires 19 lines of code and 303 characters.

```
module alu_accum4 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (.alu_out, .zero(), .ones(), .ain, .bin, .opcode);

  accum accum (.dataout(dataout[7:0]), .datain(alu_out),
               .clk, .rst_n);

  xtend xtend (.dout(dataout[15:8]),   .din(alu_out[7]),
               .clk, .rst_n);
endmodule
```

Example 4 - alu_accum **model built using** .name **implicit port connections**

4 .* and .name details

This section outlines advantages, rules and details related to **.*** and **.name** implicit port instantiations.

4.1 .* and .name advantages

There are two strong advantages to using **.*** implicit port connections over Verilog positional or named port connections: (1) more concise designs, and (2) strong port type checking. These are two excellent reasons to use then new **.*** implicit port connections.

For engineers who were not ready for the abstract designs using **.*** implicit ports, we introduced **.name** implicit port connections. Two reasons to use **.name** implicit port connections are (1) semi-concise designs, and (2) strong port type checking.

4.2 .* usage

According to IEEE Std 1800-2005, the **.*** may only be listed once per instantiation and may be placed anywhere in the instantiated port list.

Legal: **.*** at the beginning of the port list.

```
accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out));
```

Legal: **.*** in the middle of the port list.

```
accum accum (.dataout(dataout[7:0]), .*, .datain(alu_out));
```

Recommended: **.*** at the end of the port list.

```
accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);
```

Illegal: **.*** in the port list twice.

```
accum accum (.*, .dataout(dataout[7:0]), .datain(alu_out), .*);
```

To take advantage of the EMACS mode port collapsing and expansion, the **.*** must be placed last in the instantiated port list.

4.3 .* and .name rules

When instantiating modules using **.*** or **.name** implicit ports, the following rules apply:

(1) It is illegal to mix positional ports with new SystemVerilog **.*** or **.name** implicit port connections.

```
alu   alu   (alu_out, , , .*);                    // ILLEGAL
```

In the above example, the first three ports are listed by position and the remaining ports are connected using SystemVerilog **.*** implicit ports. This is not legal in SystemVerilog.

```
accum accum (dataout[7:0], alu_out, .clk, .rst_n); // ILLEGAL
```

In the above example, the first two ports are listed by position and the remaining ports are connected using SystemVerilog **.name** implicit ports. This is not legal in SystemVerilog.

This restriction is not surprising. The new SystemVerilog implicit port styles are alternate forms of named port connections and it has never been legal in Verilog to mix positional ports with named ports in the same instantiation.

```
// ILLEGAL Verilog-2001 instantiation
xtend xtend (dataout[15:8], alu_out[7], clk, .rst_n(rst_n));
```

In the above example, the first three ports are listed by position and the last port is connected using a Verilog-2001 named port connection. This is not legal in Verilog or SystemVerilog.

(2) It is legal to mix SystemVerilog **.*** or **.name** implicit port connections with Verilog named port connections. As a matter of fact, it is required to mix SystemVerilog **.*** or **.name** implicit port connections with named ports if rules 3 - 6 apply (see below).

The following rules apply if the individual module instantiation includes SystemVerilog **.*** or **.name** implicit port connections.

(3) If a port name does not match a connecting net name, then Verilog named port connections must be used.

```
xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*);
```

In the above example, two ports are connected to buses that have different names. In this same example, the upper 8 bits of a 16-bit **dataout** bus is connected to an 8-bit **dout** port and the MSB of an 8-bit **alu_out** bus is connected to a 1-bit **din** input.

(4) If a port size is smaller than a connecting bus size, then Verilog named port connections must be used to show which bits of the connecting bus are connected to the port. Port sizes and connecting nets or buses must match.

```
accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);
```

In the above example, the lower 8 bits of a 16-bit **dataout** bus are connected to an 8-bit **dataout** port. Although the names matched, the sizes did not match so a named port connection was required. In this same example an 8-bit **alu_out** bus is connected to an 8-bit port with a different name (**datain**).

(5) If a port size is larger than a connecting bus or net size, then Verilog named port connections must be used and a concatenated bus of matching size (using constants, nets and or buses) must be connected to the port. Port sizes and connecting nets or buses must match.

```
mymod1 u1 (.din({4'b0,nibble}), .*);
```

In the above example, a 4-bit **nibble** bus is connected to an 8-bit **din** port so the 4 MSBs are connected to **0**'s through the use of a concatenation of **4'b0** and the 4-bit **nibble** bus.

(6) All unconnected ports must be shown as named empty port connections. Empty ports cannot be omitted, unlike in Verilog-2001 instantiations.

```
alu   alu   (.zero(), .ones(), .*);
```

In the preceding example, the **zero** and **ones** ports are left unconnected so they must be explicitly listed.

Rev 1.1 Update - The **.name** implicit port connections do not require that unconnected ports be listed in the instantiation. The original intent of the enhancement was to require the empty named port connections but this verbiage was omitted from the IEEE1800-2005 Standard and efforts by users who participate on the 2008 SystemVerilog Standards Group were unsuccessful in updating the P1800-2008 Standard with the stronger requirements due to backward compatibility with designs that have already used the **.name** port connection style while omitting unconnected ports. See update to requirement (8) below. Cliff believes this is just another good reason to choose **.*** implicit ports over **.name** implicit ports.

(7) All internal 1-bit nets must be declared. In Verilog-2001 designs, undeclared identifiers automatically turned into 1-bit wires. For SystemVerilog, 1-bit nets connected to .* or .name implicit ports must be declared. . Empty ports cannot be omitted, unlike in Verilog-2001 instantiations.

```
alu   alu   (.zero(), .ones(), .*);
```

In the preceding example, the **zero** and **ones** ports are left unconnected so they must be explicitly listed.

(8) According to IEEE Std 1800-2005, it is legal to mix SystemVerilog **.*** and **.name** implicit port connections in the same instantiation. The need to do this is rare and it is discussed further in section 4.6.

Rev 1.1 Update - To force checking for named unconnected ports when using **.name** implicit port connections, simply add a **.*** to the end of the instantiated **.name** implicit port list. Note that most vendors do not support mixing of **.*** and **.name** in the same instantiation at the time that Rev 1.1 was published.

4.4 Comparing Verilog-2001 to SystemVerilog implicit ports

The comparisons between Verilog-1995/2001 positional or named port connections to SystemVerilog **.*** or **.name** implicit port connections are shown in the following table.

Verilog-2001 Port Connections	SystemVerilog Implicit Ports
If the port size does not match the connecting net or bus • port-size mismatch WARNING • simulations run without modification *(this is almost always an error)*	If the port size does not match the connecting net or bus • port-size mismatch ERROR • sizes must be corrected before simulations will run
Omitting ports from the instantiated module • LEGAL !! - no warning • simulations run without modification and treat the unlisted port as an unconnected port	Omitting ports from the instantiated module • ERROR • Must list all unconnected ports before simulations will run
Undeclared 1-bit wires in top module • LEGAL !! - no warning • Implicitly creates a 1-bit wires	Undeclared 1-bit wires in top module • ERROR • All implicitly connected ports require an upper-level net declaration

Table 1 - Verilog port connections compared to SV implicit port connections

What does this mean? It means that using **.*** implicit ports permits the creation of more concise top-level designs with stronger port type checking. This is a win-win enhancement!

So why not add the additional checking to Verilog-2001 ports? The stronger port type checking cannot be added to Verilog-2001 style port connections for reasons of backward compatibility. The stronger port type checking would break too many existing designs.

4.5 Packages

SystemVerilog added **`package`**-**`endpackage`** with the ability to capture and encapsulate type definitions, tasks, functions and even net and variable declarations.

Note: It is illegal to include **`output`**, **`inout`** and **`input`** declarations in a package.

Packaging internal net and variable declarations would be similar to declaring a SystemVerilog interface, but packaged declarations are not quite as versatile or powerful as the **`interface`** construct.

```
package alu_pkg;
  logic [7:0] alu_out;
endpackage
```

When a package is imported using the form:

```
import alu_pkg::*;
```

the wildcard import in reality does not actually import anything until first use. This means that the 8-bit **alu_out** logic bus is not visible unless explicitly referenced from the code. Instantiating a module using **.*** implicit port instantiation does not explicitly access the **alu_out** bus by name so **.*** would fail to find any **alu_out** variable declaration. On the other hand, **.name** instantiation of the form **.alu_out** does explicitly list the variable so it is visible to **.name** instantiation.

In contrast, explicit importing of package items, as shown below, does import and make visible the item explicitly imported. In the following example, the **alu_out** bus is explicitly imported from the **alu_pkg** and is therefore visible to **.*** and **.name** instantiation.

```
import alu_pkg::alu_out;
```

4.6 Mixing .name and .*

If internal signal declarations were declared within a package and then wild-card imported inside of a module (as shown in fig###), the SystemVerilog **.name** implicit port connections would find the packaged declarations and use them in a design, but SystemVerilog **.*** implicit port connections would not find the declarations and would therefore fail. For this reason, the IEEE SystemVerilog standards committee added the ability to mix **.name** and **.*** connections in the same instantiation to the IEEE Std 1800-2005.

```
module alu_accum1a (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  import alu_pkg::*;

  alu   alu   (.alu_out, .zero(), .ones(), .*);

  accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*);
endmodule
```

Example 5 - Mixed .name and .* ports in alu instance

In the above example, the **alu** module is instantiated using **.*** implicit ports but the **alu_out** bus connected to the **alu** module would not be found, so in the **alu** instantiation the **alu_out** port is instantiated using a **.name** syntax. This is an example of mixed **.name** and **.*** instantiation. Note: this capability is currently not working in any known simulator.

```
module alu_accum2a (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  import alu_pkg::alu_out; // imported alu_out is visible to .*

  alu   alu   (.zero(), .ones(), .*);

  accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*);
endmodule
```

Example 6 - Explicit package variable with .* instantiation

In the above example, the **`alu`** module is instantiated using **.*** implicit ports and the instantiation finds the **`alu_out`** bus because it was explicitly imported by the package import syntax shown. Note: this works with VCS but does not work with all simulators at this time.

In theory, an engineer could capture all of the internal signal declarations into a package, wildcard import (**`import`** *pkg*`::*;`) all of the internal declarations into a module and reference some of the signal declarations using **.name** implicit ports and then complete the port connections using **.*** implicit ports. This is documented in section 19.11.4 of the IEEE Std 1800-2005[4].

IEEE Std 1800-2005, 19.11.4 Instantiation using implicit **.*** port connections:

"... A named or implicit **.name** connection can be mixed with a **.*** connection to create a sufficient reference for a wildcard import of a name from a package."

In earlier versions of the SystemVerilog standard (Accellera 3.1a[5]), mixing **.name** and **.*** connections in the same instantiation was not explicitly allowed and was treated as an error by at least two simulators, including VCS. Although this capability is now explicitly allowed in IEEE Std 1800-2005 as cited above, I know of no simulator that supports this mixed **.name** and **.*** instantiation style. I do not consider this to be a severe bug since I cannot think of a good reason to package internal signal declarations only to import them into a design. Importing bundles of signals is better left to the declaration and use of the SystemVerilog **`interface`** construct.

4.7 Abstraction and fear

Many engineers are very enthusiastic about **.*** (myself included) while others are terribly afraid of the abbreviated syntax because they cannot see the port lists of instantiated modules.

When SystemVerilog **.*** implicit port connections were first introduced to engineers, about half of the engineers I taught enthusiastically embraced the use of **.*** implicit ports while the other half expressed fear related to debugging a design where the port names were not visible. For this reason, a large number of engineers elected to use **.name** implicit port connections instead of the more concise **.*** implicit ports. I was frustrated that I could not convince more engineers to use the simple, elegant and more concise **.*** implicit ports.

I questioned how was I going to get more engineers excited about using the new **.*** implicit ports over using **.name** implicit ports.

5 Verilog EMACs mode port expansion assistance

At the urging of my colleague and friend, Erich Whitney, I approached Wilson Snyder to see if he could offer support for **.*** implicit ports in his Verilog EMACS mode. Wilson's EMACS mode already supported the insertion of **`/*AUTOINST*/`**, which offered support for the exact same capability that we wanted with **.***. The request was made to Wilson in early November, 2005 and the first version of the EMACS mode with **.*** implicit port-expansion support was ready by the end of the same month.

5.1 EMACS basic setup

There were two main issues that had to be addressed to make **.*** support within EMACS a reality: (1) automated port expansion and collapsing of implicit ports, and (2) finding all the files of the instantiated modules so they could be used to match and check the port expansions. The second requirement could be most easily satisfied if the Verilog EMACS mode could recognize common Verilog command line options and simple Verilog command files.

5.2 EMACS mode requirements for SystemVerilog

There were two main issues that had to be addressed to make **.*** support within EMACS a reality: (1) automated port expansion and collapsing of implicit ports, and (2) finding all the files of the instantiated modules so they could be used to match and check the port expansions. The second requirement could be most easily satisfied if the Verilog EMACS mode could recognize common Verilog command line options and simple Verilog command files.

If all of the design files are in the same directory, the EMACS mode can expand and collapse the ports automatically with no additional information provided to the EMACS Verilog mode.

The EMACS command sequence to expand all the ports for debugging is ctl-c ctl-a (often abbreviated as (C-cC-a) by EMACS users).

Note: the EMACS mode cannot expand and collapse **.name** implicit ports.

When the **`alu_accum3`** module of Example 3 is edited using the new EMACS Verilog mode, and when the **.*** implicit ports are expanded by issuing the command-key sequence (C-cC-a), the expanded version of the design is shown in Example 7

```
module alu_accum3 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (.zero(), .ones(), .*,
               // Outputs
               .alu_out      (alu_out[7:0]),  // Implicit .*
               // Inputs
               .ain          (ain[7:0]),      // Implicit .*
               .bin          (bin[7:0]),      // Implicit .*
               .opcode       (opcode[2:0]));  // Implicit .*

  accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*,
               // Inputs
               .clk          (clk),           // Implicit .*
               .rst_n        (rst_n));        // Implicit .*

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*,
               // Inputs
               .clk          (clk),           // Implicit .*
               .rst_n        (rst_n));        // Implicit .*
endmodule
```

Example 7 - alu_accum3 design with EMACS expanded ports

After debugging the **`alu_accum3`** module and when there is no more need to see all of the additional named ports, the expanded ports can be easily collapsed. The EMACS command sequence to collapse all of the ports is: (C-cC-k)

```
module alu_accum3 (
  output [15:0] dataout,
  output        zero,
  input   [7:0] ain, bin,
  input   [2:0] opcode,
  input         clk, rst_n);

  logic   [7:0] alu_out;

  alu   alu   (.zero(), .ones(), .*);

  accum accum (.dataout(dataout[7:0]), .datain(alu_out), .*);

  xtend xtend (.dout(dataout[15:8]), .din(alu_out[7]), .*);
endmodule
```

Example 8 - alu_accum3 design with ports collapsed by EMACS

Note that the EMACS mode will automatically collapse the ports when the EMACS editor buffer is saved.

5.3 EMACS mode file-trailer comments

In reality, it is rare to find all of the necessary files for a large design in a single directory. Most Verilog power-users use command files with command options to designate the locations of all of the design files. Assuming that the command file is named **`run.f`** and that the command file is in the same directory as the top-level file that uses **.*** implicit ports, then it is a simple matter to add the following four lines of comment-code to the bottom of the top-level design module after the **`endmodule`** statement:

```
// Local Variables:
// mode: Verilog
// verilog-library-flags:("-f run.f")
// End:
```

With this EMACS directive explicitly placed at the bottom of the top-level design, the EMACS Verilog mode will now search all of the necessary design files to match and expand all of the ports for the top-level instantiations.

The expansion and collapsing of ports is done for all instances in the active module scope, which makes it easy to debug the design. When the debugging task is complete executing the command sequence (C-cC-k), or saving the design buffer will automatically collapse all of the expanded ports to save the very concise **.*** version of the design.

5.4 .* Adoption

As noted earlier, about one half of he engineers that I used to train would not use **.*** because they were afraid that **.*** designs would be difficult to debug. Now that I can use the EMACS Verilog mode to demonstrate an easy way to expand and collapse ports to help debug designs, I no longer hear objections to using the **.*** implicit ports.

5.5 EMACS -vs- vi

I am a **vi** and **vim** user and have not seriously used EMACS in over 10 years. The intent of this paper is not to convince you that you should abandon all other editors and exclusively use EMACS. Even though I continue to primarily use **vi** and **vim** for almost all file editing, when I need to do automated port expansion for design debug, I now open the **.*** version of the design using Wilson's EMACS mode and with a couple of keystrokes I can easily expand the ports for debugging purposes, and then collapse the ports for storage and normal use.

6 Converting existing designs

When the first EMACS Verilog mode with **.*** support was ready, Wilson Snyder emailed me to let me know that it would be relatively easy to retro-fit existing Verilog designs using another feature of the EMACS mode, verilog-auto-inject (C-cC-z)

The verilog-auto-inject function of Wilson's EMACS mode instruments a design with AUTO commands, including the all-important **/*AUTOINST*/**. The AUTOINST commands are inserted after all of the required named port connections for each instance, which is exactly what we want for .* implicit port connections. After executing (C-cC-z), one can then substitute **.*** for all of the **/*AUTOINST*/** commands and the design is now ready for port collapsing (C-cC-k) and port expansion (C-cC-a).

6.1 Large ASIC design #1

I worked with one company to look into retro-fitting an existing design that the company already had in production. We decided to benchmark the coding reduction by modifying their top-level core model. The original top-level core model had 2830 lines of code.

Using the EMACS mode on large and convoluted existing designs has sometimes proven to be a significant challenge. The EMACS mode works well on designs where the instantiations are rather straight forward, but on some large designs, engineers have used some very "interesting" and convoluted styles to control designs and design configurations.

In the process of doing the verilog-auto-insert command, we found about a half dozen subtle bugs related to some of these convoluted styles in the EMACS Verilog mode. For each bug, we were able to put together simple models with similar syntax that would fail the verilog-auto-insert step. These bugs were emailed back to Wilson and he fixed them all rather quickly.

After using the verilog-auto-inject command sequence (C-cC-z), we found that the comment-style **/*AUTOINST*/** would either show up somewhere at the end of an instantiation or in the middle of an instantiation, as shown in the example instantiations below.

```
mod1 u1 (.bus1(new1), ... , .bus2(new2) /*AUTOINST*/ );

mod2 u2 (.bus1(new1), /*AUTOINST*/ .out1(out1), ...);
```

When **/*AUTOINST*/** was at the end of the instantiation, we needed to substitute **, .*** (with leading comma) because **.*** was the last port in the list. When **/*AUTOINST*/** was somewhere in the middle of the instantiation, we needed to substitute **.*,** (with trailing comma) to continue the port list. The substituted example code is shown below.

```
mod1 u1 (.bus1(new1), .*, .out1(out1), ...); // .* in middle

mod2 u2 (.bus1(new1), ... , .bus2(new2), .*); // .* at end
```

The two **vim** commands that we used to make the substitutions were:

```
:%s/\/\*AUTOINST\*\/)/, .*/g
```

16 copies of **/*AUTOINST*/)** were modified. These were non-collapsable module instantiations.

```
:%s/\/\*AUTOINST\*\// .*,/g
```

29 copies of **/*AUTOINST*/** were modified. These were collapsable module instantiations.

With **.*** instatiations in place, we then collapsed the ports (C-cC-k) and measured the improvement. We were somewhat disappointed to see that we still had 2643 lines of code. This was only a reduction of 167 lines of code or a 7.1% reduction.

We have not finished exploring the existing design but we have made some interesting observations. The design uses a large number of internal signals written in all lower case, connected to instantiated module ports written with a leading upper case character. Obviously, **.*** implicit ports will not match these opposite-case port and net names. With a few substitutions to match port-name-case to net-name-case, we were able to again go through the process of verilog-auto-inject and **AUTOINST**-**.***-substitution and we now measured the design to have 2184 lines of code. This represents a reduction of 646 lines of code or a 29.6% reduction in code.

We have noticed that there are other net names that could be modified to match the port names at one end of the connection and yet other simplifications that would also help to automatically reduce the code that would be needed to build the top-level design.

There are a couple of interesting observations from this experiment:

(1) Besides the reduction in coding effort, the designs have stronger port type checking over Verilog named port connections. Since this is a production design, we were not entirely surprised that all of the port connections were correctly sized; however, had SystemVerilog **.*** ports been used on the original design, we anticipate that more bugs would have been discovered earlier in the design process.

(2) We believe a greater reduction in coding effort is possible if a sane naming convention is used in anticipation of adopting **.*** implicit ports. The retro-fit exercise showed that some rather "interesting" naming conventions and configuration techniques had been employed. These techniques were obviously used to help engineers follow the large amount of confusing code.

I hope to perform many more experiments with large ASIC designs. I would not be surprised to find a few more subtle bugs in the EMACS mode but Wilson has a great record for quickly fixing bugs in his EMACS mode.

6.2 Signal naming conventions

There are many design teams that have adopted signal naming conventions that are incompatible with the use of **.*** or **.name** port connections. Many of these naming conventions appear to have been formulated to help engineers track all the signals that could occur in large designs.

Two common naming conventions include:

(1) module ports named with a leading "**p**_" (port signal) prefix while internal signals have a leading "**i**_" (internal signal) prefix. There are many variations of this naming convention using different prefix characters or variations that only use the prefix characters on either the port signals or the internal signals but not both.

(2) module ports with upper case first character and internal signals with the same name only the leading character is lowercase.

Design teams would do well to review their project naming conventions with respect to the new capabilities offered by SystemVerilog implicit port connections. Old limitations and assumptions that lead to existing company coding guidelines might not be applicable any more. Many coding guidelines could be streamlined to facilitate more efficient top-level design.

7 Arrays of instances & generate statements

The **.*** and **.name** implicit port connections can both be used with arrays of instances and generated arrays. These capabilities are described in this section.

The simple example design used in this section is a 4-stage pipeline register using 16-bit register with a chip-eneable (ce) load pin. The code for the 16-bit register is shown in section 7.1.

The pipeline register instantiates four copies of the 16-bit register. Simple Verilog named port connections are shown in section 7.2. The simple .* implementation is shown in section 7.3. A pipelined array of instance implementation with some ports connected using .* implicit ports is shown in section 7.4. A generated pipelined implementation with some ports connected using .* implicit ports is shown in section 7.5.

7.1 Pipeline register

The simple 16-bit register code is shown in Example 9.

```
module reg16 (
  output logic [15:0] q,
  input        [15:0] d,
  input               ce, clk, rst_n);

  always_ff @(posedge clk or negedge rst_n)
    if      (!rst_n) q <= '0;
    else if (   ce) q <= d;
endmodule
```

Example 9 - 16-bit register - to be instantiated into 4-stage pipeline register designs

7.2 Pipelined design - named port connections

The simple Verilog code for the pipelined register using named port connections is shown in Example 10.

```
module pipeline_reg1 (
  output [15:0] q,
  input  [15:0] d,
  input         ce, clk, rst_n);

  logic [15:0] n [1:3];

  reg16 u3 (.q(q),    .d(n[3]), .ce(ce), .clk(clk), .rst_n(rst_n));
  reg16 u2 (.q(n[3]), .d(n[2]), .ce(ce), .clk(clk), .rst_n(rst_n));
  reg16 u1 (.q(n[2]), .d(n[1]), .ce(ce), .clk(clk), .rst_n(rst_n));
  reg16 u0 (.q(n[1]), .d(d),    .ce(ce), .clk(clk), .rst_n(rst_n));
endmodule
```

Example 10 - 4-stage pipeline register with Verilog named port connections

7.3 Pipelined design - named port connections with .*

The simple Verilog code for the pipeline design implementated using **.*** implicit port connections is shown in Example 11.

```
module pipeline_reg2 (
  output [15:0] q,
  input  [15:0] d,
  input         ce, clk, rst_n);

  logic [15:0] n [1:3];

  reg16 u3 (          .d(n[3]), .*);
  reg16 u2 (.q(n[3]), .d(n[2]), .*);
  reg16 u1 (.q(n[2]), .d(n[1]), .*);
  reg16 u0 (.q(n[1]),           .*);
endmodule
```

Example 11 - 4-stage pipeline register with SystemVerilog .* implicit port connections

7.4 Pipelined design - arrays of instance

The simple Verilog code for the array-of-instance pipelined implementation using **.*** implicit port connections is shown in Example 12.

```
module pipeline_aoi (
  output [15:0] q,
  input  [15:0] d,
  input         ce, clk, rst_n);

  logic [15:0] n [1:3];

  reg16 u[3:0] (.q({q,n[3],n[2],n[1]  }),
                .d({  n[3],n[2],n[1],d}), .*);
endmodule
```

Example 12 - 4-stage pipeline register using an array of instance and SystemVerilog .* implicit ports

When there is a common bus or net that connects to a common port of the same size and name on all copies of the arrayed instances, **.*** will make the common connection to each instance. In this example, a 4-deep pipeline of **`reg16`** modules is instantiated using an array of instance. The **`ce`**, **`clk`** and **`rst_n`** port connections are common to all four pipeline stages so **.*** can be used to make those connections.

If there is a instance port bus that connects to multiple buses, the names will not match and named port connections will be required. In this example, the **`q`**-outputs and **`d`**-inputs of the pipeline registers connect to buses with different names so these parts are connected with appropriately sized concatenated bus names with named port connections.

7.5 Pipelined design - generate instances

The simple Verilog code for the generated pipelined implementation using **.*** implicit port connections is shown in Example 13.

```
module pipeline_gen (
  output [15:0] q,
  input  [15:0] d,
  input         ce, clk, rst_n);

  logic [15:0] n [0:4];
  genvar i;

  assign q    = n[4];
  generate for (i=0; i<4; i++) begin: R
    reg16 u1 (.q(n[i+1]), .d(n[i]), .*);
  end endgenerate
  assign n[0] = d;
endmodule
```

Example 13 - 4-stage pipeline register using a generate statement and SystemVerilog .* implicit ports

Just like the arrayed instance example, in this example, a 4-deep pipeline of **reg16** modules is instantiated using a **generate**-loop. The **ce**, **clk** and **rst_n** port connections are common to all four pipeline stages so **.*** can be used to make those connections.

And in a manner similar to the arrayed instance example, the **q**-outputs and **d**-inputs of the generated pipeline registers connect to buses with different names so these parts are connected named port connections.

7.6 Array Of Instance - Arrayed Instance Bus Matching

If a higher-level bus is connected to an instance array of 1-bit ports by the same name and if the instance array size matches the higher-level bus size, then **.*** can be used to make the arrayed instance port connections.

```
module ibuf (
  output logic a,
  input  logic din);

  buf b1 (a, din);
endmodule
```

Example 14 - 1-bit input buffer

The **ibuf** module of Example 14 has ports named **a** and **din**. When this **ibuf** is instantiated as an array of eight instances, the 1-bit ports become buses with names **a[7:0]** and **din[7:0]**, which match the port names and sizes of the top-level **ibuf_mod2** module of Example 15, so the **.*** ports are implicitly connected.

```
module ibuf_mod2 (
  output logic [7:0] a,
  input  logic [7:0] din);

  ibuf i[7:0] (.*);
endmodule
```

Example 15 - ibuf-top-module with ports that match the arrayed ibuf instance ports

8 Interfaces

The SystemVerilog **`interface`** construct allows the grouping of multiple signals into a single bundle that can be instantiated into a design and used to connect two or more sub-modules in a design.

An in depth description of interfaces and their usage is beyond the scope of this paper, but a quick description of how interfaces can work with **.*** implicit ports is appropriate.

8.1 SystemVerilog Interface Usage with .* Implicit Ports

Can the SystemVerilog **`interface`** construct and **.*** implicit ports be used together?

The answer to this question is, absolutely yes!

An interface is instantiated with an instance name. If that instance name matches the interface reference handle name in an instantiated module, the connection in the top-level module can be made using a **.*** implicit ***interface***-port connection.

If there are ports on an instantiated interface that match signal names and sizes of declared nets and buses in the enclosing module, then the matching ports to the interface can also be connected using **.*** implicit port connections.

9 Other Issues

Other issues related to port connections are worth noting. These miscellaneous issues are documented in this section.

9.1 Limitation - gate-level netlists

Will engineers use **.name** or **.*** on a gate-level netlist? No!

Why?

Most gate-level netlists instantiate 100's to 1000's of primitives many with input ports named **a**, **b**, **c**, **d**, and many with output ports named **y** and **q**. Using implicit port instantiations would short together 100' to 1000's of unrelated ports.

Implicit port connections help designers at the top-levels of a design and with block-level testbenches but they do not help with low-level netlists, which contain 100's to 1000's of same-named ports.

The good news is, low-level testbenches are generally generated by synthesis or netlisting tools and the tools do a good job of creating low-level netlists with named port connections.

9.2 EMACS Availability

I was on a customer site to teach a class and asked for permission to retro-fit one of their existing designs using the EMACS mode tricks. Nobody in that particular class was an EMACS user and we had troubles making the local EMACS installation recognize the EMACS mode files. Due to very limited time constraints, we did not have time to get the EMACS mode running at that company.

I am sure with time and perhaps some System Administration help, we could have figured out the EMACS mode startup issues. The point is, if nobody has setup the EMACS mode at your company, there may be some startup overhead to make the EMACS Verilog mode work.

9.3 Regular Expression Matching

Many engineers have asked for the ability to use regular expressions with **.*** implicit ports. At this time, the EDA vendors on the SystemVerilog standards group are unwilling to take on that challenge. It would be a nice capability, but one that the vendors are unwilling to support at this time.

10 Recommended Warnings & Errors

There are warnings and errors that vendors could issue that would help users find bugs in their designs. Although these warnings and errors are not required by the IEEE SystemVerilog Standard,

Although not required by the SystemVerilog standards documents, there is feature that tool vendors could add to their SystemVerilog compilers or to linting tools to assist users to find implicit port connection problems.

Whenever a instantiated module port is mistakenly connected to the wrong net, it generally means that some other net at the top-level is only connected at one end of the correct net.

Compilers or linting tools could check all nets in a design and report:

(1) Nets that do not have drivers (indicating that one or more inputs are connected together but no output is present on the net). This is almost always an error.

(2) Nets that only have one or more drivers but no receivers (indicating that there are no inputs connected to the net).

Both of these conditions are likely to be errors and will help the savvy designer to investigate anomalous cases. This would most likely catch 90-100% of all incorrectly named implicit net connections.

Users unite! Ask your vendors to support additional warnings and errors related to connectivity (not just declarations).

11 Submit Your Design Experiences!

Cliff is trying to collect as many design experiences as he can to add to this research. Do you have an existing large ASIC or FPGA design that you would like to retro-fit with **.*** implicit ports? If so, please follow these steps and email the results to Cliff Cummings. Cliff will continue to post updates to this paper on the www.sunburst-design.com web page with more data about a variety of different ASIC & FPGA projects.

To get accurate and consistent measurements for each ASIC or FPGA design, please copy the original top-level design to a new file and do the following:

(1) Remove all comments (we are interested in changes in executable code).
(2) Remove all blank lines.
(3) Count and record the number of lines of code.
(4) Count and record the number of characters in the code.
(5) Save the temporary file.

With the EMACS Verilog mode properly installed and running (see the Veripool web site[6] for file downloading and installation instructions), do the following:

(6) Open the temporary file with EMACS
(7) Instrument the design using the command-key sequence: (C-cC-z)
(8) Replace all occurrences of **`/*AUTOINST*/`** with **`.*,`** (if **`/*AUTOINST*/`** was at the end of the instantiation port list, you will need to remove the extra "**`,`**" before the closing "**`)`**").
(9) Execute the command sequence (C-cC-K) to collapse the **.*** ports.
(10) Count and record the new number of lines of code.
(11) Count and record the new number of characters in the code.

Please submit the following if permitted (otherwise, default values are shown in parentheses):

- Company Name - (company #n).
- ASIC Type - (blank).
- Gate Count - (blank).
- Lines of code before .* auto-insertion (required).
- Lines of code after .* auto-insertion (required).

- Any issues running the EMACS Verilog mode? (Cliff will help submit any suspected bugs to Wilson for corrections or updates).
- Would you like to be acknowledged in the paper? If so, let Cliff know that it is okay to use your name in the acknowledgements section of the updated paper (name will be left anonymous).

Cliff will keep confidential any information that you do not want to be made public.

12 Conclusions

For large top-level ASIC and FPGA designs **.*** implicit port connections facilitate the assembly of the top-level design with less effort and stronger port type checking.

The ability to expand and collapse **.*** ports using the EMACS Verilog mode removes the greatest concern about debugging a design using the powerful but abstract **.*** implicit ports.

When starting a new ASIC or FPGA design, choose a naming convention that will allow maximum usage of **.*** implicit ports and put the **.*** at the end of each port list to take advantage of the EMACS Verilog mode.

13 Acknowledgements

Supreme kudos to Wilson Snyder for implementing **.*** port expansion and collapsing in the EMACS Verilog mode.

14 References

[1] Clifford E. Cummings, “SystemVerilog Implicit Port Connections - Simulation & Synthesis,” *DesignCon 2005 (Santa Clara, CA)*, February 2005. Also available at www.sunburst-design.com/papers

[2] Erich Whitney, personal communication.

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

[4] "IEEE Standard For SystemVerilog - Unified Hardware Design, Specification and Verification Language," IEEE Computer Society, IEEE, New York, NY, IEEE Std 1800-2005

[5] SystemVerilog 3.1a Language Reference Manual, Accellera’s Extensions to Verilog, Accellera Organization, Inc., Napa, CA, 2004. Available at www.eda.org/sv

[6] Veripool web site: www.veripool.com

[7] Wilson Snyder, personal communication.

15 Revision 1.1 (November 2007) - What Changed?

Updates were made to section 4.3 regarding .* and .name rules. Recent exchanges by IEEE SystemVerilog Standards Group members clarified that the IEEE1800-2005 Standard omitted the requirement that unconnected ports be listed when using **.name** implicit port instantiations. Rules (6) & (8) were updated with important information related to the clarifications.

16 Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 25 years of ASIC, FPGA and system design experience and 15 years of Verilog, synthesis and methodology training experience.

Mr. Cummings has presented more than 80 SystemVerilog seminars and training classes in the past five years and was the featured speaker at the world-wide SystemVerilog NOW! seminars.

Mr. Cummings has participated on every IEEE & Accellera Verilog, Verilog Synthesis, SystemVerilog committee, and has presented some 40 papers on Verilog & SystemVerilog related design, synthesis and verification techniques.

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

Sunburst Design, Inc. offers World Class Verilog & SystemVerilog training courses. For more information, visit the www.sunburst-design.com web site.
Email address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers
(Last updated November 12th, 2007)