***Expert* Verilog, SystemVerilog & Synthesis Training**

SystemVerilog Implicit Port Connections - Simulation & Synthesis

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

Abstract

The Accellera SystemVerilog language[3] includes two new features designed to remove much of the tedium and verbosity related to building top-level ASIC and FPGA designs from instantiated sub-blocks. These enhancements permit one of two forms of implicit port connections

NOTE: An updated copy of this paper can be found at www.sunburst-design.com/papers

1. Implicit port connections

Verilog[2] and VHDL both have the ability to instiantiate modules using either positional or named port connections. Positional ports are subject to mis-ordered incorrect connections, which is why most experienced companies have internal guidelines requiring the use of named port connections. Unfortunately the use of named port connections in a top-level ASIC or FPGA design is typically a very verbose and redundant set of connections that requires multiple pages of coding to describe. Often, most of the top-level module port names match the equivalent net or bus connections.

Whenever a design review is conducted using a verbose top-level model, the reviewing engineers always ask the same question, “did you simulate it?” The instantiations are so tedious and verbose that nobody intends to read and verify every connection in the top-level HDL design.

SystemVerilog[3] addresses the top-level verbosity issue with two new concise and powerful implicit port connection enhancements: **`.name`** and **`.*`** connection.

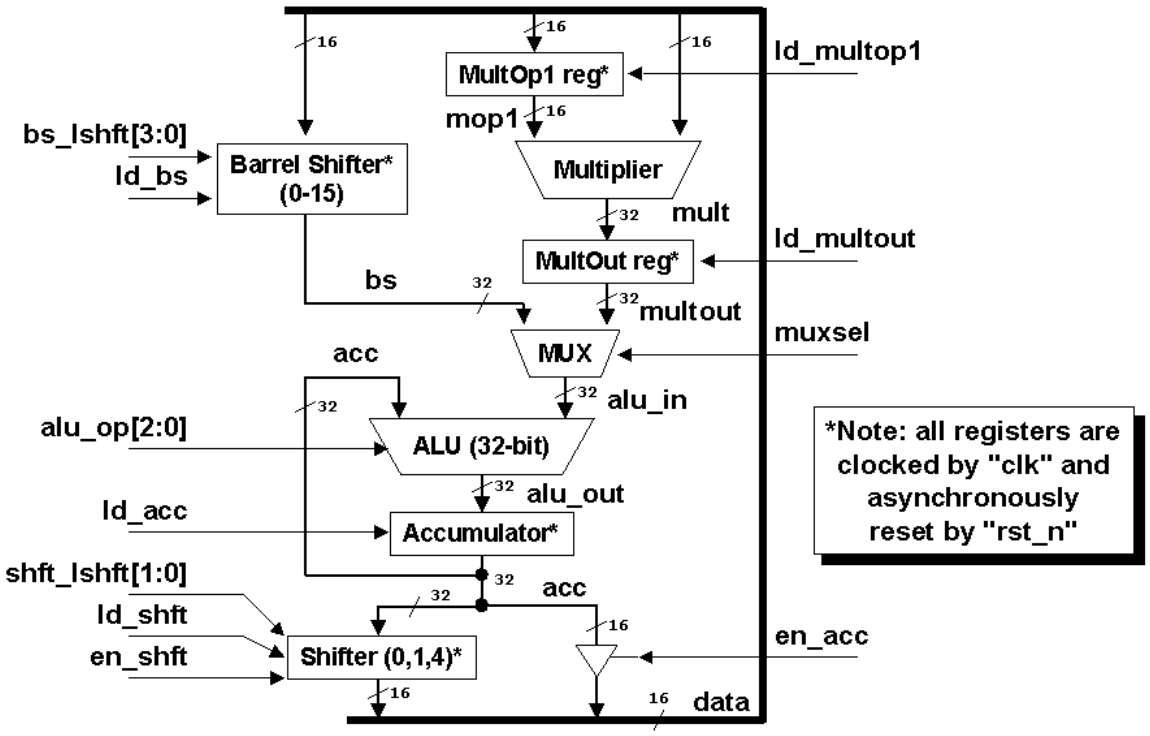

Figure 1 - Central Arithmetic Logic Unit (CALU) Block Diagram

Figure 1 shows a re-drawn version of the Texas Instruments First-Generation TMS320 CALU block diagram[1]. In this paper, this simple model will be built by instantiating each of the shown sub-modules, using multiple instantiation methods, into top-level **`calu`** modules.

2. Different port connection styles

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

The styles are compared for coding effort and efficiency.

2.1 Verilog positional port connections

Verilog has always permitted positional port connections. The Verilog code for the positional port connections for the CALU block diagram is shown in Example 1. The model requires 31 lines of code and 679 characters.

```
module calu1 (
  inout  [15:0] data,
  input  [ 3:0] 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            (alu_in, multout, bs,
                                  calu_muxsel);
  alu            alu            (alu_out, , ,
                                  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[15:0],
                                  en_acc);
endmodule
```

Example 1 - CALU model built using positional port connections

2.2 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 an 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 CALU block diagram is shown in Example 2. The model requires 43 lines of code and 1,019 characters.

```
module calu2 (
  inout  [15:0] data,
  input  [ 3:0] 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(bs),
                                  .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 2 - CALU model built using named port connections

2.3 The *.name* implicit port connection enhancement

SystemVerilog introduces the ability to do **.name** implicit port connections. 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 3. The model requires 32 lines of code and 756 characters.

```
module calu3 (
  inout  [15:0] data,
  input  [ 3:0] 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(bs),
                                 .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 3 - CALU model built using *.name* implicit port connections

2.4 The .* implicit port connection enhancement

SystemVerilog also introduces the ability to do **.*** implicit port connections. Just like the **.name** implicit port connection enhancement, 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 23 lines of code and 517 characters.

```
module calu4 (
  inout  [15:0] data,
  input  [ 3:0] 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(bs) , .sel1(calu_muxsel));
  alu            alu            (.* , .zero(), .neg());
  accumulator    accumulator    (.*);
  shifter        shifter        (.*);
  tribuf         tribuf         (.* , .acc(acc[15:0]));
endmodule
```

Example 4 - - CALU model built using .* implicit port connections

3. Important implicit port connection rules

There are six important rules related to the implicit port connection enhancements. They are:

(1) **`.name`** and **`.*`** implicit ports are not allowed to be mixed in the same instantiation. Instantiating one module with **`.name`** implicit ports and another module with **`.*`** implicit ports is permitted.
(2) **`.name`** or **`.*`** implicit ports are not allowed to be mixed in the same instantiation with positional port connections.
(3) A named port connection is required if the port size does not match the size of the connecting net or bus. For example: a 16-bit **`data`** bus connected to an 8-bit **`data`** port requires a named port connection to show which of the 16 bits are connected to the 8-bit **`data`** port.
(4) A named port connection is required if the port name does not match the connecting net or bus name. For example the 32-bit pad address named **`paddr`** connecting to a 32-bit **`addr`** port would require a named port connections **`(... .addr(paddr), ...);`**
(5) A named port connection is required if the port is unconnected. For example, if the above instantiations have an unconnected bus error (**`berr`**) port, the unconnected port must be listed as a named empty port **`(... .berr(), ...);`**
(6) All nets or variables connected to the implicit ports must be declared in the instantiating module, either as explicit net or variable declarations or as explicit port declarations.

Rule #6 requires that 1-bit nets be declared if the net is to be implicitly connected to a port of the instantiated module. Similarly, multi-bit buses must still be declared. Implicit port connection does not support automatic 1-bit net declaration.

My experience so far suggests that typically only a few dozen additional wire declarations are required to take advantage of the **`.name`** and **`.*`** implicit port connection enhancements. The **`.*`** enhancement can reduce 10 pages of top-level ASIC or FPGA instantiation code down to three pages of equivalent code while highlighting the differences in the port connections.

4. Stronger port connection-typing

An interesting side-effect of the implicit port connection enhancements is that not only are the coding styles more concise and less error prone, but the coding style actually imposes some VHDL-like stronger typing on the port connections that did not previously exist in Verilog.

The Verilog Standard allows connections of unequal sizes and then issues a port-size mismatch warning when the design is elaborated. The **`.name`** and **`.*`** implicit instantiation enhancements require that all sizes be matched; hence, reducing port-size instantiation errors.

Verilog allows unconnected ports to be omitted from the instantiation port list. The **`.name`** and **`.*`** implicit instantiation enhancements require that all unconnected ports be listed; hence, reducing instantiation errors related to accidental omission of ports.

Verilog does not require declaration of 1-bit nets and declaring 1-bit nets does not increase the name checking of 1-bit nets. The **`.name`** and **`.*`** implicit instantiation enhancements require that connections be made to declared nets and variables in the instantiating module. This means that declarations will be required and tested in the instantiating module without the onerous use of the Verilog-2001 **`` `default_nettype none ``** directive (which also requires the keyword wire to be added to all net-ports).

The SystemVerilog designer will now get stronger size and declaration checking with an enhancement that reduces top-level RTL coding by as much as 70%. A very nice trade-off!

The **`.*`** implicit port instantiation enhancement not only offers better port checking, it also makes the code more concise and highlights net-connections that are exceptions to like-named connections. Reviewers will more easily focus on the important parts of an upper-level netlist as opposed to pages of redundant and error-prone verbose connections.

5. Potential implicit port connection problems

There is a new type of potential error associated with implicit port connections: what if a port name in an instantiated module accidentally and unintentionally matches a net name in the instantiating module? The **`.*`** implicit connection enhancement will erroneously connect the same-named port and net together and it will have to be debugged (a bug which may not be easy to find). This problem is similar to the potential misconnection caused by scripts that

automatically generate named port lists. In both cases, the wrong port may be connected to a same-name net.

The **.*** implicit port instantiation enhancement was actually added to the SystemVerilog language at the request of Intel engineers that had a very similar capability with Intel's internal IHDL language, so the SystemVerilog committee took the opportunity to ask Intel engineers if they had encountered significant difficulties debugging the problem described above. Intel engineers reported that using IHDL they had seen the above problems but that the problems were rare and relatively easy to find and correct.

6. Intelligent Tools

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 nets that do not have drivers (indicating that one or more inputs are connected together but no output is present on the net) and 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.

7. 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's 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, gate-level netlists are generally generated by synthesis or netlisting tools and the tools do a good job of creating low-level netlists with named port connections.

8. Naming conventions

Many companies employ net-naming conventions that add prefixes and suffixes to net names if they are I/O pads or connected to module ports or if they are internal signals. Engineers from these companies have already expressed interest in having SystemVerilog enhanced with some form of regular expression matching and generation capabilities.

Although the request is very interesting, most tool vendors on the SystemVerilog committees do not relish the idea of adding regular expression capabilities to their tools and it is doubtful that such enhancements will be readily added to SystemVerilog standards in the near future.

Companies that employ the prefix-suffix naming conventions have two obvious choices if they would still like to use **`.name`** or **`.*`** implicit port connections:

(1) create an in-house tool that can take a prefix-suffix-style top-level netlist and convert it to a **`.name`** or **`.*`** implicit port connection-friendly style, then use existing SystemVerilog tools,

or

(2) make a change to their in-house naming convention strategies to take advantage of the **`.name`** or **`.*`** implicit port connection enhancements.

The second option may be very attractive considering the connection and debug capabilities of the implicit port connection enhancements.

9. Debugging & the .name compromise

In reality, when **`.*`** was first proposed as an implicit port instantiation enhancement to the SystemVerilog language, there were some members of the SystemVerilog standards group who were very uncomfortable about what this enhancement would do to debugging efficiency and design clarity because it effectively hid module port names for all instantiated modules connected using the **`.*`** implicit port connections.

The debug-concern related to hidden **`.*`** port connections is reminiscent of similar concerns held by experienced schematic-based design engineers about 5-10 years ago. Schematic-based designers used to debug designs by tracing visible wires between blocks on one or multiple schematic pages. There are no visible wires connecting module ports in an RTL design, which made some designers very nervous about the ease of debugging an RTL design.

RTL designers quickly adapted to RTL designs by using the search command in their favorite text editors. Debugging was not any more difficult, it was just different!

Similarly, there have been RTL designers who have expressed concern about the ease of debugging an RTL design with hidden **`.*`** implicit ports. An engineer cannot easily search within their favorite text editor to find the end points of nets for designs assembled with **`.*`** implicit

ports. This makes some experienced RTL designers very nervous about the ease of debugging an RTL design with **.*** implicit port connections.

SystemVerilog RTL designers are quickly adapting to the hidden **.*** implicit ports by using the UNIX grep command to find all of the Verilog files within the same directory that are connected to common net. Again, debugging is not any more difficult, it is just different!

Recognizing that some engineers would be hesitant to employ the design-abstraction imposed by the **.*** implicit port connection enhancement, the SystemVerilog standards group added the **.name** implicit port connection style as a very useful compromise to the abstraction imposed by **.*** implicit ports.

Engineers who insist on seeing all of the port connections in the top-level models will use the **.name** implicit port connection style, which offers all the benefits of the strong SystemVerilog port type checking, all the advantages of named port connections, and still reduces the amount of code required to build a top-level design to an effort close to what engineers now have with positional port connections.

10. Block-level testbenches using .* implicit ports

Assembling a block-level testbench is simple using the **.*** implicit port connections. The block-level testbench for the **calu** models (shown in Example 5) was built using the steps outlined below:

To build a block-level testbench:
(1) copy the header and declarations from the Design Under Test (DUT) module and paste the header and declarations into a testbench file.
(2) change all DUT **inout** port declarations into testbench **wire** declarations.
(3) change all DUT **input** port declarations into testbench **reg** declarations (or **logic** declarations).
(4) change all DUT **output** port declarations into testbench **wire** declarations (or **logic** declarations).
(5) if the DUT port declarations were Verilog-2001 ANSI-C-style port declarations, each end-of-line must be changed from a "**,**" to a "**;**"
(6) instantiate the DUT with **.*** (all ports now match declared testbench variables).
(7) add appropriate stimulus to test the DUT.
(8) add appropriate verification code for the DUT.

The preceding steps were used to build a block-level testbench for the **calu4** module of Example 4 (the same block-level testbench techniques would also work for the Verilog-2001 **calu** blocks shown in Example 1 and Example 2, or the SystemVerilog **calu** block of Example 3).

```
module blk_tb;
  wire   [15:0] data;
  reg    [ 3:0] bs_lshft;
```

```
  reg    [ 2:0] alu_op;
  reg    [ 1:0] shft_lshft;
  reg           calu_muxsel, en_shft, ld_acc, ld_bs;
  reg           ld_multop1, ld_multout, ld_shft, en_acc;
  reg           clk, rst_n;

  calu4 u1 (.*);  // DUT instantiated using .*

  initial begin // Stimulus
  ...

  initial begin // Verification
  ...
endmodule
```

Example 5 - Block-level testbench for the calu models using SystemVerilog .* implicit port-connections

Step (6) is the biggest difference between block-level testbenches using Verilog-2001 and SystemVerilog. No longer is it necessary to elaborate all the named port connections, now the DUT can be easily instantiated using **.*** implicit port connections.

11. Use it right! Don't blame the EDA tools!

Some EDA tool developers are worried about this enhancement because they are concerned that engineers will use it wrong, blame the EDA tools when errors are reported and tie up EDA support resources to debug engineering mistakes. This is a valid concern - untrained engineers making mistakes and blaming the EDA tools.

To all engineers that intend to use this extremely powerful enhancement - when EDA tools report compile errors, please examine your code carefully before pointing the finger at the EDA vendor. We do not want to give EDA tool development engineers reason to reject future powerful enhancements due to a few untrained engineers!

I believe that the stronger port-typing will actually remove more support problems than will be introduced by untrained engineers using the **.*** enhancement incorrectly.

12. Conclusions

Both **.name** and **.*** implicit port instantiation enhancements offer stronger port type checking with a much more powerful and much more concise coding style.

The concise nature of the **.*** implicit port connections will show more design blocks per page, emphasize where there are net-port connection differences and will speed the development of top-level designs.

These enhancements were added to the SystemVerilog language to help the design engineers to complete the job more quickly.

13. Tool versions & command aliases

For this paper, Verilog and SystemVerilog experiments were conducted using the ModelSim Verilog simulator version 6.0, VCS Verilog simulator, version 7.2, and Design Compiler (DC) with Design Vision GUI, version V-2004.06-SP2. At the time of this publication, Design Compiler required a special license to enable recognition of SystemVerilog features. Engineers interested in using Design Compiler SystemVerilog features should contact their local Synopsys sales or field personnel. I also used the following command aliases to run my Verilog-2001 & SystemVerilog simulations

ModelSim SystemVerilog (**`-sv`**) alias to: compile a SystemVerilog design

```
alias svlog="vlog -sv +define+SV"
```

VCS SystemVerilog (**`-sverilog`**) aliases to: compile & run; compile, run & dump

```
alias    svcsr="vcs     -R -sverilog +define+SV"
alias svcsdump="vcs -PP -R -sverilog +define+SV +define+VPD"
```

The **`+define+SV`** and **`+define+VPD`** options are Sunburst Design options to enable conditionally compiled SystemVerilog code and for dumping the VCS dumpfile format respectively.

NOTE: the **`-sverilog`** switch is new with VCS7.2. Earlier versions of VCS used the switch **`+sysvcs`** to enable SystemVerilog simulation. The **`-sverilog`** switch is an improvement because now both VCS and Design Compiler use a similar **`sverilog`** switch name, as shown below.

Within Design Vision, the tcl commands used to either read or analyze SystemVerilog files are:

```
read_sverilog <filename>
read_sverilog { <filename> <filename>... }

analyze -f sverilog <filename>
analyze -f sverilog { <filename> <filename>... }
```

Analyzing sub-module design files is important to using the **`.*`** implicit port connections enhancement, and is discussed in Appendix C.

14. Revision 1.1 (January 2005) - What Changed?

The examples in the first version of the paper mistakenly had the acc bus connected to one of the mux inputs instead of the bs bus. The bs bus was left dangling which was discovered when the design was synthesized and the barrel_shifter block was optimized away (since the bs output was not connected to any logic). The barrel_shifter also indicated it was possible to rotate by 0-16, but a rotate of 16 is equivalent to no rotation at all, so the rotation comment was changed to 0-15 and the extra rotate control signal was removed from the design.

As short section on using **`.*`** to build a simple block level testbench was added (Section 10).

Synopsys synthesis commands and explanation was added to Section 13.

Some of the sections in the paper were re-ordered into a more logical flow. The author contact information was moved to the end of the paper instead of the DesignCon prescribed beginning of the paper. There were also a number of minor typos in the paper that were corrected.

Module headers for the **`calu`** sub-blocks are now included in Appendix A at the end of this paper.

Many people have asked me for the VIM key-mapping that I use to auto-generate named port connections. The key-mapping and explanation are now included in Appendix B at the end of this paper.

Many people have asked why not just use the new SystemVerilog interface constructs to connect up the top-level designs. Although the new SystemVerilog interfaces allow engineers to encapsulate port connections and combine them with testing tasks and assertions, they can be somewhat verbose when used to connect multiple sub-blocks with different interfaces that must eventually connect to the ports of the top-level module. I am not convinced that interfaces with their respective overhead are well suited to building the top-level design of large ASICs or FPGAs. Since I am not sure if there are some unique tricks that one can employ to efficiently use to replace implicit port connections, I have included "**The Great Sunburst Design Interface / .* Implicit Ports Challenge!!**" This challenge can be found in Appendix C.

Note - to execute the challenge, you must already have a working knowledge of SystemVerilog interfaces.

15. Revision 1.2 (April 2005) - What Changed?

Minor typo corrections were made to Section 7. The end of the section erroneously referred to a low-level testbench instead of a gate-level netlist.

I also added minor correction and clarifications to the content of Appendix B - VIM Named Port Connections Macro.

References

[1] *First-Generation TMS320 User's Guide,* Section 3.5, Texas Instruments, 1989

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

[3] SystemVerilog 3.1a Language Reference Manual, Accellera's Extensions to Verilog, Accellera, 2004, freely downloadable from: www.eda.org/sv

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 23 years of ASIC, FPGA and system design experience and 13 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, and is currently a member of the IEEE P1800 SystemVerilog Standards Group. Mr. Cummings is the only Verilog and SystemVerilog trainer to co-develop and co-author all of the IEEE Verilog Standards, the IEEE Verilog RTL Synthesis Standard and all of the Accellera SystemVerilog Standards.

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

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

A freely downloadable and updated version of this paper can be found at the web site:
www.sunburst-design.com/papers
(Data accurate as of November 29, 2004)

Appendix A - CALU sub-module header files

This section contains the header files for the sub-modules that were used to test the **`calu`** designs. All of the header files use the Verilog-2001 ANSI-C style port headers.

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

Example 6 - multop1.v source file (header file only)

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

Example 7 - multiplier.v source file (header file only)

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

Example 8 - multoutreg.v source file (header file only)

```
module barrel_shifter (
  output [31:0] bs,
  input  [15:0] data,
  input  [ 3:0] bs_lshft,
  input         ld_bs, clk, rst_n);
  // RTL code for the barrel shifter
endmodule
```

Example 9 - barrel_shifter.v source file (header file only)

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

Example 10 - mux2.v source file (header file only)

```
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 11 - alu.v source file (header file only)

```
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 12 - accumulator.v source file (header file only)

```
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 13 - shifter.v source file (header file only)

```
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 14 - tribuf.v source file (header file only)

Appendix B - VIM Named Port Connections Macro

Many people have asked me for the VIM key-mapping that I use to auto-generate named port connections. The key-mapping and explanation of same follows.

In a Linux environment, edit the **.exrc** VIM startup file in the users home directory and add the following key-mapping command:

```
map = 0/(/<cr>i<cr><esc>:s/\([^  (),;][^      (),;]*\)/.\1(\1)/g<cr>kJ
```

Explanation of the commands and regular expression:

- **0** - go to the beginning of the line (the cursor can be anywhere in the instantiation line).
- **/(/<cr>** - search for the first "**(**" - Note: for modules instantiated with **#(parameter redefinition)** causes minor problems that are manually corrected after all the named ports are generated.
- **i<cr><esc>** - insert a new line at the first "**(**" to do the substitutions on a separate line. The lines are re-joined at the end of this command.
- **:s/** - VIM substitution command.
- **\([^***<blank><tab>***(),;][^***<blank><tab>***(),;]*\)** - collect each group of characters that does not include white space (***<blank><tab>***), parentheses, comma or semicolon. These are all the identifiers in the positional port connection-version of the instantiation. Note ***<blank>*** is just the space-bar pressed once and ***<tab>*** is the tab-key pressed once.
- **/** - end of search.
- **.\1(\1)** - replace each collected identifier with **.identifier(identifier)**
- **/g<cr>** - end of global replacement (global on this line).
- **kJ** - go up one line(**k**) and (**J**)oin the substituted line to the line with the module name and instance name.

To map the same command in a UNIX environment, carriage returns and escape characters are entered into the key-mapping as follows:

- **<cr>** - hold down the **<ctl-shift>** keys and then press the "**V**" key followed by the "**L**" key, then release the **<ctl-shift>** keys. You will see **^M**
- **<esc>** - hold down the **<ctl-shift>** keys and then press the "**V**" key followed by the "**ESC**" key, then release the **<ctl-shift>** keys. You will see **^[**

USAGE: To use this command, instantiate a module with positional ports all on one line and then position the cursor on this line and press the "**=**" key.

Appendix C - Challenge

The Great Sunburst Design Interface / `.*` Implicit Ports Challenge!! (January, 2005)

A number of my colleagues have given me limited amounts of grief over my enthusiasm for the SystemVerilog `.*` implicit ports enhancement. Many colleagues have strongly suggested that interfaces are always superior to "those dangerous `.*` implicit ports!"

Below you will find a `.*` / interface challenge that should be interesting and amusing!

The first engineers that show me a simple solution to the top-level **`calu`** design using interfaces will be prominently mentioned in a later version of the paper, *SystemVerilog Implicit Port Connections - Simulation & Synthesis*, along with their clever interface-implementation of the top-level **`calu`** design. This paper is posted on the www.sunburst-design.com/papers web page.

Note - to execute the challenge, you must already have a working knowledge of SystemVerilog interfaces.

I believe that `.*` works wonders for large top-level ASIC designs, and that interfaces are often cumbersome at this level. So here is the challenge:

A block diagram for a **`calu`** (Central Arithmetic Logic Unit - from the old first generation TI DSP Data Book) is shown in Figure 1. The code for Verilog positional ports (**`calu1.v`**) is shown in Example 1, the code for the Verilog named ports (**`calu2.v`**)is shown in Example 2, the code for the SystemVerilog **`.name`** implicit ports (**`calu3.v`**) is shown in Example 3, the code for the SystemVerilog `.*` implicit ports (**`calu4.v`**) is shown in Example 4, and the code for the **`calu`** sub-block header files is shown in Appendix A. Although the entire design could easily be coded in one or two modules, I have chosen to code each block separately and the challenge is to assemble the sub-blocks and get them to communicate with the top-level CALU ports.

This is comparable to actual ASIC top-level designs, that would connect multiple much-larger blocks to form the top-level ASIC.

As mentioned above, the header files for the sub-blocks are shown in Appendix A.

This design can be synthesized with Synopsys DC using the commands:

```
analyze -f  sverilog { accumulator.v alu.v barrel_shifter.v       \
                       multiplier.v multop1.v multoutreg.v mux2.v \
                       shifter.v tribuf.v }
read_sverilog calu4.v
set current_design calu4
compile
```

Example 15 - Synopsys DC TCL commands to synthesize the calu .* implementation

I would like to see anyone implement this design efficiently using interfaces. You should use at least one interface for the top-level **`calu`** ports, and you may optionally use one or more additional interfaces to connect the sub-blocks.

I know that interfaces have great value with SystemVerilog. I know that one can easily add testing tasks and assertions to the interface. I am just trying to find the balance between using interfaces and top-level implicit port connections.

One enhancement that I suggested in a 2004 Accellera SystemVerilog committee meeting (but never filed) was to allow connections to interface signals without using the interface_name .hierarchical notation that is so cumbersome whenever there was no overlap between interface signal names and module identifiers. One possibility was to instantiate an interface with a leading "**`default`**" keyword.

May we learn lots from this exercise and develop great guidelines or propose enhancements that will make interfaces easier and more attractive to users!!

Let the challenge begin!! (now we find out who really believes in their preferred SystemVerilog enhancement!)

Regards - Cliff **`.*`** Cummings - cliffc@sunburst-design.com