***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.

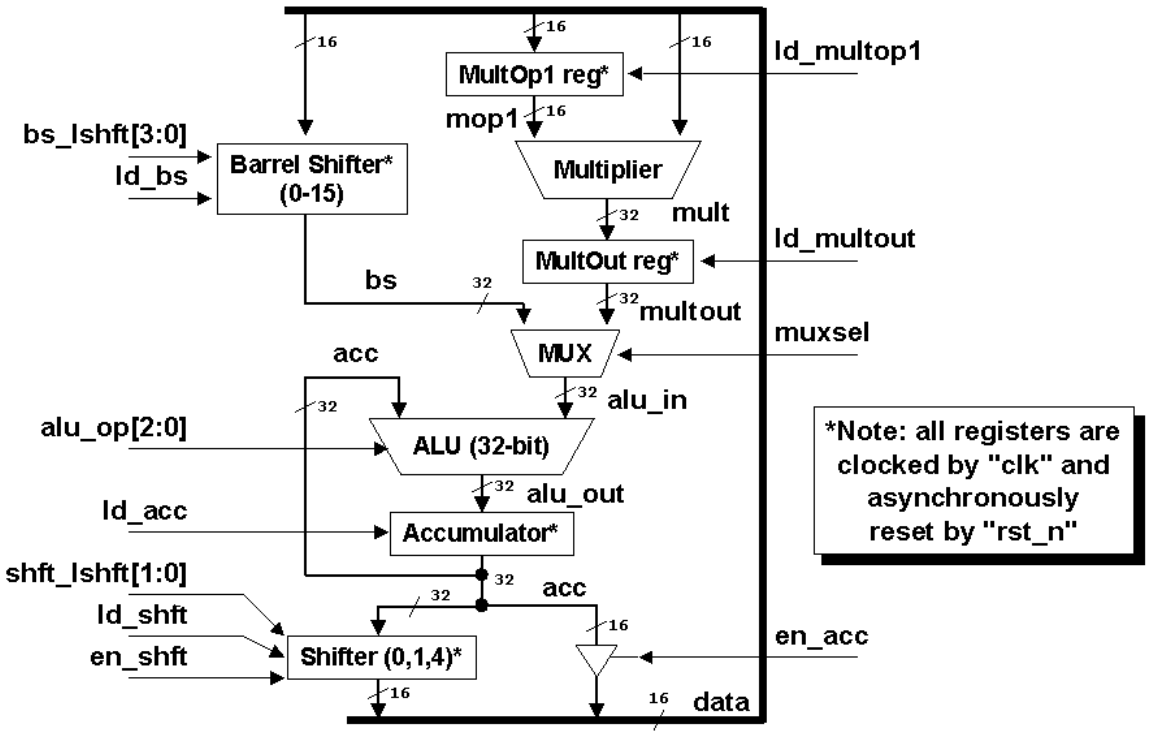


Figure 1 - Central Arithmetic Logic Unit (CALU) Block Diagram

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

THE IEEE VERILOG-2001 SIMULATION TOOL SCOREBOARD

Clifford E. Cummings - Sunburst Design, Inc., Beaverton, OR

POST-DVCon NOTES (Rev 1.2):

The testing of so many simulators and synthesis tools proved to be more than I could do in a reasonable amount of time, so this paper only focuses on simulators and the title of the paper was changed to reflect this fact. Perhaps I will get around to testing synthesis tools by the next DVCON.

Verilog-XL was removed from the compliance tables because Cadence does not intend to add support for Verilog-2001 features to Verilog-XL. See Section 12.0 for more details.

ABSTRACT

Verilog-2001 added many valuable enhancements to the IEEE1364-1995 Verilog Standard, but when can we safely use them? When the full suite of tools used by your company to do design all support Verilog-2001 enhancements, your company can safely start taking advantage of the enhancements.

This paper details a number Verilog-2001 coding examples and indicates which simulation tools support the enhancement. This paper is not intended to run performance benchmarks against the different simulation vendors and indeed does not include performance data. This paper is intended to inform the Verilog design and synthesis community which Verilog-2001 enhancements have been implemented by the various vendors so that the end-user can scan the list of vendors for implemented enhancements to determine when their company can start coding with the enhanced Verilog-2001 coding styles.

This paper includes multiple "scorecards" (tables) to show which simulation vendors support the important Verilog-2001 enhancements. The latest version of tools from major EDA vendors are represented on the "scorecards."

1.0 Introduction

The IEEE Verilog-2001 Standard introduced a number of enhancements intended to make designs more concise and more powerful. Stuart Sutherland has published a book on Verilog-2001 enhancements and ordered those enhancements by number. This paper re-orders the enhancements, according to user requested priorities and RTL-coding partitions, but I do cross reference the enhancements discussed in this paper with the enhancement numbers as reported in Sutherland's book for easy correlation.

At the time that this paper went to publication, I was not done testing as much as I had wanted. This paper will continue to be updated and readers are encouraged to go to the Sunburst Design web page referenced at the end of this paper to download copies of this paper with updated information.

2.0 Test Suite and Tool Versions

Simulation tools that were tested with the beta version of the Sunburst Design Basic Verilog-2001 Compliance Commercial Test Suite, included (abbreviations used in the tables):

[SR#] Sutherland Reference # - *Verilog 2001*
VCS - Synopsys, VCS version 7.0
SS - Synopsys SystemSim version 2.1.1 (the Superlog simulator)
MTI - Model Technology ModelSim version 5.7
NC - Cadence NC-Verilog version 4.2 (beta)
SIL - Simucad Silos version 2002.100
VXL - Cadence Verilog-XL - will not support Verilog-2001. See Section 12.0.

2.1 KEY - Table abbreviations

Some abbreviations were used in the tables shown in this paper. The following abbreviations were used in the compliance data tables.

X - Feature is supported
- - Feature not supported - syntax error reported
IG - Syntax was ignored - feature not supported
MSG - Tool recognized the syntax but gave a message indicating that the feature is not yet supported.

3.0 Top Five Enhancements

At the International Verilog Conference (IVC) in 1996, a "Birds Of a Feather" panel session was held where panelists and audience members submitted enhancement ideas and the entire group voted for the top-five enhancements that they wanted added to the Verilog language.

Although numerous enhancements were ultimately considered and many enhancements added to the Verilog 2001 Standard, the top-five requested enhancements were:

#1 - Verilog generate statement
#2 - Multi-dimensional arrays
#3 - Better Verilog file I/O
#4 - Re-entrant tasks
#5 - Better configuration control

3.1 #1 Generate Statements

Verilog generate statements are divided into three main groups: the generate for-loop, the generate if-else statement and the generate case statement. In conversations with vendors, the flexibility of the Verilog generate for-loops seems to be proving the most difficult aspect to implement of this requested enhancement.

As shown in Table 1, vendors have started to implement the generate statements but as of this publication, none of the vendors had supported nested generate for-loops. Future testing will also examine generate for-loops with non-contiguous incrementing and decrementing.

3.2 Array of Instance

It should be noted that most contiguous generate for-loops could be more easily coded using the Array of Instance construct that was added to Verilog-1995 and is now well supported by vendors. Engineers should think first about the array of instance and then fall back to a generate for-loop. For

instantiating a simple contiguous set of I/O pads, the array of instance is better supported and far simpler than an equivalent generate for-loop.

Table 1 - Verilog-2001 - The Top Five Requested Enhancements

[SR#]	Top-Five Requested Enhancements	VCS	SS	MTI	NC	SIL
	Verilog-1995 Array of Instance	X	X	X	X	X
[36]	(1) generate for-loop	X	X	X	-	??
[36]	(1) nested generate for-loop	-	-	-	-	-
[36]	(1) generate if-else	X	X	X	-	X
[36]	(1) generate if-else-if	X	X	X	-	**BUG**
[36]	(1) generate case	X	X	X	-	X
[15-17]	(2) multi-dimensional arrays	X	X	X	-	-
[16]	(2) 2-D array of reals	-	X	X	-	-
[30-31]	(3) enhanced file I/O	X	X	X	X	X
[30]	(3) file I/O opening files for modification	X	X	X	-	X
[7]	(4) automatic tasks	X	X	**MSG**	X	-
	(4) recursive functions (Verilog-1995)	X	X	X	X	-
[8]	(4) recursive automatic functions	X	X	**MSG**	X	-
[37]	(5) Verilog configuration files	-	-	-	-	-

?? - The generate for-loop is almost useless without multi-dimensional arrays

3.3 #2 Multi-Dimensional Arrays

Verilog-2001 permits the declaration and use of multidimensional arrays. Former Verilog-1995 restrictions that only allowed two dimensional arrays, and then only word access into the arrays, have been removed. The Sunburst suite tested 2-D arrays with word, part-select and bit access as well as 3-D arrays also with word, part-select and bit access. The suite also tested for 2-D declarations of real values.

3.4 #3 Enhanced File I/O

Verilog-2001 offers much more powerful file I/O and string manipulation capability over Verilog-1995. As of this date, the Sunburst suite is incomplete in testing all of the numerous new file I/O capabilities, but the suite will be expanded and used to do additional testing of vendor tools. The suite did open and close files using every new read, write and append mode, and did some other file I/O testing. Although the file I/O capabilities are now native to many simulation tools, users can still download a nearly identical set of capabilities using PLI routines from Chris Spear's web site, referenced at the end of this paper. Many of the Verilog-2001 file I/O enhancements were patterned after Chris' pre-existing file I/O PLI routines.

3.5 #4 Reentrant Tasks and Functions

Verilog-1995 tasks and functions use static variables, which means that a task with delays that is called a second time before the first invocation is finished, will share common-static variable, usually with undesirable results. Verilog-2001 allows users to add the keyword "automatic" to Verilog tasks and functions to force the automatic versions to dynamically allocated variables for each task or function call.

Some simulators have started to support automatic tasks and functions, while other simulators like ModelSim do not support this functionality yet but give a cute message that "this Verilog-2001 feature is not yet supported."

In the testing that I did with recursive function calls, I noted that some vendors even partially support recursive function calls in Verilog-1995 while others do not.

3.6 #5 Verilog Configuration Files

Verilog-2001 configuration files are intended to give the user better control of binding files to instances in a design during simulation (to replace the ugly and non-standard **`uselib** directive) while also offering a language method for selecting library directories (to replace the command line switches **-y** and **+libext**) as well as library files (to replace the **-v** command line switch). Verilog configuration files add new keywords such as **library**, **config-endconfig**, **design**, **default**, **liblist** and **instance**.

Unfortunately, none of the vendors tested supports any of the features of this valuable design-control enhancement.

4.0 The ANSI-Port Enhancements

ANSI style port enhancements provide a concise, non-redundant way to make port declarations in Verilog-2001. The most useful and powerful form of ANSI style ports is making port directions, data types and port names all in the same declaration and all vendors seem to support this correctly with one notable exception. Some vendors seem to have problems when port directions are separated from explicit wire declarations. Although the latter is the less important part of the enhancements, it is nonetheless annoying.

ANSI style parameters, vital to supporting parameterized reusable or re-sizable models is somewhat poorly supported or subject to bugs. Equally important is the ability to redefine parameters on an instance by instance basis, and support for the new named parameter passing capability is also somewhat shaky from some vendors.

Cleaning up ANSI style module ports and parameters should be a priority for every vendor. Under the category of credit where credit is due, ModelSim passed all of the ANSI port tests in the Sunburst Design Basic Verilog-2001 Compliance Test Suite. Kudos to the ModelSim team for getting it right!

Vendors have mixed records of success when it comes to extending ANSI styles to tasks, functions, User Defined Primitives (UDPs), etc., but again, these are not as commonly needed as the ANSI style ports. ANSI support for tasks and functions is still relatively important but should be implemented.

Table 2 - Verilog-2001 - ANSI-Port Enhancements

[SR#]	**ANSI Port Enhancements**	VCS	SS	MTI	NC	SIL
[1]	Combined port-data type declarations	X	X	X	X	X
[1]	Combined port-data types - explicit wires	X	**BUG**	X	X	**BUG**
[2]	ANSI style module ports	X	X	X	X	X
[3]	ANSI style parameters	-	**BUG**	X	X	X
[27]	Named parameter redefinition	X	X	X	X	X
[6]	ANSI style task/function ports	X	X	X	X	X
[4]	ANSI style UDP ports	-	-	X	-	X
[26]	Real & integer parameters	-	X	X	-	-
[26]	Sized parameters	**IG**	**IG**	X	**IG**	**IG**

5.0 The Fundamental RTL Enhancements

There is another subset of Verilog-2001 enhancements that is important to RTL coders and that should not be too hard to implement, this is the fundamental RTL enhancement subset.

5.1 Comma-Separated Sensitivity Lists

In VHDL, the signals in a process sensitivity list are separated by commas. In Verilog the signals are separated by the "**or**" keyword. This means that a Verilog sensitivity list is generally more verbose than an equivalent VHDL sensitivity list. I personally found this to be offensive! We added comma-separated sensitivity lists to Verilog and this really should be the simplest of enhancements to implement, but one vendor flagged the comma as an error when placed into a sensitivity list with posedges and negedges and another vendor did not support it at all! This should be fixed post haste!

5.2 The @* Combinational Sensitivity List

Even better than the comma separated sensitivity list is the very concise **@*** combinational sensitivity list. The **@*** basically was added to mean, if the synthesis tool wants it, then so does the simulator! Note that although, **@(*)** is also currently a legal form of this sensitivity list, there is some consideration being made by the Verilog standards groups to remove support for this form. The problem is that **(*** is also the opening delimiter for the new Verilog-2001 attributes, and tool vendors have complained about the difficulty in distinguishing between the two. I personally would support removal of the **@(*)** style, and if your company writes any in-house tools, adopting a coding standard that prohibits **@(*)** will probably make your tool-creation job easier.

Guideline: Use @* for combinational sensitivity lists and do not use @(*)

Some vendors are doing a good job of supporting the **@*** feature while one vendor has implemented the SystemVerilog **always_comb** statement, which is currently a slight super-set of the **@*** capability. It would be nice to have the **@*** capability implemented universally by all vendors.

5.3 Implicit Internal 1-bit Wire Declarations

Verilog-2001 no longer requires that 1-bit internal wires be declared that are driven from continuous assignments. They should come into existence automatically. This was a non-orthogonal wart on the Verilog-1995 language. Unfortunately, most vendors still have not fixed this for Verilog-2001.

5.4 `default_nettype none

Verilog-2001 also added a compiler directive to force engineers to declare all variables, including 1-bit wires and port wires for those misguided souls who think that more declarations equates to better design practices (actually you have now doubled the number of places where you can introduce a typo into your code but now a typo in a declaration will cause an error to show up on your good RTL code - hopefully you will spot the error in the declaration instead of severely altering RTL code before realizing the RTL code really was good the whole time). Only one vendor has correctly implemented this "feature" but I am not encouraging other vendors to spend much effort on this "enhancement" until the good stuff has been implemented.

5.5 X & Z Extension Past 32 Bits

Making an assignment of `bz or `bx in Verilog-1995 to a left-hand-side (LHS) variable that is larger than 32 bits causes only the 32 LSBs to be assigned the X and Z values. This is fixed in Verilog-2001 but only one vendor has made the change.

5.6 Variable Declaration Assignments

Verilog-2001 permits variables to be declared with and initial value. The initial value is assigned as if it were in an initial block so there is no guarantee that the initial value will be assigned first, and this coding style is generally not a good idea for synthesizable RTL coding (the real hardware may not be capable of initializing to the selected value, causing a mismatch between pre- and post-synthesis simulations).

Some vendors have implemented this feature, some still flag it as an error and one vendor just ignores the initializations. Although useful, this is not a "must-have" enhancement.

5.7 Enhanced Conditional Compilation

Verilog-2001 adds the **`ifndef** and **`elsif** conditional compilation compiler directives to the set that already includes **`ifdef**, **`else**, **`endif**. Again, all vendors with one notable exception have implemented this very useful enhancement.

5.8 Standardized Random!

Courtesy of Cadence, the Verilog-2001 Standard now includes the exact code used to generate random numbers so that a user can get the exact same random numbers using any Verilog simulator. Testing shows that all simulators have implemented the same **$random** system function. The Verilog-2001 standard also standardized the less frequently used random distribution functions and all vendors have similarly implemented these functions except one. The latter functions are not frequently used by RTL coders and are not as important to fix as other enhancements.

Table 3 - Verilog-2001 - Fundamental RTL Enhancements

[SR#]		VCS	SS	MTI	NC	SIL
[10]	Comma-separated - combinational logic	X	X	X	X	X
[10]	Comma-separated - sequential logic	X	X	X	X	X
[11]	@* combinational sensitivity list	X	**X**	X	X	X
[25]	Single attributes	X	-	-	-	-
[25]	Multiple attributes	-	-	-	-	-
[12]	Implicit internal 1-bit wires	-	-	X	X	X
[13]	`default nettype none	-	-	**BUG**	X	-
[23]	X & Z extension past 32 bits	-	-	-	X	-
[5]	Variable declaration assignments	X	**IG**	X	X	-
[34]	Enhanced conditional compilation	X	X	X	X	-
[26]	Standard random number generation	X	X	X	X	X
[26]	Standard distribution number generation	**BUG**	X	X	X	**BUG**

6.0 Signed Arithmetic and Power Operator

Signed arithmetic was a much requested enhancement to the Verilog-2001 language. Although all vendors have addressed signed arithmetic, a couple of the implementations have a few notable bugs. A rule of thumb to remember when working with Verilog signed arithmetic is that all operands and the destination must all be signed variables, otherwise you generally end up with an unsigned result.

The power operator has also been implemented by a subset of the simulation tool vendors.

Table 4 - Verilog-2001 - Signed Arithmetic & Power Operator Enhancements

[SR#]		VCS	SS	MTI	NC	SIL
[18-22]	Signed arithmetic	X	X	**BUG**	X	-
[24]	Power operator	X	X	X	-	-

7.0 IP Block Enhancements

Two enhancements added to the Verilog language to assist in the development of robust Intellectual Property (IP) blocks are the **`localparam`** and the constant function.

7.1 localparam

The new **`localparam`** keyword makes it possible to declare parameters that cannot be changed by **`defparam`** or other parameter redefinition techniques. The **`localparam`** will generally be calculated from other parameters that are passed to a module.

One example would be to calculate the memory depth of a RAM device based on the size of the address bus. Allowing a user to modify both the address bus size and the memory depth could lead to incompatible parameter values. Vendors have been slow to implement this useful enhancement.

7.2 Constant Functions

A constant function is a function that is run at compile time to calculate values that will be used to size portions of a design or to assign logical values to parameters based on other parameter values.

One example would be to calculate the *ceiling of the log base-2* of a number, such as calculating the number of address bits that are required to address an odd-sized memory.

Although a very useful enhancement, this is going to be a hard enhancement to implement and may take some time to show up in a vendor's tool. This enhancement is very important to IP developers who are trying to create complex parameterizable models.

Table 5 - Verilog-2001 - IP Block Enhancements

[SR#]		VCS	SS	MTI	NC	SIL
[28]	localparam	-	X	**BUG**	-	-
[9]	Constant functions	-	-	-	-	-

8.0 Miscellaneous Nice Enhancements

Other nice enhancements that were added to Verilog-2001 include enhanced +command option testing, constant part-select indexing and a standardized SDF **`$sdf_annotate`** command.

8.1 Enhanced +Command Option Testing

Verilog-1995 had the ability to recognize +command options using the **`$test$plusargs`** system function to initiate specific desired simulation activity. Veilog-2001 adds the ability to read the values of the +command options using the **`$value$plusargs`** system function. Implementation by vendors ranges from sporadic to buggy.

One interesting note is that verification teams often like to use the **`+all`** user-define plusarg to tell a compiled regression suite to run all tests. When I tried this with NC-Verilog, I discovered that NC-Verilog already has **`+all`** reserved for a tool-specific command (ouch!)

8.2 Constant Part-Select Indexing

Verilog does not permit variable part-select indexing in a for-loop, even if the part-select really is a fixed width. Most Verilog users avoid this problem by using a shift operator to get the desired bits moved to the correct position for assignment. Verilog-2001 adds **`+:`** and **`-:`** tokens that permit a fixed width to be specified on the right-hand side of the token. A couple of vendors have implemented this enhancement and one vendor has implemented it with bugs.

Table 6 - Verilog-2001 - Miscellaneous Enhancements

[SR#]		VCS	SS	MTI	NC	SIL
[33]	$value$plusargs	**BUG**	-	X	(+all)	-
[14]	Constant part-selects +: -:	X	X	X	-	-

8.3 Standard $sdf_annotate

Although not tested in the current compliance suite, most vendors have recognized the standard **`$sdf_annotate`** command for years. Now it is just official.

9.0 Non-Tested Enhancements

There are other Verilog-2001 enhancements that have not yet been tested by the Sunburst suite, these include: the line-file compiler directive (Sutherland #35), a series of enhancements added to increase the accuracy of Verilog ASIC models and timing (Sutherland 38-41), Extensions to VCD files, enhanced PLA system tasks and enhanced PLI support for Verilog-2001 (Sutherland 43-45).

Tests for these enhancements may be added at a later date, but these are enhancements that should be driven and verified by ASIC library modelers and Verilog tool vendors.

10.0 Conclusions

In the first revision of this paper, I found it disappointing to find that there was no Verilog-2001 subset that was reliably supported by all vendors. In revision 1.2, I found that ANSI-style module ports and both comma-separated and @* sensitivity lists are now supported by the simulation vendors I tested. If your company uses a certain subset of the tools shown in this paper, you should be able to identify Verilog-2001 features that are fairly safe to use now.

One thing that vendors have told me time and time again, is that they prioritize their development efforts based on their users feedback. After looking at these tables, perhaps you and your company can

do some serious pounding on your favorite vendor to get certain important features implemented by all of your chosen vendors.

Make your requests known. The more you pester your vendor the quicker the vendor will dedicate resources to the request. If your vendor says they are going to support the new Accellera SystemVerilog enhancements, ask them when they intend to support the Verilog-2001 enhancements!

11.0 Acknowledgements and Apologies

I would like to thank the many people from engineering, marketing and sales at Synopsys, Model Technology, Cadence, Simucad and Mentor for lending their simulation tools to make this evaluation possible.

12.0 About Cadence's Verilog-XL

For rev1.1 of this paper, I tested and reported compliance data related to Verilog-XL, version 4.2 beta. I was surprised to discover that Verilog-XL failed almost every single Verilog-2001 compliance test. Concerned that I would be reporting this situation to hundreds of engineers, I asked Cadence for a comment. Michael Munsey[7], NC-Verilog Product Marketing Manager for Cadence emailed that Cadence has,

> "no plans either now or in the future to enhance Verilog-XL *with Verilog-2001 extensions*. There are no plans to end of life it either ... we still actively support it. Verilog-XL is still used by a majority of *Cadence* customers for legacy designs and gate level regression runs. Our customers who require Verilog 2001 features are all NC-Verilog customers." (Italicized text added).

Since Cadence has no plans to add Verilog-2001 features to Verilog-XL, Verilog-XL was removed from the tables in this paper and will not be tested again.

Conclusion - do not use Verilog-XL for any new design work. Either use NC-Verilog or another Verilog-2001 compliant Verilog simulator.

Revision 1.2 (April 2003) - What Changed?

For the reasons stated in Section 12.0, Verilog-XL was removed from the tables.

I was chided by VCS marketing for not using the latest version of the VCS simulator in the original paper. Feeling bad, I went to the Synopsys Electronic Software Transfer site and discovered that the latest version was not listed (now I didn't feel so bad!) VCS 7.0 was released but had not been added to the Synopsys EST web site (that situation has been corrected); nevertheless, I did acquire VCS version 7.0 and tested it for this revision of the paper. VCS 7.0 did add support for a number of important Verilog-2001 enhancements and they are reflected in the paper.

I also was able to test Simucad's SILOS 2002.10 version, which has implemented some of the Verilog-2001 enhancements as shown in the tables.

13.0 References

[1] Clifford E. Cummings, "Verilog-2001 Behavioral and Synthesis Enhancements," *Delivered at HDLCON-2001 but missed publication in the Proceedings*, March 2001. Available at www.sunburst-design.com/papers

[2] Donald Thomas, and Philip Moorby, The Verilog Hardware Description Language, Fourth Edition, Kluwer Academic Publishers, 1998

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

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

[5] Stuart Sutherland, "Verilog 2001 - A Guide to the New Features of the Verilog Hardware Description Language," Kluwer Academic Publishers, 2002.

[6] www.chris.spear.net

[7] Michael Munsey - personal communication

Author & Contact Information

Cliff Cummings is President of Sunburst Design, Inc., a company that specializes in Verilog, Verilog synthesis and SystemVerilog training. Mr. Cummings is an independent consultant and trainer with 21 years of ASIC, FPGA, system design and verification experience and 11 years of Verilog, synthesis and methodology training experience.

Mr. Cummings has co-authored four Verilog books: the 1995 and 2001 IEEE Verilog Standards, the 2002 IEEE Verilog RTL Synthesis Standard and the 2002 Accellera SystemVerilog Standard. Mr. Cummings is the only Verilog trainer to co-develop all four of these standards.

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

E-mail Address: cliffc@sunburst-design.com

An updated version of this paper can be downloaded from the web site:
www.sunburst-design.com/papers

Correct Methods For Adding Delays To Verilog Behavioral Models

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

Abstract

Design engineers frequently build Verilog models with behavioral delays. Most hardware description languages permit a wide variety of delay coding styles but very few of the permitted coding styles actually model realistic hardware delays. Some of the most common delay modeling styles are very poor representations of real hardware. This paper examines commonly used delay modeling styles and indicates which styles behave like real hardware, and which do not.

1.0 Introduction

One of the most common behavioral Verilog coding styles used to model combinational logic is to place delays to the left of blocking procedural assignments inside of an always block. This coding style is flawed as it can either easily produce the wrong output value or can propagate inputs to an output in less time than permitted by the model specifications.

This paper details delay-modeling styles using continuous assignments with delays, and procedural assignments using blocking and nonblocking assignments with delays on either side of the assignment operator.

To help understand delay modeling, the next section also includes a short description on inertial and transport delays, and Verilog command line switches that are commonly used to simulate a model that is neither a fully inertial-delay model nor a fully transport-delay model.

2.0 Inertial and transport delay modeling

Inertial delay models only propagate signals to an output after the input signals have remained unchanged (been stable) for a time period equal to or greater than the propagation delay of the model. If the time between two input changes is shorter than a procedural assignment delay, a continuous assignment delay, or gate delay, a previously scheduled but unrealized output event is replaced with a newly scheduled output event.

Transport delay models propagate all signals to an output after any input signals change. Scheduled output value changes are queued for *transport delay* models.

Reject & Error delay models propagate all signals that are greater than the error setting, propagate unknown values for signals that fall between the reject & error settings, and do not propagate signals that fall below the reject setting.

For most Verilog simulators, reject and error settings are specified as a percentage of propagation delay in multiples of 10%.

Pure *inertial delay* example using reject/error switches.
Add the Verilog command line options:

```
+pulse_r/100 +pulse_e/100
```

reject all pulses less than 100% of propagation delay.

Pure *transport delay* example using reject/error switches.
Add the Verilog command line options:

```
+pulse_r/0 +pulse_e/0
```

pass all pulses greater than 0% of propagation delay.

Semi-realistic delay example using reject/error switches.
Add the Verilog command line options:

```
+pulse_r/30 +pulse_e/70
```

reject pulses less than 30%, propagate unknowns for pulses between 30-70% and pass all pulses greater than 70% of propagation delay.

3.0 Blocking assignment delay models

Adding delays to the left-hand-side (LHS) or right-hand-side (RHS) of blocking assignments (as shown in Figure 1) to model combinational logic is very common among new and even experienced Verilog users, but the practice is flawed.

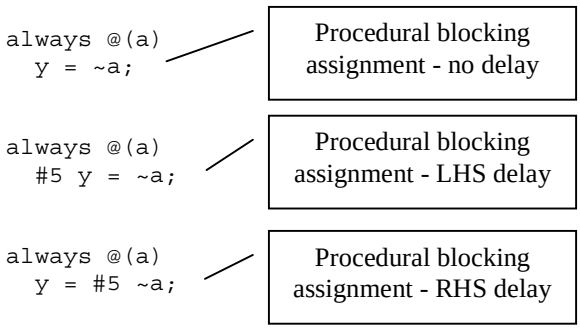


Figure 1 - Blocking Assignments with Delays

For the **adder_t1** example shown in Figure 2, the outputs should be updated 12ns after input changes. If the **a** input changes at time 15 as shown in Figure 3, then if the **a**, **b** and **ci** inputs all change during the next 9ns, the outputs will be updated with the latest values of **a**, **b** and **ci**. This modeling style has just permitted the **ci** input to propagate a value to the **sum** and **carry** outputs after only 3ns instead of the required 12ns propagation delay.

```
module adder_t1 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} = a + b + ci;
endmodule
```

Figure 2 - LHS Blocking Assignment

Adding delays to the left hand side (LHS) of any sequence of blocking assignments to model combinational logic is also flawed.

The **adder_t7a** example shown in Figure 4 places the delay on the first blocking assignment and no delay on the second assignment. This will have the same flawed behavior as the **adder_t1** example.

The **adder_t7b** example, also shown in Figure 4, places the delay on the second blocking assignment and no delay on the first. This model will sample the inputs on the first input change and assign the outputs to a temporary location until after completion of the specified blocking delay. Then the outputs will be written with the old temporary output values that are no longer valid. Other input changes within the 12ns delay period will not be evaluated, which means old erroneous values will remain on the outputs until more input changes occur.

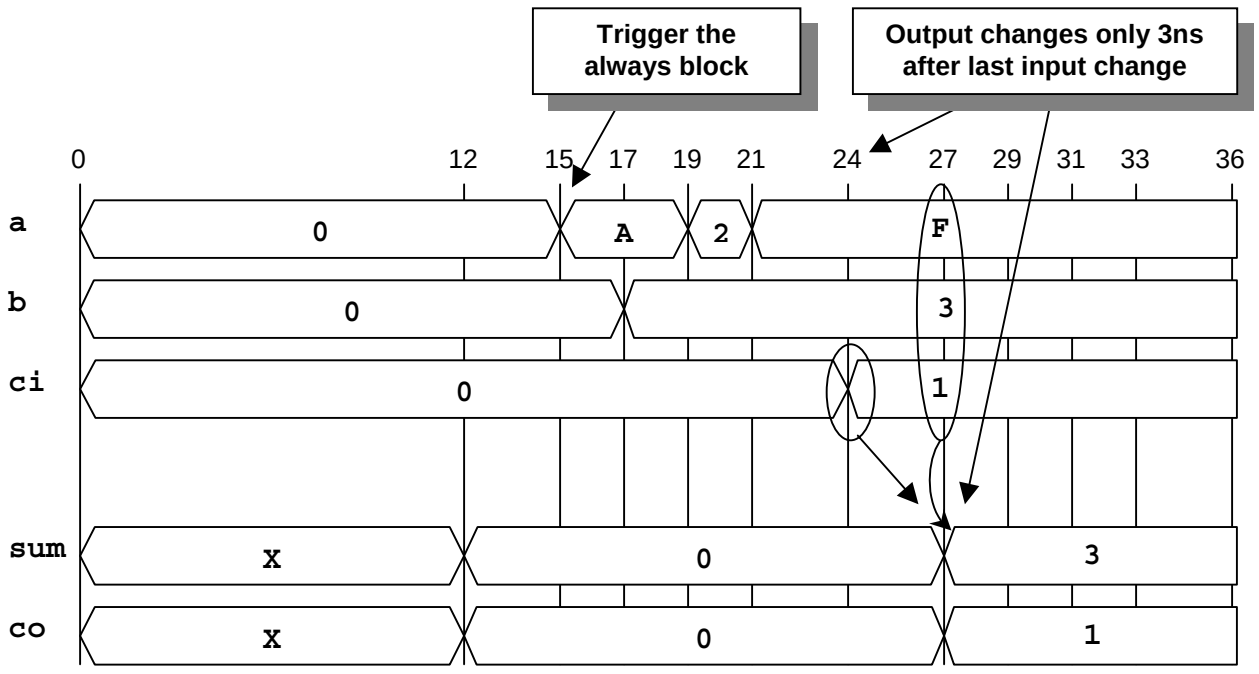


Figure 3 - Waveforms for adder_t1.v Example

These adders do not model any known hardware.

Modeling Guideline: do not place delays on the LHS of blocking assignments to model combinational logic. This is a bad coding style.

Testbench Guideline: placing delays on the LHS of blocking assignments in a testbench is reasonable since the delay is just being used to time-space sequential input stimulus events.

```
module adder_t7a (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci) begin
    #12 tmp       = a + b + ci;
        {co, sum} = tmp;
  end
endmodule


module adder_t7b (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci) begin
        tmp       = a + b + ci;
    #12 {co, sum} = tmp;
  end
endmodule
```

Figure 4 - Multiple LHS Blocking Assignments

3.1 RHS blocking delays

Adding delays to the right hand side (RHS) of blocking assignments to model combinational logic is also flawed.

```
module adder_t6 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    {co, sum} = #12 a + b + ci;
endmodule
```

Figure 5 - RHS Blocking Assignment

For the **`adder_t6`** example shown in Figure 5, the outputs should be updated 12ns after input changes. If the **`a`** input changes at time 15, the RHS input values will be sampled and the outputs will be updated with the sampled value, while all other **`a`**, **`b`** and **`ci`** input changes during the next 12ns will not be evaluated. This means old erroneous values will remain on the outputs until more input changes occur.

The same problem exists with multiple blocking assignments when delays are placed on the RHS of the assignment statements. The **`adder_t11a`** and **`adder_t11b`** examples shown in Figure 6 demonstrate the same flawed behavior as the **`adder_t6`** example .

None of the adder examples with delays on the RHS of blocking assignments behave like any known hardware.

Modeling Guideline: do not place delays on the RHS of blocking assignments to model combinational logic. This is a bad coding style.

Testbench Guideline: do not place delays on the RHS of blocking assignments in a testbench.

General Guideline: placing a delay on the RHS of any blocking assignment is both confusing and a poor coding style. This Verilog coding practice should be avoided.

```
module adder_t11a (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci) begin
    tmp       = #12 a + b + ci;
    {co, sum} =     tmp;
  end
endmodule


module adder_t11b (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci) begin
    tmp       =     a + b + ci;
    {co, sum} = #12 tmp;
  end
endmodule
```

Figure 6 - Multiple RHS Blocking Assignments

4.0 Nonblocking assignment delay models

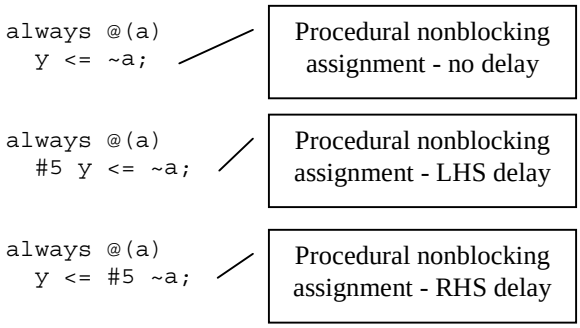


Figure 7 - Nonblocking Assignments with Delays

Adding delays to the left-hand-side (LHS) of nonblocking assignments (as shown in Figure 7) to model combinational logic is flawed.

The same problem exists in the **adder_t2** example shown in Figure 8 (nonblocking assignments) that existed in the **adder_t1** example shown in Figure 2 (blocking assignments). If the **a** input changes at time 15, then if the **a**, **b** and **ci** inputs all change during the next 9ns, the outputs will be updated with the latest values of **a**, **b** and **ci**. This modeling style permitted the **ci** input to propagate a value to the **sum** and **carry** outputs after only 3ns instead of the required 12ns propagation delay.

```
module adder_t2 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    #12 {co, sum} <=  a + b + ci;
endmodule
```

Figure 8 - LHS Nonblocking Assignment

It can similarly be shown that adding delays to the left hand side (LHS) of any sequence of nonblocking assignments to model combinational logic is also flawed.

Adders modeled with LHS nonblocking assignments do not model any known hardware.

Modeling Guideline: do not place delays on the LHS of nonblocking assignments to model combinational logic. This is a bad coding style.

Testbench Guideline: nonblocking assignments are less efficient to simulate than blocking assignments; therefore, in general, placing delays on the LHS of nonblocking assignments for either modeling or testbench generation is discouraged.

4.1 RHS nonblocking delays

Adding delays to the right hand side (RHS) of nonblocking assignments (as shown in Figure 9) will accurately model combinational logic with *transport delay*s.

In the **adder_t3** example shown in Figure 9, if the **a** input changes at time 15 as shown in Figure 10 (next page), then all inputs will be evaluated and new output values will be queued for assignment 12ns later. Immediately after the outputs have been queued (scheduled for future assignment) but not yet assigned, the always block will again be setup to trigger on the next input event. This means that all input events will queue new values to be placed on the outputs after a 12ns delay. This coding style models combinational logic with *transport delays*.

```
module adder_t3 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;

  always @(a or b or ci)
    {co, sum} <= #12 a + b + ci;
endmodule
```

Figure 9 - RHS Nonblocking Assignment

Recommended Application: Use this coding style to model behavioral delay-line logic.

Modeling Guideline: place delays on the RHS of nonblocking assignments only when trying to model transport output-propagation behavior. This coding style will accurately model delay lines and combinational logic with pure *transport delays*; however, this coding style generally causes slower simulations.

Testbench Guideline: This coding style is often used in testbenches when stimulus must be scheduled on future clock edges or after a set delay, while not blocking the assignment of subsequent stimulus events in the same procedural block.

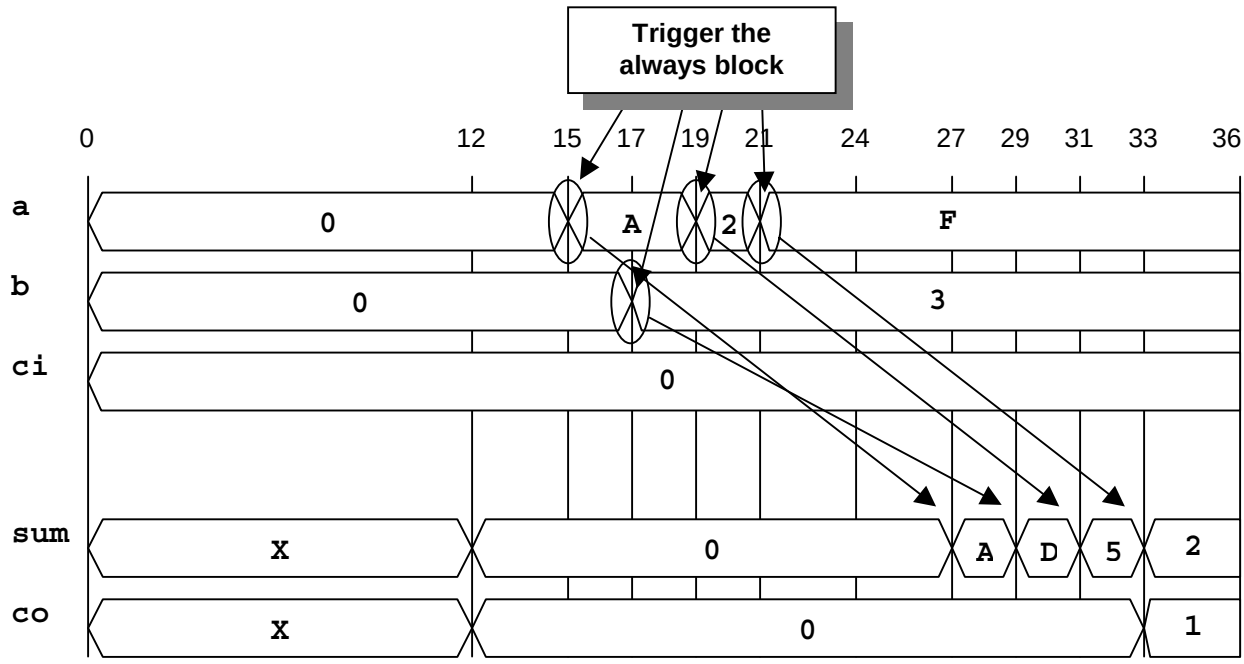


Figure 10 - Waveforms for adder_t3 Example

4.2 Multiple RHS nonblocking delays

Adding delays to the right hand side (RHS) of multiple sequential nonblocking assignments to model combinational logic is flawed, unless all of the RHS input identifiers are listed in the sensitivity list, including intermediate temporary values that are only assigned and used inside the always block, as shown in Figure 11.

For the **adder_t9c** and **adder_t9d** examples shown in Figure 11, the nonblocking assignments are executed in parallel and after **tmp** is updated, since **tmp** is in the sensitivity list, the always block will again be triggered, evaluate the RHS equations and update the LHS equations with the correct values (on the second pass through the always block).

Modeling Guideline: in general, do not place delays on the RHS of nonblocking assignments to model combinational logic. This coding style can be confusing and is not very simulation efficient. It is a common and sometimes useful practice to place delays on the RHS of nonblocking assignments to model clock-to-output behavior on sequential logic.

Testbench Guideline: there are some multi-clock design verification suites that benefit from using multiple nonblocking assignments with RHS delays; however, this coding style can be confusing, therefore placing delays on the RHS of nonblocking assignments in testbenches is not generally recommended.

```
module adder_t9c (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci or tmp) begin
    tmp       <= #12 a + b + ci;
    {co, sum} <=     tmp;
  end
endmodule


module adder_t9d (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg          co;
  reg    [3:0] sum;
  reg    [4:0] tmp;

  always @(a or b or ci or tmp) begin
    tmp       <=     a + b + ci;
    {co, sum} <= #12 tmp;
  end
endmodule
```

Figure 11 - Multiple Nonblocking Assignments with Delays

5.0 Continuous assignment delay models

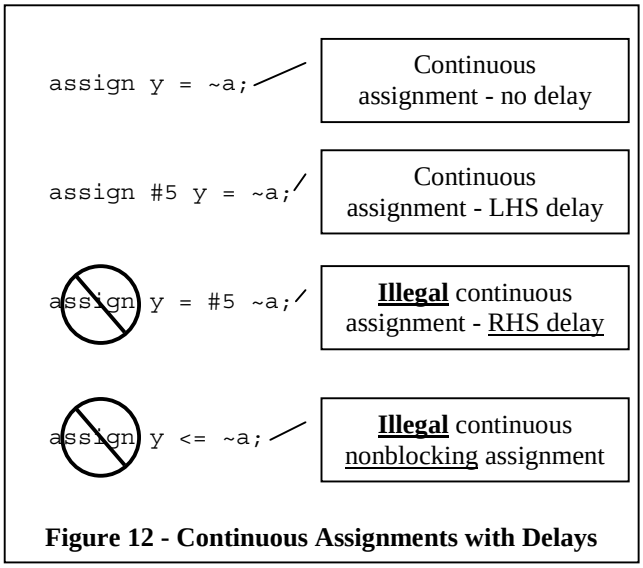


Figure 12 - Continuous Assignments with Delays

Adding delays to continuous assignments (as shown in Figure 12) accurately models combinational logic with *inertial delays* and is a recommended coding style.

For the **adder_t4** example shown in Figure 13, the outputs do not change until 12ns after the last input change (12ns after all inputs have been stable). Any sequence of input changes that occur less than 12ns apart will cause any future scheduled *output-event* (output value with corresponding assignment time) to be replaced with a new output-event.

```
module adder_t4 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;

  assign #12 {co, sum} = a + b + ci;
endmodule
```

Figure 13 - Continuous Assignment with Delay

Figure 14 shows the output waveforms for a simulation run on the **adder_t4** code shown in Figure 13. The first **a**-input change occurs at time 15, which causes an output event to be scheduled for time 27, but a change on the **b**-input and two more changes on the **a**-input at times 17, 19 and 21 respectively, cause three new output events to be scheduled. Only the last output event actually completes and the outputs are assigned at time 33.

Continuous assignments do not "queue up" output assignments, they only keep track of the next output value and when it will occur; therefore, continuous assignments model combinational logic with *inertial delays*.

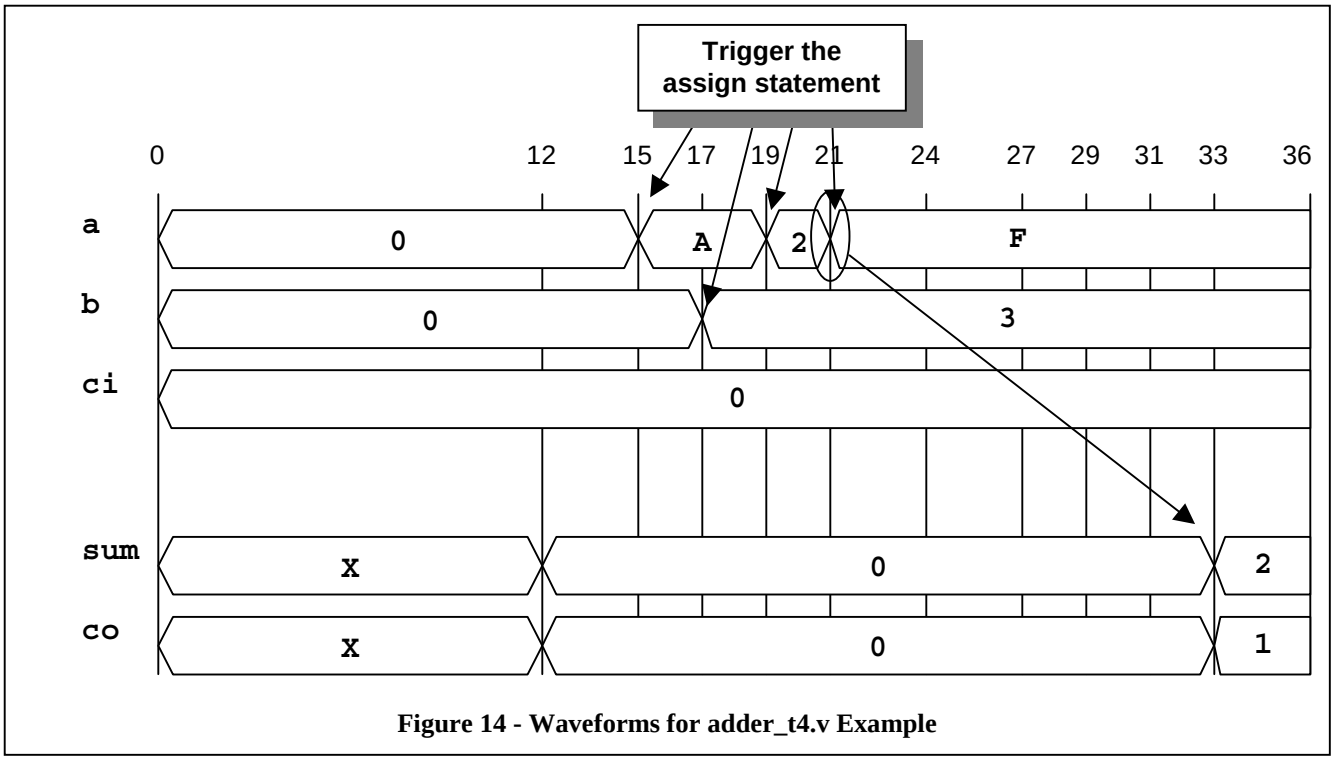


Figure 14 - Waveforms for adder_t4.v Example

5.1 Multiple continuous assignments

It can similarly be shown that modeling logic functionality by adding delays to continuous assignments, whose outputs are used to drive the inputs of other continuous assignments with delays, as shown in Figure 15, also accurately models combinational logic with *inertial delays*.

```
module adder_t10a (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  wire   [4:0] tmp;

  assign     tmp       = a + b + ci;
  assign #12 {co, sum} = tmp;
endmodule

module adder_t10b (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  wire   [4:0] tmp;

  assign #12 tmp       = a + b + ci;
  assign     {co, sum} = tmp;
endmodule
```

Figure 15 - Multiple Continuous Assignments

5.2 Mixed no-delay always blocks and continuous assignments

Modeling logic functionality in an always block with no delays, then passing the always block intermediate values to a continuous assignment with delays as, shown in Figure 16, will accurately model combinational logic with *inertial delays*.

```
module adder_t5 (co, sum, a, b, ci);
  output       co;
  output [3:0] sum;
  input  [3:0] a, b;
  input        ci;
  reg    [4:0] tmp;

  always @(a or b or ci) begin
    tmp = a + b + ci;
  end

  assign #12 {co, sum} = tmp;
endmodule
```

Figure 16 - No-Delay Always Block & Continuous Assignment

For the **`adder_t5`** example shown in Figure 16, the **`tmp`** variable is updated after any and all input events. The continuous assignment outputs do not change until 12ns after the last change on the **`tmp`** variable. Any sequence of always block input changes will cause **`tmp`** to change, which will cause a new output event on to be scheduled on the continuous assignment outputs. The continuous assignment outputs will not be updated until **`tmp`** remains unchanged for 12ns. This coding style models combinational logic with *inertial delays*.

Modeling Guideline: Use continuous assignments with delays to model simple combinational logic. This coding style will accurately model combinational logic with *inertial delays*.

Modeling Guideline: Use always blocks with no delays to model complex combinational logic that are more easily rendered using Verilog behavioral constructs such as "case-casez-casex", "if-else", etc. The outputs from the no-delay always blocks can be driven into continuous assignments to apply behavioral delays to the models. This coding style will accurately model complex combinational logic with *inertial delays*.

Testbench Guideline: Continuous assignments can be used anywhere in a testbench to drive stimulus values onto input ports and bi-directional ports of instantiated models.

6.0 Conclusions

Any delay added to statements inside of an always block does not accurately model the behavior of real hardware and should not be done. The one exception is to carefully add delays to the right hand side of nonblocking assignments, which will accurately model *transport delays*, generally at the cost of simulator performance.

Adding delays to any sequence of continuous assignments, or modeling complex logic with no delays inside of an always block and driving the always block outputs through continuous assignments with delays, both accurately model *inertial delays* and are recommended coding styles for modeling combinational logic.

Author & Contact Information

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

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, chaired the VSG Behavioral Task Force, which was charged with proposing enhancements to the Verilog language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.
Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

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

(Data accurate as of March 7th, 2001)

A Proposal To Remove Those Ugly Register Data Types From Verilog

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

Abstract

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

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

The proposal

Proposal:

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

Reasons:

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

Introduction

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

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

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

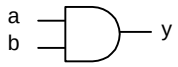
Register & net declarations - simple rule

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

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

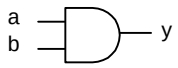
Let's look at two simple Verilog examples to help understand the declarations of register and net data types.

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

  assign y = a & b;
endmodule
```



Example 1 - Valid continuous assignment with no wire declaration

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

  assign y = a & b;
endmodule
```



Example 2 - Valid continuous assignment with wire declaration

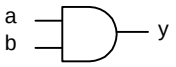
In Example 1, a 2-input and gate is modeled using a continuous assignment statement. The y-output does not have to be declared because it is a 1-bit wire. Example 2

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

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

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

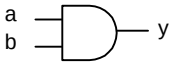
```
module and2c (y, a, b);
  output y;
  input  a, b;
  wire   y;

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



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

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

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

  assign y = a & b;
endmodule
```



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

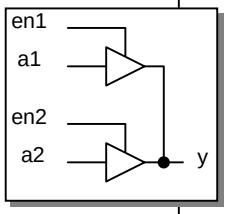
Simple rule: In Verilog, anything on the left hand side (LHS) of a procedural assignment must be declared as a register data type. Everything else in Verilog is a net data type. No exceptions!

Why differentiate nets & registers?

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

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

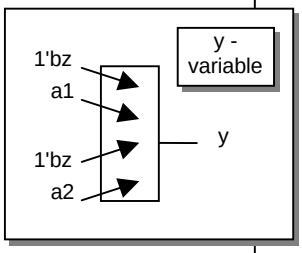
```
module drivers1 (y, a1, en1, a2, en2);
  output y;
  input  a1, en1, a2, en2;

  assign y = en1 ? a1 : 1'bz;

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



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

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

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

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

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



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

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

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

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

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

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

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

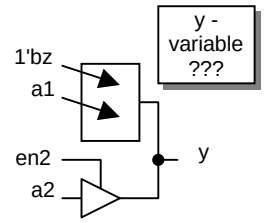


Figure 1 - Illegal variable/driver combination

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

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

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

  assign y = y_tmp;

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

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

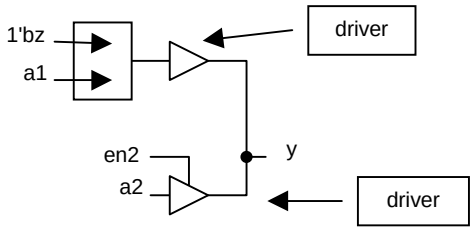


Figure 2 - Variable assigned to a driver

Conditional compilation

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

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

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

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

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

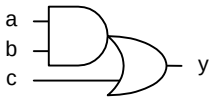
Example 9 - Conditional compilation with mid-code reg declaration

Verilog-2000 port enhancements

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

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

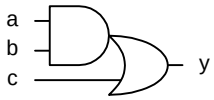
```
module and2ora (y, a, b, c);
  output y;
  input  a, b, c;
  reg    y;
  reg    tmp;

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

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



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

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

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

  assign tmp = a & b;

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



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

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

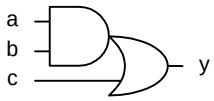
For the purposes of this paper, the following port declaration style definitions are used:

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

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

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

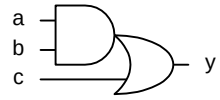
```
module and2orc (y, a, b, c);
  output reg  y;
  input  wire a, b, c;
  reg    tmp;

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

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



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

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

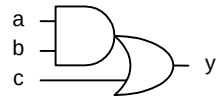
```
module and2ord (y, a, b, c);
  output reg  y;
  input       a, b, c;
  reg    tmp;

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

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



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

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

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

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

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



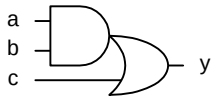
Example 14 - Verilog-2000 and-or gate with single-declared port (style #2) and "`reg tmp`"

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

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

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

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

  assign tmp = a & b;

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

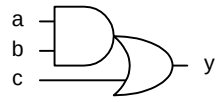


Example 15 - Verilog-2000 and-or gate with single-declared port (style #2) and "`wire tmp`"

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

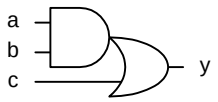
If separate register and net data type requirements are eliminated, the same enhanced port declarations as shown in Example 16 and Example 17 will be both abbreviated and legal. Note that in both examples, the declarations are identical and no net or register declarations are required.

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

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

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



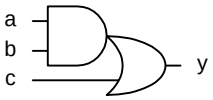
Example 16 - Verilog-2005(?) and-or gate with single-declared port (always y-output)

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

  assign tmp = a & b;

  assign y = tmp | c;
endmodule
```



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

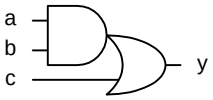
```
module and2ori (
  output y;
  input  a, b, c;
  );

  assign tmp = a & b;

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



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

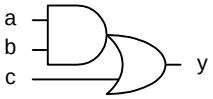
Of course, the and-or model can also be simplified in Verilog-2005(?) by combining the separate assignments seen in earlier examples into either a single continuous assignment as shown in Example 19 or into a single always block as shown in Example 20. Example 20 also shows the combinational sensitivity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

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

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



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

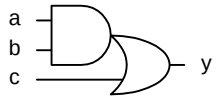
```
module and2ork (
  output y;
  input  a, b, c;
  );

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



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

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

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

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



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

Pros & cons of declaring wires

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

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

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

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

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

Example 22 - Undetected bad 1-bit wire declaration

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

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

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

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

Example 23 - Testbench for inverter modules

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

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

  always @(a) tmp = ~a;

  assign      y   = tmp;
endmodule
```

Example 24 - Undetected bad 1-bit reg declaration

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

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

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

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

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

Example 25 - Good 8-bit wire reg declaration

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

  always @(a) tmp = ~a;

  assign      y   = tmp;
endmodule
```

Example 26 - Good 8-bit wire reg declaration

Net and register differences

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

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

Figure 3 - Net and register differences

Handling existing register types

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

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

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

Figure 4 - Net implementations of existing register types

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

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

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

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

A new type of syntax check

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

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

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

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

Further assignment restrictions might include:

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

Conclusions

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

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

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

References

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

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

Author & Contact Information

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

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

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

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

(Data accurate as of March 7th, 2001)

Revised - April 2002
Important correction to ANSI style parameter lists added to this revision

Verilog-2001 Behavioral and Synthesis Enhancements

Clifford E. Cummings
cliffc@sunburst-design.com / www.sunburst-design.com

Sunburst Design, Inc.
14314 SW Allen Blvd.
PMB 501
Beaverton, OR 97005

ABSTRACT

The Verilog-2001 Standard includes a number of enhancements that are targeted at simplifying designs, improving designs and reducing design errors.

This paper details important enhancements that were added to the Verilog-2001 Standard that are intended to simplify behavioral modeling and to improve synthesis accuracy and efficiency. Information is provided to explain the reasons behind the Verilog-2001 Standard enhancement implementations..

1.0 Introduction

For the past five years, experienced engineers and representatives of EDA vendors have wrestled to define enhancements to the Verilog language that will offer increased design productivity, enhanced synthesis capability and improved verification efficiency.

The guiding principles behind proposed enhancements included:
1. do not break existing designs,
2. do not impact simulator performance,
3. make the language more powerful and easier to use.

This paper details many of the behavioral and synthesis enhancements that were added to the Verilog-2001 Standard[1], including some of the rational that went into defining the added enhancements. This paper will also discuss a few errata and corrections to the yet unpublished 2001 Verilog Standard.

Immediately after the header for each enhancement, I make predictions on when you will likely see each enhancement actually implemented by EDA vendors.

1.1 Glossary of Terms

The Verilog Standards Group used a set of terms and abbreviations to help concisely describe current and proposed Verilog functionality. Many of those terms are used in this paper and are therefore defined below:

- ASIC - Application Specific Integrated Circuit
- EDA - Electronic Design Automation.
- HDLCON - International HDL Conference.
- IP - Intellectual Property (not internet protocol).
- IVC - International Verilog Conference - precursor to HDLCON when the Spring VIUF and IVC conferences merged.
- LHS - Left Hand Side of an assignment.
- LSB - Least Significant Bit.
- MSB - Most Significant Bit.
- PLI - the standard Verilog Programming Language Interface
- RHS - Right Hand Side of an assignment.
- RTL - Register Transfer Level or the synthesizable subset of the Verilog language.
- VHDL - VHSIC Hardware Description Language.
- VHSIC - Very High Speed Integrated Circuits program, funded by the Department of Defense in the late 1970's and early 1980's [2].
- VIUF - VHDL International Users Forum - the Spring VIUF conference was a precursor to HDLCON when the Spring VIUF and IVC conferences merged.
- VSG - Verilog Standards Group.

2.0 What Broke in Verilog-2001?

While proposing enhancements to the Verilog language, the prime directive of the Verilog Standards Group was to not break any existing code. There are only two Verilog-2001 behavioral enhancement proposals that potentially break existing designs. These two enhancements are described below.

2.1 31 open files

Verilog-1995[3] permitted users to open up to 31 files for writing. The file handle for Verilog-1995-style files is called an MCD (Multi-Channel Descriptor) where each open file is represented by one bit set in an integer. Only the 31 MSBs of the integer could be set for open files since bit 0 represented the standard output (STDOUT) terminal. The integer identifier-name was the file handle used in the Verilog code.

MCDs could be bit-wise or'ed together into another integer with multiple bits set to represent multiple open files. Using an MCD with multiple valid bits set, a designer can access multiple open files with a single command.

In recent years, engineers have found reasons to access more than 31 files while doing design verification. The 31 open-file limit was too restrictive.

At the same time, engineers were demanding better file I/O capabilities, so both problems were addressed in a single enhancement. The file I/O enhancement requires the use of the integer-MSB to indicate that the new file I/O enhancement is in use. When the integer-MSB is a "0", the file in use is a Verilog-1995-style file with multi-channel descriptor capability. When the integer-MSB is a "1", the file in use is a Verilog-2001-style file where it is now possible to have 2**31 open files at a time, each with a unique binary number file-handle representation (multi-channel descriptors are not possible with the new file I/O-style files.

Any existing design that currently uses exactly 31 open files will break using Verilog-2001. The fix is to use the new file I/O capability for at least one of the current 31 open files. It was necessary to steal the integer MSB to enhance the file I/O capabilities of Verilog.

2.2 `bz assignment

Verilog-1995 and earlier has a peculiar, not widely known "feature" (documented-bug!) that permits assignments like the one shown below in Example 1 to assign up to 32 bits of "Z" with all remaining MSBs being set to "0".

```
assign databus = en ? dout : 'bz;
```

Example 1 - Simple continuous assignment using 'bz to do z-expansion

If the databus in Example 1 is 32 bits wide or smaller, this coding style works fine. If the databus is larger than 32 bits wide, the lower bits are set to "Z" while the upper bits are all set to "0". All synthesis tools synthesize this code to 32 tri-state drivers and all upper bits are replaced with and-gates so that if the en input is low, the and-gate outputs also drive "0"s.

The correct Verilog-1995 parameterized model for a tri-state driver of any size is shown Example 2:

```
module tribuf (y, a, en);
  parameter SIZE = 64;
  output [SIZE-1:0] y;
  input  [SIZE-1:0] a;
  input             en;

  assign y = en ? a : {SIZE{1'bz}};
endmodule
```

Example 2 - Synthesizble and parameterizable Verilog-1995 three-state buffer model

In Verilog-2001, making assignments of 'bz or 'bx will respectively z-extend or x-extend the full width of the LHS variable.

The VSG determined that any engineer that intentionally made 'bz assignments, intending to drive 32 bits of "Z" and all remaining MSBs to "0" deserved to have their code broken! An engineer could easily make an assignment of 32'bz wherever the existing behavior is desired and the assignment will either truncate unused Z-bits or add leading zeros to the MSB positions to fill a larger LHS value.

2.3 Minimal risk

The VSG decided that there would be minimal impact from the file I/O enhancement that could not be easily solved using the new Verilog-2001 file I/O enhancement, and the 'bz assignment enhancement is not likely to appear in the code of any reasonably proficient Verilog designer, plus there is an easy work-around for the 'bz functionality if the existing silly behavior is actually desired!

3.0 LRM Errors

Unfortunately, adding new functionality to the Verilog language also required the addition of new and untested descriptions to the IEEE Verilog Standard documentation. Until the enhanced functionality is implemented, the added descriptions are unproven and might be short on intended enhancement functionality detail. What corner cases are not accurately described? The VSG could not compile the examples so there might be syntax errors in the newer examples.

One example of an error that went unnoticed in the new IEEE Verilog-2001 Standard is the Verilog code for a function that calculates the "ceiling of the log-base 2" of a number. This

example, given in section 10.3.5, makes use of constant functions. The clogb2 function described in the example from the IEEE Verilog Standard, duplicated below, has a few notable errors:

```
//define the clogb2 function
function integer clogb2;
  input depth;
  integer i,result;
  begin
    for (i = 0; 2 ** i < depth; i = i + 1)
      result = i + 1;
    clogb2 = result;
  end
endfunction
```

Example 3 - Verilog-2001 Standard constant function example from section 10 with errors

Errors in this model include:

(1) the input "depth" to the function in this example is only one bit wide and should have included a multi-bit declaration.

(2) the result is not initialized. If the depth is set to "1", the for-loop will not execute and the function will return an unknown value.

A simple and working replacement for this module that even works with Verilog-1995 is shown in Example 4:

```
function integer clogb2;
  input [31:0] value;
  for (clogb2=0; value>0; clogb2=clogb2+1)
    value = value>>1;
endfunction
```

Example 4 - Working function to calculate the ceiling of the log-base-2 of a number

4.0 Top Five Enhancements

At a "Birds Of a Feather" session at the International Verilog Conference (IVC) in 1996, Independent Consultant Kurt Baty moderated an after-hours panel to solicit enhancement ideas for future enhancements to the Verilog standard.

Panelists and audience members submitted enhancement ideas and the entire group voted for the top-five enhancements that they would like to see added to the Verilog language. These top-five enhancements gave focus to the VSG to enhance the Verilog language.

Although numerous enhancements were considered and many enhancements added to the Verilog 2001 Standard, the top-five received the most attention from the standards group and all five were added in one form or another. The top-five enhancements agreed to by the audience and panel were:

#1 - Verilog generate statement
#2 - Multi-dimensional arrays
#3 - Better Verilog file I/O
#4 - Re-entrant tasks
#5 - Better configuration control

Many enhancements to the Verilog language were inspired by similar or equivalent capabilities that already existed in VHDL. Many Verilog designers have at one time or another done VHDL design. Any VHDL capability that we personally liked, we tried adding to Verilog. Anything that we did not like about VHDL we chose not to add to Verilog.

4.1 Multi-Dimensional Arrays

Expected to be synthesizable? Yes. This capability is already synthesizable in VHDL and is needed for Verilog IP development.
When? Soon!

Before describing the generate statement, it is logical to describe the multi-dimensional array enhancement, that is essentially required to enable the power of generate statements.

Multidimensional arrays are intended to be synthesizable and most vendors will likely have this capability implemented around the time that the Verilog 2001 LRM becomes an official IEEE Standard.

In Verilog-1995, it was possible to declare register variable arrays with two dimensions. Two noteworthy restrictions were that net types could not be declared as arrays and only one full array-word could be referenced, not the individual bits within the word.

In Verilog-2001, net and register-variable data types can be used to declare arrays and the arrays can be multidimensional. Access will also be possible to either full array words or to bit or part selects of a single word.

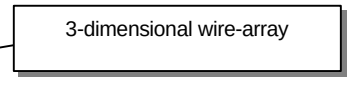
In Verilog-2001, it shall still be illegal to reference a group of array elements greater than a single word; hence, one still cannot initialize a partial or entire array by referencing the array by the array name or by a subset of the index ranges. Two-dimensional array elements must be accessed by one or two index variables, Three dimensional array elements must be accessed by two or three index variables, etc.

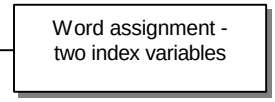
In Example 5, a structural model of a dual-pipeline model with one 2-bit data input is fanned out into two 2-bit by 3-deep pipeline stages and two 2-bit data outputs are driven by the two respective pipeline outputs. The flip-flops in the model have been wired together using a 3-dimensional net array called data. The data-word-width is listed before the identifier data, and the other two dimensions are placed after the identifier data.

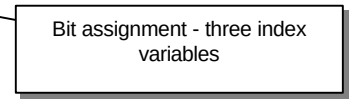
The connections between flip-flops are made using all three dimensions to indicate which individual nets are attached to the flip-flop data input and output, while connections to the ports

are done using only two dimensions to tie the 2-bit buses to the 2-bit data input and output ports of the model.

```
module dualpipe_v2k (dout1, dout0, din, en, clk, rst_n);
  output [1:0] dout1, dout0;
  input  [1:0] din, en;
  input        clk, rst_n;

  wire   [1:0] data [1:0] [3:0];

  assign data[1][0] = din;
  assign data[0][0] = din;

  dff u000 (.q(data[0][1][0]), .d(data[0][0][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u010 (.q(data[0][2][0]), .d(data[0][1][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u020 (.q(data[0][3][0]), .d(data[0][2][0]), .clk(clk), .en(en[0]), .rst_n(rst_n));

  dff u001 (.q(data[0][1][1]), .d(data[0][0][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u011 (.q(data[0][2][1]), .d(data[0][1][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));
  dff u021 (.q(data[0][3][1]), .d(data[0][2][1]), .clk(clk), .en(en[0]), .rst_n(rst_n));

  dff u100 (.q(data[1][1][0]), .d(data[1][0][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u110 (.q(data[1][2][0]), .d(data[1][1][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u120 (.q(data[1][3][0]), .d(data[1][2][0]), .clk(clk), .en(en[1]), .rst_n(rst_n));

  dff u101 (.q(data[1][1][1]), .d(data[1][0][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u111 (.q(data[1][2][1]), .d(data[1][1][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));
  dff u121 (.q(data[1][3][1]), .d(data[1][2][1]), .clk(clk), .en(en[1]), .rst_n(rst_n));

  assign dout1 = data[1][3];
  assign dout0 = data[0][3];
endmodule
```







Example 5 - Verilog-2001 structural dual-pipeline model using multidimensional wire arrays for connections

4.2 The Verilog Generate Statement
Expected to be synthesizable? Yes
When? Soon.

Inspired by the VHDL generate statement, the Verilog generate statement extends generate-statement capabilities beyond those of the VHDL-1993 generate statement.

In VHDL there is a for-generate (for-loop generate) and an if-generate statement. In Verilog-2001 there will be a for-loop generate statement, an if-else generate statement and a case generate statement.

4.3 The genvar index variable

After much debate, the VSG decided to implement a new index variable data type that can only be used with generate statements. The keyword for the generate-index variable is "genvar." This variable type is only used during the evaluation of generated instantiations and shall not be referenced by other statements during simulation. The VSG felt it was safest to define a new variable type with restrictive usage requirements as opposed to imposing rules on integers when used in the context of a generate statement.

Per the IEEE Verilog-2001 Draft Standard, a Verilog genvar must adhere to the following restrictions:

- Genvars shall be declared within the module where the genvars are used.
- Genvars can be declared either inside or outside of a generate scope.
- Genvars are positive integers that are local to, and shall only be used within a generate loop that uses them as index variables.
- Genvars are only defined during the evaluation of the generate blocks.
- Genvars do not exist during simulation of a Verilog design.
- Genvar values shall only be defined by generate loops.
- Two generate loops using the same genvar as an index variable shall not be nested.
- The value of a genvar can be referenced in any context where the value of a parameter could be referenced.

The Verilog generate for-loop, like the Verilog procedural for-loop, does not require a contiguous loop-range and can therefore be used to generate sparse matrices of instances that might prove useful to DSP related designs.

The Verilog if-else generate statement can be used to conditionally instantiate modules, procedural blocks, continuous assignments or primitives.

The Verilog case generate statement was added to enhance the development of IP. Perhaps a model could be written for a multiplier IP that chooses an implementation based on the width of the multiplier operands. Small multipliers might be implemented best one or two different ways but large multipliers might be implemented better another way. Perhaps the multiplier model could chose a different implementation based on power_usage parameters passed to the model.

A FIFO model might be created that infers a different implementations based on whether the model uses synchronous or asynchronous clocks.

4.4 Enhanced File I/O

Expected to be synthesizable? No
When? Soon.

Verilog has always had reasonable file-writing capabilities but it only has very limited built-in file-reading capabilities.

Standard Verilog-1995 file reading capabilities were limited to reading binary or hex data from a file into a pre-declared Verilog array and then extracting the data from the array using Verilog commands to make assignments elsewhere in the design or testbench.

Verilog-1995 file I/O can be enhanced through the PLI and the most popular package used to enhance Verilog file I/O is the package maintained by Chris Spear on his web site[4]. Any Verilog

simulator with built-in standard PLI can be compiled to take advantage of most of the Verilog-2001 file I/O enhancements today.

Chris' file I/O PLI code was the starting point for Verilog-2001 file I/O enhancements, and since Chris has already done most of the work of enhancing file I/O, it is likely that most Verilog vendors will leverage off of Chris' work to implement the new file I/O enhancements.

4.5 Re-entrant Tasks and Functions

Expected to be synthesizable? Maybe?
When? Probably not soon.

Verilog functions are synthesizable today and Verilog tasks are synthesizable as long as there are no timing controls in the body of the task, such as @(posedge clk). The #delay construct is ignored by synthesis tools.

This enhancement might be one of the last enhancements to be implemented by most Verilog vendors. Most existing Verilog vendors have complained that this enhancement is a departure from the all-static variables that currently are implemented in the Verilog language. Automatic tasks and functions will require that vendors push the current values of task variables onto a stack and pop them off when the a recursively executing task invocation completes. Vendors are wrestling with how they intend to implement this functionality.

This enhancement is especially important to verification engineers who use tasks with timing controls to apply stimulus to a design. Unknown to many Verilog users, Verilog-1995 tasks use static variables, which means that if a verification task is called a second time before the first task call is still running, they will use the same static variables, most likely causing problems in the testbench. The current work-around is to place the task into a separate verification module and instantiate the module multiple times in the testbench, each with a unique instance name, so that the task can be called multiple times using hierarchical references to the instantiated tasks.

By adding the keyword "automatic" after the keyword "task," Verilog compilers will treat the variables inside of the task as unique stacked variables.

What about synthesis? Tektronix, Inc. of Beaverton Oregon has had an in-house synthesis tool that was first used to design ASICs starting in the late 1980's, and that tool has had the capability to synthesize recursive blocks of code also since the late 1980s. The recursive capabilities made certain DSP blocks very easy to code. Some creative synthesis vendor might find some very useful abilities by permitting recursive RTL coding; however, it is not likely that recursive tasks will be synthesizable in the near future.

4.6 Configurations

Expected to be synthesizable? Yes, Synthesis tools should be capable of reading configuration files to extract the files need to be included into a synthesized design.
When? This could be implemented soon.

Configuration files will make it possible to create a separate file that can map the instances of a source file to specific files as long as the files can be accessed with a UNIX-like path name. This enhancement should remove the need to employ `uselib directives in the source model to change the source files that are used to simulate specific instances within a design.

The `uselib directive has never been standardized because it requires a designer to modify the source models to add directives to call specific files to be compiled for specific instances. Modifying the source files to satisfy the file mapping requirements of a simulation run is a bad idea and the VSG hopes that usage of the `uselib directives will eventually cease. The configuration file also offers an elegant replacement for the common command line switches: -y, -v and +libext+.v, etc. These non-standard command line switches should also slowly be replaced with the more powerful Verilog-2001configuration files.

5.0 More Verilog Enhancements

In addition to the top-five enhancement requests, the VSG considered and added other powerful and useful enhancements to the Verilog language. Many of these enhancements are described below.

5.1 ANSI-C style port declarations

Expected to be synthesizable? Yes.
When? Almost immediately.

Verilog-1995 requires all module header ports to be declared two or three times, depending on the data type used for the port. Consider the simple Verilog-1995 compliant example of a simple flip-flop with asynchronous low-true reset, as shown in Example 6.

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

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 6 - Verilog-1995 D-flip-flop model with verbose port declarations

The Verilog-1995 model requires that the "q" output be declared three times, once in the module header port list, once in an output port declaration and once in a reg data-type declaration. The Verilog-2001 Standard combines the header port list declaration, port direction declaration and data-type declaration into a single declaration as shown in Example 7, patterned after ANSI-C style ports. Declaring all 1-bit inputs as wires is still optional.

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

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 7 - Verilog 2001 D-flip-flop model with new-style port declarations

This enhancement is a more compact way of making port declarations and should be easy to implement for simulation and synthesis soon.

5.2 Parameter passing by name (explicit & implicit)

Expected to be synthesizable? Yes.
When? Almost immediately.

Verilog-1995 standardized two ways to change parameters for instantiated modules, (1) parameter redefinition and (2) defparam statements.

(1) Parameter redefinition is accomplished by instantiating a module and adding #(new_value1 , new_value2 , ...) immediately after the module name.

Advantage: this technique insures that all parameters are passed to a module at the same time that the module is referenced.

Disadvantage: all parameters must be explicitly listed, in the correct order, up to and including the parameter(s) that are changed. For example, if a module contains 10 parameter definitions, and if the module is to be instantiated requires that the seventh parameter be changed, the instantiation must include seven parameters within the parentheses, listed in the correct order and including the first six values even though they did not change for this instantiation. It is not permitted to simply list six commas followed by the new seventh parameter value.

(2) Using defparam redefinition is accomplished by instantiating a module and including a separate defparam statement to change the instance_name.parameter_name value to its new value.

Advantage: this technique gives a simple and direct correspondence between the instance-name, parameter-name pair and the new value.

Disadvantage: defparam statements can appear anywhere in the Verilog source code and can change any parameter on any module. Translation - when compiling a Verilog design, none of the parameters in any module are fixed until the last Verilog source file is read, because the last file might hierarchically change every single parameter in the design! A "grand-child" module might change all of the parameters of the "grand-parent" module, which might pass new parameter values to the "parent/child" module. It gets ugly and probably slows the compilation of a Verilog design.

Verilog-2001 adds a superior way of passing parameters to instantiated modules, using named parameter passing, using the same technique as named port instantiation.

Advantage #1: Only the parameters that change need to be referenced in named port instantiations. The same advantage that exists when using defparam statements.

Advantage #2: All parameter information is available when the module instantiation is parsed and parameters are passed down the hierarchy; they do not cause side-effects up the hierarchy.

This is the best solution for IP development and usage.

The current defparam statement will not be fully usable in some Verilog-2001 enhancements and the VSG hopes that the addition of named parameter redefinition will eventually cause defparam statement usage to die.

Vendors might want to flag defparam statements as Verilog-2001 compiler errors with the following message:

```
   "The Verilog compiler found a defparam statement in the source code at (file-
line#). To use defparam statements in the Verilog source code, you must include the
switch +Iamstupid on the command line which will degrade compiler performance.
Defparam statements can be replaced with named parameter redefinition as define by
the Verilog-2001 standard"
```

5.3 Signed Arithmetic

Expected to be synthesizable? Could be(?)
When? Synthesis vendor dependent.

The signed arithmetic enhancement removes a frequent complaint about Verilog, that the design has to explicitly code signed arithmetic functionality into the model.

Any vendor that already handles synthesis of signed arithmetic operations should be able to take advantage of this enhancement to facilitate signed arithmetic design tasks.

5.4 `ifndef & `elsif

Expected to be synthesizable? Yes.
When? This could be implemented soon.

The `ifdef / `else / `endif conditionally-compiled-code compiler directives have been a part of the Verilog language since before the Verilog-1995 Standard. Two additions have been added to help generate conditionally compile code: `ifndef and `elsif.

The `ifdef set of compiler directives have been synthesizable by most synthesis tools for a long time and they became synthesizable by Synopsys tools starting with Synopsys version 1998.02 (full usage within Synopsys tools requires that the switch hdlin_enable_vpp be set to true).

The `ifndef switch adds a small simplification to Verilog code where the intent is to compile a block of code only when a specific text macro has not been defined.

```
`ifdef SYNTHESIS
`else
  initial $display("Running RTL Model");
`endif
```

Example 8 - Verilog-1995 coding style to replicate the Verilog-2001 `ifndef capability

```
`ifndef SYNTHESIS
  initial $display("Running RTL Model");
`endif
```

Example 9 - Using the new Verilog-2001 `ifndef compiler directive

Since the `ifndef and `elseif statements are used to simply determine when code should be compiled, these compiler directives could easily be implemented in both simulation and synthesis without much effort.

5.5 Exponential Operator

Expected to be synthesizable? Yes, if the operands are constants.
When? This could be implemented soon.

The ** (exponential) operator is a straightforward way of determining such things as memory depth. If a model has 10 address bits, it should have 1024 memory locations.

If the two operands of the ** operator are constants at compile-time, there is no reason a synthesis tool could not calculate the final value to be used during synthesis.

5.6 Local Parameters

Expected to be synthesizable? Yes.
When? This could be implemented soon.

Parameters, local to a module, that cannot be changed by parameter redefinition during instantiation is another enhancement to the Verilog-2001 Standard. Local parameters are declared using the keyword localparam.

This enhancement is needed by IP developers who want to create a parameterized design where only certain non-local parameters can be manually changed while other local parameters are manipulated within a design based on the parameters that are passed to a particular design instance. Restricting access to some parameters helps to insure that a IP users cannot inadvertently set incompatible parameter values for a particular module. The memory models in Example 10 and Example 11 both use local parameters to calculate one of the memory parameters based on other memory parameters.

5.7 Comma separated sensitivity list
Expected to be synthesizable? Yes.
When? Almost immediately.

Verilog-1995 uses the keyword "or" as a separator in the sensitivity list. New users of the Verilog language often ask the question, "can I use ***and*** in the sensitivity list?" The answer is no.

The "or" keyword is merely a separator between signals in the sensitivity list and nothing more. The Verilog sensitivity list is one of the few places where Verilog is more verbose than VHDL. I personally found this to be offensive!

VHDL separates signals in the sensitivity list with a comma character, which most Verilog users would agree is a better separator token. For this reason, the comma character has been added as an alternate way of separating signals in a Verilog sensitivity list.

Because this enhancement is really just a parsing change, it should be very easy to implement. There is no reason this capability should not be available by all Verilog vendors as soon as the Verilog-2001 Standard is released by the IEEE.

The Verilog code for a parameterized ram model in Example 10 uses a comma-separated sensitivity list in the always block just two lines before the endmodule statement.

```
//-----------------------------------------
// ram1 model - Verilog-2001 @(a, b, c)
// requires ADDR_SIZE & DATA_SIZE parameters
// MEM_DEPTH is automatically sized
//-----------------------------------------
module ram1 (addr, data, en, rw_n);
  parameter ADDR_SIZE = 10;
  parameter DATA_SIZE = 8;
  parameter  MEM_DEPTH = 1<<ADDR_SIZE;
  output [DATA_SIZE-1:0] data;
  input  [ADDR_SIZE-1:0] addr;
  input                  en, rw_n;

  reg    [DATA_SIZE-1:0] mem [0:MEM_DEPTH-1];

  assign data = (rw_n && en) ? mem[addr] : {DATA_SIZE{1'bz}};

  always @(addr, data, rw_n, en)
    if (!rw_n && en) mem[addr] = data;
endmodule
```

Example 10 - Parameterized Verilog ram model with comma-separated sensitivity list

5.8 @* combinational sensitivity list
Expected to be synthesizable? Yes.
When? Almost immediately.

The Verilog-2001 Standard refers to the @* operator as the implicit event expression list; however, members of the VSG called the always @* keyword-pair, the combinational logic

sensitivity list and that was its primary intended purpose, to be used to model and synthesize combinational logic.

Experienced synthesis engineers are aware of the problems that can occur if combinational always blocks are coded with missing sensitivity list entries. Synthesis tools build combinational logic strictly from the equations inside of an always block but then synthesis tools check the sensitivity list to warn the user of a potential mismatch between pre-synthesis and post-synthesis simulations [4].

The always @* procedural block will eliminate the need to list every single always-block input in the sensitivity list. This enhancement will reduce typing, and reduce design errors. The intent was to reduce effort when coding combinational sensitivity lists and to reduce opportunities for coding errors that could lead to a pre-synthesis and post-synthesis simulation mismatch.

The @* was really intended to be used at the top of an always block, but the VSG chose not to restrict its use to just that location. The VSG could not think of a good reason not to use, nor did the VSG think it was wise to restrict, the @* operator only to the top of the always block. This enhancement was made orthogonal but it should be used with caution and has the potential to be abused.

The Verilog code for a parameterized ram model in Example 11 uses an @* sensitivity list in the always block just two lines before the endmodule statement.

```
//-----------------------------------------
// ram1 model - Verilog-2001
// requires ADDR_SIZE & DATA_SIZE parameters
// MEM_DEPTH is automatically sized
//-----------------------------------------
module ram1 #(parameter  ADDR_SIZE = 10,
              parameter  DATA_SIZE = 8)
             (output [DATA_SIZE-1:0] data,
              input  [ADDR_SIZE-1:0] addr,
              input                  en, rw_n);

  localparam MEM_DEPTH = 1<<ADDR_SIZE;
  reg    [DATA_SIZE-1:0] mem [0:MEM_DEPTH-1];

  //-------------------------------------------------
  // Memory read operation
  //-------------------------------------------------
  assign data = (rw_n && en) ? mem[addr] : {DATA_SIZE{1'bz}};

  //-------------------------------------------------
  // Memory write operation - modeled as a latch-array
  //-------------------------------------------------
  always @*
    if (!rw_n && en) mem[addr] <= data;
endmodule
```

Example 11 - Parameterized Verilog ram model with @* combinational sensitivity list

The Verilog-2001 Standard notes that nets and variables which appear on the RHS of assignments, in function and task calls, or case expressions and if expressions shall all be included

in the implicit sensitivity list. Missing from the Verilog-2001 Standard is the fact that variables on the LHS of an expression when used as an index range and variables used in case items should also be included in the implicit sensitivity list. In the 3-to-8 decoder with output enable shown in Example 12, the en input and the a-inputs should included in the @* implied sensitivity list.

```
module decoder (
  output reg [7:0] y,
  input      [2:0] a,
  input            en);

  always @* begin
    y = 8'hff;
    y[a] = !en;
  end
endmodule
```

Example 12 - 3-to-8 Decoder model using the @* implicit sensitivity list

5.9 Constant functions

Expected to be synthesizable? Yes.
When? This might take some time to implement.

Perhaps the most contentious enhancement to the Verilog-2001 Standard, the enhancement that raised the most debate and that was almost removed from the standard on multiple occasions in the past five years, was the constant function. EDA vendors opposed this enhancement because of the perceived difficulty in efficiently implementing this enhancement, and its potential impact on compile-time performance.

A quote from an EDA vendor who requested that constant functions not be added to Verilog summarizes some of the opposition:

"Constant functions are another example of how a VHDL concept does not map will into Verilog ... The Verilog language is simply too powerful and unrestricted to support such functionality."

The users on the VSG also recognize that constant functions might not only be difficult to implement, but also impact compile times. Despite this potential impact on compile-time performance, users deemed this functionality too important to omit from the Verilog-2001 Standard. Vendors might want to publicize that a design modeled without constant functions will compile faster than designs that include constant functions.

Constant functions are important to IP developers. The objective of the constant function is to permit an IP developer to add local parameters to a module that are calculated from other parameters that could be passed into the module when instantiated.

Constant functions will require vendors to calculate some parameters at compile time, which will require that some parameters not be immediately calculated when read, but that they will be calculated after a function is used to determine the actual value of a parameter.

Consider the example of a simple ROM model. To make a parameterized version of a ROM model, we need to know the number of address bits, number of memory locations and number of data bits. The data bus width should be passed to the model, but only the memory size or number of address bits should be passed to the model. If we are given the number of address bits, we should calculate the memory depth at compile time. If we are given the number of memory locations, we should be able to calculate how many address bits are required at compile time.

In order to make constant functions somewhat more agreeable to EDA vendors, they were defined with significant restrictions including some that do not apply to normal Verilog functions. Significant restrictions that apply to constant functions include:

- Constant functions shall not contain hierarchical references.
- Constant functions are defined and invoked in the same module.
- Constant functions shall ignore system tasks. This permits a regular Verilog function with system tasks such as $display commands to be changed into a constant function without requiring removal of the system tasks.
- Constant functions shall not permit system functions.
- Constant functions shall have no side effects (they shall not make assignments to variables that are defined outside of the constant function).
- If a constant function uses an external parameter within the internal calculations of the function, the external parameter must be declared before the constant function call.
- All variables used in a constant function that are not parameters or functions must be declared locally in the constant function.
- If the constant function uses a parameter that is directly or indirectly modified by a defparam statement, the behavior of the Verilog compiler is undefined. The compiler can return an unknown value or it can issue a syntax error.
- Constant functions cannot be defined inside of a generate statement.
- Constant functions shall not call other constant functions in any context that requires a constant expression.

The VSG anticipated that the typical use of a constant function would be to perform simple calculations to generate local parameters to insure compatibility with passed parameters. The above restrictions insure that constant functions do not cause undue compile-time problems.

5.10 Attributes

Expected to be synthesizable? Partially.
When? As soon as the IEEE synthesis committee finishes its work and includes attributes into the synthesis spec.

Verilog-2001 will add a new construct (new to Verilog) called an attribute. The attribute uses (* *) tokens (named "funny braces" by members of the VSG) as shown in Figure 1.

```
(* attribute_name = constant_expression *)
                                        -or-
(* attribute_name *)
```

Figure 1 - Legal attribute definition syntax using (* *)

Attributes were primarily added to the Verilog language enable other tools to use Verilog as an input language and still pass non-Verilog information to those tools. For many years now, vendors have been adding hooks into the Verilog language by way of synthetic comments. The most famous (infamous) example is the deadly[6] synthetic comment:

```
// synopsys full_case parallel_case
```

The biggest problem with the synthetic comment approach is that attaching tool-specific information to a Verilog comment forces those same tools to parse all Verilog comments to see if the comment contains a tool-specific directive.

To assist vendors who use Verilog as an input language, the VSG decided to add attributes to the Verilog language that for the most part will be ignored by Verilog compilers the same as any Verilog comment. The attributes permit third-party vendors to add tool-related information to the source code without impacting simulation and without having to parse every Verilog comment.

5.11 Required net declarations
Expected to be synthesizable? N/A.
When? Soon.

Verilog-1995 has an odd and non-orthogonal requirement that all 1-bit nets, driven by a continuous assignment, that are not declared to be a ports, must be declared. It is the only 1-bit net type that must be declared in Verilog. This non-orthogonal restriction is removed in Verilog-2001.

```
module andor1 (y, a, b, c);
  output y;
  input  a, b, c;
  wire   n1; // not required in Verilog-2001

  assign n1 = a & b;
  assign y  = n1 | c;
endmodule
```

Example 13 - Verilog-1995 required net declaration for the LHS of a continuous assignment to an internal net

5.12 `default_nettype none
Expected to be synthesizable? N/A.
When? Soon.

In the Verilog-1995 Standard, any undeclared identifier, except for the output of a continuous assignment that drives a non-port net, is by default a 1-bit wire. Verilog never required these 1-bit

net declarations and adding the declarations to a model yielded no additional checking to insure that all 1-bit nets were declared.

The Verilog-2001 Standard adds a new option to the `defult_nettype compiler directive called "none." If the "none" option is selected, all 1-bit nets must be declared.

Whether or not forcing all 1-bit nets to be declared is a good coding practice or not is open to debate. Some engineers believe that all nets should be declared before they are used. Other engineers find that declaring all 1-bit nets can be both time and space-consuming.

Editorial comment: I personally find the practice of declaring all 1-bit nets to be a waste of time, effort and lines of code. VHDL requires all 1-bit nets (signals) to be declared, and on a VHDL ASIC design that I worked on in 1996, while instantiating and connecting the major sub-blocks and I/O pads at the top-level model of an ASIC design, I spent as much time debugging flawed signal declarations as I did debugging real hardware problems. The only declarations that I personally found useful were multi-bit signals (buses), which are also required in Verilog-1995. My signal declarations extended over three pages of code and offered no additional useful information about the design. Nevertheless, one can now inflict similar pain and suffering into a Verilog design using the `default_nettype none compiler directive.

6.0 Array of Instance

Expected to be synthesizable? Yes.
When? Soon.

A noteworthy enhancement to the Verilog language is the Array of Instance that was added to the 1995 IEEE Verilog Standard. This enhancement was implemented by Cadence more than two years ago, but the other simulation vendors and all synthesis vendors were slow to follow.

An array of instance allows an simple one-dimensional linear array of instances to be declared in a single statement.

Most ASIC designers build a top-level module that only permits instantiation of other modules, no RTL code allowed. The RTL code is used in sub-modules but not in the top-level module.

In the top-level module, all of the major sub-blocks are instantiated along with all of the ASIC I/O pads. Consider the task of instantiating the I/O pads for a 32-bit address bus and a 16-bit data bus. Verilog engineers have always been required to make 32 address-pad and 16 data-pad instantiations in the top-level model as shown in Example 14.

```
module top_pads1 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
          .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));
```

```
  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  IBUF i15 (.O(data[15]), .pI(pdata[15]));
  IBUF i14 (.O(data[14]), .pI(pdata[14]));
  IBUF i13 (.O(data[13]), .pI(pdata[13]));
  IBUF i12 (.O(data[12]), .pI(pdata[12]));
  IBUF i11 (.O(data[11]), .pI(pdata[11]));
  IBUF i10 (.O(data[10]), .pI(pdata[10]));
  IBUF i9  (.O(data[ 9]), .pI(pdata[ 9]));
  IBUF i8  (.O(data[ 8]), .pI(pdata[ 8]));
  IBUF i7  (.O(data[ 7]), .pI(pdata[ 7]));
  IBUF i6  (.O(data[ 6]), .pI(pdata[ 6]));
  IBUF i5  (.O(data[ 5]), .pI(pdata[ 5]));
  IBUF i4  (.O(data[ 4]), .pI(pdata[ 4]));
  IBUF i3  (.O(data[ 3]), .pI(pdata[ 3]));
  IBUF i2  (.O(data[ 2]), .pI(pdata[ 2]));
  IBUF i1  (.O(data[ 1]), .pI(pdata[ 1]));
  IBUF i0  (.O(data[ 0]), .pI(pdata[ 0]));

  BIDIR b31 (.N2(addr[31]), .pN1(paddr[31]), .WR(wr));
  BIDIR b30 (.N2(addr[30]), .pN1(paddr[30]), .WR(wr));
  BIDIR b29 (.N2(addr[29]), .pN1(paddr[29]), .WR(wr));
  BIDIR b28 (.N2(addr[28]), .pN1(paddr[28]), .WR(wr));
  BIDIR b27 (.N2(addr[27]), .pN1(paddr[27]), .WR(wr));
  BIDIR b26 (.N2(addr[26]), .pN1(paddr[26]), .WR(wr));
  BIDIR b25 (.N2(addr[25]), .pN1(paddr[25]), .WR(wr));
  BIDIR b24 (.N2(addr[24]), .pN1(paddr[24]), .WR(wr));
  BIDIR b23 (.N2(addr[23]), .pN1(paddr[23]), .WR(wr));
  BIDIR b22 (.N2(addr[22]), .pN1(paddr[22]), .WR(wr));
  BIDIR b21 (.N2(addr[21]), .pN1(paddr[21]), .WR(wr));
  BIDIR b20 (.N2(addr[20]), .pN1(paddr[20]), .WR(wr));
  BIDIR b19 (.N2(addr[19]), .pN1(paddr[19]), .WR(wr));
  BIDIR b18 (.N2(addr[18]), .pN1(paddr[18]), .WR(wr));
  BIDIR b17 (.N2(addr[17]), .pN1(paddr[17]), .WR(wr));
  BIDIR b16 (.N2(addr[16]), .pN1(paddr[16]), .WR(wr));
  BIDIR b15 (.N2(addr[15]), .pN1(paddr[15]), .WR(wr));
  BIDIR b14 (.N2(addr[14]), .pN1(paddr[14]), .WR(wr));
  BIDIR b13 (.N2(addr[13]), .pN1(paddr[13]), .WR(wr));
  BIDIR b12 (.N2(addr[12]), .pN1(paddr[12]), .WR(wr));
  BIDIR b11 (.N2(addr[11]), .pN1(paddr[11]), .WR(wr));
  BIDIR b10 (.N2(addr[10]), .pN1(paddr[10]), .WR(wr));
  BIDIR b9  (.N2(addr[ 9]), .pN1(paddr[ 9]), .WR(wr));
  BIDIR b8  (.N2(addr[ 8]), .pN1(paddr[ 8]), .WR(wr));
  BIDIR b7  (.N2(addr[ 7]), .pN1(paddr[ 7]), .WR(wr));
  BIDIR b6  (.N2(addr[ 6]), .pN1(paddr[ 6]), .WR(wr));
  BIDIR b5  (.N2(addr[ 5]), .pN1(paddr[ 5]), .WR(wr));
  BIDIR b4  (.N2(addr[ 4]), .pN1(paddr[ 4]), .WR(wr));
  BIDIR b3  (.N2(addr[ 3]), .pN1(paddr[ 3]), .WR(wr));
  BIDIR b2  (.N2(addr[ 2]), .pN1(paddr[ 2]), .WR(wr));
  BIDIR b1  (.N2(addr[ 1]), .pN1(paddr[ 1]), .WR(wr));
  BIDIR b0  (.N2(addr[ 0]), .pN1(paddr[ 0]), .WR(wr));

endmodule
```

Example 14 - Verilog-1995 structural top-level ASIC model with multiple I/O pad instantiations

VHDL engineers have been able to use two generate for-loops to instantiate the same 32 address and 16 data pad models. With Verilog-2001, Verilog engineers can now use similarly simple generate for-loops to instantiate the 32 address and 16 data pads, as shown in Example 15.

```
module top_pads2 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
               .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));

  genvar i;

  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  generate for (i=0; i<16; i=i+1) begin: dat
    IBUF i1 (.O(data[i]), .pI(pdata[i]));

  generate for (i=0; i<32; i=i+1) begin: adr
    BIDIR b1 (.N2(addr[i]), .pN1(paddr[i]), .WR(wr));

endmodule
```

Generated instance names dat[0].i1 to dat[15].i1

Generated instance names adr[0].b1 to adr[31].b1

Example 15 - Top-level ASIC model with address and data I/O pads instantiated using a generate statement

For simple contiguous one-dimensional arrays, the array of instance construct is even easier to use and has a more intuitive syntax. Finally, simulation and synthesis vendors are now starting to support the Verilog-1995 Array of Instance construct that makes placement of 32 consecutively named instances possible with an easy instantiation by bus names as ports and applying a range to the instance name as shown in Example 16.

```
module top_pads3 (pdata, paddr, pctl1, pctl2, pctl3, pclk);
  inout  [15:0] pdata;                    // pad data bus
  input  [31:0] paddr;                    // pad addr bus
  input         pctl1, pctl2, pctl3, pclk; // pad signals
  wire   [15:0] data;                     // data bus
  wire   [31:0] addr;                     // addr bus

  main_blk u1 (.data(data), .addr(addr),
               .sig1(ctl1), .sig2(ctl2), .sig3(ctl3), .clk(clk));

  IBUF c4 (.O(ctl3), .pI(pctl3));
  IBUF c3 (.O(ctl2), .pI(pctl2));
  IBUF c2 (.O(ctl1), .pI(pctl1));
  IBUF c1 (.O( clk), .pI( pclk));

  IBUF i[15:0] (.O(data), .pI(pdata));

  BIDIR b[31:0] (.N2(addr), .pN1(paddr), .WR(wr));

endmodule
```

Arrayed instance names i[15] to i[0]

Arrayed instance names b[31] to b[0]

Example 16 - Top-level ASIC model with address and data I/O pads instantiated using arrays of instance

7.0 Conclusions

The Verilog-2001 enhancements are coming. These enhancements will increase the efficiency and productivity of Verilog designers.

8.0 Honorable Mention

Although the Behavioral Task Force benefited from the expertise and contributions of numerous synthesis experts, a particular honorable mention must go out to Kurt Baty of WSFDB.

Kurt has experience designing some 50 ASICs and has written a significant number of Design Ware models that are used in Synopsys synthesis tools. Kurt complains that he had to write all of the models using VHDL because Verilog lacked a few of the key features that are required to make parameterized models. Kurt's insight into the 1995 Verilog limitations lead to enhancements that will make future IP model creation not only doable, but also easier to do in Verilog than it was in VHDL.

9.0 References

[1] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Std P1364/D5

[2] Douglas L. Perry, *VHDL*, McGraw-Hill, Inc., 1994, p. 1.

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

[4] www.chris.spear.net

[5] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG'99 (Synopsys Users Group San Jose, CA, 1999) Proceedings*, section-TA2 (1st paper), March 1999.

[6] Clifford E. Cummings, "'full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, section-FA1 (2nd paper), October 1999.

Revision 1.2 - What Changed?

The ANSI style ports in previous versions of this paper incorrectly showed semi-colons between port declarations and between the parameter list and the port list. These errors were fixed in this version of the document.

Revision 1.3 (April 2002) - What Changed?

Example 11 still incorrectly showed ANSI style parameters separated by a semicolon instead of a comma and also included an illegal comma at the end of the ANSI style parameter list. Note that the second **`parameter`** keyword is not required in the ANSI style parameter list and would typically be removed.

Example 11 also included a localparam declaration in the ANSI style parameter list, which is illegal. The localparam has been moved outside of the ANSI style header.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

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

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

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

(Data accurate as of December 17th, 2001)

New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!)

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

Abstract

Creating reusable models typically requires that general-purpose models be written with re-definable parameters such as SIZE, WIDTH and DEPTH.

With respect to coding parameterized Verilog models, two Verilog constructs that are over-used and abused are the global macro definition (`define) and the infinitely abusable parameter redefinition statement (defparam).

This paper will detail techniques for coding proper parameterized models, detail the differences between parameters and macro definitions, present guidelines for using macros, parameters and parameter definitions, discourage the use of defparams, and detail Verilog-2001 enhancements to enhance coding and usage of parameterized models.

1. Introduction

Two Verilog constructs that are overused and abused are the Verilog macro definition statement (**`define**) and the infinitely abusable **defparam** statement. It is the author's opinion that macro definitions are largely over-used to avoid the potential abuse of the dangerous **defparam** statement by design teams.

Respected Verilog and verification texts over-promote the usage of the macro definition (**`define**) statement, and those recommendations are being followed without recognition of the dangers that these recommendations introduce.

Note: even though multiple questionable parameter and macro definition recommendations are cited from *Principles of Verifiable RTL Design* by Bening and Foster[13] and from *Writing Testbenches, Functional Verification of HDL Models* by Bergeron[8], I still recommend both texts for the other valuable material they both contain, especially the text by Bening and Foster.

2. Verilog Constants

In Verilog-1995[6], there are two ways to define constants: the **parameter**, a constant that is local to a module and macro definitions, created using the **`define** compiler directive.

A **parameter**, after it is declared, is referenced using the parameter name.

A **`define** macro definition, after it is defined, is referenced using the macro name with a preceding ` (back-tic) character.

It is easy to distinguish between **parameters** and macros in a design because macros have a **`***identifier_name* while a **parameter** is just the *identifier_name* without back-tic.

3. Parameters

Parameters must be defined within module boundaries using the keyword **parameter**.

A **parameter** is a constant that is local to a module that can optionally be redefined on an instance-by-instance basis. For parameterized modules, one or more parameter declarations typically precede the port declarations in a Verilog-1995 style model, such as the simple register model in Example 1.

```
module register (q, d, clk, rst_n);
  parameter SIZE=8;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

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

Example 1 - Parameterized register model - Verilog-1995 style

The Verilog-2001[5] version of the same model can take advantage of both the ANSI-C style ports and module header parameter list, as shown in Example 2.

```
module register2001 #(parameter SIZE=8)
  (output reg [SIZE-1:0] q,
   input      [SIZE-1:0] d,
   input                 clk, rst_n);

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

Example 2 - Parameterized register model - Verilog-2001 style

4. Parameters and Parameter Redefinition

When instantiating modules with parameters, in Verilog-1995 there are two ways to change the parameters for some or all of the instantiated modules; parameter redefinition in the instantiation itself, or separate **defparam** statements.

Verilog-2001 adds a third and superior method to change the parameters on instantiated modules by using named parameter passing in the instantiation itself (see section 7).

5. Parameter redefinition using

Parameter redefinition during instantiation of a module uses the **#** character to indicate that the parameters of the instantiated module are to be redefined.

In Example 3, two copies of the register from Example 1 are instantiated into the **two_regs1** module. The **SIZE** parameter for both instances is set to 16 by the **#(16)** parameter redefinition values on the same lines as the register instantiations themselves.

```
module two_regs1 (q, d, clk, rst_n);
  output [15:0] q;
  input  [15:0] d;
  input         clk, rst_n;
  wire   [15:0] dx;

  register #(16) r1 (.q(q), .d(dx),
          .clk(clk), .rst_n(rst_n));

  register #(16) r2(.q(dx), .d(d),
          .clk(clk), .rst_n(rst_n));
endmodule
```

Example 3 - Instantiation using parameter redefinition

This form of parameter redefinition has been supported by all synthesis tools for many years.

The biggest problem with this type of parameter redefinition is that the parameters must be passed to the instantiated module in the order that they appear in the module being instantiated.

Consider the myreg module of Example 4.

```
module myreg (q, d, clk, rst_n);
  parameter Trst = 1,
            Tckq = 1,
            SIZE = 4,
            VERSION = "1.1";
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  reg    [SIZE-1:0] q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= #Trst 0;
    else        q <= #Tckq d;
endmodule
```

Example 4 - Module with four parameters

The **myreg** module of Example 4 has four parameters, and if the module, when instantiated, requires that just the third parameter, (for example the **SIZE** parameter) be changed, the module cannot be instantiated with a series of commas followed by the new value for the **SIZE** parameter as shown in Example 5. This would be a syntax error.

```
module bad_wrapper (q, d, clk, rst_n);
  output [7:0] q;
  input  [7:0] d;
  input        clk, rst_n;

  // illegal parameter passing example
  myreg #(,,8) r1 (.q(q), .d(d),
               .clk(clk), .rst_n(rst_n));
endmodule
```

Example 5 - Parameter redefinition with #(,,8) syntax error

In order to use the parameter redefinition syntax when instantiating a module, all parameter values up to and including all values that are changed, must be listed in the instantiation. For the **myreg** module of Example 4, the first two parameter values must be listed, even though they do not change, followed by the new value for the **SIZE** parameter, as shown in Example 6.

```
module good_wrapper (q, d, clk, rst_n);
  output [7:0] q;
  input  [7:0] d;
  input        clk, rst_n;

  // the first two parameters must be
  // explicitly passed even though the
  // values did not change
  myreg #(1,1,8) r1 (.q(q), .d(d),
                .clk(clk), .rst_n(rst_n));
endmodule
```

Example 6 - Parameter redefinition with correct #(1,1,8) syntax

Aware of this limitation, engineers have frequently rearranged the order of the parameters to make sure that the most frequently used parameters are placed first in a module, similar to the technique described by Thomas and Moorby[4].

Despite the limitations of Verilog-1995 parameter redefinition, it is still the best supported and cleanest method for modifying the parameters of an instantiated module.

Verilog-2001 actually enhances the above parameter redefinition capability by adding the ability to pass the parameters by name, similar to passing port connections by name. See section 7 for information on this new and preferred way of passing parameters to instantiated modules.

6. Death to defparams!

First impressions of **defparam** statements are very favorable. In fact, many authors, like Bergeron, prefer usage of the **defparam** statement because "it is self documenting and robust to changes in parameter declarations"[9].

The **defparam** statement explicitly identifies the instance *and* the individual parameter that is to be redefined by each **defparam** statement. The **defparam** statement can be placed before the instance, after the instance or anywhere else in the file.

Until the year 2000, Synopsys tools did not permit parameter redefinition using **defparam** statements. Synopsys was to be commended for this restriction. Unfortunately, Synopsys developers bowed to pressure from uninformed engineers and added the ability to use **defparam** statements in recent versions of Synopsys tools.

Unfortunately, the well-intentioned **defparam** statement is easily abused by:

(1) using **defparam** to hierarchically change the parameters of a module.

(2) placing the **defparam** statement in a separate file from the instance being modified.

(3) using multiple **defparam** statements in the same file to change the parameters of an instance.

(4) using multiple **defparam** statements in multiple different files to change the parameters of an instance.

6.1. Hierarchical defparams

It is legal to hierarchically change the values of parameters using a **defparam** statement. This means that any **parameter** in a design can be changed from any input file in the design. Potentially, the abuse could extend to changing the parameter value of the module that instantiated the module with the **defparam** statement and pass that **parameter** to the instantiated module that in turn re-modifies the **parameter** of the instantiating module again, etc.

In Example 7, the **testbench** module (**tb_defparam**) instantiates a model and passes the **SIZE** parameter to the register module (passed to the **WIDTH** parameter), which passes the **WIDTH** parameter to the **dff** module (passed to the **N** parameter). The **dff** module has an erroneous hierarchical **defparam** statement that changes the testbench **SIZE parameter** from **8** to **1** and that value is again passed down the hierarchy to change the register **WIDTH** and the **dff N** values again.

```
module tb_defparam;
  parameter SIZE=8;
  wire [SIZE-1:0] q;
  reg  [SIZE-1:0] d;
  reg             clk, rst_n;

  register2 #(SIZE) r1
             (.q(q), .d(d), .clk(clk),
              .rst_n(rst_n));
  // ...
endmodule

module register2 (q, d, clk, rst_n);
  parameter WIDTH=8;
  output [WIDTH-1:0] q;
  input  [WIDTH-1:0] d;
  input              clk, rst_n;

  dff #(WIDTH) d1
       (.q(q), .d(d), .clk(clk),
        .rst_n(rst_n));
endmodule
```

Example 7 - Dangerous use of hierarchical defparam
(example continues on next page)

```
module dff (q, d, clk, rst_n);
  parameter N=1;
  output [N-1:0] q;
  input  [N-1:0] d;
  input             clk, rst_n;
  reg    [N-1:0] q;

  // dangerous, hierarchical defparam
  defparam tb_defparam.SIZE = 1;

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

Example 7 - Dangerous use of hierarchical defparam

All of the ports and variables in the designs in Example 7 are now just one bit wide, while synthesis of the **register2** and **dff** modules will be eight bits wide. This type of **defparam** use can easily escape detection and cause design and debug problems.

```
module register3 (q, d, clk, rst_n);
  parameter WIDTH=8;
  output [WIDTH-1:0] q;
  input  [WIDTH-1:0] d;
  input              clk, rst_n;

  dff3 #(WIDTH) d1
        (.q(q), .d(d), .clk(clk),
         .rst_n(rst_n));
endmodule

module dff3 (q, d, clk, rst_n);
  parameter N=1;
  output [N-1:0] q;
  input  [N-1:0] d;
  input             clk, rst_n;
  reg    [N-1:0] q;

  // dangerous, hierarchical defparam
  defparam register3.WIDTH = 1;

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

Example 8 - Dangerous hierarchical defparams enclosed within the register3/dff3 models

Example 8 is similar to Example 7 except that the **defparam** redefines the bus widths of the **register3** model and appears to be self-contained. Unfortunately, even though this model will simulate like a 1-bit wide model, it still synthesizes to an 8-bit wide model.

6.2. defparams in separate files

It is not uncommon to find **defparams** being abused by placing them in a completely different file from the instances being modified[10].

Unfortunately, this practice was semi-encouraged by the following comment in section 12.2.1 of the Verilog-1995[6] and Verilog-2001[5] Standards documents:

> The defparam statement is particularly useful for grouping all of the parameter value override assignments together in one module.

The above text probably should have been deleted from the Verilog-2001 Standard, but it was not.

It should be noted that the Verilog Standards Group (VSG) introduced and encourages the use of the superior capability of passing parameters by name (see section 7), similar to passing ports by name, when instantiating modules. The VSG hopes that engineers will take advantage of this new capability and that **defparam** statements eventually die (see section 6.6).

6.3. Multiple defparams in the same file

defparams are abused by placing multiple **defparams** in the same file that modify the same **parameter**. The Verilog-2001 Standard defines the correct behavior to be:

> **In the case of multiple defparams for a single parameter, the parameter takes the value of the last defparam statement encountered in the source text.[5]**

In Example 9, two copies of the **register** from Example 1 are instantiated into the **two_regs2** module. The **SIZE** parameter for both instances is set to **16** by **defparam** statements placed before the corresponding **register** instantiations. A third **defparam** statement is placed after the second **register** instantiation, changing the size of the second register to **4** by mistake.

```
module two_regs2 (q, d, clk, rst_n);
  parameter SIZE = 16;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             clk, rst_n;
  wire   [SIZE-1:0] dx;

  defparam r1.SIZE=16;
  register r1 (.q(q), .d(dx), .clk(clk),
               .rst_n(rst_n));

  defparam r2.SIZE=16;
  register r2 (.q(dx), .d(d), .clk(clk),
               .rst_n(rst_n));
  defparam r2.SIZE=4; // Design error!
endmodule
```

Example 9 - Instantiation using defparam statements

Because this is a small design and because compilers will issue "port-size mismatch" warnings, this design will not be difficult to debug.

Unfortunately, frequently when a second stray **defparam** statement is added by mistake, it is added into a large design with pages of RTL code because the designer did not notice that an earlier **defparam** statement had been used to redefine the same parameter value. This type of design is typically more confusing and more difficult to debug.

6.4. Multiple defparams in separate files

defparams are even abused by placing them in multiple different files.

The practice of placing multiple **defparam** statements in different files that make assignments to the same **parameter** is very problematic. Multiple **defparam** statements are treated differently by different vendors because the behavior for this scenario was never defined in the Verilog-1995 Standard.

The Verilog-2001 Standards Group did not want to encourage this behavior so we added the following disclaimer to the Verilog-2001 Standard.

> **When defparams are encountered in multiple source files, e.g., found by library searching, the defparam from which the parameter takes its value is undefined.[5]**

The Verilog-2001 Standards Group basically wanted to discourage this practice altogether so we left the behavior undefined and documented that fact, hoping to discourage anyone from requiring vendors to support this flawed strategy.

6.5. defparams and tools

Since **defparams** can be placed anywhere in a design and because they can hierarchically change the **parameter** values of any module in a design, **defparams** in their current incarnation make it very difficult to write either a vendor tool or an in-house tool that can accurately parse a design that is permitted to include **defparam** statements[14].

A Verilog compiler cannot determine the actual values of any parameters until all of the Verilog input files have been read, because the last file read might change every single parameter in the design!

I know of some companies that ban the use of **defparams** in their Verilog code in order to facilitate the creation of useful in-house Verilog tools. I agree with this practice and propose the following guideline:

Guideline: do not use **defparams** in any Verilog designs.

A superior alternative to **defparam** statements is discussed in section 7.

6.6. Deprecate defparam

The VSG is not the only organization that hopes that the defparam statement will die (see the end of section 6.2).

The IEEE Verilog Synthesis Interoperability Group voted not to support **defparam** statements in the IEEE Verilog Synthesis Standard[7].

And in April 2002, The SystemVerilog Standards Group voted unanimously (with one abstention) to deprecate the **defparam** statement (possibly remove support for the **defparam** statement from future versions of the Verilog language)[1].

After **defparams** have been deprecated, the author suggests that future Verilog tools report errors whenever a **defparam** statement is found in any Verilog source code and then provide a switch to enable **defparam** statement use for backward compatibility. An error message similar to the following is suggested:

> **"The Verilog compiler found a defparam statement in the source code at (file_name/line#).**
> **To use defparam statements in the Verilog source code, you must include the switch +Iamstupid on the command line which will degrade compiler performance and introduce potential problems but is bug-compatible with Verilog-1995 implementations.**
> **Defparam statements can be replaced with named parameter redefinition as define by the IEEE Verilog-2001 standard."**

The preceeding **defparam** warning is annoyingly long. Hopefully users will tire of these long annoying warnings and remove **defparams** from their code.

7. Verilog-2001 named parameter redefinition

An enhancement added to the Verilog-2001 Standard is the ability to instantiate modules with named parameters in the instantiation itself[3][5].

This enhancement is superior to and eliminates the need for **defparam** statements.

```
module demuxreg (q, d, ce, clk, rst_n);
  output [15:0] q;
  input  [ 7:0] d;
  input         ce, clk, rst_n;
  wire   [15:0] q;
  wire   [ 7:0] n1;

  not          u0 (ce_n, ce);
  regblk #(.SIZE( 8)) u1
          (.q(n1), .d (d), .ce(ce),
           .clk(clk), .rst_n(rst_n));
  regblk #(.SIZE(16)) u2
          (.q (q), .d({d,n1}), .ce(ce_n),
           .clk(clk), .rst_n(rst_n));
endmodule

module regblk (q, d, ce, clk, rst_n);
  parameter SIZE = 4;
  output [SIZE-1:0] q;
  input  [SIZE-1:0] d;
  input             ce, clk, rst_n;
  reg    [SIZE-1:0] q;

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

Example 10 - Instantiation using named parameter passing

This new technique offers the advantage of specifically indicating which **parameter** is modified (like the **defparam** statement) and also places the **parameter** values conveniently into the instantiation syntax, like Verilog-1995 **#** parameter redefinition.

This is the cleanest way to instantiate models from any vendor and this is a technique that should be encouraged by designers and vendors of reusable models.

Because all of the parameter information is included in the instantiation of the model, this coding style will also be easiest to parse by vendor and in-house tools.

Guideline: require all passing of parameters to be done using the new Verilog-2001 named parameter redefinition technique.

8. `define Macro Substitution

The **`define** compiler directive is used to perform "global" macro substitution, similar to the C-language #define directive. Macro substitutions are global from the point of definition and remain active for all files read after the macro definition is made or until another macro definition changes the value of the defined macro or until the macro is undefined using the **`undef** compiler directive.

Macro definitions can exist either inside or outside of a module declaration, and both are treated the same. **parameter** declarations can only be made inside of module boundaries.

Since macros are defined for all files read after the macro definition, using macro definitions generally makes compiling a design file-order dependent.

A typical problem associated with using macro definitions is that another file might also make a macro definition to the same macro name. When this occurs, Verilog compilers issue warnings related to "macro redefinition" but an unnoticed warning can be costly to the design or to the debug effort.

Why is it bad to redefine macros? The Verilog language allows hierarchical referencing of identifiers. This proves to be very valuable for probing and debugging a design. If the same macro name has been given multiple definitions in a design, only the last definition will be available to the testbench for probing and debugging purposes.

If you find yourself making multiple macro definitions to the same macro name, consider that the macro should probably be a local **parameter** as opposed to a global macro.

9. `define Usage

Guideline: only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design.

Guideline: where possible, place all macro definitions into one "**definitions.vh**" file and read the file first when compiling the design.

Alternate Guideline: place all macro definitions in the top-level testbench module and read this module first when compiling the design.

Reading all macro definitions first when compiling a design insures that the macros exist when they are needed and that they are globally available to all files compiled in the design.

Pay attention to warnings about macro redefinition.

Guideline: do not use macro definitions to define constants that are local to a module.

10. `define Inclusion

One popular technique to insure that a macro definition exists before its usage is to use an **`ifdef**, or the new Verilog-2001 **`ifndef** compiler directives to query for the existence of a macro definition followed by either a **`define** macro assignment or a **`include** of a file name that contains the require macro definition.

```
`ifdef CYCLE
  // do nothing (better to use `ifndef)
`else
  `define CYCLE 100
`endif

`ifndef CYCLE
  `include "definitions.vh"
`endif
```

Example 11 - Testing and defining macro definitions

11. The `undef compiler directive

Verilog has the **`undef** compiler directive to remove a macro definition created with the **`define** compiler directive.

Bergeron recommends avoiding the use of macro definitions[11]. I agree with this recommendation. Bergeron further recommends that all macro definitions should be removed using **`undef** when no longer needed[11]. I disagree with this recommendation. This seems to be overkill to correct a problem that rarely exists. Using the **`define** compiler directive to create global macros where appropriate is very useful. Losing sleep over the existence of global macro definitions and tracking all of the **`undef**'s in a design is not a good use of time.

For the rare occasion where it might make sense to redefine a macro, use **`undef** in the same file and at the end of the file where the **`define** macro was defined.

Make sure that the last compiled macro definition is likely to be the macro that you might want to access from a testbench, because only one macro definition can exist during runtime debug.

Again, using a **`define**-**`undef** pair should be considered the last resort to a problem that could probably be better handled using a better method.

12. Clock cycle definition

Bergeron's somewhat justified paranoia over the use of the **`define** macro definition leads him to recommend that clock cycles be defined using **parameters** as opposed to using the **`define** compiler directive[12]. This recommendation is flawed.

Guideline: make clock cycle definitions using the **`define** compiler directive. Example:

```
`define CYCLE 10
```

Guideline: place the clock cycle definitions in the "**definitions.vh**" file or in the top-level testbench. Example:

```
`define CYCLE 10
module tb_cycle;
  // ...
  initial begin
    clk = 1'b0;
    forever #(`CYCLE/2) clk = ~clk;
  end
  // ...
endmodule
```

Example 12 - Global clock cycle macro definition and usage (recommended)

Reason: Clock cycles are a fundamental constant of a design and testbench. The cycle of a common clock signal should not change from one module to another; the cycle should be constant!

Verilog power-users do most stimulus generation and verification testing on clock edges in a testbench. In general, this type of testbench scales nicely with changes to the global clock cycle definition.

13. State Machines and `define do not mix

Bening and Foster[13] and Keating and Bricaud[15] both recommend using the **`define** compiler directive to define state names for a Verilog state machine design. After recommending the use of **`define**, Keating and Bricaud subsequently show an example using **parameter** definitions instead of using the **`define**[16]. The latter is actually preferred.

Finite State Machine (FSM) designs should use **parameters** to define state names because the state name is a constant that applies only to the FSM module. If multiple state machines are added to a large design, it is not uncommon to want to reuse certain state names in multiple FSM designs[1]. Example state names that are common to multiple designs include: RESET, IDLE, READY, READ, WRITE, ERROR and DONE.

Using **`define** to assign state names would either preclude reuse of a state name because the name has already been taken in the global name space, or one would have to **`undef** state names between modules and re-**`define** state names in the new FSM modules. The latter case makes it difficult to probe the internal values of FSM

state buses from a testbench and running comparisons to the state names.

There is no good reason why state names should be defined using **`define**. State names should not be considered part of the global name space. State names should be considered local names to the FSM module that encloses them.

Guideline: do not make state assignments using **`define** macro definitions for state names.

Guideline: Make state assignments using **parameters** with symbolic state names.

14. Verilog-2001 localparam

An enhancement added to the Verilog-2001 Standard is the **localparam**.

Unlike a **parameter**, a **localparam** cannot be modified by parameter redefinition (positional or named redefinition) nor can a **localparam** be redefined by a **defparam** statement.

The **localparam** can be defined in terms of **parameters** that can be redefined by positional parameter redefinition, named parameter redefinition (preferred) or **defparam** statements.

The idea behind the **localparam** is to permit generation of some local parameter values based on other **parameters** while protecting the **localparams** from accidental or incorrect redefinition by an end-user.

In Example 13, the size of the memory array **mem** should be generated from the size of the address bus. The memory depth-size **MEM_DEPTH** is "protected" from incorrect settings by placing the **MEM_DEPTH** in a **localparam** declaration. The **MEM_DEPTH** parameter will only change if the **ASIZE** parameter is modified.

```
module ram1 #(parameter ASIZE=10,
                        DSIZE=8)
   (inout  [DSIZE-1:0] data,
    input  [ASIZE-1:0] addr,
    input              en, rw_n);

   // Memory depth equals 2**(ASIZE)
   localparam MEM_DEPTH = 1<<ASIZE;
   reg [DSIZE-1:0] mem [0:MEM_DEPTH-1];

   assign data = (rw_n && en) ? mem[addr]
                 : {DSIZE{1'bz}};

   always @(addr, data, rw_n, en)
     if (!rw_n && en) mem[addr] = data;
endmodule
```

Example 13 - Verilog-2001 ANSI-parameter and port style model with localparam usage

We want to protect the local **MEM_DEPTH** parameter and calculate it from the size parameter value of the address bus.

Note: the Verilog-2001 Standard does not extend the capabilities of the localparam enhancement to the module header parameter list. Specifically, localparam currently cannot be added to an ANSI-style parameter list as shown in Example 14.

```
module multiplier2
    #(parameter  AWIDTH=8, BWIDTH=8,
      localparam YWIDTH=AWIDTH+BWIDTH)
     (output [YWIDTH-1:0] y,
      input  [AWIDTH-1:0] a,
      input  [BWIDTH-1:0] b);
  assign y = a * b;
endmodule
```

Example 14 - Illegal use of localparam in the ANSI-parameter header

15. `timescale Definitions

The **`timescale** directive gives meaning to delays that may appear in a Verilog model. The timescale is placed above the module header and takes the form:

```
`timescale time_unit / time_precision
```

The **`timescale** directive can have a huge impact on the performance of most Verilog simulators. It is a common new-user mistake to select a time_precision of **1ps** (1 pico-second) in order to account for every last pico-second in a design. adding a **1ps** precision to a model that is adequately modeled using either **1ns** or **100ps** time_precisions can increase simulation time by more than 100% and simulation memory usage by more than 150%. I know of one very popular and severely flawed synthesis book that shows Verilog coding samples using a `timescale of **1 ns / 1 fs**[17] (measuring simulation performance on this type of design typically requires a calendar watch!)

I have seen some engineers use a macro definition to facilitate changing all **`timescales** in a design. All modules coded by these engineers include the timescale macro before every module header that they ever write. Example 15 shows a macro definition for a global **`timescale** and usage of the global **`timescale** macro.

```
`define tscale `timescale 1ns/1ns

`tscale
module mymodule (...);
  ...
```

Example 15 - Global maroc definition of a timescale macro (not recommended)

These well-meaning engineers typically hope to control simulation efficiency by changing a global **`timescale** definition to potentially modify both the time_units and time_precisions of every model and enhance simulator performance.

Globally changing the time_units of every **`timescale** in a design can adversely impact the integrity of an entire design. Any design that includes **#**delays relies on the accuracy of the specified time_units in the **`timescale** directive. In Example 16, the model requires that the time_units of the **`timescale** be in units of **100ps**. Changing the time_units to **1ns** changes the delay from 160ps to 1.6ns, introducing an error into the model.

```
`timescale 100ps/10ps
module tribuf2001 #(parameter SIZE=8)
  (output [SIZE-1:0] y,
   input  [SIZE-1:0] a,
   input             en_n);

  assign #1.6 y = en_n ? {SIZE{1'bz}}:a;
endmodule
```

Example 16 - Module with 100ps time_units

Since the time_precision must always be equal to or smaller than the time_unit in a **`timescale** directive, additional guidelines should probably be followed if a global **`timescale** strategy is being employed:

Guideline: Make all time_units of user defined **`timescales** equal to **1ns** or larger.

Reason: if a smaller time_unit is used in any model, globally changing all time_precisions to **1ns** will break an existing design.

Note: If a vendor model is included in the simulation and if the vendor used a very small time_precision in the their model, the entire simulation will slow down and very little will have been accomplished by globally changing the time_precisions of the user models.

To enhance simulator performance, using a unit-delay simulation mode or using cycle based simulators are better options than macro-generating all of the **`timescales** in a design.

16. Conclusions

Macro definitions should be used to define system-global constants, such as a user-friendly set of names for PCI commands or global clock cycle definitions.

Each time a new macro definition is made, that macro name cannot be safely used elsewhere in the design (name-space pollution). As more and more modules are compiled into large system simulations, the likelihood of macro-name collision increases. The practice of making macro definitions for constants such as port or data sizes and state names is an ill-advised practice.

Macro definitions using the **`define** compiler directive should not be used to define constants that can be better localized to individual modules.

Verilog **parameters** are intended to represent constants that are local to a module. A **parameter** has the added benefit that each different instance of the module can have different values for the **parameters** in each module.

The following is a summary of important guidelines outlined in this paper:

Guideline: do not use **defparams** in any Verilog designs.

Guideline: require all passing of parameters to be done using the new Verilog-2001 named parameter redefinition technique.

Guideline: only use macro definitions for identifiers that clearly require global definition of an identifier that will not be modified elsewhere in the design.

Guideline: where possible, place all macro definitions into one "**definitions.vh**" file and read the file first when compiling the design.

Alternate Guideline: place all macro definitions in the top-level testbench module and read this module first when compiling the design.

Guideline: do not use macro definitions to define constants that are local to a module.

Guideline: make clock cycle definitions using the **`define** compiler directive.

Guideline: place the clock cycle definitions in the "**definitions.vh**" file or in the top-level testbench.

Guideline: do not make state assignments using **`define** macro definitions for state names.

Guideline: Make state assignments using **parameters** with symbolic state names.

Guideline: To improve simulation efficiency, make all time_units of user defined **`timescales** equal to **1ns** or larger.

In his book Writing Testbenches, Functional Verification of HDL Models, Bergeron claims that VHDL and Verilog both have the same area under the learning curve[8]. Due to the misinformation that has been spread through numerous Verilog books and training courses, I am afraid Bergeron may be right. When Verilog is taught correctly, I believe the area under the Verilog learning

curve is much smaller (and Verilog simulations run much faster).

"Long live named parameter redefinition!"

"Death to **`defparams`**!"

17. References

[1] Accellera - SystemVerilog 3.0 Accellera's Extensions to Verilog/Draft 6. www.accellera.org (not publicly available at this time).

[2] Clifford E. Cummings, "State Machine Coding Styles for Synthesis," *SNUG'98 (Synopsys Users Group San Jose, CA, 1998) Proceedings*, March 1998. Also available at www.sunburst-design.com/papers

[3] Clifford E. Cummings, "Verilog-2001 Behavioral and Synthesis Enhancements," *Delivered at HDLCON-2001 but missed publication in the Proceedings*, March 2001. Available at www.sunburst-design.com/papers

[4] Donald Thomas, and Philip Moorby, The Verilog Hardware Description Language, Fourth Edition, Kluwer Academic Publishers, 1998, pg. 142. (re-ordering the parameter redefinition list)

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

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

[7] IEEE P1364.1/D2.1 Draft Standard for Verilog Register Transfer Level Synthesis, http://www.eda.org/vlog-synth/drafts.html

[8] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. xxi. (Verilog learning curve)

[9] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 265. (why people use defparam)

[10] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 267. (defparam file)

[11] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 340. (avoid `define *and* `undef all `defines)

[12] Janick Bergeron, Writing Testbenches, Functional Verification of HDL Models, Kluwer Academic Publishers, 2000, pg. 341. (parameter CYCLE)

[13] Lionel Bening, and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001, pg. 146. (recommendation to use `define for FSM state names)

[14] Lionel Bening, and Harry Foster, Principles of Verifiable RTL Design, Second Edition, Kluwer Academic Publishers, 2001, pg. 147. (tool implementation of parameters are more difficult than `define)

[15] Michael Keating, and Pierre Bricaud, Reuse Methodology Manual, Second Edition, Kluwer Academic Publishers, 1999, pg. 110. (recommendation to use `define for state names)

[16] Michael Keating, and Pierre Bricaud, Reuse Methodology Manual, Second Edition, Kluwer Academic Publishers, 1999, pg. 112. (example state machine design uses parameters)

[17] Pran Kurup, and Taher Abbasi, Logic Synthesis Using Synopsys, Second Edition, Kluwer Academic Publishers, 1997 (book not recommended!)

Revision 1.2 (May 2002) - What Changed?

The text before Example 16 incorrectly stated that "Changing the time_units to **`1ns`** changes the delay from 1.6ns to 16ns, introducing an error into the model." The corrected text reads, "Changing the time_units to **`1ns`** changes the delay from 160ps to 1.6ns, introducing an error into the model."

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

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

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

E-mail Address: cliffc@sunburst-design.com

This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of April 19th, 2002)

VERILOG CODING STYLES FOR IMPROVED SIMULATION EFFICIENCY

Clifford E. Cummings
cliffc@sunburst-design.com / www.sunburst-design.com

Sunburst Design, Inc.
14314 SW Allen Blvd.
PMB 501
Beaverton, OR 97005

INTERNATIONAL CADENCE USER GROUP CONFERENCE
OCTOBER 5-9, 1997
SAN DIEGO, CALIFORNIA

Verilog Coding Styles for Improved Simulation Efficiency

Clifford E. Cummings
Sunburst Design, Inc.
14314 SW Allen Blvd., PMB 501, Beaverton, OR 97007
Phone: 503-641-8446 Email: cliffc@sunburst-design.com.com

Abstract

This paper details different coding styles and their impact on Verilog-XL simulation efficiency.

1. Introduction

What are some of the more optimal ways to code Verilog models and testbenches to shorten simulation times? This paper is a collection of interesting coding style comparisons that have been run on Verilog-XL.

2. Verilog Efficiency Testing

The intent of this paper is to help identify which Verilog coding styles are more efficient than others; thereby, increasing design and simulation efficiency.

All of the Verilog benchmarks in this paper are intended to show the efficiency of how specific coding styles run on the same simulator. This paper is not intended to be a benchmark comparison between different simulators.

All of the benchmarks were run on a Sparc 5, running Solaris 2.5, with 32MB of memory and 500MB of swap space. Verilog-XL, version 2.21, compiled with Undertow version 5.3.3, was used for these benchmarks.

Note: over time, measured efficiency will likely change as newer versions of the same simulator are introduced and as simulators become more efficient.

3. Case Statements Vs. Large If/Else-If Structures

Question: Is there a simulation efficiency difference in coding large case statements as equivalent if/else-if statements?

The testcase for this benchmark is a synthesizable 8-to-1 multiplexer written as both a case statement and as a large if/else-if construct. Figure 1 shows the code for the case-statement multiplexer, Figure 2 shows the code for the equivalent if/else-if-statement multiplexer.

3.1 Case Vs. If Efficiency Summary

The results in Table 1 show that, using Verilog-XL, this 8-item case structure took about the same amount of memory to implement as the if/else-if structure but the case statement was about 6% faster.

```
module CaseMux8 (y, i, sel);
    output       y;
    input  [7:0] i;
    input  [2:0] sel;

    reg          y;
    wire   [7:0] i;
    wire   [2:0] sel;

    always @(i or sel)
      case (sel)
        3'd0: y = i[0];
        3'd1: y = i[1];
        3'd2: y = i[2];
        3'd3: y = i[3];
        3'd4: y = i[4];
        3'd5: y = i[5];
        3'd6: y = i[6];
        3'd7: y = i[7];
      endcase
  endmodule
```

Figure 1

```
module IfMux8 (y, i, sel);
    output       y;
    input  [7:0] i;
    input  [2:0] sel;

    reg          y;
    wire   [7:0] i;
    wire   [2:0] sel;

    always @(i or sel)
      if      (sel == 3'd0) y = i[0];
      else if (sel == 3'd1) y = i[1];
      else if (sel == 3'd2) y = i[2];
      else if (sel == 3'd3) y = i[3];
      else if (sel == 3'd4) y = i[4];
      else if (sel == 3'd5) y = i[5];
      else if (sel == 3'd6) y = i[6];
      else if (sel == 3'd7) y = i[7];
  endmodule
```

Figure 2

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
CaseMux8	92252	100.00%	1209.3	100.00%
IfMux8	92912	100.72%	1282.0	106.01%

Table 1

4. Begin-End Statements

Question: Is there a simulation efficiency difference when extra begin-end pairs are added to Verilog models?

The testcase for this benchmark is a synthesizable D flip-flop with asynchronous reset. In Figure 3 the synthesizable flip-flop was written with no begin-end statements in the always block (they are not needed for this model). In Figure 4, the same flip-flop was written with three unnecessary begin-end statements. 1000 flip-flops were then instantiated into a testbench and simulated.

```
// Removed unneeded begin-end pairs
module dff (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

always @(posedge clk or posedge rst)
      if (rst == 1) q = 0;
      else          q = d;

endmodule
```

Figure 3

4.1 Begin-End Efficiency Summary

The results in Table 2 show that, using Verilog-XL, the flip-flop with three extra begin-end statements took about 6% more memory to implement and about 6% more time to simulate as the flip-flop with no begin-end statements.

```
// Includes unneeded begin-end pairs
module dff (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

    always @(posedge clk or posedge rst)
    begin
      if (rst == 1) begin
        q = 0;
      end
      else begin
        q = d;
      end
    end

endmodule
```

Figure 4

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
nobegin	4677676	100%	9403.2	100%
begin	4953264	105.89%	9960.4	105.93%

Table 2

5. `define Vs. Parameters

Question: What simulation efficiency difference is there between using `define macros and parameters? How is parameter efficiency affected by redefining parameters using parameter redefinition in the model instantiation and by using defparam statements?
The testcases for this benchmark were models with nine parameter delays (Figure 5) and nine `define-macro delays (Figure 6).

Testbenches were created with defined-macros (Figure 7), defparam statements (Figure 8) and #-parameter redefinition (Figure 9).

5.1 Define & Parameter Efficiency Summary

The results in Table 3 show that, parameter redefinition and defparams occupy about 25% - 35% more memory than do `define statements.

```
module paramchk (y, i, en);
  output [9:0] y;
  input        i, en;
  parameter Tr_min = 2;
  parameter Tr_typ = 5;
  parameter Tr_max = 8;
  parameter Tf_min = 1;
  parameter Tf_typ = 4;
  parameter Tf_max = 7;
  parameter Tz_min = 3;
  parameter Tz_typ = 6;
  parameter Tz_max = 9;

  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b9 (y[9],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b8 (y[8],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b7 (y[7],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b6 (y[6],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b5 (y[5],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b4 (y[4],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b3 (y[3],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b2 (y[2],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b1 (y[1],i,en);
  bufif1 #(Tr_min:Tr_typ:Tr_max,Tf_min:Tf_typ:Tf_max,Tz_min:Tz_typ:Tz_max) b0 (y[0],i,en);
endmodule
```

Figure 5

```
module definechk (y, i, en);
  output [9:0] y;
  input        i, en;

  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b9 (y[9], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b8 (y[8], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b7 (y[7], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b6 (y[6], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b5 (y[5], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b4 (y[4], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b3 (y[3], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b2 (y[2], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b1 (y[1], i, en);
  bufif1 #(`Tr_min:`Tr_typ:`Tr_max,`Tf_min:`Tf_typ:`Tf_max,`Tz_min:`Tz_typ:`Tz_max) b0 (y[0], i, en);
endmodule
```

Figure 6

```
`define Tr_min 2
`define Tr_typ 5
`define Tr_max 8
`define Tf_min 1
`define Tf_typ 4
`define Tf_max 7
`define Tz_min 3
`define Tz_typ 6
`define Tz_max 9

`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_define;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en;
  reg        clk;

  definechk i9 (y9, i, en);
  definechk i8 (y8, i, en);
  definechk i7 (y7, i, en);
  definechk i6 (y6, i, en);
  definechk i5 (y5, i, en);
  definechk i4 (y4, i, en);
  definechk i3 (y3, i, en);
  definechk i2 (y2, i, en);
  definechk i1 (y1, i, en);
  definechk i0 (y0, i, en);

  initial begin
    clk = 0;
    forever #(`cycle/2) clk = ~clk;
  end

  initial begin
    i = 0; en = 1;
    repeat (`CNT) begin
      @(negedge clk) i = ~i;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
  `ifdef RUN
    @(negedge clk) $finish(2);
  `else
    @(negedge clk) $stop(2);
`endif
  end
endmodule
```

Figure 7

```
`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_defparam;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en, clk;

  paramchk i9 (y9, i, en);
  defparam i9.Tr_min=0;  defparam i9.Tr_typ=1;  defparam i9.Tr_max=2;
  defparam i9.Tf_min=3;  defparam i9.Tf_typ=4;  defparam i9.Tf_max=5;
  defparam i9.Tz_min=6;  defparam i9.Tz_typ=7;  defparam i9.Tz_max=8;
  paramchk i8 (y8, i, en);
  defparam i8.Tr_min=0;  defparam i8.Tr_typ=1;  defparam i8.Tr_max=2;
  defparam i8.Tf_min=3;  defparam i8.Tf_typ=4;  defparam i8.Tf_max=5;
  defparam i8.Tz_min=6;  defparam i8.Tz_typ=7;  defparam i8.Tz_max=8;
  paramchk i7 (y7, i, en);
  defparam i7.Tr_min=0;  defparam i7.Tr_typ=1;  defparam i7.Tr_max=2;
  defparam i7.Tf_min=3;  defparam i7.Tf_typ=4;  defparam i7.Tf_max=5;
  defparam i7.Tz_min=6;  defparam i7.Tz_typ=7;  defparam i7.Tz_max=8;
  paramchk i6 (y6, i, en);
  defparam i6.Tr_min=0;  defparam i6.Tr_typ=1;  defparam i6.Tr_max=2;
  defparam i6.Tf_min=3;  defparam i6.Tf_typ=4;  defparam i6.Tf_max=5;
  defparam i6.Tz_min=6;  defparam i6.Tz_typ=7;  defparam i6.Tz_max=8;
  paramchk i5 (y5, i, en);
  defparam i5.Tr_min=0;  defparam i5.Tr_typ=1;  defparam i5.Tr_max=2;
  defparam i5.Tf_min=3;  defparam i5.Tf_typ=4;  defparam i5.Tf_max=5;
  defparam i5.Tz_min=6;  defparam i5.Tz_typ=7;  defparam i5.Tz_max=8;
  paramchk i4 (y4, i, en);
  defparam i4.Tr_min=0;  defparam i4.Tr_typ=1;  defparam i4.Tr_max=2;
  defparam i4.Tf_min=3;  defparam i4.Tf_typ=4;  defparam i4.Tf_max=5;
  defparam i4.Tz_min=6;  defparam i4.Tz_typ=7;  defparam i4.Tz_max=8;
  paramchk i3 (y3, i, en);
  defparam i3.Tr_min=0;  defparam i3.Tr_typ=1;  defparam i3.Tr_max=2;
  defparam i3.Tf_min=3;  defparam i3.Tf_typ=4;  defparam i3.Tf_max=5;
  defparam i3.Tz_min=6;  defparam i3.Tz_typ=7;  defparam i3.Tz_max=8;
  paramchk i2 (y2, i, en);
  defparam i2.Tr_min=0;  defparam i2.Tr_typ=1;  defparam i2.Tr_max=2;
  defparam i2.Tf_min=3;  defparam i2.Tf_typ=4;  defparam i2.Tf_max=5;
  defparam i2.Tz_min=6;  defparam i2.Tz_typ=7;  defparam i2.Tz_max=8;
  paramchk i1 (y1, i, en);
  defparam i1.Tr_min=0;  defparam i1.Tr_typ=1;  defparam i1.Tr_max=2;
  defparam i1.Tf_min=3;  defparam i1.Tf_typ=4;  defparam i1.Tf_max=5;
  defparam i1.Tz_min=6;  defparam i1.Tz_typ=7;  defparam i1.Tz_max=8;
  paramchk i0 (y0, i, en);
  defparam i0.Tr_min=0;  defparam i0.Tr_typ=1;  defparam i0.Tr_max=2;
  defparam i0.Tf_min=3;  defparam i0.Tf_typ=4;  defparam i0.Tf_max=5;
  defparam i0.Tz_min=6;  defparam i0.Tz_typ=7;  defparam i0.Tz_max=8;

  initial begin
    clk = 0;
    forever #(`cycle/2) clk = ~clk;
  end

  initial begin
    i = 0; en = 1;
    repeat (`CNT) @(negedge clk) i  = ~i;
                  @(negedge clk) i  = ~i;
    repeat (`CNT) @(negedge clk) en = ~en;
                  @(negedge clk) i  = ~i;
    repeat (`CNT) @(negedge clk) en = ~en;
  `ifdef RUN  @(negedge clk) $finish(2);
  `else       @(negedge clk) $stop(2);
  `endif
  end
endmodule
```

Figure 8

```
`define CNT 1000000
`define cycle 20
`timescale 1ns / 1ns

module tb_param;
  wire [9:0] y9, y8, y7, y6, y5, y4, y3, y2, y1, y0;
  reg        i, en;
  reg        clk;

  paramchk #(0,1,2,3,4,5,6,7,8) i9 (y9, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i8 (y8, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i7 (y7, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i6 (y6, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i5 (y5, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i4 (y4, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i3 (y3, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i2 (y2, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i1 (y1, i, en);
  paramchk #(0,1,2,3,4,5,6,7,8) i0 (y0, i, en);

  initial begin
    clk = 0;
    forever #(`cycle/2) clk = ~clk;
  end

  initial begin
    i = 0; en = 1;
    repeat (`CNT) begin
      @(negedge clk) i = ~i;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
    @(negedge clk) i = ~i;
    repeat (`CNT) begin
      @(negedge clk) en = ~en;
    end
  `ifdef RUN
    @(negedge clk) $finish(2);
  `else
    @(negedge clk) $stop(2);
  `endif
  end
endmodule
```

Figure 9

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
define	162448	100.00%	1436.9	100.00%
paramredef	201236	123.88%	1427.2	99.32%
defparam	220568	135.78%	1420.0	98.82%

Table 3

6. Grouping of Assignments Within Always Blocks

Question: Is there a penalty for splitting assignments into multiple always blocks as opposed to grouping the same assignments into fewer always blocks?

The testcase for this benchmark is a conditionally complied set of four assignments in four separate always blocks, or the same four assignments in a single always block (Figure 10).

6.1 Always Block Grouping Efficiency Summary

The results in Table 4 show that, four assignments in four separate always blocks is about 34% slower than placing the same assignments in a single always block

```
`define ICNT 10000000
`define cycle 100
`timescale 1ns / 100ps

module AlwaysGroup;
  reg       clk;
  reg [7:0] a, b, c, d, e;

  initial begin
    clk = 0;
    forever #(`cycle) clk = ~clk;
  end

  initial begin
    a = 8'haa;
    forever @(negedge clk) a = ~ a;
  end

  initial begin
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN @(posedge clk) $finish(2);
  `else      @(posedge clk) $stop(2);
  `endif
  end

  `ifdef GROUP4 // Group of four always blocks
    always @(posedge clk) begin
      b <= a;
    end
    always @(posedge clk) begin
      c <= b;
    end
    always @(posedge clk) begin
      d <= c;
    end
    always @(posedge clk) begin
      e <= d;
    end

  `else // Four assignments grouped into a
        // single always block
    always @(posedge clk) begin
      b <= a;
      c <= b;
      d <= c;
      e <= d;
    end
  `endif

endmodule
```

Figure 10

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
group1	85712	100.00%	1196.6	100.00%
group4	87544	102.14%	1607.2	134.31%

Table 4

7. Port Connections

Question: Is there a penalty for passing data over module ports as opposed to passing data by hierarchical reference?

The idea for this benchmark came from a paper presented by Martin Gravenstein[1] of Ford Microelectronics at the 1994 International Verilog Conference.

The testcase for this benchmark is a pair of flip-flops. The first flip-flop has no ports and testbench communication with this model was conducted by hierarchical reference. The second flip-flop is a model with normal port communication with a testbench (Figure 11).

7.1 Ports/No Ports Efficiency Summary

The results in Table 5 show that, communicating with a four-port model as opposed to referencing the ports hierarchically required about 46% more simulation time

Model port usage and communication is still recommended; however, passing monitor data over ports would be simulation-time expensive. It is better to reference monitored state and bus data hierarchically

These results also suggest that models with extra levels of hierarchy will significantly slow down a simulation

```
`ifdef NOPORTS
  module PortModels;
    reg    [15:0] q;
    reg    [15:0] d;
    reg           clk, rstN;

    always @(posedge clk or negedge rstN)
      if (rstN == 0) q <= 0;
      else           q <= d;

  endmodule
`else
  module PortModels (q, d, clk, rstN);
    output [15:0] q;
    input  [15:0] d;
    input         clk, rstN;
    reg    [15:0] q;

    always @(posedge clk or negedge rstN)
      if (rstN == 0) q <= 0;
      else           q <= d;

  endmodule
`endif
```

Figure 11

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
NoPorts	90948	100.00%	1343.7	100.00%
Ports	89436	98.34%	1967.4	146.42%

Table 5

8. `timescale Efficiencies

Question: Are there any simulator performance penalties for using a higher precision `timescale during simulation?

The testcase for this benchmark is a buffer and inverter with propagation delays that are simulated with a variety of `timescales (Figure 12).

8.1 `timescale Efficiency Summary

The results in Table 6 show that, using a `timescale of 1ns/1ps requires about 156% more memory and about 99% more time to simulate than the same model using a timescale of 1ns/1ns.

9. Displaying $time Values

Question: Are there any simulator performance penalties for displaying different $time values to STDOUT during simulation? Do the $time display format specifiers affect simulation performance?

The idea for this benchmark came from a paper presented by Jay Lawrence[2] of Cadence at the 1995 ICU. The testcase for this benchmark is a set of different display commands of time values (Figure 13).

9.1 Display $time Efficiency Summary

```
`define ICNT 10000000
`define cycle 10

`ifdef Time_1ns      `timescale 1ns / 1ns
`endif
`ifdef Time_100ps    `timescale 1ns / 100ps
`endif
`ifdef Time_10ps     `timescale 1ns / 10ps
`endif
`ifdef Time_1ps      `timescale 1ns / 1ps
`endif

module TimeModel;
  reg         i;
  wire [1:2]  y;

  initial begin
    i = 0;
    forever #(`cycle) i = ~i;
  end

  initial begin
    repeat(`ICNT) @(posedge i);
  `ifdef RUN  @(posedge i) $finish(2);
  `else       @(posedge i) $stop(2);
  `endif
  end

  buf #(2.201, 3.667) i1 (y[1], i);
  not #(4.633, 7.499) i2 (y[2], i);

endmodule
```

Figure 12

The results in Table 7 show that, needless display of $time values is very costly in simulation time

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
Time_1ns	83916	100.00%	1395.0	100.00%
Time_100ps	83920	100.00%	1459.4	104.62%
Time_10ps	92620	110.37%	1718.1	123.16%
Time_1ps	214476	255.58%	2777.8	199.13%

Table 6

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
No_Display	84784	100.00%	906.6	100.00%
One_Display	84924	100.17%	1297.3	143.10%
Two_Display	85060	100.33%	1646.1	181.57%
Two_Display0	85064	100.33%	1624.5	179.19%
Two_Format0	85024	100.28%	1348.4	148.73%
Two_RtimeFormat0	85832	101.24%	1328.1	146.49%

Table 7

```
`define hcycle 12.5
`define ICNT 1000000
`timescale 1ns / 100ps

module DisplayTime;
  reg     clk;
  time    rClkTime,  fClkTime;
  real   rRealTime, fRealTime;

  initial begin
    clk = 0;
    forever #(`hcycle) clk = ~clk;
  end

  initial begin
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN  @(posedge clk) $finish(2);
  `else       @(posedge clk) $stop(2);
  `endif
  end

  `ifdef NoDisplay // display with no time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Negedge Clk");
    end
  `endif

  `ifdef OneDisplay // display with one time value
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Negedge Clk at %d", fClkTime);
    end
  `endif

  `ifdef TwoDisplay // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %d - Negedge Clk at %d", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoDisplay0 // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %0d - Negedge Clk at %0d", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoFormat0 // display with two time values
    always @(negedge clk) begin
      fClkTime = $time;
      $display ("Posedge Clk at %0t - Negedge Clk at %0t", rClkTime, fClkTime);
    end
  `endif

  `ifdef TwoRtimeFormat0 // display with two time values
    initial $timeformat(-9,2,"ns",15);

    always @(negedge clk) begin
      fRealTime = $realtime;
      $display ("Posedge Clk at %0t - Negedge Clk at %0t", rRealTime,fRealTime);
    end
  `endif

  `ifdef TwoRtimeFormat0  always @(posedge clk) rRealTime = $realtime;
  `else                   always @(posedge clk) rClkTime = $time;
  `endif

endmodule
```

Figure 13

10. Clock Oscillators

Question: Are there any significant simulator performance advantages to implementing a clock oscillator using an always block, forever loop or Verilog gate primitives?

The testcase for this benchmark are three conditionally compiled clock oscillator implementations, clocking a flip-flop model (Figure 14).

10.1 Clock Oscillator Efficiency Summary

The results in Table 8 show that, gate clock oscillators were about %85% slower than behavioral clock oscillators.

11. Efficiency - Final Word

Simulation efficiency should not be the only Verilog coding criteria.

Code readability, simulation accuracy and displaying timely simulation and diagnostic information might actually increase design productivity. However, reckless use of inefficient coding styles when a more efficient alternative exists is detrimental to simulation productivity.

Again, these benchmarks were only run on Verilog-XL. Mileage may vary on other simulators.

12. References

[1] M. Gravenstein, "Modeling Techniques to Support System Level Simulation and a Top-Down Development Methodology," *International Verilog HDL Conference Proceedings 1994*, pp 43-50

[2] J. Lawrence & C. Ussery, "INCA: A Next-Generation Architecture for Simulation," *International Cadence Users Group Conference Proceedings 1995*, pp 105-109


```
`define ICNT 100_000
`define cycle 100
`timescale 1ns / 1ns

module Clocks;
  reg d, rstN;

  `ifdef ALWAYS
    reg clk;     // driven by a procedural block
    initial clk = 0;
    always        // free running behave clk #1
      #(`cycle/2) clk = ~clk;
  `endif

  `ifdef FOREVER
    reg clk;     // driven by a procedural block
    initial begin
      clk = 0;
      forever #(`cycle/2) clk = ~clk;
    end
  `endif

  `ifdef GATE  // free running clk #3 (gate)
    reg  start;
    wire clk;    // driven by a gate
    initial begin
      start = 0; #(`cycle/2) start = 1;
    end
    nand #(`cycle/2) (clk, clk, start);
  `endif

  dff d1 (q, d, clk, rstN);

  initial begin
    rstN = 0; d = 1;
    @(negedge clk) rstN = 1;
    repeat(`ICNT) @(posedge clk);
  `ifdef RUN   @(posedge clk) $finish(2);
  `else        @(posedge clk) $stop(2);
  `endif
  end

// Veritools Undertow-dumpfile option
`ifdef UT
  initial begin
    $dumpfile(dump.ut);  $vtDumpvars;
  end
`endif
endmodule
```

Figure 14

Sparc5/32MB RAM 491MB Swap Solaris version 2.5 Verilog-XL 2.2.1 - Undertow 5.3.3	Data Structure (bytes of memory)	Memory Usage Percentages	Simulation CPU Time (in seconds)	CPU Simulation Percentages
Always	88104	100.00%	1359.2	100.00%
Forever	88168	100.07%	1371.0	100.87%
Gate	88588	100.55%	2562.7	188.54%
AlwaysUT	90368	102.57%	2446.5	180.00%
ForeverUT	90432	102.64%	2481.5	182.57%
GateUT	90696	102.94%	3448.8	253.74%

Table 8

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

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

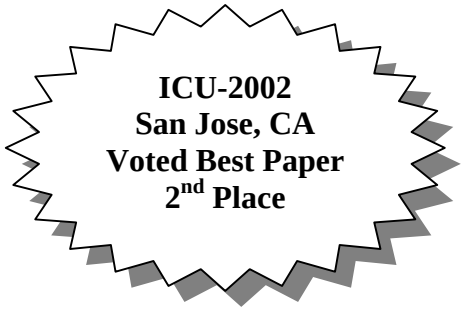
Mr. Cummings holds a BSEE from Brigham Young University and an MSEE from Oregon State University.

E-mail Address: cliffc@sunburst-design.com

This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of January 4th, 2002)

The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates



Clifford E. Cummings

Sunburst Design, Inc.
503-641-8446
cliffc@sunburst-design.com

INTERNATIONAL CADENCE USERGROUP CONFERENCE
September 16-18, 2002
San Jose, California

Abstract

This paper details proven RTL coding styles for efficient and synthesizable Finite State Machine (FSM) design using IEEE-compliant Verilog simulators. Important techniques related to one and two always block styles to code FSMs with combinational outputs are given to show why using a two always block style is preferred. An efficient Verilog-unique onehot FSM coding style is also shown. Reasons and techniques for registering FSM outputs are also detailed. Myths surrounding erroneous state encodings, full-case and parallel-case usage are also discussed. Compliance and enhancements related to the IEEE 1364-2001 Verilog Standard, the proposed IEEE 1364.1 Verilog Synthesis Interoperability Standard and the proposed Accellera SystemVerilog Standard are also discussed.

1. Introduction

FSM is an abbreviation for *Finite State Machine*.

There are many ways to code FSMs including many very poor ways to code FSMs. This paper will examine some of the most commonly used FSM coding styles, their advantages and disadvantages, and offer guidelines for doing efficient coding, simulation and synthesis of FSM designs.

This paper will also detail Accellera SystemVerilog enhancements that will facilitate and enhance future Verilog FSM designs.

In this paper, multiple references are made to combinational always blocks and sequential always blocks. Combinational always blocks are always blocks that are used to code combinational logic functionality and are strictly coded using blocking assignments (see Cummings[4]). A combinational always block has a combinational sensitivity list, a sensitivity list without "posedge" or "negedge" Verilog keywords.

Sequential always blocks are always blocks that are used to code clocked or sequential logic and are always coded using nonblocking assignments (see Cummings[4]). A sequential always block has an edge-based sensitivy list.

2. Mealy and Moore FSMs

A common classification used to describe the type of an FSM is Mealy and Moore state machines[9][10].

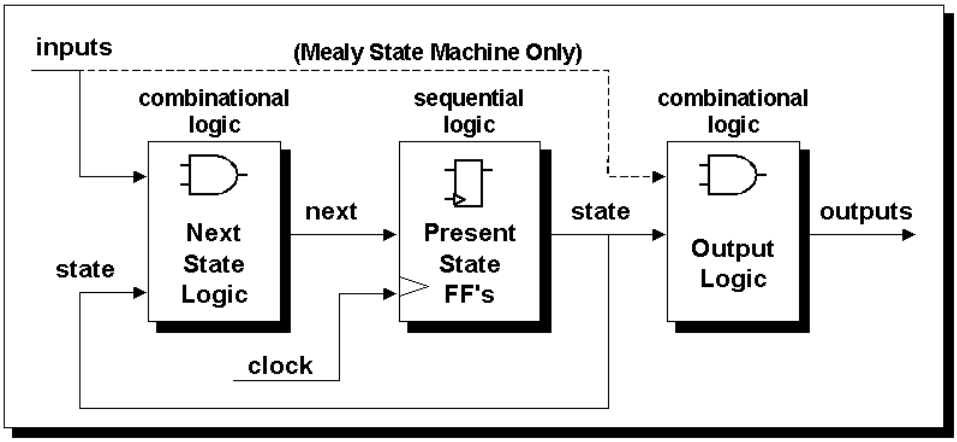


Figure 1 - Finite State Machine (FSM) block diagram

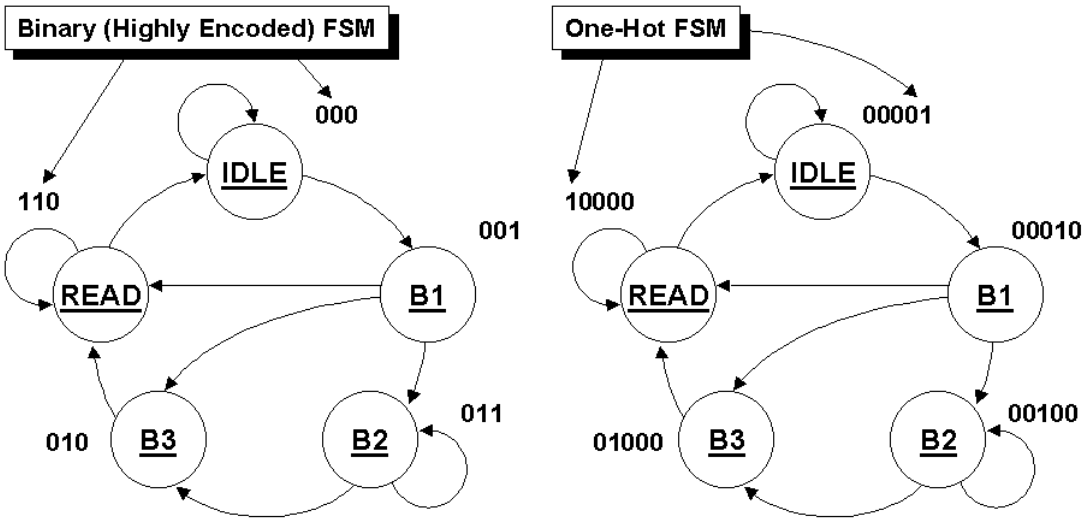
A Moore FSM is a state machine where the outputs are only a function of the present state. A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs. A block diagram for Moore and Mealy FSMs is shown Figure 1.

3. Binary Encoded or Onehot Encoded?

Common classifications used to describe the state encoding of an FSM are Binary (or highly encoded) and Onehot.

A binary-encoded FSM design only requires as many flip-flops as are needed to uniquely encode the number of states in the state machine. The actual number of flip-flops required is equal to the ceiling of the log-base-2 of the number of states in the FSM.

A onehot FSM design requires a flip-flop for each state in the design and only one flip-flop (the flip-flop representing the current or "hot" state) is set at a time in a onehot FSM design. For a state machine with 9-16 states, a binary FSM only requires 4 flip-flops while a onehot FSM requires a flip-flop for each state in the design (9-16 flip-flops).



FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks[8].

4. FSM Coding Goals

To determine what constitutes an efficient FSM coding style, we first need to identify HDL coding goals and why they are important. After the HDL coding goals have been identified, we can then quantify the capabilities of various FSM coding styles.

The author has identified the following HDL coding goals as important when doing HDL-based FSM design:

- The FSM coding style should be easily modified to change state encodings and FSM styles.
- The coding style should be compact.
- The coding style should be easy to code and understand.

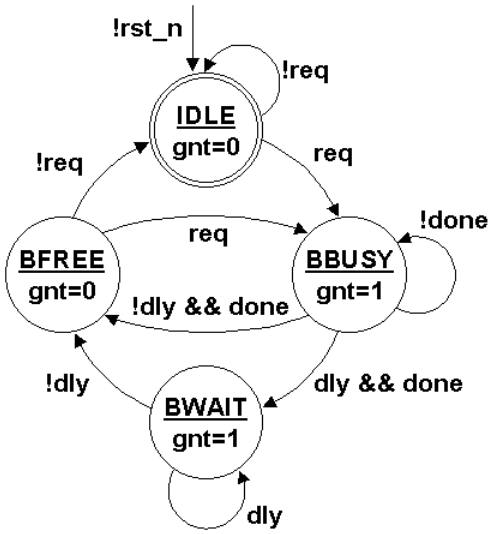
- The coding style should facilitate debugging.
- The coding style should yield efficient synthesis results.

Three different FSM designs will be examined in this paper. The first is a simple 4-state FSM design labeled fsm_cc4 with one output. The second is a 10-state FSM design labeled fsm_cc7 with only a few transition arcs and one output. The third is another 10-state FSM design labeled fsm_cc8 with multiple transition arcs and three outputs. The coding efforts to create these three designs will prove interesting.

5. Two Always Block FSM Style (Good Style)

One of the best Verilog coding styles is to code the FSM design using two always blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

```
module fsm_cc4_2
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'b00,
                  BBUSY = 2'b01,
                  BWAIT = 2'b10,
                  BFREE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    gnt  = 1'b0;
    case (state)
      IDLE :   if (req)       next = BBUSY;
               else           next = IDLE;
      BBUSY: begin
               gnt = 1'b1;
               if (!done)     next = BBUSY;
               else if ( dly) next = BWAIT;
               else           next = BFREE;
             end
      BWAIT: begin
               gnt = 1'b1;
               if (!dly)      next = BFREE;
               else           next = BWAIT;
             end
      BFREE:   if (req)       next = BBUSY;
               else           next = IDLE;
    endcase
  end
endmodule
```



Example 1 - fsm_cc4 design - two always block style - 37 lines of code

5.1 Important coding style notes:

- Parameters are used to define state encodings instead of the Verilog `define macro definition construct[3]. After parameter definitions are created, the parameters are used throughout the rest of the

design, not the state encodings. This means that if an engineer wants to experiment with different state encodings, only the parameter values need to be modified while the rest of the Verilog code remains unchanged.

- Declarations are made for **`state`** and **`next`** (next state) after the parameter assignments.
- The sequential always block is coded using nonblocking assignments.
- The combinational always block sensitivity list is sensitive to changes on the **`state`** variable and all of the inputs referenced in the combinational always block.
- Assignments within the combinational always block are made using Verilog blocking assignments.
- The combinational always block has a default **`next`** state assignment at the top of the always block (see section 5.3 for details about making default-X assignments).
- Default output assignments are made before coding the **`case`** statement (this eliminates latches and reduces the amount of code required to code the rest of the outputs in the **`case`** statement and highlights in the **`case`** statement exactly in which states the individual output(s) change).
- In the states where the output assignment is not the default value assigned at the top of the always block, the output assignment is only made once for each state.
- There is an **`if`**-statement, an **`else`**-**`if`**-statement or an **`else`** statement for each transition arc in the FSM state diagram. The number of transition arcs between states in the FSM state diagram should equal the number of **`if`**-**`else`**-type statements in the combinational always block.
- For ease of scanning and debug, all of the **`next`** assignments have been placed in a single column, as opposed to finding **`next`** assignments following the contour of the RTL code.

5.2 The unfounded fear of transitions to erroneous states

In engineering school, we were all cautioned about "what happens if you FSM gets into an erroneous state?" In general, this concern is both invalid or poorly developed.

I do not worry about most of my FSM designs going to an erroneous state any more than I worry about any other register in my design spontaneously changing value. It just does not occur!

There are exceptions, such as satellites (subject to alpha particle bombardment) or medical implants (subject to radiation and requiring extra robust design), plus other examples. In these situations, one does have to worry about FSMs going to an erroneous state, but most engineering schools fail to note that getting back to a known state is typically not good enough! Even though the FSM is now in a known state, the rest of the hardware is still expecting activity related to another state. It is possible for the design to lockup waiting for signals that will never arrive because the FSM changed states without resetting the rest of the design. At the very least, the FSM should transition to an error state that communicates to the rest of the design that resetting will occur on the next state transition, "get ready!"

5.3 Making default `next` equal all X's assignment

Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style. This default assignment is updated by next-state assignments inside the case statement. There are three types of default next-state assignments that are commonly used: (1) next is set to all X's, (2) next is set to a predetermined recovery state such as IDLE, or (3) next is just set to the value of the state register.

By making a default next state assignment of X's, pre-synthesis simulation models will cause the state machine outputs to go unknown if not all state transitions have been explicitly assigned in the case statement. This is a useful technique to debug state machine designs, plus the X's will be treated as "don't cares" by the synthesis tool.

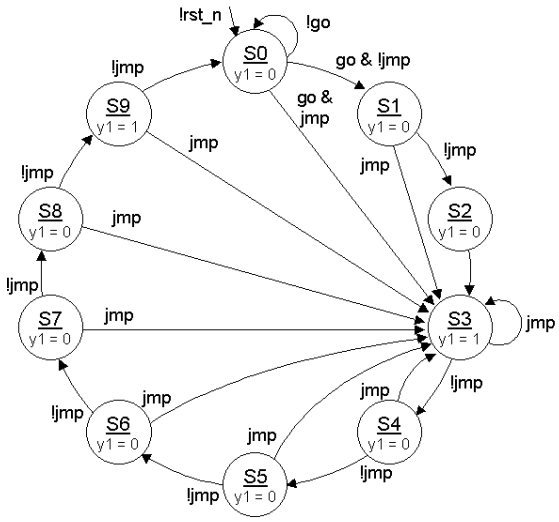
Some designs require an assignment to a known state as opposed to assigning X's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan

chain and some designs that are equivalence checked with formal verification tools. Making a default next state assignment of either IDLE or all 0's typically satisfies these design requirements and making the initial default assignment might be easier than coding all of the explicit next-state transition assignments in the case statement.

5.4 10-state simple FSM design - two always blocks

Example 2 is the fsm_cc7 design implemented with two always blocks. Using two always blocks, the fsm_cc7 design requires 50 lines of code (coding requirements are compared in a later section).

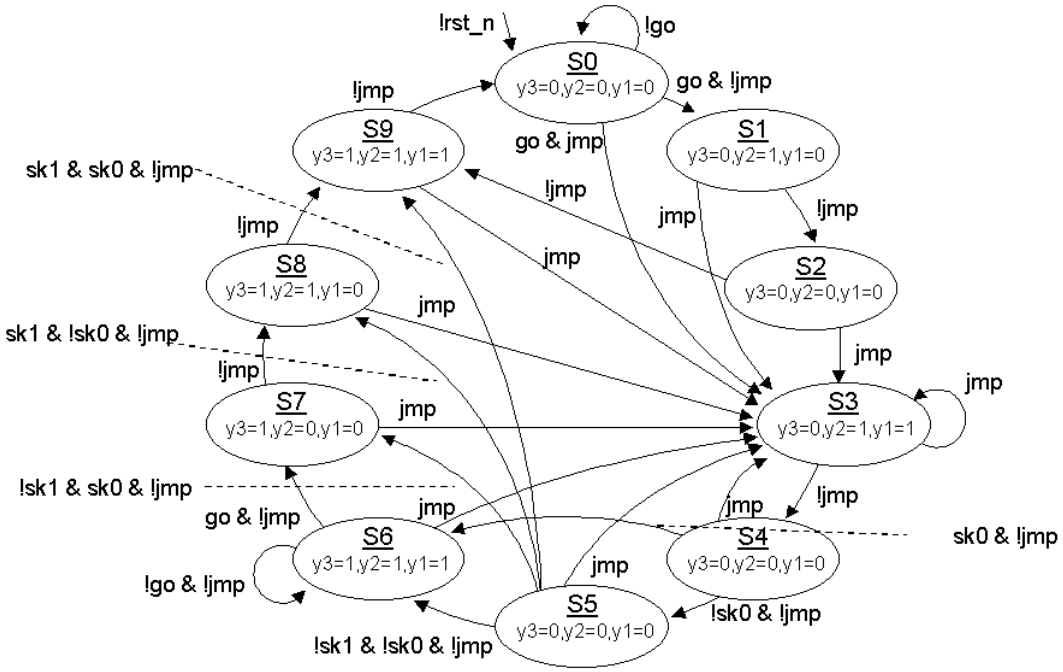
```
module fsm_cc7_2
        (output reg y1,
         input      jmp, go, clk, rst_n);

        parameter S0  = 4'b0000,
                  S1  = 4'b0001,
                  S2  = 4'b0010,
                  S3  = 4'b0011,
                  S4  = 4'b0100,
                  S5  = 4'b0101,
                  S6  = 4'b0110,
                  S7  = 4'b0111,
                  S8  = 4'b1000,
                  S9  = 4'b1001;

        reg [3:0] state, next;

        always @(posedge clk or negedge rst_n)
          if (!rst_n) state <= S0;
          else        state <= next;

        always @(state or go or jmp) begin
          next = 4'bx;
          y1 = 1'b0;
          case (state)
            S0 : if (!go)      next = S0;
                 else if (jmp) next = S3;
                 else          next = S1;
            S1 : if (jmp)      next = S3;
                 else          next = S2;
            S2 :               next = S3;
            S3 : begin y1 = 1'b1;
                   if (jmp)    next = S3;
                   else        next = S4;
                 end
            S4 : if (jmp)      next = S3;
                 else          next = S5;
            S5 : if (jmp)      next = S3;
                 else          next = S6;
            S6 : if (jmp)      next = S3;
                 else          next = S7;
            S7 : if (jmp)      next = S3;
                 else          next = S8;
            S8 : if (jmp)      next = S3;
                 else          next = S9;
            S9 : if (jmp)      next = S3;
                 else          next = S0;
          endcase
        end
      endmodule
```



Example 2 - fsm_cc7 design - two always block style - 50 lines of code

5.5 10-state moderately complex FSM design - two always blocks

Example 3 is the fsm_cc8 design implemented with two always blocks. Using two always blocks, the fsm_cc8 design requires 80 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_2
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 4'bx;
    y1 = 1'b0;
    y2 = 1'b0;
    y3 = 1'b0;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
```

```
            else                              next = S1;
       S1 : begin
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S2;
            end
       S2 : if (jmp)                          next = S3;
            else                              next = S9;
       S3 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S4;
              end
       S4 : if      (jmp)                     next = S3;
            else if (sk0 && !jmp)             next = S6;
            else                              next = S5;
       S5 : if      (jmp)                     next = S3;
            else if (!sk1 && !sk0 && !jmp)    next = S6;
            else if (!sk1 &&  sk0 && !jmp)    next = S7;
            else if ( sk1 && !sk0 && !jmp)    next = S8;
            else                              next = S9;
       S6 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if      (jmp)                   next = S3;
              else if (go && !jmp)            next = S7;
              else                            next = S6;
            end
       S7 : begin
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S8;
            end
       S8 : begin
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S9;
            end
       S9 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S0;
            end
    endcase
  end
endmodule
```

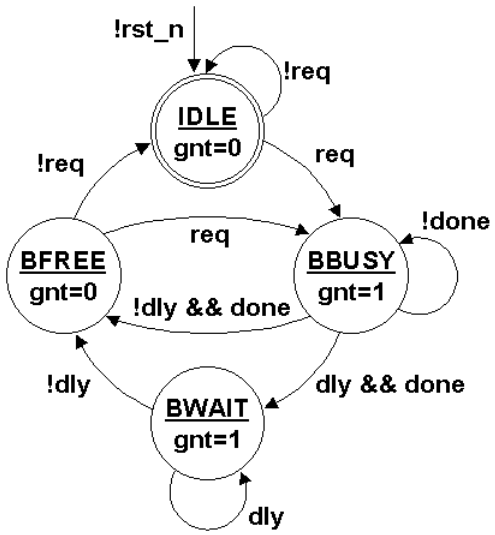
Example 3 - fsm_cc8 design - two always block style - 80 lines of code

6. One Always Block FSM Style (Avoid This Style!)

One of the most common FSM coding styles in use today is the one sequential always block FSM coding style. This coding style is very similar to coding styles that were popularized by PLD programming languages of the mid-1980s, such as ABEL. For most FSM designs, the one always block FSM coding style is more verbose, more confusing and more error prone than a comparable two always block coding style.

Reconsider the fsm_cc4 design shown in section 5.

```
module fsm_cc4_1
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'd0,
                  BBUSY = 2'd1,
                  BWAIT = 2'd2,
                  BFREE = 2'd3;

  reg [1:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state <= IDLE;
      gnt   <= 1'b0;
    end
    else begin
      state <= 2'bx;
      gnt   <= 1'b0;
      case (state)
        IDLE : if (req) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else              state <= IDLE;
        BBUSY: if (!done) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else if ( dly) begin
                                  state <= BWAIT;
                  gnt <= 1'b1;
                end
                else              state <= BFREE;
        BWAIT: if ( dly) begin
                                  state <= BWAIT;
                  gnt <= 1'b1;
                end
                else              state <= BFREE;
        BFREE: if (req) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else              state <= IDLE;
      endcase
    end
endmodule
```



Example 4 - fsm_cc4 design - one always block style - 47 lines of code

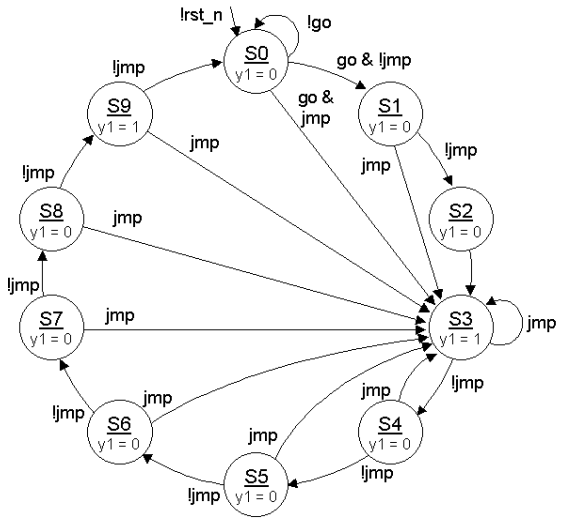
6.1 Important coding style notes:

- Parameters are used to define state encodings, the same as the two always block coding style.
- A declaration is made for **`state`**. Not for **`next`**.
- There is just one sequential always block, coded using nonblocking assignments.
- The there is still a default **`state`** assignment before the **`case`** statement, then the **`case`** statement tests the **`state`** variable. Will this be a problem? No, because the default **`state`** assignment is made with a nonblocking assignment, so the update to the **`state`** variable will happen at the end of the simulation time step.
- Default output assignments are made before coding the **`case`** statement (this reduces the amount of code required to code the rest of the outputs in the **`case`** statement).
- A **`state`** assignment must be made for each transition arc that transitions to a **`state`** where the output will be different than the default assigned value. For multiple outputs and for multiple transition arcs into a **`state`** where the outputs change, multiple **`state`** assignments will be required.
- The **`state`** assignments do not correspond to the current **`state`** of the **`case`** statement, but the **`state`** that **`case`** statement is transitioning to. *This is error prone* (but it does work if coded correctly).
- Again, for ease of scanning and debug, the all of the **`state`** assignments have been placed in a single column, as opposed to finding **`state`** assignments following the contour of the RTL code.
- All outputs will be registered (unless the outputs are placed into a separate combinational always block or assigned using continuous assignments). No asynchronous Mealy outputs can be generated from a single synchronous always block.
- Note: some misinformed engineers fear that making multiple assignments to the same variable, in the same always block, using nonblocking assignments, is undefined and can cause race conditions. This is not true. Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins! (See reference [5] for details).

6.2 10-state simple FSM design - one always blocks

Example 5 is the fsm_cc7 design implemented with one always blocks. Using one always blocks, the fsm_cc7 design requires 79 lines of code (coding requirements are compared in a later section).

```
module fsm_cc7_1
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state <= S0;
      y1 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
```

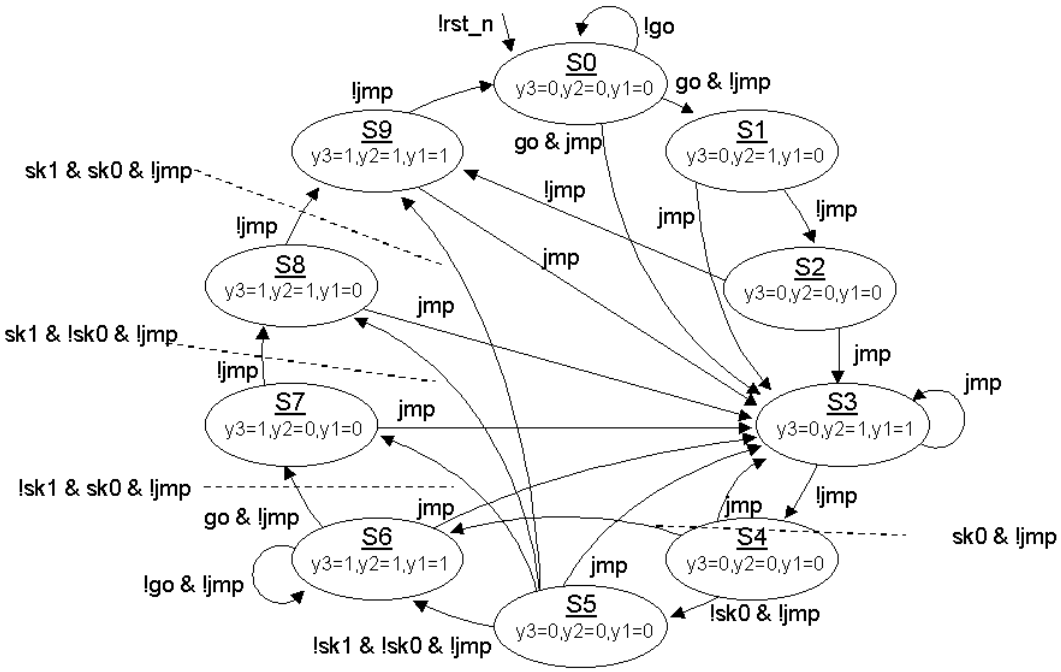


```
        state <= 4'bx;
        case (state)
          S0 : if (!go)      state <= S0;
               else if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S1;
          S1 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S2;
          S2 : begin
                 y1 <= 1'b1;
                             state <= S3;
               end
          S3 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S4;
          S4 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S5;
          S5 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S6;
          S6 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S7;
          S7 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S8;
          S8 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S9;
          S9 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S0;
        endcase
      end
    endmodule
```

Example 5 - fsm_cc7 design - one always block style - 79 lines of code

6.3 10-state moderately complex FSM design - one always blocks

Example 6 is the fsm_cc8 design implemented with one always blocks. Using one always blocks, the fsm_cc8 design requires 146 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_1
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                            state <= S0;
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
                            state <= 4'bx;
      y1 <= 1'b0;
      y2 <= 1'b0;
```

```
y3 <= 1'b0;
case (state)
  S0 : if (!go)      state <= S0;
       else if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S1;
         y2 <= 1'b1;
       end
  S1 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S2;
  S2 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S9;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
  S3 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S4;
  S4 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else          state <= S5;
  S5 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (!sk1 && !sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else if (!sk1 && sk0 && !jmp) begin
                     state <= S7;
         y3 <= 1'b1;
       end
       else if (sk1 && !sk0 && !jmp) begin
```

```
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S6 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else if (go && !jmp) begin
                        state <= S7;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S6;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S7 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S8 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S9 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else          state <= S0;
        endcase
      end
endmodule
```

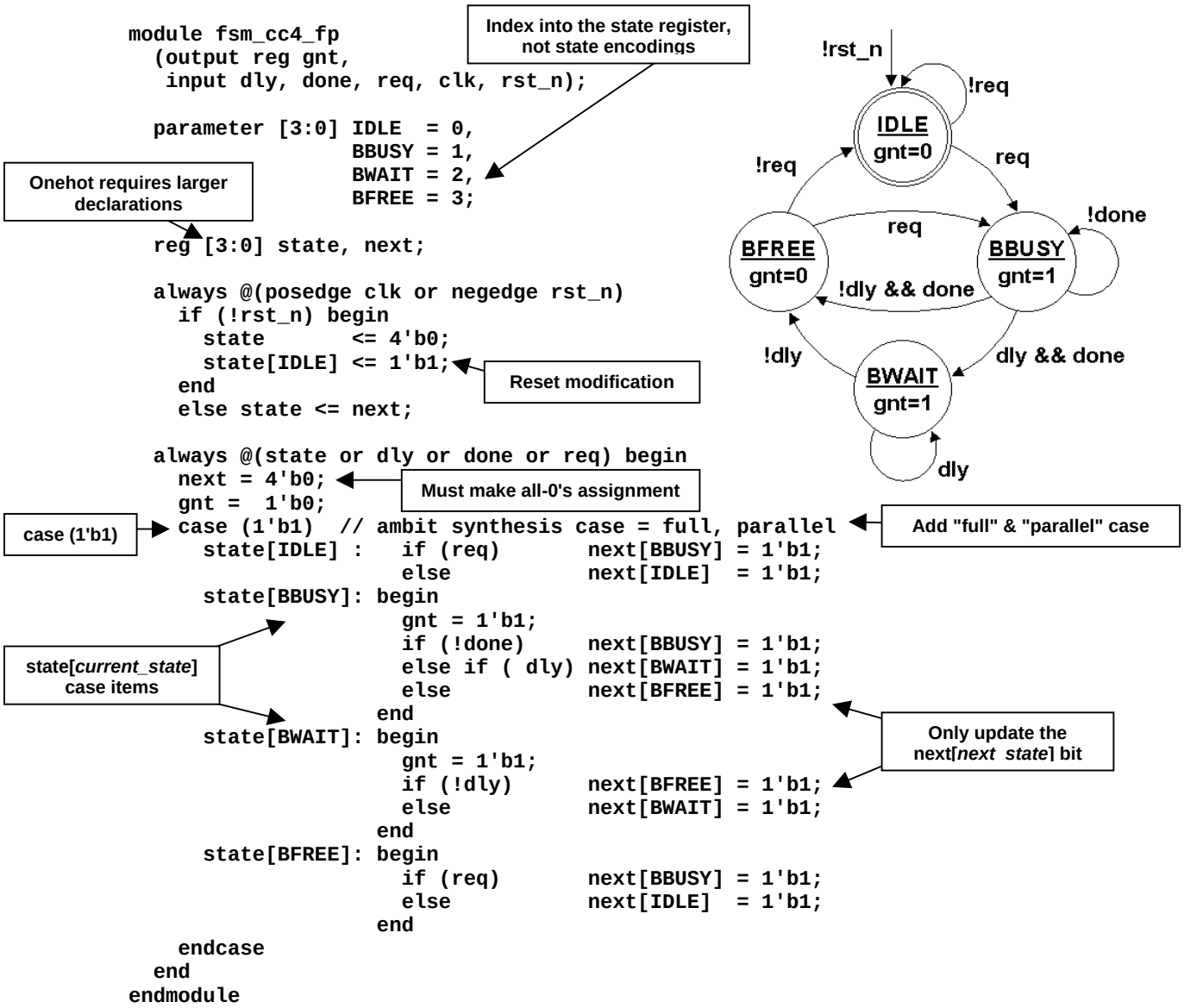
Example 6 - fsm_cc8 design - one always block style - 146 lines of code

7. Onehot FSM Coding Style (Good Style)

Efficient (small and fast) onehot state machines can be coded using an inverse case statement; a case statement where each case item is an expression that evaluates to true or false.

Reconsider the fsm_cc4 design shown in section 5. Eight coding modifications must be made to the two always block coding style of section 5 to implement the efficient onehot FSM coding style.

The key to understanding the changes is to realize that the parameters no longer represent **`state`** encodings, they now represent an *index* into the **`state`** vector, and comparisons and assignments are now being made to single bits in either the **`state`** or **`next`**-state vectors. Notice how the case statement is now doing a 1-bit comparison against the onehot state bit.

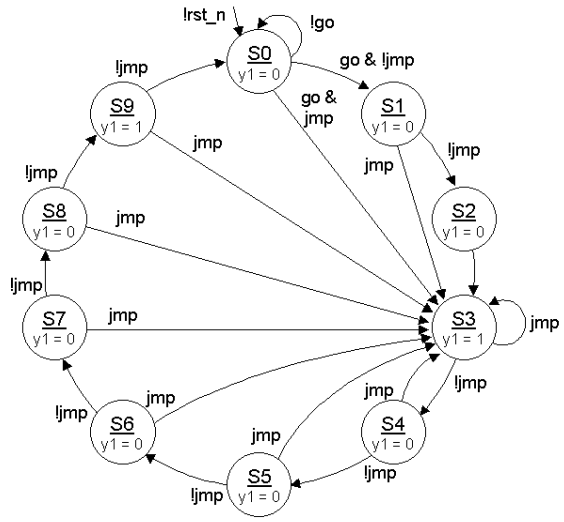



```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state       <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit synthesis case = full, parallel
      state[IDLE] :   if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)     next[BBUSY] = 1'b1;
                      else if ( dly) next[BWAIT] = 1'b1;
                      else           next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)      next[BFREE] = 1'b1;
                      else           next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Example 7 - fsm_cc4 design - case (1'b1) onehot style - 42 lines of code

7.1 10-state simple FSM design - case (1'b1) onehot coding style

Example 8 is the fsm_cc7 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc7 design requires 53 lines of code (coding requirements are compared in a later section).

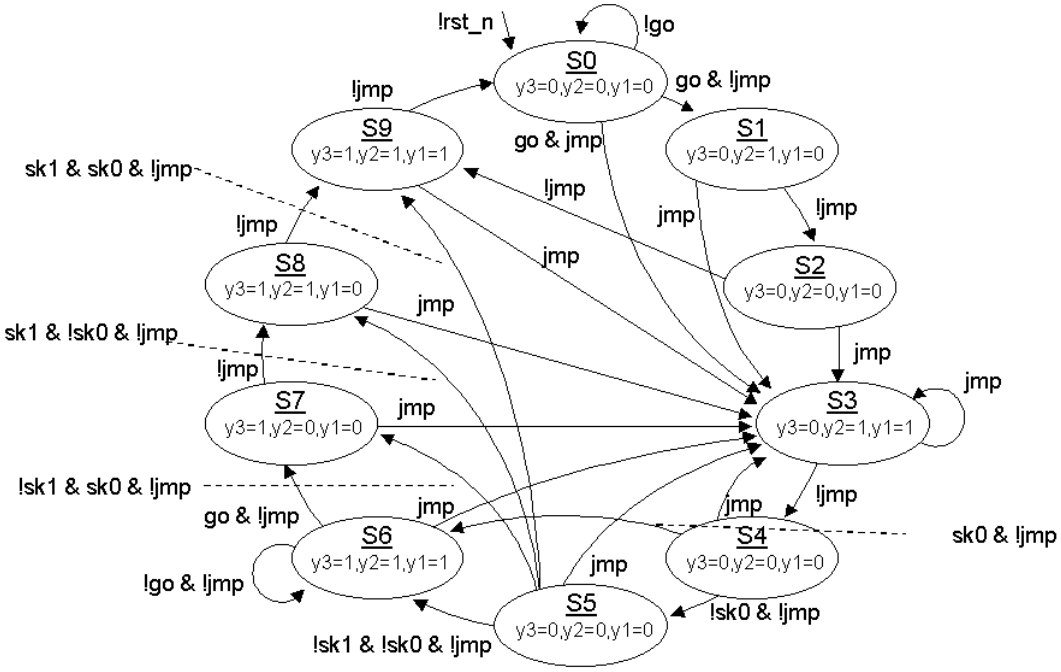



```
module fsm_cc7_onehot_fp
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 0,
            S1  = 1,
            S2  = 2,
            S3  = 3,
            S4  = 4,
            S5  = 5,
            S6  = 6,
            S7  = 7,
            S8  = 8,
            S9  = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                  state     <= 0;
                  state[S0] <= 1'b1;
                end
    else          state     <= next;

  always @(state or go or jmp) begin
    next = 10'b0;
    y1 = 1'b0;
    case (1'b1) // ambit synthesis case = full, parallel
      state[S0] : if (!go)      next[S0]=1'b1;
                  else if (jmp) next[S3]=1'b1;
                  else          next[S1]=1'b1;
      state[S1] : if (jmp)      next[S3]=1'b1;
                  else          next[S2]=1'b1;
      state[S2] :               next[S3]=1'b1;
      state[S3] : begin y1 = 1'b1;
                    if (jmp)    next[S3]=1'b1;
                    else        next[S4]=1'b1;
                  end
      state[S4] : if (jmp)      next[S3]=1'b1;
                  else          next[S5]=1'b1;
      state[S5] : if (jmp)      next[S3]=1'b1;
                  else          next[S6]=1'b1;
      state[S6] : if (jmp)      next[S3]=1'b1;
                  else          next[S7]=1'b1;
      state[S7] : if (jmp)      next[S3]=1'b1;
                  else          next[S8]=1'b1;
      state[S8] : if (jmp)      next[S3]=1'b1;
                  else          next[S9]=1'b1;
      state[S9] : if (jmp)      next[S3]=1'b1;
                  else          next[S0]=1'b1;
    endcase
  end
endmodule
```

Example 8 - fsm_cc7 design - case (1'b1) onehot style - 53 lines of code

7.2 10-state moderately complex FSM design - case (1'b1) onehot coding style

Example 9 is the fsm_cc8 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc8 design requires 86 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_onehot_fp
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0 = 0,
            S1 = 1,
            S2 = 2,
            S3 = 3,
            S4 = 4,
            S5 = 5,
            S6 = 6,
            S7 = 7,
            S8 = 8,
            S9 = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                state     <= 0;
                state[S0] <= 1'b1;
              end
    else          state     <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 0;
    case (1'b1) // ambit synthesis case = full, parallel
      state[S0] : if      (!go)                 next[S0]=1'b1;
                  else if (jmp)                 next[S3]=1'b1;
                  else                          next[S1]=1'b1;
```

```
      state[S1] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S2]=1'b1;
      state[S2] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S3] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S4]=1'b1;
      state[S4] : if      (jmp)                  next[S3]=1'b1;
                  else if (sk0 && !jmp)          next[S6]=1'b1;
                  else                           next[S5]=1'b1;
      state[S5] : if      (jmp)                  next[S3]=1'b1;
                  else if (!sk1 && !sk0 && !jmp) next[S6]=1'b1;
                  else if (!sk1 &&  sk0 && !jmp) next[S7]=1'b1;
                  else if ( sk1 && !sk0 && !jmp) next[S8]=1'b1;
                  else                           next[S9]=1'b1;
      state[S6] : if      (jmp)                  next[S3]=1'b1;
                  else if (go && !jmp)           next[S7]=1'b1;
                  else                           next[S6]=1'b1;
      state[S7] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S8]=1'b1;
      state[S8] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S9] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S0]=1'b1;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (1'b1)
        next[S0], next[S2], next[S4], next[S5] : ; // default outputs
        next[S7] :   y3 <= 1'b1;
        next[S1] :   y2 <= 1'b1;
        next[S3] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                   end
        next[S8] : begin
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
        next[S6], next[S9] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
    endcase
  end
endmodule
```

Example 9 - fsm_cc8 design - case (1'b1) onehot style - 86 lines of code

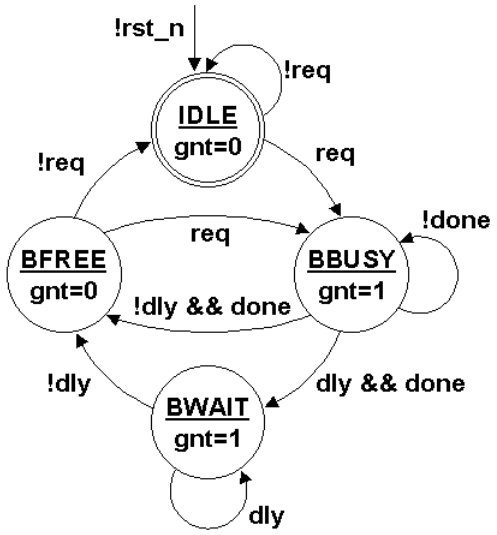
This is the only coding style where I recommend using full_case and parallel_case statements. The parallel case statement tells the synthesis tool to not build a priority encoder even though in theory, more than one of the state bits could be set (as engineers, we know that this is a onehot FSM and that only one bit can be set so no priority encoder is required). The value of the full_case statement is still in question.

8. Registered FSM Outputs (Good Style)

Registering the outputs of an FSM design insures that the outputs are glitch-free and frequently improves synthesis results by standardizing the output and input delay constraints of synthesized modules (see reference [1] for more information).

FSM outputs are easily registered by adding a third always sequential block to an FSM module where output assignments are generated in a case statement with case items corresponding to the next state that will be active when the output is clocked.

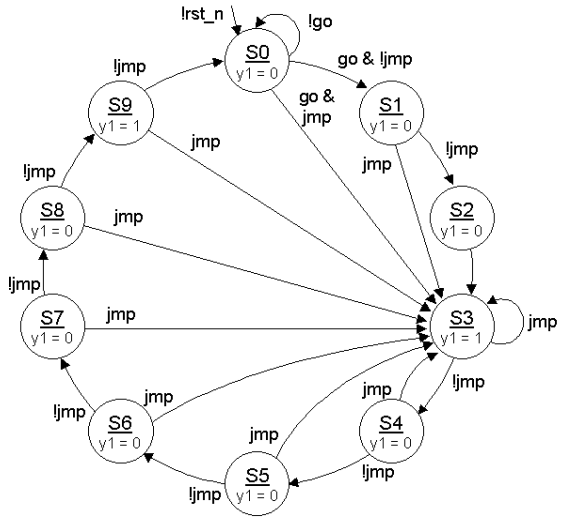
```
module fsm_cc4_2r
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'b00,
                  BBUSY = 2'b01,
                  BWAIT = 2'b10,
                  BFREE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    case (state)
      IDLE : if (req)       next = BBUSY;
             else           next = IDLE;
      BBUSY: if (!done)     next = BBUSY;
             else if ( dly) next = BWAIT;
             else           next = BFREE;
      BWAIT: if (!dly)      next = BFREE;
             else           next = BWAIT;
      BFREE: if (req)       next = BBUSY;
             else           next = IDLE;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) gnt <= 1'b0;
    else begin
      gnt <= 1'b0;
      case (next)
        IDLE,  BFREE: ; // default outputs
        BBUSY, BWAIT: gnt <= 1'b1;
      endcase
    end
endmodule
```



Example 10 - fsm_cc4 design - three always blocks w/registered outputs - 40 lines of code

8.1 10-state simple FSM design - three always blocks - registered outputs

Example 11 is the fsm_cc7 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc7 design requires 60 lines of code (coding requirements are compared in a later section).



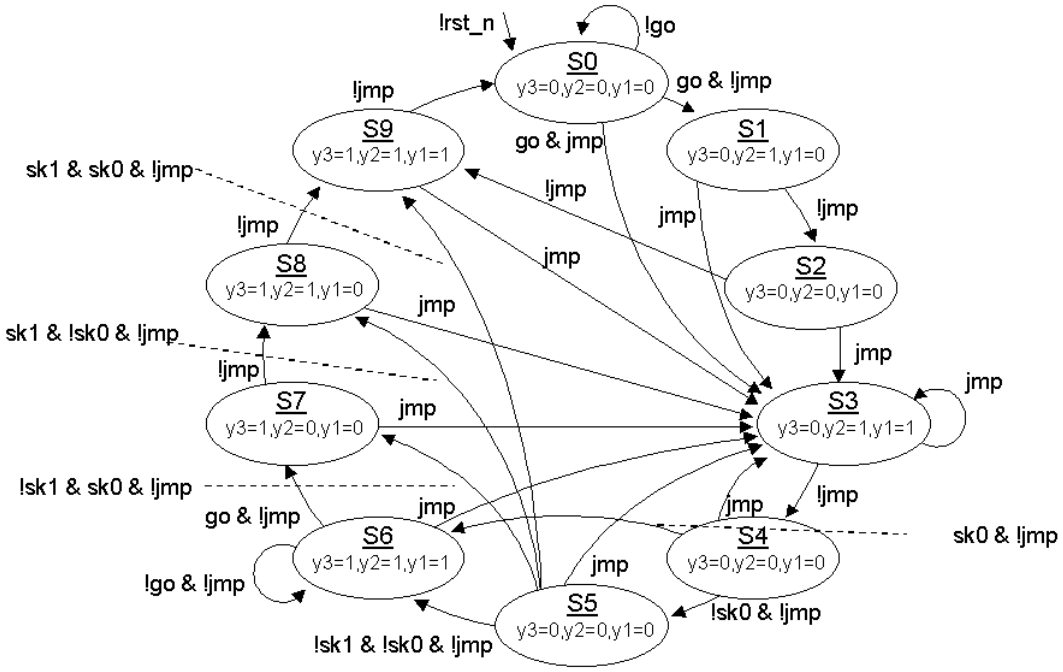

```
module fsm_cc7_3r
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or go or jmp) begin
    next = 4'bx;
    y1 = 1'b0;
    case (state)
      S0 : if (!go)      next = S0;
           else if (jmp) next = S3;
           else          next = S1;
      S1 : if (jmp)      next = S3;
           else          next = S2;
      S2 :               next = S3;
      S3 : begin y1 = 1'b1;
             if (jmp)    next = S3;
             else        next = S4;
           end
      S4 : if (jmp)      next = S3;
           else          next = S5;
      S5 : if (jmp)      next = S3;
           else          next = S6;
      S6 : if (jmp)      next = S3;
           else          next = S7;
      S7 : if (jmp)      next = S3;
           else          next = S8;
      S8 : if (jmp)      next = S3;
           else          next = S9;
      S9 : if (jmp)      next = S3;
           else          next = S0;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) y1 <= 1'b0;
    else begin
      y1 <= 1'b0;
      case (state)
        S0, S1, S2, S4, S5, S6, S7, S8, S9:; // default
```

```
        S3 : y1 <= 1'b1;
      endcase
    end
endmodule
```

Example 11 - fsm_cc7 design - three always blocks w/registered outputs - 60 lines of code

8.2 10-state moderately complex FSM design - three always blocks - registered outputs

Example 12 is the fsm_cc8 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc8 design requires 83 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_3r
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
```

```
    next = 4'bx;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
           else                          next = S1;
      S1 : if (jmp)                      next = S3;
           else                          next = S2;
      S2 : if (jmp)                      next = S3;
           else                          next = S9;
      S3 : if (jmp)                      next = S3;
           else                          next = S4;
      S4 : if      (jmp)                 next = S3;
           else if (sk0 && !jmp)         next = S6;
           else                          next = S5;
      S5 : if      (jmp)                 next = S3;
           else if (!sk1 && !sk0 && !jmp) next = S6;
           else if (!sk1 &&  sk0 && !jmp) next = S7;
           else if ( sk1 && !sk0 && !jmp) next = S8;
           else                          next = S9;
      S6 : if      (jmp)                 next = S3;
           else if (go && !jmp)          next = S7;
           else                          next = S6;
      S7 : if (jmp)                      next = S3;
           else                          next = S8;
      S8 : if (jmp)                      next = S3;
           else                          next = S9;
      S9 : if (jmp)                      next = S3;
           else                          next = S0;
    endcase
  end

always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (next)
        S0, S2, S4, S5 : ; // default outputs
        S7             :   y3 <= 1'b1;
        S1             :   y2 <= 1'b1;
        S3             : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                         end
        S8             : begin
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
        S6, S9         : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
    endcase
  end
endmodule
```

Example 12 - fsm_cc8 design - three always blocks w/registered outputs - 83 lines of code

9. Comparing RTL Coding Efforts

In the preceding sections, three different FSM designs were coded four different ways: (1) two always block coding style, (2) one always block coding style, (3) onehot, two always block coding style, and (4) three always block coding style with registered outputs.

	Two always block coding style	One always block coding style (12%-83% larger)	Onehot, two always block coding style	Three always block coding style w/ registered outputs
fsm_cc4 (4 states, simple)	37 lines of code	47 lines of code (12%-27% larger)	42 lines of code	40 lines of code
fsm_cc7 (10 states, simple)	50 lines of code	79 lines of code (32%-58% larger)	53 lines of code	60 lines of code
fsm_cc8 (10 states, moderate complexity)	80 lines of code	146 lines of code (70%-83% larger)	86 lines of code	83 lines of code

Table 1 - Lines of RTL code required for different FSM coding styles

From Table 1, we see that the one always block FSM coding style is the least efficient coding style with respect to the amount of RTL code required to render an equivalent design. In fact, the more outputs that an FSM design has and the more transition arcs in the FSM state diagram, and thus the faster the one always block coding style increases in size over comparable FSM coding styles.

If you are a contractor or are paid by the line-of-code, clearly, the one always block FSM coding style should be your preferred style. If you are trying to complete a project on time and code the design in a concise manner, the one always block coding style should be avoided.

10. Synthesis Results

Synthesis results were not complete by the time the paper was submitted for publication.

11. Running Cadence BuildGates

```
ac_shell          (for command-line mode)
ac_shell -gui &   (for GUI mode with process running  in background)
```

12. Verilog-2001 Enhancements

As of this writing, the Cadence Verilog simulators do not support many (if any) of the new Verilog-2001 enhancements. All of the preceding examples were coded with Verilog-2001 enhanced and concise ANSI-style module headers. In reality, to make the designs work with the Cadence Verilog simulators, I had to also code Verilog-1995 style module headers and select the appropriate header using macro definitions. To ease the task, I have created two aliases for 1995-style Verilog simulations.

```
alias ncverilog95 "ncverilog +define+V95"
alias verilog95 "verilog +define+V95"
```

12.1 ANSI-Style port declarations

ANSI-style port declarations are a nice enhancement to Verilog-2001 but they are not yet supported by version 3.4 of NC-Verilog or Verilog-XL, but they are reported to work with BuildGates. This enhancement permits module headers to be declared in a much more concise manner over traditional Verilog-1995 coding requirements.

Verilog-1995 required each module port be declared two or three times. Verilog-1995 required that (1) the module ports be listed in the module header, (2) the module port directions be declared, and (3) for reg-variable output ports, the port data type was also required.

Verilog-2001 combined all of this information into single module port declarations, significantly reducing the verbosity and redundancy of Verilog module headers. Of the major Verilog vendors, only the Cadence Verilog simulators do not support this Verilog-2001 feature. This means that users who want to take advantage of this feature and who use simulators from multiple vendors, including Cadence, must code both styles of module headers using `ifdef statements to select the appropriate module header style.

I prefer the following coding style to support retro-style Verilog simulators:

```
`ifdef V95
  // Verilog-1995 old-style, verbose module headers
`else
  // Verilog-2001 new-style, efficient module headers
`endif
```

The following example is from the actual fsm_cc4_1.v file used to test one always block FSM coding styles in this paper.

```
`ifdef V95
module fsm_cc4_1 (gnt, dly, done, req, clk, rst_n);
  output gnt;
  input  dly, done, req;
  input  clk, rst_n;
  reg    gnt;
`else
module fsm_cc4_1
  (output reg gnt,
   input dly, done, req, clk, rst_n);
`endif
```

It should be noted that this is an easy enhancement to implement, significantly improves the coding efficiency of module headers and that some major Verilog vendors have supported this enhanced coding style for more than a year at the time this paper was written. The author strongly encourages Cadence simulator developers to quickly adopt this Verilog-2001 enhancement to ease the Verilog coding burden for Cadence tool users.

12.2 @* Combinational sensitivity list

Verilog-2001 added the much-heralded @* combinational sensitivity list token. Although the combinational sensitivy list could be written using any of the following styles:

```
always @*
always @(*)
always @( * )
always @ ( * )
```

or any other combination of the characters @ (*) with or without white space, the author prefers the first and most abbreviated style. To the author, "always @*" clearly denotes that a combinational block of logic follows.

The Verilog-2001 "always @*" coding style has a number of important advantages over the more cumbersome Verilog-1995 combinational sensitivity list coding style:

- Reduces coding errors - the code informs the simulator that the intended implementation is combinational logic, so the simulator will automatically add and remove signals from the sensitivity list as RTL code is added or deleted from the combinational always block. The RTL coder is no longer burdened with manually insuring that all of the necessary signals are present in the sensitivity list. This will reduce coding errors that do not show up until a synthesis tool or linting tool reports errors in the sensitivity list. The basic intent of this enhancement is to inform the simulator, "if the synthesis tool wants the signals, so do we!"

- Abbreviated syntax - large combinational blocks often meant multiple lines of redundant signal naming in a sensitivity list. The redundancy served no appreciable purpose and users will gladly adopt the more concise and abbreviated @* syntax.

- Clear intent - an always @* procedural block informs the code-reviewer that this block is intended to behave like, and synthesize to, combinational logic.

13. SystemVerilog Enhancements

In June of 2002, Accellera released the SystemVerilog 3.0 language specification, a superset of Verilog-2001 with many nice enhancements for modeling, synthesis and verification. The basis for the SystemVerilog language comes from a donation by CoDesign Automation of significant portions of their Superlog language.

Key functionality that has been added to the Accellera SystemVerilog 3.0 Specification to support FSM design includes:

Enumerated types - Why do engineers want to use enumerated types? (1) Enumerated types permit abstract state declaration without defining the state encodings, and (2) enumerated types can typically be easily displayed in a waveform viewer permitting faster design debug. Enumerated types allow abstract state definitions without required state encoding assignments. Users also wanted the ability to assign state encodings to control implementation details such as output encoded FSM designs with simple registered outputs.

One short coming of traditional enumerated types was the inability to make X-state assignments. As discussed earlier in this paper, X-state assignments are important to simulation debug and synthesis optimization. SystemVerilog enumerated types will permit data type declaration, making it possible to declare enumerated types with an all-X's definitions.

Other SystemVerilog proposals under consideration for FSM enhancement include:

Different enumerated styles - the ability to declare different enumerated styles, such as enum_onehot, to make experimentation with different encoding styles easier to do. Currently, when changing from a binary encoding to an efficient onehot encoding style, 8 different code changes must be made in the FSM module. Wouldn't it be nice if the syntax permitted easier handling of FSM styles without manual intervention.

Transition statement and ->> next state transition operator -

These enhancements were removed from the SystemVerilog 3.0 Standard only because their definition was not fully elaborated and understood. Some people like the idea of a next-state transition operator that closely corresponds to the transition arcs that are shown on an FSM state diagram.

The infinitely abusable "goto" statement - Concern about a "goto" statement that could "cause spaghetti-code" could be avoided by limiting a goto-transition to a label within the same procedural block. Implicit FSM coding styles are much cleaner with a goto statement. A goto statement combined with a carefully crafted disable statement makes reset handling easier to do. A goto statement alleviates the problem of multiple transition arcs within a traditional implicit FSM design. Goto is just a proposal and may not pass.

14. Conclusions

There are many ways to code FSM designs. There are many inefficient ways to code FSM designs!

Use parameters to define state encodings. Parameters are constants that are local to a module. After defining the state encodings at the top of the FSM module, never use the state encodings again in the RTL code. This makes it possible to easily change the state encodings in just one place, the parameter definitions, without having to touch the rest of the FSM RTL code. This makes state-encoding-experimentation easy to do.

Use a two always block coding style to code FSM designs with combinational outputs. This style is efficient and easy to code and can also easily handle Mealy FSM designs.

Use a three always block coding style to code FSM designs with registered outputs. This style is efficient and easy to code. Note, another recommended coding style for FSM designs with registered outputs is the "output encoded" FSM coding style (see reference [1] for more information on this coding style).

Avoid the one always block FSM coding style. It is generally more verbose than an equivalent two always block coding style, output assignments are more error prone to coding mistakes and one cannot code asynchronous Mealy outputs without making the output assignments with separate continuous assign statements.

15. Acknowledgements

I would like to especially thank both Rich Owen and Nasir Junejo of Cadence for their assistance and tips enabling the use of the BuildGates synthesis tool. Their input helped me to achieve very favorable results in a short period of time.

16. References

[1] Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs," *SNUG'2000 Boston (Synopsys Users Group Boston, MA, 2000) Proceedings*, September 2000. (Also available online at www.sunburst-design.com/papers)

[2] Clifford E. Cummings, "'full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, October 1999. (Also available online at www.sunburst-design.com/papers)

[3] Clifford E. Cummings, "New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!)," *International HDL Conference 2002 Proceedings*, pp. 17-24, March 2002. (Also available online at www.sunburst-design.com/papers)

[4] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG'2000 Boston (Synopsys Users Group San Jose, CA, 2000) Proceedings*, March 2000. (Also available online at www.sunburst-design.com/papers)

[5] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995, pg. 47, section 5.4.1 - Determinism.

[6] Nasir Junejo, personal communication

[7] Rich Owen, personal communication

[8] The Programmable Logic Data Book, Xilinx, 1994, pg. 8-171

[9] William I. Fletcher, An Engineering Approach To Digital Design, New Jersey, Prentice-Hall, 1980

[10] Zvi Kohavi, Switching And Finite Automata Theory, Second Edition, New York, McGraw-Hill Book Company, 1978

Revision 1.2 (July 2004) - What Changed?

Version 1.1 of the paper had misspelled the name of the Data IO PLD programming language in Section 6. The corrected spelling is ABEL. My thanks to a reader who found and reported this mistake. Also, the [10] reference title was corrected.

Author & Contact Information

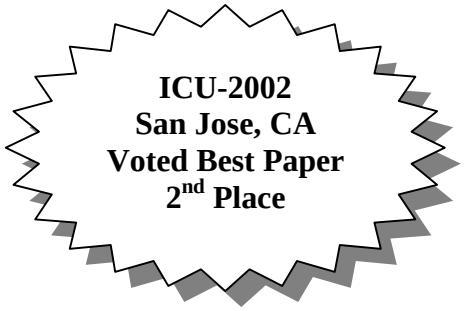
Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

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

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

E-mail Address: cliffc@sunburst-design.com
An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers (Data accurate as of July 22nd, 2002)

The Fundamentals of Efficient Synthesizable Finite State Machine Design using NC-Verilog and BuildGates



Clifford E. Cummings

Sunburst Design, Inc.
503-641-8446
cliffc@sunburst-design.com

INTERNATIONAL CADENCE USERGROUP CONFERENCE
September 16-18, 2002
San Jose, California

Abstract

This paper details proven RTL coding styles for efficient and synthesizable Finite State Machine (FSM) design using IEEE-compliant Verilog simulators. Important techniques related to one and two always block styles to code FSMs with combinational outputs are given to show why using a two always block style is preferred. An efficient Verilog-unique onehot FSM coding style is also shown. Reasons and techniques for registering FSM outputs are also detailed. Myths surrounding erroneous state encodings, full-case and parallel-case usage are also discussed. Compliance and enhancements related to the IEEE 1364-2001 Verilog Standard, the proposed IEEE 1364.1 Verilog Synthesis Interoperability Standard and the proposed Accellera SystemVerilog Standard are also discussed.

1. Introduction

FSM is an abbreviation for *Finite State Machine*.

There are many ways to code FSMs including many very poor ways to code FSMs. This paper will examine some of the most commonly used FSM coding styles, their advantages and disadvantages, and offer guidelines for doing efficient coding, simulation and synthesis of FSM designs.

This paper will also detail Accellera SystemVerilog enhancements that will facilitate and enhance future Verilog FSM designs.

In this paper, multiple references are made to combinational always blocks and sequential always blocks. Combinational always blocks are always blocks that are used to code combinational logic functionality and are strictly coded using blocking assignments (see Cummings[4]). A combinational always block has a combinational sensitivity list, a sensitivity list without "posedge" or "negedge" Verilog keywords.

Sequential always blocks are always blocks that are used to code clocked or sequential logic and are always coded using nonblocking assignments (see Cummings[4]). A sequential always block has an edge-based sensitivy list.

2. Mealy and Moore FSMs

A common classification used to describe the type of an FSM is Mealy and Moore state machines[9][10].

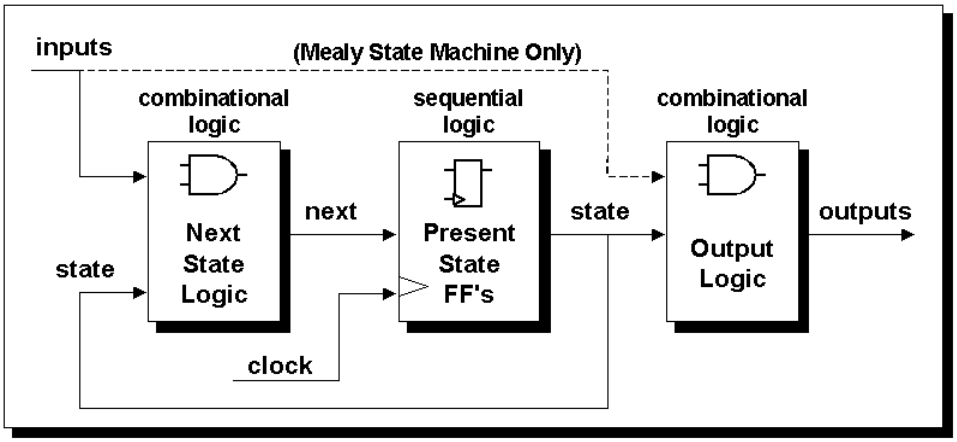


Figure 1 - Finite State Machine (FSM) block diagram

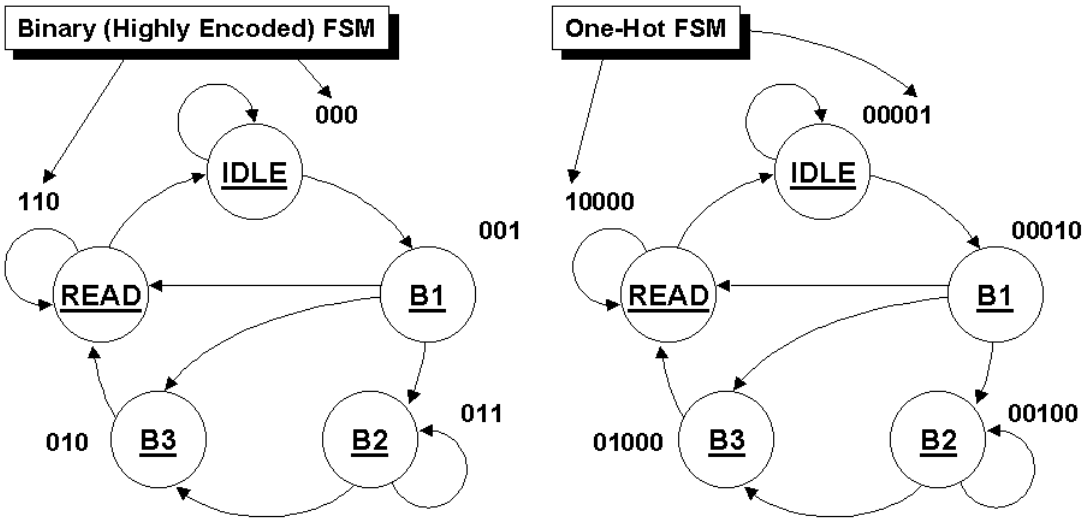
A Moore FSM is a state machine where the outputs are only a function of the present state. A Mealy FSM is a state machine where one or more of the outputs is a function of the present state and one or more of the inputs. A block diagram for Moore and Mealy FSMs is shown Figure 1.

3. Binary Encoded or Onehot Encoded?

Common classifications used to describe the state encoding of an FSM are Binary (or highly encoded) and Onehot.

A binary-encoded FSM design only requires as many flip-flops as are needed to uniquely encode the number of states in the state machine. The actual number of flip-flops required is equal to the ceiling of the log-base-2 of the number of states in the FSM.

A onehot FSM design requires a flip-flop for each state in the design and only one flip-flop (the flip-flop representing the current or "hot" state) is set at a time in a onehot FSM design. For a state machine with 9-16 states, a binary FSM only requires 4 flip-flops while a onehot FSM requires a flip-flop for each state in the design (9-16 flip-flops).



FPGA vendors frequently recommend using a onehot state encoding style because flip-flops are plentiful in an FPGA and the combinational logic required to implement a onehot FSM design is typically smaller than most binary encoding styles. Since FPGA performance is typically related to the combinational logic size of the FPGA design, onehot FSMs typically run faster than a binary encoded FSM with larger combinational logic blocks[8].

4. FSM Coding Goals

To determine what constitutes an efficient FSM coding style, we first need to identify HDL coding goals and why they are important. After the HDL coding goals have been identified, we can then quantify the capabilities of various FSM coding styles.

The author has identified the following HDL coding goals as important when doing HDL-based FSM design:

- The FSM coding style should be easily modified to change state encodings and FSM styles.
- The coding style should be compact.
- The coding style should be easy to code and understand.

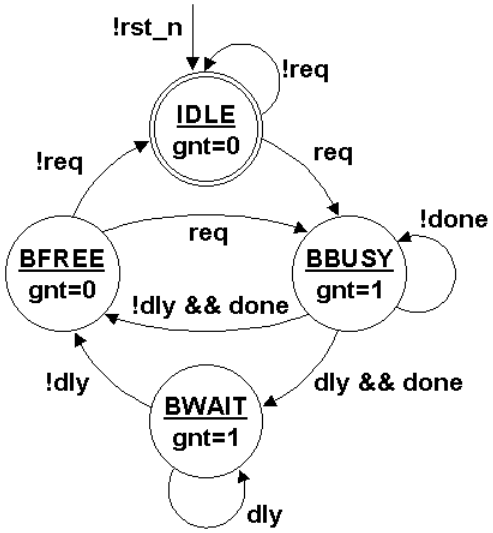
- The coding style should facilitate debugging.
- The coding style should yield efficient synthesis results.

Three different FSM designs will be examined in this paper. The first is a simple 4-state FSM design labeled fsm_cc4 with one output. The second is a 10-state FSM design labeled fsm_cc7 with only a few transition arcs and one output. The third is another 10-state FSM design labeled fsm_cc8 with multiple transition arcs and three outputs. The coding efforts to create these three designs will prove interesting.

5. Two Always Block FSM Style (Good Style)

One of the best Verilog coding styles is to code the FSM design using two always blocks, one for the sequential state register and one for the combinational next-state and combinational output logic.

```
module fsm_cc4_2
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'b00,
                  BBUSY = 2'b01,
                  BWAIT = 2'b10,
                  BFREE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    gnt  = 1'b0;
    case (state)
      IDLE :   if (req)       next = BBUSY;
               else           next = IDLE;
      BBUSY: begin
               gnt = 1'b1;
               if (!done)     next = BBUSY;
               else if ( dly) next = BWAIT;
               else           next = BFREE;
             end
      BWAIT: begin
               gnt = 1'b1;
               if (!dly)      next = BFREE;
               else           next = BWAIT;
             end
      BFREE:   if (req)       next = BBUSY;
               else           next = IDLE;
    endcase
  end
endmodule
```



Example 1 - fsm_cc4 design - two always block style - 37 lines of code

5.1 Important coding style notes:

- Parameters are used to define state encodings instead of the Verilog `define macro definition construct[3]. After parameter definitions are created, the parameters are used throughout the rest of the

design, not the state encodings. This means that if an engineer wants to experiment with different state encodings, only the parameter values need to be modified while the rest of the Verilog code remains unchanged.

- Declarations are made for **`state`** and **`next`** (next state) after the parameter assignments.
- The sequential always block is coded using nonblocking assignments.
- The combinational always block sensitivity list is sensitive to changes on the **`state`** variable and all of the inputs referenced in the combinational always block.
- Assignments within the combinational always block are made using Verilog blocking assignments.
- The combinational always block has a default **`next`** state assignment at the top of the always block (see section 5.3 for details about making default-X assignments).
- Default output assignments are made before coding the **`case`** statement (this eliminates latches and reduces the amount of code required to code the rest of the outputs in the **`case`** statement and highlights in the **`case`** statement exactly in which states the individual output(s) change).
- In the states where the output assignment is not the default value assigned at the top of the always block, the output assignment is only made once for each state.
- There is an **`if`**-statement, an **`else`**-**`if`**-statement or an **`else`** statement for each transition arc in the FSM state diagram. The number of transition arcs between states in the FSM state diagram should equal the number of **`if`**-**`else`**-type statements in the combinational always block.
- For ease of scanning and debug, all of the **`next`** assignments have been placed in a single column, as opposed to finding **`next`** assignments following the contour of the RTL code.

5.2 The unfounded fear of transitions to erroneous states

In engineering school, we were all cautioned about "what happens if you FSM gets into an erroneous state?" In general, this concern is both invalid or poorly developed.

I do not worry about most of my FSM designs going to an erroneous state any more than I worry about any other register in my design spontaneously changing value. It just does not occur!

There are exceptions, such as satellites (subject to alpha particle bombardment) or medical implants (subject to radiation and requiring extra robust design), plus other examples. In these situations, one does have to worry about FSMs going to an erroneous state, but most engineering schools fail to note that getting back to a known state is typically not good enough! Even though the FSM is now in a known state, the rest of the hardware is still expecting activity related to another state. It is possible for the design to lockup waiting for signals that will never arrive because the FSM changed states without resetting the rest of the design. At the very least, the FSM should transition to an error state that communicates to the rest of the design that resetting will occur on the next state transition, "get ready!"

5.3 Making default `next` equal all X's assignment

Placing a default next state assignment on the line immediately following the always block sensitivity list is a very efficient coding style. This default assignment is updated by next-state assignments inside the case statement. There are three types of default next-state assignments that are commonly used: (1) next is set to all X's, (2) next is set to a predetermined recovery state such as IDLE, or (3) next is just set to the value of the state register.

By making a default next state assignment of X's, pre-synthesis simulation models will cause the state machine outputs to go unknown if not all state transitions have been explicitly assigned in the case statement. This is a useful technique to debug state machine designs, plus the X's will be treated as "don't cares" by the synthesis tool.

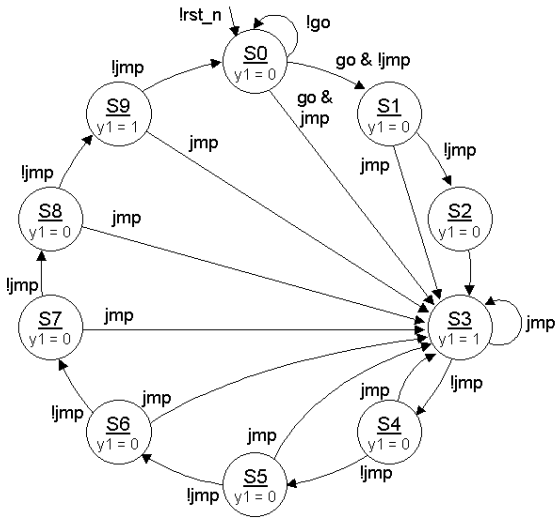
Some designs require an assignment to a known state as opposed to assigning X's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan

chain and some designs that are equivalence checked with formal verification tools. Making a default next state assignment of either IDLE or all 0's typically satisfies these design requirements and making the initial default assignment might be easier than coding all of the explicit next-state transition assignments in the case statement.

5.4 10-state simple FSM design - two always blocks

Example 2 is the fsm_cc7 design implemented with two always blocks. Using two always blocks, the fsm_cc7 design requires 50 lines of code (coding requirements are compared in a later section).

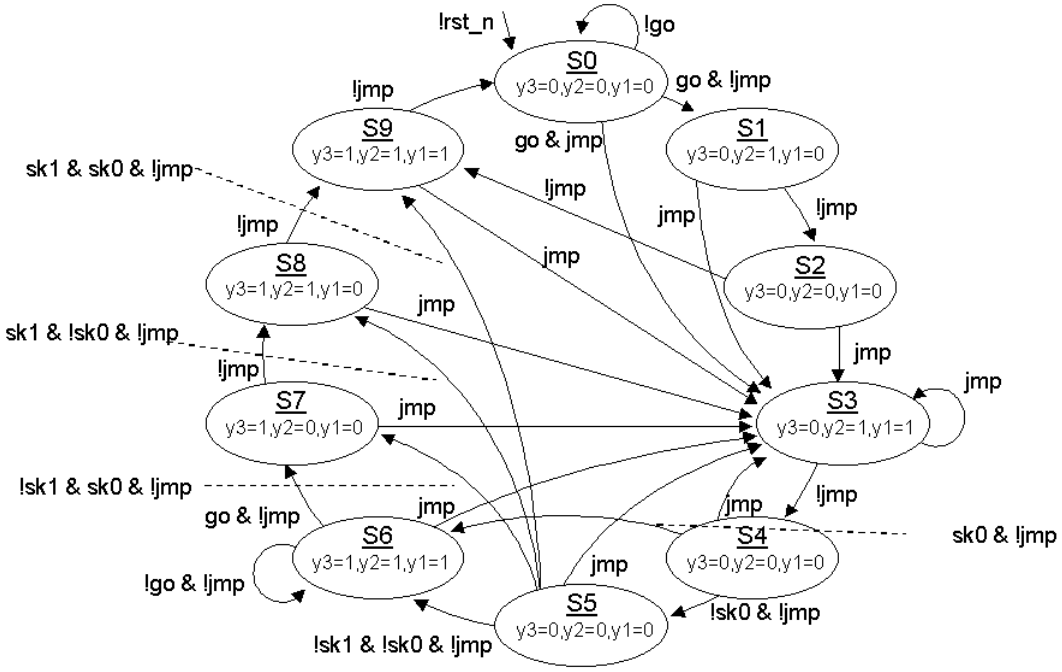
```
module fsm_cc7_2
        (output reg y1,
         input      jmp, go, clk, rst_n);

        parameter S0  = 4'b0000,
                  S1  = 4'b0001,
                  S2  = 4'b0010,
                  S3  = 4'b0011,
                  S4  = 4'b0100,
                  S5  = 4'b0101,
                  S6  = 4'b0110,
                  S7  = 4'b0111,
                  S8  = 4'b1000,
                  S9  = 4'b1001;

        reg [3:0] state, next;

        always @(posedge clk or negedge rst_n)
          if (!rst_n) state <= S0;
          else        state <= next;

        always @(state or go or jmp) begin
          next = 4'bx;
          y1 = 1'b0;
          case (state)
            S0 : if (!go)      next = S0;
                 else if (jmp) next = S3;
                 else          next = S1;
            S1 : if (jmp)      next = S3;
                 else          next = S2;
            S2 :               next = S3;
            S3 : begin y1 = 1'b1;
                   if (jmp)    next = S3;
                   else        next = S4;
                 end
            S4 : if (jmp)      next = S3;
                 else          next = S5;
            S5 : if (jmp)      next = S3;
                 else          next = S6;
            S6 : if (jmp)      next = S3;
                 else          next = S7;
            S7 : if (jmp)      next = S3;
                 else          next = S8;
            S8 : if (jmp)      next = S3;
                 else          next = S9;
            S9 : if (jmp)      next = S3;
                 else          next = S0;
          endcase
        end
      endmodule
```



Example 2 - fsm_cc7 design - two always block style - 50 lines of code

5.5 10-state moderately complex FSM design - two always blocks

Example 3 is the fsm_cc8 design implemented with two always blocks. Using two always blocks, the fsm_cc8 design requires 80 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_2
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 4'bx;
    y1 = 1'b0;
    y2 = 1'b0;
    y3 = 1'b0;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
```

```
            else                              next = S1;
       S1 : begin
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S2;
            end
       S2 : if (jmp)                          next = S3;
            else                              next = S9;
       S3 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S4;
              end
       S4 : if      (jmp)                     next = S3;
            else if (sk0 && !jmp)             next = S6;
            else                              next = S5;
       S5 : if      (jmp)                     next = S3;
            else if (!sk1 && !sk0 && !jmp)    next = S6;
            else if (!sk1 &&  sk0 && !jmp)    next = S7;
            else if ( sk1 && !sk0 && !jmp)    next = S8;
            else                              next = S9;
       S6 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if      (jmp)                   next = S3;
              else if (go && !jmp)            next = S7;
              else                            next = S6;
            end
       S7 : begin
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S8;
            end
       S8 : begin
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S9;
            end
       S9 : begin
              y1 = 1'b1;
              y2 = 1'b1;
              y3 = 1'b1;
              if (jmp)                        next = S3;
              else                            next = S0;
            end
    endcase
  end
endmodule
```

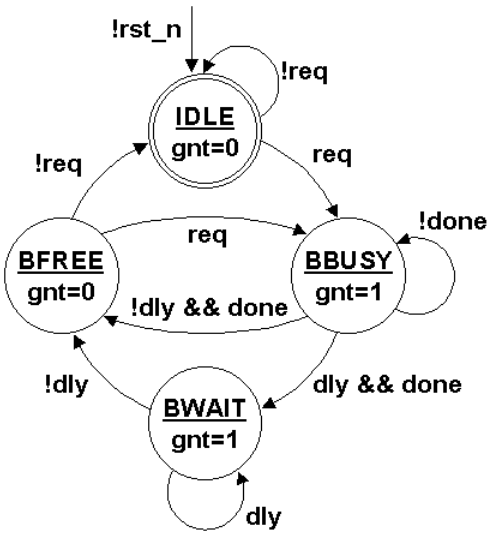
Example 3 - fsm_cc8 design - two always block style - 80 lines of code

6. One Always Block FSM Style (Avoid This Style!)

One of the most common FSM coding styles in use today is the one sequential always block FSM coding style. This coding style is very similar to coding styles that were popularized by PLD programming languages of the mid-1980s, such as ABLE. For most FSM designs, the one always block FSM coding style is more verbose, more confusing and more error prone than a comparable two always block coding style.

Reconsider the fsm_cc4 design shown in section 5.

```
module fsm_cc4_1
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'd0,
                  BBUSY = 2'd1,
                  BWAIT = 2'd2,
                  BFREE = 2'd3;

  reg [1:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state <= IDLE;
      gnt   <= 1'b0;
    end
    else begin
      state <= 2'bx;
      gnt   <= 1'b0;
      case (state)
        IDLE : if (req) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else              state <= IDLE;
        BBUSY: if (!done) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else if ( dly) begin
                                  state <= BWAIT;
                  gnt <= 1'b1;
                end
                else              state <= BFREE;
        BWAIT: if ( dly) begin
                                  state <= BWAIT;
                  gnt <= 1'b1;
                end
                else              state <= BFREE;
        BFREE: if (req) begin
                                  state <= BBUSY;
                  gnt <= 1'b1;
                end
                else              state <= IDLE;
      endcase
    end
endmodule
```



Example 4 - fsm_cc4 design - one always block style - 47 lines of code

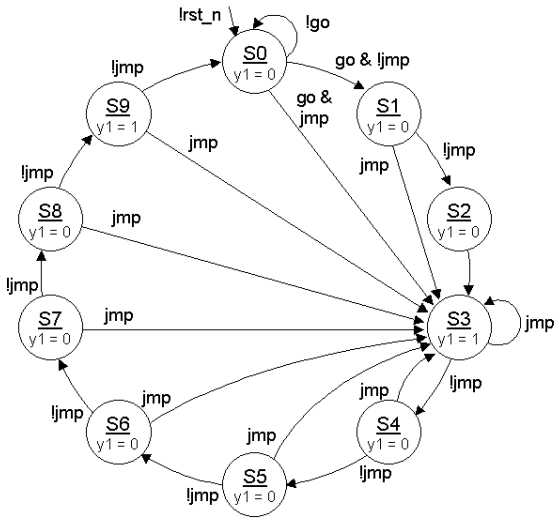
6.1 Important coding style notes:

- Parameters are used to define state encodings, the same as the two always block coding style.
- A declaration is made for **`state`**. Not for **`next`**.
- There is just one sequential always block, coded using nonblocking assignments.
- The there is still a default **`state`** assignment before the **`case`** statement, then the **`case`** statement tests the **`state`** variable. Will this be a problem? No, because the default **`state`** assignment is made with a nonblocking assignment, so the update to the **`state`** variable will happen at the end of the simulation time step.
- Default output assignments are made before coding the **`case`** statement (this reduces the amount of code required to code the rest of the outputs in the **`case`** statement).
- A **`state`** assignment must be made for each transition arc that transitions to a **`state`** where the output will be different than the default assigned value. For multiple outputs and for multiple transition arcs into a **`state`** where the outputs change, multiple **`state`** assignments will be required.
- The **`state`** assignments do not correspond to the current **`state`** of the **`case`** statement, but the **`state`** that **`case`** statement is transitioning to. *This is error prone* (but it does work if coded correctly).
- Again, for ease of scanning and debug, the all of the **`state`** assignments have been placed in a single column, as opposed to finding **`state`** assignments following the contour of the RTL code.
- All outputs will be registered (unless the outputs are placed into a separate combinational always block or assigned using continuous assignments). No asynchronous Mealy outputs can be generated from a single synchronous always block.
- Note: some misinformed engineers fear that making multiple assignments to the same variable, in the same always block, using nonblocking assignments, is undefined and can cause race conditions. This is not true. Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins! (See reference [5] for details).

6.2 10-state simple FSM design - one always blocks

Example 5 is the fsm_cc7 design implemented with one always blocks. Using one always blocks, the fsm_cc7 design requires 79 lines of code (coding requirements are compared in a later section).

```
module fsm_cc7_1
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state <= S0;
      y1 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
```

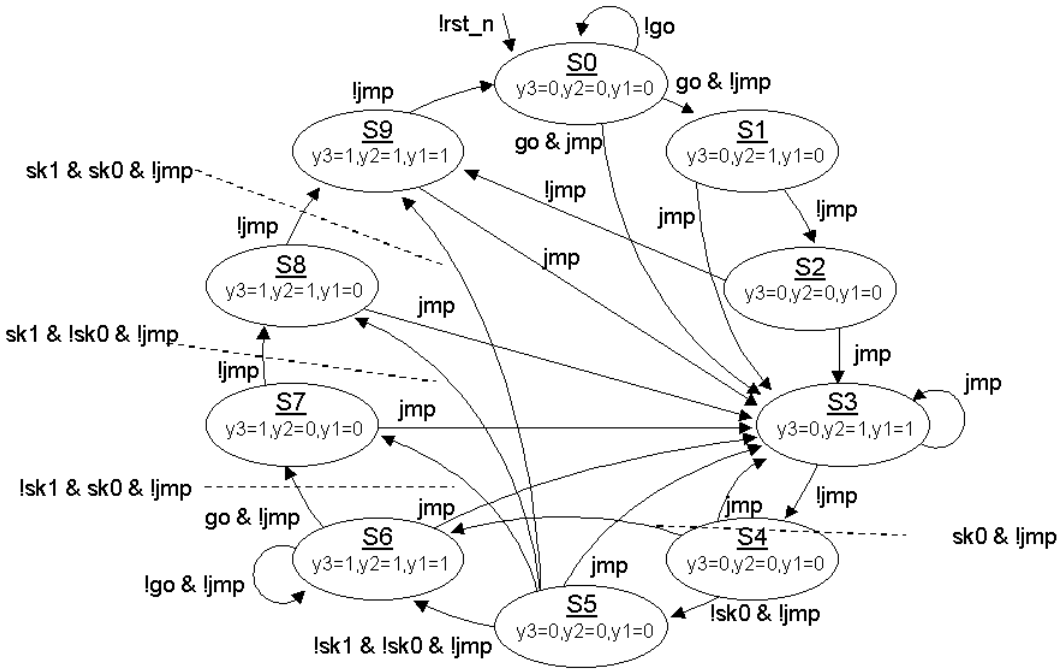


```
        state <= 4'bx;
        case (state)
          S0 : if (!go)      state <= S0;
               else if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S1;
          S1 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S2;
          S2 : begin
                 y1 <= 1'b1;
                             state <= S3;
               end
          S3 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S4;
          S4 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S5;
          S5 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S6;
          S6 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S7;
          S7 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S8;
          S8 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S9;
          S9 : if (jmp) begin
                 y1 <= 1'b1;
                             state <= S3;
               end
               else          state <= S0;
        endcase
      end
    endmodule
```

Example 5 - fsm_cc7 design - one always block style - 79 lines of code

6.3 10-state moderately complex FSM design - one always blocks

Example 6 is the fsm_cc8 design implemented with one always blocks. Using one always blocks, the fsm_cc8 design requires 146 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_1
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                           state <= S0;
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
                           state <= 4'bx;
      y1 <= 1'b0;
      y2 <= 1'b0;
```

```
y3 <= 1'b0;
case (state)
  S0 : if (!go)      state <= S0;
       else if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S1;
         y2 <= 1'b1;
       end
  S1 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S2;
  S2 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else begin
                     state <= S9;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
  S3 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else          state <= S4;
  S4 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else          state <= S5;
  S5 : if (jmp) begin
                     state <= S3;
         y1 <= 1'b1;
         y2 <= 1'b1;
       end
       else if (!sk1 && !sk0 && !jmp) begin
                     state <= S6;
         y1 <= 1'b1;
         y2 <= 1'b1;
         y3 <= 1'b1;
       end
       else if (!sk1 && sk0 && !jmp) begin
                     state <= S7;
         y3 <= 1'b1;
       end
       else if (sk1 && !sk0 && !jmp) begin
```

```
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S6 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else if (go && !jmp) begin
                        state <= S7;
                  y3 <= 1'b1;
                end
                else begin
                        state <= S6;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S7 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S8;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S8 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else begin
                        state <= S9;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                  y3 <= 1'b1;
                end
          S9 : if (jmp) begin
                        state <= S3;
                  y1 <= 1'b1;
                  y2 <= 1'b1;
                end
                else          state <= S0;
        endcase
      end
endmodule
```

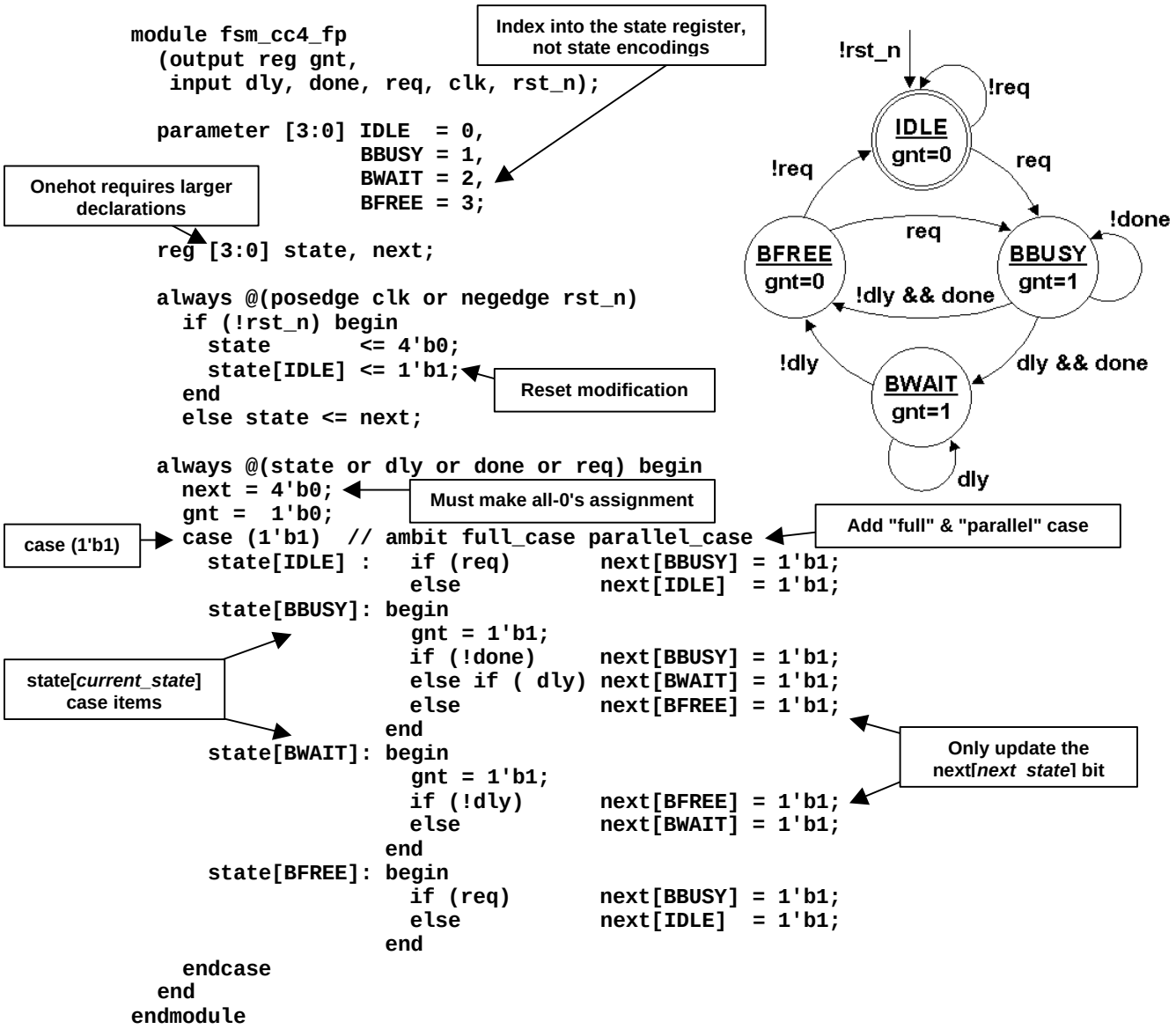
Example 6 - fsm_cc8 design - one always block style - 146 lines of code

7. Onehot FSM Coding Style (Good Style)

Efficient (small and fast) onehot state machines can be coded using an inverse case statement; a case statement where each case item is an expression that evaluates to true or false.

Reconsider the fsm_cc4 design shown in section 5. Eight coding modifications must be made to the two always block coding style of section 5 to implement the efficient onehot FSM coding style.

The key to understanding the changes is to realize that the parameters no longer represent **`state`** encodings, they now represent an *index* into the **`state`** vector, and comparisons and assignments are now being made to single bits in either the **`state`** or **`next`**-state vectors. Notice how the case statement is now doing a 1-bit comparison against the onehot state bit.



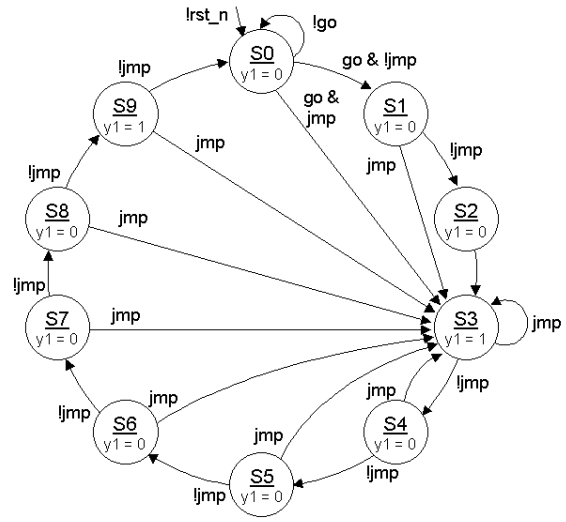

```
module fsm_cc4_fp
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [3:0] IDLE  = 0,
                  BBUSY = 1,
                  BWAIT = 2,
                  BFREE = 3;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      state       <= 4'b0;
      state[IDLE] <= 1'b1;
    end
    else state <= next;

  always @(state or dly or done or req) begin
    next = 4'b0;
    gnt =  1'b0;
    case (1'b1)  // ambit full_case parallel_case
      state[IDLE] :   if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
      state[BBUSY]: begin
                      gnt = 1'b1;
                      if (!done)     next[BBUSY] = 1'b1;
                      else if ( dly) next[BWAIT] = 1'b1;
                      else           next[BFREE] = 1'b1;
                    end
      state[BWAIT]: begin
                      gnt = 1'b1;
                      if (!dly)      next[BFREE] = 1'b1;
                      else           next[BWAIT] = 1'b1;
                    end
      state[BFREE]: begin
                      if (req)       next[BBUSY] = 1'b1;
                      else           next[IDLE]  = 1'b1;
                    end
    endcase
  end
endmodule
```

Example 7 - fsm_cc4 design - case (1'b1) onehot style - 42 lines of code

7.1 10-state simple FSM design - case (1'b1) onehot coding style

Example 8 is the fsm_cc7 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc7 design requires 53 lines of code (coding requirements are compared in a later section).

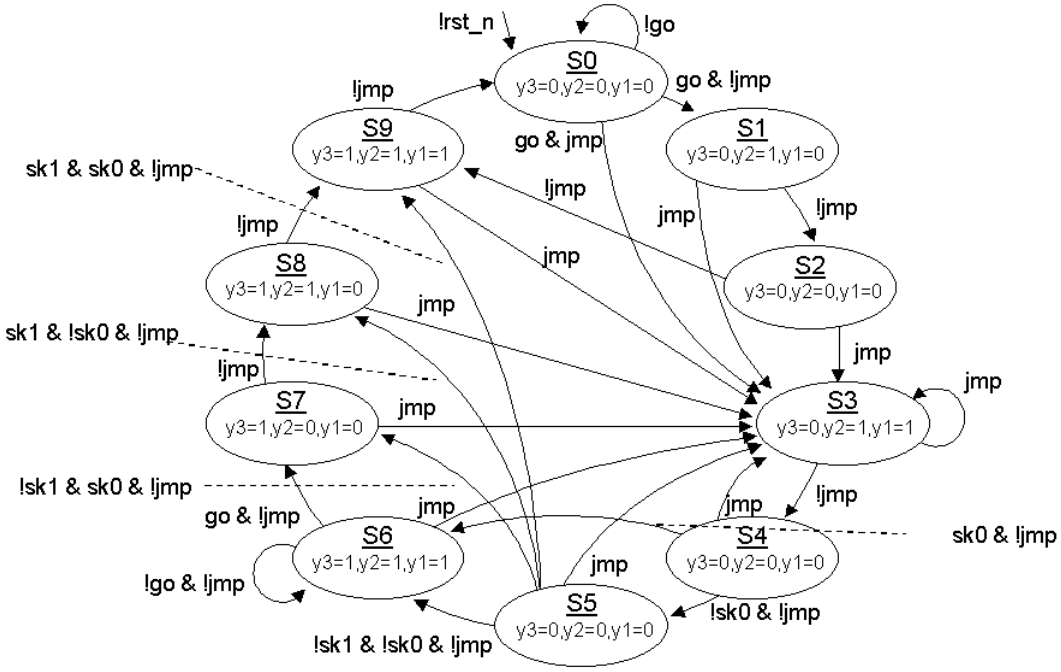
```
module fsm_cc7_onehot_fp
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 0,
            S1  = 1,
            S2  = 2,
            S3  = 3,
            S4  = 4,
            S5  = 5,
            S6  = 6,
            S7  = 7,
            S8  = 8,
            S9  = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                  state     <= 0;
                  state[S0] <= 1'b1;
                end
    else          state     <= next;

  always @(state or go or jmp) begin
    next = 10'b0;
    y1 = 1'b0;
    case (1'b1) // ambit full_case parallel_case
      state[S0] : if (!go)      next[S0]=1'b1;
                  else if (jmp) next[S3]=1'b1;
                  else          next[S1]=1'b1;
      state[S1] : if (jmp)      next[S3]=1'b1;
                  else          next[S2]=1'b1;
      state[S2] :               next[S3]=1'b1;
      state[S3] : begin y1 = 1'b1;
                    if (jmp)    next[S3]=1'b1;
                    else        next[S4]=1'b1;
                  end
      state[S4] : if (jmp)      next[S3]=1'b1;
                  else          next[S5]=1'b1;
      state[S5] : if (jmp)      next[S3]=1'b1;
                  else          next[S6]=1'b1;
      state[S6] : if (jmp)      next[S3]=1'b1;
                  else          next[S7]=1'b1;
      state[S7] : if (jmp)      next[S3]=1'b1;
                  else          next[S8]=1'b1;
      state[S8] : if (jmp)      next[S3]=1'b1;
                  else          next[S9]=1'b1;
      state[S9] : if (jmp)      next[S3]=1'b1;
                  else          next[S0]=1'b1;
    endcase
  end
endmodule
```



Example 8 - fsm_cc7 design - case (1'b1) onehot style - 53 lines of code

7.2 10-state moderately complex FSM design - case (1'b1) onehot coding style

Example 9 is the fsm_cc8 design implemented with the case (1'b1) onehot coding style. Using this style, the fsm_cc8 design requires 86 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_onehot_fp
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0 = 0,
            S1 = 1,
            S2 = 2,
            S3 = 3,
            S4 = 4,
            S5 = 5,
            S6 = 6,
            S7 = 7,
            S8 = 8,
            S9 = 9;

  reg [9:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
                state     <= 0;
                state[S0] <= 1'b1;
              end
    else          state     <= next;

  always @(state or jmp or go or sk0 or sk1) begin
    next = 0;
    case (1'b1) // ambit full_case parallel_case
      state[S0] : if      (!go)                 next[S0]=1'b1;
                  else if (jmp)                 next[S3]=1'b1;
                  else                          next[S1]=1'b1;
```

```
      state[S1] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S2]=1'b1;
      state[S2] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S3] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S4]=1'b1;
      state[S4] : if      (jmp)                  next[S3]=1'b1;
                  else if (sk0 && !jmp)          next[S6]=1'b1;
                  else                           next[S5]=1'b1;
      state[S5] : if      (jmp)                  next[S3]=1'b1;
                  else if (!sk1 && !sk0 && !jmp) next[S6]=1'b1;
                  else if (!sk1 &&  sk0 && !jmp) next[S7]=1'b1;
                  else if ( sk1 && !sk0 && !jmp) next[S8]=1'b1;
                  else                           next[S9]=1'b1;
      state[S6] : if      (jmp)                  next[S3]=1'b1;
                  else if (go && !jmp)           next[S7]=1'b1;
                  else                           next[S6]=1'b1;
      state[S7] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S8]=1'b1;
      state[S8] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S9]=1'b1;
      state[S9] : if (jmp)                       next[S3]=1'b1;
                  else                           next[S0]=1'b1;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (1'b1)
        next[S0], next[S2], next[S4], next[S5] : ; // default outputs
        next[S7] :   y3 <= 1'b1;
        next[S1] :   y2 <= 1'b1;
        next[S3] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                   end
        next[S8] : begin
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
        next[S6], next[S9] : begin
                     y1 <= 1'b1;
                     y2 <= 1'b1;
                     y3 <= 1'b1;
                   end
      endcase
    end
endmodule
```

Example 9 - fsm_cc8 design - case (1'b1) onehot style - 86 lines of code

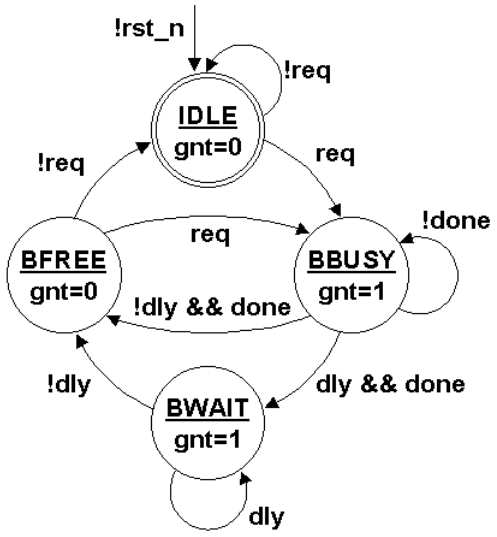
This is the only coding style where I recommend using full_case and parallel_case statements. The parallel case statement tells the synthesis tool to not build a priority encoder even though in theory, more than one of the state bits could be set (as engineers, we know that this is a onehot FSM and that only one bit can be set so no priority encoder is required). The value of the full_case statement is still in question.

8. Registered FSM Outputs (Good Style)

Registering the outputs of an FSM design insures that the outputs are glitch-free and frequently improves synthesis results by standardizing the output and input delay constraints of synthesized modules (see reference [1] for more information).

FSM outputs are easily registered by adding a third always sequential block to an FSM module where output assignments are generated in a case statement with case items corresponding to the next state that will be active when the output is clocked.

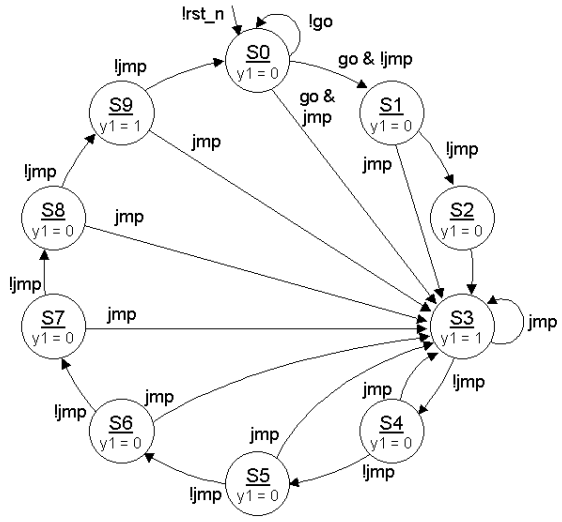
```
module fsm_cc4_2r
  (output reg gnt,
   input dly, done, req, clk, rst_n);

  parameter [1:0] IDLE  = 2'b00,
                  BBUSY = 2'b01,
                  BWAIT = 2'b10,
                  BFREE = 2'b11;

  reg [1:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= IDLE;
    else        state <= next;

  always @(state or dly or done or req) begin
    next = 2'bx;
    case (state)
      IDLE : if (req)       next = BBUSY;
             else           next = IDLE;
      BBUSY: if (!done)     next = BBUSY;
             else if ( dly) next = BWAIT;
             else           next = BFREE;
      BWAIT: if (!dly)      next = BFREE;
             else           next = BWAIT;
      BFREE: if (req)       next = BBUSY;
             else           next = IDLE;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) gnt <= 1'b0;
    else begin
      gnt <= 1'b0;
      case (next)
        IDLE,  BFREE: ; // default outputs
        BBUSY, BWAIT: gnt <= 1'b1;
      endcase
    end
endmodule
```



Example 10 - fsm_cc4 design - three always blocks w/registered outputs - 40 lines of code

8.1 10-state simple FSM design - three always blocks - registered outputs

Example 11 is the fsm_cc7 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc7 design requires 60 lines of code (coding requirements are compared in a later section).



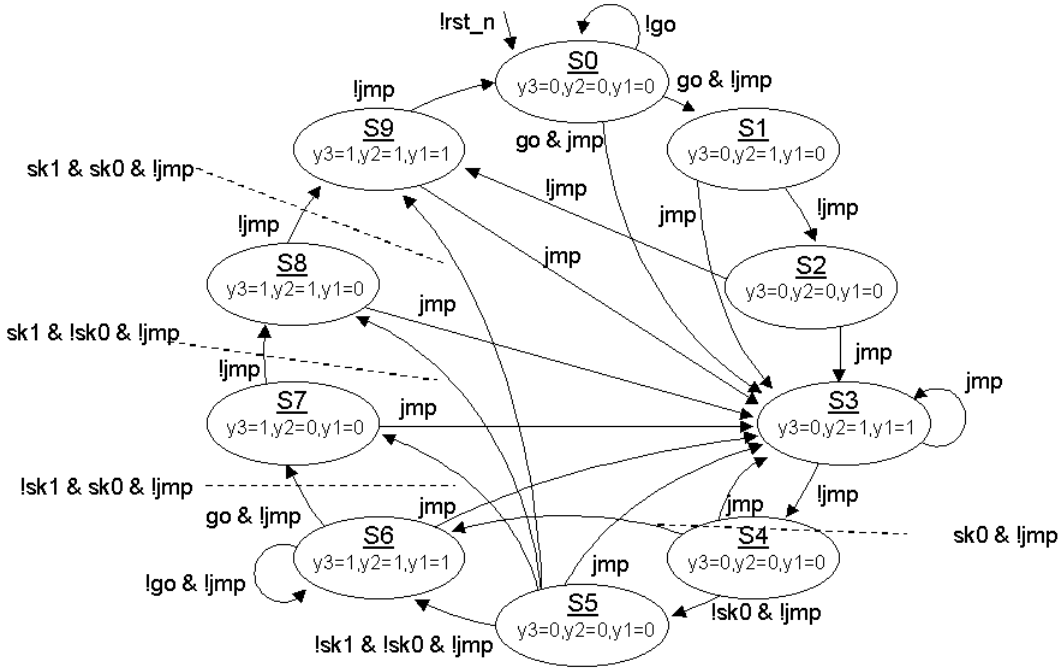

```
module fsm_cc7_3r
  (output reg y1,
   input      jmp, go, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or go or jmp) begin
    next = 4'bx;
    y1 = 1'b0;
    case (state)
      S0 : if (!go)      next = S0;
           else if (jmp) next = S3;
           else          next = S1;
      S1 : if (jmp)      next = S3;
           else          next = S2;
      S2 :               next = S3;
      S3 : begin y1 = 1'b1;
             if (jmp)    next = S3;
             else        next = S4;
           end
      S4 : if (jmp)      next = S3;
           else          next = S5;
      S5 : if (jmp)      next = S3;
           else          next = S6;
      S6 : if (jmp)      next = S3;
           else          next = S7;
      S7 : if (jmp)      next = S3;
           else          next = S8;
      S8 : if (jmp)      next = S3;
           else          next = S9;
      S9 : if (jmp)      next = S3;
           else          next = S0;
    endcase
  end

  always @(posedge clk or negedge rst_n)
    if (!rst_n) y1 <= 1'b0;
    else begin
      y1 <= 1'b0;
      case (state)
        S0, S1, S2, S4, S5, S6, S7, S8, S9:; // default
```

```
        S3 : y1 <= 1'b1;
      endcase
    end
endmodule
```

Example 11 - fsm_cc7 design - three always blocks w/registered outputs - 60 lines of code

8.2 10-state moderately complex FSM design - three always blocks - registered outputs

Example 12 is the fsm_cc8 design with registered outputs implemented with three always blocks. Using three always blocks, the fsm_cc8 design requires 83 lines of code (coding requirements are compared in a later section).




```
module fsm_cc8_3r
  (output reg y1, y2, y3,
   input      jmp, go, sk0, sk1, clk, rst_n);

  parameter S0  = 4'b0000,
            S1  = 4'b0001,
            S2  = 4'b0010,
            S3  = 4'b0011,
            S4  = 4'b0100,
            S5  = 4'b0101,
            S6  = 4'b0110,
            S7  = 4'b0111,
            S8  = 4'b1000,
            S9  = 4'b1001;

  reg [3:0] state, next;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) state <= S0;
    else        state <= next;

  always @(state or jmp or go or sk0 or sk1) begin
```

```
    next = 4'bx;
    case (state)
      S0 : if      (!go)                 next = S0;
           else if (jmp)                 next = S3;
           else                          next = S1;
      S1 : if (jmp)                      next = S3;
           else                          next = S2;
      S2 : if (jmp)                      next = S3;
           else                          next = S9;
      S3 : if (jmp)                      next = S3;
           else                          next = S4;
      S4 : if      (jmp)                 next = S3;
           else if (sk0 && !jmp)         next = S6;
           else                          next = S5;
      S5 : if      (jmp)                 next = S3;
           else if (!sk1 && !sk0 && !jmp) next = S6;
           else if (!sk1 &&  sk0 && !jmp) next = S7;
           else if ( sk1 && !sk0 && !jmp) next = S8;
           else                          next = S9;
      S6 : if      (jmp)                 next = S3;
           else if (go && !jmp)          next = S7;
           else                          next = S6;
      S7 : if (jmp)                      next = S3;
           else                          next = S8;
      S8 : if (jmp)                      next = S3;
           else                          next = S9;
      S9 : if (jmp)                      next = S3;
           else                          next = S0;
    endcase
  end

always @(posedge clk or negedge rst_n)
    if (!rst_n) begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
    end
    else begin
      y1 <= 1'b0;
      y2 <= 1'b0;
      y3 <= 1'b0;
      case (next)
        S0, S2, S4, S5 : ; // default outputs
        S7             :   y3 <= 1'b1;
        S1             :   y2 <= 1'b1;
        S3             : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                         end
        S8             : begin
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
        S6, S9         : begin
                           y1 <= 1'b1;
                           y2 <= 1'b1;
                           y3 <= 1'b1;
                         end
    endcase
  end
endmodule
```

Example 12 - fsm_cc8 design - three always blocks w/registered outputs - 83 lines of code

9. Comparing RTL Coding Efforts

In the preceding sections, three different FSM designs were coded four different ways: (1) two always block coding style, (2) one always block coding style, (3) onehot, two always block coding style, and (4) three always block coding style with registered outputs.

	Two always block coding style	One always block coding style (12%-83% larger)	Onehot, two always block coding style	Three always block coding style w/ registered outputs
fsm_cc4 (4 states, simple)	37 lines of code	47 lines of code (12%-27% larger)	42 lines of code	40 lines of code
fsm_cc7 (10 states, simple)	50 lines of code	79 lines of code (32%-58% larger)	53 lines of code	60 lines of code
fsm_cc8 (10 states, moderate complexity)	80 lines of code	146 lines of code (70%-83% larger)	86 lines of code	83 lines of code

Table 1 - Lines of RTL code required for different FSM coding styles

From Table 1, we see that the one always block FSM coding style is the least efficient coding style with respect to the amount of RTL code required to render an equivalent design. In fact, the more outputs that an FSM design has and the more transition arcs in the FSM state diagram, the faster the one always block coding style increases in size over comparable FSM coding styles.

If you are a contractor or are paid by the line-of-code, clearly, the one always block FSM coding style should be your preferred style. If you are trying to complete a project on time and code the design in a concise manner, the one always block coding style should be avoided.

10. Synthesis Results

Synthesis results were not complete by the time the paper was submitted for publication.

11. Running Cadence BuildGates

```
ac_shell          (for command-line mode)
ac_shell -gui &   (for GUI mode with process running  in background)
```

12. Verilog-2001 Enhancements

As of this writing, the Cadence Verilog simulators do not support many (if any) of the new Verilog-2001 enhancements. All of the preceding examples were coded with Verilog-2001 enhanced and concise ANSI-style module headers. In reality, to make the designs work with the Cadence Verilog simulators, I had to also code Verilog-1995 style module headers and select the appropriate header using macro definitions. To ease the task, I have created two aliases for 1995-style Verilog simulations.

```
alias ncverilog95 "ncverilog +define+V95"
alias verilog95 "verilog +define+V95"
```

12.1 ANSI-Style port declarations

ANSI-style port declarations are a nice enhancement to Verilog-2001 but they are not yet supported by version 3.4 of NC-Verilog or Verilog-XL, but they are reported to work with BuildGates. This enhancement permits module headers to be declared in a much more concise manner over traditional Verilog-1995 coding requirements.

Verilog-1995 required each module port be declared two or three times. Verilog-1995 required that (1) the module ports be listed in the module header, (2) the module port directions be declared, and (3) for reg-variable output ports, the port data type was also required.

Verilog-2001 combined all of this information into single module port declarations, significantly reducing the verbosity and redundancy of Verilog module headers. Of the major Verilog vendors, only the Cadence Verilog simulators do not support this Verilog-2001 feature. This means that users who want to take advantage of this feature and who use simulators from multiple vendors, including Cadence, must code both styles of module headers using `ifdef statements to select the appropriate module header style.

I prefer the following coding style to support retro-style Verilog simulators:

```
`ifdef V95
  // Verilog-1995 old-style, verbose module headers
`else
  // Verilog-2001 new-style, efficient module headers
`endif
```

The following example is from the actual fsm_cc4_1.v file used to test one always block FSM coding styles in this paper.

```
`ifdef V95
module fsm_cc4_1 (gnt, dly, done, req, clk, rst_n);
  output gnt;
  input  dly, done, req;
  input  clk, rst_n;
  reg    gnt;
`else
module fsm_cc4_1
  (output reg gnt,
   input dly, done, req, clk, rst_n);
`endif
```

It should be noted that this is an easy enhancement to implement, significantly improves the coding efficiency of module headers and that some major Verilog vendors have supported this enhanced coding style for more than a year at the time this paper was written. The author strongly encourages Cadence simulator developers to quickly adopt this Verilog-2001 enhancement to ease the Verilog coding burden for Cadence tool users.

12.2 @* Combinational sensitivity list

Verilog-2001 added the much-heralded @* combinational sensitivity list token. Although the combinational sensitivy list could be written using any of the following styles:

```
always @*
always @(*)
always @( * )
always @ ( * )
```

or any other combination of the characters @ (*) with or without white space, the author prefers the first and most abbreviated style. To the author, "always @*" clearly denotes that a combinational block of logic follows.

The Verilog-2001 "always @*" coding style has a number of important advantages over the more cumbersome Verilog-1995 combinational sensitivity list coding style:

- Reduces coding errors - the code informs the simulator that the intended implementation is combinational logic, so the simulator will automatically add and remove signals from the sensitivity list as RTL code is added or deleted from the combinational always block. The RTL coder is no longer burdened with manually insuring that all of the necessary signals are present in the sensitivity list. This will reduce coding errors that do not show up until a synthesis tool or linting tool reports errors in the sensitivity list. The basic intent of this enhancement is to inform the simulator, "if the synthesis tool wants the signals, so do we!"

- Abbreviated syntax - large combinational blocks often meant multiple lines of redundant signal naming in a sensitivity list. The redundancy served no appreciable purpose and users will gladly adopt the more concise and abbreviated @* syntax.

- Clear intent - an always @* procedural block informs the code-reviewer that this block is intended to behave like, and synthesize to, combinational logic.

13. SystemVerilog Enhancements

In June of 2002, Accellera released the SystemVerilog 3.0 language specification, a superset of Verilog-2001 with many nice enhancements for modeling, synthesis and verification. The basis for the SystemVerilog language comes from a donation by CoDesign Automation of significant portions of their Superlog language.

Key functionality that has been added to the Accellera SystemVerilog 3.0 Specification to support FSM design includes:

Enumerated types - Why do engineers want to use enumerated types? (1) Enumerated types permit abstract state declaration without defining the state encodings, and (2) enumerated types can typically be easily displayed in a waveform viewer permitting faster design debug. Enumerated types allow abstract state definitions without required state encoding assignments. Users also wanted the ability to assign state encodings to control implementation details such as output encoded FSM designs with simple registered outputs.

One short coming of traditional enumerated types was the inability to make X-state assignments. As discussed earlier in this paper, X-state assignments are important to simulation debug and synthesis optimization. SystemVerilog enumerated types will permit data type declaration, making it possible to declare enumerated types with an all-X's definitions.

Other SystemVerilog proposals under consideration for FSM enhancement include:

Different enumerated styles - the ability to declare different enumerated styles, such as enum_onehot, to make experimentation with different encoding styles easier to do. Currently, when changing from a binary encoding to an efficient onehot encoding style, 8 different code changes must be made in the FSM module. Wouldn't it be nice if the syntax permitted easier handling of FSM styles without manual intervention.

Transition statement and ->> next state transition operator -

These enhancements were removed from the SystemVerilog 3.0 Standard only because their definition was not fully elaborated and understood. Some people like the idea of a next-state transition operator that closely corresponds to the transition arcs that are shown on an FSM state diagram.

The infinitely abusable "goto" statement - Concern about a "goto" statement that could "cause spaghetti-code" could be avoided by limiting a goto-transition to a label within the same procedural block. Implicit FSM coding styles are much cleaner with a goto statement. A goto statement combined with a carefully crafted disable statement makes reset handling easier to do. A goto statement alleviates the problem of multiple transition arcs within a traditional implicit FSM design. Goto is just a proposal and may not pass.

14. Conclusions

There are many ways to code FSM designs. There are many inefficient ways to code FSM designs!

Use parameters to define state encodings. Parameters are constants that are local to a module. After defining the state encodings at the top of the FSM module, never use the state encodings again in the RTL code. This makes it possible to easily change the state encodings in just one place, the parameter definitions, without having to touch the rest of the FSM RTL code. This makes state-encoding-experimentation easy to do.

Use a two always block coding style to code FSM designs with combinational outputs. This style is efficient and easy to code and can also easily handle Mealy FSM designs.

Use a three always block coding style to code FSM designs with registered outputs. This style is efficient and easy to code. Note, another recommended coding style for FSM designs with registered outputs is the "output encoded" FSM coding style (see reference [1] for more information on this coding style).

Avoid the one always block FSM coding style. It is generally more verbose than an equivalent two always block coding style, output assignments are more error prone to coding mistakes and one cannot code asynchronous Mealy outputs without making the output assignments with separate continuous assign statements.

15. Acknowledgements

I would like to especially thank both Rich Owen and Nasir Junejo of Cadence for their assistance and tips enabling the use of the BuildGates synthesis tool. Their input helped me to achieve very favorable results in a short period of time.

16. References

[1] Clifford E. Cummings, "Coding And Scripting Techniques For FSM Designs With Synthesis-Optimized, Glitch-Free Outputs," *SNUG'2000 Boston (Synopsys Users Group Boston, MA, 2000) Proceedings*, September 2000. (Also available online at www.sunburst-design.com/papers)

[2] Clifford E. Cummings, '"full_case parallel_case", the Evil Twins of Verilog Synthesis,' *SNUG'99 Boston (Synopsys Users Group Boston, MA, 1999) Proceedings*, October 1999. (Also available online at www.sunburst-design.com/papers)

[3] Clifford E. Cummings, "New Verilog-2001 Techniques for Creating Parameterized Models (or Down With `define and Death of a defparam!)," *International HDL Conference 2002 Proceedings*, pp. 17-24, March 2002. (Also available online at www.sunburst-design.com/papers)

[4] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG'2000 Boston (Synopsys Users Group San Jose, CA, 2000) Proceedings*, March 2000. (Also available online at www.sunburst-design.com/papers)

[5] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Computer Society, IEEE Std 1364-1995, pg. 47, section 5.4.1 - Determinism.

[6] Nasir Junejo, personal communication

[7] Rich Owen, personal communication

[8] The Programmable Logic Data Book, Xilinx, 1994, pg. 8-171

[9] William I. Fletcher, An Engineering Approach To Digital Design, New Jersey, Prentice-Hall, 1980

[10] Zvi Kohavi, Switching And Finite Automota Theory, Second Edition, New York, McGraw-Hill Book Company, 1978

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 20 years of ASIC, FPGA and system design experience and ten years of Verilog, synthesis and methodology training experience.

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

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

E-mail Address: cliffc@sunburst-design.com
An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers (Data accurate as of July 22nd, 2002)

State Machine Coding Styles for Synthesis

Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

This paper details efficient Verilog coding styles to infer synthesizable state machines. HDL considerations such as advantages and disadvantages of one-always block FSMs Vs. two-always block FSMs are described.

Introduction

Steve Golson's 1994 paper, "State Machine Design Techniques for Verilog and VHDL" [1], is a great paper on state machine design using Verilog, VHDL and Synopsys tools. Steve's paper also offers in-depth background concerning the origin of specific state machine types.

This paper, "State Machine Coding Styles for Synthesis," details additional insights into state machine design including coding style approaches and a few additional tricks.

State Machine Classification

There are two types of state machines as classified by the types of outputs generated from each. The first is the Moore State Machine where the outputs are only a function of the present state, the second is the Mealy State Machine where one or more of the outputs are a function of the present state and one or more of the inputs.

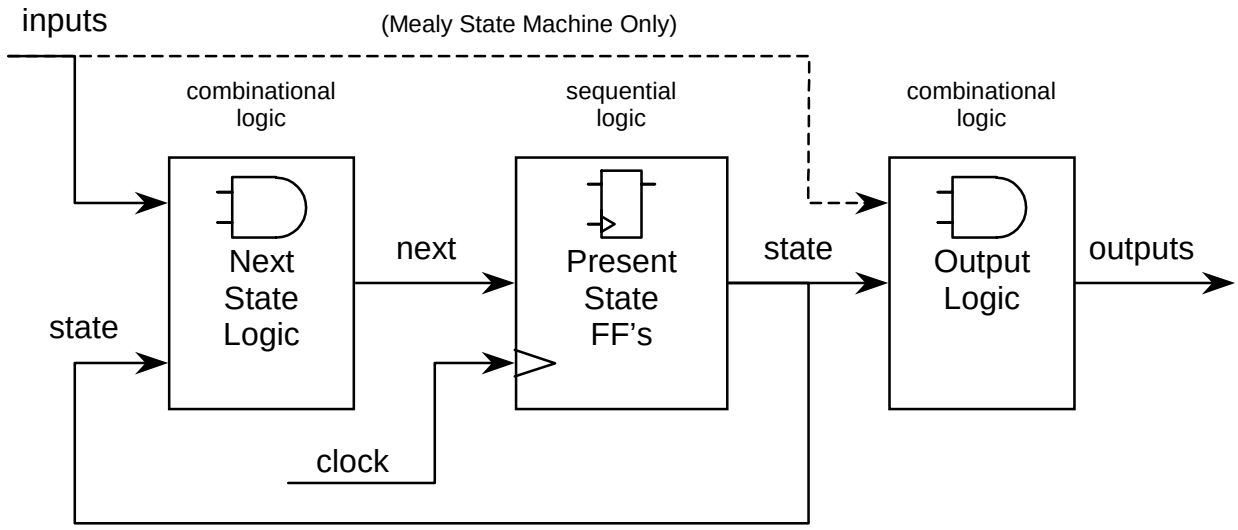


Figure 1 - FSM Block Diagram

In addition to classifying state machines by their respective output-generation type, state machines are also often classified by the state encoding employed by each state machine. Some of the more common state encoding styles include [1] [2] [3]: highly-encoded binary (or binary-sequential), gray-code, Johnson, one-hot, almost one-hot and one-hot with zero-idle (note: in the absence of a known official designation for the last encoding-style listed, the author selected the "one-hot with zero-idle" title. A more generally accepted name may exist).

Using the Moore FSM state diagram shown in Figure 2, this paper will detail synthesizable Verilog coding styles for highly-encoded binary, one-hot and one-hot with zero-idle state machines. This paper also details usage of the Synopsys FSM Tool to generate binary, gray and one-hot state machines. Coded examples of the three coding styles for the state machine in Figure

2, plus an example with the correct Synopsys FSM Tool comments, have been included at the end of this paper.

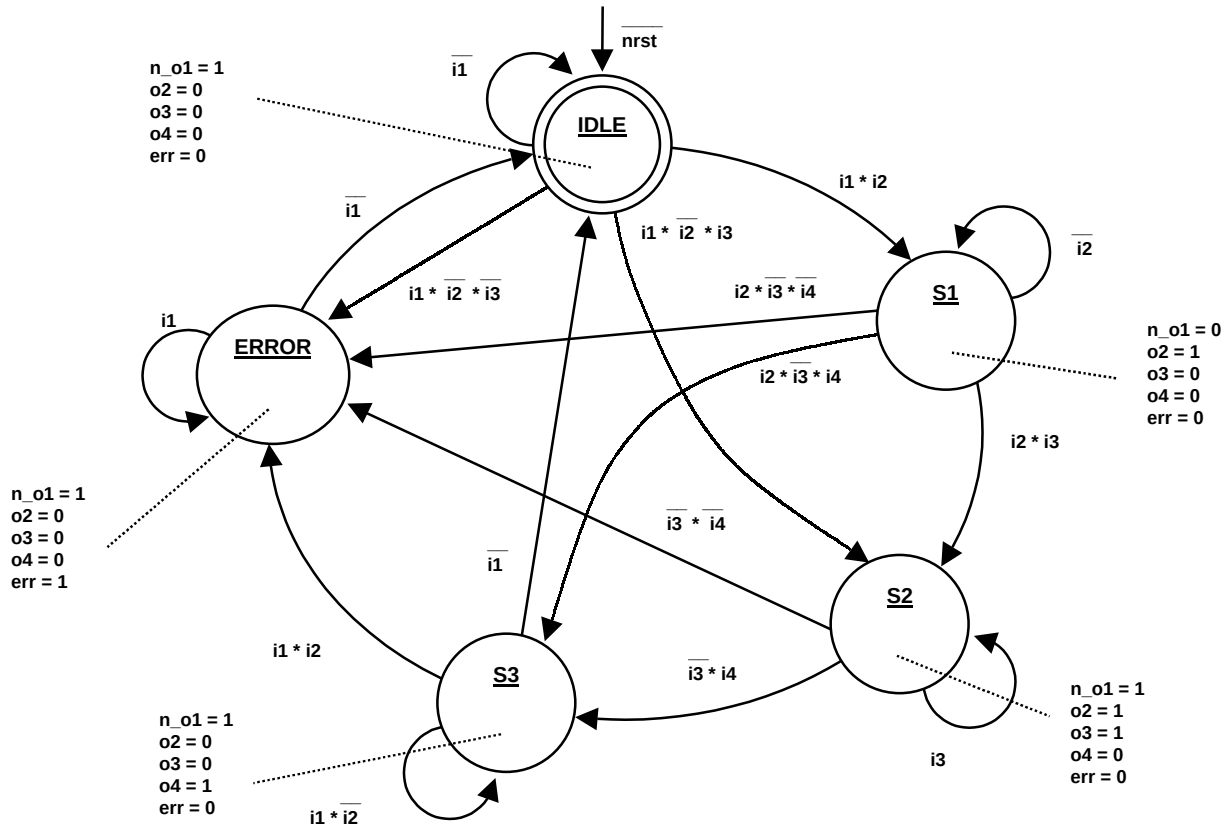


Figure 2 - Benchmark 1 (bm1) State Diagram

FSM Verilog Modules

Guideline: make each state machine a separate Verilog module.

Keeping each state machine separate from other synthesized logic simplifies the tasks of state machine definition, modification and debug. There are also a number of EDA tools that assist in the design and documentation of FSMs, but in general they only work well if the FSM is not mingled with other logic-code.

State Assignments

Guideline: make state assignments using parameters with symbolic state names.

Defining and using symbolic state names makes the Verilog code more readable and eases the task of redefining states if necessary. Examples 1-3 show binary, one-hot and one-hot with zero-idle parameter definitions for the FSM state diagram in Figure 2.

```
parameter [2:0] // synopsys enum code
                  IDLE = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                 ERROR = 3'd4;
```

Example 1 - Parameter definitions for binary encoding

```
parameter [4:0] IDLE = 5'b00001,
                    S1 = 5'b00010,
                    S2 = 5'b00100,
                    S3 = 5'b01000,
                 ERROR = 5'b10000;
```

Example 2 - Parameter definitions for verbose one-hot encoding

```
parameter [4:0] IDLE = 5'd0,
                    S1 = 5'd1,
                    S2 = 5'd2,
                    S3 = 5'd3,
                 ERROR = 5'd4;
```

Example 3 - Parameter definitions for simplified one-hot encoding

The simplified one-hot encoding shown Example 3 uses decimal numbers to index into the state register. This technique permits comparison of single bits as opposed to comparing against the entire state vector using the full state parameters shown in Example 2.

```
parameter [4:1] // ERROR is 4'b0000
                  IDLE = 4'd1,
                    S1 = 4'd2,
                    S2 = 4'd3,
                    S3 = 4'd4;
```

Example 4 - Parameter definitions for one-hot with zero-idle encoding

The one-hot with zero-idle encoding can yield very efficient FSMs for state machines that have many interconnections with complex equations, including a large number of connections to one particular state. Frequently, multiple transitions are made either to an IDLE state or to another common state (such as the ERROR-state in this example).

One could also define symbolic state names using the macro-definition compiler directives (`define), but `define creates a global definition (from the point where the definition is read in the Verilog-code input stream). Unlike `define constants, parameters are constants local to the module where they are declared, which allows a design to have multiple FSMs with duplicate state names, such as IDLE or READ, each with a unique state encoding.

Occasionally, FSM code is written with parameter-defined state definitions, but subsequent code still includes explicit binary state encodings elsewhere in the module. This defeats the purpose of using symbolically labeled parameters. Only use the pre-defined parameter names for state testing and next-state assignment.

Additional notes on experimenting with different state definitions using Synopsys generated binary, gray and one-hot encodings are detailed in the section, "Synopsys FSM Tool."

Two-Always Block State Machine

A synthesizable state machine may be coded many ways. Two of the most common, easily understood and efficient methods are two-always block and one-always block state machines.

The easiest method to understand and implement is the two-always block state machine with output assignments included in either the combinational next-state always block or separate continuous-assignment outputs. This method partitions the Verilog code into the major FSM blocks shown in Figure 1: clocked present state logic, next state combinational logic and output combinational logic.

Sequential Always Block

Guideline: only use Verilog nonblocking assignments in the sequential always block.

Guideline: only use Verilog nonblocking assignments in all always blocks used to generate sequential logic.

For additional information concerning nonblocking assignments, see reference [4].

Verilog nonblocking assignments mimic the pipelined register behavior of actual hardware and eliminate many potential Verilog race conditions. Many engineers make nonblocking assignments using an intra-assignment timing delay (as shown in Example 5). There are two good reasons and one bad reason for using intra-assignment timing delays with nonblocking assignments.

Good reasons: (1) gives the appearance of a clk->q delay on a clocked register (as seen using a waveform viewer); (2) helps avoid hold-time problems when driving most gate-level models from an RTL model.

Bad reason: "we add delays because Verilog nonblocking assignments are broken!" - This is not true.

When implementing either a binary encoded or a verbose one-hot encoded FSM, on reset the state register will be assigned the IDLE state (or equivalent) (Example 5).

```
always @(posedge clk or posedge rst)
    if (rst)   state <= #1 IDLE;
    else       state <= #1 next;
```

Example 5 - Sequential always block for binary and verbose one-hot encoding

When implementing a simplified one-hot encoded FSM, on reset the state register will be assigned all zeros followed immediately by reassigning the IDLE bit of the state register (Example 6). Note, there are two nonblocking assignments assigning values to the same bit. This is completely defined by the IEEE Verilog Standard [5] and in this case, the last nonblocking assignment supercedes any previous nonblocking assignment (updating the IDLE bit of the state register).

```
always @(posedge clk or posedge rst)
    if (rst) begin
               state <= 5'b0;
               state[IDLE] <= 1'b1;
             end
    else       state <= next;
```

Example 6 - Sequential always block for simplified one-hot encoding

When implementing a one-hot with zero-idle encoded FSM, on reset the state register will be assigned all zeros (Example 7).

```
always @(posedge clk or posedge rst)
    if (rst) state <= 4'b0;
    else     state <= next;
```

Example 7 - Sequential always block one-hot with zero-idle encoding

Combinational Always Block

Guideline: only use Verilog blocking assignments in combinational always blocks.

Code a combinational always block to update the next state value. This always block is triggered by a sensitivity list that is sensitive to the state register from the synchronous always block and all of the inputs to the state machine.

Place a default next state assignment on the line immediately following the always block sensitivity list. This default assignment is updated by next-state assignments inside the case statement. There are three types of default next-state assignments that are commonly used: (1) next is set to all x's, (2) next is set to a predetermined recovery state such as IDLE, or (3) next is just set to the value of the state register.

By making a default next state assignment of x's, pre-synthesis simulation models will cause the state machine outputs to go unknown if not all state transitions have been explicitly assigned in the case statement. This is a useful technique to debug state machine designs, plus the x's will be treated as "don't cares" by the synthesis tool.

Some designs require an assignment to a known state as opposed to assigning x's. Examples include: satellite applications, medical applications, designs that use the FSM flip-flops as part of a diagnostic scan chain and designs that are equivalence checked with formal verification tools. Making a default next state assignment of either IDLE or all 0's typically satisfy these design requirements and making the initial default assignment might be easier than coding all of the explicit next-state transition assignments in the case statement.

Making the default next-state assignment equal to the present state is a coding style that has been used by PLD designers for years.

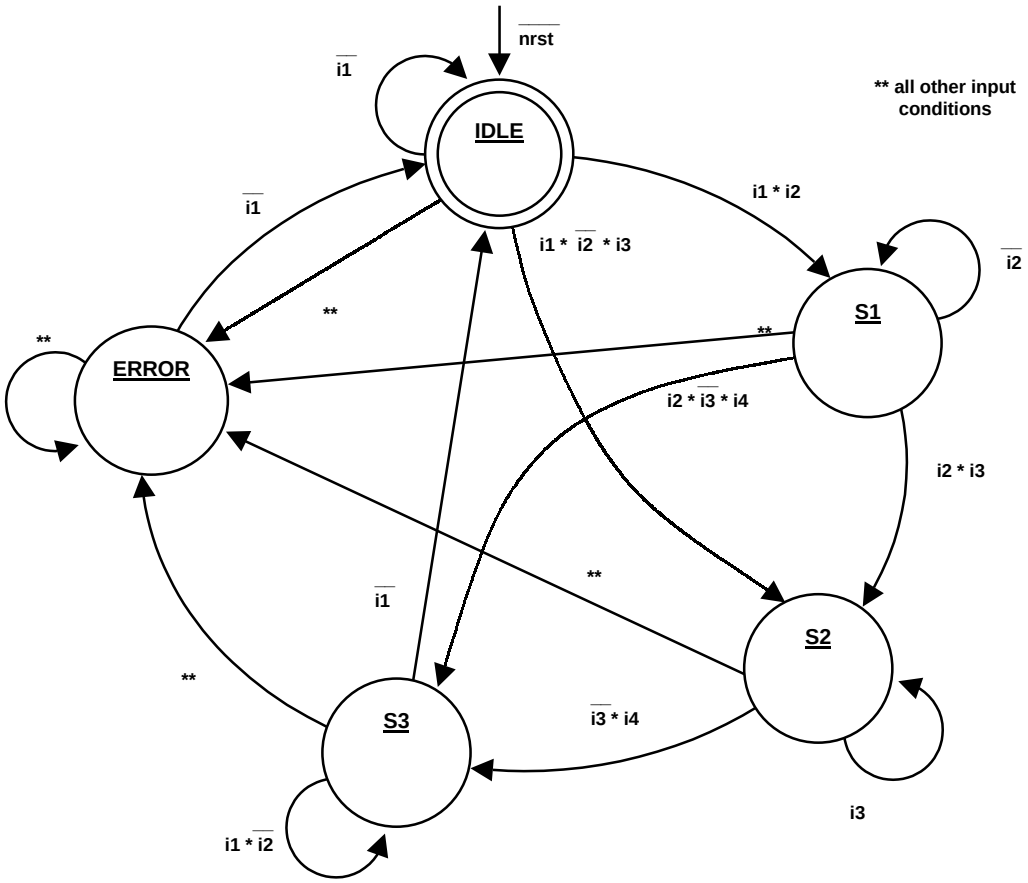


Figure 3 - Next State Transitions

Next state assignments are efficiently updated from within a case statement.

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 3'bx;
    case (state)
    IDLE: begin
                              next = ERROR;
          if (!i1)            next = IDLE;
          if (i1 & i2)        next = S1;
          if (i1 & !i2 & i3)  next = S2;
        end

    S1: ...
```

Example 8 - Next state assignments for binary and verbose one-hot encoding

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 5'b0;
    case (1'b1)  // synopsys full_case parallel_case
    state[IDLE]: begin
          if      (!i1) next[IDLE]  = 1'b1;
          else if ( i2) next[S1]    = 1'b1;
          else if ( i3) next[S2]    = 1'b1;
          else          next[ERROR] = 1'b1;
        end
    state[S1]: ...
```

Example 9 - Next state assignments for simplified one-hot encoding

```
always @(state or i1 or i2 or i3 or i4) begin
    next = 4'b0;
    case (1'b1)  // synopsys full_case parallel_case
    ~|state: begin // IDLE
          if ( i1 &  i2)       next[S1]    = 1'b1;
          if ( i1 & !i2 &  i3) next[S2]    = 1'b1;
          if ( i1 & !i2 & !i3) next[ERROR] = 1'b1;
        end
    state[S1]: ...
```

Example 10 - Next state assignments for simplified one-hot with zero-idle encoding

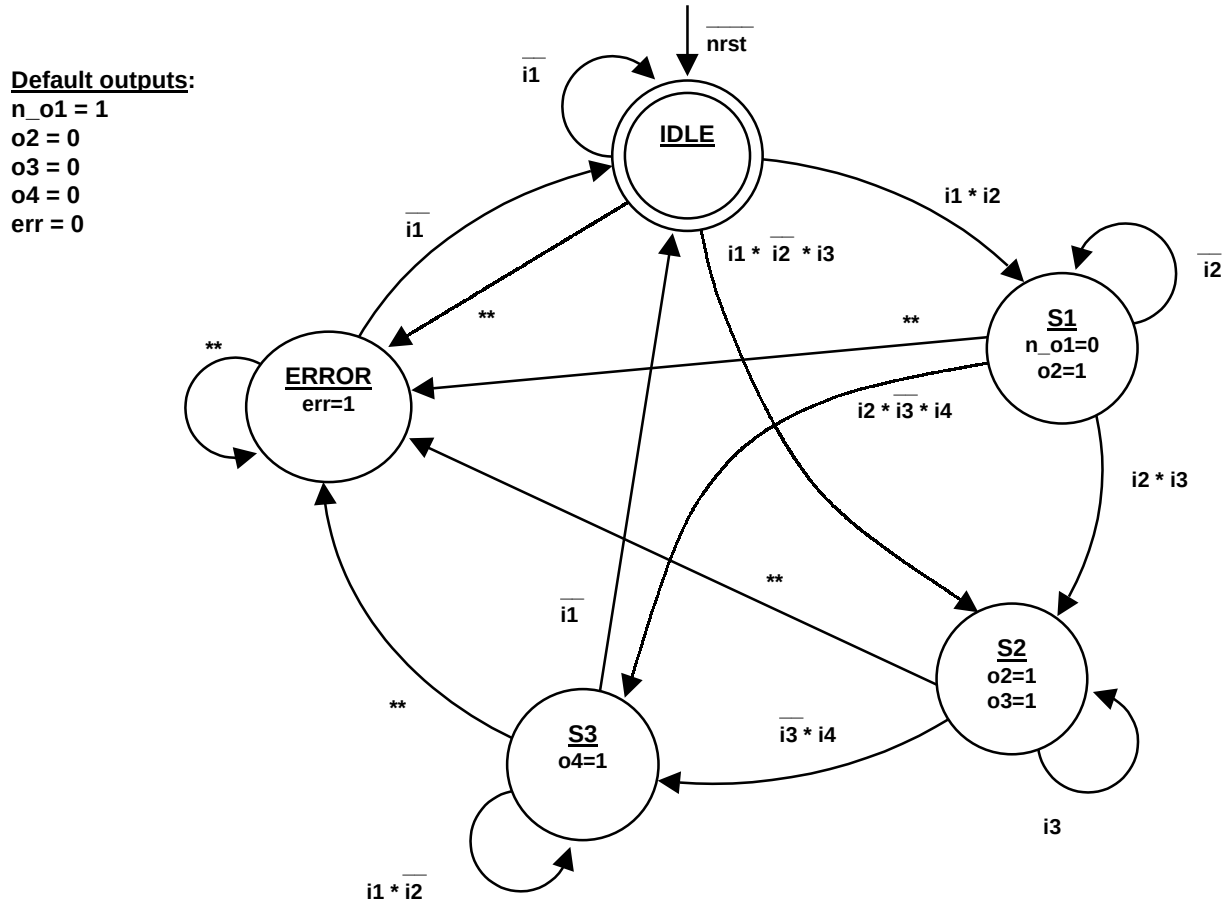


Figure 4 - State Diagram

FSM Output Generation

Code the output logic either as a separate block of continuous assignments or within the combinational logic always block. If the output assignments are coded as part of the combinational always block, output assignments could also be put into Verilog tasks with meaningful names, as shown in Figure 5. The tasks are called from within each state in the case statement.

Isolation of the output assignments makes changes to the output logic easy if modification is required. It also helps to avoid the creation of additional unwanted latches by the synthesis tool.

When placing output assignments inside the combinational always block of a Two-Always Block State Machine, make default output assignments at the top of the always block, then modify the appropriate output assignments in the case statement.

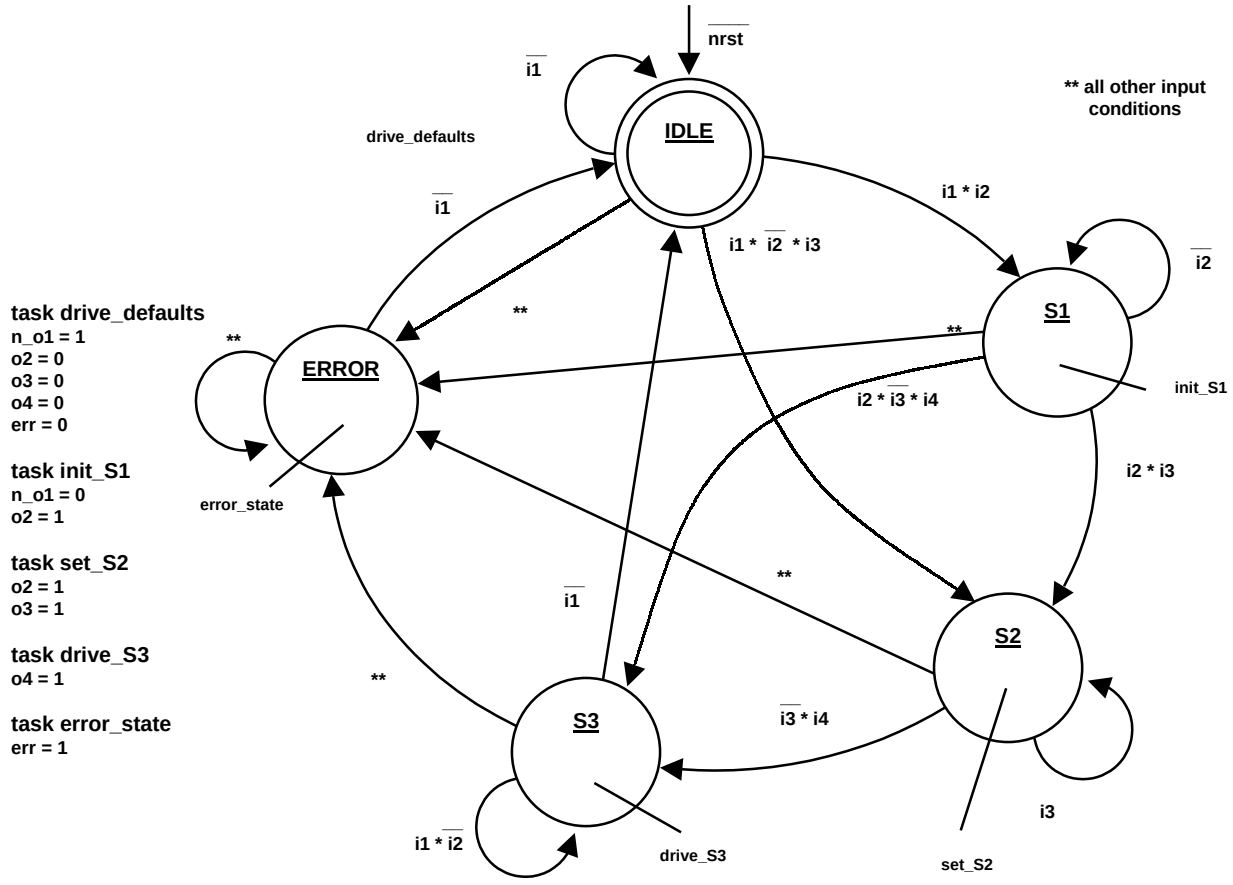


Figure 5 - Task State Outputs

In general this method requires less coding than making all output assignments for each state (case item) and highlights when outputs are supposed to change.

Mealy and Registered Outputs

Mealy outputs are easily added to the Verilog code either by qualifying an output continuous assignment:

```
assign rd_out = (state == READ) & !rd_strobe_n;
```

or by qualifying an output assignment in the combinational always block:

```
case (state)
...
READ: if (!rd_strobe_n) rd_out = 1'b1;
```

Registered outputs may be added to the Verilog code making assignments to an output using nonblocking assignments in a sequential always block. The FSM can be coded as one sequential always block or a second sequential always block can be added to the design.

One-Always Block State Machine

In general, the one-always block state machine is slightly more simulation-efficient than the two-always block state machine since the inputs are only examined on clock changes; however, this state machine can be more difficult to modify and debug.

When placing output assignments inside the always block of a one-always block state machine, one must consider the following:

Placing output assignments inside of the always block will infer output flip-flops. It must also be remembered that output assignments placed inside of the always block are "next output" assignments which can be more error-prone to code.

Note: output assignments inside of a sequential always block cannot be Mealy outputs.

Full_case / parallel_case

A case statement is a "select-one-of-many" construct in both Verilog and VHDL. A case statement is composed of the keyword, case, followed by a case expression that is compared to subsequent case items. The case items are tested against the case expression, one by one, in sequential order and when a match between the case expression and one of the case items is detected, the corresponding actions executed, the rest of the case items are skipped and program execution resumes with the first statement after the endcase statement.

```
case (case_expression (with 2^n possible combinations))
   case_item1 : <action #1>;
   case_item2 : <action #2>;
   case_item3 : <action #3>;
   ...
```

```
    case_item2^n-1: <action #2^n-1>;
    case_item2^n: <action #2^n>;
    default: <default action>;
endcase
```

A full case statement is defined to be a case statement where every possible input pattern is explicitly defined. A parallel case statement is defined to be a case statement with no overlapping conditions in the case items.

VHDL case statements are required to be "full," which means that every possible case item shall either be explicitly listed as a case item, or there must be an "others =>" clause after the last-defined case item. In practice, almost all VHDL case statements utilizing non bit-type data types include an "others =>" statement to cover the non-binary data patterns.

VHDL case statements are also required to be "parallel," which means that no case item shall overlap any other in the list of case items.

Verilog case statements are not required to be either "full" or "parallel."

Adding "// synopsys full_case" to the end of a case statement (before any case items are declared) informs the synthesis tool that all outputs from non-explicitly declared case items should be treated as "don't-cares" for synthesis purposes.

Adding "// synopsys parallel_case" to the end of a case statement (before any case items are declared) informs the synthesis tool that all case items should be tested individually, even if the case items overlap.

Adding either or both "// synopsys full_case parallel_case" directives to the Verilog FSM source code is generally beneficial when coding one-hot or one-hot with zero-idle FSMs. In these cases, it is given that only one bit of the state vector is set and that all other bit-pattern combinations should be treated as "don't cares." It is also given that there should be no overlap in the list of case items.

Note that the usage of full_case parallel case may cause pre-synthesis design simulations to differ from post-synthesis design simulations because these directives are effectively giving Synopsys tools information about the design that was not included in the original Verilog model.

Adding full_case parallel_case to every case statement in a design is not recommended. The practice can change the functionality of a design, and can also cause some binary encoded FSM designs to actually get larger and slower.

Synopsys FSM Tool

The Synopsys FSM tool can be used to experiment with different state encodings styles, such as binary, gray and one-hot codes. In order to use the FSM tool, the Verilog code must include Synopsys synthetic comments, plus a few unusual Verilog code statements. The Synopsys FSM tool is very strict about how these comments and code segments are ordered and it is very easy to code this incorrectly for the FSM tool.

First, the parameter must include a range (very unusual Verilog coding style). If no range is included in the parameter declaration, the error message "Declaration of enumeration type requires range specification" will be reported.

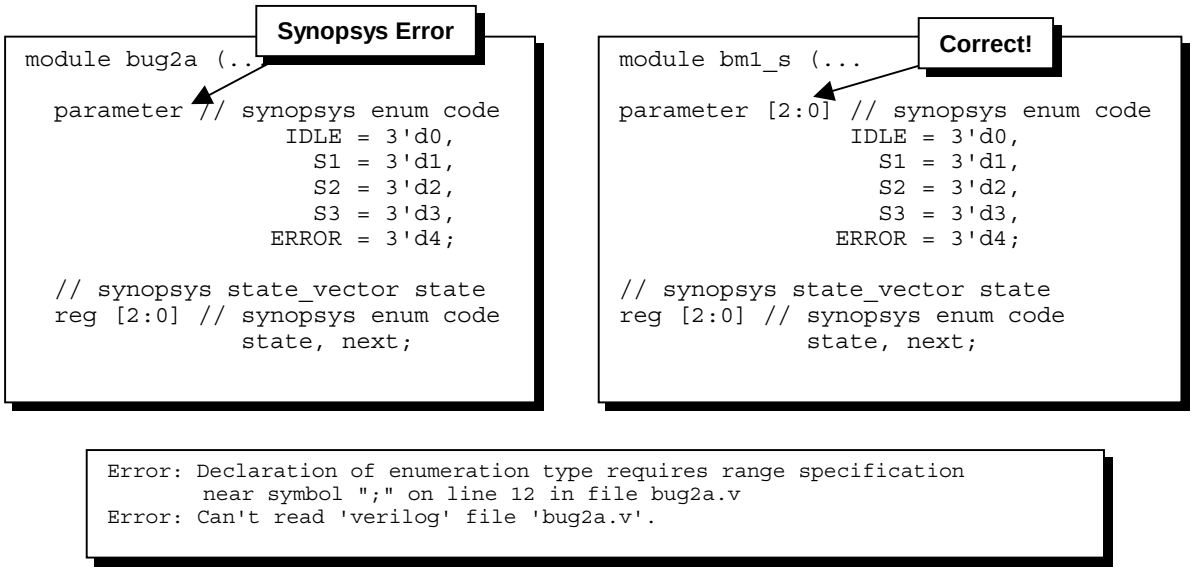


Figure 6 - FSM Tool - parameter range

Second, numeric parameter definitions must be sized, otherwise the FSM tool interprets all numbers as 32-bit numbers and reports an invalid encoding error.

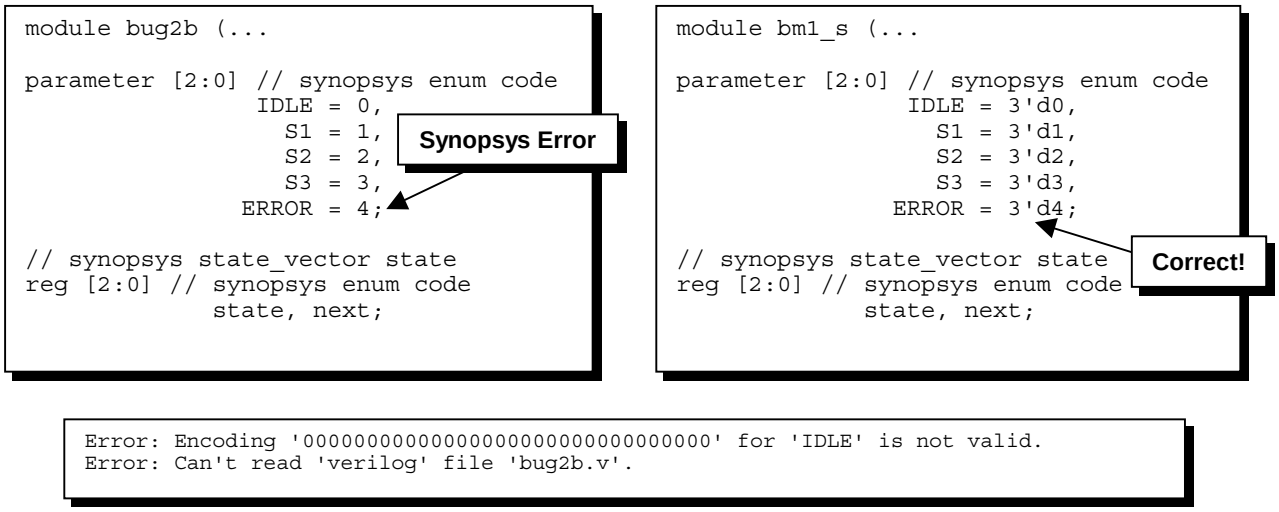


Figure 7 - FSM Tool - sized numbers

Third, the required placement of the Synopsys synthetic comments is exactly as shown. The "// synopsys enum <name>" must be placed after the parameter range declaration and before any of the parameters are declared,

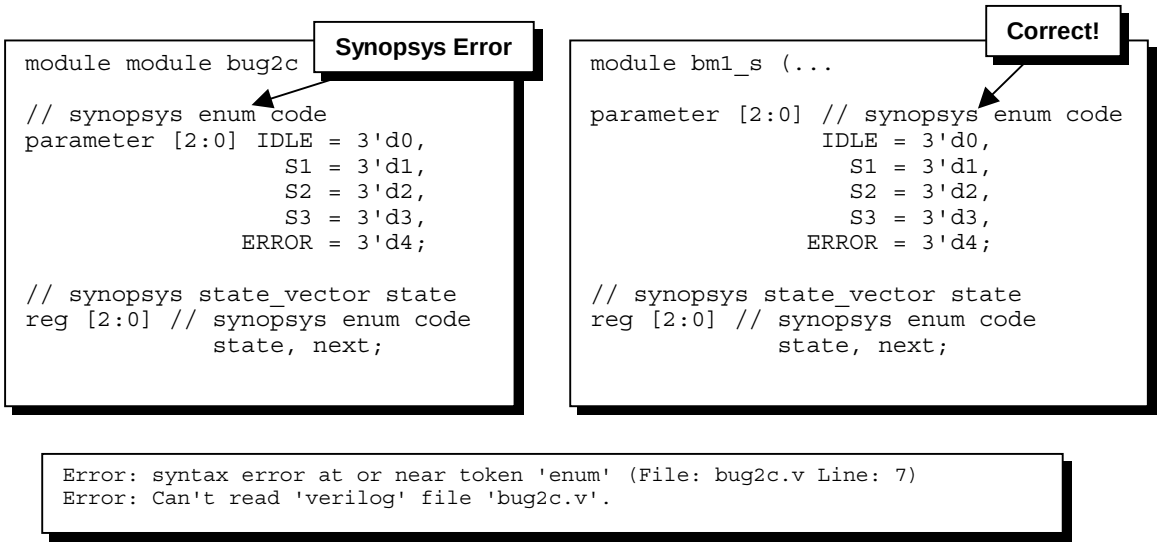


Figure 8 - FSM Tool - synopsys enum

a "/ synopsys state_vector <state_vector_name>" Synopsys comment must be placed immediately before the state-reg declaration and the exact same "// synopsys enum <name>" comment, used above, must be placed after the reg range declaration but before the state (and next) declarations. "

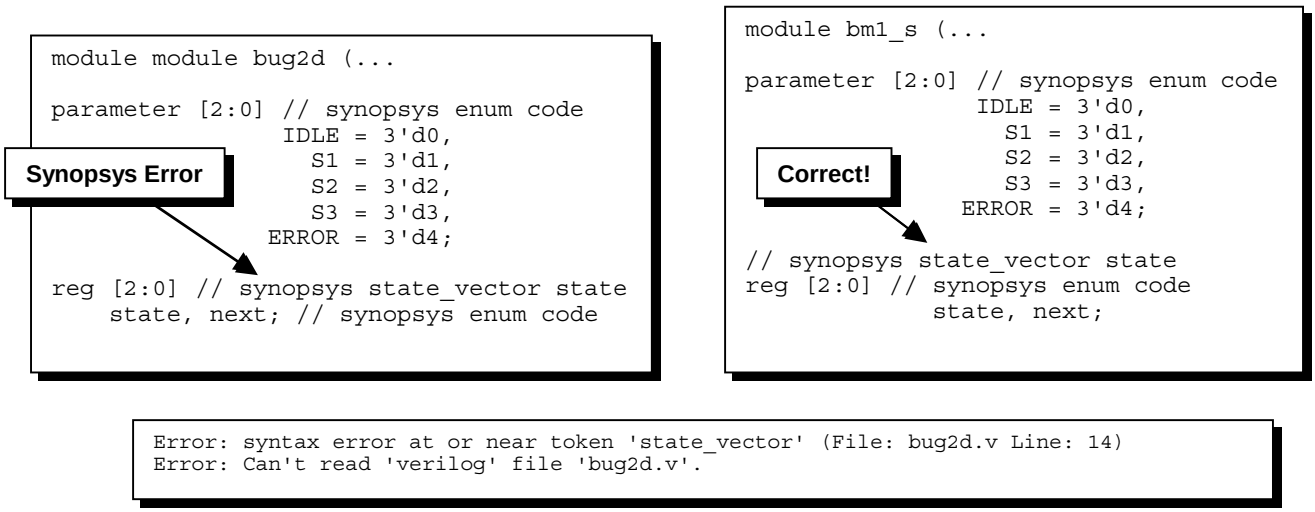


Figure 9 - FSM Tool - synopsys state_vector

Below are example dc_shell commands that are used to invoke the Synopsys FSM tools on a state machine design.

```
(read the design)
compile
extract
set_fsm_encoding_style binary
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_binary.db"
report_area > "rpt/" + DESIGN + "_fsm_binary.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_binary.rpt"

(read the design)
compile
extract
set_fsm_encoding_style gray
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_gray.db"
report_area > "rpt/" + DESIGN + "_fsm_gray.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_gray.rpt"

(read the design)
compile
extract
set_fsm_encoding_style one_hot
compile
write -f db -hier -output "db/" + DESIGN + "_fsm_onehot.db"
report_area > "rpt/" + DESIGN + "_fsm_onehot.rpt"
create_clock -p 0 clk
report_timing >> "rpt/" + DESIGN + "_fsm_onehot.rpt"
```

Example 11 - FSM Tool - dc_shell script

Acknowledgements

I would like to thank both Mike McNamara of Silicon Sorcery and Steve Golson of Trilobyte Systems for information and tips they have shared with me concerning Finite State Machine design. For more information about coding State Machines in both Verilog and VHDL, I highly recommend reading Steve's paper, "State Machine Design Techniques for Verilog and VHDL" [1].

References

[1] S. Golson, "State Machine Design Techniques for Verilog and VHDL," Synopsys Journal of High-Level Design, September 1994, pp. 1-48.

[2] Z. Kohavi, "Switching and Finite Automata Theory," McGraw-Hill Book Company, New York, 1978, pp. 275-321.

[3] D.J. Smith, "HDL Chip Design," Doone Publications, Madison, Alabama, 1997, pp. 193-270.

[4] C.E. Cummings, "Verilog Nonblocking Assignments Demystified," International Verilog HDL Conference Proceedings 1998.

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

Author & Contact Information

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

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

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

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

(Data accurate as of September 7th, 2001)

```
module bm1_s (err, n_o1, o2, o3, o4,
              i1, i2, i3, i4, clk, rst);
  output    err, n_o1, o2, o3, o4;
  input     i1, i2, i3, i4, clk, rst;
  reg       err, n_o1, o2, o3, o4;

  parameter [2:0] // synopsys enum code
                  IDLE = 3'd0,
                    S1 = 3'd1,
                    S2 = 3'd2,
                    S3 = 3'd3,
                 ERROR = 3'd4;

  // synopsys state_vector state
  reg [2:0] // synopsys enum code
              state, next;

  always @(posedge clk or posedge rst)
    if (rst)   state <= IDLE;
    else       state <= next;

  always @(state or i1 or i2 or i3 or i4) begin
    next = 3'bx;
    err = 0; n_o1 = 1;
    o2 = 0;  o3 = 0;  o4 = 0;
    case (state)
    IDLE: begin
                              next = ERROR;
          if (!i1)            next = IDLE;
          if (i1 & i2)        next = S1;
          if (i1 & !i2 & i3)  next = S2;
        end

    S1: begin
                              next = ERROR;
          if (!i2)            next = S1;
          if (i2 & i3)        next = S2;
          if (i2 & !i3 & i4)  next = S3;
          n_o1 = 0;
          o2 = 1;
        end

    S2: begin
                              next = ERROR;
          if (i3)             next = S2;
          if (!i3 & i4)       next = S3;
          o2 = 1;
          o3 = 1;
        end

    S3: begin
                              next = S3;
          if (!i1)            next = IDLE;
          if (i1 & i2)        next = ERROR;
          o4 = 1;
        end

    ERROR: begin
                              next = IDLE;
          if (i1)             next = ERROR;
          err = 1;
        end
    endcase
  end
endmodule
```

Synopsys FSM Tool synthetic comment

Highly encoded state-parameter definitions

Synopsys FSM Tool synthetic comments

next = 2'bx (synthesis "don't care" assignment)

Initial default output assignments

Default assignment followed by parallel if statements

Only update output assignments that change in each state

Figure 10 - FSM Tool synthetic comments

```
module bm1_1afp (err, n_o1, o2, o3, o4,
                 i1, i2, i3, i4, clk, rst);
  output  err, n_o1, o2, o3, o4;
  input   i1, i2, i3, i4, clk, rst;
  reg     err, n_o1, o2, o3, o4;

  parameter [4:0] IDLE = 5'b00001,
                    S1 = 5'b00010,
                    S2 = 5'b00100,
                    S3 = 5'b01000,
                 ERROR = 5'b10000;

  reg [4:0]  state, next;

  always @(posedge clk or posedge rst)
    if (rst) state <= IDLE;
    else     state <= next;

  always @(state or i1 or i2 or i3 or i4) begin
    next = 5'bx;
    err = 0; n_o1 = 1;
    o2 = 0;    o3 = 0;  o4 = 0;
    case (state)  // synopsys full_case parallel_case
    IDLE: begin
          if      (!i1) next = IDLE;
          else if ( i2) next = S1;
          else if ( i3) next = S2;
          else          next = ERROR;
        end

    S1: begin
          if      (!i2) next = S1;
          else if ( i3) next = S2;
          else if ( i4) next = S3;
          else          next = ERROR;
          n_o1 = 0;
          o2 = 1;
        end

    S2: begin
          if      ( i3) next = S2;
          else if ( i4) next = S3;
          else          next = ERROR;
          o2 = 1;
          o3 = 1;
        end

    S3: begin
          if      (!i1) next = IDLE;
          else if ( i2) next = ERROR;
          else          next = S3;
          o4 = 1;
        end

    ERROR: begin
          if (i1)       next = ERROR;
          else          next = IDLE;
          err = 1;
        end
    endcase
  end
endmodule
```

Verbose one-hot state-parameter definitions

next = 5'bx (synthesis "don't care" assignment)

Initial default output assignments

If/else-if statements

Final else statements

Only update output assignments that change in each state

Figure 11 - Verbose one-hot FSM

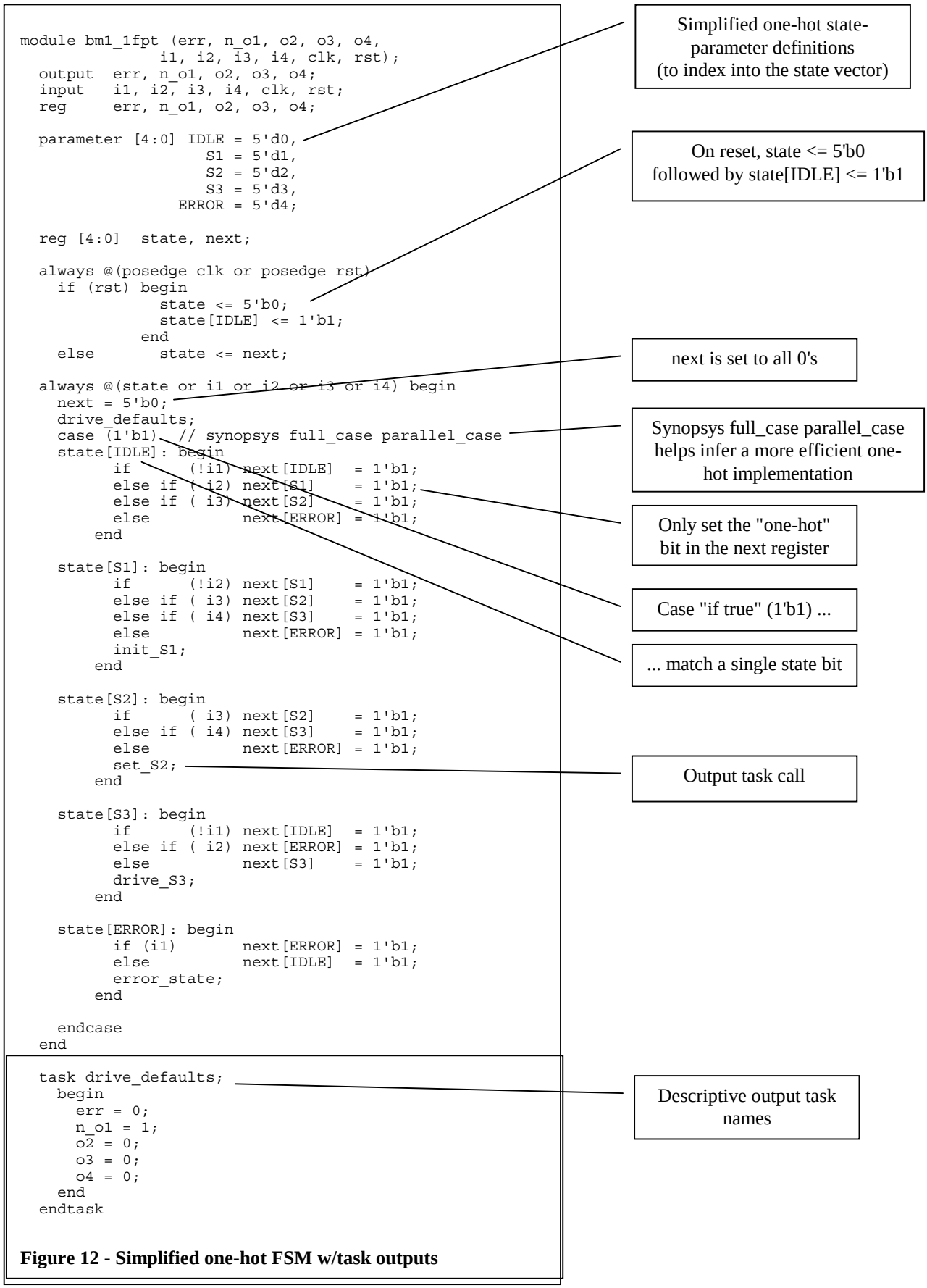


Figure 12 - Simplified one-hot FSM w/task outputs

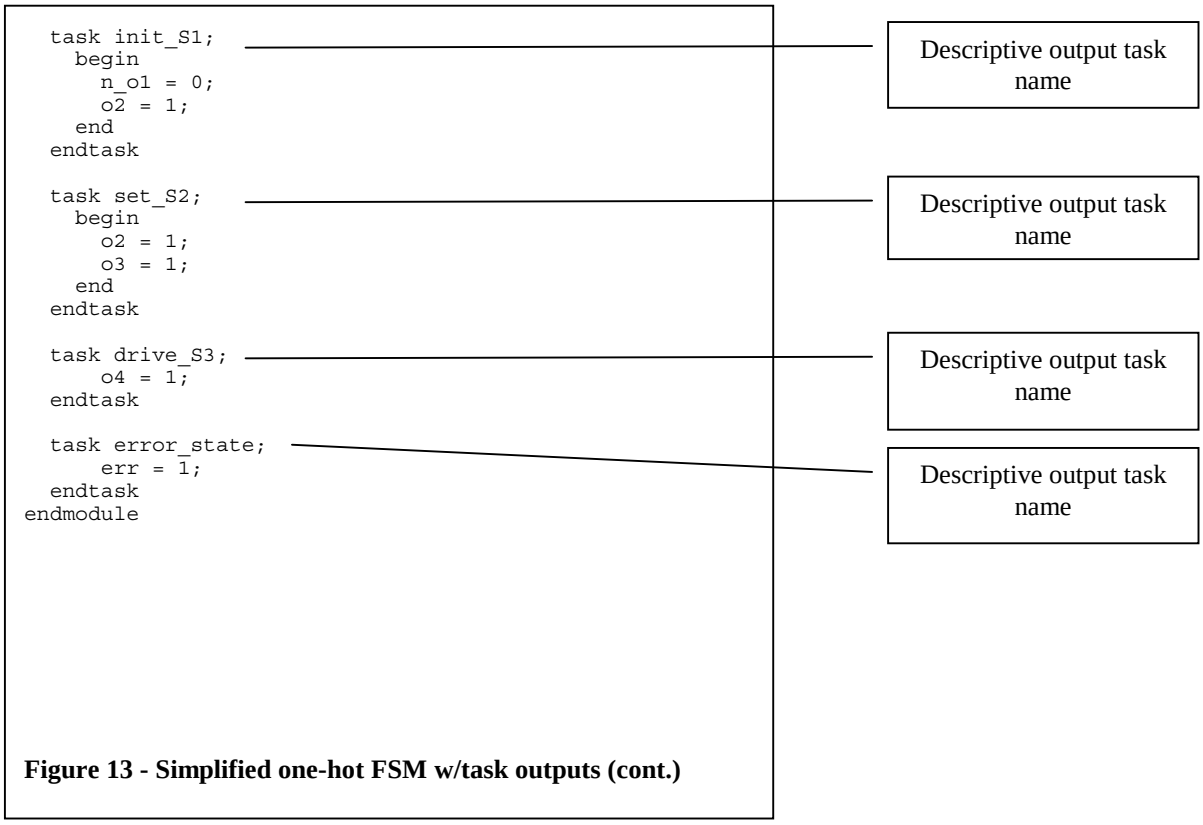


Figure 13 - Simplified one-hot FSM w/task outputs (cont.)

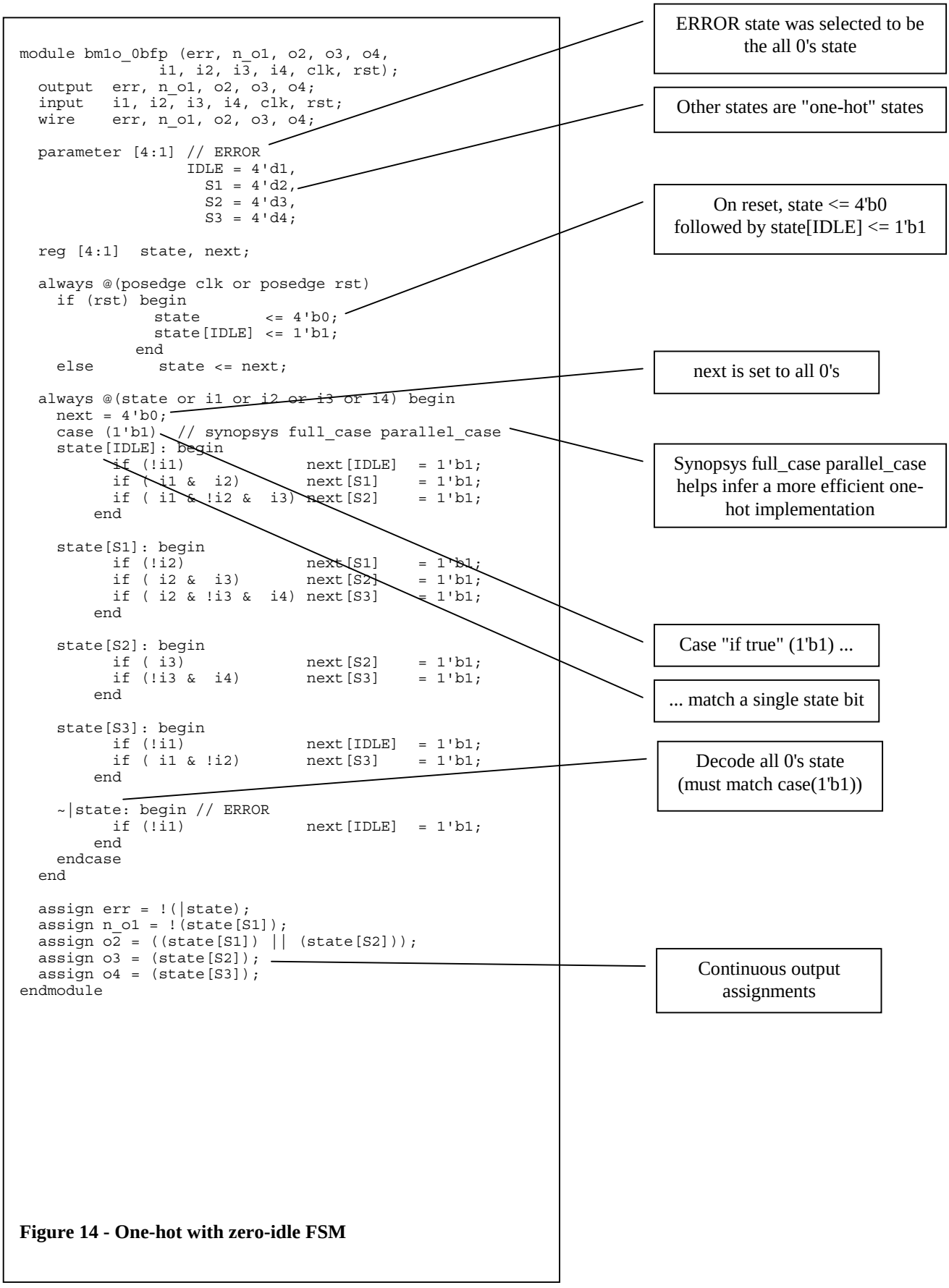
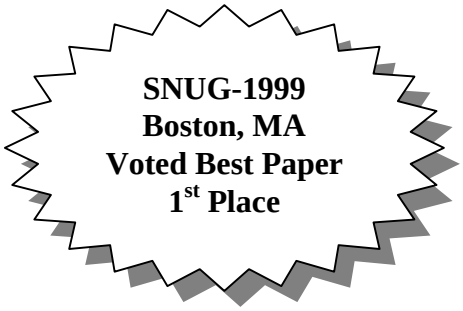


Figure 14 - One-hot with zero-idle FSM

"full_case parallel_case", the Evil Twins of Verilog Synthesis

Clifford E. Cummings

Sunburst Design, Inc.



ABSTRACT

Two of the most over used and abused directives included in Verilog models are the directives "//synopsys full_case parallel_case". The popular myth that exists surrounding "full_case parallel_case" is that these Verilog directives always make designs smaller, faster and latch-free. This is false! Indeed, the "full_case parallel_case" switches frequently make designs larger and slower and can obscure the fact that latches have been inferred . These switches can also change the functionality of a design causing a mismatch between pre-synthesis and post-synthesis simulation, which if not discovered during gate-level simulations will cause an ASIC to be taped out with design problems.

This paper details the effects of the "full_case parallel_case" directives and includes examples of flawed and inefficient logic that is inferred using these switches. This paper also gives guidelines on the correct usage of these directives.

1.0 Introduction

The "full_case parallel_case" commands are two of the most abused synthesis directives employed by Verilog synthesis design engineers. The reasons cited most often to the author for using "full_case parallel_case" are:

- "full_case parallel_case" makes my designs smaller and faster.
- "full_case" removes latches from my designs.
- "parallel_case" removes large, slow priority encoders from my designs.

The above reasons are either inaccurate or dangerous. Sometimes these directives don't affect a design at all, sometimes these switches make a design larger and slower, sometimes these directives change the functionality of a design, and ***these directives are always most dangerous when they work!***

This paper will define "full" and "parallel" case statements, detail case statement usage and show the effects that the "full_case parallel_case" directives have on synthesized code. An alternate title for this paper could be: "How to add $200,000 to the cost and 3-6 months to the schedule of your ASIC design without trying!"

2.0 Case statement definitions

To fully understand how the "full_case parallel_case" directives work, a common set of terms is needed to describe the different parts of a case statement. This section defines a common set of terms that will be used to describe case statement functionality throughout the rest of the paper.

2.1 Case statement

In Verilog, a case statement includes all of the code between the Verilog keywords, "case" ("casez", "casex") and "endcase" [1].

A case statement is a select-one-of-many construct that is roughly equivalent to an if-else-if statement. The general case statement in Figure 1 is equivalent to the general if-else-if statement shown in Figure 2.

```
case (case_expression)
  case_item1 : case_item_statement1;
  case_item2 : case_item_statement2;
  case_item3 : case_item_statement3;
  case_item4 : case_item_statement4;
  default    : case_item_statement5;
endcase
```

Figure 1 - Case Statement - General Form

```
if      (case_expression === case_item1) case_item_statement1;
else if (case_expression === case_item2) case_item_statement2;
else if (case_expression === case_item3) case_item_statement3;
else if (case_expression === case_item4) case_item_statement4;
else                                    case_item_statement5;
```

Figure 2 - If-else-if Statement - General Form

2.2 Case statement header

A case statement header consists of the "case" ("casez", "casex") keyword followed by the case expression, usually all on one line of code.

When adding "full_case" or "parallel_case" directives to a case statement, the directives are added as a comment immediately following the case expression at the end of the case statement header and before any of the case items on subsequent lines of code.

2.3 Case expression

A Verilog case expression is the expression enclosed between parentheses immediately following the "case" keyword. In Verilog, a case expression can either be a constant, such as "1'b1" (one bit of '1', or "true"), it can be an expression that evaluates to a constant value, or most often it is a bit or vector of bits that are used to compare against case items.

2.4 Case item

The case item is the bit, vector or Verilog expression that is used to compare against the case expression.

Unlike other high-level programming languages such as 'C', the Verilog case statement includes implied break statements. The first case item that matches the current case expression causes the corresponding case item statement to be executed and then all of the rest of the case items are skipped (ignored) for the current pass through the case statement.

2.5 Case item statement

A case item statement is one or more Verilog statements that are executed if the case item matches the current case expression.

Unlike VHDL, Verilog case items can themselves be expressions. To simplify parsing of Verilog source code, Verilog case item statements must be enclosed between the keywords "begin" and "end" if more than one statement is to be executed for a selected case item. This is one of the few places were Verilog syntax requirements are considered by VHDL-literate engineers to be too verbose.

2.6 Case default

An optional case "default" can be included in the case statement to indicate what actions to perform if none of the defined case items matches the current case expression. It is good coding style to place the case default last, even though the Verilog standard does not require it.

2.7 Casez

In Verilog there is a casez statement, a variation of the case statement that permits "z" and "?" values to be treated during case-comparison as "don't care" values. "Z" and "?" are treated as a don't care if they are in the case expression *and/or* if they are in the case item.

More information on the precautions that should be taken when using casez for RTL modeling and synthesis are detailed by Mills [2].

Guideline: Exercise caution when coding synthesizable models using the Verilog casez statement [2].

Coding Style Guideline: When coding a case statement with "don't cares," use a casez statement and use "?" characters instead of "z" characters in the case items to indicate "don't care" bits.

2.8 Casex

In Verilog there is a casex statement, a variation of the case statement that permits "z", "?" and "x" values to be treated during comparison as "don't care" values. "x", "z" and "?" are treated as a don't care if they are in the case expression *and/or* if they are in the case item.

More information on the dangers of using casex for RTL modeling and synthesis are detailed by Mills [2]

Guideline: Do not use casex for synthesizable code [2].

3.0 What is a "full" case statement?

A "full" case statement is a case statement in which all possible case-expression binary patterns can be matched to a case item or to a case default. If a case statement does not include a case default and if it is possible to find a binary case expression that does not match any of the defined case items, the case statement is not "full."

3.1 Synopsys case statement reports - "full_case"

For each case statement that is read by Synopsys tools, a case statement report is generated that indicates one of the following conditions with respect to the "full" nature of a each case statement:

- Full / auto (Figure 3) - Synopsys tools have determined that the case statement as coded is "full."

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |    auto/auto     |
================================================
```

Figure 3 - full / auto - Case statement is "full"

- Full / no (Figure 4) - The case statement was not recognized to be "full" by Synopsys.

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |     no/auto      |
================================================
```

Figure 4 - full / no - Case statement not "full"

- Full / user (Figure 5) - A Synopsys "full_case" directive was added to the case statement header by the user.

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |    user/auto     |
================================================
```

Figure 5 - full / user - "// synopsys full_case" added to the case header

3.2 HDL "full" case statement

From an HDL simulation perspective, a "full" case statement is a case statement in which every possible binary, non-binary and mixture of binary and non-binary patterns is included as a case item in the case statement. Verilog non-binary values are, and VHDL non-binary values include, "z" and "x" and are called metalogical characters by both the IEEE Draft Standard For VHDL RTL Synthesis [3] and the IEEE Draft Standard For Verilog RTL Synthesis [4].

3.3 Synthesis "full" case statement

From a synthesis tool perspective, a "full" case statement is a case statement in which every possible binary pattern is included as a case item in the case statement.

Verilog does not require case statements to be either synthesis or HDL simulation "full," but Verilog case statements can be made full by adding a case default. VHDL requires case statements to be HDL simulation "full," which generally requires an "others" clause.

Example 1 shows a case statement, with case default, for a 3-to-1 multiplexer. The case default causes the case statement to be "full." During Verilog simulation, when binary pattern 2'b11 is driven onto the select lines, the y-output will be driven to an unknown, but the synthesis will treat the y-output as a "don't care" for the same select-line combination, causing a mismatch to occur between simulation and synthesis. To insure that the pre-synthesis and post-synthesis simulations match, the case default could assign the y-output to either a predetermined constant value, or to one of the other multiplexer input values.

```
module mux3c (y, a, b, c, sel);
  output       y;
  input  [1:0] sel;
  input        a, b, c;
  reg          y;

  always @(a or b or c or sel)
      case (sel)
        2'b00:   y = a;
        2'b01:   y = b;
        2'b10:   y = c;
        default: y = 1'bx;
      endcase
endmodule
```

Example 1 - A case default, "full" case statement

```
Statistics for case statements in always block at line 7 in file
        '.../mux3c.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |    auto/auto     |
================================================
```

Figure 6 - Case statement report for a case statement with a case default

3.4 Non-"full" case statements

Example 2 shows a case statement for a 3-to-1 multiplexer that is not "full." The case statement does not define what happens to the y-output when binary pattern 2'b11 is driven onto the select lines. In this example, the Verilog simulation will hold the last assigned y-output value and synthesis will infer a latch on the y-output as shown in the latch inference report of Figure 7.

```
module mux3a (y, a, b, c, sel);
  output       y;
  input  [1:0] sel;
  input        a, b, c;
  reg          y;

  always @(a or b or c or sel)
      case (sel)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
      endcase
endmodule
```

Example 2 - Non-full case statement

```
Statistics for case statements in always block at line 7 in file
        '.../mux3a.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |     no/auto      |
================================================

Inferred memory devices in process
     in routine mux3a line 7 in file
          '.../mux3a.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|        y_reg        |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
===============================================================================
```

Figure 7 - Latch inference report for non-full case statement

3.5 Synopsys "full_case"

Synopsys tools recognize two directives when added to the end of a Verilog case header. The directives are "// synopsys full_case parallel_case." The directives can either be used together or an engineer can elect to use only one of the directives for a particular case statement. The Synopsys "parallel_case" directive is described in section 4.4.

When "// synopsys full_case" is added to a case statement header, there is no change in the Verilog simulation for the case statement, since "// synopsys ..." is interpreted to be nothing more than a Verilog comment; however, Synopsys parses all Verilog comments that start with "// synopsys ..." and interprets the "full_case" directive to mean that if a case statement is not "full" that the outputs are "don't care's" for all unspecified case items. If the case statement includes a case default, the "full_case" directive will be ignored.

Example 3 shows a case statement for a 3-to-1 multiplexer that is not "full" but the case header includes a "full_case" directive. During Verilog simulation, when binary pattern 2'b11 is driven onto the select lines, the y-output will behave as if it were latched, the same as in Example 2, but the synthesis will treat the y-output as a "don't care" for the same select-line combination, causing a functional mismatch to occur between simulation and synthesis.

```
module mux3b (y, a, b, c, sel);
  output       y;
  input  [1:0] sel;
  input        a, b, c;
  reg          y;

  always @(a or b or c or sel)
      case (sel) // synopsys full_case
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
      endcase
endmodule
```

Example 3 - Non-full case statement with "full_case" directive

```
Warning: You are using the full_case directive with a case statement in which not all cases
are covered.

Statistics for case statements in always block at line 7 in file
        '.../mux3b.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |    user/auto     |
================================================
```

Figure 8 - Case statement report for a non-full case statement with "full_case" directive

4.0 What is a "parallel" case statement?

A "parallel" case statement is a case statement in which it is only possible to match a case expression to one and only one case item. If it is possible to find a case expression that would match more than one case item, the matching case items are called "overlapping" case items and the case statement is not "parallel."

4.1 Synopsys case statement reports - "parallel_case"

For each case statement that is read by Synopsys tools, a case statement report is generated that indicates one of the following conditions with respect to the "parallel" nature of each case statement:

- Parallel / no (Figure 9) - The case statement was not recognized to be "parallel" by Synopsys.

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |     auto/no      |
================================================
```

Figure 9 - parallel / no - Case statement not "parallel"

- Parallel / auto (Figure 10) - Synopsys tools have determined that the case statement as coded is "parallel."

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |    auto/auto     |
================================================
```

Figure 10 - parallel / auto - Case statement is "parallel"

- Parallel / user (Figure 11) - A Synopsys "parallel_case" directive was added to the case statement header by the user.

```
Statistics for case statements in always block at line ...
================================================
|           Line           |  full/ parallel  |
================================================
|            X             |    auto/user     |
================================================
```

Figure 11 - parallel / user - "// synopsys parallel_case" added to the case header

4.2 Non-parallel case statements

Example 4 shows a casez statement that is not parallel because if the 3-bit irq bus is 3'b011, 3'b101, 3'b110 or 3'b111, more than one case item could potentially match the irq value. This will simulate like a priority encoder where irq[2] has priority over irq[1], which has priority over irq[0]. This example will also infer a priority encoder when synthesized.

```
module intctl1a (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;

  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 4 - Non-parallel case statement

```
Statistics for case statements in always block at line 6 in file
        '.../intctl1a.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9             |      no/no       |
================================================
```

Figure 12 - Case statement report for Example 4

4.3 Parallel case statements

Example 5 is a modified version of Example 4 such that each of the case items is now unique and therefore parallel. Even though the case items are parallel, this example happens to infer priority encoder logic when synthesized.

```
module intctl2a (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;

  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq)
      3'b1??: int2 = 1'b1;
      3'b01?: int1 = 1'b1;
      3'b001: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 5 - Parallel case statement

```
Statistics for case statements in always block at line 6 in file
        '.../intctl2a.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9             |     no/auto      |
================================================
```

Figure 13 - Case statement report for Example 5

4.4 Synopsys "parallel_case"

Example 6 is the same as Example 4 except that a Synopsys "parallel_case" directive has been added to the case header. This example will simulate like a priority encoder but will infer non-priority encoder logic when synthesized.

```
module intctl1b (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;

  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq) // synopsys parallel_case
      3'b1??: int2 = 1'b1;
      3'b?1?: int1 = 1'b1;
      3'b??1: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 6 - Non-parallel case statement with "parallel_case" directive

```
Warning: You are using the parallel_case directive with a case statement in
which some case-items may overlap

Statistics for case statements in always block at line 6 in file
        '.../intctl1b.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |     no/user      |
================================================
```

Figure 14 - Case statement report for Example 6

In Example 6, the "parallel_case" directive has "worked" and now the synthesized logic does not match the Verilog functional model.

4.5 A "parallel" case statement with "parallel_case" directive

The casez statement in Example 7 is parallel! If a "parallel_case" directive is added to the casez statement, it will make no difference. The design will synthesize the same as without the "parallel_case" directive.

The point is, the "parallel_case" directive is always most dangerous when it works! When it does not work, it is just extra characters at the end of the case header.

```
module intctl2b (int2, int1, int0, irq);
  output       int2, int1, int0;
  input  [2:0] irq;
  reg          int2, int1, int0;

  always @(irq) begin
    {int2, int1, int0} = 3'b0;
    casez (irq) // synopsys parallel_case
      3'b1??: int2 = 1'b1;
      3'b01?: int1 = 1'b1;
      3'b001: int0 = 1'b1;
    endcase
  end
endmodule
```

Example 7 - Parallel case statement with "parallel_case" directive

4.6 Verilog & VHDL case statements

VHDL case statements are required to have no overlap in of any case items, are therefore "parallel" and cannot infer priority encoders. VHDL case items are constants that are used to compare against the VHDL case expression. For this reason, it is also easy to parse multiple VHDL case item statements without the need to include "begin" and "end" keywords for case item statements.

Verilog case statements are permitted to have overlapping case items. Verilog case items can be separate and distinct Boolean expressions where one or more of the expressions can evaluate to "true" or "false." In those instances where more than one case item can match a "true" or "false" case expression, the first matching case item has priority over subsequent matching case items; therefore, the corresponding priority logic will be inferred by synthesis tools.

Verilog casez and casex statements can also include case items with constant vector expressions that include "don't-cares" that would permit a case expression to match multiple case_items in the casez or casex statements, also inferring a priority encoder.

If all goes well, "full_case parallel_case" will do nothing to your design, and it will work fine. The problem happens when "full_case parallel_case" DO work to change the functionality of your design or increase the size and area of your design.

There is one other style of Verilog case statement that frequently infers a priority encoder. The style is frequently referred to as the "case if true" or "reverse case" statement coding style.

This case statement style evaluates expressions for each case item and then tests to see of they are "true" (equal to 1'b1). This coding style is used to infer very efficient one-hot finite state machines, but is otherwise a somewhat dangerous coding practice.

4.7 Coding priority encoders

Non-parallel case statements infer priority encoders. It is a poor coding practice to code priority encoders using case statements. It is better to code priority encoders using if-else-if statements.

Guideline: Code all intentional priority encoders using if-else-if statements. It is easier for a typical design engineer to recognize a priority encoder when it is coded as an if-else-if statement.

Guideline: Case statements can be used to create tabular coded parallel logic. Coding with case statements is recommended when a truth-table-like structure makes the Verilog code more concise and readable.

Guideline: Examine all synthesis tool case-statement reports [5].

Guideline: Change the case statement code, as outlined in the above coding guidelines, whenever the synthesis tool reports that the case statement is not parallel (whenever the synthesis tool reports "no" for "parallel_case") [5].

Although good priority encoders can be inferred from case statements, following the above coding guidelines will help to prevent mistakes and mismatches between pre-synthesis and post-synthesis simulations.

5.0 Synthesis coding styles

Sunburst Design Assumption: it is generally a bad coding practice to give the synthesis tool different information about the functionality of a design than is given to the simulator.

Whenever either "full_case" or "parallel_case" directives are added to the Verilog source code, more information is potentially being given about the design to the synthesis tool than is being given to the simulator.

Guideline: In general, do not use "full_case parallel_case" directives with any Verilog case statements.

Guideline: There are exceptions to the above guideline but you better know what you're doing if you plan to add "full_case parallel_case" directives to your Verilog code.

Guideline: Educate (or fire) any employee or consultant that routinely adds "full_case parallel_case" to all case statements in their Verilog code, especially if the project involves the design of medical diagnostic equipment, medical implants, or detonation logic for thermonuclear devices!

Guideline: only use full_case parallel_case to optimize onehot FSM designs.

Other exceptions might exist and will be acknowledged by the author as they are discovered.

6.0 Latch example using "full_case"

Myth: "// synopsys full_case" removes all latches that would otherwise be inferred from a case statement.

Truth: The "full_case" directive only removes latches from a case statement for missing case items. One of the most common ways to infer a latch is to make assignments to multiple outputs from a single case statement but neglect to assign all outputs for each case item. Even adding the "full_case" directive to this type of case statement will not eliminate latches [6].

Example 8 shows Verilog code for a simple address decoder that will infer a latch for the mce0_n mce1_n and rce_n outputs, despite the fact that the "full_case" directive was used with the case statement. In this example, the case statement is "full" but not all outputs are assigned for each case item; therefore, latches were inferred for all three outputs. The easiest way to eliminate latches is to make initial default value assignments to all outputs immediately beneath the sensitivity list, before executing the case statement, as shown in Example 9.

```
module addrDecode1a (mce0_n, mce1_n, rce_n, addr);
  output         mce0_n, mce1_n, rce_n;
  input  [31:30] addr;
  reg            mce0_n, mce1_n, rce_n;

  always @(addr)
    casez (addr) // synopsys full_case
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:           rce_n  = 1'b0;
    endcase
endmodule
```

Example 8 - "full_case" directive with latched outputs

```
Statistics for case statements in always block at line 6 in file
       ...'addrDecode1a.v'
================================================
|           Line           |  full/ parallel   |
================================================
|            8             |    user/auto      |
================================================

Inferred memory devices in process
     in routine addrDecode1a line 6 in file
          ...'addrDecode1a.v'.
===============================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST |
===============================================================================
|     mce0_n_reg      |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
|     mce1_n_reg      |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
|      rce_n_reg      |   Latch   |   1   |  -  | -  | N  | N  | -  | -  | -  |
===============================================================================
```

Figure 15 - Case statement report and latch report for "full_case" latched example

```
module addrDecode1d (mce0_n, mce1_n, rce_n, addr);
  output         mce0_n, mce1_n, rce_n;
  input  [31:30] addr;
  reg            mce0_n, mce1_n, rce_n;

  always @(addr) begin
    {mce1_n, mce0_n, rce_n} = 3'b111;
    casez (addr)
      2'b10: {mce1_n, mce0_n} = 2'b10;
      2'b11: {mce1_n, mce0_n} = 2'b01;
      2'b0?:           rce_n  = 1'b0;
    endcase
  end
endmodule
```

Example 9 - Initial default value assignments to remove latches

```
Statistics for case statements in always block at line 6 in file
        ...'addrDecode1d.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            9            |    auto/auto     |
================================================
```

Figure 16 - Case statement report for Example 9

7.0 Synopsys warnings

When Verilog files are read by design_analyzer or dc_shell, Synopsys issues warnings when the "full_case" directive is used with a case statement that was not "full" (see Synopsys "full_case" description in section 3.5).

Example 10 shows a non-full case statement with "full_case" directive. Figure 17 shows the warning that is reported when the "full_case" directive is used with a non-full case statement.

```
module fcasewarn1b (y, d, en);
  output y;
  input  d, en;
  reg    y;

  always @(d or en)
    case (en) // synopsys full_case
      1'b1: y = d;
    endcase
endmodule
```

Example 10 - Non-full case statement with "full_case" directive

```
"Warning: You are using the full_case directive with a case statement in which not all
cases are covered."

Statistics for case statements in always block
          at line 6 in file ..."/fcasewarn1b.v"
================================================
|           Line           |  full/ parallel  |
================================================
|            8             |    user/auto     |
================================================
```

Figure 17 - Synopsys "full_case" warning

The warning in Figure 17 is really saying, "watch out! the *full_case* directive might work and cause your design to break!!" Unfortunately this warning is easy to miss when running a synthesis script and the design might be adversely affected by the "full_case" directive.

Similarly, when Verilog files are read by design_analyzer or dc_shell, Synopsys issues warnings when the "parallel_case" directive is used with a case statement that was not "parallel." (see Synopsys "parallel_case" description in section 4.4).

Example 11 shows a non-parallel case statement with "parallel_case" directive. Figure 18 shows the warning that is reported when the "parallel_case" directive is used with a non-parallel case statement.

```
module pcasewarn1b (y, z, a, b, c, d);
  output y, z;
  input  a, b, c, d;
  reg    y, z;

  always @(a or b or c or d) begin
    {y,z} = 2'b00;
    casez ({a,b,c,d}) // synopsys parallel_case
      4'b11??: y = 1'b1;
      4'b??11: z = 1'b1;
    endcase
  end
endmodule
```

Example 11 - Non-parallel case statement with "parallel_case" directive

```
"Warning: You are using the parallel_case directive with a case statement in which some
case-items may overlap."

Statistics for case statements in always block
          at line 6 in file ..."/pcasewarn1b.v"
================================================
|           Line           |  full/ parallel  |
================================================
|            9             |     no/user      |
================================================
```

Figure 18 - Synopsys "parallel_case" warning

The warning in Figure 18 is really saying, "watch out! the *parallel_case* directive might work and cause your design to break!!" Unfortunately this warning, like the "full_case" warning, is also easy to miss when running a synthesis script and the design might be adversely affected by the "parallel_case" directive.

8.0 Actual "full_case" design problem

The 2-to-4 decoder with enable in Example 12, uses a case statement that is coded without using any synthesis directives. The resultant design was a decoder built from 3-input and gates and inverters. No latch is inferred because all outputs are given a default assignment before the case statement. For this example, the pre-synthesis and post-synthesis designs and simulations matched. The 2-to-4 decoder with enable in Example 13, uses a case statement with the "full_case" synthesis directive. Because of this synthesis directive, the enable input (en) was optimized away during synthesis and left as a dangling input. The pre-synthesis simulation results of modules code4a and code4b matched the post-synthesis simulation results of module code4a, but did not match the post-synthesis simulation results of module code4b [2].

```
// no full_case
// Decoder built from four 3-input and gates
//   and two inverters
module code4a (y, a, en);
  output [3:0] y;
  input  [1:0] a;
  input        en;
  reg    [3:0] y;

  always @(a or en) begin
    y = 4'h0;
    case ({en,a})
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

Example 12 - Decoder example with no "full_case" directive

```
Statistics for case statements in always block at line 9 in file
        '.../code4a.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            12            |     no/auto      |
================================================
```

Figure 19 - Case statement report for Example 12

```
// full_case example
// Decoder built from four 2-input nor gates
//   and two inverters
// The enable input is dangling (has been optimized away)
module code4b (y, a, en);
  output [3:0] y;
  input  [1:0] a;
  input        en;
  reg    [3:0] y;

  always @(a or en) begin
    y = 4'h0;
    case ({en,a}) // synopsys full_case
      3'b1_00: y[a] = 1'b1;
      3'b1_01: y[a] = 1'b1;
      3'b1_10: y[a] = 1'b1;
      3'b1_11: y[a] = 1'b1;
    endcase
  end
endmodule
```

Example 13 - Decoder example with "full_case" directive

```
Warning: You are using the full_case directive  with a case statement in which
not all cases are covered

Statistics for case statements in always block at line 10 in file
        '.../code4b.v'
================================================
|           Line           |  full/ parallel  |
================================================
|            13            |    user/auto     |
================================================
```

Figure 20 - Case statement report for Example 13

9.0 Actual "parallel_case" design problem

One consultant shared the experience where "parallel_case" was added to the Verilog code for a large ASIC design to remove stray priority encoders and infer a smaller and faster design. The Verilog case statement was coded as a priority encoder and all RTL simulations worked correctly. Unfortunately, the gate-level design without priority encoder did not function correctly and the gate-level simulations did not catch the problem. This ASIC had to be re-designed, costing $100,000's of actual dollars, delayed product release, and unknown lost dollars for being months late to market.

10.0 Summary of guidelines and conclusions

Summary of guidelines and conclusions

Guideline: Exercise caution when coding synthesizable models using the Verilog casez statement [2].

Guideline: Do not use casex for synthesizable code [2].

Guideline: In general, do not use "full_case parallel_case" directives with any Verilog case statements.

Guideline: There are exceptions to the above guideline but you better know what you're doing if you plan to add "full_case parallel_case" directives to your Verilog code.

Guideline: Code all intentional priority encoders using if-else-if statements. It is easier for a typical design engineer to recognize a priority encoder when it is coded as an if-else-if statement.

Guideline: Coding with case statements is recommended when a truth-table-like structure makes the Verilog code more concise and readable.

Guideline: Examine all synthesis tool case-statement reports [5].

Guideline: Change the case statement code, as outlined in the above coding guidelines, whenever the synthesis tool reports that the case statement is not parallel (whenever the synthesis tool reports "no" for "parallel_case") [5].

Guideline: only use full_case parallel_case to optimize onehot FSM designs.

Coding Style Guideline: When coding a case statement with "don't cares," use a casez statement and use "?" characters instead of "z" characters in the case items to indicate "don't care" bits.

Guideline: Educate (or fire) any employee or consultant that routinely adds "full_case parallel_case" to all case statements in their Verilog code.

Conclusion: "full_case" and "parallel_case" directives are most dangerous when they work! It is better to code a full and parallel case statement than it is to use directives to make up for poor coding practices.

References

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

[2] Don Mills and Clifford Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," in *SNUG 1999 Proceedings.*

[3] IEEE P1076.6 Draft Standard For VHDL Register Transfer Level Synthesis, section 5

[4] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis, section 4

[5] Steve Golson, personal communication

[6] "HDL Compiler for Verilog Reference Manual," section 9. Synopsys Online Documentation v1999.05

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 18 years of ASIC, FPGA and system design experience and eight years of Verilog, synthesis and methodology training experience.

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

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

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

(Data accurate as of October 9th, 2000)

fsm_perl: A Script to Generate RTL Code for State Machines and Synopsys Synthesis Scripts

Clifford E. Cummings

Sunburst Design, Inc.

ABSTRACT

Coding a Verilog RTL model of a state machine requires significant effort to generate an efficient synthesizable implementation. There are a number of different coding styles that can yield different results with varying degrees of efficiency. Because of the effort required to code a Verilog state machine, an engineer typically makes a guess as to which coding style will yield a good implementation and then rarely experiments with other styles after the first model simulates correctly.

This paper details a new and highly abbreviated language for coding a state machine and then describes the use of a Perl script called fsm_perl to turn the abbreviated code into a variety of synthesizable models for synthesis experimentation.

The fsm_perl also generates an accompanying dc_shell script to synthesize and compare the area and timing of each synthesized implementation.

1.0 **Introduction**

Coding a Finite State Machine (FSM) is not a difficult task but does involve a fair amount of typing. Efficient Verilog coding styles are well known but which FSM state-encoding style will give the best results is not obvious. The ability to easily generate different Verilog FSM designs and the accompanying synthesis scripts was the reason fsm_perl was developed. Fsm_perl is a freely available Perl script designed to make Finite State Machine (FSM) coding, experimentation and synthesis easy and efficient. Instructions on how to download fsm_perl from the Sunburst Design web site are included at the end of this paper.

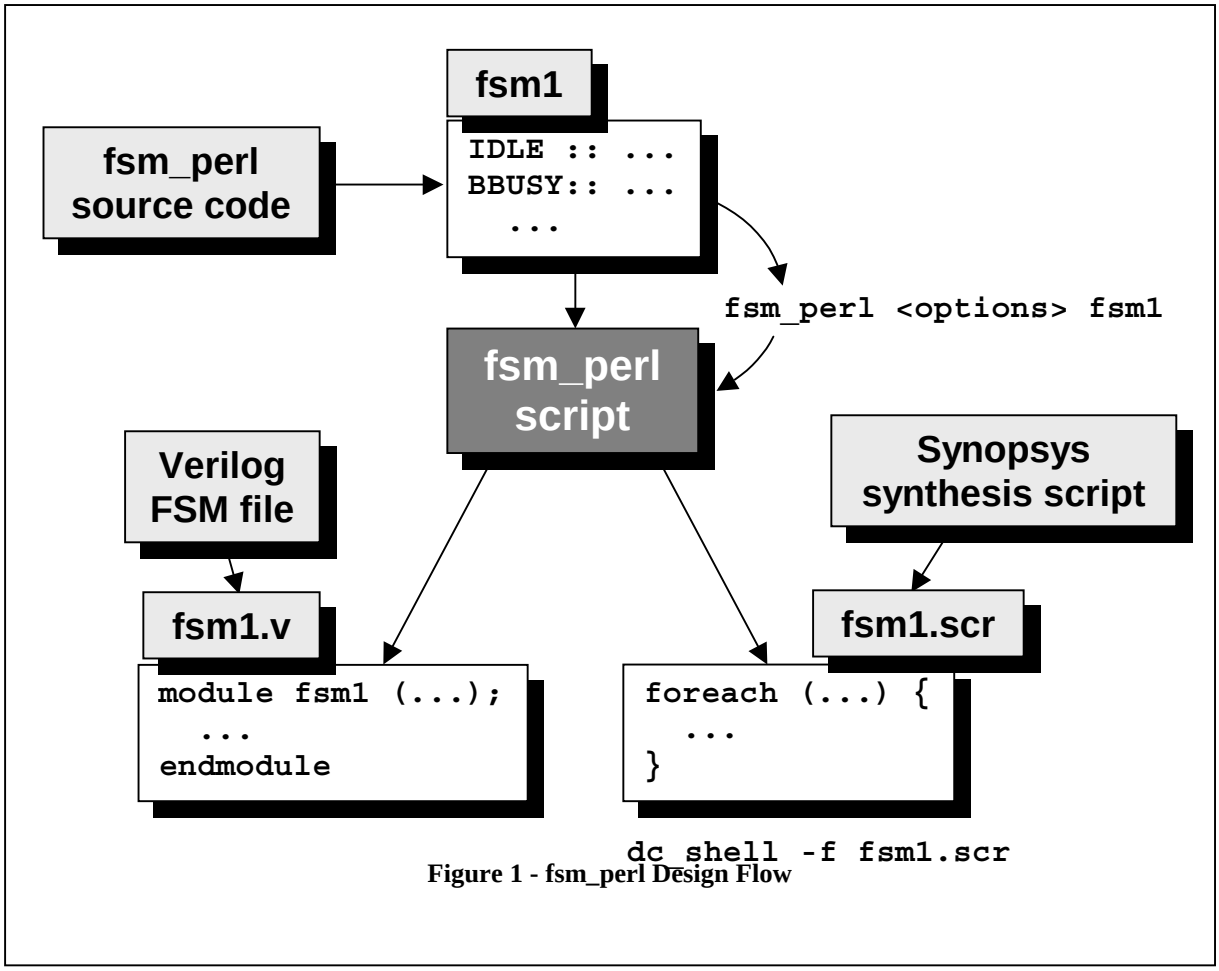


Figure 1 - fsm_perl Design Flow

Figure 1 shows the basic fsm_perl design flow. An fsm_perl source file is coded using any text editor, and then the source file is compiled using the fsm_perl command. Fsm_perl generates two files, the synthesizable Verilog source code file and a Synopsys synthesis script. The Synopsys synthesis script can then be run using dc_shell to read and compile the Verilog FSM code, produce a Verilog gate-level netlist, produce the corresponding SDF timing file and an area/timing report file.

2.0 **Basic fsm_perl Syntax**

The fsm_perl syntax was designed to make FSM coding simple, compact and easily interpreted; indeed, the fsm_perl syntax was intended to be easier to read and maintain than an equivalent Verilog source file for the same FSM. To this end, the basic fsm_perl syntax largely revolves around triplets to describe state diagram transition arcs.

In its most basic form, an fsm_perl source code file consists of state names, followed by one or more triplets consisting of input statements, next state values and Mealy- or Moore-output(s) assignments. Specifying all three fields is not required for every triplet. Legal triplet combinations are detailed in section 5.

3.0 **State Names and Encodings**

Each state in the state machine must appear as a left-side argument to a state-separator operator (a pair of adjacent colons ::). Only one adjacent colon-pair is permitted for each defined state and no white space is permitted between the colons.

State name example:

```
IDLE::  ...
READ::  ...
WAIT::  ...
DONE::  ...
```

The first state listed will be the reset-state (the state that the state machine will go to on reset).

The state names may be optionally followed by a binary state encoding enclosed within parentheses between the state name and the state-separator operator.

State name and encoding example:

```
IDLE (00)::  ...
READ (01)::  ...
WAIT (11)::  ...
DONE (10)::  ...
```

If user-defined state encodings are specified, then all of the states must specify a user-defined state encoding; otherwise, fsm_perl will report a state encoding error.

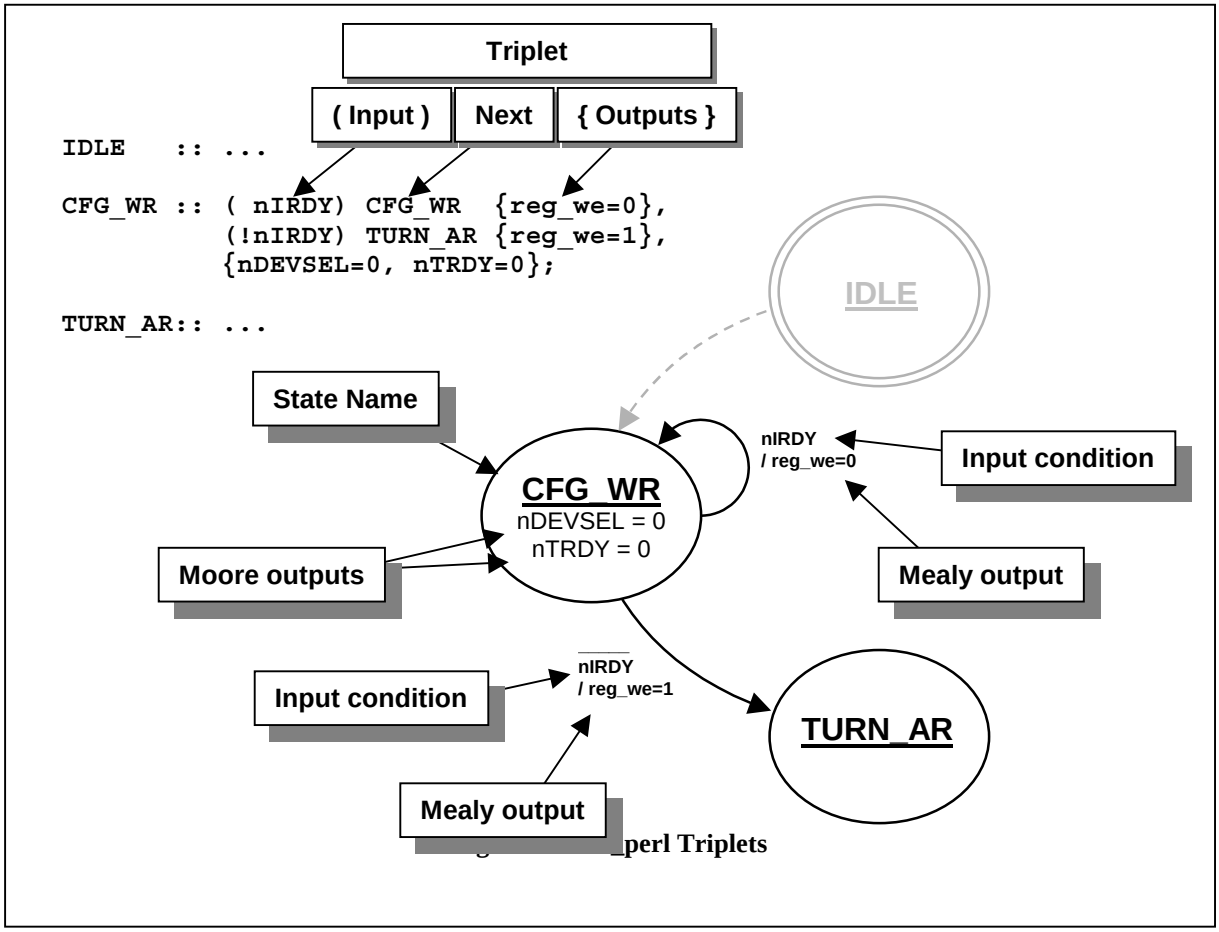
4.0 **Triplets**

Triplets are one or more comma-separated groups consisting of input statements, next state values and Mealy or Moore outputs assignments. Specifying all three fields is not required for every triplet. Legal triplet combinations are detailed in section 5. Triplets must appear as a right-side argument to a state-separator operator (::).

5.0 **Legal Triplet Statements**

5.1 **(Input)**

The input statement is enclosed in parentheses ()'s and is an expression that is used as a Boolean test. The code between the parentheses must be legal Verilog code since this expression will be copied directly into the Verilog code generated by fsm_perl. At this time, fsm_perl does no syntax checking of the Verilog expression between the parentheses; therefore, a Verilog syntax error placed in the fsm_perl source code will be written to the generated Verilog output file and will not be detected until the Verilog output file is compiled.



5.2 **Next State**

The next state statement is not enclosed in either parentheses or curly braces. The next state value must exactly match one of the state names used in the fsm_perl source code.

5.3 **{Outputs}**

Output statements are one or more Mealy outputs enclosed in curly braces {}'s or one or more Moore outputs enclosed in curly braces {}'s.

5.3.1 **{Mealy Outputs}**

Mealy outputs are enclosed within curly braces {}'s and must follow either an input statement, or an input statement and a next state statement. Mealy outputs are a function of the present state and one or more inputs, so if a next state is specified but an input statement is missing, inclusion of an output statement is illegal and fsm_perl issues a syntax error and halts. An output statement by itself with no input statement and no next state statement is a legal Moore output since Moore outputs are not dependent on either inputs or next state transitions.

Only one Mealy output statement is permitted for each state diagram transition-arc triplet. Multiple Mealy output definitions per transition-arc are coded as comma separated output assignments enclosed within one set of curly braces.

5.3.2 **{Moore Outputs}**

Moore outputs are enclosed within curly braces {}'s and are not preceded by either an input statement or a next state statement. Only one Moore output statement is permitted for each defined state. An fsm_perl syntax error is reported if more than one Moore output is detected per state definition.

5.4 **Polarities and Bus Assignments**

Input expressions are copied directly to the generated Verilog output code and input expressions are parsed to detect input vectors. A vector range should be included in the first input vector expression in the fsm_perl source file. Subsequent vector expressions do not require a range specification. Whenever a range specification is found in the input expression, fsm_perl tests the range to determine if a new minimum or maximum range value has been specified and updates the stored identifier range if a new minimum or maximum limit is detected.

Output expressions are parsed to determine both vector ranges (if the output is a non-scalar output) and default assignment value.

Fsm_perl has a source code directive that permits more control over vector declarations and default assignments. The directive is called "//fsm default" and is explained in more detail in section 7.4.

5.5 **Comments**

Fsm_perl only recognizes the Verilog single-line comment style (//). Everything in the fsm_perl source file from "//" to the end of the line is considered a comment.

6.0 **Simple Example**

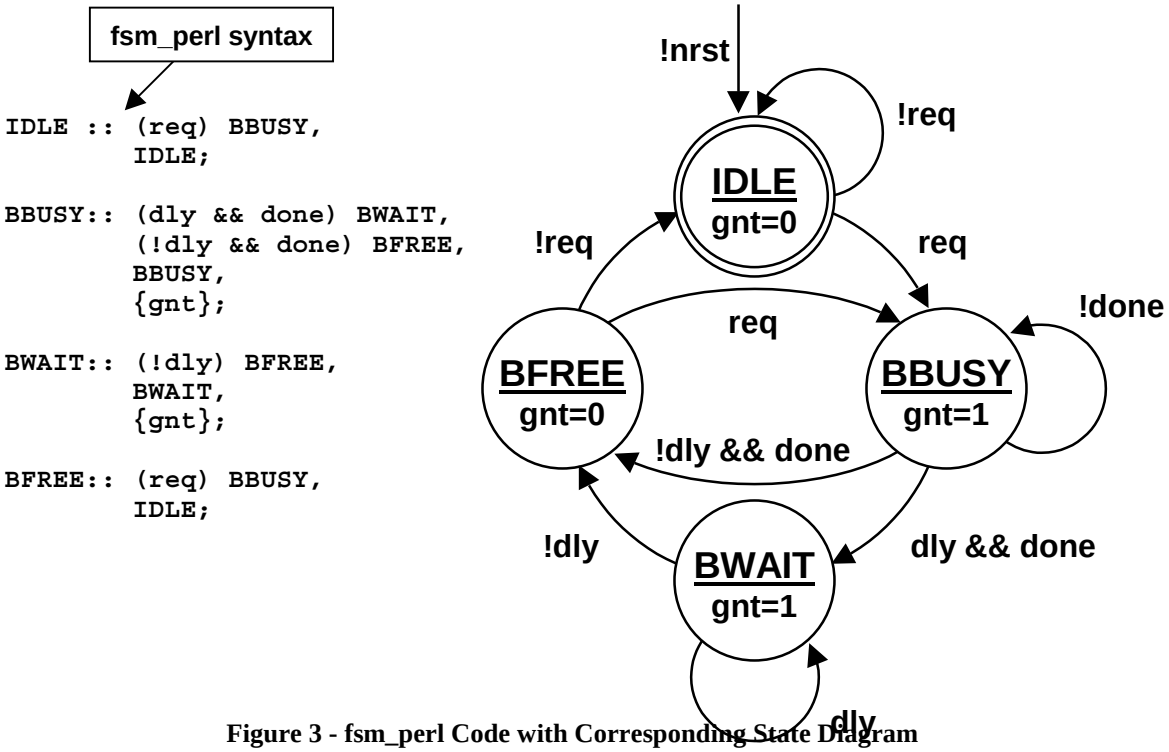


Figure 3 - fsm_perl Code with Corresponding State Diagram

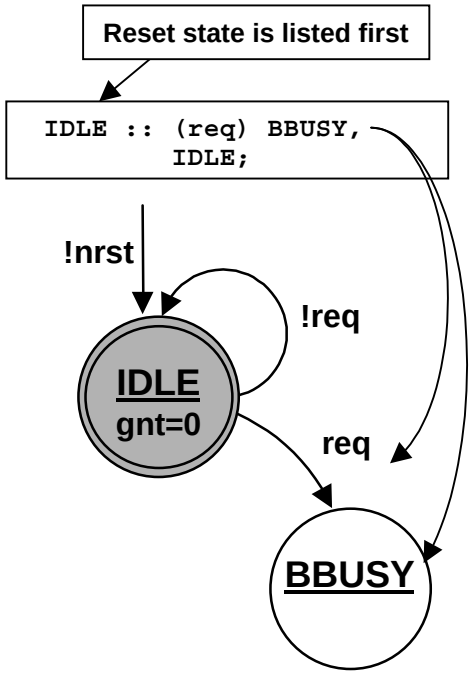


Figure 4 - IDLE State Definition

Figure 3 shows the state diagram for a simple 4-state, Moore state machine.

6.1 **IDLE State**

Figure 4 shows the fsm_perl syntax for the **`IDLE`** state of the state machine.

Since there are two transition arcs leaving the **`IDLE`** state, there will be two triplets to describe these arcs. The first triplet:

```
IDLE :: (req) BBUSY,
```

indicates that if **`req`** is true, a transition to the **`BBUSY`** state will occur. The second triplet:

```
IDLE;
```

could have been coded as **`(!req) IDLE;`** but for this example, a default next-state transition

triplet was used, indicating that there will be a transition to the **IDLE** state unless **req** is true, in which case the transition to **BBUSY** will occur.

The **IDLE** state does not list any outputs. This means that this state will use the default reset output assignments, as determined by the other output assignments in the fsm_perl source file.

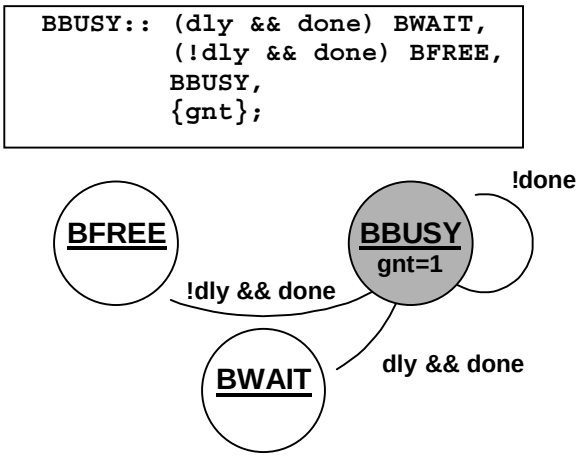


Figure 5 - BBUSY State Definition

6.2 BBUSY State

Figure 5 shows the fsm_perl syntax for the Bus BUSY (**BBUSY**) state of the state machine.

The **BBUSY** state has one Moore and no Mealy outputs. The output **gnt** is the first occurrence of this output in the fsm_perl code; therefore, this output is assumed to have a default-reset value opposite to the assigned value in this state. The default value for **gnt** will be a **0** after reset, and is assigned to **1** for the **BBUSY** state.

The **BBUSY** state also has three transition arcs leaving this state so there will be three triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(dly && done) BWAIT,
```

indicates that if **dly** and **done** are both true, a transition to the **BWAIT** state will occur. The second triplet:

```
(!dly && done) BFREE,
```

indicates that if **dly** is not true and **done** is true, a transition to the **BFREE** state will occur. The third triplet:

```
BBUSY,
```

could have been coded as **(!done) BBUSY;** but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the **BBUSY** state unless **done** is false, in which case a transition to one of the other two destination states will occur. There is one more triplet without either an input or a next state statement:

```
{gnt};
```

This triplet represents the Moore output for the **BBUSY** state, which was described above.

```
BWAIT:: (!dly) BFREE,
        BWAIT,
        {gnt};
```

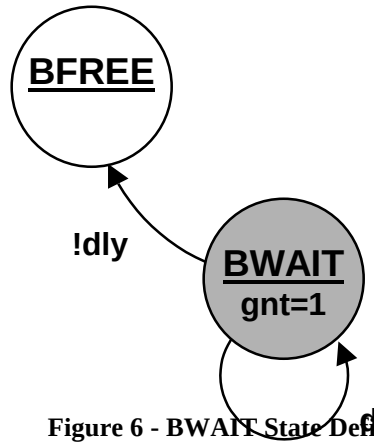


Figure 6 - BWAIT State Definition

6.3 BWAIT State

Figure 6 shows the fsm_perl syntax for the Bus WAIT (`BWAIT`) state of the state machine.

The `BWAIT` state has one Moore and no Mealy outputs. The output:

```
{gnt};
```

is assigned to `1` for the `BWAIT` state.

The `BWAIT` state also has two transition arcs leaving this state so there will be two triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(!dly) BFREE,
```

indicates that if `dly` is not true, a transition to the `BFREE` state will occur. The second triplet:

```
BWAIT,
```

could have been coded as `(dly) BWAIT;` but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the `BWAIT` state unless `dly` is false, in which case the transition to the `BFREE` state will occur.

```
BFREE:: (req) BBUSY,
        IDLE;
```

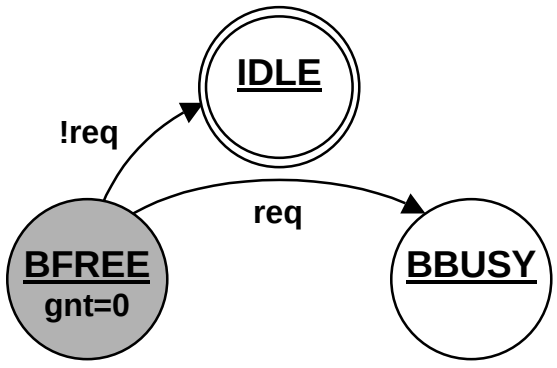


Figure 7 - BFREE State Definition

6.4 BFREE State

Figure 7 shows the fsm_perl syntax for the Bus FREE (`BFREE`) state of the state machine.

The `BFREE` state has two transition arcs leaving this state so there will be two triplets in the fsm_perl code to describe these arcs. The first triplet:

```
(req) BBUSY,
```

indicates that if `req` is true, a transition to the `BBUSY` state will occur. The second triplet:

```
IDLE;
```

could have been coded as **`(!req) IDLE;`** but for this example, a default next-state transition triplet was used, indicating that there will be a transition to the **`IDLE`** state unless **`req`** is true, in which case the transition to the **`BBUSY`** state will occur.

7.0 Fsm_perl Directives

Although fsm_perl is able to extract most of the information from the fsm_perl source code, "fsm" options can be specified to alter the default settings used by fsm_perl.

The fsm_perl user-selected defaults can be changed by adding directives (synthetic comments) to the fsm_perl source code. An fsm synthetic comment must start with "//fsm". No space is permitted between the "//" and "fsm". Legal "//fsm" commands are listed below.

7.1 //fsm clock

By default, fsm_perl generates Verilog code with clock name "clk" and "posedge" polarity. These values can be changed by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm clock [negative_polarity]new_clock_name

Examples:   //fsm clock !clock      (low-true "clock")
            //fsm clock CLK         (high-true "CLK")
```

Negative edge polarities are specified by preceding the clock signal name with either "!" or "~".

7.2 //fsm period

The //fsm period directive is the only fsm_perl directive that does not affect the generated Verilog code. The "period" directive helps specify clock constraints for the generated Synopsys synthesis script. By default, fsm_perl generates a synthesis script with no clock constraints. A clock constraint can be added to the Synopsys synthesis script by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm period clock_period

Examples:   //fsm period 20         ("create_clock clk -period 20")

            //fsm clock CK
            //fsm period 10         ("create_clock CK -period 10")
```

7.3 //fsm reset

By default, fsm_perl generates Verilog code with reset name "nrst" and "negedge" polarity. These values can be changed by adding an fsm synthetic comment to the fsm_perl source code of the form:

```
//fsm reset [negative_polarity]new_reset_name

Examples:   //fsm reset !rst_N      (low-true "rst_N")
            //fsm reset reset       (high-true "reset")
```

Negative edge polarities are specified by preceding the clock signal name with either "!" or "~".

7.4 //fsm default

Fsm_perl extracts scalar or vector information from the fsm_perl source and makes a default assignment when the outputs are assigned within the combinational always block. The default assignment value is determined by the scalar polarity or vector assignment value of each identifier that is found in the fsm_perl source code. The first time an output identifier is found, fsm_perl assumes that an assignment other than the reset-default is being made. Scalar reset-defaults are set to the opposite polarity of the first scalar assignment and vector reset-defaults outputs are assigned to all 0's. For example:

```
S1 :: S2,
      {y1};
...
```

State `S1` unconditionally transitions to state `S2` on the next clock edge and State `S1` has one scalar Moore output, `y1`, that is set to `1'b1`; therefore, fsm_perl assumes the reset-default setting for the `y1` output must have been `1'b0`. This assumption means that only when the output is assigned to a non-default value, must the output assignment be specified in the fsm_perl source code.

When an fsm_perl source file includes vector assignments and Mealy output assignments, fsm_perl might assume an incorrect reset-default output assignment value. //fsm default can be used to change the reset-default output assignment value. For the above example, if `y1` should be set to `1'b1` by default, the example could include the directive:

```
//fsm default y1=1'b1
...
S1 :: S2,
      {y1};
...
```

7.5 //fsm state

By default, fsm_perl generates Verilog code with the state name "state". To select a different state name, use the fsm_perl directive:

//fsm state *new_state_name*

```
Examples:   //fsm state PS
            //fsm state STATE
```

7.6 //fsm next

By default, fsm_perl generates Verilog code with the next state name "next". To select a different next state name, use the fsm_perl directive:

//fsm next *new_next_name*

```
Examples:   //fsm next NS
            //fsm next NEXT
```

7.7 //fsm define

It is sometimes desirable to make input comparisons against a macro that is defined by the Verilog `define compiler directive. When a `define comparison is used, fsm_perl must be notified of the existing definition; otherwise, fsm_perl will assume the `define identifier is a scalar input

and include the identifier as both a port identifier and as a declared input. To specify the existence of a macro definition, use the fsm_perl directive:

//fsm define *macro_name*

```
Examples:   //fsm define `CONFIG_READ
            //fsm define `CONFIG_WRITE
Usage example
triplet:   (!nFRAME && IDSEL && (nC_BE==`CONFIG_WRITE)) CFG_WR,  ...
```

7.8 //fsm filename

By default, fsm_perl uses the fsm_perl source file name as the root of the Verilog output file name. To select a different Verilog output file name, use the fsm_perl directive:

//fsm filename *new_file_name*

```
Example:    //fsm filename myfile
```

7.9 //fsm module

By default, fsm_perl uses the fsm_perl source file name as the root of the Verilog module name and output file name. If there is a period in the fsm_perl source file name, unless the "//fsm module" directive is included in the fsm_perl source code, an illegal Verilog module name will be generated. To select a different Verilog module name, use the fsm_perl directive:

//fsm module *new_module_name*

```
Example:    //fsm module mymodule
```

In the absence of a separate "//fsm filename" directive, the "//fsm module" directive also changes the output file name. To generate unique Verilog module and output file names, use both the "//fsm module" and "//fsm filename" directives.

```
Example:    //fsm module mymodule
            //fsm filename myfile
```

8.0 Command Invocation

Fsm_perl is invoked from the UNIX command prompt as follows:

fsm_perl [options] *fsm_perl_source_file*

The fsm_perl source code, Verilog output file and Synopsys synthesis script for the **fsm1** design used in the "Simple Example" (in section 6) are shown in Figure 11 at the end of this paper.

8.1 Fsm_perl Output Files

Fsm_perl generates two output files: a Verilog source file and a Synopsys synthesis script to compile the Verilog source file. See section 9.0 for details about the generated Synopsys synthesis scripts.

8.2 **Fsm_perl Options**

Fsm_perl has command line options that help generate different Verilog coding styles and Synopsys synthesis scripts for synthesis experimentation. Each option generates just one Verilog output file and one synthesis script.

8.2.1 **-e Option**

The -e option generates a Verilog file with binary encoded state variables and standard Synopsys enumeration comments. The -e option also generates a synthesis script that will compile the Verilog design four different ways. The script compiles the design (1) with no special processing, (2) using the FSM compiler gray encoding style setting, (3) using the FSM compiler one_hot encoding style setting, and (4) using the FSM compiler binary encoding style setting.

The Synopsys enumeration comments help the Synopsys FSM compiler to find and process the state variables and state encodings.

For the following command invocation:

`fsm_perl -e fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

`fsm1_e.v` (the synthesizable Verilog FSM source file)
`fsm1_e.scr` (the Synopsys synthesis script to compile the **`fsm1_e.v`** file).

The "_e" appendage indicates the "enumerated" coding style. The fsm_perl source code, Verilog output file and Synopsys synthesis script using the "-e" command option are shown in Figure 13 at the end of this paper.

8.2.2 **-1 Option**

The -1 (the number "one") option generates a Verilog file with one-hot encoded state variables and adds "synopsys full_case parallel_case" to the case statement. This is the only place where "full_case parallel_case" is automatically added to the Verilog source code since this coding style is the only coding style where full and parallel directives generally seem to make a positive difference in the quality of the synthesized design.

For the following command invocation:

`fsm_perl -1 fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

`fsm1_1fp.v` (the synthesizable Verilog FSM source file)
`fsm1_1fp.scr` (the Synopsys synthesis script to compile the **`fsm1_1fp.v`** file)

The "_1fp" appendage indicates the "one-hot full_case parallel_case" coding style. The fsm_perl source code, Verilog output file and Synopsys synthesis script using the "-1" command option are shown in Figure 12 at the end of this paper.

8.2.3 **-f Option**

The -f option generates a Verilog file with "synopsys full_case" appended to the case statement header code. Using this option is not recommended since the pre-synthesis simulation might not match the post-synthesis implementation, plus there are Verilog coding styles that can accomplish the same or better synthesis optimization without using this potentially dangerous switch; however, the switch is included to permit easy experimentation with "full_case" usage.

For the following command invocation:

`fsm_perl -f fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

`fsm1_f.v` (the synthesizable Verilog FSM source file)
`fsm1_f.scr` (the Synopsys synthesis script to compile the **`fsm1_f.v`** file)

The "_f" appendage indicates that " full_case" has been added to the Verilog output file.

8.2.4 **-p Option**

The -p option generates a Verilog file with "synopsys parallel_case" appended to the case statement header code. Using this option is not recommended since the pre-synthesis simulation might not match the post-synthesis implementation, plus there are Verilog coding styles that can accomplish the same or better synthesis optimization without using this potentially dangerous switch; however, the switch is included to permit easy experimentation with "parallel_case" usage.

For the following command invocation:

`fsm_perl -p fsm1` (where **`fsm1`** is the fsm_perl source file)

fsm_perl will create two output files named:

`fsm1_p.v` (the synthesizable Verilog FSM source file)
`fsm1_p.scr` (the Synopsys synthesis script to compile the **`fsm1_p.v`** file)

The "_p" appendage indicates that " parallel_case" has been added to the Verilog output file.

8.2.5 **Multiple Options**

Multiple options can be used at the same time when fsm_perl is invoked. When multiple options are used, the resultant file names will contain appended letters indicating which file options were used to run the fsm_perl script.

9.0 **Synthesis Scripts**

Not only does fsm_perl generate the Verilog code for an FSM, it also generates the Synopsys synthesis script required to compile the design and report performances.

```
design_list = { fsm1 }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" > rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context verilog -o DESIGN + ".sdf"
  write -f verilog -hier -output DESIGN + ".vg"
  report_area   >> rpt_file
  report_timing >> rpt_file
}
```

Figure 8 - fsm1.scr File

Figure 8 shows the synthesis script that is typically generated when using fsm_perl. The synthesis script reads the Verilog source file, compiles the design, writes out the SDF timing file, writes out the Verilog gate-level netlist, then reports area usage and worst case timing to a report file.

```
design_list = { fsm1_e }
foreach (DESIGN, design_list) {
  rpt_file = DESIGN + ".rpt"
  echo DESIGN + " Synthesis Run" > rpt_file
  read -f verilog DESIGN + ".v"
  current_design = DESIGN
  compile
...
  echo DESIGN + " Synopsys Gray-Code Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style gray
  compile
  create_schematic -size infinite
  write_timing -f sdf-v2.1 -context verilog -o DESIGN + "_xg.sdf"
  write -f verilog -hier -output DESIGN + "_xg.vg"
...
  echo DESIGN + " Synopsys One-Hot Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style onehot
  compile
...
  echo DESIGN + " Synopsys Binary Synthesis Run" >> rpt_file
  extract
  set_fsm_encoding_style binary
  compile
...
}
```

Figure 9 - Synopsys FSM Compiler-Options Script (fsm1_e.scr)

If the FSM file is compiled with the "-e" command option, fsm_perl will generate a synthesis script to compile the design four different ways: (1) with no special processing, (2) using the FSM compiler gray encoding style setting, (3) using the FSM compiler one_hot encoding style setting, and (4) using the FSM compiler binary encoding style setting. Figure 9 shows a section of the synthesis script that is typically generated using the "-e" option.

If the fsm_perl source file contains the "//fsm period" option, fsm_perl will generate a synthesis script to compile the design with clock constraints. Both "//fsm period" and "//fsm clock" affect the "create_clock" command that is put into the synthesis script, as shown in figure 10.

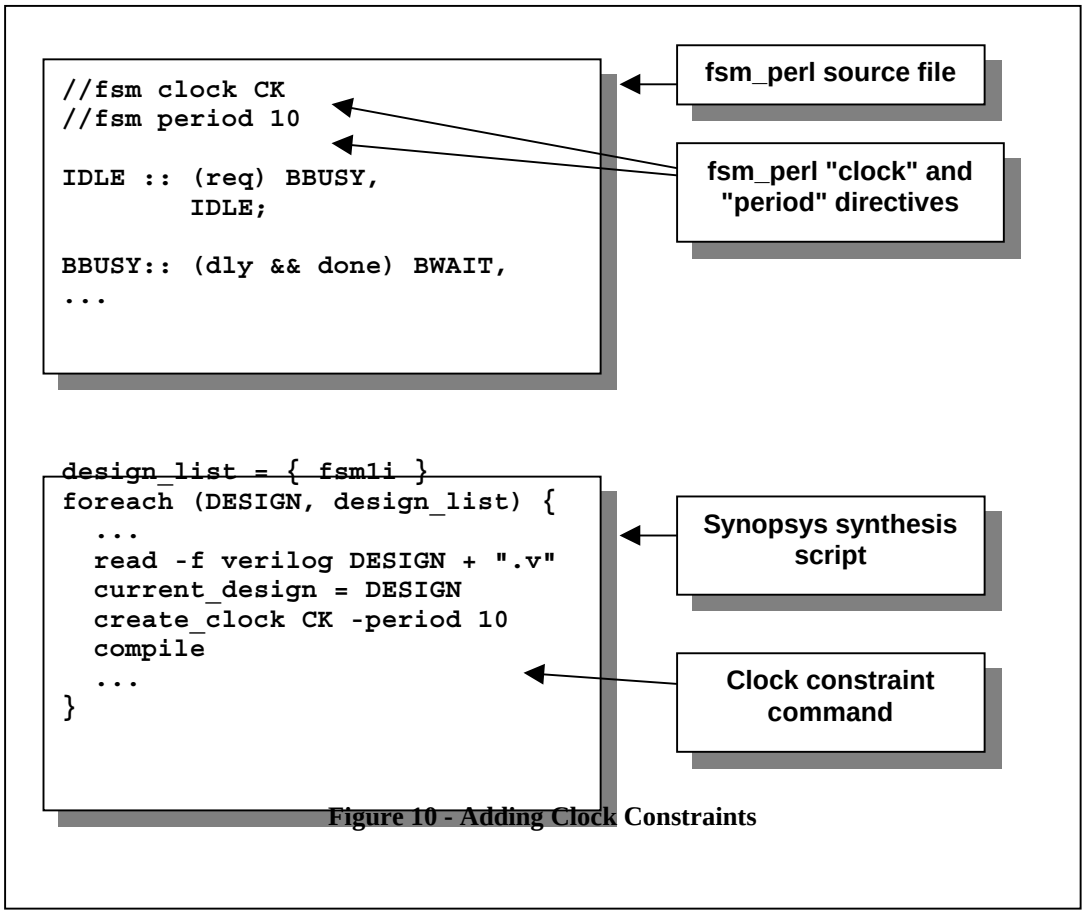


Figure 10 - Adding Clock Constraints

10.0 **Download fsm_perl**

This paper and the fsm_perl script are available for download at the Sunburst Design web site:

www.sunburst-design.com

The fsm_perl source code contains the GNU copyright header that permits free distribution of the fsm_perl code as long as the copyright header is included and remains unchanged.

Both fsm_perl and the paper can be freely downloaded from the Sunburst Design web site. Enhancement requests can be sent to **cliffc@sunburst-design.com**. The subject line should contain "**fsm_perl enhancement request**".

10.1 Fsm_perl Development & Enhancements

Fsm_perl was first created with the intent of simplifying the task of generating Verilog source code for simulation and synthesis. Later, other capabilities were added to generate state machines using different FSM state-encoding styles and different Verilog coding styles. The next step was to permit the creation of multiple Verilog files with different coding styles and an accompanying Synopsys synthesis script to permit easy experimentation with Verilog styles, Synopsys switches and to report the various area and timing results. The latter capability greatly accelerates the selection of an optimal coding style and synthesis strategy.

One enhancement in progress is the generation of all fsm_perl Verilog output files and a synthesis script to compile and report results from all coding styles. This enhancement will facilitate selection of the best coding style for a design project.

A one-hot output registered coding style permits the generation of non-glitching registered outputs. This is another enhancement under present consideration.

With the release of the Synopsys 1999.05 TCL interface, generation of a TCL script output file might also be a valuable future enhancement.

Another potential enhancement would be the generation of synthesizable VHDL code from fsm_perl source code. This should require little more than an alternate code generator.

As the art of FSM design using HDLs progresses, it is anticipated that additional styles will be generated by fsm_perl.

11.0 Conclusion

The fsm_perl syntax is short, simple to generate and easy to understand. Fsm_perl removes much of the tedious effort associated with the entry of synthesizable Verilog FSM code, plus fsm_perl generates the Synopsys script that is necessary to compile and examine the synthesized FSM design.

Author & Contact Information

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

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

language. Mr. Cummings is also a member of the IEEE Verilog Synthesis Interoperability Working Group.

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

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

(Data accurate as of September 7th, 2001)

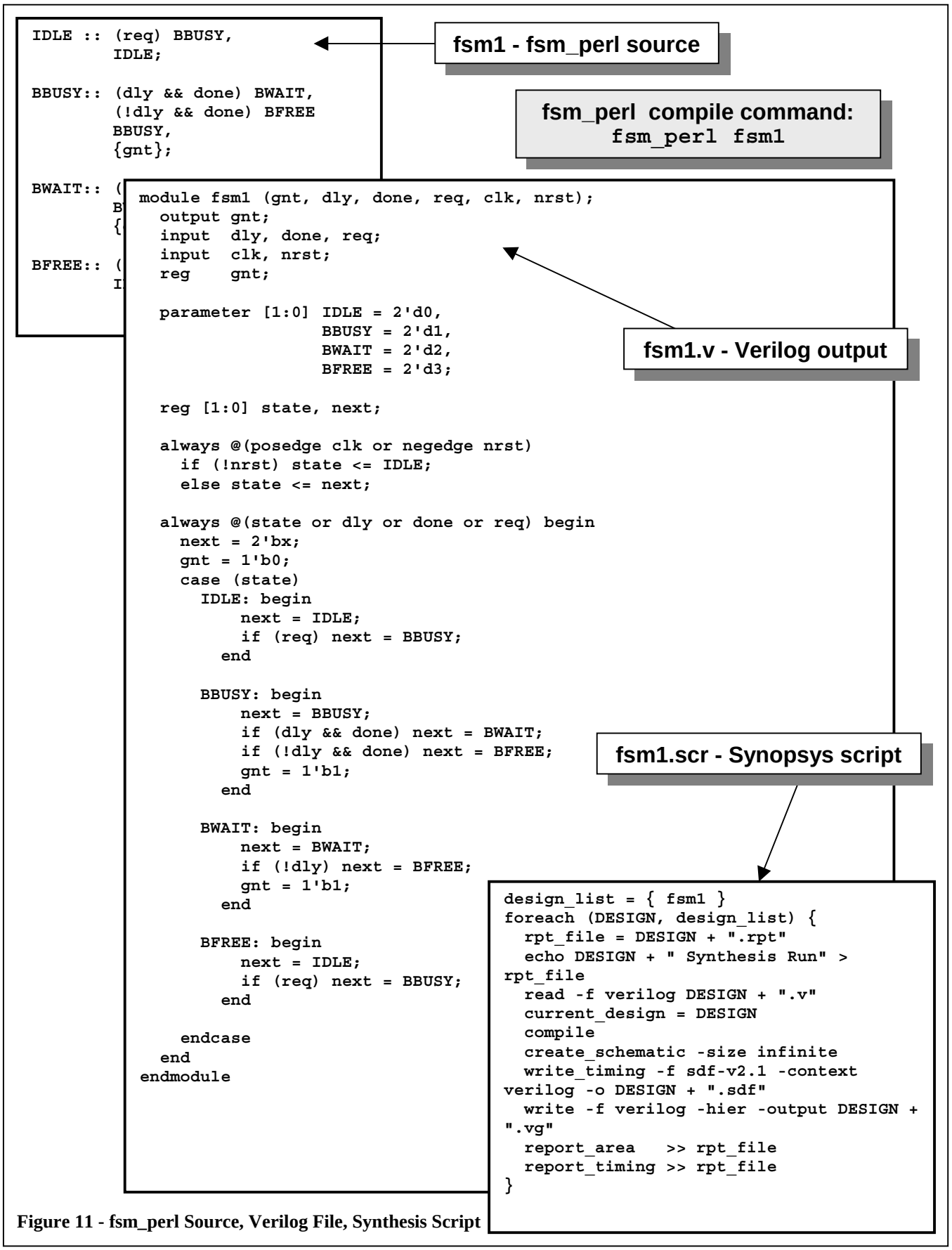


Figure 11 - fsm_perl Source, Verilog File, Synthesis Script

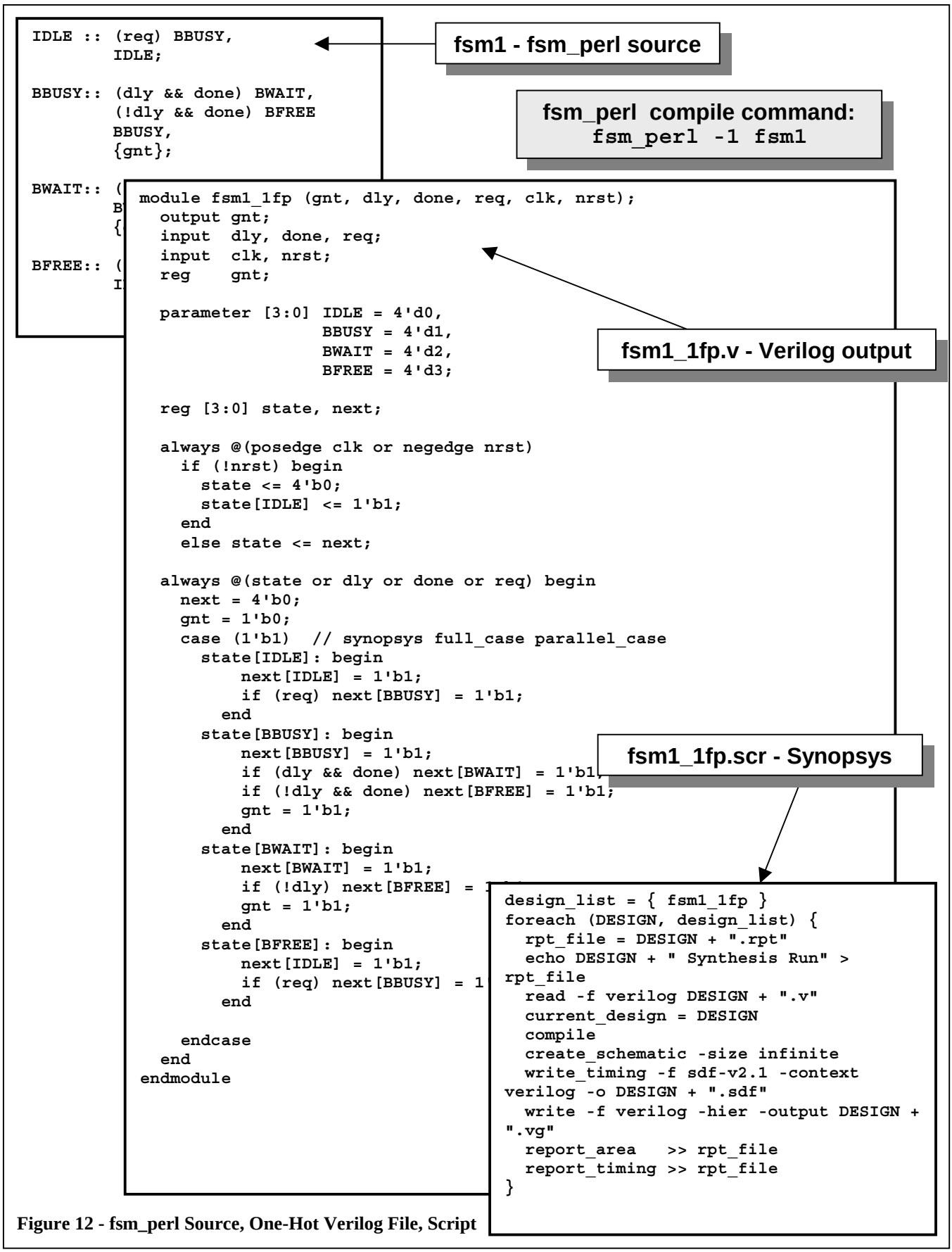


Figure 12 - fsm_perl Source, One-Hot Verilog File, Script

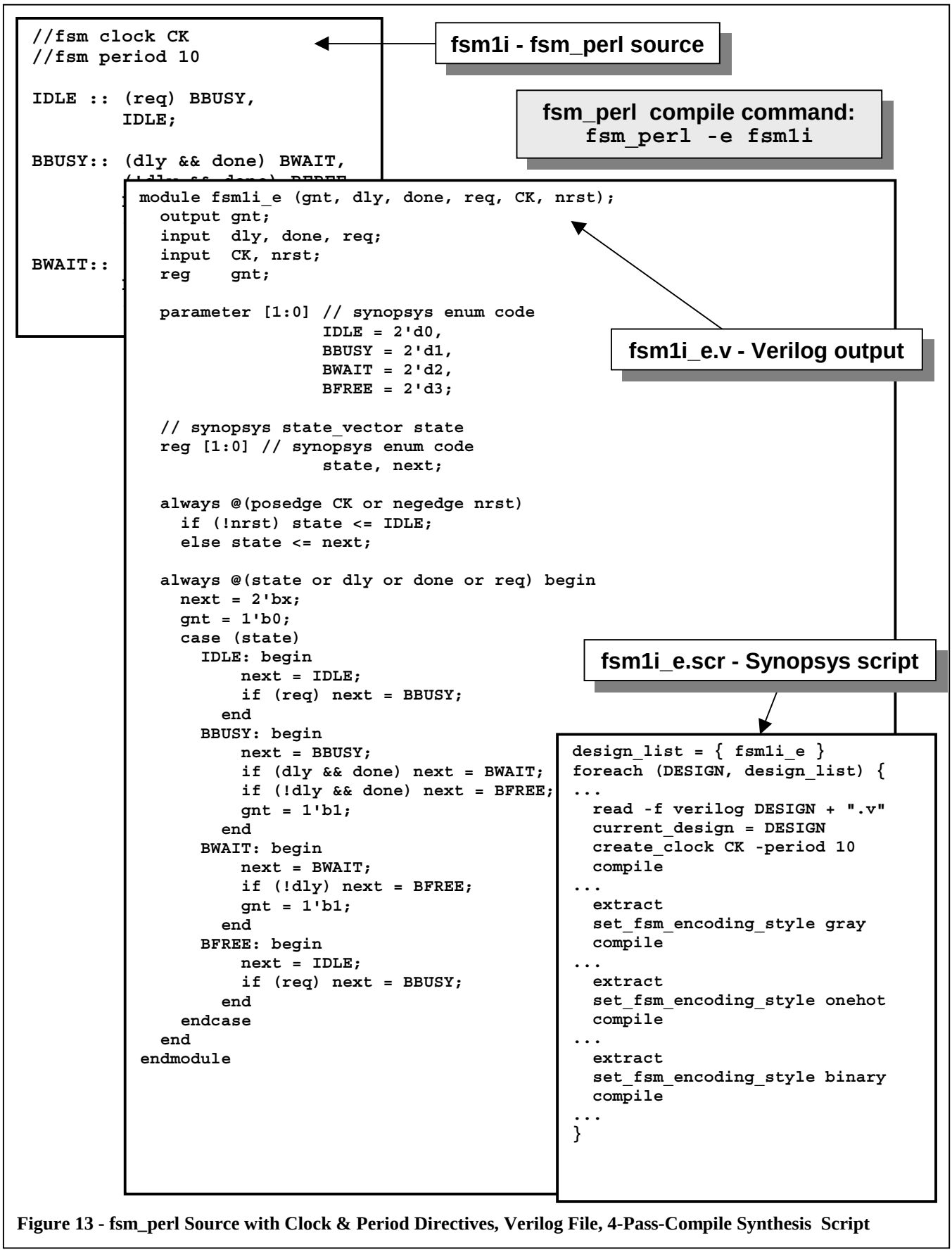
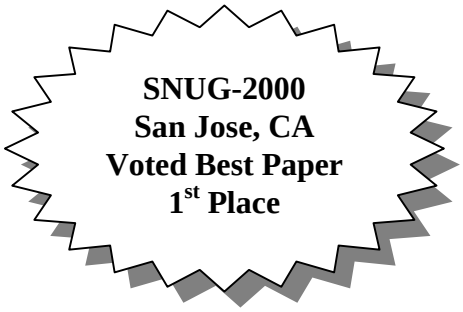


Figure 13 - fsm_perl Source with Clock & Period Directives, Verilog File, 4-Pass-Compile Synthesis Script

Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!

Clifford E. Cummings

Sunburst Design, Inc.



ABSTRACT

One of the most misunderstood constructs in the Verilog language is the nonblocking assignment. Even very experienced Verilog designers do not fully understand how nonblocking assignments are scheduled in an IEEE compliant Verilog simulator and do not understand when and why nonblocking assignments should be used. This paper details how Verilog blocking and nonblocking assignments are scheduled, gives important coding guidelines to infer correct synthesizable logic and details coding styles to avoid Verilog simulation race conditions.

1.0 Introduction

Two well known Verilog coding guidelines for modeling logic are:

- Guideline: Use blocking assignments in always blocks that are written to generate combinational logic [1].
- Guideline: Use nonblocking assignments in always blocks that are written to generate sequential logic [1].

But why? In general, the answer is simulation related. Ignoring the above guidelines can still infer the correct synthesized logic, but the pre-synthesis simulation might not match the behavior of the synthesized circuit.

To understand the reasons behind the above guidelines, one needs to have a full understanding of the functionality and scheduling of Verilog blocking and nonblocking assignments. This paper will detail the functionality and scheduling of blocking and nonblocking assignments.

Throughout this paper, the following abbreviations will be used:
RHS - the expression or variable on the right-hand-side of an equation will be abbreviated as RHS equation, RHS expression or RHS variable.
LHS - the expression or variable on the left-hand-side of an equation will be abbreviated as LHS equation, LHS expression or LHS variable.

2.0 Verilog race conditions

The IEEE Verilog Standard [2] defines: which statements have a guaranteed order of execution ("Determinism", section 5.4.1), and which statements do not have a guaranteed order of execution ("Nondeterminism", section 5.4.2 & "Race conditions", section 5.5).

A Verilog race condition occurs when two or more statements that are scheduled to execute in the same simulation time-step, would give different results when the order of statement execution is changed, as permitted by the IEEE Verilog Standard.

To avoid race conditions, it is important to understand the scheduling of Verilog blocking and nonblocking assignments.

3.0 Blocking assignments

The blocking assignment operator is an equal sign ("="). A blocking assignment gets its name because a blocking assignment must evaluate the RHS arguments and complete the assignment without interruption from any other Verilog statement. The assignment is said to "block" other assignments until the current assignment has completed. The one exception is a blocking assignment with timing delays on the RHS of the blocking operator, which is considered to be a poor coding style [3].

Execution of blocking assignments can be viewed as a one-step process:
1. Evaluate the RHS (right-hand side equation) and update the LHS (left-hand side expression) of the blocking assignment without interruption from any other Verilog statement.

A blocking assignment "blocks" trailing assignments in the same always block from occurring until after the current assignment has been completed

A problem with blocking assignments occurs when the RHS variable of one assignment in one procedural block is also the LHS variable of another assignment in another procedural block and both equations are scheduled to execute in the same simulation time step, such as on the same clock edge. If blocking assignments are not properly ordered, a race condition can occur. When blocking assignments are scheduled to execute in the same time step, the order execution is unknown.

To illustrate this point, look at the Verilog code in Example 1.

```
module fbosc1 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 = 0;  // reset
    else     y1 = y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 = 1;  // preset
    else     y2 = y1;
endmodule
```

Example 1 - Feedback oscillator with blocking assignments

According to the IEEE Verilog Standard, the two always blocks can be scheduled in any order. If the first always block executes first after a reset, both y1 and y2 will take on the value of 1. If the second always block executes first after a reset, both y1 and y2 will take on the value 0. This clearly represents a Verilog race condition.

4.0 Nonblocking assignments

The nonblocking assignment operator is the same as the less-than-or-equal-to operator ("<="). A nonblocking assignment gets its name because the assignment evaluates the RHS expression of a nonblocking statement at the beginning of a time step and schedules the LHS update to take place at the end of the time step. Between evaluation of the RHS expression and update of the LHS expression, other Verilog statements can be evaluated and updated and the RHS expression of other Verilog nonblocking assignments can also be evaluated and LHS updates scheduled. The nonblocking assignment does not block other Verilog statements from being evaluated.

Execution of nonblocking assignments can be viewed as a two-step process:
1. Evaluate the RHS of nonblocking statements at the beginning of the time step.
2. Update the LHS of nonblocking statements at the end of the time step.

Nonblocking assignments are only made to register data types and are therefore only permitted inside of procedural blocks, such as initial blocks and always blocks. Nonblocking assignments are not permitted in continuous assignments.

To illustrate this point, look at the Verilog code in Example 2.

```
module fbosc2 (y1, y2, clk, rst);
  output y1, y2;
  input  clk, rst;
  reg    y1, y2;

  always @(posedge clk or posedge rst)
    if (rst) y1 <= 0;  // reset
    else     y1 <= y2;

  always @(posedge clk or posedge rst)
    if (rst) y2 <= 1;  // preset
    else     y2 <= y1;
endmodule
```

Example 2 - Feedback oscillator with nonblocking assignments

Again, according to the IEEE Verilog Standard, the two always blocks can be scheduled in any order. No matter which always block starts first after a reset, both nonblocking RHS expressions will be evaluated at the beginning of the time step and then both nonblocking LHS variables will be updated at the end of the same time step. From a users perspective, the execution of these two nonblocking statements happen in parallel.

5.0 Verilog coding guidelines

Before giving further explanation and examples of both blocking and nonblocking assignments, it would be useful to outline eight guidelines that help to accurately simulate hardware, modeled using Verilog. Adherence to these guidelines will also remove 90-100% of the Verilog race conditions encountered by most Verilog designers.

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

Guideline #6: Do not make assignments to the same variable from more than one always block.

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

Guideline #8: Do not make assignments using #0 delays.

Reasons for these guidelines are given throughout the rest of this paper. Designers new to Verilog are encouraged to memorize and use these guidelines until their underlying functionality is fully understood. Following these guidelines will help to avoid "death by Verilog!"

6.0 The Verilog "stratified event queue"

An examination of the Verilog "stratified event queue" (see Figure 1) helps to explain how Verilog blocking and nonblocking assignments function. The "stratified event queue" is a fancy name for the different Verilog event queues that are used to schedule simulation events.

The "stratified event queue" as described in the IEEE Verilog Standard is a conceptual model. Exactly how each vendor implements the event queues is proprietary, helps to determine the efficiency of each vendor's simulator and is not detailed in this paper.

As defined in section 5.3 of the IEEE 1364-1995 Verilog Standard, the "stratified event queue" is logically partitioned into four distinct queues for the current simulation time and additional queues for future simulation times.

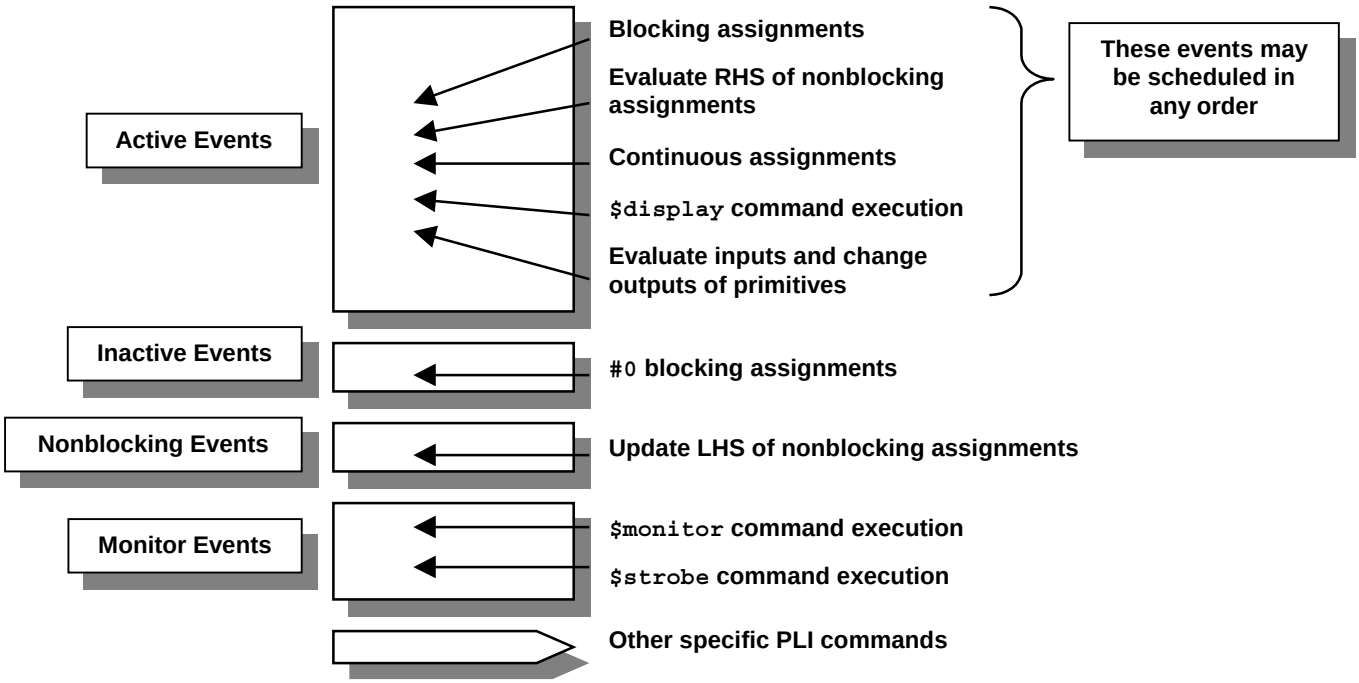


Figure 1 - Verilog "stratified event queue"

The *active events* queue is where most Verilog events are scheduled, including blocking assignments, continuous assignments, $display commands, evaluation of instance and primitive inputs followed by updates of primitive and instance outputs, and the evaluation of nonblocking RHS expressions. The LHS of nonblocking assignments are not updated in the *active events* queue.

Events are added to any of the event queues (within restrictions imposed by the IEEE Standard) but are only removed from the *active events* queue. Events that are scheduled on the other event queues will eventually become "activated," or promoted into the active events queue. Section 5.4 of the IEEE 1364-1995 Verilog Standard lists an algorithm that describes when the other event queues are "activated."

Two other commonly used event queues in the current simulation time are the *nonblocking assign updates* event queue and the *monitor* events queue, which are described below.

The *nonblocking assign updates* event queue is where updates to the LHS expression of nonblocking assignments are scheduled. The RHS expression is evaluated in random order at the beginning of a simulation time step along with the other *active events* described above.

The *monitor* events queue is where $strobe and $monitor display command values are scheduled. $strobe and $monitor show the updated values of all requested variables at the end of a simulation time step, after all other assignments for that simulation time step are complete.

A fourth event queue described in section 5.3 of the Verilog Standard is the *inactive* events queue, where #0-delayed assignments are scheduled. The practice of making #0-delay assignments is generally a flawed practice employed by designers who try to make assignments to the same variable from two separate procedural blocks, attempting to beat Verilog race conditions by scheduling one of the assignments to take place slightly later in the same simulation time step. Adding #0-delay assignments to Verilog models needlessly complicates the analysis of scheduled events. The author knows of no condition that requires making #0-delay assignments that could not be easily replaced with a different and more efficient coding style and hence discourages the practice.

Guideline #8: Do not make assignments using #0 delays.

The "stratified event queue" of Figure 1 will be frequently referenced to explain the behavior of Verilog code examples shown later in this paper. The event queues will also be referenced to justify the eight coding guidelines given in section 5.0.

7.0 Self-triggering always blocks

In general, a Verilog always block cannot trigger itself. Consider the oscillator example in Example 3. This oscillator uses blocking assignments. Blocking assignments evaluate their RHS expression and update their LHS value without interruption. The blocking assignment must complete before the **`@(clk)`** edge-trigger event can be scheduled. By the time the trigger event has been scheduled, the blocking **`clk`** assignment has completed; therefore, there is no trigger event from within the always block to trigger the **`@(clk)`** trigger.

```
module osc1 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk = ~clk;
endmodule
```

Example 3 - Non-self-triggering oscillator using blocking assignments

In contrast, the oscillator in Example 4 uses nonblocking assignments. After the first **`@(clk)`** trigger, the RHS expression of the nonblocking assignment is evaluated and the LHS value scheduled into the *nonblocking assign updates* event queue. Before the *nonblocking assign updates* event queue is "activated," the **`@(clk)`** trigger statement is encountered and the always block again becomes sensitive to changes on the **`clk`** signal. When the nonblocking LHS value is updated later in the same time step, the **`@(clk)`** is again triggered. The osc2 example is self-triggering (which is not necessarily a recommended coding style).

```
module osc2 (clk);
  output clk;
  reg    clk;

  initial #10 clk = 0;

  always @(clk) #10 clk <= ~clk;
endmodule
```

Example 4 - Self-triggering oscillator using nonblocking assignments

8.0 Pipeline modeling

Figure 2 shows a block diagram for a simple sequential pipeline register. Example 5 - Example 8 show four different ways that an engineer might choose to model this pipeline using blocking assignments.

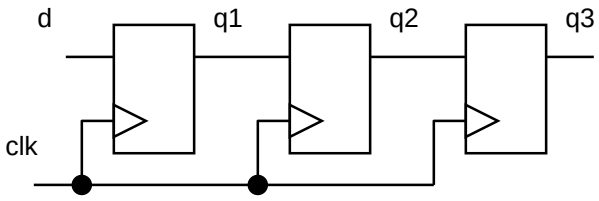


Figure 2 - Sequential pipeline register

In the pipeb1, Example 5 code, the sequentially ordered blocking assignments will cause the input value, **`d`**, to be placed on the output of every register on the next **`posedge clk`**. On every clock edge, the input value is transferred directly to the **`q3`**-output without delay. This clearly does not model a pipeline register and will actually synthesize to a single register! (See Figure 3).

```
module pipeb1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 = d;
    q2 = q1;
    q3 = q2;
  end
endmodule
```

Example 5 - Bad blocking-assignment sequential coding style #1

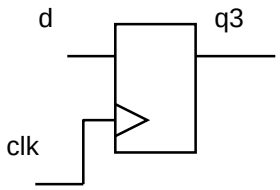


Figure 3 - Actual synthesized result!

In the pipeb2 example, the blocking assignments have been carefully ordered to cause the simulation to correctly behave like a pipeline register. This model also synthesizes to the pipeline register shown in Figure 2.

```
module pipeb2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 = q2;
    q2 = q1;
    q1 = d;
  end
endmodule
```

Example 6 - Bad blocking-assignment sequential coding style #2 - but it works!

In the pipeb3 example, the blocking assignments have been split into separate always blocks. Verilog is permitted to simulate the always blocks in any order, which might cause this pipeline simulation to be wrong. This is a Verilog race condition! Executing the always blocks in a different order yields a different result. However, this Verilog code will synthesize to the correct pipeline register. This means that there might be a mismatch between the pre-synthesis and post-synthesis simulations. The pipeb4 example, or any other ordering of the same always block statements will also synthesize to the correct pipeline logic, but might not simulate correctly.

```
module pipeb3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1=d;

  always @(posedge clk) q2=q1;

  always @(posedge clk) q3=q2;
endmodule
```

Example 7 - Bad blocking-assignment sequential coding style #3

```
module pipeb4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2=q1;
  always @(posedge clk) q3=q2;
  always @(posedge clk) q1=d;
endmodule
```

Example 8 - Bad blocking-assignment sequential coding style #4

If each of the four blocking-assignment examples is rewritten with nonblocking assignments, each will simulate correctly and synthesize the desired pipeline logic.

```
module pipen1 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
    q3 <= q2;
  end
endmodule
```

Example 9 - Good nonblocking-assignment sequential coding style #1

```
module pipen2 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) begin
    q3 <= q2;
    q2 <= q1;
    q1 <= d;
  end
endmodule
```

Example 10 - Good nonblocking-assignment sequential coding style #2

```
module pipen3 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q1<=d;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;
endmodule
```

Example 11 - Good nonblocking-assignment sequential coding style #3

```
module pipen4 (q3, d, clk);
  output [7:0] q3;
  input  [7:0] d;
  input        clk;
  reg    [7:0] q3, q2, q1;

  always @(posedge clk) q2<=q1;

  always @(posedge clk) q3<=q2;

  always @(posedge clk) q1<=d;
endmodule
```

Example 12 - Good nonblocking-assignment sequential coding style #4

Upon examination of the pipeline coding styles shown in this section:

- 1 out of 4 blocking assignment coding styles was guaranteed to simulate correctly
- 3 out of 4 blocking assignment coding styles were guaranteed to synthesize correctly
- 4 out of 4 nonblocking assignment coding styles were guaranteed to simulate correctly
- 4 out of 4 blocking assignment coding styles were guaranteed to synthesize correctly.

Although, if confined to one always block with carefully sequenced assignments, it was possible to code the pipeline logic using blocking assignments. On the other hand, it was easy to code the same pipeline logic using nonblocking assignments; indeed, the nonblocking assignment coding styles all would work for both synthesis and simulation.

9.0 Blocking assignments & simple examples

There are many Verilog and Verilog synthesis books that show simple sequential examples that are successfully coded using blocking assignments. Example 13 shows a flipflop model that appears in most Verilog text books.

```
module dffb (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q = 1'b0;
    else     q = d;
endmodule
```

Example 13 - Simple flawed blocking-assignment D-flipflop model - but it works!

If an engineer is willing to limit all modules to a single always block, blocking assignments can be used to correctly model, simulate and synthesize the desired logic. Unfortunately this reasoning leads to the habit of placing blocking assignments in other, more complex sequential always blocks that will exhibit the race conditions already detailed in this paper.

```
module dffx (q, d, clk, rst);
  output q;
  input  d, clk, rst;
  reg    q;

  always @(posedge clk)
    if (rst) q <= 1'b0;
    else     q <= d;
endmodule
```

Example 14 - Preferred D-flipflop coding style with nonblocking assignments

It is better to develop the habit of coding all sequential always blocks, even simple single-block modules, using nonblocking assignments as shown in Example 14.

Now consider a more complex piece of sequential logic, a Linear Feedback Shift-Register or LFSR.

10.0 Sequential feedback modeling

A Linear Feedback Shift-Register (LFSR) is a piece of sequential logic with a feedback loop. The feedback loop poses a problem for engineers attempting to code this piece of sequential logic with correctly ordered blocking assignments as shown in Example 15.

```
module lfsrb1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 = 1'b1;
      q2 = 1'b1;
      q1 = 1'b1;
    end
    else begin
      q3 = q2;
      q2 = n1;
      q1 = q3;
    end
endmodule
```

Example 15 - Non-functional LFSR with blocking assignments

There is no way to order the assignments in Example 15 to model the feedback loop unless a temporary variable is used.

One could group all of the assignments into one-line equations to avoid using a temporary variable, as shown in Example 16, but the code is now more cryptic. For larger examples, one-line equations might become very difficult to code and debug. One-line equation coding styles are not necessarily encouraged.

```
module lfsrb2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} = 3'b111;
    else        {q3,q2,q1} = {q2,(q1^q3),q3};
endmodule
```

Example 16 - Functional but cryptic LFSR with blocking assignments

If the blocking assignments in Example 15 and Example 16 are replaced with nonblocking assignments as shown in Example 17 and Example 18, all simulations function as would be expected from an LFSR.

```
module lfsrn1 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;
  wire   n1;

  assign n1 = q1 ^ q3;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) begin
      q3 <= 1'b1;
      q2 <= 1'b1;
      q1 <= 1'b1;
    end
    else begin
      q3 <= q2;
      q2 <= n1;
      q1 <= q3;
    end
endmodule
```

Example 17 - Functional LFSR with nonblocking assignments

```
module lfsrn2 (q3, clk, pre_n);
  output q3;
  input  clk, pre_n;
  reg    q3, q2, q1;

  always @(posedge clk or negedge pre_n)
    if (!pre_n) {q3,q2,q1} <= 3'b111;
    else        {q3,q2,q1} <= {q2,(q1^q3),q3};
endmodule
```

Example 18 - Functional but cryptic LFSR with nonblocking assignments

Based on the pipeline examples of section 8.0 and the LFSR examples of section 10.0, it is recommended to model all sequential logic using nonblocking assignments. A similar analysis would show that it is also safest to use nonblocking assignments to model latches

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

11.0 Combinational logic - use blocking assignments

There are many ways to code combinational logic using Verilog, but when the combinational logic is coded using an always block, blocking assignments should be used.

If only a single assignment is made in the always block, using either blocking or nonblocking assignments will work; but in the interest of developing good coding habits one should always using blocking assignments to code combinational logic.

It has been suggested by some Verilog designers that nonblocking assignments should not only be used for coding sequential logic, but also for coding combinational logic. For coding simple combinational always blocks this would work, but if multiple assignments are included in the always block, such as the and-or code shown in Example 19, using nonblocking assignments with no delays will either simulate incorrectly, or require additional sensitivity list entries and multiple passes through the always block to simulate correctly. The latter would be inefficient from a simulation time perspective.

The code shown in Example 19 builds the `y`-output from three sequentially executed statements. Since nonblocking assignments evaluate the RHS expressions before updating the LHS variables, the values of `tmp1` and `tmp2` were the original values of these two variables upon entry to this always block and not the values that will be updated at the end of the simulation time step. The `y`-output will reflect the old values of `tmp1` and `tmp2`, not the values calculated in the current pass of the always block.

```
module ao4 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

Example 19 - Bad combinational logic coding style using nonblocking assignments

The code shown in Example 20 is identical to the code shown in Example 19, except that **tmp1** and **tmp2** have been added to the sensitivity list. As describe in section 7.0, when the nonblocking assignments update the LHS variables in the *nonblocking assign update* events queue, the always block will self-trigger and update the **y**-outputs with the newly calculated **tmp1** and **tmp2** values. The **y**-output value will now be correct after taking two passes through the always block. Multiple passes through an always block equates to degraded simulation performance and should be avoided if a reasonable alternative exists.

```
module ao5 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d or tmp1 or tmp2) begin
    tmp1 <= a & b;
    tmp2 <= c & d;
    y    <= tmp1 | tmp2;
  end
endmodule
```

Example 20 - Inefficient multi-pass combinational logic coding style with nonblocking assignments

A better habit to develop, one that does not require multiple passes through an always block, is to only use blocking assignments in always blocks that are written to model combinational logic.

```
module ao2 (y, a, b, c, d);
  output y;
  input  a, b, c, d;
  reg    y, tmp1, tmp2;

  always @(a or b or c or d) begin
    tmp1 = a & b;
    tmp2 = c & d;
    y    = tmp1 | tmp2;
  end
endmodule
```

Example 21 - Efficient combinational logic coding style using blocking assignments

The code in Example 21 is identical to the code in Example 19, except that the nonblocking assignments have been replaced with blocking assignments, which will guarantee that the **y**-output assumes the correct value after only one pass through the always block; hence the following guideline:

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

12.0 Mixed sequential & combinational logic - use nonblocking assignments

It is sometimes convenient to group simple combinational equations with sequential logic equations. When combining combinational and sequential code into a single always block, code the always block as a sequential always block with nonblocking assignments as shown in Example 22.

```
module nbex2 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= a ^ b;
endmodule
```

Example 22 - Combinational and sequential logic in a single always block

The same logic that is implemented in Example 22 could also be implemented as two separate always blocks, one with purely combinational logic coded with blocking assignments and one with purely sequential logic coded with nonblocking assignments as shown in Example 23.

```
module nbex1 (q, a, b, clk, rst_n);
  output q;
  input  clk, rst_n;
  input  a, b;
  reg    q, y;

  always @(a or b)
    y = a ^ b;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= y;
endmodule
```

Example 23 - Combinational and sequential logic separated into two always blocks

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

13.0 Other mixed blocking & nonblocking assignment guidelines

Verilog permits blocking and nonblocking assignments to be freely mixed inside of an always block. In general, mixing blocking and nonblocking assignments in the same always block is a poor coding style, even if Verilog permits it.

The code in Example 24 will both simulate and synthesize correctly because the blocking assignment is not made to the same variable as the nonblocking assignments. Although this will work, the author discourages this coding style.

```
module ba_nba2 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q;

  always @(posedge clk or negedge rst_n) begin: ff
    reg tmp;
    if (!rst_n) q <= 1'b0;
    else begin
      tmp = a & b;
      q <= tmp;
    end
  end
endmodule
```

Example 24 - Blocking and nonblocking assignment in the same always block - generally a bad idea!

The code in Example 25 will most likely simulate correctly most of the time, but Synopsys tools will report a syntax error because the blocking assignment is assigned to the same variable as one of the nonblocking assignments. This code must be modified to be synthesizable.

```
module ba_nba6 (q, a, b, clk, rst_n);
  output q;
  input  a, b, rst_n;
  input  clk;
  reg    q, tmp;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q = 1'b0;  // blocking assignment to "q"
    else begin
      tmp = a & b;
      q <= tmp;            // nonblocking assignment to "q"
    end
endmodule
```

Example 25 - Synthesis syntax error - blocking and nonblocking assignment to the same variable

As a matter of forming good coding habits, the author encourages adherence to the following:

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

14.0 Multiple assignments to the same variable

Making multiple assignments to the same variable from more than one always block is a Verilog race condition, even when using nonblocking assignments.

In Example 26, two always blocks are making assignments to the `q`-output, both using nonblocking assignments. Since these always blocks can be scheduled in an order, the simulation output is a race condition.

```
module badcode1 (q, d1, d2, clk, rst_n);
  output q;
  input  d1, d2, clk, rst_n;
  reg    q;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d1;

  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d2;
endmodule
```

Example 26 - Race condition coding style using nonblocking assignments

When Synopsys tools read this type of coding example, the following warning message is issued:

```
Warning: In design 'badcode1', there is 1 multiple-driver net with
unknown wired-logic type.
```

When the warning is ignored and the code from Example 26 is compiled, two flipflops with outputs feeding a 2-input and gate are inferred. The pre-synthesis simulation does not even closely match the post-synthesis simulation in this example.

Guideline #6: Do not make assignments to the same variable from more than one always block.

15.0 Common nonblocking myths

15.1 Nonblocking assignments & $display

Myth: "Using the $display command with nonblocking assignments does not work"

Truth: Nonblocking assignments are updated after all $display commands

```
module display_cmds;
  reg a;

  initial $monitor("\$monitor: a = %b", a);

  initial begin
    $strobe  ("\$strobe : a = %b", a);
    a = 0;
    a <= 1;
    $display ("\$display: a = %b", a);
    #1 $finish;
  end
endmodule
```

The below output from the above simulation shows that the **`$display`** command was executed in the *active events* queue, before the *nonblocking assign update* events were executed.

```
$display: a = 0
$monitor: a = 1
$strobe : a = 1
```

15.2 #0-delay assignments

Myth: “#0 forces an assignment to the end of a time step”

Truth: #0 forces an assignment to the "inactive events queue"

```
module nb_schedule1;
  reg a, b;

  initial begin
    a = 0;
    b = 1;
    a <= b;
    b <= a;

       $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
    #0 $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $monitor ("%0dns: \$monitor: a=%b  b=%b", $stime, a, b);
       $display ("%0dns: \$display: a=%b  b=%b", $stime, a, b);
       $strobe  ("%0dns: \$strobe : a=%b  b=%b\n", $stime, a, b);
       $display ("%0dns: #0      : a=%b  b=%b", $stime, a, b);

    #1 $finish;
  end
endmodule
```

The below output from the simulation on the previous page shows that the #0-delay command was executed in the *inactive events* queue, before the *nonblocking assign update* events were executed.

```
0ns: $display: a=0  b=1
0ns: #0      : a=0  b=1
0ns: $monitor: a=1  b=0
0ns: $strobe : a=1  b=0

1ns: $display: a=1  b=0
1ns: #0      : a=1  b=0
1ns: $monitor: a=1  b=0
1ns: $strobe : a=1  b=0
```

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

15.3 Multiple nonblocking assignments to the same variable

Myth: “Making multiple nonblocking assignments to the same variable in the same always block is undefined”

Truth: Making multiple nonblocking assignments to the same variable in the same always block is defined by the Verilog Standard. The last nonblocking assignment to the same variable wins! Quoting from the IEEE 1364-1995 Verilog Standard [2], pg. 47, section 5.4.1 - Determinism

> "Nonblocking assignments shall be performed in the order the statements were executed. Consider the following example:
>
> ```
> initial begin
>     a <= 0;
>     a <= 1;
> end
> ```
>
> When this block is executed, there will be two events added to the nonblocking assign update queue. The previous rule requires that they be entered on the queue in source order; this rule requires that they be taken from the queue and performed in source order as well. Hence, at the end of time-step 1, the variable a will be assigned `0` and then `1`."

Translation: “Last nonblocking assignment wins!”

Summary of guidelines and conclusions

All of the guidelines detailed in this paper are list below:

Guideline #1: When modeling sequential logic, use nonblocking assignments.

Guideline #2: When modeling latches, use nonblocking assignments.

Guideline #3: When modeling combinational logic with an always block, use blocking assignments.

Guideline #4: When modeling both sequential and combinational logic within the same always block, use nonblocking assignments.

Guideline #5: Do not mix blocking and nonblocking assignments in the same always block.

Guideline #6: Do not make assignments to the same variable from more than one always block.

Guideline #7: Use $strobe to display values that have been assigned using nonblocking assignments.

Guideline #8: Do not make assignments using #0 delays.

Conclusion: Following the above guidelines will accurately model synthesizable hardware while eliminating 90-100% of the most common Verilog simulation race conditions.

16.0 Final note: the spelling of "nonblocking"

The word nonblocking is frequently misspelled as "non-blocking." The author believes this is the Microsoftization of the word. Engineers have inserted a dash between "non" and "blocking" to satisfy Microsoft, as well as other, spell checkers. The correct spelling of the word as noted in the IEEE 1364-1995 Verilog Standard is in fact: nonblocking.

References

[1] IEEE P1364.1 Draft Standard For Verilog Register Transfer Level Synthesis

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

[3] Clifford Cummings, "Correct Methods For Adding Delays To Verilog Behavioral Models," *International HDL Conference 1999 Proceedings*, pp. 23-29, April 1999.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 18 years of ASIC, FPGA and system design experience and eight years of Verilog, synthesis and methodology training experience.

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

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

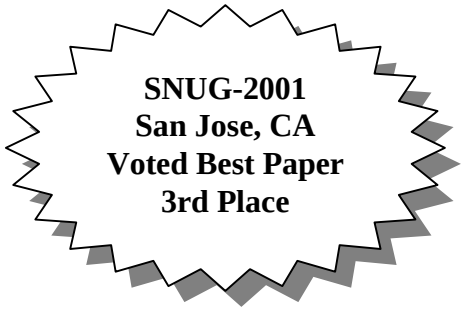
E-mail Address: cliffc@sunburst-design.com
This paper can be downloaded from the web site: www.sunburst-design.com/papers

(Data accurate as of October 9th, 2000)

Synthesis and Scripting Techniques for Designing Multi-Asynchronous Clock Designs

Clifford E. Cummings

Sunburst Design, Inc.



ABSTRACT

Designing a pure, one-clock synchronous design is a luxury that few ASIC designers will ever know. Most of the ASICs that are ever designed are driven by multiple asynchronous clocks and require special data, control-signal and verification handling to insure the timely completion of a robust working design.

1.0 Introduction

Most college courses teach engineering students prescribed techniques for designing completely synchronous (single clock) logic. In the real ASIC design world, there are very few single clock designs. This paper will detail some of the hardware design, timing analysis, synthesis and simulation methodologies to address multi-clock designs.

This paper is not intended to provide exhaustive coverage of this topic, but is presented to share techniques learned from experience.

2.0 Metastability

Quoting from Dally and Poulton's book[1] concerning metastability:

> "When sampling a changing data signal with a clock ... the order of the events determines the outcome. The smaller the time difference between the events, the longer it takes to determine which came first. When two events occur very close together, the decision process can take longer than the time allotted, and a synchronization failure occurs."

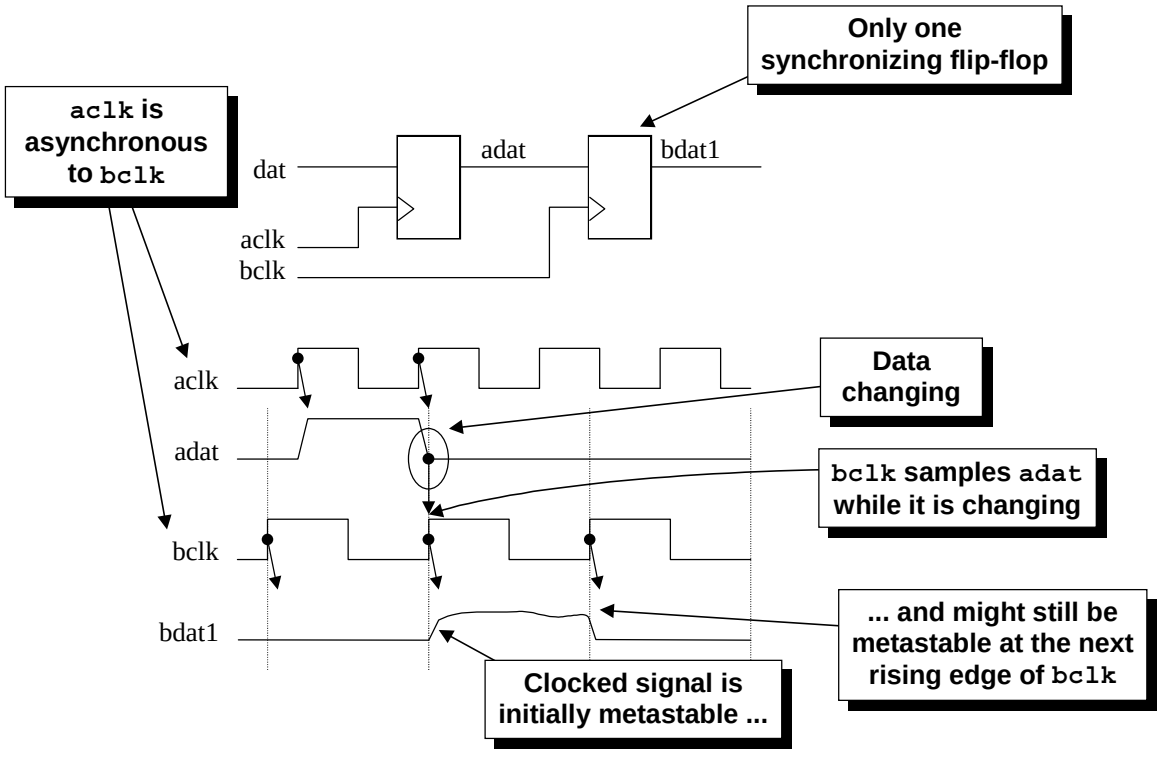


Figure 1 - Asynchronous clocks and synchronization failure

Figure 1 shows a synchronization failure that occurs when a signal generated in one clock domain is sampled too close to the rising edge of a clock signal from another clock domain.

Synchronization failure is caused by an output going metastable and not converging to a legal stable state by the time the output must be sampled again. Figure 2 shows that a metastable output can cause illegal signal values to be propagated throughout the rest of the design.

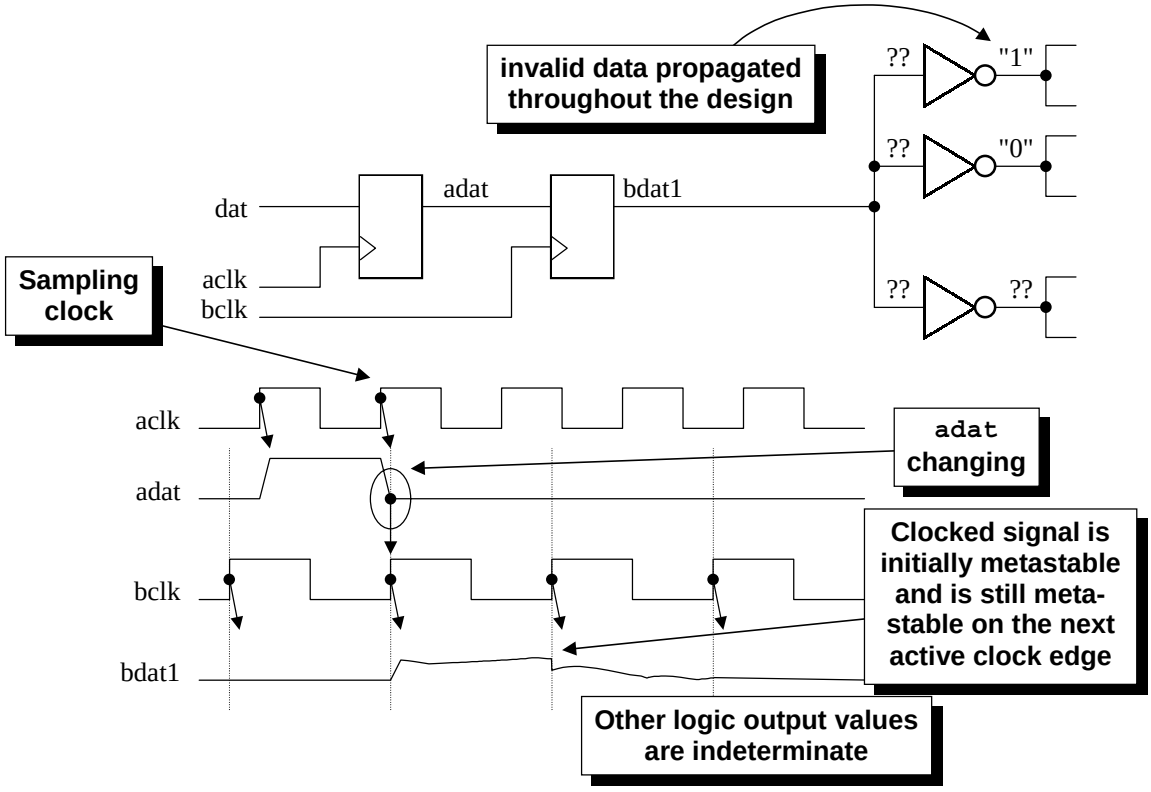


Figure 2 - Metastable `bdat1` output propagating invalid data throughout the design

Every flip-flop that is used in any design has a specified setup and hold time, or the time in which the data input is not legally permitted to change before and after a rising clock edge. This time window is specified as a design parameter precisely to keep a data signal from changing too close to another synchronizing signal that could cause the output to go metastable.

The metastable output problem shown in Figure 2 is sometimes known as the John Cooley ESNUG effect, or in other words, the propagation of unwanted information!
(Just kidding, John! ☺)

3.0 Synchronizers

Quoting again from Dally and Poulton[2] concerning synchronizers:

> "A synchronizer is a device that samples an asynchronous signal and outputs a version of the signal that has transitions synchronized to a local or sample clock."

The most common synchronizer used by digital designers is a two-flip-flop synchronizer as shown in Figure 3.

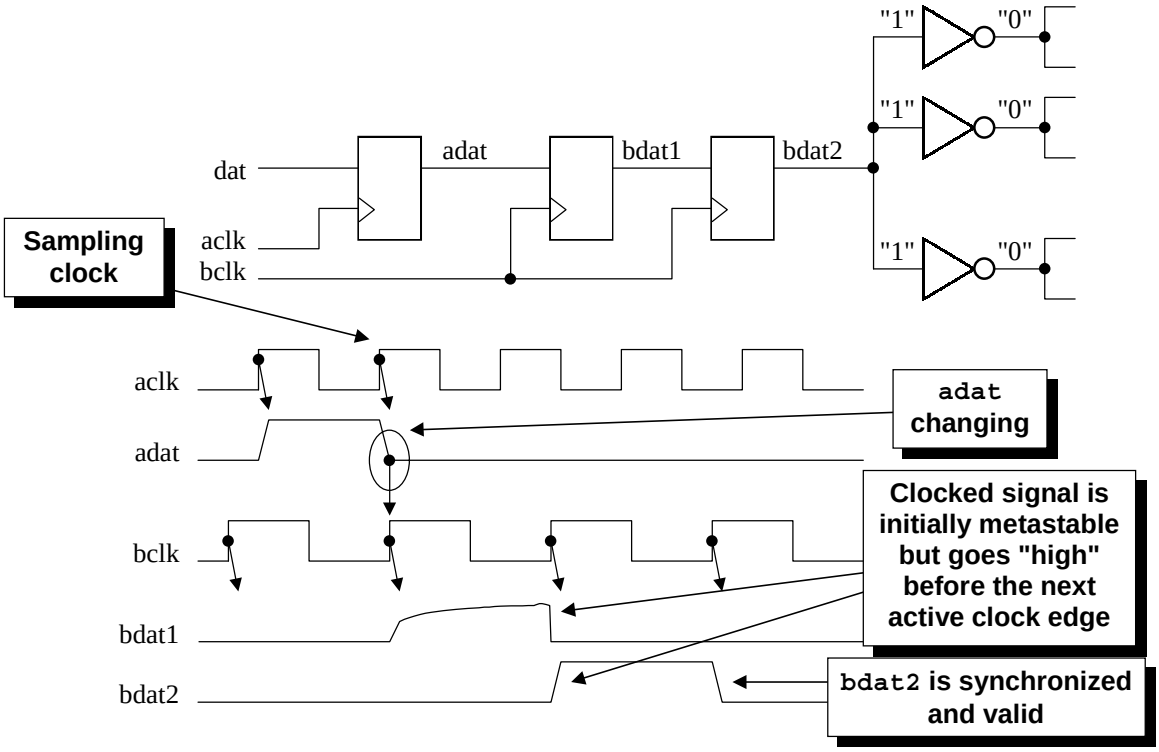


Figure 3 - Two flip-flop synchronizer

The first flip-flop samples the asynchronous input signal into the new clock domain and waits for a full clock cycle to permit any metastability on the stage-1 output signal to decay, then the stage-1 signal is sampled by the same clock into a second stage flip-flop, with the intended goal that the stage-2 signal is now a stable and valid signal synchronized into the new clock domain.

It is theoretically possible for the stage-1 signal to still be sufficiently metastable by the time the signal is clocked into the second stage to cause the stage-2 signal to also go metastable. The calculation of the probability of the time between synchronization failures (MTBF) is a function of multiple variables including the clock frequencies used to generate the input signal and to clock the synchronizing flip-flops. One description of the MTBF calculation can be found in Dally and Poulton[3].

For most synchronization applications, the two flip-flop synchronizer is sufficient to remove all likely metastability.

4.0 Static Timing Analysis

Performing static timing analysis is the process of verifying that every signal path in a design meets required clock-cycle timing, whether or not all of the signal paths are even possible. Static timing analysis is not used to verify the functionality of the design, only that the design meets timing goals. In theory, timing verification could be accomplished by running exhaustive gate-level simulations with SDF backannotation of actual timing values after a design is placed and routed. This is often referred to as dynamic timing verification.

Static timing analysis has three principal advantages over dynamic timing verification: (1) static timing analysis tools verify every single path between any two sequential elements, (2) static timing analysis does not require the generation of any test vectors, and (3) static timing analysis tools are orders of magnitude faster than trying to do timing verification running exhaustive gate-level simulations[4].

Timing analysis using Synopsys tools on a completely synchronous design is relatively easy to perform using either DesignTime within the Synopsys Design Compiler or Design Analyzer environments, or by using PrimeTime.

Timing analysis on modules with two or more asynchronous clocks is error prone, more difficult and can be time consuming. Static timing analysis on signals generated from one clock domain and latched into sequential elements within a second, asynchronous clock domain is inaccurate and for the most part worthless. The timing information for a signal latched by a clock that is asynchronous to the latched signal is inaccurate because the phase relationship between the signal and the asynchronous clock is always changing; therefore, the static timing analysis tool would have to check an infinite number of phase relationships between the signal and asynchronous clock. The fact is, one must assume that signals that pass from one clock domain to another at some point will violate either setup or hold times on the destination sequential element.

There is no good reason to perform timing analysis on signals that are generated in one clock domain and registered in another asynchronous clock domain. It is a given that these signals DO violate setup and hold times on the destination register. This is why synchronizers (see section 3.0) are needed, to alleviate the problems that can occur when a signal is passed from one clock domain to another.

For RTL modules that have two or more asynchronous clocks as inputs, a designer will be required to indicate to the static timing analysis tool which signal paths should be ignored. This is accomplished by "setting false paths" on signals that cross from one clock domain to another. This can be a tedious and error prone job unless the guidelines in the next two sections are followed.

5.0 Clock Naming Conventions

Guideline: Use a clock naming convention to identify the clock source of every signal in a design.
Reason: A naming convention helps all team members to identify the clock domain for every signal in a design and also makes grouping of signals for timing analysis easier to do using regular expression "wild-carding" from within a synthesis script.

A number of useful clock naming conventions have been used by various design teams. One that was used by design engineers in 1995 while designing video ASICs for In Focus projectors required that a leading prefix character be used to identify the various asynchronous clock domains. Examples included: uClk for the microprocessor clock, vClk for the video clock and dClk for the display clock.

Each signal was synchronized to one of the clock domains in the design and each signal-name had to include a prefix character identifying the clock domain for that signal. Any signal that was clocked by the uClk would have a u-prefix in the signal name, such as uaddr, udata, uwrite, etc. Any signal that was clocked by the vClk would similarly have a v-prefix in the signal name, such as vdata, vhsync, vframe, etc. The same signal naming convention was used for all signals generated by any of the other clocks in the design.

Using this technique, any engineer on the ASIC design team could easily identify the clock-domain source of any signal in the design and either use the signals directly or pass the signals through a synchronizer so that they could be used within a new clock domain.

The naming convention alone contributed significantly to the productivity of the design team. How do we know there was a productivity gain? One of the design engineers started his part of the ASIC design using his own naming convention, ignoring the convention in use by the other design team members. After much confusion about the signals entering and leaving his design partition, a team meeting was called and the non-compliant designer was "strongly encouraged" to rename the signals in his part of the design to conform to the team naming convention. After the signal names were changed, it became easier to interface to the partition in question. Fewer questions and less confusions occurred after the change.

6.0 Design Partitioning

Guideline: Only allow one clock per module.
Reason: Static timing analysis and creating synthesis scripts is more easily accomplished on single-clock modules or groups of single-clock modules.

Guideline: Create a synchronizer module for each set of signals that pass from just one clock domain into another clock domain.
Reason: It is given that any signal passing from one clock domain to another clock domain is going to have setup and hold time problems. No worst-case (max time) timing analysis is required for synchronizer modules. Only best case (min time) timing analysis is required between

first and second stage flip-flops to ensure that all hold times are met. Also, gate-level simulations can more easily be configured to ignore setup and hold time violations on the first stage of each synchronizer.

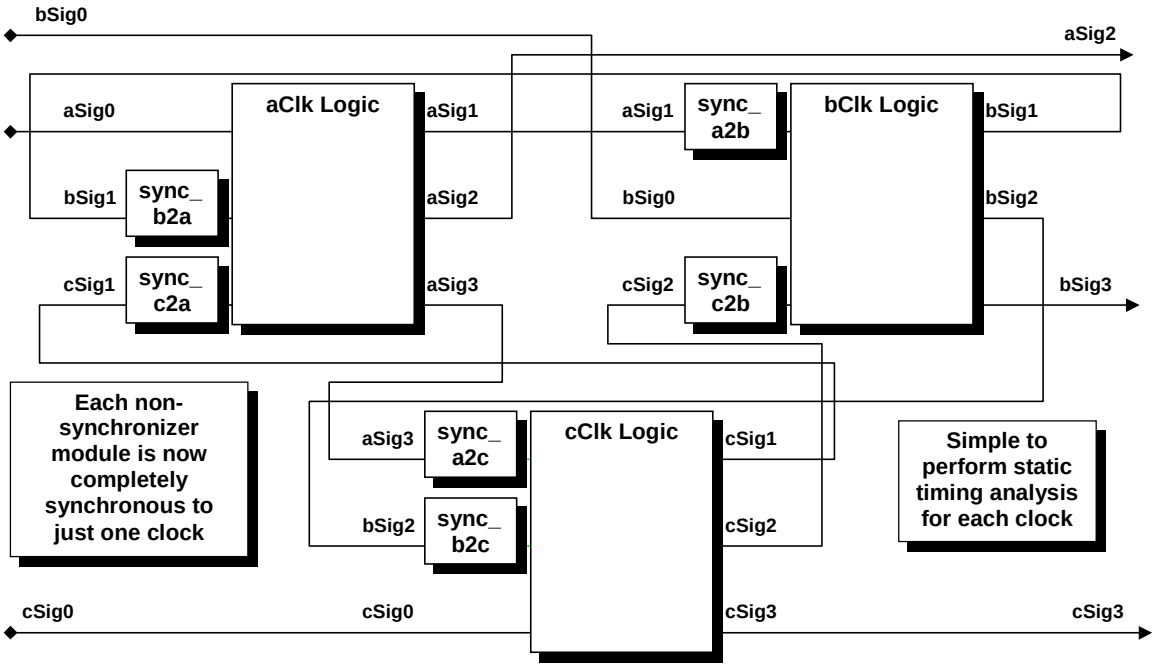


Figure 4 - Design partitioned on clock boundaries

In 1995, while working on a multi-asynchronous-clock ASIC design to be used in In Focus projectors, I received an e-mail message from Steve Golson in which he gave me the strong recommendation to only allow one clock per module for each module in the ASIC design[5]. At that time we were permitting multiple clocks per module and trying to handle timing analysis by including a large number of set_false_path commands in our synthesis scripts to eliminate invalid timing-error messages.

After giving consideration to Steve's recommendation, I decided to completely re-partition the ASIC design I was working on and to adhere to the recommendation to only permit one clock per module. I took a two-week hit to my schedule to re-partition the entire ASIC. After repartitioning the design, many of the timing analysis and synthesis tasks became trivial.

By partitioning a design to permit only one clock per module, static timing analysis becomes a significantly easier task.

The next logical step was to partition the design so that every input module signal was already synchronized to the same clock domain before entering the module. Why is this significant? If all signals entering and leaving the module are synchronous to the clock used in the module, the design is now completely synchronous! Now the entire module can be static timing analyzed

without any "false paths" and Design Compiler can be used to "group" all of the same-clock synchronous modules to perform complete, sequential static timing analysis within each clock domain.

There is one exception to the above recommendation. Multi-clock designs require at least some RTL modules to pass signals from one clock domain to modules that are clocked within a different clock domain. For the In Focus ASIC designs, we created separate synchronizer modules that permitted signals from one and only one clock domain to be passed into a module that synchronized the signals into a new clock domain.

Using the naming convention described in section 5.0, all processor-clock generated signals (u-*signals*) would be used as inputs to a module that might be clocked by the video clock. This module was called the "sync_u2v" module and the RTL code did nothing more than take each u-*signal* input and run it through a pair of flip-flops clocked by vClk. Aside from the vClk and reset inputs, every other input signal to the "sync_u2v" module had a "u" prefix and every output signal from that same module had a "v" prefix.

No worst-case timing analysis is required on the "sync" modules because we know that every input signal to these modules will have timing problems; otherwise, we would not have to pass the signals through synchronizers. The only timing analysis that we need to perform within synchronizer modules is min-time (hold time) analysis between the first and second flip-flop stages for each signal.

In general, if there are **n** asynchronous clock domains, the design will require **n(n-1)** synchronizer modules, two for each pair of clock signals (example: using the uClk and vClk signals: the two synchronizer modules required would be sync_u2v and sync_v2u). Only if there are no signals that pass between two specific clock domains will a pair of synchronizer modules not be required.

By the way, what happened to that repartitioned In Focus ASIC design? After modifying all of the RTL files to create either completely synchronous modules or synchronizer modules, the task of generating synthesis scripts became trivial. All of the script files which previously included "set_false_path" commands were either deleted or significantly simplified. All timing problems were easily identified and fixed (because they were all within single-clock domain groupings) and the final synthesis runs completed two weeks earlier than anticipated, putting the project back on schedule and completely justifying the decision to repartition the design.

7.0 Synthesis Scripts & Timing Analysis

Following the guidelines of section 6.0, to only permit one clock per module, to require that all signals entering non-synchronizer modules are also in the same clock domain that is used to clock that module and to require that synchronizer modules only permit input signals from one other clock domain, helps to simplify the timing analysis and synthesis scripting tasks associated with a multi-clock design.

Synthesis script commands used to address multiple clock domain issues now become a matter of grouping, identifying false paths and performing min-max timing analysis.

7.1 Grouping

Group together all non-synchronizer modules that are clocked within each clock domain. One group should be formed for each clock domain in the design. These groups will be timing verified as if each were a separate, completely synchronous design.

7.2 Identifying False Paths

In general, only the inputs to the synchronizer modules require "set_false_path" commands. If a clock-prefix naming scheme is used (see section 5.0), then wild-cards can be used to easily identify all asynchronous inputs. For example, the sync_u2v module should have inputs that all start with the letter "u". The following dc_shell command should be sufficient to eliminate all asynchronous inputs from timing analysis:

```
set_false_path -from { u* }
```

7.3 Performing Min-Max Timing Analysis

Each grouped set of modules for each clock domain is now a completely synchronous sub-design and tools such as DesignTime or PrimeTime can be used to verify worst case timing (including setup time checks) and best case timing (including hold time checks).

The synchronizer blocks are timing verified separately. Worst case timing checks are not required because these modules are just composed of flip-flops to synchronize asynchronous input signals; therefore, there are no long path delays and the outputs are fully registered. After setting false paths on all of the asynchronous inputs, best case (minimum) timing verification is conducted to insure that hold times are met on all signals that are passed from the first to second stage synchronizing flip-flops.

8.0 Synchronizing Fast Signals Into Slow Clock Domains

A general problem associated with synchronizers is the problem that a signal from a sending clock domain might change values twice before it can be sampled into a slower clock domain. This problem must be considered any time signals are sent from one clock domain to another.

Synchronizing slower control signals into a faster clock domain is generally not a problem since the faster clock signal will sample the slower control signal one or more times. Recognizing that sampling slower signals into faster clock domains causes fewer potential problems than sampling faster signals into slower clock domains, a designer might want to take advantage of this fact and try to steer control signals towards faster clock domains.

8.1 Passing A Slow Control Signal

When passing one control signal between clock domains, a simple two-flip-flop synchronizer is typically sufficient if other rules are followed (described below).

An exception to this rule occurs when trying to pass a control signal from a faster clock domain to a slower clock domain, the control signal must be wider than the cycle time of the slower clock. If the control signal is only asserted for one fast-clock cycle, the control signal could go high and low between the rising edges of a slower clock and not be captured into the slower clock domain as shown in Figure 5 .

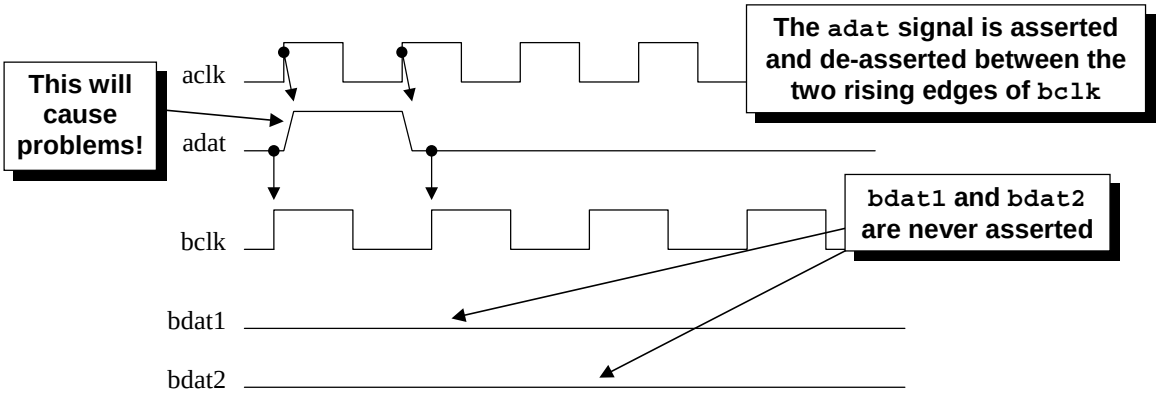


Figure 5 - Short control signal pulse missed during synchronization

One potential solution to this problem is to assert control signals for a period of time that exceeds the cycle time of the sampling clock as shown in Figure 6. The assumption is that the control signal will be sampled at least once and possibly twice by the receiver clock.

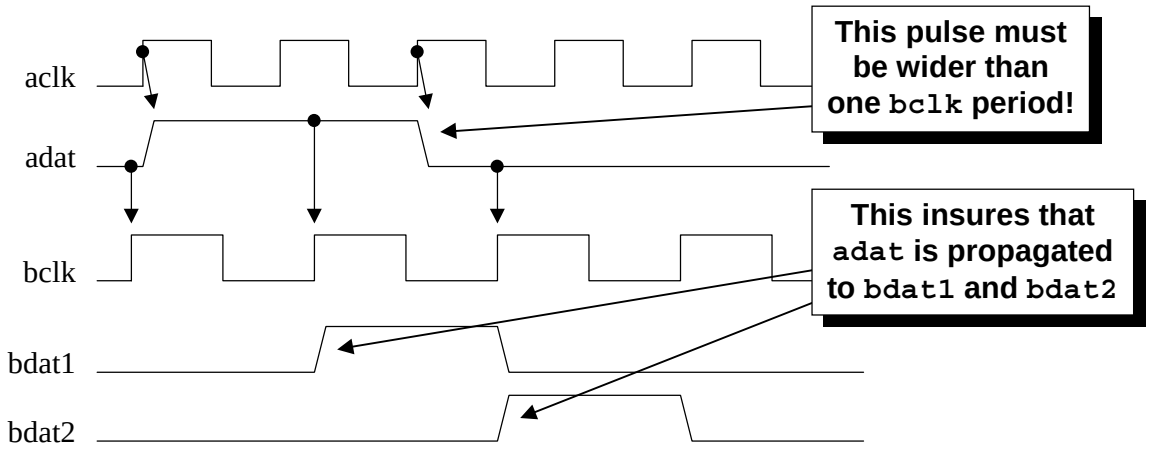


Figure 6 - Lengthened pulse to guarantee that the control signal will be sampled

A second potential solution to this problem is to assert a control signal, synchronize it into the new clock domain and then pass the synchronized signal back through another synchronizer into the sending clock domain as an acknowledge signal. Although synchronizing a feedback signal is a very safe technique to acknowledge that the first control signal was recognized and sampled into the new clock domain, there is considerable delay associated with synchronizing control signals in both directions before releasing the control signal[6].

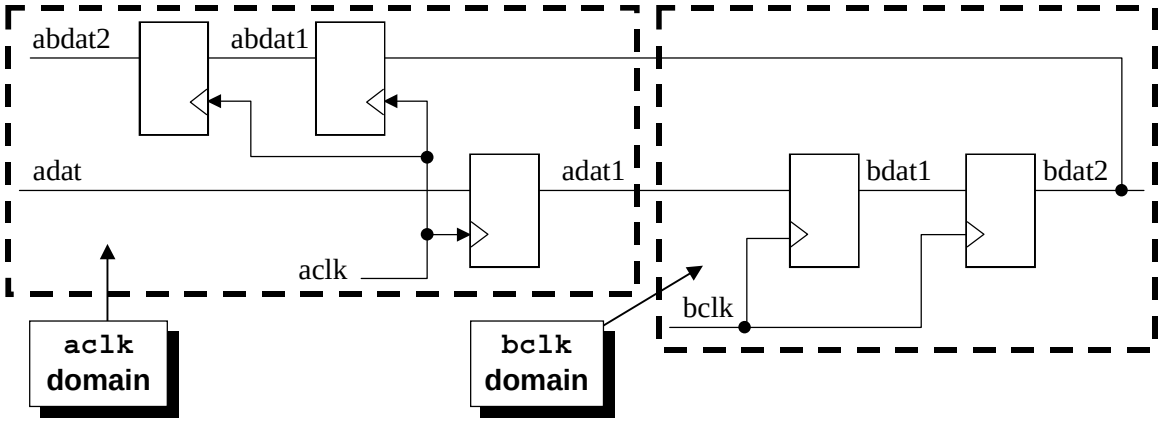


Figure 7 - Feedback synchronization of a control signal

9.0 Passing Multiple Control Signals

A frequent mistake made by engineers when working on multi-clock designs is passing multiple control signals from one clock domain to another and overlooking the importance of the sequencing of the control signals. Simply using synchronizers on all control signals is not always good enough as will be shown in the following examples.

If the order or alignment of the control signals is significant, care must be taken to correctly pass the signals into the new clock domain. All of the examples shown in this section are overly simplistic but they closely mimic situations that often arise in real designs.

9.1 Problem - Two simultaneously required control signals.

In the simple example shown in Figure 8, a register in the new clock domain requires both a load signal and an enable signal in order to load a data value into the register. If both the load and enable signals are being sent from one clock domain, there is a chance that a small skew between the control signals could cause the two signals to be synchronized into different clock cycles within the new clock domain. In this example, this would cause the data to the register to not be loaded.

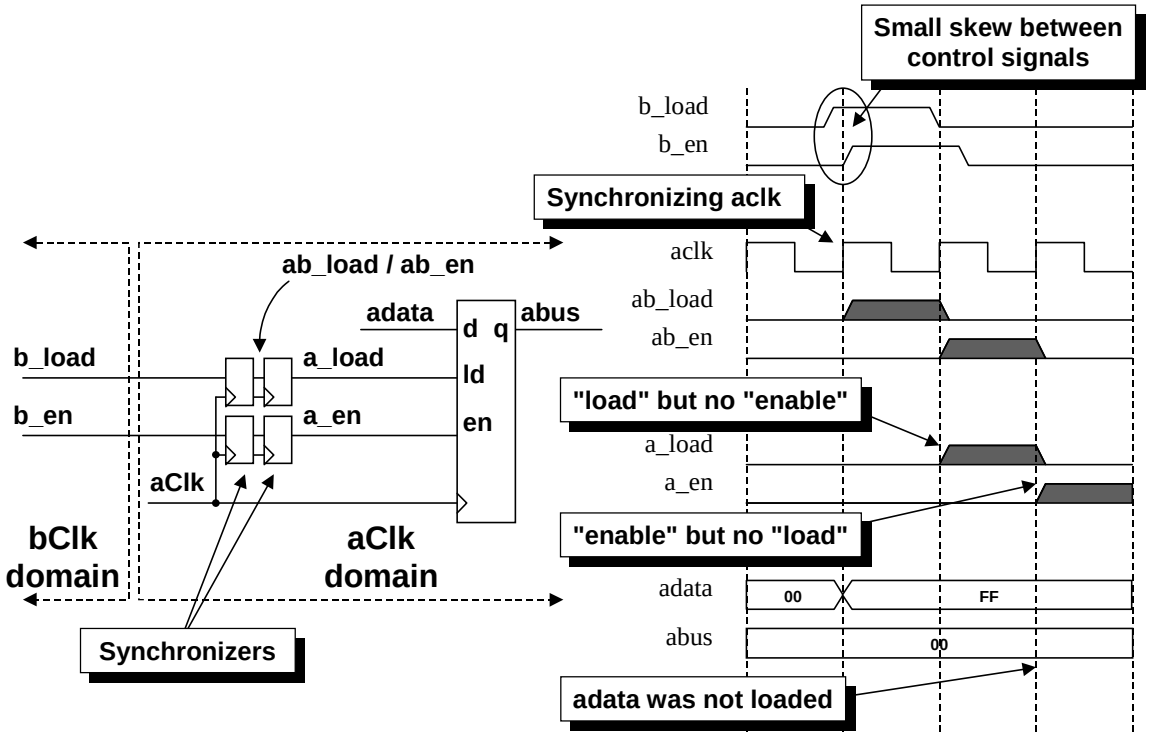


Figure 8 - Problem - Passing multiple control signals between clock domains

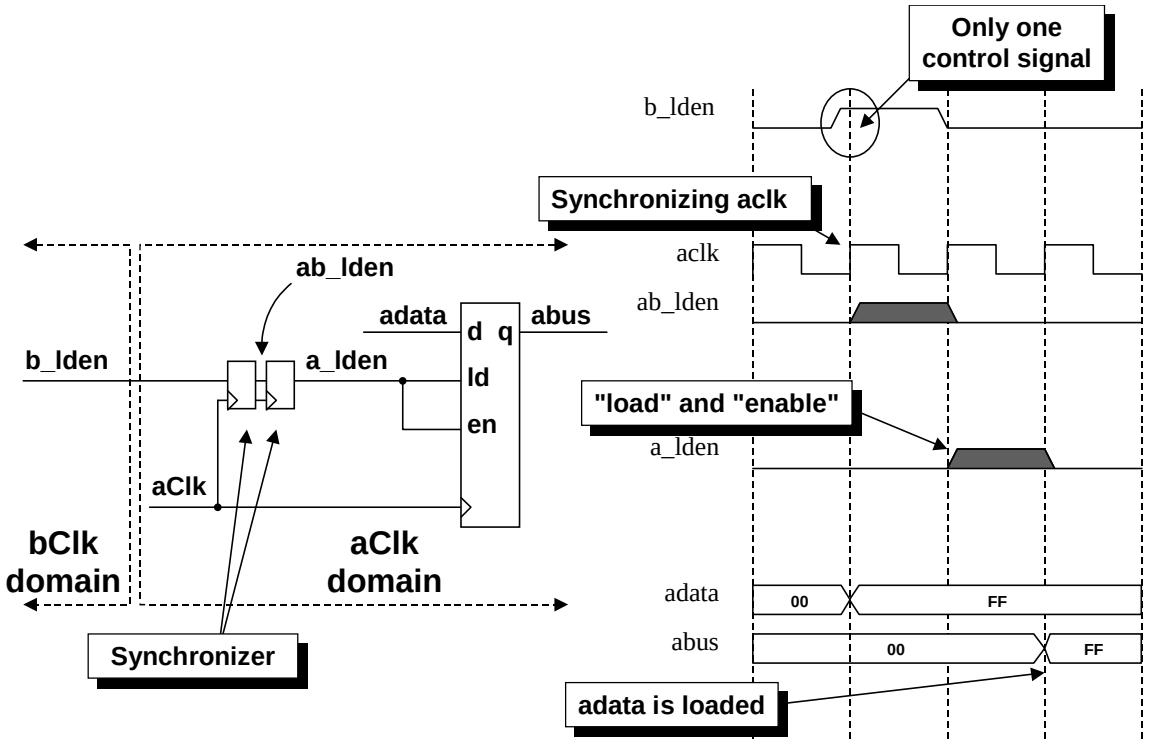


Figure 9 - Solution - Consolidating control signals before passing them between clock domains

The solution to the problem in this simple example is easy. As shown in Figure 9, drive both the load and enable register input signals in the new clock domain from just one control signal. This will remove the potential for the control signals arriving shifted in time.

9.2 Problem - Two phase-shifted sequencing control signals.

The diagram in Figure 10, shows two enable signals, aen1 and aen2, that are used to enable the sequential passing of a data signal through a short pipeline design. The problem is that in the first clock domain, the aen1 control signal might terminate slightly before the aen2 control signal is asserted, and the second clock domain might try to sample the aen1 and aen2 control signals in the middle of this slight time gap, causing a one-cycle gap to form in the enable control-signal chain in the second clock domain. This would cause the a2 output signal to be missed by the second flip-flop.

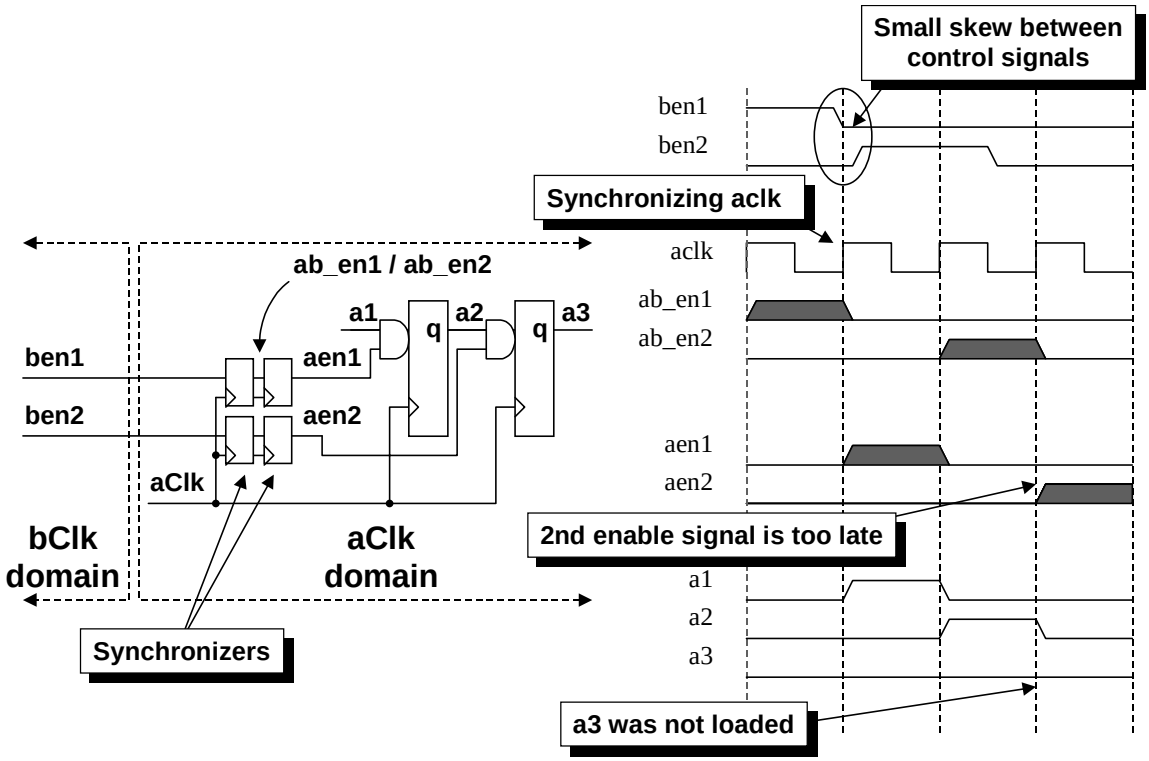


Figure 10 - Problem - Passing sequential control signals between clock domains

The solution to the problem, as shown in Figure 11, is to send only one control signal into the new clock domain and generate the second phase-shifted sequential control signal within the new clock domain.

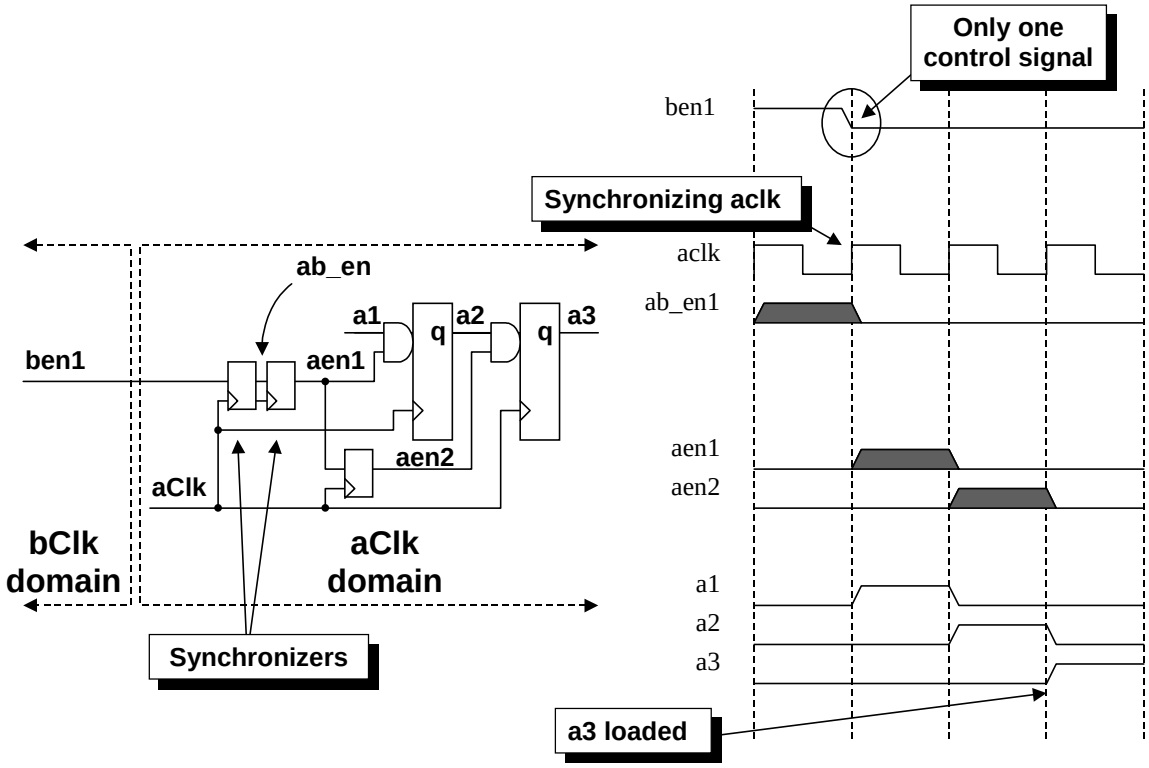


Figure 11 - Solution - Logic to generate the proper sequencing signals in the new clock domains

9.3 Problem - Two encoded control signals.

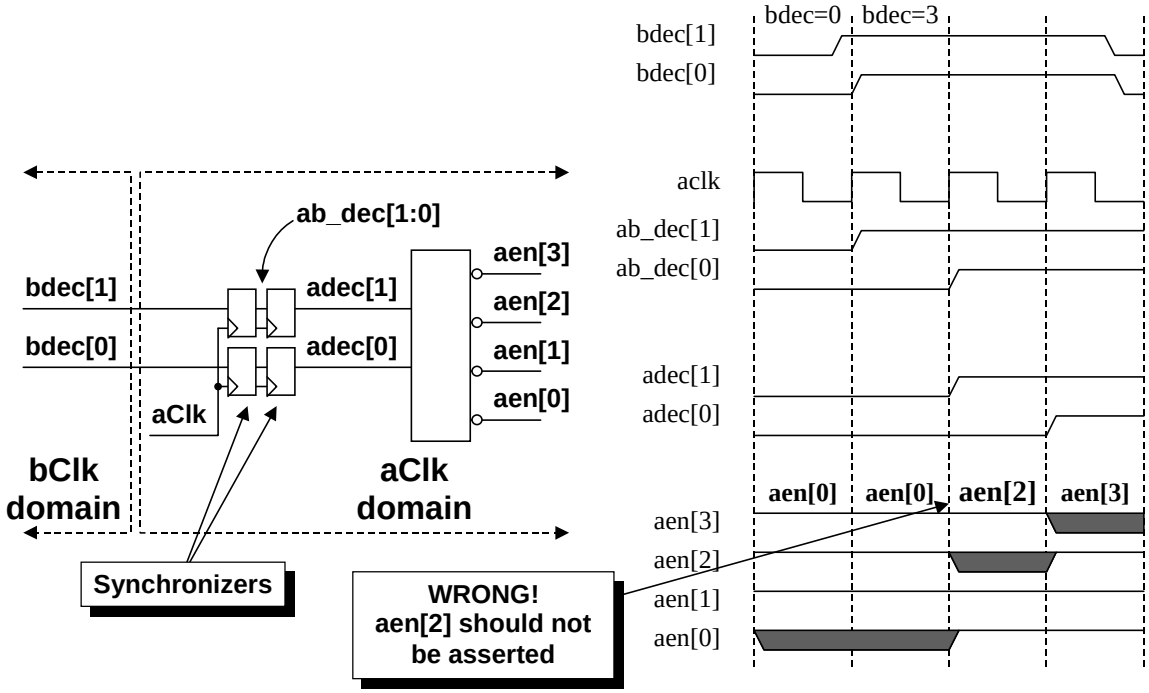


Figure 12 - Problem - Encoded control signals passed between clock domains

The diagram in Figure 12 shows two encoded control signals being passed between clock domains. If the two encoded signals are slightly skewed when sampled, an erroneous decoded output could be generated for one clock period in the new clock domain.

One potential solution to this problem, as shown in Figure 13, is to send a shaped enable signal to act as a "ready flag" in the new clock domain. The sending clock domain must generate and enable signal one clock cycle after asserting the decoder inputs. The sending clock domain must also remove the enable signal one clock cycle before de-asserting the decoder inputs. As described earlier, the enable signal must be asserted for a time period that is longer than the cycle time of the receiving clock domain.

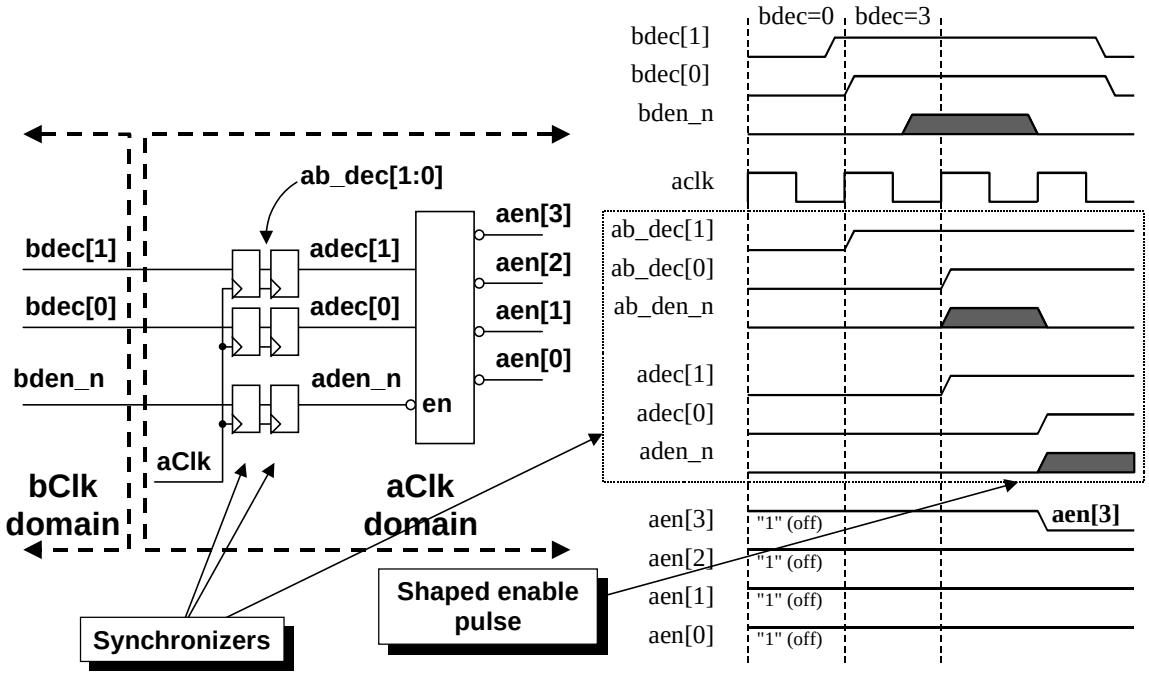


Figure 13 - Solution #1 - Logic to synchronize and wave-shape an enable pulse to pass between clock domains

Under worst case conditions, the shaped enable signal will either be sampled at the same time as the encoded inputs are sampled into the receiving clock domain, or the shaped enable signal will be de-asserted at the same time as the encoded inputs are de-asserted in the receiving clock domain. Under best case conditions, the shaped enable pulse will be asserted one receiving clock cycle later than the assertion of the encoded inputs and de-asserted one receiving clock cycle before the de-assertion of the encoded inputs. This method insures that the encoded inputs are valid before they are enabled into the receiving clock domain.

A second potential solution to this problem, as shown in Figure 14, is to decode the signals back in the sending clock domain and then send the decoded outputs (where only one of the outputs is asserted) through synchronizers into the new clock domain. Within the new clock domain, a state machine is used to determine when a new decoded output has been asserted. If there are no decoded outputs, it means that one decoded output has been de-asserted and that another decoded

output is about to be asserted. If there are two asserted decoded output signals, the last decoded output signal will cause the state machine to change states and the older decoded output signal will turn off on the next rising clock edge in the new clock domain. It is important that the sender insure that the decoded outputs are each asserted for a time period that is longer than the cycle time of the receiving clock domain.

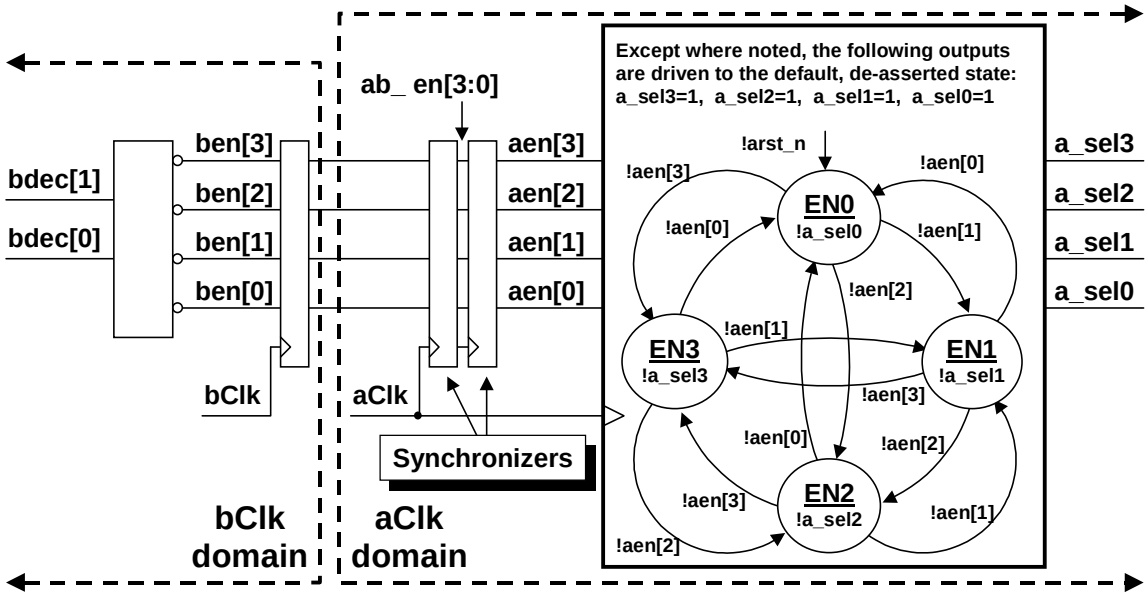


Figure 14 - Solution #2 - FSM logic to detect one-hot control signals passed from a different clock domain

Any time there are multiple control signals crossing clock boundaries, caution must be taken to insure that the sequencing of the control signals being passed is correct or that any potential mis-sequencing of the control signals will not adversely impact the correct operation of the design.

10.0 Data-Path Synchronization

Passing data from one clock domain to another is an example of passing multiple randomly changing signals between clock domains. Using synchronizers to handle the passing of data is generally unacceptable. There are far too many opportunities for multi-bit data changes to be incorrectly sampled using synchronizers.

Two common methods for synchronizing data between clock domains are: (1) use handshake signals to pass data between clock domains or, (2) use FIFOs (First In First Out memories) to store data using one clock domain and to retrieve data using another clock domain.

10.1 Handshaking Data Between Clock Domains

Data can be passed between clock domains using two or three handshake control signals, depending on the application and the paranoia of the design engineer. When it comes to handshaking, the more control signals that are used, the longer the latency to pass data from one

clock domain to another. The biggest disadvantage to using handshaking is the latency required to pass and recognize all of the handshaking signals for each data word that is transferred.

For many open-ended data-passing applications, a simple two-line handshaking sequence is sufficient. The sender places data onto a data bus and then synchronizes a "data_valid" signal to the receiving clock domain. When the "data_valid" signal is recognized in the new clock domain, the receiver clocks the data into a register in the new clock domain (the data should have been stable for at least two rising clock edges in the sending clock domain) and then passes an "acknowledge" signal through a synchronizer to the sender. When the sender recognizes the synchronized "acknowledge" signal, the sender can change the value being driven onto the data bus.

Under some circumstances, it might be useful to use a third control signal, "ready", sent through a synchronizer from the receiver to the sender to indicate that the receiver is indeed "ready" to receive data. The "ready" signal should not be asserted while the "data_valid" signal is true. When the "data_valid" signal is de-asserted, a "ready" signal can be passed to the sender. Of course, with the added handshake signal comes the penalty of longer latency to synchronize and recognize the third control signal.

10.2 Passing Data By FIFO Between Clock Domains

One of the most popular methods of passing data between clock domains is to use a FIFO. A dual port memory is used for the FIFO storage. One port is controlled by the sender which puts data into the memory as fast a one data word (or one data bit for serial applications) per write clock. The other port is controlled by the receiver, which pulls data out of memory one data word per read clock. Two control signals are used to indicate if the FIFO is empty, full or partially full. Two additional control signals are frequently used to indicate if the FIFO is almost full or almost empty.

In theory, placing data into a shared memory with one clock and removing the data from the shared memory with another clock seems like an easy and ideal solution to passing data between clock domains. For the most part it is, but generating accurate full and empty flags can be challenging.

10.3 FIFO Full & Empty

Determining that a FIFO is full or empty requires some type of mathematical manipulation and/or comparison of write and read pointers. The problem is that the two pointers are generated in two different clock domains, so one or both pointers must be synchronized into the opposite clock domain before mathematical and comparison operations can be safely performed.

10.4 FIFO Pointers - Implemented as Binary Counters

Any FIFO pointer that must be synchronized into a different clock domain should not be implemented as a binary counter.

One characteristic of binary counters is that half of all sequential binary incrementing operations require that two or more counter bits must change. Trying to synchronize a binary counter into a new clock domain is more problematic than trying to synchronize multiple control signals into a new clock domain. If a simple 4-bit binary counter changes from address 7 (binary 0111) to address 8 (binary 1000), all four counter bits will change at the same time. If a synchronizing clock edge comes in the middle of this transition, it is possible that any 4-bit binary pattern could be sampled and synchronized into the new clock domain as shown in Figure 15.

Binary Count Values
00 0 0 0 0
01 0 0 0 1
02 0 0 1 0
03 0 0 1 1
04 0 1 0 0
05 0 1 0 1
06 0 1 1 0
07 0 1 1 1
08 1 0 0 0
09 1 0 0 1
10 1 0 1 0
11 1 0 1 1
12 1 1 0 0
13 1 1 0 1
14 1 1 1 0
15 1 1 1 1

07 -> 08 possible binary transitions		
0 1 1 1	-> 1 0 0 0	(07->08)
0 1 1 1	-> 0 0 0 0	(07->00)
0 1 1 1	-> 0 0 0 1	(07->01)
0 1 1 1	-> 0 0 1 0	(07->02)
0 1 1 1	-> 0 0 1 1	(07->03)
0 1 1 1	-> 0 1 0 0	(07->04)
0 1 1 1	-> 0 1 0 1	(07->05)
0 1 1 1	-> 0 1 1 0	(07->06)
0 1 1 1	-> 0 1 1 1	(07->07)
0 1 1 1	-> 1 0 0 0	(07->08)
0 1 1 1	-> 1 0 0 1	(07->09)
0 1 1 1	-> 1 0 1 0	(07->10)
0 1 1 1	-> 1 0 1 1	(07->11)
0 1 1 1	-> 1 1 0 0	(07->12)
0 1 1 1	-> 1 1 0 1	(07->13)
0 1 1 1	-> 1 1 1 0	(07->14)
0 1 1 1	-> 1 1 1 1	(07->15)

Figure 15 - Binary count values sampled in mid-transition

The new, synchronized binary value might trigger a false full or empty flag, or even worse, it might *not* trigger a *real* full or empty flag causing data to be lost due to FIFO overflow or causing bogus data to be read from the FIFO due to attempting to read data when the FIFO is really empty.

10.5 FIFO Pointers - Implemented as Gray-Code Counters

Although binary counters work fine for addressing the memory, trying to synchronize binary counters into a new clock domain is problematic. A better approach for passing pointers between clock domains is to use a gray-code counter for the two FIFO pointers. Gray code counters only change one bit at a time. If a synchronizing clock signal comes in the middle of a gray code counter transition, the synchronized value will either be the old value or the new value because only one bit is changing at a time.

10.6 Designing Gray Code Counters

A block diagram for a gray-code counter is shown in Figure 16. To design a gray code counter, a register is used to store the gray code values. The register output is fed back to a gray-to-binary converter, the binary value is incremented by one, the incremented binary value is then passed to a binary-to-gray converter that drives the inputs to the gray-code register.

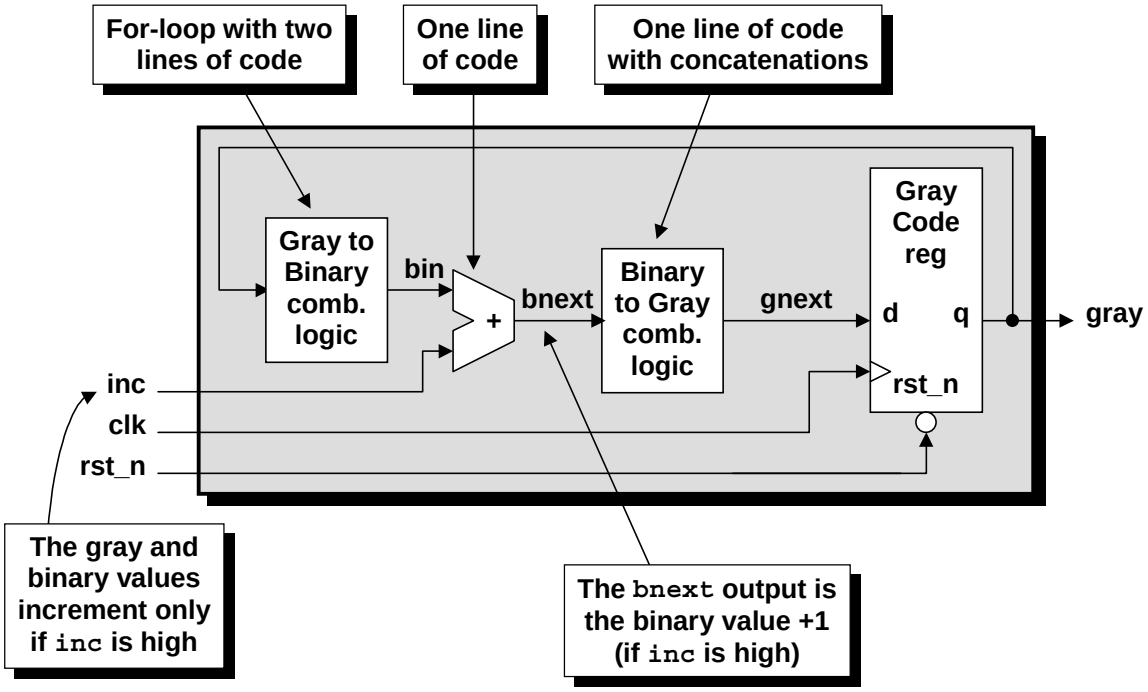


Figure 16 - Gray-code counter block diagram

10.7 Gray To Binary Conversion

To convert a gray-code value to an equivalent binary-code value, using an n-bit gray code value as an example, binary bit 0 is equal to the exclusive-or of gray code bit 0 exclusive-ored with all other gray code bits from 1 to n. Binary bit 1 is equal gray code bit 1 exclusive-ored with all other gray code bits from 2 to n, etc. The most significant binary bit is just equal to the most significant gray code bit. The equations for a 4-bit gray-to-binary conversion are shown in Figure 17.

```
bin[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0];
bin[1] = gray[3] ^ gray[2] ^ gray[1];
bin[2] = gray[3] ^ gray[2];
bin[3] = gray[3];
```

Figure 17 - 4-bit gray-to-binary conversion equations

The easiest way to code a gray-to-binary converter is to code a for-loop and do an exclusive-or reduction on a gray code vector with variable index range, where each time through the loop the

LSB of the index range increases until we are left with a simple assignment of bin[MSB] = ^gray[MSB:MSB] (just the 1-bit MSB of the gray code vector), as shown in Example 1.

```
module gray2bin_bad (bin, gray);
  parameter SIZE = 4;
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;
  reg    [SIZE-1:0] bin;
  integer           i;

  // Syntax Error - variable index range
  always @(gray)
    for (i=0; i<SIZE; i=i+1)
      bin[i] = ^(gray[SIZE-1:i]);
endmodule
```

Example 1 - Non-working but conceptually correct gray-to-binary Verilog model

Unfortunately, Verilog does not permit part selects using a variable index range so the code in Example 1, although conceptually correct, will not compile.

Another way to think of a gray-to-binary conversion is to exclusive-or the significant gray-code bits with padded 0's as shown in Figure 18.

```
bin[0] = gray[3] ^ gray[2] ^ gray[1] ^ gray[0] ; // gray>>0
bin[1] =   1'b0  ^ gray[3] ^ gray[2] ^ gray[1] ; // gray>>1
bin[2] =   1'b0  ^   1'b0  ^ gray[3] ^ gray[2] ; // gray>>2
bin[3] =   1'b0  ^   1'b0  ^   1'b0  ^ gray[3] ; // gray>>3
```

Figure 18 - 4-bit gray-to-binary conversion equations - 2nd method

The corresponding parameterized Verilog model for this algorithm is shown in Example 2. This example is syntactically correct, will compile and does work.

```
module gray2bin (bin, gray);
  parameter SIZE = 4;
  output [SIZE-1:0] bin;
  input  [SIZE-1:0] gray;
  reg    [SIZE-1:0] bin;
  integer           i;

  always @(gray)
    for (i=0; i<SIZE; i=i+1)
      bin[i] = ^(gray>>i);
endmodule
```

Example 2 - Parameterized and correct gray-to-binary Verilog model

10.8 Binary To Gray Conversion

To convert a binary value to an equivalent gray-code value, using an n-bit binary value as an example, gray-code bit 0 is equal to the exclusive-or of binary bits 0 and 1. Gray-code bit 1 is

equal to the exclusive-or of binary bits 1 and 2, etc. The most significant gray-code bit is just equal to the most significant binary bit. The equations for a 4-bit binary-to-gray conversion are shown in Figure 19.

```
bin[0] = gray[0] ^ gray[1];
bin[1] = gray[1] ^ gray[2];
bin[2] = gray[2] ^ gray[3];
bin[3] = gray[3];
```

Figure 19 - 4-bit binary-to-gray conversion equations

The easiest way to code a binary-to-gray converter is to code a simple continuous assignment that performs a bit-wise exclusive-or operation between the binary vector and a right-shifted version of the same binary vector as shown in Example 3. This example is syntactically correct, will compile and does work.

```
module bin2gray (gray, bin);
  parameter SIZE = 4;
  output [SIZE-1:0] gray;
  input  [SIZE-1:0] bin;

  assign gray = (bin>>1) ^ bin;
endmodule
```

Example 3 - Parameterized binary-to-gray Verilog model

10.9 Gray Code Counter

The Verilog code for a gray-code counter incorporates a gray-to-binary converter, a binary-to-gray converter and increments the binary value between conversions. The parameterized Verilog model for the gray-code counter is shown in Example 4.

```
module graycntr (gray, clk, inc, rst_n);
  parameter SIZE = 4;
  output [SIZE-1:0] gray;
  input             clk, inc, rst_n;
  reg    [SIZE-1:0] gnext, gray, bnext, bin;
  integer           i;

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

  always @(gray or inc) begin
    for (i=0; i<SIZE; i=i+1)
      bin[i] = ^(gray>>i);
    bnext = bin + inc;
    gnext = (bnext>>1) ^ bnext;
  end
endmodule
```

Example 4 - Parameterized gray-code counter Verilog model

11.0 FIFO Design

When passing data between two different clock domains, FIFOs, or First-In, First-Out memories, are the design-block of choice for most engineers. Figure 20 shows a block diagram for a FIFO design.

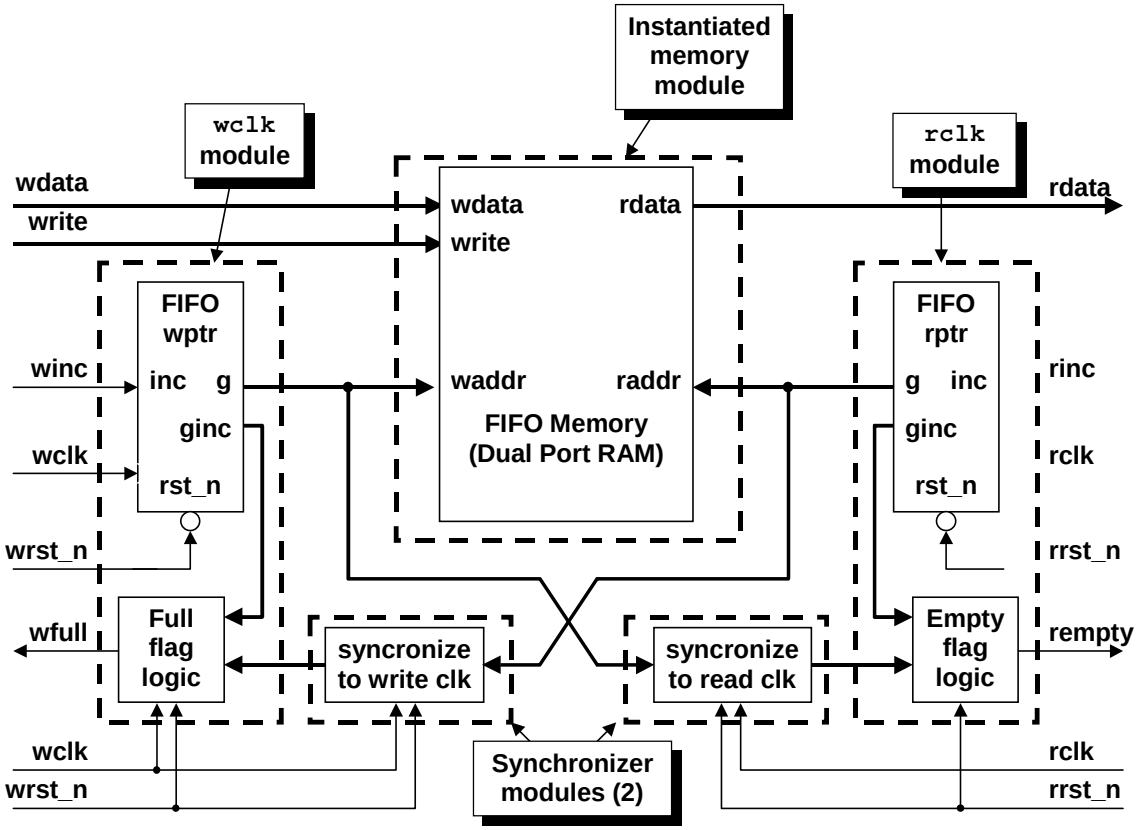


Figure 20 - FIFO Block Diagram - partitioned on clock boundaries

11.1 FIFO Write and Read Operations

For the purposes of this paper, a FIFO write operation is an operation that loads a data word into the FIFO. FIFO write operations are sometimes called FIFO fill, FIFO load, etc.

For the purposes of this paper, a FIFO read operation is an operation that removes a data word from the FIFO. FIFO read operations are sometimes called FIFO drain, etc.

Since full and empty flags are generated by pointers where at least one of the pointers must be synchronized into a second clock domain, clock-cycle accurate assertion and de-assertion of full and empty flags is not completely possible.

One FIFO design technique is to insure that a full or empty flag is asserted exactly when full or empty conditions occur, but de-asserting the flags might come a few clock cycles late. This is sometimes referred to as pessimistic full and empty flags.

11.2 Pessimistic full and empty flags

A pessimistic full flag is a full signal that is asserted immediately when a FIFO becomes full but is de-asserted late (it is not de-asserted until a few read-clock cycles later).

Because the write pointer does not have to be synchronized before testing for a full condition, the full flag will be asserted immediately when the FIFO goes full. The FIFO might not actually be completely full because the read pointer might have incremented but the new read pointer value might not have been synchronized into the write clock domain. Using the block diagram shown in Figure 20, the read pointer synchronized into the write clock domain is always two write clocks behind the actual read pointer value, so the full flag might be asserted for two extra write clocks. This typically is not a problem since the full flag is simply holding off transmission of more data from the data sending source for two extra write clock cycles. Pointers being synchronized into a new clock domain should be gray code counters for reasons explained in sections 10.4 and section 10.5.

Similarly, because the read pointer does not have to be synchronized before testing for an empty condition, the empty flag will be asserted immediately when the FIFO goes empty. The FIFO might not actually be completely empty because the write pointer might have incremented but the new write pointer value might not have been synchronized into the read clock domain. Using the block diagram shown in Figure 20, the write pointer synchronized into the read clock domain is always two read clocks behind the actual write pointer value, so the empty flag might be asserted for two extra read clocks. This typically is not a problem since the empty flag is merely informing the data receiver that data is not ready to be sent for another two read clock cycles. Again, pointers being synchronized into a new clock domain should be gray code counters for reasons explained in sections 10.4 and section 10.5.

11.3 Full & Empty

A FIFO is full when both pointers are equal. A FIFO is also empty when both pointers are equal, so the FIFO pointers should be one bit larger than is necessary to address the full memory range. The extra bit is used as a flag to help determine if the FIFO is empty or full. If the extra, pointer MSBs are equal, it means that the FIFO pointers have wrapped back to address 0 an equal number of times and if the rest of the FIFO bits are equal, the FIFO is empty. If the extra, pointer MSBs are not equal, it means that the write pointer has wrapped back to address 0 one more time than the read pointer and if the rest of the FIFO bits are equal, the FIFO is full.

12.0 Simulation Issues

As mentioned in section 4.0, signals crossing clock boundaries through a synchronizer will experience setup and hold violations. That is why synchronizers are added to a design, to filter out the metastability effects of a signal that changes too close to the rising edge of a new clock domain clock signal.

When doing gate-level simulations on a multi-clock design, the ASIC library models of flip-flops are modeled with setup and hold time expressions to match the timing specifications of the actual flip-flops. ASIC libraries typically model flip-flops to drive X's (unknowns) on the flip-flop outputs when a timing violation occurs. When simulating gate-level synchronizers, setup and hold time violations might cause ASIC libraries to issue setup and hold time error messages and the offending signals are frequently driven to an X value. These X-values propagate to the rest of the design causing problems when trying to verify the functionality of the entire gate-level design.

Most Verilog simulators have a command option to ignore all timing checks, but this would also ignore the desired timing checks for the rest of the design.

It is possible to change the setup and hold time setting to zero for any ASIC library flip-flop that is used in a synchronizer, but that would cause all setup and hold time checks of all instances of that same type of flip-flop to be set to zero, including the flip-flops that you might want to use to test the rest of the design.

You could make copies of flip-flops from an ASIC library and store them into a new Verilog library with different names, set to zero all setup and hold times, then modify the design gate-level netlist, replacing all first stage synchronizer ASIC library flip-flops with the modified library flip-flops without timing checks, but this could be an error prone and tedious process that might have to be repeated each time a new netlist is generated or it might require the creation of a makefile and scripts to automatically make the modifications each time a new netlist is generated.

A clever way to approach this problem suggested by Bhatnagar[7] is to use Synopsys commands to modify the SDF backannotation of the setup and hold time on just the first stage flip-flop cells in the design. Bhatnagar points out that the SDF file is instance based and therefore targeting the setup and hold times for the offending cells is more easily accomplished. Bhatnagar notes:

> Instead of manually removing the setup and hold-time constructs from the SDF file, a better way is to zero out the setup and hold-times in the SDF file, only for the violating flops, i.e., replace the existing setup and hold-time numbers with zero's.

Bhatnagar further points out that setup hold times of zero means that there can be no timing violation, therefore no unknowns propagated to the rest of the design. The following dc_shell command, given by Bhatnagar, is used to make setup and hold times zero:

```
set_annotated_check 0 -setup -hold -from REG1/CLK -to REG1/D
```

Using a creative naming convention for the output of the first stage flip-flop of a synchronizer might make wild card expressions possible to easily backannotate all first stage flip-flop SDF setup and hold time values to zero using very few dc_shell commands.

13.0 Conclusions

Completely synchronous one-clock design techniques are well known. Synthesis tools do their best work on synchronous designs. Timing analysis tools are designed to report timing problems on one-clock synchronous designs. Synthesis scripts are easy to create for one-clock synchronous clock designs. The techniques in this paper are aimed at making the design look like multiple single clock designs!

- **Partitioning non-synchronizer blocks** so that there is only one clock per module permits easy verification of correct timing by creating clock-domain sub-blocks that can be more easily verified with static timing analysis tools.

- **Partitioning synchronizer blocks** to permit inputs from one and only one clock domain and clocking the signals with only one asynchronous clock creates manageable synchronizer sub-blocks that can also be easily timed.

- **A clock-oriented naming convention** can be useful to help identify signals that need to be timed within the different asynchronous clock domains.

- **Multiple control signals crossing clock domains** require special attention to ensure that all control signals are properly sequenced into a new clock domain.

The techniques described in this paper were developed to facilitate robust development and verification of multi-clock designs.

References

[1] William J. Dally and John W. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998, pg. 468.

[2] William J. Dally and John W. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998, pp. 462-513.

[3] William J. Dally and John W. Poulton, *Digital Systems Engineering*, Cambridge University Press, 1998, pp. 469-470.

[4] Samir Palnitkar, *Verilog HDL, A Guide to Digital Design and Synthesis*, Sunsoft Press A Prentice Hall Title, 1996, pg. 193.

[5] Steve Golson, personal communication.

[6] ESNUG #281 - http://www.deepchip.com/posts/0281.html

[7] Himanshu Bhatnagar, Advanced ASIC Chip Synthesis, Kluwer Academic Publishers, 1999, pp. 202-203.

Author & Contact Information

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

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

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

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

(Data accurate as of March 7th, 2001)

Synchronous Resets? Asynchronous Resets?
I am so confused!
How will I ever know which to use?

Clifford E. Cummings
Sunburst Design, Inc.

Don Mills
LCDM Engineering

ABSTRACT

This paper will investigate the pros and cons of synchronous and asynchronous resets. It will then look at usage of each type of reset followed by recommendations for proper usage of each type.

This paper will also detail an interesting synchronization technique using digital calibration to synchronize reset removal on a multi-ASIC design.

1.0 resets, Resets, RESETS, and then there's RESETS

One cannot begin to consider a discussion of reset usage and styles without first saluting the most common reset usage of all. This undesired reset occurs almost daily in systems that have been tested, verified, manufactured, and integrated into the consumer, education, government, and military environments. This reset follows what is often called "The Blue Screen of Death" resulting from software incompatibilities between the OS from a certain software company, the software programs the OS is servicing, and the hardware on which the OS software is executing.

Why be concerned with these annoying little resets anyway? Why devote a whole paper to such a trivial subject? Anyone who has used a PC with a certain OS loaded knows that the hardware reset comes in quite handy. It will put the computer back to a known working state (at least temporarily) by applying a system reset to each of the chips in the system that have or require a reset.

For individual ASICs, the primary purpose of a reset is to force the ASIC design (either behavioral, RTL, or structural) into a known state for simulation. Once the ASIC is built, the need for the ASIC to have reset applied is determined by the system, the application of the ASIC, and the design of the ASIC. For instance, many data path communication ASICs are designed to synchronize to an input data stream, process the data, and then output it. If sync is ever lost, the ASIC goes through a routine to re-acquire sync. If this type of ASIC is designed correctly, such that all unused states point to the "start acquiring sync" state, it can function properly in a system without ever being reset. A system reset would be required on power up for such an ASIC if the state machines in the ASIC took advantage of "don't care" logic reduction during the synthesis phase.

It is the opinion of the authors that in general, every flip-flop in an ASIC should be resetable whether or not it is required by the system. Further more, the authors prefer to use asynchronous resets following the guidelines detailed in this paper. There are exceptions to these guidelines. In some cases, when follower flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs. This type of environment requires a number of clocks during the reset active period to put the ASIC into a known state.

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset, how will the reset tree be laid out and buffered, how to verify timing of the reset tree, how to functionally test the reset with test scan vectors, and how to apply the reset among multiple clock zones.

In addition, when applying resets between multiple ASICs that require a specific reset release sequence, special techniques must be employed to adjust to variances of chip and board manufacturing. The final sections of this paper will address this latter issue.

2.0 General flip-flop coding style notes

2.1 Synchronous reset flip-flops with non reset follower flip-flops

Each Verilog procedural block or VHDL process should model only one type of flip-flop. In other words, a designer should not mix resetable flip-flops with follower flip-flops (flops with no resets)[12]. Follower flip-flops are flip-flops that are simple data shift registers.

In the Verilog code of Example 1a and the VHDL code of Example 1b, a flip-flop is used to capture data and then its output is passed through a follower flip-flop. The first stage of this design is reset with a synchronous reset. The second stage is a follower flip-flop and is not reset, but because the two flip-flops were inferred in the same procedural block/process, the reset signal **`rst_n`** will be used as a data enable for the second flop. This coding style will generate extraneous logic as shown in Figure 1.

```
module badFFstyle (q2, d, clk, rst_n);
  output q2;
  input  d, clk, rst_n;
  reg    q2, q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

Example 1a - Bad Verilog coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity badFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end badFFstyle;

architecture rtl of badFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q1 <= '0';
      else
        q1 <= d;
        q2 <= q1;
      end if;
    end if;
  end process;
end rtl;
```

Example 1b - Bad VHDL coding style to model dissimilar flip-flops

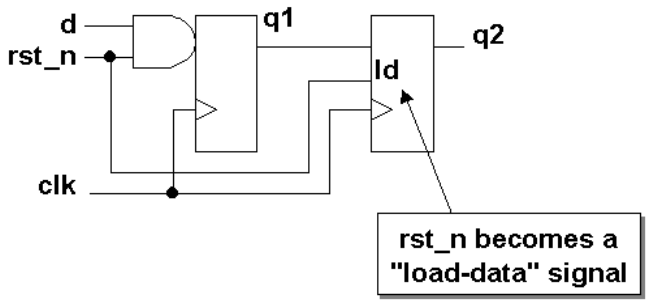


Figure 1 - Bad coding style yields a design with an unnecessary loadable flip-flop

The correct way to model a follower flip-flop is with two Verilog procedural blocks as shown in Example 2a or two VHDL processes as shown in Example 2b. These coding styles will generate the logic shown in Figure 2.

```
module goodFFstyle (q2, d, clk, rst_n);
  output q2;
  input  d, clk, rst_n;
  reg    q2, q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else        q1 <= d;

  always @(posedge clk)
    q2 <= q1;
endmodule
```

Example 2a - Good Verilog coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity goodFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end goodFFstyle;

architecture rtl of goodFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q1 <= '0';
      else
        q1 <= d;
      end if;
    end if;
  end process;

  process (clk)
  begin
    if (clk'event and clk = '1') then
        q2 <= q1;
    end if;
  end process;
end rtl;
```

Example 2b - Good VHDL coding style to model dissimilar flip-flops

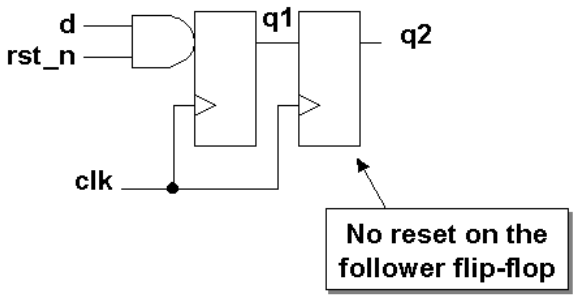


Figure 2 - Two different types of flip-flops, one with synchronous reset and one without

It should be noted that the extraneous logic generated by the code in Example 1a and Example 1b is only a result of using a synchronous reset. If an asynchronous reset approach had be used, then both coding styles would synthesize to the same design without any extra combinational logic. The generation of different flip-flop styles is largely a function of the sensitivity lists and **`if-else`** statements that are used in the HDL code. More details about the sensitivity list and **`if-else`** coding styles are detailed in section 3.1.

2.2 Flip-flop inference style

Each inferred flip-flop should **not** be independently modeled in its own procedural block/process. As a matter of style, all inferred flip-flops of a given function or even groups of functions should be described using a single procedural block/process. Multiple procedural blocks/processes should be used to model macro level functional divisions within a given module/architecture. The exception to this guideline is that of follower flip-flops as discussed in the previous section (section 2.1) where multiple procedural blocks/processes are required to efficiently model the function itself.

2.3 Assignment operator guideline

In Verilog, all assignments made inside the always block modeling an inferred flip-flop (sequential logic) should be made with nonblocking assignment operators[3]. Likewise, for VHDL, inferred flip-flops should be made using signal assignments.

3.0 Synchronous resets

As research was conducted for this paper, a collection of ESNUG and SOLV-IT articles was gathered and reviewed. Around 80+% of the gathered articles focused on synchronous reset issues. Many SNUG papers have been presented in which the presenter would claim something like, “we all know that the best way to do resets in an ASIC is to strictly use synchronous resets”, or maybe, “asynchronous resets are bad and should be avoided.” Yet, little evidence was offered to justify these statements. There are some advantages to using synchronous resets, but there are also disadvantages. The same is true for asynchronous resets. The designer must use the approach that is appropriate for the design.

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. The reset can be applied to the flip-flop as part of the combinational logic generating the d-input to the flip-flop. If this is the case, the coding style to model the reset should be an **`if`**/**`else`** priority style with the reset in the **`if`** condition and all other combinational logic in the **`else`** section. If this style is not strictly observed, two possible problems can occur. First, in some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a hardware issue, but remember, one of the prime objectives of a reset is to put the ASIC into a known state for simulation. Second, the reset could be a “late arriving signal” relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic. This style of synchronous reset can be used with any logic or library. Example 3 shows an implementation of this style of synchronous reset as part of a loadable counter with carry out.

```
module ctr8sr ( q, co, d, ld, rst_n, clk);
  output [7:0] q;
  output       co;
  input  [7:0] d;
  input        ld, rst_n, clk;
  reg    [7:0] q;
  reg          co;

  always @(posedge clk)
    if      (!rst_n) {co,q} <= 9'b0;     // sync reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 3a - Verilog code for a loadable counter with synchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8sr is
  port (
    clk        : in  std_logic;
    rst_n      : in  std_logic;
    d          : in  std_logic;
    ld         : in  std_logic;
    q          : out std_logic_vector(7 downto 0);
    co         : out std_logic);
end ctr8sr;

architecture rtl of ctr8sr is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        count <= (others => '0');       -- sync reset
      elsif (ld = '1') then
        count <= '0' & d;               -- sync load
      else
        count <= count + 1;             -- sync increment
      end if;
    end if;
  end process;
end rtl;
```

Example 3b - VHDL code for a loadable counter with synchronous reset

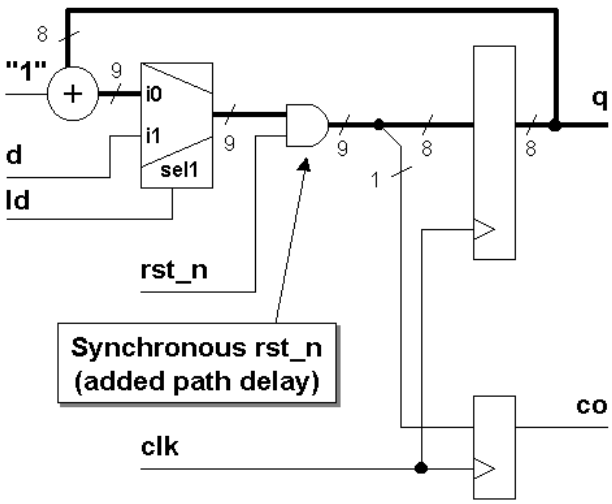


Figure 3 - Loadable counter with synchronous reset

A second style of synchronous resets is based on the availability of flip-flops with synchronous reset pins and the ability of the designer and synthesis tool to make use of those pins. This is sometimes the case, but more often the first style discussed above is the implementation used[22][26].

3.1 Coding style and example circuit

The Verilog code of Example 4a and the VHDL code of 4b show the correct way to model synchronous reset flip-flops. Note that the reset is not part of the sensitivity list. For Verilog omitting the reset from the sensitivity list is what makes the reset synchronous. For VHDL omitting the reset from the sensitivity list and checking for the reset after the "**`if clk'event and clk = 1`**" statement makes the reset synchronous. Also note that the reset is given priority over any other assignment by using the **`if-else`** coding style.

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

  always @(posedge clk)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 4a - Correct way to model a flip-flop with synchronous reset using Verilog

```
library ieee;
use ieee.std_logic_1164.all;
entity syncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end syncresetFFstyle;

architecture rtl of syncresetFFstyle is
begin
  process (clk)
  begin
```

```
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end rtl;
```

Example 4b - Correct way to model a flip-flop with synchronous reset using VHDL

For flip-flops designed with synchronous reset style #1 (reset is gated with data to the d-input), Synopsys has a switch that the designer can use to help infer flip-flops with synchronous resets.

Compiler directive: **`sync_set_reset`**

In general, the authors recommend only using Synopsys switches when they are required and make a difference; however, our colleague Steve Golson pointed out that the **`sync_set_reset`** directive does not affect the functionality of a design, so its omission would not be recognized until gate-level simulation, when discovery of a failure would require re-synthesizing the design late in the project schedule. Since this directive is only required once per module, adding it to each module with synchronous resets is recommended[19].

A few years back, another ESNUG contributor recommended adding the **`compile_preserve_sync_resets = "true"`** compiler directive[13]. Although this directive might have been useful a few years ago, it was discontinued starting with Synopsys version 3.4b[22].

3.2 Advantages of synchronous resets

Synchronous reset will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the d-input. But in such a case, the combinational logic gate count grows, so the overall gate count savings may not be that significant. If a design is tight, the area savings of one or two gates per flip-flop may ensure the ASIC fits into the die. However, in today's technology of huge die sizes, the savings of a gate or two per flip-flop is generally irrelevant and will not be a significant factor of whether a design fits into a die.

Synchronous reset can be much easier to work with when using cycle based simulators. For this very reason, synchronous resets are recommend in section 3.2.4(2nd edition, section 3.2.3 in the 1st edition) of the Reuse Methodology Manual (RMM)[18].

Synchronous resets generally insure that the circuit is 100% synchronous.

Synchronous resets insure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches; however, if these glitches occur near the active clock edge, the flip-flop could go metastable.

In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

By using synchronous resets and a number of clocks as part of the reset process, flip-flops can be used within the reset buffer tree to help the timing of the buffer tree keep within a clock period.

3.3 Disadvantages of synchronous resets

Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock[14].

A designer must work with pessimistic vs. optimistic simulators. This can be an issue if the reset is generated by combinational logic in the ASIC or if the reset must traverse many levels of local combinational logic. During simulation, based on how the reset is generated or how the reset is applied to a functional block, the reset can be masked by X's. A large number of the ESNUG articles addressed this issue. Most simulators will not resolve some X-logic conditions and therefore block out the synchronous reset[5][6][7][8][9][10][11][12][13][20].

By it's very nature, a synchronous reset will require a clock in order to reset the circuit. This may not be a disadvantage to some design styles but to others, it may be an annoyance. The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate a tristate bus when a chip is powered up, the chip must have a power on asynchronous reset[17].

4.0 Asynchronous resets

Asynchronous resets are the authors preferred reset approach. However, asynchronous resets alone can be very dangerous. Many engineers like the idea of being able to apply the reset to their circuit and have the logic go to a known state. The biggest problem with asynchronous resets is the reset release, also called reset removal. The subject will be elaborated in detail in section 5.0.

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. The reset pin is typically active low (the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.)

4.1 Coding style and example circuit

The Verilog code of Example 5a and the VHDL code of Example 5b show the correct way to model asynchronous reset flip-flops. Note that the reset *is* part of the sensitivity list. For Verilog, adding the reset to the sensitivity list is what makes the reset asynchronous. In order for the Verilog simulation model of an asynchronous flip-flop to simulate correctly, the sensitivity list should only be active on the leading edge of the asynchronous reset signal. Hence, in Example 5a, the always procedure block will be entered on the leading edge of the reset, then the **`if`** condition will check for the correct reset level.

Synopsys requires that if any signal in the sensitivity list is edge-sensitive, then all signals in the sensitivity list must be edge-sensitive. In other words, Synopsys forces the correct coding style. Verilog simulation does not have this requirement, but if the sensitivity list were sensitive to more than just the active clock edge and the reset leading edge, the simulation model would be incorrect[4]. Additionally, only the clock and reset signals can be in the sensitivity list. If other signals are included (legal Verilog, illegal Verilog RTL synthesis coding style) the simulation model would not be correct for a flip-flop and Synopsys would report an error while reading the model for synthesis.

For VHDL, including the reset in the sensitivity list and checking for the reset before the "**`if clk'event and clk = 1`**" statement makes the reset asynchronous. Also note that the reset is given priority over any other assignment (including the clock) by using the **`if/else`** coding style. Because of the nature of a VHDL sensitivity list and flip-flop coding style, additional signals can be included in the sensitivity list with no ill effects directly for simulation and synthesis. However, good coding style recommends that only the signals that can directly change the output of the flip-flop should be in the sensitivity list. These signals are the clock and the asynchronous reset. All other signals will slow down simulation and be ignored by synthesis.

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

  // Verilog-2001: permits comma-separation
  // @(posedge clk, negedge rst_n)
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 5a - Correct way to model a flip-flop with asynchronous reset using Verilog

```
library ieee;
```

```
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end asyncresetFFstyle;

architecture rtl of asyncresetFFstyle is
begin
  process (clk, rst_n)
  begin
    if (rst_n = '0') then
      q <= '0';
    elsif (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end rtl;
```

Example 5b - Correct way to model a flip-flop with asynchronous reset using VHDL

The approach to synthesizing asynchronous resets will depend on the designers approach to the reset buffer tree. If the reset is driven directly from an external pin, then usually doing a **`set_drive 0`** on the reset pin and doing a **`set_dont_touch_network`** on the reset net will protect the net from being modified by synthesis. However, there is at least one ESNUG article that indicates this is not always the case[16].

One ESNUG contributor[15] indicates that sometimes **`set_resistance 0`** on the reset net might also be needed.

And our colleague, Steve Golson, has pointed out that you can **`set_resistance 0`** on the net, or create a custom wireload model with resistance=0 and apply it to the reset input port with the command:

```
set_wire_load -port_list reset
```

A recently updated SolvNet article also notes that starting with Synopsys release 2001.08 the definition of ideal nets has slightly changed[24] and that a **`set_ideal_net`** command can be used to create ideal nets and "get no timing updates, get no delay optimization, and get no DRC fixing."

Another colleague, Chris Kiegle, reported that doing a set_disable_timing on a net for pre-v2001.08 designs helped to clean up timing reports[2], which seems to be supported by two other SolvNet articles, one related to synthesis and another related to Physical Synthesis, that recommend usage of both a **`set_false_path`** and a **`set_disable_timing`** command[21][25].

4.2 Modeling Verilog flip-flops with asynchronous reset and asynchronous set

One additional note should be made here with regards to modeling asynchronous resets in Verilog. The simulation model of a flip-flop that includes both an asynchronous set and an asynchronous reset in Verilog might not simulate correctly without a little help from the designer. In general, most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but on the occasion such a flip-flop is required. The coding style of Example 6 can be used to correct the Verilog RTL simulations where both reset and set are asserted simultaneously and reset is removed first.

First note that the problem is only a simulation problem and not a synthesis problem (synthesis infers the correct flip-flop with asynchronous set/reset). The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals. If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state. With both

these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the translate_off/translate_on directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset. The code in Example 6 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations. This code uses the translate_off/translate_on directives to force the correct output for the exception condition[4].

```
// Good DFF with asynchronous set and reset and self-
// correcting set-reset assignment
module dff3_aras (q, d, clk, rst_n, set_n);
  output q;
  input d, clk, rst_n, set_n;
  reg q;

  always @(posedge clk or negedge rst_n or negedge set_n)
    if      (!rst_n) q <= 0; // asynchronous reset
    else if (!set_n) q <= 1; // asynchronous set
    else             q <= d;

  // synopsys translate_off
  always @(rst_n or set_n)
    if (rst_n && !set_n) force   q = 1;
    else                 release q;
  // synopsys translate_on
endmodule
```

Example 6 – Verilog Asynchronous SET/RESET simulation and synthesis model

4.3 Advantages of asynchronous resets

The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, can not afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Of course this argument does not hold if the vendor library has flip-flops with synchronous reset inputs *and* the designer can get Synopsys to actually use those pins. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path. The code in Example 7 infers asynchronous resets that will not be added to the data path.

```
module ctr8ar ( q, co, d, ld, rst_n, clk);
  output [7:0] q;
  output       co;
  input  [7:0] d;
  input        ld, rst_n, clk;
  reg    [7:0] q;
  reg          co;
```

```
  always @(posedge clk or negedge rst_n)
    if      (!rst_n) {co,q} <= 9'b0;     // async reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 7a- Verilog code for a loadable counter with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8ar is
  port (
    clk        : in  std_logic;
    rst_n      : in  std_logic;
    d          : in  std_logic;
    ld         : in  std_logic;
    q          : out std_logic_vector(7 downto 0);
    co         : out std_logic);
end ctr8ar;

architecture rtl of ctr8ar is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk)
  begin
    if (rst_n = '0') then
      count <= (others => '0');       -- sync reset
      elsif (clk'event and clk = '1') then
        if (ld = '1') then
          count <= '0' & d;           -- sync load
        else
          count <= count + 1;         -- sync increment
        end if;
      end if;
  end process;
end rtl;
```

Example 7b- VHDL code for a loadable counter with asynchronous reset

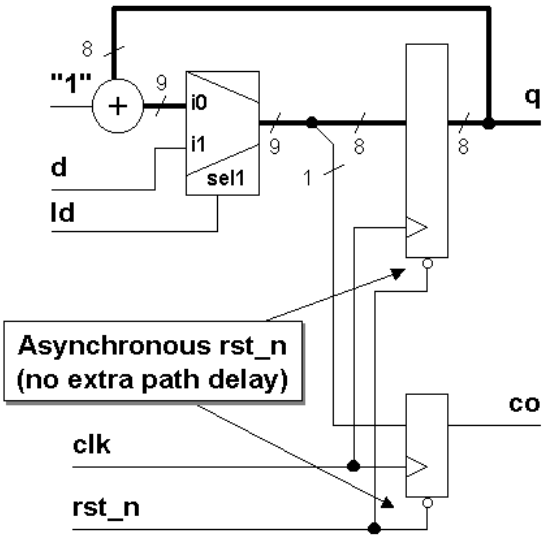


Figure 4 - Loadable counter with asynchronous reset

Another advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present.

The experience of the authors is that by using the coding style for asynchronous resets described in this section, the synthesis interface tends to be automatic. That is, there is generally no need to add any synthesis attributes to get the synthesis tool to map to a flip-flop with an asynchronous reset pin.

4.4 Disadvantages of asynchronous resets

There are many reasons given by engineers as to why asynchronous resets are evil.

The Reuse Methodology Manual (RMM) suggests that asynchronous resets are not to be used because they cannot be used with cycle based simulators. This is simply not true. The basis of a cycle based simulator is that all inputs change on a clock edge. Since timing is not part of cycle based simulation, the asynchronous reset can simply be applied on the inactive clock edge.

For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing. This is required for the synchronizer circuit shown in section 6.

Some designers claim that static timing analysis is very difficult to do with designs using asynchronous resets. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met.

The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the ASIC could be lost.

Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. See section 8.0 for a possible solution to reset glitches. If this is a real problem in a system, then one might think that using synchronous resets is the solution. A different but similar problem exists for synchronous resets if these spurious reset pulses occur near a clock edge, the flip-flops can still go metastable.

5.0 Asynchronous reset problem

In discussing this paper topic with a colleague, the engineer stated first that since all he was working on was FPGAs, they do not have the same reset problems that ASICs have (a misconception). He went on to say that he always had an asynchronous system reset that could override everything, to put the chip into a known state. The engineer was then asked what would happen to the FPGA or ASIC if the release of the reset occurred on or near a clock edge such that the flip-flops went metastable.

Too many engineers just apply an asynchronous reset thinking that there are no problems. They test the reset in the controlled simulation environment and everything works fine, but then in the system, the design fails intermittently. The designers do not consider the idea that the release of the reset in the system (non-controlled environment) could cause the chip to go into a metastable unknown state, thus voiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock

As shown in Figure 5, an asynchronous reset signal will be de-asserted asynchronous to the clock signal. There are two potential problems with this scenario: (1) violation of reset recovery time and, (2) reset removal happening in different clock cycles for different sequential elements.

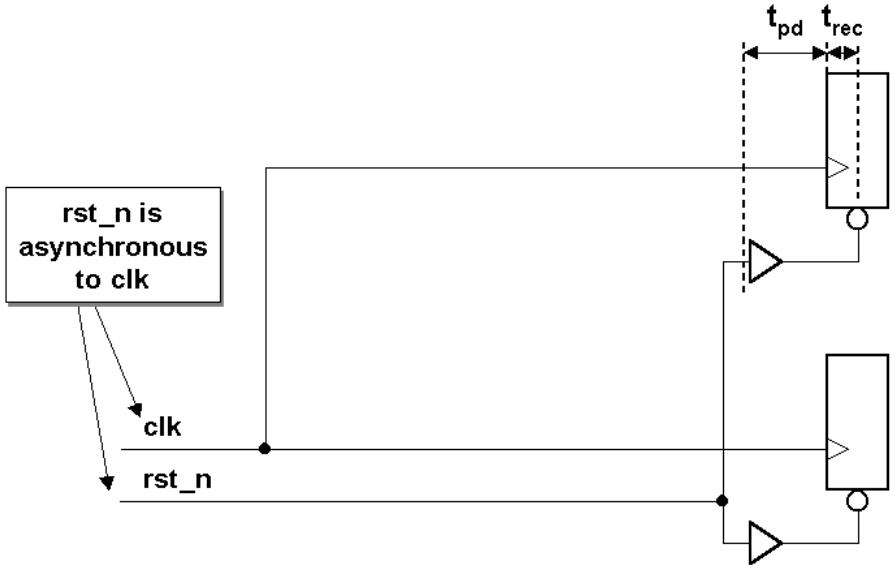


Figure 5 - Asynchronous reset removal recovery time problem

5.1 Reset recovery time

Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. The Verilog-2001 Standard[17] has three built-in commands to model and test recovery time and signal removal timing checks: $recovery, $removal and $recrem (the latter is a combination of recovery and removal timing checks).

Recovery time is also referred to as a **tsu** setup time of the form, "PRE or CLR inactive setup time before CLK↑"[1].

Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.

5.2 Reset removal traversing different clock cycles

When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

6.0 Reset synchronizer

Guideline: EVERY ASIC USING AN ASYNCHRONOUS RESET SHOULD INCLUDE A RESET SYNCHRONIZER CIRCUIT!!

Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

The reset synchronizer logic of Figure 6 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

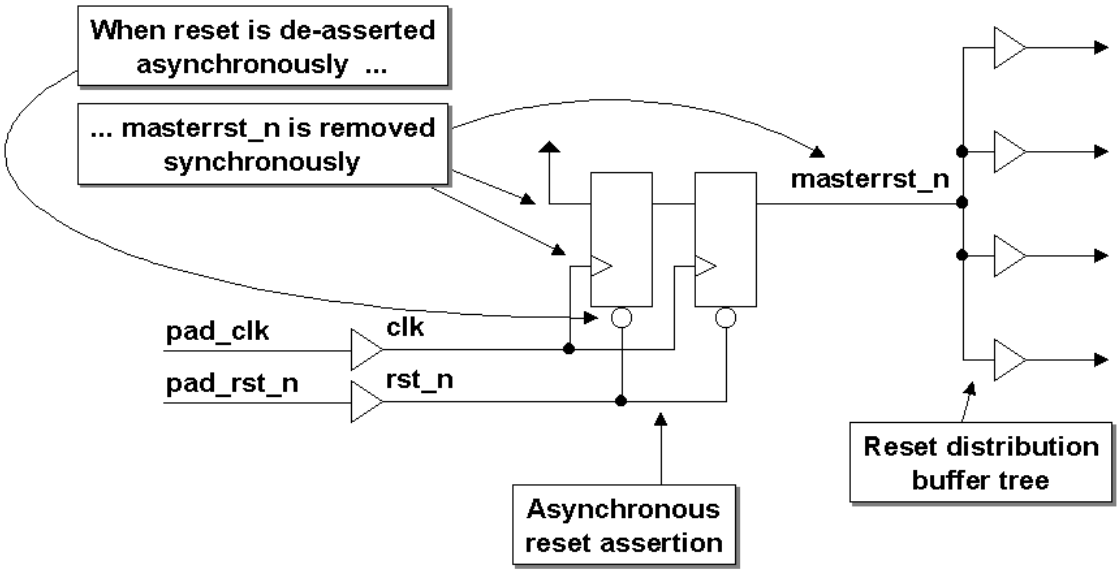


Figure 6 - Reset Synchronizer block diagram

An external reset signal asynchronously resets a pair of master reset flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge. As discussed in section 4.4, these synchronization flip-flops must be kept off of the scan chain.

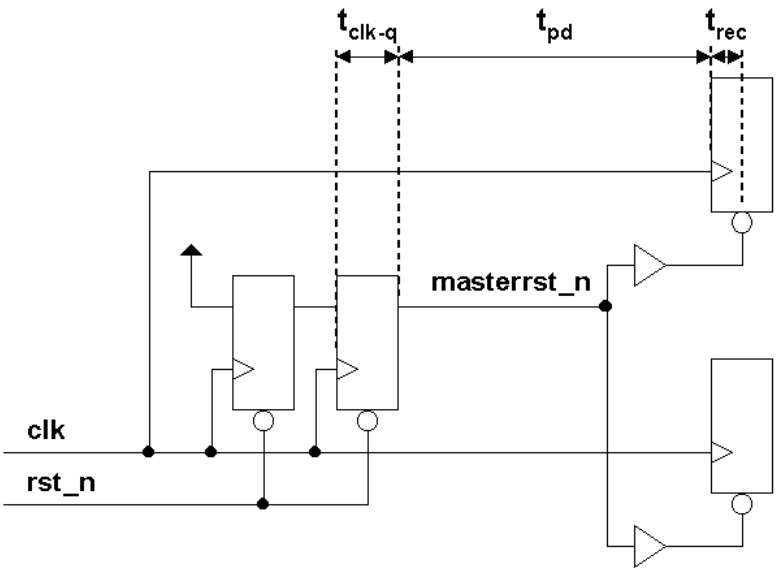


Figure 7 - Predictable reset removal to satisfy reset recovery time

A closer examination of the timing now shows that reset distribution timing is the sum of the a clk-to-q propagation delay, total delay through the reset distribution tree and meeting the reset recovery time of the destination registers and flip-flops, as shown in Figure 7.

The code for the reset synchronizer circuit is shown in Example 8.

```
module async_resetFFstyle2 (rst_n, clk, asyncrst_n);
  output rst_n;
  input  clk, asyncrst_n;
  reg    rst_n, rff1;

  always @(posedge clk or negedge asyncrst_n)
    if (!asyncrst_n) {rst_n,rff1} <= 2'b0;
    else             {rst_n,rff1} <= {rff1,1'b1};
endmodule
```

Example 8a - Properly coded reset synchronizer using Verilog

```
library ieee;
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk        : in  std_logic;
    asyncrst_n : in  std_logic;
    rst_n      : out std_logic);
end asyncresetFFstyle;

architecture rtl of asyncresetFFstyle is
  signal rff1 : std_logic;
begin
  process (clk, asyncrst_n)
  begin
    if (asyncrst_n = '0') then
```

```
      rff1  <= '0';
      rst_n <= '0';
    elsif (clk'event and clk = '1') then
      rff1  <= '1';
      rst_n <= rff1;
    end if;
  end process;
end rtl;
```

Example 8b - Properly coded reset synchronizer using VHDL

7.0 Reset distribution tree

The reset distribution tree requires almost as much attention as a clock distribution tree, because there are generally as many reset-input loads as there are clock-input loads in a typical digital design, as shown in Figure 8. The timing requirements for reset tree are common for both synchronous and asynchronous reset styles.

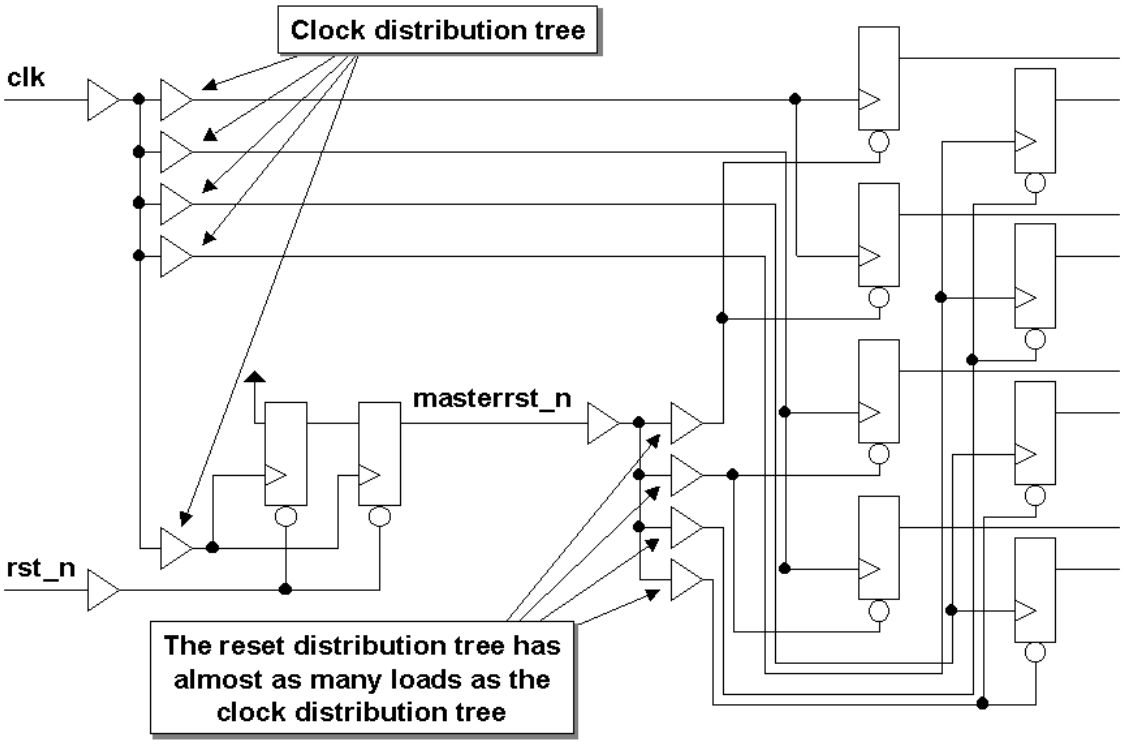


Figure 8 - Reset distribution tree

One important difference between a clock distribution tree and a reset distribution tree is the requirement to closely balance the skew between the distributed resets. Unlike clock signals, skew between reset signals is not critical as long as the delay associated with any reset signal is short enough to allow propagation to all reset loads within a clock period and still meet recovery time of all destination registers and flip-flops.

Care must be taken to analyze the clock tree timing against the clk-q-reset tree timing. The safest way to clock a reset tree (synchronous or asynchronous reset) is to clock the internal-master-reset flip-flop from a leaf-clock of the clock tree as shown in Figure 9. If this approach will meet timing, life is good. In most cases, there is not enough time to have a clock pulse traverse the clock tree, clock the reset-driving flip-flop and then have the reset traverse the reset tree, all within one clock period.

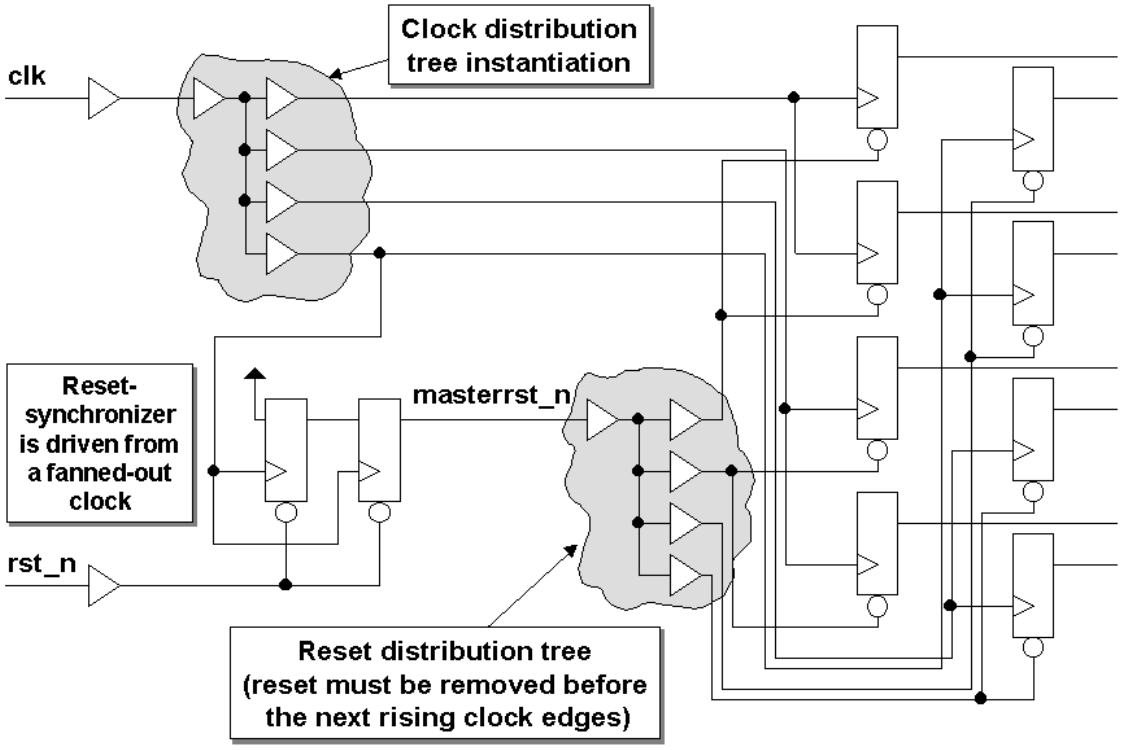


Figure 9 - Reset tree driven from a delayed, buffered clock

In order to help speed the reset arrival to all the system flip-flops, the reset-driver flip-flop is clocked with an early clock as shown in Figure 10. Post layout timing analysis must be made to ensure that the reset release for asynchronous resets and both the assertion and release for synchronous reset do not beat the clock to the flip-flops; meaning the reset must not violate setup and hold on the flops. Often detailed timing adjustments like this can not be made until the layout is done and real timing is available for the two trees.

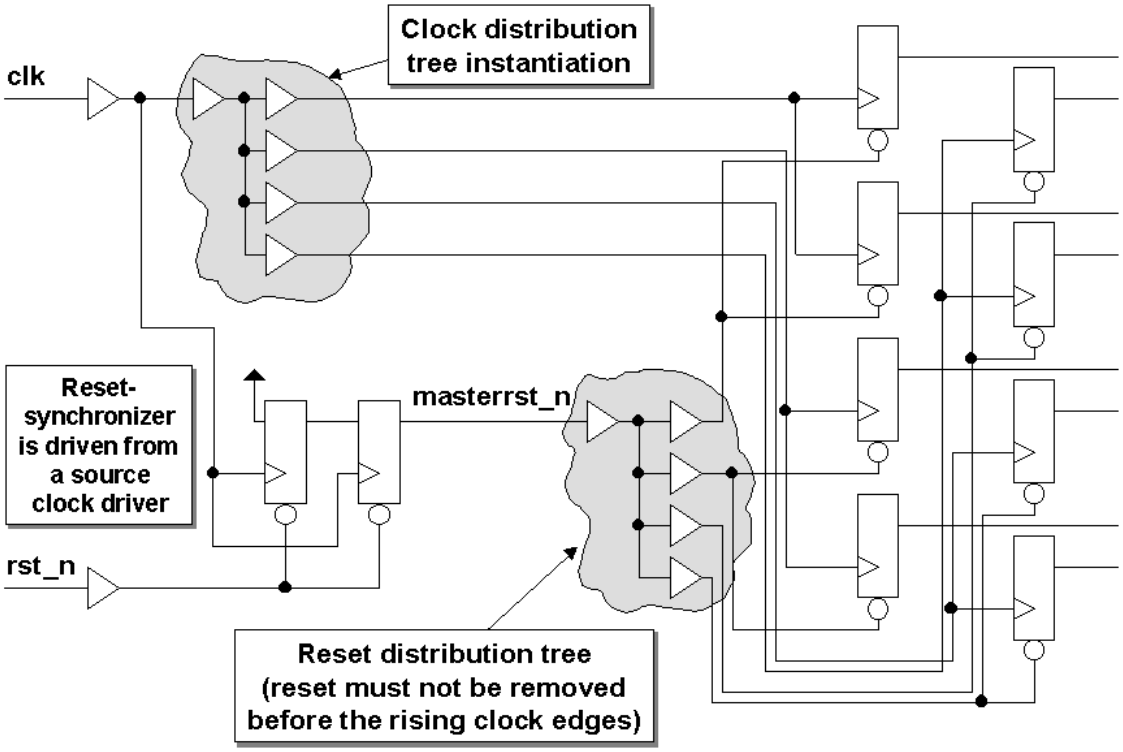


Figure 10 - Reset synchronizer driven in parallel to the clock distribution tree

Ignoring this problem will not make it go away. Gee, and we all thought resets were such a basic topic.

8.0 Reset-glitch filtering

As stated earlier in this paper, one of the biggest issues with asynchronous resets is that they are asynchronous and therefore carry with them some characteristics that must be dealt with depending on the source of the reset. With asynchronous resets, any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitching, this can be a real problem. Presented here is one approach that will work to filter out the glitches, but it is ugly! This solution requires that a digital delay (meaning the delay will vary with temperature, voltage and process) to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 11 shows the implementation of this approach.

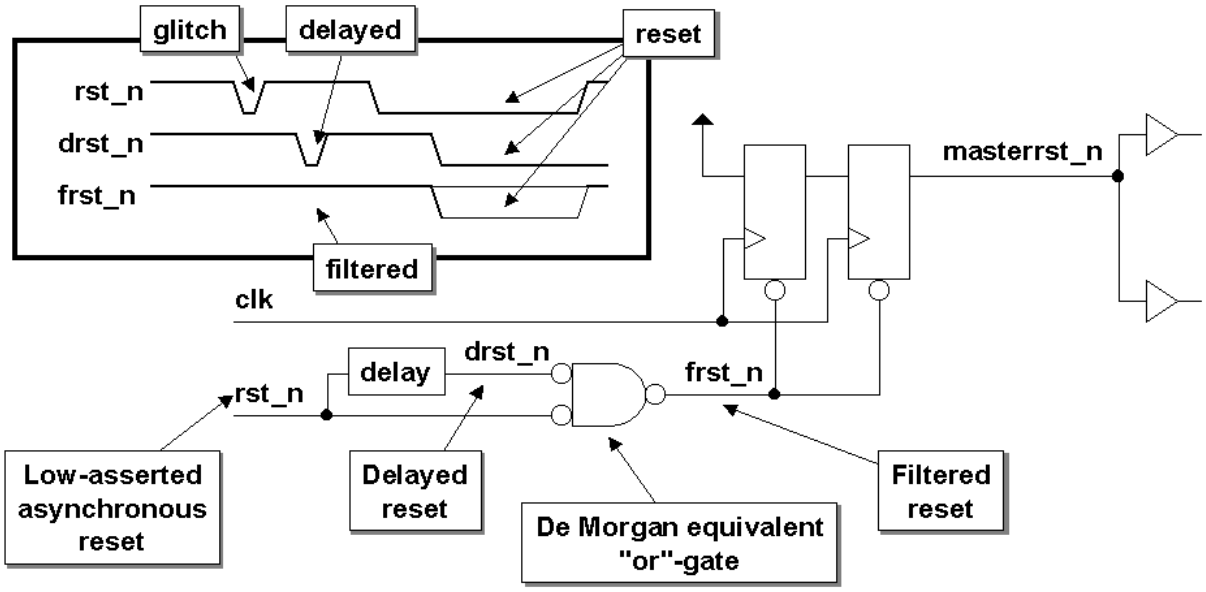


Figure 11 - Reset glitch filtering

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization – remember not to optimize this block after the delay has been inserted or it will be removed. Of course the elements could have don't touch attributes applied to prevent them from being removed. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

This glitch filter is not needed in all systems. The designer must research the system requirements to determine whether or not a delay is needed.

9.0 DFT for asynchronous resets

Applying Design for Test (DFT) functionality to a design is a two step process. First, the flips-flops in the design are stitched together into a scan chain accessible from external I/O pins, this is called scan insertion. The scan chain is typically not part of the functional design. Second, a software program is run to generate a set of scan vectors that, when applied to the scan chain, will test and verify the design. This software program is called Automatic Test Program Generation or ATPG. The primary objective of the scan vectors is to provide foundry vectors for manufacture tests of the wafers and die as well as tests for the final packaged part.

The process of applying the ATPG vectors to create a test is based on:

1. scanning a known state into all the flip-flops in the chip,
2. switching the flip-flops from scan shift mode, to functional data input mode,
3. applying one functional clock,
4. switching the flip-flops back to scan shift mode to scan out the result of the one functional clock while scanning in the next test vector.

The DFT process usually requires two control pins. One that puts the design into "test mode." This pin is used to mask off non-testable logic such as internally generated asynchronous resets, asynchronous combinational feedback loops, and many other logic conditions that require special attention. This pin is usually held constant during the entire test. The second control pin is the shift enable pin.

In order for the ATPG vectors to work, the test program must be able to control all the inputs to the flip-flops on the scan chain in the chip. This includes not only the clock and data, but also the reset pin (synchronous or

asynchronous). If the reset is driven directly from an I/O pin, then the reset is held in a non-reset state. If the reset is internally generated, then the master internal reset is held in a non-reset state by the test mode signal. If the internally generated reset were not masked off during ATPG, then the reset condition might occur during scan causing the flip-flops in the chip to be reset, and thus lose the vector data being scanned in.

Even though the asynchronous reset is held to the non-reset state for ATPG, this does not mean that the reset/set cannot be tested as part of the DFT process. Before locking out the reset with test mode and generating the ATPG vectors, a few vectors can be manually generated to create reset/set test vectors. The process required to test asynchronous resets for DFT is very straight forward and may be automatic with some DFT tools. If the scan tool does not automatic test the asynchronous resets/sets, then they must be setup manually. The basic steps to manually test the asynchronous resets/sets are as follows:

1. scan in all ones into the scan chain
2. issue and release the asynchronous reset
3. scan out the result and scan in all zeros
4. issue and release the reset
5. scan out the result
6. set the reset input to the non reset state and then apply the ATPG generated vectors.

This test approach will scan test for both asynchronous resets and sets. These manually generated vectors will be added to the ATPG vectors to provide a higher fault coverage for the manufacture test. If the design uses flip-flops with synchronous reset inputs, then modifying the above manual asynchronous reset test slightly will give a similar test for the synchronous reset environment. Add to the steps above a functional clock while the reset is applied. All other steps would remain the same.

For the reset synchronizer circuit discussed in this paper, the two synchronizer flips-flops should not be included in the scan chain, but should be tested using the manual process discussed above.

10.0 Multi-clock reset issues

For a multi-clock design, a separate asynchronous reset synchronizer circuit and reset distribution tree should be used for each clock domain. This is done to insure that reset signals can indeed be guaranteed to meet the reset recovery time for each register in each clock domain.

As discussed earlier, asynchronous reset assertion is not a problem. The problem is graceful removal of reset and synchronized startup of all logic after reset is removed.

Depending on the constraints of the design, there are two techniques that could be employed: (1) non-coordinated reset removal, and (2) sequenced coordination of reset removal.

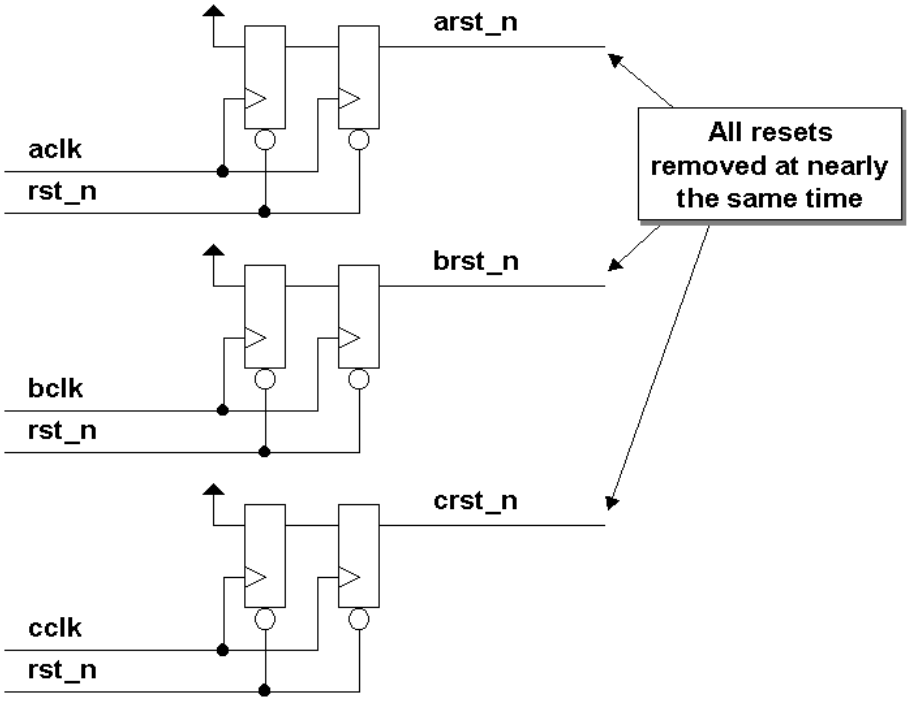


Figure 12 - Multi-clock reset removal

10.1 Non-coordinated reset removal

For many multi-clock designs, exactly when reset is removed within one clock domain compared to when it is removed in another clock domain is not important. Typically in these designs, any control signals crossing clock boundaries are passed through some type of request-acknowledge handshaking sequence and the delayed acknowledge from one clock domain to another is not going to cause invalid execution of the hardware. For this type of design, creating separate asynchronous reset synchronizers as shown in Figure 12 is sufficient, and the fact that **`arst_n`**, **`brst_n`** and **`crst_n`** could be removed in any sequence is not important to the design.

10.2 Sequenced coordination of reset removal

For some multi-clock designs, reset removal must be ordered and proper sequence. For this type of design, creating prioritized asynchronous reset synchronizers as shown in Figure 13 might be required to insure that all **`aclk`** domain logic is activated after reset is removed before the **`bclk`** logic, which must also be activated before the **`cclk`** logic becomes active.

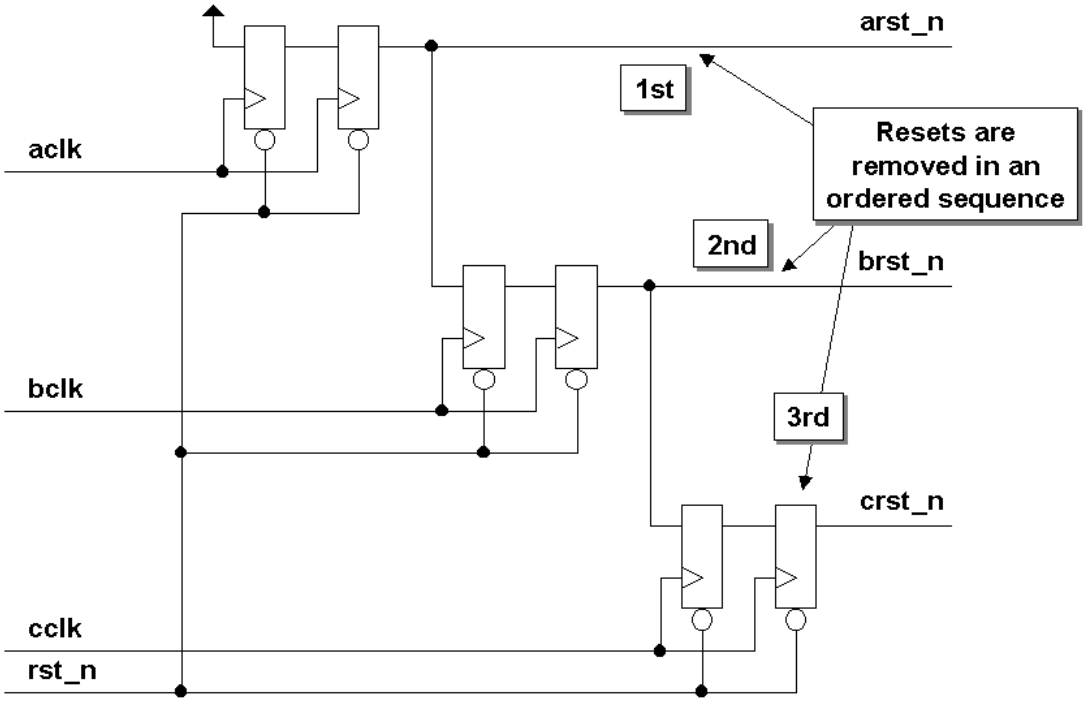


Figure 13 - Multi-clock ordered reset removal

For this type of design, only the highest priority asynchronous reset synchronizer input is tied high. The other asynchronous reset synchronizer inputs are tied to the master resets from higher priority clock domains.

11.0 Multi-ASIC reset synchronization

There are designs with multiple ASICs that require precise synchronization of reset removal across all of the multiple ASICs. One approach to satisfy this type of design, described in this section, is to use a different asynchronous reset synchronization scheme, one that only requires one reset removal flip-flop instead of the two flip-flops described in section 6.0, plus a digitally calibrated synchronization delay to properly sequence reset removal from the multiple ASICs.

Consider the actual design of a data acquisition board on a Digital Storage Oscilloscope (DSO). In rudimentary terms, a DSO is a test instrument that probes an analog signal, continuously does sampling and Analog-to-Digital (A2D) conversion of the signal, and continuously stores the sampled digital data into memory as fast as it can. After the requested trigger condition occurs, the rest of the data associated with the trigger condition is stored to memory and then DSO control logic (typically a commercial microprocessor) accesses the data and draws a waveform of the data values onto a screen for visual inspection.

For an actual design of this type, the data acquisition board contained four digital demultiplexer (demux) ASICs, each of which captured one-fourth of the **`datain`** samples to send to memory, as shown in Figure 14.

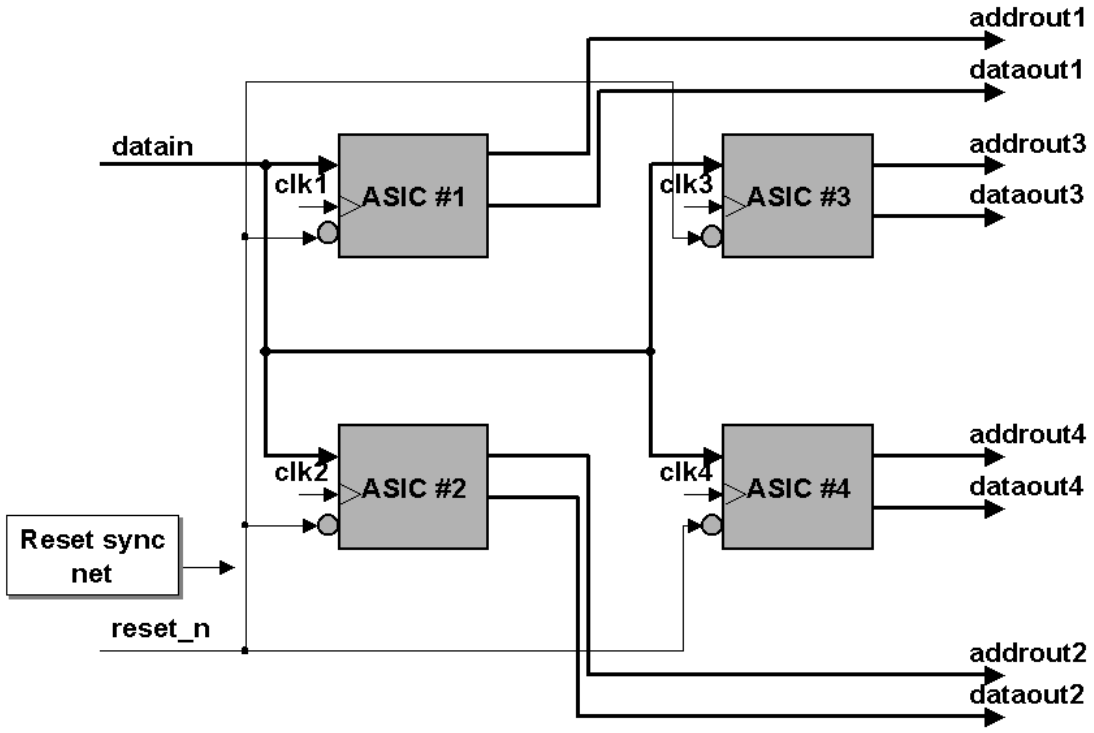


Figure 14 - Multi-ASIC design with synchronized reset removal problem

For this digital acquisition system, as soon as reset is removed, the ASICs must start capturing data and generating memory addresses to write the data to memory. Both data acquisition and address generation are continuously running, capturing data samples and overwriting previous written memory locations until a trigger circuit causes the address counters to stop and hold the data that has been most recently captured. Frequently, the trigger is set to hold and show 90% of the waveform as pre-trigger data and 10% of the waveform as post-trigger data. Since it is generally impossible to predict when the trigger will occur, it is necessary to continuously acquire data after reset removal until a trigger signal stops the data acquisition.

The approach that was used in this design to do high-speed data acquisition was to use four demux ASICs that capture every fourth point of the digitized waveform. Since the demux ASICs typically ran at very fast clock rates, and since each demux ASIC also had to generate accompanying address count values to store the data samples to memory, it was important that all four demux ASICs start their respective address counters in the correct sequence to insure that the data samples stored in memory could be easily read-back to draw waveforms on the DSO display.

The problem with this type of design was to accurately remove the reset signal from the four ASIC devices at the same time (in the same relative clock period) so that the four ASICs captured the correctly sequenced data samples that corresponded to address-#0 on all four ASICs, followed by address-#1 on all four ASICs, etc., so that the data stored to memory could be read back from memory (after triggering the DSO) in the correct sequence to display an accurate waveform on the DSO screen.

For this type of design, there are a number of factors that work against correct reset-removal and hence correct sequencing of the data values being written to memory.

First, for very high-speed designs (DSOs are typically very high-speed designs in order to capture an adequate number of data samples while probing other high-speed circuits), the relative board trace length of reset signals to the four ASICs would have to be held to a very tight tolerance; hence, board layout is an issue.

Second, process variations within or between batches of manufactured ASICs can create delays that exceed the ultra-short ASIC clock periods. Choosing four ASICs to insert during manufacture can result in selection of four devices

with different relative delays being placed on the same data acquisition board. The relative process speeds of the four ASICs placed on a board cannot be guaranteed (which of the four ASICs will always be the fastest? Who knows!)

Third, temperature swings in different test environments can also add to differences in delays. Relative positioning of the ASICs inside of a DSO enclosure might account for significant differences in temperature for this high-speed system.

Fourth, removing the covers of the DSO to troubleshoot prototypes could introduce different temperature variations across the four ASICs than when the covers are closed.

For the actual design, a common reset signal (**`reset_n`**) was routed to all four demux ASICs to assert reset, but the reset signal did not de-assert reset from the demux ASICs. A separate sync signal was used to flag reset removal permission on each demux ASIC.

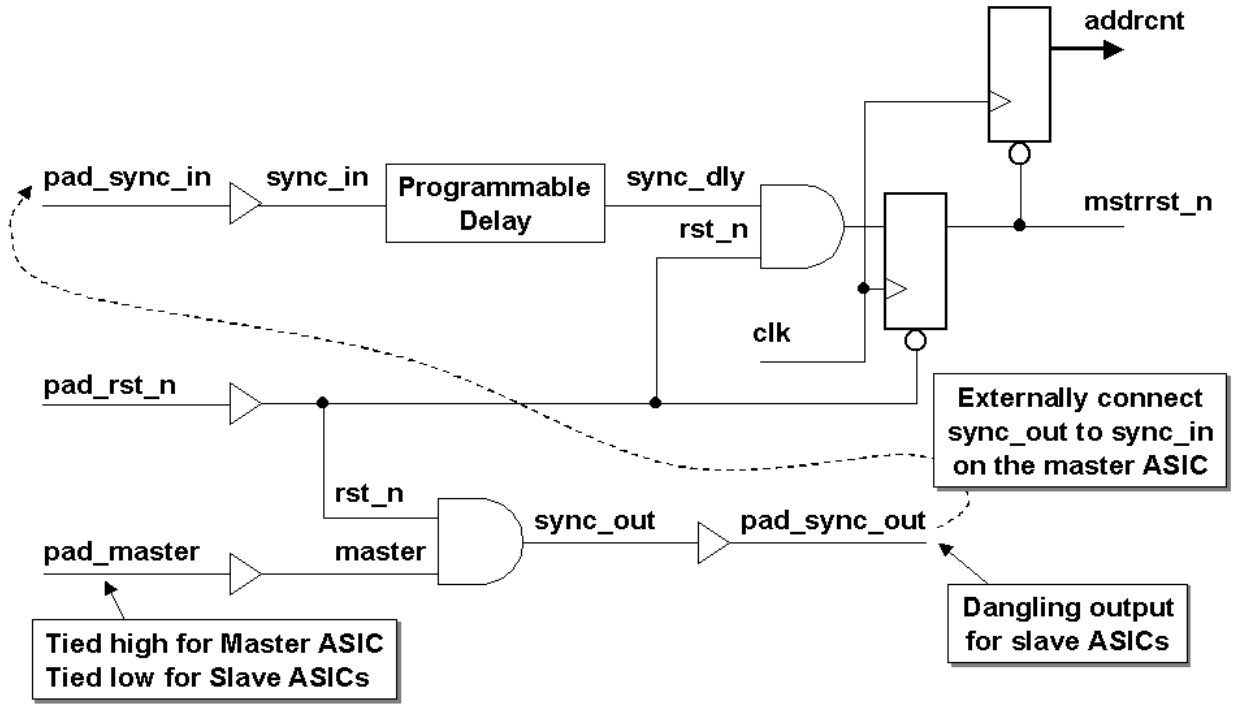


Figure 15 - Reset removal synchronization logic block diagram

The multi-ASIC reset removal synchronization logic is handled using the logic shown in Figure 15. This logic is common to both master and slave ASICs.

Asserting reset (**`rst_n`** going low in Figure 15) asynchronously resets the master reset signal, **`mstrrst_n`**, which is driven through a reset-tree to the rest of the resetable logic on all ASICs (both master and slave ASICs); therefore, reset is asynchronous and immediate.

Each ASIC has three pins dedicated to reset-removal synchronization.

The first pin on each ASIC is a dedicated master/slave selection pin. When this pin is tied high, the ASIC is placed into master mode. When the pin is tied low, the ASIC is placed into slave mode.

The second pin on each ASIC is the **`sync_out`** pin. On the slave ASICs, the **`sync_out`** pin is unused and left dangling. The master ASIC generates the **`sync_out`** pulse when reset is removed (when **`reset_n`** goes high). The **`sync_out`** signal is driven out of the master ASIC and is tied to the **`sync_in`** input on both master and slave ASICs through board-trace connections. The **`sync_out`** pin is the pin that controls reset removal on both the master ASIC and the slave ASICs.

The third pin on each ASIC is the **sync_in** pin. The **sync_in** pin is the input pin that is used to control reset removal on both master and slave ASICs. The **sync_in** signal is connected to a programmable delay block and is then enabled by a high-assertion on the reset input, that is then passed to a synchronous reset removal flip-flop. The next rising clock edge on the ASIC will cause the reset to be synchronously removed, permitting the address counters on each ASIC to start counting in a synchronized and orderly manner.

The problem, as explained earlier, is to insure that the **sync_in** signal removes the reset on the four ASICs in the correct order.

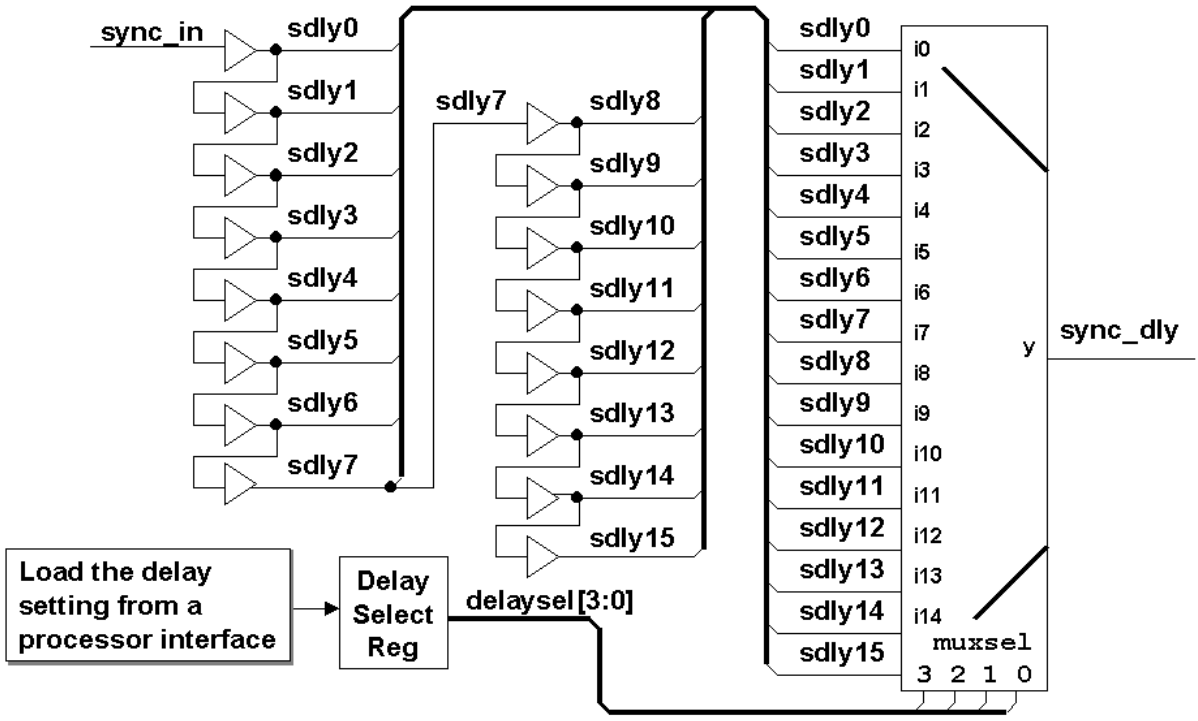


Figure 16 - Programmable digital delay block diagram

The programmable digital delay block, shown in Figure 16, is a set of delay stages connected in series with each delay-stage output driving both the next delay-stage input and an input on a multiplexer. The delay stages could be simple buffers or they could be pairs of inverters. The number of delay stages selected was equal to almost three ASIC clock cycles.

A processor interface is used to program the delay select register, which enables the multiplexer select lines to choose which delayed **sync_in** signal (**sdly0** to **sdly15**) would be driven to the mux output and used to remove the reset on the ASIC.

In order to determine the correct delay settings for each ASIC, a software digital calibration technique was employed.

To help calibrate the demux ASICs, as well as other analog devices on the data acquisition board, the board was designed to capture a selectable on-board ramp signal through the data acquisition path. The ramp signal was used to calibrate the delays on the four demux ASICs.

In Figure 17-Figure 19, the software programmable, digital calibration procedure is shown with just two of the four demux ASICs.

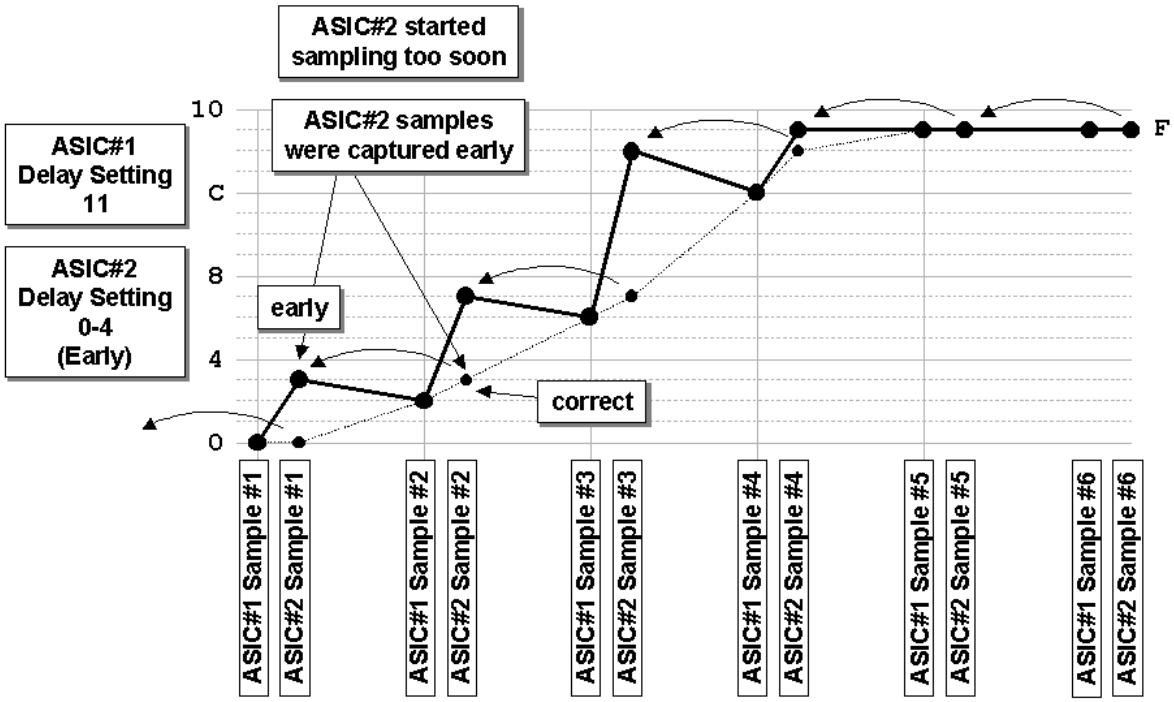


Figure 17 - Two-ASIC reset-removal calibration - early data sampling on ASIC #2

ASIC #1 is given the initial delay setting of 11 (to drive the **`sdly11`** signal to the mux output). ASIC #2 is given another delay setting and a ramp signal is captured by the data acquisition board. If the delay setting on ASIC #2 is too small, such as a delay value of 0-4 as shown in Figure 17, the ramp values captured by ASIC #2 will be sampled early compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is larger than the next data point captured by ASIC #1.

If the delay setting on ASIC #2 is in the correct range, such as a delay value of 5-11 as shown in Figure 18, the ramp values captured by ASIC #2 will be sampled in the correct order compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is larger than the previous data point captured by ASIC #1 and smaller than the next data point captured by ASIC #1.

If the delay setting on ASIC #2 is too large, such as a delay value of 12-15 as shown in Figure 19, the ramp values captured by ASIC #2 will be sampled late compared to the data points sampled by ASIC #1. This is manifest by the fact that each ramp data point captured by ASIC #2 is smaller than the next data point captured by ASIC #1.

Once the correct range is determined for ASIC #2, the center point in the range is chosen to be the ASIC #2 **`sync_in`** delay setting. The center point is the safest setting in the range since this setting is approximately a half-cycle between the previous and next rising clock edges for the reset-removal synchronization flip-flop.

After determining the correct ASIC #2 setting, the correct ASIC #1 range surrounding the initial setting (the setting of 11 is used in Figure 17) must be determined to find the correct ASIC #1 mid-point setting. After determining the correct ASIC #1 setting, a similar process is used to find the correct delay setting for ASIC #3, followed by finding the correct setting for ASIC #4.

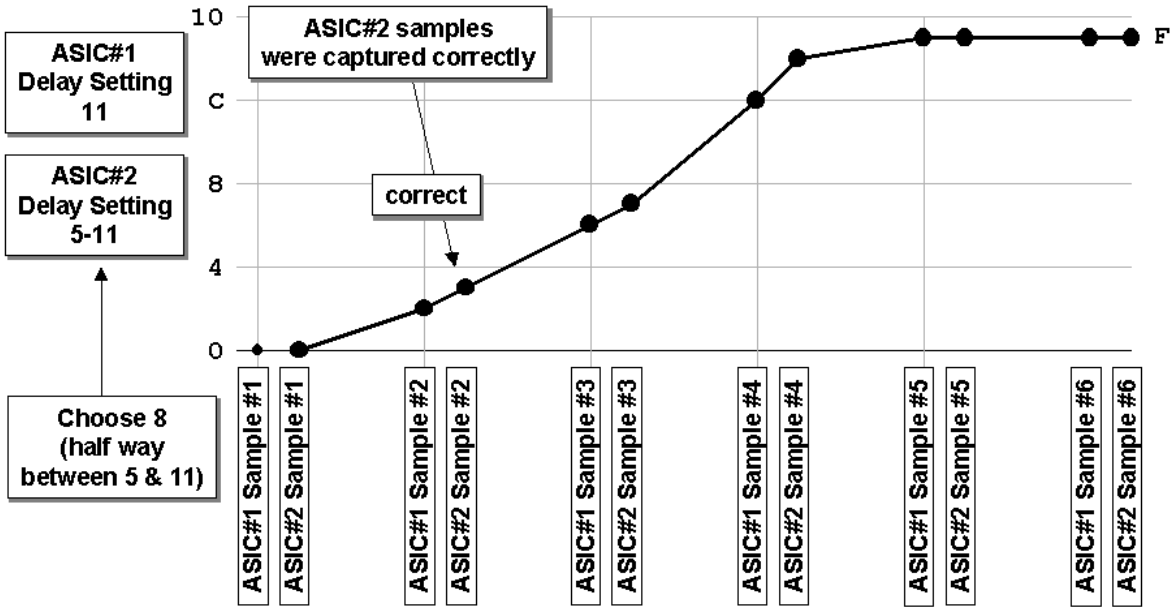


Figure 18 - Two-ASIC reset-removal calibration - correctly timed data sampling on ASIC #2

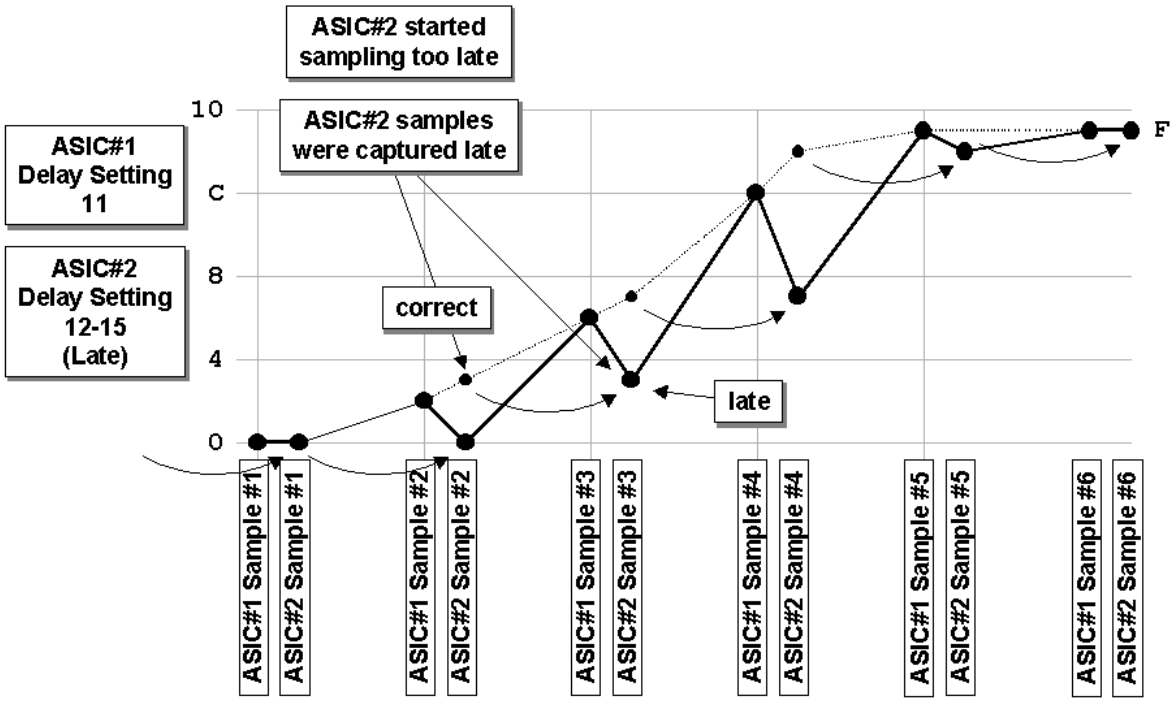


Figure 19 - Two-ASIC reset-removal calibration - late data sampling on ASIC #2

After digital calibration, there was no need to use a second reset-removal synchronization flip-flop because a mid-clock setting was used to insure that the flip-flop recovery time was met and to insure that no metastability problems would arise.

The full block diagram of the four-demux ASIC design with master/slave pin and **`sync_in`**/**`sync_out`** pins on each ASIC and how they were connected is shown in Figure 20.

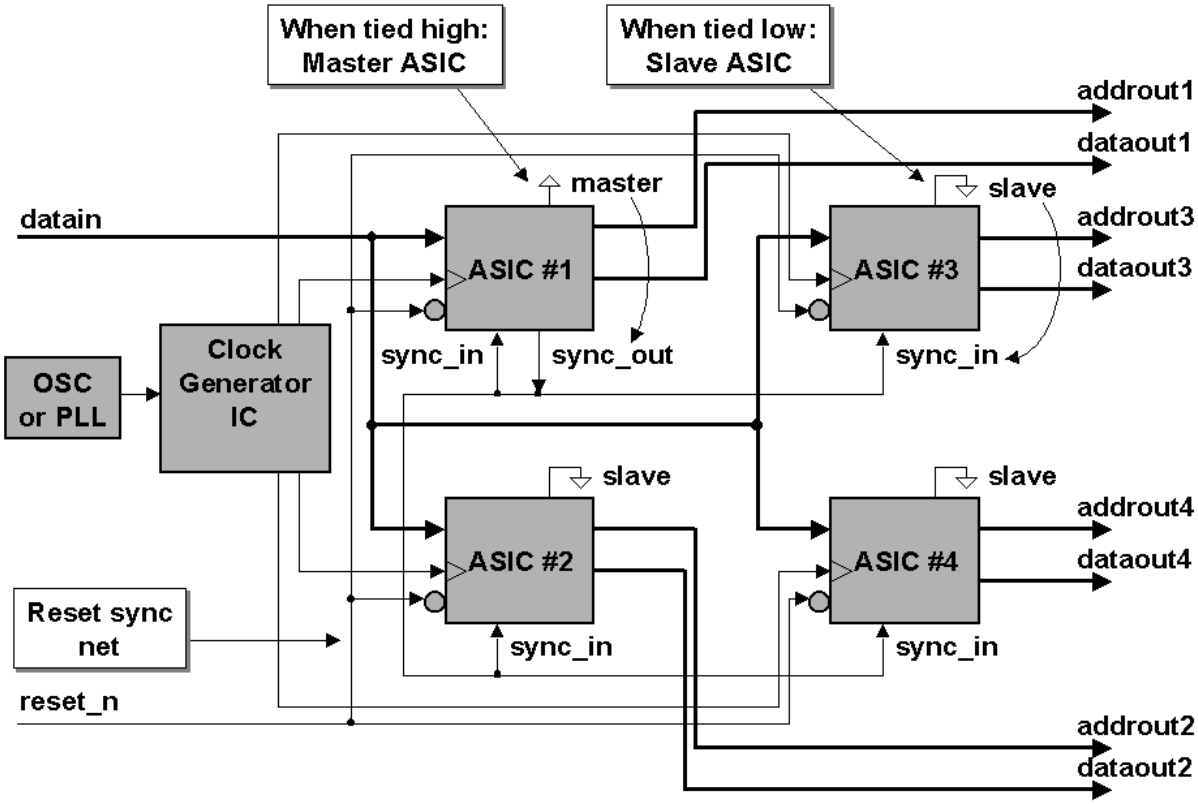


Figure 20 - Multi-ASIC synchronized reset removal solution

In the actual design, after determining a valid set of mid-point delay settings for the four ASICs on one of the data acquisition prototype boards, these values were programmed into a ROM and used as initial settings for all manufactured boards and variations from the initial settings were tracked. What was interesting was that the calibrated delay values for each board rarely strayed more than one or two delay stages up or down from the original settings of the initial data acquisition prototype board.

12.0 Conclusions

Using asynchronous resets is the surest way to guarantee reliable reset assertion. Although an asynchronous reset is a safe way to reliably reset circuitry, removal of an asynchronous reset can cause significant problems if not done properly.

The proper way to design with asynchronous resets is to add the reset synchronizer logic to allow asynchronous reset of the design and to insure synchronous reset removal to permit safe restoration of normal design functionality.

Using DFT with asynchronous resets is still achievable as long as the asynchronous reset can be controlled during test.

References

[1] *ALS/AS Logic Data Book*, Texas Instruments, 1986, pg. 2-78.

[2] Chris Kiegle, personal communication

[3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG (Synopsys Users Group) 2000 User Papers*, section-MC1 (1st paper), March 2000.
Also available at www.sunburst-design.com/papers

[4] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999.
Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[5] ESNUG #60, Item 1- http://www.deepchip.com/posts/0060.html

[6] ESNUG #240, Item 7- http://www.deepchip.com/posts/0240.html

[7] ESNUG #242, Item 6 - http://www.deepchip.com/posts/0242.html

[8] ESNUG #243, Item 4 - http://www.deepchip.com/posts/0243.html

[9] ESNUG #244, Item 5 - http://www.deepchip.com/posts/0244.html

[10] ESNUG #246, Item 5 - http://www.deepchip.com/posts/0246.html

[11] ESNUG #278, Item 7 - http://www.deepchip.com/posts/0278.html

[12] ESNUG #280, Item 4 - http://www.deepchip.com/posts/0280.html

[13] ESNUG #281, Item 2 - http://www.deepchip.com/posts/0281.html

[14] ESNUG #355, Item 2 - http://www.deepchip.com/posts/0355.html

[15] ESNUG #356, Item 4 - http://www.deepchip.com/posts/0356.html

[16] ESNUG #373, Item 6 - http://www.deepchip.com/posts/0373.html

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

[18] Michael Keating, and Pierre Bricaud, *Reuse Methodology Manual*, Second Edition, Kluwer Academic Publishers, 1999, pg. 35.

[19] Steve Golson, personal communication

[20] Synopsys SolvNet, Doc Name: METH-933.html, "Methodology and limitations of synthesis for synchronous set and reset," Updated 09/07/2001.

[21] Synopsys SolvNet, Doc Name: Physical_Synthesis-231.html, "Handling High Fanout Nets in 2001.08" Updated: 11/01/2001.

[22] Synopsys SolvNet, Doc Name: Star-15.html, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 09/07/2001.

[23] Synopsys SolvNet, Doc Name: Synthesis-452.html, "Why can't I synthesize synchronous reset flip-flops?," Updated: 08/16/1999.

[24] Synopsys SolvNet, Doc Name: Synthesis-780.html, "How can I use the high_fanout_net_threshold commands to simplify the net delay calculation?" Updated: 01/25/2002.

[25] Synopsys SolvNet, Doc Name: Synthesis-482109.html, "How to Eliminate Transition Time Calculation Side Effects From Arcs That Are Fal" Updated: 08/11/1997

[26] Synopsys SolvNet, Doc Name: Synthesis-799.html, "Data and Synchronous Reset Swapped," Updated: 05/01/2001.

Author & Contact Information

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

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

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

E-mail Address: cliffc@sunburst-design.com

Don Mills is an independent EDA consultant, ASIC designer, and Verilog/VHDL trainer with 16 years of experience.

Don has inflicted pain on Aart De Geuss for too many years as SNUG Technical Chair. Aart was more than happy to see him leave! Not really, Don chaired three San Jose SNUG conferences: 1998-2000, the first Boston SNUG 1999, and is currently chair of the Europe SNUG 2001- present.

Don holds a BSEE from Brigham Young University.

E-mail Address: mills@lcdm-eng.com

An updated version of this paper can be downloaded from the web site: www.sunburst-design.com/papers or from www.lcdm-eng.com

(Data accurate as of April 19th, 2002)

Asynchronous & Synchronous Reset Design Techniques - Part Deux

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

Don Mills
LCDM Engineering
mills@lcdm-eng.com

Steve Golson
Trilobyte Systems
sgolson@trilobyte.com

ABSTRACT

This paper will investigate the pros and cons of synchronous and asynchronous resets. It will then look at usage of each type of reset followed by recommendations for proper usage of each type.

1.0 Introduction

The topic of reset design is surprisingly complex and poorly emphasized. Engineering schools generally do an inadequate job of detailing the pitfalls of improper reset design. Based on our industry and consulting experience, we have compiled our current understanding of issues related to reset-design and for this paper have added the expertise of our colleague Steve Golson, who has done some very innovative reset design work. We continually solicit and welcome any feedback from colleagues related to this important design issue.

We presented our first paper on reset issues and techniques at the March 2002 SNUG conference[4] and have subsequently received numerous email responses and questions related to reset design issues.

We obviously did not adequately explain all of the issues related to the asynchronous reset synchronizer circuit because many of the emails we have received have asked if there are metastability problems related to the described circuit. The answer to this question is, no, there are no metastability issues related to this circuit and the technical analysis and explanation are now detailed in section 7.1 of this paper.

Whether to use synchronous or asynchronous resets in a design has almost become a religious issue with strong proponents claiming that their reset design technique is the only way to properly approach the subject.

In our first paper, Don and Cliff favored and recommended the use of asynchronous resets in designs and outlined our reasons for choosing this technique. With the help of our colleague, Steve Golson, we have done additional analysis on the subject and are now more neutral on the proper choice of reset implementation.

Clearly, there are distinct advantages and disadvantages to using either synchronous or asynchronous resets, and either method can be effectively used in actual designs. When choosing a reset style, it is very important to consider the issues related to the chosen style in order to make an informed design decision.

This paper presents updated techniques and considerations related to both synchronous and asynchronous reset design. This version of the paper includes updated Verilog-2001 ANSI-style ports in all of the Verilog examples.

The first version of this paper included an interesting technique for synchronizing the resetting of multiple ASICs of a high speed design application. That material has been deleted from this paper and readers are encouraged to read the first version of the paper if this subject is of interest.

2.0 Resets Purpose

Why be concerned with these annoying little resets anyway? Why devote a whole paper to such a trivial subject? Anyone who has used a PC with a certain OS loaded knows that the hardware reset comes in quite handy. It will put the computer back to a known working state (at least temporarily) by applying a system reset to each of the chips in the system that have or require a reset.

For individual ASICs, the primary purpose of a reset is to force the ASIC design (either behavioral, RTL, or structural) into a known state for simulation. Once the ASIC is built, the need for the ASIC to have reset applied is determined by the system, the application of the ASIC, and the design of the ASIC. For instance, many data path communication ASICs are designed to synchronize to an input data stream, process the data, and then output it. If sync is ever lost, the ASIC goes through a routine to re-acquire sync. If this type of ASIC is designed correctly, such that all unused states point to the "start acquiring sync" state, it can function properly in a system without ever being reset. A system reset would be required on power up for such an ASIC if the state machines in the ASIC took advantage of "don't care" logic reduction during the synthesis phase.

We believe that, in general, every flip-flop in an ASIC should be resetable whether or not it is required by the system. In some cases, when pipelined flip-flops (shift register flip-flops) are used in high speed applications, reset might be eliminated from some flip-flops to achieve higher performance designs. This type of environment requires a predetermined number of clocks during the reset active period to put the ASIC into a known state.

Many design issues must be considered before choosing a reset strategy for an ASIC design, such as whether to use synchronous or asynchronous resets, will every flip-flop receive a reset, how will the reset tree be laid out and buffered, how to verify timing of the reset tree, how to functionally test the reset with test scan vectors, and how to apply the reset across multiple clocked logic partitions.

3.0 General flip-flop coding style notes

3.1 Synchronous reset flip-flops with non reset follower flip-flops

Each Verilog procedural block or VHDL process should model only one type of flip-flop. In other words, a designer should not mix resetable flip-flops with follower flip-flops (flops with no resets) in the same procedural block or process[14]. Follower flip-flops are flip-flops that are simple data shift registers.

In the Verilog code of Example 1a and the VHDL code of Example 1b, a flip-flop is used to capture data and then its output is passed through a follower flip-flop. The first stage of this design is reset with a synchronous reset. The second stage is a follower flip-flop and is not reset, but because the two flip-flops were inferred in the same procedural block/process, the reset signal **`rst_n`** will be used as a data enable for the second flop. This coding style will generate extraneous logic as shown in Figure 1.

```
module badFFstyle (
  output reg q2,
  input      d, clk, rst_n);
  reg        q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else begin
      q1 <= d;
      q2 <= q1;
    end
endmodule
```

Example 1a - Bad Verilog coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity badFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end badFFstyle;

architecture rtl of badFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q1 <= '0';
      else
        q1 <= d;
        q2 <= q1;
      end if;
    end if;
  end process;
end rtl;
```

Example 1b - Bad VHDL coding style to model dissimilar flip-flops

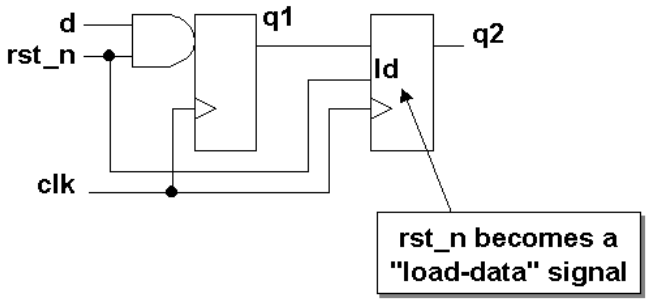


Figure 1 - Bad coding style yields a design with an unnecessary loadable flip-flop

The correct way to model a follower flip-flop is with two Verilog procedural blocks as shown in Example 2a or two VHDL processes as shown in Example 2b. These coding styles will generate the logic shown in Figure 2.

```
module goodFFstyle (
  output reg q2,
  input      d, clk, rst_n);
  reg        q1;

  always @(posedge clk)
    if (!rst_n) q1 <= 1'b0;
    else        q1 <= d;

  always @(posedge clk)
    q2 <= q1;
endmodule
```

Example 2a - Good Verilog-2001 coding style to model dissimilar flip-flops

```
library ieee;
use ieee.std_logic_1164.all;
entity goodFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q2    : out std_logic);
end goodFFstyle;

architecture rtl of goodFFstyle is
  signal q1 : std_logic;
begin
  process (clk)
  begin
```

```
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q1 <= '0';
      else
        q1 <= d;
      end if;
    end if;
  end process;

  process (clk)
  begin
    if (clk'event and clk = '1') then
        q2 <= q1;
    end if;
  end process;
end rtl;
```

Example 2b - Good VHDL coding style to model dissimilar flip-flops

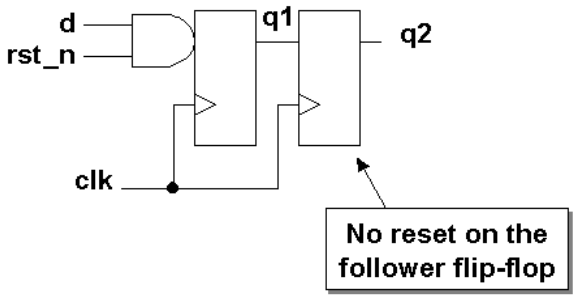


Figure 2 - Two different types of flip-flops, one with synchronous reset and one without

It should be noted that the extraneous logic generated by the code in Example 1a and Example 1b is only a result of using a synchronous reset. If an asynchronous reset approach had be used, then both coding styles would synthesize to the same design without any extra combinational logic. The generation of different flip-flop styles is largely a function of the sensitivity lists and **`if-else`** statements that are used in the HDL code. More details about the sensitivity list and **`if-else`** coding styles are detailed in section 4.1.

3.2 Flip-flop inference style

Each inferred flip-flop should **not** be independently modeled in its own procedural block/process. As a matter of style, all inferred flip-flops of a given function or even groups of functions should be described using a single procedural block/process. Multiple procedural blocks/processes should be used to model larger partitioned blocks within a given module/architecture. The exception to this guideline is that of follower flip-flops as discussed in section 3.1 where multiple procedural blocks/processes are required to efficiently model the function itself.

3.3 Assignment operator guideline

In Verilog, all assignments made inside the always block modeling an inferred flip-flop (sequential logic) should be made with nonblocking assignment operators[3]. Likewise, for VHDL, inferred flip-flops should be made using signal assignments.

4.0 Synchronous resets

As research was conducted for this paper, a collection of ESNUG and SOLV-IT articles was gathered and reviewed. Around 80+% of the gathered articles focused on synchronous reset issues. Many SNUG papers have been presented in which the presenter would claim something like, "we all know that the best way to do resets in an ASIC is to strictly use synchronous resets", or maybe, "asynchronous resets are bad and should be avoided." Yet, little evidence was offered to justify these statements. There are both advantages and disadvantages to using either synchronous or asynchronous resets. The designer must use an approach that is appropriate for the design.

Synchronous resets are based on the premise that the reset signal will only affect or reset the state of the flip-flop on the active edge of a clock. The reset can be applied to the flip-flop as part of the combinational logic generating the d-input to the flip-flop. If this is the case, the coding style to model the reset should be an **`if`**/**`else`** priority style with the reset in the **`if`** condition and all other combinational logic in the **`else`** section. If this style is not strictly observed, two possible problems can occur. First, in some simulators, based on the logic equations, the logic can block the reset from reaching the flip-flop. This is only a simulation issue, not a hardware issue, but remember, one of the prime objectives of a reset is to put the ASIC into a known state for simulation. Second, the reset could be a "late arriving signal" relative to the clock period, due to the high fanout of the reset tree. Even though the reset will be buffered from a reset buffer tree, it is wise to limit the amount of logic the reset must traverse once it reaches the local logic. This style of synchronous reset can be used with any logic or library. Example 3 shows an implementation of this style of synchronous reset as part of a loadable counter with carry out.

```
module ctr8sr (
  output reg [7:0] q,
  output reg       co,
  input      [7:0] d,
  input            ld, clk, rst_n);

  always @(posedge clk)
    if      (!rst_n) {co,q} <= 9'b0;     // sync reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 3a - Verilog-2001 code for a loadable counter with synchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8sr is
  port (
    clk        : in  std_logic;
    rst_n      : in  std_logic;
    d          : in  std_logic;
    ld         : in  std_logic;
    q          : out std_logic_vector(7 downto 0);
    co         : out std_logic);
end ctr8sr;

architecture rtl of ctr8sr is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        count <= (others => '0');       -- sync reset
      elsif (ld = '1') then
        count <= '0' & d;               -- sync load
      else
        count <= count + 1;             -- sync increment
      end if;
    end if;
  end process;
end rtl;
```

Example 3b - VHDL code for a loadable counter with synchronous reset

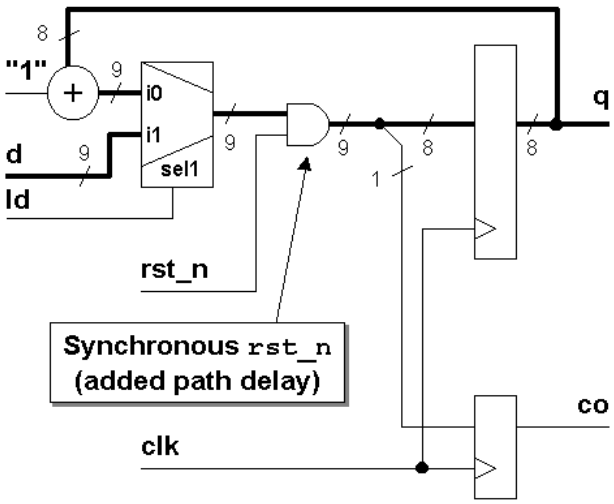


Figure 3 - Loadable counter with synchronous reset

4.1 Coding style and example circuit

The Verilog code of Example 4a and the VHDL code of 4b show the correct way to model synchronous reset flip-flops. Note that the reset is not part of the sensitivity list. For Verilog omitting the reset from the sensitivity list is what makes the reset synchronous. For VHDL omitting the reset from the sensitivity list and checking for the reset after the "**`if clk'event and clk = 1`**" statement makes the reset synchronous. Also note that the reset is given priority over any other assignment by using the **`if-else`** coding style.

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

  always @(posedge clk)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 4a - Correct way to model a flip-flop with synchronous reset using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity syncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end syncresetFFstyle;

architecture rtl of syncresetFFstyle is
```

```
begin
  process (clk)
  begin
    if (clk'event and clk = '1') then
      if (rst_n = '0') then
        q <= '0';
      else
        q <= d;
      end if;
    end if;
  end process;
end rtl;
```

Example 4b - Correct way to model a flip-flop with synchronous reset using VHDL

One problem with synchronous resets is that the synthesis tool cannot easily distinguish the reset signal from any other data signal. Consider the code from Example 3, which resulted in the circuit of Figure 3. The synthesis tool could alternatively have produced the circuit of Figure 4.

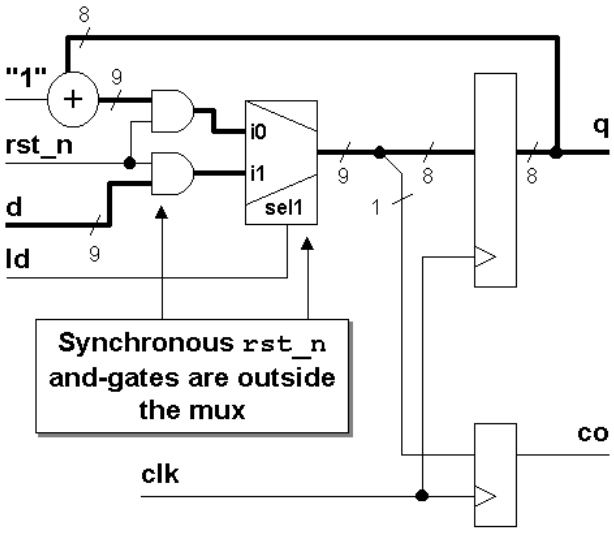


Figure 4 - Alternative circuit for loadable counter with synchronous reset

This is functionally identical to Figure 3. The only difference is that the reset and-gates are outside the MUX. Now, consider what happens at the start of a gate-level simulation. The inputs to both legs of the MUX can be forced to 0 by holding **`rst_n`** asserted low, however if **`ld`** is unknown (X) and the MUX model is pessimistic, then the flops will stay unknown (X) rather than being reset. Note this is only a problem during simulation! The actual circuit would work correctly and reset the flops to 0.

Synopsys provides the compiler directive **`sync_set_reset`** which tells the synthesis tool that a given signal is a synchronous reset (or set). The synthesis tool will "pull" this signal as close to the flop as possible to prevent this initialization problem from occurring. In this example the directive would be used by adding the following line somewhere inside the module:

```
// synopsys sync_set_reset "rst_n"
```

In general, we recommend only using Synopsys switches when they are required and make a difference; however the **sync_set_reset** directive does not affect the logical behavior of a design, instead it only impacts the functional implementation of a design. A wise engineer would prefer to avoid re-synthesizing the design late in the project schedule and would add the **sync_set_reset** directive to all RTL code from the start of the project. Since this directive is only required once per module, adding it to each module with synchronous resets is recommended.

Alternatively the synthesis variable **hdlin_ff_always_sync_set_reset** can be set to **true** prior to reading in the RTL, which will give the same result without requiring any directives in the code itself.

A few years back, another ESNUG contributor recommended adding the **compile_preserve_sync_resets = "true"** synthesis variable [15]. Although this variable might have been useful a few years ago, it was discontinued starting with Synopsys version 3.4b[38].

4.2 Advantages of synchronous resets

Synchronous reset logic will synthesize to smaller flip-flops, particularly if the reset is gated with the logic generating the d-input. But in such a case, the combinational logic gate count grows, so the overall gate count savings may not be that significant. If a design is tight, the area savings of one or two gates per flip-flop may ensure the ASIC fits into the die. However, in today's technology of huge die sizes, the savings of a gate or two per flip-flop is generally irrelevant and will not be a significant factor of whether a design fits into a die.

Synchronous resets generally insure that the circuit is 100% synchronous.

Synchronous resets insure that reset can only occur at an active clock edge. The clock works as a filter for small reset glitches; however, if these glitches occur near the active clock edge, the flip-flop could go metastable. This is no different or worse than every other data input; any signal that violates setup requirements can cause metastability.

In some designs, the reset must be generated by a set of internal conditions. A synchronous reset is recommended for these types of designs because it will filter the logic equation glitches between clocks.

By using synchronous resets and a pre-determined number of clocks as part of the reset process, flip-flops can be used within the reset buffer tree to help the timing of the buffer tree keep within a clock period.

According to the Reuse Methodology Manual (RMM)[32], synchronous resets might be easier to work with when using cycle based simulators. For this reason, synchronous resets are recommend in section 3.2.4(2nd edition, section 3.2.3 in the 1st edition) of the RMM. We believe using asynchronous resets with a good testbench coding style, where reset stimulus is only changed on clock edges, removes any simulation ease or speed advantages attributed to synchronous reset designs by the RMM. Translation: it is doubtful that reset style makes much difference in either ease or speed of simulation.

4.3 Disadvantages of synchronous resets

Not all ASIC libraries have flip-flops with built-in synchronous resets. However since synchronous reset is just another data input, you don't really need a special flop. The reset logic can easily be synthesized outside the flop itself.

Synchronous resets may need a pulse stretcher to guarantee a reset pulse width wide enough to ensure reset is present during an active edge of the clock[16]. This is an issue that is important to consider when doing multi-clock design. A small counter can be used that will guarantee a reset pulse width of a certain number of cycles.

A designer must work with simulator issues. A potential problem exists if the reset is generated by combinational logic in the ASIC or if the reset must traverse many levels of local combinational logic. During simulation, depending on how the reset is generated or how the reset is applied to a functional block, the reset can be masked by X's. A large number of the ESNUG articles address this issue. Most simulators will not resolve some X-logic conditions and therefore block out the synchronous reset[7][8][9][10][11][12][13][14][15][34]. Note this can also be an issue with asynchronous resets. The problem is not so much what type of reset you have, but whether the reset signal is easily controlled by an external pin.

By its very nature, a synchronous reset will require a clock in order to reset the circuit. This may not be a disadvantage to some design styles but to others, it may be an annoyance. For example, if you have a gated clock to save power, the clock may be disabled coincident with the assertion of reset. Only an asynchronous reset will work in this situation, as the reset might be removed prior to the resumption of the clock.

The requirement of a clock to cause the reset condition is significant if the ASIC/FPGA has an internal tristate bus. In order to prevent bus contention on an internal tristate bus when a chip is powered up, the chip should have a power-on asynchronous reset (see Figure 5). A synchronous reset could be used, however you must also directly de-assert the tristate enable using the reset signal (see Figure 6). This synchronous technique has the advantage of a simpler timing analysis for the reset-to-HiZ path.

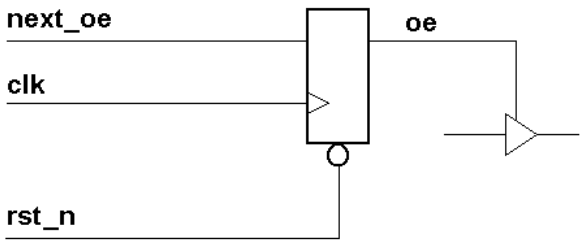


Figure 5 - Asynchronous reset for output enable

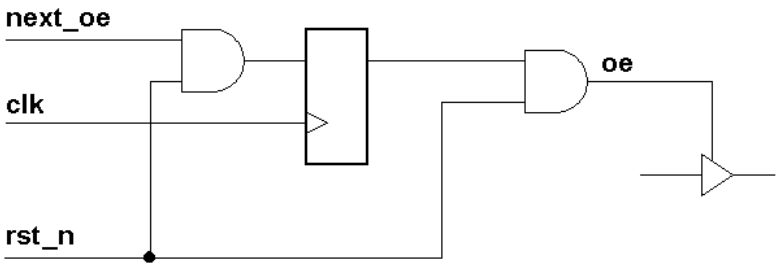


Figure 6 - Synchronous reset for output enable

5.0 Asynchronous resets

Improper implementation of asynchronous resets in digital logic design can cause serious operational design failures.

Many engineers like the idea of being able to apply the reset to their circuit and have the logic go to a known state. The biggest problem with asynchronous resets is the reset release, also called reset removal. The subject will be elaborated in detail in section 6.0.

Asynchronous reset flip-flops incorporate a reset pin into the flip-flop design. The reset pin is typically active low (the flip-flop goes into the reset state when the signal attached to the flip-flop reset pin goes to a logic low level.)

5.1 Coding style and example circuit

The Verilog code of Example 5a and the VHDL code of Example 5b show the correct way to model asynchronous reset flip-flops. Note that the reset *is* part of the sensitivity list. For Verilog, adding the reset to the sensitivity list is what makes the reset asynchronous. In order for the Verilog simulation model of an asynchronous flip-flop to simulate correctly, the sensitivity list should only be active on the leading edge of the asynchronous reset signal. Hence, in Example 5a, the always procedure block will be entered on the leading edge of the reset, then the **`if`** condition will check for the correct reset level.

Synopsys requires that if any signal in the sensitivity list is edge-sensitive, then all signals in the sensitivity list must be edge-sensitive. In other words, Synopsys forces the correct coding style. Verilog simulation does not have this requirement, but if the sensitivity list were sensitive to more than just the active clock edge and the reset leading edge, the simulation model would be incorrect[5]. Additionally, only the clock and reset signals can be in the sensitivity list. If other signals are included (legal Verilog, illegal Verilog RTL synthesis coding style) the simulation model would not be correct for a flip-flop and Synopsys would report an error while reading the model for synthesis.

For VHDL, including the reset in the sensitivity list and checking for the reset before the "**`if clk'event and clk = 1`**" statement makes the reset asynchronous. Also note that the reset is given priority over any other assignment (including the clock) by using the **`if/else`** coding style. Because of the nature of a VHDL sensitivity list and flip-flop coding style, additional signals can be included in the sensitivity list with no ill effects directly for simulation and synthesis. However, good coding style recommends that only the signals that can directly

change the output of the flip-flop should be in the sensitivity list. These signals are the clock and the asynchronous reset. All other signals will slow down simulation and be ignored by synthesis.

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

  // Verilog-2001: permits comma-separation
  // @(posedge clk, negedge rst_n)
  always @(posedge clk or negedge rst_n)
    if (!rst_n) q <= 1'b0;
    else        q <= d;
endmodule
```

Example 5a - Correct way to model a flip-flop with asynchronous reset using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk   : in  std_logic;
    rst_n : in  std_logic;
    d     : in  std_logic;
    q     : out std_logic);
end asyncresetFFstyle;

architecture rtl of asyncresetFFstyle is
begin
  process (clk, rst_n)
  begin
    if (rst_n = '0') then
      q <= '0';
    elsif (clk'event and clk = '1') then
      q <= d;
    end if;
  end process;
end rtl;
```

Example 5b - Correct way to model a flip-flop with asynchronous reset using VHDL

The approach to synthesizing asynchronous resets will depend on the designers approach to the reset buffer tree. If the reset is driven directly from an external pin, then usually doing a **`set_drive 0`** on the reset pin and doing a **`set_dont_touch_network`** on the reset net will protect the net from being modified by synthesis. However, there is at least one ESNUG article that indicates this is not always the case[18].

One ESNUG contributor[17] indicates that sometimes **`set_resistance 0`** on the reset net might also be needed.

Alternatively rather than having **`set_resistance 0`** on the net, you can create a custom wireload model with resistance=0 and apply it to the reset input port with the command:

```
set_wire_load -port_list reset
```

A recently updated SolvNet article also notes that starting with Synopsys release 2001.08 the definition of ideal nets has slightly changed[41] and that a **`set_ideal_net`** command can be used to create ideal nets and "get no timing updates, get no delay optimization, and get no DRC fixing."

Our colleague, Chris Kiegle, reported that doing a set_disable_timing on a net for pre-v2001.08 designs helped to clean up timing reports[2], which seems to be supported by two other SolvNet articles, one related to synthesis and another related to Physical Synthesis, that recommend usage of both a **`set_false_path`** and a **`set_disable_timing`** command[35].

5.2 Modeling Verilog flip-flops with asynchronous reset and asynchronous set

One additional note should be made here with regards to modeling asynchronous resets in Verilog. The simulation model of a flip-flop that includes both an asynchronous set and an asynchronous reset in Verilog might not simulate correctly without a little help from the designer. In general, most synchronous designs do not have flop-flops that contain both an asynchronous set and asynchronous reset, but on the occasion such a flip-flop is required. The coding style of Example 6 can be used to correct the Verilog RTL simulations where both reset and set are asserted simultaneously and reset is removed first.

First note that the problem is only a simulation problem and not a synthesis problem (synthesis infers the correct flip-flop with asynchronous set/reset). The simulation problem is due to the always block that is only entered on the active edge of the set, reset or clock signals. If the reset becomes active, followed then by the set going active, then if the reset goes inactive, the flip-flop should first go to a reset state, followed by going to a set state. With both these inputs being asynchronous, the set should be active as soon as the reset is removed, but that will not be the case in Verilog since there is no way to trigger the always block until the next rising clock edge.

For those rare designs where reset and set are both permitted to be asserted simultaneously and then reset is removed first, the fix to this simulation problem is to model the flip-flop using self-correcting code enclosed within the translate_off/translate_on directives and force the output to the correct value for this one condition. The best recommendation here is to avoid, as much as possible, the condition that requires a flip-flop that uses both asynchronous set and asynchronous reset. The code in Example 6 shows the fix that will simulate correctly and guarantee a match between pre- and post-synthesis simulations. This code uses the translate_off/translate_on directives to force the correct output for the exception condition[5].

```
// Good DFF with asynchronous set and reset and self-
// correcting set-reset assignment
module dff3_aras (
```

```
  output reg q,
  input      d, clk, rst_n, set_n);

  always @(posedge clk or negedge rst_n or negedge set_n)
    if      (!rst_n) q <= 0; // asynchronous reset
    else if (!set_n) q <= 1; // asynchronous set
    else             q <= d;

  // synopsys translate_off
  always @(rst_n or set_n)
    if (rst_n && !set_n) force   q = 1;
    else                 release q;
  // synopsys translate_on
endmodule
```

Example 6 – Verilog Asynchronous SET/RESET simulation and synthesis model

5.3 Advantages of asynchronous resets

The biggest advantage to using asynchronous resets is that, as long as the vendor library has asynchronously reset-able flip-flops, the data path is guaranteed to be clean. Designs that are pushing the limit for data path timing, can not afford to have added gates and additional net delays in the data path due to logic inserted to handle synchronous resets. Using an asynchronous reset, the designer is guaranteed not to have the reset added to the data path. The code in Example 7 infers asynchronous resets that will not be added to the data path.

```
module ctr8ar (
  output reg [7:0] q,
  output reg       co;
  input      [7:0] d;
  input            ld, rst_n, clk;

  always @(posedge clk or negedge rst_n)
    if      (!rst_n) {co,q} <= 9'b0;     // async reset
    else if (ld)     {co,q} <= d;        // sync load
    else             {co,q} <= q + 1'b1; // sync increment
endmodule
```

Example 7a - Verilog-2001 code for a loadable counter with asynchronous reset

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
entity ctr8ar is
  port (
    clk        : in  std_logic;
    rst_n      : in  std_logic;
    d          : in  std_logic;
```

```
    ld         : in  std_logic;
    q          : out std_logic_vector(7 downto 0);
    co         : out std_logic);
end ctr8ar;

architecture rtl of ctr8ar is
  signal count : std_logic_vector(8 downto 0);
begin
  co <= count(8);
  q  <= count(7 downto 0);

  process (clk, rst_n)
  begin
    if (rst_n = '0') then
      count <= (others => '0');       -- async reset
      elsif (clk'event and clk = '1') then
        if (ld = '1') then
          count <= '0' & d;           -- sync load
        else
          count <= count + 1;         -- sync increment
        end if;
      end if;
  end process;
end rtl;
```

Example 7b- VHDL code for a loadable counter with asynchronous reset

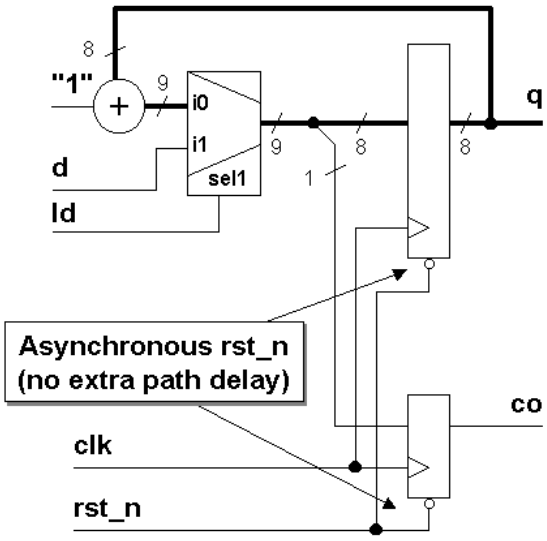


Figure 7 - Loadable counter with asynchronous reset

Another advantage favoring asynchronous resets is that the circuit can be reset with or without a clock present.

The experience of the authors is that by using the coding style for asynchronous resets described in this section, the synthesis interface tends to be automatic. That is, there is generally no need to add any synthesis attributes to get the synthesis tool to map to a flip-flop with an asynchronous reset pin.

5.4 Disadvantages of asynchronous resets

There are many reasons given by engineers as to why asynchronous resets are evil.

The Reuse Methodology Manual (RMM) suggests that asynchronous resets are not to be used because they cannot be used with cycle based simulators. This is simply not true. The basis of a cycle based simulator is that all inputs change on a clock edge. Since timing is not part of cycle based simulation, the asynchronous reset can simply be applied on the inactive clock edge.

For DFT, if the asynchronous reset is not directly driven from an I/O pin, then the reset net from the reset driver must be disabled for DFT scanning and testing. This is required for the synchronizer circuit shown in section 6.

Some designers claim that static timing analysis is very difficult to do with designs using asynchronous resets. The reset tree must be timed for both synchronous and asynchronous resets to ensure that the release of the reset can occur within one clock period. The timing analysis for a reset tree must be performed after layout to ensure this timing requirement is met. This timing analysis can be eliminated if the design uses the distributed reset synchronizer flip-flop tree discussed in section 8.2.

The biggest problem with asynchronous resets is that they are asynchronous, both at the assertion and at the de-assertion of the reset. The assertion is a non issue, the de-assertion is the issue. If the asynchronous reset is released at or near the active clock edge of a flip-flop, the output of the flip-flop could go metastable and thus the reset state of the ASIC could be lost.

Another problem that an asynchronous reset can have, depending on its source, is spurious resets due to noise or glitches on the board or system reset. See section 8.0 for a possible solution to reset glitches. If this is a real problem in a system, then one might think that using synchronous resets is the solution. A different but similar problem exists for synchronous resets if these spurious reset pulses occur near a clock edge, the flip-flops can still go metastable (but this is true of any data input that violates setup requirements).

6.0 Asynchronous reset problem

In discussing this paper topic with a colleague, the engineer stated first that since all he was working on was FPGAs, they do not have the same reset problems that ASICs have (a misconception). He went on to say that he always had an asynchronous system reset that could override everything, to put the chip into a known state. The engineer was then asked what would happen to the FPGA or ASIC if the release of the reset occurred on or near a clock edge such that the flip-flops went metastable.

Too many engineers just apply an asynchronous reset thinking that there are no problems. They test the reset in the controlled simulation environment and everything works fine, but then in the system, the design fails intermittently. The designers do not consider the idea that the release of the reset in the system (non-controlled environment) could cause the chip to go into a metastable

unknown state, thus voiding the reset all together. Attention must be paid to the release of the reset so as to prevent the chip from going into a metastable unknown state when reset is released. When a synchronous reset is being used, then both the leading and trailing edges of the reset must be away from the active edge of the clock.

As shown in Figure 8, an asynchronous reset signal will be de-asserted asynchronous to the clock signal. There are two potential problems with this scenario: (1) violation of reset recovery time and, (2) reset removal happening in different clock cycles for different sequential elements.

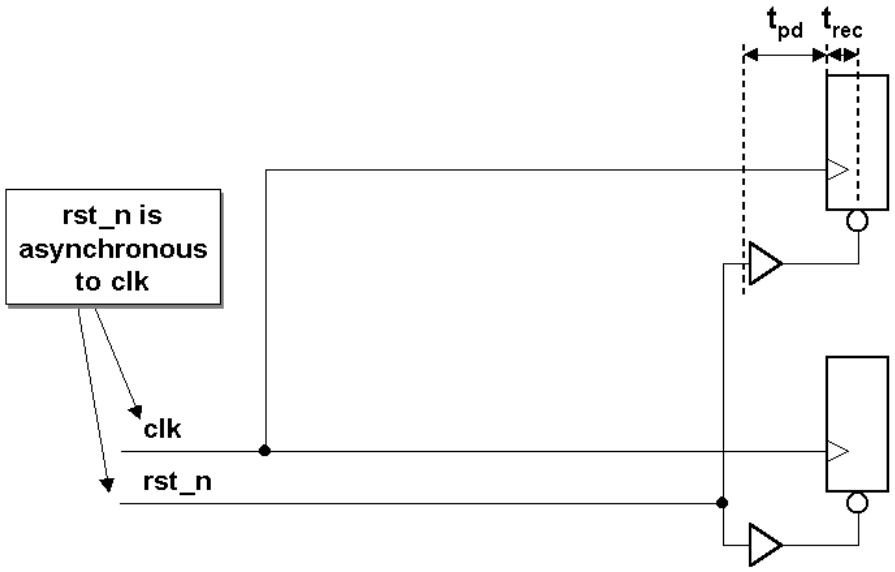


Figure 8 - Asynchronous reset removal recovery time problem

6.1 Reset recovery time

Reset recovery time refers to the time between when reset is de-asserted and the time that the clock signal goes high again. The Verilog-2001 Standard[29] has three built-in commands to model and test recovery time and signal removal timing checks: $recovery, $removal and $recrem (the latter is a combination of recovery and removal timing checks).

Recovery time is also referred to as a **tsu** setup time of the form, “PRE or CLR inactive setup time before CLK↑”[1].

Missing a recovery time can cause signal integrity or metastability problems with the registered data outputs.

6.2 Reset removal traversing different clock cycles

When reset removal is asynchronous to the rising clock edge, slight differences in propagation delays in either or both the reset signal and the clock signal can cause some registers or flip-flops to exit the reset state before others.

7.0 Reset synchronizer

Guideline: EVERY ASIC USING AN ASYNCHRONOUS RESET SHOULD INCLUDE A RESET SYNCHRONIZER CIRCUIT!!

Without a reset synchronizer, the usefulness of the asynchronous reset in the final system is void even if the reset works during simulation.

The reset synchronizer logic of Figure 9 is designed to take advantage of the best of both asynchronous and synchronous reset styles.

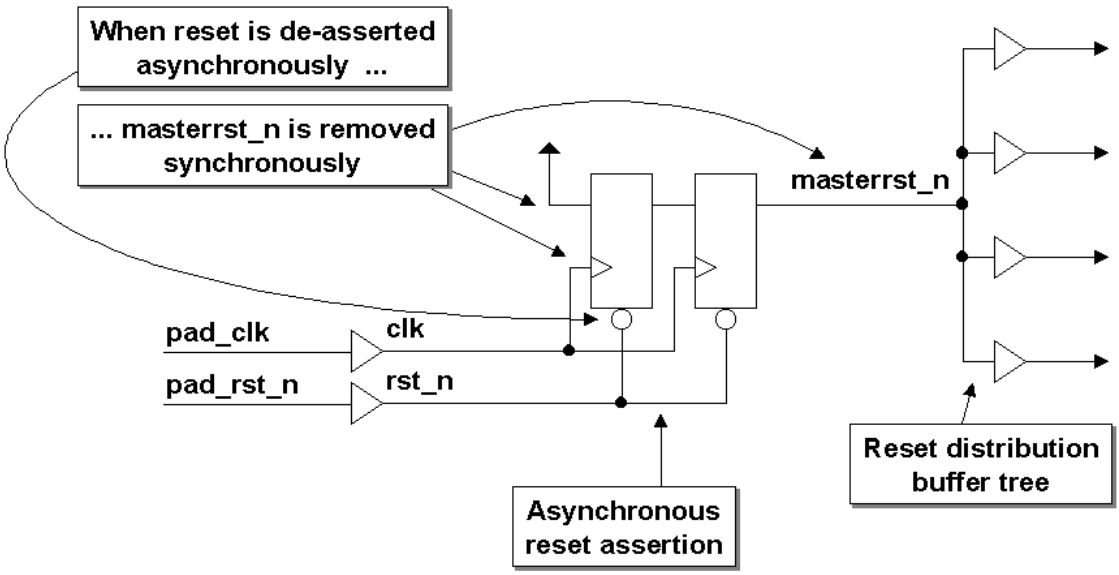


Figure 9 - Reset Synchronizer block diagram

An external reset signal asynchronously resets a pair of master reset flip-flops, which in turn drive the master reset signal asynchronously through the reset buffer tree to the rest of the flip-flops in the design. The entire design will be asynchronously reset.

Reset removal is accomplished by de-asserting the reset signal, which then permits the d-input of the first master reset flip-flop (which is tied high) to be clocked through a reset synchronizer. It typically takes two rising clock edges after reset removal to synchronize removal of the master reset.

Two flip-flops are required to synchronize the reset signal to the clock pulse where the second flip-flop is used to remove any metastability that might be caused by the reset signal being removed asynchronously and too close to the rising clock edge. As discussed in section 5.4, these synchronization flip-flops must be kept off of the scan chain.

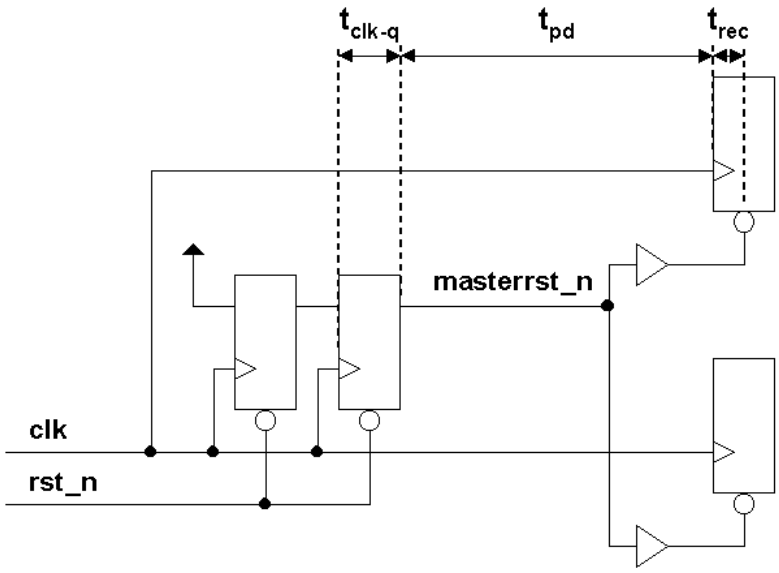


Figure 10 - Predictable reset removal to satisfy reset recovery time

A closer examination of the timing now shows that reset distribution timing is the sum of the a clk-to-q propagation delay, total delay through the reset distribution tree and meeting the reset recovery time of the destination registers and flip-flops, as shown in Figure 10.

The code for the reset synchronizer circuit is shown in Example 8.

```
module async_resetFFstyle2 (
  output reg rst_n,
  input      clk, asyncrst_n);
  reg        rff1;

  always @(posedge clk or negedge asyncrst_n)
    if (!asyncrst_n) {rst_n,rff1} <= 2'b0;
    else             {rst_n,rff1} <= {rff1,1'b1};
endmodule
```

Example 8a - Properly coded reset synchronizer using Verilog-2001

```
library ieee;
use ieee.std_logic_1164.all;
entity asyncresetFFstyle is
  port (
    clk        : in  std_logic;
    asyncrst_n : in  std_logic;
    rst_n      : out std_logic);
end asyncresetFFstyle;
```

```
architecture rtl of asyncresetFFstyle is
  signal rff1 : std_logic;
begin
  process (clk, asyncrst_n)
  begin
    if (asyncrst_n = '0') then
      rff1  <= '0';
      rst_n <= '0';
    elsif (clk'event and clk = '1') then
      rff1  <= '1';
      rst_n <= rff1;
    end if;
  end process;
end rtl;
```

Example 8b - Properly coded reset synchronizer using VHDL

7.1 Reset Synchronizer Metastability??

Ever since the publication of our first resets paper[4], we have received numerous email messages asking if the reset synchronizer has potential metastability problems on the second flip-flop when reset is removed. The answer is that the reset synchronizer DOES NOT have reset metastability problems. The analysis and discussion of related issues follows.

The first flip-flop of the reset synchronizer *does have* potential metastability problems because the input is tied high, the output has been asynchronously reset to a 0 and the reset could be removed within the specified reset recovery time of the flip-flop (the reset may go high too close to the rising edge of the clock input to the same flip-flop). This is why the second flip-flop is required.

The second flip-flop of the reset synchronizer *is not* subject to recovery time metastability because the input and output of the flip-flop are both low when reset is removed. There is no logic differential between the input and output of the flip-flop so there is no chance that the output would oscillate between two different logic values.

7.2 Erroneous ASIC Vendor Modeling

One engineer emailed to tell us that he had run simulations with four different ASIC libraries and that the flip-flop outputs of two of the ASIC libraries were going unknown during gate-level simulation when the reset was removed too close to the rising clock edge[44]. This is typically an ASIC library modeling problem. Some ASIC vendors make the mistake of applying a general recovery time specification without consideration of the input and output values being the same. When we asked the engineer to examine the transistor-level version of the model, he emailed back that the circuit was indeed not susceptible to metastability problems if the d-input was low when a reset recovery violation occurred; translation, the vendor had mistakenly applied a general reset recovery time to the flip-flop model.

7.3 Flawed Reset De-Metastabilization Circuit

One engineer suggested using the circuit in Figure 11 to remove metastability. The flip-flop in the circuit is an asynchronously reset flip-flop.

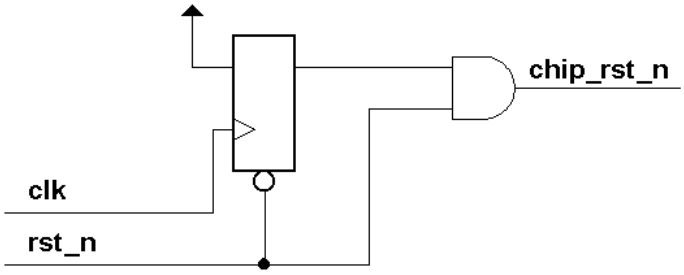


Figure 11 - Flawed reset synchronizer #1

Upon further query, the engineer reported that the output and-gate was used to remove metastability if reset is asserted too close to an active clock edge[28]. This is not necessary. There is no reset metastability issue when reset is asserted because the reset signal bypasses the clock signal in a flip-flop circuit to cleanly force the output low. The metastability issue is always related to reset removal.

This engineer handled reset recovery issues as a post place & route task. The reset delays would be measured and if necessary, a falling-clock flip-flop would be substituted for the flip-flop shown in Figure 11.

We are not convinced that this is a robust solution to the problem because min-max process variations may cause some reset circuits to fail if they have significantly different timing characteristics than the measured prototype device.

7.4 Simulation testing with resets

One EDA support engineer reported that design engineers are running simulations and releasing reset on the active edge of the clock. It should be noted that most of the time, this is a Verilog race condition and is almost always a real hardware race condition.

On real hardware, if the reset signal is removed coincident with a rising clock edge, the reset signal will violate the reset recovery time specification for the device and the output of the flip-flop could go metastable. This is another important reason why the reset synchronizer circuit described in section 7.0 is used for designs that include asynchronous reset logic.

In a simulation, if reset is removed on a posedge clock, there is usually no guarantee what the simulation result will be. Even if the RTL code behaves as expected, the gate-level simulation may behave differently due to event scheduling race conditions and different IEEE-Verilog compliant simulators may even yield different RTL simulation results. Most ASIC libraries will drive an X-output from the gate-level flip-flop simulation model when a reset recovery time violation occurs (typically modeled using a User Defined Primitive, or UDP for short).

Since one important goal related to testbench creation is to make sure that the same testbench can be used to verify the same results for both pre- and post-synthesis simulations, in our testbenches we always change the reset signal on the inactive clock edge, far away from any potential recovery time violation and simulation race condition.

Guideline: In general, change the testbench reset signal on the inactive clock edge using blocking assignments.

Another good testbench strategy is to assert reset at time 0 to initialize all resetable registers and flip-flops. Asserting reset at time zero could also cause a Verilog race condition but this race condition can be easily avoided by making the first testbench assignment to reset using a nonblocking assignment as shown in Example 9. Using a time-0 nonblocking assignment to reset causes the reset signal to be updated in the nonblocking update events region of the Verilog event queue at time 0, forcing all procedural blocks to become active before the reset signal is asserted, which means all reset-sensitive procedural blocks are guaranteed to trigger at time 0 (no Verilog race issues).

```
initial begin              // clock oscillator
  clk <= 0;                // time 0 nonblocking assignment
  forever #(`CYCLE/2) clk = ~clk;
end

initial begin
  rst_n <= 0;              // time 0 nonblocking assignment
  @(posedge clk);          // Wait to get past time 0
  @(negedge clk) rst = 1; // rst_n low for one clock cycle
  ...
end
```

Example 9 - Good coding style for time-0 reset assertion

One EDA tool support engineer who receives complaints about Verilog race conditions by engineers that release reset coincident with the active clock edge in their testbenches (as noted above, this is a real hardware race condition, a Verilog simulation race condition, and in our opinion a sign of a poorly trained Verilog engineer) recommended that design engineers avoid asynchronous-reset flip-flops to eliminate the potential Verilog race conditions related to reset removel. He then showed a typical asynchronous reset flip-flop model similar to the one shown in Example 10.

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

Example 10 - Typical coding style for flip-flops with asynchronous resets

He correctly noted that either the clk would go high while rst_n is low, causing q to be reset, or clk could go high after rst_n is released, causing q to be assigned the value of d.

We pointed out that synchronous reset flip-flops can experience the same non-deterministic simulation results for the exact same reason and that synchronous reset flip-flops do not change the fact that this would still be a real hardware problem. Conclusion: do not release reset coincident with the active clock edge of the design from a testbench. This might make a good interview question for design and verification engineers!

8.0 Reset distribution tree

The reset distribution tree requires almost as much attention as a clock distribution tree, because there are generally as many reset-input loads as there are clock-input loads in a typical digital design, as shown in Figure 12. The timing requirements for reset tree are common for both synchronous and asynchronous reset styles.

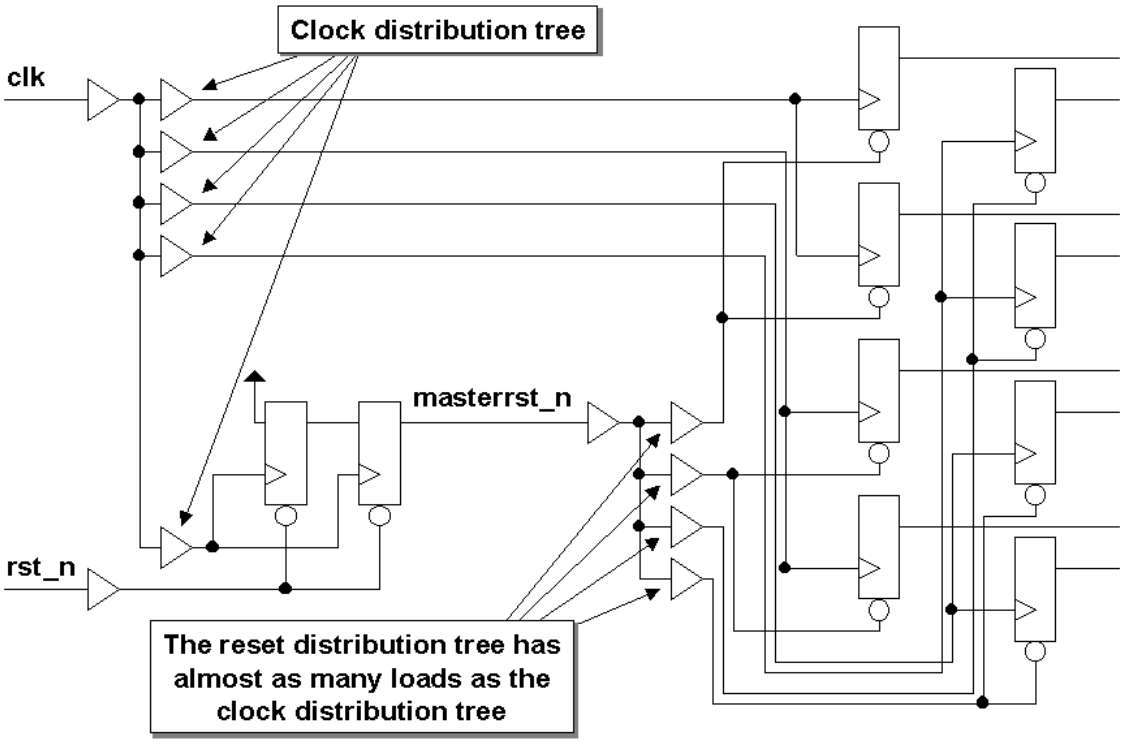


Figure 12 - Reset distribution tree

One important difference between a clock distribution tree and a reset distribution tree is the requirement to closely balance the skew between the distributed resets. Unlike clock signals, skew between reset signals is not critical as long as the delay associated with any reset signal is short enough to allow propagation to all reset loads within a clock period and still meet recovery time of all destination registers and flip-flops.

Care must be taken to analyze the clock tree timing against the clk-q-reset tree timing. The safest way to clock a reset tree (synchronous or asynchronous reset) is to clock the internal-master-reset flip-flop from a leaf-clock of the clock tree as shown in Figure 13. If this approach will meet timing, life is good. In most cases, there is not enough time to have a clock pulse traverse the clock tree, clock the reset-driving flip-flop and then have the reset traverse the reset tree, all within one clock period.

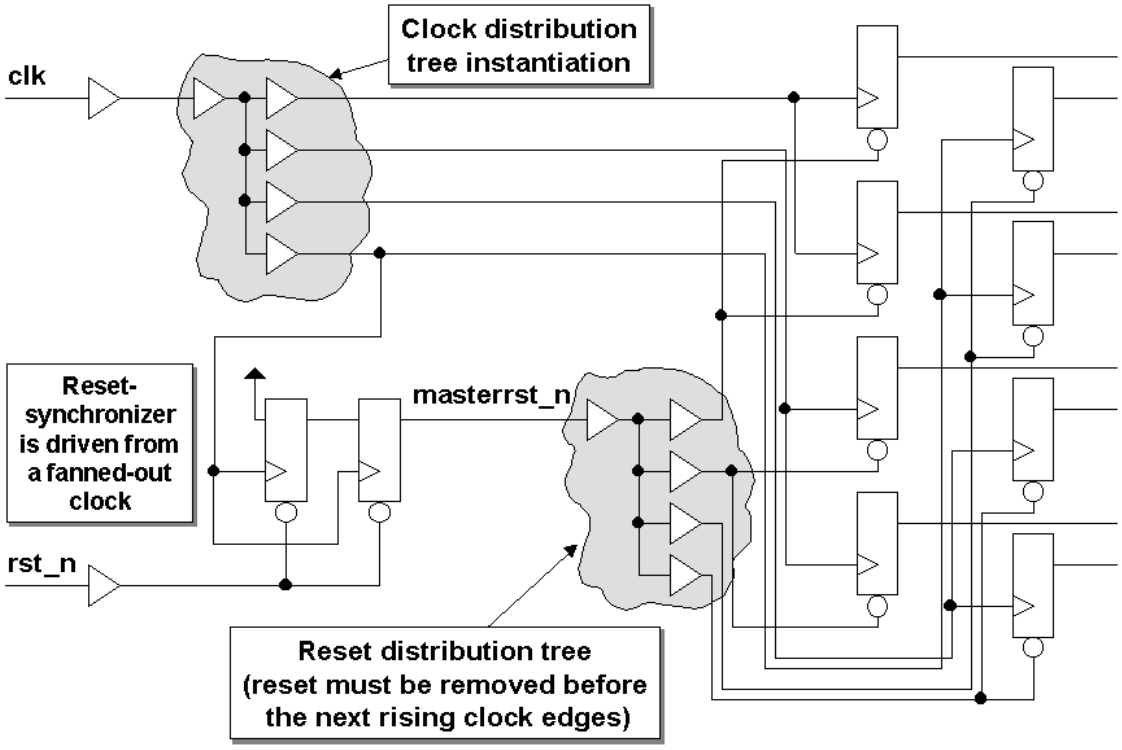


Figure 13 - Reset tree driven from a delayed, buffered clock

In order to help speed the reset arrival to all the system flip-flops, the reset-driver flip-flop is clocked with an early clock as shown in Figure 14. Post layout timing analysis must be made to ensure that the reset release for asynchronous resets and both the assertion and release for synchronous reset do not beat the clock to the flip-flops; meaning the reset must not violate setup and hold on the flops. Often detailed timing adjustments like this can not be made until the layout is done and real timing is available for the two trees.

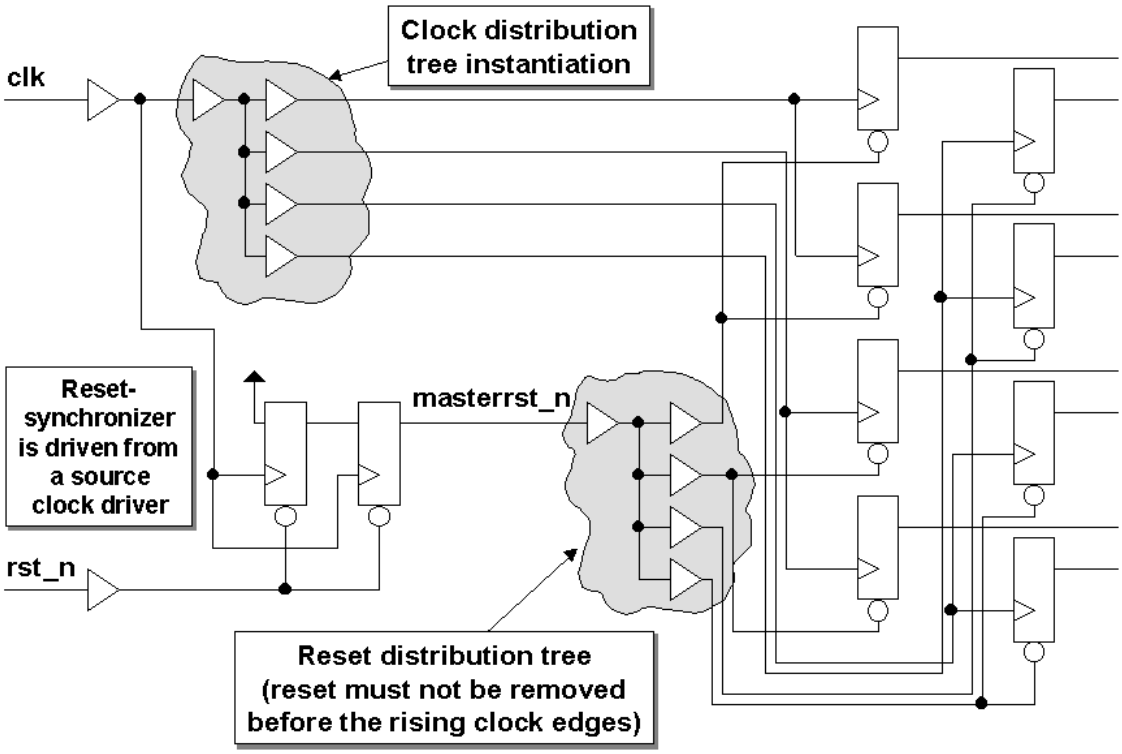


Figure 14 - Reset synchronizer driven in parallel to the clock distribution tree

Ignoring this problem will not make it go away. Gee, and we all thought resets were such a basic topic.

8.1 Synchronous reset distribution technique

For synchronous resets, one technique is to build a distributed reset buffer tree with flops embedded in the tree (see Figure 15). This keeps the timing requirements fairly simple, because you don't have to reach every flip-flop in one clock period. In each module, the reset input to the module is run through a simple D-flip-flop, and then this delayed reset is used to reset logic inside the module *and* to drive the reset input of any submodules. Thus it may take several clocks for all flip-flops in the design to be reset (Note: similar problems are seen with multi-clock designs where the reset signal must cross clock domains). Thus each module would contain code such as

```
input reset_raw;

// synopsys sync_set_reset "reset"
always @ (posedge clk) reset <= reset_raw;
```

where **`reset`** is used to synchronously reset all logic within the enclosed module, and is also connected to the **`reset_raw`** port of any submodules.

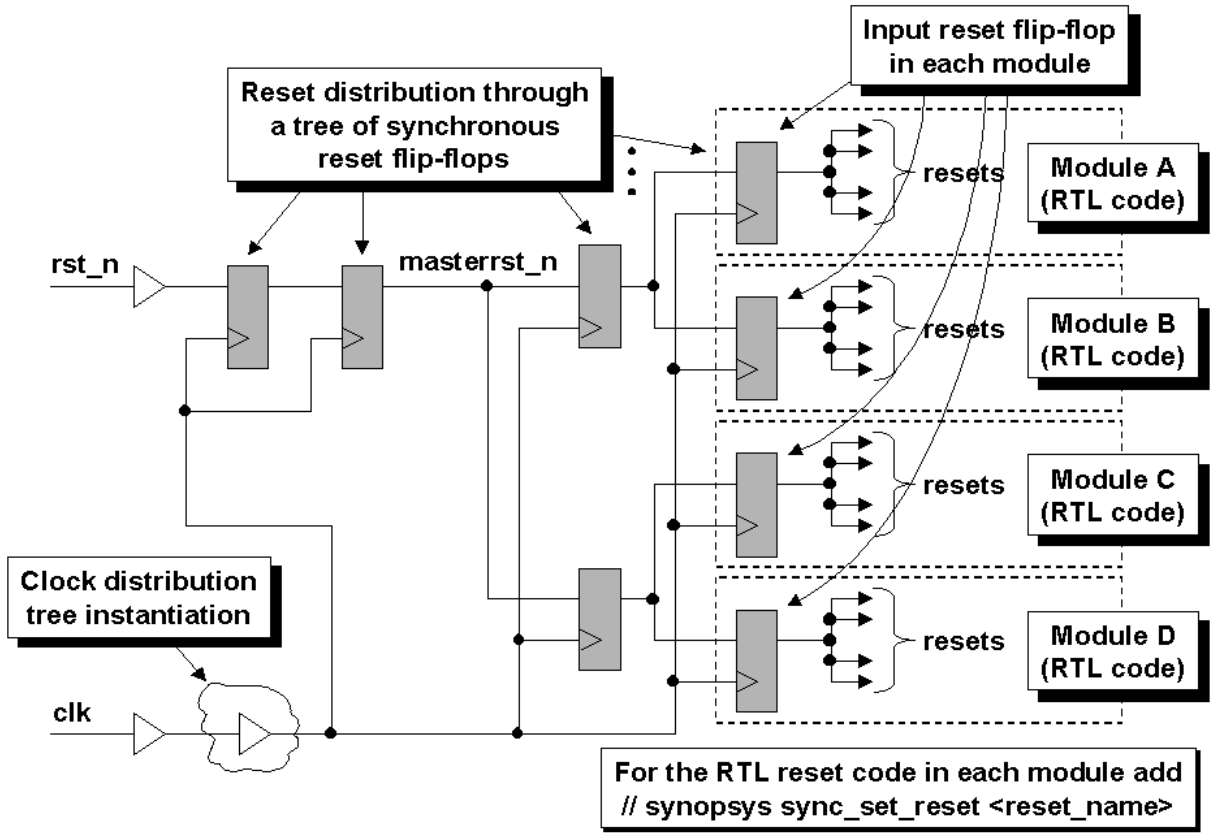


Figure 15 - Synchronous reset distribution method using distributed synchronous flip-flops

With such a technique the synchronous reset signal can be treated like any other data signal, with easy timing analysis for every module in the design, and reasonable fanouts at any stage of the reset tree.

8.2 Asynchronous reset distribution technique

For asynchronous resets, an interesting technique is to again use a distributed asynchronous reset synchronizer scheme, similar to the reset tree described in section 8.1, to replace the reset buffer tree (see Figure 16).

This approach for asynchronous resets places reset synchronizers at every level of hierarchy of the design. This is the same approach as distributing synchronous reset flip-flops as discussed in section 8.1. The difference, is that there are two flip-flops per reset synchronizer at each level instead of one flip-flop used for the synchronous reset approach. The local reset drives the asynchronous reset inputs to local flip-flops instead of being gated into the data path as done with the synchronous reset technique.

This method of distributed reset synchronizers will reset the same as having one reset synchronizer at the top level, in that the design will asynchronously reset when reset is applied and will be synchronously released from the reset. However, the design will be released from

reset over a number of clock cycles as the release of reset trickles through the hierarchical reset tree.

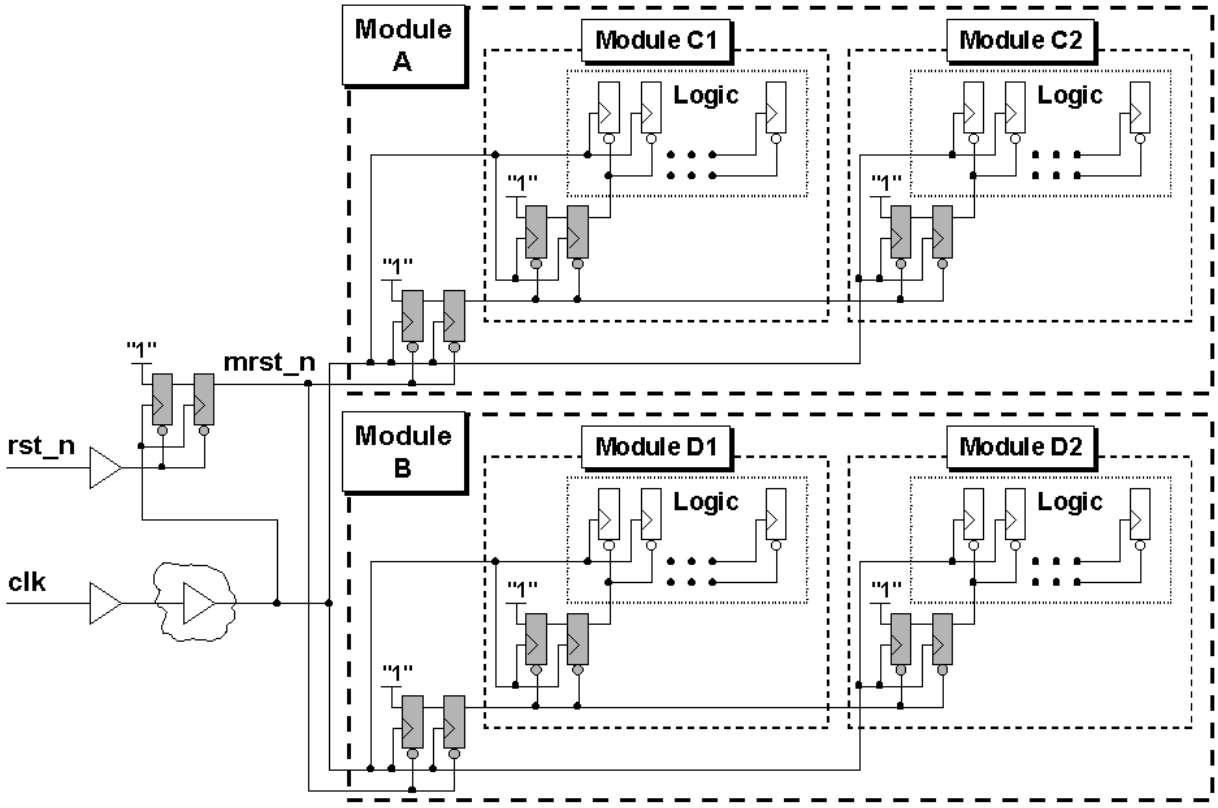


Figure 16 - Asynchronous reset distribution method using distributed reset synchronizers

Note that using this technique, the whole design may not come out of reset at the same time (within the same clock cycle). Whether or not this is a problem is design dependent. Most designs can deal with the release of reset across many clocks. If the design functionality is such that the whole design must come out of reset within the same clock cycle, then the reset tree of reset synchronizers must be balanced at all end points. This is true for both synchronous and asynchronous resets.

Section 8.0 discussed details about buffering the global asynchronous reset tree. The biggest problem with this approach is the timing verification of the reset tree to ensure that the release of the reset occurs within one clock period. Preliminary analysis can be done prior to place and route, but the reset tree from section 8.0 must be analyzed after place & route (P&R). Unfortunately, if timing adjustments are required, the designer most often must make these adjustments by hand in the P&R domain and then re-time the routed design, repeating this process until the timing requirements are met. The approach discussed in this section using the distributed reset synchronizers removes the backend manual adjustments and will allow the synthesis tools to do the job of timing and buffering the design automatically. Using this distribution technique, the reset buffering is completely local to the current level (the same as with the synchronous approach discussed in section 8.1).

When using asynchronous resets, it is vitally important that the designer uses the proper variables set to the proper settings in both DC and PT to ensure that the asynchronous reset driven from the q-output of the reset synchronizing flip-flops are buffered (if needed) and timed. Details on these settings can be found in SolvNet article #901989[43]. The article states, both DC and PT can and will time to the asynchronous reset input against the local clock if the following variables are set:

```
pt_shell> set timing_disable_recovery_removal_checks "false"
dc_shell> enable_recovery_removal_arcs "true"
```

These settings should be the default setting from Synopsys (just make sure they are set for your environment). With these flags set correctly, and the distributed reset synchronizers, the clock-tree-like task of building a buffered reset tree is eliminated.

If you are designing FPGAs, then the reset synchronizer distribution method discussed in this section may be preferred[30]. There are two good reasons this may be true: (1) The Global Set/Reset (GSR) buffer on most FPGAs does both asynchronous reset *and asynchronous reset removal* with all of the associated problems related to asynchronous reset removal already detailed in this paper. Unless an FPGA vendor has implemented a reset synchronizer on-chip, the engineer will need to implement an off-chip asynchronous reset synchronizer and the inter-chip pin-pad delays may be too slow to effectively implement. (2) It is not unusual to have multiple clock buffers with multiple clock domains but only one GSR buffer and each clock domain should control a corresponding reset synchronizer (discussed in section 11.0).

There is also a good reason to consider using asynchronous resets instead of synchronous resets in an FPGA device. In general, FPGAs have an abundance of flip-flops, but FPGA design speed is often limited by the size of the combinational blocks required for the design. If a block of combinational logic does not fit into a single cell of FPGA lookup tables, the combinational logic must be continued in additional lookup tables with corresponding lookup delays and inter-cell routing delays. The use of synchronous resets typically requires at least part of a lookup table that might be needed by a combinational logic block.

And finally, DFT for FPGAs is a non-issue since FPGA designs do not include DFT internal scan, thus the issues regarding DFT with asynchronous resets on an FPGA do not exist.

9.0 Reset-glitch filtering

As stated earlier in this paper, one of the biggest issues with asynchronous resets is that they are asynchronous and therefore carry with them some characteristics that must be dealt with depending on the source of the reset. With asynchronous resets, any input wide enough to meet the minimum reset pulse width for a flip-flop will cause the flip-flop to reset. If the reset line is subject to glitching, this can be a real problem. Presented here is one approach that will work to filter out the glitches, but it is ugly! This solution requires that a digital delay (meaning the delay will vary with temperature, voltage and process) to filter out small glitches. The reset input pad should also be a Schmidt triggered pad to help with glitch filtering. Figure 17 shows the implementation of this approach.

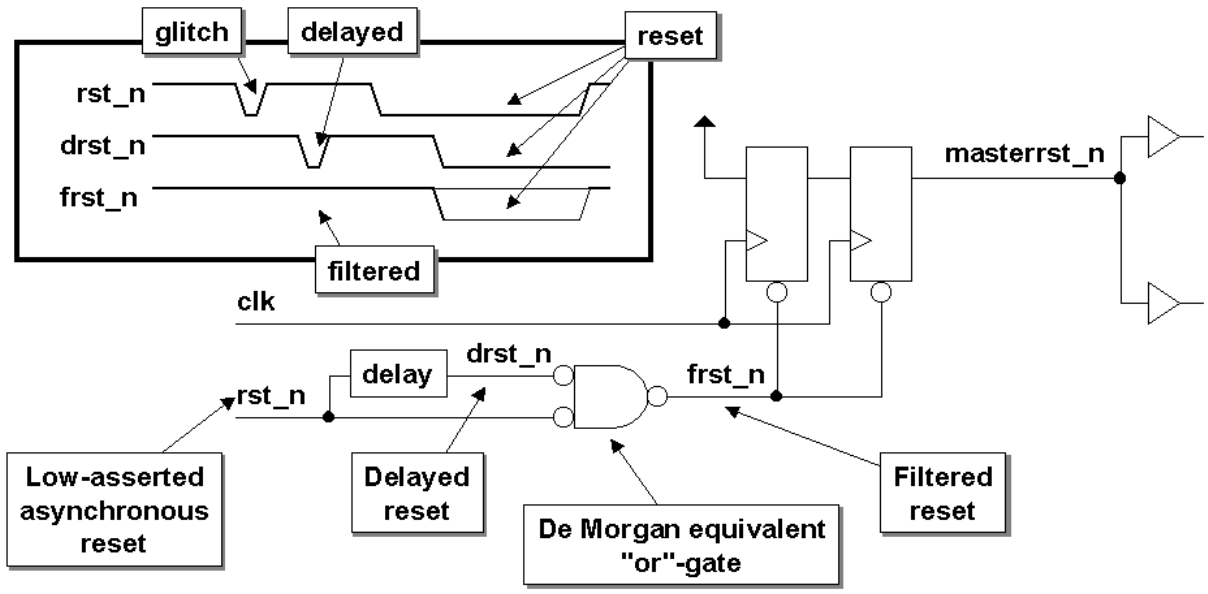


Figure 17 - Reset glitch filtering

In order to add the delay, some vendors provide a delay hard macro that can be hand instantiated. If such a delay macro is not available, the designer could manually instantiate the delay into the synthesized design after optimization – remember not to optimize this block after the delay has been inserted or it will be removed. Of course the elements could have don't touch attributes applied to prevent them from being removed. A second approach is to instantiated a slow buffer in a module and then instantiated that module multiple times to get the desired delay. Many variations could expand on this concept.

This glitch filter is not needed in all systems. The designer must research the system requirements to determine whether or not a delay is needed.

10.0 DFT for asynchronous resets

One important issue related to asynchronous resets has to do with the use of Design For Test (DFT). Engineers have commented that the toughest part about using asynchronous resets in a design is related to DFT[26] while other engineers that are using DFT with asynchronous resets claim it is not difficult[6]. If an asynchronous reset is being gated and used as an active functional input, DFT becomes difficult.

If the asynchronous reset is used as part of the functional design, then all the comments in ESNUG #409 item 11[26] regarding DFT difficulties are correct. DFT will be hard, if not impossible. The functional design is no longer following the base design guidelines for synchronous design that the synthesis and timing analysis tools require for accurate and correct results. The guidelines recommended in this paper with regards to asynchronous reset are based on the reset being only an initialization reset and that reset is not part of the functionality of the

device. The only logic in the reset path would be the reset synchronizers. If this design approach is used, then the DFT approach discussed below provides a very simple and thorough approach to DFT for asynchronous reset.

Applying Design for Test (DFT) functionality to a design is a two step process. First, the flips-flops in the design are stitched together into a scan chain accessible from external I/O pins, this is called scan insertion. The scan chain is typically not part of the functional design. Second, a software program is run to generate a set of scan vectors that, when applied to the scan chain, will test and verify the design. This software program is called Automatic Test Program Generation or ATPG. The primary objective of the scan vectors is to provide foundry vectors for manufacture tests of the wafers and die as well as tests for the final packaged part.

The process of applying the ATPG vectors to create a test is based on:

1. scanning a known state into all the flip-flops in the chip,
2. switching the flip-flops from scan shift mode, to functional data input mode,
3. applying one functional clock,
4. switching the flip-flops back to scan shift mode to scan out the result of the one functional clock while scanning in the next test vector.

The DFT process usually requires two control pins. One that puts the design into “test mode.” This pin is used to mask off non-testable logic such as internally generated asynchronous resets, asynchronous combinational feedback loops, and many other logic conditions that require special attention. This pin is usually held constant during the entire test. The second control pin is the shift enable pin.

In order for the ATPG vectors to work, the test program must be able to control all the inputs to the flip-flops on the scan chain in the chip. This includes not only the clock and data, but also the reset pin (synchronous or asynchronous). If the reset is driven directly from an I/O pin, then the reset is held in a non-reset state. If the reset is internally generated, then the master internal reset is held in a non-reset state by the test mode signal. If the internally generated reset were not masked off during ATPG, then the reset condition might occur during scan causing the flip-flops in the chip to be reset, and thus lose the vector data being scanned in.

Even though the asynchronous reset is held to the non-reset state for ATPG, this does not mean that the reset/set cannot be tested as part of the DFT process. Before locking out the reset with test mode and generating the ATPG vectors, a few vectors can be manually generated to create reset/set test vectors. The process required to test asynchronous resets for DFT is very straight forward and may be automatic with some DFT tools. If the scan tool does not automatic test the asynchronous resets/sets, then they must be setup manually. The basic steps to manually test the asynchronous resets/sets are as follows:

1. scan in all ones into the scan chain
2. issue and release the asynchronous reset
3. scan out the result and scan in all zeros
4. issue and release the reset
5. scan out the result
6. set the reset input to the non reset state and then apply the ATPG generated vectors.

This test approach will scan test for both asynchronous resets and sets. These manually generated vectors will be added to the ATPG vectors to provide a higher fault coverage for the manufacture test. If the design uses flip-flops with synchronous reset inputs, then modifying the

above manual asynchronous reset test slightly will give a similar test for the synchronous reset environment. Add to the steps above a functional clock while the reset is applied. All other steps would remain the same.

For the reset synchronizer circuit discussed in this paper, the two synchronizer flips-flops should not be included in the scan chain, but should be tested using the manual process discussed above.

11.0 Multi-clock reset issues

For a multi-clock design, a separate asynchronous reset synchronizer circuit and reset distribution tree should be used for each clock domain. This is done to insure that reset signals can indeed be guaranteed to meet the reset recovery time for each register in each clock domain.

As discussed earlier, asynchronous reset assertion is not a problem. The problem is graceful removal of reset and synchronized startup of all logic after reset is removed.

Depending on the constraints of the design, there are two techniques that could be employed: (1) non-coordinated reset removal, and (2) sequenced coordination of reset removal.

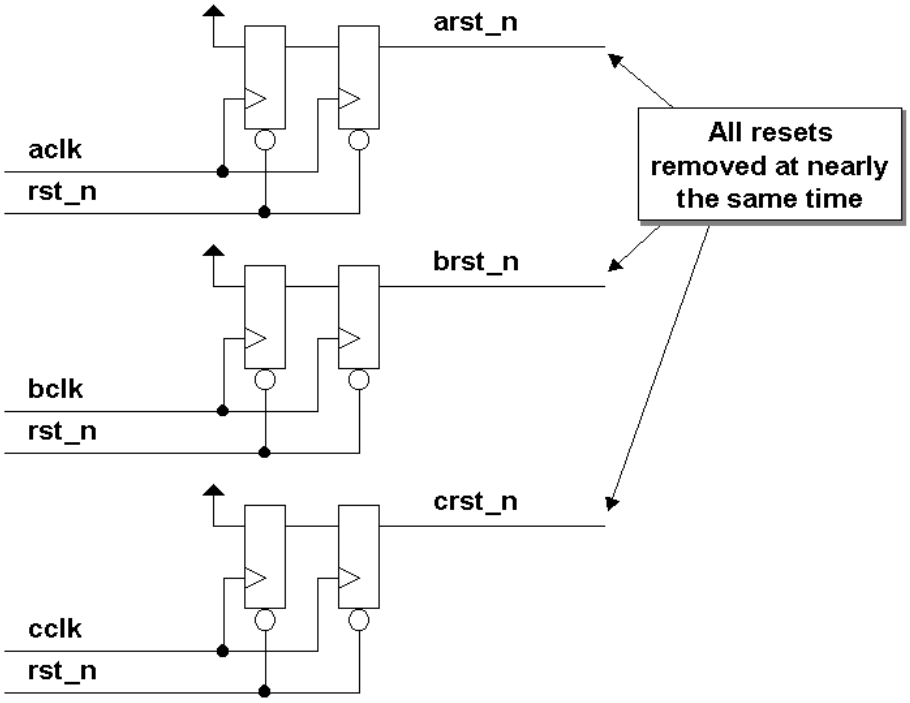


Figure 18 - Multi-clock reset removal

11.1 Non-coordinated reset removal

For many multi-clock designs, exactly when reset is removed within one clock domain compared to when it is removed in another clock domain is not important. Typically in these designs, any control signals crossing clock boundaries are passed through some type of request-acknowledge

handshaking sequence and the delayed acknowledge from one clock domain to another is not going to cause invalid execution of the hardware. For this type of design, creating separate asynchronous reset synchronizers as shown in Figure 18 is sufficient, and the fact that **`arst_n`**, **`brst_n`** and **`crst_n`** could be removed in any sequence is not important to the design.

11.2 Sequenced coordination of reset removal

For some multi-clock designs, reset removal must be ordered and proper sequence. For this type of design, creating prioritized asynchronous reset synchronizers as shown in Figure 19 might be required to insure that all **`aclk`** domain logic is activated after reset is removed before the **`bclk`** logic, which must also be activated before the **`cclk`** logic becomes active.

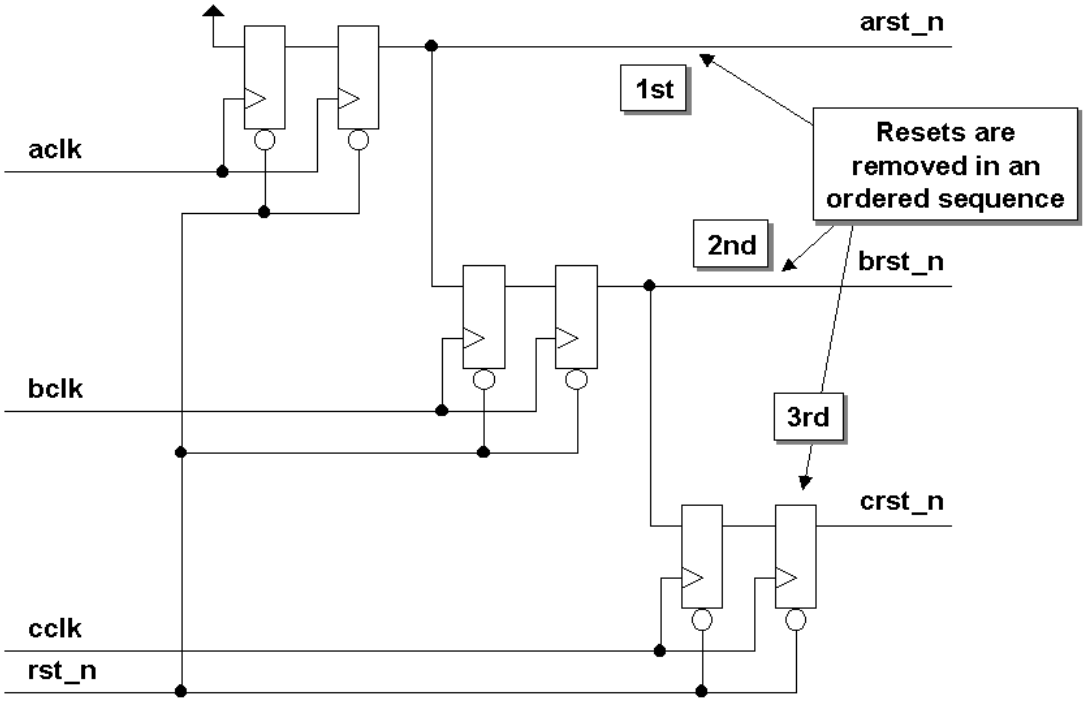


Figure 19 - Multi-clock ordered reset removal

For this type of design, only the highest priority asynchronous reset synchronizer input is tied high. The other asynchronous reset synchronizer inputs are tied to the master resets from higher priority clock domains.

12.0 Conclusions

Properly used, synchronous and asynchronous resets can each guarantee reliable reset assertion. Although an asynchronous reset is a safe way to reliably reset circuitry, removal of an asynchronous reset can cause significant problems if not done properly.

The proper way to design with asynchronous resets is to add the reset synchronizer logic to allow asynchronous reset of the design and to insure synchronous reset removal to permit safe restoration of normal design functionality.

Using DFT with asynchronous resets is still achievable as long as the asynchronous reset can be controlled during test.

Whether the design uses synchronous or asynchronous resets, using one of the distributed flip-flop trees as described in this paper may be worthy of consideration by the designer since they remove many of the issues related to buffering, timing and layout of a reset tree.

In conclusion, simple little resets ... aren't!

References

[1] *ALS/AS Logic Data Book*, Texas Instruments, 1986, pg. 2-78.

[2] Chris Kiegle, personal communication

[3] Clifford E. Cummings, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill!," *SNUG (Synopsys Users Group) 2000 User Papers*, section-MC1 (1st paper), March 2000. Also available at www.sunburst-design.com/papers

[4] Clifford E. Cummings and Don Mills, "Synchronous Resets? Asynchronous Resets? I am so confused! How will I ever know which to use?" *SNUG (Synopsys Users Group) San Jose, 2002 User Papers*, March 2002. Also available at www.sunburst-design.com/papers and www.lcdm-eng.com/papers.htm

[5] Don Mills and Clifford E. Cummings, "RTL Coding Styles That Yield Simulation and Synthesis Mismatches," *SNUG (Synopsys Users Group) 1999 Proceedings*, section-TA2 (2nd paper), March 1999. Also available at www.lcdm-eng.com/papers.htm and www.sunburst-design.com/papers

[6] Erick Pew, personal communication

[7] ESNUG #60, Item 1- www.deepchip.com/items/0060-01.html

[8] ESNUG #240, Item 7- www.deepchip.com/items/0240-07.html

[9] ESNUG #242, Item 6 - www.deepchip.com/items/0242-06.html

[10] ESNUG #243, Item 4 - www.deepchip.com/items/0243-04.html

[11] ESNUG #244, Item 5 - www.deepchip.com/items/0244-05.html

[12] ESNUG #246, Item 5 - www.deepchip.com/items/0246-05.html

[13] ESNUG #278, Item 7 - www.deepchip.com/items/0278-07.html

[14] ESNUG #280, Item 4 - www.deepchip.com/items/0280-04.html

[15] ESNUG #281, Item 2 - www.deepchip.com/items/0281-02.html

[16] ESNUG #355, Item 2 - www.deepchip.com/items/0355-02.html

[17] ESNUG #356, Item 4 - www.deepchip.com/items/0356-04.html

[18] ESNUG #373, Item 6 - www.deepchip.com/items/0373-06.html

[19] ESNUG #375, Item 14 - www.deepchip.com/items/0375-14.html

[20] ESNUG #379, Item 14 - www.deepchip.com/items/0379-14.html

[21] ESNUG #380, Item 13 - www.deepchip.com/items/0380-13.html

[22] ESNUG #381, Item 13 - www.deepchip.com/items/0381-13.html

[23] ESNUG #393 Item 1 - www.deepchip.com/items/0393-01.html

[24] ESNUG #396, Item 1 - www.deepchip.com/items/0396-01.html

[25] ESNUG #404, Item 15 - www.deepchip.com/items/0404-15.html

[26] ESNUG #409, Item 11 - www.deepchip.com/items/0409-11.html

[27] ESNUG #410, Item 3 - www.deepchip.com/items/0410-03.html

[28] Gzim Derti, personal communication

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

[30] Ken Chapman, "Get Smart About Reset (Think Local, Not Global)," Xilinx TechXclusives, downloaded from www.xilinx.com/support/techxclusives/global-techX19.htm

[31] Lee Tatistcheff, personal communication

[32] Michael Keating, and Pierre Bricaud, *Reuse Methodology Manual*, Second Edition, Kluwer Academic Publishers, 1999, pg. 35.

[33] Synopsys SolvNet, Doc Name: 902298, "Recovery and Removal timing checks on Primetime," Updated 02/13/2002 - solvnet.synopsys.com/retrieve/902298.html

[34] Synopsys SolvNet, Doc Name: 903391, "Methodology and limitations of synthesis for synchronous set and reset," Updated 09/07/2001 - solvnet.synopsys.com/retrieve/903391.html

[35] Synopsys SolvNet, Doc Name: 900214, "Handling High Fanout Nets in 2001.08" Updated: 11/01/2001 - solvnet.synopsys.com/retrieve/900214.html

[36] Synopsys SolvNet, Doc Name: 004186, "Reset Pros and Cons" Updated: 03/10/2003 - solvnet.synopsys.com/retrieve/004186.html

[37] Synopsys SolvNet, Doc Name: 901644, "Multiple Synchronous Resets" Updated: 09/07/2001 - solvnet.synopsys.com/retrieve/901644.html

[38] Synopsys SolvNet, Doc Name: 901093, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 09/07/2001 - solvnet.synopsys.com/retrieve/901093.html

[39] Synopsys SolvNet, Doc Name: 902448, "Is the compile_preserve_sync_reset Switch Still Valid?," Updated: 05/22/1998 - solvnet.synopsys.com/retrieve/902448.html (this article and the previous reference have the same name but are different articles.)

[40] Synopsys SolvNet, Doc Name: 901811, "Why can't I synthesize synchronous reset flip-flops?," Updated: 08/16/1999 - solvnet.synopsys.com/retrieve/901811.html

[41] Synopsys SolvNet, Doc Name: 901241, "Commands for High Fanout Nets: high_fanout_net_threshold and report_high_fanout" Updated: 01/31/2003 - solvnet.synopsys.com/retrieve/901241.html

[42] Synopsys SolvNet, Doc Name: 901264, "Data and Synchronous Reset Swapped," Updated: 06/18/2003 - solvnet.synopsys.com/retrieve/901264.html

[43] Synopsys SolvNet, Doc Name: 901989, "Default Settings for Recovery/Removal Arcs in PrimeTime and Design Compiler," Updated: 01/12/1999 - solvnet.synopsys.com/retrieve/901989.html

[44] Zeeshan Sarwar, personal communication

Revision 1.2 (September 2003) - What Changed?

Figure 15 - Synchronous reset distribution method using distributed synchronous flip-flops and Figure 16 - Asynchronous reset distribution method using distributed reset synchronizers were added to the paper to help show how the described reset distribution methods are done.

Revision 1.3 (July 2004) - What Changed?

The example 7b code incorrectly showed a VHDL example with synchronous reset. The code has been corrected to show asynchronous resets. Multiple readers pointed out the problem but the update was just made.

Author & Contact Information

Cliff Cummings, President of Sunburst Design, Inc., is an independent EDA consultant and trainer with 21 years of ASIC, FPGA and system design experience and 11 years of Verilog, synthesis and methodology training experience.

Mr. Cummings, a member of the IEEE 1364 Verilog Standards Group (VSG) since 1994, is the only Verilog and SystemVerilog trainer to co-develop and co-author the IEEE 1364-1995 & IEEE 1364-2001 Verilog Standards, the IEEE 1364.1-2002 Verilog RTL Synthesis Standard and the Accellera SystemVerilog 3.0 & 3.1 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

Don Mills is an independent EDA consultant, ASIC designer, and Verilog/VHDL trainer with 17 years of experience.

Don has inflicted pain on Aart De Geuss for too many years as SNUG Technical Chair. Aart was more than happy to see him leave! Not really, Don chaired three San Jose SNUG conferences: 1998-2000, the first Boston SNUG 1999, and three Europe SNUG conferences 2001- 2003.

Don holds a BSEE from Brigham Young University.

E-mail address: mills@lcdm-eng.com

Steve Golson designed his first IC in 1982 in 4-micron NMOS. Since 1986 he has provided contract engineering services in VLSI design (full custom, semi-custom, gate array, FPGA); computer architecture and memory systems; and digital hardware design. He has extensive experience developing synthesis methodologies for large ASICs using a variety of design tools including Verilog and Synopsys. Other services include Synopsys and Verilog training classes, patent infringement analysis, reverse engineering, and expert witness testimony.

Steve holds a BS in Earth, Atmospheric, and Planetary Science from Massachusetts Institute of Technology. Hey, if a seismologist can design ASICs, it can't be that hard!

E-mail address: sgolson@trilobyte.com

An updated version of this paper can be downloaded from the web sites: www.sunburst-design.com/papers, www.lcdm-eng.com or from www.trilobyte.com

(Data accurate as of August 12, 2003)