Synthesizable Watchdog Logic: A Key Coding Strategy for Managing Complex Designs

Duane Galbi
Conexant Systems, Inc.
duaneg@iname.com

Wilson Snyder
Conexant Systems, Inc

ABSTRACT

As HDL models become larger, the growth of module interfaces and the participation of multiple designers complicate the verification effort. Adding watchdog logic within an HDL model is a key tool for managing and verifying module interfaces, as well as a key tool for allowing one to more safely make changes to logic which they do not fully understand. A common approach is to treat watchdog logic as nonsynthesizable code and to include "//synopsys translate_off/on" directives around it. However, since this directive affects only the synthesis results, misplacing it can easily cause a designer's worst nightmare – a mismatch between pre and post synthesis simulation. This paper will detail how, by utilizing Design Compiler's extensive ability to optimize away unused logic, the "translate_off" directive is almost never needed. In addition, using freely available Verilog preprocessing software, complex watchdog logic can be disguised to appear like a small set of Verilog system tasks. This approach makes it easy to include watchdog logic, and it allows this logic to be automatically handled in a manner compatible with Verilog lint and coverage checking tools.

1.0 Introduction

In order to increase one's productivity when writing HDL code, one typically uses specialized tools such as graphical interfaces [1] or specialized language sensitive Emacs modes and macros [2]. However, as the code is being written, one typically does not take into account trying to increase verification productivity. Even though tools such as coverage analysis tools are, by necessity, run on finished HDL code, it is a design oversight to not consider verification as the code is being written. Verification productivity not design productivity is what gets a design to market quickly, for it is a generally accepted fact that the verification of HDL code takes longer, much longer at times, than its initial writing.

Hence, as HDL designs grow in size, coding in a manner that reduces the verification complexity of design can become a key factor in reducing the design cycle time. In order to manage the growing amount of HDL code required by today's designs, an HDL designer needs to start thinking more like a software developer. In order to partition the design into smaller pieces, most designers use the 1960's software technique of modular design. It is no doubt a very fundamental and effective strategy, but the state of art in software development has moved forward over the last 40 years, and some of it is applicable to today's HDL designer.

The software concept of "Design by Contract" [3][4][5] is an attempt to address the verification issue directly at the initial stage of coding. Full implementation of the "Design by Contract" approach requires pre-conditioning, post-conditioning, and invariants which in essence allow one to generate a "contact" on the affects of the routine. A full implementation of this approach is not common; however, the general approach of using assertion checks to verify that certain conditions remain valid has become a very common software technique. For instance, "C" has a standard assert() library function for this purpose.

This approach not only helps the initial debug of the code, but it also enhances the reusability of partial pieces of the code. Using assertion checks to verify key input, output, and internal conditions serves to document key requirements of the code. If the code is used in a different environment, or modified so that these requirements no longer exist, these assertion checks can unambiguously notify the user/designer of this condition.

An HDL designer can enjoy the same vital benefits that a software designer does by including assertion checks in the HDL code they generate. In relating to HDL code, *"assertion checks"* are often referred to as *"watchdog logic"*. We will use the terms *"assertion check"* and *"watchdog logic"* interchangeably in the rest of the paper. In both these cases, no real hardware logic is intended to be built, and the *"watchdog logic"* is only intended to be a software modeling artifact.

2.0 Problems with including Watchdog Logic

One problem with getting people to include “watchdog logic” as they create HDL code is that adding these statements is often viewed as too time consuming. People unfamiliar with adding “watchdog logic” often view time spent writing “watchdog logic” as time that could be better spent writing HDL code. However, this is false economy, for the code is not really done until is it debugged. In our experience, the time required to create “watchdog logic”, can easily result in a corresponding greater reduction in the amount of time required to debug the HDL code.

However, given the fact that HDL code runs through various software tools for simulation, lint checking, coverage checking, and synthesis, adding “watchdog logic” is often not as simple as just adding a few $display() statements to one’s Verilog code. If coded by hand, adding “watchdog logic” can easily require one to add Synopsys, lint, and coverage checking directives to one’s code. Furthermore, since one often does not want “watchdog logic” to trigger until the design leaves reset, it can also require simulation directives as well. Adding all these directives to simple “watchdog logic” is time consuming and can easily clutter one’s HDL beyond the point of readability. Fortunately, there are better ways to include “watchdog logic”.

One approach is to include all the checking logic in a non-synthesized module. This approach removes the need to address all the non-simulation issues related to the “watchdog logic”. The main drawback is it requires the designer to maintain another module, and removes the benefits of proximity between the check logic and what is being checked. Because the checker and design are in effect two separate pieces, it is easy for them to get separated, and for the design to get instantiated without including the checker logic. In total, a “watchdog logic” only module can require a fair amount of effort to create and maintain.

Fortunately there is a better way.

We have found that including the “watchdog logic” directly in the HDL module being created is the best way to make creating this logic easy and transparent to the design process. Coupling Design Compiler’s aggressive optimizing away of unessential logic with a simple Verilog preprocessor, enables this to happen without the designer needing to worry about adding directives for any of the various downstream software tools being used. This approach will be the focus of the rest of the paper.

3.0 Assertion Macros Disguised as System Calls

We have found that the best way to add assertion checks to a design module is to disguise the assertion check macros as Verilog system calls. Left unexpanded, these "system calls" are ignored by Synopsys, lint checking tools, and coverage checking tools. However, before simulation, the design modules are passed through a simple preprocessor (as example is included at http://www.ultranet.com/~wsnyder/veripool) which expands these system calls into the appropriate Verilog required by each assertion check. In order to kept the module's absolute line numbers unchanged, each "system call" is expanded with all the Verilog statements it contains placed on a single line. Hence, the number of the lines in the file does not change when the assertion check "system calls" are expanded.

Over the years, we have found that there are only a few key assertion check macros which are needed to make writing assertion checks easy. The following five assertion check "system calls" are the ones we have found the most useful:

Example 1 – Five Key Assertion Check Macros

Assertion System Macro	**Definition and Example Use**
$assert(<condition>,<msg>);	Check condition is true. $assert(!(rd1 && rd2),"Multiple Reads\n");
$assert_onehot([<variables>],<msg>);	Check variable or variable list is one hot. $assert_onehot(sel_a,sel_b,"mux selects\n"); $assert_onehot(state_r[7:0],"State_r not one-hot\n");
$assert_amone([<variables>],<msg>);	Check variable or variable list contains at most one valid logical condition. $assert_amone(gnt_a,gnt_b,"Multiple grants active\n");
$error(<level>,<msg>);	Print out an error message and stop simulation. $error(0,"Bad counter value=%x\n",count);
$info(<level>,<msg>);	Print out an information message and continue simulation. (message only printed in message level for the module is less that or equal to the <level> in the $info() statement) $info(1,"Reading bank-%x\n",mbank_r);

In order to keep these macros from triggering due to a transient change in a variable value, they are typically placed within an "@posedge" always block.

As mentioned earlier, before simulation, the assertion check "system macros" are replaced with appropriate Verilog code by passing all the design modules though a simple Verilog preprocessor. For example, the "$assert" system macro is expanded as follows (the full macro is really placed on one file line, but for readability we have broken it into multiple lines):

Example 2 – Expansion of $assert(!(rd1 && rd2),"Multiple Reads\n");

```
/*vpm*/begin
if ((!(rd1 && rd2))==0 && `c_esim_subrs.__message_on!=0) begin
$write("[%0t] %%E:%stest2.v:0070          : Multiple Reads\n %%E: In %m\n",
       $time,idm._id_ascii);
`pli.errors = `pli.errors+1;
end end/*vpm*/
```

If the condition in the "assert" macro is false, and the global flag enabling messages is active, then the appropriate message along with the simulation time, module file name, module line number, and module instance name are displayed. In order to make debugging easier, we do not terminate the simulation immediately; rather, we have the simulation set up to terminate a few cycles after an assertion macro evaluates false.

The expansion of the "$assert_onehot" system macro is slightly more complicated. The exact expansion of the macro depends on the number of variables being checked to make sure they are one-hot. An error message is printed if multiple bits being checked are active or if none of the bits being checked are active. An example expansion of "$assert_onehot" is as follows (again, for improved readability, we have artificially spit the macro into multiple lines):

Example 3 – Expansion of $assert_onehot(d1,d2,d3,"mux selects\n");

```
/*vpm*/begin
case ({d1,d2,d3})
  3'b100, 3'b010, 3'b001, 3'bXXX: ; 3'b000:
     if (`c_esim_subrs.__message_on!=0) begin
       $write("[%0t] %%E:%stest2.v:0080      : NONE ACTIVE (%x) --> ",
              $time,idm._id_ascii,({d1,d2,d3}));
       $write("mux selects\n");
       `pli.errors = `pli.errors+1; end
  default:
     if (`c_esim_subrs.__message_on!=0) begin
       $write("[%0t] %%E:%stest2.v:0080      : MULTIPLE ACTIVE (%x) --> ",
              $time,idm._id_ascii, ({d1,d2,d3}));
       $write("mux selects\n");
       `pli.errors = `pli.errors+1;
     end
endcase end/*vpm*
```

All the arguments to "$assert_onehot" before the first quoted argument are assumed to be arguments one wants to check to make sure they are one-hot. All the arguments after the first quoted argument are assumed to be arguments to pass to the $write() system call. Full vectors can be checked to make sure they are one-hot by including the index along with the variable name (ex: state_r[3:0]). If there are no explicit indexes associated with a variable, it is assumed that this variable is only one bit wide.

The Verilog preprocessor run on the modules before simulation also adds a "__message" variable to each module and assigns this variable a default value of five. This variable is used to control whether a message associated with a "$info" watchdog macro is displayed. The first argument to the "$info" macro is the debugging level associated with the message. A message will be displayed only if the module debugging level "__message" variable is greater than or equal to the debugging level associated with a "$info" statement. The addition of the module debugging level variable has the benefit of allowing the number of module information messages displayed to be dynamically controlled at run time. We have found this feature to be very useful. An example expansion of the "$info" system macro is as follows:

Example 4 – Expansion of $info(1,"Reading bank-%x\n",mbank_r);

```
/*vpm*/begin
if (__message >= (1)) begin
  $write("[%0t] -I:%stest2.v:0082          : Reading bank -%x\n",
         $time,idm._id_ascii, mbank_r);
end
end/*vpm*/
```

As with the "$assert" macro, all the macro arguments other that the first one, are passed directly to the included "$write" system call. This allows complicated write statements displaying multiple variable values to be used.

The other "watchdog logic" macros listed in Example 1 are expanded in an analogous manner to the "$assert", "$assert_onehot", and "$info" macros.

4.0 Design Compiler Aggressively Optimizes Away Unneeded Logic

In the process of generating "assertion checks" one often needs to create temporary variables and registers. For instance, one might want to create a delayed version of some logic to make sure it does or does not occur at the same time as another signal. In addition, the generation of a temporary variable might make an "assertion check" easier to code, or it make it easier to understand the common relationship between multiple assertion checks.

One can add "watchdog logic" to a sythesized design module without needing to worry about this logic appearing in the final design, for Design Compiler aids the "watchdog logic" writer by aggressively optimizing away nonessential RTL logic cones. It will remove unneeded combinatorial logic and unneeded sequential logic as well as optimize away the unneeded sequential and combinatorial logic derived from this logic.

Example 5 below is an example of module with a extra logic added.

Example 5 – Optimizing Module with Extraneous Logic

Input Module

```
module test3(e, e1, clk, a, a1, a2, a3);
   input clk;
   input a, a1, a2, a3;
   output e, e1;

   reg       d1, d2,d3, a_d1r, e, e1;
   reg [1:0] d4,d5;
   wire      a1_inv, a_inv, a3_inv;

   wire  t1 = d3 && a;
   wire  t2 = t1 && a2;

   inv_ccr     inv1(.in(a1),.out(a1_inv));
   inv_ccr_clk inv2(.in(a),.out(a_inv),
                    .clk(clk));
   inv_ccr     inv3(.in(a_d1r),
                    .out(a3_inv));

   always @(posedge clk) begin
      e <= #1 (a & a2 & a3);
      e1 <= #1 ~a1_inv;

      d1 <= #1 (a || a2);
      d2 <= #1 (d1 && a);
      d4 <= #1 (d2 + a1 + a2 + a3);
      d5 <= #1 d4;

      a_d1r <= #1 a;
      d3 <= #1 a_inv;
      $assert(d1==a3,"Input Overlap\n");
  end
endmodule
```

Design Compiler Results

```
module test3 (e,e1,clk,a,a1,a2,a3);
input  clk, a, a1, a2, a3;
output e, e1;
  wire a_d1r, a1_inv, n_3;
  inv_ccr_clk inv2(.in(a),.clk(clk));
  inv_ccr_1 inv3(.in(a_d1r));
  inv_ccr_0 inv1(.in(a1),
                 .out(a1_inv));

  mfntnq1 e1_reg(.da(1'b0), .db(1'b1),
                 .sa(a1_inv),.cp(clk),
                 .q(e1));

  dfptnq0 a_d1r_reg(.d(a),.cp(clk),
                    .sdn(1'b1),
                    .q(a_d1r));

  mfntnq1 e_reg(.da(a3), .db(1'b0),
                .sa(n_3),.cp(clk),
                .q(e));
  an02d1 SG9 (.a1(a), .a2(a2),
              .z(n_3));
endmodule
```

As one can see from the example above, Design Compiler will optimize away logic which has no affect on the output of the module AND which has no affect on the inputs to any user defined modules. In Example 5, flip-flop "a_d1r" is not optimized away because it is an input to user module "inv1". Design Compiler will not optimized away user defined modules. However, if an output of these modules is not used, the output will be optimized away. For instance, in Example 5, the output of "inv2" is optimized away because it is not used.

Design Compiler optimizes away nonessential RTL logic in its verilog input stage as the RTL is being mapped to synthetic library elements. Example 6 shows the Design Compile logfile output for the code shown in Example 5. One can see that in the initial mapping, even before the compile, Design Compiler has optimized away flip-flops "d1" through "d5". Hence, the removal of this logic is independent of the compile options utilized.

Example 6 – Design Compiler Logfile

```
Inferred memory devices in process
      in routine inv_ccr_clk line 15 in file
================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
================================================================================
|       out_reg       | Flip-flop |   1   |  -  | -  | N  | N  | N  | N  | N
================================================================================

Inferred memory devices in process
      in routine test3 line 36 in file
================================================================================
|    Register Name    |   Type    | Width | Bus | MB | AR | AS | SR | SS | ST
================================================================================
|      a_d1r_reg      | Flip-flop |   1   |  -  | -  | N  | N  | N  | N  | N
|       e1_reg        | Flip-flop |   1   |  -  | -  | N  | N  | N  | N  | N
|        e_reg        | Flip-flop |   1   |  -  | -  | N  | N  | N  | N  | N
================================================================================
Current design is 'test3'.
  Uniquifying cell 'inv1' in design 'test3'.  New design is 'inv_ccr_0'.
  Uniquifying cell 'inv3' in design 'test3'.  New design is 'inv_ccr_1'.
```

Unused task logic is removed in the same manner. In essence, Design Compiler flattens the instances of task logic and then works to optimize them away in the same manner as other RTL code. Although Design Compile will not optimize away unused user defined modules, it will work to optimize away unused library modules. However, this optimization happens at compile time and the amount of optimization depends on the compile options used. Applying an “ungroup –all” to the results of Example 6 (ungroup itself does not remove logic) and using a low effort compile will not remove all the extraneous library modules. If one uses a high effort compile instead, all the extraneous library modules will be removed from the design

5.0 Eliminate “synopsys translate_off” From Code

At first inspection, including “synopsys translate_off/on” pairs to assure that “watchdog” related logic does not appear in the final design, might seem like a good idea. However, the problem with this directive and any other synthesis only directive is that they can lead to a mismatch between pre- and post-synthesis simulation. A misplaced “synopsys translate_on” directive, or a changed to the design to use a variable inside the “translate_on/translate_off” block is all it takes for the design to become cursed with a pre- to post-synthesis simulation mismatch.

Avoiding this problem is as simple as eliminating the “translate_off” directives from one’s HDL code. For the most part, this directive is really not needed. As shown above, Synopsys will aggressively eliminate unneeded logic from the design, so extra “watchdog logic” can be freely added without needing to worry about it appearing in the final design. Combining this with the “assertion” macro methodology shown above allows one to virtually eliminate the need for the “translate_off” directive in their code. With the methodology shown earlier, one does not need to

hand code any “translate_off/on” blocks around the assertion check logic they are including. Instead, the assertion check macros disguised as system calls serve to hide the assertion check logic from the synthesis step.

In general we have found only three conditions which really require the “translate_off” directive:

Example 7 – Uses for “translate_off” Directive

1) Model instances you want to avoid being compiled by synopsys

```
//synopsys translate_off
gen_pc_log_pc1_log();    // generate PC logfile
//synopsys translate_on
```

2) Hierarchical name references in synthesizable code

```
//synopsys translate_off
$write(“count=%w\n”,`c_buf.count);
//synopsys translate_on
```

3) Modeling asynchronous set/reset flip-flips correctly [6]

```
always @(posedge clk or negedge rstn or negedge setn)
    if  (!rstn) q     q<= 0;
    else if (!setn) q<= 1;
    else q <= d;
//synopsys translate_off
always @(rstn or setn)
    if (rstn && !setn) force q = 1;
    else  release q;
//synopsys translate_on
```

In cases other than above, this directive should just be avoided.

Actually, this directive does not need to be used at all. It is better to use Verilog preprocessor commands to control the inclusion of simulation specific code. Using a “`ifdef synthesis `else ... `endif” preprocessor line instead of a “synopsys translate_off/on” pair allows one more direct control over the inclusion of code. In this example, the code in the “`else” clause can be excluded by defining the variable “synthesis”. Furthermore, valid Verilog syntax requires the closing “`endif”, so with this approach one does not need to worry about the dangling “synopsys translate_off” problem.

6.0 Guidelines for Including Watchdog Logic

For those not used to thinking in terms of pre-conditions and post-conditions, determining what to use “assertion” checks to verify can initially be a little troublesome. However, selecting where to use assertion checks is actually very easy. Input conditions required by the code and output conditions guaranteed by the code are candidates for assertion checks to assure that these conditions remain invariant. In less formal terms, when writing code, those conditions required for the proper operation of the code are candidates for assertion checks. When modifying code,

conditions required for the code change to work correctly or even conditions which reflect one's general belief in how the code works are good candidates for assertion checks.
Although we use assertion checks to cover a wide variety of conditions, we find that many of our assertion checks can be categorized into five or six basic types. These types are as follows:

Example 8 – Basic Assertion Check Categories

1) Checking that the state of a state machine is one hot

```
$assert_onehot(req_sm[7:0],"Req state vector is not one hot\n");
```

2) Taming non-fully specified case statements (those requiring //synopsys full_case)

```
$assert_onehot(adr_sel==3'b010,adr_sel==3'b100,adr_sel==3'b000,
                "Bad adr_sel mux select = %x\n",adr_sel);
```

3) Verifying interface logic

```
$assert(!(grant && !req),"Grant without request\n");
$assert_amone(req_a,req_b,req_c,req_d,"Multiple Requestors\n");
$assert(!(req && req_d1r), "Back to back request signal\n");
```

4) Checking to make sure counters do not wrap

```
$assert(!(count==4'hf && add_q && !del_q), "Q overflowed\n");
```

5) Informing the user that something has happened

```
$info(0, "just received a new event: type=%x",event_type);
```

6) Printing an error message if "default" case is reached

```
default: $error(0, "Default xx-yy condition has  been reached\n");
```

In our experience, assertion checks end up being most useful when they are included up front by the module designer. The payback for adding these checks as the design is created can be great, for when a design bug is caught by an assertion check it eliminates the need to determine why a module has failed. The assertion check can directly tell us what has gone wrong. In general, assertion checks serve to constrain the operating space of a module. In doing so, they make it more readily apparent when the module begins to operate outside its design parameters.

7.0 Conclusion

As HDL models become larger and begin to include HDL code from an increasing number of sources, modular design practices alone are not sufficient to manage the growth in the diversity of the module interfaces. Adopting principles of "Design by Contract" and using assertion checks to assure certain characteristics of the individual modules remain invariant is the fundamental design strategy required by this additional complexity. In our experience, assertion checks will be readily added to a module by the HDL designer, only if these checks can be easily and transparently added directly to the HDL code. Disguising the assertion checks as Verilog system calls and expanding these "macros" only before simulation is the best method to simplify adding these checks. Combining this approach with Design Compiler's aggressive optimizing away of unneeded logic, allows assertion checks to be freely constructed without the need to add additional synthesis directives to the HDL code.

8.0 References

[1] v2html, http://www.burbleland.com/v2html/v2html.html

[2] Emacs Verilog Mode, http://www.ultranet.com/~wsnyder/veripool/verilog-mode.html

[3] Betrand Meyer: “Applying Design by Contract”, in Computer (IEEE), vol 25, no 10, October 1992, pages 40-51

[4] Betrand Meyer: Eiffel: The Language, Prentice Hall, 1992

[5] Jean-Marc Jezequel: “Put it in the Contract: The Lessons of Ariane”, in Computer (IEEE), vol 30, no 2, pages 129-130

[6] Don Mills, Clifford E Cummings: “RTL Coding Styles That Yield Simulation and Synthesis Mismatches”, SNUG San Jose 1999, user session