



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

Duane Galbi
Wilson Snyder
Conexant Systems, Inc.



The Fundamental Design Issue

- Even with the oddities of Verilog the actual writing of HDL code is relatively quick
 - General sensitivity lists, what were they thinking!!
- Verilog emacs auto-modes (www.ultranet.com/~wsnyder/veripool) key for removing the worst drudgery
- Real time sink is getting code to work “correctly”
- Problem only gets worse when include interfaces between multiple designers and multiple standard buses
- Multiple coding standards and strategies adopted to deal with the issue



Coding Standards

- Help find problems early by identifying common problems
 - Naming conventions
 - wires vs registers, clock domain suffix, module prefix, etc
 - Lint checking software
 - Limits on module size
 - Standards for case statements
- Very beneficial but taken to an extreme can be painful
 - Prefix requirements leading to very long names
- Limits to what coding standards can accomplish




Coding Strategy - Powerful Verification Medicine

- Modular Design
- Reference Designs
- Bus Checkers
- **Design by Contract - Module Level Assertion Checks**
 - Add checks to verify key input/output/internal requirements of module
 - Approach fundamental part of Eiffel programming language
 - Offshoot of formal program verification
 - Enhances reusability and debuggability of code
 - Acts to limit allowable operating space of the module
 - Documents and verifies key requirements of the code
 - Unambiguously notifies user/designer when key conditions violated



Agenda

- Motivate use of module level watchdog logic 
- Illustrate how watchdog logic is easy to include
 - Synthesizable watchdog logic
 - Watchdog logic macros disguised as Verilog system calls
 - Verilog preprocessor to expand watchdog macros
 - Synopsys' ability to optimize away unneeded logic
 - Synopsys translate on/off pairs elimination
- Guidelines for using watchdog logic
- Conclusions



Synthesizable Watchdog Logic

- Add watchdog logic as module is created
 - Serves to highlight key requirements during module creation
- Put watchdog logic directly in synthesized code
 - Close proximity between checking logic and what is being checked
 - No separate module to maintain
- Insert watchdog logic in module in manner which does not generate any corresponding real hardware
 - Check code should only be software modeling artifact
 - Needs to be compatible with lint, coverage, and synthesis tools
- Disguise watchdog logic as Verilog system calls



Assertion Macros Disguised as Verilog System Calls

Duane Galbi
Conexant
Systems, Inc

7

- Watchdog logic tends to follow predictable form
 - Check condition, and print message if it is invalid
 - Want global variable to disable watchdog logic during chip initialization
 - Want watchdog logic to be ignored by non-simulation software
- Assertion macros greatly simplify writing of watchdog logic
- Discovered over the last few years, are only a few key macros needed to simplify writing watchdog logic
 - Disguising these macros as Verilog system calls causes them to be ignored by synthesis, lint, and coverage checking software
- For simulation only, macros are expanded to Verilog code using simple Verilog preprocessor
 - Macros expanded into just one long line to keep the absolute line numbers in the file unchanged



Five Key Assertion Macros

- 1) `$assert(<condition>, <msg>);`
Checks if condition is true
Example: `$assert(!(rd1 && rd2),"Multiple Reads\n");`
- 2) `$assert_onehot([<variables>], <msg>);`
Checks variable or variable list is one hot
Example: `$assert_onehot(sel_a,sel_b,"mux_selects\n");`
`$assert_onehot(state_r[7:0],"State_r not one-hot\n");`
- 3) `$assert_amine([<variables>], <msg>);`
Checks variable/variable list contains at most one logically valid condition
Example: `$assert_amine(gnt_a,gnt_b,"Multiple grants active\n");`
- 4) `$error(<level>, <msg>);`
Prints out error message and stops the simulation
Example: `$error(0,"Bad counter value=%x\n",count);`
- 5) `$info(<level>, <msg>);`
Prints informational message and continues the simulation
Example: `$info(1,"Reading bank -%x\n",mbank_r);`



Run Verilog Preprocessor to Expand Macros

- Assertion macros expanded by simple Verilog preprocessor
- Example preprocessor at: www.ultranet/~wsnyder/veripool

```
vpm --date -o .vpm project/
```

- Traverses the “project/” directory tree and macro-preprocesses on all the Verilog files in the tree
- Puts all the resultant files in the “.vpm” directory (typically local to the machine where running simulation)
- The “--date” option indicates only want to preprocess those files which have be modified since the last time the preprocessor was run



Expansion of \$assert()

\$assert(!(rd1 && rd2), "Multiple Reads\n");

Global disable

```
/*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/
```

Module name and line number added by vpm

grep of log for %E will show errors

Error count used to terminate simulation a few cycles after error detected

Expansion of \$assert_onehot()

\$assert_onehot(d1,d2,d3, "Mux Selects\n");

```

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 (%s) --> ",
      $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 (%s) --> ",
      $time,idm._id_ascii,({d1,d2,d3}));
    $write( "Mux Selects\n" ); `pli.errors = `pli.errors + 1;
    `pli.errors = `pli.errors + 1;
  end
endcase
  
```

No error if only one input active

Error: no inputs active

Construction of case statement varies based on number of variables checking

Include state of inputs in error messages

Error: multiple inputs active



Expansion of \$info()

\$info(1, "Reading bank - %x\n",mbank_r);

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

Module variable **__message**
automatically added by vpm
and initialized to 5

For all the macros, extra macro
arguments are passed directly to
the \$write() system call



Synopsys Aids the Watchdog Logic Writer

- Temporary variables often needed for watchdog logic
 - Delayed version of some logic
- Synopsys aids by aggressively optimizing away unused logic
 - Will optimize away logic which has no affect on the outputs of the module and no affect on the inputs to any user defined module
 - Will optimize away the full RTL logic cones
 - Will not optimize away user defined modules
- Optimization away of the logic happens in the input stage as the logic is being mapped to synthetic library elements
 - Logic removed independent of the compile options used
- Design Compile will optimize away library elements
 - Optimization happens in the compile stage and amount of optimization is dependant on the compile options used

Original Code

```

module test3(e,clk,a,a1,a2,a3);
  wire t1 = d3 && a;
  always @(posedge clk) begin
    e <= #1 (a & a2 & a3);
    d1 <= #1 (a || s2);
    d2 <= #1 (d1 & a);
    d3 <= #1 (d2 + a1 + a2);
    $assert(d1==a3,"Inputs Overlap\n");
  end
endmodule

```

Output from Synopsys

```

module test3(e,clk,a,a1,a2,a3);
  an02d1 SG9 ( .a1(a), .a2(a2), .z(n_3) );
  mfntnq1 e_reg (.da(a3), .db(1'b0),
                .sa(n_3), .cp(clk), .q(e));
endmodule

```

Combinatorial and sequential logic unrelated to the module outputs are optimized away



Logic Optimized Away in RTL input Stage

Design Compiler Logfile

Inferred memory devices in process
in routine test3 line 36 in file

Register Name	Type	Width	Bus	MB	AR	AS	SR
e_reg	Flip-Flop	1	-	-	N	N	N

Current design is 'test3'.

In initial mapping flip-flops d1
through d3 have already been
optimized away

Original Code

```

module test3(e1,clk,a,a1,a2,a3);
  inv_m iv1(.in(a), .out(a_i));
  inv_m iv2(.in(a1),.out(e1));
  inv_m iv3(.in(a1_d1r),.out(a1_i3));
  always @(posedge clk) begin
    a1_d1r <= #1 a1;
  end
endmodule
  
```

Inputs to user modules are not optimized away

Output from Synopsys

```

module test3(e1,clk,a,a1,a2,a3);
  inv_m iv3(.in(a1_d1r));
  inv_m iv2(.in(a1), .out(e1));
  inv_m iv1(.in(a));
  dfptnq0 a_d1r_reg(.d(a), .cp(clk),
    .sdn(1'b1), .q(a1_d1r));
endmodule
  
```

Unused outputs of user modules are optimized away



Avoid Synopsys Translate On/Off Pairs

- Assertion macros and Synopsys' aggressive optimization away of unused logic virtually eliminate need for excluding code from synthesis
- **Better way to exclude code is use Verilog preprocessor directives**
 - Allows user more direct control over inclusion of code
 - Valid syntax requires closing `endif eliminating dangling “translate off” problem
- In general, we have found only three cases where code really needs to be excluded from synthesis

1) Model instances want to avoid being compiled by Synopsys

```
`ifdef synthesis `else  
gen_pc_log_pc1_log(); // generate pc logfile  
`endif
```

2) Hierarchical name references in synthesizable code

```
`ifdef synthesis `else  
$write("count=%x\n", `c_buf.count);  
`endif
```

3) Modeling asynchronous set/reset flip-flops correctly

**code setup to only
define “synthesis”
during synthesis step**

- Selecting where to use watchdog logic is easy
- Useful in a wide variety of situations but can often be categorized into five or six basic types

Typically in @posedge block to keep transient values from triggering

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

```
always @(posedge clk) begin
  $assert_onehot(req_sm[7:0], "Req state vector is not one hot\n");
end
```

2) Taming not fully specified case statements (those requiring //synopsys full_case)

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

3) Verifying Interface Logic

```
$assert(!(grant & !req), "Grant without request\n");  
$assert_among(reqa, reqb, reqc, "Multiple Requestors\n");  
$assert(!(req & req_d1r), "Back to back request signals\n");
```

4) Checking counters do not wrap

```
$assert(!(count==4'hf & add_q & !del_q), "Q overflowed\n");  
$assert(!(count==4'h0 & !add_q & del_q), "Q underflowed\n");
```

5) Informing user that something has happened

```
$info(0, "Just received new event: type=%x\n", event_type);
```

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

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



Conclusions

- Adding watchdog logic to verify characteristics of a module remain invariant is a fundamental design strategy improvement
- In our experience, watchdog logic will be readily added by HDL designer only if these checks can easily and transparently be added directly to HDL code
 - Disguise assertion checks as Verilog system calls
 - Utilize Design Compiler's optimization away of unneeded logic
- This approach allows assertion checks to be freely constructed without need to add additional directives to HDL code