

# Questa Formal AutoCheck QuickStart

Rick White

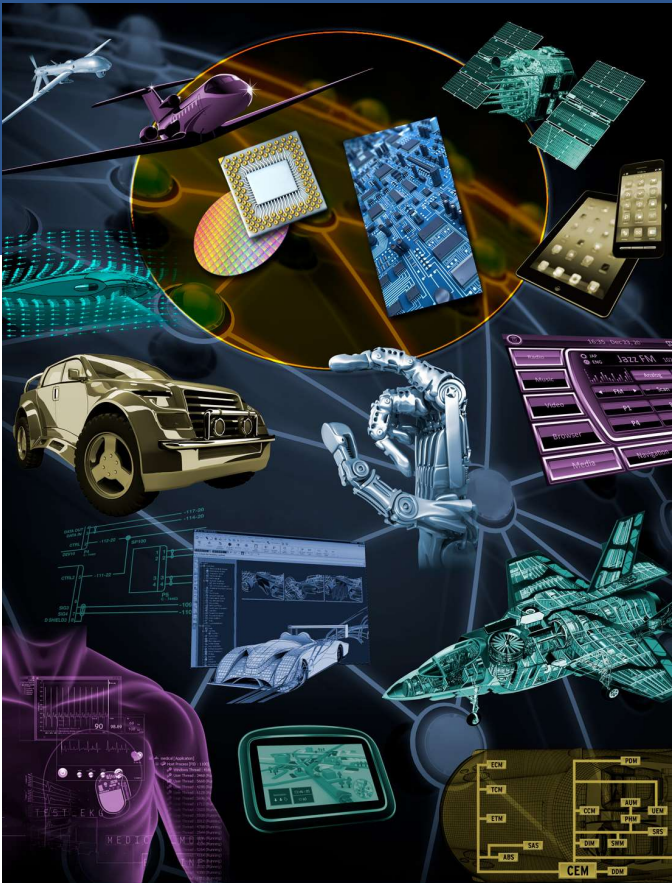
Applications Engineer Consultant  
IC Verification Solutions Division

Dominic Lucido

Technical Account Manager

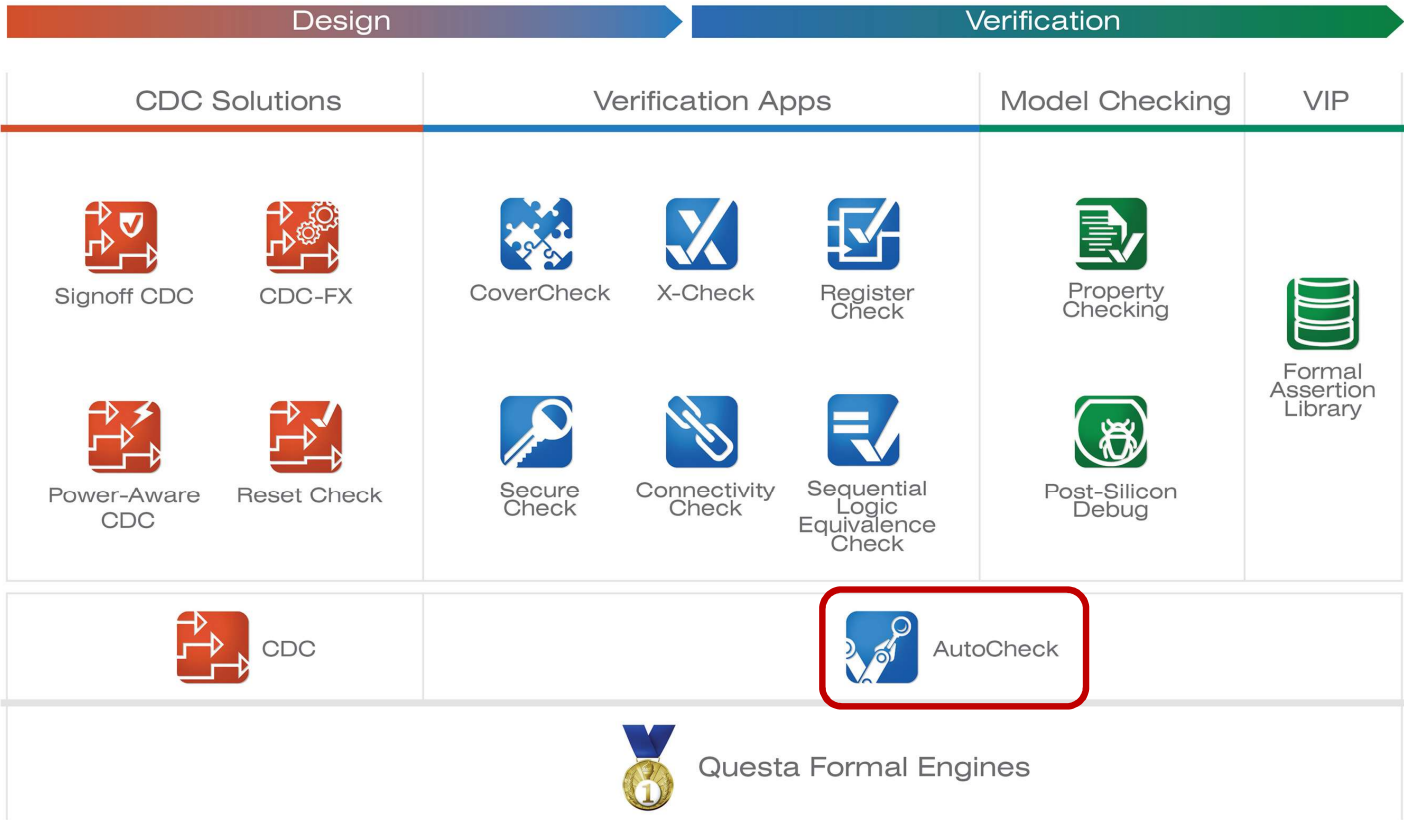
March, 2020

**Mentor**<sup>®</sup>  
A Siemens Business



# Questa Formal Solutions & Apps

*Automated, Exhaustive Verification For Complex Challenges*



# Objectives and Prerequisites

---

## ■ Objectives

- Provide students a brief understanding of what Questa AutoCheck is
- Learn how to run Questa AutoCheck on your design
- Learn how to debug with Questa AutoCheck
- Cover some more advanced topics regarding AutoCheck
- Provide a description of the Questa AutoCheck design checks

## ■ Prerequisites

- A basic understanding of design and verification
- Ability to run various verification software
- Knowledge of an HDL language

# Agenda

---

- Level I Training
  - Overview
  - Controlling AutoCheck
  - Running AutoCheck
  - Debugging with AutoCheck
  - Lab
- Appendix Level II Training
- Appendix Reference

# Automatic Checks

## Easy-to-use predefined checks for common problems

- Push-button functional verification, for checks such as:

### Initialization Checks

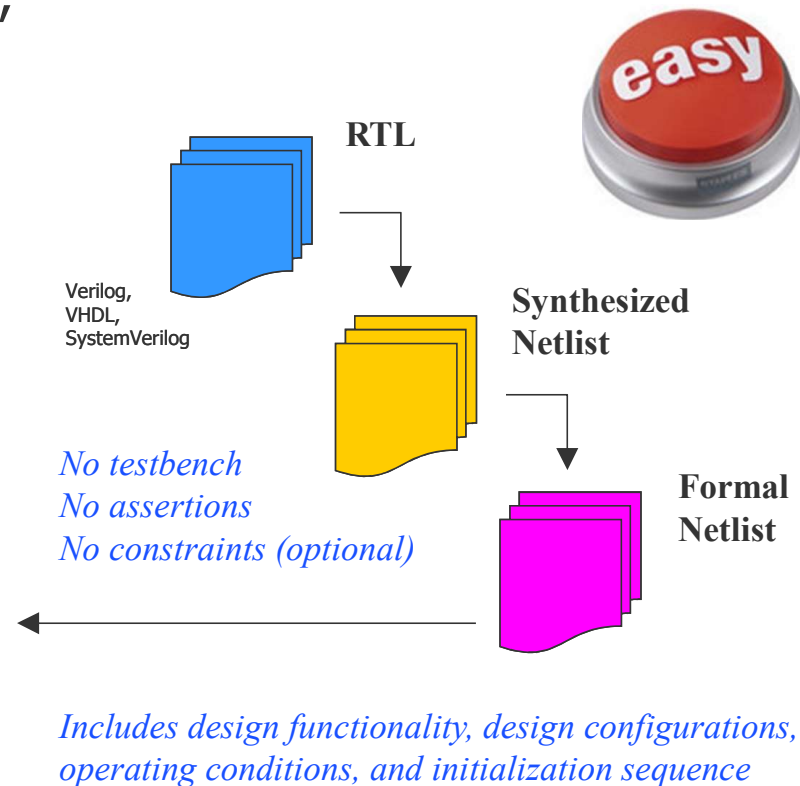
- Uninitialized registers
- X propagation/reachability

### Functional Issue Checks

- Combinational loops
- Case statement checks
- Arithmetic checks
- Bus checks
- FSM checks

### Coverage Reachability Checks

- Unreachable Logic
- Unreachable FSM state
- Unreachable FSM transition
- Register stuck at constant



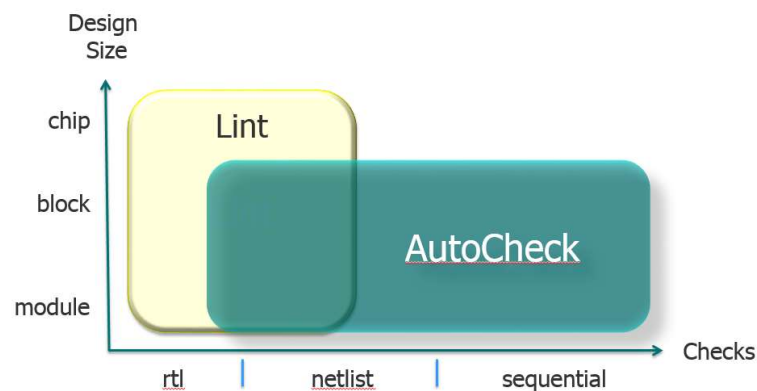
# AutoCheck and Lint are Complimentary

## ■ Questa AutoCheck

- Automated for the user
- Run at block level primarily
- **Assertion based checks**
- **Functional verification**
- Operates on the rtl
- **Operates on synthesized netlist**
- **Operates on a formal netlist with formal engines**
- **Does sequential analysis**
- **Full Functional Debug**

## ■ Lint Checks

- Automated for the user
- Run at block/chip level
- Rule based checks
- Syntax verification
- Design style verification
- Operates on the code (rtl)
- Some netlist checks



© 2020 Mentor Graphics Corporation

**Mentor**  
A Siemens Business

# Agenda

---

- Level I Training
  - Overview
  - Controlling AutoCheck
  - Running AutoCheck
  - Debugging with AutoCheck
  - Lab
- Appendix Level II Training
- Appendix Reference

# AutoCheck Directives

---

- Questa AutoCheck is controlled by using Tcl directives
- General directives can be used here
  - `netlist clock`
  - `netlist constant`
  - `netlist blackbox`
  - `netlist blackbox instance`
- Directives specific to AutoCheck
  - `autocheck enable`
  - `autocheck disable`
- All Tcl directives follow the “last one wins” rule



# Directive: autocheck enable

- Syntax:

usage: autocheck enable

```
[-type <type_name>...]  
[-module <module_name>...]  
[-instance <module_instance_name>...]  
[-name <name_name>...]  
[-recursive] [-match_local_scope]  
[-help]
```

- Can use wildcards "\*"

- -recursive applies this to all hierarchy below –module/instance

- Example Directives:

autocheck enable –type FSM\*

autocheck enable –type ARITH\_OVERFLOW\_VAL –module M

autocheck enable –type ARITH\_ZERO\_DIV –name u1.Z\* -match\_local\_scope

# Directive: autocheck disable

- Syntax:

```
usage: autocheck disable
      [-type <type_name>...]
      [-module <module_name>...]
      [-instance <module_instance_name>...]
      [-name <name_name>...]
      [-recursive] [-match_local_scope]
      [-help]
```

- Can use wildcards "\*"

- -recursive applies this to all hierarchy below –module/instance

- Example:

```
autocheck disable –type FSM*
autocheck disable –type ARITH_OVERFLOW_VAL –module M
autocheck disable –type ARITH_ZERO_DIV –name u1.Z* -match_local_scope
```

# Example: AutoCheck Directives File

- AutoCheck can be run with all default settings without a directives file
  - All clocks will have the same frequency
  - A default init sequence will be used
  - Default checks will be used
- A directives file can be used to refine the AutoCheck settings for the run

```
> cat ac_directives.do
```

```
# this is a comment
```

```
netlist clock pci_clk -period 30
```

```
netlist clock wb_clk -period 10
```

```
netlist constant test_mode 1'b0
```

```
netlist blackbox ip_3rd_party
```

```
autocheck enable
```

```
autocheck disable -type ARITH*
```

# Agenda

---

- Level I Training
  - Overview
  - Controlling AutoCheck
  - Running AutoCheck
  - Debugging with AutoCheck
  - Lab
- Appendix Level II Training
- Appendix Reference

# Inputs to the Questa AutoCheck

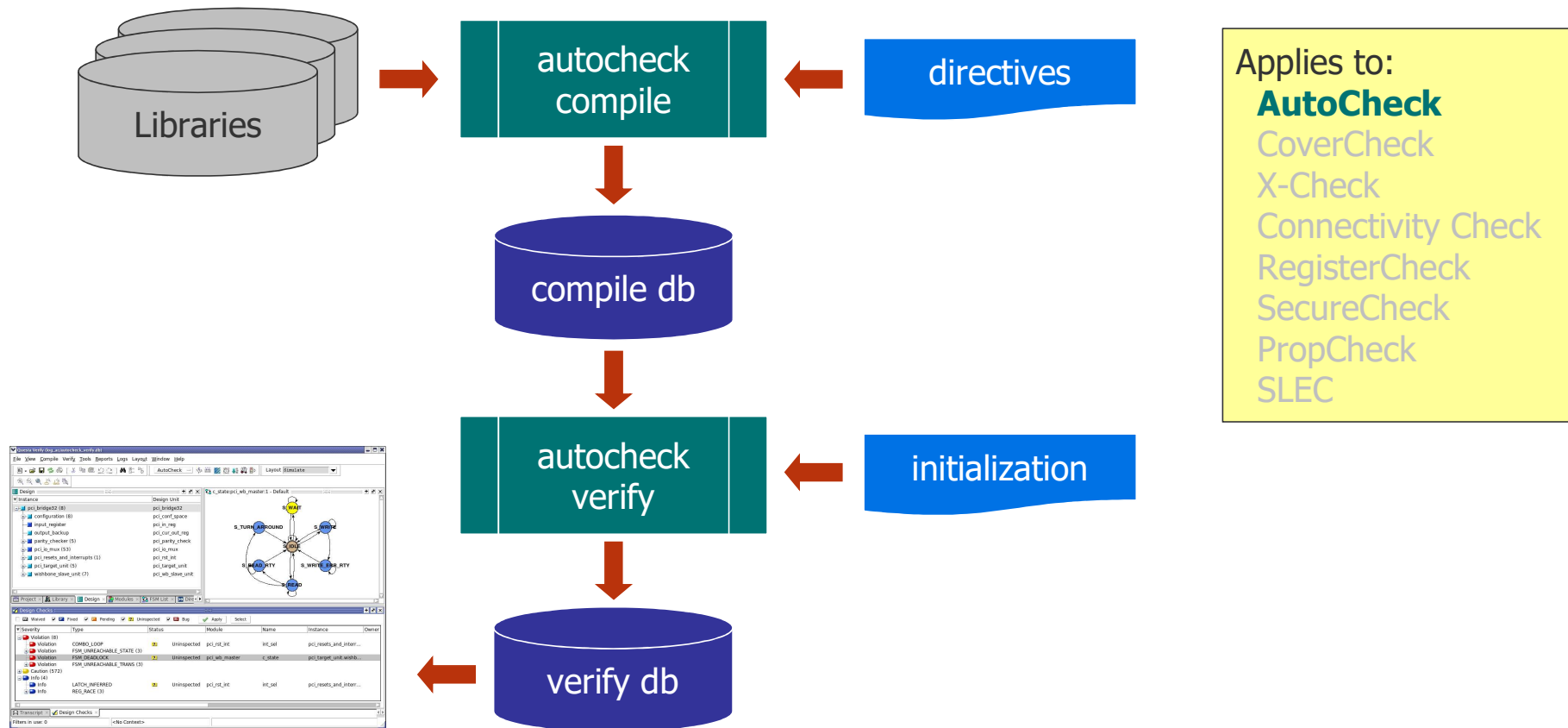
---

- Input data needed:
  - RTL(synthesizable) design files
  - Directives file to:
    - Setup your clocks (Optional)
    - Setup any simple constraints

# Running Questa AutoCheck

- There are multiple ways to run AutoCheck
  - Simple command line run (preferred)
    - vlib/vmap/vlog/vcom (compile design)
    - qverify -c ... (AutoCheck analysis)
  - Running from the GUI
- When using Questa Sim 2019.1 or greater you can use those work libraries
  - It is required that the design files in the work library be synthesizable
  - If not will require recompilation with synthesizable models if possible
    - Non synthesizable logic is blackboxed by AutoCheck
- Assertions are ignored by AutoCheck
  - Assumptions will be honored
  - Follow methods for Questa PropCheck assertion compilation

# Questa AutoCheck Flow



# AutoCheck Command: `autocheck compile`

- Tcl command to compile AutoCheck: `autocheck compile`
- Common options:
  - `-d` specifies the top of the DUT
    - Required: use top module name or entity name
  - `-cuname <compile_unit_name>`
    - Used if SVA assumptions are being bound to the design
- AutoCheck will look in the work directories for the compiled design files
  - `-work` locally or with mappings specified in local/specified modelsim.ini file
- How you run `autocheck compile` is the same no matter what language is being used

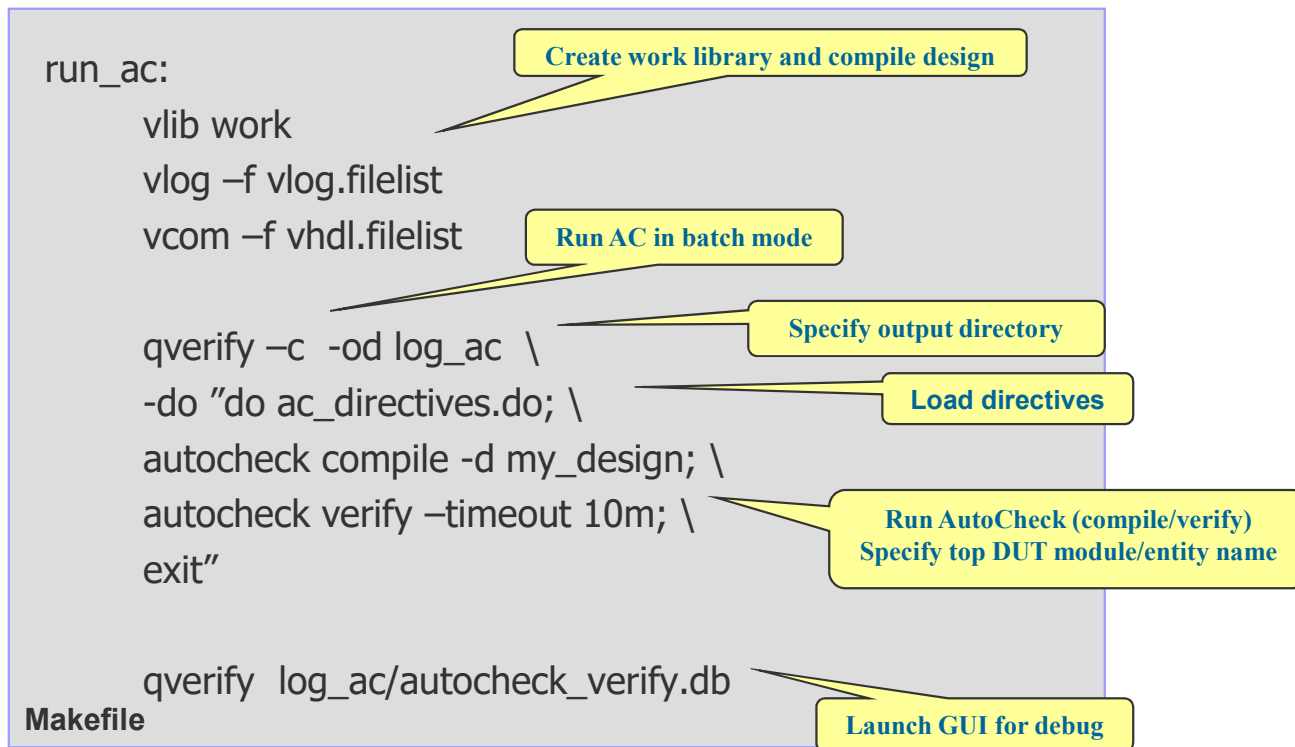


# AutoCheck Command: `autocheck verify`

- Tcl command to run AutoCheck: `autocheck verify`
- Common options:
  - `-timeout N<s|m|h|d>` ( Example: `-timeout 5m` )
    - For initial runs specify a few minutes to pipe clean, then extend the time
    - It is recommended to run with the timeout option
- The `autocheck verify` uses the `autocheck_compile.db` passed down in the same `qverify` Tcl shell process or specified with the `autocheck load db` command
- How you run `autocheck verify` is the same no matter what language is being used

# Example: Mixed Design

- qverify analysis is run in batch mode with `-c` option
  - Can run from unix shell/make scripts



# Questa AutoCheck Output Files

- Useful AutoCheck output files
  - `autocheck_settings.rpt`
    - List of all settings applied to AutoCheck
  - `autocheck_compile.log`
    - Log file, useful for compile issues
  - `autocheck.db`
    - Database containing the formal model and checks
  - `autocheck_design.rpt`
    - Contains design and other useful info
  - `autocheck_verify.rpt`
    - Contains the results summary
    - Also contains more info for the advanced user
  - `autocheck_verify.log`
    - Log file, useful for looking at verify issue and summary table
  
- View and debug results in the AutoCheck GUI
  - > `qverify autocheck.db`

# Agenda

---

- Level I Training
  - Overview
  - Controlling AutoCheck
  - Running AutoCheck
  - Debugging with AutoCheck
  - Lab
- Appendix Level II Training
- Appendix Reference

# Debugging AutoCheck Results

---







- AutoCheck results are viewed and debugged using the qverify debugger
- Results are looked at and corrected or waived
  - AutoCheck is then rerun to verify any fixes
  - Run until all checks are fixed or waived
- qverify GUI is used to view the AutoCheck results

Usage:

```
> qverify autocheck.db
```

# qverify: Design Checks Severity Icons

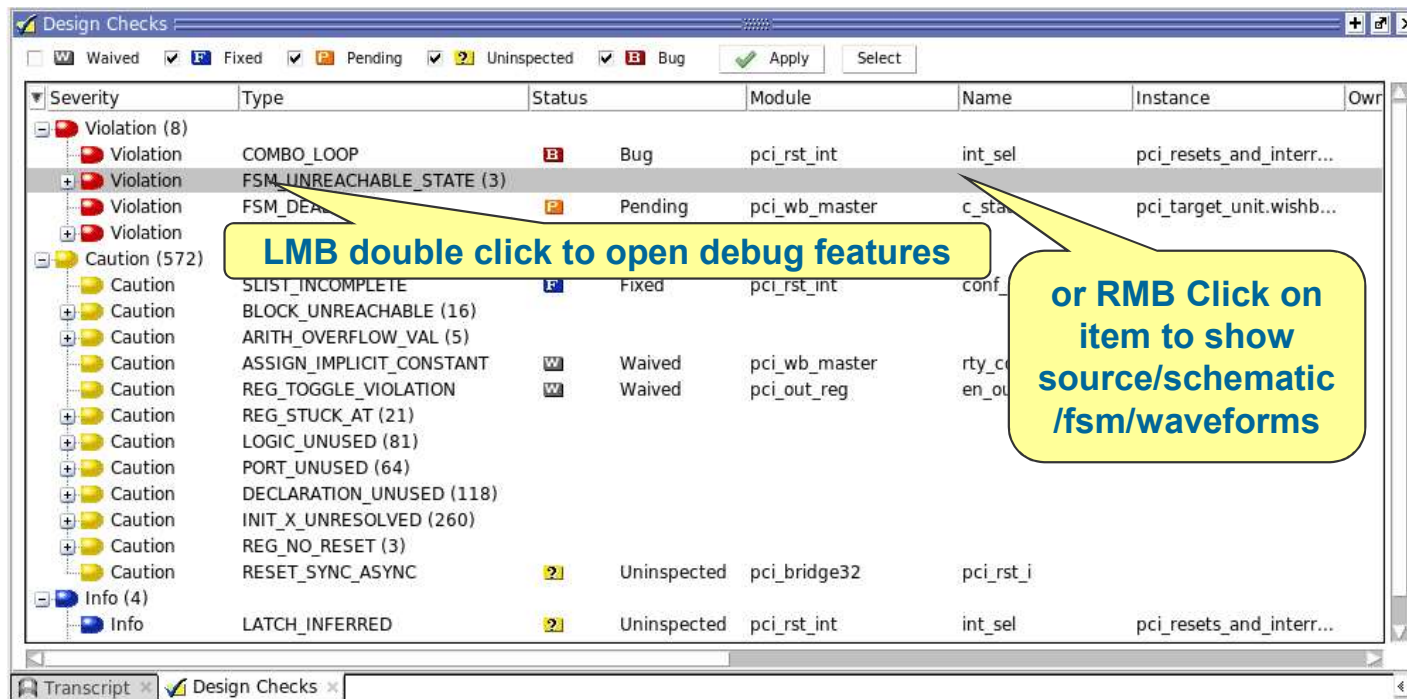
## ■ Design Checks Results

	<b>Violations **:</b>	Check found likely illegal behavior or design bug: Fix it
	<b>Cautions **:</b>	Check found a potential problem that requires inspection: Should fix it
	<b>Infos:</b>	Check is for information: May still want to fix it
	<b>Evaluations:</b>	Check was evaluated but no results found
	<b>Off :</b>	Check has been disabled
	<b>Inconclusive:</b>	Check was inconclusive

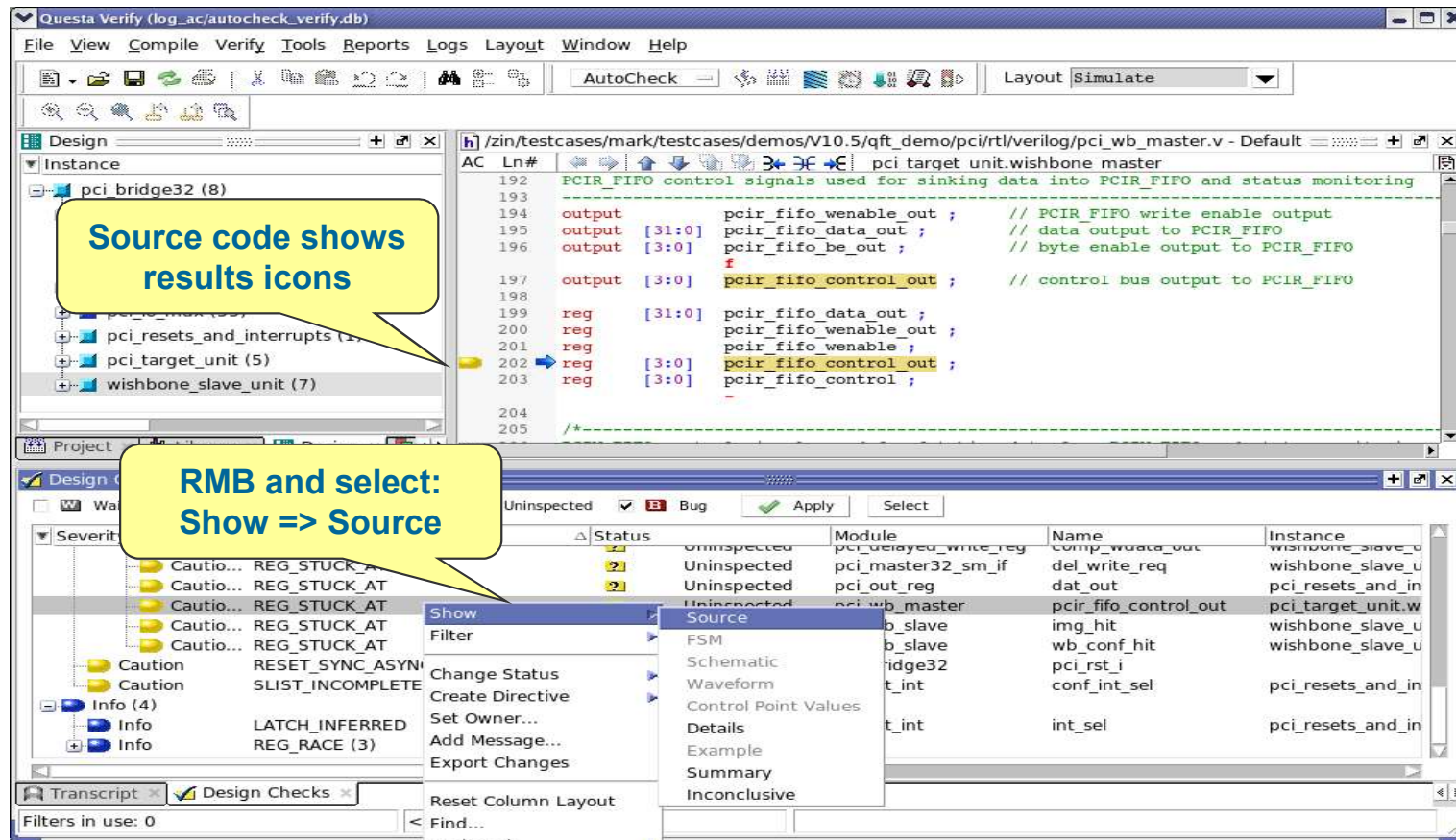
\*\* Indicates "Default" settings seen in GUI

# qverify: Design Checks Tab

- Results are grouped into categories
- Double click with LMB to debug (or select with RMB)

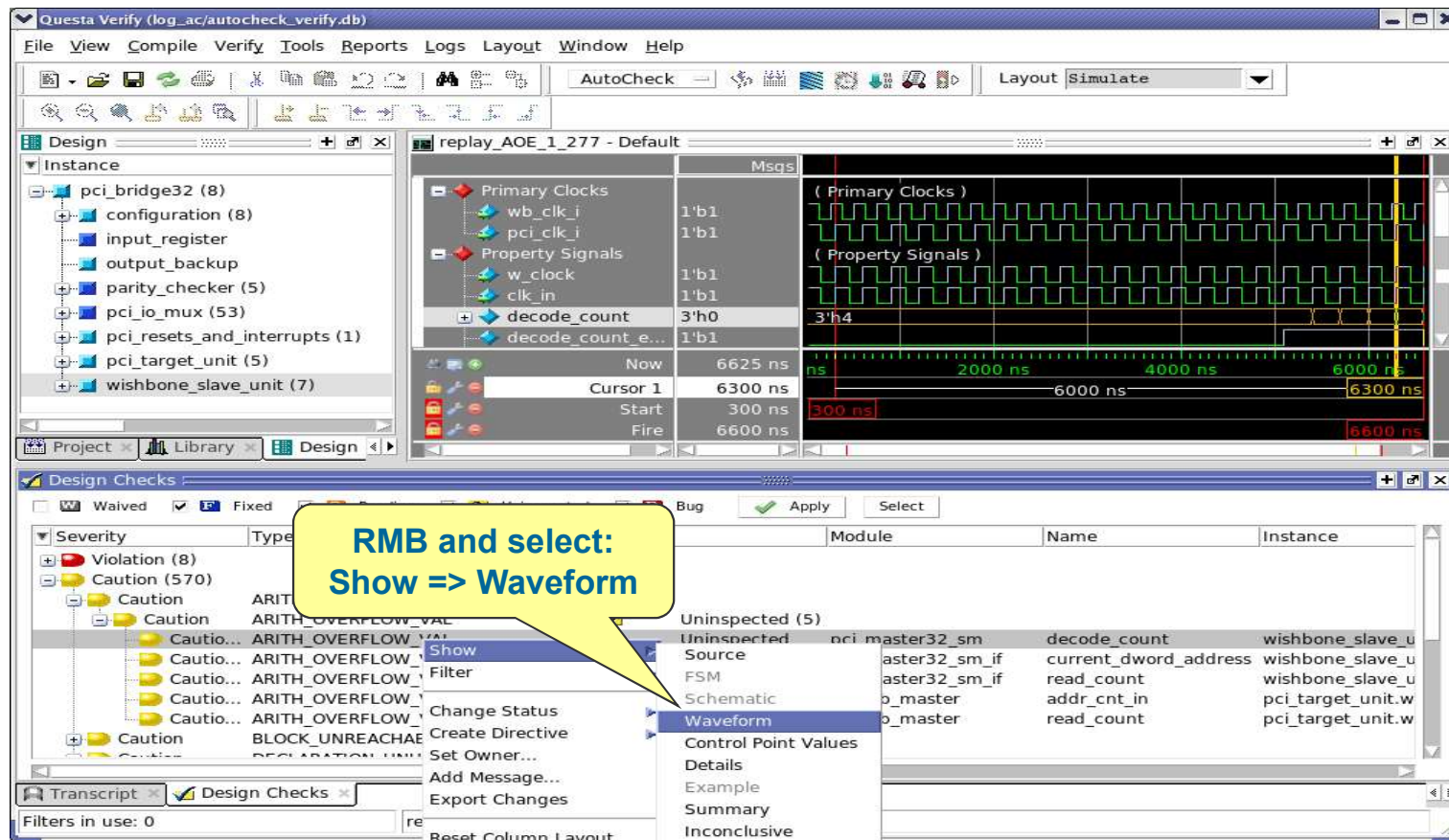


# qverify : View Source Code



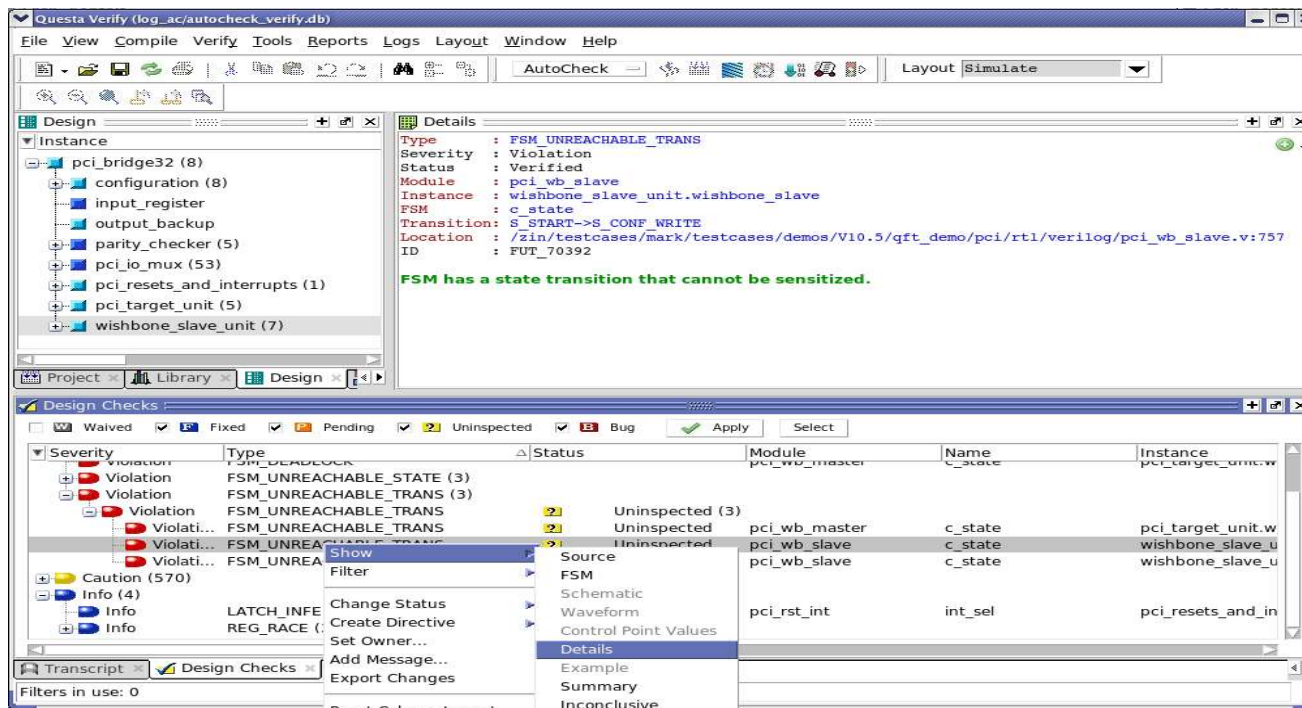


# qverify : View Waveforms



# qverify: Details Window

- Provides detailed/unique info for selected Design Check
  - Only common categories are shown in design checks tab



# qverify : View FSM

The screenshot shows the Questa Verify software interface. The top menu bar includes File, View, Compile, Verify, Tools, Reports, Logs, Layout, Window, and Help. The Design window displays a hierarchical tree of components under 'Instance', including pci\_bridge32 (8), configuration (8), input\_register, output\_backup, parity\_checker (5), pci\_io\_mux (53), pci\_resets\_and\_interrupts (1), pci\_target\_unit (5), and wishbone\_slave\_unit (7). The Design Checks window is open, showing a table of violations. A context menu is open over the 'FSM\_DEADLOCK' violation, with options: Show, Filter, Change Status, Create Directive, Set Owner..., Add Message..., Export Changes, Reset Column Layout, and Find... A yellow callout bubble points to the 'FSM' option in the menu.

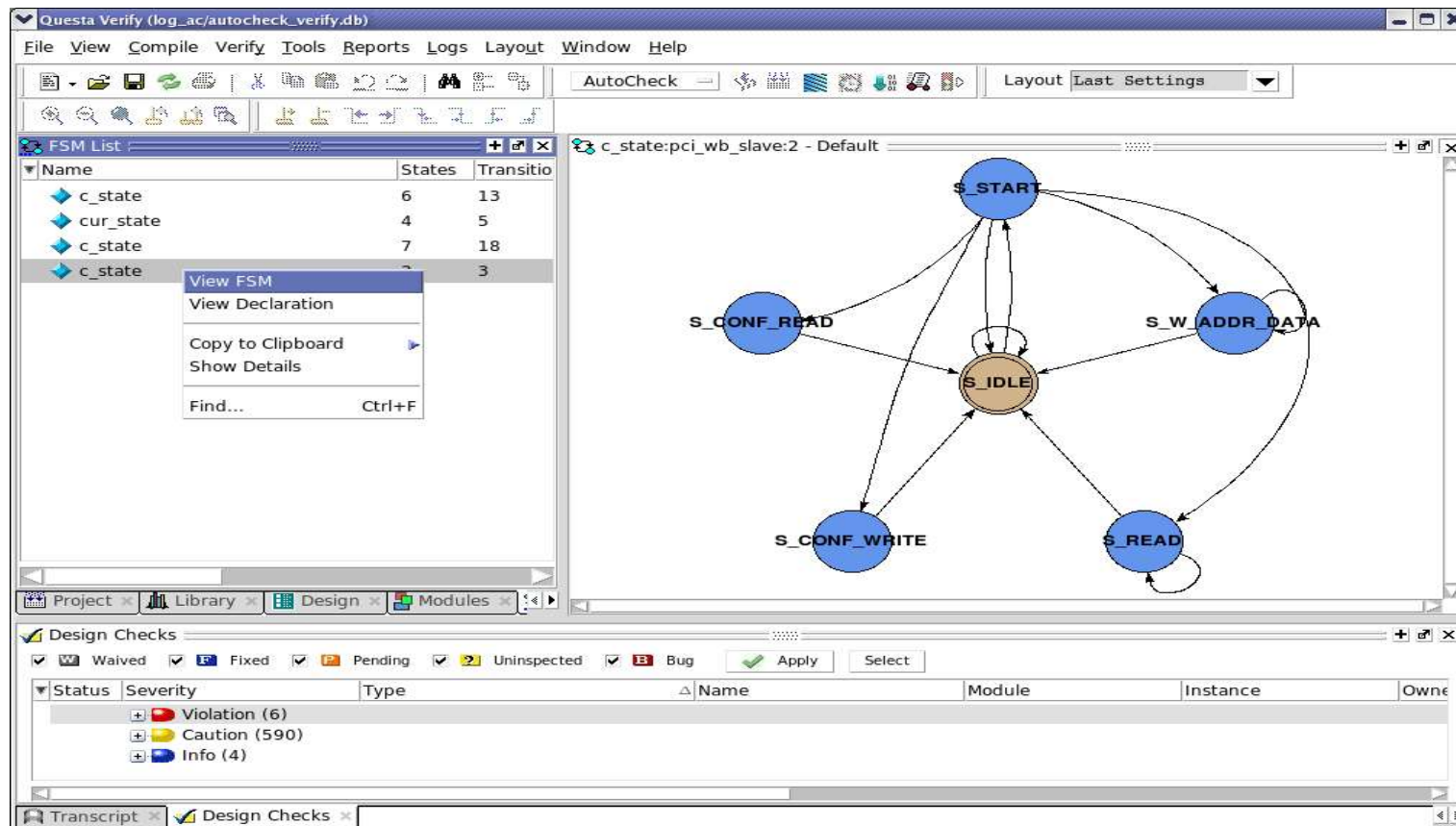
Design Checks

Severity	Type	Status	Module
Violation (8)	COMBO_LOOP	Bug	pci_rst_int
Violation	FSM_DEADLOCK	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Violation	FSM_UNREACHAE	Bug	pci_rst_int
Caution (570)			

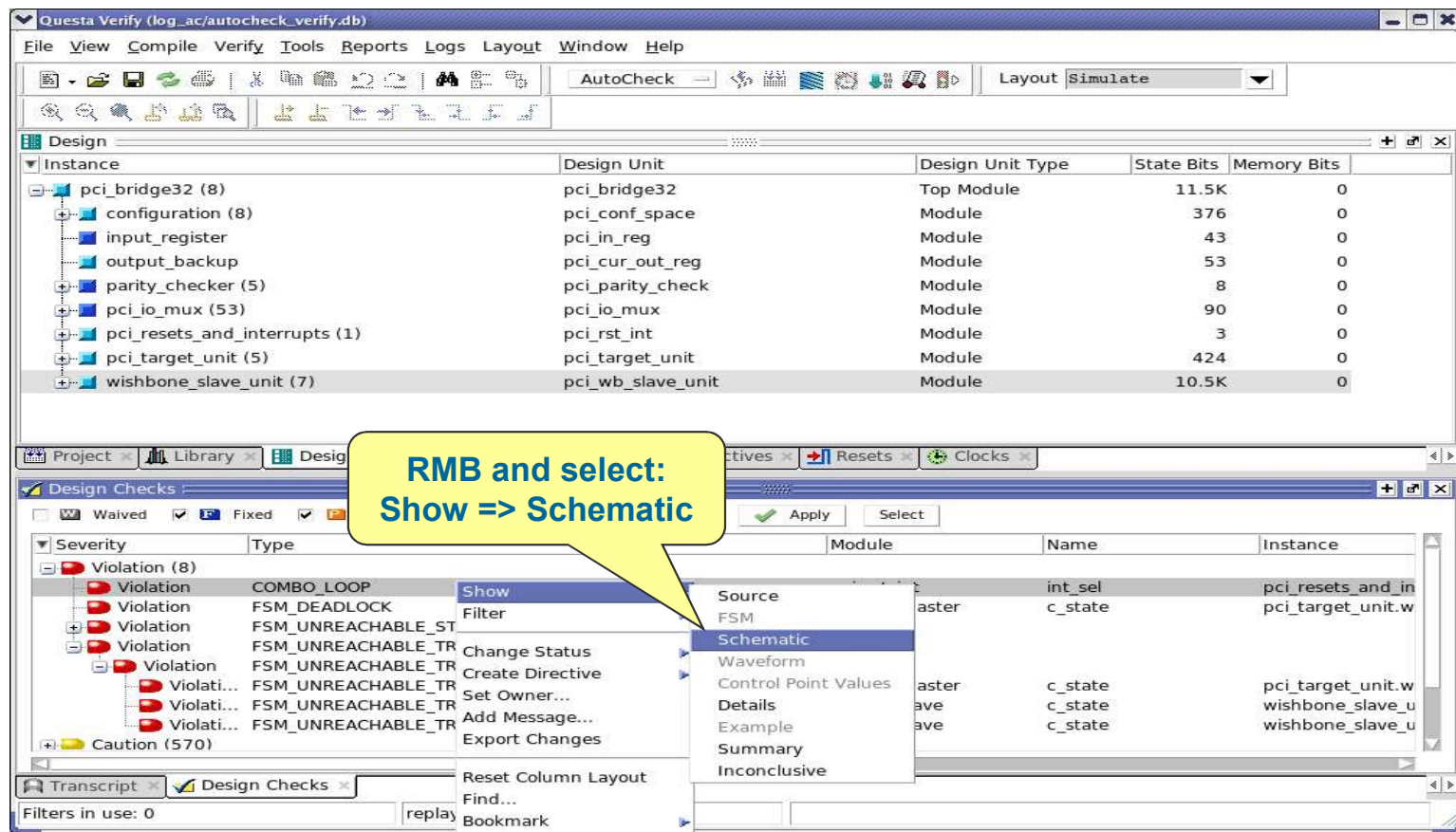
Filters in use: 0

FSM

# qverify: Design-wide FSM Tab

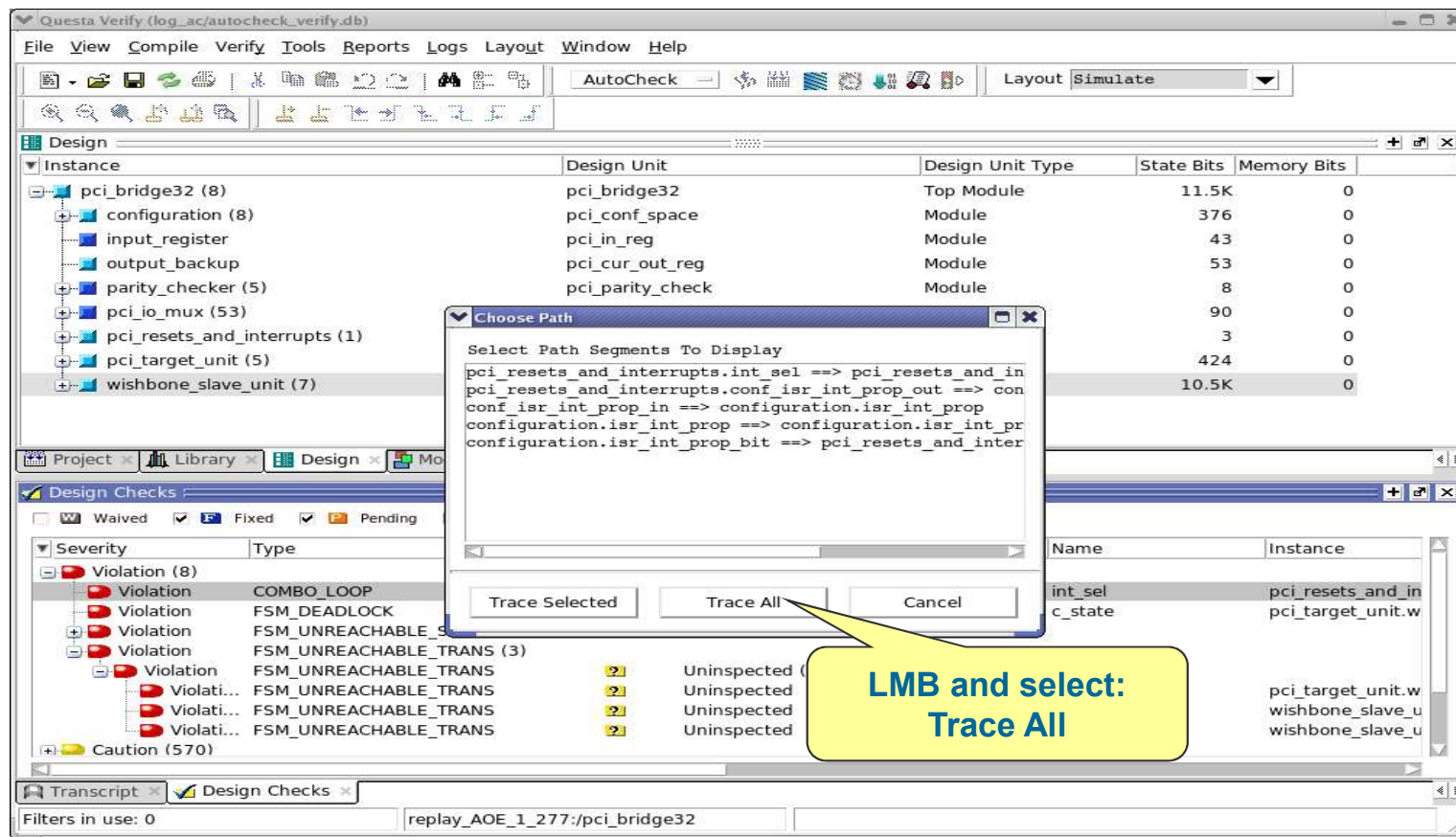


# qverify : View Schematic





# qverify : View Schematic



# qverify : View Schematic

The screenshot displays the Questa Verify (log\_ac/autocheck\_verify.db) interface. The top menu bar includes File, View, Compile, Verify, Tools, Reports, Logs, Layout, Window, and Help. The toolbar contains various icons for file operations, design navigation, and simulation. The main window is divided into two panes. The left pane, titled 'Design', shows a hierarchical tree of components under 'Instance': pci\_bridge32 (8), configuration (8), input\_register, output\_backup, parity\_checker (5), pci\_io\_mux (53), pci\_resets\_and\_interrupts (1), pci\_target\_unit (5), and wishbone\_slave\_unit (7). The right pane, titled 'CFL\_18805 - Default', shows a schematic diagram with components like 'pci\_resets\_and\_interrupts' and 'configuration' connected by signal lines. Below the schematic, the 'Design Checks' pane is visible, showing a table of violations and cautions.

Severity	Type	Status	Module	Name	Instance
Violation (8)					
Violation	COMBO_LOOP	Bug	pci_rst_int	int_sel	pci_resets_and_in
Violation	FSM_DEADLOCK		pci_wb_master	c_state	pci_target_unit.w
Violation	FSM_UNREACHABLE_STATE (3)				
Violation	FSM_UNREACHABLE_TRANS (3)				
Violation	FSM_UNREACHABLE_TRANS	Uninspected (3)			
Violati...	FSM_UNREACHABLE_TRANS	Uninspected	pci_wb_master	c_state	pci_target_unit.w
Violati...	FSM_UNREACHABLE_TRANS	Uninspected	pci_wb_slave	c_state	wishbone_slave_u
Violati...	FSM_UNREACHABLE_TRANS	Uninspected	pci_wb_slave	c_state	wishbone_slave_u
Caution (570)					

Filters in use: 0 | replay\_AOE\_1\_277:/pci\_bridge32

# Agenda

---

- Level I Training
  - Overview
  - Controlling AutoCheck
  - Running AutoCheck
  - Debugging with AutoCheck
  - Lab
- Appendix Level II Training
- Appendix Reference



# Questa AutoCheck Lab

---

- Copy the AutoCheck lab over into your work area
  - `cp -r $MYINSTALL/share/examples/labs/autocheck .`
- Run the lab
  - `cd autocheck`
  - Pick what language (vlog or vhdI) you want to run
  - Follow lab instructions in the doc directory (use `acroread` to view .pdf file)
- Follow the instructions in the lab document(pick appropriate language):
  - Questa\_AutoCheck\_Lab\_vhdl\_v1.pdf
  - Questa\_AutoCheck\_Lab\_vlog\_v1.pdf
  - Do the Level I section of the lab

# APPENDIX

# Agenda

---

- Level I Training
- Level II Training
  - AutoCheck Directives and Commands
  - Considerations when running AutoCheck
  - Status flow and debugging features with AutoCheck
  - FPGA flows with AutoCheck
  - Lab
- Reference

# AutoCheck Directives

- Questa AutoCheck is controlled by using Tcl directives
- General directives can be used here
  - netlist clock
  - netlist constant
  - netlist blackbox
  - netlist blackbox instance
- Directives specific to AutoCheck in addition to enable/disable
  - autocheck preference
  - autocheck report inconclusives
  - autocheck report type
  - autocheck report item
- All Tcl directives follow the “last one wins” rule

# Directive: autocheck preference

- Syntax:

```
usage: autocheck preference
        [-deadcode_case_default ]
        [-loop_no_latch]
        [-nostatus]
        [-help]
```

- By default, AutoCheck does not report case statement default/others as deadcode, the `-deadcode_case_default` will enable reporting these
- The `-loop_no_latch` option will turn off reporting of a CFL involving a latch
- The `-nostatus` option disables the status column
- Example:

```
autocheck preference -deadcode_case_default
```

# Directive: autocheck report inconclusives

- Syntax:

usage: autocheck report inconclusives  
[-help]

- By default, AutoCheck does not list inconclusives in the "AutoCheck Summary" table in the autocheck\_compile.log or autocheck\_verify.rpt
  - User can report inconclusive count in the summary table using this option
  - The number of inconclusives should not be fixated on, the value is in what AutoCheck finds for the user

- Example:

autocheck report inconclusives

# Directive: autocheck report type

- Syntax:

```
usage: autocheck report type  
      [<type_name>...]  
      [-severity <violation|caution|info>]  
      [-help]
```

- This directive is used to change the default severity type of specified design checks
- Can use wildcards "\*"
- Example:

`autocheck report type FSM* violation`

# Directive: autocheck report item

## ■ Syntax:

```
usage: autocheck report item
[-status <fixed|bug|pending|waived>]
[-owner <owner>...]
[-type <type_name>...]
[-module <module_name>...]
[-instance <module_instance_name>...]
[-name <name_name>...]
[-checksum <checksum>...]
[-message <user_comment>...]
[-reviewer <reviewer>...]
[-recursive]
[-add]
[-state <state>...]
[-from_state <from_state>...]
[-to_state <to_state>...]
[-bit_index <bit_index>...]
[-composite_signal]
[-help]
```

Typically reserved for GUI usage with waivers.  
Expert users can try it manually.

- This directive is used for status/comments/owner/reviewer etc.
  - These are typically generated from the qverify GUI

## ■ Example

```
autocheck report item -status waived -type REG_STUCK_AT -instance rstn_ints.inta -name dat_out
```



# Example: AutoCheck Directives File

```
> cat ac_directives.do
# this is a comment
netlist clock pci_clk -period 30
netlist clock wb_clk -period 10

netlist constant test_mode 1'b0

autocheck enable
autocheck disable -type ARITH*
autocheck disable -type FSM* -module XYZ

autocheck preference -deadcode_case_default
```

# Example: Mixed Design with Assumptions

- qverify analysis is run in batch mode with `-c` option
  - Can run from unix shell/make scripts
  - Example with SVA bind and PSL vunit used for assumptions (all assertions ignored by AC)

run\_ac:

```
vlib work  
vlog -f vlog.flst  
vlog -sv assumes.sv bind.sv -mfcu -cunome my_bind  
vcom file1.vhd -pslfile file1.vunit
```

Create work library and compile design and assumptions

Run AC in batch mode

```
qverify -c -od log_ac \  
-do "do ac_directives.do; \  
autocheck compile -d my_design -cunome my_bind; \  
autocheck verify -timeout 1h; \  
exit"
```

Specify output directory

Load directives

PSL/SVA read automatically  
Specify SVA bind cunome if used

Run AutoCheck for 1 hour

```
qverify log_ac/autocheck_verify.db
```

Makefile

Launch UI for debug

# Reporting Enhancements

- AutoCheck compile.log summary table contains only compile time checks but the summary table in autocheck verify.log includes all checks.
- CPU time table (see checks which consume the most time)
  - Contains only issues checked during verify time
  - Number of jobs run now taken into account

-----  
CPU Time per Check Type

%	Seconds	Check
0.0%	0	ARITH_OVERFLOW_SUB
0.0%	0	ARITH_OVERFLOW_VAL
0.0%	0	ARITH_ZERO_DIV
0.0%	0	ARITH_ZERO_MOD
9.5%	6	BLOCK_UNREACHABLE
0.0%	0	BUS_MULTIPLY_DRIVEN
0.0%	0	BUS_UNDRIVEN
0.0%	0	BUS_VALUE_CONFLICT
0.3%	0	CASE_DEFAULT
0.0%	0	CASE_FULL
0.0%	0	CASE_PARALLEL
1.8%	1	FSM_STUCK_STATE
0.5%	0	FSM_UNREACHABLE_STATE
31.7%	20	FSM_UNREACHABLE_TRANS
0.0%	0	INDEX_ILLEGAL
0.0%	0	INIT_X_OPTIMISM
0.2%	0	INIT_X_PESSIMISM
0.2%	0	INIT_X_UNRESOLVED
0.0%	0	INIT_X_UNRESOLVED_MEM
0.0%	0	ONE_COLD
0.0%	0	ONE_HOT
54.5%	35	REG_STUCK_AT
1.4%	0	REG_TOGGLE_VIOLATION

-----

**This check is taking a  
lot of the run time**

**This check is taking a  
lot of the run time**

# Enable/Disable AutoCheck Checks Post Compile

- Checks supported: Verify Checks
  - ARITH\_(OVERFLOW\_SUB/VAL, ARITH\_ZERO\_DIV/MOD)
  - BLOCK\_UNREACHABLE
  - BUS\_(MULTIPLY\_DRIVEN/UNDRIVEN/VALUE\_CONFLICT)
  - CASE\_(DEFAULT/FULL/PARALLEL)
  - FSM\_(DEADLOCK\_STATE/LOCKOUT\_STATE/STUCK\_BIT/UNREACHABLE\_STATE/TRANS)
  - INDEX\_ILLEGAL
  - INIT\_X\_(OPTIMISM/PESSIMISM/UNRESOLVED/UNRESOLVED\_MEM)
  - ONE\_COLD/ONE\_HOT
  - REG\_(MULTIPLY\_DRIVEN/STUCK\_AT/TOGGLE\_VIOLATION)
- The covercheck\_verify.log/rpt file contain only current run data
- The covercheck\_verify.db file contains all data from previous runs
- Can split up the runs based on check difficulty

# Example: autocheck enable/disable

- User breaks up the AC runs based on check type and timeout

```
onerror {exit}  
##### add directives  
netlist constant use_1clk_i 1'b0
```

Enable all checks,  
Disable the ARITH checks

```
autocheck enable  
autocheck disable -type ARITH*  
##### Run AutoCheck  
autocheck compile -d axi4lite_to_apb4
```

compile

```
autocheck disable -type FSM*  
autocheck verify -timeout 5m
```

Run everything except FSM  
checks shorter time

```
autocheck disable  
autocheck enable -type FSM*  
autocheck verify -timeout 10m
```

Run the FSM checks for  
longer

```
exit
```

# autocheck load db => autocheck verify

---

## ■ Incremental AutoCheck

- Checks with conclusive results are imported, so that in the second run only inconclusive checks are targeted
- Can load an autocheck\_verify.db and a subsequent run only targets the inconclusives
- All results displayed

# Agenda

---

- Level I Training
- Level II Training
  - AutoCheck Directives and Commands
  - Considerations when running AutoCheck
  - Status flow and debugging features with AutoCheck
  - FPGA flows with AutoCheck
  - Lab
- Reference

# Considerations When Running AutoCheck

---

- The sweet spot for running AutoCheck is at the block or unit level
  - Module level runs are fine as well
- AutoCheck makes use of formal technology
  - Thus it has some of the same limitations as the formal tool has when it comes to capacity and runtimes
  - The more states in the design the longer the runtimes and memory needed
  - Running at the chip level could require hours of runtime and 10's of Gbits of memory (depending on design size of course)
- If you run into resource and time limitations on your AutoCheck runs the following ideas may help



# Improving Performance with AutoCheck

- **Note: Improving performance will most likely reduce the results you achieve with AutoCheck**
  - This is a tradeoff you will need to make
- **Method 1: Move down a level of hierarchy if needed**
  - This will reduce the state space and number of checks
- **Method 2: Disable Checks**
  - Use `autocheck disable` to disable checks that use heavy formal analysis
  - Category's to target first: BLOCK, BUS, CASE, INDEX, X
  - Example: `autocheck disable -type BLOCK*`
- **Method 3: Black Box Modules**
  - This removes logic from analysis thus reducing runtimes
  - Target blocks: silicon proven, non synthesizable, 3<sup>rd</sup> party IP

# Improving Performance with AutoCheck

## ■ Method 4: Simplify the Clocks

- For multiple clock designs, run without “**netlist clock**” directives
  - AutoCheck will automatically run with all the same frequency
- Or modify the `–period` ratios to minimize the least common multiple of the clocks used
  - View the clock active edges in the `autocheck_verify.rpt` file
  - AutoCheck has some automation in this area to minimize clk ratios

## ■ Method 5: Use the Timeout Option

- Example: `qverify -c ... -do “..; autocheck verify ... -timeout 5m”`
- This is the same option as PropCheck (units: s,m,h,d; default s)

## ■ Method 6: ctrl-C

- If the process is swapping (memory max, CPU min) ctrl-C out
- The `autocheck_verify.db` will be saved and you can view results

# Inconclusives ...

- Are a natural part of formal and AutoCheck
  - Every property starts as an inconclusive
  - You can expect for large/complex designs that you will have them
  - The primary benefit of AutoCheck is in the bugs it finds
    - Minimize the time spent trying to resolve inconclusive
- Are due to
  - The initial state of the design
  - The number of inputs to the design
  - The constraints used on the design
  - The design itself
  - The algorithms used for evaluation of the design
  - The amount of memory available
- Are primarily dependent on the state space of the design
  - The number of states in the design
  - The number of states in the properties

# AutoCheck: How to Resolve Inconclusives

---

- To some degree you can use tool functionality
- You can play with initial states and constraints
- The primary way is to reduce the state space of the design
  - For AutoCheck you have no control over reduction of the property state space
  - You may have minimal ability to effect the design state space
- How you resolve inconclusives is about the same for all formal tools
  - It mostly involves effort and time on the user's part
  - The question is: How much effort/time will you commit to?

# Resolving Inconclusives: **The Easy Way**

- Run for a longer period of time
  - Recommend the `-timeout` option in most cases
  - Make sure you have enough memory
- Use the default clocking (all clocks same frequency)
  - This is most efficient for formal, allows deepest exploration
- Run with multi-core option and increase the number of processes
  - Default is 4 cores out of the box
- Targets specific checks and/or disable unneeded checks
  - Do multiple runs targeting groups of checks
- Consider running at a lower level of the hierarchy
  - Block level is the sweet spot

# Agenda

---

- Level I Training
- Level II Training
  - AutoCheck Directives and Commands
  - Considerations when running AutoCheck
  - Status flow and debugging features with AutoCheck
  - FPGA flows with AutoCheck
  - Lab
- Reference

## qverify: AutoCheck Status






---

- Status category, displayed in the left most column in the design checks tab
  - uninspected, waived, fixed, pending, bug
- RMB click on status icon to change status
- Can filter on various status by selecting/deselecting icons in the top bar of the Design Checks tab
- Can save status changes in `autocheck_verify.db`
- Can reload `autocheck.db` with current status level
  - Can output directives in case you need to run from scratch or document the status

# qverify: Status Icons

## ■ AutoCheck Status Icons

— Note: The user is free to use these icons any way they want

	<b>Uninspected:</b>	<b>Check has not been looked at: Default Status</b>
	<b>Waived:</b>	<b>Check is being waived</b>
	<b>Fixed:</b>	<b>Check will be fixed by next AC run</b>
	<b>Pending:</b>	<b>Check has been inspected but needs more analysis</b>
	<b>Bug:</b>	<b>Check has been determined to be a bug</b>

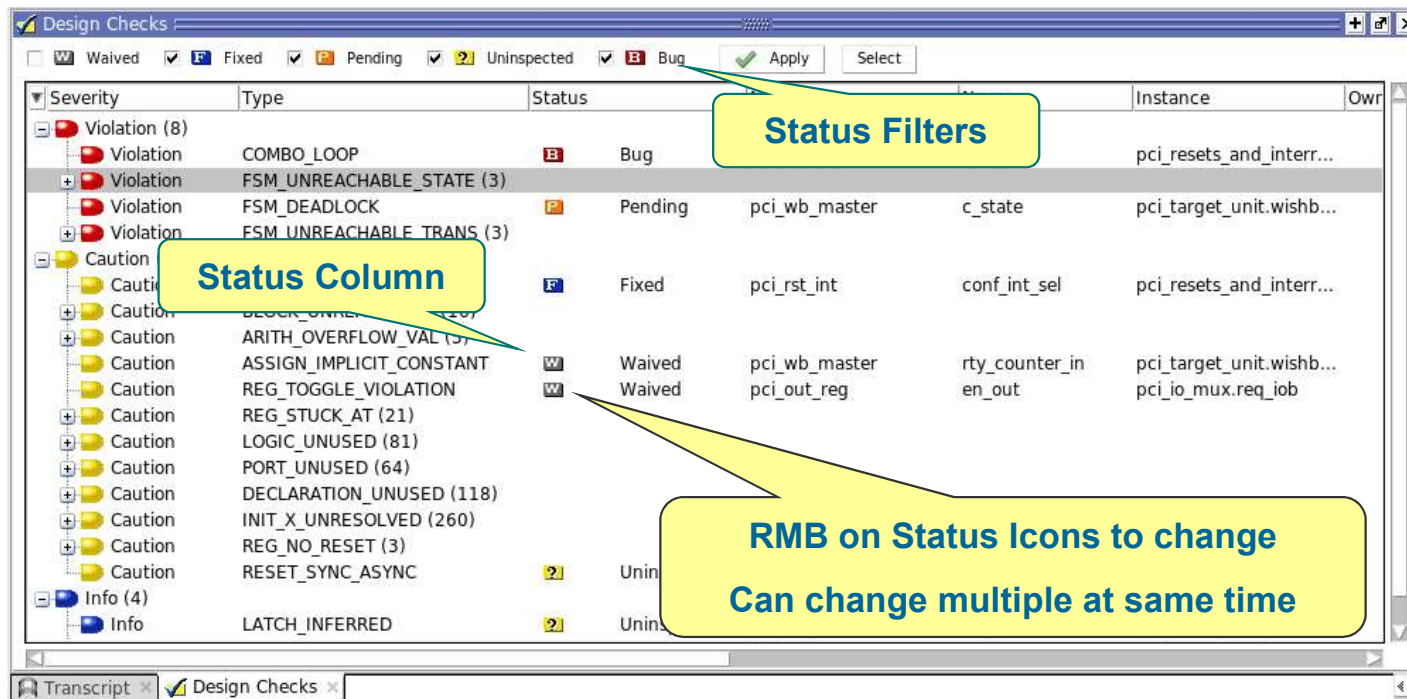


# qverify: Using the Status Column

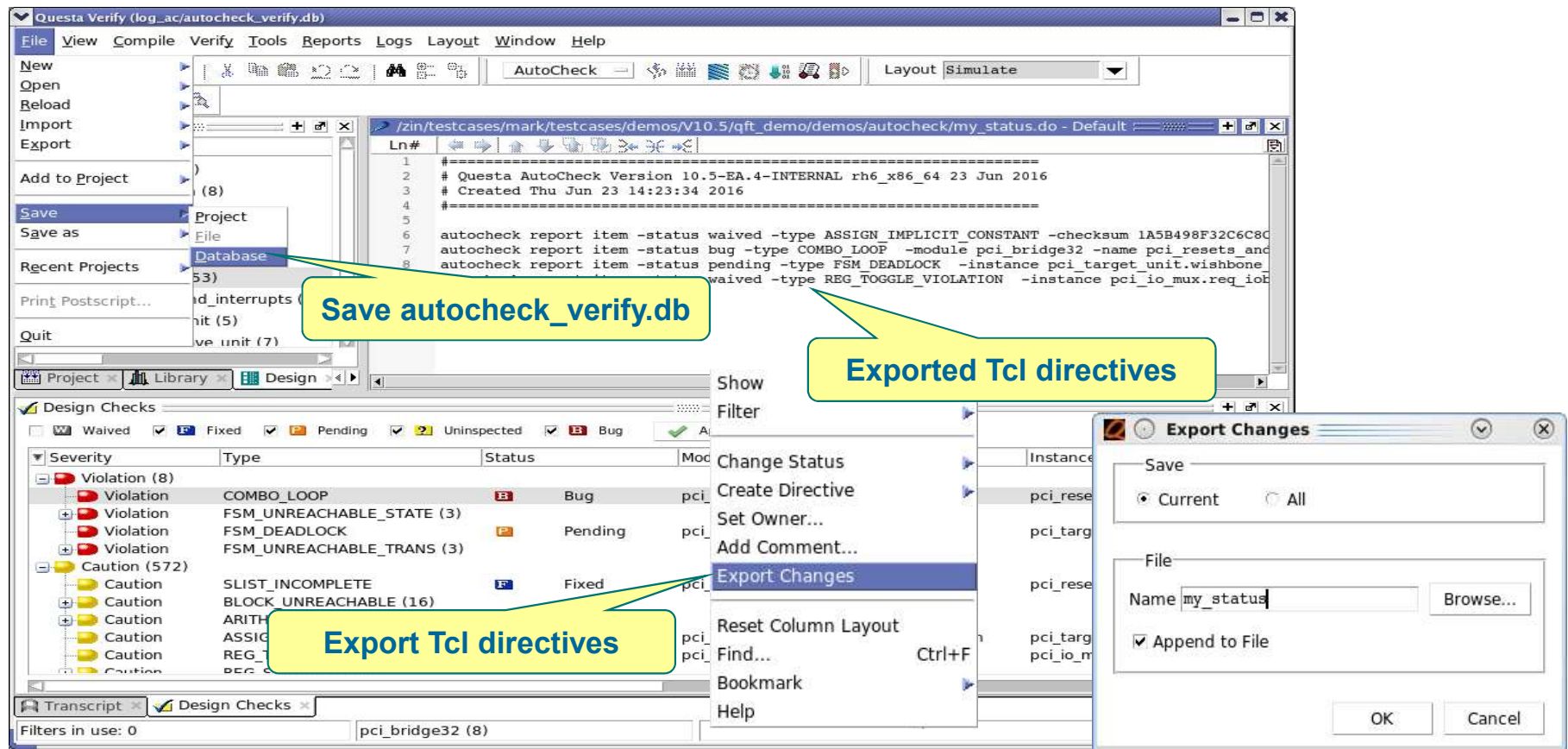
- RMB on the Status Icon to ...
  - Waive
  - Schedule to Fix
  - Mark for additional info
  - Ultimately it is up to the user on how they want to use this feature
- GUI automatically updates the Design Checks view
- Status results can be saved
  - Save in the autocheck\_verify.db
    - Previously run autocheck\_verify.db files can be loaded
  - Tcl directives for the status can be saved and used in subsequent runs
- Comments can be added with a RMB selection
  - User and reviewer names can also be added

# qverify: Design Checks Tab

- Results are grouped into categories
- Double click with LMB to debug (or select with RMB)

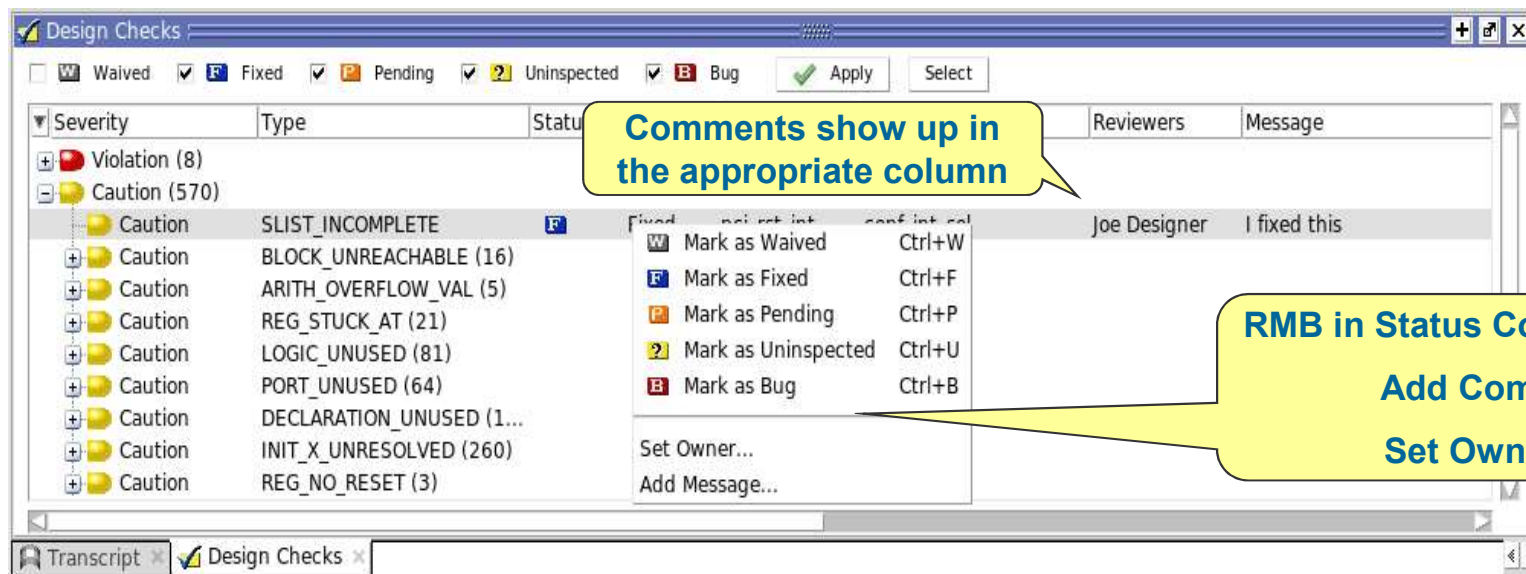


# qverify: Save Status in .db or .tcl



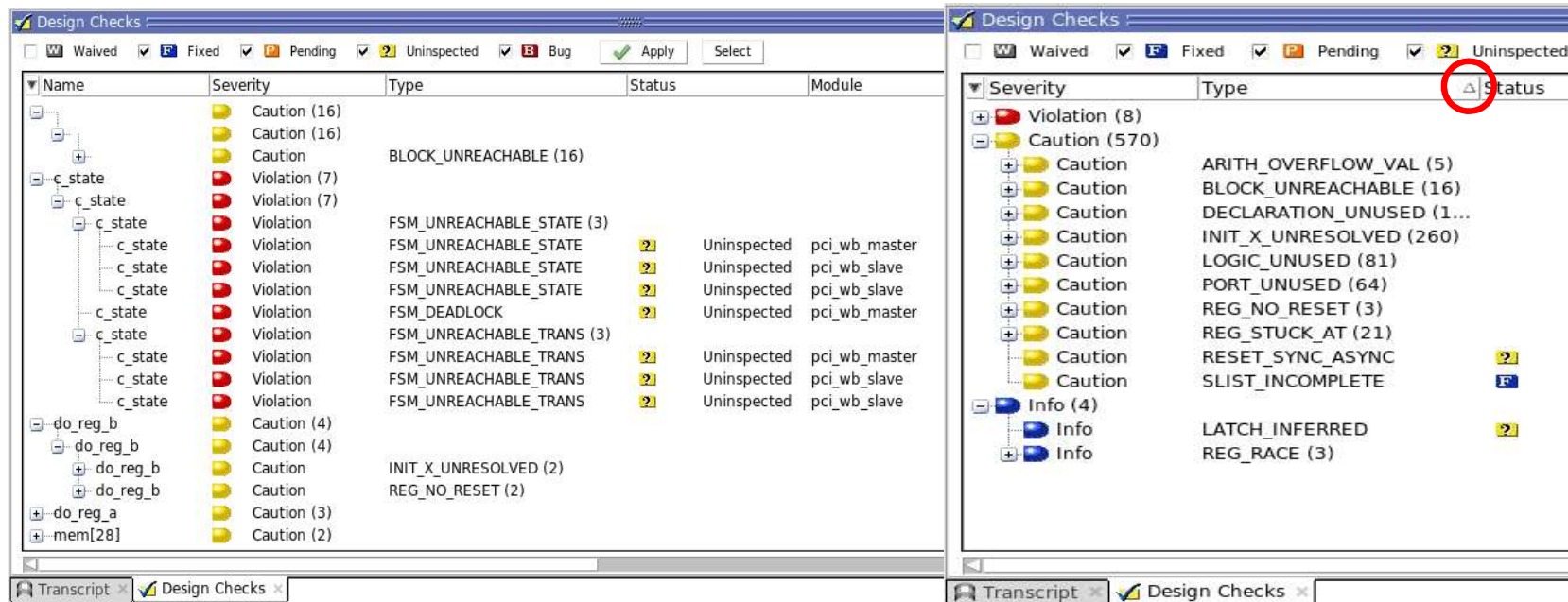
# qverify: Modify Status/Owner/Comment

- RMB in the Design Checks tab allows you to modify the status and add comments or owner. These show up in the appropriate column — RMB in the status column also brings up a window



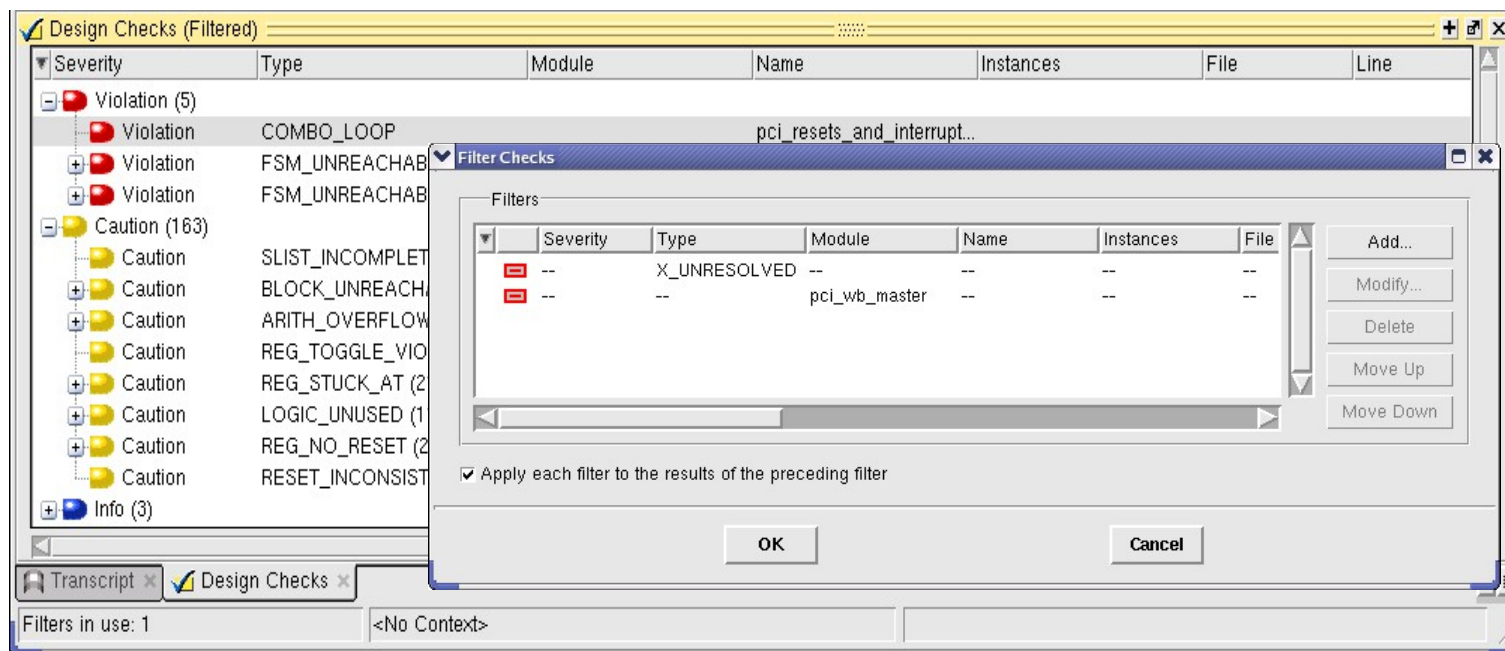
# qverify: Design Checks Sort by Columns

- Drag columns, sort by left most column  
— Sort by Name, Module etc.
- Click a column to sort alphabetically in that column



# qverify: Filtering

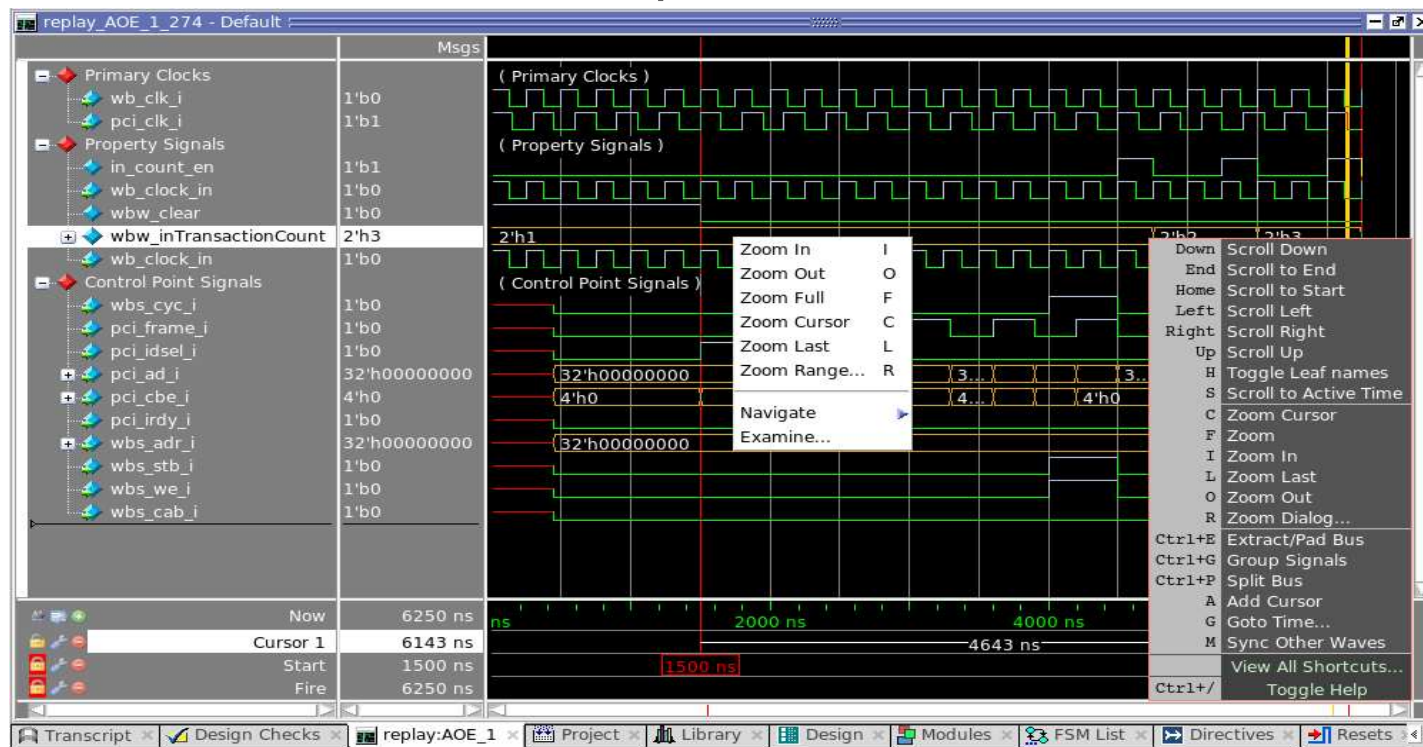
- Filter on various objects
- Design Checks header color changes when filtered
  - Lower left hand corner shows how many filters in use





# qverify : Waveform Hot Keys

- In the waveform view type: Ctrl + /
  - RMB window also shows Hot Keys

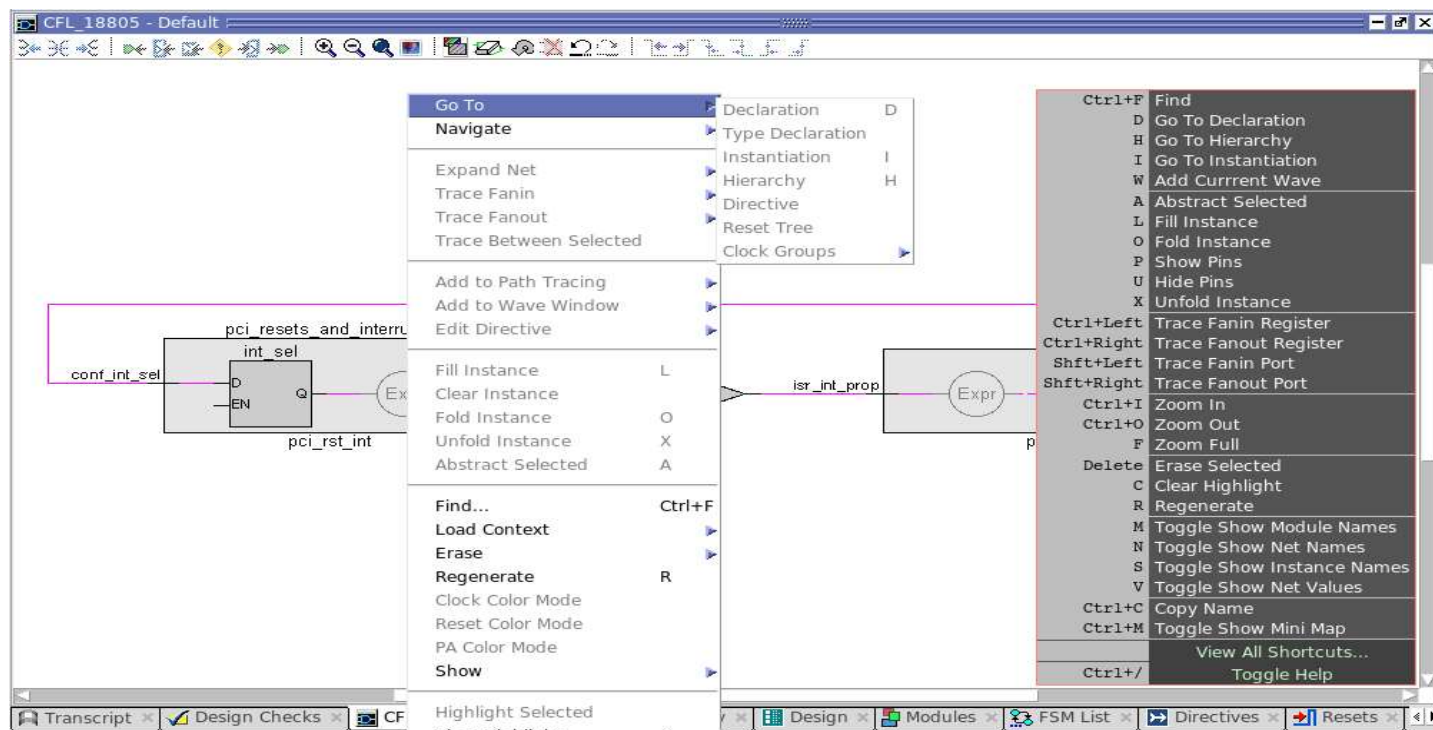


© 2020 Mentor Graphics Corporation

**Mentor**  
A Siemens Business

# qverify : Schematic Hot Keys

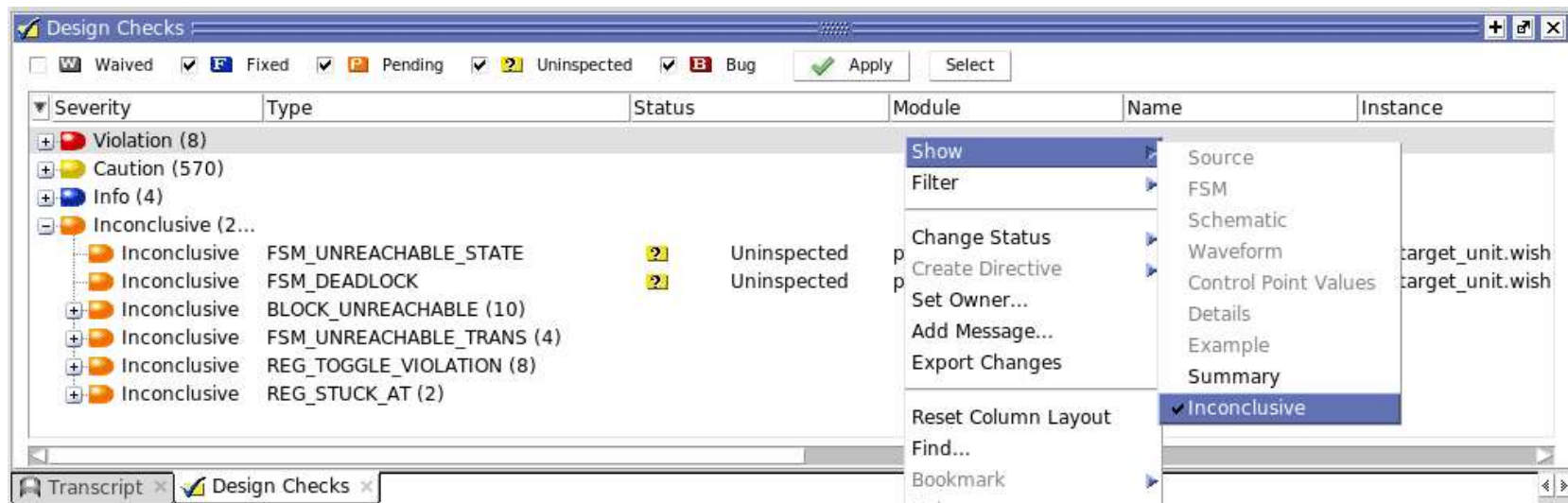
- In the schematic view type: Ctrl + /
  - RMB window also shows Hot Keys





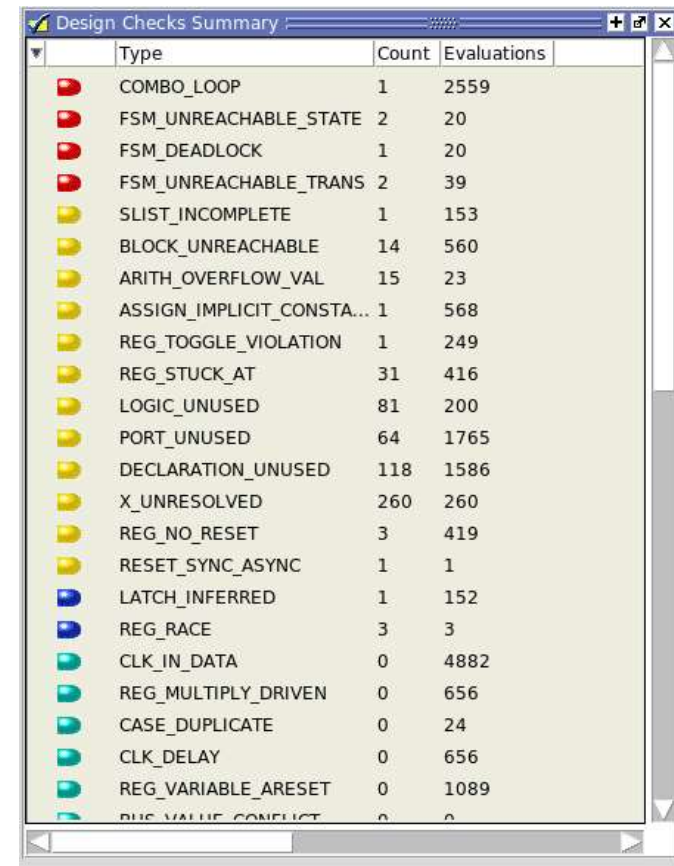
# qverify: Inconclusives

- To display Inconclusives in Design Checks tab:
  - RMB: Show -> Inconclusive



# qverify: Design Checks Summary

- Shows all Design Checks results summarized
- The Design Checks tab only shows actionable results
- From Design Checks tab
  - RMB: Show -> Summary



The screenshot shows a window titled "Design Checks Summary" with a table of design checks. The table has four columns: Type, Count, and Evaluations. The rows list various design checks with their respective counts and evaluation counts. The checks are color-coded: red for errors, yellow for warnings, and blue for information.

Type	Count	Evaluations
COMBO_LOOP	1	2559
FSM_UNREACHABLE_STATE	2	20
FSM_DEADLOCK	1	20
FSM_UNREACHABLE_TRANS	2	39
SLIST_INCOMPLETE	1	153
BLOCK_UNREACHABLE	14	560
ARITH_OVERFLOW_VAL	15	23
ASSIGN_IMPLICIT_CONSTA...	1	568
REG_TOGGLE_VIOLATION	1	249
REG_STUCK_AT	31	416
LOGIC_UNUSED	81	200
PORT_UNUSED	64	1765
DECLARATION_UNUSED	118	1586
X_UNRESOLVED	260	260
REG_NO_RESET	3	419
RESET_SYNC_ASYNC	1	1
LATCH_INFERRED	1	152
REG_RACE	3	3
CLK_IN_DATA	0	4882
REG_MULTIPLY_DRIVEN	0	656
CASE_DUPLICATE	0	24
CLK_DELAY	0	656
REG_VARIABLE_ARESET	0	1089
DUP_VALUE_CONFLICT	0	0

# Agenda

---

- Level I Training
- Level II Training
  - AutoCheck Directives and Commands
  - Considerations when running AutoCheck
  - Status flow and debugging features with AutoCheck
  - FPGA flows with AutoCheck
  - Lab
- Reference

# Running AutoCheck on FPGA Designs

- Similar flow to what is done with CDC can be used
- Make use of pre-compiled libraries (Xilinx, Altera)
  - Use the netlist fpga command to point to the correct one
  - Example: netlist fpga –vendor Xilinx –library vivado
  - Put this at the beginning of the do file before the compile command
- Most gate level features are supported
- Anything like a memory or more complicated will be BB'd
  - Not synthesizable
- Unlike CDC which uses HDM models, AutoCheck needs functionality
  - BB outputs are controllable, so some checks may be impacted
  - The more rtl the better

# Xilinx Tcl app Store: [write\\_questa\\_autocheck\\_script](#)

- ❑ Enable Xilinx Users to easily run AutoCheck on their FPGA designs
- ❑ AutoCheck Vivado App will be integrated into the Xilinx IDE
- ❑ Once installed, the AutoCheck app appears like built-in vivado command
- ❑ The app uses Xilinx Tcl APIs to extract the design files, compilation options and constraints files from the Vivado project
- ❑ The app generates scripts/Makefile to run AutoCheck (and uses "netlist fpga")

[/2019\\_install/share/fpga\\_libs/Xilinx/write\\_questa\\_autocheck\\_script.tcl](#)

```
# Usage: write_questa_autocheck_script <top_module> [-output_directory <output_directory>] [-use_existing_xdc]
#####
# write_questa_autocheck_script.tcl (Routine for Mentor Graphics Questa AutoCheck Application)
#
# Script created on 12/20/2016 by Islam Ahmed (Mentor Graphics Inc) &
#                               Ravi Kurlagunda
#
#####

namespace eval ::tclapp::mentor::questa_autocheck {
    # Export procs that should be allowed to import into other namespaces
    namespace export write_questa_autocheck_script
}

proc ::tclapp::mentor::questa_autocheck::matches_default_libs {lib} {
    # Summary: internally used routine to check if default libs used

    # Argument Usage:
    # lib: name of lib to check if default lib

    # Return Value:
    # 1 is returned when the passed library matches on of the names of the default libraries

    # Categories: xilixtclstore, mentor, questa_autocheck

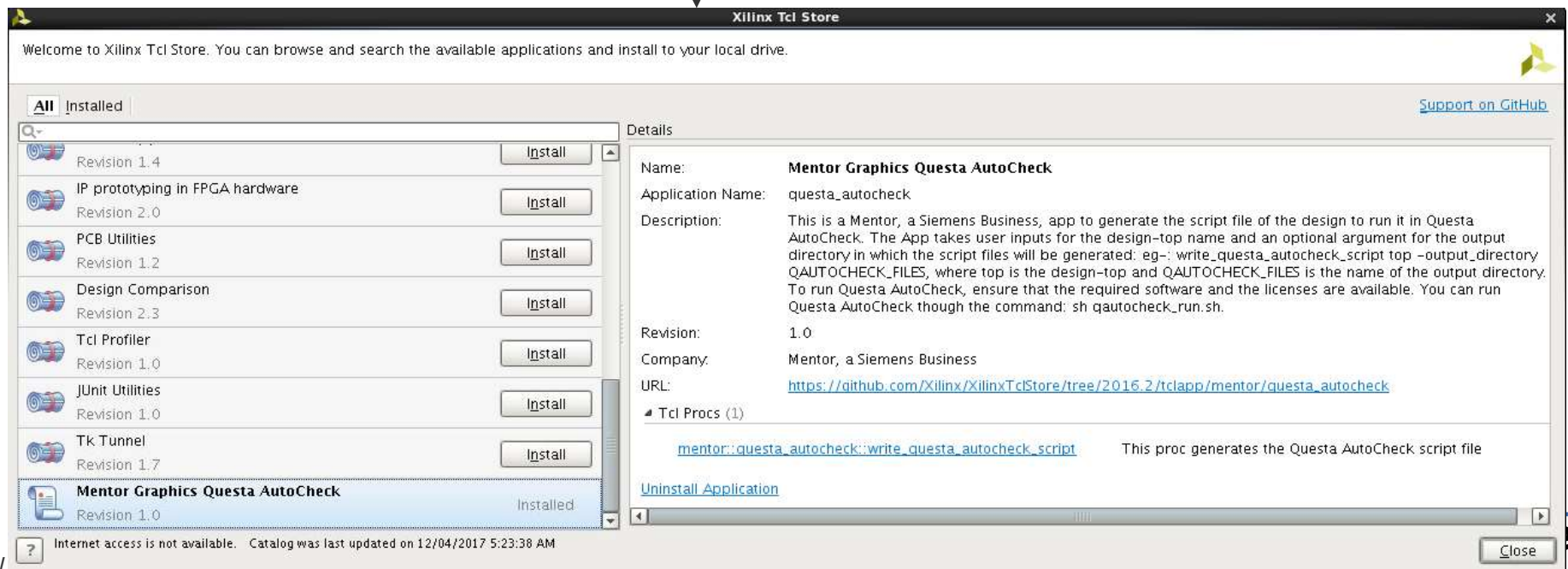
    regsub ".*" $lib {} lib
    if {[string match -nocase $lib "xil_defaultlib"]} {
        return 1
    }
}
```

# Xilinx Tcl app Store: `write_questa_autocheck_script`

## How to run

### Use Model #1

1. Install Questa AutoCheck app
2. Open Vivado project (.xpr) file
3. Invoke the AutoCheck proc from the Vivado Tcl Console



# Xilinx Vivado AutoCheck Button:

## Run Questa AutoCheck through Vivado GUI Button

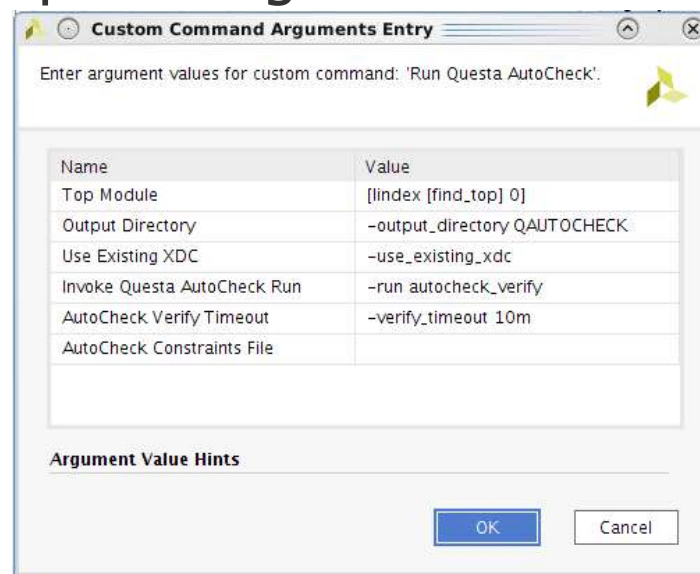
- To simplify the use model and make it more user friendly, a button is added to the Vivado GUI to invoke Questa AutoCheck scripts generation and run AutoCheck
- First install the button through the following script in UNIX shell:  
`$QHOME/share/fpga_libs/Xilinx/setup_qautocheck_vivado_button.csh`
  - This updates the users ~/.Xilinx directory so when Vivado is invoked the AutoCheck button is visible.
- Once the Vivado GUI is invoked, the Questa AutoCheck button will be there



# Xilinx Vivado AutoCheck Button:

## [write\\_questa\\_autocheck\\_script](#)

- Once the user clicks the Questa AutoCheck button, he will be requested to fill the required arguments to run.
- The default ones are:



- Once, the user presses OK, the scripts will be generated, Questa AutoCheck run will be invoked and GUI will be loaded for debug



# Agenda

---

- Level I Training
- Level II Training
  - AutoCheck Directives and Commands
  - Considerations when running AutoCheck
  - Status flow and debugging features with AutoCheck
  - FPGA flows with AutoCheck
  - Lab
- Reference

# Questa AutoCheck Lab

---

- Copy the AutoCheck lab over into your work area
  - `cp -r $MYINSTALL/share/examples/labs/autocheck .`
- Run the lab
  - `cd autocheck`
  - Pick what language (vlog or vhdl) you want to run
  - Follow lab instructions in the doc directory (use `acroread` to view .pdf file)
- Follow the instructions in the lab document(pick appropriate language):
  - Questa\_AutoCheck\_Lab\_vhdl\_v1.pdf
  - Questa\_AutoCheck\_Lab\_vlog\_v1.pdf
  - Do the Level II section of the lab

# Agenda

---

- Basic Training
- Advanced Training
- Reference
  - Description of AutoCheck design checks

# ARITH\_OVERFLOW\_SUB

- Checks for overflow that occurs within a sub-expression
- Contrasts with standard arithmetic overflow that checks for overflow as data passes from RHS to LHS

—enable with:

**autocheck enable -type ARITH\_OVERFLOW\_SUB**

- Example:

```
reg [3:0] A, C, E;  
reg BOOL;  
always @(posedge clk)  
if (en)  
    BOOL = (A + C > E);
```

Debug:






Formal Check

Notes: OFF by default

# ARITH\_OVERFLOW\_VAL

- Checks cases where an assignment overflows the range of the LHS variable
  - enable with:  
**autocheck enable -type ARITH\_OVERFLOW\_VAL**
- Example:

```
reg [3:0] a, b, c;  
reg [4:0] var ;  
always @(posedge clk or posedge rst)  
if (rst) var <= 5'b00000;  
else var[4:2] <= a + b - c;
```

Debug:   

Formal Check

Notes: OFF by default

# ARITH\_ZERO\_DIV

- Checks for cases when division operation divides by 0
- Example:

```
reg [2:0] a;  
reg [3:0] b, var ;  
...  
var <= a / b + 4'sb0001;
```

Evaluates to 0

Debug: 

Formal Check

Notes: -

# ARITH\_ZERO\_MOD

- Checks if the divisor (RHS) of a modulus expression was 0 during evaluation
- Example:

```
int a, b, c, var;  
...  
var <= c % a - b;
```

Evaluates to 0

Debug: 

Formal Check

Notes: -



# ASSIGN\_IMPLICIT\_CONSTANT

- Checks for RHS of assignments that includes a non-constant expression, but the statement only assigns a constant value
- Example:

```
reg a, b, q;  
...  
if (a == 1'b0)  
    q <= a;  
else  
    q <= b;
```

a is always 1'b0 in this assignment

Debug: 

Netlist Check

Notes: -



# BLOCK\_UNREACHABLE

- Checks for blocks of code that can't be sensitized and therefore never executed
  - Also known as deadcode check
- Example:

```
reg [1:0] R;  
always @* begin  
    if (a)      R = 2'b00;  
    else if (b) R = 2'b01;  
    else        R = 2'b11;  
end
```

```
reg T;  
always @* begin  
    T = 1'bX;  
    case (R)  
        2'b00:    T = 1'b0;  
        2'b01:    T = 1'b1;  
        2'b10:    T = 1'b1;  
        2'b11:    T = 1'b0;  
    endcase  
end
```

Statement is unreachable

Debug: 

Formal Check

Notes: -

# BUS\_MULTIPLY\_DRIVEN

- Checks for a bus or signal that has more than one active driver in the same cycle
  - even if drivers are all driving the same value
- Example:

```
wire [7:0] bus;  
assign bus = enable1 ? data1 : 8'bzzzz_zzzz;  
assign bus = enable2 ? data2 : 8'bzzzz_zzzz;  
...  
enable1 <= 1'b1;  
enable2 <= 1'b1;
```

Debug: 

Formal Check

Notes: -



# BUS\_UNDRIVEN

- Checks for a bus or signal that has no driver in a particular cycle
- Example:

```
wire [7:0] bus;  
assign bus = enable1 ? data1 : 8'bzzzz_zzzz;  
assign bus = enable2 ? data2 : 8'bzzzz_zzzz;  
...  
enable1 <= 1'b0;  
enable2 <= 1'b0;
```

Debug: 



Formal Check

Notes: -

# BUS\_VALUE\_CONFLICT

- Checks for a bus or signal that has multiple active drivers and which are driving conflicting values in a cycle
- Example:

```
wire [7:0] bus;  
assign bus = enable1 ? data1 : 8'bzzzz_zzzz;  
assign bus = enable2 ? data2 : 8'bzzzz_zzzz;  
...  
enable1 <= 1'b1;  
enable2 <= 1'b1;  
data1 <= ~data2;
```

Debug: 



Formal Check

Notes: -

# CASE\_DEFAULT

- Checks if the “default” or “when others” item of a case statements ever sensitizes in a cycle
  - enable with:  
**autocheck enable -type CASE\_DEFAULT**
- Example:

```
case (a)
  0 : q <= b;
  1 : q <= c;
  2 : q <= d;
  default : e;
endcase
```

Debug: 



Formal Check

Notes: Off by default

# CASE\_DUPLICATE

- Checks for duplicate case items
- Example:

```
case (a)
  0 : q <= b;
  1 : q <= c;
  2 : q <= d;
  0 : q <= e;
endcase
```

Debug: 

RTL Check

Notes: -



# CASE\_FULL

- Checks that each time the case statement is evaluated a case item expression matches the predicate expression value
  - priority case()
  - case() // synopsys full\_case
- Example:

```
priority case (a)
  0 : q <= b;
  1 : q <= c;
  2 : q <= d;
endcase
```

a must always be 0, 1, or 2

Debug: 

Formal Check



Notes: vlog only

# CASE\_PARALLEL

- Checks that each time the case statement is evaluated, only one case item expression matches the predicate expression value
  - unique case()
  - case() // synopsys parallel\_case
- Example:

```
priority case (a)
  0 : q <= b;
  1 : q <= c;
  2 : q <= d;
  s : q <= d;
endcase
```

s can never be 0, 1, or 2

Debug:    
Formal Check  
Notes: vlog only



# CLK\_DELAY

- Checks if a continuous clock assignment has a delay factor in it
- Example:

```
assign #20 clkA = clkB & gate;  
reg r;  
always @(posedge clkA)  
    r <= rst ? 1'b0 : d;
```

Debug: 

RTL Check

Notes: -



# CLK\_IN\_DATA

- Checks if a clock is used in the data input of a state element (other than it's own clock)
- Example:

```
assign data = clk1 ^ i1;  
reg r;  
always @(posedge clk2)  
    if (rst)  
        r <= 1'b0;  
    else  
        r <= data;
```

Debug: 

Netlist Check

Notes: -



# COMBO\_LOOP

- Checks if a feedback loop has only combinatorial logic
  - Can also take open latches into account
- Example:

```
wire a, b, c;  
assign a = c & i1;  
assign b = ~a;  
assign c = (i2 | i3) & b;
```

Debug: 

Netlist Check

Notes: -



# DECLARATION\_UNDRIVEN

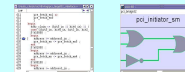
- Checks if there is a declaration which is used but not driven
  - This check points out internal signals that can be driven by formal
- Example:

```
module sub (clk, rst, din, dout);  
  input clk, rst, din;  
  output dout;  
  reg dout, r1;  
  wire gclk = clk & r1;  
  always @(posedge gclk)  
    dout <= rst ? 1'b0 : din;  
endmodule
```

Debug: 

Netlist Check

Notes: -



# DECLARATION\_UNUSED

- Checks if a declaration is defined but not used
  - Logic can be removed without effecting the external behavior of the model
- Example:

```
module sub (clk, rst, din0, din1, dout);  
  input clk, rst, din0, din1;  
  output dout;  
  reg dout;  
  wire w1 = din1;  
  always @(posedge clk) begin  
    dout <= rst ? 1'b0 : din0;  
  end  
endmodule
```

Debug: 

Netlist Check

Notes: -

# DECLARATION\_UNUSED\_UNDRIVEN

- Checks if a declaration is defined and not used and not driven
  - Logic can be removed without effecting the external behavior of the model
  - Enable with: `autocheck enable -type DECLARATION_UNUSED_UNDRIVEN`
- Example:

```
module sub (clk, rst, din0, din1, dout);  
  input clk, rst, din0, din1;  
  output dout;  
  reg dout, r1;  
  always @(posedge clk) begin  
    dout <= rst ? 1'b0 : din0;  
  end  
endmodule
```

Debug:    
Netlist Check  
Notes: OFF by default

# FSM\_DEADLOCK\_STATE

- Checks FSM which has a state from which it eventually cannot transition
  - Compile time check, module level check
- Example:

```
if (rst) state <= S0;  
else case(state)  
    S0 : state <= S1;  
    S1 : state <= S2;  
    S2 : state <= S2;  
endcase
```

S2 state is deadlocked

Debug: 

Formal Check

Notes: -







# FSM\_LOCKOUT\_STATE

- Checks for states which an FSM transitions from but cannot return
  - Enable with: **autocheck enable -type FSM\_LOCKOUT\_STATE**
- Example:

```
if (rst) state <= S0;  
else case(state)  
  S0 : state <= S1;  
  S1 : state <= S2;  
  S2 : state <= S1;  
endcase
```

S0 is locked out

Debug:      
Formal Check  
Notes: OFF by default



# FSM\_STUCK\_BIT

- Checks for FSM state variable bits that don't toggle  
— ie. stuck FSM bits
- Example:

```
reg [2:0] state;  
if (rst) state <= 3'b000;  
else case(state)  
    3'b000 : state <= 3'b001;  
    3'b001 : state <= 3'b010;  
    3'b010 : state <= 3'b000;  
endcase
```

Debug:



Netlist Check

Notes: -

# FSM\_UNREACHABLE\_STATE

- Checks for FSM states that cannot be reached
  - The state can't be transitioned to
- Example:

```
if (rst) state <= S0;  
else case(state)  
    S0 : state <= S1;  
    S1 : state <= S0;  
    S2 : state <= S0;  
endcase
```

S2 state is unreachable

Debug: 

Formal Check

Notes: -



# FSM\_UNREACHABLE\_TRANS

- Checks for a FSM that has a state transition that cannot be sensitized
- Example:

```
if (rst | c) state <= S0;  
else case(state)  
    S0 : state <= S1;  
    S1 : state <= S2;  
    S2 : state <= c ? S1 : S0;  
endcase
```

S2 -> S1 transition unreachable

Debug: 

Formal Check

Notes: -



# FUNCTION\_INCOMPLETE\_ASSIGN

- Checks that a function output is always assigned a value
  - enable with:  
**autocheck enable -type FUNCTION\_INCOMPLETE\_ASSIGN**
- Example:

```
function foo;  
  input a;  
  if (a)  
    foo = a;  
endfunction
```

Function is not assigned a value when a = 0

Debug: 

Netlist Check

Notes: Off by default

# INDEX\_ILLEGAL

- Checks that each time it's evaluated, an array index is in the legal range
- Example:

```
reg [7:0] v;  
always @(posedge clk)  
if (rst)  
    v[0] <= 1'b0;  
else  
    v[a] <= c;
```

**a = 8 is illegal**

Debug: 

Formal Check

Notes: -



# INDEX\_UNREACHABLE

- Checks that all locations of a vector or memory can assume all possible values

- May indicate over constrained logic (stuck bit)

- Example:

```
logic [1:0] mem [0:3][0:3];  
wire b = addr1 > 1;  
always @(posedge clk)  
    if (b) begin  
        mem[addr1][addr2] <= vec1;  
    end
```

**mem[0][addr2] and mem[1][addr2] are not accessible**

Debug: 

Netlist Check

Notes: -

# INIT\_X\_OPTIMISM

- Reports cases where simulation state value (0 or 1) doesn't match what happens in the real design coming out of initialization


- Example:

```
reg rega, soft_rstn;  
always @(posedge clk) begin  
    if (ld) soft_rstn <= dat[0]; end  
always @(posedge clk) begin  
    if (!soft_rstn)  
        rega <= 1'b0;  
    else  
        rega <= cstate[0];  
end
```

soft\_rstn not initialized, is X

Simulation initializes rega to cstate[0]  
(which is initialized to 1)

H/W can assign either path so rega  
should be X after initialization

Debug: 

Notes: -



# INIT\_X\_PESSIMISM

- Reports cases where simulation state value is X but in the real design the value will be initialized (0 or 1)
- Example:

```
reg rega, regb, regc;  
always @(posedge clk) begin  
    rega <= !rega; regb <= rega; end  
always @(posedge clk or negedge rstn) begin  
    if (!rstn)  
        regc <= rega & regb;  
    else  
        regc <= d;  
end
```

Simulation initializes this to X  
H/W initializes it to 0

Debug: 

Notes: -





# INIT\_X\_UNRESOLVED

- Checks that a register has an X in it after the initialization sequence has completed
  - Doesn't list memories, those are shown in INIT\_X\_UNRESOLVED\_MEM
- Example:

```
wire n1 = c && ~d;  
reg r, q;  
always @(posedge clk) begin  
    r <= d && n1;  
    q <= d;  
end
```

r resolved to 1'b0 after init sequence  
q is still 1'bx after init sequence completion so is unresolved

Init Sequence:

rstn = 1'b0

##5

rstn = 1'b1

Debug: 

Notes: -

# INIT\_X\_UNRESOLVED\_MEM

- Checks that a Memory has an X in it after the initialization sequence has completed
  - Same as INIT\_X\_UNRESOLVED, only showing memories
- Example:

```
wire n1 = c && ~d;  
reg r;  
reg [31:0] q [0:7];  
always @(posedge clk) begin  
    r <= d && n1;  
    q[0] <= data;  
end
```

r resolved to 1'b0 after init sequence  
q[0] is still 32'dx after init sequence completion so is unresolved

## Init Sequence:

rstn = 1'b0

##5

rstn = 1'b1

Debug: 

Notes: -



# LATCH\_INFERRED

- Checks when a latch is inferred by synthesis
  - enable with:  
**autocheck enable -type LATCH\_INFERRED**
- Example:

```
always @*  
    if (enable)  
        q <= a;
```

Debug: 

RTL Check

Notes: Off by default



# LOGIC\_UNDRIVEN

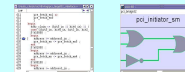
- Checks if there is logic not driven from the fanout of the primary inputs
  - This check points out internal signals that can be driven by formal
  - Typically this falls into the category of bits of a bus which aren't driven
- Example:

```
module sub (clk, rst, din, dout);  
  input clk, rst, din;  
  output dout; reg dout;  
  wire [1:0] r1;  
  assign r1[1] = din;  
  wire gclk = clk & r1[0];  
  always @(posedge gclk)  
    dout <= rst ? 1'b0 : r[1];  
endmodule
```

Debug: 

Netlist Check

Notes: -



# LOGIC\_UNUSED

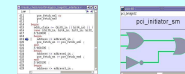
- Checks if logic is not in the fanin of its outputs
  - Logic can be removed without effecting the external behavior of the model
  - Typically this is displayed as bits of a bus
- Example:

```
module sub (clk, rst, din0, din1, dout);  
input clk, rst, din0, din1;  
output dout;  
reg [1:0] r1;  
always @* r1[1] <= din1;  
always @(posedge clk) begin  
    dout <= rst ? 1'b0 : r1[1];  
    r1[0] <= rst ? 1'b0 : din1;  
end  
endmodule
```

Debug: 

Netlist Check

Notes: -



# ONE\_COLD

- Checks each cycle that a vector of signals is “one cold”
  - Only works with Synopsys pragmas
  - No busses can be specified, only concatenation of bits
- Example:

...

```
// synopsys one_cold "enable1, enable2, enable3"
```

...

Debug: 



Formal Check

Notes: -

# ONE\_HOT

- Checks each cycle that a vector of signals is “one hot”
  - Only works with Synopsys pragmas
  - No busses can be specified, only concatenation of bits
- Example:

...

```
// synopsys one_hot "enable1, enable2, enable3"
```

```
// synopsys one_hot "state[2],state[1],state[0]"
```

...

Debug: 

Formal Check

Notes: -



# PORT\_UNDRIVEN

- Checks if there is output or inout ports which aren't driven
  - This check points out ports that can be driven by formal
- Example:

```
module sub (clk, rst, din, dout, r1);  
  input clk, rst, din;  
  output dout, r1;  
  reg dout;  
  always @(posedge clk)  
    dout <= rst ? 1'b0 : din;  
endmodule
```

Debug: 

Netlist Check

Notes: -





# PORT\_UNUSED

- Checks if input or inout ports aren't used
  - Ports can be removed without effecting the external behavior of the model
- Example:

```
module sub (clk, rst, din0, din1, dout);  
  input clk, rst, din0, din1;  
  output dout;  
  reg dout;  
  always @(posedge clk) begin  
    dout <= rst ? 1'b0 : din0;  
  end  
endmodule
```

Debug: 

Netlist Check

Notes: -

# REG\_MIXED\_ASSIGNNS

- Checks for signals driven by both blocking and non-blocking assignments
- Example:

```
always @(posedge clk or posedge rst)
  if (rst)
    q = 1'b0;
  else
    q <= d;
```

Debug: 

RTL Check

Notes: vlog only



# REG\_MULTIPLY\_DRIVEN

- Checks for registers & latches driven from multiple always blocks or processes
- Example:

```
always @(posedge clk)
  r2 <= rst ? 0 : 4'b0000;
always @(negedge clk)
  r2 <= rst ? 0 : d1;
```

Debug: 

Netlist Check

Notes: -



# REG\_NO\_RESET

- Checks for signals that are not reset in a process that has a reset  
— enable with:  
**autocheck enable -type REG\_NO\_RESET**
- Example:

```
always @(posedge clk or posedge rst)
  if (rst)
    q <= 1'b0;
  else begin
    q <= d;
    r <= e;
  end
```

**r has not been reset**

Debug:



RTL Check

Notes: OFF by default



# REG\_RACE

- Checks for register race conditions caused by two always blocks
  - enable with:  
**autocheck enable -type REG\_RACE**
- Example:

```
always @(posedge clk)
  c = b;
always @(posedge clk)
  d <= c;
```

**d is loaded with either b or  
previous b value**

**Debug:** 

**RTL Check**

**Notes: OFF by default**

# REG\_STUCK\_AT

- Checks for registers & latches that have one or more bits stuck at a constant value
- Example:

```
reg [3:0] out;  
wire [3:0] dat = {1'b0,data[2:0]};  
always @(posedge clk)  
  if (rst)  
    out <= 4'b0000;  
  else  
    out <= dat;
```

out[3] is stuck at 1'b0

Debug: 

Formal Check

Notes: -

# REG\_TOGGLE\_VIOLATION

- Checks for registers & latches that have one or more bits that can be stuck in a dead-end value
- Example:

```
wire w = !q | b;
```

```
always @(posedge clk)
```

```
  if (w || a)
```

```
    q <= 1'b0;
```

```
  else
```

```
    q <= 1'b1;
```

q cannot toggle from 0 to 1 in normal operation

Debug: 

Formal Check

Notes: -

# REG\_VARIABLE\_ARESET

- Checks for registers that have async non-constant (i.e. variable) reset/load data
  - This check is a warning about race conditions
- Example:

```
reg q;  
always @(posedge clk or posedge rst or posedge en)  
  if (rst)  
    q <= 1'b0;  
  else if (en)  
    q <= r;  
  else  
    q <= d;
```

Loading a variable

Debug: 

RTL Check

Notes: -





# RESET\_HIGH\_LOW

- Checks for signals used as both active-high and active-low resets
- Example:

```
always @(posedge clk or posedge rst)
    if (rst)    r1 <= 1'b0;
    else       r1 <= d1;
always @(posedge clk or negedge rst)
    if (~rst)  r2 <= 1'b0;
    else      r2 <= d2;
```

Debug: 

RTL Check

Notes: -



# RESET\_SYNC\_ASYNC

- Checks if reset is used both asynchronously and synchronously
- Example:

```
always @(posedge clk)
    if (~rstn) r1 <= 1'b0;
    else      r1 <= d1;
always @(posedge clk or negedge rstn)
    if (~rstn) r2 <= 1'b0;
    else      r2 <= d2;
```

Debug: 

RTL Check

Notes: -

# SLIST\_INCOMPLETE

- Checks that the sensitivity list for a process is complete
- Example:

```
always @(r or d)
  if (clr)
    q <= 1'b0;
  else if (en)
    q <= r;
  else
    q <= d;
```

clr and en are missing from the sensitivity list

Debug: 

RTL Check

Notes: -

# X\_ASSIGN\_REACHABLE

- Checks for literal X assignments in RTL that can be sensitized by a set of known driven values
  - Using synthesis semantics
- Example:

```
reg q;  
  
always @(posedge clk or negedge rstn)  
  if (!rstn)   q <= 1'b0;  
  else if (en) q <= 1'b1;  
  else        q <= 1'bx;
```

This can be sensitized if en = 1'b0

Debug: 

Formal Check

Notes: -

**Q&A**

**Mentor<sup>®</sup>**  
A Siemens Business

[www.mentor.com](http://www.mentor.com)