ZeBu® Synthesis Verification User Guide

Version V-2024.03-1, July 2024

SYNOPSYS®

Feedback

Contents

Feedback

Preface

This section discusses the following topics:

About This Book

The **ZeBu® Synthesis Verification Guide** describes the **zFmCheck** tool. This tool validates whether the ZeBu synthesis engine has correctly synthesized Verilog modules, VHDL entity, or architecture pairs.

Audience

This guide is an internal document and is intended for Synopsys AEs.

Contents of This Book

The **ZeBu® Synthesis Verification Guide** has the following chapters:

Chapter	Describes...
Introduction to zFmCheck	Introduces **zFmCheck** tool
Debugging Using zFmCheck	Describes the prerequisites for using the **zFmCheck** tool, and provides instructions to enable and invoke the tool.

Chapter	Describes...
Formality Usage with zFmCheck	Describes how to use Formality with **zFmCheck**
Simulating the Testbench	Describes how to simulate the Testbench using **zFmCheck**
Using zFmCheck for Large Design	Describes how to use **zFmCheck** for large designs
Tutorial	Discusses the best practices for using **zFmCheck**
Limitations	Provides limitations of the **zFmCheck**tool

Hardware Documentation

Document Name	Description
ZeBu Getting Started Guide	Provides brief information about Synopsys' emulation system - ZeBu.
ZeBu Power Estimation User Guide	Provides the power estimation flow and the tools required to estimate the power on a System on a Chip (SoC) in emulation.
ZeBu Server 4 Smart Z-ICE Interface User Guide	Provides physical description of the Smart Z-ICE interface and the steps to instantiate and use it on ZeBu Server 4.
ZeBu Server 4 Release Notes	Provides enhancements and limitations for new ZeBu Server 3 and ZeBu Server 4 releases.

Document Name	Description
ZeBu Server 5 Site Planning Guide	Describes planning for ZeBu Server 5 hardware installation.
ZeBu Server 5 Site Administration Guide	Provides information on administration tasks for ZeBu Server 5 hardware. It includes software installation.
ZeBu Server 5 Getting Started Guide	Provides brief information about Synopsys' ZeBu Server 5.
ZeBu Server 5 Hardware Performance Technologies User Guide	Provides information on the performance technologies available with the ZeBu Server 5 hardware architecture.

Document Name	Description
ZeBu Server 5 Release Notes	Provides enhancements and limitations for a new ZeBu Server 5 release.

Document Name	Description
ZeBu EP1 Getting Started Guide	Provides brief information about Synopsys' ZeBu EP1.
ZeBu EP1 Site Administration Guide	Provides information on administration tasks for ZeBu EP1 hardware. It includes software installation.
ZeBu EP1 Site Planning Guide	Describes planning for ZeBu EP1 hardware installation
ZeBu EP1 Hardware Performance Technologies Guide	Provides information on the performance technologies available with the ZeBu EP1 hardware architecture.
ZeBu EP1 Release Notes	Provides enhancements and limitations for the new EP1 release.

Related Documentation

Document Name	Description
ZeBu User Guide	Provides detailed information on using ZeBu.
ZeBu Debug Guide	Provides information on tools you can use for debugging.
ZeBu Debug Methodology Guide	Provides debug methodologies that you can use for debugging.
ZeBu Unified Command-Line User Guide	Provides the usage of Unified Command-Line Interface (UCLI) for debugging your design.
ZeBu UTF Reference Guide	Describes Unified Tcl Format (UTF) commands used with ZeBu.
ZeBu Power Aware Verification User Guide	Describes how to use Power Aware verification in ZeBu environment, from the source files to runtime.
ZeBu Functional Coverage User Guide	Describes collecting functional coverage in emulation.
Simulation Acceleration User Guide	Provides information on how to use Simulation Acceleration to enable cosimulating SystemVerilog testbenches with the DUT
ZeBu Verdi Integration Guide	Provides Verdi features that you can use with ZeBu. This document is available in the Verdi documentation set.

Feedback

Document Name	Description
ZeBu Runtime Performance Analysis With zTune User Guide	Provides information about runtime emulation performance analysis with zTune.
ZeBu Custom DPI Based Transactors User Guide	Describes ZEMI-3 that enables writing transactors for functional testing of a design.
ZeBu LCA Features Guide	Provides a list of Limited Customer Availability (LCA) features available with ZeBu.
ZeBu Transactors Compilation Application Note	Provides detailed steps to instantiate and compile a ZeBu transactor.
ZeBu zManualPartitioner Application Note	Describes the zManualPartitioner feature for ZeBu. It is a graphical interface to manually partition a design.
ZeBu Hybrid Emulation Application Note	Provides an overview of the hybrid emulation solution and its components.

Other Useful References

Document Name	Description
VCS User Guide	Provides information on using VCS to simulate a design.
Verdi User Guide and Tutorial P	Provides information on using Verdi.
Verdi Coverage Technology User Guide, Coverage Technology Reference Guide, Verification Planner User Guide, and Tutorial	Provides information on using Verdi to debug coverage data.

Typographical Conventions

This document uses the following typographical conventions:

To indicate	Convention Used
Program code	`OUT <= IN;`
Object names	`OUT`
Variables representing objects names	`<sig-name>`

To indicate	Convention Used
Message	Active low signal name '<sig-name>' must end with _X
Message location	`OUT` <= IN;
Reworked example with message removed	`OUT_X` <= IN;
Important Information	**NOTE:** This rule...

The following table describes the syntax used in this document:

Syntax	Description
[ ] (Square brackets)	An optional entry
{ } (Curly braces)	An entry that can be specified once or multiple times
| (Vertical bar)	A list of choices out of which you can choose one
... (Horizontal ellipsis)	Other options that you can specify

Synopsys Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

1

Introduction to zFmCheck

zFmCheck is a tool to validate that the ZeBu synthesis engine has correctly synthesized Verilog modules, VHDL entity, or architecture pairs. **zFmCheck** verification is based on the comparison between the reference and implementation models using Formality Equivalence Checking and VCS simulation technologies.

In the **zCui** compile directory, **zFmCheck** uses either the synthesizer outputs or the final the EDIF output as the implementation model and the decompiled Verilog or VHDL as the reference model.

In this guide, the term “model” is used to refer a Verilog module, a VHDL entity or an architecture pair.

This section discusses the following topics:

- Supported Options

Supported Options

zFmCheck supports the following options:

Table 1 zFmCheck Command Options

Command Options	**Description**
`-r`	Run formality equivalence check
`-v`	Run simulation check
`-o output_directory`	Use a directory to generate files, the directory is relative to the current directory
`-c remote_command`	Remote command to launch Formality
`-m <module_name>`	Verify specified modules
`-mf <modules_list_file>`	Module list file, only the modules listed in this file will be verified
`-l <hh:mm:ss>`	Time-outlimit for each job

Table 1 zFmCheck Command Options (Continued)

Command Options	Description
`-j N`	Number of Formality jobs to launch in parallel
`-d <compilation-directory>`	Path to synthesis directory. Example: `Zcui.work`
`-gencase`	Extract standalone reproduced case

Note:
To view the list of options supported with **zFmCheck**, use `zFmCheck -help`.

2

Debugging Using zFmCheck

This section discusses the following topics:

Prerequisites

- Make sure you are using same environment as for your emulation runtime with ZeBu:
  - Same `$ZEBU_ROOT`
  - Same `$VCS_HOME`
  - Latest Formality version
- Compile design

  Make sure there are no synthesis compilation errors.

Enabling zFmCheck

To enable **zFmCheck**, add the following UTF option before compiling a design in ZeBu:

```
synthesis -generate_db_for_fmcheck true
```

After the design is synthesized, **zFmCheck** can be invoked from the compile directory.

Note:

- If you only want to prove equivalence and not to build the entire design, add the `-d` option with the **zCui** invocation command to run just through the front-end.
- **zFmCheck** can be invoked when you see that the front-end is complete. After synthesis executes, it means that the front-end has completed

Invoking zFmCheck

To invoke **zFmCheck**, use the following command:

```
zFmCheck -d "zcui.work directory" -r -v -j NUMBER -c "remote_cmd" -o
 fm.out
```

The log is available at `fm.out/formality.log`.

The following is a sample summary log:

```
###################Formality summary####################
EQUIVALENT                    :    574 (98%)
DIFFERENT                     :    3 (1%)

UNKNOWN                       :    9 (2%)
    |--REF SETUP FAIL         :    3
    |--MULTIPLY DRIVEN        :    1
    |--VERIFY INCONCLUSIVE    :    1
    |--OTHERS                 :    4
  BLACKBOX
  UNVERIFIED                  :    0 (0%)
  SKIPPED                     :    0 (0%)

                                   588
####################################################
################Simulation summary#####################
Modules                       :    12
PASS                          :    8
FAIL                          :    1
ERROR/MISSING                 :    3
RUNTIME-ERROR                 :    0
####################################################
```

3

Formality Usage with zFmCheck

You must set the Formality for equivalence checking.

This section describes the following sub-topics.

- Equivalence Checking
- Formality Results

Equivalence Checking

To use Formality for equivalence checking, use the `-r` argument in the `zFmCheck` command:

```
zFmCheck -r -d zcui.work
```

This is the simplest form of the command and is only meaningful for small examples that do not require significant compute intensive resources. Additional options should be used for designs with many models.

This option instructs **zFmCheck** to prepare the reference and implementation models for use with the Formality tool. In addition, match files are created for use by Formality.

Formality Results

The Formality results are saved within the `fm_edf` sub directory and are summarized at the completion of the `zFmCheck` command. Each model has a log of the Formality run captured in the `fm_edf/job_<model>.log` file. This file is accessed to determine the state of models as follows:

- **Equivalent**: It indicates that the reference and implementation models match according to Formality. Synthesis is correct for this set. You can find the list of models within `fm_edf/equivalent.log`.
- **Different**: It indicates that the reference and implementation models do not match according to Formality. Synthesis is not correct for this set. You can find the list of models within `fm_edf/different.log`.

- **Unknown**: It indicates that Formality is unable to conclude a result for the given reference and implementation models. These results are further categorized into other sets. You can find the complete set of models within `fm_edf/unknown.log`.
    - **Timeout**: It indicates that Formality is unable to conclude in the given period. Often, this period is too short for a typical Formality proof and so simply increasing the time limit substantially on a subsequent iteration proves equivalent or different. You can find the models that fall into this category in `fm_edf/unknown_timeout.log`.
    - **Ref Setup Fail**: It indicates that Formality tool itself was unable to handle the reference model. Often, this occurs because of a bug in Formality or simply because there are language constructs that the Formality tool is unable to handle. A recent example of this is the SystemVerilog if in an event control. Models that fall in to this category can be found in `fm_edf/unknown_ref_setup_fail.log`.The only way to show results for the unknowns of this type is through constrained random testbench simulation.
    - **Multiply Driven**: It indicates that Formality tool was unable to handle the comparison due to multiply driven signals. Sometimes, you know multiple drivers present in a part of the code (for example, transactor) for which they are tolerating this. Otherwise, it is not acceptable. You can find the models that fall into this category in `fm_edf/unknown_multiply_driven.log`.The multiply driven signals are reported in `fm_edf/multi_driven_<model>.log`
    - **Verify Inconclusive**: It indicates that Formality was unable to converge and stopped in the given period. You can find the models that fall into this category in `fm_edf/unknown_verify_inconclusive.log`.
    - **Unmatched Point**: It indicates that **zFmCheck** was unable to provide an appropriate match point between the reference and implementation models. The models that fall into this category might be correctly handled in constrained random testbench simulation. You can find the models that fall into this category at `fm_edf/unknown_unmatched_point.log`.
    - **Others**: It indicates that **zFmCheck** is unable to deduce why Formality is unable to prove the equivalence. You can find models that fall in to this category at `fm_edf/unknown_others.log`
- **Blackbox**: It indicates that Formality sees the model as empty. Models of this type can be found in `fm_edf/blackbox.log`.
- **Unverified**: It indicates that Formality did not try to verify the model although it was requested to. Models of this type can be found in `fm_edf/unverified.log`. When the model is encrypted and the encryption method cannot be supported, **zFmCheck** moves the model into this category.

- **Skipped**: It indicates that **zFmCheck** has detected that the model or entity or architecture pair cannot be proved by Formality and so it is skipped. This can occur for certain memory types. Instead of Formality, constrained random simulation should be used on these models. Models of this type can be found in `fm_edf/skipped.log`.

4

Simulating the Testbench

zFmCheck uses VCS for the constrained random testbench simulation.

To use constrained random testbench simulation, use the `-v` argument to the `zFmCheck` command.

```
zFmCheck -v -d zcui.work
```

This is the simplest form of the command and is only meaningful for small examples that do not require significant compute intensive resources. Additional options should be used for designs with many models.

Using this option instructs **zFmCheck** to generate a constrained random testbench with an instance of the reference and implementation models for validation. Input stimulus from randomly generated data and output data is asserted to match between the reference and implementation models. Simulation is not exhaustive and concludes after a default set of cycles.

This section describes the following topics.

- Simulation Results
- Advanced Simulation Options

Simulation Results

Simulation results are saved in the simulation subdirectory and a summary is displayed on the screen at the completion of **zFmCheck**. Each model has its simulation compile and runtime log captured within `simulation/logs/<model>_*.log`. These files are often accessed to determine what occurred in specific to that model.

- **Pass**: This result indicates that simulation has successfully simulated the set of random data and found no differences between the reference and implementation models. You can find the models that fall into this category within the `simulation/ sim_pass.log` file.

Note:

A pass in simulation does not mean that the model is correctly synthesized. It simply means that applying the set of random stimuli for the defined set of cycles detected no difference between the reference and implementation models.

- **Fail**: This set indicates those models that failed during simulation by comparing the reference and implementation results. You can find the models that fall into this category at `simulation/sim_fail.log`. Failures in this category are concerning and could mean there is a problem in synthesis.
  - **SYN_SIM_MISMATCH**: The `SYN_SIM_MISMATCH`means there are some synthesis and simulation mismatch in the model. The mismatch reasons are as follows:
    - `SAME_RESET_SIGNAL`
    - `TREAT_PROCESS_COMPLETELY_SENSITIVE`
    - `LATCH_RACE_CONDITION`
    - `CONVERT_NONBLOCKING_TO_BLOCKING_IN_FUNCTION`
    - `VDHL_EXPLICT_ENUM_ENCODING`
    - `NON_BLOCKING_ASSIGNMENT_INSIDE_TASK_OR_FUNCTION`
- **Error/Missing**: This set indicates that there was an error in trying to produce the simulation executable. The issue could be in testbench creation, reference model compile, implementation model compile, and design elaboration. To see the models from this category, see the `simulation/sim_error.log` file.
  - **Runtime error**: This set indicates that there was an error in running the simulation. The errors in this category typically indicate a problem with the constraints. Often, the models are not able to handle the random stimulus on the input to the model whereas in the real design, the stimulus is further constrained. Errors in this category need investigation to determine if it is a real issue with the model or simply the provided stimulus is too vast. To see the models from this category, see the `simulation/sim_runtime_error.log` file.
  - **Timeout**: This set indicates that the simulation is unable to conclude within the given timeout. You can typically increase the timeout to conclude on these, but sometimes,the longer timeout does not help. You can find the list of models that fall into this category within the `simulation/sim_timeout.log` file.

Advanced Simulation Options

When constructing a simulation testbench, simulation aspects such as clock frequency,cycles to run, and register initialization can be set through **zFmCheck**. Following is a sampling of some of those options and what they control:

- `-initcycleN` : Specifies the number of initial cycles for simulation. Use this to determine how long you want the simulation to run in the initial state.
- `-gl` : Gate level memory simulation. The state machine in the gate level ensures the last written part take effect.
- `-xprop<1/0>` : Enables or disables `-xprop` in simulation.
- `-freq` : Specifies the user clock frequency in KHz for random simulation.
- `-cycleN` : Specifies the number of cycles to run the random stimulus.

5

Using zFmCheck for Large Design

For a large design, run **zFmCheck** in multiple iterations to find the set of models that need attention to a minimal set in the following sections:

- Initial Iteration (With or Without -v)
- Subsequent Iterations
- Timeout Iteration

Initial Iteration (With or Without -v)

In the first iteration, minimize the concerning models and give any details into identifiable issues. In the first iteration, you should use the `-r`, `-j`, `N`, and `-c "cmd"` options with **zFmCheck**. You might also want to add `-v` to automatically perform simulation on the set that is not equivalent in Formality.

To distribute the jobs in parallel, use `-j`. For example, use `-j 100` with a design that has 200 models and runs 100 processes where each job contains 2 models to process.

To distribute the processing onto the grid, use `-c "cmd"`. For example, use `-j 100` with a design that has 200 models and runs 100 invocations of the -c option. In addition, it distributes all 100 processes onto 100 machines where each machine processes 2 jobs per machine.

Formality runs on the entire design and then a summary is provided for all blocks that Formality (`-r`) was able to prove. For items that Formality is not able to prove equivalent, a constrained random testbench is created to stimulate with random simulation (`-v`). A simulation summary is also provided for the set of simulated models.

```
prompt> zFmCheck -r -v -j 10 -c "grid_cmd" -d zcui.work
 ###################Formality summary######################
EQUIVALENT                     : 574 (98%)
DIFFERENT                      : 3 (1%)
UNKNOWN                        : 9 (2%)
    |--REF SETUP FAIL          : 3
    |--MULTIPLY DRIVEN         : 1
    |--VERIFY INCONCLUSIVE     : 1
    |--OTHERS                  : 4
  BLACKBOX                     : 2 (0%)
  UNVERIFIED                   : 0 (0%)
  SKIPPED                         : 0 (0%)
                                        588
####################Simulation summary####################
  Modules                         :       12
  PASS                            :        8
  FAIL                            :        1
  ERROR/MISSING                   :        3
  RUNTIME-ERROR                   :        0
##########################################################
```

In this case, 588 models were processed and 574 were proven correct by Formality. The remaining 14 models, 2 are black boxes. It means, there is really nothing in those 2 models. The final 12 models are passed to constrained random simulation using VCS. In the 12 models, consider 8 models are passed, 1 failed, and 3 are not able to prove due to errors in compile or missing results.

Subsequent Iterations

Subsequent iterations are typically used to find the concerned models.

The following are some typical iteration to explain subsequent iteration set: (all are on grid with the large `-j` option)

1. Run `zFmCheck -r` with large timeout for those models that timed out. This is described in the Timeout Iteration section.
2. Run `zFmCheck -v` with mid-size timeout that are not equivalent.
   a. Set the `-j` option to the remaining set, if possible or some evenly distributed fraction of 400 models left
   b. Run `-j 100` to get 4 models per job.

Timeout Iteration

For those models that Formality fails to prove due to timeout, launch **zFmCheck** that focuses on such models. For this, use the `-l`option to increase the default timeout (that is, `-l 24:00:00`to specify a run for at least 24 hours). Some models might never converge in Formality. This iteration should minimize the results even further.

```
zFmCheck -l 12:00:00 -mf zcui.work/design/synth_Default_RTL_Group/
 formality/fm_edf/timeout.log -r -d zcui.work -j 30 -c "grid_cmd"
```

Assuming a functional grid with 30 machines and 30 models that timed out, the command runs just those models that timed out in Formality for 12 hours. The results should be available in just over 12 hours.

Note:

Use the same number of jobs (`-j`) as models that timed out (`wc -l timeout.log`) to get the optimal distribution. If the grid is not large enough for the number of models that timed out, use a fraction of the time based on when you want to see results.

For example, if you want results in 12 hours (next morning) and you have 100 models to validate with a grid of only 25 machines, use `-j 25` with a timeout of 3 hours (1/4 of 12 hours). Since the number of machines (25) is only a quarter of the timeouts, it should complete the analysis of the 100 within 12 hours given 4 models per job.

After this set completes, you might have some that still timed out, but you should likewise have many that did not timeout.

6 Tutorial

This section discusses the best practices for using **zFmCheck** in the following subtopics:

Organize Your Results

When working on a large design, **zFmCheck** is invoked multiple times. Each execution of **zFmCheck** is important to keep a track of results and examine the results in the future. Some **zFmCheck** runs encompass a single `zcui.work` directory and some encompass many `zcui.work` directories. The results must be organized according to the expected usage.

You must adhere to the following rules when starting to analyze a full design:

1. Delete the `zcui.work` directory because the space occupied by `zcui.work` is huge and the space is required for a different compile.
2. Do not store the results directly in `zcui.work`. Instead, use the `-g` option to place the results in an area that has ample space and is not within `zcui.work`.

   Note:
   By default the results are stored within `zcui.work`.
3. Use a fully qualified path name with the `-g` or `-o` option. This is stored within the results and helps you to know exactly where the results are stored. The `-o` option uses the current path as the relative path. For example, `-g 'pwd'/fm` is same as `-o fm`.

4. Use a name for your output directory to indicate what you are doing. Following is an example of iterations that occur:

   ```
   fm_r      #Run zFmCheck with just -r(no -v) on entire design.
   fm_r_timeouts #Run zFmCheck with just -r(no -v) on design models that
    timed out.
   fm_v_348      #Run with just -v on a set of 348. This implies there is
    348 models left.
   ```

   This naming helps to indicate what has occurred.

Using -norm to Keep Simulation Data

When executing `zFmCheck` with `-v`, use `-norm`to preserve all the data needed to produce the `simv` (VCS) executable for each module.

Example:

```
zFmCheck -g `pwd`/fm -v -norm -m dut -d zcui.work
```

The `fm/simulation/sim_dir/dut/*sh` file compiles all the necessary files and runs `simv`. You can execute this directly to re-create the **zFmCheck** run.

The `fm/simulation/sim_dir/dut/simon_dut_tb.*` file is the generated testbench for simulating the reference and implementation models.

Note:

If you want to debug a simulation failure, modify the `*sh` file to comment the testbench generation, modify the existing testbench to (insert an FSDB dump) and then rerun the `*sh` file.

Using DVE to Debug a Simulation Failure

To launch a waveform debugger on the simulation run, there are multiple ways to do it.

To launch the waveform debugger from the **zFmCheck** command line, add `-debug` to the VCS command line and then the `-gui` option to the `simv` execution as follows:

```
zFmCheck -v -opt_simv "-gui" -opt_vcs "-debug" -d zcui.work -m model_name
 -norm
```

Where, `-norm` retains all the simulation data used to produce `simv` for the model named `"model_name"`.

Using Verdi to Debug a Simulation Failure

To debug the waveform of a failed simulation module, use Verdi as follows:

```
zFmCheck -v -verdi_debug -d zcui.work
```

This command generates Verdi command files under simulation/scripts named as: `${model_name}_verdi_cmd.sh`.

To perform debug, launch Verdi using the following steps:

1. Navigate to the `simulation/sim_dir/model/` directory.
2. Execute `${model_name}_verdi_cmd.sh`.

Using -save du to Launch Formality GUI

To use the formality GUI on a given module, make sure you have `-save du` on the **zFmCheck** command line. You can now directly launch Formality using the following:

```
# Replace bundle.tcl and mod with the specific one for your case.
fm_shell -f bundle.tcl -x "set Module mod"
```

Now, from the `fm_shell` prompt, type `start_gui`.

Checking Status of a zFmCheck Run

To check the status of a **zFmCheck** run, use the `-p` option. A summary of the results is displayed and `formality.log` is updated for the selected directory.

```
zFmCheck -p -g fm_r
```

Using -gencase to Create Failing Module Standalone Testcase

To create standalone testcases of failing modules, use `-gencase` as follows:

```
zFmCheck -d zcui.work -o zfm.work -r -gencase
```

To create testcase for specific module, use `-m/-mf` with `gencase` as follows:

```
zFmCheck -d zcui.work -o zfm.work -gencase -m <module>
zFmCheck -d zcui.work -o zfm.work -gencase -mf <module_list_file>
```

It creates testcases under `gen_cases/<module>/` directory.

Error may not be reproduced:

- For modules having XMR
- If VCS options are passed in zCui run that can affect pre-simon stages, so make sure to verify UTF settings

7 Limitations

zFmCheck does not support both Formality and simulation.

Following are the limitations with Simulation:

- UPF flow
- simXl flow
- Combinational loop in RTL

Following are limitations with Formality:

- Presence of resettable memory
- Multiple clock multiple write for memory
- Multiple write for memory
- Presence of Verilog force/release
- If there is condition which is not in the event control list
- If there is blocking and non-blocking assignment on the same variable
- If there is implicit state machine
- Presence of multipliers in RTL (possibility of formality unable to conclude)

Note:

- Simulation does not provide proof, but only captures the issues
- In simulation, pass does not indicate correct model, but does give a high level of confidence
- Use simulation only on the aspects of design which Formality is unable to converge
- Simulation cycle times are not always optimal for quick results

- Simulation results may not be reliable in the following scenarios:
  - when there are multiple drivers either in reference or implementation
  - when there are memories (zMem) with multiple write ports