Verification of Multi-Cycle Path Timing Exceptions in Simulation with Automatically Generated SystemVerilog Assertions

Akshaya Prashanthi Lakshmi Narayanan[1], James Gillespie[2], Abhinav Singla[3], Sudeep Mondal[4], Barsneya Chakrabarti[5]

[1]Infineon Technologies AG, Munich, Germany

[2]Synopsys GmbH, Munich, Germany

[3,4,5]Synopsys India Pvt Ltd, Noida, India

akshaya.narayanan@infineon.com

ABSTRACT

This paper describes the application of a novel approach to validate Multi-Cycle Path (MCP) Timing Exceptions during Register Transfer Level (RTL) simulation using the Synopsys Spyglass TXV tool. The Hybrid Verification Flow generates SystemVerilog Assertions (SVA) for every defined MCP in the design, which are then validated during simulation/regression. This paper will describe the flow, metrics and results based on a real-life application on a complex design. It will also discuss former approaches which were used for MCP validation and their setbacks.

This automated flow, with SVA based on absolute time as opposed to clock cycles, provides a key advantage when working with large, complex, designs in which clock frequency can often change during simulation. It is also therefore easier to maintain the validation environment throughout design changes in a project.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

1.1 Motivation

As process nodes shrink, designs become more complex and target higher performance, it is becoming increasingly necessary to utilize Multi-Cycle Path (MCP) timing exceptions for synthesis and static timing analysis in order to close timing. Design specification and area/timing constraints drive designers to implement large combinational clouds between register stages which make the MCPs unavoidable. An MCP is defined when more than one clock cycle is required to propagate data from one register to another [4]. MCPs are very hard, in many cases impossible, to manually review to verify that they match with the actual intended design behaviour. This introduces a significant design risk as an incorrectly defined MCP can lead to functional failure in silicon. The intention of such a verification is to identify and correct implementation bugs early in the design flow.

This paper describes a new, state-of-the-art, automated approach to verify MCPs at Register Transfer Level (RTL) with SystemVerilog Assertions and Dynamic Simulation testbench.

1.2 Background

Prior to the automated approach explained in this paper, an alternate technique, based on RTL delay elements, was adopted in order to make the MCP exceptions visible and hence verifiable during functional verification. The following section describes the implementation and setbacks.

1.2.1 Implementation

RTL units with variable delay specification were manually instantiated in portions of the design where the signals had to be delayed due to MCP. These units would force the corresponding signals in simulation go to an "X" VHDL value (unknown) for a specified time period . This would make the Dynamic Simulation testbench fail if the software or hardware didn't respect an exception as defined for synthesis. Figure 1 depicts the implementation .

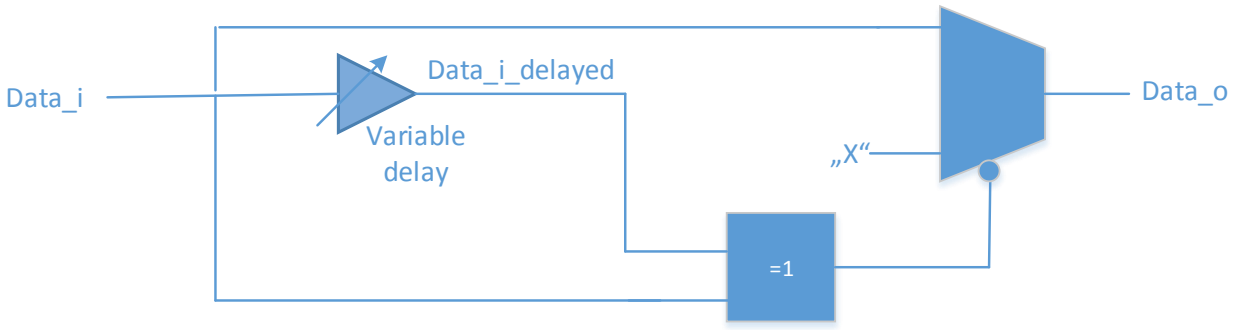

Figure 1. RTL Delay Elements

1.2.2 Setback

a) The RTL delay units were instantiated manually in the design.

b) It is not a simple, one-to-one mapping between the MCP exception definition and the RTL delay unit instantiation in the design. This is because the exceptions are defined for a specific timing path, as considered by the synthesis tool, Synopsys Design Compiler (DC). On the other hand, we cannot insert the delay elements in RTL by blindly following these timing paths as the functionality is interleaved and it might lead to an accumulation of delays and therefore incorrect verification. This explains why the insertion of RTL delay

units by itself needed a separate manual review.

c) Such an approach is not robust to design changes. Everytime the design or exception is changed, the RTL delay units have to be recalculated and carefully reviewed once again.

d) It is not possible to evaluate any coverage information for MCP timing verification in order to define a confidence level. At every stage of implementation and verification, this approach is susceptible to human errors.

2. The Hybrid Verification Flow from Spyglass TXV

2.1 Static and Dynamic Verification

With more and more integration in the System On Chip (SoC) designs, it has become necessary to use dynamic and static verification "hand in hand" in order to improve confidence in the design quality and have better coverage [1]. Dynamic verification refers to simulation-based verification which include techniques like directed tests and constrained-random verification. It is a testbench based approach and is not exhaustive [2]. On the other hand, static verification (also known as formal verification) uses mathematical methods to prove properties of the design and is exhaustive. Table 1 [3] illustrates the advantages and disadvantages of these two approaches.

Verification	**Dynamic**	**Static**
Depth	Non-Exhaustive	Exhaustive
Preparation	Complex test-bench	No dedicated test-bench
Complexity	Always conclusive	Inconclusiveness increases with increased complexity

Table 1: Comparison of Dynamic Verification and Static Verification

The hybrid flow from Spyglass TXV performs static verification on the design and later can be used for dynamic verification to close coverage holes.

2.2 Usage of the Flow for MCP Verification

The hybrid flow is depicted in Figure 2.

In the context of our project, a Register Transfer Level (RTL) description of the design and Synopsys Design Constraints (SDC) were given as inputs to the tool. The SDC file included

- MCP timing exceptions which needed to be verified dynamically
- Description for design clocks

With the provided inputs, Spyglass TXV will identify all valid timing paths in the design between the start, through and end points described in the MCP exceptions. The enable logic for the destination flip-flop is then formulated for every identified path. Enable detection is explained in detail in section 2.3. At this stage, the tool can be forced to skip formal verification and generate only SystemVerilog assertions for the given exceptions as shown in Figure 2. These generated assertions could be plugged into a Dynamic simulation regression environment so that it is dynamically verified and coverage information can be obtained. The tool generates unique assertions files for a wide range of simulators with a "bind" file so that the assertions can be easily integrated into the regression.

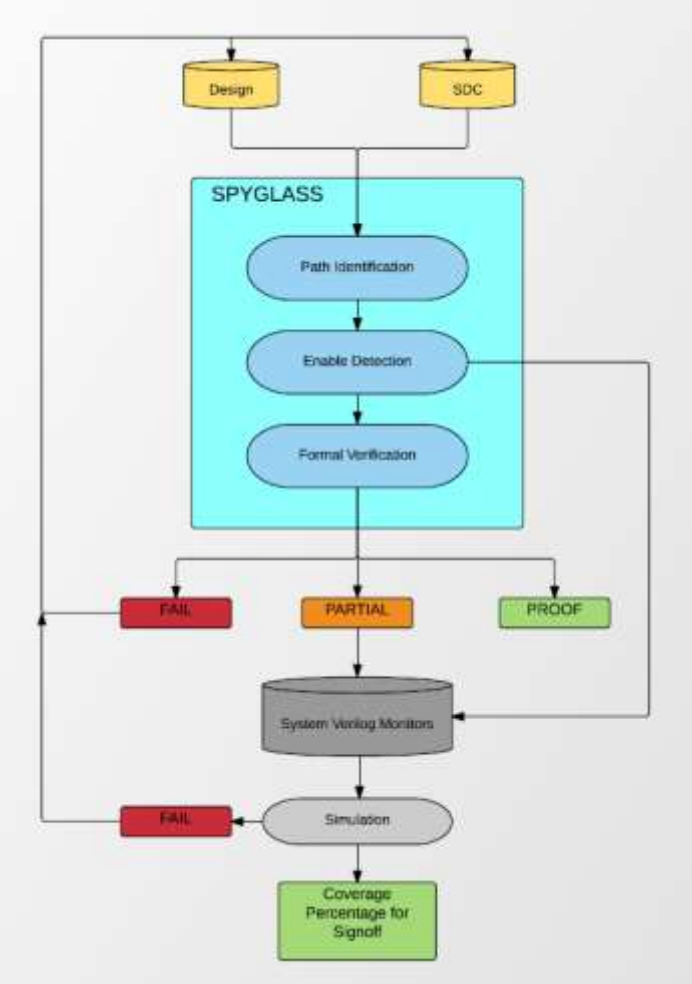

Figure 2. Spyglass TXV Hybrid Flow

2.3 Enable Detection

Spyglass TXV detects the enable conditions corresponding to each data path by doing a structural analysis of the design. It involves a smart traversal of the design which identifies the enabling conditions for each data path and generates the SystemVerilog Assertions.

Spyglass TXV provides the flexibility to the user to choose the control logic structure type to detect:

- MUX based enable
- AND gate based enable
- NAND gate based enable
- OR gate based enable
- NOR gate based enable

Spyglass synthesizes MUX structures for most of the control logic in RTL and therefore detects the MUX based enables in the default setting. It can also detect other enable structures from above mentioned types if the user provides the corresponding setting in the tool.

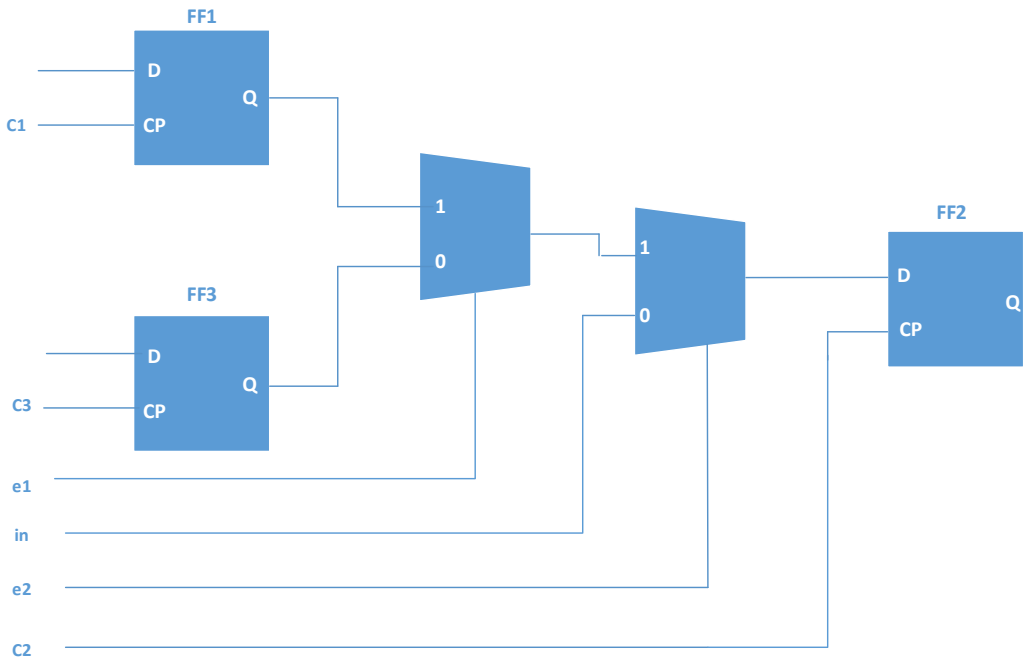

Figure 3. Enable Detection

In Figure 3, for an MCP path defined from C1 to C2, Spyglass TXV detects the enable condition for data path from FF1 to FF2 as e1 & e2 .

2.4 Cycle Vs Timing Based SVA

Spyglass TXV can be used to generate two kinds of assertions depeding on the design : cycle based SVA or timing based SVA. Details of cycle based and timing based SVA are given below, followed by a waveform example from our design simulation to illustrate on the necessity for timing based SVA for our project.

The cycle based SystemVerilog assertion verifies multicycle path property with the cycles of MCP clock as:

```
property mcp_user;
@(posedge mcpclk) disable iff(~Assertion_mod.spyglass_assert)
(from != $past(from)) |=> (!((to != $past(to)) && $past(den)))
[*PATH_MULT]; // for a (PATH_MULT + 1) cycle mcp
Endproperty
```

Here, mcpclk refers to the actual design clock that goes to the destination flip-flop.

The cycle based SVA, does not take into account the absolute time available for the path. If the sdc file has following definitions:

```
create_clock -name C1 -period 20 clk1
set_multicycle_path 3 -to C1 -end
```

Then the absolute time available for data to propagate is 3*20 = 60 time units.

However, if in the simulation, the clock reaching the destination flop has period 40, and the actual path multiplier for the corresponding path is 2 instead of 3, then assertion should not fail because the actual data propagation would take place after 2*40 = 80 time units .

Cycle based assertion will result in a false assertion failure as the actual path multiplier for this simulation testcase is 2 and not 3 as defined by the MCP, but this is still safe from a design timing point of view, based on slower destination clock..

Following is the same assertion with timing based flow:

```
reg TXV_C1;
initial begin
    TXV_C1 <= 1'b0;
end
always #10.000000 TXV_C1 = ~TXV_C1;

mcpclk = TXV_C1;

property mcp_user;
@(posedge mcpclk) disable iff(~Assertion_mod.spyglass_assert)
(from != $past(from)) |=> (!((to != $past(to)) && $past(den)))
[*PATH_MULT]; // for a (PATH_MULT + 1) cycle mcp
Endproperty
```

In the timing based SVA, the clock specified in the sdc will be generated at run time of the Spyglass-TXV tool (TXV_C1 in the above example), and will be used for cycle count. Therefore, even if the clock at destination flop in simulation has different period w.r.t. clock in sdc, the absolute time available for data propagation is always fixed which is SDC_CLOCK_PERIOD*PATH_MULTIPLIER.

In our project, we had to use timing based SVA generation due to the following reason :

We have a clock that has a different time period for varying scenarios. Figure 4 illustartes this issue. Depending on the enable conditions (value of “clk_freq_changer”) from the testcase, the “designclk” in simulation can be slower than its definition for synthesis. The number of multicycle delays which have been defined for synthesis will not be valid anymore. This leads to a lot of false failures for our design. Therefore the SVA is checked based on “mcpclk” , which has a constant period throughout the simulation and has a periodicity as defined in synthesis for the “designclk”.

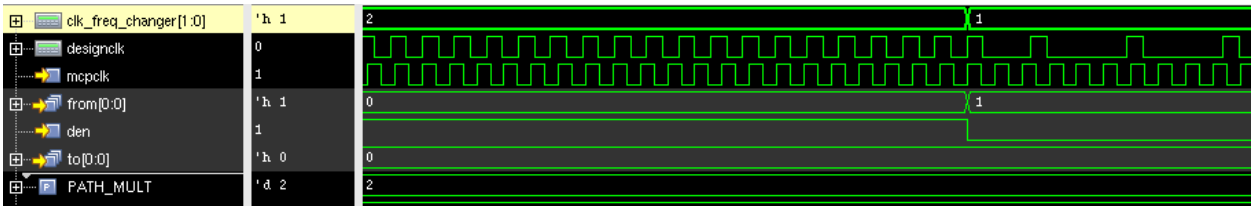

Figure 4. Timing based SVA

2.5 Example with a Simple Design

Let us consider a simple example design as shown below:

```
module test (in1,in2,clk1,clk2,out);
input in1,in2,clk1,clk2;
output out;
reg ff1,ff2,ff3;
reg en;
always @(posedge clk2)
```

```
  en <= ~en;
always @(posedge clk1)
  if (en)
    ff1 <= in1;
always @(posedge clk2)
  ff3 <= in2;
wire w1;
assign w1 = en ? ff1 : ff3;
always @(posedge clk2)
  ff2 <= w1;
assign out = ff2;
endmodule
```

For this design , we have defined a MCP exception of 3 between ff1_reg and ff2_reg in the SDC file as follows:

```
set_multicycle_path 3 -from clk1 -through ff1_reg/Q -through ff2_reg/D
-to clk2
```

where clk2 has been defined as :

```
create_clock -period 10 -name clk2 [get_ports clk2]
```

According to this exception, we do not expect any valid data to be available at ff2_reg before 3*10 time units .

```
MCP_Check_mod #("Spyglass - MCP failure (test.sdc:3)", 2,
1)TXV_MCP_s0_m1_p1(
.mcpclk(TXV_clk2),
.from(test.ff1),
.to(test.ff2),
.den(test.en ));
```

Here MCP_Check_mod module defines the property as stated in section 2.4.

Once the assertion file has been generated, it can be plugged into the dynamic simulation testbench and verified. The quality and number of the testcases determines the simulation coverage for the MCP assertions. Figure 5 shows an example waveform (with Incisive Enterprise Simulator) that could be used to debug a failing assertion.

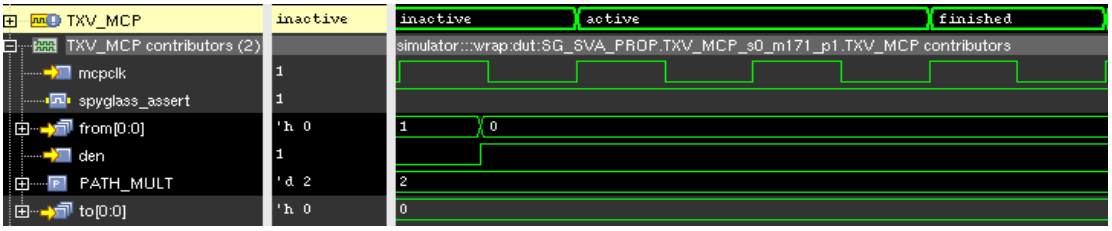

Figure 5. Waveform of a simulated SVA

3. Used Commands for this Paper

A set of parameters had to be specially added in the Spyglass project configuration file used by the tool to enable the generation of SVA in a certain manner for our project. These additional parameters are stated below:

```
set_parameter txv_mcp_dp_enable_type 'mux_and'
set_parameter txv_run_audit yes
set_goal_option addrules { Txv_Gen_Assert }
set_parameter txv_mcp_common_enable_only no
set_parameter txv_mcp_time_based_sva_gen yes
```

These commands instruct the tool not to make any structural analysis or formal verification. We forced the tool to generate only SVA for the given exceptions. MUX and AND based enable structures were used to identify enable logic and time-based SVA generation was used.

The flow also generates bind statements for instantiating the SVA module into the design hierarchy. This offered a convenient plug-in solution to integrate the SVA into our VHDL simulation. These bind statements helped to keep the verification logic separated from the design logic.

4. Results from Real Life Design Application

Inputs to the tool were a Tcl file defining all the MCP exceptions which was originally written for synthesis with Synopsys DC and the design description at RTL. With the current design, our challenge was also to ensure that the software running on the hardware for which MCP exceptions have been defined, was properly written so that it does not pick up and propagate data earlier than desired. Finding such mistakes in the software, early in the design flow whilst still at the RTL simulation stage, is very beneficial as necessary changes can be more easily fixed.

In our design, most of the MCP enables that control the data flow are driven from software. In order to have used a formal verification approach to validate the MCPs, we would have needed to either provide complex constraints or a full behavioral model of the software to the tool. Both of these require significant effort to develop and maintain. Therefore we chose to target dynamic verification using SpyGlass TXV generated timing based SVA, which saves significant time in the development phase and existing dynamic simulation testcases could be utilized.

Table 2 lists the number of generated SVA for the defined MCP exceptions.

MCP Exceptions	**Generated SVA**
Clock to Clock : 3	725
Register to Register : 152	441

Table 2: Generated SVA for defined MCP Exceptions

Assertion coverage metrics along with the number of failing assertions were obtained using Cadence Vmanager regression tool. It is shown in figure 6. As we can observe, 66.1 % of the assertions passed and the rest failed. We were able to cover 66.19% of the assertions with the testcases included for the regression run. Further below, we get a list of the SVAs , along with their individual status.

Current Node: /SG_SVA_PROP

Exclusion Rule Type	UNR	Name	Assertion Average Grade	Assertion Covered	Assertion Status Grade	Valid Metrics
None	none	SG_SVA_PROP	66.19%	736 / 1112 (66.19%)	66.1%	0

Sub-Nodes:

Exclusion Rule Type	UNR	Name	Assertion Average Grade	Assertion Covered	Assertion Status Grade	Valid Metrics
None	none	TXV_MCP_s0_m1_p1	100%	1 / 1 (100%)	100%	256
None	none	TXV_MCP_s0_m1_p2	100%	1 / 1 (100%)	100%	256
None	none	TXV_MCP_s0_m1_p3	100%	1 / 1 (100%)	100%	256
None	none	TXV_MCP_s0_m1_p4	100%	1 / 1 (100%)	100%	256
None	none	TXV_MCP_s0_m1_p5	100%	1 / 1 (100%)	100%	256
None	none	TXV_MCP_s0_m1_p6	100%	1 / 1 (100%)	100%	256

Figure 6. Assertion Coverage and Status

There was virtually no simulation time overhead when the assertion file was used with the normal regression. This was a big benefit for our design as it normally takes ~24 hours for our regressions to complete and we cannot afford significantly more time to verify MCP exceptions additionally.

Figure 7 depicts a failing MCP exception (due to incorrect software) which was identified by Spyglass TXV but not by any other means of verification. This lead to design changes.

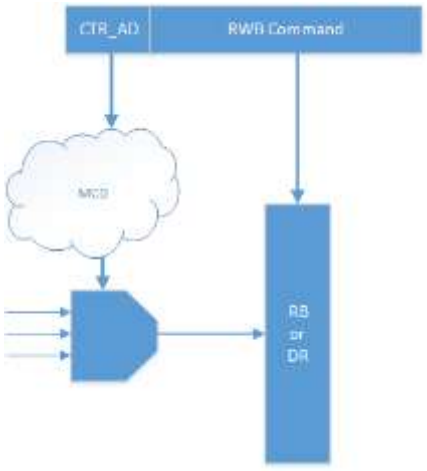

Figure 7. Failing MCP

In our setup , the software was fetching data from “RB or DR” register in the very next clock cycle after “CTR_AD” was fed. This could not be detected with normal functional verification. A waveform of the fail is shown in figure 8.

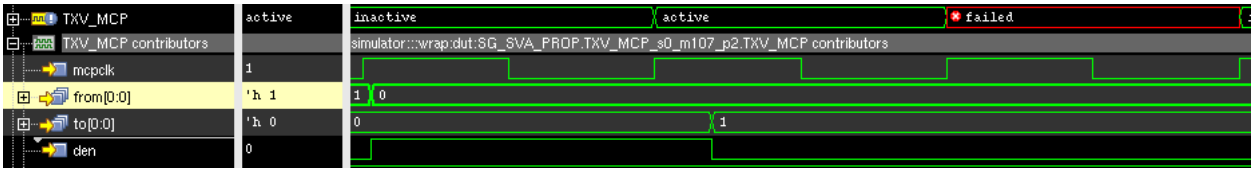

Figure 8. Waveform of the failing MCP

5. Conclusions and Future Work

This paper described an automated approach to verify MCP exceptions which leads to better design quality and faster closure of design verification. The tool helped us to obtain better verification coverage information for the defined MCP exceptions as opposed to tedious manual reviews that was done prior. This increased our confidence in the design quality.

In future, we intend to extend the application of the tool to more complex designs with deeper levels of hierarchy. It would also be useful to think of generating a schematic for every enable logic that is inferred by the tool. This would help in easier debugging of failing MCPs. A more verbose list of exceptions which could not be translated into SVA, along with more detailed explanation from the tool would improve the usability.

6. References

[1] Ping Yeung,"Advanced static verification for SoC designs," SoC Design Conference (ISOCC), 2009 International

[2] Walter Soto Encinas,"Infrastructure for formal and dynamic verification of peripheral programming model," Test Symposium (LATS), 2016 17th Latin-American

[3] Sudeep Mondal, "Hybrid Verification Flow," Atrenta Technical Conference 2013

[4] Hongbin Zheng,"High-level synthesis with behavioral level multi-cycle path analysis," Field Programmable Logic and Applications (FPL), 2013 23rd International Conference