2024 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Automation of Glitch Checker Implementation on Various Design Interfaces/ Boundaries

Nikhila Pranavi Adari
NXP Semiconductors
Bengaluru, Karnataka
{nikhila.pranaviadari@nxp.com}

***Abstract*- Glitches are one of the most common and critical errors in any RTL designs. It is a real challenge to detect spikes on signals in RTL simulations. Glitches are often identified in GLS simulations, which comes at the end of project cycle and thereby difficult to fix without significant design changes. This paper discusses on RTL glitch detection methodology to expose any issues earlier to gate-level simulation stage, an approach where the design interface signals are tapped and automated to implement glitch checker assertions for each corresponding interface signal and their respective design parameters. Furthermore, there is only a CSV file taken as input and the automation script transforms them to property files or SVA that can be seamlessly integrated into our testbench.**

Keywords - SoC; Glitch checker; Automation; RTL; GLS Simulations; target module; CSV; clock frequency

I. INTRODUCTION

In recent years, with advancements in technology across various fields like telecommunications, computing, consumer electronics, automotive, health care etc., semiconductors play a crucial role. There is an immense demand for more powerful and efficient SoC's in this current technology driven world. Multiple functionalities are being integrated into the SoC's, to meet the customer requirements, thus increasing design complexities day by day to accommodate higher performance, greater functionality, and miniaturization [1]. It is quite a challenge for the verification team to sign off such complex designs.

Glitches are the most common bugs which often escapes from the testbench simulations and caught as silicon bugs. By detecting and correcting glitches early in the design phase, glitch checkers significantly enhances the reliability of digital systems, helps in avoiding costly redesigns and rework [2]. There is a high chance for glitch transmission when a signal is connected from analog to digital boundary. There might be a transient disturbances or irregularities in the design, which can impact the smooth and continuous analog signals. This small spikes on digital signals when connected to analog domain can have a huge impact. Glitches could lead to data corruption or loss if they affect integrity of transmitted signals. Also, they could lead to false triggering of unwanted logic, which impacts the system functionality and increases power consumption.

In GLS simulations and design synthesis, glitch occurrence is more significant as the netlist, gate delays from combinational logic, race conditions and timing constraints comes into picture. To capture these unwanted signal transients, it is more common in verification testbench to implement checkers on the signals prone to glitches. In this paper, we discuss on how we automated glitch checker on large number of signals from different module/ boundaries with minimal efforts.

II. IMPLEMENTATION

A. *Proposed Methodology:*

Glitch checker is implemented at the top level of design hierarchy, so that it is easy to tap the signals from modules of DUT at different hierarchical levels anywhere in the design, as shown in Figure 1. This approach also gives us the feasibility to extend the checker to any number of modules. Instead of tapping output signals of source block, checker was implemented on the input signals of the destination block. It gives us the confidence that the later releases of RTL do not impact glitch checker results, which was implemented on the interface of target modules. In this way the checker stays immune to any changes on the signal path (E.g: Some wrapper updates, synchronizer, or glue logic additions etc.).

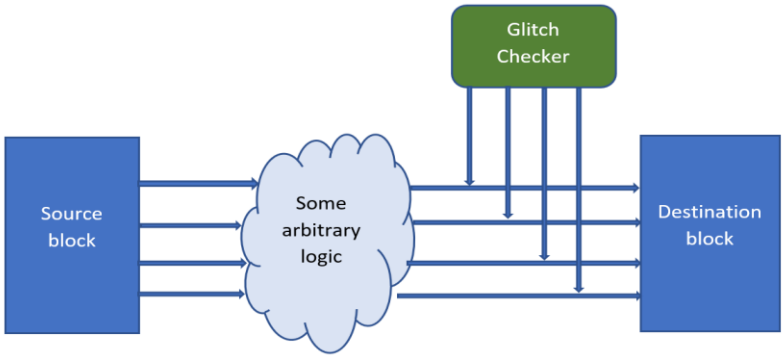

Figure 1. Implementation of checker on the interface/ boundary of target module

Integrating glitch checkers into verification testbench requires careful planning to ensure that they complement all the signal operating conditions are taken care of, on which the checker is implemented. For each signal there might be a different reset, enable and disable conditions. Also, when they function on different clocks their glitch width varies. Some signals like clocks might take few cycles before reaching a stable condition, in such scenarios we need to instantiate the checker after delayed cycles. Considering all these parameters and implementing an assertion manually for each individual interface signal of a module/ boundary is quite a hideous task.

B. *Glitch checker module and csv:*

Implementation of glitch checker, with automation brings down lots of effort and improves efficiency by reducing human errors. The primary input for the script to implement this flow is a csv file with the information of all the interface signals of the target module in different rows, with their respective reset, enable, disable, delay and glitch width information in the columns (i.e., signal_name, reset_signal, maximum and minimum (ns) to calculate glitch width, enable_condition, disable_condition, checker_delay) as shown in Table 1. Automation script iterates through each row entry in csv by probing the target signal with the glitch checker assertion and their corresponding configuration.

Glitch checker module pseudo code:

```
module glitch_checker #(parameter real MIN_NS=0.0, parameter real MAX_NS=2.0, parameter int DELAY=0) (input logic sig, input logic reset, input logic enable, input logic disable )”

   bit skip;
   bit [$clog2(DELAY):0]count;

   property glitch;
     realtime tm_1;
     disable iff(reset !== 1 || enable !== 1 || disable_ == 1 || skip)
      (1,tm_1=$realtime) |=> ($realtime - tm_1 >= MAX_NS || $realtime - tm_1 <=MIN_NS);
   endproperty

    if(DELAY > 0) begin
      always @(posedge sig) begin
      if (count < DELAY)
      begin
          count = count + 1'd1;
          skip = 1;
      end
      else
            skip =0;
      end
  end
endmodule
```

TABLE I
SAMPLE CSV FORMAT

Signal_name	Reset	Max_ns	Min_ns	Delay	Enable	Disable
TB.top.moduleA.signal1	CPU_RST	8	0	0	TB.top.moduleA.Reset && !TB.top.enable_a	TB.top.Isolate_moduleA

The automation script generates two files, one file with a glitch checker wrapper module encapsulating individual checkers implemented on all the interface signals from csv and the second file which instantiates the glitch checker wrapper and with all the hierarchies probed into it as shown in Figure 2. These two generated files can be included in the testbench or any other checker file.

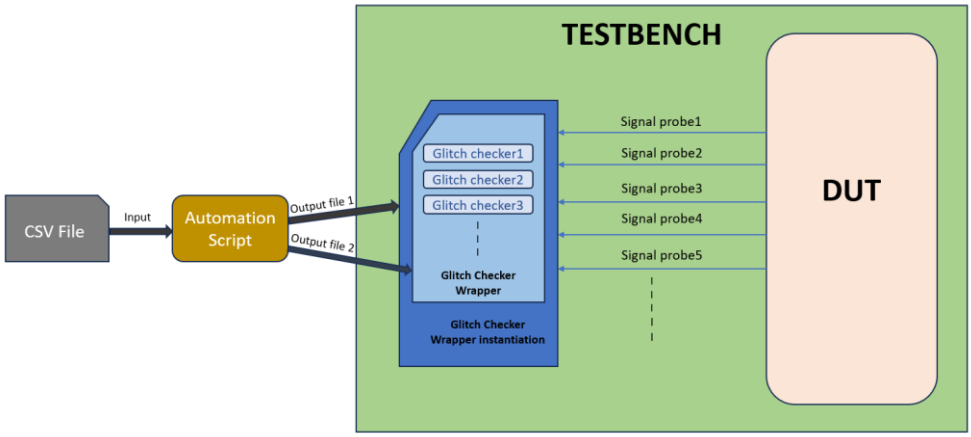

Figure 2. Glitch checker file generation from automation script

C. Clock timing mismatch:

Clock gating is widely used in design to save power by disabling clocks parts of circuit that are not actively processing. Improper clock gating can lead to timing mismatches, and this will trigger the logic driven by clock unnecessarily. Sometimes there is an inconsistency in clock frequency due to improper power up or boot sequence. The glitch checker is implemented in such a way that, it can detect clock glitches or even if the clock is running other than its specified frequency. There is a flexibility to implement checker for clocks with different duty cycle ratios as shown below.

Checker properties for clocks with different duty cycle ratios:

```
 property clk_glitch1;
      realtime tm_1;
      disable iff(reset !== 1 || enable !== 1 || disable_ == 1 || skip || MIN_NS == 0 || high == 0)
      (1,tm_1=$realtime) |=> ($realtime - tm_1 >= MAX_NS);
endproperty


property clk_glitch2;
      realtime tm_1;
      disable iff(reset !== 1 || enable !== 1 || disable_ == 1 || skip || MIN_NS == 0 || high == 1)
      (1,tm_1=$realtime) |=> ($realtime - tm_1 >= MIN_NS);
Endproperty

if(MIN_NS > 0) begin
      always @(sig) begin
          if(high == 1)
              high = 0;
          else
              high =1;
      end
 end
```

D. Results:

When a glitch occurs on the interface signal or when clock frequency is not within the specified range, then the glitch assertion triggers with the failure signature of *"*E,ASRTST (<assertion_file_path>.sv,<line_number>): (time) Assertion <assertion_hierarchy> has failed*" in RTL/ GLS simulations. This failure can be further traced back in the simulation waveform and root cause the failure.

E. Challenges faced:

As designs become more intricate, the analysis performed by glitch checkers can become more complex and time consuming. As the assertion triggers at each signal toggle, it sure leads to longer simulation times. In an SOC for a given module there will be many interface signals and to know all the signal operating conditions is quite challenging and requires tight collaboration between architect, design, verification.

III. CONCLUSION

We have used this flow to create glitch checker assertions on each interface signals of analog boundary, register wrapper and I/O wrapper. This method of automating and implementing glitch checker on each targeted interface signals ensures that the checker operates in sync with the verification environment and the DUT, also enhances the verification results by reducing manual errors and saves lot of time. The automation script demonstrated viability to scale the checker and extend to any module in the design. Since, all the signal parameters and hierarchy information are maintained in in the form of csv, it is easy to maintain with the further design updates in project cycle. This method of integration of glitch checker into verification environment aided in detecting unwanted signal transitions in RTL and GLS simulations more effectively.

REFERENCES

[1] Hopkins, Andrew BT, and Klaus D. McDonald-Maier. "Debug support for complex systems on-chip: A review." *IEE Proceedings-Computers and Digital Techniques* 153, no. 4 (2006): 197-207.

[2] M. Kasim, V. Gupta and M. Jebin, "Methodology for Detecting Glitch on Clock, Reset and CDC path," *2020 5th International Conference on Communication and Electronics Systems (ICCES)*, Coimbatore, India, 2020, pp. 300-304, doi: 10.1109/ICCES48766.2020.9138064.