# Automated Connectivity Test Creation for System-in-Package Analog Mixed-Signal Verification

Samantha Pandez
Analog Devices, Inc.
Gen. Trias, Cavite
Sam.Pandez@analog.com

Christopher Geen
Analog Devices, Inc.
Wilmington, MA
Christopher.Geen@analog.com

***Abstract*- Mixed-signal verification consumes more time than ever, driven by increasing product complexity, power constraints, and higher performance. An important aspect of mixed-signal verification is validating connectivity from digital pins to analog cells. This work proposes a novel mixed-signal verification method that simplifies connectivity checking for complicated ASIC developments. This method uses python to automatically generate SystemVerilog connectivity tests based on an intuitive, human-readable spreadsheet description. Tedious, error-prone manual steps are reduced, and model synchronization is enhanced by enabling comparison against a golden standard. This method was successfully implemented on the latest System-in-Package (SiP) Pin Driver ATE Project at Analog Devices, Inc. where checkers were generated for >2000 interface pins.**

## I. Introduction

Verification is one of today's most important product development challenges. The latest Wilson Research Group Functional Verification Study [1], shows the percentage of ASIC development spent on verification averages around 50-60%, which is higher than prior year surveys. The same study found verification engineers spend most of their time debugging, creating tests, and running simulations. Figure 1 from the same study shows these tasks, combined, consume 68% of total verification time. Clearly productivity can be improved with verification methodologies promising to reduce errors during code development.

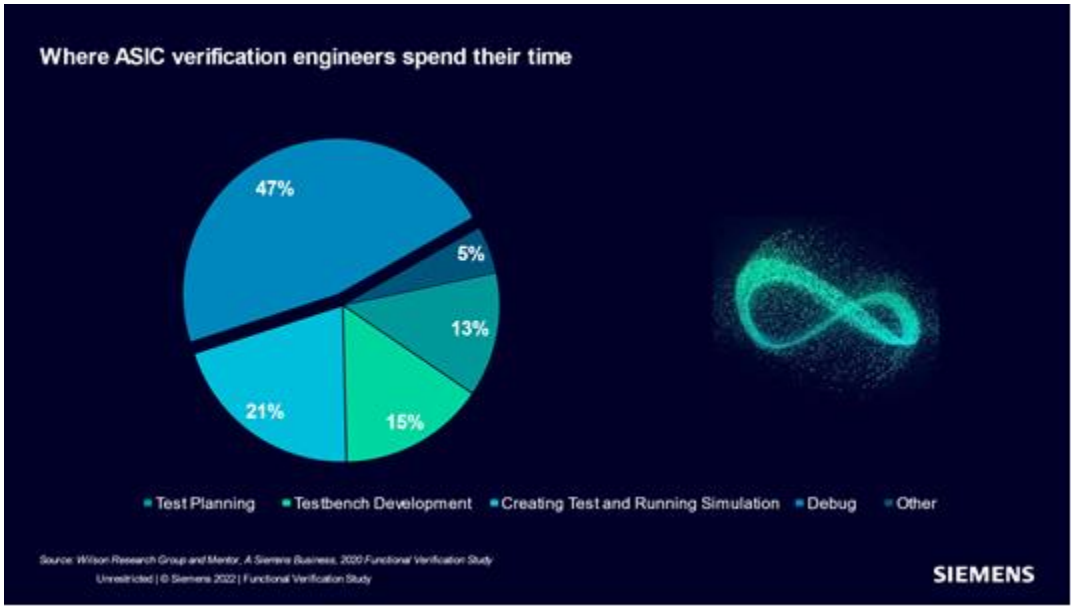


Figure 1. Where ASIC/IC Verification Engineers Spend their time (Siemens EDA and Wilsons Research Group, 2022)

Connectivity checking verifies the intended connectivity from digital pins to their analog circuit destinations, ensuring correct voltage levels and logic states. These tests catch errors such as switched pin positions, incorrect node names, unmatched bus bit positions, and improper reset states. This task is challenging, especially with large complex designs containing multiple levels of hierarchy, gating conditions, and register logic. Manual test creation is error

prone, as large designs contain thousands of connections with varying stimulus conditions, all of which are important to ensure that the design will work as intended.

Our proposal reduces verification effort avoiding errors inherent with manually generating connectivity code. It utilizes a python script to convert a human-readable spreadsheet into SystemVerilog connectivity test code which includes voltage checkers. This approach was used successfully in the top-level verification of the latest Analog Devices, Inc. Pin Driver ATE project. The routine generated connectivity tests for over 40 priority 16-bit registers in 8 channels, generating 591 voltage checkers, saving days of code debug effort.

In Section II we describe the traditional, manual connectivity checking approach. In Section III, we introduce the new automated connectivity checker methodology and demonstrate how a spreadsheet is used to generate the connectivity checking code. Section IV discusses the advantages, limitations, and best practices when implementing our methodology. Section V presents summary of our results and proposes future enhancements.

## II. Traditional Manual Connectivity Checking

Modern SiP designs contain complicated subsystems with complex IP blocks and numerous configurable modules, making manual connectivity test development challenging. Writing checkers for every digital logic signal and manually confirming each signal arrives at the proper destination with the intended voltage level is daunting since, checkers must be hand crafted for every relevant connection. This is possible with small designs but becomes untenable with complex products. Complexity invites code errors which impact schedule and productivity. Our proposed automated approach reduces code errors resulting in increased productivity and efficiency.

## III. Automated Connectivity Test Creation

Connectivity tests, including voltage checkers, can be automatically generated using a standardized format consisting of a human readable spreadsheet which is operated on by a Python script producing System Verilog code.

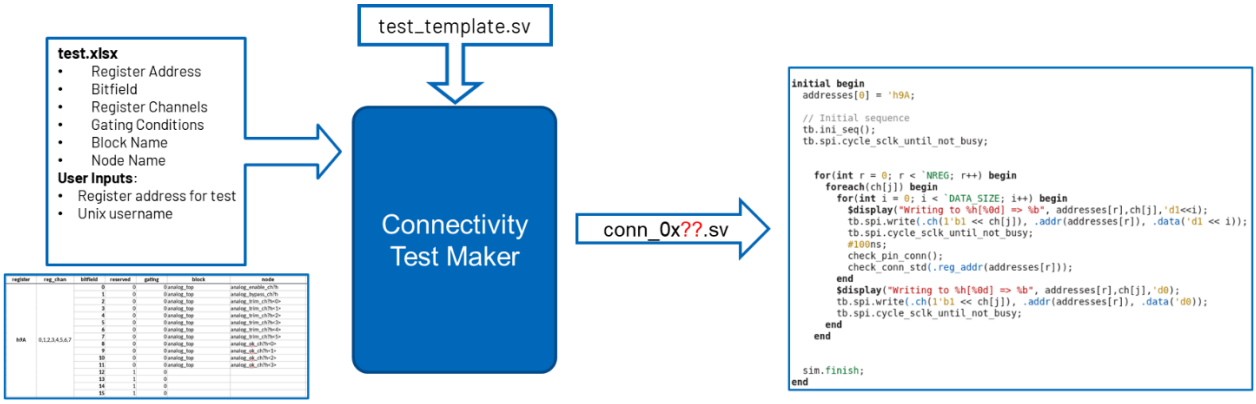


Figure 2. Automated Connectivity Test Generation Flow

Figure 2 shows the process of generating the connectivity tests. Each part is discussed as follows:

A. *Digital to Analog Connectivity Spreadsheet* - The input spreadsheet is a document that contains detailed information about the signals in each bitfield of the register. This can include: the applicable channels, if the bitfield is reserved, if R/W operations are permitted, and if the bitfield is gated. We propose using a project connectivity spreadsheet document to record the proper, intended connectivity between the various circuit elements. Updating this document is the responsibility of the project system architect, who must ensure connectivity changes are captured throughout the project development. An example connectivity document is as follows:

- register – Digital register location
- reg_chan – Channels this register exists on
- bitfield – The bit within the register
- reserved – If this bit can be written

- gating – If there is gating logic
- block – The connecting analog block instance
- node – The node name on the connecting analog block

| register | reg_chan | bitfield | reserved | gating | block | node |
|---|---|---|---|---|---|---|
| h9A | 0,1,2,3,4,5,6,7 | 0 | 0 | 0 | analog_top | analog_enable_ch?h |
| | | 1 | 0 | 0 | analog_top | analog_bypass_ch?h |
| | | 2 | 0 | 0 | analog_top | analog_trim_ch?h<0> |
| | | 3 | 0 | 0 | analog_top | analog_trim_ch?h<1> |
| | | 4 | 0 | 0 | analog_top | analog_trim_ch?h<2> |
| | | 5 | 0 | 0 | analog_top | analog_trim_ch?h<3> |
| | | 6 | 0 | 0 | analog_top | analog_trim_ch?h<4> |
| | | 7 | 0 | 0 | analog_top | analog_trim_ch?h<5> |
| | | 8 | 0 | 0 | analog_top | analog_ok_ch?h<0> |
| | | 9 | 0 | 0 | analog_top | analog_ok_ch?h<1> |
| | | 10 | 0 | 0 | analog_top | analog_ok_ch?h<2> |
| | | 11 | 0 | 0 | analog_top | analog_ok_ch?h<3> |
| | | 12 | 1 | 0 | | |
| | | 13 | 1 | 0 | | |
| | | 14 | 1 | 0 | | |
| | | 15 | 1 | 0 | | |

Figure 3. Sample spreadsheet data

B. *Connectivity Maker Script* - The spreadsheet data is parsed by a python script, which treats the spreadsheet as a data frame and converts it into a multi-level dictionary data structure. The python script associates the block and node information contained in the spreadsheet to a valid hierarchical path to the analog node. For example, Figure 3 shows that the bits in register 0x9A connect to the cmp_top block in each channel. The script then utilizes the gating logic and register bitfields to construct a logic statement ensuring that each node is toggled. The python routine then defines a checker to validate the circuit connectivity.

C. *test.sv output* - The output is a test that operates on each register bitfield (of all channels), checking the voltage levels of the corresponding analog nodes. The test sets each pin to a known state and checks that the voltage level of the target analog node is at the correct level.

Figure 4 shows an output code example which cycles through the bits in a register without gating. The custom *check_reg* function is generated in each test and verifies that the readback of the register matches the expected value based on a golden reference. The *check_conn* function checks the voltage levels of the analog node corresponding to each register bitfield. Figure 5 shows the automated checkers generated by the script that are part of the check_conn function.

```
initial begin
  addresses[0] = 'h9A;

  // Initial sequence
  tb.ini_seq();
  tb.spi.cycle_sclk_until_not_busy;


    for(int r = 0; r < `NREG; r++) begin
      foreach(ch[j]) begin
        for(int i = 0; i < `DATA_SIZE; i++) begin
          $display("Writing to %h[%0d] => %b", addresses[r],ch[j],'d1<<i);
          tb.spi.write(.ch(1'b1 << ch[j]), .addr(addresses[r]), .data('d1 << i));
          tb.spi.cycle_sclk_until_not_busy;
          #100ns;
          check_pin_conn();
          check_conn_std(.reg_addr(addresses[r]));
        end
        $display("Writing to %h[%0d] => %b", addresses[r],ch[j],'d0);
        tb.spi.write(.ch(1'b1 << ch[j]), .addr(addresses[r]), .data('d0));
        tb.spi.cycle_sclk_until_not_busy;
      end
    end
```

Figure 4. Snippet from the output test code of the script

```
if (ch[j] == 0 || ch[j] == 2 || ch[j] == 4 || ch[j] == 6) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_trim_chxh<1>"), .exp(reg_tmp[3]), .vtol(vtol));
end else if  (ch[j] == 1 || ch[j] == 3 || ch[j] == 5 || ch[j] == 7) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_trim_chyh<1>"), .exp(reg_tmp[3]), .vtol(vtol));
end

if (ch[j] == 0 || ch[j] == 2 || ch[j] == 4 || ch[j] == 6) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_trim_chxh<0>"), .exp(reg_tmp[2]), .vtol(vtol));
end else if  (ch[j] == 1 || ch[j] == 3 || ch[j] == 5 || ch[j] == 7) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_trim_chyh<0>"), .exp(reg_tmp[2]), .vtol(vtol));
end

if (ch[j] == 0 || ch[j] == 2 || ch[j] == 4 || ch[j] == 6) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_bypass_chxh"), .exp(reg_tmp[1]), .vtol(vtol));
end else if  (ch[j] == 1 || ch[j] == 3 || ch[j] == 5 || ch[j] == 7) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_bypass_chyh"), .exp(reg_tmp[1]), .vtol(vtol));
end

if (ch[j] == 0 || ch[j] == 2 || ch[j] == 4 || ch[j] == 6) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_enable_chxh"), .exp(reg_tmp[0]), .vtol(vtol));
end else if  (ch[j] == 1 || ch[j] == 3 || ch[j] == 5 || ch[j] == 7) begin
       check_vlogic(.ch(ch_str), .block("analog_top"), .node("analog_enable_chyh"), .exp(reg_tmp[0]), .vtol(vtol));
end
```

Figure 5. Sample pin checkers generated by the script for the check_conn function

This methodology ensures digital pins are correctly wired to their respective analog blocks guaranteeing no pins are mis-wired (e.g. ensuring bits 7 and 6 are not reversed and that channel 0 is not connected to channel2). It also verifies the intended voltage signal levels arrive at the analog block (e.g. ensuring the driver enable pin is not enabled when enabling the load). This methodology also addresses the complexity of circuit hierarchy. Instead of manually coding a complex hierarchical path (e.g. itop.ianalog.i0.icore.i0.ichannel3.idesigneroversight.icmp_top), the script automatically constructs the path. These paths can often be over 10 levels deep and vary depending on where a block is located in the hierarchy (i.e. on different die in a multi-die laminate designs as illustrated in Figure 6).

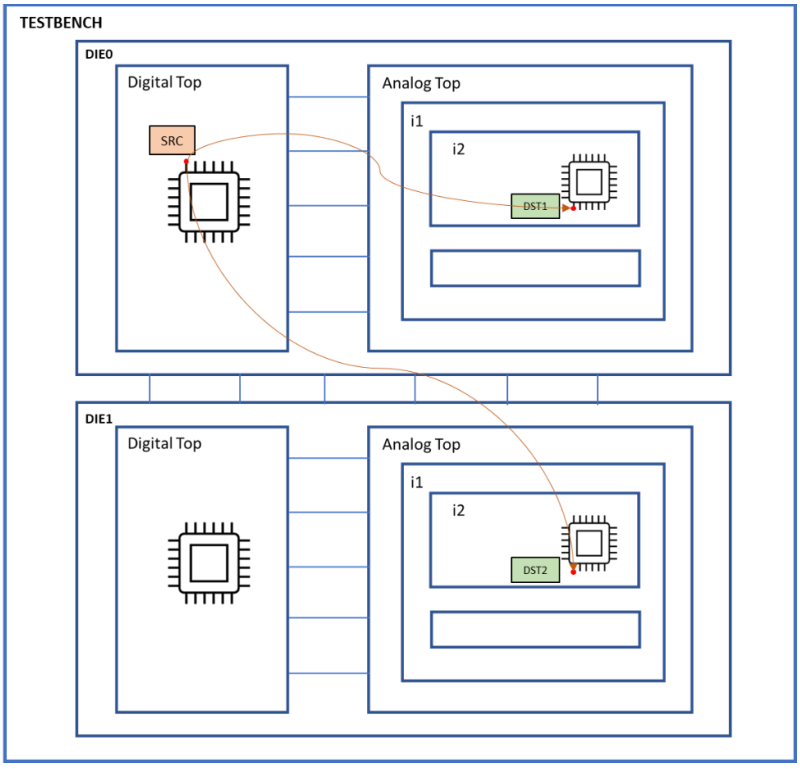


Figure 6. Digital source to its analog block node destination going through several layers of hierarchy for separate die

## IV. Advantages, Limitations and Best Practices

Our new methodology's main advantage is it quickly produces a standardized test structure which can accurately parse a complex analog hierarchy. Hand crafting hundreds of connections and checkers for read/write transactions takes an extremely long time and is error prone. Our proposal reduces this effort from weeks to minutes. This methodology is flexible accommodating hierarchical changes since re-running the script will generate updated checkers. Since many of our projects utilize same register definitions or analog block names, we can reuse a majority of the spreadsheet between projects. Also, since the hierarchy information is extracted from the netlist, the methodology can be utilized in either a top-down or bottom-up approach. It can also cover blocks which uses SystemVerilog models if it is included on the netlist. The abstraction level changes based on the test requirements. This work has been implemented in a mixed-signal design project, and it was simulated in a co-simulation environment. One limitation of the automated connectivity checker is that connections to analog blocks which reside behind a state machine require a verification engineer's manual intervention.

## V. Summary of Results

This work introduces an alternative to manual code generation whereby a python script auto-generates verification code reducing development time and cost. The procedure has successfully created connectivity tests with voltage checkers in a span of minutes as opposed to the days it would require when using the typical manual coding approach. This work created almost 600 checks during a complex ADI pin electronics project using over 2000 digital pin connections. Productivity gains allow the verification engineer to spend more time on other mission critical tests.

Future work planned to expand this work are as follows:

- Expand script to allow for more complex gating conditions.
- Add capability to cater UVM testbench to reach wider set of users so they can also benefit from this automation.


## Acknowledgment

The authors would like to express their gratitude to ADI ATE Design Verification Team - Carl Heerbrugg Ramirez, Roberto Suarez-Valle and Anton Mark Sumollo, for their valuable feedback during the development of this work. The authors would also like to acknowledge the ADI ATE Design Team, and ADGT IMM Leadership for the support, resources, and suggestions on the pursuant of this work.


## References


[1] Wilson Research Group and Siemens EDA, "2022 Functional Verification Study", *November 2022*