SoC Verification of Analog IP Integration through Automated, Formal-Based, Rule-driven Spec Generation

Murugesh Palaniswamy[1], Ravi Kalyanaraman[2], Gargi Sharma[3], Bharat Baliga-Savel[4]

[1]Senior Staff Verification Engineer (Synaptics, Inc. San Jose, CA, USA.
Tel: +1-408-904-1868 Murugesh.Palaniswamy@synaptics.com),
[2]Director of Design Verification (Synaptics, Inc. San Jose, CA, USA.
Tel: +1-408-904-1933 Ravi.Kalyanaraman@synaptics.com),
Application Engineer (Mentor Graphics, A Siemens Business, Fremont, CA, USA
Tel: +1-510-354-7400 Gargi_Sharma@mentor.com),
[4]Functional Verification Application Engineer (Mentor Graphics, A Siemens Business, Fremont, CA, USA.
Tel: +1-510-354-7400 Bharat_Savel@mentor.com).

***Abstract*- The increasing trend to pack more functionality into SoCs has resulted in the inclusion of complex analog IPs alongside digital IP blocks. While most of the full chip focus is on functional integration verification, a rigorous static and conditional connectivity verification among the digital and analog IPs is of equal importance in the SoC integration verification process. An incorrect connection to an analog IP can prove fatal and cost a silicon re-spin! It is imperative to therefore verify the connectivity map of these analog IPs. Formal connectivity applications require a valid connectivity specification to verify the default values and the connectivity states. The biggest challenge for SoCs is in extracting this connectivity specification by tracing the design connectivity. Manually generating such a specification is tedious, error prone and cannot satisfy the needs of a quick turnaround in the late stages of the project cycle. Consequently, we developed and deployed a new, formal-based flow that can read the entire design and extract such a connectivity specification. The connectivity specification creation process required feeding customized rules to the formal connectivity EDA tool to guide it during the connectivity traversal process. While connectivity verification using formal techniques is not new, the challenge here is in creating a golden connectivity spec that uses these techniques mentioned above. Rules were defined to handle every unique connectivity type for various analog IPs, and these rules were then fed into the tool to extract the connectivity information as a specification. In this paper, we present the details of this new approach and how we have deployed this tool successfully into our projects. We have seen a big boost in productivity and accuracy of specification generation, with a time savings of almost 95% compared to prior methods -- from 120-160 hours with the old process down to few hours with the new methodology.**

I. Introduction

Today's complex SoCs are made up of numerous digital and analog IP blocks that are obtained from internal libraries and third-party vendors. The digital IP blocks are for supporting various features in an SoC (e.g., USB3, PCIE, SATA, Gigabit Ethernet etc.), while the analog IP blocks constitute functionalities such as signal conditioning, clock synthesis, etc. (e.g. PLLs, ADCs, PHYs). Typically, a lot of attention is paid to the functional integration aspect of the IP blocks to the other blocks within the SoC. However, the correctness of connectivity among the IP blocks in the SoC needs to be addressed with equal importance as well, as an improper connection bug can result in a re-spin!

Generation of the connectivity specification for each analog IP usually involves a lengthy manual process. Because connectivity verification is often pushed to the final stages of a design cycle, this tedious and manual effort often consumes critical verification resources late in a project. The problem is exacerbated by the restricted and not-so-thorough documentation from the respective analog IP providers.

II. Problem Description

Connectivity verification of an analog IP (which will be referred as DUT hereinafter in this paper) refers to the verification of all port properties and connections of an analog IP (PHY or Macro) to their respective intended specifications. Typically, the ports of an analog IP fall into the following types of interfaces but can include other types depending on the nature of the analog IP:

- Power (voltage/current)
- Functional
- Scan

- External communication interfaces to components outside the SoC.

To see this in the context of a typical SoC, refer to Figure 1 below.

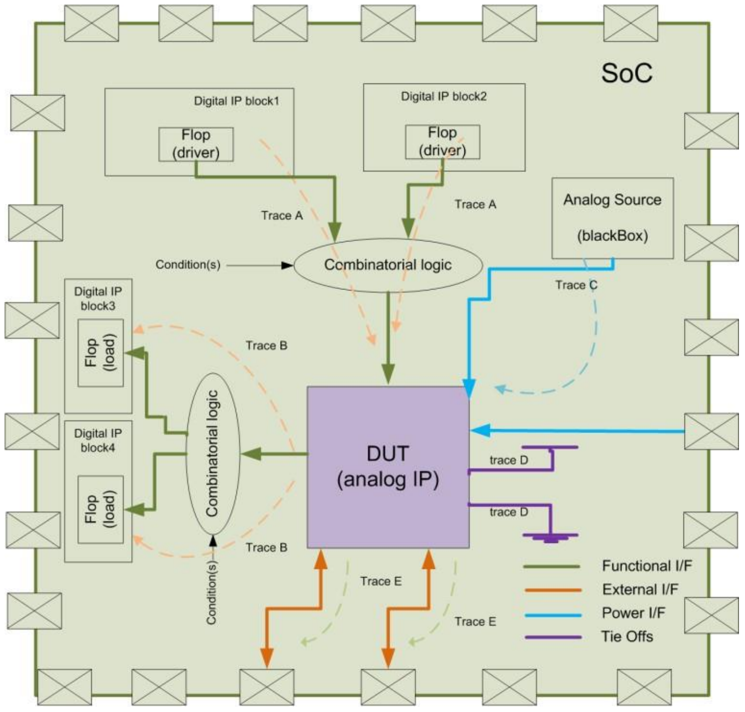


Figure 1: IP Connectivity paths in a typical SoC

The basis of SoC analog IP connectivity verification is determining all the driver(s) of each of the input ports of the DUT (i.e., the analog IP), and the load(s) of each of the output ports of the DUT that are connected to it in a SoC (i.e., the digital blocks).

As shown in Figure 1 above, this entails the following start point/end point types:

(a) The first driving flop output or a primary port of the SoC that connects to an input port of the DUT (trace A in Figure 1)
(b) The first destination flop input or a primary port inside the SOC that connects to an output/input port of the DUT (trace B),
(c) The path from any specified black box (trace C) to the DUT,
(d) Tie-offs on any ports of the DUT (trace D), and
(e) External primary ports to ports of the DUT (trace E).

To manually extract all the above types of connectivity traces takes a lot of manual verification effort.

III. Prior Approach

As mentioned above, the conventional approach is waveform tracing or RTL code tracing. As such, the verification man-hours for this step increases linearly with the number of ports of the DUT. For example, with a 200MG gate SoC with an internal configuration common to our products, this typically takes about 12-14 verification man-hours for an IP with an average size of 100 ports.

The next step involves creation of the connectivity tests themselves, which verify the connectivity through force/release constructs of Verilog. Fortunately, much of this step can be scripted such that the effort and time required is fixed irrespective of size or number of ports on the IP.

Finally, running the tests themselves, takes a considerable amount of compute time, followed by results analysis. Again, to some extent the results analysis is automated with scripting to flag obvious failures, but overall, the simulation and review cycle can take weeks of tedious, error prone work. As each new SoC we develop is larger than the previous device, it is clear that the engineering and computing resources consumed by this approach would soon outstrip what we could economically provide, while still try to keep within our project schedule constraints.

IV. NEW FORMAL-BASED APPROACH

Given that the prior solution can no longer adequately scale, our new solution is to "change the game" entirely by adopting a hybrid approach based on simulation and formal-based techniques to exhaustively verify the internal SoC connectivity – both the static, point-to-point connections and the myriad of muxed/switched "conditional connectivity" points. The modified flow is shown in Figure 2 below.

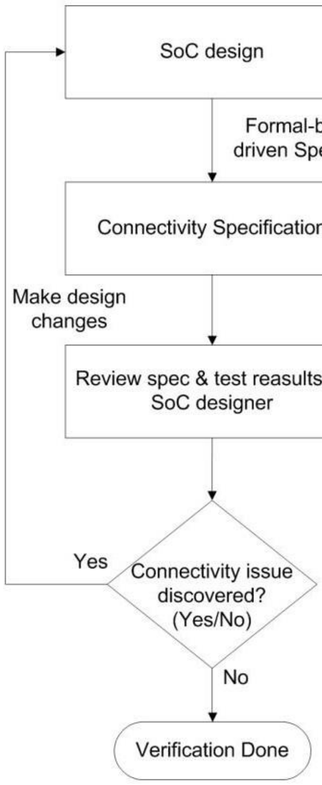

SoC design
Formal-based, Rule-driven Spec generation
Connectivity Specification
Make design changes
Review spec & test reasults w/ SoC designer
Yes
Connectivity issue discovered? (Yes/No)
No
Verification Done

Figure 2: New connectivity specification generation

The key steps are as follows:

A. *Formal-based, rule driven spec generation*

Recall that the initial operational challenge we faced is that the connectivity specifications for all IPs inside the SoC were either poorly documented, or somehow became obsolete (e.g., due to engineering change orders (ECOs) not propagated into the written specification quick enough). As such, even if inputs are suspected to be imperfect, the process needs to start somewhere. Hence, we start this whole process accepting that the baseline, fully assembled SoC probably has as-yet undiscovered issues. Indeed, we leverage this cynicism to our advantage later by employing an exhaustive formal analysis.

To automate this "spec generation" process for connectivity, we collaborated with an EDA vendor to leverage their existing tool that already reads in a design, and augment this tool to develop the capability that would read the connectivity directives (such as connectivity trace levels, conditional expression) supplied and export this information in a human and machine-readable specification. This tool is guided by specific rules that we write up-front which eventually gets fed into a formal engine that extracts the connectivity specification against the DUT.

We came up with an *Input specification format* and the *Path traversal rules* that is then taken in by the tool to generate this specification. The Input specification format lists the hierarchical start points and the direction of traversal (i.e., from core to analog IP or from analog IP to core). The Path traversal rules specify rules that control the traversal such as navigating through junction points in combinatorial logic, navigating through flops, the stopping condition for the traversal etc.

Example rules are as follows:

```
:
:
-inst<instance_name>: traces the connectivity of this particular instance of the analog IP
 -sig <name>: traces both sides of the connectivity of signal/port belonging to the above instance
:
 :
```

Figure 3: Sample input file used by the formal tool

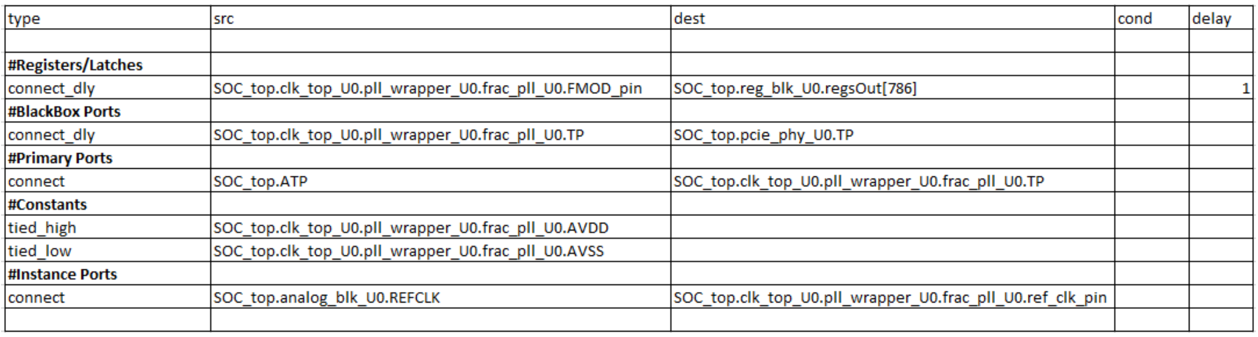
Additional switches can be added to control the direction of traversal.

Again, these rules and the DUT code are input to the "Connectivity Explorer" program, which generates a human and machine-readable specification. One of the salient features of this automated tool-based approach is that the connectivity information of all the analog IPs can be extracted with a single compiled database of the SoC, significantly reducing verification time and effort.

B. Creating & Reviewing the Connectivity Specification

As noted above, we don't "trust" our IPs to begin with, so after the first pass of the automated spec generation we perform a brief manual review of the spec before sharing it with our design counterparts. This doesn't take much time, but it often exposes the more blatant issues that can be quickly corrected.

After the initial review and correction cycle, we rerun the spec extraction and generation utility to produce a "golden" baseline spec that we feed into the formal analysis part of the flow.



type	src	dest	cond	delay
#Registers/Latches				
connect_dly	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.FMOD_pin	SOC_top.reg_blk_U0.regsOut[786]		1
#BlackBox Ports				
connect_dly	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.TP	SOC_top.pcie_phy_U0.TP		
#Primary Ports				
connect	SOC_top.ATP	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.TP		
#Constants				
tied_high	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.AVDD			
tied_low	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.AVSS			
#Instance Ports				
connect	SOC_top.analog_blk_U0.REFCLK	SOC_top.clk_top_U0.pll_wrapper_U0.frac_pll_U0.ref_clk_pin		

Table 1 - Sample snippet of connectivity map generated by the formal tool

C. Running the formal analysis

The specification – and the DUT Verilog code -- is provided as input to the "Connectivity Check" formal application. This tool automatically derives all the properties that describe the static and dynamic connections, then executes a formal analysis on each property to mathematically prove whether the property is true for all time and all inputs – i.e., does the specification match the actual circuitry. All this is automated, so it is not necessary for the person performing the verification to know how to run a formal analysis – the tool performs the tasks and reviews the results automatically without human intervention.

Additionally, the extracted connectivity information can be automated to regress throughout the design cycle by employing formal based techniques to catch any inadvertent bug slip. In this way, even if the baseline was flawed, the successive regression runs shake out any such issues.

D. Results Processing and Review

When the actual connections match the specification – in formal terms, if the connectivity properties are formally proven – the successful results will be reported in a log file. In contrast, the detection of errors by this flow is illustrated in waveforms called “counter examples”. In this context, if port A is supposed to be connected to port B, but it’s connected to port C, the tool will generate a waveform that shows how the signal emanating from port A can activate port C. In a nutshell, the tool delivers clear “root cause” result, which makes debug and fixing of issues very straightforward.

IV. Putting This Flow Into Service

Using a 200 million gate SoC as a pilot design for this flow, we devised a set of rules to be read into the tool to extract the trace paths mentioned above. One of the salient features of this automated tool-based approach is that the connectivity information of all the analog IPs can be extracted with a single compiled database of the SoC, significantly reducing verification time. Also, the extracted connectivity information can be automated to regress throughout the design cycle by employing formal based techniques to catch any inadvertent bug slip.

V. Results

This new formal-based flow is very easy to set up for new projects, and does not require knowledge of formal or assertions. The new process flow has been validated on an SoC having approximately 25 analog IPs, and over 200MG in scale. It has significantly reduced the connectivity verification man-hour effort from 120-160 hours using the old process, down to 3-4 hours -- roughly a 95% savings of verification effort with much higher quality of results due to the exhaustive nature of the formal analysis performed under-the-hood.

VI. Acknowledgment

We would like to thank Mr. Bill Au from Mentor Graphics, for the technical engagement and refining the formal EDA tool to support our requirements.

VII. References

(1) Nuni Srikanth and Maddipatla Shankar Naidu, Lakshman Easwaran, DVCON, Sept 25-26, 2014. Expediting Verification of critical SoC components using Formal methods.

(2) Ram Narayan, Oracle Labs, Austin, Texas.U.S.A. The future of formal model checking is NOW!