2025 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE
MUNICH, GERMANY
OCTOBER 14-15, 2025

Out of The Box Techniques for Data-Path Verification

Atharva Mahesh Kakde, Cadence Design Systems, Bengaluru, India (atharvam@cadence.com)

Ketki Gosavi, Cadence Design Systems, Bengaluru, India (ketkig@cadence.com)

Pradeep Bagavathiappan, Cadence Design Systems, Bengaluru, India (bpradeep@cadence.com)

Anshul Singhal, Cadence Design Systems, Noida, India (anshuls@cadence.com)

***Abstract*— Algorithmic datapath designs are typically modelled at a high level in C/C++ and can be verified at early stages before the corresponding RTL design is ready. The criticality for the equivalence verification of datapath designs (RTL) with their reference high-level C/C++ models, is well accepted. Simulation-based approaches like DPI or scoreboarding suffer from challenges of achieving verification completeness. The means of verifying this equivalence is shifting from simulation to formal gradually, thanks to new solver capabilities available in commercial formal tools and the exhaustive nature of formal. However complex image processing algorithms often run into the challenge of compiling i.e. building the formal model from C++ of in reasonable time. Also, we often struggle to achieve proof convergence on the equivalence check targets, even if we manage to compile the C/C++ models. While formal methods can be potentially effective, the right methodology is required to overcome these challenges and scale to larger and complex designs. In this paper, we propose several generic techniques that have helped us to generate formal models for FFT and Decompression units in a couple of minutes and converge on the properties which was impossible to achieve out of the box. In this process, we uncovered bugs that traditional verification methods failed to detect. All these techniques are reusable. This paper presents details of these techniques by taking an FFT and a Decompression algorithm as an example.**

Keywords*— *"Formal Verification" "Datapath verification" "C vs RTL equivalence checking" "Convergence techniques" "Image processing" "C/C++" "FFT" "Decompression" "assume guarantee" "case split" "system verilog assertion modeling"

I. INTRODUCTION

Nowadays, a variety of digital image processing algorithms are an integral part of many biomedical and industrial applications for recognition systems, digital watermarking, medical imaging and diagnosis, gaming, camera sensing, and so forth. Some of the commonly used image processing algorithms consist of Fast Fourier Transform (FFT) to decompose an image into its sine and cosine components for image enhancement. Compression/decompression algorithms are used to reduce/restore the image size by reducing redundancy of the image data to be able to store or transmit data in an efficient form [1]. Such algorithms can process a huge amount of data and are often categorized as datapath units owing to the data manipulations they do. They typically have huge number of scenarios to verify which are impossible to simulate exhaustively in reasonable time and compute resources. Most datapath algorithms are developed at a high level in C/C++ first and RTL designers use these models as a reference while implementing the RTL. Hence, it makes sense to verify the RTL by proving its functional equivalence to the C/C++ model. Traditional approach is to verify this in simulation using score boarding and DPI (Direct Programing Interface) that many commercial simulators provide [2]. This can have challenges such as building up heavy testbenches, need multi seed constrained random runs to maximize scenarios covered, may sometimes need directed tests to test the individual functions of C/C++, etc. Simulation alone can never exhaustively cover all the scenarios to achieve verification completeness. For example, for a simple 32x32bit integer multiplication, there are 2^64 input combinations and verifying them all with simulation can take thousands of CPU years [3]. And nowadays all designs have many such functions and when it comes to floating point arithmetic, the problem becomes manyfold harder [4], [5]. Even if we rely on random simulations, there is a risk of missing a corner case scenario, which can be very costly to uncover late in the design cycle. Formal verification mathematically validates the equivalence between two models. It also conducts an exhaustive exploration of all possible behaviors implicitly as against to the explicit exponential number of testcases that simulation and emulation require [6]. Datapath verification is beyond the capacity of the traditional bit-level formal engines; hence, they require special solving techniques. Formal methods were historically intractable for these problems and were known to work best with control path designs. But new solvers and techniques in the formal tools are making it possible to mathematically verify such datapath designs [7]. Using these formal methods, one can

efficiently prove the critical functional equivalence between RTL and its golden reference C/C++ model. Due to its nature of exhaustiveness and lack of need of elaborate testbench, formal equivalence checking for datapath designs is accelerating chip design in the era of More-than-Moore [8], [9]. The C vs RTL equivalence checking is further challenging as the internal structures, data sizes, dataflow, order of operations can differ significantly between the high-level models and RTL. Formal tools internally build a mathematical formal model of a C/C++ model, RTL and checks if these formal models are equivalent. Based on the complexity of the algorithm and the huge number of scenarios to exercise, we often run into challenge of generating formal model in reasonable time frame. Further, for the units where formal model is generated, we often struggle with equivalence check proof convergence. To take advantage of formal methods on such complex designs. In this paper, we propose several techniques that are generic in nature and have helped overcome these challenges at datapath units having image processing algorithms. We would like to cite some of the sate-of-the art work done in datapath verification using formal verification – Tackling control and datapath verification [10], A 101 guide for datapath verification [11], Solving datapath proof convergence [12], Prediction algorithm simplified using formal verification [13].

II. DESIGN DETAILS

The problem at hand – Image Processing Unit, had a lot of data transformations and simulation could have never covered all possible input combinations. This unit mainly had FFT, compression, decompression and Inverse FFT algorithms as shown in the figure 1.

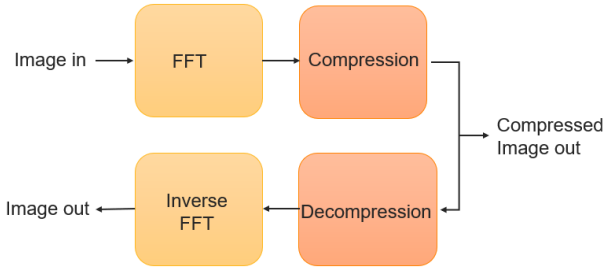

Figure 1: Image Processing Unit Block Diagram

We had C++ models for each of these algorithms and separate RTL boundaries for them, which made it easy to create independent setups for each. In this paper we are going to focus on FFT, and decompression specific flow and methodologies implemented to check their C vs RTL functional equivalence. The FFT unit implemented a 2048-point FFT and consisted of a large data_in, butterfly, twiddle factors, scale and a 8-point FFT operation as the base. It had sine and cosine lookup tables and specific values that were passed to internal functions. At the lowest level, it performed shift, add and complex multiplication operations in each stage. The previous stage output was fed to next stage as input. The corresponding RTL implementation was using a similar flow of functions, making it more suitable target for C vs RTL equivalence checking. For the whole image of a large data stream to process, the core 8-point FFT function was called 16K times with a specific pattern of data_in and twiddle factors in the C++ code. Whereas RTL was processing a single data sample per clock cycle.

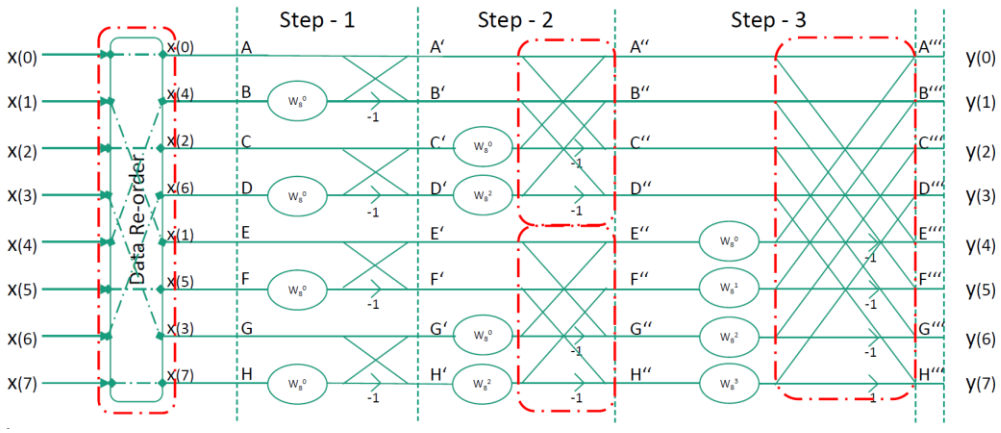

Figure 2: FFT Model

Another setup was targeting decompression logic from the image processing unit. The decompression core algorithm was referenced from an open source [14]. The input to the decompression unit was the compressed data stream of variable size up to 4K bytes, in terms of a set of instructions. Based on the instructions type, the corresponding internal functions were executed. The output was a decompressed stream of data of size max 4K bytes. The C++ model processed the entire data in one go, whereas RTL was taking multiple valid cycles to do that.

III. FLOW USED

This section explains the setup flow for formal equivalence checking solutions offered by EDA vendors. It is based on a miter model with the equivalence goal as - for the same set of inputs to C/C++ model and RTL, both are generating the same outputs, taking into account the appropriate pipeline delay. Figure 3 illustrates the miter model having corresponding inputs and outputs from C/C++ model and RTL connected, referred as “mappings” [15]

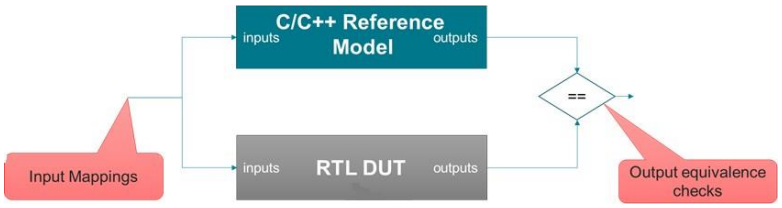

Figure 3: Miter Model

The input signal mapping generates assumptions to ensure both model and RTL gets same inputs and output signal mapping generates assertion to check the equivalence between the two. C/C++ specification model is untimed, and RTL can be timed, so this mapping can have fixed or variable latency involved. Below are the basic setup steps we followed in our FFT and decompression setups.

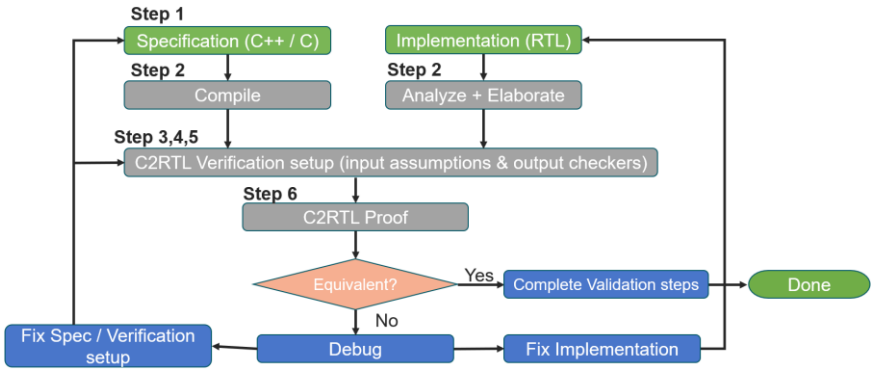

Figure 4: C2RTL Flow Diagram

Step 1: The entry-level function (main) at C/C++ model typically does not have interface signals. So, to create the signal mapping with RTL, we instrumented the C/C++ model to specify its inputs and outputs using EDA tool defined macros.
Step 2: This instrumented C/C++ model is then compiled to generate a formal model. Based on the model complexity, it is possible that the formal model generation may take a long time. RTL was analyzed and elaborated as part of bring up next.
Step 3: When the C/C++ and RTL ports have matching hierarchical names then corresponding ports get automatically mapped by the EDA tool that we used. But as is often case, ports at these units had different naming and data sizes between C/C++ and RTL, so we used manual mappings. For input and output signal mappings we added SVA assume and assert properties, respectively.
Step 4: For the extra functional ports on RTL, we added required constraints on them to ensure their valid values using SVA assumes.
Step 5: Once all properties were added, environment settings such as clock and reset declaration were needed before proving the properties.
Step 6: We then proved the equivalence assertions and any related cover properties.

This flow automatically created sanity assertions for the C/C++ model to catch issues in the formal model generation, which can lead to false results. Typically, these assertions are generated to catch issues related to the completeness of the formal model generated from the C/C++ code. If any of these assertions is falsified by the formal engines, the compilation of the software model might be incomplete; therefore, any results based on such model might not be reliable [14]. Hence before proving the equivalence checks, we proved these sanity assertions and took corrective measures if there are any failures. One of the sanity assertions that failed in our case was to check whether a loop (for/while) or recursive function has more iterations or calls than the specified unroll limit during the compilation. We checked whether the unroll limit needed to be increased at compile time or, if the loop iterations were controlled by a free variable, then do we need to add assumptions to limit the unwanted iterations. After the sanity checks were proven, we proved the C2RTL equivalence checks.
Formal is very quick to find failures, if any. We noted that initially there can be setup issues which need to be fixed and rerun but once the setup is clean, then only the specification/ implementation model needs to be debugged and fixed until we get the equivalence. The visualize waveform debugger, available in the formal tool that we used, was employed to debug the failing properties and root cause the failures.

IV. CHALLENGES AND SOLUTIONS

1) Formal model generation

We created separate setups for decompression and FFT units from our image processing block. The input for the decompression function was the compressed data stream of variable size up to 4K Bytes, in terms of set of instructions. When we tried to compile the original C++ decompression function, the process was stuck at formal model generation for more than 3 days. The profiler available in the tool, helped us understand the complexity bottleneck and we made changes to the loop iterations. However, because model generation was still not through, we tried to take an instruction-based restricted setup approach. In this case, at compilation itself, we specified the instructions for which the code will run and for all other instructions, the code will exit. This is different from the assumptions that we apply, as the assumptions will come in picture after compilation and would only help in the proof. But what we added here is the over constraint at compilation itself. With this, the formal model generation for decompression was completed in a few minutes. The details of the approach are described in the following sections.

a) Compile time over constraint

We applied over constraints to restrict the formal model for a certain set/range of inputs. This in turn reduced the size of the generated formal model. We used "*return 0*" if the inputs do not lie under a certain set/range of inputs and we can generate multiple smaller models with a different set/range of input. For example, for a 16-bit signed int data type, its values can vary from -32768 < 0 < 32767; that is, its formal model is to be generated for (2^16) = 65536 possible values. But when we say return 0 for all the values less than 0 or greater than 256 using "If (c_input < 0 || c_input >256) return 0;", then model generation will only happen for values from 0 to 256. This in turn reduced the size of the formal model generated.

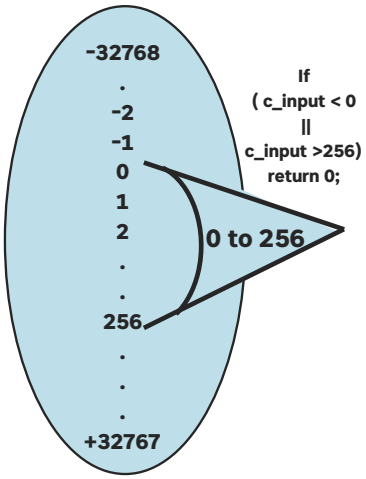

Figure 5: Compile Time Over Constraint

In this manner, we worked on multiple ranges based on the algorithm instructions. Although they were not covering the entire state space, we focused on critical instructions. However, the same technique did not work with the FFT unit. Because there were too many input data samples, it was not practical to cover them with a handful of restricted setups. As a result, we opted for a divide and conquer approach.

b) Divide and Conquer

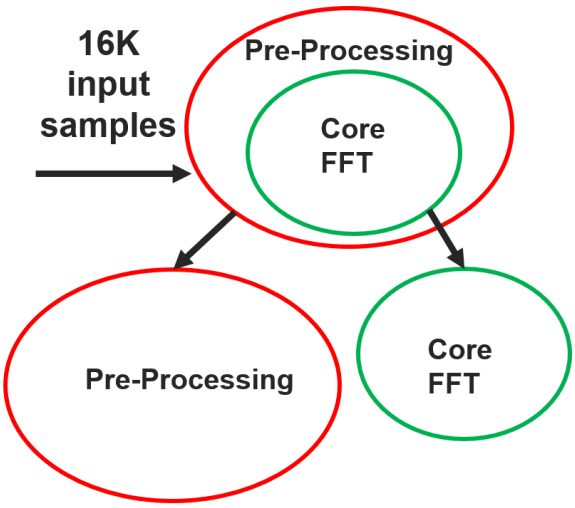

Figure 6: Verifying pre-processing and core FFT separately

The FFT unit consisted of 16K data samples pre-processing and was passed to the core FFT function. With the help of the designer and simulation team we understood that the RTL also has separate module for core FFT functionality. So, we divided this setup into two separate setups. One where only preprocessing was checked and the other where the core FFT logic was verified. For the second setup we still had loop iterating 2K times to process 8 data samples per 8-point FFT function call. It was still challenging to generate the formal model for this part. Therefore, we explored the symbolic variable approach.

c) Symbolic Variable

We got the working equivalence check setup with over-constrained input data to the first 8 fixed reference values. We needed to get these eight data samples symbolically represent any eight data samples out of 16K samples to cover all the scenarios. We added a few assumptions in the C++ code to symbolically pick 8 twiddle and their corresponding scale values with random data. As these signals will now be driven as per the assumption, we added cutpoint on their corresponding signals on RTL side and assumed them to have same value as generated in C++ model. Later we proved these assumptions separately. This greatly helped in downsizing the formal model and it was generated in matter of seconds.

2) Convergence

Once we got the decompression formal model generated, there was the challenge of getting the convergence on the equivalence check. Each formal property has a cone of influence logic and the complexity of this logic impacts the proof performance. Datapath problems are very different from control logic problems. Usually, datapath problems don't have much sequential depth, and the complexity is mainly due to arithmetic operations. Specialized datapath engines are used to prove such targets, but not all problems can be solved out of the box with these proof engines. In such hard-to-prove cases, a specific methodology and user guidance are needed.

a) Arbitrary Miter Model

The C++ model is untimed, but the decompression RTL required multiple cycles to initialize and exercise all relevant scenarios spread across time. In this case, multiple instances of the equivalence check need to be run as part of the standard miter model and this can make it difficult to converge.

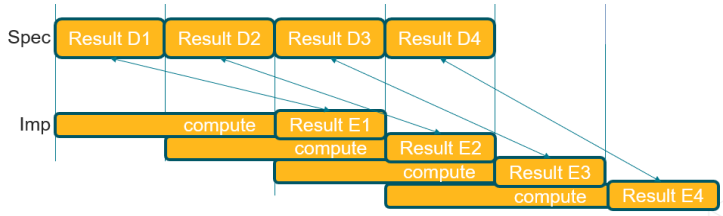

Figure 7: Standard Miter Model

Instead, we kept the RTL uninitialized, to start from any arbitrary state, making it cover all the scenarios in only a single transaction. For this approach, we might require additional constraints at the RTL to avoid invalid scenarios.

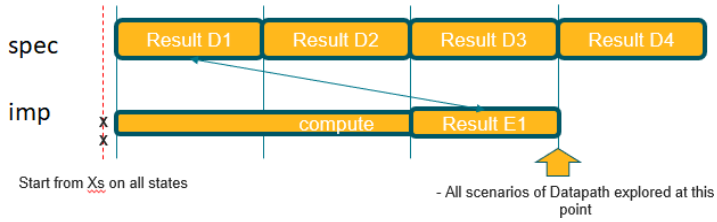

Figure 8: Arbitrary Slice Miter Model

b) Avoiding the use of $past

We had to compare the cycle 1 C++ output with the N^{th} cycle at the RTL, due to the RTL sequential delay. For this requirement, we used SVA assert like {$past(c_out, N) == rtl_out}. $past saves previous values of the signal in flops and then, evaluates at the required clock cycle. The higher the N value, the more the number of flops. When it comes to the C2RTL arbitrary slice miter where we are comparing a single transaction, we need to check the first instance of this assertion using a property like {first_cycle |-> ##N rtl_out == $past(c_out,N)}. We reduce the property's flop count and, in turn, its complexity by storing the c_out value from the first cycle and later compare when the rtl_out is available.

In formal $past will create a flop per cycle
Example :- Property comparing 1st cycle c_out to 100th clock cycle rtl_out

```
assert -name P1 {first_cycle |->
##100 rtl_out == $past(c_out,100)}
# Flops: 0 (201) (201 property flop bits)
```

Property flop count reduced:

```
virtual_net save
assume -bound 1 {save == c_out}
assume {##1 $stable(save)}
assert -name P2 {first_cycle |->
##100 rtl_out == save}
# Flops: 0 (101) (101 property flop bits)
```

Figure 9: Avoid $past example

c) Assume Guarantee

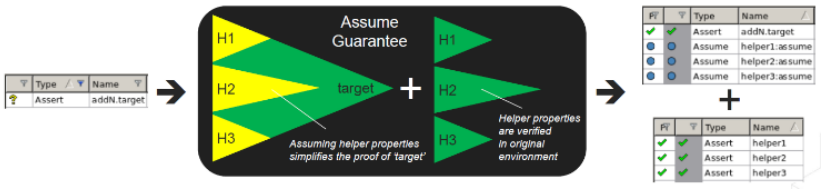

Figure 10: Assume Guarantee

We applied the assume-guarantee approach for the FFT stage by stage comparison when the final stage 4 equivalence check was not converging. In this technique, when end-to-end properties are hard to converge, one can target the intermediate stages of the design and prove them first. These intermediate proven properties act as assumptions for the top-level target property. Stage 1 equivalence was proven and was assumed for proving stage 2, and so on stage 3 and 4. This helped us to locate the stage wise complexity and the property convergence.

d) Case Split

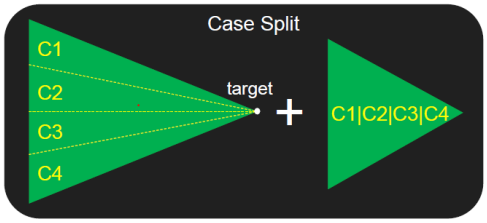

Figure 11: Case Splitting

When the property state space is huge, it can be difficult to target all the cases in COI. This can lead to non-convergence. Case split methodology applies case wise pre-condition to the target property. This generates multiple splits of target properties with reduced number of scenarios to check and ensures its completeness.

V. RESULTS AND CONCLUSION

Table 1 illustrates the results we achieved on instruction wise restricted setup approach for the decompression unit with input size up to 128bit.

Parameter	**Technique**	**Before application**	**After Application**
Model Generation	**1.a**	Unable to generate	<5 Mins
Prove Time (Single Instruction)	**2.a** **2.b** **2.e**	>1day	<10mins
Prove Time (Multiple Instruction)		Not Converged	Max 2 days
Bug found	**2.a** **2.b**	Not converged	2 bugs* found in <1min

Table 1: Decompression Result

We found two bugs in the decompression unit which was ran through simulation-based equivalence check already. One of the bugs was related to FSM hang scenario and other non-equivalence was due to unimplemented RTL scenario.

Table 2 shows the results achieved on the FFT unit. Here property failure showing non-equivalence was reported in seconds, stating stage wise tolerance differences in RTL and C++ model.

Parameter	Technique	Before Application	After Application
Model Generation	**1.b** **1.c**	Unable to generate	2mins
Prove time	**2.a** **2.b** **2.c** **2.d**	Not Converged	Stage 1 - 113 sec Stage 2 - 88 Mins Stage 3 - 36 Mins Stage 4 - 119 Mins
Bugs found	**2.a** **2.b** **2.c** **2.d**	Not converged	1Bug* found in <1min

Table 2: FFT Result

Many of these techniques are useful to other datapath algorithms such as AES, SAM, FMUL etc. Even in cases where full convergence was not achieved, this method helped us gain confidence to explore deeper bounds. We can also inject bugs and ensure that they are found as a part of negative testing. This formal approach does not need heavy testcase creation like in dynamic verification so it can be started early in the cycle and left shift the verification.

VI. FUTURE WORK

There are lot of advancements in this area and the scope of formal is widening. The future of this technology has many formal applications such as formal property verification for C/C++ models. Many automatic formal checks commonly performed for RTL designs can also be applied for high- level models. Furthermore, coverage-based flow to sign off on bounded proofs is underway.

VII. REFERENCES

[1] DIGITAL IMAGE PROCESSING: AN ALGORITHMIC APPROACH. N.p.: PHI Learning Pvt. Ltd., 2018.
[2] S. Konale and N. B. Rao, "C-based predictor for scoreboard in Universal Verification Methodology," 2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014), Unnao, India, 2014, pp. 1-5, doi: 10.1109/ICAETR.2014.7012913
[3] P. Kalla, "Formal verification of arithmetic datapaths using algebraic geometry and symbolic computation," 2015 Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, USA, 2015, pp. 2-2, doi: 10.1109/FMCAD.2015.7542240.
[4] A. Kapoor, W. Ferguson, H. Jain and S. Kundu, "Formal Verification of Floating-Point Division," 2023 IEEE 30th Symposium on Computer Arithmetic (ARITH), Portland, OR, USA, 2023, pp. 93-96, doi: 10.1109/ARITH58626.2023.00018
[5] NXP_Exhaustive Formal Datapath Verification Of RISCV-based Floating Point Unit
https://www.cadence.com/en_US/home/multimedia-secured.html/content/dam/cadence-www/global/en_US/videos/about-cadence/events/cadencelive/secured/2022/india/system-verification-advanced-verification-methodology/sva-2-exhaustive-formal-datapath-verification
[6] Li, Lun., Thornton, Mitchel. Digital System Verification: A Combined Formal Methods and Simulation Framework. Poland: Springer International Publishing, 2022.
[7] Article that states the 100x performance improvement with new solvers
https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/jasperc2rtl
[8] N. Kikkeri and P. . -M. Seidel, "Formal co-verification of pipelined datapaths," 48th Midwest Symposium on Circuits and Systems, 2005., Covington, KY, USA, 2005, pp. 104-107 Vol. 1, doi: 10.1109/MWSCAS.2005.1594050.
[9] B. Xue, P. Chatterjee and S. K. Shukla, "Simplification of C-RTL equivalent checking for fused multiply add unit using intermediate models," 2013 18th Asia and South Pacific Design Automation Conference (ASP-DAC), Yokohama, Japan, 2013, pp. 723-728, doi: 10.1109/ASPDAC.2013.6509686
[10] Ravindra Aneja et al, Tackling the Complexity Problem in Control and Datapath Designs with Formal Verification,Design Automatic Conference (DAC) 2019
[11] P. McLellan, “Datapath Formal Verification 101: Technology and Technique”, JUG 2021
[12] Disha Puri et al, Revolutionizing Proof Convergence for Algorithmic Designs: Combining Multiple Formal Verification Tools, Design Automatic Conference (DAC) 2023
[13] Demetrio Bori, Functional verification of prediction algorithms: make it simple with C vs RTL, Jasper User Group (JUG) 2023
[14] LZO Decompression algorithm - https://github.com/torvalds/linux/blob/master/lib/lzo/lzo1x_decompress_safe.c
[15] Jasper C2RTL userguide –
https://support.cadence.com/apex/techpubDocViewerPage?xmlName=jasper_c2rtl_userguide.xml&title=Jasper%20C%20to%20RTL%20Equivalence%20Checking%20App%20User%20Guide%20%20Contents&hash=&c_version=2023.03&path=jasper_c2rtl_userguide/jasper_c2rtl_userguide2023.03/jasper_c2rtl_userguideTOC.html