Breaking the Formal Verification Bottleneck: Faster and More Comprehensive Testing of Parameterized Modules

Menachem Rappaport, Ariel Ansbacher, Elchanan Rappaport
Veriest Solutions
9 Shimshon St., Petach Tikva, Israel
menachemr@veriests.com, ariela@veriests.com, elchananr@veriests.com

***Abstract*- This work addresses the challenges of verifying parameterized SoC (Systems on a Chip) designs using Formal Verification. Each parametrization is a new RTL, with a different internal model, and for that reason we require new Formal Verification runs to try to cover all possible configuration cases. Traditional methods, which involve testing each parameter configuration separately, result in increased test counts and longer runtimes often resulting in incomplete coverage due to schedules. These drawbacks have a direct influence on the verification quality, increasing the risk of missed bugs, potential increase on engineering effort and increased time to market. To reduce the verification time of complicated SOCs and improve the verification quality, we propose a novel solution, P2S (Parameter-to-Signal), which replaces design parameters with input signals to enable a single test case that covers multiple configurations. A custom compiler automates the conversion, eliminating the need for manual engineering effort, removing the possibility of human errors and ensuring functional equivalence. Using LEC (Logical Equivalence Checking) we are not just making sure that the two modules (one without applying P2S and the other one after we have applied P2S) are equivalent, we are getting a formal proof for it. Experimental results show that the P2S approach significantly reduces verification time, offering a more efficient method for verifying parameterized modules.**

I. Introduction

The design of SoCs and other integrated circuits frequently relies on parameters. These parameters are essential in configuring and controlling the behavior of various Verilog modules. Parameters serve multiple purposes, as shown in Fig. 1, including:

- Defining the width and depth of signals and arrays
- Specifying modes of operation
- Performing calculations with predefined values
- Affect timing

```
module advanced_fifo
  #(
    parameter FIFO_DEPTH = 8,
    parameter FIFO_WIDTH = 11,
    parameter NUM_LOOPS  = 3,
    parameter logic ADD_MODE   = 1
    )
```

Figure 1. An example of advanced FIFO interface that uses parameters

Imagine that for the Fig. 1 example we want to cover all possible configuration cases when we have the following legal combinations for the given parameters:

- **FIFO_DEPTH:** 2,4,8 (3 options)
- **FIFO_WIDTH:** 8-11 (4 options)
- **NUM_LOOPS:** 3-6 (4 options)
- **ADD_MODE:** 0,1 (2 options)

In practice, it will result in 96 separate formal runs as we need 96 different test runs to cover the behavior of all individual RTL instances. We are still using a simple RTL module as an example, and we are already at a regression

which has a significant number of test cases. In Formal Verification, mathematical techniques are used to prove the correctness of designs under all possible conditions. When verifying designs with parameters, it's crucial to ensure verification across all parameter configurations. This presents several challenges. Our goal is to highlight the shortcomings of traditional methods and present an innovative solution that simplifies, accelerates, and enhances the efficiency of the process, ultimately improving the verification quality of parameterized modules using Formal.

II. The Challenge in Traditional Methods

Since in Formal Verification, the goal is to verify all possible scenarios, the traditional approach for handling modules with parameters involves creating separate tests for each parameter configuration. However, this method introduces several drawbacks:

- **Increased Test Count:** Each unique configuration requires its own test, leading to a dramatic increase in the total number of tests.
- **Extended Runtime:** Running each test individually consumes significant time, lengthening the overall verification process
- **Bug Escape! :** When numerous tests with many parameter combinations are required, it's often not feasible to run them all. As a result, only a few key combinations are selected for testing, which reduces test coverage and increases the risk of missing bugs.

A common partial solution is to use the *generate* function to create tests with different parameters. In Verilog, *generate* allows repeated or conditional instantiation of modules or code blocks, making designs more efficient and adaptable. By Using the *generate* function and loops, we can flexibly create multiple instances or conditionally include code, reducing effort by enabling a single-command test execution. However, this still results in tests for all parameter combinations, maintaining a high-test count and long runtimes. Fig. 2 shows an example of how to use generate loops for each parameter combination.

```
generate

   for (gen_FIFO_DEPTH = 2; gen_FIFO_DEPTH <= 8; gen_FIFO_DEPTH++)
     begin: gen_FIFO_DEPTH_block
      if ((gen_FIFO_DEPTH ==  2) || (gen_FIFO_DEPTH ==  4) || (gen_FIFO_DEPTH ==  8))
        for (gen_FIFO_WIDTH = 8; gen_FIFO_WIDTH <= 11; gen_FIFO_WIDTH++)
          begin: gen_FIFO_WIDTH_block
           if ((gen_FIFO_WIDTH >=  8) &&  (gen_FIFO_WIDTH <=  11))

             begin

                logic  push ;
                logic [gen_FIFO_WIDTH-1:0] push_data ;
                logic                      pop ;
                logic [gen_FIFO_WIDTH-1:0] pop_data ;
                logic                      empty ;
                logic                      full ;

                fifo
                  #(
                     .FIFO_DEPTH (gen_FIFO_DEPTH),
                     .FIFO_WIDTH (gen_FIFO_WIDTH)
                     )
                u_dut
                  (
                    .*
                    );

             end
        end   //gen_FIFO_WIDTH_block
   end   //gen_FIFO_DEPTH_block

endgenerate
```

Figure 2. Use of generate loops to instantiate the block for each parameter combination

III. Proposal for a New Solution

A. *Proposed Solution*

Our proposed solution introduces an innovative approach by replacing the design parameters with input signals. This allows us to create a single, comprehensive test case that dynamically evaluates multiple parameter configurations, significantly reducing the need for numerous individual tests. As shown in Fig. 3, to ensure the input signals behave like parameters, we implemented specific assume properties to address two key aspects:

- **Stability of Signals Replacing Parameters:** Ensuring that each test scenario is consistently validated throughout the testing process.
- **Constraints on Signal Values:** Defining the signals to cover all possible values originally assigned to the parameters.

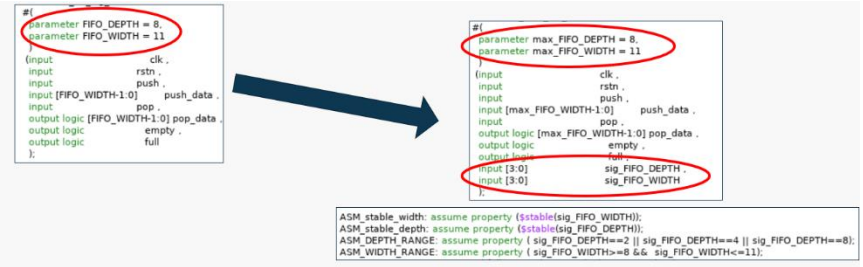


Figure 3. Write the Interface's Parameters as Input Signals

It's important to note that we couldn't simply replace parameters defining signal widths with input signals, as Verilog syntax does not allow signals to define other signals' widths. In such cases, we replaced these parameters with ones representing their maximum value, then created a mask signal to apply in assignments as shown in Fig. 4. The masking method employed in this process will be explained in greater detail in the conversion techniques paragraph.

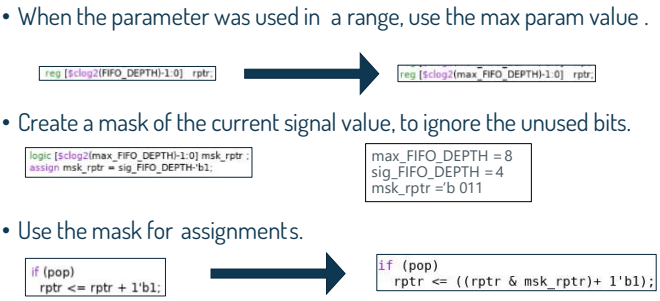


Figure 4. An example of P2S replacement process of signal's width parameters

B. Custom Compiler

To automate the P2S process, we developed a custom compiler that takes a Verilog module as input and automatically converts all its parameters into input signals and adds all the necessary logic. This ensures that the entire design adapts smoothly to the signal-based approach. The compiler leverages an Abstract Syntax Tree (AST) to represent and modify Verilog code. This approach involves parsing the Verilog source code to create an AST, transforming specific constructs—particularly by converting parameters into signals—within the AST, and regenerating Verilog code based on the modified AST. Currently, our implementation uses Pyverilog, a tool that uses YACC and Lex software for parsing and AST generation. However, to enhance parsing capabilities and support more advanced transformations, we plan to transition to a compiler architecture based on ANTLR4 software. This upgrade will allow for a more robust handling of Verilog's syntax and offer greater flexibility in future modifications. Fig. 5 represents the flow from parameter code to signal code.

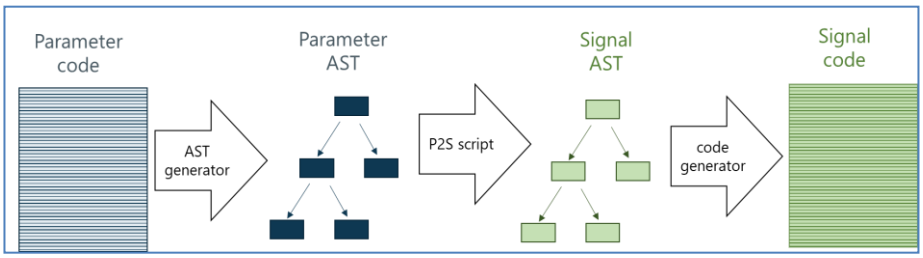


Figure 5. Flow from parameter code to signal code

As previously noted, directly converting all parameters to signals can lead to compilation issues. To address this, we initially convert specific parameters to their maximum values and apply necessary adjustments, such as masking, to ensure the new design's functionality matches that of the original. Additionally, modifications are made for other parameters that cannot be directly converted to signals, such as those used in for loop conditions. Once these adjustments are complete, the remaining parameters can then be safely converted to signals. This approach currently supports the blocks we have worked on, and the script will be extended to accommodate additional blocks as we

encounter new design requirements. Fig. 6 shows the main part of the script, which is responsible for converting the parameter AST to the signal AST.

```
## change all needed to "max_PARAM" with changes
#  update params widths and add masks
update_param_width(top_node,param_names,width_signals)
add_masks(top_node,width_signals,param_names)

# update param dimensions
update_param_dimensions(top_node,param_names)

# update param for
update_param_for_loops(top_node, param_names)

# update params in repetition
update_param_repetition(top_node, param_names)

# change all the rest to "sig_PARAM"
update_param_to_sig(top_node,param_names)
```

Figure 6. Steps for converting the AST

C. Conversion Techniques

To extend signals to their maximum size without impacting the block's functionality, we selectively apply masking techniques. Although masking is generally unnecessary, there are three specific cases where it becomes essential. First, masking with zeros is the default method; it is used during indexing or bitwise OR operations to prevent higher-order bits from interfering with the intended functionality. Second, masking with ones is required for bitwise AND operations to preserve specific bit patterns and maintain functionality. Third, in sign extension with two's complement, we apply a mask using the most significant bit (MSB) to extend the signal based on the value of the highest bit. These masking strategies ensure that signal extensions are accurate and functionally consistent with the original design. Table 1 shows some examples of the masks generated by the P2S script.

Table 1
Examples for generated masks by P2S script

Option	Examples		
	Operation	3 bits	8 bits
Mask with 0s	indexing	a[xxx]	a[0000_0xxx]
	Bitwise or	|(xxx)	|(000_0xxx)
Mask with 1s	Bitwise and	&(xxx)	&(1111_1xxx)
Mask with MSB	Signed	$signed(0xx) $signed(1xx)	$signed(0000_00xx) $signed(1111_11xx)

Implementing a for loop requires more than simply converting parameters to signals, as static Formal tools necessitate a predefined loop count. For basic loops, this can be addressed by incorporating a condition that utilizes the maximum parameter value, ensuring compatibility with fixed loop constraints. However, in more complex cases, an additional if condition within the *for* loop may be required to handle variable or conditional iterations effectively. This approach allows the loop to adapt to advanced scenarios while meeting tool requirements for defined iteration limits. Fig. 7 represents the way of constructing for loops with our P2S script.

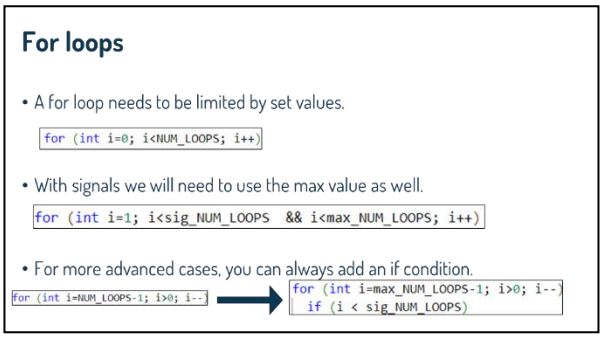


Figure 7. Way of constructing *for* loops with P2S script

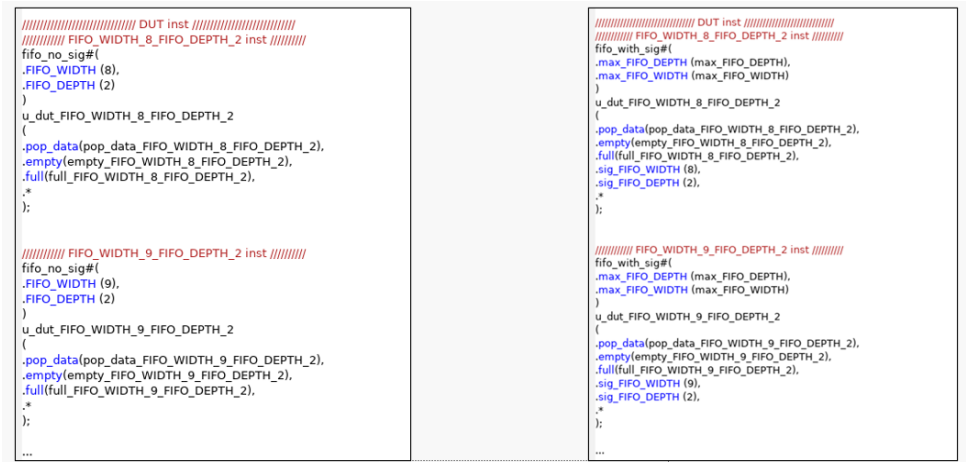
D. *Logical Equivalence Checking*

Since our compiler modifies the design itself, one concern we addressed was ensuring that the modified designs retain the same functionality as the originals despite these changes [1]. Furthermore, as the script has not been tested across all block types, it is essential to verify that the results align with the expected functionality. Additionally, some minor manual modifications were required during the generation of new blocks, as the script is still under development. These measures help maintain functional integrity while refining the script to support a wider range of design blocks.

For solving this problem, we decided to use an equivalence check tool to verify that the logic of the modified designs remains identical to the originals. Logical Equivalence Checking (LEC) examines the combinational structure of a design to determine whether two different implementations exhibit the same behavior. The process begins by matching all storage elements, such as flops and arrays, and then analyzes the combinational paths between these flops, as well as the connections to and from primary inputs and outputs. To prove that the original module is logically equivalent to the P2S module, we have run LEC. It's worth mentioning that the equivalence check process is straightforward, with an insignificant runtime for the blocks we tested.

Our verification method involved generating two design instantiations for each combination of parameters to be tested. The first instantiation used the original design with the parameters set explicitly, while the second used the modified design, in which parameters were replaced by input signals. For the signal-based design, we applied hard-coded values to drive these inputs. This approach allowed us to verify that, for each parameter combination, the modified design exhibited identical functionality to the original.

In our initial implementation using generate loops, we left the primary inputs and outputs of each block undriven, resulting in the LEC tool not reporting results for these primary signals. To ensure that the LEC tool evaluates primary inputs and outputs, we modified our approach. Rather than relying on generate loops, we used Python to explicitly write out every instantiation combination, thereby creating unique inputs and outputs for each instance. This adjustment allows the LEC tool to accurately report on the primary inputs and outputs, improving the verification process. Fig. 8 shows the implementations for which we have applied the LEC.



```
//////////////////////////////// DUT inst ////////////////////////////
////////////// FIFO_WIDTH_8_FIFO_DEPTH_2 inst //////////
fifo_no_sig#(
.FIFO_WIDTH (8),
.FIFO_DEPTH (2)
)
u_dut_FIFO_WIDTH_8_FIFO_DEPTH_2
(
.pop_data(pop_data_FIFO_WIDTH_8_FIFO_DEPTH_2),
.empty(empty_FIFO_WIDTH_8_FIFO_DEPTH_2),
.full(full_FIFO_WIDTH_8_FIFO_DEPTH_2),
.*
);


////////////// FIFO_WIDTH_9_FIFO_DEPTH_2 inst //////////
fifo_no_sig#(
.FIFO_WIDTH (9),
.FIFO_DEPTH (2)
)
u_dut_FIFO_WIDTH_9_FIFO_DEPTH_2
(
.pop_data(pop_data_FIFO_WIDTH_9_FIFO_DEPTH_2),
.empty(empty_FIFO_WIDTH_9_FIFO_DEPTH_2),
.full(full_FIFO_WIDTH_9_FIFO_DEPTH_2),
.*
);

...
```

```
//////////////////////////////// DUT inst ////////////////////////////
////////////// FIFO_WIDTH_8_FIFO_DEPTH_2 inst //////////
fifo_with_sig#(
.max_FIFO_DEPTH (max_FIFO_DEPTH),
.max_FIFO_WIDTH (max_FIFO_WIDTH)
)
u_dut_FIFO_WIDTH_8_FIFO_DEPTH_2
(
.pop_data(pop_data_FIFO_WIDTH_8_FIFO_DEPTH_2),
.empty(empty_FIFO_WIDTH_8_FIFO_DEPTH_2),
.full(full_FIFO_WIDTH_8_FIFO_DEPTH_2),
.sig_FIFO_WIDTH (8),
.sig_FIFO_DEPTH (2),
.*
);


////////////// FIFO_WIDTH_9_FIFO_DEPTH_2 inst //////////
fifo_with_sig#(
.max_FIFO_DEPTH (max_FIFO_DEPTH),
.max_FIFO_WIDTH (max_FIFO_WIDTH)
)
u_dut_FIFO_WIDTH_9_FIFO_DEPTH_2
(
.pop_data(pop_data_FIFO_WIDTH_9_FIFO_DEPTH_2),
.empty(empty_FIFO_WIDTH_9_FIFO_DEPTH_2),
.full(full_FIFO_WIDTH_9_FIFO_DEPTH_2),
.sig_FIFO_WIDTH (9),
.sig_FIFO_DEPTH (2),
.*
);

...
```

Figure 8. Instantiations for the LEC testing

E. *P2S Concerns*

Besides the advantages of the P2S solution, it also presents a few challenges. For example, writing the testbench becomes more complex. In certain situations, within SystemVerilog Assertions (SVA), only parameters—not signals—can be used, such as when applying delays, which are compatible only with parameters. Adapting these properties to accommodate signals can, in some cases, increase tool runtime.

IV. Experimental Results

For experimental purposes, multiple DUT's were used to increase the confidence that our approach works properly and more efficiently for various types of designs. We have applied P2S for different classes of DUT's, to show that our proposed solution is the one to go with when you want to achieve good verification quality across diverse design structures. The models were chosen to include cases of parameters affecting signal widths, modes of operation, calculations and timing. To ensure the designs were unbiased and objective, we enlisted the help of ChatGPT to generate our base designs, with some manual tweaking to reach correct functionality (We look forward to the day

when GenAI can reliably create correct designs). The following chapter contains a short description of the DUT's that were used to get the experimental results.

A. DUT Descriptions

ADVANCED FIFO: A synchronous FIFO is a First-In-First-Out queue in which the same clock is used for both writing and reading. The number of rows is called the depth of the FIFO, and number of bits in each row is called the width of the FIFO. The advanced FIFO is designed with the capability to add or subtract a specified number to the data stored within it. The precise operation and timing are determined by the NUM_LOOPS and ADD_MODE parameters. The following parameters have been implemented for the advanced FIFO:

- **FIFO_DEPTH:** Refers to the size of the First-In-First-Out (FIFO) buffer, which plays a crucial role in synchronizing data transfer (It can be set to 2, 4, 8)
- **FIFO_WIDTH:** Represents the data width of the FIFO elements, at the write port and at the read port (It can be set to 8, 9, 10, 11 or 2, 3, 4, 5)
- **NUM_LOOPS: S**pecifies the values utilized for the operation and establishes the timing associated with it. (It can be set to 3, 4, 5, 6)
- **ADD_MODE:** Determines whether the operation would be addition or subtraction (It can be set to 0 or 1)

ALU: An ALU (Arithmetic Logic Unit) is a fundamental building block in digital systems design that performs arithmetic and logical operations on binary data. It is mainly used by the processor for performing various arithmetic and logical operations like addition, subtraction, logical AND operation etc. It may have one or more than one operand and an opcode. The opcode will tell the ALU which operations to perform. If the processor is n-bit, then ALU will perform the operation on n-bit operands. Depending on the operation, the output data width may be different than the input data width. (e.g. for a two-input multiplier, the output data width is double the input data width) The implementation that we have used supports two input operands and a few signed and unsigned arithmetic operations (add, subtract, and multiply). The following parameters have been implemented for the ALU:

- **IWIDTH:** Represents the input data width of the operands and it is used to calculate output data width (It can be set to 8, 9, 10, 12 or 5, 6, 7, 8)
- **OP:** Represents the operation that the ALU instance is going to perform (It can be set to 0, 1, 2)
- **SIGNED:** Represents the way of processing the input operands, whether we are applying signed or unsigned operations (It can be set to 0 or 1)

APB_XBAR: The APB (Advanced Peripheral Bus) protocol is a fundamental part of ARM's AMBA (Advanced Microcontroller Bus Architecture) suite. The APB protocol is designed to offer minimal latency and power consumption while maintaining simplicity. Its non-pipelined, simple architecture makes it highly efficient for connecting peripheral devices that don't require high-speed data transfers. It is generally used for low-performance peripherals like GPIO, UART, timers, SPI, and other slow-speed modules. APB bridges operate between higher-speed buses like AHB or AXI and those low-speed peripherals, making it an integral part of many SoCs. The implementation that we have used is an APB Cross-bar. It is used to connect one or more APB compliant master devices to one or more APB compliant slave devices. A crossbar is a piece of logic aiming to connect any master to any slave connected upon it. The core consists of a collection of switches, routing the master requests to the slaves and driving back completions to the agents. It provides advanced routing capabilities, such as arbitration and prioritization, that can help optimize system performance. A crossbar is a common piece of logic to connect peripherals like memories, IOs and co-processors to the processor(s) in a SOC. The following parameters have been implemented for the APB_XBAR:

- **IPORTS:** Determines the number of ingress ports (It can be set to 1, 2, 3, 4)
- **EPORTS:** Determines the number of egress ports (It can be set to 1, 2, 3, 4)

AXI BRIDGE: An AXI (Advanced eXtensible Interface) Bridge is a hardware component used in systems that implement the AXI protocol, which is fundamental part of the ARM AMBA family of interconnect standards. It serves as an interface between two different AXI-based components or subsystems, enabling them to communicate with each other despite potentially having different configurations, clock domains, or data transfer protocols. It can remap or translate addresses between different address spaces used by the connected subsystems, allowing seamless communication even if the components have different address layouts. It can also perform width conversion, such as transforming 64-bit data to 32-bit or 128-bit depending on the needs of the connected modules. AXI Bridges often take care of different latency requirements by buffering and timing the data transfer between systems with varying

speed or processing requirements. We have created an AXI Bridge that supports address translation, width conversion and data buffering. The following parameters have been implemented for the AXI Bridge:

- **M_DATA_WIDTH:** Determines master data width (it can be set to 8, 16, 32)
- **S_DATA_WIDTH:** Determines slave data width (It can be set to 8, 16, 32)
- **ID_WIDTH:** Determines the width of the transaction ID (it can be set to 2, 3, 4)
- **ADDR_WIDTH:** Determines the address width (It can be set to any value supported by AXI)
- **FIFO_DEPTH:** Determines the depth of FIFO used for storing IDs
- **FIFO_DEPTH_SIZE:** Calculated as $clog2(FIFO_DEPTH)

B. Runtime Results

We ran tests on all the designs mentioned above. It is important to note that we did not run every possible property to test the design. Instead, we selected the more time-consuming tests and based our comparisons on these properties. The results are presented in Table 2. Due to differing licensing limits across tools, a direct comparison between tools was not feasible. Additionally, our objective was not to determine which tool is superior but to compare the runtime differences between the original design and the signal-based design. Therefore, all times have been normalized, and only the relative time differences are reported. We note that results for the APB design in the third tool were not obtained due to tool-related issues, which we were unable to resolve within the available timeframe.

Table 2
Run Time Comparison

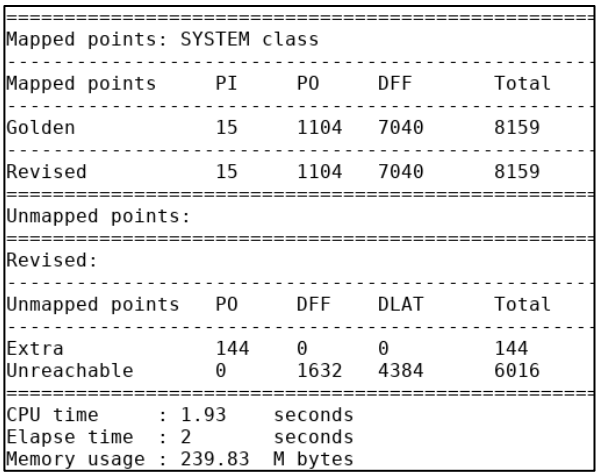
Design example	Number of permutations	Tool 1			Tool 2			Tool 3		
		PAR runtime	SIG runtime	Speedup	PAR runtime	SIG runtime	Speedup	PAR runtime	SIG runtime	Speedup
Advanced FIFO	96	1.0	0.387	**258%**	1.0	0.25	**400%**	1.0	0.155	**644%**
ALU	24	1.0	0.148	**675%**	1.0	0.196	**511%**	1.0	0.289	**346%**
APB XBAR	16	1.0	0.28	**357%**	1.0	0.300	**334%**	-	-	-
AXI Bridge	27	1.0	0.329	**304%**	1.0	0.67	**147%**	1.0	0.54	**184%**

The results show significant improvements across the tested blocks. The highest improvement reached 675%, while the lowest was 147%. In the vast majority of cases, performance improvements exceeded 300%. We have not yet analyzed why certain blocks exhibited more significant improvements than others. Future work will focus on studying these differences to determine the scenarios in which the methodology is more efficient and those where it is less effective.

C. LEC Results

In the LEC testing, we confirmed that there are no unmapped points in the original design, and all mapped points are equivalent. Consequently, we have established that the original DUTs are logically equivalent to the P2S DUTs across all designs tested. For illustration, we will present the results for the Advanced FIFO, noting that all designs exhibited similar results, demonstrating complete equivalence for all parameters.

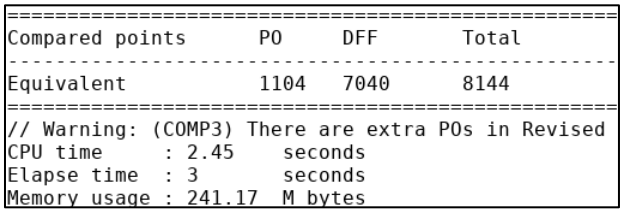
Fig. 9 illustrates the mapping results for the Advanced FIFO. In this representation, the original parameterized design serves as the golden design, while the modified signal design is identified as the revised design. We categorize the mapping points into four types: primary inputs (PIs), primary outputs (POs), D flip-flops (DFFs), and D latches (DLATs). All pins of the golden design were successfully mapped, with all unmapped points arising from the revised design. Unmapped points are anticipated due to the modified structure of the P2S DUTs, which utilize maximum parameter values for every instance instead of the specific parameters assigned to each instance.



```
================================================
Mapped points: SYSTEM class
------------------------------------------------
Mapped points     PI     PO     DFF       Total
------------------------------------------------
Golden            15     1104   7040      8159
------------------------------------------------
Revised           15     1104   7040      8159
================================================
Unmapped points:
================================================
Revised:
------------------------------------------------
Unmapped points   PO     DFF    DLAT      Total
------------------------------------------------
Extra             144    0      0         144
Unreachable       0      1632   4384      6016
================================================
CPU time     : 1.93    seconds
Elapse time  : 2       seconds
Memory usage : 239.83  M bytes
```

Figure 9. Mapping results for the advanced FIFO

Fig. 10 displays the comparison results for the Advanced FIFO. The results indicate that all compared points are equivalent, and there is no entry for inequivalent points, as none were found. A warning is noted regarding extra primary outputs in the revised design, which is again expected due to the use of maximum parameters. Both the mapping and comparison processes were executed with short and insignificant runtimes.



```
================================================
Compared points      PO     DFF       Total
------------------------------------------------
Equivalent           1104   7040      8144
================================================
// Warning: (COMP3) There are extra POs in Revised
CPU time     : 2.45    seconds
Elapse time  : 3       seconds
Memory usage : 241.17  M bytes
```

Figure 10. comparing results for the advanced FIFO

V. CONCLUSION

This study presents the P2S approach as a transformative solution for the formal verification of parameterized modules. By replacing design parameters with input signals, we achieved a streamlined verification process that not only reduced runtime but also enhanced test coverage. The results demonstrate substantial improvements over traditional methods, with speedups reaching as high as **675%** in some cases.

Our experiments highlight the ability of the P2S approach to address the critical challenges faced in verifying complex SoCs, such as increased test counts and extended runtimes. The custom compiler developed as part of this work along with LEC ensures that the transition to the P2S methodology is both efficient and functional, allowing for the verification of parameterized designs without compromising correctness.

Looking ahead, we aim to further refine the P2S compiler to accommodate a wider range of design scenarios and to continue exploring the potential of this approach in various applications within formal verification. By advancing the automation and applicability of the P2S method, we are optimistic about its integration into diverse design environments, providing engineers with a faster and more effective pathway to achieving high-quality verification.

REFERENCES

[1] Jonathan Bromley, Jason Sprott, Formal Verification in the Real World. https://dvcon-proceedings.org/wp-content/uploads/formal-verification-in-the-real-world.pdf, Accessed: 11.09.24.