Enabling Scalable Sequential Synthesis and Formal Verification in an Industrial Flow

Eleonora Testa
Synopsys, US

Dewmini Marakkalage
EPFL, Switzerland

Michael Quayle
Synopsys, US

Sudipta Kundu
Synopsys, US

Abhishek Kumar
Synopsys, India

Diptanshu Ghosh
Synopsys, India

Giulia Meuli
Synopsys, Italy

Giovanni De Micheli
EPFL, Switzerland

Luca Amaru
Synopsys, US

ABSTRACT

This paper introduces the first industrial flow capable to harness the full power of sequential logic synthesis accompanied by scalable sequential formal verification. Sequential synthesis is a generalization of combinational logic synthesis, exploiting the fact that not all combinations of flops values are reachable. This opens the opportunity to improve *Power Performance Area* (PPA) of digital circuits, at the cost of solving tougher computational design problems in the sequential world. As of today, industrial flows mainly focus on combinational synthesis and equivalence checking, with rare exceptions, e.g., clock gating or retiming, receiving ad hoc support. In this work, we tightly orchestrate sequential synthesis and sequential verification, enabling them to scale in an industrial *Electronic Design Automation* (EDA) flow. First, we present innovations on the synthesis engines, stemming from the concept of sequential sat-sweeping and extending to new strong sequential simplifications. Second, we alleviate the complexity of generic sequential equivalence checking by making our verification models synthesis aware. Finally, we show impressive PPA improvements on 10 commercial benchmarks post place and route, up to -3.6%/-5.4% area/power improvement accompanied by successful verifications, not attainable by other EDA flows.

KEYWORDS

Sequential synthesis, sequential formal verification, industrial EDA

1 INTRODUCTION

Logic synthesis and formal verification form the backbone of industrial *Electronic Design Automation* (EDA) flows. Sequential logic synthesis is a stronger notion than regular (i.e., combinational) logic synthesis, which is not bound by combinational equivalence and can harness the fact that not all combinations of flop values are in fact reachable. While sequential synthesis is widely known to be able to explore a larger solution space, and generally provide better *Power Performance Area* (PPA) in the implemented circuits [6, 7, 12, 13], formally verifying sequential equivalence reveals to be a much tougher task than combinational equivalence checking [8, 13, 21]. Providing proof of correctness of the implemented circuits via formal verification tools, is virtually a must for today's commercial design flows. Indeed, combinational equivalence checking is becoming increasingly scalable, and modern EDA tools provide strong solutions [1], also supporting some specific instances of sequential problems, e.g., clock-gating synthesis. Consequently, combinational logic synthesis is the most popular, and often only, flavor of synthesis deployed in commercial design flows. However, this means that some PPA opportunities may be missed in today's chips design, which is becoming increasingly expensive in the sub-10nm context where every mW and unit of area matter [2].

In this paper we enable for the first time true sequential logic synthesis in an industrial EDA flow, without compromising on the support of an automated and scalable formal verification solution. First, we enhance our logic synthesis flow to natively harness the sequential reachability nature of logic circuits. We improve on top of the most scalable sequential synthesis methods [13] and we extend our optimizations to unlock more PPA opportunities. Our method is scalable and can efficiently be applied to large designs within modern industrial EDA flows. Second, we describe an industrial verification flow which is synthesis-aware, combining the high capacity of combinational equivalence checking, for general logic, with the proving power of sequential verification methods, targeted for regions of logic where sequential optimization happened. The proposed verification flow is enhanced to make best use of hints and guidance provided by synthesis, in order to improve the convergence and runtime of both sequential and combinational verification tasks. Finally, we present experimental results showing important PPA improvements over 10 commercial benchmarks, up to

-3.6%/-5.4% area/power improvement and -2.0%/-1.9% on average post *Place & Route* (P&R), accompanied by successful verification for each synthesized circuit. Such improvements have not been reachable by traditional EDA flows.

The paper is organized as follows: Section 2 presents relevant background. Section 3 proposes our novel sequential logic synthesis algorithms and industrial flow, while Section 4 details the novel verification approach. Section 5 shows experimental results for 10 commercial benchmarks and Section 6 concludes this work.

2 BACKGROUND

Here we provide some background on sequential logic synthesis and verification.

2.1 Sequential Boolean Network

A Boolean network is a *Directed Acyclic Graph* (DAG) where nodes correspond to logic gates and directed edges are the wires connecting them. The sources of the graph are the *Primary Inputs* (PIs), while the sinks are the *Primary Outputs* (POs). The fanin of a node n is the set of input nodes driving the node, while its fanout is the set of nodes driven by n. The *Transitive FanOut* (TFO) of a node is defined as the cone of a node n with nodes reachable from n towards to the network POs. If the Boolean network is sequential, the memory elements are D flip-flops with initial states. All registers in a Boolean network are technology-independent, have one input D and one output Q, and are assumed to have the same effective clock. A Boolean network composed of only AND and inverters is called an *And-Inverter Graph* (AIG, [16]). Inverters are usually represented as attributes on the edges (complemented edges). Structural hashing of AIGs (strashed AIG) ensures that, for each pair of nodes, all constants are propagated and there is at most one AND node having them as fanins (up to permutation).

2.2 Sequential Synthesis

Sequential logic synthesis is a stronger notion of logic synthesis. While in regular (i.e., combinational) logic synthesis, optimizations need to preserve combinational equivalence, in sequential synthesis we can make use of the fact that not all combinations of registers' values are in fact reachable. While multiple works have focused on sequential synthesis in the past years [6, 7, 9, 12, 13], our work extend sequential logic synthesis to be fully integrated within an industrial flow, with automated verification support. We review here one sequential technique, known as *Sequential SAT-sWeeping* (SSW, [13]), which is part of the new proposed flow.

To introduce SSW, we first briefly describe *combinational SAT-sweeping* [17]. This is a technique for detecting and merging nodes that are equivalent - also up to complementation - in a combinational Boolean network. When *merging* a node n into m, the fanouts of n are transferred to m, and node n and its *Maximum Fanout Free Cone* (MFFC [14]) can be removed from the logic network. This results in logic optimization (e.g., reduction in the number of nodes). Merging is often applied to a set of nodes that are proved to be equivalent. Usually, one node of the class is denoted as the representative of an equivalence class, and all other nodes of the class are merged onto the representative. In the case of SAT-sweeping, the equivalence is proven using simulation and *Boolean Satisfiability* (SAT). In the sequential scenario, two sequentially equivalent nodes are such that they compute the same value, up to complementation, in all states reachable from the initial state. It follows that combinational equivalent nodes are also sequentially equivalent, but the contrary does not hold. An efficient implementation of SSW (based on strashed AIGs) is the one presented in [13]. The computation of sequentially equivalent classes is obtained using *induction* and it is referred to as *signal correspondence* [5, 18, 21]. We refer the reader to [13] for more details on the implementation.

2.3 Sequential Equivalence Checking

Sequential Equivalence Checking (SEC, [4, 19, 21]) techniques are used to formally check RTL-to-RTL sequential transformations. They are designed to compare two RTL designs and verify that they are equivalent on a cycle-by-cycle basis. The design before sequential optimization is the *specification* (or spec) and the design after sequential optimization is the *implementation* (or impl). This is in contrast to (traditional) *Combinational Equivalence Checking* (CEC, [10, 11, 15]) techniques that instead are usually involved to formally verify netlists synthesized using only combinational techniques [15]. SEC techniques have limited use in modern industrial EDA as compared to CEC techniques, as they have scalability issues due to state-space explosion problems inherent to model checking algorithms [8].

SEC is a specialized form of model checking algorithm and uses verification engines such as induction, interpolation, binary decision diagrams and property-directed reachability [4, 10, 19, 21]. For a given SEC verification, there are three possible outcomes: (i) All assertions are proved, meaning that the spec is sequentially equivalent to the impl. (ii) Some assertions are inconclusive, meaning the problem is too complex. This may be due to resource constraints (including time limits) or other fundamental limitations of formal algorithms. (iii) Some assertions failed, meaning that "bad logic" has been created by synthesis. In this case, a counterexample (CEX) is produced to help determining the root cause.

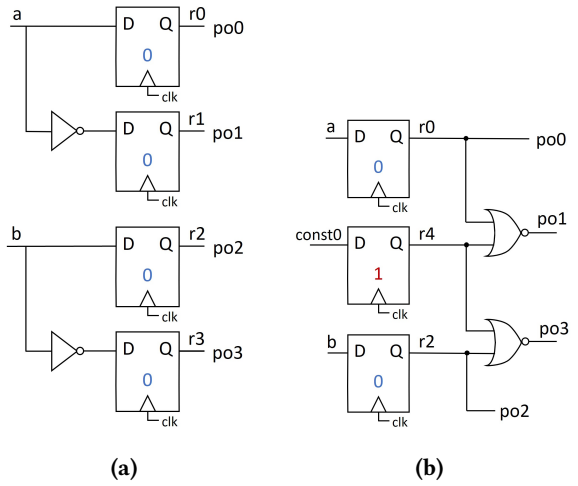

Figure 1: Example of complemented merges. The original network (a), is optimized into (b): One register and one NOR gate are added to allow merging of registers with opposite input pins but same initial state.

3 SCALABLE SEQUENTIAL SYNTHESIS

In this section, we first discuss our enhancements to state-of-the-art sequential logic synthesis. Then, we present improvements to deal with complex registers of practical industrial designs and, we conclude with the overall proposed flow with focus on scalability.

3.1 Enhanced Sequential Synthesis

We implemented our novel version of SSW and sequential synthesis optimization within an industrial framework. The core of our SSW algorithm is implemented based on [13] (e.g., uses AIG as underlying data structure and SAT to find merging opportunities). We present next further enhancements.

3.1.1 Extended Merging Opportunities. We enhanced state-of-the-art SSW to find more merging using two approaches:

(1) *Complemented merges and pin swapping:* First, we modified our sweeping algorithm to merge registers with inversions, even when the registers' initial states are not in the opposite state. We call this optimization "complemented merges". Consider as an example the circuit in Figure 1(a). The two pairs of registers have the same clock and same initial value but opposite D signal. Default SSW algorithms will thus not be able to find any merging opportunities. Our algorithm is instead enhanced to perform such merging, at cost of adding one register and one NOR gate. The idea is depicted in Figure 1(b). Each pair of equivalent registers (i.e., {r0,r1} and {r2,r3}) can be merged by adding a total of two NOR gates and one register {r4}. Costing is needed to accept only advantageous merging (resulting in area optimization). Note also that only one extra register is needed if the reset is shared among different registers, thus decreasing the effective registers overhead. Moreover, we implemented a symmetric *pin swapping* technique. Such technique swaps symmetric pins of nodes to rewrite the logic. This helps us escaping local minima when running multiple iterations of sequential synthesis.

(2) *Speculation of initial states:* For our sequential synthesis every register is required to have an associated initial state. Our assumption is that if a register has an asynchronous or synchronous preset, its initial value will be 1 and the initial value will be 0 for registers with clear signals. Historically, in industrial synthesis, unresettable registers can be set to the initial value providing the best PPA. Finding the ideal initial state beforehand (i.e., the initial state that gives the largest improvement) is a non-trivial theoretical problem. In our flow, we propose the following heuristic approach: all the non-resettable flip-flops are first assigned the same initial state 0. Note that instead initial states for resettable flip-flops are determined by their reset signals and cannot be modified. Then, a first pass of SSW is performed. From the results of this optimization, we determine all the registers which are *unchanged*. A register is *unchanged* if its TFO is not being changed by the optimization, i.e., no logic in its TFO cone is merged to any other logic, or wired as constant. A second pass of the optimization is then run (on the output network of the first pass) by flipping the initial state of these *unchanged* registers. We consider this as the final optimized network. Note that the initial state that we consider during verification should be consistent with that of synthesis.

3.1.2 Mapping and Decomposition. Our sequential synthesis algorithms are developed to work within an EDA flow, and thus need to be able to work on already mapped networks. While the work in [13] is developed on strashed AIGs, we enhanced our new SSW to work on general Boolean networks. In such networks, each internal node has an arbitrary logic function associated with a library cell (logic gate), and can support both mapped and unmapped nodes that may arise from optimizations steps. To achieve such goal, we use a shadow AIG (decomposed to run SSW) and a look-up table to store the correspondence between the original (unoptimized) mapped network and the AIG. When working on the shadow AIG, merging is allowed only on nodes corresponding to root nodes of the mapped network. The merging is performed on the mapped network using the merging equivalences found on the underlying AIG.

We further improve our mapped SSW by introducing a targeted decomposition pre-processing step. Such pre-processing performs a first analysis of the shadow AIG and finds potential merging opportunities within *all* nodes of the AIGs. That

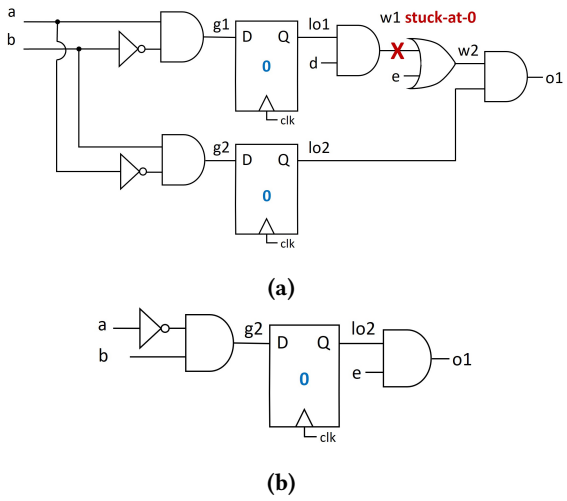

Figure 2: Example of sequential synthesis with SODCs.

is, not only the root nodes of the mapped nodes, but assuming full decomposition into AND/INV. If the merging opportunities within a decomposed node are found to be advantageous, the node is unmapped and actually decomposed before running SSW. This node will lose mapping information but will result in more merging opportunities, not achievable otherwise. Differently from decomposing the entire networks [13], the costing of the pre-process step account for decomposition of only those nodes that add advantageous merging to the optimization. Limit on the maximum fanin of nodes to decompose and sharing opportunities are also evaluated to guide the decomposition pre-step.

3.1.3 Sequential Synthesis with Observability Don't Cares. Finally, we further enhanced our flow to support additional sequential optimizations, i.e., not limited to SSW. We implemented new sequential synthesis algorithms based on the recent method presented in [12]. The idea is to perform logic synthesis optimizations under *Sequential Observability Don't Cares* (SODCs). In particular, we extended to run redundancy removal and 1-resubstitution. Consider the example in Figure 2. Sequential synthesis with SODCs is able to find that signal $w1$ is stuck-at-0, thus allowing to remove part of the logic. The method is based on induction, and is orthogonal to SSW. Finding redundancy removal and resubstitution opportunities are solved using SAT as proposed in [12].

All mentioned techniques are part of our novel sequential synthesis flow. They allow further improvements w.r.t. state-of-the-art SSW. Consider as an example a sequential version of the *i2c* benchmark. After mapping and applying SSW until saturation of results, the number of equiv. AND-2 nodes is

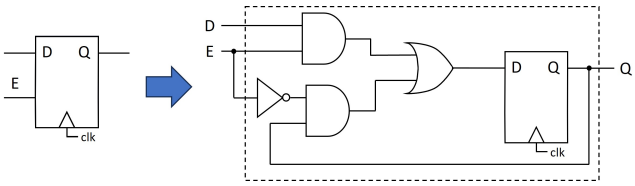

Figure 3: Example of complex flop decomposition for the enable pin.

optimized to 2319. Our techniques instead further reduce them to 1250.

3.2 Complex Registers

Besides above improvements, our novel sequential synthesis flow also needs to deal with complex registers of industrial designs.

3.2.1 Compatible Groups. The primary assumption for modelling any sequential element in the Boolean network is that they can only be in the form of a simple D-flip flop. This means that they cannot have any other control pins (synchronous or asynchronous) which can change the output at Q. Also, they need to be controlled by the same effective clock. However, industrial designs usually have registers with different control pins (e.g., asynchronous preset/clear, enable, QN, etc), different clocks and, can be clock-gated. To integrate our sequential optimizations in such scenarios, we create *compatible groups* of registers. A compatible group is defined as a group of registers that have the same (or equivalent) clock and same input signals (except D). With compatible groups, we ensure that the registers in the same group have the same clock and hence can be analyzed together as part of the same Boolean network. Input signals other than D can now be safely ignored.

3.2.2 Decomposition of Complex Registers. The number of compatible groups can reach a large number for designs with a variety of different input signals to registers. To overcome this, we perform *pull out* of the synchronous input pins by modelling them with the D pin of every register. This results in compatible groups having registers with different synchronous input signals, but the same clock and same asynchronous inputs. This solution highly reduces the total number of groups. Also, since more registers are considered in the same group, it helps to explore a larger state-space in a single optimization problem, thereby leading to more optimization opportunities. Synchronous input signals are modelled by modifying the Boolean logic at D input by inserting equivalent combinational logic (containing the synchronous signal) to mimic the same synchronous behavior. For example, consider the register with synchronous enable pin (E) in Figure 3, the effective logic at D after pull-out is:

Algorithm 1 High-level pseudocode of proposed flow

Input: Hierarchies candidate_hiers to be optimized
Output: Optimized hierarchies

```
1:  for each hier h ∈ candidate_hiers do
2:      all_regs ← optimizable registers ∈ h
3:      reg_groups ← classify all_regs into compatible groups
4:      Generate guides used as 'reference' for verification
5:      for each reg group R ∈ reg_groups do
6:          C ← combinational instances connected to R
7:          D ← registers of R
8:          create sequential Boolean network N using C and D
9:          for each reg r ∈ D do
10:             Pullout synchronous pins and add logic to N
11:             Store id of pulled out gates
12:             Add mark don't_touch on them
13:         end for
14:         sequential optimization on N
15:         survivor_regs ← regs after optimization
16:         for each reg r ∈ survivor_regs do
17:             Undo pullout logic
18:             Rewire to original signals
19:         end for
20:         Generate further guides for verification
21:         Map and commit changes to the hierarchy
22:     end for
23: end for
```

$(E\&D)|(\bar{E}\&Q)$. This works similarly for other synchronous signals.

3.3 Novel Sequential Synthesis Flow

In this section we summarize the overall industrial flow for sequential synthesis. In particular, we give details on how to make it scalable to be applied on large designs. The pseudocode of the approach is depicted in Algorithm 1. We present here an approach that works on one hierarchy at the time; extension to multiple hierarchies is discussed below. For each hierarchy, the registers are first classified into different compatible groups (see Section 3.2.1). Such classification is done considering the decomposition described in Section 3.2.2. The sequential Boolean network is built for each group of registers. To build the sequential network for each group, we keep the problem size (and thus runtime) under constraint. This is achieved on two fronts: (i) we only add the combinational logic that is relevant to the optimization problem (line 6). In other words, we collect enough logic around the registers to give the sequential algorithm all the context it needs, without including any useless information, that would increase the SAT problem size (and thus its runtime). With a larger SAT problem the algorithm would be slower to solve and may give up in proving some optimization. (ii) We partition the network by identifying groups of registers that do not share any logic, and that can be optimized separately to further reduce the problem size. The algorithm proceeds then with performing the actual logic pull-out (Section 3.2.2). Lines [10-12] make sure that the pullout nodes are considered in the optimization problem, but marked as *do-not-touch* to prevent their optimization. The *do-not-touch* is an attribute on the nodes of the Boolean network: the core optimization engine is enhanced to recognize such attributes and is bound to preserve the corresponding nodes. The sequential optimization in line 14 runs the mentioned optimization presented in Section 3.1 (e.g., enhanced SSW, sequential synthesis with SODCs, etc.). Regarding technology mapping, we tailored our methods to work incrementally when dealing with mapped networks. Similarly to [20], our methods can be tuned to preserve all the mapped gates that are not modified by the optimization. In this way, we perform technology mapping only on the subset of nodes that are unmapped (i.e., those modified by the sequential optimization); this allows us to save additional runtime. The changes are then committed to the hierarchy and the next group is considered. Note that line 4 and 20 are used to provide information to the verification (e.g., on the initial states of registers). Synthesis-aware verification is essential to contain runtime, as will be discussed in details in Section 4.

3.3.1 Parallelization. Even after reducing the number of compatible groups and partitioning the logic, runtime can still be an issue, especially when working on designs with multiple (large) hierarchies. To address scalability, we present here a parallelization scheme based on groups. Note that parallelization based on hierarchies is not a doable option because multiple compatible groups in a hierarchy need to commit changes to the same design, which cannot be done in parallel. We address this with the following approach: Let us consider hierarchies $H_1, H_2, \ldots, H_n$. Each hierarchy can have multiple compatible groups $H_1G_1, H_1G_2, \ldots\ H_nG_1, \ldots, H_nG_m$. Our new approach processes the first group of each hierarchy in parallel (i.e., $H_1G_1, H_2G_1, \ldots, H_nG_1$), commit the changes to the respective hierarchies, then proceed to group 2 of each hierarchy, and so on. Since at each parallel step, every group is from a different hierarchy, it is guaranteed that there is no sharing of logic among the groups. Commit of every group is done sequentially. This novel parallelization further reduced the runtime and improved scalability of our flow. For example, for "Design10" from our industrial designs set of experiments, it enabled more than 2x speedup.

3.3.2 Hierarchical Visibility Flow. In the flow presented in Alg. 1, optimization is applied on one hierarchy at the time. This means that, while analyzing a hierarchy, we consider any connected signal coming from other hierarchies (parent

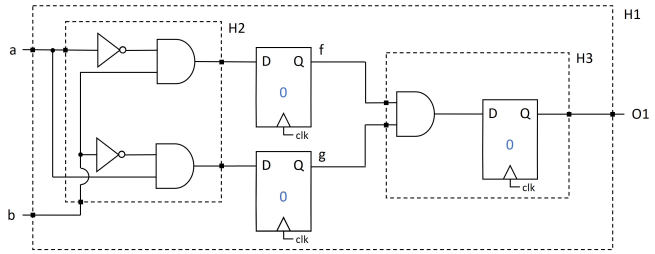

Figure 4: Hierarchical visibility: motivation and example.

or children) as PIs/POs of the Boolean network. In this section, we describe an enhancement that considers multiple hierarchies to improve logic visibility. For example, consider the design in Figure 4. Hierarchies $H1$, $H2$ and $H3$ share signals with each other. If the optimization considers these hierarchies individually, it won't be able to find any optimization. If they are instead considered as one Boolean network, SSW will be able to detect that output of $O1$ is stuck-at-0.

Our hierarchical visibility method works similarly to Alg. 1, but is further modified as follows:

- Pre-process the design to group hierarchies into different groups. Each group should be disjoint, meaning that a particular hierarchy should not be part of more than one group. This is done to not increase the problem size for verification and allow parallelization.
- Instead of working on one hierarchy, perform "virtual ungrouping" of hierarchies in the same group and include all their instances in the optimization problem.
- The sequential optimization is thus applied on a Boolean network that contains nodes coming from different hierarchies. It is important to consider that merging (or any other optimization) should be done so that nodes belonging to different hierarchies must not get merged. Our algorithms are thus modified to store such information to avoid cross-hierarchies optimization.
- Once a group is processed, we don't include any of its member hierarchies for any other optimization problem. The information related to the group of hierarchies is passed for verification.

Note that different heuristics can be involved to create the hierarchy groups. In our implementation, we give priority to decrease the number of compatible groups. Hierarchies are also grouped based on their size to avoid large size Boolean network and contain the runtime.

4 AUTOMATED SCALABLE VERIFICATION

Equivalence checking of the implemented circuits using formal verification tools is essential in today's commercial EDA flows. We describe here the first industrial automated flow for sequential synthesis verification. This is essential to verify our novel sequential synthesis flow. Here, we use SSW as running example of sequential optimization; a similar approach is used with the other sequential techniques.

4.1 Proposed Approach

Our novel flow for verification aims at providing a scalable solution. This is achieved by:

(1) using a hybrid approach that uses both CEC and SEC techniques. The key idea is to use CEC tools for verifying non-sequential transformations and to isolate the sequential optimizations during synthesis for the SEC tool. This novel approach uses the best of both worlds, can formally verify both combinational and sequential optimizations, and can scale to huge design sizes.
(2) allowing a synthesis-aware verification. The idea is to drive the verification using guides (i.e., hints describing details of various optimizations) obtained from synthesis through an automated file for verification. Verification is required to independently confirm that the guide or hint is correct before accepting. However, when verified, the guide is incorporated and becomes part of the reference.

4.1.1 Sequential Guides. We propose new sequential optimization specific guides to communicate key information needed by the SEC tool for sequential synthesis verification (see Alg. 1). This information includes: (i) Pre- and post-netlists for the region in which sequential optimizations occurred; (ii) Hierarchy grouping; (iii) Register initial state specification; (iv) Register groups and number of frames for analysis; (v) Constant and merged registers.

4.2 Novel Verification Flow

Here we detail our scalable hybrid verification flow with guides. The flow works as follows:

(1) A CEC tool is processing guides for the reference during verification.
(2) When a sequential optimization guide is encountered, the CEC tool extracts netlists for the modules being processed both before and after the SSW transformation occurs. The CEC tools verifies the pre-SSW netlist and register initial states against the reference.
(3) If successful, the CEC tool invokes the SEC tool with both pre- and post-netlists and initial states to verify the proposed sequential changes.
(4) If SEC is successful, the CEC tool performs additional nonsequential checks on the transformation, e.g., verifying that *Unified Power Format* (UPF, [3]) power intent remains correct.

Table 1: Average gain post P&R on 10 industrial designs. Negative numbers represent improvement w.r.t. the baseline.

Flow	Area	WNS	TNS	Tot. Power	Runtime
Design 1	-0.8 %	-0.9 %	-0. 2%	-0.6 %	0.1 %
Design 2	-1.8 %	-0.7 %	0.1 %	-0.4 %	0.2 %
Design 3	-2.1 %	-2.7 %	-0.7 %	-5.4 %	0.4 %
Design 4	-3.3 %	0.4 %	-0.2 %	-4.4 %	0.5 %
Design 5	-1.4 %	0.4 %	-0.7 %	-1.1 %	0.5 %
Design 6	-1.9 %	-2.1%	7.8 %	-1.6 %	1.4 %
Design 7	-0.9 %	-0.8 %	0.2 %	-0.4 %	1.4 %
Design 8	-3.6 %	1.2 %	-0.1 %	-2.1 %	1.7 %
Design 9	-1.0 %	-0.0 %	-0.0 %	-0.2 %	2.9 %
Design 10	-3.5 %	-0.8 %	-0.9 %	-2.9 %	3.3 %
Average	**-2.0 %**	**-0.6 %**	**0.5 %**	**-1.9 %**	**1.2 %**

(5) If all steps have succeeded, the CEC tool replaces the original module in the reference with the post-SSW-netlist, and the rest of the verification continues.

Note, this flow can be performed in parallel, assuming the CEC and SEC verifications are independent from each other.

4.2.1 Enhanced Sequential Equivalence Checking. Our proposed approach addresses the scalability issue of state-of-the-art SEC tools, resulting in more conclusive outcomes. For this, our tight handshake with synthesis is essential. The new sequential guides provided by the synthesis are key to controlling the verification complexity. Each SEC verification executes the following steps:

(1) Compile the spec and impl designs into a combined *Data-Flow Graph* (DFG).
(2) For each set of register and hierarchies groups provided in the sequential guides, perform SSW with the #frames provided.
(3) Check if each constant register guide can be proven in our model. If yes, then replace with the constant.
(4) Check if each merged register can be proven in our model. If yes, then replace them with the representative equivalent register.

After these steps, we use traditional sequential equivalence checking technologies to solve the problem.

Our novel flow for verification is able to successfully verify truly sequential transformations within an industrial flow, as will be detailed in the next section.

5 EXPERIMENTAL RESULTS

Our proposed approach requires to be tested on industrial designs with multiple large hierarchies, complex registers, and to be fully integrated within an industrial EDA flow. We thus run experiments on 10 benchmarks coming from major electronic industries with mentioned characteristics. Experiments are run using 4 cores. To present our synthesis results, we compare our new flow (enhanced with novel sequential synthesis) to a baseline flow running a state-of-the-art commercial EDA flow without our optimization. Our novel flow mainly targets area (and power) optimization; The flow runs a complete synthesis flow including *Place & Route* (P&R).

The results are presented in Table 1. For each design, we compare various metrics w.r.t. the baseline flow: area, timing and, power. We also report the "Runtime" as the % of runtime increased by the sequential synthesis as compared to the runtime of the whole flow. On average, our flow achieves -2.0% improvement in area and -1.9% in total power post P&R. Such major improvement is achieved with minor negative effect on timing (measured as both *Worst Negative Slack* (WNS) and *Total Negative Slack* (TNS)). The area/power improvements are up to -3.6%/ -5.4%. On average, the runtime is contained to only +1.2% of the entire flow. For the verification results, default verification tool cannot verify our novel transformations and proposed flow. Our novel hybrid automated approach for formal equivalence checking was able to automatically verify all benchmarks to be equivalent. Meaning that all sequential transformation were fully proved to be equivalent, without incurring in any "inconclusive" results.

6 CONCLUSIONS

In this work, we introduced the first industrial flow for sequential logic synthesis and scalable sequential formal verification. From a synthesis perspective, we described enhancements to state-of-the-art methods and provided details on our scalable industrial framework. We also described a novel verification flow, which focuses on combining the high capacity of combinational equivalence checking with the proving power of sequential verification methods. We demonstrated results not attainable by any other industrial EDA flow. We showed impressive PPA results on 10 commercial benchmarks, up to -3.6%/-5.4% area/power improvement post P&R, accompanied by successful formal equivalence proofs.

REFERENCES

[1] "Synopsys Formality, [Nov. 2023]: https://www.synopsys.com/implementation-and-signoff/signoff/formality-equivalence-checking.html."
[2] "TSMC's New 3nm Chip Wafers Priced at $20,000, [Nov. 2023]: https://www.siliconexpert.com/blog/tsmc-3nm-wafer/."
[3] "UPF: Standard for design and verification of low power integrated circuits, 1801-2009 - IEEE standard association," March 2009.
[4] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, "Scalable sequential equivalence checking across arbitrary design transformations," in *Int'l Conf. on Computer Design*, 2006, pp. 259–266.

[5] P. Bjesse and K. Claessen, "SAT-based verification without state space traversal," ser. FMCAD, 2000, p. 372–389.
[6] R. K. Brayton and A. Mishchenko, "Sequential rewriting and synthesis," in *Int'l Workshop on Logic and Synthesis*, 2007.
[7] M. L. Case, V. N. Kravets, A. Mishchenko, and R. K. Brayton, "Merging nodes under sequential observability," in *Design Automation Conference*, 2008, pp. 540–545.
[8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Progress on the state explosion problem in model checking," *Informatics: 10 Years Back, 10 Years Ahead*, pp. 176–194, 2001.
[9] G. De Micheli, "Synchronous logic synthesis: algorithms for cycle-time minimization," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 10, no. 1, pp. 63–73, 1991.
[10] A. Kuehlmann and C. A. van Eijk, "Combinational and sequential equivalence checking," in *Logic synthesis and Verification*. Springer, 2002, pp. 343–372.
[11] H. H. Kwak, I.-H. Moon, J. H. Kukula, and T. R. Shiple, "Combinational equivalence checking through function transformation," in *Int'l Conf. on Computer-Aided Design*, 2002, pp. 526–533.
[12] D. S. Marakkalage, E. Testa, W. Lau Neto, A. Mishchenko, and et al., "Scalable sequential optimization under observability don't cares," in *arXiv:2311.09967, cs.LO*, 2023.
[13] A. Mishchenko, M. Case, R. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis," in *Int'l Conf. on Computer-Aided Design*, 2008.
[14] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Int'l Workshop on Logic and Synthesis*, 2006, pp. 15–22.
[15] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking," in *Int'l Conf. on Computer-Aided Design*, 2006, pp. 836–843.
[16] A. Mishchenko, S. Chatterjee, and R. K. Brayton, "DAG-aware AIG rewriting a fresh look at combinational logic synthesis," in *Design Automation Conference*, 2006, pp. 532–535.
[17] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "FRAIGs: A unifying representation for logic synthesis and verification," UC Berkeley, Tech. Rep., 2005.
[18] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman, "Exploiting suspected redundancy without proving it," in *Design Automation Conference*, 2005, pp. 463–466.
[19] C. Pixley, "A theory and implementation of sequential hardware equivalence," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 11, no. 12, pp. 1469–1478, 1992.
[20] V. Possani, L. Amaru, and P. Vuillod, "Revisiting SAT-based resubstitution for incremental mapped optimization," in *Int'l Workshop on Logic and Synthesis*, 2022.
[21] C. van Eijk, "Sequential equivalence checking based on structural similarities," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 19, no. 7, pp. 814–819, 2000.