Improving Standard-Cell Design Flow using Factored Form Optimization

Alessandro Tempia Calvino[1,2], Alan Mishchenko[3], Herman Schmit[4], Ethan Mahintorabi[4],
Giovanni De Micheli[1], Xiaoqing Xu[2]
[1]Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
[2]X, the moonshot factory, Mountain View, USA
[3]Department of EECS, University of California, Berkeley, USA
[4]Google, Mountain View, USA

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10247905

***Abstract*—Factored form is a powerful multi-level representation of a Boolean function that readily translates into an implementation of the function in CMOS technology. In particular, the number of literals in a factored form correlates strongly with the number of transistors in the CMOS implementation. This paper develops novel methods for optimizing factored forms while working on the efficient and-inverter graph (AIG) representation of combinational logic. This is in contrast to the traditional logic synthesis based on logic networks, and other AIG-based methods that minimize the AIG nodes count. Experiments show that applying these methods helps to reduce the area after technology mapping by an additional 2.8% on average, compared to a high-effort area-oriented baseline. It is expected that deploying these methods as part of an industrial standard-cell design flow will reduce design costs and power consumption. Additionally, this work enables efficient transistor-level logic synthesis of large designs with various applications in design automation.**

I. Introduction

A standard-cell design flow aims at transforming a high-level description of a hardware design into a placed and routed representation that depends on the target technology. A design flow consists of numerous steps. In this paper, we focus on technology-independent synthesis and technology mapping into standard cells. The goal of these two steps is to meet delay, area, power, and other constraints by deriving an optimized circuit representation of the design, which is then placed and routed, making it suitable for fabrication.

The delay is characterized by the time it takes for a signal to propagate from inputs to outputs. The area is correlated to the number of transistors needed to implement the cells that constitute the design.

It is known [1] that the *factored form literal count* (FFLC) correlates with the number of transistors needed to implement a Boolean function. Thus, past research efforts in technology-independent synthesis [2], [3] concentrated on synthesizing small Boolean networks with a minimal FFLC. To our knowledge, this approach is still in use in many EDA tools for standard-cell designs.

However, in the last few decades, substantial progress has been made to leverage the simplicity of an *and-inverter-graph* (AIG) representation [4] for the technology-independent synthesis of Boolean networks [5]. The known AIG optimization methods can efficiently synthesize large AIGs while minimizing the number of AIG nodes and logic levels. This, however, does not guarantee the minimization of the FFLC. Consequently, the current technology-independent synthesis based on AIGs does not work at its best for optimizing standard-cell-based designs.

In this paper, we propose several efficient AIG-based FFLC minimization methods that work without converting AIGs into logic networks, as required by traditional FFLC minimization techniques [2]. The portfolio FFLC optimization includes (1) an enhanced Boolean resubstitution [6], (2) a modified version of AIG rewriting [7] and refactoring, and (3) a dedicated FFLC minimization that performs AIG re-mapping using a versatile technology mapper [8]. In the experimental results, we show up to a 5.3% reduction in the literals count and up to a 7% improvement in the area after technology mapping compared to a high-effort area-oriented AIG optimization with no runtime increase. Additionally, we discuss possible applications of the presented methods, beyond logic optimization, for transistor-level synthesis.

II. Background

A. Logic representations

Logic representations are key for developing robust EDA tools. They enable compact data storage in memory and efficient implementation of optimization algorithms. One of the first standard representations of Boolean logic was the *Sum-Of-Products* (SOP) [2]. An SOP is a two-level representation consisting of the logic OR of *product terms*, which are logic ANDs of *literals* (variables or their complements). This representation was motivated by *programmable logic arrays* (PLAs) whose primitives are modeled directly using SOPs. Because of the simple structure of a two-level circuit, the optimization problems for SOPs are well understood, which led to the development of efficient heuristic and exact minimization methods. A powerful extension of SOPs into a multi-level representation are *factored forms* [2]. A factored form is defined recursively as follows. A literal is a factored form, and the logic OR or logic AND of two factored forms is a factored form. Informally, a factored form is an SOP whose inputs are other SOPs, etc.

Example:

$$\textit{Sum-of-products} : ab + ac + ad + cbd$$
$$\textit{Factored Form} : a(b + c + d) + cbd$$

Another well known data structure for Boolean functions is the *Binary Decision Diagram* (BDD) [9] which is a canonical representation based on *if-then-else* formulae. With the advent of large-scale integration, the design size increased and logic synthesis moved to work on multi-level logic representations.

A multi-level circuit is represented as a *Directed Acyclic Graph* (DAG) composed of internal nodes having logic functions, primary inputs, and primary outputs. Generally, multi-level circuits for both ASIC and FPGA designs tend to be much smaller, power efficient, and faster, compared to the two-level counterparts. The functionality of DAGs is expressed using a small set of primitives, making DAGs easy to manipulate. The most common multi-level representation is the *And-Inverter Graph* (AIG) [4] where nodes are two-input ANDs. Recently, other logic representations have been proposed, such as the *Majority-Inverter Graph* (MIG) [10].

B. Notations and Definitions

A *Boolean function* is a mapping from an n-dimensional Boolean space into a 1-dimensional one: $\{0,1\}^n \rightarrow \{0,1\}$.

A *Boolean network* is modeled as a directed acyclic graph (DAG) with nodes represented by Boolean functions. The sources of the graph are the *primary inputs* (PIs) of the network, the sinks are the *primary outputs* (POs). For any node n, the *fanins* of n is a set of nodes driving n, i.e. nodes that have an outgoing edge towards n. Similarly, the *fanouts* of n is a set of nodes that are driven by node n, i.e., nodes that have an incoming edge from n. If there is a path from node a to node b, then a is in the *transitive fanin* of b and b is said to be in the *transitive fanout* of a. The transitive fanin of b includes node b and the nodes in its transitive fanin, including the PIs. The *transitive fanout* of b includes b and all the nodes in its transitive fanout including the POs.

The *maximum fanout free cone* (MFFC) of a node n is a subset of the transitive fanin of n such that every path from the nodes in the MFFC to the POs passes through n. Informally, the MFFC of a node contains the node itself and all the logic exclusively used by the node. When a node is removed (or substituted) the logic in the MFFC can also be removed.

A *cut* C of a node n is a collection of nodes of the network, called *leaves*, such that every path from the PIs to node n passes through at least one leaf. Node n is called the *root* of C. The *size* of a cut is defined as the number of leaves. A cut is k-feasible if its size does not exceed k. The *volume* of the cut is the total number of nodes encountered on all the paths between the leaves and the root.

A *cover* of a Boolean network is a set of cuts such that each node in the network is contained in the volume of at least one cut and all the cuts in the set are leaves of other cuts in the set or are rooted in the POs. A cover can be extracted top-down by selecting cuts starting at the POs and recurring on the leaves.

C. Logic optimization

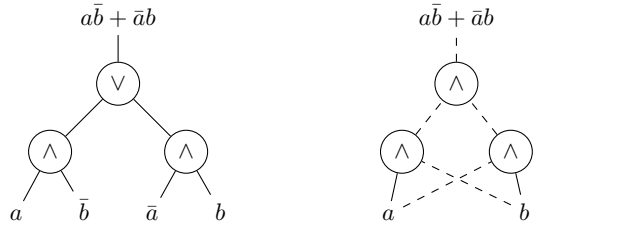
Logic optimization is a key step that enables the design of efficient circuits. Over the years, many techniques working on DAGs have been proposed. Choosing a few primitives to represent circuits as DAGs helps navigate through the logic and extract properties. State-of-the-art methods are primarily working on *And-Inverter Graphs* (AIGs). The tool ABC [11] is considered the state-of-the-art academic tool for logic optimization. ABC uses AIGs as the main logic representation. The most common and powerful optimization algorithms are *resubstitution*, *rewriting*, *refactoring*, and *balancing* [6], [7]. Most of the optimization scripts are composed of a combination of these algorithms:

- *Resubstitution*: Resubstitution [6], shortened to *resub*, (re)expresses the function of a node using other nodes, called *divisors*, that are already present in the network. The transformation is accepted if the new implementation of a node is better, according to a target metric (e.g., size), compared to the current implementation of the node in terms of its immediate fanins. This approach generalizes to *k-resubstitution*, which adds k new nodes and removes at least $k + 1$ nodes. The removed nodes are the ones present in the *maximum fanout free cone* (MFFC) [6] of the node. The functionality of the new nodes is derived from a library of primitives used for resubstitution. In the AIG implementation, added gates are 2-input ANDs with optional inverters at the inputs/outputs.
- *Rewriting*: Rewriting [7] is a fast greedy algorithm that aims at minimizing the size of a logic network by iteratively replacing sub-graphs rooted in a node with smaller pre-computed structures while preserving the functionality at the root node. Typically, pre-computed structures cover all the 4-variable functions classified into the NPN equivalence classes for compactness [12].
- *Refactoring*: Refactoring is similar to rewriting. It iterates over large logic cones rooted in a node and tries to replace the logic structure of the cone with a factored form of the root function. The replacement is accepted if there is an improvement in the selected cost metric (usually the number of gates) [6], [7].
- *Balancing*: Balancing is a fast algorithm that reconstructs logic by balancing the structure using the associative property such that the logic depth is minimized.

III. Factored Forms in AIGs

In this section, we describe the relationship between factored forms (FFs) and and-inverter graphs (AIGs). This allows us to introduce the notion of factored form literals of an AIG and propose algorithms to reduce the factored form literal count (FFLC). Unlike traditional logic synthesis [2], [3], our approach does not need to convert an AIG into a logic network. Hence, it offers better scalability on large designs.

One application of AIGs in synthesis is the representation of DAGs derived by Boolean decomposition or factoring. In particular, FFs can be represented as syntax trees where



(a) Factored form (b) AIG

Fig. 1: Translation of a factored form of a XOR2 (a) into an AIG (b). Dashed lines represent negations.

nodes are AND or OR operations, and leaves are literals (variables or their complements). Thus, FFs can be directly represented by an AIG by translating ANDs to ANDs, and ORs to ANDs using De Morgan's law $x \vee y = \overline{\bar{x} \wedge \bar{y}}$. An AIG representation of a FF is composed of primary inputs with multiple fanouts, 2-input AND gates with a single fanout (and possibly complemented inputs), and an output associated with a primary input or a 2-input AND. As an example, Figure 1 shows a representation of a XOR2 in FF and its translation into an AIG. In a FF, the number of literals is given by its number of leaves. For instance, in Figure 1a the number of literals is 4. In the AIG representation of FFs, the number of literals is equal to the fanout count of the inputs of the graph.

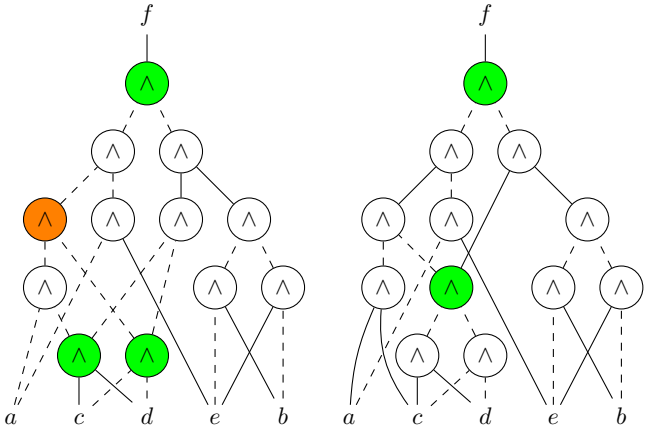
AIGs representing combinational logic are not FFs because AND nodes may have multiple fanouts. Nevertheless, FFs can be used to cover an AIG. Deriving the FF cover can be done by a technology mapper.

In this paper, we follow [13] and refer to nodes with a single fanout as *tree nodes*, and to nodes with two or more fanouts as *dag nodes*. Let us consider an arbitrary AIG. For each primary output, let us define cuts such that each node in the volume of cut is a tree node (except for the root) and the leaves are either dag nodes or primary inputs. Note that this definition is different from the notion of the MFFC of a node because the MFFC may contain dag nodes. By definition, each such cut covers a FF. Using this definition of cuts, we can cover the whole AIG by recursively creating new cuts from the leaves of the existing ones. Thus, FFs can cover an AIG and their number depends on the number of cuts in the cover. Hence, we can define the FFLC of an AIG as the sum of the literals of each FF contained in an AIG. Without employing the notion of the cover, the FFLC of an AIG can be computed using a simple formula:

$$FFLC = O + 2 \times G - M \tag{1}$$

where O is the number of primary outputs, G is the number of 2-input nodes, and M is the number of 2-input tree nodes. Alternatively, the FFLC of an AIG is the sum of the fanout counts of the primary inputs and dag nodes.

Figure 2a depicts an AIG representation of the 5-input Boolean function of a partial product of a radix-4 Booth multiplier [14]. The AIG can be covered using 3 FFs rooted in the green nodes. The one rooted in f is connected to a, e, b, and the other two green nodes. The remaining two FFs rooted in the green nodes at the bottom are connected to c and d. The total number of FF literals is 15 and it is given by Formula 1 ($1 + 2 \times 12 - 10 = 15$) or by adding the fanout counts of the primary inputs and the two dag nodes in green.



(a) Initial network: 15 lits (b) After *resub*: 14 lits

Fig. 2: Optimization of an AIG for factored form literals. Figure (a) shows the initial network. Figure (b) shows the result with reduced literal count after *resubstitution* is applied to the orange node. Nodes in green are roots of factored forms.

From the definition, it follows that FF literals in an AIG can be used as an alternative cost function to carry the optimization of combinational logic. The simple definition of literal count makes it very efficient to compute.

IV. Logic Optimization for Literal Count

In this section, we propose optimization algorithms targeting ASIC designs that can reduce the factored form literal count in AIGs. We re-formulate Boolean resubstitution, Boolean rewriting, refactoring, and re-mapping to perform the literal count minimization. We suppose that each node n in an AIG has a reference counter showing the number of its fanouts [6]. Reference counting is used for counting nodes in an MFFC and for efficient addition/removal operations for individual nodes and their MFFCs. We can also use the reference counters of the nodes to classify them into tree nodes and dag nodes and compute the FFLC.

Using Formula 1, we establish how local transformations affect the FFLC. Generally, if a 2-input dag node is removed from the graph, 2 literals are saved. If a tree node is removed from the graph, 1 literal is saved. However, every time a restructuring step increases the fanout count of a tree node, the number of tree nodes decreases leading to an FFLC increase by 1. Consequently, some transformations may decrease AIG size but increase the FFLC.

For instance, Figure 3 shows a case in which the FFLC increases due to a node substitution. In the example, the tree node p is substituted with a new node q. Removing p also removes any node in the MFFC such as t. Consequently, two tree nodes are removed decreasing G and M by two,

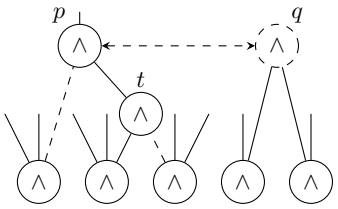


Fig. 3: Substitution of p using a new node q that increases the number of factored form literals by one.

and no dag node is transformed into a tree node. Node q is added starting from 2 tree nodes and substituted into p. Therefore, one new tree node (q) is added, increasing G and M by one. After adding a new node, two nodes increase their fanout counts and become dag nodes. Hence, even though the AIG size is reduced, the total FFLC increases by one. Other transformations may increase or keep the AIG size constant but decrease the FFLC, as shown in Figure 2. This is not exploited by state-of-the-art methods.

A. Resubstitution for factored form literals

In the standard resubstitution, the improvement is measured by the difference between the node count removed and added. The nodes that can be removed are the ones included in the MFFC. Hence, the improvement is measured as $|\text{MFFC}| - k$ where k is the number of added nodes. Similarly, in resubstitution for literals minimization, the literal saving is given by the difference between the literal count removed and added. The change in the number of literals can be computed locally using node reference counters. The reference counters track the number of nodes removed/added (MFFC) and the tree nodes created/removed during the manipulation. Finally, the change is evaluated using Formula 1. The algorithm employed is a recursive dereferencing (referencing) that decreases (increases) the reference counter of a node and recurs over the fanins if the reference count is 0 (1). In particular, recursive dereferencing (referencing) is used to measure the MFFC. Literal savings are measured by recursively dereferencing the node to substitute (root node) and counting the nodes removed and the change in the number of tree nodes. Similarly, the creation of new nodes is measured using recursive referencing.

This approach also supports filtering rules for candidates to speed up resubstitution. To simplify the filtering rule, we do not account for the change in M for the root node when we compute the savings, because any new node created will inherit the fanout of the root node leading to a zero change in M for the root. We can then use the support of the resubstitution as a filtering rule. For instance, 1-resub adds one node to the network increasing G by one adding 2 literals. For 2-resub, a minimum of 3 literals are added (one tree node), and so on.

Figure 2 depicts the FF literal optimization based on resubstitution on a Booth partial product. The orange node in Figure 2a is the target node for resubstitution. Figure 2b shows the result where the total literal count and the number of FFs needed to cover the AIG are reduced by one. Consequently, the structure of the graph is more suitable for technology mapping leading to a 17% area reduction after mapping the two implementations to a *7nm* technology[1] [15].

B. Rewriting and refactoring for factored form literals

Rewriting and refactoring are enhanced similarly to resubstitution. Standard rewriting enumerates the 4-input cuts at a root node to match and evaluate the replacements. Refactoring uses MFFCs or reconvergence-driven cuts [6]. The improvement of a replacement is measured by counting the number of nodes in the cut that can be removed, i.e. the MFFC contained in the cut, minus the number of nodes added when the structure is inserted. Rewriting and refactoring for literals minimization evaluates the reduction equivalently for literals. Savings in terms of nodes and literals are calculated using the same method since they are both based on recursive dereferencing and referencing. This enables the use of both cost functions with one as a primary cost criterion and the other one as a tie-breaker. Our implementation of rewriting for FF literals optimizes for literals primarily and uses node savings as a tie-breaker. This is motivated by compactness since an AIG with fewer nodes is easier to manipulate.

C. Mapping for factored form literals

We implemented a global re-mapping method for AIGs targeting the minimization of FF literals. The method is similar to cost-based mappers applied to graphs [8]. It consists of cost-driven mapping, followed by Boolean decomposition of each cut in the cover into an AIG, which can be seen as re-mapping. The algorithm works by computing cuts for each node using the fast cut enumeration procedure [16] and assigning to each cut a cost based on the FF representation. The FF is computed using the irredundant SOP (ISOP) extracted from the Boolean function of the cut. The SOP is then factored using algebraic or Boolean factoring [17]. Next, the technology mapper selects a cover to minimize the number of FF literals in the Boolean functions of the cuts used to cover the AIG.

V. Experiments

In this section, we evaluate the factored form optimization methods for technology-independent logic synthesis by showing the literal reductions and the results after technology mapping. We propose a resynthesis script called *compress2ff* for factored form optimization. This script has the same commands as *compress2rs* in ABC [6], but each command is modified to minimize FFLC rather than the node count. The two scripts have roughly the same runtime because the FF literal counting has negligible runtime overhead.

We set up our experiments according to one of the use cases of this novel approach which is the manipulation of optimized designs for size and literal reduction before technology mapping. When performing technology-independent synthesis, we compare the new script to *compress2rs*, which is the default script in ABC for high-effort AIG size minimization [6].

[1]The first design in (a) has been obtained after synthesis in ABC using the script *compress2rs*. We used *amap* in ABC for technology mapping.

Our baseline consists of two runs of *compress2rs* to obtain the initial compact representation of the AIG. Then, we create two flows: one running *compress2rs* two times, and the other running *compress2ff* two times. Finally, for technology mapping, we use `&nf -R 1000` in ABC for area-oriented mapping. We use ASAP7 [15] as the target technology library.

Table I shows the experimental results for the designs from the IWLS'05 benchmark suite [18]. Our methods reduce the AIG size, factored form literal count (FFLC), and area by 1.6%, 2.3%, and 2.8%, respectively, compared to the baseline. Instead, the flow that runs *compress2rs* has a limited improvement of only 0.6% in the AIG size, FFLC, and area compared to the baseline. In this experiment, we used a strong baseline to evaluate our methods. Even so, our approach enables further improvement showing the importance of FFLC optimization. For a fair comparison, our script mirrors *compress2rs* without exploring other FFLC optimization opportunities.

Generally, some designs respond to the FFLC optimization better than others. The optimization can lead to a significant improvement in the literal count, AIG size, and area after mapping for some benchmarks, such as *DMA*, *i2c*, *systemcaes*, *systemcdes*, and *tv80*. For instance, our approach reduces the area of *systemcaes* by 7%. For other benchmarks, our flow does not lead to significantly better results, compared to the standard flow, such as in *des_area*, *ethernet*, and *vga_lcd*. We noticed that our approach is more effective for control and random logic. On the other hand, arithmetic circuits are less impacted by the proposed optimization due to structural regularity. We expect better results with richer standard cell libraries, which can map large factored forms better.

VI. Applications

A. Logic Optimization

In the previous section, we presented novel optimization algorithms to reduce the FF literal count in combinational logic, aiming at improving the area after technology mapping into standard cells. The positive impact of the proposed FF-based network optimization on the CMOS implementation offers new opportunities to restructure combinational logic represented as an AIG while preparing it for technology mapping. While size is currently the main measure of the graph complexity for the area, we found that the literal count is a powerful metric to guide fine-grain optimization leading to better quality after mapping. Deploying the aforementioned methods as part of an industrial synthesis flow would improve power, performance, and area.

B. Transistor-level synthesis

The literal count in FFs is a well-known proxy for transistor count in CMOS transistor networks. Transistor count is a fundamental measure that strongly correlates with the area. Even if transistor count alone does not capture other important factors affecting area and power, such as transistor ordering, placement, and routing, it is one of the best indicators. In particular, FFs describe the serial-parallel connection of transistors. A serial connection is described using an AND gate. A parallel connection is described using an OR gate. This relation allows us to generate CMOS transistor networks from factored forms. Since the pulldown and pullup networks in CMOS are complementary, two FFs are needed, one being the dual of the other. Figure 4 depicts a mapping of a function in FF into a transistor network.

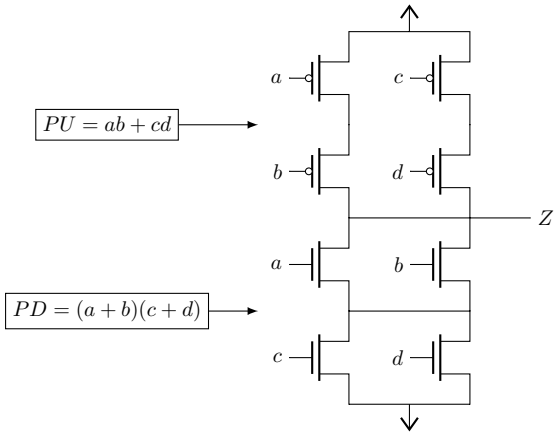


Fig. 4: CMOS network for function $Z = \overline{(a+b)(c+d)}$ and the respective pullup (PU) and pulldown (PD) networks.

Since an AIG can contain many FFs, it naturally describes the connection of transistors in a multi-stage network. Using this relation, we can extract a transistor-level network after minimizing the inverters and mapping each FF into CMOS using the natural translation of factored forms, or using other methods [19], [20]. This property opens up to transistor-level synthesis offering flexibility in functionality, not restricted by standard cell libraries, and compact layout from automated transistor-level placement and routing approaches [21], [22]. In particular, the methods discussed in this paper enable efficient transistor-level optimization and synthesis for large designs while working directly on an AIG representation thanks to the correlation between the number of literals and transistors.

As an example, for the Booth partial product function our approach optimizes and translates the function into a transistor-level network with 34 transistor after employing our factored form-based mapper[2]. Instead, if we map the function directly to the ASAP7 standard-cell library [15] while minimizing area, the resulting netlist consists of 40 transistors. This means that our approach generates a 15% reduction in transistor count for the Booth partial product implementation, which generally translates into a similar amount of area reduction after transistor placement and routing.

Our preliminary tests on transistor-level synthesis show that some useful transistor-level networks are not included in the ASAP7 standard cell library. The layout creation of those transistor-level networks and integration in an industrial flow is beyond the scope of this paper.

[2]The transistor-level network generated is not included in the ASAP7 standard cell library [15]. It is a custom implementation at the transistor level assuming a pullup/pulldown network structure, as shown in Figure 4, and transistor stacking limits.

TABLE I: Experimental results for factored form literals optimization and technology mapping

Benchmark	**Baseline: 2×*compress2rs***					**ABC: 2×*compress2rs***					**Factored form opt: 2×*compress2ff***				
	Size	FFLC	Depth	Area	Delay	Size	FFLC	Depth	Area	Delay	Size	FFLC	Depth	Area	Delay
ac97_ctrl	10203	13039	10	7012.21	100.42	10145	12971	10	6976.53	99.33	10175	12979	10	6831.41	109.19
aes_core	19493	24738	23	14012.98	245.4	19167	24511	24	13864.05	244.74	19210	23972	23	13702.23	233.87
des_area	4274	5177	32	2835.45	295.91	4263	5162	32	2825.57	295.63	4255	5093	32	2856.12	289.84
des_perf	67904	99730	28	60836.15	292.37	67141	99000	28	60431.14	287.44	67145	95820	30	59012.16	294.43
DMA	21974	26614	26	14254.06	240.5	21954	26589	26	14242.05	253.43	21358	25706	26	13533.02	247.11
DSP	37559	47734	100	26216.06	958.63	37068	47187	103	25927.76	998.47	36849	46561	98	25573.38	866.02
ethernet	55708	69226	34	35478.61	427.08	55659	69160	34	35482.25	441.46	55600	69011	34	35382.93	435.39
i2c	858	1136	19	648.46	188.32	849	1124	19	642.85	188.32	832	1087	19	604.65	188.32
mem_ctrl	7983	10410	44	5717.15	417.43	7881	10303	46	5675.89	458.48	7815	10150	45	5624.12	385.83
pci_bridge32	16092	20628	46	11262.44	505.24	16077	20616	46	11252.62	505.24	16014	20477	47	11210.79	556.1
RISC	60048	75005	100	40591.26	1055.71	59723	74631	105	40332.13	1034.93	59012	73260	101	39634.05	1126.49
sasc	546	729	9	430.9	101.77	546	729	9	430.9	101.77	542	722	9	426.24	108.77
simple_spi	732	961	19	548.09	154.85	731	960	20	548.05	155.43	728	950	19	544.11	187.98
spi	3112	3797	32	2096.5	303.82	3070	3736	32	2062.54	307.17	3055	3692	31	2035.31	294.66
ss_pcm	389	497	9	301.26	70.78	389	497	9	301.26	70.78	389	497	9	298.22	70.78
systemcaes	9582	11542	38	7002.48	426.35	9548	11444	40	7007.27	453.54	9307	11157	38	6512.72	423.49
systemcdes	2276	3252	28	1872.14	333.19	2256	3231	28	1851.12	290.58	2164	3061	27	1727.16	309.89
tv80	6856	8603	48	4752.52	417.16	6768	8507	53	4664.8	456.63	6506	8158	49	4449.26	411.61
usb_funct	12582	16145	36	8920.83	329.01	12509	16059	41	8854.99	374.44	12482	15986	41	8801.6	407.19
usb_phy	347	524	10	307.72	104.98	347	524	10	307.72	104.98	346	512	10	297.72	104.98
vga_lcd	88633	112806	35	59943.02	353.96	88628	112802	35	59933.86	346.89	88616	112747	35	59755.3	339.89
wb_conmax	37146	43041	18	20746.44	195.03	36640	42338	18	20442.97	213.27	36258	41476	20	20144.39	193.95
Geomean	7508.1	9662.5	27.3	5382.8	273.8	7460.1	9605.6	27.9	5351.4	278.8	7387.8	9435.6	27.7	5232.7	278.3
Ratio	1.000	1.000	1.000	1.000	1.000	0.994	0.994	1.023	0.994	1.018	**0.984**	**0.977**	1.013	**0.972**	1.016

VII. Conclusion

This work presents novel methods for minimizing the factored form literal count (FFLC) of AIGs representing combinational logic. Unlike traditional FFLC minimization methods, our approach does not convert an AIG into a logic network offering better scalability. Since the FFLC is a proxy for the number of transistors and the number of transistors strongly correlates with the area, our method has a positive impact when mapping to standard cells. We employ our methods in a synthesis flow showing an average reduction in the number of literals of 2.3% which translates into an area improvement of 2.8% after technology mapping over highly optimized designs. Additionally, we discuss how the proposed methods enable transistor-level optimization for large designs.

References

[1] R. L. Ashenhurst, "The decomposition of switching functions," in *Proc. Int. Symp. on the Theory of Switching*, 1957.

[2] R. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. CAD*, 1987.

[3] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A system for sequential circuit synthesis," tech. rep., EECS Department, University of California, Berkeley, 1992.

[4] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. CAD*, vol. 21, pp. 1377–1394, 2002.

[5] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification," in *IEEE/ACM ICCAD*, 2004.

[6] A. Mishchenko and R. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. IWLS*, 2006.

[7] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis," in *Proc. DAC*, 2006.

[8] A. T. Calvino, H. Riener, S. Rai, A. Kumar, and G. De Micheli, "A versatile mapping approach for technology mapping and graph optimization," in *ASP-DAC*, 2022.

[9] Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. on Computers*, no. 8, pp. 677–691, 1986.

[10] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. CAD*, vol. 35, no. 5, 2016.

[11] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification* (T. Touili, B. Cook, and P. Jackson, eds.), 2010.

[12] L. Benini and G. De Micheli, "A survey of Boolean matching techniques for library binding," *ACM Trans. Design Autom. Electr. Syst.*, July 1997.

[13] S. Chatterjee, A. Mishchenko, and R. Brayton, "Factor cuts," in *IEEE/ACM ICCAD*, 2006.

[14] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs, 2nd edition*. USA: Oxford University Press, 2010.

[15] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, 2016.

[16] P. Pan and C.-C. Lin, "A new retiming-based technology mapping algorithm for LUT-based FPGAs," in *Proc. ACM/SIGDA Sixth International Symposium on FPGA*, 1998.

[17] R. Brayton and C. McMullen, "The decomposition and factorization of boolean expression," in *Proc. ISCAS*, 1982.

[18] "IWLS 2005 benchmarks." http://iwls.org/iwls2005/benchmarks.html. Accessed: 2022-11-21.

[19] F. Mailhot and G. DeMicheli, "Automatic layout and optimization of static CMOS cells," in *Proceedings 1988 IEEE International Conference on Computer Design: VLSI*, 1988.

[20] V. N. Possani, V. Callegaro, A. I. Reis, R. P. Ribas, F. de Souza Marques, and L. S. da Rosa, "Graph-based transistor network generation method for supergate design," *IEEE Trans. VLSI Systems*, 2016.

[21] T.-C. Lee, C.-Y. Yang, and Y.-L. Li, "ITPlace: Machine learning-based delay-aware transistor placement for standard cell synthesis," in *Proc. ICCAD*, 2020.

[22] D. Lee, D. Park, C.-T. Ho, I. Kang, H. Kim, S. Gao, B. Lin, and C.-K. Cheng, "SP&R: SMT-based simultaneous place-and-route for standard cell synthesis of advanced nodes," *IEEE Trans. CAD*, 2021.