

Dear Editor-in-Chief,

Please find the attached manuscript entitled “Three-Input Gates for Logic Synthesis” to be considered for publication as a short paper in the IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. The work has never been published before. It has been recently submitted to the International Workshop on Logic and Synthesis (IWLS 2020), which is a workshop with no proceedings and no publications.

Best Regards,
Dewmini S. Marakkalage

Three-Input Gates for Logic Synthesis

Dewmini Sudara Marakkalage, *Student Member IEEE*, Eleonora Testa, *Student Member IEEE*,
Heinz Riener, *Member IEEE*, Alan Mishchenko, *Senior Member IEEE*, Mathias Soeken, *Member IEEE*,
Giovanni De Micheli, *Fellow IEEE*

Abstract—Most logic synthesis algorithms work on graph representations of logic functions with nodes associated with arbitrary logic expressions or simple logic functions and iteratively optimize such graphs. While recent multilevel logic synthesis efforts focused primarily on graphs with 2-input nodes such as AND and OR gates, the recently proposed paradigm of Majority-Inverter Graphs instead uses the 3-input Majority gate as the node function. As this technique proved to be a success, it is natural to ask: Are there other 3-input gates better suited for logic synthesis? Motivated by this question, we investigate the relative advantages of 3-input gates as constituents of logic networks. We consider representative gates from each of the ten non-degenerate 3-input NPN classes and study how powerful they are at representing Boolean functions. Using SAT-based exact synthesis, we evaluate each 3-input gate using the minimum number of such gates (together with inverters) needed to synthesize all 4-input Boolean functions and a subset of frequent 5-input and 6-input Boolean functions. We show that the logic gate $\text{Dot}(x, y, z) := x \oplus (z \vee xy)$ outperforms the rest in terms of expressive power. We further confirm this observation by showing that Dot-Inverter Graph representations are more than 14% smaller as compared to Majority-Inverter Graph representations of EPFL benchmarks.

Index Terms—Logic synthesis, Exact synthesis, 3-input gates.

I. INTRODUCTION

GIVEN a Boolean function, what is the minimum-size circuit that computes it? This is one of the driving questions in logic synthesis, which is the process of optimizing logic representations under various criteria. Decades of research on this have considered various circuit models and produced many synthesis algorithms [1], [2], [3], [4]. In general, the problem of finding the smallest circuit for a given Boolean function is a computationally difficult task, and exact minimization is reasonably fast only for Boolean functions with a small number of inputs. Consequently, most synthesis algorithms do not find optimum representations but focus on heuristic solutions. They usually work on graph representations of logic functions where each node computes an arbitrary logic expression (e.g., YLE [5], MIS [6]) or a simple logic function, and they try to incrementally modify such graphs to minimize the size or depth.

Recent multilevel logic synthesis efforts on this front considered graphs with nodes computing 2-input ANDs and

ORs together with inverters (or NANDs and NORs), which included the well-known And-Inverter Graphs (AIGs). An AIG is a Directed Acyclic Graph (DAG) where each internal node has in-degree two and represents a 2-input AND gate, and each directed edge is either complemented or regular indicating the presence/absence of an inverter. As an example, the logic synthesis tool ABC [2] uses AIGs as the primary logic representation and implements associated re-writing techniques [7], [8] for optimizing them. We call graph representations such as AIGs *homogeneous* as each internal node in the graph computes the same logic function. Optimizing homogeneous multilevel logic representations is typically more scalable and yields better results.

Recently, Amarú et al. [9] proposed Majority-Inverter Graphs (MIGs) as a new paradigm for logic synthesis. An MIG is also a homogeneous DAG representation similar to an AIG, but the internal nodes have in-degree *three* and represent *3-input majority* gates. The authors further introduced a set of algebraic rules for manipulating MIGs, proposed new synthesis algorithms, and showed significant results for depth optimization over ASIC and FPGA designs. The success of MIGs for logic synthesis begs the question: Are there other Gate-Inverter Graphs (i.e., homogeneous DAGs where each internal node implements the same 3-input logic gate¹) that are better suited for logic synthesis?

The answer depends on several criteria, such as the expressive power of the 3-input gate, the rules to manipulate such gate networks, and practical considerations such as the suitability of the gate for physical design. This work primarily focuses on the expressive power which measures how succinctly different Gate-Inverter Graphs can represent Boolean functions in general. We consider technology-independent logic synthesis, and hence we define the size of a Gate-Inverter Graph as the number of gates in such a graph, excluding inverters. We note that the homogeneity of Gate-Inverter Graphs makes their manipulation using synthesis algorithms easier as compared to the non-homogeneous ones.

There are $2^8 = 256$ 3-input Boolean functions to be considered as potential 3-input gates. However, due to the zero cost of inverters and the ability to rewire a gate's inputs in any order, many of these 3-input gates can be considered equivalent (the transformations do not incur any additional cost). This notion of equivalence is called NPN (Negation-Permutation-Negation) equivalence [10], [11]. Formally, two Boolean functions are NPN equivalent if one can be obtained











¹Unlike the 3-input majority gate, a general 3-input gate can be asymmetric, thus in such a DAG representation, the ordering of the fan-in must be specified as well.

This research was supported in part by the Swiss National Science Foundation (200021-169084 MAJesty), by H2020-ERC-2014-ADG 669354 CyberCare, by the EPFL Open Science Fund, and by SRC contract 2867.001

D. S. Marakkalage, E. Testa, H. Riener, M. Soeken, and G. De Micheli are with the Integrated Systems Laboratory, Swiss Federal Institute of Technology Lausanne, 1015 Lausanne, Switzerland.

A. Mishchenko is with Department of EECS, UC Berkeley, Berkeley, California, USA.

Table I: 10 NPN classes for 3-input functions and their representative functions. The column *Class* is the class representative, which is the lexicographically smallest truth table (as two hexadecimal digits) for each NPN class, and *Function* represents the truth table of the candidate function selected from each class. The column *Cube* shows the minterms on the Boolean hypercube.

Class	Expression	Function	Cube	Name
#01	xyz	#80		And3
#06	$x(y \oplus z)$	#28		XorAnd
#07	$x(y \vee z)$	#a8		OrAnd
#16	$x\bar{y}\bar{z} \oplus \bar{x}y\bar{z} \oplus \bar{x}\bar{y}z$	#16		Onehot
#17	$\langle xyz \rangle$	#e8		Majority
#18	$xyz \oplus \bar{x}\bar{y}\bar{z}$	#81		Gamble
#19	$x \oplus (z \vee xy)$	#52		Dot
#1b	$x?y:z$	#d8		Mux
#1e	$x \oplus yz$	#6a		AndXor
#69	$x \oplus y \oplus z$	#96		Xor3

from the other using a combination of input negations, input permutations, and output negation. Consequently, all n -input Boolean functions can be partitioned into NPN equivalence classes. The 256 3-input functions fall into 14 different NPN classes, out of which, four classes only depend on at most two variables ($f(x, y, z) = 0$, $f(x, y, z) = x$, $f(x, y, z) = x \wedge y$, and $f(x, y, z) = x \oplus y$). Thus, we only consider the remaining ten 3-input NPN classes that depend on all three variables and select one function from each class as candidate gates for building Gate-Inverter Graphs. Table I summarizes the considered ten 3-input NPN classes.

As our main result, we present a comparison of the expressive powers of the 3-input gates mentioned in Table I. To this end, we measure the minimum size of a Gate-Inverter Graph of each type needed to compute each 4-input Boolean function using *SAT-based exact synthesis* [12], [13]. These results imply that *Dot* gates have the highest expressive power closely followed by *Onehot* gates whereas *And3* gates have the least expressive power. We further confirm these results by running SAT-based exact synthesis for some frequent 5-input and 6-input Boolean functions and using exact node re-synthesis on EPFL benchmarks [14]. We omit Xor3 gates from our analysis as Xor3-Inverter Graphs are not universal representations (see Section III-A).

II. SAT-BASED EXACT SYNTHESIS

We now introduce SAT-based exact synthesis as presented in [12], [13], [15], and show how it is applied to synthesize 3-input gate networks.

Exact synthesis is the problem of finding a logic network that exactly meets its specification or determines whether it is impossible to do so. In our case, given a 3-input gate \mathcal{T} , a non-negative integer r , and a Boolean function f , the goal is to find whether there exists a \mathcal{T} -Inverter Graph of size r that computes f . Starting with $r = 0$ and incrementing it until the synthesis algorithm finds a valid circuit, we determine the minimum number of gates to compute f .

In the following, we formalize the notion of 3-input gate networks and show how to encode the exact synthesis problem as a Boolean satisfiability (SAT) problem [13]. The exact synthesis algorithm uses a SAT solver to find a satisfying assignment to the problem or to determine its unsatisfiability. If a satisfying assignment is found, the algorithm decodes it into a valid logic network. We refer the interested reader to [15], [16] for a detailed review of exact synthesis, while the first example of SAT-based exact synthesis can be found in [17], and successive analyses and improvements have been considered in [12], [13].

A. 3-Input Gate Networks

Let P be a collection of 3-input Boolean operators $\phi: \mathbb{B}^3 \rightarrow \mathbb{B}$ where $\mathbb{B} = \{0, 1\}$ is the Boolean alphabet. We call P the set of *primitives*. (For the purpose of encoding as a SAT problem, we will describe how to construct P in Section II-B.) Let $f(x_1, \dots, x_n)$ be a Boolean function on n inputs, and for notational convenience, define $x_0 = 0$. We call a sequence $(x_{n+1}, x_{n+2}, \dots, x_{n+r})$ a 3-input gate network of size r if $x_j = \phi_j(x_{j_1}, x_{j_2}, x_{j_3})$ for all $n+1 \leq j \leq n+r$ where $\phi_j \in P$ and $0 \leq j_1, j_2, j_3 < j$. Note that such a network corresponds to a DAG where each leaf node corresponds to an input variable or constant 0, and each non-leaf node has in-degree 3 and corresponds to some Boolean operator in P . The sequence $(x_{n+1}, \dots, x_{n+r})$ defines a topological ordering of non-leaf vertices. We say a given gate network $(x_{n+1}, x_{n+2}, \dots, x_{n+r})$ computes f if $f(x_1, \dots, x_n) = x_{n+r}$.

B. Encoding as a SAT Problem

We encode our exact synthesis problem as a SAT problem using the Single Selection Variable (SSV) encoding [15], [16]. The SSV encoding uses a single variable per Boolean operator to encode the inputs of the operator. Namely, it uses binary variables $s_{\ell, i, j, k}$ which are set to 1 if x_i, x_j , and x_k are the inputs for the ℓ -th operator in the network.

In the SSV encoding, to reduce the number of variables, the variables $s_{\ell, i, j, k}$ are only defined for $i < j < k < \ell$. However, since we need repeated inputs in a gate's fan-in and different fan-in orderings for non-associative operators, we add all input-permuted (with repetitions) versions of an operator as primitives for SAT-based exact synthesis.

To allow inverters at no additional cost, for every Boolean operator $\phi(x, y, z)$, we add input negated versions (with appropriate normalization as discussed later) to P . Furthermore, to avoid explicitly considering constants as inputs, for each operator $\phi(x, y, z) \in P$, we also add its versions where subsets of inputs are replaced by constants.

Note that the different primitives in P correspond to the different fan-in configurations a 3-input gate in a Gate-Inverter Graph can have. The primitives derived from input permutations with repetitions take care of different fan-in orders and the possibility of repeated inputs while the input negated versions of the primitives cover the existence of complemented edges.

The SSV encoding also uses the symmetry-breaking assumption that all logic primitives are *normal* (i.e., the

output is zero when all inputs are zero), thus, we negate any primitive that is not normal. Note that, due to the zero-cost inverters, this normality assumption does not affect the accuracy of the SAT-based exact synthesis.

III. EVALUATION METHOD AND RESULTS

In this section, we study how the 3-input gates of Table I can be used as a basis of representing logic functions with larger support. Using exact methods, we investigate the minimum number of each 3-input gates needed to compute each 2-input (Section III-A), 3-input (Section III-B), and 4-input (Section III-C) logic functions. As the main result, we summarize our findings for 4-input logic functions in Table IV, which serves as a relative measure of the expressive power of the considered 3-input gates. In Section III-D, we present synthesis results of some frequent 5-input and 6-input functions, and in Section III-E, we present the experimental results for Dot-Inverter Graphs on EPFL benchmarks. Note that applying the same exact methods for all 5-input NPN functions would be vastly time-consuming in a conventional computing setting, and prohibitively so for functions with even larger supports.

For exact synthesis, we use the SAT-based exact synthesis library *percy*² from EPFL logic synthesis libraries [18] with the SSV encoding described in Section II as our SAT encoding method. Given a logic gate $\mathcal{T} : \mathbb{B}^3 \rightarrow \mathbb{B}$, to synthesize a Gate-Inverter Graph using base gate \mathcal{T} with *percy*, we first compute the correct set of primitives. Note that \mathcal{T} is one of the ten gates from Table I and recall that the primitives should represent all versions of \mathcal{T} obtained with input permutations (with repetition), input negations, and replacing some inputs with constants. However, recall that the SSV encoding uses the assumption that all primitives are normal. Hence, we replace any not normal primitive with its complement. Formally, let $Q_{\mathcal{T}} = \{\mathcal{T}(a, b, c) : (a, b, c) \in \{x, \bar{x}, y, \bar{y}, z, \bar{z}, 0, 1\}^3\}$. Then, we define the set of primitives $P_{\mathcal{T}}$ as follows: For each $q \in Q_{\mathcal{T}}$, if $q(0, 0, 0) = 0$ then we add q to $P_{\mathcal{T}}$, otherwise we add \bar{q} to $P_{\mathcal{T}}$. Using the set of primitives $P_{\mathcal{T}}$, we invoke *percy*'s exact synthesis algorithm using the standard synthesis engine with ABC's BSAT2 as the SAT solver, which is a modified MiniSAT solver [2], [19].

A. Synthesizing 2-Input NPN Classes

The only two non-trivial NPN classes of 2-inputs are the AND and the XOR function. Table II shows the number of gates needed to compute the 2-input AND and the 2-input XOR functions using each 3-input gate type. Concerning the 2-input XOR function, note that it takes two 3-input OrAnd gates to implement the 2-input XOR gate, while the Majority and And3 gates need three instances each. Indeed, these three are the only monotone³ ones out of the ten functions. Moreover, the Majority and And3 gates are also symmetric.

As Table II shows, all the gates except Xor3 can construct the 2-input AND gate. Note that AND and Inverter are

Table II: Number of gates to synthesize 2-input And and Xor gates.

	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3	Xor3
2-input And	1	1	1	1	1	1	1	1	1	-
2-input Xor	1	1	1	1	1	1	2	3	3	1

Table III: Number of gates to synthesize 3-input NPN classes.

	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
Dot	1	2	2	2	2	2	2	4	3
Onehot	2	1	3	3	3	3	4	4	4
Mux	2	2	1	2	2	3	2	3	3
AndXor	2	2	2	1	2	2	3	4	4
XorAnd	2	2	2	2	1	2	2	3	3
Gamble	3	3	3	3	2	1	3	4	3
OrAnd	2	2	2	2	2	2	1	2	2
Majority	2	2	2	3	3	2	2	1	4
And3	2	2	2	2	2	2	2	2	1
Xor3	2	2	2	2	2	2	4	3	6
Total	20	20	21	22	21	21	25	30	33

a complete set of Boolean operators. It means AIGs can represent any Boolean function and are thus called here a *universal* representation. The results of Table II imply that all Gate-Inverter Graphs where "Gate" is any 3-input gate from Table I except Xor3 are universal (i.e., each "Gate" forms together with inverter a complete set of Boolean operators). The Xor3-Inverter Graph is not a universal representation as the truth-table of each node in such a graph must have an even number of '1's in the output column which prevents them from representing a Boolean function that has an odd number of '1's in the output column. Hence, in the following, we restrict our focus to the remaining nine 3-input gates.

B. Synthesizing 3-Input NPN Classes

In this section, we synthesize the ten 3-input NPN classes using the selected gates. Note that these classes are the 3-input functions we considered in Table I. Table III summarizes the synthesis results. Each column represents one of the nine candidate gates for universal 3-input Gate-Inverter Graphs and shows the number of gates to compute each of the other 3-input functions. For example, the third column is for 3-input Mux gates. It means that, to implement the Dot function, 2 Muxes are needed, while, to implement the Onehot function, we need 3 Muxes, etc..

It is worth noting that Dot and Onehot gates need the smallest number of gates to compute the remaining three input classes. As noted previously, the monotone functions OrAnd, Majority, and And3 need a high number of gates to synthesize other functions. However, the property of symmetry does not appear to affect the expressiveness as demonstrated by the Onehot gate.

Also note that no 3-input function pair needs five gates, whereas implementing Xor3 needs six And3 gates. This implies that for And3 gates, implementing the Xor3 function is significantly more complex than implementing the remaining functions in Table III.

²Available at: <https://github.com/lilsil/percy>

³The representative function is monotone. For other NPN classes, no member function is monotone.

C. Synthesizing 4-Input NPN Classes

The results of the previous section hint that Dot and Onehot gates seem to outperform the rest in their expressive power. We further confirm this claim by synthesizing all 222 4-input NPN classes using each type of 3-input gates. Table IV shows, for each 3-input gate \mathcal{T} and for each gate count r , the number of NPN classes (number of functions) that need r gates of type \mathcal{T} . For example, consider the first column: There are two 4-input NPN classes (ten functions) that do not need any Dot gate (i.e., the trivial functions), three classes (252 functions) that need only one Dot gate (2-input And and Xor, and the Dot function itself), 32 NPN classes (9128 functions) that need two Dot gates, etc.

As shown in Table IV, the nine 3-input gates roughly correspond to three categories. The Dot and Onehot gates outperform the rest and use the smallest number of gates to represent 4-input functions. On the other side of the spectrum, OrAnd, Majority, and And3 gates use the most number of gates. Notice that, for the And3 gate, there is one NPN class whose synthesis needs nine gates. Unsurprisingly, it turns out to be the NPN class of the 4-input Xor function (recall that the synthesis of the 3-input Xor (Xor3) function needed six 3-input And gates). The remaining four gates fall in the middle of the two former categories.

D. Synthesizing Frequent 5-Input and 6-Input Functions

In this section, we consider the effectiveness of the considered nine 3-input gates in representing some frequent 5-input and 6-input functions. For this purpose, we computed the 50k most popular 5-input and 6-input functions from the LUT mapping of the EPFL benchmarks [14] and classified them according to NPN-classification. We obtained 387 5-input NPN classes and 1905 6-input NPN classes on which we apply our exact synthesis method. To keep the runtime under control, we set a maximum conflict limit for the SAT solver of 10M (5-input) and 1M (6-input), respectively.

For the 5-input classes, 316 (out of 387) classes are synthesized – within the conflict limit – by all nine Gate-Inverter Graphs. In Table V, we summarize the results for these 316 NPN classes which confirm the trend observed in Table IV: Dot uses a fewer number of gates, while And3 and Majority have the highest number of gates. It is also worth mentioning that And3 and Majority have the highest number of non-synthesized classes (conflict limit exceeded) equal to 59 and 30 classes (out of a total of 387), respectively.

Regarding the 6-input classes, only 213 classes (out of 1905) were synthesized by all 9 Gate-Inverter Graphs within the conflict limit of 1M. We thus present the complete set of results on all 1905 classes, showing also the number of classes that could not be synthesized within the conflict limit. The results are shown in Table VI, and they confirm our previous observations as well. The Dot gate synthesizes the highest number of classes (1488) within the conflict limit, and it uses at most 5 gates on each synthesized class. On the other side, And3 and Majority could synthesize only 779 and 685 classes within the conflict limit, respectively.

E. Benchmark Results

Motivated by the high expressive power of Dot gates, in this section, we evaluate Dot-Inverter Graphs (DIGs) on EPFL benchmarks suite [14]. The experimental setup is to alternatively perform 4-LUT mapping and exact node re-synthesis for several iterations. To elaborate, we first map each benchmark into a 4-LUT network using ABC command `'if -a -K 4'`, and then use the node re-synthesis implementation of EPFL synthesis library *mockturtle*⁴ to rewrite each 4-LUT with the smallest DIG using SAT-based exact synthesis results generated using *percy*. To speed up the process, we use a database of pre-computed DIGs for all 4-input NPN classes. The same process of mapping into a 4-LUT network with ABC followed by node re-synthesis is repeated until the size of the resulting Gate-Inverter Graph size does not reduce any further. For comparison, we perform the same experiment where node re-synthesis is performed with exact MIGs and exact 2-input AIGs, and our results, presented in Table VII, indicate an overall size reduction of more than 14% for DIGs as compared to MIGs.

IV. DISCUSSION

In this paper, we studied whether there exist better logic primitives for homogeneous logic representations. In particular, we considered different 3-input logic gates and analyzed their expressive power using SAT-based exact synthesis. We show that the 3-input logic gate Dot is the most powerful in terms of expressibility and it uses a significantly fewer number of gates to represent 4-input functions as compared to And3 or Majority gates. We further supported this observation with results for frequent 5-input and 6-input NPN classes.

Our results show that monotone gates (And3, Majority, and OrAnd) have less expressive power as compared to non-monotone ones even in the presence of zero-cost inverters. This is intuitive as we certainly need a combination of monotone gates to represent non-monotone functions. However, somewhat counter-intuitively, the symmetric property of the gates seems not to affect their expressibility as suggested by the results for Onehot gates when synthesizing 4-input functions. In general, if a function is highly asymmetric in its variables, this means that permuting the inputs makes it compute different logic functions. In particular, a single Dot gate can compute six different logic functions by just permuting its inputs. Thus, intuitively, a few of them are sufficient to compute a large number of logic functions. On the other hand, a symmetric function such as Onehot stays the same when inputs are permuted, hence one would expect its expressive power to be relatively low.

The high expressive power of Dot gates yields concise logic representations as evident from our experiments. Thus DIGs are attractive candidates for technology-independent intermediary logic representations as optimization algorithms can run faster with less memory on smaller graphs. Moreover, this also inspires emerging technologies to consider physical implementations of Dot gates.

⁴Available at: <https://github.com/lsils/mockturtle>

Table IV: Classification of the 222 4-input classes (65536 4-input functions). This shows, for each 3-input gate \mathcal{T} and for each gate count r , the number of 4-input NPN classes (number of 4-input functions) whose synthesis needs r gates of type \mathcal{T} . The last row shows the total number of gates needed to separately synthesize all 4-input NPN classes (all 4-input functions).

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
0	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)
1	3 (252)	3 (124)	3 (156)	3 (156)	3 (156)	3 (92)	2 (80)	2 (80)	2 (112)
2	32 (9128)	18 (2856)	17 (3224)	15 (2776)	18 (2336)	13 (1272)	14 (3020)	5 (640)	4 (544)
3	158 (51770)	151 (50490)	92 (31554)	86 (27202)	83 (26786)	63 (14242)	46 (14528)	18 (3300)	13 (2508)
4	27 (4376)	48 (12056)	100 (29936)	110 (34864)	109 (35032)	115 (41856)	89 (30854)	42 (10352)	46 (14944)
5	-	-	8 (656)	6 (528)	7 (1216)	26 (8064)	55 (15064)	117 (40064)	68 (24024)
6	-	-	-	-	-	-	14 (1810)	35 (11058)	55 (17376)
7	-	-	-	-	-	-	-	1 (32)	26 (5680)
8	-	-	-	-	-	-	-	-	5 (336)
9	-	-	-	-	-	-	-	-	1 (2)
Total	649 (191322)	684 (205530)	753 (224290)	761 (229410)	759 (231394)	808 (253106)	883 (259500)	1036 (319560)	1134 (335342)

Table V: Results on 316 frequent 5-input NPN classes.

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
2	-	-	2	1	3	-	4	-	2
3	97	7	49	58	68	18	72	11	12
4	208	249	204	189	180	154	136	60	73
5	11	60	56	68	65	143	90	131	94
6	-	-	5	-	-	1	14	114	83
7	-	-	-	-	-	-	-	-	52
Total	1178	1317	1277	1272	1255	1391	1302	1612	1664

Table VI: Results for 1905 frequent 6-input NPN classes. The row CLE reports the number of conflict limit exceeded cases – thus the number of non-synthesized classes.

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
3	6	-	15	31	51	1	65	7	12
4	551	33	262	400	424	75	377	55	96
5	931	923	965	771	815	796	490	296	323
6	-	-	32	-	-	31	162	421	254
CLE	417	949	631	703	615	1002	811	1126	1220

Table VII: Experimental results on EPFL benchmarks.

Benchmark Name	size	depth	Optimized size	AIG depth	Optimized size	MIG depth	Optimized size	DIG depth
adder	1020	255	1020	255	511	130	511	255
bar	3336	12	3336	12	3060	13	2246	11
div	57247	4372	43566	4417	57247	4372	43421	4401
hyp	214335	24801	214335	24801	163627	9434	147392	17159
log2	32060	444	32060	444	25866	349	23144	318
max	2865	287	2865	287	2396	233	2230	229
multiplier	27062	274	27062	274	20566	318	17273	261
sin	5416	225	5416	225	4601	187	3995	149
sqrt	24618	5058	19127	6077	22830	4379	17027	6045
square	18484	250	18484	250	14966	245	12743	251
arbitr	11839	87	11839	87	9023	63	8515	63
cavlc	693	16	693	16	693	16	644	19
ctrl	174	10	131	8	128	7	117	8
dec	304	3	304	3	304	3	304	3
i2c	1342	20	1324	16	1325	19	1105	18
int2float	260	16	248	18	255	17	202	19
mem_ctrl	46836	114	46836	114	45365	145	37114	130
priority	978	250	586	131	978	250	664	125
router	257	54	257	54	235	58	202	32
voter	13758	70	12309	74	8589	74	7385	64
Total	462884	36618	441798	37563	382565	20312	326234	29560

To the best of our knowledge, our work is the first comprehensive study of the efficiency of 3-input primitives in representing logic networks, and we believe it will trigger exciting further research in logic representations, optimizations, and implementation of logic networks in emerging technologies. One interesting research direction is to understand the manipulation capabilities of different homogeneous 3-input logic networks as it is another important factor besides the expressive power that affects the

utility of logic representations in logic optimization.

ACKNOWLEDGMENTS

We acknowledge Niklas Eén for having suggested the interesting problem addressed in this work and for the support from Synopsys Inc.

REFERENCES

- [1] E. Testa, M. Soeken, L. Amarú, and G. De Micheli, “Logic synthesis for established and emerging computing,” *Proceedings of the IEEE*, pp. 1–20, 2018.
- [2] R. K. Brayton and A. Mishchenko, “ABC: an academic industrial-strength verification tool,” in *Computer Aided Verification*, 2010, pp. 24–40.
- [3] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarú, G. De Micheli, and M. Soeken, “Scalable generic logic synthesis: One approach to rule them all,” in *Design Automation Conference*, 2019, pp. 1–6.
- [4] S. Muroga, “Logic design and switching theory,” 1979.
- [5] R. K. Brayton, “The yorktown silicon compiler,” *Silicon Compilation*, pp. 204–311, 1988.
- [6] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, “MIS: A multiple-level logic optimization system,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.
- [7] P. Bjesse and A. Boralv, “DAG-aware circuit compression for formal verification,” in *Int’l Conf. on Computer-Aided Design*, 2004, pp. 42–49.
- [8] 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.
- [9] L. Amarú, P. Gaillardon, and G. De Micheli, “Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization,” in *Design Automation Conference*, 2014, pp. 1–6.
- [10] E. Goto and H. Takahashi, “Some theorems useful in threshold logic for enumerating Boolean functions,” in *IFIP Congress*, 1962, pp. 747–752.
- [11] S. L. Hurst, D. M. Miller, and J. C. Muzio, “Spectral techniques in digital logic,” 1985.
- [12] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavlsev, “Finding efficient circuits using SAT-solvers,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.
- [13] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.
- [14] L. Amarú, P.-E. Gaillardon, and G. De Micheli, “The epfl combinational benchmark suite,” in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, no. CONF 2015.
- [15] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. Amarú, R. K. Brayton, and G. De Micheli, “Practical exact synthesis,” in *Design, Automation and Test in Europe*, 2018, pp. 309–314.
- [16] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, “SAT-based exact synthesis: Encodings, topology families, and parallelism,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, pp. 1–1, 2019.
- [17] N. Eén, “Practical SAT - a tutorial on applied satisfiability solving,” 2007, slides of invited talk at FMCAD.
- [18] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, “The EPFL logic synthesis libraries,” Nov. 2019, arXiv:1805.05121v2.
- [19] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.