INVITED PAPER

Logic Synthesis for Established and Emerging Computing

This paper provides a state-of-the-art view on the status of logic design flows in conventional silicon CMOS as well as using several of the emerging technologies.

By Eleonora Testa, *Student Member IEEE*, Mathias Soeken, *Member IEEE*, Luca Gaetano Amarù, *Member IEEE*, and Giovanni De Micheli, *Fellow IEEE*

ABSTRACT | Logic synthesis is an enabling technology to realize integrated computing systems, and it entails solving computationally intractable problems through a plurality of heuristic techniques. A recent push toward further formalization of synthesis problems has shown to be very useful toward both attempting to solve some logic problems exactly—which is computationally possible for instances of limited size today—as well as creating new and more powerful heuristics based on problem decomposition. Moreover, technological advances including nanodevices, optical computing, and quantum and quantum cellular computing require new and specific synthesis flows to assess feasibility and scalability. This review highlights recent progress in logic synthesis and optimization, describing models, data structures, and algorithms, with specific emphasis on both design quality and emerging technologies. Example applications and results of novel techniques to established and emerging technologies are reported.

KEYWORDS | Algorithms; discrete optimization; emerging technologies; logic networks; logic synthesis; quantum computing; satisfiability

I. INTRODUCTION

The fast evolution of computing and communication technologies has been enabled by a wide body of knowledge on how to represent and manipulate digital functions as well as on how to optimize their realization. Logic synthesis is a key component of digital design, as logic functions are often extracted from high-level models, such as programming (e.g., C, C++) or specialized hardware languages (e.g., VHDL), and their optimization is crucial to achieve effective implementations. Indeed, it was clear from the early days that logic design is such a daunting task, because of the problem size and plurality of choices, that design automation is essential. Logic synthesis has progressed through the years by combining theoretical results and engineering practices. The notion of optimal design has been obscured by the superposition of various concerns, mainly related to the definition of the cost function (e.g., circuit complexity, delay, and power consumption) in terms of the properties of the physical medium used to fabricate the circuit and its interconnections. With the lack of a precise objective function and due to the large design space, engineers have relied on complex tool flows that apply heuristics. These approaches have been shown to be successful in designing large chips and represent the state of the art [1]–[3]. Today logic synthesis is an essential instrument to push the limits of performance (upwards) and power consumption (downwards). These objectives are often at odds and compounded by other goals such as enhancing testability, reliability, and reducing area. Thus, competitive synthesis tools are necessary for the design of leading-edge chips.

Today, synthesis is a critical area of research for two main reasons. 1) The computational fabric, in terms of devices of various nature, is evolving. Postsilicon technologies have been shown to be viable and may provide us with better substrates for computation. By the same token, new architectures (e.g., neuromorphic, optical, and quantum

Manuscript received March 21, 2018; revised August 9, 2018; accepted September 1, 2018. Date of publication October 1, 2018; date of current version December 21, 2018. This work was supported by the Swiss National Science Foundation (200021-169084 MAJesty) and by H2020-ERC-2014-ADG 669354 CyberCare. *(Corresponding author: Eleonora Testa.)*

E. Testa, **M. Soeken** and **G. De Micheli** are with the École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland (e-mail: eleonora.testa@epfl.ch).

L. G. Amarù is with Synopsys Inc., Mountain View, CA 94043 USA.

Digital Object Identifier 10.1109/JPROC.2018.2869760

This work is licensed under a Creative Commons Attribution 3.0 License. For more information, see http://creativecommons.org/licenses/by/3.0/

computing) can take advantage of this change in terrain to provide us with solutions to our unstoppable appetite for computing. 2) The current computing and storage means make it possible to solve exactly problems that were only approximated before, providing good working solutions but whose quality may be far from optimum. Moreover, new specialized circuits can be used as engines for computational solvers, thus enabling a virtuous cycle to achieve increasingly higher quality hardware [4]. The frontier of computing architectures and digital systems has been moving fast over the last two decades. The availability of a large number of devices on a single chip has enabled multicore design in the last two decades. New disruptive architectures are exploiting circuit arrangements for supporting (deep) learning. Recent digital systems have shown the capability to leverage (to some limited extent as of this writing) devices that perform quantum computation by exploiting superposition and entanglement [5]. As a result, the circuit primitives for logic design have increased and changed over the years. Complementary metal–oxide–semiconductor (CMOS) technology has favored circuits based on NORs, NANDs, and their extensions, which can be abstracted as negative unate single-output functions. Today most product-level circuits leverage these primitives and their extensions (e.g., AOI gates) collected into libraries. With the downscaling of technologies, the number of stacked transistors decreases thus reducing also the fanin (or support size) of these functions. Field-programmable gate arrays (FPGAs) are built out of programmable lookup tables that realize small-scale logic functions. New emerging technologies, such as some optical technologies and quantum-dot cellular automata (QCA, [6]), leverage majority (MAJ), and inverter (INV) gates as primitives. Neuromorphic architectures exploit threshold gates, which can be seen as majority gates with weighted inputs. Combinational operations in quantum computing (QC) [7] can be abstracted in terms of libraries of components, such as the Toffoli gate [8], [9] that implements a generalized form of exclusive OR (EXOR) operation. This abstraction is further refined in terms of quantum gates targeting devices available in specific QC technologies.

The objective of synthesis is to map data flows (i.e., sets of logic and arithmetic operations in a partial order) into optimal interconnections of circuits. We consider these circuits as atomic primitives, because we want to use an abstraction of the underlying computation valid in a large class of technologies. Such circuits can be exemplified by library cells in CMOS, optical computational devices, or quantum circuits that realize one or more computation steps in a quantum medium. Thus, a circuit is both—according to the context—a physical device and an abstraction of a computation in terms of a stimulus/response pair that can be represented by a logic function.

Since we address a plurality of technologies, we consider logic synthesis as a task performed independently of physical design. Whereas we are cognizant of the importance of coupling physical and logic design on nanotechnologies, we believe that we need to separate the issues to formulate clear scientific problems of broad applicability. It can also be argued, and it was demonstrated before [10], that new robust logic synthesis algorithms can lead eventually to better circuits as evaluated after physical design.

In view of the progress and opportunities of technology, logic synthesis has to be revisited while considering the plurality of primitives that can be of interest, and as a result the corresponding objective functions and optimization problems. The objective of this paper is to present the state of the art in a succinct manner (as other tutorials and books are available [11]–[15]) to provide the basis to describe data structures and algorithms for emerging technologies and architectures. Whereas we presented in [11] an array of novel computing technologies and one computational approach for their synthesis, here we review critically several logic synthesis methods and we show their applicability to established and emerging technological platforms. Namely, we want to capture here the essential features of logic synthesis at the onset of architectural and technological changes and thus we focus on combinational synthesis and refer the interested reader to [12]–[14] for the sequential counterpart.

This paper is organized as follows. After a brief historical perspective, we first consider data structures for logic synthesis that have been used in synthesis, such as two-level and multilevel structures (e.g., including tables, expressions, and diagrams). We consider logic synthesis algorithms of various kinds. First, we describe algorithms based on algebraic properties of the representation. Next we consider Boolean methods that exploit specific Boolean properties. As a third kind we consider exact approaches for logic synthesis. We will also comment on decomposition methods that can be coupled to exact methods to make the approach viable. Last, we will address specific application technologies and their relations to logic synthesis. Namely, we consider CMOS technologies, majority-based nanoemerging technologies, and technologies that exploit quantum effects, such as information quantization (e.g., QCA), superposition, and entanglement (e.g., quantum computing).

II. A BRIEF HISTORICAL PERSPECTIVE

Broadly speaking, the overall problem of logic synthesis is the one of finding "the best implementation" of a logic function, where that term "best" is used because it is imprecise as it may depend on goals and computational methods and it may not be unique. Thus, synthesis encompasses also logic optimization and the two terms are interchangeable. We focus here on the combinational logic synthesis approach, where "best" is understood in terms of complexity, delay, and/or power consumption. The first approaches to logic synthesis addressed sum-of-product (SOP) representations, and attempted to reduce the cardinality of logic covers (i.e., the number of product terms also called implicants). Structured representations of SOP representations,

such as PLAs, have rectangular shapes with rows associated with product terms. Hence, reducing the number of product terms reduces the area. The first logic synthesis algorithm, the Quine–McCluskey algorithm [16], solves the minimization of logic covers exactly. Subsequent implementations of this algorithm, enhanced by appropriate data structures, enabled designers to solve most benchmarks of relevant size [17]. Several approaches to heuristic minimization of two-level forms [12], preferable to the exact approach for computing time reasons, culminated with program ESPRESSO, that provides irredundant covers of near-optimum size. Irredundant covers are minimal with respect to containment and have the advantage that the corresponding AND/OR realizations are fully testable for stuck-at faults. Thus, the program ESPRESSO [13], [17] had a large impact on the design automation community. Unfortunately, two-level logic implementations have two major drawbacks. First, the delay is not correlated with the stages of delay (as originally thought) but with fanin and capacitive load. Second, efficient implementations in CMOS require dynamic operation (which is complicated at high speed) or pseudo-NMOS loads (which consume excessive power) [18].

As a result, two-level logic optimization is used as a method to reduce the complexity of a logic block, which may have one or more outputs, as an intermediate step in logic optimization. In a similar vein, extensive logic synthesis research has been devoted to exclusive-sum-of-product (ESOP) minimization. This problem can be solved exactly [19], [20] or heuristically. At present, the most used program is EXORCISM [20]. Also in this context, single-output or multiple-output functions are optimized as an intermediate step of a logic synthesis flow, with no direct relation to the implementation. Nevertheless, in the domain of design of quantum computing circuits, ESOP minimization is important because ESOP forms can be mapped into a cascade of Toffoli gates providing a reversible logic solution.

Contemporary logic synthesis and its scientific and commercial successes have risen in the 1980 with the establishment of CMOS technology, semicustom design styles, and libraries of components. The problem consists of mapping logic functions into the "best" interconnection of instances of library elements, and it bears a relation to computing the complexity [21] of a Boolean function, which is computationally intractable. Hence, most approaches divide synthesis into a technology-independent phase, where the interconnection of logic blocks is minimized independently of the library, followed by a technology mapping step where the instances of library elements are chosen. In practice, such an approach tends to provide a "good starting condition" to the mapping problem. Recent approaches to synthesis have tackled the problem from a different angle. Rather than relying on various layers of heuristics to find the solution, the following question is asked: "How large can a logic block be so that an optimum realization (possibly under constraints) can be found?" Optimum may mean minimum area, which is the sum of the areas of the chosen cells [22]–[24], or minimum delay [25], [26], which is the critical path delay through the circuit that can be computed once the cells are selected and possibly verified after physical design. The optimum problem can be cast in terms of satisfiability (SAT), and a SAT solver is used to attempt its solution. It is not surprising that increasingly larger optimum circuits (for area or delay) can be computed as more powerful computer resources become available. The main issue is the practicality of such an approach for very large circuits that today can involve millions of NAND-equivalent gates. Nevertheless, the divide-and-conquer approach still applies: logic networks can be decomposed into blocks, and blocks synthesized by exact methods. Moreover, the "best" realization of functional blocks can be cached in libraries and instantiated by synthesis algorithms at runtime. It is important to note that the divide-and-conquer paradigm is also a cornerstone of heuristic logic synthesis. Indeed logic networks are interconnection of blocks, each block represented by a logic function. This hybrid structure helps containing the possible blow-up in size of the Boolean function representations, and it provides an underlying substrate for optimization algorithms. In the rest of the paper, we will outline the data structures that capture efficiently logic circuits as interconnection of blocks and support optimization methods based on heuristic and exact techniques.

III. DATA STRUCTURES

We present here various data structures that are commonly used by logic synthesis algorithms. The subsections are ordered according to the scalability of the data structures, starting from truth tables, which are suitable for functions with a small support (i.e., number of variables), to multilevel logic networks, which are the ubiquitous data structure (in various forms and shapes) to represent Boolean functions in almost all research and commercial tools. Each section also briefly mentions some implementation hints to enable efficient algorithms.

A. Truth Tables

A truth table is an explicit representation where the function values are listed for all possible input combinations. Formally, a truth table for a Boolean function $f(x_1,\ldots,x_n)$ is a bitstring $b_{2^n-1}b_{2^n-2}\ldots b_1b_0$ of 2^n bits, where $f(x_1,\ldots,x_n) = b_x$ such that $x = (x_n\ldots x_1)_2$ is the integer representation of the input assignment. Consequently, we may also consider a truth table as a number in the half-open interval $[0, 2^{2^n})$, for which the truth table representation is the binary expansion of that number.

Example 1: The truth table for a majority-of-three (majority-3) function $\langle x_1x_2x_3\rangle = (x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3)$ is $1110\,1000$. Since the binary notation can quickly become very large, it is customary to use a hexadecimal

notation, in which each block of 4 b is represented by the corresponding hexadecimal digit. For the majority-of-three function, the hexadecimal truth table is #e8. (We use the hash prefix to indicate a hexadecimal number.)

Clearly, truth tables cannot provide a scalable function representation. Nevertheless, for small functions they can be beneficial as they enable very fast implementations. For example, a truth table for a six-variable function requires $2^6 = 64$ b and therefore fits into a single unsigned integer of a 64-b computer architecture. Many operations, e.g., computing the AND of two functions can be performed using bitwise AND, which accounts for a single processor instruction. Such an approach works reasonably well in practice up to 16-variable functions, which require $2^{10} = 1024$ 64-b unsigned integers, and therefore 8 MB of memory.

A truth table is a canonical (i.e., unique) representation of a function. Consequently, for small functions, truth tables can be used for a simple equivalence check of two functions, if a truth table can be efficiently derived from them.

B. Two-Level Representations

Logic functions can be represented in disjunctive normal form, also referred to as sum-of-products

$$f = p_1 \vee p_2 \vee \cdots \vee p_k \tag{1}$$

where each

$$p_i = x_1^{q_{i,1}} \wedge x_2^{q_{i,2}} \wedge \cdots \wedge x_n^{q_{i,n}} \tag{2}$$

is a product of literals with $0 \leq q_{i,j} \leq 2^R - 1$ for $1 \leq i \leq k$ and $1 \leq j \leq n$ and where R is a radix. We have $R = 2$ for binary Boolean logic, $R = 3$ for ternary logic, etc. This represents the so-called positional cube notation [12] where usually the $q_{i,j}$ are represented in binary form. Therefore, for binary-valued logic, the negative and positive literals are $x^1 = x^{\{01\}} = \bar{x}$ and $x^2 = x^{\{10\}} = x$, respectively, $x^3 = x^{\{11\}}$ is a don't care term (i.e., both values of a variable are possible), and $x^0 = x^{\{00\}} = \emptyset$ is the empty set (i.e., no value).

Example 2: Let $f(x_1, x_2, x_3) = x_1 \,?\, x_2 : x_3$, which is also called the if–then–else operator. A disjunctive normal form is $f = x_1x_2\bar{x}_3 \vee x_1x_2x_3 \vee \bar{x}_1x_2x_3 \vee \bar{x}_1\bar{x}_2x_3$. An alternative, shorter form is $f = x_1x_2 \vee \bar{x}_1x_3$. In general, one is interested in finding a disjunctive normal form that minimizes the number of product terms k.

Many algorithms have been presented to find disjunctive normal forms with some minimality properties (see, e.g., [19]). Also, other two-level representations have been investigated. Examples are conjunctive normal forms, or product-of-sums [that interchange "$\vee$" and "$\wedge$" in (1) and (2)] or exclusive sum-of-products [which use "$\oplus$" instead of "$\vee$" in (1)]. Conjunctive normal forms play a central role in Boolean satisfiability solving (see, e.g., [27] and [28]) and can be seen as the dual representation of disjunctive normal forms [29]. Exclusive sum-of-product representations find extensive use in cryptography applications (see, e.g., [30]–[32]) and quantum computing (see, e.g., [33] and [34]). Recently, also exclusive product-of-sum representations, which are the dual of exclusive sum-of-products, have been investigated in the context of Boolean satisfiability of cryptography applications [35].

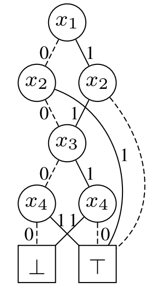


Fig. 1. *BDD for the function $(x_1 \oplus x_2) \vee (x_3 \oplus x_4)$.*

It is often feasible to represent Boolean functions in two-level representations for 20–30 variables. Conjunctive normal forms are possible for functions with many variables, if one allows additional helper variables [36]. Product terms for functions with up to 32 variables can be represented in a computer using 64-b unsigned integers: the first 32 b are used to represent which variables occur in the product term, and the second 32 b are used to represent whether the occurring literals are positive or negative.

C. Binary Decision Diagrams

Logic functions can be expressed by decision diagrams in many ways. The most common representation is the binary decision diagram (BDD) [21], [37] which is a directed acyclic graph where internal nodes are associated with the Shannon (also credited to Boole) expansion of the function, i.e., $f = x_i f_{x_i} \oplus \bar{x}_i f_{\bar{x}_i}$, where f_{x_i} and $f_{\bar{x}_i}$ are the cofactors obtained from f when the variable x_i is assigned 1 or 0, respectively. When referring to BDDs, it is usually implicitly understood that the variables are ordered and the diagram reduced (i.e., BDD refers to ROBDDs [37]). Moreover BDDs are constructed and manipulated so that redundancy is avoided, and thus they are canonical representation of logic functions.

Example 3: Fig. 1 shows the BDD for the function $(x_1 \oplus x_2) \vee (x_3 \oplus x_4)$. Solid and dashed lines represent here positive and negative cofactors, respectively.

BDDs exploit the fact that for many functions of practical interest, smaller subfunctions occur repeatedly and need to be represented only once. Combined with an efficient recursive algorithm that makes use of caching

techniques and hash tables to implement elementary operations, BDDs are a powerful data structure for Boolean function representation and manipulation. Indeed, algorithms for BDD manipulation have polynomial-time complexity (usually quadratic or cubic) in the number of nodes, and such a number grows mildly with the problem size (i.e., variables) in many—but not all—cases, e.g., multipliers.

The variable order in BDDs affects their size. Improving the variable ordering for BDDs (i.e., minimizing the BDD graph size) is NP-complete [38]. An exact algorithm [39] and many heuristics [40] have been presented that aim at finding a good ordering. It is easy to fit a single BDD node, which contains the variable index and pointers to its two children, into a single 64-b unsigned integer [21]. Thus, BDDs can represent a good scalable representation for logic functions. They can cope with larger functions as compared to truth tables. When their storage becomes excessive, functions are usually decomposed into blocks forming logic networks.

D. Multilevel Logic Networks

A multilevel logic network (LN) is interconnection of blocks, each implementing a logic function and whose representation style may vary. The interconnection is modeled by a directed acyclic graph where nodes represent primary inputs and outputs, as well as local functions. In most cases, such functions are restricted to have a single output, by similarity to CMOS logic gates. For internal nodes, the indegree and outdegree are referred to as fanin and fanout, respectively. Note that LNs can be extended to deal with sequential cyclic circuits [12], but such cases are not considered here.

We use a formal notation for Boolean logic networks that is also referred to as straightline programs or Boolean chains in the literature. Given primary inputs $x_1, \ldots, x_n$, a Boolean logic network consisting of r local functions is a sequence

$$x_i = f_i(x_{i_1}, x_{i_2}, \ldots, x_{i_{\mathrm{ar}(f_i)}}) \qquad \text{for } n < i \leq n + r \tag{3}$$

where f_i is a gate function with $\mathrm{ar}(f_i)$ inputs and $0 \leq i_j < i$ for $1 \leq j \leq \mathrm{ar}(f_i)$ are indexes to primary inputs or previous gates in the sequence. For convenience, we define $x_0 = 0$. Also, we define a sequence of primary outputs $y_1 = x_{o_1}, \ldots, y_m = x_{o_m}$.

Example 4: A full adder with inputs x_1, x_2, x_3 can be realized by the network

$$x_4 = x_1 \oplus x_2 \oplus x_3, \qquad x_5 = \langle x_1 x_2 x_3 \rangle$$

with outputs $y_1 = x_4$ for the sum and $y_2 = x_5$ for the carry. The network uses the parity function f_4 and the majority function f_5.

Logic networks can be specialized by placing restrictions on the internal nodes. A homogeneous LN is one where the fanin of each internal node is fixed. Restrictions can be applied to local functions as well (e.g., networks consisting of NANDs and/or NORs). For example, AND-inverter graphs (AIGs), [41], [42] employ AND and inverters (or equivalently apply AND functions to positive/negative literals). Majority-inverter graphs (MIGs), [43] use majority and inverter gates and XOR-majority graphs (XMG) [44] use majority and EXOR gates. For FPGA design, bounded input lookup tables k-LUT networks are used, where $\mathrm{ar}(f) \leq k$.

Example 5: Fig. 2 shows logic networks for a 4-b full adder, which computes $(x_4x_3x_2x_1)_2 + (x_8x_7x_6x_5)_2 = (y_5y_4y_3y_2y_1)_2$. Fig. 2(a)–(c) shows an AIG, an MIG, and an XMG, respectively. Inverted inputs are drawn using dashed edges. Fig. 2(d) shows a 4-LUT network. The gate functions are $f_9 = {}^{\#}6$, $f_{10} = {}^{\#}936c$, $f_{11} = {}^{\#}137f$, $f_{12} = {}^{\#}69$, $f_{13} = {}^{\#}2b$, $f_{14} = {}^{\#}69$, and $f_{15} = {}^{\#}d4$.

Combinational logic functions can be represented by many different logic networks. A central task in logic synthesis is to optimize some figure of merit that relates to area, performance, and/or power consumption of the final implementation. Commonly used cost functions are the size r of the logic network, measured in the number of nodes, the depth d of the logic network, which is the longest path from any primary input to any primary

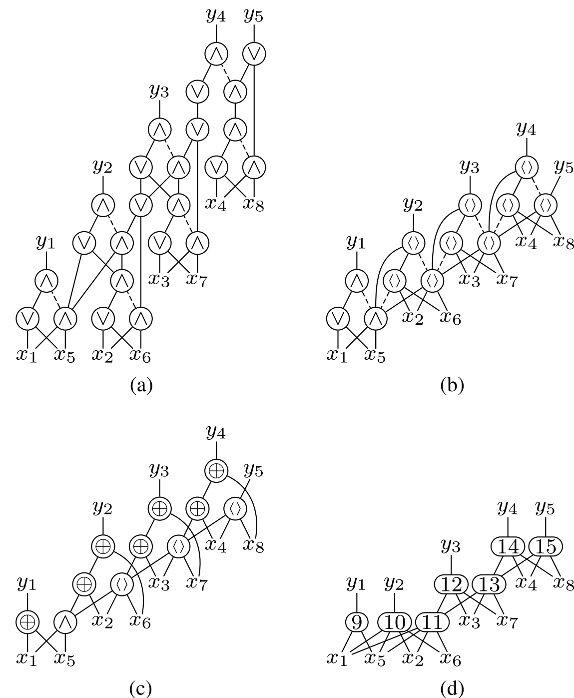


Fig. 2. *Logic networks for a 4-b adder: (a) and-inverter graph; (b) majority-inverter graph; (c) xor-majority graph; and (d) 4–LUT network.*

output, and the switching activity.[1] Most synthesis methods use stepwise refinement, i.e., an iterated replacement of fragments of the network while preserving input/output (I/O) behavior, mainly driven by heuristics descent strategies. Heuristics are chosen for the optimization goal (e.g., area recovery subject to timing constraints) and select network nodes where logic transformations are likely to have a beneficial effect. Recently, exact methods have emerged as a means to achieve directly the optimum network for a function, but its applicability is limited by the size of the network. Heuristic and exact methods are reviewed next.

IV. ALGORITHMS

We present here the underlying techniques for logic optimization algorithms. The minimization of two-level SOPs can be achieved by the program ESPRESSO [17], which embodies both an efficient implementation of the Quine–McCluskey [16] algorithm for exact minimization and fast near-optimum heuristics [13], [17]. The latter is used most often. The minimization of BDDs has been addressed by Drechsler [39] (exact method) and by others through heuristics. Here, we concentrate on multilevel networks as this model is the most widespread, and we present heuristics first and exact methods later. It is interesting to remark that artificial intelligence methods (in particular expert systems) were used in the early 1980s [45], [46] and later abandoned. A resurgent interest in machine learning synthesis is noticeable at the time of this writing [47], [48], but the related results are not (yet) strong enough and broadly used to deserve a report.

Various approaches to LN optimization have historical names that we preserve here, namely algebraic methods (based on polynomial algebra), algebraic rewriting (based on algebraic axioms, possibly of Boolean algebra), and Boolean methods (based on Boolean algebra). Heuristics are used in these approaches to select the type and sequence of transformations. While a combination of these methods (as often provided by the scripts of commercial tools) provide adequate engineering solutions, very few properties can be claimed on the synthesized circuits. This has motivated the recent search for exact methods that can yield subcircuits with provable properties.

A. Algebraic Methods

Traditional algebraic methods represent each LN node in SOP form (minimal with respect to single-cube containment [49]) and treat them as polynomials [49], [50]. This simplifying abstraction enables fast manipulation of very large LNs. Algorithms are designed as operators that iterate one type of transformation until the LN reaches a local minimum (with respect to the transformation itself). Examples of transformations are extraction, substitution, decomposition, and algebraic rewriting [12], [49].

1) Extraction: Extraction consists of searching common subexpressions of two (or more) functions, expressed as polynomials, in order to simplify the original ones. It relies on the search of appropriate common divisors (called kernels) that can be extracted to represent a new local function; the associated variable can thus be used to simplify the original expressions. The extraction problem can further be characterized as extraction of single-cube expression (i.e., of a monomial), and of multiple-cube expressions (i.e., of a polynomial). The algorithm for computing kernels was proposed by Brayton and McMullen [49].

Example 6: Consider the logic network given by

$$\begin{aligned} y_i &= x_1x_3x_5 + x_2x_3x_5 + x_4 \\ y_j &= x_3x_4x_5 + x_2. \end{aligned} \tag{4}$$

The common expression $y_k = x_3x_5$ can be extracted, and the network can be reexpressed as

$$\begin{aligned} y_i &= y_k(x_1 + x_2) + x_4 \\ y_j &= y_kx_4 + x_2 \\ y_k &= x_3x_5. \end{aligned} \tag{5}$$

2) Substitution: Substitution (also called resubstitution) means simplifying a local function by using an additional input coming from a node already present in the network. This input realizes already a part of the function that thus needs not to be replicated. Algebraic substitution makes use of algebraic division [49]: an expression y_i can be expressed as $y_jy_{\text{quotient}} + y_{\text{remainder}}$, where y_j is a divisor of the original function y_i.

Example 7: Consider the logic network given by

$$\begin{aligned} y_i &= x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4 + x_5 \\ y_j &= x_1 + x_2. \end{aligned} \tag{6}$$

Function y_j is a divisor of y_i and can therefore be used to express the network as

$$\begin{aligned} y_i &= y_j(x_3 + x_4) + x_5 \\ y_j &= x_1 + x_2. \end{aligned} \tag{7}$$

The implementation of algebraic substitution algorithms [49] can be very fast and provide an efficient algorithm for logic optimization.

3) Decomposition: Decomposition splits a local function (that may be too complex) into two smaller ones. The reverse transformation, i.e., merging to local functions is called elimination. There are many ways to per-

[1]We use abstract logic models to assess the quality of the network because we present and compare various emerging technologies. For established technologies, physical design and logic synthesis are combined, and optimality indicators are extracted from the circuit physical layout.

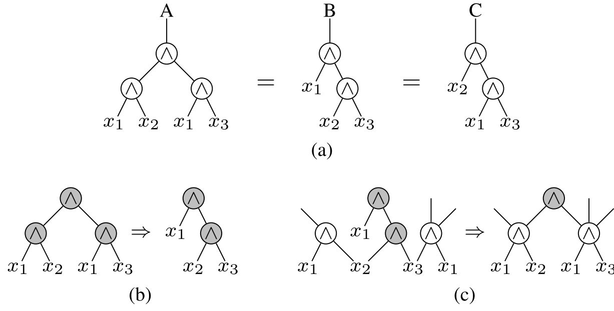


Fig. 3. ***Example of AIG rewriting from [52]. (a) Functionally equivalent AIG structures. (b) Rewrite structure A into B. (c) Rewrite structure B into A.***

form decomposition of logic functions [12], [50], [51]. A straightforward way is to divide algebraically a function y_i by one of its kernels that becomes a new node in the LN associated with variable j. Thus, y_i can be expressed as $jy_{\text{quotient}} + y_{\text{remainder}}$. In contrast to substitution, decomposition associates a new variable with the divisor. Decomposition can be applied recursively on the quotient and remainder.

Example 8: Consider the expression

$$y_i = x_1x_2x_5 + x_2x_3x_5 + x_4. \tag{8}$$

Let us introduce a new variable $j = x_1x_2 + x_2x_3$; it follows y_i can be decomposed as

$$y_i = jy_{\text{quotient}} + y_{\text{remainder}} = jx_5 + x_4. \tag{9}$$

4) Algebraic Rewriting: The purpose of algebraic rewriting is to reshape portions of an LN in order to improve the number of nodes and levels [52]. The general idea consists of applying transformation rules (based on algebraic axioms) with the objectives of improving some figure of merit. Rewriting is more effective when LNs are homogeneous (e.g., AIGs, MIGs, and XMGs), because logic transformations can be made specific. Algebraic rewriting has been used extensively in ABC [52]. For example, one can hold a database of precomputed circuit structures for a function. For any subcircuit, one can compute its function and check whether replacing the subcircuit by a precomputed structure leads to an improvement. If other nodes in the circuit are reused in the rewriting, it may be even beneficial to replace a smaller structure by a larger one.

Example 9: An example for AIG rewriting as it is implemented in ABC is shown in Fig. 3. Fig. 3(a) shows three functionally equivalent AIGs structures. These equivalences are employed in Fig. 3(b) and (c) to reshape the structure of AIGs into functionally equivalent ones.

Refactoring is a variant of rewriting, in which large cones of logic feeding a node are iteratively selected with the aim to replace them by a factored form of the function. The change is accepted if there is an improvement in the selected cost metric (usually the number of nodes) [52], [53].

Algebraic rewriting is very effective for MIGs and XMGs. The related majority algebra and axiomatic system Ω have been described in [43], where it is shown that Ω is sound and complete, providing reachability in the solution space. In simple words, this means that for MIGs and XMGs there exist a sequence of steps leading to the optimum solution. Such a path may not exist in other representation frameworks. Indeed, experimental evidence has shown that the MIGhty program [43] implementing algebraic rewriting has outperformed other tools on several benchmarks [43], and especially on large arithmetic functions.

The MIG axiomatic system Ω consists of five primitive transformation rules that can be used to rewrite MIGs

$$\Omega \begin{cases} \textbf{Commutativity} - \Omega.C \\ \langle xyz \rangle = \langle yxz \rangle = \langle zyx \rangle \\ \textbf{Majority} - \Omega.M \\ \langle xxy \rangle = x \qquad \langle x\bar{x}y \rangle = y \\ \textbf{Associativity} - \Omega.A \\ \langle xu\langle yuz \rangle \rangle = \langle zu\langle yux \rangle \rangle \\ \textbf{Distributivity} - \Omega.D \\ \langle xy\langle uvz \rangle \rangle = \langle \langle xyu \rangle \langle xyv \rangle z \rangle \\ \textbf{Inverter Propagation} - \Omega.I \\ \overline{\langle xyz \rangle} = \langle \bar{x}\bar{y}\bar{z} \rangle. \end{cases} \tag{10}$$

An MIG can be transformed into another MIG by just using the rules in Ω in either direction as well as additional rules. Such rules can reduce the number of nodes and depth of an LN, or any other metric [10]. Note that MIGs can be generalized to using majority-n functions [54], [55].

Example 10: An MIG for function y and its optimized version are presented in Fig. 4. The optimized version [see Fig. 4(b)] has been obtained by applying the distributivity rule and by considering that $\langle 10x \rangle = x$. Both depth (number of levels) and size (number of nodes) are optimized. Details about MIG theory and implementation in program MIGhty can be found in [10] and [43].

B. Boolean Methods

When using Boolean methods, each node of the LN is associated with both a local function and a local don't care set [12], [56]. The full power of Boolean algebra is used to construct local transformations that attempt to improve the LN characteristics [12], [49], [57]. This can be typically done by checking that the perturbation introduced by the optimization step is contained in the don't care set,

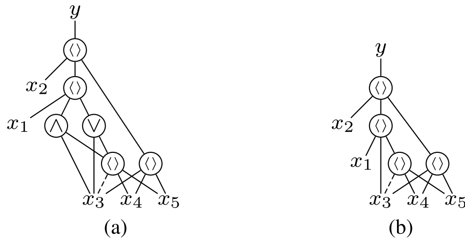


Fig. 4. ***MIG algebraic rewriting. (a) MIG: before algebraic rewriting. (b) MIG: after rewriting.***

which represents the tolerance on the perturbation. In this scenario, often the function at a node n can be changed to another function without affecting the functionality at the primary outputs [58], [59]. The new function was called a permissible function for node n by Muroga, who did pioneering work on these methods [59]. Unfortunately, his work based on tabular descriptions had limited impact because the data structure and algorithms, as well as the contemporary computers, were not efficient enough to operate on LN of reasonable size.

Boolean methods evolved through time as different engines became available for doing the essential task of detecting the existence of permissible functions. The MIS/SIS program [50] used program ESPRESSO to find permissible functions by optimizing the literal count of two-level logic expressions associated with LN nodes and their don't care sets. This approach is fast and practical, but not general enough as it may miss good solutions.

Other tools used BDDs to check if a function is a permissible replacement of another by checking the tautology of their equivalence. Fast tautology check can be provided by BDD tools [60] and thus desirable permissible replacements of a local function can be quickly evaluated. As a specific example, technology mapping with Boolean matching aims at replacing a portion of an LN by an element of a cell library. The feasibility of the matching is done using BDDs [61], [62]. Moreover, when the candidate permissible functions are many, their implicit enumeration through BDDs [62] makes their search very effective.

In general, Boolean methods can also be enabled by casting the search for permissible functions as a satisfiability problem, and using an effective SAT solver for this task [63]. Overall, Boolean methods leverage a variety of transformations that eventually resort to an engine for verifying their applicability. Examples of engines are two-level minimizers, BDD, and SAT packages. We review some transformations next.

1) Substitution: As in the algebraic methods, substitution reexpresses the local function of an existing node using the input of other nodes already present in the LN. Substitution can be computed in various ways and it is inherently more expensive than algebraic substitution. Traditionally, it is realized by minimizing a local function with its local don't care set, which may contain variables not originally present in the function support. Indeed the local don't care set expresses the mutual controllability and observability links among local functions in an LN, and thus enables the reexpression of a portion of a function through a new input.

Example 11: Consider the logic network given by

$$\begin{aligned} y_1 &= x_1 + x_2x_3x_4 + x_5 \\ y_2 &= x_1 + x_3x_4. \end{aligned} \tag{11}$$

Assume that we want to substitute y_2 into y_1. Then, we minimize y_1 with the don't care conditions induced by the second assignment $y_2 \oplus (x_1 + x_3x_4)$. The minimized expression yields

$$\begin{aligned} {y_1}^* &= x_1 + x_2y_2 + x_5 \\ y_2 &= x_1 + x_3x_4 \end{aligned} \tag{12}$$

thus effectively reducing one literal in the first expression by using the output of the second one.

2) Rewriting: Rewriting aims at minimizing the size of an LN by iteratively selecting subnetworks and by replacing them with smaller precomputed subgraphs, while preserving the functionality. This is achieved by applying a Boolean equivalence check (modulo the don't cares).

Example 12: Examples of typical precomputed subnetworks are all four variables functions, or their 222 NPN equivalence classes [23], [44], [52]. Here, the idea is to replace four-input subnetworks with their optimum precomputed representation.

3) Redundancy Removal and Rewiring: Redundancy removal is a common technique that uses automatic test pattern generators (ATPGs) to detect untestable stuck-at faults in an LN and modifies the network at the faulty net by setting it to a constant value [64], [65]. Rewiring improves on redundancy removal because it adds new connections in an LN to create redundancies that later can be removed. In practice, it adds and removes nets and it aims at removing nets related to long wires [66].

Rewiring fits into a general paradigm where an LN is optimized by changing a local function (to improve overall size and/or depth) by introducing errors that are then corrected by changing the local functionality somewhere else [67]. The following example shows this type of transformation for MIGs.

Example 13: We show an example that makes use of an induced error correction technique for MIGs, which was first explained in [68]. The technique is based on the

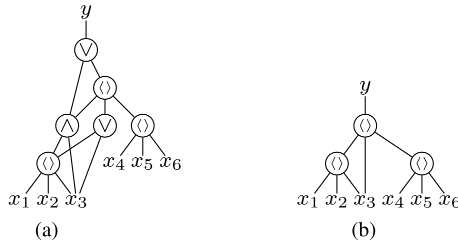


Fig. 5. *Rewiring based on induced error correction in MIGs. (a) Initial MIG. (b) Optimized MIG.*

property that

$$y = \langle y_1 y_2 y_3 \rangle \text{ if, and only if } (y \oplus y_i)(y \oplus y_j) = 0 \text{ for all } 1 \leq i < j \leq 3. \tag{13}$$

We can think of each y_i, $i = 1, 2, 3$ as a convenient (i.e., reduced and thus incorrect) version of y. The difference between y_i and y is expressed by $(y \oplus y_i)$ in the local error. The condition on the right-hand side of (13) states that all three errors must be pairwise orthogonal, i.e., the pairwise differences have an empty intersection. In this condition, the majority operator restores the correct functionality. Fig. 5(a) shows an MIG for the function y, which has the truth table #f8f8f8e0f8e0e0e0. One can easily verify that the right-hand side condition in (13) is satisfied for $y_1 = \langle x_1 x_2 x_3 \rangle$, $y_2 = x_3$, and $y_3 = \langle x_4 x_5 x_6 \rangle$, and therefore, $y = \langle \langle x_1 x_2 x_3 \rangle x_3 \langle x_4 x_5 x_6 \rangle \rangle$, for which an MIG is shown in Fig. 5(b). The optimized MIG reduces both size and depth to half of their original values. More details on this technique including methods to derive valid fault candidates are described in [68].

C. Exact Methods

Exact synthesis is the problem of finding the optimum logic representation for a given Boolean function with respect to some cost criterion. We consider here logic networks where the cost is either the number of gates (or equivalently nodes and correlated to area) or the depth of the LN (or equivalently the critical path and correlated to delay). For example, a well-known exact algorithm is *FlowMap* that determines a minimum-depth mapping of an LN into k-LUTs in polynomial time [69]. Note that an optimum circuit implementation is not necessarily unique. For example, the majority-5 function can be realized with the minimum number of majority-3 gates in more than one way, for example

$$\langle \langle x_3 x_4 x_5 \rangle x_2 \langle x_1 x_2 \langle x_3 x_4 x_5 \rangle \rangle \rangle$$

and

$$\langle x_1 x_2 \langle \langle x_3 x_4 x_5 \rangle x_2 \langle x_3 x_4 x_5 \rangle \rangle \rangle.$$

Theoretical bounds can be derived under various assumption. For example all four-variable Boolean functions can be represented using SOPs with at most eight implicants [19]. All five-variable Boolean functions can be represented using two-input LNs with at most 12 gates [21]. Since the number of Boolean functions grows double exponentially with the support size, it is hard to compute bounds for a larger number of variables. To give a sense of the kind and applicability of exact synthesis, we present here two exact synthesis methods: 1) an implicit LN enumeration method for area minimization; and 2) an explicit LN enumeration method for delay minimization. We refer the interested reader to [70] for details and other approaches.

1) Implicit Network Enumeration Methods: Implicit network enumeration methods aim at exploring the logic representation space, in search for optimum networks, with the help of constraint satisfaction and optimization techniques, such as integer linear programming or Boolean satisfiability [71]. Implicit enumeration methods are considered the most scalable ones for exact synthesis, especially considering Boolean functions of five, six, or more variables, implemented in common technologies. In this work, we focus on Boolean satisfiability as a main reasoning engine for exact synthesis.

To showcase how implicit enumeration methods for exact synthesis can be driven by SAT engines, we present in the following details on "SAT-based exact synthesis for minimum gate count." The same approach can be naturally extended to minimum delay, minimum power, and other types of network costs, hence it will not be discussed here for the sake of brevity.

a) SAT-based exact synthesis: Given an m-tuple of m functions over n variables

$$(y_1(x_1, \ldots, x_n), \ldots, y_m(x_1, \ldots, x_n))$$

we can formulate the exact synthesis of these functions as a sequence of decision problems $P_0, P_1, P_2, \ldots$. Problem P_r corresponds to the question: Can functions $y_1, \ldots, y_m$ be computed by an r-gates Boolean LN? Without loss of generality, we assume as a default situation that any two-input logic gate is available for synthesis, but the problem formulation can be tailored to a given technology library that implements a universal gate set. Each instance P_r can be described by a SAT formula.[2] Hereafter, we describe what such a formula looks like, how additional constraints can speed up the synthesis process, as well as some experimental results. We attribute the SAT formulation described here to Kojevnikov *et al.*, [22], Knuth [28], and Eén[71]. Knuth [28] improved previous approaches, by restricting synthesis to normal Boolean functions.

[2] In practice, P_0 is often handled as a trivial special case, since it means that all $y_1, \ldots y_m$ are constants or variable projections.

b) Definitions and variables: For our SAT-based exact synthesis purposes, an r-gates LN with n inputs $x_1, \ldots, x_n$ is a sequence of (two-input) gates $(x_{n+1}, \ldots, x_{n+r})$ with

$$x_i = x_{j(i)} \circ_i x_{k(i)}, \qquad \text{for } n+1 \le i \le n+r. \qquad (14)$$

That is, each gate combines two previous gates or inputs with $j(i) < k(i) < i$ using $\circ_i$, which is one of the two-input Boolean functions. For single-output functions, the last gate x_{n+r} is considered the network's output. For multiple-output networks, each gate could potentially be an output. We call a single-output function f normal, if $f(0, \ldots, 0) = 0$. A multiple-output function is normal, if all of its component functions are normal. An LN represents normal functions if all of its gate functions are normal.

To proceed with the formulation of this exact synthesis problem, we define the variables to be used in the SAT formula. For $1 \le h \le m$, $n < i \le n+r$, and $0 < t < 2^n$, define the following:

$$\begin{aligned} x_{it} &: t^{\text{th}}\text{bit of} x_i{}'\text{struthtable} \\ g_{ih} &: [y_h = x_i] \\ s_{ijk} &: [x_i = x_j \circ_i x_k] \text{for } 1 \le j < k < i \\ f_{ipq} &: \circ_i(p, q) \text{for } 0 \le p, q \le 1, p+q > 0. \end{aligned}$$

The variables x_{it} correspond to the value (at row t) of the global truth table for gate x_i. The g_{ih} variables determine which outputs point to which gates. Thus, if g_{ih} is true, it means that function y_h is computed by gate i. The s_{ijk} variables determine, for each gate i, the inputs j and k. Also known as selection variables, their assignments control the underlying DAG structure of the LN. The f_{ipq} encode for all gates i what the corresponding Boolean operator is. Since we synthesize normal logic networks, we do not need to consider row 0 of the gate's truth tables and require only $2^n - 1$ truth table indices t. Also $p + q > 0$, since the local function describing a gate's operation does not need to be specified for the case $p = q = 0$.

c) Constraints: We now constrain the variables by a set of clauses which ensure that the network computes the correct functions. With the addition of these clauses, the SAT formula is satisfiable if and only if the given functions can be computed by an r-gate logic network. For $0 \le a, b, c \le 1$ and $1 \le j < k < i$, the main clauses are

$$((s_{ijk} \wedge (x_{it} \oplus \bar{a}) \wedge (x_{jt} \oplus \bar{b}) \wedge (x_{kt} \oplus \bar{c})) \rightarrow (f_{ibc} \oplus \bar{a})).$$

In other words, if gate i has inputs j and k, and the tth bit of x_i is a, and the tth bit of x_j is b, and the tth bit of x_k is c, then we must have $\circ_i(b, c) = a$. We can rewrite these constraints to CNF

$$(\bar{s}_{ijk} \vee (x_{it} \oplus a) \vee (x_{jt} \oplus b) \vee (x_{kt} \oplus c) \vee (f_{ibc} \oplus \bar{a})).$$

Here, a, b, and c are constants used to set the proper variable polarities. In fact, these constraints may be simplified in several cases. When $b = c = 0$, the final term encodes f_{i00}. If $a = 0$, this is trivially true, due to the normality of the network. Hence, in that case, the entire clause may be omitted. If $a = 1$, the final literal is omitted from the clause. Similarly, x_{jt} and x_{kt} are constants if $j \le n$ or $k \le n$, and the appropriate simplifications can be made.

Next, let $(t_1, \ldots, t_n)_2 = t$ be the binary encoding of t, such that t_i refers to the ith bit of t. In order to fix the proper output values, we add the clauses $(\bar{g}_{hi} \vee \bar{x}_{it})$ or $(\bar{g}_{hi} \vee x_{it})$ depending on the value $y_h(t_1, \ldots, t_n)$. Finally, we add the clauses $\bigvee_{i=n+1}^{n+r} g_{hi}$ and $\bigvee_{1 \le j < k < i} s_{ijk}$, so that every output h points to a gate in the network and to ensure that every gate i has two inputs.

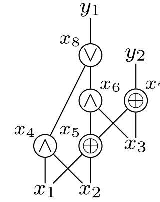


Fig. 6. ***Illustration of a size-optimum (in a number of two-input gates) logic network for a full adder with carry y_1 and sum y_2.***

Example 14: We illustrate the encoding by means of the example network in Fig. 6 that is size-optimum to realize a full adder. Let us consider a variable assignment that would synthesize it. It has five gates, so the corresponding decision problem is P_5 and $r = 5$. Further, it has three inputs and two outputs. Hence, indices i and t range from 4 to 5 and from 1 to 7, respectively

$$\begin{aligned} t &= 7\ 6\ 5\ 4\ 3\ 2\ 1 \\ x_{4t} &= 1\ 0\ 0\ 0\ 1\ 0\ 0 \\ x_{5t} &= 0\ 1\ 1\ 0\ 0\ 1\ 1 \\ x_{6t} &= 0\ 1\ 1\ 0\ 0\ 0\ 0 \\ x_{7t} &= 1\ 0\ 0\ 1\ 0\ 1\ 1 \\ x_{8t} &= 1\ 1\ 1\ 0\ 1\ 0\ 0. \end{aligned}$$

There are two outputs, each of which can point to five gates, making for a total of ten g_{hi} variables. In this case, we have $g_{17} = 1$, $g_{28} = 1$, and $g_{hi} = 0$ for all other g_{hi}.

From the DAG structure of the network, we can see that $s_{412} = 1$, $s_{512} = 1$, $s_{635} = 1$, $s_{735} = 1$, and $s_{846} = 1$. All other s_{ijk} are zero.

Table 1 Exact Area Synthesis of all Four and Five-Input NPN Classes and a Set of Six-Input DSD Functions. All Runtimes Are in Milliseconds

Function set	Nr. of functions	Mean runtime (ms)	Total runtime (ms)
NPN4	222	225.46	50052.12
NPN5	616,126	553.29e+03	5,506,943.47e+03
FDSD6	1000	69.00	69000.00

Finally, the Boolean operators for the different gates are assigned the following values:

$$\begin{array}{rcccc} (p,q) & = & (1,1) & (0,1) & (1,0) \\ f_{4pq} & = & 1 & 0 & 0 \\ f_{5pq} & = & 0 & 1 & 1 \\ f_{6pq} & = & 1 & 0 & 0 \\ f_{7pq} & = & 0 & 1 & 1 \\ f_{8pq} & = & 1 & 1 & 1. \end{array}$$

d) Additional clauses: The above clauses are the minimum ones necessary to ensure that a valid logic network is found. However, we may add additional constraints to boost synthesis speed, such as clauses to force a colexicographic order on the gates. We refer the reader to [28] for the details.

e) Algorithms: Now that we know how to create the SAT formula for P_r, we can use that to construct an exact synthesis algorithm. We would start by solving SAT(P_i), with $i = 0$, and then increasing i as long as the answer is unSAT. It is evident that the first satisfiable answer corresponds to an exact circuit solution.

f) Experimental results: Table 1 shows experimental results for the synthesis of three sets of functions, using the algorithm described in this section. The set NPN4 consists of all 222 four-input NPN classes. The set NPN5 consists of all 616 126 five-input NPN classes. The set FDSD6 consists of 1000 fully disjoint support set (DSD) decomposable functions [72]. NPN4 and FDSD6 sets of functions can be fully synthesized in less than 70 s. On average, all functions are synthesized in (much) less than 1 s. Interestingly, the six-input DSD functions have a lower average runtime than the four-input NPN classes. This is due to the fact DSD functions are rather special functions, at times easier to synthesize. Considering NPN5, not only the number of classes is about $3000\times$ larger than NPN4, but also the average complexity of each function increases. As a result, exact synthesis of each five-variable function is more difficult than in the four-variable case.

2) Explicit Network Enumeration Methods: Explicit enumeration methods aim at exhaustively exploring the logic representation space, or a well-defined subportion, looking for optimum networks. For Boolean functions with four, five variables maximum, or considering exact synthesis problems with special constraints on network topology, a small number of gates, etc., explicit enumeration can outperform implicit enumeration in terms of execution runtime. For example, it takes less than 2 min to generate all delay-optimal circuits of four variables, for more than 200 input arrival time patterns, using a recently introduced explicit enumeration method [25]. On the other hand, SAT-based methods can take more than 3 h to find the same circuits. However, one has to be cautious when using explicit network enumeration: when the number of variables grows too much, or the filters on the search space are not tight enough, explicit network enumeration may be inapplicable because of the memory footprint, without even considering the super-exponential runtime blowup. Nevertheless, there are still applications in EDA where explicit enumeration is of interest [73].

To showcase how explicit enumeration methods for exact synthesis can be implemented effectively, we present in the following a procedure for optimum delay circuits enumeration. The same approach can be extended to the minimum area and other metrics: we refer the reader to [25] for more details.

a) Optimum delay circuits enumeration: We consider the problem of finding all minimum delay logic circuits of n variables, given a technology library L and input arrival pattern T. This problem arises when a complete exact delay database needs to be populated [25]. The following procedure that we are going to describe depicts a high-level flow for explicit circuit enumeration for exact delay synthesis. We first store trivial circuits for the logic constants and input variables. These circuits, which are simple wires, are delay optimal by construction. Then, we start an enumeration loop where we try to add a new gate from L, in increasing delay order, having as fanin some of the already stored functions, also in increasing arrival time order. If the generated function is not already stored, we save it. Otherwise, we already have a better delay implementation stored for the generated function. We keep iterating this procedure until we have stored circuits for all the 2^{2^n} functions. It can be proven that this procedure only stores optimum delay circuits. Note that such procedure can be sped up by taking into account library considerations and function filtering. On the library side, we can filter based on the gate properties, e.g., functional symmetry, delay dominance and decomposition, etc. On the function side, we can filter based on considerations on NPN classification properties of the already stored functions. With all the filtering, explicit enumeration is fast. It takes less than 2 min to generate all optimum delay circuits of four variables for a typical L in CMOS technology [25].

Fig. 7 shows a sample entry for an optimum delay database, generated by the aforementioned explicit circuit enumeration algorithm. The minimum delay precision is set to 0.5 delay units for this example, for practical reasons. The gate delays are extracted from a characteristic CMOS library in a 45-nm technology node.

D. Scalable Synthesis Flows

Boolean methods achieve better results than algebraic methods; they are also more precise. However, the price

	$t_y = 8$
$t_{\text{NOR}} = 1$	$t_{x_1} = 0$
$t_{\text{AND}} = 1.5$	$t_{x_2} = 3.5$
$t_{\text{NOT}} = 0.5$	$t_{x_3} = 2$
$t_{\text{XOR}} = 2$	$t_{x_4} = 1$

Fig. 7. *Sample entry of an optimum delay circuit database.*

for better quality is worse scalability. Exact methods can find optimum circuits and are applicable when circuits are sought in the presence of many constraints; but these methods are only applicable to functions with a few number of inputs. This section discusses how such techniques can nevertheless be integrated into a robust and scalable synthesis flow, by making use of partitioning techniques. Breaking down the logic network into smaller fractions allows us to apply runtime-intensive methods in a controllable manner.

1) Windowing: A generic scalable logic synthesis framework is described by Mishchenko and Brayton [53]. The general idea is to move a small window (with restricted and controllable fanin and unlimited fanout) over the logic network. Algorithms can be applied on the window resulting in local optimization of subgraphs. Generally, a windowing procedure takes as input a directed acyclic graph and two positive integers which denote the maximal number of primary inputs of the window and the maximal number of nodes, respectively. We refer the reader to [53] for a detailed description of the algorithm.

2) LUT Mapping: LUT mapping allows us to cover the logic network with k-input lookup tables. LUT mapping is usually based on k-feasible cut enumeration [74]–[76], and depth-optimal LUT mapping is based on the FlowMap algorithm [69]. k-LUT mapping can be used as partition method of large networks in order to apply exact solutions on smaller functions of k inputs. Results of exact synthesis algorithm on LUT-mapped networks can be found in [44] to optimize size and in [26] to optimize logic depth.

V. APPLICATIONS

Logic synthesis is a technology enabler [11], in the sense that it allows us to validate the effectiveness of new technologies on established and emerging architectures. Some system and technology trends are reviewed in this special issue. It can be argued that future computing systems will involve a plurality of hardware solutions, in the search of the best match to the application of interest.

Whereas the results of using logic synthesis in various emerging technologies has been reported elsewhere [77]–[79] and reviewed in [11], we consider here the applications of logic synthesis methods to CMOS technology, optical computing (focusing on plasmonic logic), and quantum-dot cellular automata as well as quantum computing. This choice is motivated by the belief that CMOS will continue to be the mainstream technology, enhanced by emerging optical computing and communication technology. Similarly, we think that a design flow including specialized logic synthesis is necessary to scale up quantum computing (in various technologies and embodiments) to provide us with versatile and powerful computing means.

A. CMOS Targeted Synthesis

This section demonstrates applications for classical CMOS technologies in which the novel logic synthesis data structures and algorithms described in this paper lead to significant improvements. First, we show how majority-based logic networks as intermediate representation enable further logic optimization that carries through to technology mapping. Second, we show examples in which SAT-based techniques are successfully employed into scalable synthesis flows for standard cell and LUT-mapping.

The most recent results obtained using majority-based (i.e., MIGhty) logic optimization are reported in [10]. Logic functions are represented by MIGs and further optimized using both algebraic and Boolean methods, here summarized in Section IV. A selection of circuits from both IWLS'05 benchmarks and HDL arithmetic benchmarks have been considered and synthesis results obtained with MIGhty are compared to AIGs optimized by ABC [80] in terms of size and depth. Considering the IWLS'05 benchmarks, an average 14% reduction in depth and 4% in size are achieved by MIGhty. Focusing on the arithmetic HDL benchmarks, MIGhty enables about 33% depth reduction combined with 4% reduction in size. Note that these improvements are obtained with similar runtime between MIGhty and ABC. The benefits in area and delay are also found after place and route, although part of the advantage is absorbed by place and route. For the case of FPGA design (synthesis followed by place and route) on a commercial 28-nm technology node, employing MIGhty as front-end to the FPGA design flow, better final circuits are obtained both in terms of LUT count, delay, and power metrics. When targeting delay as main optimization objective, a 10% improvement is achieved [10, Table 3]. The application-specific integrated circuit (ASIC) design follows a similar trend on a commercial 22-nm technology node. MIGhty achieves better final circuits both for area, delay and power. Also in this case, when targeting delay as main optimization objective, a 13% improvement is obtained [10, Table 4].

The first SAT-based framework dealing with standard cells has been recently included in ABC [53] and reported in [81]. A novel SAT-based procedure is used to enumerate standard-cell implementations of the target node. It is employed as a postprocessing step, aiming at reducing the size of already mapped networks. The experimental results show that a 2.5% further reduction [81, Table 1] in the number of standard cells is obtained when optimizing circuits mapped using heuristics from ABC. A most

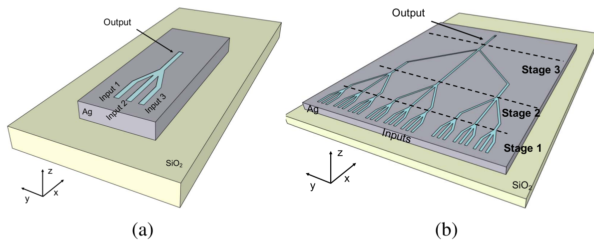


Fig. 8. *(a) Layout for a plasmonic-based majority-3, and (b) three-stage cascaded plasmonic majority circuit.*

recent work has considered SAT-based area recovery for technology mapping [82]. The SAT-based procedure has been implemented in ABC and tested on a suite of EPFL benchmarks [68] mapped into 6-LUT logic network. When targeting area reduction, an average area reduction of 3.5% is obtained and a delay improvement is achieved in most examples. For several arithmetic benchmarks, the area reduction is very substantial, with values up to 11% [82, Table 2].

Open source implementations of most of aforementioned synthesis algorithms for CMOS are available at *github.com/lsils/mockturtle*.

B. Majority-Based Technologies

With transistor dimensions reaching their scaling limits, it is interesting to look at disruptive computation paradigms offered by emerging nanotechnologies. Examples of majority-based beyond CMOS technologies include, but are not limited to, quantum-dot cellular automata (QCA, [6]), nanomagnet logic [83], spin-based devices (e.g., spin-wave devices [84] and spin torque majority gates [85]), and plasmonic-based devices [86]. Here, we describe as examples plasmonic-based devices and QCA, and we illustrate how logic synthesis data structures and algorithms described so far can be employed in order to realize logic circuits based on these new paradigms of computation. It is worth noting that logic synthesis for spin-based devices has already been extensively studied [11], [77]; nevertheless, the following discussion can be easily extended and applied to other majority-based nanotechnologies.

1) Plasmonic-Based Devices: Plasmonic-based devices [86] described hereafter are based on the propagation of surface plasmon polaritons (SPP, [87]), which are electromagnetic waves propagating at the interface between a dielectric and a metal. In particular, plasmonic-based logic makes use of the phase ϕ of the SPP as logic variable. The computation is based on the interference of waves: in general, the output depends on the number of inputs with phase ϕ and $\phi + \pi$.

Functionality: The phase of interfering SPP waves follows the majority rule; this makes the three-input majority function easy to realize with plasmonic-based devices [86]. Fig. 8(a) shows a single-stage three-input plasmonic majority gate layout.

Thanks to the physics of plasmonic devices that can be abstracted as multivalued logic, it has been shown [86] that a nine-input majority gate can be easily realized using four three-input plasmonic devices. Note that the best realization of majority-9 in binary-valued logic known so far uses 15 majority-3 gates [55], and thus plasmonic devices may be more efficient (as compared to other wave-based devices) to realize logic circuits. The wave nature of the computation allows us to easily implement the INVerter by using a waveguide of half the length of the SPP wavelength. Thanks to this property, a complete set of logic primitives (INV and MAJ) can be build using plasmonic-based devices.

a) Constraints and costs: As stated above, plasmonic-based devices make a complete set of Boolean primitives; however, some constraints arise due to the wave nature and the physics of this device. As an example, the propagation losses of SPP puts a limitation on the number of cascaded stages (i.e., the number of levels of the circuits). Currently, it is not efficient to have more than three stages, which means that after the third stage, either an amplifier or a converter to voltage domain is necessary. An example of three-stage cascaded plasmonic majority is shown in Fig. 8(b). The propagation losses across the first stage are around 30%, and keep increasing at every cascaded stage. The increase in propagation losses between the different stages is a direct consequence of the size difference between the devices in different stages [as shown in Fig. 8(b)]. As the size of the majority gates increases with the number of stages, also the delay of devices at different stages follows a similar trend. Furthermore, since the SPP wavelength has different values according to the stage, also the inversion cost depends on the stage at which it is implemented. It should also be noted that most emerging nanodevices target ultralow energy operation, with an inherent low amplification and reduced driving capabilities. Thus, in addition to the constraints already considered, we expect this technology to have limitations on the number of outgoing waves (i.e., to the maximum fanout of each device).

b) Logic synthesis algorithms: The constraints in depth, fanout, and functionality that arise with the use of plasmonic-based technology are best dealt with using SAT-based exact synthesis (see Section IV-C). In fact, any additional application constraint corresponds to an additional constraint added to the SAT formula; at the same time, the circuit size that the SAT solver has to work with is limited by the depth constraint. A SAT constraint can be used to target the use of the most suitable type of logic primitives, according to the technology in use. For example, working with plasmonic-based devices, one might wish to enable the use of the compact MAJ-9 implementation. Depending on the logic representations for which the synthesis has to be performed (depth, fanout, etc.), additional constraints may also be easily implemented. In [88], a SAT-based method that works on MIGs is used to produce majority-based networks that can be mapped using devices with

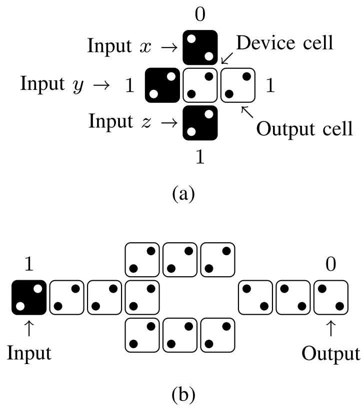


Fig. 9. ***QCA layout for (a) majority, and (b) inverter. The black cells are the primary inputs.***

restricted fan-out and depth. However, due to the limited scalability of the SAT-based method [88] and the technological constraints of plasmonic-based devices, decomposition solutions need to be addressed in order to be able to support large functions. The core idea is to partition the main function into smaller ones, where each of these small functions meets the given constraints (for further details refer to Section IV-D). In [88], a partition method based on LUT-mapping is presented, showing that, on average, 86.6% of 6-LUTs from EPFL benchmarks [68] can be realized using majority-based plasmonic circuits with maximum depth and fanout of 3.

2) Quantum-Dot Cellular Automata: QCA technology [6] is based on the interaction of QCA cells; each cell consists of four quantum dots and two free electrons. The two electrons are coupled by tunnel barriers and they can tunnel between the dots. The electrons are forced by Coulomb repulsion in opposite corners of the cell producing two energetically equivalent polarizations, i.e., $P = 1$ and $P = -1$. The two polarizations are used as logic variables; i.e., to represent logic 1 and 0, respectively. However, for these polarization states to be energetically stable, the operating temperature is limited to $\sim$ 1K [89].

a) Functionality: The fundamental logic element of QCA is the three-input majority gate [90]. Fig. 9(a) shows the layout of a QCA majority gate; five QCA cells are needed to build one single majority gate. The polarization of the central logic cell, called device cell, is the majority of the three inputs, while the output cell follows the polarization of the device cell.

The inverter can be realized as shown in Fig. 9(b) [90], using 13 quantum cells, and thus it is more expensive as compared to majority.

In the last few years, five-input majority gate realizations using QCA cells have been intensively studied [93]. The majority-5 is a versatile primitive and it can be employed to realize a variety of functions. Fig. 10 shows one of the first implementations of the majority-5 [91]. It only requires ten QCA cells; on the other hand, the input cells are close to each other and difficult to be accessed. Improved versions of five-input majority have been recently proposed [92], [93]. In these implementations, the five inputs are easier to reach, allowing single-layer accessibility to the input and output cells.

b) Constraints and costs: QCA technology enables the realization of three- and five-input majority gates, plus the inversion. As in the previous case, some limitations and costs for circuits realization need to be discussed.

The cost used to compare QCA blocks is the number of QCA cells [93], i.e., the area. The area of a QCA layout can be obtained by analyzing the layout with QCADesigner [94], a tool for the layout and analysis of QCA technology circuits. In this scenario, the inverter implementation is very expensive in terms of number of cells as compared to the majorities. Note that 11 QCA cells are needed for a single five-input majority, while 13 are necessary to change the polarity of each cell. It is thus preferable to limit the number of inversions in the circuit. Considering further constraints, each three- and five-input majority block has a fanout limited to 3. This is due to the fact that in order to have the same polarization, two cells should be aligned and close to each other on one of the square borders (being 3 for each output signal). Moreover, the fabrication of interconnections between building blocks needs to be handled efficiently for a better stability. Until now, an efficient and robust realization of wire crossing is not available, thus a river routing is needed [93].

c) Logic synthesis algorithms: QCA technology is mainly based on three-input majority and INV. This makes MIGs a perfect data structure to fully exploit QCA functionality. Several MIG optimization algorithms have been proposed which can be employed to lower the cost metrics with respect to both area and delay [10].

Logic synthesis algorithms need also to consider that QCA technology does not offer efficient implementation of inverters. Therefore, it is required to minimize their application or even to eliminate them from implemented circuits. In [95], an algorithm to minimize inversions with application to QCA technology is presented, providing on average an additional 5% area reduction on the EPFL arithmetic benchmarks [68], already optimized using MIGhty.

A different approach is presented in [79] where circuits are realized using only majorities, after moving all the INVs to the primary inputs. The same work also tackles the issue of gates with large fanout, describing an algorithm to limit the fanout of each MIG node to a given maximum value. Despite the strict constraints used, the results presented

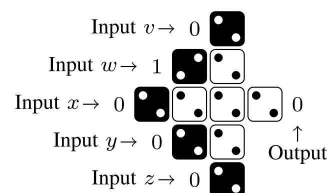


Fig. 10. ***Layout of five-input majority with QCA [91].***

in [79] show that having inverter-free circuits enables a $3.1\times$ reduction in area delay energy product (ADEP).

Both plasmonic-based devices and QCA offer majority gates with different arities as primitives. Larger majority-gates can lead to cost reductions if properly exerted by the logic synthesis algorithm. In general, majority-n logic synthesis addresses such problems (see, e.g., [54], [96], and [97]), including mapping combinational logic into large majority gates and decomposing large majority gates into smaller ones.

C. Quantum Computing

In this section, we illustrate how logic synthesis techniques can be used in applications different from conventional computing. We show how logic synthesis helps to compile combinational logic for quantum computers.

Quantum computers are computers that exploit the principles of quantum mechanics. Their premise is to execute quantum algorithms, which can be computationally superior to their classical counterparts. Several quantum algorithms have already been conceived, which can harness the power of a quantum computer to eventually solve complex problems more efficiently. The most prominent one is arguably Shor's algorithm [98] that can factorize integers in polynomial time, whereas for classical computing nothing better than a subexponential upper bound is known [99]. Consequently, Shor's algorithm can break public-key cryptography which is based on the assumption that integer factorization is a hard task. Other more generic algorithms play a significant role in scientific applications of high interest. Examples are as follows:

- Grover's search algorithm [100], which enables faster database queries;
- the HHL algorithm [101], which brings an exponential speedup to solve linear equations;
- quantum simulation (see, e.g., [102]) to model atomic-scale interactions efficiently, allowing to approximate behavior in drugs, organics, and materials in areas such as medicine, chemistry, and engineering, respectively.

1) Functionality: A quantum computer consists of an array of quantum bits, also called qubits, that in contrast to classical bits, can be in a superposition state and can be entangled [7]. Formally, a qubit is in a quantum state that is a column vector $|\varphi\rangle = \binom{\alpha}{\beta}$ of two complex numbers α and β, called amplitudes, such that $|\alpha|^2 + |\beta|^2 = 1$. The squared amplitudes $|\alpha|^2$ and $|\beta|^2$ indicate the probability that the quantum state will collapse to the classical state $|0\rangle = \binom{1}{0}$ or $|1\rangle = \binom{0}{1}$ after the qubit is measured. A quantum state can be transformed into another quantum state by applying quantum gates, which are represented by 2×2 unitary matrices. For example, the Hadamard gate $H = (1/\sqrt{2})\left(\begin{smallmatrix}1 & 1\\ 1 & -1\end{smallmatrix}\right)$ transforms the classical quantum state $|0\rangle$ into the state $(1/\sqrt{2})\binom{1}{1}$, which is in the perfect superposition between 0 and 1. Quantum states over n qubits are represented by a column vector of 2^n complex values α_x with $x \in 2^n$ such that $\sum|\alpha_i|^2 = 1$. Each squared amplitude $|\alpha_i|^2$ indicates the probability that after measurement the n qubits are in classical states x. Quantum states can be combined by applying the Kronecker product to produce larger ones, e.g., $\binom{1}{0} \otimes (1/\sqrt{2})\binom{1}{1} = (1/\sqrt{2})\left(\begin{smallmatrix}1\\1\\0\\0\end{smallmatrix}\right)$, which represents a 2-qubit state that is in the perfect superposition between the classical states 00 and 01. On the contrary, larger states cannot always be represented in terms of smaller ones. For example, there are no two independent qubit states $|\varphi_1\rangle$ and $|\varphi_2\rangle$ such that $|\varphi_1\rangle \otimes |\varphi_2\rangle = (1/\sqrt{2})\left(\begin{smallmatrix}1\\0\\0\\1\end{smallmatrix}\right)$, the state that is in the perfect superposition between the classical states 00 and 11. This phenomenon is called entanglement. Quantum gates that act on n qubits are represented in terms of $2^n \times 2^n$ unitary matrices. One frequently used two-input gate is the CNOT gate that inverts one qubit conditioned on the other qubit. Its 4×4 unitary matrix is $\left(\begin{smallmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{smallmatrix}\right)$.

Quantum algorithms describe problem solutions by manipulating quantum states using quantum operations. Algorithm 1 shows the pseudocode for a Grover search [100]. Given a Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ such that there exists exactly one $\hat{x}$ with $f(\hat{x}) = 1$, the algorithm finds $\hat{x}$ using only $O(\sqrt{2^n})$ evaluations of f. A classical computer cannot solve this problem in fewer than $O(2^n)$ evaluations of f.

Algorithm 1 Grover Search Algorithm

Input: Boolean function $f : \mathbb{B}^n \to \mathbb{B}$ such that exists exactly one $\hat{x}$ with $f(\hat{x}) = 1$
Output: $\hat{x}$ with high probability

```
1: Let |φ⟩ = |φ1⟩ ⊗ ... ⊗ |φn⟩
2: Set |φ⟩|ψ⟩ ← |0⟩^⊗n |1⟩
3: Set |φ⟩|ψ⟩ ← H^⊗n |φ⟩ ⊗ H|ψ⟩
4: for O(√(2^n)) times do
5:   Set |φ⟩|ψ⟩ ← |φ⟩|ψ ⊕ f(φ)⟩
6:   Set |φ⟩ ← D|φ⟩
7: end for
8: return Measure(|φ⟩)
```

The algorithm works on $n + 1$ qubits and it can easily be seen that the function f is only evaluated once in each iteration of the loop. The exact number of required iterations is $(\pi/4)\arcsin(1/\sqrt{2^n}) \approx \pi/(4\sqrt{2^n})$, therefore a polynomial speedup is achieved using a linear number of resources. We now explain the individual steps of the algorithm and explicitly describe the quantum state of the n-qubit register $|\varphi\rangle$, composed of 2^n amplitudes α_x for all $x \in \mathbb{B}^n$. The power of quantum computing is enabled by simultaneously acting on exponentially many amplitudes while applying an operation to a linear number of qubits.

- Line 2: Each qubit in $|\varphi\rangle$ is initialized to $|0\rangle$, i.e., $\alpha_{00\ldots0} = 1$, and $\alpha_x = 0$ for all $x \neq 00\ldots0$.
- Line 3: Apply the Hadamard gate to each qubit in $|\varphi\rangle$, resulting in $\alpha_x = 1/\sqrt{2^n}$ for all $x \in \mathbb{B}^n$. (At this point,

measuring $|\varphi\rangle$ corresponds to sampling a value $x \in \mathbb{B}^n$ uniform at random; and the probability of finding $\hat{x}$ is $1/2^n$.)

- Line 5: When applying f to $|\varphi\rangle$, the amplitude $\alpha_{\hat{x}}$ of the satisfying assignment is inverted and becomes $-\alpha_{\hat{x}}$; all other amplitudes remain unchanged. The additional qubit $|\psi\rangle$ is used here to apply f as a unitary operation. (Note that the sign inversion does not change the probabilities of measurement outcome.)
- Line 6: The operation D, called Grover diffusion operator, is described by the $2^n \times 2^n$ unitary matrix $D = H^{\otimes n} \otimes \mathrm{diag}(1, -1, -1, \ldots, -1) \otimes H^{\otimes n}$. Its effect on an amplitude is to reflect it with respect to the mean value $\mu = \sum_{x \in \mathbb{B}^x} \alpha_x$ of all amplitudes, i.e., the operation maps α_x to $2\mu - \alpha_x$. Since $\alpha_{\hat{x}}$ is negative, the reflection amplifies its absolute value such that after the operation $|\alpha_{\hat{x}}| > |\alpha_x|$ for all $x \in \mathbb{B}^x$. The difference grows in every iteration, as does the probability of obtaining $\hat{x}$ by measuring $|\varphi\rangle$.
- Line 8: After a sufficient number of iterations—and not more than that—the measurement of $|\varphi\rangle$ yields $\hat{x}$ with a very high probability. The probability of returning the wrong result is in $O(1/\sqrt{2^n})$. [Checking whether the returned result is correct is in $O(1)$, since it can be done by evaluating f classically once; if the result turns out to be wrong, the Grover algorithm is executed again.]

As an alternative representation to the textual one in Algorithm 1, one can use quantum circuits to express the interaction of quantum operations with qubits. Fig. 11 shows the quantum circuit representation of Algorithm 1. The boxes denote gates, e.g., the Hadamard operation H and the diffusion operator D. The gate in between expresses the application of f onto $|\varphi\rangle$. Instruction sequences for a quantum computer are also expressed in terms of quantum circuits, with the requirement that all quantum gates in the circuit correspond to actual physical operations that can be executed on the physical quantum computer [5]. Quantum compilers are software programs that take a high-level description of a quantum algorithm and map them into low-level quantum circuits. The objective of quantum compilers is to find a quantum circuit that meets the number of available qubits and minimizes the number of quantum gates. Fig. 12 shows a compiled and optimized quantum circuit for an instance of the Grover algorithm in which $n = 2$ and $f = x_1 \wedge x_2$, for which $\hat{x} = 11$. The gate set that is used in this quantum computation is called Clifford+T [and generated by the matrices H, $T = \mathrm{diag}(1, e^{\mathrm{i}\pi/4})$, and CNOT] and is, e.g., supported by the IBM and Google superconducting quantum computers [103], [104].

2) Constraints and Costs: Logic synthesis can help to automatically translate high-level quantum operations that appear in quantum algorithms or high-level quantum circuit descriptions into the low-level quantum operations supported by a quantum computer. However, quantum circuits differ significantly in comparison to classical circuits, which needs to be addressed by design automation tools [105].

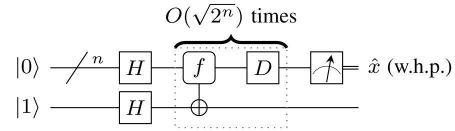


Fig. 11. ***Quantum circuit representation of Algorithm 1.***

Although qubits may be in superposition or entangled, when targeting purely Boolean functions, it is sufficient to assume that all input values are Boolean inside the synthesis algorithm, even though entangled qubits in superposition are eventually acted upon by the quantum hardware.

Since quantum gates are described in terms of unitary matrices, also classical Boolean functions must be implemented in terms of unitary operations. A direct translation is not always possible. For example, in the Grover algorithm, an additional qubit, called ancilla, is necessary in order to apply the function f to the quantum state. For Boolean functions and algorithms, which are more involved, the problem of determining how many ancillae are necessary is not a simple task. Qubits are a limited resource; therefore, the use of ancillae is restricted and synthesis must find circuits that satisfy the number of available qubits.

3) Logic Synthesis Algorithms: Quantum compilation requires several different steps, such as decomposing arbitrary unitary operations into a given gate set [106]–[108], optimizing low-level quantum circuits [109]–[112], mapping quantum circuits while respecting architectural constraints [113], [114], and applying error correction [115], [116]. In the remainder, we focus on the task of translating classical combinational operations, such as the operation f in the Grover algorithm, into quantum circuits.

The translation of classical combinational operations into quantum circuits involves reversible logic synthesis [117]. State-of-the-art approaches first create a reversible logic circuit with reversible gates, which are Boolean abstractions of classical reversible operations. Other methods translate reversible gates into quantum circuits [9], [118], [119]. Many approaches for reversible logic synthesis have been proposed in the last 15 years (e.g., [120]–[123]). Initial algorithms are based on explicit truth table or permutation-based representations [120], [124], [125]. All algorithms guarantee quantum circuits with the minimum number of qubits, because the input function is already reversible. But since truth tables grow

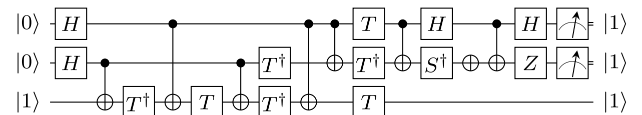


Fig. 12. ***Compiled and optimized instance of the Grover algorithm for $n = 2$ and $f = x_1 \wedge x_2$.***

exponentially in size, the algorithms require exponential space and runtime in the number of input variables. This rules out their applicability to problem sizes of 20 qubits or more. Symbolic implementations of theses algorithms (see, e.g., [123], [126], and [127]) can overcome these limitations for functions that have a compact symbolic representation (e.g., in terms of a BDD) and result in reversible circuits with a few number of gates.

By using data structures that allow a more compact representation for the input function, and by relaxing the constraint that the input function is reversible, more scalable algorithms can be achieved, with the cost of requiring additional qubits in their implementation. If it is possible to find a two-level ESOP representation for a Boolean function with n inputs and m outputs (i.e., m ESOP representations), ESOP-based reversible logic synthesis [33], [34] can find a reversible circuit using $m + n$ qubits, which requires as many reversible gates as product terms in the ESOP expressions. Finding a two-level ESOP representation for a function is not always simple, but recent approaches show that it is possible to find representations for functions with up to 32 variables in a reasonable amount of runtime [128].

Scalable reversible logic synthesis can be achieved when using a multilevel logic representation for the input function. In so-called hierarchical reversible logic synthesis, a reversible or quantum circuit is determined for substructures in the multilevel representation, e.g., a gate in a logic network, or a node in a BDD. The resulting circuits are composed by using additional ancillae qubits to store temporary results. Proposed approaches make use of decision diagrams [122], [129], logic networks in general [130], AIGs [131], XMGs [132], and most recently LUT networks [105]. Hierarchical reversible logic synthesis is as scalable as classical logic synthesis, since it depends on the representation size of the input function. However, this scalability comes at the cost of a significant number of additional qubits—often far more than current and near-term quantum computers support. Strategies based on the reversible pebble game [133] can be used to reduce the number of qubits for the cost of more gates [134].

VI. CONCLUSION

The plurality of post-CMOS nanoemerging technologies enforced a reorientation of classical logic synthesis methods. Different logic primitives are addressed by new logic representations, such as majority-inverter graphs. Exact methods guarantee valid solutions even in the presence of various constraints, which need to be taken into consideration especially for very recent emerging technologies.

Conventional CMOS technologies, such as FPGAs and ASICs, can benefit largely from modern logic synthesis data structures and techniques. If new data structures are used as intermediate representations, logic synthesis algorithms can exploit more optimization opportunities which results in significant reductions after technology mapping. Furthermore, the increase of compute power in combination with the advances in SAT solvers enable the use of SAT-based logic synthesis techniques in a scalable manner. Exact synthesis can find optimum logic networks for various cost functions and constraints, thus being appealing for emerging technologies. Decomposition techniques allow us to apply exact synthesis locally to large logic networks.

In this paper, we have shown a selection of modern data structures, algorithms, and applications. The common principle is to consider logic synthesis as a technology-independent tool that can—efficiently and effectively—express functionality in terms of small logic primitives, and optimize the representation in terms of an application and technology-dependent cost function. This general view makes logic synthesis a key ingredient to numerous computing platforms. ■

Acknowledgments

The authors would like to thank A. Mishchenko, R. Brayton, P.-E. Gaillardon, and W. Haaswijk for fruitful discussions.

REFERENCES

[1] (2018). *Synopsys Design Compiler Graphical*. [Online]. Available: https://www.synopsys.com/content/dam/synopsys/implementation&signoff/datasheets/dc-graphical-ds.pdf
[2] (2018). *Cadence Genus Synthesis Solution*. [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_us/documents/tools/digital-design-signoff/genus-synthesis-solution-ds.pdf
[3] (2018). *Mentor Graphics Oasys-RTL*. [Online]. Available: http://s3.mentor.com/public_documents/datasheet/products/ic_nanometer_design/place-route/realtime-designer/realttime-designer.pdf
[4] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. Int. Symp. Field Program. Gate Arrays*, 2015, pp. 161–170.
[5] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017.
[6] C. S. Lent, P. D. Tougaw, W. Porod, and G. H. Bernstein, "Quantum cellular automata," *Nanotechnology*, vol. 4, no. 1, pp. 49–57, 1993.
[7] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*. Cambridge, U.K.: Cambridge Univ. Press, 2000.
[8] T. Toffoli, "Reversible computing," in *Proc. Int. Coll. Automat., Lang., Program.*, 1980, pp. 632–644.
[9] D. Maslov, "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Phys. Rev. A, Gen. Phys.*, vol. 93, p. 022311, Feb. 2016.
[10] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A new paradigm for logic optimization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 5, pp. 806–819, May 2016.
[11] L. Amaru, P. E. Gaillardon, S. Mitra, and G. De Micheli, "New logic synthesis as nanotechnology enabler," *Proc. IEEE*, vol. 103, no. 11, pp. 2168–2195, Nov. 2015.
[12] G. De Micheli, *Synthesis and Optimization of Digital Circuits*. New York, NY, USA: McGraw-Hill, 1994.
[13] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, vol. 2. Springer, 1984.
[14] G. D. Hachtel and F. Somenzi, *Logic Synthesis and Verification Algorithms*. Springer, 2006.
[15] S. Hassoun and T. Sasao, *Logic Synthesis and Verification*, vol. 654. Springer, 2012.
[16] E. J. McCluskey, "Minimization of Boolean functions," *Bell Syst. Tech. J., The*, vol. 35, no. 6, pp. 1417–1444, Nov. 1956.
[17] R. Rudell and A. Sangiovanni-Vincentelli,"Logic synthesis for VLSI design," Ph.D. dissertation, Univ. California, Berkeley, Berkeley, CA, USA, 1989.
[18] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, MA, USA: Addion-Wesley, 1993, ch. 8.
[19] T. Sasao, *Switching Theory for Logic Synthesis*. Springer, 1999.
[20] N. Song and M. A. Perkowski, "Minimization of exclusive sum-of-products expressions for

multiple-valued input, incompletely specified functions," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 15, no. 4, pp. 385–395, Apr. 1996.

[21] D. E. Knuth, *The Art of Computer Programming*, vol. 4A. Reading, MA, USA: Addison-Wesley, 2011.

[22] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, "Finding efficient circuits using SAT-solvers," in *Proc. Int. Conf. Theory Appl. Satisfiability Testing*, 2009, pp. 32–44.

[23] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 11, pp. 1842–1855, Nov. 2017.

[24] R. Drechsler and W. Günther, "Exact circuit synthesis," in *Proc. Int. Workshop Logic Synth.*, 1998.

[25] L. G. Amarù *et al.*, "Enabling exact delay synthesis," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 2017, pp. 352–359.

[26] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Proc. Design, Automat. Test Eur.*, Mar. 2017, pp. 830–835.

[27] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam, The Netherlands: IOS Press, 2009.

[28] D. E. Knuth, *The Art of Computer Programming, Fascicle 6: Satisfiability*, vol. 4. Reading, MA, USA: Addison-Wesley, 2015.

[29] F. M. Brown, *Boolean Reasoning: The Logic of Boolean Equations*. Springer, 2012.

[30] J. Boyar and R. Peralta, "A new combinational logic minimization technique with applications to cryptology," in *Proc. Int. Symp. Exp. Algorithms*, 2010, pp. 178–189.

[31] J. Boyar and R. Peralta, "A small depth-16 circuit for the AES S-box," in *Proc. Inf. Secur. Privacy Conf.*, 2012, pp. 287–298.

[32] D. Canright and L. Batina, "A very compact 'perfectly masked' S-box for AES," in *Proc. Int. Conf. Appl. Cryptogr. Netw. Secur.*, 2008, pp. 446–459.

[33] K. Fazel, M. A. Thornton, and J. E. Rice, "ESOP-based Toffoli gate cascade generation," in *Proc. Pacific Rim Conf. Commun., Comput. Signal Process.*, Aug. 2007, pp. 206–209.

[34] A. Mishchenko and M. Perkowski, "Logic synthesis of reversible wave cascades," in *Proc. Int. Workshop Logic Synth.*, 2002.

[35] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Proc. Int. Conf. Theory Appl. Satisfiability Test.*, 2009, pp. 244–257.

[36] G. S. Tseitin, "On the complexity of derivation in propositional calculus," in *Studies in Constructive Mathematics and Mathematical Logic Part 2* (Seminars in Mathematics), A. P. Slisenko, Ed. Springer, 1970, pp. 115–125.

[37] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 677–691, Aug. 1986.

[38] B. Bollig and I. Wegener, "Improving the variable ordering of OBDDs is NP-complete," *IEEE Trans. Comput.*, vol. 45, no. 9, pp. 993–1002, Sep. 1996.

[39] R. Drechsler, N. Drechsler, and W. Gunther, "Fast exact minimization of BDD's," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 19, no. 3, pp. 384–389, Mar. 2000.

[40] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," in *Proc. Conf. Eur. Design Automat.*, Feb. 1991, pp. 50–54.

[41] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, "Robust Boolean reasoning for equivalence checking and functional property verification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 21, no. 12, pp. 1377–1394, Dec. 2002.

[42] L. Hellerman, "A catalog of three-variable or-invert and and-invert logical circuits," *IEEE Trans. Electron. Comput.*, vol. EC-12, no. 3, pp. 198–223, Jun. 1963.

[43] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Proc. Design Automat. Conf.*, Jun. 2014, p. 194:1–194:6.

[44] W. Haaswijk, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic optimization," in *Proc. Asia South Pacific Design Automat. Conf.*, 2017, pp. 151–156.

[45] J. A. Darringer, W. H. Joyner, C. L. Berman, and L. Trevillyan, "Logic synthesis through local transformations," *IBM J. Res. Develop.*, vol. 25, no. 4, pp. 272–280, Jul. 1981.

[46] A. J. de Geus and D. J. Gregory, "The Socrates logic synthesis and optimization system," in *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation*, vol. 136, G. De Micheli, A. Sangiovanni-Vincentelli, and P. Antognetti, Eds. 1987, p. 473.

[47] W. Haaswijk *et al.*, "Deep learning for logic optimization algorithms," in *Proc. Int. Symp. Circuits Syst.*, May 2018, pp. 1–4.

[48] C. Yu, H. Xiao, and G. De Micheli, "Developing synthesis flows without human knowledge," in *Proc. Design Automat. Conf.*, 2018, Art. no. 50.

[49] R. K. Brayton and C. T. McMullen, "The decomposition and factorization of Boolean expressions," in *Proc. Int. Symp. Circuits Syst.*, 1982, pp. 49–54.

[50] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: A multiple-level logic optimization system," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 6, no. 6, pp. 1062–1081, Nov. 1987.

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

[52] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis," in *Proc. Design Automat. Conf.*, Jul. 2006, pp. 532–535.

[53] A. Mishchenko and R. K. Brayton, "Scalable logic synthesis using a simple circuit structure," in *Proc. Int. Workshop Logic Synth.*, 2006, pp. 15–22.

[54] L. G. Amarù, P.-E. Gaillardon, A. Chattopadhyay, and G. De Micheli, "A sound and complete axiomatization of majority-n logic," *IEEE Trans. Comput.*, vol. 65, no. 9, pp. 2889–2895, Sep. 2016.

[55] E. Testa, M. Soeken, L. Amarù, W. Haaswijk, and G. D. Micheli, "Mapping monotone boolean functions into majority," 2018.

[56] K. A. Bartlett *et al.*, "Multi-level logic minimization using implicit don't cares," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 7, no. 6, pp. 723–740, Jun. 1988.

[57] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 25, no. 5, pp. 743–755, May 2006.

[58] S. Muroga, "Logic synthesizers, the transduction method and its extension, Sylon," in *Logic Synthesis and Optimization*, T. Sasao, Ed. Springer, 1993, pp. 59–86.

[59] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The transduction method-design of logic networks based on permissible functions," *IEEE Trans. Comput.*, vol. 38, no. 10, pp. 1404–1424, Oct. 1989.

[60] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 21, no. 7, pp. 866–876, Jul. 2002.

[61] F. Mailhot and G. Di Micheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 12, no. 5, pp. 599–620, May 1993.

[62] P. Vuillod, L. Benini, and G. De Micheli, "Generalized matching from theory to application," in *Proc. Int. Conf. Comput.-Aided Design*, Nov. 1997, pp. 13–20.

[63] A. Mishchenko and R. K. Brayton, "SAT-based complete don't-care computation for network optimization," in *Proc. Design, Automat. Test Eur.*, Mar. 2005, pp. 412–417.

[64] F. Brglez, D. Bryan, J. Calhoun, G. Kedem, and R. Lisanke, "Automated synthesis for testability," *IEEE Trans. Ind. Electron.*, vol. 36, no. 2, pp. 263–277, May 1989.

[65] F. D. Bryan, Brglez and R. Lisanke, "Redundancy identification and removal," in *Proc. IWLS*, 1991.

[66] S.-C. Chang, L. P. P. P. V. Ginneken, and M. Marek-Sadowska, "Circuit optimization by rewiring," *IEEE Trans. Comput.*, vol. 48, no. 9, pp. 962–970, Sep. 1999.

[67] M. Damiani, J. C. Y. Yang, and G. D. Micheli, "Optimization of combinational logic circuits based on compatible gates," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 14, no. 11, pp. 1316–1327, Nov. 1995.

[68] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proc. Int. Workshop Logic Synth.*, 2015.

[69] J. Cong and Y. Ding, "FlowMap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 13, no. 1, pp. 1–12, Jan. 1994.

[70] E. A. Ernst, "Optimal combinational multi-level logic synthesis," Ph.D. dissertation, Dept. Comput. Sci. Eng., Univ. Michigan, Ann Arbor, MI,USA, 2009.

[71] N. Éen, "Practical SAT—A tutorial on applied satisfiability solving," FMCAD, Tech. Rep., 2007.

[72] V. Bertacco and M. Damiani, "Boolean function representation based on disjoint-support decompositions," in *Proc. Int. Conf. Comput. Design*, Oct. 1996, pp. 27–32.

[73] L. G. Amarù, P. Vuillod, J. Luo, and J. Olson, "Logic optimization and synthesis: Trends and directions in industry," in *Proc. Design, Automat. Test Eur.*, Mar. 2017, pp. 1303–1305.

[74] J. Cong, C. Wu, and Y. Ding, "Cut ranking and pruning: Enabling a general and efficient FPGA mapping solution," in *Proc. Int. Symp. Field Program. Gate Arrays*, 1999, pp. 29–35.

[75] S. Ray, A. Mishchenko, N. Eén, R. K. Brayton, S. Jang, and C. Chen, "Mapping into LUT structures," in *Proc. Design, Automat. Test Eur.*, 2012, pp. 1579–1584.

[76] A. Mishchenko, A. S. Chatterjee, and R. K. Brayton, "Improvements to technology mapping for LUT-based FPGAs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 2, pp. 240–253, Feb. 2007.

[77] O. Zografos, L. Amaru, P.-E. Gaillardon, P. Raghavan, and G. De Micheli, "Majority logic synthesis for spin wave technology," in *Proc. Euromicro Conf. Digit. Syst. Design*, Aug. 2014, pp. 691–694.

[78] R. Zhang, P. Gupta, and N. K. Jha, "Majority and minority network synthesis with application to QCA-, SET-, and TPL-based nanotechnologies," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 7, pp. 1233–1245, Jul. 2007.

[79] E. Testa *et al.*, "Inverter propagation and fan-out constraints for beyond-CMOS majority-based technologies," in *Proc. Annu. Symp. VLSI*, Jul. 2017, pp. 164–169.

[80] R. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Computer Aided Verification*. 2010, pp. 24–40.

[81] A. Mishchenko, R. Brayton, T. Besson, S. Govindarajan, H. Arts, and P. van Besouw, "Versatile SAT-based remapping for standard cells," in *Proc. Int. Workshop Logic Synth.*, 2016, pp. 1–5.

[82] B. Schmitt, A. Mishchenko, and R. K. Brayton, "SAT-based area recovery in technology mapping," in *Proc. Asia South Pacific Design Automat. Conf.*, 2018, pp. 1–6.

[83] G. Csaba, A. Imre, G. H. Bernstein, W. Porod, and V. Metlushko, "Nanocomputing by field-coupled nanomagnets," *IEEE Trans. Nanotechnol.*, vol. 99, no. 4, pp. 209–213, 2002.

[84] O. Zografos *et al.*, "Design and benchmarking of hybrid CMOS-spin wave device circuits compared

to 10 nm CMOS," in *Proc. Int. Conf. Nanotechnol.*, Jul. 2015, pp. 686–689.

[85] D. E. Nikonov, G. I. Bourianoff, and T. Ghani, "Proposal of a spin torque majority gate logic," *IEEE Electron Device Lett.*, vol. 32, no. 8, pp. 1128–1130, Aug. 2011.

[86] S. Dutta *et al.*, "Proposal for nanoscale cascaded plasmonic majority gates for non-Boolean computation," *Sci. Rep.*, vol. 7, no. 1, p. 17866, 2017.

[87] W. L. Barnes, A. Dereux, and T. W. Ebbesen, "Surface plasmon subwavelength optics," *Nature*, vol. 424, no. 6950, pp. 824–830, 2003.

[88] E. Testa, M. Soeken, O. Zografos, F. Catthoor, and G. De Micheli, "Exact synthesis for logic synthesis applications with complex constraints," in *Proc. Int. Workshop Logic Synth.*, 2017.

[89] C. S. Lent and P. D. Tougaw, "A device architecture for computing with quantum dots," *Proc. IEEE*, vol. 85, no. 4, pp. 541–557, Apr. 1997.

[90] P. D. Tougaw and C. S. Lent, "Logical devices implemented using quantum cellular automata," *Appl. Phys. Lett.*, vol. 75, no. 3, pp. 1818–1825, 1993.

[91] K. Navi, R. Farazkish, S. Sayedsalehi, and M. R. Azghadi, "A new quantum-dot cellular automata full-adder," *Microelectron. J.*, vol. 41, no. 12, pp. 820–826, 2010.

[92] S. Sheikhfaal, S. Angizi, S. Sarmadi, M. H. Moaiyeri, and S. Sayedsalehi, "Designing efficient QCA logical circuits with power dissipation analysis," *Microelectron. J.*, vol. 46, no. 6, pp. 462–471, Jun. 2015.

[93] T. N. Sasamal, A. K. Singh, and A. Mohan, "An optimal design of full adder based on 5-input majority gate in coplanar quantum-dot cellular automata," *Optik*, vol. 127, no. 20, pp. 8576–8591, Oct. 2016.

[94] K. Walus, T. J. Dysart, G. A. Jullien, and R. A. Budiman, "QCADesigner: A rapid design and simulation tool for quantum-dot cellular automata," *IEEE Trans. Nanotechnol.*, vol. 3, no. 1, pp. 26–31, Mar. 2004.

[95] E. Testa *et al.*, "Inversion optimization in majority-inverter graphs," in *Proc. Int. Symp. Nanosc. Architect.*, Jul. 2016, pp. 15–20.

[96] S. Amarel, G. E. Cooke, and R. O. Winder, "Majority gate networks," *IEEE Trans. Electron. Comput.*, vol. 13, no. 1, pp. 4–13, 1964.

[97] A. Chattopadhyay, L. G. Amarù, M. Soeken, P.-E. Gaillardon, and G. De Micheli, "Notes on majority Boolean algebra," in *Proc. Int. Symp. Multiple-Valued Logic*, May 2016, pp. 50–55.

[98] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, pp. 1484–1509, 1997.

[99] C. Pomerance, "A tale of two sieves," *Notices AMS*, vol. 43, no. 12, pp. 1473–1485, 1996.

[100] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proc. Symp. Theory Comput.*, 1996, pp. 212–219.

[101] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Phys. Rev. Lett.*, vol. 103, no. 15, p. 150502, Oct. 2009.

[102] T. H. Johnson, S. R. Clark, and D. Jaksch, "What is a quantum simulator?" *EPJ Quantum Technol.*, vol. 1, no. 10, pp. 1–12, 2014.

[103] N. M. Linke *et al.*, "Experimental comparison of two quantum computing architectures," *Proc. Nat. Acad. Sci. USA*, vol. 114, no. 13, pp. 3305–3310, 2017.

[104] I. L. Markov, A. Fatima, S. V. Isakov, and S. Boixo (2018). "Quantum supremacy is both closer and farther than it appears." [Online]. Available: https://arxiv.org/abs/1807.10749

[105] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Hierarchical reversible logic synthesis using LUTs," in *Proc. Design Automat. Conf.*, 2017, pp. 78:1–78:6.

[106] V. Kliuchnikov and J. Yard (2015). "A framework for exact synthesis." [Online]. Available: https://arxiv.org/abs/1504.04350

[107] S. Forest, D. Gosset, V. Kliuchnikov, and D. McKinnon, "Exact synthesis of single-qubit unitaries over Clifford-cyclotomic gate sets," *J. Math. Phys.*, vol. 56, no. 8, p. 082201, 2015.

[108] A. Bocharov, Y. Gurevich, and K. M. Svore, "Efficient decomposition of single-qubit gates into V basis circuits," *Phys. Rev. A, Gen. Phys.*, vol. 88, Jul. 2013, Art. no. 012313.

[109] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *Quantum Inf.*, vol. 4, no. 23, pp. 1–12, 2018.

[110] D. Maslov, "Basic circuit compilation techniques for an ion-trap quantum machine," *Quantum Inf.*, vol. 19, Feb. 2017, Art. no. 023035.

[111] L. Heyfron and E. T. Campbell (2018). "An efficient quantum compiler that reduces T count." [Online]. Available: https://arxiv.org/abs/1712.01557

[112] M. Amy, D. Maslov, and M. Mosca, "Polynomial-time T-depth optimization of cliffordT circuits via matroid partitioning," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 10, pp. 1476–1489, Oct. 2014.

[113] A. Zulehner, A. Paler, and R. Wille (2018). "An efficient methodology for mapping quantum circuits to the IBM QX architectures." [Online]. Available: https://arxiv.org/abs/1712.04722

[114] L. Lao *et al.* (2018). "Mapping of lattice surgery-based quantum circuits on surface code architectures." [Online]. Available: https://arxiv.org/abs/1805.11127

[115] C. Horsman, A. G. Fowler, S. Devitt, and R. Van Meter, "Surface code quantum computing by lattice surgery," *Quantum Inf.*, vol. 14, pp. 1–27, Dec. 2012.

[116] R. Chao and B. W. Reichardt (2017). "Fault-tolerant quantum computation with few qubits." [Online]. Available: https://arxiv.org/abs/1705.05365

[117] M. Saeedi and I. L. Markov, "Synthesis and optimization of reversible circuits—A survey," *ACM Comput. Surveys*, vol. 45, no. 2, pp. 21:1–21:34, 2013.

[118] A. Barenco *et al.*, "Elementary gates for quantum computation," *Phys. Rev. A, Gen. Phys.*, vol. 52, no. 5, p. 3457, 1995.

[119] N. Abdessaied, M. Amy, M. Soeken, and R. Drechsler, "Technology mapping of reversible circuits to cliffordT quantum circuits," in *Proc. Int. Symp. Multiple-Valued Logic*, May 2016, pp. 150–155.

[120] D. M. Miller, D. Maslov, and G. W. Dueck, "A transformation based algorithm for reversible logic synthesis," in *Proc. Design Automat. Conf.*, Jun. 2003, pp. 318–323.

[121] V. V. Shende, A. K. Prasad, I. L. Markov, and J. P. Hayes, "Synthesis of reversible logic circuits," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 22, no. 6, pp. 710–722, Jun. 2003.

[122] R. Wille and R. Drechsler, "BDD-based synthesis of reversible logic for large functions," in *Proc. Des. Automat. Conf.*, Jul. 2009, pp. 270–275.

[123] M. Soeken, R. Wille, C. Hilken, N. Przigoda, and R. Drechsler, "Synthesis of reversible circuits with minimal lines for large functions," in *Proc. Asia South Pacific Des. Automat. Conf.*, Jan./Feb. 2012, pp. 85–92.

[124] A. De Vos and Y. Van Rentergem, "Young subgroups for reversible computers," *Adv. Math. Commun.*, vol. 2, no. 2, pp. 183–200, 2008.

[125] M. Saeedi, M. S. Zamani, M. Sedighi, and Z. Sasanian, "Reversible circuit synthesis using a cycle-based approach," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 6, no. 4, p. 13, 2010.

[126] M. Soeken, L. Tague, G. W. Dueck, and R. Drechsler, "Ancilla-free synthesis of large reversible functions using binary decision diagrams," *J. Symbolic Comput.*, vol. 73, pp. 1–26, Mar./Apr. 2016.

[127] M. Soeken, G. W. Dueck, and D. M. Miller, "A fast symbolic transformation based algorithm for reversible logic synthesis," in *Proc. Int. Conf. Reversible Comput.*, 2016, pp. 307–321.

[128] B. Schmitt, M. Soeken, A. Mishchenko, and G. D. Micheli, "Scaling up collapsing into ESOP expressions enables ancilla-free quantum compilation," 2018.

[129] M. Soeken, R. Wille, and R. Drechsler, "Hierarchical synthesis of reversible circuits using positive and negative Davio decomposition," in *Proc. Int. Design Test Symp.*, 2010, pp. 143–148.

[130] M. Rawski, "Application of functional decomposition in synthesis of reversible circuits," in *Proc. Int. Conf. Reversible Comput.*, 2015, pp. 285–290.

[131] M. Soeken and A. Chattopadhyay, "Unlocking efficiency and scalability of reversible logic synthesis using conventional logic synthesis," in *Proc. Design Automat. Conf.*, Jun. 2016, pp. 149:1–149:6.

[132] M. Soeken, M. Roetteler, N. Wiebe, and G. De Micheli, "Design automation and design space exploration for quantum computers," in *Proc. Design, Automat. Test Eur.*, 2017, pp. 470–475.

[133] C. H. Bennett, "Time/space trade-offs for reversible computation," *SIAM J. Comput.*, vol. 18, no. 4, pp. 766–776, 1989.

[134] A. Parent, M. Roetteler, and K. M. Svore, "REVS: A tool for space-optimized reversible circuit synthesis," in *Proc. Int. Conf. Reversible Comput.*, 2017, pp. 90–101.

ABOUT THE AUTHORS

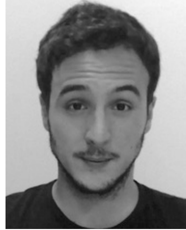
Eleonora Testa (Student Member, IEEE) received the B.Sc. degree in physical engineering from the Politecnico di Torino, Turin, Italy, in 2013 and the joint M.Sc. degree in micro and nanotechnologies for integrated systems from Politecnico di Torino (Italy), Grenoble INP (France), and EPFL (Switzerland) in 2015. Currently, she is working toward the Ph.D. degree at the Integrated Systems Laboratory at EPFL, Switzerland.

Her research interests include logic synthesis, electronic design automation, and post-CMOS technologies.

Mathias Soeken (Member, IEEE) received the Ph.D. degree in computer science and engineering from the University of Bremen, Bremen, Germany, in 2013.

He is a Scientist at the École Polytechnique Fédéderale de Lausanne (EPFL), Lausanne, Switzerland. His current research interests include the many aspects of logic synthesis and formal verification. He investigates constraint-based techniques in logic synthesis and industrial-strength design automation for quantum computing. He is actively maintaining the logic synthesis frameworks CirKit and RevKit.

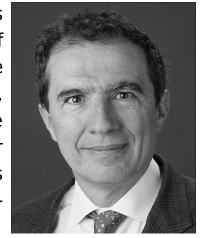
Dr. Soeken received a scholarship from the German Academic Scholarship Foundation. He has been serving as TPC member for several conferences, including DAC, DATE, and ICCAD and is a reviewer for *Mathematical Reviews* as well as for several journals.



Luca Gaetano Amarù (Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from the Politecnico di Torino, Turin, Italy, in 2009 and 2011, respectively, and the Ph.D. degree in computer science from the Swiss Federal Institute of Technology Lausanne, Lausanne, Switzerland, in 2015.

He is a Staff R&D Engineer in the Design Group, Synopsys Inc., Mountain View, CA, USA, where he is responsible for designing efficient data structures and algorithms for logic synthesis. Prior to joining Synopsys, he was a visiting researcher at Stanford University and research assistant at EPFL. His current research interests include electronic design automation, logic in computer science, and beyond CMOS technologies.

Dr. Amaru was a recipient of the IEEE TCAD Donald O. Pederson Best Paper Award in 2018, the EDAA Outstanding Dissertation Award in 2015, the Best Presentation Award at FETCH conference in 2013, and a Best Paper Award Nomination at ASP-DAC conference in 2013. He received fellowships and research contribution awards at EPFL. He has been serving as TPC member for several conferences, including DATE, IWLS and DSD. He is a reviewer for several IEEE journals.



Giovanni De Micheli (Fellow, IEEE) is Professor and Director of the Institute of Electrical Engineering, École Polytechnique Fédéderale de Lausanne (EPFL), Lausanne, Switzerland. His research interests include several aspects of design technologies for integrated circuits and systems, such as synthesis for emerging technologies, networks on chips, and 3-D integration.

Prof. De Micheli is a Fellow of the Association for Computing Machinery (ACM) and a member of the Academia Europaea and an International Honorary member of the American Academy of Arts and Sciences. He is the recipient of the 2016 IEEE/CS Harry Goode award for seminal contributions to design and design tools of networks on chips, the 2016 EDAA Lifetime Achievement Award, the 2012 IEEE/CAS Mac Van Valkenburg award for contributions to theory, practice, and experimentation in design methods and tools, the 2003 IEEE Emanuel Piore Award for contributions to computer-aided synthesis of digital systems, and the D. Pederson Award for the best paper in the IEEE Transactions on Computer-Aided Design and ICAS in 1987 and 2018.