Compilation flow for classically defined quantum operations

Bruno Schmitt[1,2], Ali Javadi-Abhari[3], Giovanni De Micheli[1]
[1]*Integrated Systems Laboratory (LSI), EPFL, Switzerland*
[2]*IBM Quantum, IBM Research, Switzerland*
[3]*IBM Quantum, IBM T.J. Watson Research Center, United States*

***Abstract*—We present a flow for synthesizing quantum operations that are defined by classical combinational functions. The discussion will focus on out-of-place computation, i.e., $U_f : |x\rangle|y\rangle|0\rangle^k \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^k$. Our flow allows users to express this function at a high level of abstraction. At its core, there is an improved version of the current state-of-the-art algorithm for synthesizing oracles [1]. As a result, our synthesized circuits use up to 25% fewer qubits and up to 43% fewer Clifford gates. Crucially, these improvements are possible without increasing the number of T gates nor the execution time.**

***Index Terms*—quantum, design automation, compilation, synthesis**

I. Introduction

In this paper, we present an automatic compilation flow for classically defined quantum operations, i.e., operations with the behavior described by a classical function $f : \{0,1\}^n \mapsto \{0,1\}^m$. Many quantum algorithms make use of such operations [2]–[4] and in particular oracular algorithms [5]–[10], since their oracles are often classically defined.

We must implement these oracles in terms of elementary quantum operations to run on a quantum computer—very much like classical computer programs need to be expressed in terms of low-level machine instructions. Furthermore, due to the physical properties of quantum states, all quantum operations need to be reversible. Therefore, a classical function that defines the behavior of a quantum operation must be reversible.

Most often, however, real-world problems require the use of oracles defined by functions that are both complex and irreversible. Demanding an algorithm designer to deal with all these constraints is impractical. Our work allows designers to implement the complicated classical subroutines on a high level of abstraction, and then automatically translate such implementations into low-level quantum circuits.

Contributions: We present an automatic compilation flow for classically defined quantum operations. We present its core algorithm: An improved version of the current state-of-the-art algorithm [1] for oracle synthesis. We present results demonstrating that our flow is capable of reducing the number of qubits by up to 24.95% and the number of Clifford gates by up to 43.3% when compared to [1]. Crucially, these improvements are possible without increasing the number of T gates nor the execution time. Also, we explain why the authors' implementation of Algorithm 1 in [1] does not properly handle the general case $U_f : |x\rangle|y\rangle|0\rangle^k \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^k$.

This research was partially supported by the European Research Council in the project H2020-ERC-2014-ADG 669354 CyberCare.

II. Preliminaries

As general notation, we are using $[n] = \{1, \ldots, n\}$.

A. Logic networks

Let $S = \{i_1, \ldots, i_k\} \subseteq [n]$, then

$$F_S(x) = f(x_{i_1}, \ldots, x_{i_k}) \quad (1)$$

is a Boolean function over the variables indexed by S. We define $F_\emptyset = 0$. For convenience we omit brackets when explicitly writing the indexes, e.g., we write $F_{1,3}$ instead of $F_{\{1,3\}}$.

We model a logic network for a Boolean function over the variables $x_n, \ldots, x_1$ as a sequence of r steps, where each step has the form:

$$x_i = F_{S_i}(x) \quad (2)$$

for $n < i \leq n + r$, where $S_i \subseteq [i-1]$. We call F_{S_i} a local function. To represent the constant function, we define $x_0 = 0$. We define a sequence of primary outputs $y_0, \ldots, y_m$ in which each element points to a step x_j with $0 \leq j \leq n + r$.

B. Quantum Computing

Quantum computers consist of an array of quantum bits, the so-called qubits, whose state can be modified by quantum operations, typically referred to as quantum gates. A quantum algorithm describes how to transform the state of a quantum computer to solve a computational task. Such an algorithm contains both classical operations and quantum operations, and its execution takes place on a classical host that decides on the sequence of quantum operations to send to the quantum co-processor. We can depict this sequence as a quantum circuit. Fig. 1c shows a quantum circuit diagram. This diagram is read from left to right with each horizontal line representing a qubit, and quantum gates represented as boxes/symbols on these lines.

A classically defined quantum operation is a 'black box' unitary operation U_f. Without loss of generality, we will restrict our discussion to operations described using a classical function $f : \{0,1\}^n \mapsto \{0,1\}^m$. We define the effect of operation on all computational basis state by:

$$U_f : |x\rangle|y\rangle|0\rangle^k \mapsto |x\rangle|y \oplus f(x)\rangle|0\rangle^k, \quad (3)$$

where the extra k qubits are used to store intermediate results for the computation of $f(x)$, the so-called ancillae qubits. Since, like all quantum operations, U_f is linear in the state it acts on, defining an operation in this way, i.e., for each computational basis state $|x\rangle|y\rangle$, also defines how U_f acts for any other state.

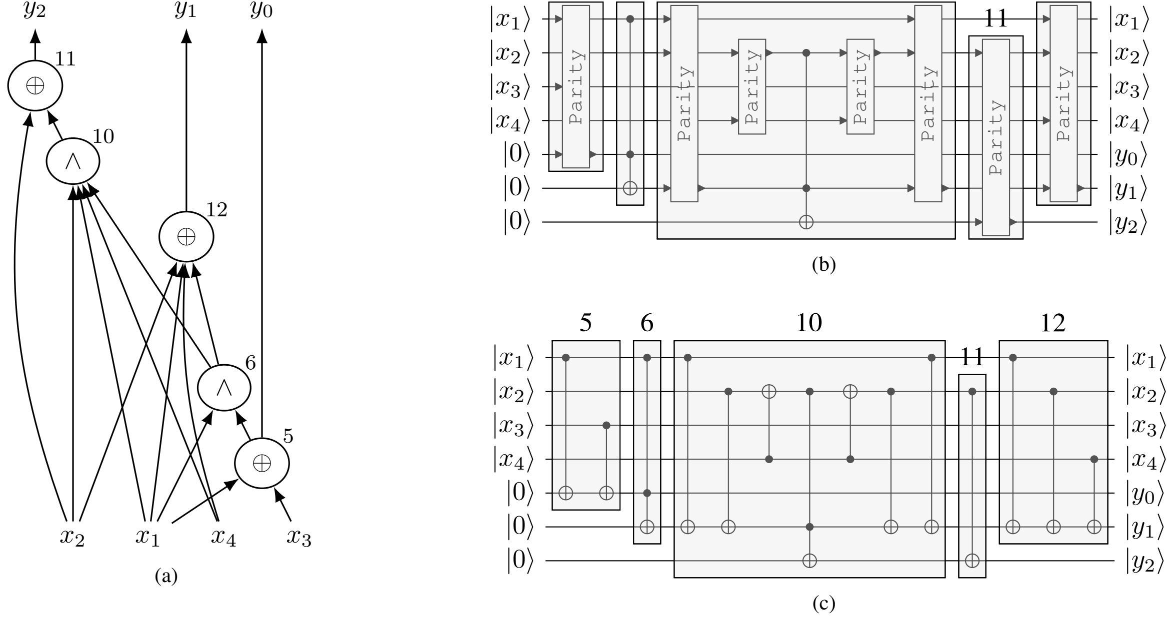


Fig. 1. Starting from the XAG, we first create a high-level XAG (a). We synthesize a quantum circuit with parity gates and Toffoli gates (b); Then we lower it to a circuit consisting of CNOT and Toffoli gates (c), which, in turn, is lowered to a circuit over the Clifford+T gate set.

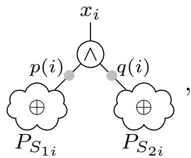
III. STATE OF THE ART

The current state-of-the-art technique considers logic networks over the gate basis $\{\neg, \oplus, \wedge\}$ [1]. Formally, a XAG is a logic networks in which each local function F_{S_i} is either a 2-input AND or a 2-input XOR, i.e., each step has one of the two following forms:

$$x_i = x_{j_{1i}} \oplus x_{j_{2i}} \quad \text{or} \quad x_i = x_{j_{1i}}^{p_{1i}} \wedge x_{j_{2i}}^{p_{2i}}, \tag{4}$$

where p_{1i} and p_{2i} are Boolean constants used to possibly complement the gate's fan-in. (Note that in a generic logic network these constants are unnecessary as the negation of a fan-in can be incorporated into the local function.)

The key idea in reference [1, Algorithm 1] is to look at the two inputs of an AND gate as parity functions over variables that are either primary inputs or preceding AND steps:



,

where $P_{S_{1i}}$ and $P_{S_{2i}}$ are the parity functions over the set of variables indexed by S_{1i}, $S_{2i} \subseteq [i-1]$. Since the parity function is reversible, the inputs can be easily computed in-place, i.e., without the need for an extra qubit. The AND steps, on the other hand, need out-of-place computation.

The algorithm starts by computing each AND gate in three simple steps: First, it computes the input parity functions in-place, then the AND gate in a new ancilla qubit, and then it cleans up the input. After that, it copies the state of the qubit holding the output step to the output qubit. Finally, it restores the ancilla qubits to $|0\rangle$ by cleaning up all AND steps.

IV. COMPILATION FLOW

As shown before, when compiling a XAG, the synthesis algorithm focuses on steps that need out-of-place computation—i.e., the outputs and the AND steps. To simplify our algorithm implementation, we represent the XAG in a higher level of abstraction that does not use nodes to represent the in-place XOR steps. We refer to this new and more general representation as high-level XAG, e.g., see Fig. 1a.

The main contributions to the synthesis technique rest in explicitly having a higher-level representation of the XAG and the analysis we do on it, which enables savings in both the number of qubits and the number of Clifford gates. Furthermore, our flow lowers the level of abstraction in small steps: Starting from a high-level XAG, we synthesize a quantum circuit with parity gates and Toffoli gates, e.g., Fig. 1b; Then we lower it to a circuit consisting of CNOT gates and Toffoli, which, in turn, is lowered to a circuit over the Clifford+T gate set. This progressive lowering process allows our flow to discover more facts about the program, thus finding better optimization opportunities.

A. High-level XAG

A high-level XAG is a logic network in which each local function F_{S_i} is either a 2-input AND or a 2-input XOR. The inputs, however, are parity functions:

$$x_i = P_{S_{1i}} \wedge P_{S_{2i}} \quad \text{or} \quad x_i = P_{S_{1i}} \oplus P_{S_{2i}} \tag{5}$$

for $n < i \leq n + r$, where $P_{S_{1i}}$ and $P_{S_{2i}}$ are parity functions over the set of variables indexed by S_{1i}, $S_{2i} \subseteq [i-1]$. Note that we can represent XOR steps as one parity function P_S. Also, we can merge XOR steps into the inputs of the AND steps, except when they drive an output.

The logic level l_i of a primary input or step i is defined as the earliest possible time in which it can be computed. Similarly, we define the reverse logic level l_i^r as the latest possible time in which gate i must be computed while not increasing the depth of the logic network

B. Synthesis

In the algorithms below, each step x_i for $0 \leq i \leq n+r$ has attributes $\mathtt{LR}(i)$, $\mathtt{L}(i)$, $\mathtt{R}(i)$, $\mathtt{FANIN}(i)$, $\mathtt{REF}(i)$, $\mathtt{LAST_REF}(i)$, $\mathtt{LVL}(i)$, and $\mathtt{TGT}(i)$, which interact as follows: $\mathtt{LR}(i)$ is the set of indexes of the variables that are common in both input parity functions, i.e., $\mathtt{LR}(i) = S_{1i} \cap S_{2i}$. $\mathtt{L}(i)$ and $\mathtt{R}(i)$ are the set of indexes of the variables that appear in only one of the input parity functions, i.e., $\mathtt{L}(i) = S_{1i} \setminus \mathtt{LR}(i)$ and $\mathtt{L}(i) = S_{2i} \setminus \mathtt{LR}(i)$. The $\mathtt{FANIN}(i)$ is a vector of indexes such that $\mathtt{FANIN}(i) = S_{1i} \cup S_{2i}$

The $\mathtt{REF}(i)$ attribute tells the algorithm the number of references to step x_i, including cleaning up. For example, given a step x_k that uses x_i in its input, if x_k needs clean up, then two references are added to x_i: one for the computation of x_k and one for its clean up. However, if x_k does not need cleanup (e.g., an output step), then just one reference is added. $\mathtt{LAST_REF}(i)$ tells the index of the last reference.

Finally, $\mathtt{LVL}(i)$ is simply the reverse logic level l_i^r, and $\mathtt{TGT}(i)$ indicates the target qubit of a step. Initially, $\mathtt{TGT}(i)$ is $\varnothing$ for all steps.

The synthesis process begins by assigning qubits to the primary inputs and primary outputs. The assignment is trivial for the inputs but more complicated for the outputs. For the sake of clarity, we look at the assignment as a separate algorithm.

Before delving into the details of the assignment algorithm, however, let's understand what are the complications with the primary outputs. Suppose we are given a high-level XAG with m outputs. An output can be: (1) The constant x_0, (2) a primary input, (3) a AND step, (4) a XOR step, or the complement of one of those. We can also have one step driving more than one output. Cases (1), (2), and (3) are simple:

(1) Either do nothing or add a NOT gate to the output qubit.
(2) Use a CNOT gate to copy the classical state.
(3) Compute the AND step directly on the output qubit—saving a qubit and the gates that would be necessary to clean up the AND step.

Careful handling of case (4) also allows us to save qubits and gates. We compute the XOR step on the output qubit. This direct computation saves us nothing, since all explicit XOR steps are outputs. To save resources, we look among the inputs of the XOR step for AND steps that we can compute on directly on the output qubit.

We now present the algorithm to assign qubits to the inputs and outputs:

Algorithm A: Given a high-level XAG with n inputs and m outputs, a quantum circuit, and a set of $n+m$ qubits $\mathtt{Q}$, this algorithm assigns target qubits to the primary inputs and primary outputs. The set $\mathtt{Q}$ is implemented as a vector where the first n elements identify the qubits used as primary inputs and the last m the primary outputs.

A1. [Assign PI.] Set the target qubits of all primary input steps, that is, $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[i-1]$, for $1 \leq i < n$.

A2. [Assign AND PO.] For each primary output i, if $\mathtt{TGT(i)} = \varnothing$ and $\circ_i = \wedge$, then set $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[n + (i-1)]$.

A3. [Assign XOR PO.] For each primary output i in which $\circ_i = \oplus$ and $\mathtt{TGT}(i) = \varnothing$. Set $\mathtt{TGT}(i) \leftarrow \mathtt{Q}[i-1]$ and look for a AND step among its inputs: First, we create vector $\mathtt{A}$ of all the AND that have not being assigned a qubit among the inputs of i, i.e., for $j \in \mathtt{FANIN}(i)$, if $\circ_j = \wedge$ and $\mathtt{TGT(j)} = \varnothing$, then add j to $\mathtt{A}$. Now, for $k \in \mathtt{A}$, if $\mathtt{REF}(i) = 1$ or $\mathtt{SIZE}(A) = 1$, and $\mathtt{LVL}(\mathtt{LAST_REF}(k)) \leq \mathtt{LVL}(i)$; then set $\mathtt{TGT}(k) \leftarrow \mathtt{TGT}(i)$, $\mathtt{REF}(k) \leftarrow \mathtt{REF}(k) - 1$.

In the following algorithm, we use functions that have the same name as quantum gates to denote the addition of the respective gate to the quantum circuit.

Algorithm B: Given a high-level XAG with n inputs and m outputs, a quantum circuit, and a set of $n+m$ qubits $\mathtt{Q}$, this algorithm assign target qubits to the primary inputs and primary outputs.

B1. [Initialize.] Execute Algorithm A.

B2. [Compute level.] We process the nodes of the high-level XAG level by level. We set $l_{\max} \leftarrow \max\{l_i \in 0 \leq i < n\}$. For $1 \leq h < l_{\max}$, we do: for all nodes i such that $\mathtt{LVL}(i) = h$, if i is an XOR node, i.e., $\circ_i = \oplus$, execute step B3; Otherwise execute step B4.

B3. [Compute XOR] We directly compute the gate on its target qubit: PARITY$(\{\mathtt{TGT}(j) : j \in \mathtt{FANIN}(i), \mathtt{TGT}(j) \neq \mathtt{TGT}(i)\}, \mathtt{TGT}(i))$. Then we update the reference counters of all nodes used by i: $\mathtt{REF}(j) = \mathtt{REF}(j) - 1$ for $j \in \mathtt{FANIN}(i)$.

B4. [Compute AND] If $\mathtt{TGT}(i) = \varnothing$, we request an ancilla a and set $\mathtt{TGT}(i) \leftarrow a$. If $|\mathtt{L}(i)| < |\mathtt{R}(i)|$, we swap them $\mathtt{L}(i) \leftrightarrow \mathtt{R}(i)$. We choose two qubits k and v to hold the in-place computation of the inputs. We set k to be an element of $\mathtt{L}(i)$. If $\mathtt{LR}(i)$ is not empty, then we choose $v \in \mathtt{LR}(i)$; Otherwise, we pick $v \in \mathtt{R}(i)$. We execute step B4 and add a Toffoli: TOFFOLI$(\{\mathtt{TGT}(k), \mathtt{TGT}(v)\}, \mathtt{TGT}(i)$. We also update the reference counters of all nodes used by i: $\mathtt{REF}(j) = \mathtt{REF}(j) - 1$ for $j \in \mathtt{FANIN}(i)$. Lastly, we clean up the inputs by executing B5 in reverse oder.

B5. [Compute inputs] Do:
- PARITY$(\{\mathtt{TGT}(j) : j \in \mathtt{L}(i), j \neq k\}, \mathtt{TGT}(k))$
- PARITY$(\{\mathtt{TGT}(j) : j \in \mathtt{LR}(i), j \neq v\}, \mathtt{TGT}(v))$
- If $\mathtt{LR}(i)$ is not empty: PARITY$(\{\mathtt{TGT}(v)\}, \mathtt{TGT}(k))$
- PARITY$(\{\mathtt{TGT}(j) : j \in \mathtt{R}(i), j \neq v\}, \mathtt{TGT}(v))$

B6. [Recursively try to cleanup] For $j \in \mathtt{FANIN}(i)$, if $\mathtt{REF}(i) = 0$, then we compute j using step B3 and try to cleanup the inputs j using this step.

B7. [Missing outputs.] This step takes care of any primary output that might not have been computed. For example, a primary output that is a primary input. Set $i = n$. For each primary output o: We do PARITY$(\{\mathtt{TGT}(i)\}\mathtt{Q}[i])$ if $\mathtt{Q}[i] \neq \mathtt{TGT}(o)$ and $o \neq 0$, and then $i \leftarrow i+1$.

B8. [Complement outputs.] Set $i = n$. For each primary output: We do X$(\mathtt{Q}[i])$ if the output is complemented, and then $i \leftarrow i+1$.

V. Experimental Results

We implemented the flow in Tweedledum[1]. We use various arithmetic and random-control functions from [11] as well as cryptographic functions and IEEE floating-point operations [12] as benchmarks. For the EPFL benchmarks, we list the currently best-known results for multiplicative complexity obtained from the state-of-the-art optimization approaches in [13]. We compile the functions into quantum operations of the form given by Eq. 3. The resulting quantum circuit representation uses the Clifford+T gate set.

In [1], the authors provide a C++ implementation of their algorithm that does not properly handle the general form, Eq. 3. The problem lies with the measurement-based cleanup [14]. Their implementation might apply this technique on Toffoli gates acting on the output qubits, thus destroying the initial state $|y\rangle$. We modify their implementation to correct this behavior.

We report the results in Figure 2. Note that we don't ignore any Clifford gates. (In [1], the Clifford gates in the decomposition of Toffoli gates to a Clifford+T gate set are ignored.) Also, we report the number of Clifford gates for the worst case, i.e., we assume that all measurement-based cleanups fail.

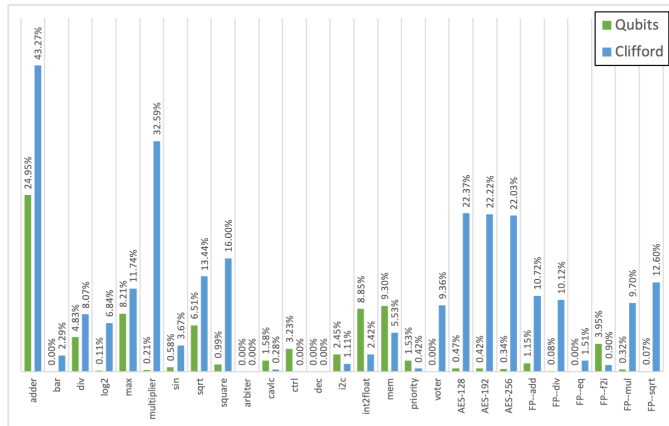


Fig. 2. Percentage improvements in the number of qubits and in the number of Clifford gates when compared against the current state of the art [13].

EPFL benchmarks: On these benchmarks our flow improves, on average, 4.07% the number of qubits and 8.72% the number of Clifford gates. The largest improvement is of 24.95% in the number of qubits and 43% in the number of Clifford gates, and occurs for the adder, when synthesizing for the general form, i.e., Eq. 3. Such impressive gain is due to Algorithm A, which is able to correctly identify that all Toffoli gates can be directly computed on the output qubits, and, thus, don't need to be cleanup.

Cryptographic functions & IEEE floating-point: We were unable to obtain results for all crypto benchmarks due to memory constraints. The state-of-the-art fails on *Keccak-f*, *SHA-256* and *SHA-512*. Our method, on the other handle, can handle *SHA-256* because of its multilevel intermediate representation, which is more memory efficient. The other results show a significant improvement on the number of Clifford gates: 22.21% on average. On the IEEE floating-point operations we also observe an important gain in the number of Clifford gates.

[1] https://github.com/boschmitt/tweedledum

VI. Conclusion and Future Directions

In this work we presented a compilation flow for classically defined quantum operations that are expressed by means of a XAG. We focused on XAGs with minimal number of AND nodes. The impact of the number of XOR nodes in the graph, on the other hand, should be better studied in the future. In our experiments, we have seen that the relation between the number of XOR steps and the number of CNOT gates is not straightforward.

Our technique achieves better results compared to other state-of-the-art automatic compilers [13]. Indeed, we can reduce the number of qubits by up to 24.95% and the number of Clifford gates by up to 43.3%. Crucially, these improvements were possible without increasing the number of T gates nor the execution time.

In addition, our multilevel intermediate representation (IR) allowed us to manipulate bigger circuits more efficiently. For example, we were able to deal with a benchmark that required to much memory when represented using a low-level IR.

References

[1] G. Meuli, M. Soeken, E. Campbell, M. Roetteler, and G. De Micheli, "The role of multiplicative complexity in compiling low T-count oracle circuits," in *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2019, pp. 1–8.

[2] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM review*, vol. 41, no. 2, pp. 303–332, 1999.

[3] P. C. Costa, S. Jordan, and A. Ostrander, "Quantum algorithm for simulating the wave equation," *Physical Review A*, vol. 99, no. 1, p. 012323, 2019.

[4] A. Suau, G. Staffelbach, and H. Calandra, "Practical quantum computing: solving the wave equation using a quantum approach," *arXiv preprint arXiv:2003.12458*, 2020.

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, 1996, pp. 212–219.

[6] A. W. Harrow, A. Hassidim, and S. Lloyd, "Quantum algorithm for linear systems of equations," *Physical review letters*, vol. 103, no. 15, p. 150502, 2009.

[7] B. D. Clader, B. C. Jacobs, and C. R. Sprouse, "Preconditioned quantum linear system algorithm," *Physical review letters*, vol. 110, no. 25, p. 250504, 2013.

[8] A. Scherer, B. Valiron, S.-C. Mau, S. Alexander, E. Van den Berg, and T. E. Chapuran, "Concrete resource analysis of the quantum linear-system algorithm used to compute the electromagnetic scattering cross section of a 2d target," *Quantum Information Processing*, vol. 16, no. 3, p. 60, 2017.

[9] G. H. Low and I. L. Chuang, "Optimal hamiltonian simulation by quantum signal processing," *Physical review letters*, vol. 118, no. 1, p. 010501, 2017.

[10] D. W. Berry, A. M. Childs, and R. Kothari, "Hamiltonian simulation with nearly optimal dependence on all parameters," in *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. IEEE, 2015, pp. 792–809.

[11] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The EPFL combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, no. CONF, 2015.

[12] V. A. Abril, P. Maene, N. Mertens, and N. Smart, "Bristol fashion MPC circuits," 2029. [Online]. Available: https://homes.esat.kuleuven.be/∼nsmart/MPC/

[13] E. Testa, M. Soeken, L. Amarù, and G. De Micheli, "Reducing the multiplicative complexity in logic networks for cryptography and security applications," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2019, pp. 1–6.

[14] C. Gidney, "Halving the cost of quantum addition," *Quantum*, vol. 2, p. 74, 2018.