Optimal Metastability-Containing Sorting via Parallel Prefix Computation

Johannes Bund, Christoph Lenzen, and Moti Medina

Abstract—Friedrichs *et al.* (TC 2018) showed that metastability can be *contained* when sorting inputs arising from time-to-digital converters, i.e., measurement values can be correctly sorted *without* resolving metastability using synchronizers first. However, this work left open whether this can be done by small circuits. We show that this is indeed possible, by providing a circuit that sorts Gray code inputs (possibly containing a metastable bit) and has asymptotically optimal depth and size. Our solution utilizes the parallel prefix computation (PPC) framework (JACM 1980). We improve this construction by bounding its fan-out by an arbitrary $f \geq 3$, without affecting depth and increasing circuit size by a small constant factor only. Thus, we obtain the first PPC circuits with asymptotically optimal size, constant fan-out, and optimal depth. To show that applying the PPC framework to the sorting task is feasible, we prove that the latter can, despite potential metastability, be decomposed such that the core operation is associative. We obtain asymptotically optimal metastability-containing sorting networks. We complement these results with simulations, independently verifying the correctness as well as small size and delay of our circuits. Proofs are omitted in this version; the article with full proofs is provided online at http://arxiv.org/abs/1911.00267.

1 Introduction

Metastability is a fundamental obstacle when crossing clock domains, potentially resulting in soft errors with critical consequences [14]. As it has been shown that metastability cannot be avoided deterministically [25], synchronizers [19] are employed to reduce the error probability to tolerable levels. This approach trades precious time for reliability: the more time is allocated for metastability resolution, the smaller the probability of metastability-induced faults.

Recently, a different approach has been proposed, coined *metastability-containing* (MC) circuits [10]. It accepts a limited amount of metastability in the input to a digital circuit and ensures limited metastability of its output, so that the result is still useful. In a series of works [3], [4], [24], we applied this approach to a fundamental primitive: sorting. The circuit given in [4] is asymptotically optimal in depth and size.

Our Contribution. In this article, we present the machinery used to obtain the circuit from [4] in detail. We prove that CMOS implementations of basic gates realize Kleene logic (cf. [20, section 64]), justifying the computational model introduced in [10] and used in this article.

The task of sorting an arbitrary number of inputs can be reduced to sorting two inputs by using sorting networks [21]. The 0-1-principle (cf. Section 2) shows that plugging an MC $2\text{-sort}(B)$ circuit (for B-bit inputs) into a sorting network (for n values) readily yields an MC circuit that is capable of sorting n inputs. Hence, we need to design a $2\text{-sort}(B)$ circuit sorting two inputs in an MC way.

As the choice of the encoding matters a lot for MC circuits, we characterize the set of input strings we want to sort ("valid strings"). A valid string is either a (standard) Gray code string or a string obtained from a Gray code string by replacing the unique bit that would change on the up-count to the "next" codeword by M for metastability (the third logic value in Kleene logic). When using non-redundant codes, the use of Gray codes is mandatory: when converting an analog value to a digital one, continuously changing the input can force any circuit (that uses the value in a non-trivial way) into metastability [25]. Moreover, for combinational circuits in the abstraction of Kleene logic, *all* output bits that change when flipping a given input bit must become unstable when the input bit is unstable, cf. [10]. For instance, encoding a value unknown to be 11 or 12 in standard binary code would result in a string that, once metastability has been resolved, may represent any number in the interval from 8 to 15, cf. Section 3.

Valid strings arise naturally when stopping a Gray code counter asynchronously [12] or, more generally, whenever performing analog-to-digital conversion; respective circuits may risk multiple metastable bits to achieve better average-case precision, but for the best worst-case precision one can stick to guaranteeing valid strings as output. Exploiting the structure of Gray code and the restriction to valid strings, we show how to reliably sort all inputs despite the uncertainty about the represented value arising from metastability.

We formally specify the $2\text{-sort}(B)$ circuit and then prove that the task of comparing two valid strings can be decomposed into first performing a four-valued comparison on each prefix pair of the two valid input strings, and then

- J. Bund and C. Lenzen are with the Max Planck Institute for Informatics, Saarland Informatics Campus, 66123 Saarbrücken, Germany. E-mail: {jbund, clenzen}@mpi-inf.mpg.de.
- M. Medina is with the School of Electrical & Computer Engineering, Ben-Gurion University of the Negev, 8410501 Beer Sheva, Israel. E-mail: medinamo@bgu.ac.il.

Manuscript received 15 Oct. 2018; revised 18 June 2019; accepted 28 Aug. 2019. Date of publication 7 Oct. 2019; date of current version 13 Jan. 2020. (Corresponding author: Johannes Bund). Recommended for acceptance by P. Gratz. Digital Object Identifier no. 10.1109/TC.2019.2939818

inferring the corresponding output bits. This reduces the design of 2-sort(B) to a parallel prefix computation (PPC) problem, which for our purposes can be phrased as follows.

Definition 1.1 (**$\mathrm{PPC}_{\oplus}(B)$**). *For associative $\oplus : D \times D \rightarrow D$ and $B \in \mathbb{N}$, a $\mathrm{PPC}_{\oplus}(B)$ circuit is specified as follows.*

Input: $d \in D^B$,
Output: $\pi \in D^B$,
Functionality: $\pi_i = \oplus_{j=1}^{i} d_j$ *for all* $i \in [1, B]$.

Fast PPC circuits that are simultaneously (asymptotically) optimal in depth and size are known due to a celebrated result by Ladner and Fischer [23]. Going beyond [4], we present the full range of solutions that can be derived using their framework, which allows for a trade-off between depth and size of the 2-sort circuit. Most prominently, optimizing for depth reduces the depth of the circuit by a factor of 2 compared to [4] to optimal $\lceil \log B \rceil$, at the expense of increasing the size by a factor of up to 2.

However, relying on the construction from [23] as-is results in a very large fan-out. We present a modification reducing fan-out to any number $f \geq 3$ without affecting depth, increasing the size by a factor of only $1 + \mathcal{O}(1/f)$ (plus at most $3B/2$ buffers). In particular, our results imply that the depth of an MC sorting circuit can match the delay of a non-containing circuit, while maintaining constant fan-out and a constant-factor size overhead. Due to the fact that PPC circuits lie at the heart of fast adders [27], we consider this result of independent interest.

We complement our theoretical findings by simulations confirming the correctness and small size of the devised circuits. Post-layout area and delay of the designed circuits compare favorably with a baseline provided by a straightforward non-containing implementation.

Organization of this Article. We discuss related work in Section 2. Some preliminaries, the computational model and its justification, as well as the problem specification are given in Section 3. Next, in Section 4, we break the task of designing a 2-sort(B) circuit down into comparing prefixes and subsequently generating the output bits out of the computed comparison values and the respective pair of input bits. The comparison can be further decomposed into sequential application of an associative operator, which enables application of the PPC framework to compute all prefixes efficiently in parallel with (asymptotically) optimal depth. In order to keep this article self-contained, we compactly review the PPC framework in Section 5. The section then proceeds to showing how to modify the construction for bounded fan-out and bounding the size of the resulting circuits. In Section 6, we implement the base operators by subcircuits and plug the pieces together to obtain complete circuits. We then simulate them up to an input width of $B = 16$ to independently verify their correctness, and provide delay and area of the laid out circuits. We compare to a non-containing version as baseline, demonstrating the controlled increase in size of the circuit. We conclude the article in Section 7, where we also briefly discuss follow-up work that generalizes our results, demonstrating that higher-level concepts of this work like sorting networks and parallel prefix computation are applicable to further MC circuits.

2 Related Work

Sorting Networks. Sorting networks (see, e.g., [21]) sort n inputs from a totally ordered universe by feeding them into n parallel wires that are connected by 2-sort elements, i.e., subcircuits sorting two inputs; these can act in parallel whenever they do not depend on each other's output. A correct sorting network sorts all possible inputs, i.e., the wires are labeled 1 to n such that the ith wire outputs the ith element of the sorted list of inputs. The *size* of a sorting network is its number of 2-sort elements and its *depth* is the maximum number of 2-sort elements an input may pass through until reaching the output.

The 0-1-principle [21] states that a sorting network — assuming the 2-sort circuits are correct — is correct if and only if it sorts 0-1 inputs correctly. Thus, we obtain sorting networks for inputs that may suffer from metastability by constructing 2-sort circuits (w.r.t. a suitable order on such inputs) and plugging them into existing sorting networks.

Sorting networks have been extensively studied. Tight lower bounds of depth $\Omega(\log n)$ (trivial) and size $\Omega(n \log n)$ (see, e.g., [8]) are known and can be simultaneously asymptotically matched [1]. More practically, for small values of n optimal depth and/or size networks are known [6], [7], [21]. Accordingly, our task boils down to finding optimal (or close to optimal) metastability-containing 2-sort circuits. For B-bit inputs, our 2-sort circuits have depth and size $\mathcal{O}(\log B)$ and $\mathcal{O}(B)$, respectively, which is (trivially) optimal up to constants; as size and depth of our circuits are close to non-containing 2-sort circuits (cf. Table 12), we conclude that our approach yields MC sorting networks that are optimal up to small constant factors in both depth and size.

Prior Work on MC Circuits. Recent work [10] shows that for any Boolean function a combinational MC circuit implementing its *metastable closure* (see Definition 3.8) exists. The metastable closure can be seen as a best effort to contain metastability: when for an input with (some) metastable bits the stable input bits already determine a given output bit of the original Boolean function, the closure attains the respective value on this output bit; otherwise it is metastable.

Unfortunately, the proof from [10], which uses a construction dating back to Huffman [16], yields circuits of exponential size in the number of input bits B. The same is true for speculative computing [28]. Unconditional lower bounds on MC circuits [17] show that this cannot be avoided in general, even if the implemented function admits a small non-containing circuit. The same work provides, assuming that at most k input bits can be metastable, a construction with multiplicative $B^{\mathcal{O}(k)}$ and additive $\mathcal{O}(k \log B)$ overheads in size and depth, respectively. For the 2-sort element, $k = 2$ (each Gray code string may contain one metastable bit), but the resulting circuits are still far from optimal.

In [10], an alternative construction relying on non-combinational logic is given, achieving (up to minor-order terms) factor $2k + 1$ increase in size and additive $\Theta(\log k)$ increase in depth of the resulting circuit; for a 2-sort circuit, $k = 2$, so these overheads are constant. Rule-of-thumb calculations suggest that optimized versions of the circuits presented here and derived by this method would have comparable performance. A fair and detailed comparison would require

TABLE 1
4-bit Binary Reflected Gray Code

#	$g_1, g_{2,4}$	#	$g_1, g_{2,4}$	#	$g_1, g_{2,4}$	#	$g_1, g_{2,4}$
0	0 000	4	0 110	8	1 100	12	1 010
1	0 001	5	0 111	9	1 101	13	1 011
2	0 011	6	0 101	10	1 111	14	1 001
3	0 010	7	0 100	11	1 110	15	1 000

fully-fledged designs of both approaches, which is beyond the scope of this article. Note, however, that our design has the advantage of being purely combinational.

Parallel Prefix Computation. Ladner and Fischer [23] studied the parallel application of an associative operator to all prefixes of an input string of length ℓ (over an arbitrary alphabet). They give parallel prefix computation circuits of depth $\mathcal{O}(\log \ell)$ and size $\mathcal{O}(\ell)$ (where the circuit implementing the operator is assumed to have size and depth 1). However, when requiring optimal depth of $\lceil \log \ell \rceil$, their corresponding solution suffers from fan-out larger than $\ell/2$. An earlier construction by Kogge and Stone [22] simultaneously achieves optimal depth and fan-out of 2. This yields the fastest adder circuits to date (cf. [27]), but at the expense of a large size of $\ell(\lceil \log \ell \rceil - 1) + 1$. A number of additional constructions have been developed for adders, including special cases ([2], [26]) of the one by Ladner and Fischer, cf. [31]. However, no other construction achieves asymptotically optimal depth and size.

3 Model and Problem

In this section, we discuss how to model metastability in a worst-case fashion and formally specify the input/output behavior of our circuits. Our model is a simplified version of the one from [10] for combinational circuits (cf. [9, Chap. 7]). This means to represent metastable "bits" by M and extend truth tables as in Kleene's 3-valued logic [20, Section 64].

Basic Notation. We set $[N] := \{0, \ldots, N-1\}$ for $N \in \mathbb{N}$ and $[i, j] = \{i, i+1, \ldots, j\}$ for $i, j \in \mathbb{N}$, $i \leq j$. We denote $\mathbb{B} := \{0, 1\}$ and $\mathbb{B}_{\mathtt{M}} := \{0, 1, \mathtt{M}\}$. For a B-bit string $g \in \mathbb{B}_{\mathtt{M}}^B$ and $i \in [1, B]$, denote by g_i its ith bit, i.e., $g = g_1 g_2 \ldots g_B$. We use the shorthand $g_{i,j} := g_i \ldots g_j$, where $i, j \in [1, B]$ and $i \leq j$. Let $\operatorname{par}(g)$ denote the parity of $g \in \mathbb{B}^B$, i.e, $\operatorname{par}(g) = \sum_{i=1}^{B} g_i \bmod 2$. For a function f and a set A we abbreviate $f(A) := \{f(y) \mid y \in A\}$.

3.1 Binary Reflected Gray Code

A standard binary representation of inputs is unsuitable: uncertainty of the input values may be arbitrarily amplified by the encoding. E.g., representing a value unknown to be 11 or 12, which are encoded as 1011 resp. 1100, would result in the bit string 1MMM, i.e., a string that is metastable in every position that differs for both strings. However, 1MMM may represent any number in the interval from 8 to 15, amplifying the initial uncertainty of being in the interval from 11 to 12. An encoding that does not lose precision for consecutive values is Gray code.

We use B-bit binary reflected Gray code, $rg_B : [N] \to \mathbb{B}^B$, which is defined recursively. For simplicity (and without loss of generality) we set $N := 2^B$. A 1-bit code is given by

TABLE 2
4-bit Valid Inputs

g	$\langle g \rangle$	g	$\langle g \rangle$	g	$\langle g \rangle$	g	$\langle g \rangle$
0000	0	0110	4	1100	8	1010	12
000M	–	011M	–	110M	–	101M	–
0001	1	0111	5	1101	9	1011	13
00M1	–	01M1	–	11M1	–	10M1	–
0011	2	0101	6	1111	10	1001	14
001M	–	010M	–	111M	–	100M	–
0010	3	0100	7	1110	11	1000	15
0M10	–	M100	–	1M10	–	–	–

$\operatorname{rg}_1(0) = 0$ and $\operatorname{rg}_1(1) = 1$. For $B > 1$, we start with the first bit fixed to 0 and counting with $\operatorname{rg}_{B-1}(\cdot)$ (for the first 2^{B-1} codewords), then toggle the first bit to 1, and finally "count down" $\operatorname{rg}_{B-1}(\cdot)$ while fixing the first bit again, cf. Table 1. Formally, this yields for $x \in [N]$

$$\operatorname{rg}_B(x) := \begin{cases} 0\ \operatorname{rg}_{B-1}(x) & \text{if } x \in [2^{B-1}] \\ 1\ \operatorname{rg}_{B-1}(2^B - 1 - x) & \text{if } x \in [2^B] \setminus [2^{B-1}]. \end{cases}$$

As each B-bit string is a codeword, the code is a bijection and the encoding function also defines the decoding function. Denote by $\langle \cdot \rangle : \mathbb{B}^B \to [N]$ the decoding function of a Gray code string, i.e., for $x \in [N]$, $\langle \operatorname{rg}_B(x) \rangle = x$.

For two binary reflected Gray code strings $g, h \in \mathbb{B}^B$, we define their maximum and minimum as

$$(\max^{\mathrm{rg}}\{g, h\}, \min^{\mathrm{rg}}\{g, h\}) := \begin{cases} (g, h) & \text{if } \langle g \rangle \geq \langle h \rangle \\ (h, g) & \text{if } \langle g \rangle < \langle h \rangle. \end{cases}$$

For example:

- $\max^{\mathrm{rg}}\{0011, 0100\} = \max^{\mathrm{rg}}\{\operatorname{rg}_B(2), \operatorname{rg}_B(7)\} = 0100$,
- $\min^{\mathrm{rg}}\{0111, 0101\} = \min^{\mathrm{rg}}\{\operatorname{rg}_B(9), \operatorname{rg}_B(10)\} = 0111$.

3.2 Valid Strings

The inputs to the sorting circuit may have some metastable bits, which means that the respective signals behave out-of-spec from the perspective of Boolean logic. Such inputs, referred to as *valid strings*, are introduced with the help of the following operator.

Definition 3.1 ($*$ Operator). *For $B \in \mathbb{N}$, define the operator $* : \mathbb{B}_{\mathtt{M}}^B \times \mathbb{B}_{\mathtt{M}}^B \to \mathbb{B}_{\mathtt{M}}^B$ by*

$$\forall i \in \{1, \ldots, B\} : (x * y)_i := \begin{cases} x_i & \text{if } x_i = y_i \\ \mathtt{M} & \text{else.} \end{cases}$$

Observation 3.2. *The operator $*$ is associative and commutative. Hence, for a set $S = \{x^{(1)}, \ldots, x^{(k)}\}$ of B-bit strings, we can use the shorthand $*S := *_{x \in S} x := x^{(1)} * x^{(2)} * \ldots * x^{(k)}$. We call $*S$* the superposition of the strings in *S.*

Valid strings have at most one metastable bit. If this bit resolves to either 0 or 1, the resulting string encodes either x or $x+1$ for some x, cf. Table 2.

Definition 3.3 (Valid Strings). *Let $B \in \mathbb{N}$ and $N = 2^B$. Then, the set of* valid strings of length *B* is

$$\mathcal{S}_{\mathrm{rg}}^B := \operatorname{rg}_B([N]) \cup \bigcup_{x \in [N-1]} \{\operatorname{rg}_B(x) * \operatorname{rg}_B(x+1)\}.$$

3.3 Resolution and Closure

To extend the specification of $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$ to valid strings, we make use of the *metastable closure* [10]. The metastable closure is defined over the possible *resolutions* of metastable bits.

Definition 3.4 (Resolution [10]). *For $x \in \mathbb{B}_{\mathtt{M}}^B$, define the resolution $\operatorname{res}(x) : \mathbb{B}_{\mathtt{M}}^B \rightarrow \mathcal{P}(\mathbb{B}^B)$ as follows:*

$$\operatorname{res}(x) := \{y \in \mathbb{B}^B \mid \forall i \in \{1, \dots, B\} : x_i \neq \mathtt{M} \Rightarrow y_i = x_i\}.$$

Thus, $\operatorname{res}(x)$ is the set of all strings obtained by replacing all Ms in x by either 0 or 1: M acts as a "wild card." For any x and y, we have that $\operatorname{res}(xy) = \operatorname{res}(x)\operatorname{res}(y)$.

We note two observations for later use.

Observation 3.5. *For any $x \in \mathbb{B}_{\mathtt{M}}^B$, $*\operatorname{res}(x) = x$.*

For example: $*\operatorname{res}(0\mathtt{M}10) = *\{0010, 0110\} = 0\mathtt{M}10$.

Observation 3.6. *For $\emptyset \neq S \subseteq \mathbb{B}^B$, we have $S \subseteq \operatorname{res}(*S)$.*

We observe that in general the reverse direction does not hold, i.e., $\operatorname{res}(*S) \nsubseteq S$. For example, consider $S = \{01, 10\}$ and thus $*S = \mathtt{MM}$ such that $\operatorname{res}(*S) = \{00, 01, 10, 11\} = \mathbb{B}^2$. Hence, $S \subseteq \operatorname{res}(*S)$ but not $\operatorname{res}(*S) \subseteq S$. In contrast, for $|\operatorname{res}(*S)| \leq 2$, we can see that the reverse direction holds.

Observation 3.7. *For any subset of strings $S \subseteq \mathbb{B}^B$, if $|\operatorname{res}(*S)| \leq 2$, then $\operatorname{res}(*S) = S$.*

The metastable closure of an operator on binary inputs extends it to inputs that may contain metastable bits. This is done by considering all resolutions of the inputs, applying the operator, and taking the superposition of the results.

Definition 3.8 (The M Closure [10]). *Given an operator $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$, its* metastable closure *$f_{\mathtt{M}} : \mathbb{B}_{\mathtt{M}}^n \rightarrow \mathbb{B}_{\mathtt{M}}^m$ is defined by $f_{\mathtt{M}}(x) := *\{f(x') | x' \in \operatorname{res}(x)\}$. Recalling the basic notation we abbreviate this by $f_{\mathtt{M}}(x) = *f(\operatorname{res}(x))$.*

The closure is the best one can achieve w.r.t. containing metastability with clocked logic using standard registers [10], i.e., when $f_{\mathtt{M}}(x)_i = \mathtt{M}$, no such implementation can guarantee that the ith output bit stabilizes in a timely fashion.

3.4 Output Specification

We want to construct a circuit computing the maximum and minimum of two valid strings, enabling us to build sorting networks for valid strings. First, however, we need to answer the question what it means to ask for the maximum or minimum of valid strings. To this end, suppose a valid string is $\operatorname{rg}_B(x) * \operatorname{rg}_B(x+1)$ for some $x \in [N-1]$, i.e., the string contains a metastable bit that makes it uncertain whether the represented value is x or $x+1$. If we wait for metastability to resolve, the string will stabilize to either $\operatorname{rg}_B(x)$ or $\operatorname{rg}_B(x+1)$. Accordingly, it makes sense to consider $\operatorname{rg}_B(x) * \operatorname{rg}_B(x+1)$ "in between" $\operatorname{rg}_B(x)$ and $\operatorname{rg}_B(x+1)$, resulting in the following total order on valid strings (cf. Table 2).

Definition 3.9 ($\prec$). *We define a total order $\prec$ on valid strings as follows. For $g, h \in \mathbb{B}^B$, $g \prec h \Leftrightarrow \langle g \rangle < \langle h \rangle$. For each $x \in [N-1]$, we define $\operatorname{rg}_B(x) \prec \operatorname{rg}_B(x) * \operatorname{rg}_B(x+1) \prec \operatorname{rg}_B(x+1)$. We extend the resulting relation on $\mathcal{S}_{\mathrm{rg}}^B \times \mathcal{S}_{\mathrm{rg}}^B$ to a total order by taking the transitive closure. Note that this also defines $\preceq$, via $g \preceq h \Leftrightarrow (g = h \vee g \prec h)$.*

TABLE 3
Extensions to Metastable Inputs of AND (Left), OR (Center), and an Inverter (Right) According to Kleene Logic

b \ a	0	1	M
0	0	0	0
1	0	1	M
M	0	M	M

b \ a	0	1	M
0	0	1	M
1	1	1	1
M	M	1	M

a	$\bar{a}$
0	1
1	0
M	M

We intend to sort with respect to this order. It turns out that implementing a 2-sort circuit w.r.t. this order amounts to implementing the metastable closure of $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$.

Lemma 3.10. *Let $g, h \in \mathcal{S}_{\mathrm{rg}}^B$. Then*

$$g \preceq h \Leftrightarrow (\max_{\mathtt{M}}^{\mathrm{rg}}\{g, h\}, \min_{\mathtt{M}}^{\mathrm{rg}}\{g, h\}) = (h, g).$$

In other words, $\max_{\mathtt{M}}^{\mathrm{rg}}$ and $\min_{\mathtt{M}}^{\mathrm{rg}}$ are the max and min operators w.r.t. the total order on valid strings shown in Table 2, e.g.,

- $\max_{\mathtt{M}}^{\mathrm{rg}}\{1001, 1000\} = \operatorname{rg}_4(15) = 1000$,
- $\max_{\mathtt{M}}^{\mathrm{rg}}\{0\mathtt{M}10, 0010\} = \operatorname{rg}_4(3) * \operatorname{rg}_4(4) = 0\mathtt{M}10$, and
- $\max_{\mathtt{M}}^{\mathrm{rg}}\{0\mathtt{M}10, 0110\} = \operatorname{rg}_4(4) = 0110$.

Hence, our task is to implement $\max_{\mathtt{M}}^{\mathrm{rg}}$ and $\min_{\mathtt{M}}^{\mathrm{rg}}$.

Definition 3.11 (2-sort(B)). *For $B \in \mathbb{N}$, a 2-sort(B) circuit is specified as follows.*

Input: $g, h \in \mathcal{S}_{\mathrm{rg}}^B$,
Output: $g', h' \in \mathcal{S}_{\mathrm{rg}}^B$,
Functionality: $g' = \max_{\mathtt{M}}^{\mathrm{rg}}\{g, h\}$, $h' = \min_{\mathtt{M}}^{\mathrm{rg}}\{g, h\}$.

3.5 Computational Model and CMOS Logic

We seek to use standard components and combinational logic only. We use the model of [10], which specifies the behavior of basic gates on metastable inputs via the metastable closure of their behavior on binary inputs, cf. Table 3. We use the standard notational convention that $a + b = \mathrm{OR}_{\mathtt{M}}(a, b)$ and $ab = \mathrm{AND}_{\mathtt{M}}(a, b)$.

Note that in this logic, most familiar identities hold: AND and OR are associative, commutative, and distributive, and DeMorgan's laws hold. However, naturally the law of the excluded middle becomes void. For instance, in general, $\mathrm{OR}(x, \bar{x}) \neq 1$, as $\mathrm{OR}(\mathtt{M}, \mathtt{M}) = \mathtt{M}$.

We now argue that basic CMOS gates behave according to this logic, justifying the model. For the sake of an intuitive notation, we apply some slightly unusual conventions. In the following, let R_1 be a wildcard that can refer to any resistance that is "low", i.e., close to being negligible, as e.g., that of a transistor in its stable conducting state (i.e., any PMOS transistor subjected to a low gate voltage or any NMOS transistor subjected to a high gate voltage). Similar, denote by R_0 any resistance that is "high", i.e., large compared to R_1, such as the resistance of a transistor in its stable non-conducting state. Thus, with a stable input $b \in \mathbb{B}$ (where we identify 0 with low and 1 with high voltage), an NMOS transistor attains resistance R_b, while a PMOS transistor attains resistance $R_{\bar{b}}$. We can extend this to unstable inputs M by making the conservative assumption that $R_{\mathtt{M}}$ is an arbitrary (possibly time-dependent) resistance.

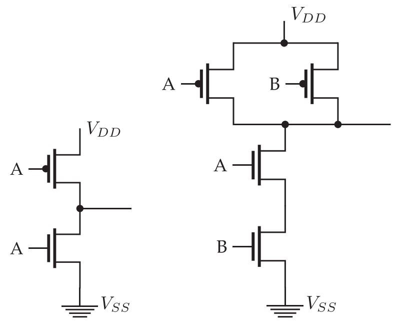


Fig. 1. Standard transistor-level implementations of inverter (left), NAND (center), and NOR (right) gates in CMOS technology. The latter can be turned into AND and OR, respectively, by appending an inverter.

With this notation, we can see that parallel and serial composition of transistors implements AND and OR in Kleene logic, respectively.

Lemma 3.12. *For $k \in \mathbb{N}$ sufficiently small so that $kR_1 \ll R_0$, let $a_1, \dots, a_k \in \mathbb{B}_{\mathtt{M}}$ be input signals fed to k NMOS transistors interconnected (i) in parallel or (ii) sequentially. Set $\sigma := \sum_{i=1}^{k} a_i$ and $\pi := \prod_{i=1}^{k} a_i$, i.e., the* OR *resp.* AND *over all inputs. Then the resistance between input and output of the resulting subcircuit is (roughly) (i) R_σ resp. (ii) R_π.*

The same arguments apply to PMOS transistors.

Corollary 3.13. *For $k \in \mathbb{N}$ sufficiently small so that $kR_1 \ll R_0$, let $a_1, \dots, a_k \in \mathbb{B}_{\mathtt{M}}$ be input signals fed to k PMOS transistors interconnected (i) in parallel or (ii) sequentially. Set $\sigma := \sum_{i=1}^{k} \bar{a}_i$ and $\pi := \prod_{i=1}^{k} \bar{a}_i$, i.e., the* OR *resp.* AND *over all inputs. Then the resistance between input and output of the resulting subcircuit is (roughly) (i) R_σ resp. (ii) R_π.*

We remark that the factor of k reduction in the gap between R_1 and R_0 may imply that a gate's output signal needs to be regenerated using a buffer. However, this is the same behavior as for logic that assumes stable signals only, so standard CMOS design techniques account for this.

From the above observations, we can readily infer that standard CMOS gate implementations behave according to Kleene logic in face of potentially metastable signals, justifying the model from [10].

Theorem 3.14. *The CMOS gates depicted in Fig. 1 implement the truth tables given in Table 3.*

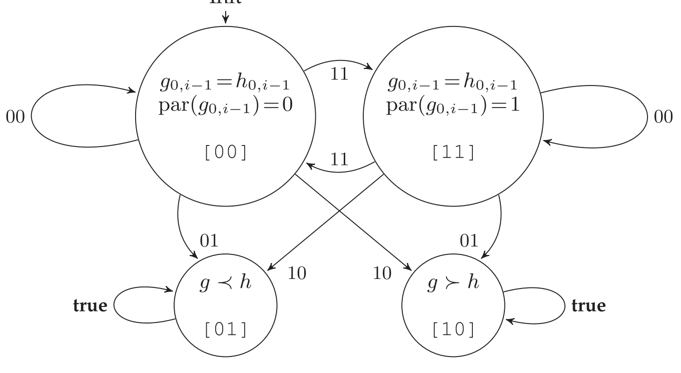


Fig. 2. Finite state machine determining which of two Gray code inputs $g, h \in \mathbb{B}^B$ is larger. In each step, it receives $g_i h_i$ as input. State encoding is given in square brackets.

TABLE 4
Run of the FSM on Inputs $g = 1001$ and $h = 1000$

i	0	1	2	3	4
$g_i h_i$		11	00	00	10
$s^{(i)} = s^{(i-1)} \diamond g_i h_i$	00	11	11	11	01
$g'_i = \text{out}(s^{(i-1)}, g_i h_i)_1$		1	0	0	0
$h'_i = \text{out}(s^{(i-1)}, g_i h_i)_2$		1	0	0	1

Similar reasoning applies to many gates, e.g., NAND and NOR gates. We stress, however, that the property of implementing the closure of the function computed by the gate on stable values is not universal for CMOS logic. For instance, standard transistor-level multiplexer implementations do not handle metastability well, cf. [11].

4 Decomposition of the Task

In this section, we show that computing $\max^{\text{rg}}_{\mathtt{M}}\{g, h\}$ and $\min^{\text{rg}}_{\mathtt{M}}\{g, h\}$ for valid strings $g, h \in \mathcal{S}^B_{\text{rg}}$ can be broken down into composing simple operators in $\mathbb{B}^2_{\mathtt{M}} \times \mathbb{B}^2_{\mathtt{M}} \to \mathbb{B}^2_{\mathtt{M}}$.

4.1 Comparing Stable Gray Codes via an FSM

Fig. 2 depicts a finite state machine performing a four-valued comparison of two Gray code strings. In each step of processing inputs $g, h \in \mathbb{B}^B$, it is fed the pair of ith input bits $g_i h_i$. In the following, we denote by $s^{(i)}(g, h)$ the state of the machine after i steps, where $s^{(0)}(g, h) := 00$ is the starting state. For ease of notation, we will omit the arguments g and h of $s^{(i)}$ whenever they are clear from context. Table 4 shows an example of a run of the finite state machine.

Because the parity keeps track of whether the remaining bits are to be compared w.r.t. the standard or "reflected" order, the state machine performs the comparison correctly w.r.t. the meaning of the states indicated in Fig. 2.

Lemma 4.1. *Let $g, h \in \mathbb{B}^B$ and $i \in [B+1]$. Then*

- *$s^{(i)} = 00$ is equivalent to $g_{1,i} = h_{1,i}$ and $g \prec h$ if and only if $g_{i+1,B} \prec h_{i+1,B}$,*
- *$s^{(i)} = 11$ is equivalent to $g_{1,i} = h_{1,i}$ and $g \prec h$ if and only if $g_{i+1,B} \succ h_{i+1,B}$,*
- *$s^{(i)} = 01$ is equivalent to $g \prec h$, and*
- *$s^{(i)} = 10$ is equivalent to $g \succ h$.*

This lemma gives rise to a sequential implementation of $2\text{-sort}(B)$ based on the given state machine, for input strings in $\mathbb{B}^B$. Table 5 lists the ith output bit as function of $s^{(i-1)}$ and the pair $g_i h_i$. Correctness of this computation follows immediately from Lemma 4.1.

We can express the transition function of the state machine as an (as easily verified) associative operator $\diamond$

TABLE 5
Computing $\max^{\text{rg}}\{g, h\}_i$ and $\min^{\text{rg}}\{g, h\}_i$ from the Current State $s^{(i-1)}$ and Inputs g_i and h_i

$s^{(i-1)}$	$\max^{\text{rg}}\{g, h\}_i$	$\min^{\text{rg}}\{g, h\}_i$
00	$\max\{g_i, h_i\}$	$\min\{g_i, h_i\}$
10	g_i	h_i
11	$\min\{g_i, h_i\}$	$\max\{g_i, h_i\}$
01	h_i	g_i

TABLE 6
Operators for Next State and Output

(a) The $\diamond$ operator

$\diamond$	00	01	11	10
00	00	01	11	10
01	01	01	01	01
11	11	10	00	01
10	10	10	10	10

(b) The out operator

out	00	01	11	10
00	00	10	11	10
01	00	10	11	01
11	00	01	11	01
10	00	01	11	10

The first operand is the current state, the second is the next input.

taking the current state and input g_ih_i as argument and returning the new state. Then $s^{(i)} = s^{(i-1)} \diamond g_ih_i$, where $\diamond$ is given in Table 6 a and $s^{(0)} = 00$. The out operator is derived from Table 5 by evaluating $\max^{\mathrm{rg}}\{g,h\}_i$ and $\min^{\mathrm{rg}}\{g,h\}_i$ for all possible values of $g_ih_i \in \mathbb{B}^2$. Noting that $s^{(0)} \diamond x = 00 \diamond x = x$ for all $x \in \mathbb{B}^2$, we arrive at the following corollary.

Corollary 4.2. *For all $i \in [1, B]$, we have that*

$$\max{}^{\mathrm{rg}}\{g,h\}_i \min{}^{\mathrm{rg}}\{g,h\}_i = \mathrm{out}\left(\bigdiamond_{j=1}^{i-1} g_jh_j, g_ih_i\right).$$

Our goal in this section is to extend this approach to potentially metastable inputs.

4.2 Dealing with Metastable Inputs

Our strategy is to replace all involved operators by their metastable closure: for $i \in [1, B]$ (i) compute $s^{(i)}_{\mathtt{M}}$, (ii) determine $\max^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}_i$ and $\min^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}_i$ according to Table 5, and finally (iii) exploit associativity of the operator computing the state $s^{(i)}_{\mathtt{M}}$ for usage in the PPC framework ([23], see Section 5). Thus, we only need to implement $\diamond_{\mathtt{M}}$ and the $\mathrm{out}_{\mathtt{M}}$ (both of constant size), plug them into the framework, and immediately obtain an efficient circuit.

The reader may ask why we compute $s^{(i)}_{\mathtt{M}}$ for all $i \in [0, B-1]$ instead of computing only $s^{(B)}_{\mathtt{M}}$ with a simple tree of $\diamond_{\mathtt{M}}$ elements, which would yield a smaller circuit. Since $s^{(B)}_{\mathtt{M}}$ is the result of the comparison of the entire strings, it could be used to compute all outputs, i.e., we could compute the output by $\mathrm{out}_{\mathtt{M}}(s^{(B)}_{\mathtt{M}}, g_ih_i)$ instead of $\mathrm{out}_{\mathtt{M}}(s^{(i-1)}_{\mathtt{M}}, g_ih_i)$. However, in case of metastability,

TABLE 7
Run of the FSM on Inputs $g = \mathtt{0M10}$ and $h = \mathtt{0010}$, Showing that Computing Only the Last State is Insufficient

i	0	1	2	3	4
g_ih_i		00	M0	11	00
$s^{(i)}_{\mathtt{M}} = s^{(i-1)}_{\mathtt{M}} \diamond_{\mathtt{M}} g_ih_i$	00	00	M0	1M	1M
$\mathrm{out}_{\mathtt{M}}(s^{(4)}_{\mathtt{M}}, g_ih_i)$		00	MM	11	00
$\mathrm{out}_{\mathtt{M}}(s^{(i-1)}_{\mathtt{M}}, g_ih_i)$		00	M0	11	00

This yields $\mathrm{out}_{\mathtt{M}}(\mathtt{1M}, \mathtt{M0}) = *\{00, 01, 10\} = \mathtt{MM}$ as second output, but $\mathrm{out}_{\mathtt{M}}(00, \mathtt{M0}) = *\{00, 10\} = \mathtt{M0}$ is correct.

TABLE 8
The $\diamond_{\mathtt{M}}$ Operator

$\diamond_{\mathtt{M}}$	00	0M	01	M1	11	1M	10	M0	MM
00	00	0M	01	M1	11	1M	10	M0	MM
0M	0M	0M	01	M1	M1	MM	MM	MM	MM
01	01	01	01	01	01	01	01	01	01
M1	M1	MM	MM	MM	0M	0M	01	M1	MM
11	11	1M	10	M0	00	0M	01	M1	MM
1M	1M	1M	10	M0	M0	MM	MM	MM	MM
10	10	10	10	10	10	10	10	10	10
M0	M0	MM	MM	MM	1M	1M	10	M0	MM
MM	MM	MM	MM	MM	MM	MM	MM	MM	MM

The first operand is the current state, the second are the next input bits.

this may lead to incorrect results. This can be seen in the example run of the FSM given in Table 7. We thus compute every intermediate state $s^{(i)}_{\mathtt{M}}$.

Unfortunately, even with this modification it is not obvious that our approach yields correct outputs. There are three hurdles to overcome:

(P1) Show that $\diamond_{\mathtt{M}}$ is associative.
(P2) Show that repeated application of $\diamond_{\mathtt{M}}$ computes $s^{(i)}_{\mathtt{M}}$.
(P3) Show that applying $\mathrm{out}_{\mathtt{M}}$ to $s^{(i-1)}_{\mathtt{M}}$ and g_ih_i results for all valid strings in $\max^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}_i \min^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}_i$.

Regarding the first point, we note the statement that $\diamond_{\mathtt{M}}$ is associative does not depend on B. In other words, it can be verified by checking for all possible $x, y, z \in \mathbb{B}^2_{\mathtt{M}}$ whether $(x \diamond_{\mathtt{M}} y) \diamond_{\mathtt{M}} z = x \diamond_{\mathtt{M}} (y \diamond_{\mathtt{M}} z)$. While it is tractable to manually verify all $3^6 = 729$ cases (exploiting various symmetries and other properties of the operator), it is tedious and prone to errors. Instead, we verified that both evaluation orders result in the same outcome by a short computer program.

Theorem 4.3. *(P1) holds, i.e., $\diamond_{\mathtt{M}}$ is associative.*

Apart from being essential for our construction, this theorem simplifies notation; in the following, we may write

$$\left(\bigdiamond_{\mathtt{M}}\right)_{i=1}^{j} g_ih_i := g_1h_1 \diamond_{\mathtt{M}} g_2h_2 \diamond_{\mathtt{M}} \ldots \diamond_{\mathtt{M}} g_jh_j\,,$$

where the order of evaluation does not affect the result.

We stress that in general the closure of an associative operator needs not be associative. A counter-example is given by binary addition modulo 4:

$$(\mathtt{0M} +_{\mathtt{M}} 01) +_{\mathtt{M}} 01 = \mathtt{MM} \neq \mathtt{1M} = \mathtt{0M} +_{\mathtt{M}} (01 +_{\mathtt{M}} 01).$$

4.3 Determining $s^{(i)}_{\mathtt{M}}$

For convenience of the reader, Table 8 gives the truth table of $\diamond_{\mathtt{M}} : \mathbb{B}^2_{\mathtt{M}} \times \mathbb{B}^2_{\mathtt{M}} \to \mathbb{B}^2_{\mathtt{M}}$. We need to show that repeated application of this operator to the input pairs g_jh_j, $j \in [1, i]$, actually results in $s^{(i)}_{\mathtt{M}}$. This is closely related to the key observation that if in a valid string there is a metastable bit at position m, then the remaining $B - m$ following bits are the maximum codeword of a $(B-m)$-bit code.

Observation 4.4. *For $g \in \mathcal{S}^B_{\mathrm{rg}}$, if there is an index $1 \leq m < B$ such that $g_m = \mathtt{M}$ then $g_{m+1,B} = 10^{B-m-1}$.*

Our reasoning will be based on distinguishing two main cases: one is that $s^{(i)}_{\mathtt{M}}$ contains at most one metastable bit, the other that $s^{(i)}_{\mathtt{M}} = \mathtt{MM}$. For each we need a technical statement.

TABLE 9
The $\mathrm{out}_{\mathtt{M}}$ Operator

$\mathrm{out}_{\mathtt{M}}$	00	0M	01	M1	11	1M	10	M0	MM
00	00	M0	10	1M	11	1M	10	M0	MM
0M	00	M0	10	1M	11	MM	MM	MM	MM
01	00	M0	10	1M	11	M1	01	0M	MM
M1	00	MM	MM	MM	11	M1	01	0M	MM
11	00	0M	01	M1	11	M1	01	0M	MM
1M	00	0M	01	M1	11	MM	MM	MM	MM
10	00	0M	01	M1	11	1M	10	M0	MM
M0	00	MM	MM	MM	11	1M	10	0M	MM
MM	00	MM	MM	MM	11	MM	MM	MM	MM

The first operand is the current state, the second is the next input bits.

Observation 4.5. *If* $|\mathrm{res}(s_{\mathtt{M}}^{(i)})| \leq 2$ *for any* $i \in [B+1]$, *then* $\mathrm{res}(s_{\mathtt{M}}^{(i)}) = \diamondsuit_{j=1}^{i} \mathrm{res}(g_j h_j)$.

Lemma 4.6. *Suppose that for valid strings* $g, h \in \mathcal{S}_{\mathrm{rg}}^{B}$, *it holds that* $s_{\mathtt{M}}^{(i)} = \mathtt{MM}$ *for some* $i \in [1, B]$. *Then* $g = h$ *and* $s_{\mathtt{M}}^{(j)} = \mathtt{MM}$ *for all* $j \in [i, B]$.

Equipped with these tools, we are ready to prove the second statement.

Theorem 4.7. *(P2) holds, i.e., for all* $g, h \in \mathcal{S}_{\mathrm{rg}}^{B}$ *and* $i \in [1, B]$, $s_{\mathtt{M}}^{(i)} = \left(\diamondsuit_{\mathtt{M}}\right)_{j=1}^{i} g_j h_j$.

4.4 Obtaining the Outputs from $s_{\mathtt{M}}^{(i)}$

Recall that $\mathrm{out} : \mathbb{B}^2 \times \mathbb{B}^2 \to \mathbb{B}^2$ is the operator given in Table 5 computing $\max^{\mathrm{rg}}\{g,h\}_i \min^{\mathrm{rg}}\{g,h\}_i$ out of $s^{(i-1)}$ and $g_i h_i$. For convenience of the reader, we provide the truth table of $\mathrm{out}_{\mathtt{M}} : \mathbb{B}_{\mathtt{M}}^2 \times \mathbb{B}_{\mathtt{M}}^2 \to \mathbb{B}_{\mathtt{M}}^2$ in Table 9. We derive the third property.

Theorem 4.8. *(P3) holds, i.e., given valid inputs* $g, h \in \mathcal{S}_{\mathrm{rg}}^{B}$ *and* $i \in [1, B]$, $\mathrm{out}_{\mathtt{M}}(s_{\mathtt{M}}^{(i-1)}, g_i h_i) = \max_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i \min_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i$.

5 The PPC Framework

In order to derive a small circuit from the results of Section 4, a straightforward approach would be to unroll the FSM. We could design a circuit implementing the transition function $\diamond_{\mathtt{M}}$ and apply it B times to the starting state $s^{(0)}$ and each input $g_i h_i$. However, computing the sequence of states step by step yields a (non-optimal) linear depth of at least B.

Hence, we make use of the PPC framework by Ladner and Fischer [23]. They describe a generic method that is applicable to *any* finite state machine translating a sequence of B input symbols to B output symbols, to obtain circuits of size $\mathcal{O}(B)$ and depth $\mathcal{O}(\log B)$. They observe that each input symbol defines a restricted transition function. Compositions of these functions evaluated on the starting state yield the state of the machine after receiving corresponding inputs. The major advantage of the technique is that compositions of restricted transition functions can be computed in parallel due to associativity, yielding a depth of $\mathcal{O}(\log B)$. This matches our needs, as we need to determine $s_{\mathtt{M}}^{(i)}$ for each $i \in [B]$. However, their generic construction involves large constants. Fortunately, we have established that $\diamond_{\mathtt{M}} : \mathbb{B}_{\mathtt{M}}^2 \times \mathbb{B}_{\mathtt{M}}^2 \to \mathbb{B}_{\mathtt{M}}^2$ is an associative operator, permitting us to directly apply the circuit templates for associative operators they provide for computing $s_{\mathtt{M}}^{(i)} = (\diamondsuit_{\mathtt{M}})_{j=1}^{i} g_j h_j$ for all $i \in [B]$. Accordingly, we discuss these templates only. During discussion of the basic construction we show a minor improvement on their results.

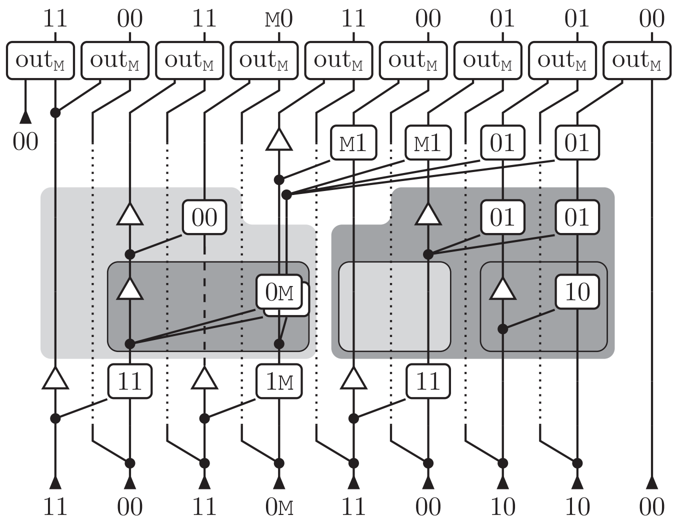


Fig. 3. An example for a computation of the $2\text{-sort}(9)$ circuit arising from our construction for fan-out $f = 3$. The inputs are $g = 101010110$ and $h = 101\mathtt{M}10000$; see Table 10 for $s_{\mathtt{M}}^{(i)}(g, h)$ and the output. We labeled each $\diamond_{\mathtt{M}}$ by its output. Buffers and duplicated gates (here the one computing $0\mathtt{M}$) reduce fan-out, but do not affect the computation. Grey boxes indicate recursive steps of the PPC construction; see also Fig. 7 for a larger PPC circuit using the one here in its "right" top-level recursion. For better readability, wires not taking part in a recursive step are dashed or dotted.

Before proceeding, the reader may want to take a look at the example given in Fig. 3, which shows how a $2\text{-sort}(9)$ derived from our construction processes an input pair.

5.1 The Basic Construction

We revisit the templates for parallel computation of all prefixes, i.e., the part of the framework relevant to our construction. To this end, recall Definition 1.1. In our case, $\oplus = \diamond_{\mathtt{M}}$ and $D = \mathbb{B}_{\mathtt{M}}^2$. [23] provides a family of recursive constructions of $\mathrm{PPC}_{\oplus}$ circuits. They are obtained by combining two different recursive patterns. The first pattern, which optimizes for size of the resulting circuits, is depicted in Fig. 4a. We distinguish between even and odd number of inputs. If B is even, we discard the rightmost gray wire and set $\bar{B} := B$; if B is odd, we set $\bar{B} := B - 1$ and include the rightmost wire. In the following, denote by $|C|$ the size of a circuit C and by $d(C)$ its depth.

Lemma 5.1. *Suppose that* C *and* P *are circuits implementing* $\oplus$ *and* $\mathrm{PPC}_{\oplus}(\lceil B/2 \rceil)$ *for some* $B \in \mathbb{N}$, *respectively. Then applying the recursive pattern given at the left of Fig. 4 yields a* $\mathrm{PPC}_{\oplus}(B)$ *circuit. It has depth* $2d(C) + d(P)$ *and size at most* $(B-1)|C| + |P|$. *Moreover, the last output is at depth at most* $d(C) + d(P)$ *of the circuit.*

The second recursive pattern, shown in Fig. 4c, avoids to increase the depth of the circuit beyond the necessary $d(C)$ for each level of recursion. Assume for now that B is a power of 2. We represent the recursion as a tree T_b, where $b := \log B$, given in the center of Fig. 4. It has depth b with all leaves (filled in white) in this depth, and there are two types of non-leaf nodes: *right* nodes (filled in black) have

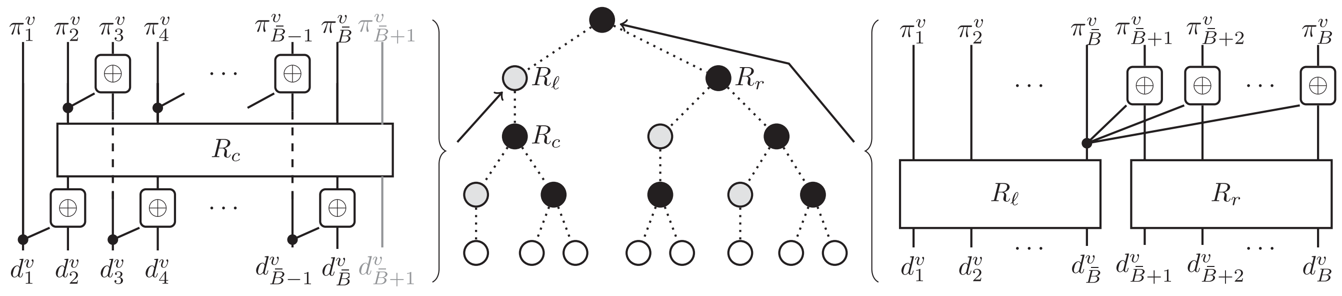

π_1^v π_2^v π_3^v π_4^v $\pi_{\bar{B}-1}^v$ $\pi_{\bar{B}}^v$ $\pi_{\bar{B}+1}^v$
R_c
d_1^v d_2^v d_3^v d_4^v $d_{\bar{B}-1}^v$ $d_{\bar{B}}^v$ $d_{\bar{B}+1}^v$
R_ℓ
R_r
R_c
π_1^v π_2^v $\pi_{\bar{B}}^v$ $\pi_{\bar{B}+1}^v$ $\pi_{\bar{B}+2}^v$ π_B^v
R_ℓ
R_r
d_1^v d_2^v $d_{\bar{B}}^v$ $d_{\bar{B}+1}^v$ $d_{\bar{B}+2}^v$ d_B^v
(a) Recursion pattern of left nodes. (b) Recursion tree T_4. (c) Recursion pattern of right nodes.

Fig. 4. The recursion tree T_4 (center). Right nodes are depicted black, left nodes gray, and leaves white. The recursive patterns applied at left and right nodes are shown on the left and right, respectively. At the root and its left child, we have that $\bar{B} = B/2$; for other nodes, $\bar{B}$ gets halved for each step further down the tree (where the leaves simply wire their single input to their single output). The left pattern comes in different variants. The gray wire with index $\bar{B}+1$ is present only if B is odd; this never occurs in $\mathrm{PPC}(C, T_b)$, but becomes relevant when initially applying the left pattern exclusively for $k \in \mathbb{N}$ steps (see Theorem 5.6), reducing the size of the resulting circuit at the expense of increasing its depth by k.

two children, a left and a right node, whereas *left* nodes (filled in gray) have a single child, which is a right node. T_b is essentially a Fibonacci tree in disguise.

Definition 5.2. *T_0 is a single leaf. T_1 consists of the (right) root and two attached leaves. For $b \geq 2$, T_b can be constructed from T_{b-1} and T_{b-2} by taking a (right) root r, attaching the root of T_{b-1} as its right child, a new left node ℓ as the left child of r, and then attaching the root of T_{b-2} as (only) child of ℓ.*

The recursive construction is now defined as follows. A right node applies the pattern given in Fig. 4 to the right. R_ℓ is the circuit (recursively) defined by the subtree rooted at the left child and R_r is the circuit (recursively) defined by the subtree rooted at the right child. Furthermore, $\bar{B} = 2^{b-d-1}$, where $d \in [b]$ is the depth of the node. A left child applies the pattern on the left. R_c is (recursively) defined by the subtree rooted at its child and $\bar{B} = 2^{b-d}$, where $d \in [b]$ is the depth of the node.

The base case for a single input and output is simply a wire connecting the input to the output, for both patterns. As $b = \log B$ and each recursive step cuts the number of inputs and outputs in half, the base case applies if and only if the node is a leaf. Note that the figure shows the recursive patterns at the root and its left child, where $\bar{B} = 2^{b-1}$ is always even (i.e., in this recursive pattern, the gray wire with index $\bar{B}+1$ is never present); when applying the patterns to nodes further down the tree, $\bar{B}$ and B are scaled down by a factor of 2 for every step towards the leaves.

In the following, denote by $\mathrm{PPC}(C, T_b)$ the circuit that results from applying the recursive construction described above to the base circuit C implementing $\oplus$. Moreover, we refer to the ith input and output of the subcircuit corresponding to node $v \in T_b$ as d_i^v and π_i^v, respectively.

Lemma 5.3. *If C implements $\oplus$, $\mathrm{PPC}(C, T_b)$ is a $\mathrm{PPC}_\oplus(2^b)$ circuit, that has depth $b \cdot d(C)$.*

It remains to bound the size of the circuit. Denote by F_i, $i \in \mathbb{N}$, the ith Fibonacci number, i.e., $F_1 = F_2 = 1$ and $F_{i+1} = F_i + F_{i-1}$ for all $2 \leq i \in \mathbb{N}$.

Lemma 5.4. $\mathrm{PPC}(C, T_b)$ *has size* $(2^{b+2} - F_{b+5} + 1)|C|$.

Asymptotically, the subtractive term of F_{b+5} is negligible, as $F_{b+5} \in (1/\sqrt{5} + o(1))((1+\sqrt{5})/2)^{b+5} \subseteq \mathcal{O}(1.62^b)$; however, unless B is large, the difference is substantial. We also get a simple upper bound for arbitrary values of B. To this end, we "split" in the recursion such that the left branch is "complete" (i.e. the number of inputs is a power of 2), while applying the same splitting strategy on the right. This is where our construction differs from and improves on [23]. They perform a balanced split and obtain an upper bound of $4B$ on the circuit size.

Corollary 5.5. *For $B \in \mathbb{N}$ and circuit C implementing $\oplus$, set $b := \lceil \log B \rceil$. Then a $\mathrm{PPC}_\oplus(B)$ of depth $\lceil \log B \rceil d(C)$ and size smaller than $(5B - 2^b - F_{b+3})|C| \leq (4B - F_{b+3})$ exists.*

We remark that one can give more precise bounds by making case distinctions regarding the right recursion, which for the sake of brevity we omit here. Instead, we computed the exact numbers for $B \leq 70$, see Fig. 5.

The construction derived from iterative application of Lemma 5.1 can be combined with $\mathrm{PPC}(C, T_b)$, achieving the following trade-off; note that if $B = 2^b$ for $b \in \mathbb{N}$, then $F_{\lceil \log B \rceil - k + 3}$ can be replaced by F_{b-k+5}.

Theorem 5.6 (improving on [23]). *Suppose C implements $\oplus$. For all $k \in [0, \lceil \log B \rceil]$ and $B \in \mathbb{N}$, there is a $\mathrm{PPC}_\oplus(B)$ circuit of depth $(\lceil \log B \rceil + k)d(C)$ and size at most*

$$\left(\left(2 + \frac{1}{2^{k-1}}\right)B - F_{\lceil \log B \rceil - k + 3}\right)|C| .$$

5.2 Constant Fan-Out at Optimal Depth

The optimal depth construction incurs an excessively large fan-out of $\Theta(B)$, as the last output of left recursive calls needs to drive all the copies of C that combine it with each of the corresponding right call's outputs. This entails that, despite its lower depth, it will not result in circuits of smaller physical delay than simply recursively applying the construction from Fig. 4a. Naturally, one can insert buffer trees to ensure a constant fan-out (and thus constantly bounded ratio between delay and depth), but this increases the depth to $\Theta(\log^2 B + d(C) \log B)$.

We now modify the recursive construction to ensure a constant fan-out, at the expense of a limited increase in size of the circuit. The result is the first construction that has size $\mathcal{O}(B)$, optimal depth, and constant fan-out.

In the following, we denote by $f \geq 3$ the maximum fan-out we are trying to achieve, where we assume that gates or memory cells providing the input to the circuit do not need

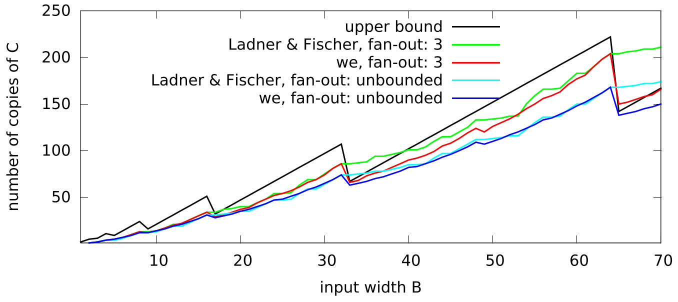


Fig. 5. Comparison of the balanced recursion from [23] and ours. The curves for unbounded fan-out are the exact sizes obtained, whereas "upper bound" refers to the bound from Corollary 5.5; the fan-out 3 curves show that the unbalanced strategy performs better also for the construction from Theorem 5.16 (for $f = 3$ and $k = 0$) we derive next.

to drive any other components. For simplicity, we consider C to be a single gate, i.e., a gate driving two C components has exactly fan-out 2.

We proceed in two steps. First, we insert $2B$ buffers into the circuit, ensuring that the fan-out is bounded by 2 everywhere except at the gate providing the last output of each subcircuit corresponding to a left node. In the second step, we will resolve this by duplicating these gates sufficiently often, recursively propagating the changes down the tree. Neither of these changes will affect the output (i.e. the correctness) of the circuit or its depth, so the main challenges are to show our claim on the fan-out and bounding the size of the final circuit.

5.2.1 Step 1: Almost Bounding Fan-Out by 2

Before proceeding to the construction in detail, we need some structural insight on the circuit.

Definition 5.7. *For node $v \in T_b$, define its* range R_v *and* left-count α_v *recursively as follows.*

- *If v is the root, then $R_v = [1, 2^b]$ and $\alpha_v = 0$.*
- *If v is the left child of p with $R_p = [i, i+j]$, then $R_v = [i, i+(j+1)/2]$ and $\alpha_v = \alpha_p$.*
- *If v is the right child of right node p with $R_p = [i, i+j]$, then $R_v = [i+(j+1)/2+1, i+j]$ and $\alpha_v = \alpha_p$.*
- *If v is the right child of left node p, then $R_v = R_p$ and $\alpha_v = \alpha_p + 1$.*

Hence, the left-count α_v tells us for every node $v \in T_b$ the number of left recursion steps preceding v, whereas R_v gives us information about the range of inputs used at node v. We observe that each recursion halves the number of inputs and that the range is only cut in half if α_v does not increase. Combining these observations with structural insights on the recursion patterns in Fig. 4a and 4c, we state the following four properties of $\mathrm{PPC}(C, T_b)$.

Lemma 5.8. *Suppose the subcircuit of $\mathrm{PPC}(C, T_b)$ represented by node $v \in T_b$ in depth $d \in [b+1]$ has range $R_v = [i, i+j]$. Then*

(i) it has 2^{b-d} inputs,
(ii) $j = 2^{b-d+\alpha_v} - 1$,
(iii) if v is a right node, all its inputs are outputs of its childrens' subcircuits, and
(iv) if v is a left node or leaf, only its even inputs are provided by its child (if it has one) and for odd $k \in [1, 2^{b-d}]$, we have that $d_k^v = \oplus_{k'=i+(k-1)2^{\alpha_v}}^{i+k2^{\alpha_v}-1} d_{k'}$.

Lemma 5.8 leads to an alternative representation of the circuit $\mathrm{PPC}(C, T_b)$, see Fig. 6, in which we separate gates in the recursive pattern from Fig. 4a that occur before the subcircuit R_c. Adding the buffers we need in our construction, this results in the modified patterns given in Fig. 6b. The separated gates appear at the bottom of Fig. 6a: for each leaf v of T_b, there is a tree of depth α_v aggregating all of the circuit's inputs from its range. Each non-root node in an aggregation tree provides its output to its parent. In addition, one of the two children of an inner node in the tree must provide its output as an input to one of the subcircuits corresponding to a node of T_b, cf. Property (iv) of Lemma 5.8.

From this representation, we will derive that the following modifications of $\mathrm{PPC}(C, T_b)$ result in a $\mathrm{PPC}_\oplus(2^b)$ circuit $\mathrm{PPC}(C, T_b)'$, for which a fan-out larger than 2 exclusively occurs on the last outputs of subcircuits corresponding to nodes of T_b.

1) Add a buffer on each wire connecting a non-root node of any of the aggregation trees to its corresponding subcircuit (see Fig. 6a).
2) For the subcircuit corresponding to left node ℓ with range $R_\ell = [i, i+j]$, add for each even $k \leq j$ (i.e., each even k but the maximum of $j+1$) a buffer before output π_k^ℓ (see bottom of Fig. 6b).
3) For each right node r with range $[i, i+j]$, add a buffer before output $\pi_{(j+1)/2}^r$ (see top of Fig. 6b).

Lemma 5.9. *With the exception of gates providing the last output of subcircuits corresponding to nodes of T_b (blue in Fig. 6b), fan-out of $\mathrm{PPC}(C, T_b)'$ is 2. Buffers or gates driving an output of the circuit drive nothing else.*

It remains to count the inserted buffers. We do so by computing a closed form expression from the linear recurrence that describes the number of nodes of a given type (left, right, leaf) in a given depth as function of the previous one. The following helper statement will be useful for this, but also later on.

Lemma 5.10. *Denote by $L_b \subseteq T_b$ the set of leaves of T_b. Then $|L_b| = F_{b+2}$ and $\sum_{v \in L_b} 2^{\alpha_v} = 2^b$.*

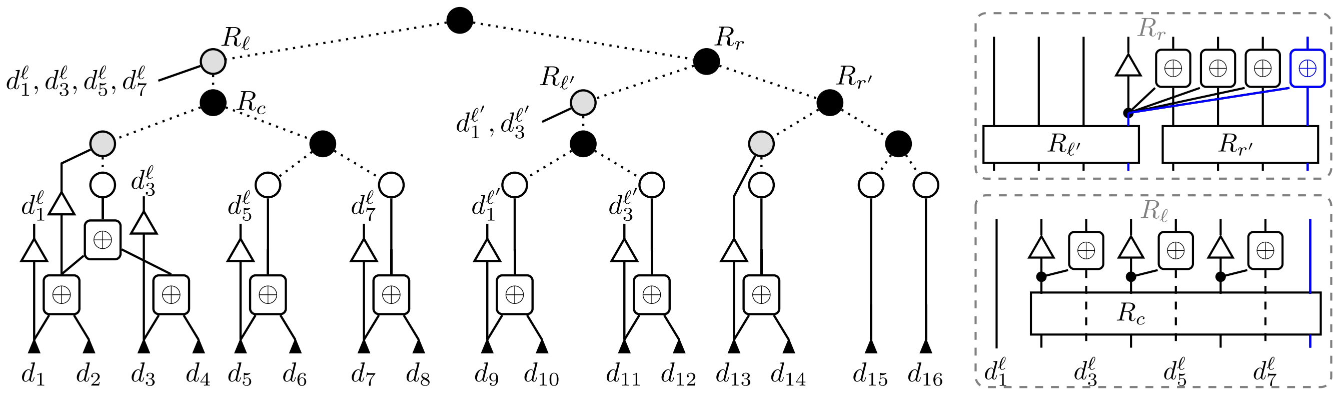
Lemma 5.11. *Denote by s the size of a buffer. Then*

$$|\mathrm{PPC}(C, T_b)'| = |\mathrm{PPC}(C, T_b)| + \left(2^b + 2^{b-1} - F_{b+3}\right)s\,.$$

Similar arguments serve later as well. The main reason why we will define the function $a(v)$ in the next section without rounding is to ensure that we again obtain linear recurrences, which can be solved using standard techniques from linear algebra. As a downside, this results in slightly overestimating the size of circuits, as we may ask for more copies of gates from children than are actually needed.

5.2.2 Step 2: Bounding Fan-Out by f

In the second step, we need to resolve the issue of high fan-out of the last output of each recursively used subcircuit in $\mathrm{PPC}(C, T_b)'$. Our approach is straightforward. Starting at the root of T_b and progressing downwards, we label each


R_ℓ
R_r
$d_1^\ell, d_3^\ell, d_5^\ell, d_7^\ell$
R_c
$R_{\ell'}$
$R_{r'}$
$d_1^{\ell'}, d_3^{\ell'}$
d_1^ℓ
d_3^ℓ
d_5^ℓ
d_7^ℓ
$d_1^{\ell'}$
$d_3^{\ell'}$
d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13} d_{14} d_{15} d_{16}
R_r
$R_{\ell'}$
$R_{r'}$
R_ℓ
R_c
d_1^ℓ d_3^ℓ d_5^ℓ d_7^ℓ

(a) Recursion tree T_4 with separated aggregation trees and added buffers. (b) Recursive patterns R_r and R_ℓ.

Fig. 6. Construction of $\mathrm{PPC}(C, T_4)'$. On the left, we see the recursion tree, with the aggregation trees separated and shown at the bottom. Inputs are depicted as black triangles. On the right, the application of the recursive patterns at the children of the root is shown. Parts marked blue will be duplicated in the second step of the construction that achieves constant fan-out; this will also necessitate to duplicate some gates in the aggregation trees.

node v with a value $a(v)$ that specifies a sufficient number of additional copies of the last output of the subcircuit represented by v to avoid fan-out larger than f. At right nodes, this is achieved by duplicating the gate computing this output sufficiently often, marked blue in Fig. 6b (top). For left nodes, we simply require the same number of duplicates to be provided by the subcircuit represented by their child (i.e., we duplicate the blue wire in the bottom recursive pattern shown in Fig. 6b). Finally, for leaves, we will require a sufficient number of duplicates of the root of their aggregation tree; this, in turn, may require to make duplicates of their descendants in the aggregation tree.

We define $a(v)$ and then utilize it to describe our fan-out f circuit. Afterwards, we will analyze the increase in size of the circuit compared to $\mathrm{PPC}(C, T_b)'$.

Definition 5.12 **($a(v)$).** *Fix $b \in \mathbb{N}_0$. For $v \in T_b$ in depth $d \in [b+1]$, define*

$$a(v) :\\ = \begin{cases} 0 & \text{if } v \text{ is the root} \\ \frac{a(p)+2^{b-d}}{f} & \text{if } v \text{ is the left child of } p \\ \frac{a(p)}{f} & \text{if } v \text{ is the right child of right node } p \\ a(p) & \text{if } v \text{ is the (only) child of left node } p. \end{cases}$$

Lemma 5.13. *Suppose that for each leaf $v \in T_b$, there are $\lfloor a(v) \rfloor$ additional copies of the root of the aggregation tree, and for each right node $v \in T_b$, we add $\lfloor a(v) \rfloor$ gates that compute (copies of) the last output of their corresponding subcircuit of $\mathrm{PPC}(C, T_b)'$. Then we can wire the circuit such that all gates that are not in aggregation trees have fan-out at most f, and each output of the circuit is driven by a gate or buffer driving only this output.*

It remains to modify the aggregation trees so that sufficiently many copies of the roots' output values are available.

Lemma 5.14. *Consider an aggregation tree corresponding to leaf $v \in T_b$ and fix $f \geq 3$. We can modify it such that the fan-out of all its non-root nodes becomes at most f, there are $\lfloor a(v) \rfloor$ additional gates computing the same output as the root, and at most $(fa(v))/(f-2) + (2^{\alpha_v - 1})/(f-1)$ gates are added.*

Finally, we need to count the total number of gates we add when implementing these modifications to the circuit.

Lemma 5.15. *For $f \geq 3$, define $\mathrm{PPC}^{(f)}(C, T_b)$ by modifying $\mathrm{PPC}(C, T_b)'$ according to Lemmas 5.13 and 5.14. Then, with $\lambda_1 := (1+\sqrt{5})/4$, $|\mathrm{PPC}^{(f)}(C, T_b)|$ is bounded by*

$$|\mathrm{PPC}(C, T_b)'| + 2^b\left(\frac{1}{2f-2} + \frac{2}{f-2} + \mathcal{O}\left(\frac{\lambda_1^b}{f^2}\right)\right)|C|\,.$$

As an example for the overall resulting construction, we show $\mathrm{PPC}^{(3)}(C, T_4)$ in Fig. 7. We summarize our findings in the following theorem.

Theorem 5.16. *Suppose that C implements $\oplus$, buffers have size s and depth at most $d(C)$, and set $\lambda_1 := (1+\sqrt{5})/4$. Then for all $k \in [b+1]$, $b \in \mathbb{N}_0$, and $f \geq 3$, there is a $\mathrm{PPC}_\oplus(2^b)$ circuit of fan-out f, depth $(b+k)d(C)$, and size at most*

$$\left(2^{b+1} + 2^{b-k}\left(2 + \frac{5f-6}{2f^2-6f+4} + \mathcal{O}\left(\frac{\lambda_1^b}{f^2}\right)\right)\right)|C|\\ + \left(2^b + 2^{b-k-1}\right)s\,.$$

We refrain from analyzing the size of the construction for values of B that are not powers of 2. However, in Fig. 8 we plot the exact bounds (without buffers) for $k = 0$ and selected values of f against B.

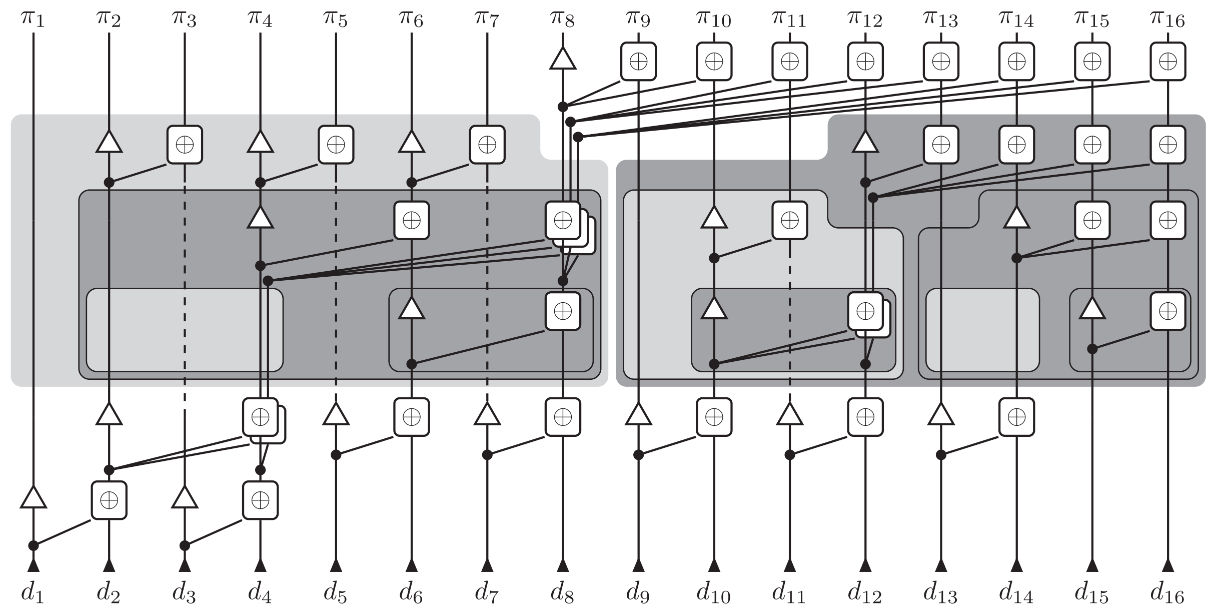
6 Simulation

In addition to the formal statements from the previous sections, we verify the correctness of our circuits by VHDL simulation. To this end, we first need to specify implementations of the subcircuits computing $\diamond_{\mathtt{M}}$ and $\mathrm{out}_{\mathtt{M}}$.

6.1 Gate-Level Implementation of Operators

From Tables 6 a and 6 b, for $s, b \in \mathbb{B}^2$ we can extract the Boolean formulas

$$\begin{aligned}(s \diamond b)_1 &= s_1\bar{s}_2 + s_1\bar{b}_1 + \bar{s}_2 b_1 \\ (s \diamond b)_2 &= \bar{s}_1 s_2 + \bar{s}_1 b_2 + s_2 \bar{b}_2 \\ \mathrm{out}(s, b)_1 &= \bar{s}_1 b_2 + \bar{s}_2 b_1 + b_1 b_2 \\ \mathrm{out}(s, b)_2 &= s_1 b_2 + s_2 b_1 + b_1 b_2\,.\end{aligned}$$



π_1 π_2 π_3 π_4 π_5 π_6 π_7 π_8 π_9 π_{10} π_{11} π_{12} π_{13} π_{14} π_{15} π_{16}
d_1 d_2 d_3 d_4 d_5 d_6 d_7 d_8 d_9 d_{10} d_{11} d_{12} d_{13} d_{14} d_{15} d_{16}

Fig. 7. $\mathrm{PPC}^{(3)}(C, T_4)$. Right recursion steps R_r are marked with dark gray, left recursion steps with light gray. The step at the root (above) and aggregation trees (below) are not marked explicitly. Duplicated gates are depicted in a layered fashion. Dashed lines indicate that a wire is not participating in a recursive step.

In general, realizing a Boolean formula f by replacing negation, multiplication, and addition by inverters, AND, and OR gates, respectively, does not result in a circuit implementing $f_{\mathtt{M}}$.[1] However, we can easily verify that the above formulas are disjunctions of all prime implicants of their respective functions. As shown in [10] (see also [16]),[2] in this special case the resulting circuits do implement the closure—provided the gates behave as in Table 3, which the implementations given in Fig. 1 do by Theorem 3.14. Using distributive laws (recall that these also hold in Kleene logic), the above formulas can be rewritten as

$$\begin{aligned}(s \diamond b)_1 &= s_1(\bar{s}_2 + \bar{b}_1) + \bar{s}_2 b_1\\ (s \diamond b)_2 &= s_2(\bar{s}_1 + \bar{b}_2) + \bar{s}_1 b_2\\ \mathrm{out}(s,b)_1 &= b_1(b_2 + \bar{s}_2) + b_2 \bar{s}_1\\ \mathrm{out}(s,b)_2 &= b_2(b_1 + s_1) + b_1 s_2\,.\end{aligned}$$

We see that, in fact, a single circuit with suitably wired (and possibly negated) inputs can implement all four operations. As for $\mathrm{sel}_1 = \overline{\mathrm{sel}_2}$ the circuit implements a multiplexer with select bit sel_1, we refer to it as *extended multiplexer*, or xmux for short. Its functionality is specified by

$$\mathrm{XMUX}(\mathrm{sel}_1, \mathrm{sel}_2, x, y) := y(x + \mathrm{sel}_2) + x\mathrm{sel}_1\,.$$

Fig. 9 shows the resulting circuit, and Table 11 lists how to map inputs to compute $\diamond_{\mathtt{M}}$ and $\mathrm{out}_{\mathtt{M}}$.

We note that this circuit is not a particularly efficient XMUX implementation; a transistor-level implementation would be much smaller. However, our goal here is to verify correctness and give some initial indication of the size of the resulting circuits—a fully optimized ASIC circuit is beyond the scope of this article. In [4], the size of the implementation is slightly reduced by moving negations. Due to space limitations, we refrain from detailing this modification here, but note that Fig. 12 and Table 12 take it into account.

6.2 Putting it All Together

We now have all the pieces in place to assemble a containing $2\text{-sort}(B)$ circuit. By Theorem 4.3, $\diamond_{\mathtt{M}}$ is associative. Thus, from a given implementation of $\diamond_{\mathtt{M}}$ (e.g., two copies of the circuit from Fig. 9 with appropriate wiring and negation, cf. Table 11) we can construct $\mathrm{PPC}_{\diamond_{\mathtt{M}}}(B-1)$ circuits of small depth and size, as shown in Section 5. We can combine such a circuit with an $\mathrm{out}_{\mathtt{M}}$ implementation (again, two XMUX es with appropriate wiring and negation will do) as shown in Fig. 10 to obtain our $2\text{-sort}(B)$ circuit.

6.3 Simulation Setup

We implemented the design given in Fig. 10 on register-transfer-level using the $\mathrm{PPC}_{\diamond_{\mathtt{M}}}(B-1)$ circuit given by Theorem 5.6 for $k = 0$.[3] *Quartus* by Altera is used for design entry, which in our case mainly consists of checking correct implementation. After design entry we use *ModelSim* by Altera for behavioral simulation. Note that we must not simulate the preprocessed Quartus output, because processing may compromise metastability-containing behavior. Instead, we simulate pure VHDL. Metastable signals are simulated using VHDL signal X, because its behavior matches the worst-case behavior assumed for M.

The correctness of this construction follows from Theorems 4.7 and 4.8, where we can plug in any $\mathrm{PPC}_{\diamond_{\mathtt{M}}}(B-1)$ circuit, cf. Section 5. For the circuits derived by relying on the

1. For instance, $(s \diamond b)_1 = s_1\bar{b}_1 + \bar{s}_2 b_1$ as Boolean formula, but the two expressions differ when evaluated on $s_1 = \bar{s}_2 = 1$ and $b_1 = \mathtt{M}$. The circuits resulting from the different formulas are implementations of a multiplexer (with select bit b_1) and its closure, respectively.

2. Alternatively, one can manually verify that these formulas evaluate to the truth tables given in Tables 8 and 9.

3. For $k > 0$, fan-out becomes an issue, requiring the more involved constructions provided by Theorem 5.16. However, the resulting numbers would be inaccurate, and a detailed comparison based on optimized ASIC implementations is beyond the scope of this work.

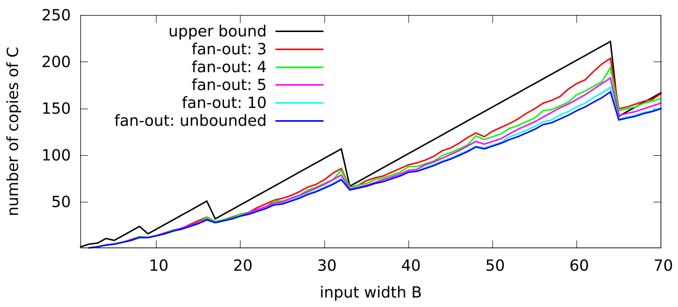


Fig. 8. Dependence of the size of the modified construction on f. For comparison, the upper bound from Corollary 5.5 on the circuit with unbounded fan-out is shown as well.

TABLE 10
Example Run of the FSM in Fig. 2 on Inputs $g = 101010110$ and $h = 101\mathtt{M}10000$

i	0	1	2	3	4	5	6	7	8	9
g_ih_i		11	00	11	0M	11	00	10	10	00
$s_{\mathtt{M}}^{(i)}$	00	11	11	00	0M	M1	M1	01	01	
$g'_ih'_i$		11	00	11	M0	11	00	01	01	00

We drop $s_{\mathtt{M}}^{(9)}$, as it is not needed to compute $g'_9h'_9$.

XMUX circuit from Fig. 9, we independently confirmed this via simulation.

6.4 Results

For the implementation of $\mathrm{PPC}_{\diamond_\mathtt{M}}(B-1)$ we used the circuits from Theorem 5.6, i.e., we did not make use of the extension to constant fan-out. Fig. 11 shows how a non-containing implementation can fail. We exhaustively checked the design from Fig. 10 for B up to 12 (and all feasible k). Simulation shows that the design works correctly for several levels of recursion, e.g., when regarding $B = 1$ and $B = 2$ as simple base cases, $B = 12$ implies 3 levels of recursion for both patterns. We refrained from simulating the constant fan-out construction, because it simply replicates intermediate results without changing functionality.

6.5 Comparison to Baseline

After behavioral simulation, we continue with a comparison of our design and a standard sorting approach $\mathrm{Bin\text{-}comp}(B)$. As mentioned earlier, the $2\text{-sort}(B)$ implementation given in

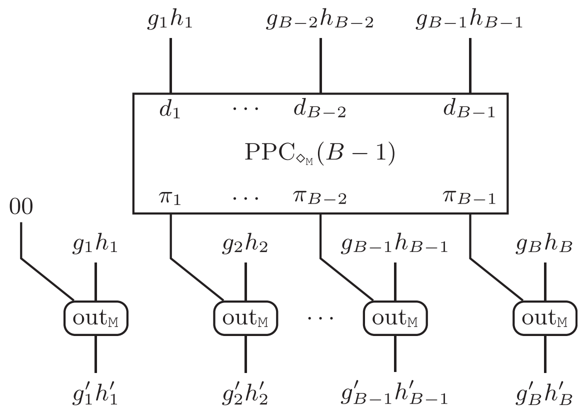


Fig. 10. Constructing $2\text{-sort}(B)$ from $\mathrm{PPC}_{\diamond_\mathtt{M}}(B-1)$ and $\mathrm{out}_\mathtt{M}$.

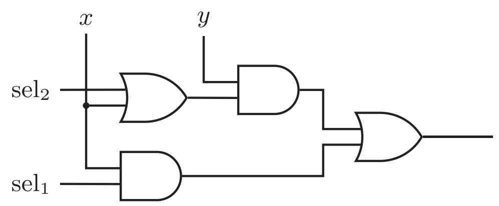


Fig. 9. XMUX circuit, used to implement $\diamond_\mathtt{M}$ and $\mathrm{out}_\mathtt{M}$.

TABLE 11
Wiring an XMUX to Compute the Various Operators

sel_1	sel_2	x	y	$\mathrm{XMUX}(\mathrm{sel}_1, \mathrm{sel}_2, a, b)$
b_1	$\bar{b}_1$	$\bar{s}_2$	s_1	$(s \diamond_\mathtt{M} b)_1$
b_2	$\bar{b}_2$	$\bar{s}_1$	s_2	$(s \diamond_\mathtt{M} b)_2$
$\bar{s}_1$	$\bar{s}_2$	b_2	b_1	$\mathrm{out}_\mathtt{M}(s,b)_1$
s_2	s_1	b_1	b_2	$\mathrm{out}_\mathtt{M}(s,b)_2$

Fig. 10 is slightly optimized by pulling out a negation from the operators in every recursive step [4].

After design entry as described above, we use *Encounter RTL Compiler* for synthesis and *Encounter* for place and route. Both tools are part of the Cadence tool set and in both steps we use NanGate 45 nm Open Cell Library as a standard cell library.

Since metastability-containing circuits may include additional gates that are not required in traditional Boolean logic, Boolean optimization may compromise metastability-containing properties [3]. Accordingly, we were forced to disable optimization during synthesis of the circuits.

Binary Benchmark Bin-comp. In short, Bin-comp consists of a simple VHDL statement comparing two binary encoded inputs and outputting the maximum and the minimum, accordingly. It follows the same design process as 2-sort, but then undergoes optimization using a more powerful set of basic gates. For example, the standard cell library provides prebuild multiplexers. These multiplexers are used by Bin-comp, but not by 2-sort, as they are not metastability-containing. We stress that these more powerful gates provide optimized implementations of multiple Boolean functions, yet each of them is still counted as a single gate. Thus, comparing our design to the binary design in terms of gate count, area, and delay disfavors our solution. Moreover, we noticed that the optimization routine switches to employing

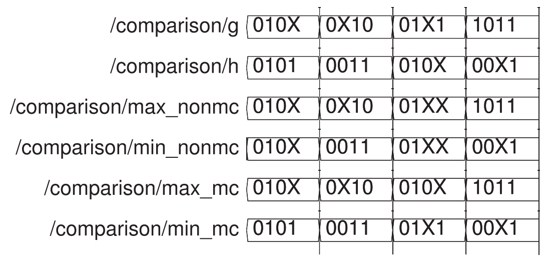

```
/comparison/g           010X  0X10  01X1  1011
/comparison/h           0101  0011  010X  00X1
/comparison/max_nonmc   010X  0X10  01XX  1011
/comparison/min_nonmc   010X  0011  01XX  00X1
/comparison/max_mc      010X  0X10  010X  1011
/comparison/min_mc      0101  0011  01X1  00X1
```

Fig. 11. Excerpt from a simulation for 4-bit inputs, where $X = M$. The rows show (from top to bottom) the inputs g and h, both outputs of the simple non-containing circuit, and both outputs of our design. As inputs g and h we randomly generated valid strings. Columns 1 and 3 show that the simpler design fails to implement a $2\text{-sort}(4)$ circuit.

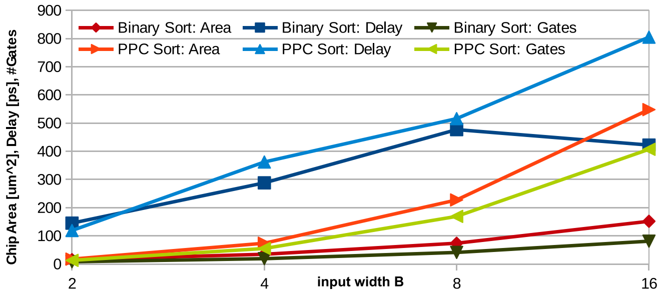


Fig. 12. Comparison of our solution PPC Sort to a standard non-containing one. For the latter, the unexpected delay reduction at $B = 16$ is the result of automatic optimization with more powerful gates, which our solution does not use.

more powerful gates when going from $B = 8$ to $B = 16$ (cf. Fig. 12), resulting in a *decrease* of the delay of the Bin-comp implementation.

Nonetheless, our design performs comparably to the non-containing binary design in terms of delay, cf. Fig. 12 and Table 12. This is quite notable, as further optimization is possible by optimizing our design on the transistor level, with significant expected gains. The same applies to gate count and area, where a notable gap remains. Recall, however, that the Bin-comp design hides complexity by using more advanced gates and does not contain metastability.

We emphasize that we refrained from optimizing the design by making use of all available gates or devising transistor-level implementations, as such an approach is tied to the utilized library or requires design of standard cells.

7 Conclusions

In this work, we demonstrated that efficient metastability-containing sorting circuits are possible. Our results indicate that optimized implementations can achieve the same delay as non-containing solutions, without a dramatic increase in circuit size. This is of high interest to an intended application motivating us to design MC sorting circuits: fault-tolerant high-frequency clock synchronization. Sorting is a key step in envisioned implementations (cf. [10], [15]) of the Lynch-Welch algorithm [30] with improved precision of synchronization. The complete elimination of synchronizer delay is possible due to the efficient MC sorting networks presented in this article; enabling an increment of the rate at which clock corrections are applied, significantly reducing the negative impact of phase drift of local clock sources on the precision of the algorithm (cf. [18]).

This goal will necessitate to devise optimized ASIC implementations of our circuits. The novel PPC circuits we devised in Section 5 are an important contribution towards this end. Note that it is crucial to take into account both depth and fan-out for devising low-delay circuits. Hence, follow-up work needs to compare the existing and our novel design based on suitable metrics that take both into account to reliably predict the achieved trade-offs between delay, area, and energy consumption of circuits. Note that this is of relevance beyond the specific application of MC sorting: PPC circuits lie at the heart of adder designs, implying that even a minor improvement can have significant impact on the overall performance of computing devices!

MC Control Loops. More generally speaking, MC circuits like those presented here are of interest in mixed-signal control loops whose performance depends on very short response times. When analog control is not desirable, traditional solutions incur synchronizer delay before being able to react to any input change. Using MC logic saves the time for synchronization, while metastability of the output corresponds to the initial uncertainty of the measurement; thus, the same quality of the computational result can be achieved in shorter time. Note that our circuits are purely combinational, so they can be used in both clocked and asynchronous control logic.

Obvious examples of such control loops are clock synchronization circuits, but MC has been shown to be useful for adaptive voltage control [13] and fast routing with an acceptable low probability of data corruption [29] as well. This type of application suggests to explore whether efficient circuits exist for a wider range of arithmetic operations, like e.g., addition or (possibly approximate) multiplication.

Redundant Encoding and Addition. On the theoretical side, our results are to be contrasted with the exponential gap between the size of non-containing and MC circuits shown in [17]. This work raised the question for which classes of functions small MC circuits exist. Given that Ladner and Fischer proved that the PPC task can be solved efficiently for any constant-sized state machine [23], it was natural to

TABLE 12
Simulation Results for Metastability-Containing Sorting Networks with $n \in \{4, 7, 10\}$ for B-bit Inputs

B	Circuit	4-sort			7-sort			10-sort$_\#$			10-sort$_\text{d}$		
		gates	area	delay	gates	area	delay	gates	area	delay	gates	area	delay
2	our work	65	87.402	357	208	279.741	714	377	506.912	912	403	541.968	833
	Bin-comp	40	77.91	478	128	249.326	953	232	451.815	1284	248	483	1145
4	our work	275	368.641	640	880	1179.528	1014	1595	2137.905	1235	1705	2285.514	1133
	Bin-comp	95	172.935	906	304	553.28	1810	551	1002.848	2429	589	1072.099	2143
8	our work	845	1136.184	1396	2704	3636.08	1921	4901	6590.283	2179	5239	7044.541	2059
	Bin-comp	205	368.641	1475	656	1179.528	2948	1189	2137.905	3945	1271	2285.514	3470
16	our work	2035	2739.961	2069	6512	8767.374	3396	11803	15891.12	4030	12617	16987.194	3844
	Bin-comp	405	530.67	1298	1296	2425.99	2600	2349	4397.085	3474	2511	4700.304	3050

10-sort$_\#$ optimizes gate count [7], 10-sort$_\text{d}$ optimizes depth [6]; for $n \in \{4, 7\}$, the sorting networks are optimal w.r.t. both measures. Simulation results are: (i) number of gates, (ii) postlayout area $[\mu m^2]$ and (iii) prelayout delay $[ps]$.

ask whether this result can be extended to MC computations. In follow-up work, we show that indeed this holds true for any constant-sized FSM [5]. However, when applying this result to addition, unlike for sorting (where the underlying operations are $\max$ and $\min$) uncertainty of inputs adds up. This means that Gray code can support meaningful computations only if the *total* uncertainty of all addends is at most 1.

Accordingly, in [5] we also consider redundant encodings, showing that using k (roughly) redundant bits, an uncertainty of $\lfloor (k+1)/2 \rfloor$ can be tolerated without loss of precision. Combined with the above result on transducers, this yields a meaningful notion of MC addition that allows for efficient circuits. As, essentially, the redundant bits are used as a unary code, it should be straightforward to apply the techniques from this article to obtain efficient sorting circuits with the encoding from [5]. We remark that the encoding from [5] turns out to be identical to that of the output of suitable time-to-digital converters [12], so relaxing their output constraints to achieve better average-case performance would provide valid input for sorting circuits that accept inputs encoded in this manner.

We believe that these results suggest applicability of our techniques to a wide range of mixed-signal control loops and call for future work further exploring to which extend basic arithmetics can be realized by efficient MC circuits.

Acknowledgments

We thank Attila Kinali and the anonymous reviewers for valuable input. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement 716562). This article generalizes and extends work presented at DATE 2018 [4]. This research was supported by the Israel Science Foundation under Grant 867/19.

References

[1] M. Ajtai, J. Komlós, and E. Szemerédi, "An $\mathcal{O}(n \log n)$ sorting network," in *Proc. 15th Annu. ACM Symp. Theory Comput.*, 1983, pp. 1–9.

[2] R. P. Brent and H. T. Kung, "A regular layout for parallel adders," *Trans. Comput.*, vol. C-31, no. 3, pp. 260–264, Mar. 1982.

[3] J. Bund, C. Lenzen, and M. Medina, "Near-optimal metastability-containing sorting networks," in *Proc. Design Automat. Test Eur. Conf. Exhib.*, 2017, pp. 226–231.

[4] J. Bund, C. Lenzen, and M. Medina, "Optimal metastability-containing sorting networks," in *Proc. Design Automat. Test Eur. Conf. Exhib.*, 2018, pp. 521–526.

[5] J. Bund, C. Lenzen, and M. Medina, "Small hazard-free transducers," *CoRR*, vol. abs/1811.12369, 2018.

[6] D. Bundala and J. Závodnỳ, "Optimal sorting networks," in *Proc. Int. Conf. Lang. Automata Theory Appl.*, 2014, pp. 236–247.

[7] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "25 comparators is optimal when sorting 9 inputs (and 29 for 10)," in *Proc. IEEE 26th Int. Conf. Tools Artif. Intell.*, 2014, pp. 186–193.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd Edition. Cambridge, MA, USA: MIT Press, 2009.

[9] S. Friedrichs, "Metastability-containing circuits, parallel distance problems, and terrain guarding," Ph.D. dissertation, Fac. Math. Comput. Sci., Saarland Univ., Saarbrücken, Germany, 2017.

[10] S. Friedrichs, M. Függer, and C. Lenzen, "Metastability-containing circuits," *IEEE Trans. Comput.*, vol. 67, no. 8, pp. 1167–1183, 2018.

[11] S. Friedrichs and A. Kinali, "Efficient metastability-containing multiplexers," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2017, pp. 332–337.

[12] M. Függer, A. Kinali, C. Lenzen, and T. Polzer, "Metastability-aware memory-efficient time-to-digital converters," in *Proc. 23rd IEEE Int. Symp. Asynchronous Circuits Syst.*, 2017, pp. 49–56.

[13] M. Függer, A. Kinali, C. Lenzen, and B. Wiederhake, "Fast all-digital clock frequency adaptation circuit for voltage droop tolerance," in *Proc. 24rd IEEE Int. Symp. Asynchronous Circuits Syst.*, 2018, pp. 68–77.

[14] R. Ginosar,"Metastability and synchronizers: A tutorial," *Design Test Comput.*, vol. 28, no. 5, pp. 23–35, 2011.

[15] F. Huemer, A. Kinali, and C. Lenzen, "Fault-tolerant clock synchronization with high precision," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI*, 2016, pp. 490–495.

[16] D. A. Huffman,"The design and use of hazard-free switching networks," *J. ACM*, vol. 4, no. 1, pp. 47–62, 1957.

[17] C. Ikenmeyer, B. Komarath, C. Lenzen, V. Lysikov, A. Mokhov, and K. Sreenivasaiah,"On the complexity of hazard-free circuits," in *Proc. STOC*, 2018, pp. 878–889.

[18] P. Khanchandani and C. Lenzen, "Self-stabilizing byzantine clock synchronization with optimal precision," *Theory Comput. Syst.*, vol. 63, pp. 261–304, 2018.

[19] D. J. Kinniment, *Synchronization and Arbitration in Digital Systems*. Hoboken, NJ, USA: Wiley Publishing, 2008.

[20] S. C. Kleene, *Introduction to Metamathematics*. Amsterdam, The Netherlands: North Holland, 1952.

[21] D. E. Knuth, *The Art of Computer Programming Volume 3: Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[22] P. M. Kogge and H. S. Stone, "A parallel algorithm for the efficient solution of a general class of recurrence equations," *IEEE Trans. Comput.*, vol. C-22, no. 8, pp. 786–793, Aug. 1973.

[23] R. E. Ladner and M. J. Fischer,"Parallel prefix computation," *J. ACM*, vol. 27, no. 4, pp. 831–838, 1980.

[24] C. Lenzen and M. Medina, "Efficient metastability-containing gray code 2-sort," in *Proc. 22nd IEEE Int. Symp. Asynchronous Circuits Syst.*, 2016, pp. 49–56.

[25] L. Marino, "General theory of metastable operation," *IEEE Trans. Comput.*, vol. C-30, no. 2, pp. 107–115, Feb. 1981.

[26] J. Sklansky,"Conditional-sum addition logic," *IRE Trans. Electron. Comput.*, vol. EC-9, no. 2, pp. 226–231, 1960.

[27] E. E. Swartzlander and C. E. Lemonds, *Computer Arithmetic*, vol. 1. Singapore: World Scientific, 2015.

[28] G. Tarawneh and A. Yakovlev, "An RTL method for hiding clock domain crossing latency," in *Proc. 19th IEEE Int. Conf. Electronics, Circuits, Syst.*, 2012, pp. 540–543.

[29] G. Tarawneh, M. Függer, and C. Lenzen, "Metastability tolerant computing," in *Proc. ASYNC*, 2017, pp. 25–32.

[30] J. L. Welch and N. A. Lynch, "A new fault-tolerant algorithm for clock synchronization," *Inf. Comput.*, vol. 77, no. 1, pp. 1–36, 1988.

[31] R. Zimmermann, *Binary Adder Architectures for Cell-Based VLSI and their Synthesis*," Citeseer, 1998.

Johannes Bund received the B Sc degree from Saarland Informatics Campus, in 2017. He is working toward the PhD degree in the Algorithms and Complexity Department, MPI for Informatics. In 2018 he joined Christoph Lenzen's group at MPI for Informatics while doing his MSc studies at Saarland University.

Christoph Lenzen received the diploma degree in mathematics from the University of Bonn, in 2007 and the PhD degree from ETH Zurich, in 2011. After postdoc positions with the Hebrew University of Jerusalem, the Weizmann Institute of Science, and MIT, he became group leader with MPI for Informatics in 2014. He received the best paper award at PODC 2009, the ETH medal for his dissertation, and in 2017 an ERC starting grant.

Moti Medina received the BSc, MSc, and PhD degrees from the School of Electrical Engineering, Tel-Aviv University, in 2014, 2009, and 2007 respectively. He is a faculty member with the Ben-Gurion University of the Negev since 2017. Previously, he was a post-doc researcher with MPI for Informatics and in the Algorithms and Complexity group at LIAFA (Paris 7). He is also a co-author of a text-book on logic design "Digital Logic Design: A Rigorous Approach", Cambridge Univ. Press, October 2012.