Optimal Metastability-Containing Sorting Networks

Johannes Bund
Saarland Informatics Campus
MPI for Informatics, Germany

Christoph Lenzen
Saarland Informatics Campus
MPI for Informatics, Germany

Moti Medina
Dept. of Electrical & Computer Engineering
Ben-Gurion University of the Negev, Israel

***Abstract*—When setup/hold times of bistable elements are violated, they may become metastable, i.e., enter a transient state that is neither digital 0 nor 1 [1]. In general, metastability cannot be avoided, a problem that manifests whenever taking discrete measurements of analog values. Metastability of the output then reflects uncertainty as to whether a measurement should be rounded up or down to the next possible measurement outcome.**

Surprisingly, Lenzen & Medina (ASYNC 2016) showed that metastability can be *contained*, i.e., measurement values can be correctly sorted *without* resolving metastability first. However, both their work and the state of the art by Bund et al. (DATE 2017) leave open whether such a solution can be as small and fast as standard sorting networks. 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. Concretely, for 10-channel sorting networks and 16-bit wide inputs, we improve by 48.46% in delay and by 71.58% in area over Bund et al. Our simulations indicate that straightforward transistor-level optimization is likely to result in performance on par with standard (non-containing) solutions.

I. Introduction

Metastability is one of the basic obstacles when crossing clock domains, potentially resulting in soft errors with critical consequences [2]. As it has been shown that there is no deterministic way of avoiding metastability [1], synchronizers [3] are employed to reduce the error probability to tolerable levels. Besides energy and chip area, this approach costs time: the more time is allocated for metastability resolution, the smaller is the probability of a (possibly devastating) metastability-induced fault.

Recently, a different approach has been proposed, coined *metastability-containing* (MC) circuits [4]. The idea is to accept (a limited amount of) metastability in the input to a digital circuit and guarantee limited metastability of its output, such that the result is still useful. The authors of [5], [6] apply this approach to a fundamental primitive: sorting. However, the state-of-the-art [5] are circuits that are by a $\Theta(\log B)$ factor larger than non-containing solutions, where B is the bit width of inputs. Accordingly, the authors pose the following question:

"What is the optimum cost of the 2-sort *primitive?"*

We argue that answering this question is critical, as the performance penalty imposed by current MC sorting primitives is not outweighed by the avoidance of synchronizers.

Our Contribution: We answer the above question by providing a B-bit MC 2-sort circuit of depth $O(\log B)$ and $O(B)$ gates. Trivially, any such building block with gates of constant fan-in must have this asymptotic depth and gate count, and it improves by a factor of $\Theta(\log B)$ on the gate complexity of [5]. Furthermore, we provide optimized building blocks that significantly improve the leading constants of these complexity bounds. See Figure 1 for our improvements over prior work; specifically, for 16-bit inputs, area and delay decrease by up to 71.58% and 48.46% respectively.

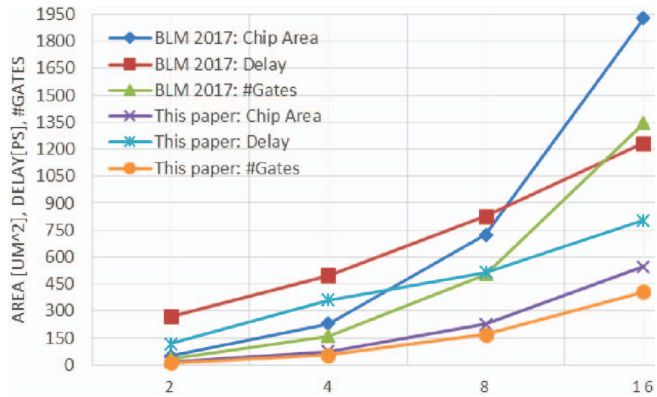

Fig. 1. Area, delay, and gate count of 2-sort(B) for $B \in \{2, 4, 8, 16\}$; we compare our solution to [5].

Plugging our circuit into (optimal depth or size) sorting networks [7], [8], [9], we obtain efficient combinational metastability-containing sorting circuits, cf. Table VIII. In general, plugging our 2-sort circuit into an n-channel sorting network of depth $O(\log n)$ with $O(n \log n)$ 2-sort elements [10], we obtain an asymptotically optimal MC sorting network of depth $O(\log B \log n)$ and $O(Bn \log n)$ gates.

Further Related Work: Ladner and Fischer [11] studied the problem of computing all the prefixes of applications of an associative operator on an input string of length n. They designed and analyze a recursive construction which computes all these prefixes in parallel. The resulting parallel prefix computation (PPC) circuit has depth of $O(\log n)$ and gate count of $O(n)$ (assuming that the implementation of the associative operator has constant size and constant depth). We make use of their construction as part of ours.

II. 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.

We use the following basic notation. For $N \in \mathbb{N}$, we set $[N] := \{0, \dots, N-1\}$. For a binary B-bit string g,

978-3-9819263-0-9/DATE18/©2018 EDAA

TABLE I
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

denote by g_i its i-th bit, i.e., $g = g_1 g_2 \dots g_B$. We use the shorthand $g_{i,j} := g_i \dots g_j$. Let $\mathrm{par}(g)$ denote the parity of g, i.e, $\mathrm{par}(g) = \sum_{i=1}^{B} g_i \bmod 2$.

Reflected Binary Gray Code: Due to possible metastability of inputs, we use Gray code. Denote by $\langle\cdot\rangle$ the decoding function of a Gray code string, i.e., for $x \in [N]$, $\langle \mathrm{rg}_B(x)\rangle = x$. As each B-bit string is a codeword, the code is a bijection and the decoding function also defines the encoding function $\mathrm{rg}_B : [N] \to \{0,1\}^B$. We define B-bit binary reflected Gray code recursively, where a 1-bit code is given by $\mathrm{rg}_1(0) = 0$ and $\mathrm{rg}_1(1) = 1$. For $B > 1$, we start with the first bit fixed to 0 and counting with $\mathrm{rg}_{B-1}(\cdot)$ (for the first $2^{B-1}-1$ codewords), then toggle the first bit to 1, and finally "count down" $\mathrm{rg}_{B-1}(\cdot)$ while fixing the first bit again, cf. Table I. Formally, this yields

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

We define the maximum and minimum of two binary reflected Gray code strings, $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$ respectively, in the usual way, as follows. For two binary reflected Gray code strings $g, h \in \{0,1\}^B$, $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$ are defined as

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

Valid Strings: In [6], the authors represent metastable "bits" by $\mathtt{M}$. 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 2.1 (The $*$ operator [6]): For $B \in \mathbb{N}$, define the operator $* : \{0,1,\mathtt{M}\}^B \times \{0,1,\mathtt{M}\}^B \to \{0,1,\mathtt{M}\}^B$ by

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

Observation 2.2: The operator $*$ is associative and commutative. Hence, for a set $S = \{x^{(1)}, \dots, x^{(k)}\}$ of B-bit strings, we can use the shorthand $* S := *_{x \in S}\, x := x^{(1)} * x^{(2)} * \dots * x^{(k)}$. 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 II.

Definition 2.3 (Valid Strings [6]): Let $B \in \mathbb{N}$ and $N = 2^B$. Then, the set of *valid strings of length* B is

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

TABLE II
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	–	–	–

where for a set A we abbreviate $f(A) := \{f(y) \mid y \in A\}$. As pointed out in [5], inputs that are valid strings may, e.g., arise from using suitable time-to-digital converters for measuring time differences [12].

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* [4], which in turn makes use of the *resolution.*

Definition 2.4 (Resolution [4]): For $x \in \{0,1,\mathtt{M}\}^B$,

$$\mathrm{res}(x) := \{y \in \{0,1\}^B | \forall i \in \{1,\dots,B\} \colon x_i \neq \mathtt{M} \Rightarrow y_i = x_i\}.$$

Thus, $\mathrm{res}(x)$ is the set of all strings obtained by replacing all $\mathtt{M}$s in x by either 0 or 1: $\mathtt{M}$ acts as a "wild card." 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 2.5 (The $\mathtt{M}$ Closure [4]): Given an operator $f \colon \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$, its *metastable closure* $f_{\mathtt{M}} \colon \{0,1,\mathtt{M}\}^n \times \{0,1,\mathtt{M}\}^n \to \{0,1,\mathtt{M}\}^n$ is defined by $f_{\mathtt{M}}(x) := * f(\mathrm{res}(x))$.

Output Specification: We want to construct a circuit that outputs the maximum and minimum of two valid strings, which will enable 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 $\mathrm{rg}_B(x) * \mathrm{rg}_B(x+1)$ 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 $x+1$. This means that the measurement the string represents was taken of a value somewhere between x and $x+1$. Moreover, if we wait for metastability to resolve, the string will stabilize to either $\mathrm{rg}_B(x)$ or $\mathrm{rg}_B(x+1)$. Accordingly, it makes sense to consider $\mathrm{rg}_B(x) * \mathrm{rg}_B(x+1)$ "in between" $\mathrm{rg}_B(x)$ and $\mathrm{rg}_B(x+1)$, resulting in the total order on valid strings given by Table II.

The above intuition can be formalized by extending $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$ to valid strings using the metastable closure.

Definition 2.6 ([5], [6]): For $B \in \mathbb{N}$, a 2-sort(B) circuit is specified as follows.

- ***Input:*** $g, h \in \mathcal{S}^B_{\mathrm{rg}}$,
- ***Output:*** $g', h' \in \mathcal{S}^B_{\mathrm{rg}}$,
- ***Functionality:*** $g' = \max^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}$, $h' = \min^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}$.

As shown in [5], this definition indeed coincides with the one given in [6], and for valid strings g and h, $\max^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}$ and $\min^{\mathrm{rg}}_{\mathtt{M}}\{g,h\}$ are valid strings, too. More specifically, $\max^{\mathrm{rg}}_{\mathtt{M}}$

TABLE III
LOGICAL EXTENSIONS TO METASTABLE INPUTS OF AND (LEFT), OR (CENTER), AND AN INVERTER (RIGHT).

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

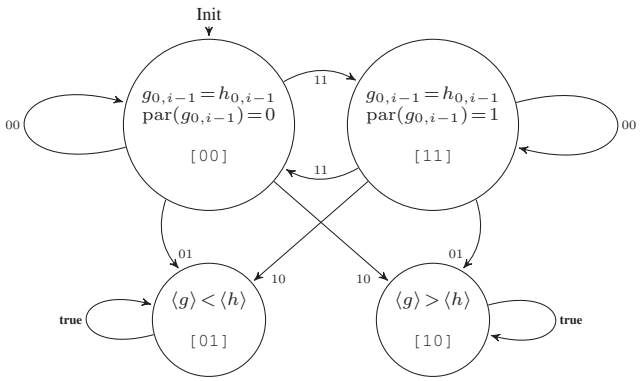

Fig. 2. Finite state automaton determining which of two Gray code inputs $g, h \in \{0,1\}^B$ is larger. In each step, the machine receives $g_i h_i$ as input. State encoding is given in square brackets.

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

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

Computational Model: We seek to use standard components and combinational logic only. We use the model of [4], which specifies the behavior of basic gates on metastable inputs via the metastable closure of their behavior on binary inputs. For standard implementations of AND and OR gates, this assumption is valid: if M represents an arbitrary, possibly time-dependent voltage between logical 0 and 1, an AND gate will still output logical 0 if the respective other input is logical 0. Similarly, an OR gate with one input being logical 1 suppresses metastability at the other input, cf. Table III.

As pointed out in [5], any additional reduction of metastability in the output necessitates the use of non-combinational masking components (e.g., masking registers), analog components, and/or synchronizers, all of which are outside of our computational model. Moreover, other than the usage of analog components, these alternatives require to spend additional time, which we avoid in this paper.

III. PRELIMINARIES ON STABLE INPUTS

Comparing Stable Gray Code Strings via an FSM: The following basic structural lemma leads to a straightforward way of comparing binary reflected Gray code strings.

Lemma 3.1: Let $g, h \in \{0,1\}^B$ such that $\langle g\rangle > \langle h\rangle$. Denote by $i \in \{1,\ldots,B\}$ the first index such that $g_i \neq h_i$. Then $g_i = 1$ (i.e., $h_i = 0$) if $\mathrm{par}(g_{1,i-1}) = 0$ and $g_i = 0$ (i.e., $h_i = 1$) if $\mathrm{par}(g_{1,i-1}) = 1$.

TABLE IV
COMPUTING $\max^{\mathrm{rg}}\{g,h\}_i$ AND $\min^{\mathrm{rg}}\{g,h\}_i$.

$s^{(i-1)}$	$\max^{\mathrm{rg}}\{g,h\}_i$	$\min^{\mathrm{rg}}\{g,h\}_i$
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 V
THE $\diamond$ OPERATOR AND THE out OPERATOR. THE FIRST OPERAND IS THE CURRENT STATE, THE SECOND THE NEXT INPUT BITS.

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

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

Lemma 3.1 gives rise to a sequential representation of 2-sort(B) as a Finite state machine (FSM), for input strings in $\{0,1\}^B$. Consider the state machine given in Figure 2. Its four states keep track of whether $g_{1,i} = h_{1,i}$ with parity 0 (state encoding: 00) or 1 (state encoding: 11), respectively, $\langle g\rangle < \langle h\rangle$ (state encoding: 01), or $\langle g\rangle > \langle h\rangle$ (state encoding: 10). Denoting by $s^{(i)}$ its state after i steps (where $s^{(0)} = 00$ is the initial state), Lemma 3.1 shows that the output given in Table IV is correct: up to the first differing bits $g_i \neq h_i$, the (identical) input bits are reproduced both for $\max^{\mathrm{rg}}$ and $\min^{\mathrm{rg}}$, and in the i-th step the state machine transitions to the correct absorbing state.

The $\diamond$ Operator and Optimal Sorting of Stable Inputs: We can express the transition function of the state machine as an operator $\diamond$ taking the current state and input $g_i h_i$ as argument and returning the new state. Then $s^{(i)} = s^{(i-1)} \diamond g_i h_i$, where $\diamond$ is given in Table V.

Observation 3.2: $\diamond$ is associative, that is, $\forall a,b,c \in \{0,1\}^2\colon (a \diamond b) \diamond c = a \diamond (b \diamond c)$. We thus have that $s^{(i)} = \bigdiamond_{j=1}^{i} g_j h_j := g_1 h_1 \diamond g_2 h_2 \diamond \ldots \diamond g_i h_i$, regardless of the order in which the $\diamond$ operations are applied.

An immediate consequence is that we can apply the results by [11] on parallel prefix computation to derive an $O(B)$-gate circuit of depth $O(\log B)$ computing all s_i, $i \in [B]$, in parallel. Our goal in the following sections is to extend this well-known approach to potentially metastable inputs.

IV. DEALING WITH METASTABLE INPUTS

Our strategy is the same as outlined in Section III for stable inputs, where we replace all involved operators by their metastable closure: (i) compute $s^{(i)}$ for $i \in [B]$, (ii) determine $\max^{\mathrm{rg}}\{g,h\}_i$ and $\min^{\mathrm{rg}}\{g,h\}_i$ according to Table IV for $i \in \{1,\ldots,B\}$, and (iii) exploit associativity of the operator computing the $s^{(i)}$ to determine all of them concurrently with $O(\log B)$ depth and $O(B)$ gates (using [11]). To make this work for inputs that are valid strings, we simply replace all involved operators by their respective metastable closure. Thus, we only need to implement $\diamond_{\mathtt{M}}$ and the closure of the operator given in Table IV (both of constant size) and immediately obtain an efficient circuit using the PPC framework [11].

Unfortunately, it is not obvious that this approach yields correct outputs. There are three hurdles to take: (i) Show that first computing $s_{\mathtt{M}}^{(i)}$ and then the output from this and the input yields correct output for all valid strings. (ii) Show that $\diamond_{\mathtt{M}}$ behaves like an associative operator on the given inputs (so we can use the PPC framework). (iii) Show that repeated application of $\diamond_{\mathtt{M}}$ actually computes $s_{\mathtt{M}}^{(i)}$.

Killing two birds with one stone, we first show the second and third point in a single inductive argument. We then proceed to prove the first point.

A. Determining $s_{\mathtt{M}}^{(i)}$

Note that for any x and y, we have that $\mathrm{res}(xy) = \mathrm{res}(x) \times \mathrm{res}(y)$. Hence, for valid strings $g, h \in \mathcal{S}_{\mathrm{rg}}^B$ and $i \in \{1, \ldots, B\}$, we have that $s_{\mathtt{M}}^{(i)} = * \bigdiamond_{j=1}^{i} \mathrm{res}(g_j h_j)$, and for convenience set $s_{\mathtt{M}}^{(0)} := s^{(0)} = 00$. Moreover, recalling Definition 2.5,

$$x \diamond_{\mathtt{M}} y = \underset{x'y' \in \mathrm{res}(xy)}{*} \{x' \diamond y'\} = *(\mathrm{res}(x) \diamond \mathrm{res}(y)). \quad (1)$$

The following theorem shows that the desired decomposition is feasible.

Theorem 4.1: Let $g, h \in \mathcal{S}_{\mathrm{rg}}^B$ and $1 \leq i \leq j \leq B$. Then

$$g_i h_i \diamond_{\mathtt{M}} g_{i+1} h_{i+1} \diamond_{\mathtt{M}} \ldots \diamond_{\mathtt{M}} g_j h_j = * \bigdiamond_{k=i}^{j} \mathrm{res}(g_k h_k), \quad (2)$$

regardless of the order in which the $\diamond_{\mathtt{M}}$ operators are applied.

We remark that we did *not* prove that $\diamond_{\mathtt{M}}$ is an associative operator, just that it behaves associatively when applied to input sequences given by valid strings. Moreover, in general the closure of an associative operator needs not be associative. A counter-example is given by binary addition modulo 4:

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

Since $\diamond_{\mathtt{M}}$ behaves associatively when applied to input sequences given by valid strings, we can apply the results by [11] on parallel prefix computation to any implementation of $\diamond_{\mathtt{M}}$.

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

Denote by $\mathrm{out}\colon \{0,1\}^2 \times \{0,1\}^2 \to \{0,1\}^2$ the operator given in Table IV computing $\max^{\mathrm{rg}}\{g,h\}_i \min^{\mathrm{rg}}\{g,h\}_i$ out of $s^{(i-1)}$ and $g_i h_i$. The following theorem shows that, for valid inputs, it suffices to implement $\mathrm{out}_{\mathtt{M}}$ to determine $\max_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i$ and $\min_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i$ from $s_{\mathtt{M}}^{(i-1)}$, g_i, and h_i.

Theorem 4.2: Given valid inputs $g, h \in \mathcal{S}_{\mathrm{rg}}^B$ and $i \in [B]$, it holds that $\mathrm{out}_{\mathtt{M}}(s_{\mathtt{M}}^{(i-1)}, g_i h_i) = \max_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i \min_{\mathtt{M}}^{\mathrm{rg}}\{g,h\}_i$.

V. The Complete Circuit

Section IV breaks the task down to using the PPC framework to compute $s_{\mathtt{M}}^{(i)}$, $i \in [B]$, using $\diamond_{\mathtt{M}}$ and then $\mathrm{out}_{\mathtt{M}}$ to determine the outputs. Thus, we need to provide implementations of $\diamond_{\mathtt{M}}$ and $\mathrm{out}_{\mathtt{M}}$, and apply the template from [11].

A. Implementations of Operators

We provide optimized implementations based on fan-in 2 AND and OR gates and inverters here, cf. Section II. Depending on target architecture and available libraries, more efficient solutions may be available.

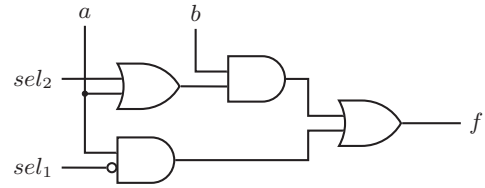

Fig. 3. Selection circuit with inputs: a, b, sel_1, sel_2 and output f, used to implement $\hat{\diamond}_{\mathtt{M}}$ and $\mathrm{out}_{\mathtt{M}}$.

TABLE VI
Connections to a selection circuit to compute the respective operator f.

sel_1	sel_2	a	b	f
$\overline{b_1}$	$\overline{b_1}$	s_2	$\overline{s_1}$	$(s \hat{\diamond}_{\mathtt{M}} b)_1$
b_2	b_2	s_2	$\overline{s_1}$	$(s \hat{\diamond}_{\mathtt{M}} b)_2$
$\overline{s_1}$	s_2	b_1	b_2	$\mathrm{out}_{\mathtt{M}}(s,b)_1$
s_2	$\overline{s_1}$	b_2	b_1	$\mathrm{out}_{\mathtt{M}}(s,b)_2$

Implementing $\diamond_{\mathtt{M}}$: We operate with the inverted first bits of the output of $\diamond_{\mathtt{M}}$. To this end, define $Nx := \overline{x_1} x_2$ for $x \in \{0, 1, \mathtt{M}\}^2$ and set $x \hat{\diamond}_{\mathtt{M}} y := N(Nx \diamond_{\mathtt{M}} Ny)$. We compute $\bar{g}$ and work with inputs $\bar{g}$ and h using operator $\hat{\diamond}_{\mathtt{M}}$. Theorem 4.1 and elementary calculations show that, for valid strings g and h, we have $(\overline{g_1} h_1 \hat{\diamond}_{\mathtt{M}} \overline{g_2} h_2) \hat{\diamond}_{\mathtt{M}} \overline{g_3} h_3 = \overline{g_1} h_1 \hat{\diamond}_{\mathtt{M}} (\overline{g_2} h_2 \hat{\diamond}_{\mathtt{M}} \overline{g_3} h_3)$, i.e., the order of evaluation of $\hat{\diamond}_{\mathtt{M}}$ is insubstantial, just as for $\diamond_{\mathtt{M}}$. Moreover, as intended we get for all $1 \leq i \leq j \leq B$ that $\overline{g_i} h_i \hat{\diamond}_{\mathtt{M}} \ldots \hat{\diamond}_{\mathtt{M}} \overline{g_j} h_j = N(g_i h_i \diamond_{\mathtt{M}} \ldots \diamond_{\mathtt{M}} g_j h_j)$. We concisely express operator $\diamond$ (Table IV) by the following logic formulas, where we already negate the first output bit.

$$\overline{(s \diamond b)}_1 = \overline{s_1} \cdot \left(s_2 + \overline{b_1}\right) + s_2 \cdot b_1$$
$$(s \diamond b)_2 = \overline{s_1} \cdot (s_2 + b_2) + s_2 \cdot \overline{b_2}$$

This gives rise to depth-3 circuits containing in total 4 AND gates, 4 OR gates, and 2 inverters.[1] From the gate behavior specified in Table III, one can readily verify that the circuit also implements $\hat{\diamond}_{\mathtt{M}}$ correctly.[2] Since these circuits are identical to the ones used to compute $\mathrm{out}_{\mathtt{M}}$, we give the implementation of such a selecting circuit once in Figure 3 and describe how to use it in Table VI. We remark that with identical select bits ($sel_1 = sel_2$), this circuit implements a CMUX (a $\mathrm{MUX}_{\mathtt{M}}$ in our terminology) as defined in [4].

Implementing $\mathrm{out}_{\mathtt{M}}$: The multiplication table of $\mathrm{out}_{\mathtt{M}}$, which is equivalent to Table IV, is given in Table V.

We can concisely express the output function given in Table V by the following logic formulas.

$$\mathrm{out}(s,b)_1 = (\overline{s_1} + b_1) \cdot b_2 + \overline{s_2} \cdot b_1$$
$$\mathrm{out}(s,b)_2 = s_1 \cdot b_2 + (s_2 + b_2) \cdot b_1$$

As mentioned before, instead of computing s_1, we determine and use as input $\overline{s_1}$. Thus, the above formulas give rise to

[1] In the base case, where $b_1 = g_i$ for some $i \in \{1, \ldots, B\}$, we can save an additional inverter.

[2] Note that this is not true for arbitrary logic formulas evaluating to $\overline{s \diamond b}$; e.g., $\overline{s \diamond b_1} = (\overline{s_1} + b_1) \cdot \left(s_2 + \overline{b_1}\right)$, but the corresponding circuit outputs $\mathtt{M} \neq \overline{(10 \diamond \mathtt{M}0)}_1 = 0$ for inputs $s = 10$ and $b = \mathtt{M}0$.

depth-3 circuits that contain in total 4 AND gates, 4 OR gates, and 2 inverters (see Figure 3 and Table VI); in fact, the circuit is identical to the one used for $\hat{\diamond}_{\mathrm{M}}$ with different inputs. From the gate behavior specified in Table III, one can readily verify that the circuit indeed also implements $\mathrm{out}_{\mathrm{M}}$.

B. Implementation of $s_{\mathrm{M}}^{(i)}$

We make use of the Parallel Prefix Computation (PPC) framework [11] to efficiently compute $s_{\mathrm{M}}^{(i)}$ in parallel for all $i \in [B]$. This framework requires an associative operator OP. In our case, $OP = \hat{\diamond}_{\mathrm{M}}$, which by Theorem 4.1 is associative on all relevant inputs. Given an implementation of OP, the circuit is recursively constructed as shown in Figure 4, where the base case $n = 1$ is trivial. For n that is a power of 2, the depth and gate counts are given as [13]

$$\begin{aligned} \mathrm{delay}(PPC_{OP}(n)) &= (2\log_2 n - 1) \cdot \mathrm{delay}(OP), \\ \mathrm{cost}(PPC_{OP}(n)) &= (2n - \log_2 n - 2) \cdot \mathrm{cost}(OP)\,. \end{aligned} \tag{3}$$

C. Putting it All Together

Theorem 5.1: The circuit depicted in Figure 5 implements 2-sort(B) according to Definition 2.6. Its delay is $O(\log B)$ and its gate count is $O(B)$.

VI. Simulation Results

Design Flow: Our design flow makes use of the following tools: (i) design entry: Quartus, (ii) behavioral simulation: ModelSim, (iii) synthesis: Encounter RTL Compiler (part of Cadence tool set) with NanGate 45 nm Open Cell Library, (iv) place & route: Encounter (part of Cadence tool set) with NanGate 45 nm Open Cell Library.

Design Flow adaptations for MC: During synthesis the VHDL description of a circuit is automatically mapped to standard cells provided by a standard cell library. The standard cell library used for the experiments provides besides simple AND, OR or Inverter gates also more powerful AOI (And-Or-Invert) gates, which combine multiple boolean connectives and optimize them on transistor level. Since we did not analyze the behaviour of more complex AOI gates in face of metastability, we restrict our implementation to use only AND, OR and

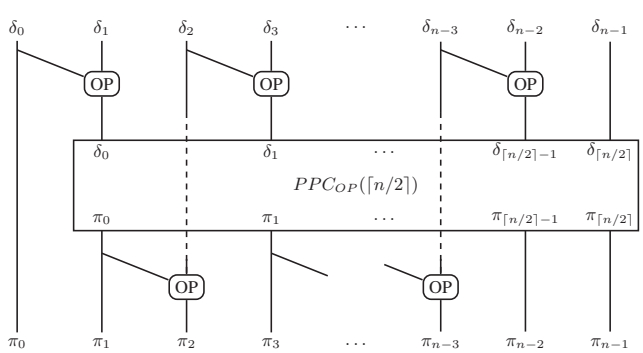

Fig. 4. Recursive construction of $PPC_{OP}(n)$ for odd n, computing $\pi_i = \delta_0 OP \ldots OP \delta_i$. For even n the rightmost input (δ_{n-1}) and output (π_{n-1}) are not present. Dashed lines are not connected to $PPC_{OP}(\lceil n/2\rceil)$.

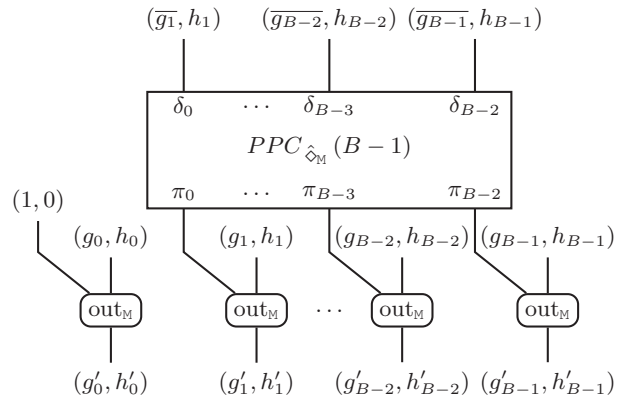

Fig. 5. Construction of 2-sort(B) from out_{M} and $PPC_{\hat{\diamond}_{\mathrm{M}}}(B-1)$. $PPC_{OP}(B-1)$ is specified in Figure 4, and we use the implementations of $\hat{\diamond}_{\mathrm{M}}$ and $\mathrm{out}_{\mathrm{M}}$ specified by Figure 3 and Table VI. For input $Ns^{(0)} = (1,0)$, $\mathrm{out}_{\mathrm{M}}$ reduces to an AND and an OR gate.

TABLE VII
Comparison of gate count, delay, and area of 2-sort(B) from this paper and [5], and Bin-comp, an optimized comparator taking binary inputs.

B	Circuit	# Gates	Area [μm^2]	Delay [ps]
$B = 2$	This paper	13	17.486	119
	[5]	34	49.42	268
	Bin-comp	8	15.582	145
$B = 4$	This paper	55	73.752	362
	[5]	160	230.3	498
	Bin-comp	19	34.58	288
$B = 8$	This paper	169	227.29	516
	[5]	504	723.52	827
	Bin-comp	41	73.752	477
$B = 16$	This paper	407	548.016	805
	[5]	1344	1928.262	1233
	Bin-comp	81	151.648	422

Inverter gates. To ensure this, we performed the mapping to standard cells by hand. The following standard cells have been used to map the logic gates to hardware: (i) INV_X1: Inverter gate, (ii) AND2_X1: AND gate, (iii) OR2_X1: OR gate. In the documentation of the NanGate 45 nm Open Cell Library it can be seen that these cells in fact compute the metastable closure of the respective Boolean connective.

After mapping the design by hand, we can disable the optimization in the synthesis step and go on with place and route. This prevents the RTL Compiler from performing Boolean optimization on the design, which may destroy the MC properties of our circuits.

The binary benchmark: Bin-comp*:* Following [5], we also compare our sorting networks to a standard (non-containing!) sorting design. Bin-comp uses a simple VHDL statement to compare both inputs:

```
1   if (a > b) then
2     greater <= '1';
3   else
4     greater <= '0';
5   end if;
```

Listing 1. VHDL code excerpt of binary comparator

TABLE VIII

SIMULATION RESULTS FOR METASTABILITY-CONTAINING SORTING NETWORKS WITH $n \in \{4, 7, 10\}$ FOR B-BIT INPUTS. $10\text{-sort}_{\#}$ OPTIMIZES GATE COUNT [8], $10\text{-sort}_{\mathrm{d}}$ DEPTH [7]; 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 $[ps]$.

B	Circuit	4-sort			7-sort			$10\text{-sort}_{\#}$			$10\text{-sort}_{\mathrm{d}}$		
		gates	area	delay	gates	area	delay	gates	area	delay	gates	area	delay
2	here	65	87.402	357	208	279.741	714	377	506.912	912	403	541.968	833
	[5]	170	247.016	846	544	790.44	1715	986	1432.62	2285	1054	1531.467	2010
	Bin-comp	40	77.91	478	128	249.326	953	232	451.815	1284	248	483	1145
4	here	275	368.641	640	880	1179.528	1014	1595	2137.905	1235	1705	2285.514	1133
	[5]	800	1151.472	1558	2560	3684.541	3147	4640	6678.294	4207	4960	7138.74	3681
	Bin-comp	95	172.935	906	304	553.28	1810	551	1002.848	2429	589	1072.099	2143
8	here	845	1136.184	1396	2704	3636.08	1921	4901	6590.283	2179	5239	7044.541	2059
	[5]	2520	3617.67	2394	8064	11576.32	4715	14616	20982.542	6252	15624	22429.176	5481
	Bin-comp	205	368.641	1475	656	1179.528	2948	1189	2137.905	3945	1271	2285.514	3470
16	here	2035	2739.961	2069	6512	8767.374	3396	11803	15891.12	4030	12617	16987.194	3844
	[5]	6720	9640.75	3396	21504	30849.875	6415	38976	55916.448	8437	41664	59772.132	7458
	Bin-comp	405	530.67	1298	1296	2425.99	2600	2349	4397.085	3474	2511	4700.304	3050

Each output is connected to a standard multiplexer, where the signal $greater$ is used as the select bit for both multiplexers.

The binary design follows a standard design flow, which uses the tools listed above. In short, $\mathrm{Bin\text{-}comp}$ follows the same design process as 2-sort, but then undergoes optimization using a more powerful set of basic gates.

We emphasize that the more powerful AOI gates combine multiple boolean functions and optimize them on gate level, yet each of them is still counted as one gate. Thus, comparing our design to the binary design in terms of gate count, area, and delay disfavors our solution. Moreover, the optimization routine switches to employing more powerful gates when going from $B = 8$ to $B = 16$ (See Table VIII) resulting in a *decrease* of the delay of the binary implementation.

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

We remark that we refrained from optimizing the design by making use of all available gates or devising transistor-level implementations for two reasons. First, such an approach is tied to the utilized library or requires design of standard cells. Second, it would have been unsuitable for a comparison with [5], which does not employ such optimizations either.

Comparison to State of the Art: Our circuits show large improvements over [5] in all performance measures. Delays, gate counts, and area are all smaller by factors between roughly 1.5 and 3.5. In particular, for $B = 16$ delay is roughly cut in half, while gate count and area decrease by factors of 3 or more.

VII. DISCUSSION

In this paper, we provide asymptotically optimal MC sorting primitives. We achieve this by applying results on parallel prefix computation [11], which requires to establish that the involved operators behave associative on the relevant inputs. Our circuits are purely combinational and are glitch-free (as they are MC). Compared to standard sorting networks, we roughly match delay, but fall behind on gate count and area. However, we used gate-level implementations of $\mathrm{out}_{\mathrm{M}}$ and $\diamond_{\mathrm{M}}$ restricted to AND and OR gates and inverters. Transistor-level implementations, which are a straightforward optimization, would decrease size and delay of the derived circuits further. We expect that this will result in circuits that perform on par with standard sorting networks. In light of these properties, we believe our circuits to be of wide applicability.

Acknowledgements: 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).

REFERENCES

[1] L. Marino, "General Theory of Metastable Operation," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 107–115, 1981.

[2] R. Ginosar, "Metastability and Synchronizers: A Tutorial," *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 23–35, 2011.

[3] D. J. Kinniment, *Synchronization and Arbitration in Digital Systems*. Wiley Publishing, 2008.

[4] S. Friedrichs, M. Függer, and C. Lenzen, "Metastability-Containing Circuits," *CoRR*, vol. abs/1606.06570, 2016.

[5] J. Bund, C. Lenzen, and M. Medina, "Near-Optimal Metastability-Containing Sorting Networks," in *(DATE)*, 2017.

[6] C. Lenzen and M. Medina, "Efficient metastability-containing gray code 2-sort," in *(ASYNC)*, 2016, pp. 49–56.

[7] D. Bundala and J. Závodnỳ, "Optimal sorting networks," in *(LATA)*. Springer, 2014, pp. 236–247.

[8] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "25 comparators is optimal when sorting 9 inputs (and 29 for 10)," in *(ICTAI)*, 2014.

[9] D. E. Knuth, "The Art of Computer Programming Vol. 3: Sorting and Searching," 1998.

[10] M. Ajtai, J. Komlós, and E. Szemerédi, "An $\mathcal{O}(n \log n)$ Sorting Network," in *(STOC)*, 1983.

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

[12] M. Függer, A. Kinali, C. Lenzen, and T. Polzer, "Metastability-aware Memory-efficient Time-to-Digital Converters," in *(ASYNC)*, 2017.

[13] G. Even, "On teaching fast adder designs: Revisiting Ladner & Fischer," in *Theoretical Computer Science*. Springer, 2006, pp. 313–347.