Near-Optimal Metastability-Containing Sorting Networks*

Johannes Bund
Saarland University
Saarland Informatics Campus, Germany
Email: s9jobund@stud.uni-saarland.de

Christoph Lenzen
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
Email: clenzen@mpi-inf.mpg.de

Moti Medina
Max Planck Institute for Informatics
Saarland Informatics Campus, Germany
Email: mmedina@mpi-inf.mpg.de

***Abstract*—Metastability in digital circuits is a spurious mode of operation induced by violation of setup/hold times of stateful components. It cannot be avoided deterministically when transitioning from continuously-valued to (discrete) binary signals. However, in prior work (Lenzen & Medina ASYNC 2016) it has been shown that it is possible to fully and deterministically *contain* the effect of metastability in sorting networks. More specifically, the sorting operation incurs no loss of precision, i.e., any inaccuracy of the output originates from mapping the continuous input range to a finite domain.**

The downside of this prior result is inefficiency: for B-bit inputs, the circuit for a single comparison contains $\Theta(B^2)$ gates and has depth $\Theta(B)$. In this work, we present an improved solution with near-optimal $\Theta(B \log B)$ gates and asymptotically optimal $\Theta(\log B)$ depth. On the practical side, our sorting networks improves over prior work for all input lengths $B > 2$, e.g., for 16-bit inputs we present an improvement of more than 70% in depth of the sorting network and more than 60% in cost of the sorting network.

I. Introduction

In the 80's, Marino proved that any bistable circuit can be forced into *metastability* [1], an unstable equilibrium state that is neither binary 0 nor 1 (regardless of how the voltage thresholds for these states are chosen). Given that metastability fundamentally disrupts the binary abstraction and, accordingly, the operation of digital circuits, the traditional methods for dealing with metastability are (i) to make sure that timing violations potentially causing metastability are avoided deterministically or (ii) to wait until metastability resolves with sufficiently high probability.

Option (i) is achieved by synchronous design, but can also be guaranteed in asynchronous circuits in many cases. However, crossing boundaries of (unsynchronized) clock domains inherently bears the risk of metastability, requiring to resort to Option (ii). This takes the form of *synchronizer* chains, which are designed to resolve metastability as fast as possible. For instance, state-of-the-art synchronizers for $65\,\text{nm}$ technology achieve roughly a factor e^{-20} decay per nanosecond in the probability of sustained metastability (i.e., $\tau \approx 50\,ps$ [2]); however, there is evidence that the resolution time will *increase* with further miniaturization [3].

The corresponding overhead in delay and buffer size can be avoided if the clock sources of different domains are synchronized. Unfortunately, this is not as straightforward as it seems: Naïve master-slave approaches essentially fuse all clock domains into one and introduce a single point of failure, negating crucial advantages of the multi-domain approach in terms of scalability and reliability.

Clock synchronization: A promising alternative is to run a fault-tolerant clock synchronization algorithm, like the one by Lynch and Welch [4], [5], preserving the benefits of multiple clock domains while removing the need for synchronizers. Inconveniently, it turns out that this does not eliminate the problem, but rather shifts it to the synchronization algorithm: any clock synchronization algorithm needs to measure time differences between the nodes executing it, which entails to map a continuous range to a discrete binary representation;[1] Marino's impossibility result now implies that this transition from analog to digital must incur the risk of metastability.

Naturally, one could now resort to Option (ii), synchronizers, to resolve this issue. However, this costs time! And losing time decreases the precision of the clock synchronization algorithm, which crucially relies on frequently updated information on clock skews to correct them.

Containing metastability: We follow a novel approach recently proposed by Friedrichs et al. [6]. They establish a theory of *metastability-containing circuits,* which allows for meaningful calculation under worst-case propagation of metastability of inputs.

Concretely, for the above problem, in an unpublished manuscript [7] it is shown how to perform the time measurements of the synchronization algorithm such that the measurements are encoded as Gray code values with (at most) one metastable bit, which captures the uncertainty of the measurement. For instance, when discretizing a time difference of $75\,\text{ps}$ in steps of $10\,\text{ps}$, one may obtain the encodings of 7 or 8 (which for a Gray code differ in exactly one bit), or the bit string for which the bit that is different is metastable. At the heart of the Lynch-Welch algorithm lies the operation of sorting such input strings according to their encoded value, and afterwards applying a respective (analog!) phase shift to the local clock.

* The proofs were omitted and will appear in the full version of the paper. We will be happy to provide them on request.

[1] Unless, of course, the algorithm is implemented in a fully analog way, which is costly.

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

This enables the following separation of concerns for the logic of the nodes executing the algorithm:

1) Measure clock differences to all other nodes (incurs metastability as described above).
2) Sort the results in a metastability-containing way.
3) Apply a corresponding phase shift (metastability is "absorbed" by the analog correction).

Metastability-containing sorting: For this approach to succeed it is essential that the initial inaccuracy of the measurement is not "amplified" by the sorting logic, despite metastability, i.e., metastability is contained. In [6], it is shown that this is possible *in principle,* but the corresponding circuits are exponentially large in the number of bits B of the Gray code inputs. Lenzen and Medina [8] proved that one can sort two Gray code inputs (i.e., output $\max\{g,h\}$ and $\min\{g,h\}$ for inputs g,h) in this manner with a circuit of depth $\mathcal{O}(B)$ and $\mathcal{O}(B^2)$ gates. Feeding n inputs into an $\mathcal{O}(\log n)$-depth [9] sorting network comprised of such metastability-containing 2-sort elements, yields a metastability-containing sorting network of depth $\mathcal{O}(B\log n)$ and $\mathcal{O}(B^2 n\log n)$ gates.

However, the 2-sort implementation from [8] is still large compared to standard 2-sort circuits, which have $\mathcal{O}(B)$ gates and depth $\mathcal{O}(\log B)$. Worse, the linear delay of the circuit defeats the purpose of resorting to metastability-containing logic, as the large computational delay eats away at the time saved by discarding all synchronizers. This leads to the following open question posed by Lenzen and Medina [8].

> *Is there a small metastability-containing* 2*-sort circuit of logarithmic depth?*

Our Contribution: We answer this question in the affirmative, by devising, proving correct, and validating in simulation a metastability-containing 2-sort circuit of depth $\Theta(\log B)$ and size $\Theta(B\log B)$. Given the trivial lower bounds of $\log B$ and $B-1$ when using fan-in 2 gates, this is close to optimal. The involved constants are moderate, leading to circuits of sufficient performance for use in the Lynch-Welch algorithm, cf. Table V. Plugging our circuit into (optimal depth or size) sorting networks [10], [11], [12], we obtain efficient combinational metastability-containing sorting circuits, cf. Table VI.

II. Encodings and Valid Inputs

Due to the potential presence of metastability at the input, we need to carefully choose and make use of suitable encodings. In this section, we formalize the respective notation and summarize basic properties of the encodings.

a) Reflected Binary Gray Code: For $N\in\mathbb{N}$, we abbreviate $[N]:=\{0,\dots,N-1\}$. For convenience, bit indices are one-based, e.g, the leftmost bit of a binary string g is $g[1]$, the second is $g[2]$, etc. For $1\le i\le j\le B$ (where g has B bits), let $g[i:j]:=g[i]g[i+1]\dots g[j]$; in case $i>j$, $g[i:j]$ is the empty string. We assume that B is a power of two. We denote the first and second halves of the string g by g_0 and g_1, i.e., $g_0=g[1:B/2]$, and $g_1=g[B/2+1:B]$. Let $\mathrm{par}(g)$ denote the parity of the binary string g, i.e, $\mathrm{par}(g)=\sum_{i=1}^{B} g[i] \bmod 2$.

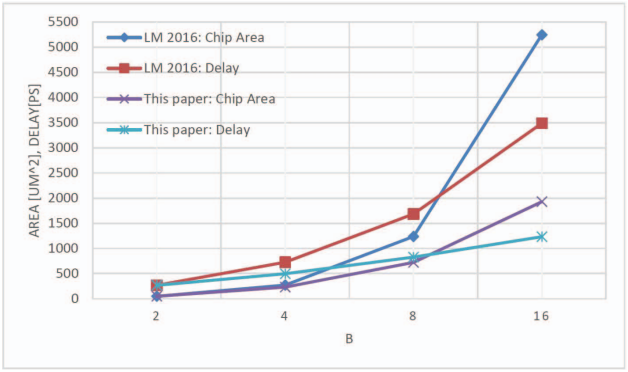

Fig. 1. Chip area and Propagation delay of 2-sort(B) for $B\in\{2,4,8,16\}$. We compare between this paper's circuit and the 2-sort from Lenzen and Medina [8].

TABLE I
The structure of a 4-bit Reflected Binary Gray Code.

g_0	g_1	g_0	g_1	g_0	g_1	g_0	g_1
00	00	01	10	11	00	10	10
00	01	01	11	11	01	10	11
00	11	01	01	11	11	10	01
00	10	01	00	11	10	10	00
$\mathrm{par}(g_0)=0$		$\mathrm{par}(g_0)=1$		$\mathrm{par}(g_0)=0$		$\mathrm{par}(g_0)=1$	

We 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 $\mathrm{rg}_B:[N]\to\{0,1\}^B$.

Intuitively, B-bit binary reflected Gray code can be recursively defined as follows (where a 1-bit code is trivial). Taking two copies $\mathrm{rg}^{(0)}$ and $\mathrm{rg}^{(1)}$ of $\mathrm{rg}_{B/2}$, we encode 0 as $\mathrm{rg}^{(0)}(0)\circ\mathrm{rg}^{(1)}(0)$. For every up-count, we increase $\mathrm{rg}^{(1)}$ until we reach the maximum value $\sqrt{N}-1$ for a $B/2$-bit code, i.e., $\mathrm{rg}^{(1)}(\sqrt{N}-1)$. The next up-count increases $\mathrm{rg}^{(0)}$. Afterwards, each up-count *decreases* $\mathrm{rg}^{(1)}$, until we reach $\mathrm{rg}^{(1)}(0)$ again. This process is repeated until we reach $\mathrm{rg}(N-1)=\mathrm{rg}^{(0)}(\sqrt{N}-1)\circ\mathrm{rg}^{(1)}(0)$. Because $\mathrm{rg}^{(0)}$ and $\mathrm{rg}^{(1)}$ are both Gray codes (i.e., each up-count changes at most one bit), the same is true for rg. Table I illustrates this structure at hand of a 4-bit code.

A crucial observation is that whether we are currently "counting up" or "counting down" the $\mathrm{rg}^{(1)}$-part of the string is given by $\mathrm{par}(\mathrm{rg}^{(0)})$: the parity of $\mathrm{rg}^{(0)}$ changes on each of its up-counts, as exactly one bit is changed, and on every such up-count we switch between counting $\mathrm{rg}^{(1)}$ up or down, respectively. An immediate consequence of this property is that $\mathrm{rg}^{(1)}$ always equals the maximum or minimum code word whenever $\mathrm{rg}^{(0)}$ contains a metastable bit. Our approach leverages these insights into the structure of the code, which we formalize mathematically in the full version of the paper.

b) Metastability Characterization of Valid Strings: Following [8], we define the set of valid strings that serve as inputs to our combinational circuits. We represent metastable bits by the symbol M. We utilize an operator that compares two strings and returns a string matching the originals where

TABLE II
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	0.5	011M	4.5	110M	8.5	101M	12.5
0001	1	0111	5	1101	9	1011	13
00M1	1.5	01M1	5.5	11M1	9.5	10M1	13.5
0011	2	0101	6	1111	10	1001	14
001M	2.5	010M	6.5	111M	10.5	100M	14.5
0010	3	0100	7	1110	11	1000	15
0M10	3.5	M100	7.5	1M10	11.5	–	–

they are both stable and equal, and M everywhere else.

Definition 2.1 (The $*$ operator [8]): Given $B \in \mathbb{N}$, define the operator $* : \{0,1,\mathtt{M}\}^B \times \{0,1,\mathtt{M}\}^B \to \{0,1,\mathtt{M}\}^B$ by

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

As discussed earlier, we may assume that Gray code inputs from time measurements contain at most one metastable bit [7], and our 2-sort elements will maintain this property. In addition, valid strings guarantee that if the single metastable bit resolves to either 0 or 1, the resulting string encodes either x or $x+1$ for some x, cf. Table II.

Definition 2.2 (Valid Strings [8]): Let $B \in \mathbb{N}$ and $N = 2^B$. Then, the set of *valid strings of length* B is

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

For simplicity, the decoding function $\langle \cdot \rangle$ is extended to also "decode" valid strings that contain a metastable bit.

Definition 2.3 (Extended Decoding [8]): Let $N = 2^B$ for some $B \in \mathbb{N}$. Denote $[N]_{\mathtt{M}} := \{z/2 \,|\, z \in [2N-1]\}$, i.e., the half-integers from the range $[0, N-1]$. For $x \in [N-1]$, we extend $\langle \cdot \rangle$ to $\mathcal{S}^B_{\mathrm{rg}}$ by

$$\langle \mathrm{rg}_B(x) * \mathrm{rg}_B(x+1) \rangle := x + 1/2 .$$

Accordingly, for $x \in [N-1]$ we extend rg_B to $[N]_{\mathtt{M}}$ by setting $\mathrm{rg}_B(x+1/2) := \mathrm{rg}_B(x) * \mathrm{rg}_B(x+1)$.

The following definition captures the span of values that a valid string can attain after metastability resolves.

Definition 2.4: For $g \in \mathcal{S}^B_{\mathrm{rg}}$, we define that $\mathrm{res}(g) := \{\mathrm{rg}_B(\lfloor \langle g \rangle \rfloor), \mathrm{rg}_B(\lceil \langle g \rangle \rceil)\}$.

III. MODEL AND PROBLEM

As we seek to design combinational circuits that accept valid strings as inputs, i.e., we need to specify circuit behavior in face of metastability. In accordance with [6], [8], we assume that gates propagate metastability to their outputs in a worst-case manner, but taking into account logical masking: an AND gates always outputs 0 if one of its inputs is stable 0, and an OR gate always outputs 1 if one of its inputs is stable 1.

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

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

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

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

A. Metastability-containing Multiplexers

Our metastability-containing circuits rely on multiplexers with a special property: if the inputs between which metastable control bits select are identical, the output is stable. Such *metastability-containing multiplexers (*CMUX*)* have been introduced in [8]. Hence, we confine ourselves to giving the specification of a $(4:1) -$ CMUX (the only type of CMUX we use) in Table IV. A $(4:1) -$ CMUX(B) exhibits the same behavior, but selects between B-bit input strings. Note that it is possible to improve our circuits further by devising better $(4:1) -$ CMUX(B) implementations.

TABLE IV
OUTPUT OF A $(4:1) -$ CMUX. THE CONTROL BITS ARE s AND t, THE SELECTABLE INPUTS ARE a, b, c, d.

s \ t	0	1	M
0	a	b	a∗b
1	c	d	c∗d
M	a∗c	b∗d	a∗b∗c∗d

B. Problem Definition

Our goal is to compute the maximum or minimum of two valid strings. To understand what this means, consider the following scenario. Suppose a valid string "encodes" $x + 1/2$ 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 $x+1$. In this case, a time-to-digital converter (TDC) generating this string has measured a time period corresponding to at least x and at most $x+1$ delay stages of the TDC. The string will stabilize to either x or $x+1$. The stabilized string is thus off by at most 1 w.r.t. the time period it represents.

In order to fully maintain the precision of the original measurement, we impose the same constraint on the extensions of max and min to valid inputs, which is formalized as follows.

Definition 3.1 ([8]): Let $N = 2^B$ for some $B \in \mathbb{N}$. For $g, h \in \mathcal{S}^B_{\mathrm{rg}}$, define

$$\max_{\mathrm{rg}}\{g,h\} := \mathrm{rg}_B(\max\{\langle g \rangle, \langle h \rangle\})$$
$$\min_{\mathrm{rg}}\{g,h\} := \mathrm{rg}_B(\min\{\langle g \rangle, \langle h \rangle\}) .$$

For example,

- $\max_{\mathrm{rg}}\{0111, 0101\} = \mathrm{rg}_4(\max\{5,6\}) = 0101$,
- $\max_{\mathrm{rg}}\{1\mathtt{M}10, 1111\} = \mathrm{rg}_4(\max\{11.5, 10\}) = 1\mathtt{M}10$,
- $\max_{\mathrm{rg}}\{010\mathtt{M}, 0100\} = \mathrm{rg}_4(\max\{6.5, 7\}) = 0100$.

Note that this definition entails that the results are valid strings as well. Our goal is to find (efficient) circuits computing $\max_{\mathrm{rg}}$ and $\min_{\mathrm{rg}}$.

Definition 3.2 ([8]): For $B \in \mathbb{N}$, a (combinational) 2-sort(B) circuit is defined as follows.

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

We remark that the results of [6] imply that this is the most restrictive specification that coincides with max and min on inputs without metastability and can be implemented by a circuit in our model. That is, any further suppression of metastability has to make use of synchronizers (which ensure probabilistic guarantees only), use analog components, or employ of masking registers (cf. [6]).

IV. The Circuit

The high-level idea of the circuit is to first determine which of the strings to select in a metastability-containing manner, represented by two control bits. It may happen that one or both of these bits become metastable, but in this case we show in the full version that the structure of the Gray code implies that the input strings were almost the same, distinguished only by which of their bits are metastable. Combining the selected strings using a CMUX, we ensure that the stable bits shared by both strings do not become metastable in the ouput, despite possible metastability of the control bits.

A. Control Signal Circuit

The subcircuit computing the control bits performs a 4-valued comparison: given stable inputs g, h, it returns 01 or 10 if $\langle g \rangle < \langle h \rangle$ or $\langle g \rangle > \langle h \rangle$, respectively. If $g = h$, it returns 00 or 11, depending on the parity of the equal strings. For inputs with metastable strings, the behavior is most conveniently specified by considering all possible resolutions of the inputs, determining the respective outputs, and obtaining the output for the original inputs as the "superposition" of all options under the $*$ operator. For example,

- $c(0111, 0101) = 01$,
- $c(1001, 1001) = 00$,
- $c(1\mathtt{M}10, 1111) = 10 * 10 = 10$,
- $c(010\mathtt{M}, 0100) = 11 * 01 = \mathtt{M}1$,
- $c(111\mathtt{M}, 111\mathtt{M}) = 11 * 10 * 01 * 00 = \mathtt{MM}$.

a) Specification:

Definition 4.1: For $B \in \mathbb{N}$, a (combinational) cont(B) circuit is defined as follows.

- ***Input*:** $g, h \in \mathcal{S}_{\mathrm{rg}}^B$,
- ***Output*:** $c \in \{0, 1, \mathtt{M}\}^2$,
- ***Functionality*:**

$$c := \mathop{*}_{(g',h') \in \mathrm{res}(g) \times \mathrm{res}(h)} \hat{c}(g', h') \,,$$

where for $g', h' \in \{0,1\}^B$, $\hat{c}(g', h')$ is defined as follows.

$\hat{c}(g',h')[1]$	$\hat{c}(g',h')[2]$	Semantics
0	0	$g' = h'$ and $\mathrm{par}(g') = 0$
0	1	$\langle g' \rangle < \langle h' \rangle$
1	0	$\langle g' \rangle > \langle h' \rangle$
1	1	$g' = h'$ and $\mathrm{par}(g') = 1$

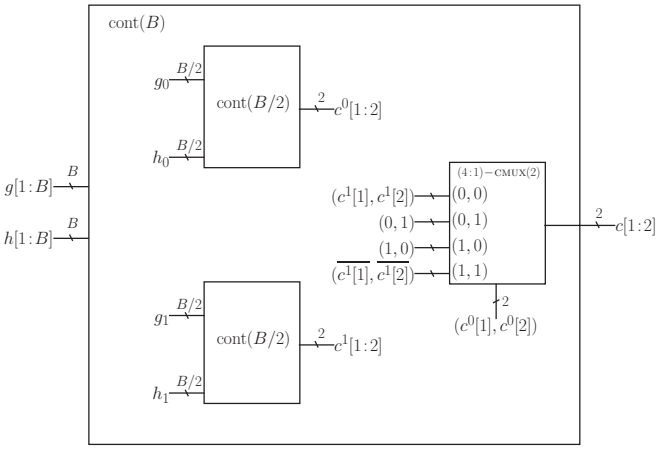

Fig. 2. Recursive implementation of cont(B) derived from Lemma 4.3.

b) Implementation: The base case is trivial.

Fact 4.2: For $B = 1$, the specification given in Definition 4.1 is met by the identity circuit returning output (g, h) for inputs $g, h \in \{0, \mathtt{M}, 1\}$.

For B that is a power of 2, we implement the specification recursively. The idea is to recursively use a $B/2$-bit circuit on inputs g_0, h_0 and g_1, h_1, respectively, and use the result of the second call to resolve a tie from the first call. For keeping track of the parity in case of a tie it is essential to use the result of the second call correctly: if the parity of $g_0 = h_0$ is odd, we need to negate the control bits returned by the second call.

Lemma 4.3: Suppose $B = 2^i$ for some $i \in \mathbb{N}$. The circuit depicted in Figure 2 implements the specification of cont(B) given in Definition 4.1.

Fact 4.2 and Lemma 4.3 yield the following theorem.

Theorem 4.4: For $B = 2^i$, $i \in \mathbb{N}$, cont(B) can be implemented by a circuit of delay and cost:

$$\mathrm{delay}(\mathrm{cont}(B)) = \log(B) \cdot \mathrm{delay}((4:1) - \mathrm{CMUX}) \,,$$
$$\mathrm{cost}(\mathrm{cont}(B)) = 2 \cdot (B-1) \cdot \mathrm{cost}((4:1) - \mathrm{CMUX}) \,.$$

B. 2-sort(B) Implementation

Again, we observe that the base case is trivial.

Fact 4.5: For $B = 1$, the specification given in Definition 3.2 is met by the circuit returning $\mathrm{AND}(g, h)$ and $\mathrm{OR}(g, h)$ for inputs $g, h \in \{0, \mathtt{M}, 1\}$.

We recursively implement the sorting circuit as follows. We determine $(\max_{\mathrm{rg}}\{g, h\})_0 = \max_{\mathrm{rg}}\{g_0, h_0\}$ by a recursive call (the same for $\min_{\mathrm{rg}}$). A second recursive call determines $\max_{\mathrm{rg}}\{g_1, h_1\}$ and $\min_{\mathrm{rg}}\{g_1, h_1\}$. Finally, we use a call to cont$(B/2)$ to compute the control bits selecting which of the strings g_1, h_1, $\max_{\mathrm{rg}}\{g_1, h_1\}$, and $\min_{\mathrm{rg}}\{g_1, h_1\}$ to select as $(\max_{\mathrm{rg}}\{g, h\})_1$. The correctness of this implementation is proven in the full version of the paper.

Lemma 4.6: The circuit depicted in Figure 3 implements the specification of 2-sort(B) given in Definition 3.2.

Note that the recursive construction requires computing control bits in each level of the recursion. However, the

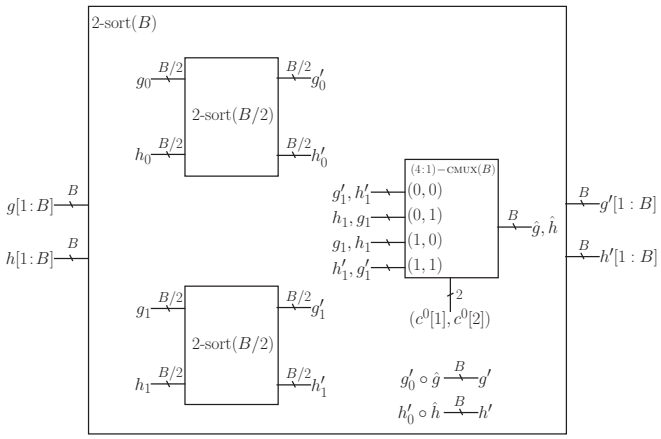

Fig. 3. Recursive implementation of 2-sort(B). To obtain the control inputs $(c^0[1], c^0[2])$, we feed g_0 and h_0 into a cont$(B/2)$ circuit, cf. Figure 2.

respective circuit recurses on the same substrings as the sorting circuit, and has slightly smaller delay. This enables reuse of the outputs of the recursive calls of the control circuit as control bits for the recursive calls of the sorting circuit. Exploiting this insight, we arrive at a highly efficient 2-sort implementation.

Theorem 4.7: For $B = 2^i$, $i \in \mathbb{N}$, 2-sort(B) can be implemented by a circuit of delay and cost:

$$\begin{aligned}\text{delay}(\text{2-sort}(B)) &= \text{delay}(\text{AND}) \\ &\quad + \log(B) \cdot \text{delay}((4{:}1)-\text{CMUX}) \\ \text{cost}(\text{2-sort}(B)) &= 2B \cdot \text{cost}(\text{AND}) \\ &\quad + B \log B \cdot \text{cost}((4{:}1)-\text{CMUX}) \\ &\quad + 2(B - \log(2B)) \cdot \text{cost}((4{:}1)-\text{CMUX})\end{aligned}$$

C. Sorting Networks

Given a metastability-containing 2-sort implementation, it is now straightforward to sort multiple inputs using standard techniques. Taking any sorting network, we can plug in our metastability-containing 2-sort circuit to obtain a metastability-containing sorting network. Cost, delay, and, in first-order approximation, area of the sorting network scale linearly with the cost and delay of the 2-sort implementation.

Suppose the sorting network has n channels, i.e., we sort n strings. The inputs are valid Gray code strings of length B. The output of the sorting network are the n input strings, sorted according to the order induced by $g < h \Leftrightarrow \langle g \rangle < \langle h \rangle$.

Bundala and Závodnỳ [11] proved the optimality w.r.t. depth of Knuth's sorting networks [10] (i.e., $n \leq 16$ channels). Codish et. al [12] proved the optimality w.r.t. number of gates of Knuth's sorting networks with up to 10 channels. In our context, we are specifically interested in sorting networks with $n = 3f + 1$ channels for some $f \in \mathbb{N}$, as this is the minimum number of nodes required to tolerate f faulty nodes in the clock synchronization algorithm by Lynch and Welch [5].

V. Simulation Results

A. Design Flow

We list the tools we used in the design flow: (i) Design entry: Quartus, (ii) Behavioral simulation: ModelSim, (iii) Synthesis: Encounter RTL Compiler (part of Cadence tool set) with the NanGate 45 nm Open Cell Library, (iv) Place & route: Encounter (part of Cadence tool set) with the NanGate 45 nm Open Cell Library.

TABLE V
Comparison of number of gates, delay, and chip-area of the of 2-sort(B) from this paper and from [8], and Binary-comp, an optimized comparator taking binary inputs.

B	Circuit	# Gates	Area [μm^2]	Delay [ps]
$B = 2$	This paper	34	49.42	268
	[8]	34	49.42	268
	Binary-comp	8	15.582	145
$B = 4$	This paper	160	230.3	498
	[8]	188	272.118	728
	Binary-comp	19	34.58	288
$B = 8$	This paper	504	723.52	827
	[8]	856	1237.425	1684
	Binary-comp	41	73.752	477
$B = 16$	This paper	1344	1928.262	1233
	[8]	3632	5247.186	3488
	Binary-comp	81	151.648	422

B. Adaptations to the Design Flow due to Metastability

In order to preserve the metastability-containment properties of our circuit, the optimizer in the synthesis had to be disabled. The simplest example is our CMUX. The optimizer realizes that a CMUX is a multiplexer and disregards additional properties introduced by the "redundant" gates in the CMUX. Hence, the optimizer simply transforms a CMUX to a regular multiplexer. A standard multiplexer may output `M` even if the two select inputs are `1` in case the control signal is `M`.

In Figure 4, we show how badly a regular sorting network behaves on inputs which contain metastable bits; shown is the output of ModelSim. Interestingly, Quartus internally mapped parts of the circuit to lookup-tables (LUTs) featuring metastability-containment properties as a side effect of glitch avoidance techniques. This resulted in metastability being masked for many inputs. The given results were obtained by simulating the VHDL code directly in the ModelSim tool.

C. The benchmarks

We consider four designs for comparators, two of which meet the specification of 2-sort. (i) the 2-sort circuit by Lenzen and Medina [8], (ii) The 2-sort(B) circuit presented in this paper, (iii) Naive-comp(B): A Gray code comparator which is not metastability-containing, (iv) Binary-comp(B): A regular comparator of binary inputs. In what follows, we elaborate on these benchmarks. The two 2-sort circuits are not optimized and they are as they appear in the schematics given here and in [8], respectively. The Binary-comp(B) design is simply the VHDL comparator of two B-bit binary strings, allowing for an estimation of the overheads introduced by metastability-containment. Note that the Binary-comp design is *optimized* for both number of gates and delay in the RTL compiler; as discussed in the conclusion, we expect optimized CMUX designs to dramatically reduce the respective gaps. The Naive-comp(B) design uses the control subcircuit cont(B), which is implemented with *regular* multiplexers. The outputs of this variant of the cont(B) circuit were fed to

TABLE VI
SIMULATION RESULTS FOR METASTABILITY-CONTAINING SORTING NETWORKS WITH $n \in \{4, 7, 10\}$ FOR B-BIT INPUTS. 10-sort$_{\#}$ OPTIMIZES GATE COUNT, 10-sort$_{\mathrm{d}}$ DEPTH; FOR $n \in \{4, 7\}$, THE SORTING NETWORKS ARE OPTIMAL W.R.T. BOTH MEASURES.

B	Circuit	4-sort			7-sort			10-sort$_{\#}$			10-sort$_{\mathrm{d}}$		
		# Gates	Area [μm^2]	Delay [ps]	# Gates	Area [μm^2]	Delay [ps]	# Gates	Area [μm^2]	Delay [ps]	# Gates	Area [μm^2]	Delay [ps]
$B = 2$	This paper	170	247.016	846	544	790.44	1715	986	1432.62	2285	1054	1531.467	2010
	[8]	170	247.016	846	544	790.44	1715	986	1432.62	2285	1054	1531.467	2010
	Binary-comp	40	77.91	478	128	249.326	953	232	451.815	1284	248	483	1145
$B = 4$	This paper	800	1151.472	1558	2560	3684.541	3147	4640	6678.294	4207	4960	7138.74	3681
	[8]	940	1360.45	2268	3008	4353.328	4584	5452	7890.372	6116	5828	8434.79	5357
	Binary-comp	95	172.935	906	304	553.28	1810	551	1002.848	2429	589	1072.099	2143
$B = 8$	This paper	2520	3617.67	2394	8064	11576.32	4715	14616	20982.542	6252	15624	22429.176	5481
	[8]	4280	6186.544	5111	13696	19796.7	10313	24824	35881.65	13771	26536	38355.66	12043
	Binary-comp	205	368.641	1475	656	1179.528	2948	1189	2137.905	3945	1271	2285.514	3470
$B = 16$	This paper	6720	9640.75	3396	21504	30849.875	6415	38976	55916.448	8437	41664	59772.132	7458
	[8]	18160	26235.755	10817	58112	83953.24	21823	105328	152165.524	29152	112592	162658.944	25484
	Binary-comp	405	530.67	1298	1296	2425.99	2600	2349	4397.085	3474	2511	4700.304	3050

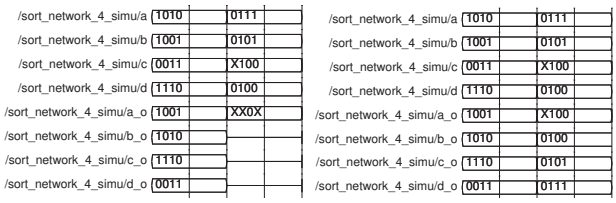

Fig. 4. The top four rows give sets of valid inputs to 4-channel sorting networks using Naive-comp(4) (left) and our 2-sort implementation (right) for comparisons, respectively. Metastable bits M are indicated as X. Using Naive-comp(4), all but one output bit is metastable (XXXX is depicted as a line). Our circuit outputs the input strings, sorted according to Table II.

regular multiplexers, which output the sorted (two) inputs. The purpose of the Naive-comp design is to demonstrate the highly undesirable behavior of a non-containing sorting network under valid inputs.

These four designs are used as comparators in optimal sorting networks with 4, 7, and 10 input channels. For 10 channels we considered two sorting networks: 10-sort$_{\#}$ optimizes the number of gates [12] and 10-sort$_{\mathrm{d}}$ optimizes the depth of the sorting network [11].

D. Summary of simulation results

Figure 4 illustrates that it is essential to utilize 2-sort circuits to guarantee correct behavior in the presence of metastability in the input. Table V and Table VI show the chip area,[2] number of gates,[3] and the delay[4] of our circuits in 45 nm technology.

The improvements are substantial. In particular, for a 10-channel sorting network with $B = 16$, we obtain (i) a 62.9% improvement in gate count (comparing the network that optimizes the number of gates), (ii) a 63.2% improvement in chip area (comparing the network that optimizes the number of gates), and (iii) a 70.7% improvement in delay (comparing the network that optimizes the depth of the sorting network).

VI. DISCUSSION

In this work, we improve prior results by Lenzen and Medina on metastability-containing sorting networks. We obtain a purely combinational circuit that is asymptotically optimal w.r.t. delay and has an exponentially smaller (logarithmic) gap to optimality w.r.t. the number of transistors. We formally show correctness alongside precise delay and cost bounds as function of B and the properties of basic components. We confirm these assertions by detailed simulations.

The main open questions this paper raises are:

- Are there better CMUX implementations, which would readily improve the constants in our bounds?
- Are there other powerful basic primitives that can be efficiently realized in a metastability-containing way?
- What is the optimum cost of the 2-sort(B) primitive?

The first question is subject to our current work. Preliminary results indicate that transistor-level CMUX implementations can cut down cost, delay, and area of our metastability-containing sorting networks by factors between 2 and 3 compared to the presented numbers. Comparing to the results for (optimized!) non-containing variants, this would result in very small overheads in terms of delay, and small overheads w.r.t. gate count, and area corresponding to the $\Theta(\log B)$ overhead of the presented 2-sort implementation in comparison to an optimal non-containing solution.

[2]We give the value returned by Encounter, without VDD and GND rings.
[3]The RTL Compiler counts an OR(3) gate as a single gate.
[4]We report the delay outputted by the RTL Compiler.

REFERENCES

[1] L. Marino, "General Theory of Metastable Operation," *IEEE Transactions on Computers*, vol. C-30, no. 2, pp. 107–115, 1981.
[2] T. Polzer and A. Steininger, "An Approach for Efficient Metastability Characterization of FPGAs through the Designer," in *19th Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2013, pp. 174–182.
[3] S. Beer, R. Ginosar, M. Priel, R. R. Dobkin, and A. Kolodny, "The Devolution of Synchronizers," in *Proc. Symposium on Asynchronous Circuits and Systems (ASYNC)*, 2010, pp. 94–103.
[4] F. Huemer, A. Kinali, and C. Lenzen, "Fault-tolerant Clock Synchronization with High Precision," in *Proc. Symp. on VLSI (ISVLSI)*, 2016.
[5] J. L. Welch and N. A. Lynch, "A New Fault-Tolerant Algorithm for Clock Synchronization," *Information and Computation*, vol. 77(1), 1988.
[6] S. Friedrichs, M. Függer, and C. Lenzen, "Metastability-Containing Circuits," *CoRR*, vol. abs/1606.06570, 2016.
[7] A. Anonymous, "Metastability-aware Memory-efficient Time-to-Digital Converters," 2016, unpublished manuscript.
[8] C. Lenzen and M. Medina, "Efficient metastability-containing gray code 2-sort," in *Proc. 22nd Symposium on Asynchronous Circuits and Systems, ASYNC*, 2016, pp. 49–56.
[9] M. Ajtai, J. Komlós, and E. Szemerédi, "An $\mathcal{O}(n \log n)$ Sorting Network," in *15th Symposium on Theory of Computing (STOC)*, 1983.
[10] D. E. Knuth, "The Art of Computer Programming Vol. 3: Sorting and Searching," 1998.
[11] D. Bundala and J. Závodnỳ, "Optimal sorting networks," in *Language and Automata Theory and Applications*. Springer, 2014, pp. 236–247.
[12] M. Codish, L. Cruz-Filipe, M. Frank, and P. Schneider-Kamp, "25 comparators is optimal when sorting 9 inputs (and 29 for 10)," in *26th Conf. on Tools with Artificial Intelligence (ICTAI)*, 2014.