

Membership Test Logic For Delay-Insensitive Codes

Stanisław J. Piestrak

Wrocław University of Technology
Institute of Engineering Cybernetics
ul. Wybrzeże Wyspiańskiego 27
50-370 Wrocław, Poland
E-mail: sjp@residue.ict.pwr.wroc.pl

&

ENSSAT — Université de Rennes
LASTI
B.P. 447, 6 rue de Kerampont
22305 Lannion CEDEX, France
E-mail: piestrak@enssat.fr

Abstract

Delay-insensitive (unordered) codes have been used to encode data in various asynchronous systems such as asynchronous circuits and buses. In this paper, a new general approach to designing completion-detection circuits (completion checkers) for asynchronous circuits and systems using delay-insensitive codes is presented. It is shown that a completion-detection circuit for many delay-insensitive codes can be easily and efficiently built in a systematic way by using multi-output threshold circuits. The results presented here remain in a sharp contrast with the conclusions reached by Akella et al. [7], where similar designs — called enumeration-based decoders — were found impractical due to excessive complexity.

1 Introduction

Asynchronous (self-timed) circuits can be loosely defined as the circuits with no global clock. Growing interest in designing asynchronous circuits can be attributed to their well known potential benefits, such as: the possibility of avoiding clock problems which are common in synchronous circuits, lower power consumption, and average rather than worst case performance. Depending on the delay model, several classes of asynchronous circuits can be distinguished [1]. Amongst them included are the following classes, which have in common that they use delay-insensitive (unordered) codes and their operation requires a completion signal: (i) Speed-independent mode — assuming finite unbounded delay in gates and no delay in wires; and (ii) Delay-insensitive (DI) mode — assuming finite unbounded delay in both gates and wires. In [2] it was shown that the DI model — to be practical — calls for an *isochronic fork* assumption, which stipulates that the delays of all forking wires are the same, or at least they are smaller than the delay of an elementary logic gate.

A *completion-detection circuit (CC)* (also called a completion checker) is a circuit needed in asynchronous circuits to determine when a functional block has completed its operation, i.e. it provides indication of the transient process completion in circuits. A variety of CCs have been presented in [3]–[7]. These include the membership test logic circuits for DI (also

called *unordered*) codes [3] and [7] — which are of our interest here, and the CCs which rely on current-sensing [4], [5] and activity- or transition-monitoring techniques [6].

The first general approach to designing CCs for various DI codes was considered in [3] (pp. 69–74), where the general rules for designing 2-level circuits were given for the *m*-out-of-*n* (*m/n*) codes and the code with identifier; for either code no particular multi-level implementation has been suggested however. Recently, two complementary designs of CCs — called *enumeration-based decoders* and *comparison-based decoders* were proposed by Akella *et al.* [7]. (However, throughout this paper, we shall use the term completion-detection circuit (CC) which has been more widely used in the literature on asynchronous circuits.) The enumeration-based CC, which can be applied for both nonsystematic and systematic codes, examines a codeword received and determines if it is valid or not. (A code is *systematic* if the information bits can be distinguished from the check bits.) In [7] it was claimed that it is generally impractical and impossible to implement due to limitations of current VLSI technologies and available CAD tools (excessive hardware complexity and delay). As a more feasible alternative, the comparison-based decoder was proposed, which, however, can be applied for systematic codes only. It detects the arrival of a codeword by: (i) recomputing the check bits using the received data bits, and (ii) comparing them with the check bits received. Its principal limitation is that it cannot be implemented as an inverter-free circuit and, as such, it cannot be implemented as a hazard-free circuit without making additional delay assumptions.

The main goal of this paper is to present some theoretical results on the design of CCs working as membership test logic circuits for the most important classes of DI codes. We will challenge the conclusions reached in [7] regarding their comparison-based CCs for DI codes (Berger codes and *m/n* codes) and show that all these circuits can be easily designed and implemented. In fact, our approach allows one to derive the general logic functions of a CC for the most commonly used DI codes. In particular, we shall show that the new enumeration-based CCs for Berger codes not

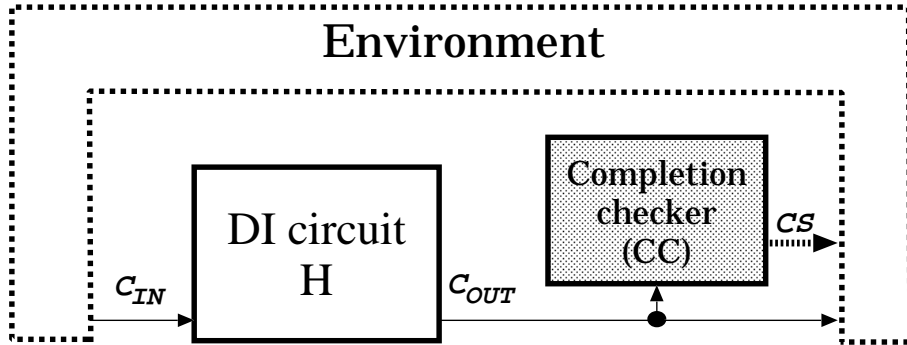


Figure 1: Model of a delay-insensitive system.

only have similar complexity to the comparison-based CCs from [7], but they are faster and implemented hazard-free with no delay limitations.

This paper is organized as follows. Section 2 describes the model of a DI system, surveys the most important DI codes, and presents the design methods of multi-output threshold circuits which will be used here as the main building block of most CCs. Section 3 presents the CCs for the 2-rail codes, the m/n codes, the Berger codes, and the codes with identifier.

2 Preliminaries

2.1 Model of a Delay-Insensitive 4-Phase System

One of self-timed systems using DI encodings can be represented as shown in Fig. 1 (for more details, see e.g. [9]–[12]) and it operates under the following assumptions:

1. A system works according to DI model with the addition of isochronic fork assumption.
2. Each operation cycle of the system follows a four-phase signaling protocol which requires that all logical variables return to the same logic state corresponding to a spacer (all-0s or all-1s). The circuit \mathbf{H} alternately receives the spacer and codeword inputs from C_{IN} in its idle and working phase, respectively. Correspondingly, it alternately generates the spacer and codewords from C_{OUT} on its outputs.
3. (00...0) is used as a spacer during idle phase both on inputs and outputs.
4. Input and output are encoded with DI codes C_{IN} and C_{OUT} .
5. The transition from a codeword X to the spacer (00...0) is performed by executing the transition $x_i \downarrow$ only on the bits x_i which are 1's in X , whereas the transition from the spacer to a codeword is performed by executing the direct transitions $x_i \uparrow$ only on the bits which are 0's in X .

Hence, in either case, the multiple transitions of some bits, e.g. $0 \rightarrow 1 \rightarrow 0 \rightarrow 1$ do not occur.

6. Initially the circuit \mathbf{H} is stable with (00...0) spacers being present both on input X and output Z .

More specifically, the system from Fig. 1 works as follows. The environment is assumed to generate input transition spacer \rightarrow codeword (working phase) and codeword \rightarrow spacer (idle phase) at any time. The environment recognizes readiness of the circuit \mathbf{H} to receive a new input codeword $X \in C_{IN}$ by checking whether the output of \mathbf{H} contains the spacer. Next, the environment recognizes that the computation performed by the circuit \mathbf{H} has been completed by checking whether the output of \mathbf{H} contains a codeword $Z \in C_{OUT}$, and then the environment sets the input to the spacer. Finally, the cycle repeats after the circuit \mathbf{H} changes its output to the spacer, in response to the input spacer provided by the environment.

2.2 DI (Unordered) Codes

A number of DI codes have been proposed in the literature, see [13]–[22], although most of them can be found in the works on self-checking circuits and error control codes under the name *unordered codes*. The importance of DI codes in designing self-timed circuits has long been recognized, see e.g. [8] and [3], although in these applications the unordered codes have been called *self-synchronizing* or *delay-insensitive* codes (which more directly reflects their most important property for DI circuits). Any DI code was shown suitable for designing a self-timed CC that follows the 4-phase (idle/working) operation discipline using two spacers (00...0) or/and (11...1) [8]–[11].

Amongst DI codes the most important DI codes are the following. For readers' convenience, the examples of all codes considered are presented in Tables 1–3.

1. A K -pair 2-rail code is a systematic code which uses K check bits (c_{K-1}, \dots, c_1, c_0) which are the bit-by-bit complements of the $I = K$ information bits (x_{I-1}, \dots, x_1, x_0), i.e. for any $0 \leq i \leq K - 1$

Table 1: Sample unordered encodings of 4-bit data.

N	Inf. Part	Check Part P_X			3/6 code
	J_X $x_3x_2x_1x_0$	2-rail $c_3c_2c_1c_0$	Berger $c_2c_1c_0$	M. Berger $c_2c_1c_0$	X $x_6x_5x_4x_3x_2x_1$
0	0000	1111	100	111	000111
1	0001	1110	011	110	001011
2	0010	1101	011	110	001101
3	0011	1100	010	101	001110
4	0100	1011	011	110	010011
5	0101	1010	010	101	010101
6	0110	1001	010	101	010110
7	0111	1000	001	100	011001
8	1000	0111	011	011	011010
9	1001	0110	010	010	011100
10	1010	0101	010	010	100011
11	1011	0100	001	001	100101
12	1100	0011	010	010	100110
13	1101	0010	001	001	101001
14	1110	0001	001	001	101010
15	1111	0000	000	000	101100

we have $c_i = \bar{x}_i$. The 2-rail codes are the most well known and practically the only that have been used in asynchronous circuits, due to their advantages: simple design of circuitry and that they are generally easy to implement in hardware (e.g. in DCVS CMOS technology). Their disadvantage is very high 100% redundancy.

- An m -out-of- n code (m/n code, constant-weight code) is one in which all valid codewords have exactly m 1s and $n - m$ 0s. Its capacity equals to $|C_{m/n}| = \binom{n}{m}$, where $C_{m/n}$ denotes all m/n codewords. The $\lfloor n/2 \rfloor/n$ code is the optimal DI code [14], in the sense that there is no other DI code of codeword length n which has more codewords than the $\lfloor n/2 \rfloor/n$ code. The disadvantage of m/n codes is that they are nonsystematic. Notice however that a K -pair 2-rail code is nothing else but a special case of an incomplete $K/2K$ code with only 2^K out of $\binom{2K}{K}$ all codewords used. (*Note:* A code is *incomplete* if some of its codewords are not used.)
- The *Berger code* $C_{(I,K)}$ [13] is a systematic code wherein the K check bits P_X are the binary representation of the count of the I information bits J_X which are 0s, where $K = \lceil \log_2(I + 1) \rceil$. (An alternative equivalent Berger code has the check part P_X defined as the bit-by-bit complemented number of 1s in the information part J_X .) The Berger codes are the optimal systematic DI codes.
- The optimal systematic DI codes equivalent to Berger codes can be constructed according to the methods given in [15] and [22]. (A code C of length n is called *equivalent to Berger code* $C_{(I,K)}$ if and only if $n = I + K$, it is systematic, DI, and

has the same number of codewords as $C_{(I,K)}$.) All these codes have an advantage over most Berger codes (except for the Berger codes with $I = 2^K - 1$) that they use all 2^K combinations as the check part. This property was found essential e.g. while constructing self-checking DI circuits [12].

- The *code with identifier* $CI_{(I,K)}$ — an extension of any Berger code with $I \neq 2^K - 1$ which can be classified as *semi-systematic*, proposed in [3]. Amongst a family of codes with identifier that can be constructed for a given I , two extreme cases of a code can be distinguished: one with the maximum number of codewords and another with the simple encoder and decoder. The examples of these codes will be provided in Subs. 3.4.

Table 1 shows sample unordered encodings of 4-bit data J_X : (i) using systematic codes — the 4-pair 2-rail code, the Berger code $C_{(4,3)}$, and the modified Berger code $C_{(4,3)}$ — for which suitable check parts P_X are shown, and (ii) a nonsystematic 3/6 code. *Note:* The nonsystematic 3/6 code allows for an arbitrary mapping of selected 16 out of a total of 20 codewords. The encoding presented in Table 1 assumes the increasing order of the decimal value of a 3/6 codeword (treated as a binary vector with x_6 and x_1 as the MSB and LSB, respectively), where the variables of encoded data are more conveniently denoted with $(x_6x_5x_4x_3x_2x_1)$.

The code with identifier $CI_{(4,3)}$ with easy encoding presented in Table 2 has 30 codewords, i.e. no codewords corresponding to two largest 5-bit numbers $N = 30$ and 31 can be formed. This code is constructed as follows. Any codeword corresponding to any 5-bit vector $(x_4x_3x_2x_1x_0)$ with $x_4 = 0$ (i.e. for

$0 \leq N \leq 15$) preserves its four LSBs, i.e. $x'_3 = x_3$, $x'_2 = x_2$, $x'_1 = x_1$, and $x'_0 = x_0$. Its check part P_X equals to (000) for $w(x_3, x_2, x_1, x_0) = 4$; otherwise, it has the MSB check bit $c_2 = 1$ and the decimal value corresponding to $3 - w(x_3, x_2, x_1, x_0)$ as $(c_1 c_0)$, where $w(x_3, x_2, x_1, x_0)$ denotes the *binary weight* of the vector $(x_3 x_2 x_1 x_0)$. On the other hand, any codeword corresponding to any 5-bit vector $(x_4 x_3 x_2 x_1 x_0)$ with $x_4 = 1$ (i.e. for $16 \leq N \leq 29$) is formed as follows. Four bits representing the 'information part' $x'_3 x'_2 x'_1 x'_0$ contain the binary representation of $N - 15$ — which corresponds to $(x_3 \cdot 2^3 + x_2 \cdot 2^2 + x_1 \cdot 2^1 + x_0 \cdot 2^0 + 1)$. The code construction ensures that for $16 \leq N \leq 29$ there are no vectors such that $w(x'_3, x'_2, x'_1, x'_0) = 0$ or 4. Hence, the check part P_X has the MSB check bit $c_2 = 0$ and the decimal value of $(c_1 c_0)$ corresponding to $4 - w(x_3, x_2, x_1, x_0)$. The code $CI_{(4,3)}$ with easy encoding can be more conveniently specified as shown in Table 3.

Table 2: Code with identifier $CI_{(4,3)}$ with easy encoding.

N	$x_4 x_3 x_2 x_1 x_0$	$x'_3 x'_2 x'_1 x'_0$	$c_2 c_1 c_0$
0	0 0000	0000	111
1	0 0001	0001	110
2	0 0010	0010	110
3	0 0011	0011	101
4	0 0100	0100	110
5	0 0101	0101	101
6	0 0110	0110	101
7	0 0111	0111	100
8	0 1000	1000	110
9	0 1001	1001	101
10	0 1010	1010	101
11	0 1011	1011	100
12	0 1100	1100	101
13	0 1101	1101	100
14	0 1110	1110	100
15	0 1111	1111	000
16	1 0000	0001	011
17	1 0001	0010	011
18	1 0010	0011	010
19	1 0011	0100	011
20	1 0100	0101	010
21	1 0101	0110	010
22	1 0110	0111	001
23	1 0111	1000	011
24	1 1000	1001	010
25	1 1001	1010	010
26	1 1010	1011	001
27	1 1011	1100	100
28	1 1100	1101	011
29	1 1101	1110	001

2.3 Multi-Output Threshold Circuits T^n

In the next section, we shall show how to construct a CC for any DI code in a systematic way. It will be shown that a CC for most DI codes can be built

Table 3: Compressed list of codewords of the code $CI_{(4,3)}$ with easy encoding.

Weight $w(J_X)$	Basic Check Part $P_X = (c_2 c_1 c_0)$	Extra Check Part $P_X = (c_2 c_1 c_0)$
0	1 1 1	—
1	1 1 0	0 1 1
2	1 0 1	0 1 0
3	1 0 0	0 0 1
4	0 0 0	—

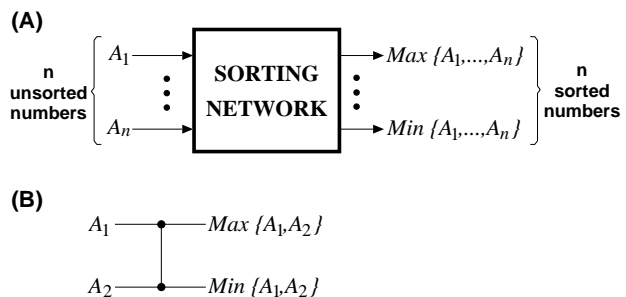


Figure 2: (A) General structure of an SN; (B) A comparator cell.

using a special class of multi-output threshold circuits, which will be presented here.

Let $S = \{x_1, x_2, \dots, x_n\}$ denote a set of n input variables and m denote a threshold.

Definition 1 A threshold function T_m^n is a switching function of n variables from the set S , which takes the value 1 if and only if at least m out of n input variables from S are 1s, $1 \leq m \leq n$.

Definition 2 A multi-output threshold circuit T^n is a circuit that implements all n T_m^n threshold functions of n variables, $1 \leq m \leq n$.

Several implementations of the threshold circuit T^n have been proposed in the literature under various names — for more references on threshold circuits and an extensive survey an interested reader may refer to [28], [29]. Basically, any realization of the circuit T^n can be used as a basic building block in any circuit presented here. However, we have shown in [28], [29] that the least complex and the most easily testable circuits T^n can be implemented as *sorting networks* (SNs). Therefore in all parameter estimations given hereafter, we assume those of the most efficient circuit T^n implemented as an SN, unless stated otherwise.

An n -input *sorting network* (SN) is a switching network with n outputs that for any combination of inputs $\{x_1, x_2, \dots, x_n\}$ generates the outputs which are a sorted permutation of inputs, e.g. in nonincreasing

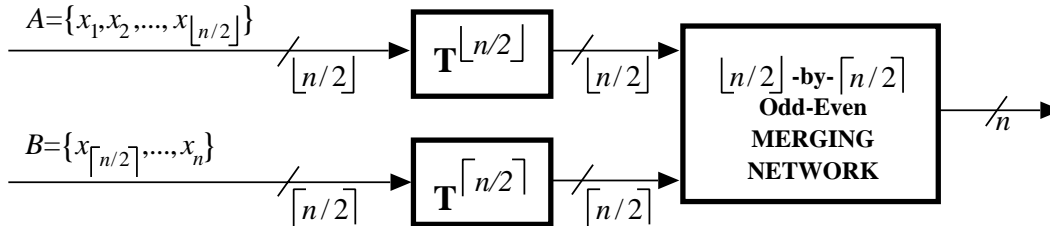


Figure 4: General structure of the n -input SN T^n using odd-even merging.

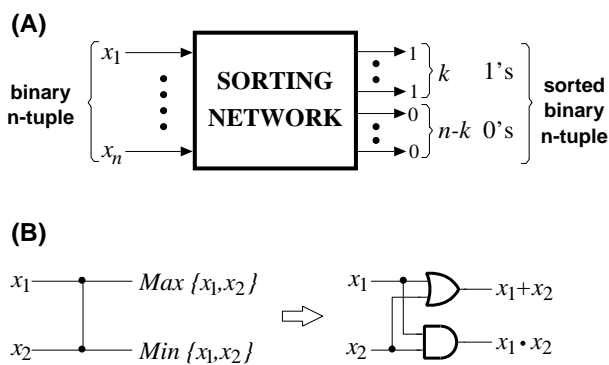


Figure 3: (A) The structure of an SN considered here; (B) Logic gate implementation of a comparator cell.

order (see Fig. 2(A)). A *binary* SN is entirely built of simple identical *comparator cells* with two inputs and two outputs, such as one shown in Fig. 2(B). For brevity, we will call such a cell simply a *comparator* (the symbol of a comparator given in Fig. 2(B) is traditionally used in the literature on sorting). A comparator executes a *pass* or *interchange* operation, depending on the inputs. As explained in Fig. 2(B), here the upper outputs are $M = MAX(A_1, A_2)$ and $m = MIN(A_1, A_2)$ which carry the maximum and the minimum of the two inputs A_1 and A_2 . An SN built using this element sorts in nonincreasing order. However, here we are interested in the simplest implementation of a comparator cell — such as shown in Fig. 3(B), which implies that in our case an SN sorts 0s and 1s — see Fig. 3(A). The SN from Fig. 3 is a special case of a circuit T^n .

The problem of designing SNs has been studied extensively in the literature for many years. The milestone result is the construction given by Batcher [23] in 1968. In 1974 Van Voorhis [25] presented constructions of SNs which provide the best upper bound for most practical values of n . An excellent survey of many SNs designs was provided by Knuth [24]. In particular, he listed some *ad hoc* constructions of the

SNs for $n \leq 16$ invented by many authors, which are either more efficient or faster than those from [23] and [25].

Batcher [23] proposed two schemes of SNs constructed from: (1) odd-even merging networks (MNs) and (2) bitonic sorters using the sorting-by-merging scheme. The two schemes have the same number of levels but only the first one, which is built of fewer comparators, is of interest here. Batcher's SNs use the optimal number of comparators for $n \leq 8$. The least complex schemes of SNs for $n > 8$ can be found for $9 \leq n \leq 16$ in [24] (Fig. 49, p. 228). The fastest schemes of the SNs for $n = 6, 9, 10, 12$, and 16 can be found in [24] (Fig. 51, p. 231). The two least complex constructions of SNs currently known as $n \rightarrow \infty$, found independently by Drysdale [26] and Van Voorhis [25], use the most efficient 16-input SN by Green (refer to Fig. 49 on p. 228 in [24]). Van Voorhis' scheme is slightly better: e.g. for $n = 256$ it requires 3651 comparators while Green's requires 3657. For most n of practical value for logic designer (i.e. for $n \leq 64$) the hardware savings offered by the schemes from [26] and [25] are not significant: for instance, Van Voorhis' SNs with $n = 16, 32$, and 64 use 60, 180, and 514 comparators compared with Batcher's 63, 191, and 543. Moreover, the irregular structures of the SNs from [24], [26], [25] make them difficult to analyze. Therefore, with a few exceptions, we concentrate on Batcher's SNs.

Figure 4 shows that any SN can be constructed recursively for any n , provided that the SNs with $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ inputs are available and we know how to build a $\lfloor n/2 \rfloor$ -by- $\lceil n/2 \rceil$ MN (more details on the construction of the MNs can be found e.g. in [23], [24], [29], [30]). It is important to note that while constructing the circuit T^n , the circuits $T^{\lfloor n/2 \rfloor}$ and $T^{\lceil n/2 \rceil}$ can be realized using *any* method, i.e. they do not have to be SNs. The recursive construction of a SN can be recognized by inspection of Fig. 5 showing the 4-input SN: the comparators $K4$ and $K5$ correspond to two circuits T^2 and the comparators $K1, K2$, and $K3$ form the 2-by-2 MN.

The complexity characteristics of the T^n circuit realized as a Batcher's odd-even MN — the number of logic gates $Ga(T^n)$ and the number of gate levels

Table 4: Parameters of threshold circuits T^n realized as optimal SNs for $3 \leq n \leq 16$

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Gates	6	10	18	24	32	38	50	62	74	78	96	106	118	126
Levels	3	3	5	5	6	6	8	9	10	9	10	10	10	10
Tests	6	6	8	9	11	12	12	15	17	16	20	21	23	24

$L(T^n)$ are the following:

$$Ga(T^n) = \frac{1}{2}n(\lceil \log_2^2 n \rceil - \lceil \log_2 n \rceil + 4) - 2 \quad (1)$$

$$L(T^n) = \frac{1}{2}\lceil \log_2 n \rceil(\lceil \log_2 n \rceil + 1). \quad (2)$$

The size of the minimal complete test set (complete means that it contains tests of all $n + 1$ weights) that detects all single stuck-at- z (s/ z) faults, $z \in \{0, 1\}$, is

$$|\mathbf{T}(T^n)| = n + \lceil n/2 \rceil. \quad (3)$$

The procedure of generating the minimal test set for any Batcher's SN and some other special SN designs can be found in [29].

The parameters of the Batcher's SNs for $3 \leq n \leq 16$ are included in Table 4.

Finally, it is worth to point out that, besides superb performance, the SNs have the following structural advantage, important for VLSI implementation: The SNs have a uniform regular structure using only one type of a simple cell composed of two gates having both fan-in and fan-out equal two.

3 Completion-Detection Circuits

In this section we will present the CCs for asynchronous circuits using the most important DI codes presented in Section 2.

The completion signal circuit CC checks whether all input and internal transitions within the DI circuit \mathbf{H} (see Fig. 1) have been completed. Here, the set of legal inputs to CC include the codewords X_i of some DI code C (i.e. $X_i \in C$) and a spacer (00...0). The CC has one output: $CS(C)$ — the completion signal which is: (i) 0 — when the spacer is present on the input of CC or the monotonic transition from the spacer to some codeword X_i has not been completed yet; and (ii) 1 — when all bits of an incoming codeword X_i has changed to 1. Throughout this paper, it is assumed that no $0 \rightarrow 1$ errors (i.e. X_i contains at least one extra bit erroneously changed from 0 to 1) occur on the input. Obviously, a $1 \rightarrow 0$ error of any multiplicity in X_i causes that the transition (00...0) $\rightarrow X_i$ cannot be completed in finite time and the CC preserves $CS(C) = 0$ and effectively halts operation of the system (it can be detected by a time-out circuit).

Preliminarily, we assume that the Boolean function $CS(C)$ of a CC for a code C are expressed in a monotonic form (which is natural for any DI code) as

$$CS(C) = \sum_{m_i \in M} m_i, \quad (4)$$

where M is the set of all implicants m_i which correspond to the codewords $X_i \in C$.

The one-to-one correspondence between every $X_i \in C$ and every $m_i \in M$, denoted $X_i \leftrightarrow m_i$, is established as follows: $X_i \leftrightarrow m_i$ implies that m_i is the product of only uncomplemented variables corresponding to bits which are 1 in X_i . For instance, for $X_i = (x_1x_2x_3x_4x_5) = (11001)$ we have $(11001) \leftrightarrow m_i = x_1x_2x_5$. Hence, a 2-level AND-OR circuit that implements (4) contains only *complete unordered products* and it is *nonconcurrent*, i.e. for any $X_i \in C$ exactly one product is 1. The latter property is important as it guarantees hazard-free implementation of the CC. Obviously, a direct realization of the function (4) would be prohibitively complex for most codes to be practical (as it was observed in [7]).

In this section we will show that a multi-level implementation of (4) is fairly easy for all codes considered. Indeed, the logic functions applicable for any code of any codeword length n that allow for a multi-level implementation of (4) will be presented.

Note: All logic functions will be conveniently expressed in terms of AND and OR operators. However, to ensure correct DI operation, the actual realizations of all these circuits will require to replace every i -input AND gate with an i -input C-element.

3.1 Completion-Detection Circuit for K -Pair 2-Rail Code

We assume that a K -pair 2-rail codeword is written as $X = (x_{K-1} \dots x_1x_0c_{K-1} \dots c_1c_0)$, where $c_i = \bar{x}_i$ for any $0 \leq i \leq K - 1$.

Basically, the CC for the K -pair 2-rail code C_{K2r} could be realized according to (4), but it is not practical. The simplified multi-level CC realizes the following well-known logic function

$$CS(C_{K2r}) = \prod_{i=0}^{K-1} (x_i + c_i), \quad (5)$$

i.e. it consists of K 2-input OR gates and one K -input C-element which must be used instead of the K -input AND gate suggested by the form of Eq. (5).

3.2 Completion-Detection Circuit for m/n Code

Detection of any m/n codeword requires checking whether the weight of a vector is equal to m . According to the survey presented in Subsection 2.3, a CC for an m/n code is nothing else but a threshold

circuit T_m^n , i.e.

$$CS(C_{m/n}) = T_m^n. \quad (6)$$

Although, the circuit T_m^n generates 1 not only when there are exactly m input bits set to 1, but also more than 1. Assuming error-free operation, the latter case never occurs however.

Basically, a CC for an m/n code can be constructed by modifying the circuit T^n , so that out of its n outputs $\{T_1^n, T_2^n, \dots, T_n^n\}$ only one output T_m^n is left. Recall, however, that most circuits T^n are constructed recursively by using a pair of circuits T^{n_a} and T^{n_b} (where, for instance, $n_a = \lfloor n/2 \rfloor$ and $n_b = \lceil n/2 \rceil$) followed by some merging network — see Fig. 4. Hence, to avoid modification of T^n which can be unnecessarily cumbersome, especially for larger n , the circuit T_m^n can be constructed directly using the subcircuits T^{n_a} and T^{n_b} (if $m \geq n_a$ and $m \geq n_b$) or the circuits $T_{1 \sim m}^{n_a}$ and $T_{1 \sim m}^{n_b}$, otherwise.

Let $T_{p \sim r}^n$, $1 \leq p < r \leq n$, denotes a circuit obtained from the circuit T^n by removing: (i) the outputs with the lowest ($\{T_1^n, T_2^n, \dots, T_{p-1}^n\}$) and the highest ($\{T_{r+1}^n, T_{r+2}^n, \dots, T_n^n\}$) thresholds, so that only the outputs T_j^n with $p \leq j \leq r$ are left, and (ii) all associated circuitry.

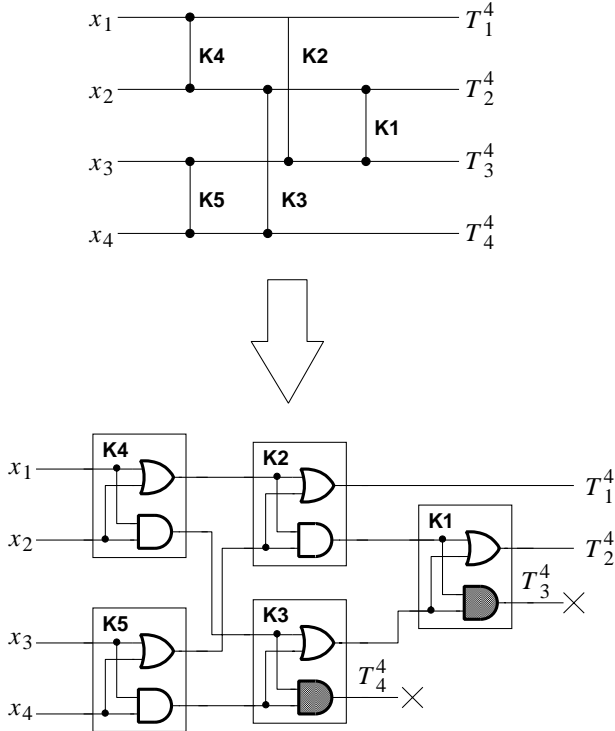


Figure 5: Construction of the circuit $T_{1 \sim 2}^4$

Figure 5 shows how to obtain the circuit $T_{1 \sim 2}^4$ by simplifying the circuit T^4 . Two shadowed AND gates

(from the comparators K1 and K3) are removed from the circuit T^4 , as they generate the signals of weight 3 and 4 (which are not needed), respectively.

Now the logic function T_m^n realized by a CC for an m/n code can be expressed as

$$CS(C_{m/n}) = \sum_{i=\underline{i}}^{\bar{i}} T_i^{n_a} T_{m-i}^{n_b}, \quad (7)$$

where: $\underline{i} = \max\{0, m - n_a\}$ and $\bar{i} = \min\{m, n_a\}$.

The CC for a complete m/n code consists of $\bar{i} - \underline{i} - 1$ 2-input AND gates and one $\bar{i} - \underline{i} + 1$ OR gate.

For instance, consider designing the CCs for the 2/8 and 6/8 codes. For either code, $n = 8$ input bits $\{x_1, x_2, \dots, x_8\}$ are first partitioned into two disjoint subsets $A = \{x_1, x_2, x_3, x_4\}$ and $B = \{x_5, x_6, x_7, x_8\}$. Then, two circuits T^{4a} and T^{4b} with inputs respectively from A and B are used in a modified form. The CC for the 2/8 code uses a pair of circuits $T_{1 \sim 2}^{4a}$ and $T_{1 \sim 2}^{4b}$, such as one shown in Fig. 5, and its function is given by

$$CS(C_{2/8}) = \sum_{i=0}^2 T_i^{4a} T_{m-i}^{4b} = T_2^{4b} + T_1^{4a} T_1^{4b} + T_2^{4a}.$$

This circuit detects if two or more inputs are 1 (however, assuming valid codewords only, there will never be more than two inputs 0 are 1).

On the other hand, the CC for the 6/8 code uses a pair of circuits $T_{2 \sim 4}^{4a}$ and $T_{2 \sim 4}^{4b}$, obtained by removing the final OR gate that generates T_1^4 from the circuit T^4 (the comparator K2 in Fig. 5), and its function is given by

$$CS(C_{6/8}) = \sum_{i=2}^4 T_i^{4a} T_{m-i}^{4b} = T_2^{4a} T_4^{4b} + T_3^{4a} T_3^{4b} + T_4^{4a} T_2^{4b}.$$

3.3 Completion-Detection Circuit for Berger Codes

Let $X = (J_X P_X)$ denotes a codeword of a Berger code $C_{(I,K)}$.

The inverter-free CC for a Berger code $C_{(I,K)}$ is shown in Fig. 6. It is built of two circuits with separated primary inputs: the information bits $J_X = \{x_1, x_2, \dots, x_I\}$ feed the threshold circuit T^I , whereas the check bits $P_X = \{c_0, c_1, \dots, c_{K-1}\}$ (where c_{K-1} is the MSB) feed the unate products generator (UPG). The UPG for the Berger code $C_{(I,K)}$ has K check bits (c_0, c_1, \dots, c_{K-1}) as inputs and $I + 1$ outputs w_i , $1 \leq i \leq I + 1$. w_i is a product of all bits c_{i_1} which are 1 in the decimal representation of w_i , i.e. $w_i = c_{i_1} \cdot c_{i_2} \cdot \dots \cdot c_{i_k}$, where $0 \leq i_1 < i_2 < \dots < i_k \leq K - 1$ such that $2^{i_1} \cdot 2^{i_2} \cdot \dots \cdot 2^{i_k} = i$. For instance, for $K = 5$ since $w_{13} \leftrightarrow (c_4 c_3 c_2 c_1 c_0) = (01101)$ $w_{13} = c_3 c_2 c_0$. The outputs of the circuits T^I and UPG are then combined by a 2-level AND-OR circuit that produces

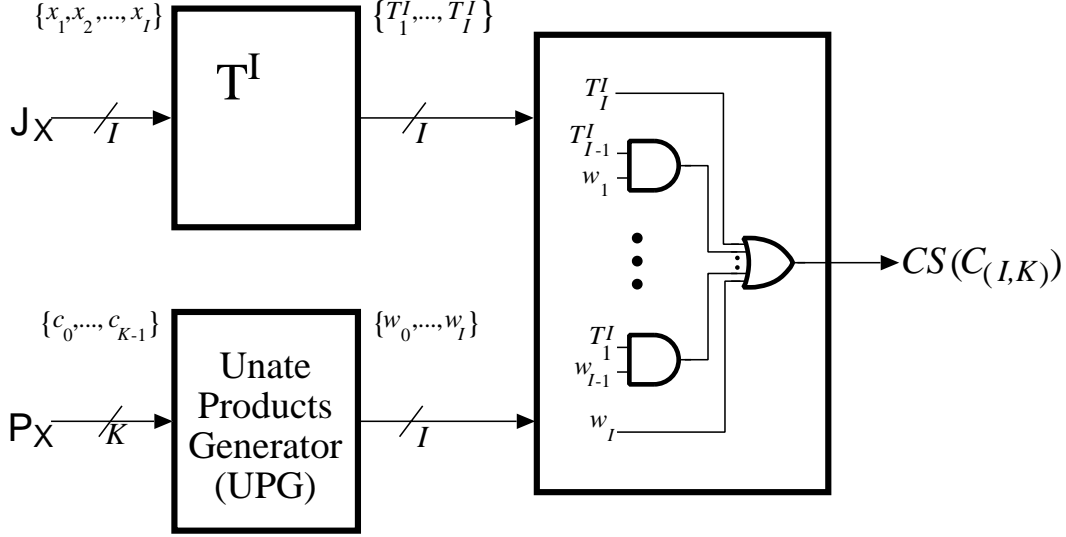


Figure 6: Completion-detection circuit for a Berger code $C_{(I,K)}$.

the completion signal. For the Berger code $C_{(I,K)}$ the function of CC is

$$CS(C_{(I,K)}) = \sum_{i=0}^I T_i^I \cdot w_i, \quad (8)$$

where, for the sake of generality, two special cases of subfunctions can occur:

- 1) $T_0^I = 1$ (which is obvious), and
- 2) $w_I = 1$ for $I = 2^K - 1$.

The functioning of the above circuit is based on the following observations. For $X = (J_X P_X) \in C_{(I,K)}$ whose information part has some weight $w(J_X)$ and for the check part holds the correspondence $P_X \leftrightarrow w_i$. we have:

- $T_j^I(J_X) = 1$ for all $1 \leq j \leq i$, and
- $w_k(P_X) = 1$ for $k = i$ and any implicant w_k for some $i \leq k \leq I$ obtained by setting some literals c_l in w_i to 1.

The construction of the Berger code guarantees that there is exactly one product $T_i^I \cdot w_i$ which is 1 for X .

Notice that the whole circuit of Fig. 6 contains only one multi-input gate — the final I -input OR, which can be implemented as a tree of $I - 1$ 2-input OR gates.

The recursive construction of an UPG — a circuit that generates all $2^K - 1$ positive unate products w_i ($1 \leq i \leq 2^K - 1$) of K bits is illustrated in Fig. 7 for $K = 4$. In general, the j -th step ($1 \leq j \leq K - 1$) allows one to form a total of $2^j - 1$ products w_i from $2^{j-1} - 1$ products w_i already formed for j variables c_k ,

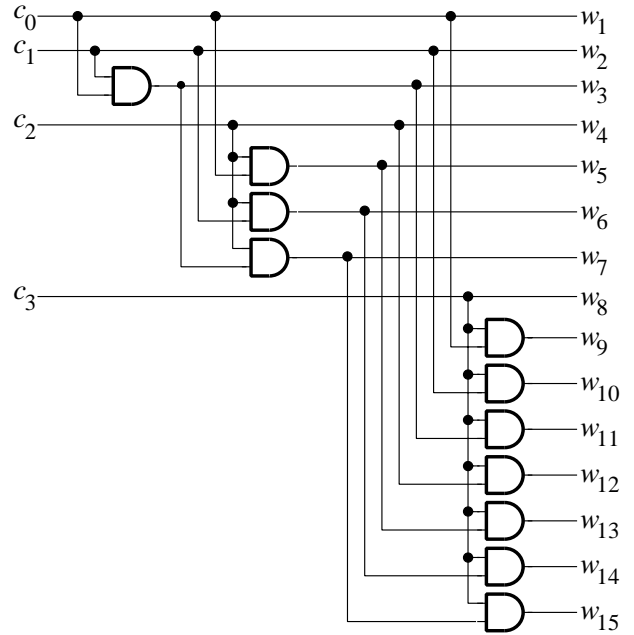


Figure 7: Construction of the 4-input circuit that generates all $2^4 - 1 = 15$ positive unate products of four bits.

$0 \leq k \leq j - 2$ and the $(j - 1)$ -th variable c_{j-1} ($1 \leq i \leq 2^{j-1} - 1$); note that the new product $w_{2^{j-1}}$ equals to

c_{j-1} . For instance, the 4-th step of the construction of the UPG from Fig. 7 consists on: (1) preserving seven products w_1, w_2, \dots, w_7 already formed in previous three steps, (2) including $w_8 = c_3$, and (3) forming seven new products $w_9, w_{10}, \dots, w_{15}$ by ANDing $w_8 = c_3$ respectively with w_1, w_2, \dots, w_7 .

For $I = 2^K - 1$ a circuit is built of a total of $\sum_{i=0}^{K-1} (2^i - 1) = (2^K - 1) - K = I - K$ 2-input AND gates. For any Berger code $C_{(I,K)}$ with $I < 2^K - 1$, $2^K - I - 1$ output lines with the largest indices (i.e. w_{15}, w_{14} , etc.) are not used and hence can be ignored.

Table 5: The Berger code $C_{(5,3)}$.

$w(J_X)$	$P_X = (c_2 \ c_1 \ c_0)$	w_i	$T_i^5 \cdot w_i$
0	1 0 1	$w_5 = c_2 c_0$	w_5
1	1 0 0	$w_4 = c_1$	$T_1^5 \cdot w_4$
2	0 1 1	$w_3 = c_1 c_0$	$T_2^5 \cdot w_3$
3	0 1 0	$w_2 = c_1$	$T_3^5 \cdot w_2$
4	0 0 1	$w_1 = c_0$	$T_4^5 \cdot w_1$
5	0 0 0	$w_0 = 1$	T_5^5

The above construction is exemplified by the Berger code $C_{(5,3)}$ specified in Table 5. The CC for $C_{(5,3)}$ realizes the function

$$CS(C_{(5,3)}) = T_5^5 w_5 + T_4^5 w_4 + T_3^5 w_3 + T_2^5 w_2 + T_1^5 w_1 + w_0.$$

The complexity of the CC for the Berger code $C_{(I,K)}$ can be evaluated by using the following formula

$$Ga(C_{(I,K)}) \leq Ga(T^I) + (I - K) + (2I - 3), \quad (9)$$

where three terms denote respectively the number of gates needed to realize: the circuit T^I , all $2^K - 1$ positive unate products, and the final $I - 2$ 2-input AND gates and a tree of $I - 1$ 2-input OR gates. The delay of this circuit is

$$L(C_{(I,K)}) \leq L(T^I) + 1 + \log_2 I. \quad (10)$$

For instance, the CC for the Berger code $C_{(31,5)}$ can be built of less than 416 2-input gates in 21 levels (assuming that the circuit T^{31} obtained by modifying the circuit T^{32} by Van Voorhis is used). Clearly, it can be built using a comparable number of gates as the similar circuit from [7] — 408 gates (assuming that the 2-input XOR gate is equivalent to 3 standard gates). However, our circuit is certainly faster, as the CC from [7] uses a tree of ripple carry adders. More importantly, our circuit is entirely inverter-free, i.e. it can be implemented hazard-free unlike any comparison-based CC from [7]. The other advantages of the CC for Berger codes presented here are:

1. extremely simple construction — one can easily derive logic functions for any Berger code $C_{(I,K)}$ — without any effort needed to run logic synthesis tools, and

2. inverter-free implementation.

3.4 Completion-Detection Circuit for a Code With Identifier

The DI code called a *code with identifier*, which will be denoted here $CI_{(I,K)}$, was introduced by Varshavsky *et al.* in [19], [3]. It is a nonsystematic extension of any Berger code with $I \neq 2^K - 1$. For any given I two classes of codes with identifier can be constructed: the code with maximum capacity and the code with easy encoding and decoding. Since the codes $CI_{(I,K)}$ with $I = 2^K - 1$ seem to be the most interesting for practical applications, we will restrict our attention to this case only.

For the code $CI_{(8,4)}$ with maximum capacity (see Table 6) the CC realizes the following functions:

$$\begin{aligned} CS(CI_{(8,4)}) = & w_{15} + T_1^8 w_{14} + T_2^8 w_{13} + T_3^8 (w_{11} + w_7) \\ & + T_4^8 (w_3 + w_5 + w_6 + w_9 + w_{10} + w_{12}) \\ & + T_5^8 (w_8 + w_4) + T_6^8 w_2 + T_7^8 w_1 + T_8^8 \end{aligned}$$

Table 6: Code with identifier $CI_{(8,4)}$ with maximum capacity.

Weight $w(J_X)$	Basic Check Part $P_X = (c_3 \ c_2 \ c_1 \ c_0)$	Extra Check Parts $P_X = (c_3 \ c_2 \ c_1 \ c_0)$
0	(1 1 1 1)	—
1	(1 1 1 0)	—
2	(1 1 0 1)	—
3	(1 0 1 1)	(0 1 1 1)
4	(0 0 1 1)	(0 1 0 1) (0 1 1 0) (1 0 0 1) (1 0 1 0) (1 1 0 0)
5	(1 0 0 0)	(0 1 0 0)
6	(0 0 1 0)	—
7	(0 0 0 1)	—
8	(0 0 0 0)	—

Table 7: Code with identifier $CI_{(8,4)}$ with easy encoding and decoding.

Weight $w(J_X)$	Basic Check Part $P_X = (c_3 \ c_2 \ c_1 \ c_0)$	Extra Check Part $P_X = (c_3 \ c_2 \ c_1 \ c_0)$
0	1 1 1 1	—
1	1 1 1 0	0 1 1 1
2	1 1 0 1	0 1 1 0
3	1 1 0 0	0 1 0 1
4	1 0 1 1	0 1 0 0
5	1 0 1 0	0 0 1 1
6	1 0 0 1	0 0 1 0
7	1 0 0 0	0 0 0 1
8	0 0 0 0	—

For the code $CI_{(8,4)}$ with easy encoding and decoding (see Table 7) the CC realizes the following functions:

$$\begin{aligned} CS(CI_{(8,4)}) &= w_{15} + T_1^8(w_{14} + w_7) + T_2^8(w_{13} + w_6) \\ &+ T_3^8(w_{12} + w_5) + T_4^8(w_{11} + w_4) \\ &+ T_5^8(w_{10} + w_3) + T_6^8(w_9 + w_2) \\ &+ T_7^8(w_8 + w_1) + T_8^8. \end{aligned}$$

It can be easily shown that for the code $CI_{(I,K)}$ ($I = 2^{K-1}$) with easy encoding and decoding, the CC realizes the following general function

$$CS(CI_{(I,K)}) = w_{2I-1} + T_I^I + \sum_{i=1}^{I-1} T_i^I(w_{I-i} + w_{2I-1-i}). \quad (11)$$

The CC for a code $CI_{(I,K)}$ with easy encoding and decoding is very similar to the CC for the Berger code, shown in Fig. 6.

4 Conclusions

In this paper, we have presented the construction of the completion-detection circuits (CCs) which perform the membership test for the most important classes of delay-insensitive (unordered) codes, including the 2-rail codes, the m -out-of- n codes, the Berger codes, and the codes with identifier. We have shown that CCs for all these codes can be easily and efficiently built in a systematic way by using multi-output threshold circuits. These results remain in a sharp contrast with the conclusions reached by Akella *et al.* [7] for similar CCs (in [7] called enumeration-based encoders). Moreover the CCs presented here do not have the limitations of the so-called comparison-based encoders which cannot be implemented in delay-insensitive manner (i.e. they are prone to glitches). We believe the design approach presented here can be easily extended to include other delay-insensitive codes as well.

The results presented here can be seen as the first step towards possible realization of DI circuits using some DI codes which are less redundant than commonly used double-rail codes. At the moment, due to the lack of any synthesis methods for DI circuits using encodings other than 2-rail, the potential advantages of these codes are difficult to confirm; moreover that the CCs are clearly more complex than for a 2-rail code with the same number of information bits. However, these codes can perhaps be found useful in asynchronous communication [31], [32], wherein the immediate potential savings are at least in a significantly reduced number of bus lines.

It is also worth to point out that the results presented here can also be useful for designing synchronous circuits which are self-checking (with concurrent error detection). In particular, any CC for a DI (unordered) code presented here can also be used as a 1-output i.e. a circuit that signals occurrence of erroneous vectors, provided that these vectors are not members of a particular code.

Acknowledgments

Partial financial support for this research was supported by the grant No. 8 T11C 05 13 from Komitet Badań Naukowych, Poland. Part of this research was done while the author was with ENSSAT — Université de Rennes, LASTI, Lannion, France. The author would like to thank Alex Yakovlev for his encouragement to write this paper, when the author was visiting the University of Newcastle upon Tyne in May 1997. The constructive feedback from the reviewers has helped improve the paper significantly.

References

- [1] S. Hauck, "Asynchronous design methodologies," *Proc. IEEE*, vol. E83, pp. 69–93, Jan. 1995.
- [2] A. J. Martin, "The limitations to delay-insensitivity in asynchronous circuits," in *Proc. Conference on Advanced Research in VLSI: 6th MIT Conf.*, C. Séquin, Ed., MIT Press, Cambridge, MA, 1990, pp. 263–278.
- [3] V. I. Varshavsky (Ed.), *Self-Timed Control of Concurrent Processes*, Kluwer Acad. Publ., Norwell, MA, 1990 (Russian edition 1986), ch. 3.
- [4] O. A. Izosimov *et al.*, "Physical approach to CMOS self-timing," *Electron. Letters*, vol. 26, pp. 1835–1836, 1990.
- [5] M. E. Dean *et al.*, "Self-timed logic using current-sensing completion detection (CSCD)," *J. of VLSI Signal Proc.*, vol. 7, pp. 7–16, No. 1, 1994.
- [6] E. Grass, V. Bartlett, and I. Kale, "Completion-detection techniques for asynchronous circuits," *IEICE Trans. Inf. & Syst.*, vol. E80-D, pp. 344–350, March 1997.
- [7] V. Akella, N. H. Vaidya, and G. R. Redinbo, "Limitations of VLSI implementation of delay-insensitive codes," in *Dig. Pap. 26th Int. Symp. on Fault-Tolerant Computing*, Sendai, Japan, June 25–27, 1996, pp. 208–217.
- [8] D. B. Armstrong *et al.*, "Design of asynchronous circuits assuming unbounded gate delays," *IEEE Trans. Comput.*, vol. C-18, pp. 1110–1120, Dec. 1969.
- [9] A. J. Martin *et al.*, "The design of an asynchronous microprocessor," in *Proc. 1989 Decennial Caltech Conf.*, C. Séquin, Ed., MIT Press, Cambridge, MA, 1989, pp. 351–373.
- [10] T. Nanya *et al.*, "TITAC: Design of a quasi-delay-insensitive microprocessor," *IEEE Design and Test*, Sp. Issue on Asynchronous Circuits and Systems, vol. 11, pp. 50–63, No. 2, 1994.
- [11] I. David, R. Ginosar, and M. Yoeli, "Implementing sequential machines as self-timed circuits," *IEEE Trans. Comput.*, vol. 41, pp. 12–17, Jan. 1992.

- [12] S. J. Piestrak and T. Nanya, "Towards totally self-checking delay-insensitive systems," in *Dig. Pap. 25th Int. Symp. on Fault-Tolerant Computing*, Pasadena, CA, June 27–30, 1995, pp. 228–237.
- [13] J. M. Berger, "A note on error detection codes for asymmetric binary channels," *Inform. Contr.*, vol. 4, pp. 68–73, Mar. 1961.
- [14] C. V. Freiman, "Optimal error detection codes for completely asymmetric binary channels," *Inform. Contr.*, vol. 5, pp. 64–71, Mar. 1962.
- [15] M. J. Ashjaee and S. M. Reddy, "On totally self-checking checkers for separable codes," *IEEE Trans. Comput.*, vol. C-26, pp. 737–744, Aug. 1977.
- [16] J. E. Smith, "The design of totally self-checking check circuits for a class of unordered codes," *J. Des. Autom. Fault-Tolerant Comput.*, vol. 2, pp. 321–342, Oct. 1977.
- [17] G. P. Mak, J. A. Abraham, and E. S. Davidson, "The design of PLAs with concurrent error detection," in *Dig. Pap. 12th Int. FTC Symp.*, Santa Monica, CA, June 1982, pp. 303–310.
- [18] M. Blaum (Ed.), *Unidirectional Error Control Codes*, IEEE Press, 1992.
- [19] V. I. Varshavsky *et al.*, "Possibility of implementing an asynchronous interface using a self-synchronizing code with identifier," *Autom. Contr. Comput. Sci.*, vol. 15, pp. 77–81, No. 5, 1981.
- [20] J. E. Smith, "On separable unordered codes," *IEEE Trans. Comput.*, vol. C-33, pp. 741–743, Aug. 1984.
- [21] T. Verhoeff, "Delay-insensitive codes—An overview," *Distr. Computing*, vol. 3, pp. 1–8, No. 1, 1988.
- [22] S. J. Piestrak, "Design of TSC code-disjoint inverter-free 4 for separable unordered codes," in *Proc. ICCD'94, Int. Conf. on Computer Design: VLSI in Computers & Processors*, Cambridge, MA, Oct. 10–12, 1994, pp. 128–131.
- [23] K. E. Batcher, "Sorting networks and their applications," in *Proc. 1968 SJCC, AFIPS*, vol. 32, 1968, pp. 307–314.
- [24] D. E. Knuth, *The Art of Computer Programming, Vol. III, Sorting and Searching*, 2nd Ed., Reading, MA, Addison-Wesley, 1973, ch. 5.
- [25] D. C. Van Voorhis, "An economical construction for sorting networks," in *Proc. AFIPS NCC*, 1974, pp. 921–927.
- [26] R. L. (Scot) Drysdale and F. H. Young, "Improved divide/sort/merge sorting networks," *SIAM J. Comput.*, vol. 4, pp. 264–270, Sept. 1975.
- [27] E. A. Lamagna, "The complexity of monotone networks for certain bilinear forms, routing problems, sorting, and merging," *IEEE Trans. Comput.*, vol. C-28, pp. 773–782, Oct. 1979.
- [28] S. J. Piestrak, "The minimal test set for sorting networks and the use of sorting networks in self-testing checkers for unordered codes," in *Dig. Pap. 20th Int. Symp. on Fault-Tolerant Computing*, Newcastle upon Tyne, UK, June 26–28, 1990, pp. 467–474.
- [29] S. J. Piestrak, "The minimal test set for multi-output threshold circuits implemented as sorting networks," *IEEE Trans. Comput.*, vol. 42, pp. 700–712, June 1993.
- [30] S. J. Piestrak, *Design of Self-Testing Checkers for Unidirectional Error Detecting Codes*, Scientific Papers of the Inst. of Techn. Cybern. of the Techn. Univ. of Wrocław, No. 92, Ser.: Monographs No. 24, Oficyna Wyd. Polit. Wrocław, Wrocław 1995.
- [31] M. Blaum and J. Bruck, "Coding for skew-tolerant parallel 0 communications," *IEEE Trans. Inform. Theory*, vol. 39, pp. 379–388, Mar. 1993.
- [32] M. Blaum and J. Bruck, "Delay-insensitive pipelined communication on parallel buses," *IEEE Trans. Comput.*, vol. 44, pp. 660–668, May 1995.