UDC 621
doi:10.15217/issn1684-8853.2018.1.68

# ERROR-CORRECTING CODES FOR TERNARY CONTENT ADDRESSABLE MEMORIES: A NEW PERSPECTIVE

**S. Engelberg**[a], *PhD, Math., Associate Professor and Dean, shlomoe@jct.ac.*
**O. Keren**[b], *PhD, Senior Lecurer, Osnat.Keren@biu.ac.il*
[a]*Jerusalem College of Technology — Lev Academic Center, 21 Havaad Haleumi, P.O.B. 16031, Jerusalem, 91160, Israel*
[b]*Bar-Ilan University, Ramat Gan, 5290002, Israel*

***Introduction:*** *In the work “Error-Correcting Codes for Ternary Content Addressable Memories”, Krishnan et al. show that under certain assumptions, using $2t + 1$ copies of a word is an optimal strategy for guaranteeing the reliable operation of a ternary content addressable memory in the presence of up to $t$ errors.* ***Purpose:*** *To present a new proof of the results of Krishnan et al. about coding for ternary content addressable memories and to extend these results somewhat.* ***Results:*** *A new logic-oriented extension of the Hamming distance is presented. Making use of this new distance, an alternate proof that repetition-based coding is optimal over the set of non-context-oriented codes is provided. The new proof allows the results of Krishnan et al. to be extended to cases where some information about the memory organization is available to the code designer. It is shown, for example, that the number of necessary redundancy bits in a non-context-oriented code cannot be reduced by assuming that the memory organizer stores codes in a particularly effective order.* ***Practical relevance:*** *The results described in this paper make clear that a repetition code is the optimal code for protecting the data stored in a ternary content addressable memories from errors. The new proof presented in this paper allows the results of Krishnan et al. to be extended to certain cases where some information about the memory organization is available to the code designer.*

***Keywords*** *— Content Addressable Memories, Ternary Content Addressable Memories, Context-Oriented Codes, Non-Context-Oriented Codes.*

**Citation:** Engelberg S., Keren O. Error-Correcting Codes for Ternary Content Addressable Memories: a New Perspective. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2018, no. 1, pp. 68–73. doi:10.15217/issn1684-8853.2018.1.68

## Overview

Ordinary content addressable memories (CAMs) store completely specified words (i. e., sequences of zeros and ones) [1]. Ternary content addressable memories (TCAMs) store completely specified words or incompletely specified words, words which are sequences of ones, zeros and wildcards (which are denoted by asterisks, “*”s). Wildcards, *s, match ones and zeros. The input to a TCAM is always a completely specified word. In its standard operating mode, a T/CAM notifies its user if a given word matches one or more completely or incompletely specified words stored in it and, if there is a match, informs the user of the address of the first location whose contents the word matches [2].

As the size of individual memory cells shrinks and the number of cells on a single chip grows, the probability of one or more errors in the contents of the memory increases [3, 4]. When a symbol in the memory is corrupted, it can go from being a zero, one, or wildcard to being any element of that set. Error correcting codes for T/CAMs are usually designed under the assumption that while a T/CAM is operating in its standard operating mode, *there is no simple way to read the contents of the T/CAM*.

There are several ways to make T/CAMs more tolerant of soft errors [5–10]. One of which is to add a sense amplifier (and some additional logic) at the end of each match line [7]. When the sense amplifier is present, T/CAMs declare a match between the input to the unit and a value stored in the unit as long as the two items differ in $t$ or fewer places. Throughout this paper, it is assumed that T/CAMs with sense amplifiers are being used.

In [11], Krishnan et al. prove that under certain assumptions, a simple repetition code, one that consists of $2t + 1$ copies of a word, is an optimal strategy for guaranteeing the reliable operation of a TCAM in the presence of up to $t$ errors. In this note, a clear distinction is made between context- and non-context-oriented coding and a weaker extension of the Hamming distance is described. Using this new logic-oriented distance, a simpler proof that the central theorem of [11] is optimal over the set of non-context-oriented codes is provided. Using the new proof the results of [11] are then extended to cases where some information about the memory organization is available to the code designer.

## Context-Oriented and Non-Context-Oriented Codes

T/CAMs are often used to associate rules with headers. If each header of interest, say $h$, is associated with a separate set of rules, $R(h)$, then one uses a CAM which searches its memory for the specific header. Often, however, a *subset* of headers has the same rules associated with each of its elements. In such a case, by using a TCAM and making judicious use of wildcards, it is possible to use many fewer entries to list all the relevant *sets* of headers and their associated rules.

When considering code design and use in TCAMs, we have found it useful to split the task into two subtasks: *code design* and *memory organization*. Code design is the design of a code by adding redundancy symbols to headers or sets of headers to form codewords, while memory organization involves taking a (subset of a) code and using it in a way that optimizes the utilization of the memory (and other resources) [12, 13]. In practice, the code designer and the memory organizer are often the same person.

When dealing with code design for TCAMs, we consider two cases: the design of context-oriented and of non-context-oriented codes. In the first case, the code's designer knows something about the set of headers to be stored, and s/he can use this information in designing the codes.

In the second case, the code designer is trying to design a code that will be useful for any possible set of completely- or incompletely-specified headers. The code designer may or may not know something about the general principles the memory organizer will use when storing codewords in the memory, but the code designer *does not know anything at all about the rules associated with the headers the memory organizer will be storing*.

When designing a non-context-oriented code, the code designer must assign a code to each of the $3^k$ possible *Boolean cubes* that can be used to represent completely- or incompletely-specified headers in the TCAM.

*Definition 1* (Boolean cube). A Boolean cube of order $w$ is a vector of length $n$ with $w$ wildcards and $n - w$ elements from {0, 1}. A Boolean cube of order $w$ represents $2^w$ distinct binary vectors (and "matches" those vectors).

In what follows, a cube of length $k$, is referred to as a header; i. e., $h \subseteq \{0, 1, *\}^k$. The term "codeword" is used to refer to a cube, $c$, of length $n$, and the input to a TCAM, $y$, is always a fully specified binary word: $y \in \{0, 1\}^n$. Both headers and codewords are Boolean cubes; in certain contexts the term header is more appropriate, and in others, the term codeword is more suitable.

To fix ideas, first several simple examples of how a memory organizer, who knows which rules are associated with each header, can combine headers are presented and then several code design examples are presented.

**Example 1.**

Denote by $H \subseteq \{0, 1\}^4$ the set of completely-specified headers of interest without added redundancy. Let

$$H = \left\{ \begin{array}{c} h_1 = (1000),\ h_2 = (1001),\ h_3 = (1100) \\ h_4 = (1101),\ h_5 = (0000) \end{array} \right\}$$

and let the header $h_i$ be associated with the rule $R(h\_i)$, $i = 1, ..., 5$. We consider three scenarios and the ways of combining headers that they lead to.

1. If the rules for $h_1, ..., h_4$ are the same and only $h_5$ has a different rule, then the memory organizer can encode the first four headers as a Boolean cube of order two of the form $h_6 = (1{*}0{*})$ and use a total of two entries.

2. If $R(h_1) = R(h_2)$, $R(h_3) = R(h_4)$, and $R(h_5)$ are three distinct rules, then the memory organizer can encode the headers using two Boolean cubes of order one, $h_7 = (100{*})$ and $h_8 = (110{*})$, and the single cube $h_5 = (0000)$.

3. If $R(h_1) = R(h_5) \neq R(h_2)$ but $R(h_2) = R(h_3) = R(h_4)$, then the memory organizer can encode the three headers $h_2, h_3, h_4$ as a cube of order two, $h_6 = (1{*}0{*})$, and the two headers $h_1$ and $h_5$ as a cube of order one, $h_9 = ({*}000)$. S/he must then place $h_9$ before the cube of order two. As T/CAMs return the *first* address at which a word matches, this *ordering* ensures that the correct rule is associated with $h_1$ and $h_5$.

In order to make the difference between context-oriented codes and non-context-oriented codes clear, the scenarios from Example 1 are now reconsidered from the point of the code designer. As the point of any code is to make different codewords "maximally distant" from one another, it is crucial that the distance between cubes be defined precisely.

*Definition 2* (a logic-oriented distance). Let $c_i = (c_{i,n}, ..., c_{i,2}, c_{i,1})$ and $c_j = (c_{j,n}, ..., c_{j,2}, c_{j,1})$ be two Boolean cubes. The distance $d(c_i, c_j)$ is defined as

$$d(c_i, c_j) = \left| \left\{ w \mid c_{i,w} \neq c_{j,w},\ c_{i,w},\ c_{j,w} \neq *,\ 1 \leq w \leq n \right\} \right|.$$

*In what follows*: distance will always be taken to mean this logic-oriented distance.

This logic-oriented distance is an extension of the Hamming distance, and when both cubes are of order zero, our logic-oriented distance reverts to the Hamming distance. Two points are particularly worth noting. The logic-oriented definition of distance given here differs from the hardware-oriented distance defined in [11] and is weaker than the distance defined there. For example, using our logic-oriented definition of the distance, $d((000), ({*}{*}{*})) = 0$. Using the more hardware-oriented definition in [11], the distance between these elements is 3. (See [7] for more information about this alternate definition of distance.)

Additionally, the logic-oriented extension of the Hamming distance between cubes is not a metric as the triangle inequality does not hold. Consider, for example, $c_1 = (000)$, $c_2 = (110)$, and $c_3 = ({*}{*}0)$.

In this case $d(c_1, c_2) = 2$, $d(c_1, c_3) = d(c_3, c_2) = 0$, and $d(c_1, c_2) > d(c_1, c_3) + d(c_3, c_2)$.

When a code designer develops a context-oriented code, s/he is assumed to know which codes may be of interest to the memory organizer and which codes are definitely not of interest to the memory organizer. S/he may have additional information as well (such as the order in which the cubes are stored in the memory).

Assume that the code designer (who knows the headers and rules the memory organizer is using) has been told to design a code that can correct a single error. At first glance, it would seem that the code designer must add enough redundancy bits to each header to make the distance between any two codewords at least three, but this is not always necessary.

The intersection of two Boolean cubes $c_1$ and $c_2$, $c_1 \cap c_2$, is defined to be

*Definition 3* (intersection of cubes). Let $c_1$ and $c_2$ be two cubes of length $n$, then

$$c_1 \cap c_2 \equiv \{c \in \{0,1\}^n \mid (c \in c_1) \text{ and } (c \in c_2)\}.$$

Cubes are said to be *disjoint* if their intersection is the null set.

It will be shown that when there are non-disjoint codewords it is sometimes possible to design a single-error-correcting code even if the distance between the codewords is less than three. For disjoint codewords, however, we find that:

**Theorem 1**. *For a (non-context-oriented or context-oriented) code to be t-error correcting, the distance between any two disjoint codewords must be at least* $2t+1$.

This is so because when two stored entries, $h_1$ and $h_2$, are disjoint, codewords that must match the most corrupted possible version of $h_2$ — which may have up to $t$ error — must not get "caught" by the corrupted version of $h_1$ — which may also have up to $t$ errors.

Now the previous scenarios are considered one by one. Redundancy is added to each header in a way that enables the correction of a single error ($t = 1$).

**Example 2.**

1. In the first case, the redundancy can be added as shown in the following table, and it makes the distance between the cubes ≥ 3. Here the codewords — composed of the headers with the requisite redundancy symbols — are each six symbols wide.

| | header | redundancy |
|---|---|---|
| $c_6$ | 1*0* | 00 |
| $c_5$ | 0000 | 11 |

2. In the second case, the redundancy can be added as follows.

| | header | redundancy |
|---|---|---|
| $c_7$ | 100* | 000 |
| $c_8$ | 110* | 110 |
| $c_5$ | 0000 | 011 |

3. In the third and final case, the redundancy can be added as follows.

| | header | redundancy |
|---|---|---|
| $c_9$ | *000 | 00 |
| $c_6$ | 1*0* | 11 |

Note the difference between the first case and the third case: in the first case $d(c_5, c_6) = 3$, and this allows the input to be correctly located in the presence of a single error. In the third case, however, the distance is only two. In classic coding theory, a distance of two does not provide error correction capability; in a TCAM, it is possible. In the third case, the order of the cubes makes the code single-error correcting. Suppose, for example, that the header and redundancy $y$ = (100000) were entered into a TCAM in which the two entries in the table above had been corrupted to

| | header | redundancy |
|---|---|---|
| corrupted $c_9$ | *000 | 10 |
| corrupted $c_6$ | 1*0* | 10 |

Because $d$(corrupted $c_9$, $y$) ≤ 1, the TCAM returns the rule associated with $c_9$. Here, the TCAM works correctly because of the order in which the codewords were stored.

Note that because (*000) ∩ (1*0*) ≠ φ, there are codewords that should be "caught" by both headers. These codewords do not have to be "protected from one another" by the final code, and that is why the final code does not need to maintain a distance of three or greater between different Boolean cubes in order to be single-error correcting.

Because of the fact that the code designer was privy to the headers to be used by the memory organizer, the code designer was able to design relatively efficient codes. As will be seen shortly, when the code designer does not have this foreknowledge, there is no way to achieve such efficiencies.

When building a complete system that employs a TCAM that uses an error-correcting code, it is necessary to implement an encoder that encodes the header being searched for. A simple repetition code turns out to be the optimal code in the non-context-oriented case, and the corresponding encoder has complexity approaching zero. In the case of context-oriented codes, the complexity of the encoder is an open question. If the redundancy were taken to be an arbitrary function of the header, then the encoder would suffer from the "Shannon effect" [14 (Ch. 5), 15], and it would generally require a number of gates that grows exponentially in the number of bits being encoded. At that point, it might be preferable to build a combinatorial circuit that returns the "address" of the header without bothering with a TCAM and error correcting codes.

As our focus is on non-context-oriented codes, we do not consider context-oriented codes further.

## Non-Context-Oriented Coding

### Constraints Imposed on the Code Designer

When designing a non-context-oriented code, the code designer does not know which headers $H$ contains and certainly does not know the rules to be associated with each header. S/he must, therefore, associate redundancies with each of the $3^k$ possible headers. S/he must encode the headers in such a way that the distance between any pair of non intersecting headers $h_i$ and $h_j$ (i. e., cubes for which $h_i \cap h_j = \varphi$), that the memory organizer might store together in the T/CAM is at least $2t+1$. In the following, the discussion is restricted to systematic codes — codes for which the information appears in the final coded binary vector in its original form.

### The Repetition Code is Optimal

One way of making it possible to correct $t$ errors is to store $2t+1$ copies of the value to be stored [11] — to use a repetition code. When using a repetition code, if $t$ (or fewer) bits of the stored value get corrupted, the TCAM still recognizes that the stored value matches any value it should have matched, and it does not match any value it should not have matched. Though this scheme looks incredibly wasteful, using our weaker, logic-oriented distance, we provide a proof that there is no error correcting code for a TCAM that is more efficient.

**Theorem 2**. *The repetition code of length 3k and size* $3^k$ *is the optimal non-context-oriented single error correcting code for headers of length k.*

*Proof*: The proof proceeds by producing a sequence that requires at least $3k$ ternary bits to protect against a single error. As the elements of the sequence used generally have many "trailing zeros," the cube ($abc$0...0) is denoted by $C_{abc}$.

Because the code designer does not know the elements of $H$, s/he must encode every possible "header" — whether completely or incompletely specified. In particular, s/he must encode the header $C_0 = (0...0)$. Without loss of generality, assume that the header $C_0$ is associated with a "tail" of redundancy bits that are all zeros. (Any completely specified header will be associated with a completely specified redundancy: using a wildcard as part of the redundancy adds nothing to the distance between the codewords.)

Since the memory organizer may choose to use the header $C_1 = (10...0)$ the code has to be able to distinguish between $C_0$ and $C_1$. Thus, $C_1$'s redundancy must have at least two ones. Denoting the codeword composed of a header $C_{pattern}$ and its associated redundancy by $\tilde{C}_{pattern}$, the distance from $\tilde{C}_0$ to $\tilde{C}_1$ is now found to be at least three.

On the other hand, the memory organizer may choose to combine $C_0$ and $C_1$ to form $C_* = (*0\ldots0)$. As the codeword associated with this header, $\tilde{C}_*$, must "catch" $\tilde{C}_0$ and $\tilde{C}_1$, $\tilde{C}_*$'s redundancy must have *s in any place where the redundancies associated with $C_0$ and $C_1$ differ. As they differ in at least two places, $\tilde{C}_*$'s redundancy has at least tvwo *s.

Next consider $C_{01} = (010...0)$. Since $C_{01} \cap C_* = \varphi$, $\tilde{C}_{01}$ has to be distinguishable from $\tilde{C}_*$. That is, $\tilde{C}_{01}$'s redundancy must differ by two non-star values from non-star values in each of the preceding redundancies. That brings us to at least four redundancy bits.

The redundancy associated with $C_{**} = (**0\ldots0)$ must have *s in any locations where any of the 0-order cubes it contains/covers have redundancy bits that differ from one another. Thus, there must be at least four *s in the redundancy associated with $C_{**}$. (The redundancies associated with $\tilde{C}_0$ and $\tilde{C}_1$ differ from one another in two places, and $\tilde{C}_{01}$'s redundancy bits differ from each of theirs in two other places.)

Finally, consider $C_{001} = (0010...0)$. Since $C_{001} \cap C_{**} = \varphi$, its associated redundancy bits must differ in at least two non-* locations from those of $C_{**}$. Thus, its redundancy has at least 6 bits. This pattern continues; for each bit that is added to the header, at least two bits must be added to the redundancy. Thus, any single-error correcting code must have at least $2k$ redundancy bits. It is simple to extend this proof the more general case of a $t$-error correcting code; at each step rather than adding two bits, one must add $2t$ bits. This proves the following theorem.

**Theorem 3**. *The repetition code of length* $(2t+1)k$ *and size* $3^k$ *is the optimal non-context-oriented t-error correcting code for headers of length k.*

A simple consequence of this theorem is that if one needs an ECC for a TCAM, one might as well use the extremely simple repetition code — for no other code can use fewer bits.

## Variations on a Theme

We now prove two corollaries of Theorem 3 that concern non-context-oriented codes for which we have some general information about the behaviour of the memory organizer.

From the proof of Theorem 2, it follows that even when the code designer knows that the memory organizer will not use headers that contain one another (for example, that the header (1000) will not be used if the header (**00) is used), the code designer cannot use this knowledge to produce a more efficient non-context-oriented code. Stated more formally, the proof of Theorems 2 and 3 also suffices to prove the following corollary.

***Corollary 1.*** *Knowing that $h_i \not\subset h_j$, $\forall$ $h_i$, $h_j$, $i \neq j$ cannot be used to reduce the number of redundancy bits in a non-context-oriented code.*

Actually, the sequence used in the proofs of Theorem 2 and 3 allows somewhat more to be said.

***Corollary 2.*** *The number of necessary redundancy bits in a non-context-oriented code cannot be reduced by assuming that the memory organizer stores codes in a particularly effective order.*

*Proof*: Consider the sequence of choices used in the proof of Theorem 2 above and the way they determine the necessary number of redundancy bits. The proof starts by considering $\tilde{C}_0$ and $\tilde{C}_1$. These codes must have a distance of at least three, and as $d(C_0, C_1) \geq 1$, order cannot be used to reduce the necessary distance. Thus $\tilde{C}_0$ and $\tilde{C}_1$'s redundancies differ in at least two places. As was seen in the proof, the code associated with $C_*$ must have at least two wildcards in its redundancy. When the code associated with $C_{01}$ is considered, nothing is known about what the memory organizer is planning. It is possible that the memory organizer will only use $\tilde{C}_*$ and $\tilde{C}_{01}$. Clearly order cannot help save bits here, and $\tilde{C}_{01}$'s redundancy must have at least four bits. At each stage in the construction of the sequence in the proof, it is clear that order cannot reduce the number of necessary redundancy bits. Thus, order-related strategies do not allow the number of redundancy bits to be reduced.

## Conclusion

Unlike CAMs, TCAMs can store wildcards. This makes them useful when one wants to reduce the size of the memory. Using the new logic-oriented distance, an alternate proof that the $2t+1$ repetition code is the optimal non-context-oriented code is provided. Moreover, it is shown that knowledge of the order in which codewords will be stored in the TCAM cannot help produce a more efficient non-context-oriented code.

## References

1. Pagiamtzis K., and Sheikholeslami A. Content-Addressable Memory (CAM) Circuits and Architectures: a Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, March 2006, vol. 41, no. 3, pp. 712–727.
2. Arsovski I., Chandler T., and Sheikholeslami A. A Ternary Content-Addressable Memory (TCAM) Based on 4T Static Storage and Including a Current-Race Sensing Scheme. *IEEE Journal of Solid-State Circuits*, Jan. 2003, vol. 38, no. 1, pp. 155–158.
3. Seifert N., Gill B., Foley K., Relangi P. Multi-Cell Upset Probabilities of 45 nm High-K Metal Gate SRAM Devices in Terrestrial and Space Environments. *Proc. IEEE Int'l Reliability Physics Symp. (IRPS '08)*, 2008, pp. 181–186.
4. Satoh S., Tosaka Y., Wender S. A. Geometric Effect of Multiple-Bit Soft Errors Induced by Cosmic Ray Neutrons on DRAMs. *IEEE Electron Device Letter*, June 2000, vol. 21, no. 6, pp. 310–312.
5. Baeg S., Wen S., and Wong R. Minimizing Soft Errors in TCAM Devices: A Probabilistic Approach to Determining Scrubbing Intervals. *IEEE Trans. on Circuits and Systems*, Reg. papers, Apr. 2010, vol. 57, no. 4, pp. 814–822.
6. Noda H., Dosaka K., Morishita F., and Arimoto K. A Soft-Error-Immune Maintenance-Free TCAM Architecture with Associated Embedded DRAM. *Proc. IEEE Custom Integrated Circuits Conf.*, 2005, pp. 451–454.
7. Pagiamtzis K., Azizi N., and Najm F. N. A Soft-Error Tolerant Content-Addressable Memory (CAM) using an Error-Correcting-Match Scheme. *Proc. IEEE Custom Integrated Circuits Conf. (CICC '06)*, 2006, pp. 301–304.
8. Lee H.-J. Immediate Soft Error Detection using Pass Gate Logic for Content Addressable Memory. *Electronics Letters*, 2008, vol. 44, no. 4, pp. 269–270.
9. Zhang W. Replication Cache: A Small Fully Associative Cache to Improve Data Cache Reliability. *IEEE Trans. Computers*, Dec. 2005, vol. 54, no. 12, pp. 1547–1555.
10. Pontarelli S., Ottavi M., Salsano A. Error Detection and Correction in Content Addressable Memories. *Proc. IEEE 25th Int'l Symp. "Defect and Fault Tolerance in VLSI Systems" (DFT '10)*, Oct. 2010, pp. 420–428.
11. Krishnan S. C., Panigrahy R., and Parthasarathy S. Error-Correcting Codes for Ternary Content Addressable Memories. *IEEE Trans. on Comp.*, 2009, vol. 58, no. 2, pp. 275–279.
12. Karthik Lakshminarayanan, Anand Rangarajan, and Srinivasan Venkatachary. Algorithms for Advanced Packet Classification with Ternary CAMs. *SIGCOMM Comput. Commun. Rev.*, Aug. 2005, vol. 35, no. 4, pp. 193–204.
13. Alex X. Liu, Chad R. Meiners, and Eric Torng. TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs. *IEEE/ACM Trans. Netw*, April 2010, vol. 18, no. 2, pp. 490–500.
14. Karpovsky M. G., Stankovic R. S., Astola J. T. *Spectral Logic and its Applications for the Design of Digital Devices*. John Wiley & Sons, 2008. 598 p.
15. Wegener I. *The Complexity of Boolean Functions*. New York, John Wiley & Sons, 1987. 458 p.

УДК 621
doi:10.15217/issn1684-8853.2018.1.68

## Использование помехоустойчивых кодов в системах троичной ассоциативной памяти: новые перспективы

Энгельберг Ш.[а], PhD, мат., доцент, декан, shlomoe@jct.ac.
Керен О.[б], PhD, старший преподаватель, Osnat.Keren@biu.ac.il
[а]Иерусалимский технологический колледж, Хавад Халейми, 21, П.О.Б. 16031, Иерусалим, 91160, Израиль
[б]Университет Бар-Илан, Рамат-Ган, 5290002, Израиль

**Введение:** в статье «Использование помехоустойчивых кодов в системах троичной ассоциативной памяти» Кришнан с соавторами показал, что при определенных предположениях использование $2t+1$ копий слов является оптимальной стратегией, гарантирующей реализацию троичных контентных адресных элементов памяти при наличии до $t$ ошибок. **Цель:** вывести новое доказательство результатов Кришнана с соавторами о кодировании для троичных контентно-адресных элементов памяти в целях расширения этих результатов для возможных случаев организации памяти. **Результаты:** представлено новое логически-ориентированное расширение расстояния Хэмминга, благодаря которому предложено альтернативное доказательство того, что кодирование, основанное на повторении, является оптимальным, охватывающим серии не контекстно-ориентированных кодов. Новое доказательство также позволяет расширить результаты Кришнана с соавторами на случаи, когда некая информация об организации памяти имеется в наличии у разработчика кодов. Показано, например, что число необходимых для репродукции битов в не контекстно-ориентированном коде не может быть выкинуто (опущено), если организатор памяти сохраняет коды в частично-эффективном порядке. **Практическая значимость:** результаты данной работы ясно показывают, что повторный код является оптимальным кодом для защиты от ошибок для информации, хранимой в троичных контекстно-адресных элементах памяти. Новое доказательство, предложенное в статье, позволяет расширить результаты Кришнана с соавторами на ряд случаев, когда некоторая информация об организации памяти находится в распоряжении разработчика кодов.

**Ключевые слова** — контентно-адресные элементы памяти, троичные контекстные адресные элементы памяти, не контекстно-ориентированные коды.

**Цитирование:** Engelberg S., Keren O. Error-Correcting Codes for Ternary Content Addressable Memories: a New Perspective // Информационно-управляющие системы. 2018. № 1. С. 68–73. doi:10.15217/issn1684-8853.2018.1.68
**Citation:** Engelberg S., Keren O. Error-Correcting Codes for Ternary Content Addressable Memories: a New Perspective. *Informatsionno-upravliaiushchie sistemy* [Information and Control Systems], 2018, no. 1, pp. 68–73. doi:10.15217/issn1684-8853.2018.1.68