*electronics*

*Article*

# Optimal Implementations of 8b/10b Encoders and Decoders for AMD FPGAs

**Stefan Popa** *, **Mihai Ivanovici** and **Radu-Mihai Coliban**

Department of Electronics and Computers, Faculty of Electrical Engineering and Computer Science, Transilvania University of Brasov, B-dul Eroilor nr. 29, 500036 Brasov, Romania; mihai.ivanovici@unitbv.ro (M.I.); coliban.radu@unitbv.ro (R.-M.C.)
* Correspondence: stefan.popa@unitbv.ro

**Abstract:** The 8b/10b IBM encoding scheme is used in a plethora of communication technologies, including USB, Gigabit Ethernet, and Serial ATA. We propose two primitive-based structural designs of an 8b/10b encoder and two of an 8b/10b decoder, all targeted at modern AMD FPGA architectures. Our aim is to reduce the amount of resources used for the implementations. We compare our designs with implementations resulting from behavioral models as well as with state-of-the-art solutions from the literature. The implementation results show that our solutions provide the lowest resource utilization with comparable maximum operating frequency and power consumption. The proposed structural designs are suitable for resource-constrained data communication protocol implementations that employ the IBM 8b/10b encoding scheme. This paper is an extended version of our paper published at the 2022 International Symposium on Electronics and Telecommunications (ISETC), Timisoara, Romania, 10–11 November 2022.

**Keywords:** 8b/10b encoding; encoder; decoder; FPGA; LUT; BRAM; SRAM; serial links

**Citation:** Popa, S.; Ivanovici, M.; Coliban, R.-M. Optimal Implementations of 8b/10b Encoders and Decoders for AMD FPGAs. *Electronics* **2024**, *13*, 1062. https://doi.org/10.3390/electronics13061062

Academic Editors: Francis Balestra and Gerard Ghibaudo

Received: 16 February 2024
Revised: 8 March 2024
Accepted: 10 March 2024
Published: 13 March 2024

## 1. Introduction

Field-Programmable Gate Arrays (FPGAs) are integrated circuits that can be configured after manufacturing. The logic functions that are used for configuration are typically obtained through synthesis from a user-provided behavioral description of the design in the form of RTL (Register Transfer Level) code written using an HDL (Hardware Description Language). Additional inputs to the synthesis tool include libraries describing the resources on the target device, a list of constraints that the design must conform to, and a list of arguments defining the synthesis strategy. The result of the synthesis process is a netlist, i.e., a structural description of the design in the form of interconnected primitive blocks. The efficiency of the resulting netlist depends strongly on the RTL description and the additional inputs, as well as on the synthesis tool. If the result is not satisfactory, the RTL description and the inputs must be changed accordingly. For greater control over the results, the user can directly instantiate primitive blocks in the HDL code to provide a structural description of the entire design or a part of it, making it more efficient, with the drawbacks of increased complexity of the design process and reduced portability.

The basic resource used for the implementation of user-defined logic in FPGAs developed by AMD is the Configurable Logic Block (CLB) [1]. A CLB consists of slices, with each slice containing several Look-Up Tables (LUTs), multiplexers, storage elements, and carry logic for arithmetic functions. All the CLBs in the FPGA are interconnected through configurable routing resources. LUTs are used as Boolean function generators, the implementation of the functions being based on the corresponding truth tables. On all FPGAs developed by AMD since Virtex-5 except for the latest Versal Adaptive Compute Acceleration Platform (ACAP) [2], each LUT has six inputs and two outputs and can be used to implement either a Boolean function of six variables, two Boolean functions of the same five variables, or two Boolean functions of three and two or fewer separate variables.

The AMD Versal ACAP integrates software-programmable processors and accelerator engines with a new FPGA fabric. Each Versal LUT has four outputs and can additionally implement two Boolean functions of the same six variables [3]. Other embedded resources available on AMD FPGAs are Blocks of Static RAM (BRAM), arithmetic logic units, and transceivers. In [4], a comparison of modern FPGA LUT characteristics from the three leading producers (including AMD) is presented.

8b/10b is a line code proposed by Widmar and Franaszek in [5]. The code defines a mapping between an eight-bit data word and a ten-bit symbol, with the objectives of achieving Direct Current (DC) balance, a running disparity (RD) between -2 and 2 and frequent transitions for clock recovery, while providing special symbols for synchronization and custom control functions. A communication channel that employs 8b/10b is implemented using an encoder–decoder pair, with the former being a component in the transmitter and the latter a part of the receiver. A multitude of 8b/10b encoding schemes have been proposed [6,7]. In this paper, we focus on the IBM 8b/10b code [5], which is used in a wide variety of communication standards such as Aurora [8], Gigabit Ethernet [9], Serial ATA [10], and USB [11].

We propose two implementations of an 8b/10b encoder and two of an 8b/10b decoder, all based on structural descriptions targeted at the configurable logic of AMD FPGAs, which minimize resource utilization without compromising features. We detail the proposed encoders and decoders and compare them with other solutions from the literature and equivalent behavioral descriptions in terms of their resource utilization, longest propagation delays, and power consumption. The proposed encoders and decoders have the lowest resource utilization while achieving comparable maximum operating frequency and power draw. The structural models can be used on all AMD FPGAs architectures since Virtex-5, including Versal ACAP, as they only require six-input LUTs with two outputs or BRAMs and do not employ architecture-specific carry or multiplexing logic. The designs might be even more compact on Versal devices due to their more complex LUTs, but are optimal for pre-Versal architectures. While the per instance gains are relatively small, they can be significant in large custom designs employing multiple communication channels using 8b/10b encoding, e.g., custom designs for complex physics experiments sich as [12] or network FPGA-based hardware with a large amount of channels.

Section 2 contains the following three subsections: (i) Section 2.1 explains the rules of the IBM encoding scheme with examples; (ii) Section 2.2 presents the proposed 8b/10b encoders; and (iii) Section 2.3 details the proposed 8b/10b decoders. Sections 2.2 and 2.3 employ a similar structure, starting with the module's block diagram and interface, then presenting the cascading feature, continuing by explaining the top-view architecture, and ending with two subsubsections, one detailing the BRAM-based structural module and the other the LUT-based one. Section 3 presents the simulation and implementation results in terms of resource utilization, propagation delay, and power consumption, followed by a comparison them with behavioral models and state-of-the-art solutions. For a fair comparison, all implementations targeted the AMD Kintex-7 KC705 evaluation board [13] having the XC7K325T-2FFG900C FPGA device. Section 4 presents a rationale concerning which codecs are suitable for various applications employing the IBM 8b/10b line code and includes various observations. Finally, the paper ends with a brief conclusion.

## 2. Materials and Methods

### *2.1. The IBM 8b/10b Code*

In the IBM 8b/10b encoding scheme, the eight input bits are labeled *HGFEDCBA*, with *H* being the most significant bit (MSB) and *A* the least significant bit (LSB). The input is mapped to a symbol consisting of ten bits, labeled *abcdeifghj*, with *a* being the LSB and *j* being the MSB. The mapping is done using two encoding schemes: (i) 5b/6b, mapping *EDCBA* to *abcdei*; and (ii) 3b/4b, mapping *HGF* to *fghj*. Several mapping examples are detailed in Tables 1 and 2 for the 5b/6b and 3b/4b encoding schemes, respectively. These are explained in the following paragraphs; for the complete encoding scheme, refer to [5].

**Table 1.** Several examples of the 5b/6b encoding scheme $(0 \le y \le 7 = 2^3 - 1)$.

| **Input** | | **Output *abcdei*** | |
|---|---|---|---|
| **Notation** | ***EDCBA*** | **RD = −1** | **RD = +1** |
| *D.0.y* | 00000 | 100111 | 011000 |
| *D.1.y* | 00001 | 011101 | 100010 |
| *D.3.y* | 00011 | 110001 | 110001 |
| *D.7.y* | 00111 | 111000 | 000111 |
| *D.23.y* or *K.23.7* | 10111 | 111010 | 000101 |
| *D.27.y* or *K.27.7* | 11011 | 110110 | 001001 |
| *D.28.y* | 11100 | 001110 | 001110 |
| *K.28.y* | 11100 | 001111 | 110000 |
| *D.29.y* or *K.29.7* | 11101 | 101110 | 010001 |
| *D.30.y* or *K.30.7* | 11110 | 011110 | 100001 |

**Table 2.** Several examples of the 3b/4b encoding scheme $(0 \le x \le 31 = 2^5 - 1)$.

| **Input** | | **Output *fghj*** | |
|---|---|---|---|
| **Notation** | ***HGF*** | **RD = −1** | **RD = +1** |
| *D.x.0* or *K.x.0* | 000 | 1011 | 0100 |
| *D.x.1* | 001 | 1001 | 1001 |
| *K.28.1* | 001 | 0110 | 1001 |
| *D.x.3* or *K.28.3* | 011 | 1100 | 0011 |
| *D.x.6* | 110 | 0110 | 0110 |
| *K.28.6* | 110 | 1001 | 0110 |
| *D.x.P7* | 111 | 1110 | 0001 |
| *D.x.A7* | 111 | 0111 | 1000 |
| *K.x.7* | 111 | 0111 | 1000 |

There are 256 *normal* data symbols, labeled in 8b/10b notation as *D.x.y* with the meaning from Equation (1). Additionally, there are twelve *special* control symbols used for synchronization or other user-defined functions, labeled as *K.28.y* and *K.x.7* and explained in Equations (2) and (3), respectively. They are detailed in Table 3.

$$\begin{aligned} &D.x.y = HGFEDCBA, H = \text{MSB}, A = \text{LSB}, \\ &\quad \text{where } x = EDCBA, 0 \le x \le 31 = 2^5 - 1 \\ &\quad \text{and } y = HGF, 0 \le y \le 7 = 2^3 - 1 \end{aligned} \tag{1}$$

$$\begin{aligned} &K.28.y = HGFEDCBA, H = \text{MSB}, A = \text{LSB}, \\ &\quad \text{where } EDCBA = 28 \text{ and } y = HGF, 0 \le y \le 7 = 2^3 - 1 \end{aligned} \tag{2}$$

$$\begin{aligned} &K.x.7 = HGFEDCBA, H = \text{MSB}, A = \text{LSB}, \\ &\quad \text{where } HGF = 7 \text{ and } x = EDCBA, x \in \{23, 27, 29, 30\} \end{aligned} \tag{3}$$

The IBM 8b/10b code is constructed in such a way that there is a maximum of five consecutive identical bits in the encoded data stream. All normal data symbols and nine special symbols contain a maximum of four consecutive bits, while the remaining three special symbols (*K.28.1*, *K.28.5* and *K.28.7*) contain five consecutive identical bits. These three symbols, called *comma* symbols, are used for transmitter-receiver synchronization and are highlighted in Table 3. The integrated circuit detailed in [14,15] employs *K.28.5* in all its input and output data channels for this purpose.

**Table 3.** The twelve *special* control symbols with their 8b/10b notation, input value in decimal and binary, and the resulting encoded binary values for both possible input disparities. The three *comma* symbols having five consecutive bits of the same value are highlighted.

| Input | | | Output *abcdei fghj* | |
|---|---|---|---|---|
| **Notation** | **Decimal** | ***HGF EDCBA*** | **RD = −1** | **RD = +1** |
| *K.28.0* | 28 | 000 11100 | 001111 0100 | 110000 1011 |
| ***K.28.1*** | 60 | 001 11100 | 00**1111 1**001 | 11**0000 0**110 |
| *K.28.2* | 92 | 010 11100 | 001111 0101 | 110000 1010 |
| *K.28.3* | 124 | 011 11100 | 001111 0011 | 110000 1100 |
| *K.28.4* | 156 | 100 11100 | 001111 0010 | 110000 1101 |
| ***K.28.5*** | 188 | 101 11100 | 00**1111 1**010 | 11**0000 0**101 |
| *K.28.6* | 220 | 110 11100 | 001111 0110 | 110000 1001 |
| ***K.28.7*** | 252 | 111 11100 | 00**1111 1**000 | 11**0000 0**111 |
| *K.23.7* | 247 | 111 10111 | 111010 1000 | 000101 0111 |
| *K.27.7* | 251 | 111 11011 | 110110 1000 | 001001 0111 |
| *K.29.7* | 253 | 111 11101 | 101110 1000 | 010001 0111 |
| *K.30.7* | 254 | 111 11110 | 011110 1000 | 100001 0111 |

The difference between the number of high bits (of value 1) and low bits (of value 0) represents the disparity of a binary code. The 5b/6b and 3b/4b encoding schemes produce either a single output code with null disparity (i.e., an equal number of high and low bits, e.g., *D.3.y* and *D.28.y* from Table 1 and *D.x.1* and *D.x.6* from Table 2) or two complementary codes (i.e., one is the bitwise negation of the other, e.g., the rest of the examples from Tables 1 and 2) with a disparity of −2, 0 or +2. The RD at the end of the previous symbol determines the chosen 5b/6b code for the current symbol, while the RD after the selected 5b/6b code establishes the chosen 3b/4b code. The initial RD is considered negative, i.e., more bits are low than high. For the non-null disparity codes, the rule is that the code with disparity opposite to the current RD is used. There are two complementary 5b/6b and 3b/4b codes with null disparity for the *D.7.x* and *D.x.3* symbols detailed in Tables 1 and 2, respectively. In these cases, the code that avoids five consecutive bits of the same value is chosen. The *D.x.7* symbols have four variants for the 3b/4b code, as depicted in Table 2, such that five consecutive bits of the same value are avoided in conjunction with the preceding 5b/6b code.

A typical communication channel employing 8b/10b encoding is depicted in Figure 1. The data to be transmitted are supplied to the 8b/10b encoder one byte at a time. The 8b/10b encoder generates the ten-bit symbols; these are then serialized and the resulting signal is amplified. These constitute the transmitter (TX). At the receiver (RX) side, the signal is deserialized and passed in units of ten bits to a module that searches for *comma* patterns on all possible positions (i.e., 8b/10b Aligner). When this pattern is detected, the alignment is established and the module begins to supply the ten-bit symbols to the 8b/10b decoder, which then outputs the corresponding bytes. A phase-locked loop can use the input serial signal to adjust the clock signal that paces the receiving logic.

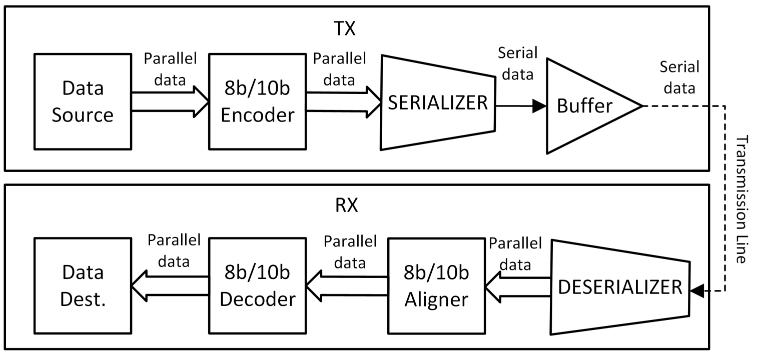


**Figure 1.** A typical communication channel which employs 8b/10b encoding.

## *2.2. The Proposed 8b/10b Encoders*

The top-level block diagram for all of the proposed 8b/10b encoders is represented in Figure 2. The input and output (IO) signals (not including the clock and reset signals) are depicted in Table 4. Except for *rd_casc_o*, all of the output signals are registered; the output signals represented with dashed lines are optional. A similar block diagram and IO signals were presented in [16], without the *rd_casc_o* signal.

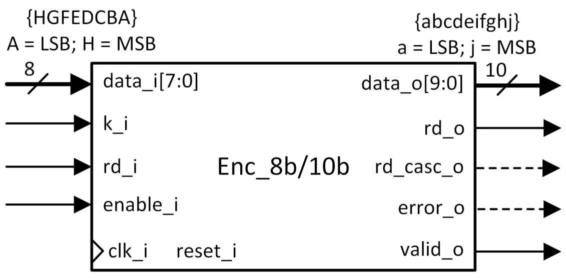


**Figure 2.** The block diagram of the proposed encoder; modified from [16].

**Table 4.** Inputs and outputs of the proposed 8b/10b encoder, based on [16] plus *rd_casc_o*.

| Name | Bits | Description |
|---|---|---|
| *data_i* | 8 | Input data bus representing the byte to be encoded (little-endian, from index 7 to 0 bits *HGFEDCBA*). |
| *k_i* | 1 | Input control signal indicating that the byte must be encoded as a control symbol. |
| *rd_i* | 1 | Input control signal indicating the RD of the 8b/10b stream of bits right before the current byte is encoded. 1 = negative, 0 = positive. |
| *enable_i* | 1 | Input control signal validating the input values. |
| *data_o* | 10 | Output data bus representing the encoded symbol (big-endian, from index 9 to 0 bits abcdeifghj). |
| *rd_o* | 1 | Output control signal indicating the RD of the 8b/10b stream of bits after the current encoded symbol. 1 = negative, 0 = positive. |
| *rd_casc_o* | 1 | Purely combinational output control signal indicating the RD of the 8b/10b stream of bits after the current input byte is encoded. This signal is used for cascading encoders. The *rd_o* signal is obtained by adding a D flip-flop that samples this signal. 1 = negative, 0 = positive. |
| *error_o* | 1 | Output error signal indicating an incorrect input byte value signaled to be encoded as a control symbol. |
| *valid_o* | 1 | Output control signal validating the output values, except for *rd_casc_o*. |

As depicted in Figure 3, to maintain DC balance on the output stream, while the data source provides one byte of data at a time to the encoder, the *rd_o* output signal must be connected directly through a wire to the *rd_i* input while the *rd_casc_o* remains unused. Although this could be achieved internally, we chose to supply the initial RD from the exterior in order to extend the functionality of the encoder. Two or more encoders can be cascaded to allow the encoding of two or more bytes into ten-bit symbols at a time. In Figure 4, two instances of the proposed 8b/10b encoder are cascaded. In this case, the data source provides two bytes of data to the encoder. The *rd_casc_o* one-bit-wide output control signal of the first encoder, representing its *rd_o* value before it is registered, must drive the *rd_i* input signal of the secondary 8b/10b encoder. Two cascaded 8b/10b encoders were used in [17].

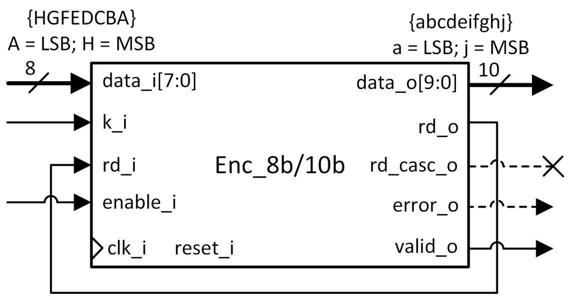


**Figure 3.** The proposed 8b/10b encoder connections when not cascaded.

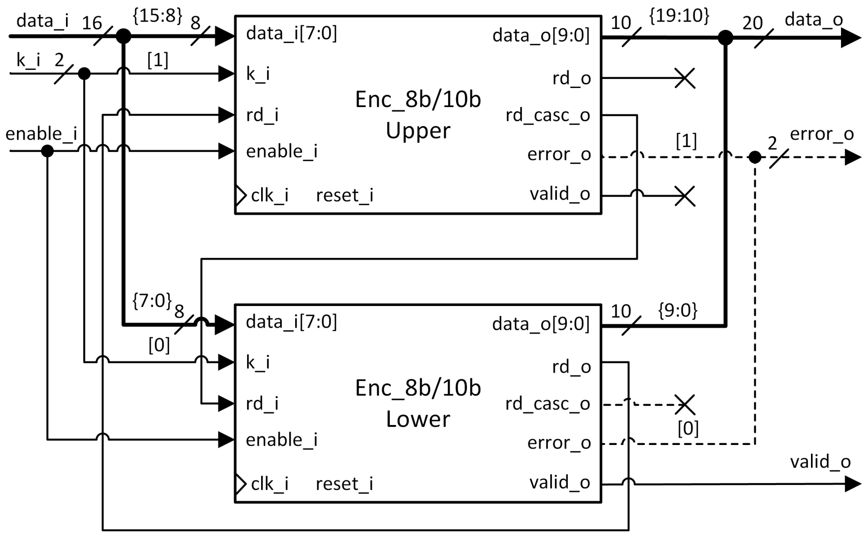


**Figure 4.** Two cascaded instances of the proposed 8b/10b encoder.

The general view of the proposed 8b/10b encoder architecture is presented in Figure 5, with emphasis on the signal dependencies between the 5b/6b and 3b/4b encoding schemes. A similar architecture was detailed in [16], although without the cascading functionality. The Boolean equations of all signals except the encoding schemes are detailed in Appendix A.

Bits *EDCBA* (i.e., the five LSBs) of the byte to be encoded are transformed by the 5b/6b encoding scheme block into bits *abcdei* (i.e., the six LSBs) of the output symbol, taking into consideration the input RD and the *k_i* signal. The 5b/6b code differs between the *K.x* and *D.x* symbols only for $x = 28$, as depicted in Table 1. The 5b/6b encoding scheme block additionally determines the intermediary RD (i.e., the RD after 5b/6b code to be used within the 3b/4b encoding scheme block) as the *rd_interm* signal. Similarly, the 3b/4b encoding scheme block translates bits *HGF* (i.e., the three MSBs) of the byte to be encoded into bits *fghj* (i.e., the four MSBs) of the output symbol, taking into consideration the intermediary RD and the *k_i* signal. To avoid five consecutive bits with identical values in *D.x.7* symbols for certain input RD, the 3b/4b code has alternative values labeled *D.x.A7* while the *D.x.P7* label is used for the primary encoding, as depicted in Table 2. The *D.x.A7* 3b/4b codes are identical to the *K.x.7* ones. The cases in which the *D.x.A7* code is used are summarized in Equation (4). The *alt_enc* signal from Figure 5 is high in these cases. The 3b/4b encoding scheme block additionally determines the RD after the symbol as the *rd_final* signal, which directly drives the *rd_casc_o* output signal and is sampled by the D flip-flop driving the *rd_o* output signal. The optional error block raises the *error* signal for input byte values that do not correspond to any special symbol if the *k_i* input signal is high.

$$D.x.A7 \text{ is used instead of } D.x.P7 \text{ for } \begin{cases} x \in \{11,13,14\}, & \text{when } rd\_i = 0 \\ x \in \{17,18,20\}, & \text{when } rd\_i = 1 \end{cases} \tag{4}$$

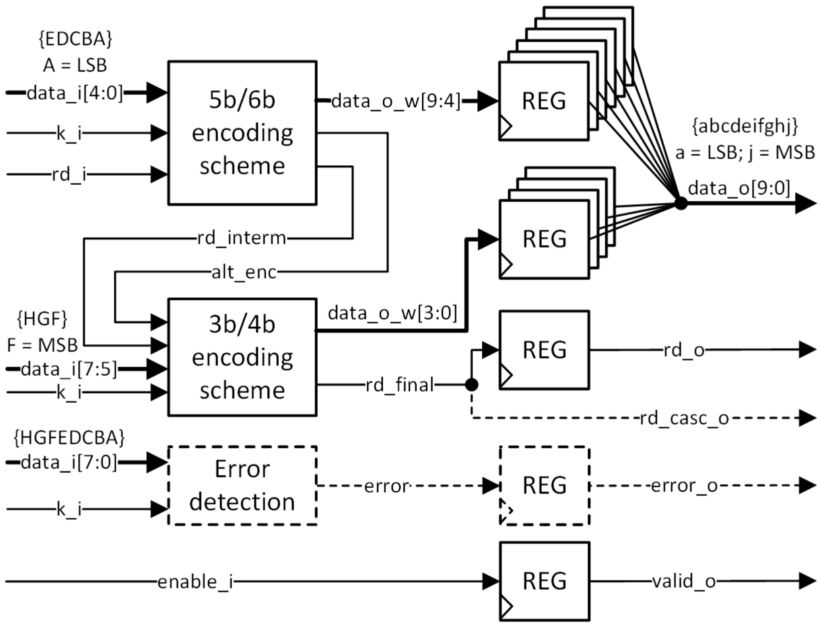


**Figure 5.** Top-view architecture of the proposed 8b/10b encoder, showing the signal dependencies. Modified from [16].

### 2.2.1. BRAM-Based 8b/10b Encoder

We propose BRAM-based organization for the 8b/10b encoder targeted at AMD FPGAs depicted in Figure 6. With the exception of the *clk_i*, *reset_i* and *enable_i* signals, the input signals of the 8b/10b encoder shown in Figure 2 and described in Table 4 are concatenated into a ten-bit-wide bus that represents the input address. The dual-port, dual-clock domain (i.e., asynchronous) BRAM primitive is configured as a dual-port, single-clock domain (i.e., synchronous, common) ROM (Read-Only Memory) with an address space of $2^{10}$ and a twelve-bit wide output data bus. The BRAM acts as a large single LUT. The twelve-bit output data represent the concatenated output signals of the 8b/10b encoder shown in Figure 2 and described in Table 4, excepting the *valid_o* and *rd_casc_o* signals. The *valid_o* output signal is formed by delaying the *enable_i* input signal by one clock cycle using a D flip-flop.

Because the primitive is dual-port and read-only, two independent 8b/10b encoders can be implemented with the same ROM at the additional cost of a D flip-flop for the second *valid_o* signal, as shown in Figure 6. In case of address collision (i.e., the same value on both address ports in the same clock cycle), both read operations are completed successfully, as described in [18].

The content of the ROM is specified during the design phase as an array of hexadecimal values, in this case, $2^{10} = 1024$ three-digit (i.e., hexadecimal, each representing a nibble) values, resulting in 12 Kb of data. A single 18 Kb BRAM configured as 1K $\times$ 18 is sufficient. There is no resource utilization/area benefit in dropping the optional output signal *error_o*.

In Figure 6, it is considered that the BRAMs have a single clock cycle latency, meaning that the behavior of the resulting 8b/10b encoder is identical (except for the lack of the *rd_casc_o* output signal) to the generic one presented in Figure 5 as well as to the one based on LUT primitives presented in the next subsection. The optional embedded registers of the BRAM primitive are not used, nor are the optional configurable logic registers. The optional *rd_casc_o* output signal is a purely combinational logic function of the input signals, as depicted in Figure 5, and as such cannot be implemented using BRAM primitives while maintaining the requirement of single clock cycle latency.

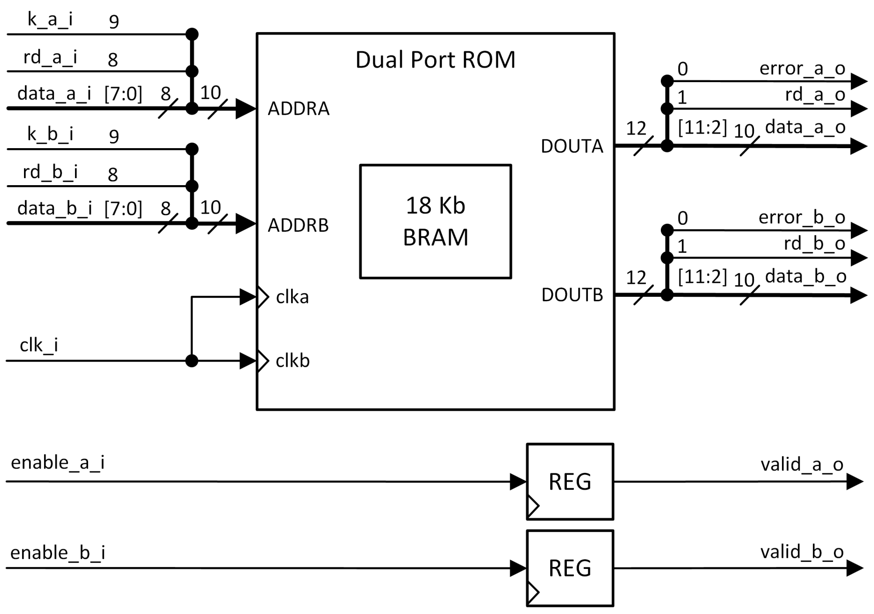


**Figure 6.** The organization of the proposed 8b/10b encoder based on a single 18 Kb AMD BRAM.

### 2.2.2. LUT-Based 8b/10b Encoder

The proposed LUT-based AMD FPGAs-specific 8b/10b encoder organization is detailed in Figure 7. A similar design was briefly presented in [16], though without the cascading functionality and the LUTs configuration. The logic/Boolean functions implemented by the LUTs are listed in Table 5 as little-endian truth table values expressed in hexadecimal. The Boolean equations of all signals except the encoding scheme are detailed in Appendix A and explained in the following paragraphs.

Our first objective was to minimize the amount resources used, while the second was to maximize the operating frequency. Depending on the presence of the error logic, the proposed 8b/10b encoder uses either seventeen LUTs and thirteen flip-flops or fifteen LUTs and twelve flip-flops. The second objective translates into minimizing the longest intra-clock propagation path. The LUTs from Figure 7 are organized by columns from left to right depending on their place in the logic chain from input to output. Regardless of the presence of the error logic, our proposed organization has at most three LUTs from input to output. We expected the routing resources between the used CLBs to have the most influence on the maximum frequency.

The *true* (i.e., implemented in the FPGA fabric with fully independent inputs) AMD six-input LUTs have two outputs: O6, which spans the entire address space (i.e., all 64 addresses), and O5, which covers only the lower half of the address space (i.e., addresses from 0 to 31). The LUT6, LUT4, and LUT3 instances from Figure 7 use solely the O6 output, marked as O. The LUTs having fewer than six inputs are implemented in the FPGA fabric as partially-driven six-input LUTs. The instances marked as LUT6_2 are the six-input LUTs that use both of the output signals and implement two Boolean functions with common inputs. The Boolean function associated with the O5 output can have up to five inputs, while the one associated with the O6 output has the same five inputs plus an additional one and shares the lower half of its truth table with the O5 function. Alternatively, the LUT6_2 can implement two five-input Boolean functions with common inputs and separate output values by driving the 6th input (i.e., I5) to logic 1. The latter configuration is solely used for all LUT6_2 instances depicted in Figure 7. An alternative use of a LUT6_2 is for implementing two Boolean functions with separate inputs, outputs, and values, one with up to two inputs and the other with up to three inputs. In this case, the sixth input is also tied to logic 1. In the proposed 8b/10b encoder organization Figure 7, there is no two-input Boolean function that could be paired with the *next_rd_5b_6b* function (i.e., the disparity after the 5b/6b encoding) in such a configuration.

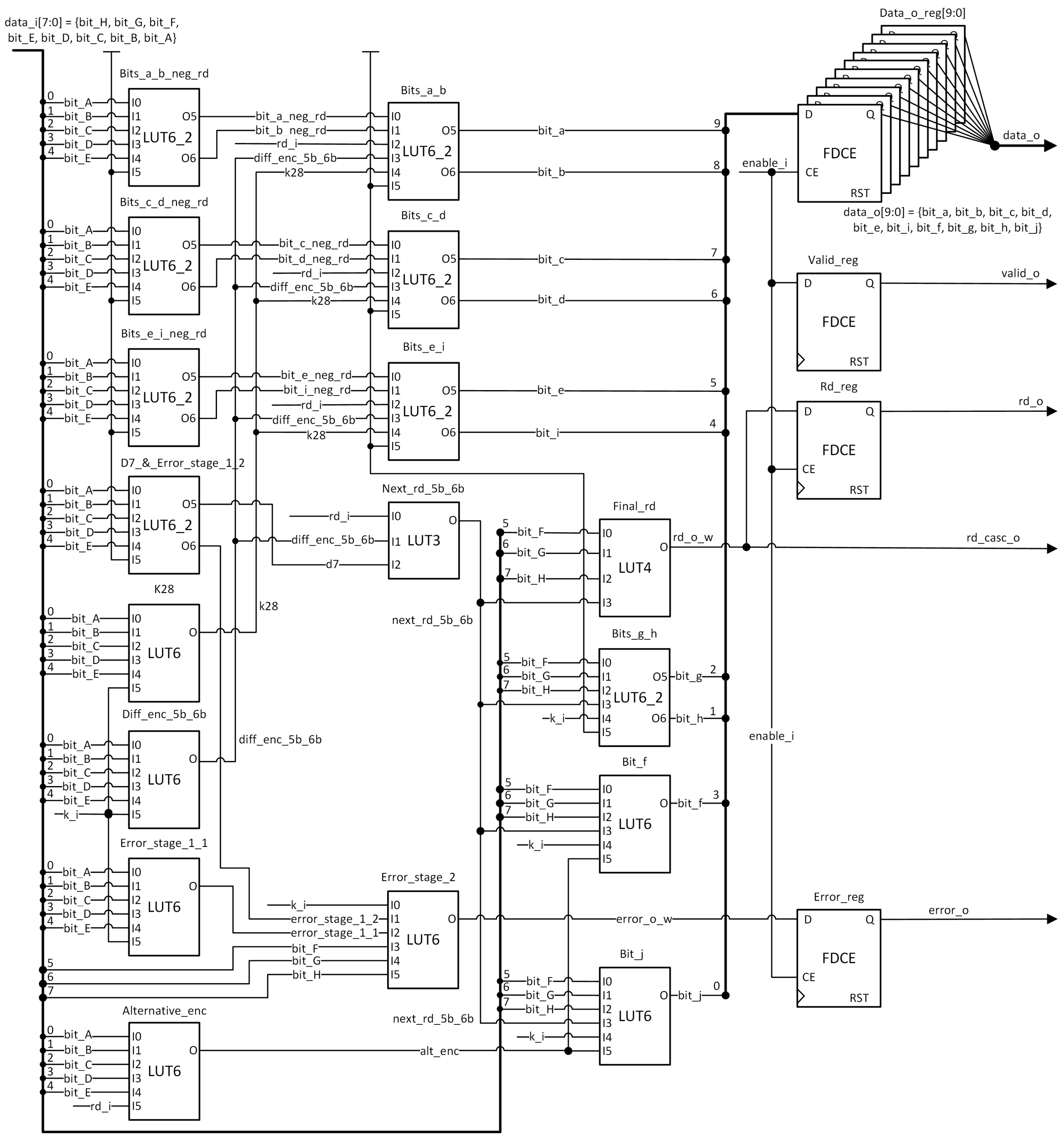


**Figure 7.** The proposed 8b/10b encoder organization based on AMD LUT primitives. Table 5 presents the logic function of each LUT. Modified from [16].

All of the outputs except for *rd_casc_o* are registered in D Flip-flops with Clock Enable (FDCEs) from within the CLBs. The Clock Enable (CE) input ports of all the FDCEs are driven by the *enable_i* input signal, except for the *Valid_reg* instance, which samples this signal in each clock cycle, i.e., its CE port is tied to logic 1.

Each of the three LUT6_2 instances from the top of the leftmost column from Figure 7 determines two bits of the 5b/6b code considering negative input RD (*rd_i* = 1). This means that the 5b/6b code formed by concatenating the *bit_a_neg_rd*, *bit_b_neg_rd*, *bit_c_neg_rd*, *bit_d_neg_rd*, *bit_e_neg_rd*, and *bit_i_neg_rd* wires has positive or null disparity. The special 5b/6b code corresponding to the eight *K.28.y* symbols is not considered, as this would require an additional input (i.e., for *k_i*) to the *Bits_e_i_neg_rd* LUT6_2 instance, which would transform it into two LUT6 instances. The single (i.e., without complement) 5b/6b code for *D.28.y* is *abcdei* = *001110*, and differs from the 5b/6b code of *K.28.y* with positive disparity (i.e., *abcdei* = *00111***1**) only by bit *i*.

**Table 5.** The configurations of all encoder LUTs from Figure 7.

| # | Instance | Hex. Config. Value |
|---|---|---|
| 1 | Bits_a_b_neg_rd | `0x4DCDCDDAABAA2BBD` |
| 2 | Bits_c_d_neg_rd | `0x7E00FE17F0F171E6` |
| 3 | Bits_e_i_neg_rd | `0x8117977FFFFF8001` |
| 4 | D7_error_stage_1_2 | `0x877FFFFF00000080` |
| 5 | K28 | `0x1000000000000000` |
| 6 | Diff_enc_5b_6b | `0xF9818197E9818197` |
| 7 | Error_stage_1_1 | `0x0880800080000000` |
| 8 | Alternative_encoding | `0x0016000000006800` |
| 9 | Bits_a_b | `0xC3CCC3CCA5AAA5AA` |
| 10 | Bits_c_d | `0xC3CCC3CCA5AAA5AA` |
| 11 | Bits_e_i | `0x3C3CC3CCA5AAA5AA` |
| 12 | Next_rd_5b_6b | `0xA6` |
| 13 | Error_stage_2 | `0xAAAAAAAA08888888` |
| 14 | Final_rd | `0x6E91` |
| 15 | Bits_g_h | `0x8778E178BA45DC45` |
| 16 | Bits_f | `0x5DA23BA25DA2BB22` |
| 17 | Bits_j | `0xF10E970EF10E178E` |

The 5b/6b codes with complementary variants as well as the separate *K.28.y* 5b/6b code are signaled directly from the inputs by the *Diff_enc_5b_6b* and *K.28* LUT6 instances, respectively. The three LUT6_2 instances from the top of the middle column determine the output 5b/6b code from its positive or null disparity form by complementing its bits as follows: for bits *a*, *b*, *c*, *d*, and *e* if the input value has two complementary 5b/6b codes (*diff_enc_5b_6b* = 1) and the input RD is positive (*rd_i* = 0), and for bit *i* if the input RD is positive (*rd_i* = 0) and the 5b/6b has two complementary 5b/6b codes (*diff_enc_5b_6b* = 1) and is not *K.28.y* (*k28* = 0) or if the input RD is negative (*rd_i* = 1) and *k28* = 1.

The RD after the 5b/6b encoding and before the 3b/4b encoding (i.e., the *rd_interm* signal in Figure 5 and *next_rd_5b_6b* signal in Figure 7) is a function of the input disparity, the existence of complementary 5b/6b codes, and the nature of the disparity of these codes (non-null or otherwise). Except for *D.7.y*, all 5b/6b codes with complementary variants have a non-null disparity. The *d7* signal differentiates this special case, and is included in the determination of the *next_rd_5b_6b* signal. The final RD (i.e., after the current input byte is encoded, the *rd_final* signal in Figure 5 and *rd_o_w* signal in Figure 7) is determined by the *Final_rd* LUT4 as a function of the *next_rd_5b_6b* signal and the three MSBs of the input byte. The *rd_final* signal is the complement of the *next_rd_5b_6b* signal for the values of the *F*, *G*, and *H* bits, which have 3b/4b codes with non-null disparities (i.e., *FGH* = *000*, *100* or *111*). In all other cases, it copies the value of the *next_rd_5b_6b* signal.

The 3b/4b encoding is determined by one LUT6_2 and three LUT6 instances. Bits *g* and *h* are determined directly from the inputs using the *Bits_g_h* LUT6_2 instance. Bits *f* and *j* might require an alternative encoding, a situation signaled by the *alternative_enc*

LUT6 instance. These two bits are supplied by the *Bit_f* and *Bit_j* LUT6 instances. Bits *g* and *h* of the 3b/4b encoding are not affected by the *alterantive_enc* signal.

The error signal is determined in two stages by three LUT instances: (i) *Error_stage_1_1* and *Error_stage_1_2* in parallel, and (ii) *Error_stage_2* as the final step. The *error_stage_1_2* and *d7* signals are generated by an LUT6_2, as they depend on the same five inputs.

### 2.3. The Proposed 8b/10b Decoders

The top-level block diagram for all of the proposed 8b/10b decoders is depicted in Figure 8. All its input and output signals are described in Table 6 except for the clock and reset signals. All the output signals are registered except for *rd_casc_o*. The output signals represented with dashed lines are optional, and consist of error signals used for debugging and signals used for cascading decoders.

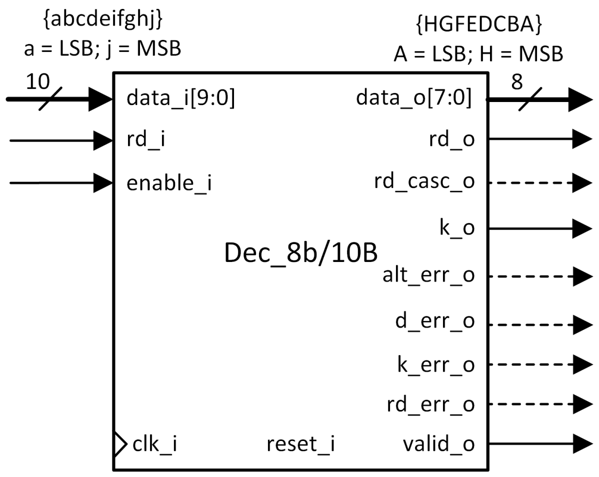


**Figure 8.** The block diagram of the proposed 8b/10b decoder.

**Table 6.** Inputs and outputs of the proposed 8b/10b decoder.

| Name | Bits | Description |
|---|---|---|
| *data_i* | 10 | Input data bus representing the symbol to be decoded (LSB to MSB, bits *abcdeifghj*). |
| *rd_i* | 1 | Input control signal indicating the RD of the 8b/10b stream of bits right before the current symbol. 1 = negative, 0 = positive. |
| *enable_i* | 1 | Input control signal validating the input values. |
| *data_o* | 8 | Output data bus representing the decoded symbol (MSB to LSB, bits *HGFEDCBA*). |
| *rd_o* | 1 | Output control signal indicating the RD of the 8b/10b stream of bits after the current symbol. 1 = negative, 0 = positive. |
| *rd_casc_o* | 1 | Purely combinational output control signal indicating the RD of the 8b/10b stream of bits after the current symbol present on *data_i*. This signal is used for cascading decoders. The *rd_o* signal is obtained by adding a D flip-flop that samples this signal. |
| *k_o* | 1 | Output signal indicating a decoded control symbol. |
| *alt_err_o* | 1 | Output error signal indicating a decoded *D.x.7* symbol with incorrect 3b/4b encoding (i.e., alternative instead of primary or vice versa) resulting in five consecutive identical bits in conjunction with the previous 5b/6b code. |

**Table 6.** *Cont.*

| Name | Bits | Description |
|---|---|---|
| *d_err_o* | 1 | Output error signal indicating input values outside the 5b/6b or 3b/4b encoding schemes. |
| *k_err_o* | 1 | Output error signal indicating a 5b/6b code corresponding to a normal symbol in conjunction with a 3b/4b code of a special code or a 5b/6b code corresponding to a *K.28.x* control symbol in conjunction with a 3b/4b code of a normal symbol. |
| *rd_err_o* | 1 | Output error signal high when the RD exceeds the [−2, +2] interval. |
| *valid_o* | 1 | Output control signal validating the other outputs, except *rd_casc_o*. |

When the data source (typically a module that finds the alignment of the deserialized 8b/10b stream of bits) provides one symbol at a time to the decoder, a single decoder instance is sufficient for decoding the data stream. In this case, the *rd_o* output signal must be connected directly through a wire to the *rd_i* input while the *rd_casc_o* signal remains unused, as depicted in Figure 9, in order to verify the DC balance of the incoming 8b/10b stream. While this could be achieved internally, we chose to supply the initial RD from the exterior to extend the functionality of the decoder. As such, two or more decoders can be cascaded to allow the simultaneous (i.e., in the same clock cycle) decoding of two or more 8b/10b symbols into the same amount of bytes. In Figure 10, two instances of the proposed 8b/10b decoder are cascaded; in this case, the data source provides two symbols per clock cycle, as in [17]. The *rd_casc_o* one-bit-wide output control signal of the first decoder, representing its *rd_o* value before it is registered, must drive the *rd_i* input signal of the secondary/lower 8b/10b decoder. All of the output signals are doubled except for *valid_o*, *rd_o*, and *rd_casc_o*, which are driven solely by the secondary decoder.

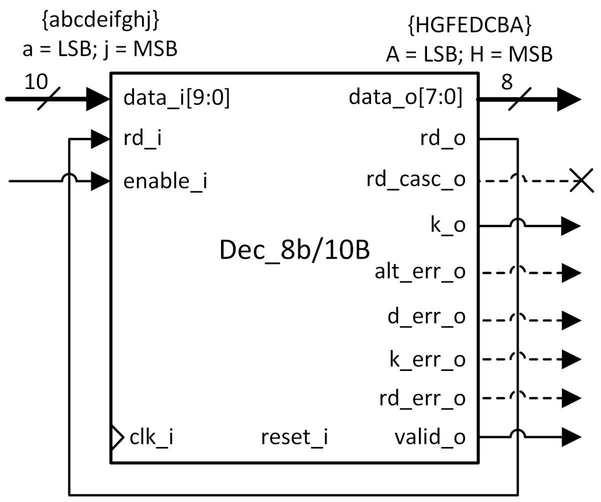


**Figure 9.** The proposed 8b/10b decoder connections when not cascaded.

The top-view architecture of the proposed 8b/10b decoder is presented in Figure 11. The decoder can be divided into five blocks of logic, each one with its own distinct functions: (i) the 5b/6b decoding block; (ii) the 3b/4b decoding block; (iii) the block that determines whether or not the symbol is special (i.e., K logic); (iv) the block that checks the RD (i.e., RD logic); and (v) the block that checks the usage of the correct *D.x.7* 3b/4b code (i.e., the logic computing the *alt_err_out* signal. The first four blocks are depicted in Figure 11 as rectangles, while the last one is represented by the remaining logic gates. All of these blocks are detailed in the following paragraphs.

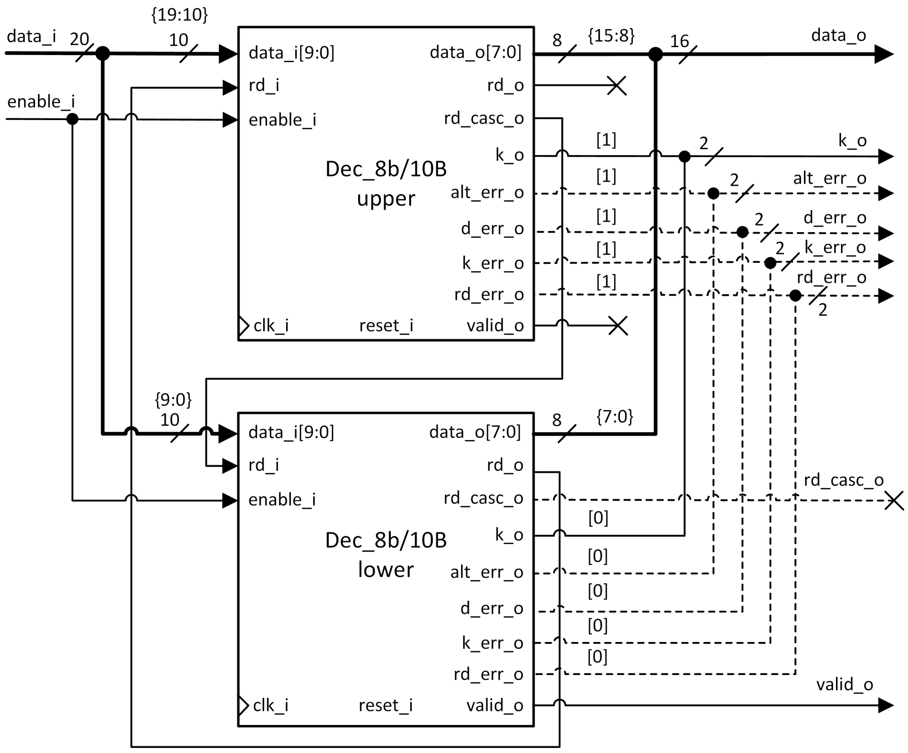


**Figure 10.** Two cascaded instances of the proposed 8b/10b decoder.

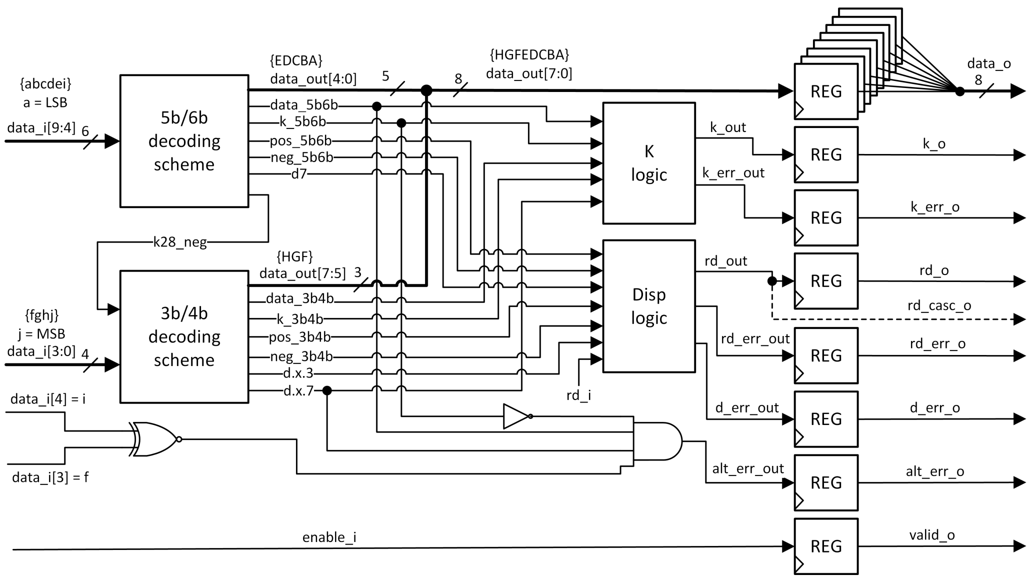


**Figure 11.** Top-view architecture of the proposed 8b/10b decoder with all of the dependencies.

The six LSBs of *data_i* (i.e., bits *abcdei*) are fed to the 5b/6b decoding scheme block, which compares their value with valid 5b/6b codes. If the six-bit input value is a valid 5b/6b code, then the block outputs the corresponding five-bit value representing the LSBs of *data_o* (i.e., bits *EDCBA*). For invalid input values, the output data bits are all 0. The *data_5b6b* signal is high when the input code may correspond to a *D.x.y* symbol, while the *k_5b6b* signal is high when the input code may correspond to a *K.x.y* symbol. The two possible codes of *K.28.y* are the only codes that will produce a high *k_5b6b* and a low *data_5b6b*. Because the complementary 5b/6b codes of *D.23.y*, *D.27.y*, *D.29.y*, and *D.30.y* are also used for *K.23.y*, *K.27.y*, *K.29.y*, and *K.30.y*, respectively (see Table 1 from Section 2), they are marked with high *k_5b6b* and *data_5b6b* signals. The rest of the valid codes produce a high *data_5b6b* and a low *k_5b6b*. Invalid codes result in both the *k_5b6b* and *data_5b6b* signals being low.

The *pos_5b6b* signal is high when the input code has positive or neutral disparity, while *neg_5b6b* is high for an input code with negative or neutral disparity. Similar to the *k_5b6b* and *data_5b6b* pair, the *pos_5b6b* and *neg_5b6b* signals are both low for invalid 5b/6b values. The *d7* signal is high for the two complementary codes of *D.7.y*, which have neutral disparity (see Table 1 from Section 2) but must be selected depending on the input RD to ensure that they do not form five consecutive bits of the same value with the previous 3b/4b code. Even if these two codes have neutral disparity, they are marked as having either a positive or negative disparity through the *pos_5b6b* and *neg_5b6b* signals to enable the signaling of incorrect uses as disparity errors (i.e., the *rd_err_o* output signal) within the RD logic block. The *d7* signal is used to separate this exception when computing the output RD. The *k28_neg* signal is high only for the *K.28.y* code with negative disparity (i.e., the *abcdei* binary value is *110000*).

The four MSBs of *data_i* (i.e., bits *fghj*) are fed to the 3b/4b decoding scheme block alongside the *k28_neg* signal. The block compares their value with valid 3b/4b codes. If the four-bit input value is a valid 3b/4b code, then the block outputs the corresponding three-bit value representing the MSBs of *data_o* (i.e., bits *HGF*). For invalid input values, the output data bits are all 0. The *k28_neg* signal is required in the 3b/4b decoding scheme block because for certain *K.28.y* symbols the *y* value depends not only on the 3b/4b code but on the intermediate RD (i.e., after the 5b/6b encoding; see Table 2 from Section 2). For instance, if the input *fghj* bits are *1001* in binary, they correspond to either *D.x.1*, *K.28.1* or *K.28.6*; if the 5b/6b decoding block indicates a *K.28* symbol with negative disparity, then bits *HGF* are *110* (*K.28.6*), or *001* otherwise (either *K.28.1* or *D.x.1*).

The *data_3b4b* signal is high when the input 3b/4b code may correspond to a *D.x.y* symbol, while the *k_3b4b* signal is high when the input 3b/4b code may correspond to a *K.x.y* symbol. Only the two codes for *D.x.P7* (i.e., *D.x.7* with primary encoding) produce a high *data_3b4b* and a low *k_3b4b*. Invalid 3b/4b codes result in both the *k_3b4b* and *data_3b4b* signals being low. A high *k_3b4b* signal in conjunction with a low *data_3b4b* signal is produced only when the *k28_neg* signal is high and the valid 3b/4b code has one of the following binary values: *0101*, *0110*, *1001*, *1010*. The rest of the valid 3b/4b codes are characterized by both the *data_3b4b* and *k_3b4b* signals being high.

The *pos_3b4b* signal is high when the input code has positive or neutral disparity, while *neg_3b4b* is high for an input code with negative or neutral disparity. Similar to the *k_3b4b* and *data_3b4b* pair, the *pos_3b4b* and *neg_3b4b* signals are both low for invalid 3b/4b values. The *d.x.3* signal is high only for the two complementary codes of *D.x.3* or *K.28.3* symbols, which have neutral disparity but must be selected depending on the intermediate RD during the encoding to ensure that they do not form five consecutive bits of the same value with the previous 5b/6b code. Similar to the *d7* case of the 5b/6b decoding scheme block, the two complementary 3b/4b codes for *D.x.3* or *K.28.3* are marked as having either positive or negative disparity through the *pos_3b4b* and *neg_3b4b* signals, even though they have neutral disparity. This allows for the detection of incorrect usage, which is signaled through the *rd_err_o* output signal. The *d.x.7* signal is high for the two complementary codes of *D.x.P7* (i.e., *D.x.7* with primary encoding), *D.x.A7* (i.e., *D.x.7* with alternative encoding), or *K.x.7* symbols (solely using the alternative encoding).

The Boolean functions implemented in the *K logic* and *Disp logic* blocks and for the *alt_err_out* output error signal are detailed in Appendix B and explained next.

The *K logic* block determines whether the decoded symbol is a control (*k_out* high) or normal (*k_out* low) symbol using the *data* and *k* tags from the two decoding blocks (i.e., the *data_5b6b*, *data_3b4b*, *k_5b6b*, and *k_3b4b* signals). The *k_out* signal is high in two cases: (i) the 5b/6b decoding block indicates a valid *K.28.y* symbol and the 3b/4b decoding block indicates any valid code except *D.x.P7*; and (ii) the 5b/6b decoding block indicates a possible *K.23.7*, *K.27.7*, *K.29.7*, or *K.30.7* symbol and the 3b/4b decoding block indicates a valid *D.x.A7* or *K.x.7* symbol. The output of the D flip-flop sampling of the *k_out* signal represents the *k_o* output signal. In addition, the *K logic* block computes the *k_err_out* signal, which after registration becomes the *k_err_o* output error signal. The *k_err_out* signal is high if the valid 5b/6b code corresponds solely to a normal symbol, while the valid 3b/4b code indicates with certainty a valid special symbol or vice versa.

The *Disp logic* block determines three signals: (i) the RD of the incoming 8b/10b stream of bits after the current symbol is decoded as the *rd_out* signal; (ii) situations when the RD goes out of bounds (i.e., outside the $[-2, +2]$ interval) as the *rd_err_out* signal; and (iii) situations when either the 5b/6b or the 3b/4b code is not valid as the *d_err_out*. The *rd_out* signal directly drives the *rd_casc_o* output signal; it is sampled by a D flip-flop, the output of which represents the *rd_o* signal.

The intermediate RD (i.e., right after the 5b/6b code is decoded), named *next_rd_5b6b*, is an internal signal of the *Disp logic* block, and is useful for determining the values of both the *rd_out* and *rd_err_out* signals. Though not depicted in Figure 11, it is used in the LUT-based decoder implementation presented in Section 2.3.2. It maintains the meaning of the logic levels; *1* means negative RD, while *0* means positive RD. The intermediate RD equals the input RD when either the 5b/6b code is invalid, the 5b/6b code is valid and has neutral disparity (indicated by both the *pos_5b6b* and *neg_5b6b* signals being high), or there is a special situation characterized by a high *d7* signal (also a 5b/6b code with neutral disparity). In all other situations, the intermediate RD is the complement of the *pos_5b6b* signal (i.e., it is equivalent to the *neg_5b6b* signal).

Similarly, the *rd_out* signal equals the intermediate RD in situations when either the 3b/4b code is invalid, the 3b/4b code is valid and has neutral disparity as indicated by both the *pos_3b4b* and *neg_3b4b* signals being high, or there is a special case indicated by a high *d.x.3* signal (also a 3b/4b code with neutral disparity). In all the other situations, the *rd_out* signal equals the complement of *next_rd_5b6b*.

The *rd_err_out* signal is high if either of the following situations is true: (i) the input RD has the same polarity as the non-neutral disparity of the valid 5b/6b code or (ii) the intermediate RD has the same polarity as the non-neutral disparity of the valid 3b/4b code.

Invalid 5b/6b codes are signaled through both the *pos_5b6b* and *neg_5b6b* signals being low, while invalid 3b/4b codes are signaled through both the *pos_3b4b* and *neg_3b4b* signals being low. If either case is true, then the *d_err_out* is high. The *data_5b6b*, *k_5b6b*, *data_3b4b*, and *k_3b4b* signals can be used for the same purpose.

As shown in Figure 11, the signal *alt_err_out* is high for *D.x.7* symbols with bits *i* and *f* having the same value. The *K.x.7* symbols should always use the alternative 3b/4b code instead of the primary one. If the decoder receives a *K.28.7* symbol with the primary 3b/4b code as input, it will produce a high *k_err_o* signal. The other potential *K.x.7* symbols (i.e., $x \in \{23, 27, 29, 30\}$) that use the primary 3b/4b code will be interpreted as *D.x.7* symbols. When the alternative 3b/4b code for *D.x.7* is used in conjunction with 5b/6b codes that can be used in both normal and special 8b/10b symbols, the resulting symbol is considered special. The *alt_err_out* signal is high when a clear *D.x.7* symbol mistakenly uses the primary 3b/4b code instead of the alternative (or vice versa), and consequently forms five consecutive bits with the same value. The alternative 3b/4b code for 7 is used for the *D.17.7*, *D.18.7* and *D.20.7* symbols when the *rd_i* signal is *1* (i.e., negative input RD) and for the *D.11.7*, *D.13.7*, and *D.14.7* symbols when the *rd_i* signal is *0* (i.e., positive input RD).

### 2.3.1. BRAM-Based 8b/10b Decoder

We propose the BRAM-based organization for the 8b/10b decoder targeted at AMD FPGAs depicted in Figure 12. The rationale is similar to the one for the BRAM-based 8b/10b encoder from Section 2.2.1. Except for the *clk_i*, *reset_i*, and *enable_i* signals, the input signals of the 8b/10b decoder shown in Figure 8 and described in Table 6 are concatenated into an eleven-bit-wide bus that represents the input address. The dual-port, dual-clock domain (i.e., asynchronous) BRAM primitive is configured as a dual-port, single-clock domain (i.e., synchronous, common) ROM with a $2^{11}$ address space and a fourteen-bit-wide output data bus. The BRAM acts as a large single LUT, while the fourteen-bit output data represents the concatenated output signals of the 8b/10b decoder shown in Figure 8 and described in Table 6, excepting the *valid_o* and *rd_casc_o* signals. The *valid_o* output signal is formed by delaying the *enable_i* input signal with one clock cycle using a D flip-flop.

Because the primitive is dual-port and read-only, two independent 8b/10b decoders can be implemented with the same ROM at the additional cost of a D flip-flop for the second *valid_o* signal, as shown in Figure 12. Address collision is not an issue for ROMs [18].

The content of the ROM is specified during the design phase as an array of hexadecimal values, in this case $2^{11} = 2048$ four-digit values (i.e., hexadecimal, each representing a nibble except for the most significant one, which represents only the two MSBs of the output bus), resulting in 28 Kb of data. Thus, a single 18 Kb BRAM is not enough, while a 36 Kb BRAM configured as 2K $\times$ 18 is sufficient. There is no resource utilization/area benefit in dropping all four optional output error signals, as in this case the required memory will be 20 Kb, which would still require a 36 Kb BRAM.

In Figure 12, it is considered that the BRAMs have a single clock cycle latency, meaning that (with the exception of the lack of the *rd_casc_o* output signal) the behavior of the resulting 8b/10b decoder is identical to that of the generic one presented in Figure 11 and the one based on LUT primitives presented in Section 2.3.2. The optional embedded registers of the BRAM primitive are not used, nor are the optional configurable logic registers. The optional *rd_casc_o* output signal is a purely combinational logic function of the input signals, as depicted in Figure 11, and as such cannot be implemented using BRAM primitives while maintaining the requirement of a single clock cycle latency.

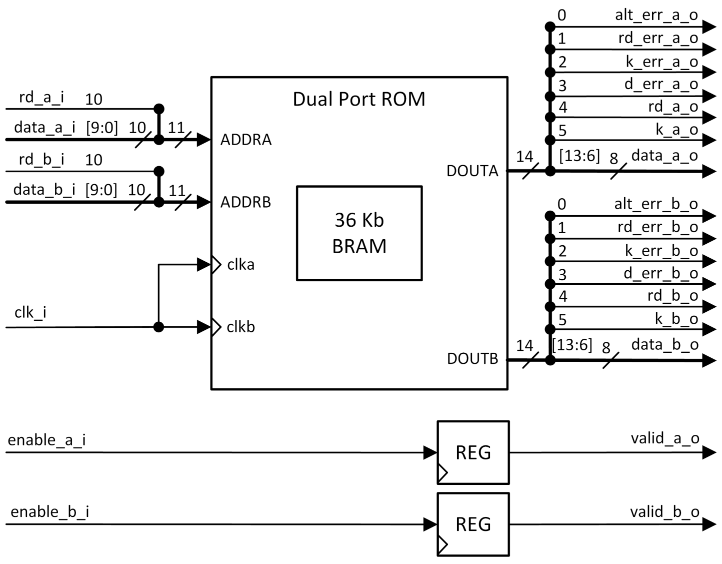


**Figure 12.** The organization of the proposed 8b/10b decoder based on a single 36 Kb AMD BRAM.

### 2.3.2. LUT-Based 8b/10b Decoder

We propose the LUT-based organization for the 8b/10b decoder targeted at AMD FPGAs depicted in Figure 13. The signals can be found in Figure 11, and their functions are described in Section 2.3. The Boolean functions implemented by the LUTs are listed in Table 7 as little-endian truth table values expressed in hexadecimal.

**Table 7.** Configurations of all the decoder LUTs from Figure 13.

| # | Instance | Hex. Config. Value |
|---|---|---|
| 1 | Bit_A | `0x05565F6C209202A0` |
| 2 | Bit_B | `0x0556290256FC02A0` |
| 3 | Bit_C | `0x05215F025692ECA0` |
| 4 | Bit_D | `0x0259596250F2F240` |

**Table 7.** *Cont.*

| # | Instance | Hex. Config. Value |
|---|---|---|
| 5 | Bit_E | 0x044D4C5A4C5ADA20 |
| 6 | K28_neg_disp | 0x0001000000000000 |
| 7 | Data_5b6b | 0x077E7FFE7FFE7EE0 |
| 8 | K_5b6b | 0x0441400240028220 |
| 9 | Neg_5b6b | 0x077E7EE87EE8E800 |
| 10 | Pos_5b6b | 0x0017177E177E7EE0 |
| 11 | Bits_F_G | 0x3317E8CC332BD4CC |
| 12 | D7 | 0x0100000000000080 |
| 13 | Data_3b4b_K_3b4b | 0x799E7FFE3FFC3FFC |
| 14 | Pos_3b4b_Neg_3b4b | 0x077E077E7EE07EE0 |
| 15 | Bit_H_D.x.7 | 0x63A665C641824182 |
| 16 | K_out_K_err_out | 0xCC00440002400240 |
| 17 | D_err_out | 0x111F |
| 18 | Rd_err_out | 0x22F244F42F224F44 |
| 19 | Toggle_5b6b_disp_bit_i_xnor_bit_f | 0x96699669FF0000FF |
| 20 | Alt_err_out | 0x2000 |
| 21 | Rd_out_Next_rd_5b6b_out | 0xF0B20F4DF0B2F0B2 |

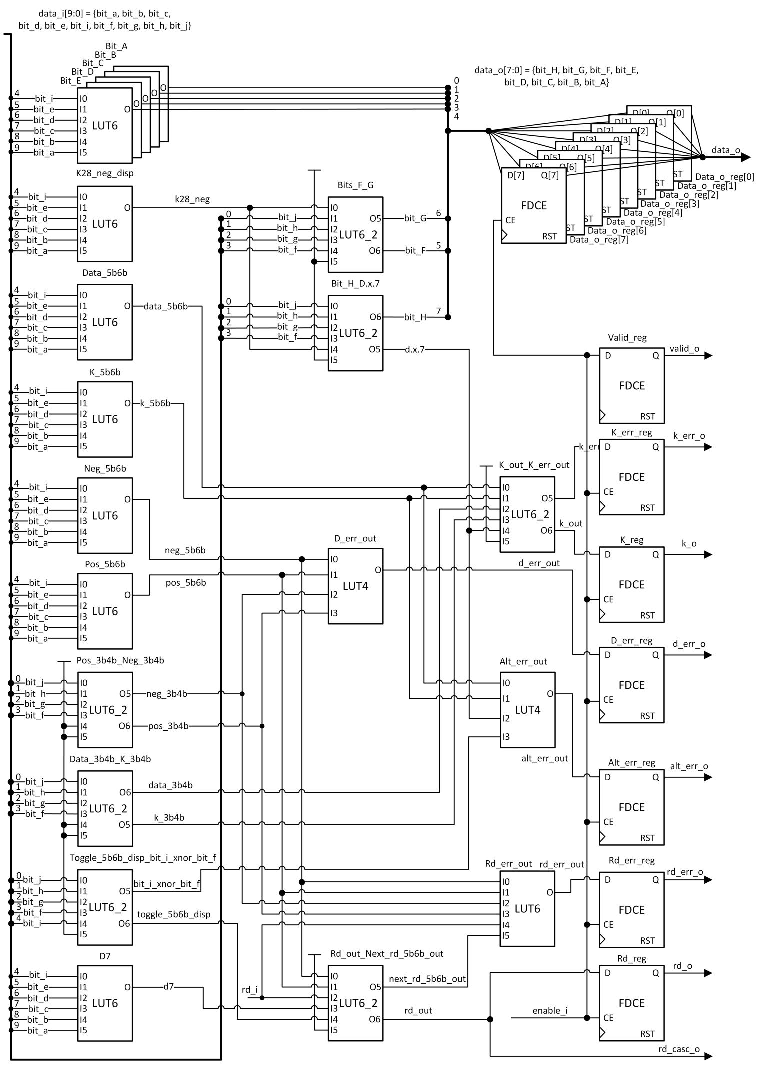


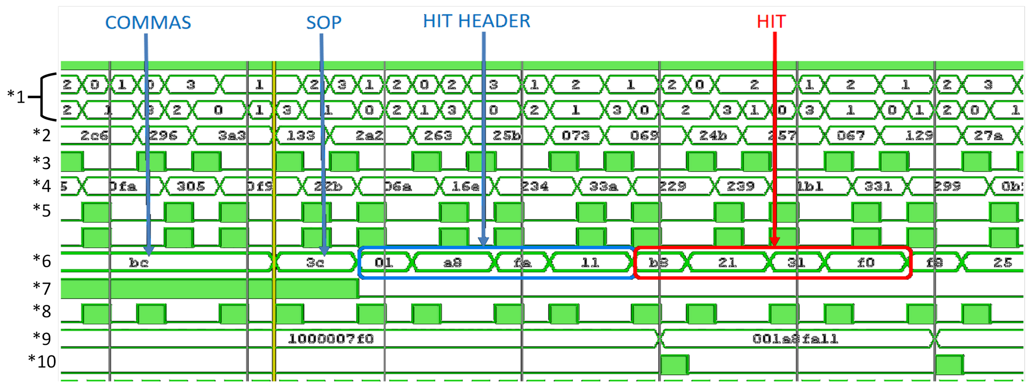
**Figure 13.** The proposed 8b/10b decoder organization based on AMD LUT primitives. Table 7 presents the logic function of each LUT.

## 3. Results

All the proposed structural 8b/10b codecs were described in Verilog HDL as interconnected instances of LUT, register, and BRAM primitives exactly as presented in the previous section. The truth table of each LUT instance was specified through the *INIT* parameter value, as described in [19]. The content of each ROM was specified as a list of hexadecimal values in a memory coefficient file, as described in [20].

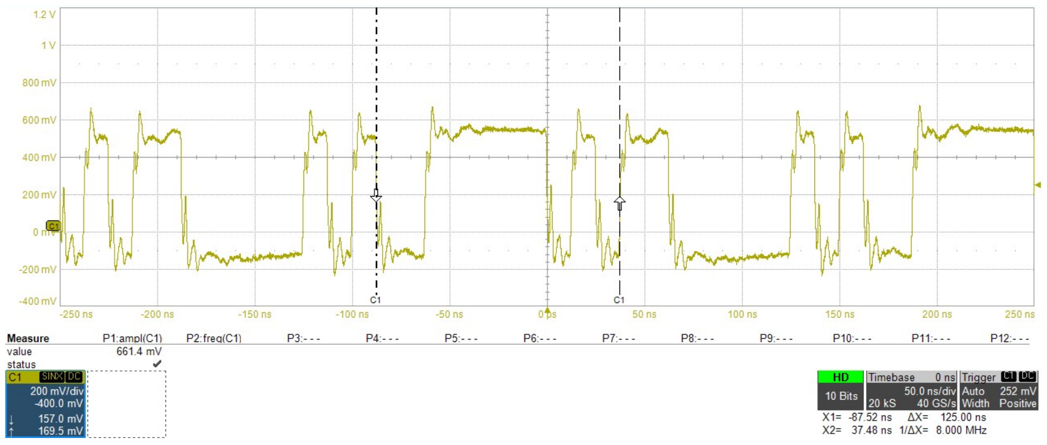
For simulation, synthesis, and implementation on AMD FPGAs, we used the AMD Vivado Design Suite version 2023.1. Our previous work regarding an LUT-based 8b/10b encoder similar to the one presented in Section 2.2.2, presented in [16], was achieved using version 2022.1 of the same tool.

Validated behavioral descriptions of 8b/10b codecs that have been successfully employed in the Application Specific Integrated Circuit (ASIC) from [14,15] and its test setup from [21] were used as references for verification of the proposed designs. From now on, these are referred to as *golden models*. They are described in Verilog HDL, are based on designs from [22], and have similar interfaces to the proposed structural designs. These models were previously exhaustively debugged and verified by referencing the state-of-the-art solutions specific to Intel FPGAs from [23]. Labeled waveforms captured using an FPGA Integrated Logic Analyzer depicting the use of the golden model 8b/10b decoder for processing encoded data arriving at 640 Mbps are presented in Figure 14. Due to the overhead introduced by the encoding, the effective throughput is 512 Mbps (i.e., 80% of 640 Mbps). A waveform captured using a digital oscilloscope depicting serialized back-to-back *K.28.5 comma* symbols produced by a golden model 8b/10b encoder used in the ASIC from [14,15] are presented in Figure 15. The two complementary encodings (i.e., *abcdei fghj = 001111 1010* and *110000 0101*) alternate at 8 MHz, resulting in a bitrate of 80 Mbps. The golden models used in this research were validated in both ASIC and FPGA implementations [21], and are compatible with each other as well as with other validated ASIC and FPGA implementations of 8b/10b codecs.



SOP = Start Of Packet
*1 = two 320 Mbps serial lines sampled by the FPGA logic
*2 = deserializer 10-bit wide output
*3 = signal validating *2
*4 = 8b/10b aligner module 10-bit wide output (decoder's data_i input)
*5 = signal validating *4 (decoder's enable_i input)
*6 = 8b/10b decoder 8-bit wide data output (i.e. data_o)
*7 = 8b/10b special symbol output (i.e. k_o)
*8 = 8b/10b output signal validating *6 and *7 (i.e. valid_o)
*9 = (32+1)-bit wide data word formed by 4 normal decoded symbols concatenated + 1 flag bit
*10 = signal validating *9

**Figure 14.** Waveforms captured using an FPGA Integrated Logic Analyzer, depicting the use of the golden model behavioral 8b/10b decoder for processing encoded data arriving at 640 Mbps. The error signals are not included.



Measure P1:ampl(C1) P2:freq(C1) P3:- - - P4:- - - P5:- - - P6:- - - P7:- - - P8:- - - P9:- - - P10:- - - P11:- - - P12:- - -
value 661.4 mV
status ✓

200 mV/div
-400.0 mV
↓ 157.0 mV
↑ 169.5 mV

10 Bits
Timebase 0 ns
50.0 ns/div
20 kS 40 GS/s
Auto 252 mV
Width Positive
X1= -87.52 ns ΔX= 125.00 ns
X2= 37.48 ns 1/ΔX= 8.000 MHz

**Figure 15.** Waveforms captured using a digital oscilloscope, depicting serialized back-to-back *K.28.5 comma* symbols produced by a golden model behavioral 8b/10b encoder used in the ASIC from [14,15]. The two complementary encodings (i.e., *abcdei fghj = 001111 1010* and *110000 0101*) alternate at 8 MHz (i.e., the bitrate is 80 Mbps).

Two additional distinct behavioral descriptions of an 8b/10b encoder were implemented on FPGA. These are referred to as *Beh. 1* and *Beh. 2* and were added for comparison. The former is a cleaner and easier to understand code of the golden model, e.g., statements in the form `assign var = condition ? 1’b0 : 1’b1;` from the golden model (i.e., conditional operator) are simplified to `assign var= ~condition;`. The *Beh. 2* encoder contains a single `case ... endcase` statement that lists all the combinations of values that the concatenated input signals can take and assigns the appropriate value in each case to the output signals. For the 8b/10b decoder, no additional behavioral descriptions were implemented.

The proposed structural codecs and additional behavioral descriptions were validated in exhaustive simulations in which their output values were compared to the golden model’s responses to the same input stimuli. These exhaustive simulations consisted of the following three tests: (i) the *rd_i* input signal was kept at *0* while all possible values for the other input signals were fed in ascending order to the Device Under Verification (DUV) at half the clock frequency rate (i.e., once every two clock cycles); (ii) the same as the previous test, except that this time the *rd_i* input signal was set to *1*; and (iii) 1,000,000 pseudo-random input data fed back-to-back. Waveforms depicting the beginning of the first test for the 8b/10b structural decoder are depicted in Figure 16. These resulted from post-implementation timing simulations using a 250 MHz ideal clock signal, ideal input stimuli (from the timing perspective), and maximum values for the back-annotated delays. The first non-zero output data, marked by the vertical cursor, were produced 100 ps after the positive edge of the clock signal, representing the setup time of the output data registers. No timing violations were reported for adequate clock frequencies. Because the first input data values are invalid 8b/10b symbols, the *d_err_out* signal is raised. These simulations were additionally used to generate switching data for the power consumption estimations in the switching activity interchange file format (SAIF).

The AMD IP catalog associated with the Vivado 2023.1 tool is a library of AMD IP cores [24]; however, it does not include any standalone 8b/10b encoder or decoder IPs. In [25,26], a parametrizable standalone 8b/10b encoder/decoder with similar interfaces and functionality to our proposed designs was presented. These AMD 8b/10b codecs were targeted at older AMD FPGA devices, being designed before the Vivado tool and the IP Catalog were available; thhus, they are not included in these tools [27]. Two hardware implementations are available for the AMD 8b/10b encoder and decoder, one based on LUTs and one based on BRAM; however, both implementations described in VHDL are behavioral, not structural. In the case of the BRAM-based implementations, two independent codecs can use the same ROM, similar to our designs. While the AMD codecs can be configured to the same functionality as our implementations, the decoders lack any signals or logic corresponding to *k_err_o* and *alt_err_o* regardless of their parameter values. We successfully simulated, synthesized, and implemented these codecs in Vivado 2023.1 and included them in the comparisons.

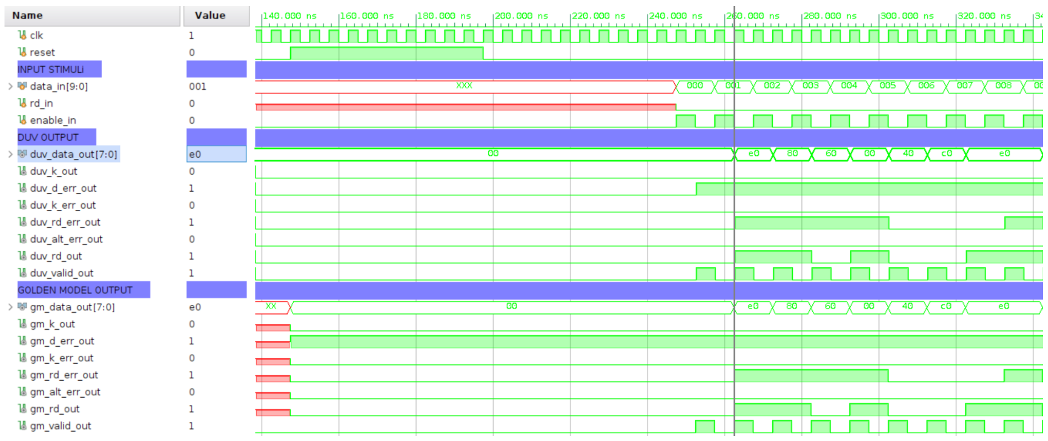


**Figure 16.** Waveforms depicting the beginning of the first test for the 8b/10b structural decoder are depicted in Figure 16.

In [28], we found the only 8b/10b encoder FPGA implementations from the literature that were well-documented in terms of resource utilization, power draw, and maximum operating frequency or longest propagation time. These implementations targeted the AMD Kintex-7 KC705 evaluation board [13] with the XC7K325T-2FFG900C FPGA device, and have a similar interface and functionality to our encoders. For a fair comparison, we targeted the same board and the AMD 8b/10b codecs for all our implementations. Similar to our structural models, one implementation is based on LUTs (further referred to as *Impl. 1*) while the other employs BRAMs (further referred to as *Impl. 2*). We did not find any well-documented FPGA implementations for 8b/10b decoders in the literature.

Our behavioral and structural designs, the golden models, and the AMD 8b/10b codecs from [25,26] were synthesized and then implemented. We used the default tool settings with the incremental optimizations turned off. In all cases, the input signals were driven by registers external to the codecs. For our BRAM-based structural implementations, the output signals were captured by external registers. We constrained our designs to be paced by a 250 MHz clock signal with a 100 ps input jitter. The same clock frequency as in [28] was used for Impl. 1 and Impl. 2. For the power draw comparisons, we used the same default tool parameters as in [28]: no output load (i.e., 0 pF), typical process, nominal power supply voltages, an input toggle rate of 12.5%, and a static probability of 50%, along with the same 250 MHz clock signal and a 25 °C environment temperature. For all the decoders, we estimated the power draw using the SAIF files resulting from the verification simulations. The results were under 1% above the estimations with the default parameters; thus, we did not apply the same analysis for the encoders. For the resource utilization comparisons, we used the *area-optimized high* synthesis strategy which applies additional optimizations to reduce resource utilization.

The resource utilization of the encoders and decoders are detailed in Tables 8 and 9, respectively. Our structural codecs are emphasized. After synthesis and implementation, the resulting netlists for our structural codecs were functionally identical to the initial designs, indicating that they are optimal and pass all design rules checks. Our LUT-based structural codecs have the lowest resource utilization among the designs that do not use any BRAM (the decoder is on par with the golden model). They use no architecture-specific multiplexers to combine LUTs. Our BRAM-based structural encoder has the lowest resource utilization of the designs that contain BRAM. However, our BRAM-based structural decoder uses an entire 36 Kb BRAM and two registers, while the AMD standalone BRAM-based decoder from [26] uses an 18 Kb BRAM, four registers, and two LUTs but does not implement the logic associated with the *k_err_o* and *alt_err_o* output signals. If we remove these signals from our BRAM-based decoder, the resource usage does not change. Both our BRAM-based codec implementations and AMD's contain two independent codec instances.

**Table 8.** Resource utilization for all encoder implementations.

| Implementation | LUTs | REGs | Muxes | BRAMs |
|---|---|---|---|---|
| Impl. 1 [28] | 18 | 21 | 6 | 0 |
| Impl. 2 [28] | 27 | 27 | 0 | 1.5 |
| Golden (default) | 22 | 13 | 0 | 0 |
| Golden (area) | 20 | 13 | 0 | 0 |
| Beh. 1. (default) | 22 | 13 | 7 | 0 |
| Beh. 1. (area) | 21 | 13 | 6 | 0 |
| Beh. 2. (default) | 12 | 2 | 1 | 1 |
| Beh. 2. (area) | 11 | 2 | 1 | 1 |
| LUT-based [25] (default) | 20 | 13 | 0 | 0 |
| LUT-based [25] (area) | 21 | 13 | 0 | 0 |
| BRAM-based [25] (default) | 74 | 26 | 12 | 0 |
| BRAM-based [25] (area) | 0 | 2 | 0 | 0.5 |
| **Prop. LUT (default)** | **17** | **13** | **0** | **0** |
| **Prop. LUT (area)** | **17** | **13** | **0** | **0** |
| **Prop. BRAM (default)** | **0** | **2** | **0** | **0.5** |
| **Prop. BRAM (area)** | **0** | **2** | **0** | **0.5** |

**Table 9.** Resource utilization for all decoder implementations.

| Implementation | LUTs | REGs | Muxes | BRAMs |
|---|---|---|---|---|
| Golden (default) | 21 | 15 | 0 | 0 |
| Golden (area) | 22 | 15 | 0 | 0 |
| LUT-based [26] (default) | 28 | 13 | 1 | 0 |
| LUT-based [26] (area) | 27 | 13 | 1 | 0 |
| BRAM-based [26] (default) | 4 | 4 | 0 | 0.5 |
| BRAM-based [26] (area) | 2 | 4 | 0 | 0.5 |
| **Prop. LUT (default)** | **21** | **15** | **0** | **0** |
| **Prop. LUT (area)** | **21** | **15** | **0** | **0** |
| **Prop. BRAM (default)** | **0** | **2** | **0** | **1** |
| **Prop. BRAM (area)** | **0** | **2** | **0** | **1** |

The *area-optimized high* synthesis strategy has a generally positive influence, with two exceptions: (i) for the AMD standalone LUT-based encoder from [25] and (ii) for the golden model decoder. It has no benefit for our structural codecs. Another interesting effect of this strategy is for the AMD standalone BRAM-based encoder from [25]; by default, the behavioral ROM description translates into 74 LUTs, 12 multiplexers and 26 registers, however, when the resource utilization optimizations are applied, the same description translates into a design similar to our BRAM-based structural encoder. This does not happen for our ROM-like encoder behavioral description (i.e., Beh. 2), which results in high usage of both LUT and BRAM regardless of the synthesis strategy.

In contrast with the results presented in [16], the optimized *Beh. 1* encoder implementation obtains slightly worse results relative to the golden model. The only changes are the addition of the *rd_casc_o* signal, which comes with no additional logic, and the tool version (2023.1 versus 2022.1). Thus, we consider this difference to be caused by the tool.

The maximum operating frequencies can be deduced from the longest paths, which are presented in Tables 10 and 11 for the encoders and the decoders, respectively. In almost all cases, the net delays corresponding to the FPGA routing resources are higher than the logic delays (i.e., propagation through LUTs, multiplexers or BRAMs). Beh. 1 and Beh. 2 obtain the lowest two propagation delays from all the encoder designs. The resource utilization of almost all behavioral encoder descriptions increased (the exception being Beh. 2), while the structural designs maintained their area; the golden model encoder uses 31 LUTs and 13 registers, Beh. 1 uses 23 LUTs, 13 registers, and 7 multiplexers, the LUT-based AMD encoder from [25] uses 27 LUTs and 13 registers, and its BRAM-based variant uses 78 LUTs, 26 registers, and 12 multiplexers.

**Table 10.** Longest path for all encoder implementations.

| Implementation | Logic [ns] | Net [ns] | Total [ns] |
|---|---|---|---|
| Impl. 1 [28] | 0.309 | 1.059 | 1.368 |
| Impl. 2 [28] | 1.929 | 1.346 | 3.275 |
| Golden | 0.395 | 1.124 | 1.519 |
| Beh. 1 | 0.498 | 0.586 | 1.084 |
| Beh. 2 | 0.388 | 0.866 | 1.254 |
| LUT-based [25] | 0.352 | 1.004 | 1.356 |
| BRAM-based [25] | 0.352 | 1.198 | 1.55 |
| **Proposed LUT** | **0.453** | **0.904** | **1.357** |
| **Proposed BRAM** | **1.8** | **0.478** | **2.278** |

**Table 11.** Longest path delay for all decoder implementations.

| Implementation | Logic [ns] | Net [ns] | Total [ns] |
|---|---|---|---|
| Golden | 0.395 | 1.031 | 1.426 |
| LUT-based [26] | 0.417 | 0.939 | 1.356 |
| BRAM-based [26] | 0.474 | 0.943 | 1.417 |
| **Proposed LUT** | **0.457** | **1.031** | **1.488** |
| **Proposed BRAM** | **1.8** | **0.494** | **2.294** |

The lowest propagation delays among the decoder designs were achieved by the LUT-based AMD implementation. The resource utilization of all behavioral decoder descriptions increased, while the structural designs maintained their area; the golden model decoder uses 26 LUTs and 15 registers, the LUT-based AMD decoder from [26] uses 32 LUTs, 13 registers, and 1 multiplexer, and its BRAM-based variant switches from using a 18 Kb BRAM to using 78 LUTs, 89 registers, and 12 multiplexers.

The number of chained LUTs of a Boolean function, with or without multiplexers, is not as important for the maximum frequency as the placement and routing congestion, e.g., the proposed structural LUT-based decoder has a chain of at most three LUTs. This is similar to the golden model, but achieves a slightly larger propagation delay; nevertheless, our proposed structural codecs maintain their low resource usage along with comparable frequency performance to the other designs.

The power draw estimates are detailed in Tables 12 and 13 for the encoders and decoders, respectively, and use the previously presented default tool parameters. The static power draw is the same for all designs, while the structural codecs are on par with the others. The designs which use BRAM have an increased dynamic power draw. The dynamic power draw of the encoders from [28] is ten times higher than the rest of the encoders, which might be due to the different tool versions (unspecified in [28]).

The power draw estimates resulting from the verification simulations are detailed in Table 14 for all the decoders considering the signal toggling activity. With the exception of the BRAM-based implementation of the AMD decoder, it can be seen that the dynamic power consumption of all models increases with 0.001 W compared to that obtained with the default tool parameters. This represents less than 1% of their total power draw. The static power draws are identical. While the dynamic power draw of the BRAM-based implementation of the AMD decoder increases with 0.003 W, its total power consumption is still less than that corresponding to our structural BRAM-based model.

**Table 12.** Power consumption for all encoder implementations with the default tool parameters.

| Implementation | Dynamic [W] | Static [W] | Total [W] |
|---|---|---|---|
| Impl. 1 [28] | 0.019 | 0.158 | 0.177 |
| Impl. 2 [28] | 0.027 | 0.158 | 0.186 |
| Golden | 0.002 | 0.156 | 0.158 |
| Beh. 1 | 0.001 | 0.156 | 0.157 |
| Beh. 2 | 0.003 | 0.156 | 0.159 |
| LUT-based [25] | 0.002 | 0.156 | 0.158 |
| BRAM-based [25] | 0.004 | 0.156 | 0.16 |
| **Proposed LUT** | **0.002** | **0.156** | **0.158** |
| **Proposed BRAM** | **0.007** | **0.156** | **0.163** |

**Table 13.** Power consumption for all decoder implementations with the default tool parameters.

| Implementation | Dynamic [W] | Static [W] | Total [W] |
|---|---|---|---|
| Golden | 0.002 | 0.156 | 0.158 |
| LUT-based [26] | 0.002 | 0.156 | 0.158 |
| BRAM-based [26] | 0.004 | 0.156 | 0.16 |
| **Proposed LUT** | **0.002** | **0.156** | **0.158** |
| **Proposed BRAM** | **0.013** | **0.156** | **0.169** |

**Table 14.** Power consumption for all decoder implementations considering the switching activity resulting from the verification simulations.

| Implementation | Dynamic [W] | Static [W] | Total [W] |
|---|---|---|---|
| Golden | 0.003 | 0.156 | 0.159 |
| LUT-based [26] | 0.003 | 0.156 | 0.159 |
| BRAM-based [26] | 0.007 | 0.156 | 0.163 |
| **Proposed LUT** | **0.003** | **0.156** | **0.159** |
| **Proposed BRAM** | **0.014** | **0.156** | **0.17** |

## 4. Discussion

Optimality has different meanings in different contexts; it can refer to a short development stage that reaches a functionally validated model quickly, to a functionally validated model characterized by lower power consumption, lower footprint (i.e., area or resource utilization), and higher performance (e.g., throughput, operating frequencies), or to a combination of these.

One of the goals of this research was to test the hypothesis that a structural design can be better in one or more ways compared to one resulting from a behavioral description. We chose to implement IBM 8b/10b codecs, which are relatively small circuits, though not straightforward. This encoding is used in a plethora of communication standards, and we previously employed it in custom communication protocols [14]. We hypothesized that more LUT6_2 instances can be inferred than in the synthesis and implementation tools, allowing us to reduce resource usage. A more compact design translates into a lower power draw. We further hypothesized that a logic path containing fewer LUTs would translate into a lower propagation time, and in turn a higher maximum operating frequency. Following Virtex-5, the different AMD FPGA architectures have varied mostly in terms of how the LUTs are grouped in CLBs, the routing resources, the implemented hard-blocks, and the used semiconductor technology. We considered that our structural designs should not employ architecture-specific resources (e.g., multiplexers for combining LUTs, dedicated carry logic); thus we set four objectives: (i) to minimize the number of LUTs, without (ii) compromising functional features, while (iii) using as few chained LUTs between any two sequential elements as possible and (iv) not employing specific resources that change from one FPGA generation to another.

A simple `case ... endcase` statement in Verilog (or an equivalent statement in other HDLs) that lists all the correspondence between the input and output values for combinational logic most likely will not have the best results in terms of resource utilization, timing, and power consumption; however, it can be developed quickly and has good portability (ASIC and FPGA regardless of its architecture). Depending on its complexity, it might additionally imply increased processing time and memory requirements on the part of synthesis and implementation tools. This led to the idea of using BRAMs as larger LUTs. While less portable, such solutions are likely to result in better implementations. A behavioral HDL description optimized by the designer is characterized by good portability (ASIC and FPGA regardless of its architecture), most likely better implementation results than the `case ... endcase` model, faster processing time, and lower memory requirements in the synthesis and implementation tools; however, it implies the need for in-depth knowledge and an increased development duration. Although synthesis and implementation tools are getting better at optimizing circuits, we consider that a good HDL model should not need to rely on the capabilities of these tools to achieve optimal results. While a model directly designed as a structure implies additional in-depth knowledge of the targeted technology (in the case of an ASIC implementation) or architecture (in the case of an FPGA implementation), and is the least portable (being very specific), it can potentially obtain even better results than an optimized behavioral one.

All of our codecs, both behavioral and structural, implement the entire 8b/10b IBM encoding scheme without compromising functional features and can be used in a complete communications channel employing this encoding regardless of the higher-level protocol used. They have been exhaustively validated within simulations and on actual hardware (i.e., FPGA and in some cases even ASIC). Their error signals proved to be crucial in debugging interfacing issues both in the ASIC test setup presented in [21] and while integrating the ASIC into its operating environment.

Several AMD IPs employ the 8b/10b encoding, e.g., the 1G/2.5G Ethernet Physical Coding Sublayer/Physical Medium Attachment (PCS/PMA), and the Serial Gigabit Media Independent Interface (SGMII) core [29] implements the 8b/10b encoding as part of the PCS functionality. For this, it uses either device-specific transceiver logic or includes the AMD codecs from [25,26], which are mentioned and used in our comparison in Section 3.

The Aurora 8B/10B protocol [8] is an open standard for point-to-point user data transfer that uses the IBM 8b/10b encoding scheme and defines the physical layer interface, initialization and error handling, data marking and mapping, and link layer flow control mechanism. The AMD Aurora 8B/10B IP core [30] implements the link layer protocol for the AMD 7 Series, UltraScale, and UltraScale+ FPGA families. According to [31], when targeting the AMD Kintex-7 KC705 evaluation board [13] an Aurora 8B10B transmitter with a single simplex line and minimal functionality uses 343 flip-flops and 184 LUTs. While this IP core implements more functional features than just 8b/10b encoding, it might benefit from the proposed structural 8b/10b codecs. We compared Impl. 1 and Impl. 2 from [28] to the Aurora 8B/10B core in terms of power consumption, propagation delay, and resource utilization.

## 5. Conclusions

In this paper, we detail four structural primitive-based 8b/10b codecs (two encoders and two decoders) targeted at modern AMD FPGAs. These models can be used on all architectures since Virtex-5. The proposed designs are compared with state-of-the-art codecs and behavioral validated descriptions. It is proven that they achieve the lowest resource utilization while reaching comparable maximum frequencies and power consumption. They do not rely on special synthesis and implementation tool strategies or architecture-specific transciever, carry, and multiplexing logic to achieve this performance. The proposed implementations are suitable for a complete communication channel employing the IBM 8b/10b encoding.

**Author Contributions:** Conceptualization, S.P., M.I., and R.-M.C.; methodology, S.P. and R.-M.C.; software, S.P.; validation, S.P.; formal analysis, S.P. and R.-M.C.; investigation, S.P.; resources, M.I. and R.-M.C.; data curation, S.P.; writing—original draft preparation, S.P., M.I., and R.-M.C.; writing—review and editing, S.P., M.I., and R.-M.C.; visualization, S.P., M.I., and R.-M.C.; supervision, M.I. and R.-M.C.; project administration, M.I. and R.-M.C.; funding acquisition, M.I. and R.-M.C. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was funded by the Romanian Ministry of Research, Innovation, and Digitization, RO-CERN collaboration, project "ATLAS experiment at LHC", contract no. 10/2022.

**Data Availability Statement:** The data presented in this study are available on request from the corresponding author.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| ACAP | Adaptive Compute Acceleration Platform |
| ASIC | Application-Specific Integrated Circuit |
| ATA | Advanced Technology Attachment |
| BRAM | Block of static RAM |
| CE | Clock Enable |
| CLB | Configurable Logic Block |
| DC | Direct Current |
| DUV | Device Under Verification |
| FDCE | Flip-flop D with Clock Enable |
| FPGA | Field-Programmable Gate Array |
| HDL | Hardware Description Language |
| IBM | International Business Machines Corporation |
| IO | Input–Output |
| LSB | Least Significant Bit |
| LUT | Look-Up Table |
| MSB | Most Significant Bit |
| PCS | Physical Coding Sublayer |
| PMA | Physical Medium Attachment |
| RAM | Random Access Memory |
| ROM | Read-Only Memory |
| RD | Running Disparity |
| RTL | Register Transfer Level |
| RX | Receiver |
| SAIF | Switching Activity Interchange Format |
| SGMII | Serial Gigabit Media-Independent Interface |
| SOP | Start Of Packet |
| TX | Transmitter |
| USB | Universal Serial Bus |

## Appendix A. Useful Encoder Boolean Functions

The following Boolean functions correlate with the descriptions of the proposed 8b/10b encoder from Section 2.2 (*pot_k*, the *potential K symbol*, is an auxiliary product asserted when the value present on *data_i* corresponds to one of the twelve possible special symbols).

$$
\begin{aligned}
\mathit{bit\_a} &= \mathit{bit\_a\_neg\_rd} \oplus (\mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}}) \\
\mathit{bit\_b} &= \mathit{bit\_b\_neg\_rd} \oplus (\mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}}) \\
\mathit{bit\_c} &= \mathit{bit\_c\_neg\_rd} \oplus (\mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}}) \\
\mathit{bit\_d} &= \mathit{bit\_d\_neg\_rd} \oplus (\mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}}) \\
\mathit{bit\_e} &= \mathit{bit\_e\_neg\_rd} \oplus (\mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}}) \\
\mathit{bit\_i} &= \mathit{bit\_a\_neg\_rd} \oplus (\mathit{rd\_i} \cdot \mathit{k28} + \mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}} \cdot \overline{\mathit{k28}}) \\
\mathit{error} &= \mathit{error\_o\_w} = \mathit{k\_i} \cdot \overline{\mathit{pot\_k}} \\
\mathit{rd\_interm} &= \mathit{next\_rd\_5b\_6b} = \overline{\mathit{diff\_enc\_5b\_6b}} \cdot \mathit{rd\_i} + \mathit{d7} \cdot \mathit{rd\_i} + \overline{\mathit{d7}} \cdot \mathit{diff\_enc\_5b\_6b} \cdot \overline{\mathit{rd\_i}} \\
\mathit{rd\_final} &= \mathit{rd\_o\_w} = \mathit{next\_rd\_5b\_6b} \oplus (\overline{\mathit{bit\_G}} \cdot \overline{\mathit{bit\_H}} + \mathit{bit\_F} \cdot \mathit{bit\_G} \cdot \mathit{bit\_H}) \\
\mathit{pot\_k} &= (\mathit{data\_i} == \mathit{K.28.0}) + (\mathit{data\_i} == \mathit{K.28.1}) + (\mathit{data\_i} == \mathit{K.28.2}) + \\
&\quad (\mathit{data\_i} == \mathit{K.28.3}) + (\mathit{data\_i} == \mathit{K.28.4}) + (\mathit{data\_i} == \mathit{K.28.5}) + \\
&\quad (\mathit{data\_i} == \mathit{K.28.6}) + (\mathit{data\_i} == \mathit{K.28.7}) + (\mathit{data\_i} == \mathit{K.23.7}) + \\
&\quad (\mathit{data\_i} == \mathit{K.27.7}) + (\mathit{data\_i} == \mathit{K.29.7}) + (\mathit{data\_i} == \mathit{K.30.7}) \\
&\quad = \mathit{bit\_E} \cdot \mathit{bit\_D} \cdot \mathit{bit\_C} \cdot \overline{\mathit{bit\_B}} \cdot \overline{\mathit{bit\_A}} + \mathit{bit\_H} \cdot \mathit{bit\_G} \cdot \mathit{bit\_F} \cdot \mathit{bit\_E} \cdot \\
&\quad (\overline{\mathit{bit\_D}} \cdot \mathit{bit\_C} \cdot \mathit{bit\_B} \cdot \mathit{bit\_A} + \mathit{bit\_D} \cdot \overline{\mathit{bit\_C}} \cdot \mathit{bit\_B} \cdot \mathit{bit\_A} + \\
&\quad \mathit{bit\_D} \cdot \mathit{bit\_C} \cdot \overline{\mathit{bit\_B}} \cdot \mathit{bit\_A} + \mathit{bit\_D} \cdot \mathit{bit\_C} \cdot \mathit{bit\_B} \cdot \overline{\mathit{bit\_A}}) \\
\mathit{alt\_enc} &= \mathit{rd\_i} \cdot ((\mathit{data\_i[4{:}0]} == \mathit{D.17}) + (\mathit{data\_i[4{:}0]} == \mathit{D.18}) + (\mathit{data\_i[4{:}0]} == \mathit{D.20})) + \\
&\quad \overline{\mathit{rd\_i}} \cdot ((\mathit{data\_i[4{:}0]} == \mathit{D.11}) + (\mathit{data\_i[4{:}0]} == \mathit{D.13}) + (\mathit{data\_i[4{:}0]} == \mathit{D.14})) \\
&\quad = \mathit{rd\_i} \cdot \mathit{bit\_E} \cdot \overline{\mathit{bit\_D}} \cdot (\overline{\mathit{bit\_C}} \cdot \overline{\mathit{bit\_B}} \cdot \mathit{bit\_A} + \overline{\mathit{bit\_C}} \cdot \mathit{bit\_B} \cdot \overline{\mathit{bit\_A}} + \\
&\quad \mathit{bit\_C} \cdot \overline{\mathit{bit\_B}} \cdot \overline{\mathit{bit\_A}}) + \overline{\mathit{rd\_i}} \cdot \overline{\mathit{bit\_E}} \cdot \mathit{bit\_D} \cdot (\overline{\mathit{bit\_C}} \cdot \mathit{bit\_B} \cdot \mathit{bit\_A} + \\
&\quad \mathit{bit\_C} \cdot \overline{\mathit{bit\_B}} \cdot \mathit{bit\_A} + \mathit{bit\_C} \cdot \mathit{bit\_B} \cdot \overline{\mathit{bit\_A}})
\end{aligned}
$$

## Appendix B. Useful Decoder Boolean Functions

The following Boolean functions correlate with the descriptions of the proposed 8b/10b decoder from Section 2.3.

$$
\begin{aligned}
\mathit{k\_out} &= \mathit{k\_5b6b} \cdot \overline{\mathit{data\_5b6b}} \cdot \mathit{k\_3b4b} + \mathit{k\_5b6b} \cdot \mathit{data\_5b6b} \cdot \mathit{k\_3b4b} \cdot \mathit{d.x.7} \\
\mathit{k\_err\_out} &= \overline{\mathit{k\_5b6b}} \cdot \mathit{data\_5b6b} \cdot \mathit{k\_3b4b} \cdot \overline{\mathit{data\_3b4b}} + \\
&\qquad \mathit{k\_5b6b} \cdot \overline{\mathit{data\_5b6b}} \cdot \overline{\mathit{k\_3b4b}} \cdot \mathit{data\_3b4b} \\
\mathit{d\_err\_out} &= \overline{\mathit{pos\_5b6b} + \mathit{neg\_5b6b}} + \overline{\mathit{pos\_3b4b} + \mathit{neg\_3b4b}} \\
\mathit{alt\_err\_out} &= \mathit{data\_5b6b} \cdot \overline{\mathit{k\_5b6b}} \cdot \mathit{d.x.7} \cdot \overline{\mathit{data\_i[4]} \oplus \mathit{data\_i[3]}} \\
\mathit{neutral\_5b6b} &= \overline{\mathit{pos\_5b6b}} \cdot \overline{\mathit{neg\_5b6b}} + \mathit{pos\_5b6b} \cdot \mathit{neg\_5b6b} + \mathit{d7} \\
&= \overline{\mathit{pos\_5b6b} \oplus \mathit{neg\_5b6b}} + \mathit{d7} \\
\mathit{neutral\_3b4b} &= \overline{\mathit{pos\_3b4b}} \cdot \overline{\mathit{neg\_3b4b}} + \mathit{pos\_3b4b} \cdot \mathit{neg\_3b4b} + \mathit{d.x.3} \\
&= \overline{\mathit{pos\_3b4b} \oplus \mathit{neg\_3b4b}} + \mathit{d.x.3} \\
\mathit{next\_rd\_5b6b} &= \mathit{neutral\_5b6b} \cdot \mathit{rd\_i} + \overline{\mathit{neutral\_5b6b}} \cdot \overline{\mathit{pos\_5b6b}} \\
&= \mathit{neutral\_5b6b} \cdot \mathit{rd\_i} + \overline{\mathit{neutral\_5b6b}} \cdot \mathit{neg\_5b6b} \\
\mathit{rd\_out} &= \mathit{neutral\_3b4b} \cdot \mathit{next\_rd\_5b6b} + \overline{\mathit{neutral\_3b4b}} \cdot \overline{\mathit{next\_rd\_5b6b}} \\
&= \overline{\mathit{neutral\_3b4b} \oplus \mathit{next\_rd\_5b6b}} \\
\mathit{rd\_err\_out} &= \mathit{pos\_5b6b} \cdot \overline{\mathit{neg\_5b6b}} \cdot \overline{\mathit{rd\_i}} + \overline{\mathit{pos\_5b6b}} \cdot \mathit{neg\_5b6b} \cdot \mathit{rd\_i} + \\
&\qquad \mathit{pos\_3b4b} \cdot \overline{\mathit{neg\_3b4b}} \cdot \overline{\mathit{next\_rd\_5b6b}} + \overline{\mathit{pos\_3b4b}} \cdot \mathit{neg\_3b4b} \cdot \mathit{next\_rd\_5b6b}
\end{aligned}
$$

## References

1. Xilinx. 7 Series FPGAs Configurable Logic Block User Guide UG474 (V1.8). 2016. Available online: https://docs.xilinx.com/v/u/en-US/ug474_7Series_CLB (accessed on 12 July 2023).
2. Gaide, B.; Gaitonde, D.; Ravishankar, C.; Bauer, T. Xilinx Adaptive Compute Acceleration Platform: VersalTM Architecture. In *PGA'19 Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 84–93. [CrossRef]
3. Xilinx. Versal Adaptive SoC Design Guide, UG1273 (v2023.2). 2023. Available online: https://docs.xilinx.com/r/en-US/ug1273-versal-acap-design (accessed on 12 December 2023).
4. Siast, J.; Łuczak, A.; Domański, M. RingNet: A Memory-Oriented Network-On-Chip Designed for FPGA. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2019**, *27*, 1284–1297. [CrossRef]
5. Widmer, A.X.; Franaszek, P.A. A DC-Balanced, Partitioned-Block, 8B/10B Transmission Code. *IBM J. Res. Dev.* **1983**, *27*, 440–451. [CrossRef]
6. Schouhamer Immink, K. Construction of binary DC-constrained codes. *Philips J. Res.* **1985**, *40*, 22–39.
7. Fukuda, S.; Kojima, Y.; Shimpuku, Y.; Odaka, K. 8/10 modulation codes for digital magnetic recording. *IEEE Trans. Magn.* **1986**, *22*, 1194–1196. [CrossRef]
8. Xilinx. Aurora 8B/10B Protocol Specification SP002 (v2.3). 2014. Available online: https://docs.xilinx.com/v/u/en-US/aurora_8b10b_protocol_spec_sp002 (accessed on 12 July 2022).
9. *IEEE Std 802.3-2018 (Revision of IEEE Std 802.3-2015)*; IEEE Standard for Ethernet. IEEE: Piscataway, NJ, USA, 2018; pp. 1–5600. [CrossRef]
10. Guha, S.; Wang, W.; Ibraheem, S.; Balakrishnan, M.; Szefer, J. Design and implementation of open-source SATA III core for Stratix V FPGAs. In Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT), Xi'an, China, 7–9 December 2016; pp. 237–240.
11. Universal Serial Bus 3.2 Specification. 2022. Available online: https://www.usb.org/document-library/usb-32-revision-11-june-2022 (accessed on 12 July 2023).
12. Ryu, S.; on behalf of the ATLAS TDAQ Collaboration. FELIX: The new detector readout system for the ATLAS experiment. *J. Phys. Conf. Ser.* **2017**, *898*, 032057. [CrossRef]
13. Xilinx. Kintex-7 FPGA KC705 Evaluation Kit Getting Started Guide UG883 (V6.0). 2014. Available online: https://docs.xilinx.com/v/u/en-US/ug883_K7_KC705_Eval_Kit (accessed on 12 July 2023).
14. Popa, S. *The Read-Out Controller ASIC for the ATLAS Experiment at LHC*; Springer: Cham, Switzerland, 2022.
15. Coliban, R.M.; Popa, S.; Tulbure, T.; Nicula, D.; Ivanovici, M.; Martoiu, S.; Levinson, L.; Vermeulen, J. The Read Out Controller for the ATLAS New Small Wheel. *J. Instrum.* **2016**, *11*, C02069. [CrossRef]
16. Popa, S.; Coliban, R.M.; Ivanovici, M. An Optimal Implementation of an 8b/10b Encoder for Xilinx FPGAs. In Proceedings of the 2022 International Symposium on Electronics and Telecommunications (ISETC), Timișoara, Romania, 10–11 November 2022; pp. 1–4. [CrossRef]
17. Audzevich, Y.; Watts, P.M.; West, A.; Mujumdar, A.; Moore, S.W.; Moore, A.W. Power Optimized Transceivers for Future Switched Networks. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2014**, *22*, 2081–2092. [CrossRef]
18. Xilinx. 7 Series FPGAs Memory Resources User Guide UG473 (V1.14). 2019. Available online: https://docs.xilinx.com/v/u/en-US/ug473_7Series_Memory_Resources (accessed on 12 July 2023).
19. Xilinx. Vivado Design Suite 7 Series FPGA and Zynq 7000 SoC Libraries Guide UG953 (V2022.1). 2022. Available online: https://docs.xilinx.com/r/en-US/ug953-vivado-7series-libraries (accessed on 12 July 2023).
20. Xilinx. Block Memory Generator v8.4 LogiCORE IP Product Guide PG058. 2021. Available online: https://docs.xilinx.com/v/u/en-US/pg058-blk-mem-gen (accessed on 12 July 2023).
21. Popa, S.; Mărtoiu, S.; Ivanovici, M. The quality-control test of the digital logic for the ATLAS new small wheel read-out controller ASIC. *J. Instrum.* **2020**, *15*, P04023. [CrossRef]
22. Lieu, J. 8b10b Encoder Decoder written in Verilog. Available online: https://github.com/jefflieu/fpga.git (accessed on 23 May 2023).
23. Altera. 8b10b Encoder/Decoder MegaCore Function (ED8B10B) ver. 1.02. 2001. Available online: https://www.intel.com/programmable/technical-pdfs/654541.pdf (accessed on 12 July 2023).
24. Xilinx. UltraFast Design Methodology Guide for FPGAs and SoCs UG949 (V2023.1). 2023. Available online: https://docs.xilinx.com/r/en-US/ug949-vivado-design-methodology (accessed on 28 June 2023).
25. Paula Vo, X. Parameterizable 8b/10b Encoder XAPP1122 (v1.1). 2008. Available online: https://docs.xilinx.com/v/u/en-US/xapp1122 (accessed on 28 June 2023).
26. Paula Vo, X. Parameterizable 8b/10b Decoder XAPP1112 (v1.1). 2008. Available online: https://docs.xilinx.com/v/u/en-US/xapp1112 (accessed on 28 June 2023).
27. Parameterizable 8b/10b Encoder/Decoder ( XAPP1122 (v1.1), XAPP1112 (v1.1) ) in Vivado/Gen 7 Chips. Available online: https://support.xilinx.com/s/question/0D52E00006iHm6OSAS/parameterizable-8b10b-encoderdecoder-xapp1122-v11-xapp1112-v11-in-vivadogen-7-chips?language=en_US (accessed on 28 June 2023).
28. Quesada-Martínez, A.; et al. Evaluation of 8b/10b FPGA Encoder Implementations for SerDes Links. In Proceedings of the 2020 IEEE 11th Latin American Symposium on Circuits & Systems (LASCAS), San Jose, Costa Rica, 25–28 February 2020; pp. 1–4. [CrossRef]

29. Xilinx. 1G/2.5G Ethernet PCS/PMA or SGMII v16.2 LogiCORE IP Product Guide. 2023. Available online: https://docs.xilinx.com/r/en-US/pg047-gig-eth-pcs-pma (accessed on 28 June 2023).
30. Xilinx. Aurora 8B/10B v11.1 LogiCORE IP Product Guide. 2022. Available online: https://docs.xilinx.com/r/en-US/pg046-aurora-8b10b/Aurora-8B/10B-v11.1-LogiCORE-IP-Product-Guide (accessed on 12 July 2022).
31. Xilinx. Available online: Resource Utilization for Aurora 8B10B v11.1. 2022. Available online: https://www.xilinx.com/htmldocs/ip_docs/pru_files/aurora-8b10b.html (accessed on 12 July 2023).