2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip

Improvement of Line Coding Overhead Targeting Both Run-Length and DC-Balance

Sarat Yoowattana
Asian Institute of Technology
sarat.yoowattana@gmail.com

Tomohiro Yoneda
National Institute of Informatics/SOKENDAI
yoneda@nii.ac.jp

***Abstract*—High-speed serial data communication is now very popular for connecting various resources in high-performance computing systems. In such high-speed serial links, a line coding is important to control the run length (RL) and the running disparity (RD), because a large run length causes insufficient transitions on data-links that make it difficult to perform reliable clock and data recovery (CDR), and a bounded running disparity is needed for maintaining the DC-balance of data links. These requirements for a line coding, however, cause additional bits to be transmitted, which decreases the throughput of data transmission. This paper presents a new approach to reducing the overhead of line codings, guaranteeing the bounded run length and running disparity. Our experimental results show that the proposed technique reduces the overhead by up to 98% compared to conventional line codings and by up to 48% compared to a latest low-overhead line coding.**

Key Words: Serial Communication, Line Codings, Run-length, DC-Balance.

I. Introduction

Serial communication is widely used because it has higher performance, compared to parallel communication, in speed, pin counts, EMI, and so on [1]. In the general serial communication, parallel data is first serialized. After this step, a line coding is applied, and then the encoded data is actually transmitted through data-links. A receiver receives the encoded serial data, decodes it, and then converts it to parallel data. This line coding is needed in order to avoid sending long sequences of "0" or "1" bit. If long sequences of the same bits are sent, it is difficult to perform reliable clock and data recovery (CDR), because the CDR circuit tries to periodically synchronize with the transmitted data by detecting transitions (rising or falling edges) on data-links. The number of same bit is called *run length* (RL). RL increases when the current bit has the same value with the previous one, and it is reset to 1 when a different bit appears. RL is always greater than 0.

Another important role that line codings play is to control the DC-balance of AC-coupled data links. In AC-coupled data links, if the numbers of "0" and "1" that appear in the links are unbalanced, the voltage levels of the links are shifted, and it finally causes errors in samplers of the receiver side. The difference of numbers of "0" and "1" is called *running disparity* (RD). RD initially takes 0, and is incremented (or decremented) when "1" (or "0") appears in data. RD can be negative, 0, or positive.

Line codings are used to bound the RL and RD values, and thus to improve the signal integrity of the transmitted serial data. They, however, introduce additional bits to be transmitted, which decreases the throughput of data transmission. The ratio of the additional bits to the raw data bits is called *bit-overhead*. It is desirable to obtain a line coding with lower bit-overhead for the given RL and RD bounds.

This paper proposes a new approach to reducing the bit-overhead of line codings, guaranteeing the bounded run length and running disparity. It is based on the existing bit stuffing and S-bit inversion techniques. However, our approach is different from the previous techniques, because suitable combination of RL handling and RD handling is pursued in our approach.

The rest of this paper is organized as follows. Section II refers to some related work. Section III introduces existing techniques for handling the RL and RD bounds. In Section IV, the proposed approach is described in details. Section V shows the experimental results of the intensive simulations, and then our approach is compared with the several conventional techniques and also a latest low-overhead line coding. Finally, Section VI summarizes this paper.

II. Related work

In 1983, A. Widmer [2] proposed 8b/10b line coding that is the most popular in serial communication and was implemented to PCIe 2.0, USB 3.0, and so on. The 8b/10b line coding has very high performance in controlling the RL and RD, but also has high bit-overhead. It can control maximum absolute RD to five and maximum RL to three. This encoding technique uses lookup-tables to transform 8-bit data into 10-bit data. It actually has two lookup-tables (*i.e.,* two 10-bit data for each 8bit data), and one table is used depending on the sign (positive or negative) of the current RD. When the current RD is positive (negative, resp.), the negative (positive) set of 10-bit data is used to control the RD. This high performance (small RL and small absolute RD bounds) of this technique is obtained from the sacrifices of the high bit-overhead, which reaches 25%.

The improvement of the 8b/10b algorithm was proposed in 1998 by A. Coles from HP, which is called 16b/18b [3]. Its bit-overhead is 12.5%, a half of the 8b/10b bit-overhead. This encoding technique adds two bits preamble to 16bits data, and those two bits tell a receiver whether the transmitted frame should be inverted or not. That is, when the current RD is positive, and the data frame to be transmitted has positive RD, this frame is inverted in order to decrease the RD. The same

978-1-5090-3531-1/16 $31.00 © 2016 IEEE
DOI 10.1109/MCSoC.2016.45

handling is also performed when the current RD is negative and the data frame has negative RD. The performance in term of bit-overhead is better compared to the 8b/10b, but the RL and RD bounds are worse than the 8b/10b. The maximum RL and maximum absolute RD is 42 and 26, respectively. Although it has the RL and absolute RD bounds higher than the 8b/10b, it is used in SONET [4] .

Another popular line coding is 64b/66b. It is used in IEEE P802.3ae standard for 10Gb Ethernet. This technique also adds two bits preamble to 64-bits scrambled data frame. The added two bits indicate whether the current frame is either a control frame or a data frame. The preamble can be "01" and "10", and if the preamble is "00" or "11", then it means errors occur in the current frame. The bit-overhead of this line coding is only 3.125%. However, this technique can control only RL bounds. That is, the maximum RL is guaranteed to be 64, but the absolute RD can be infinite. Some of similar line codings such as 128b/130b and 128b/132b are very popular, because they are used in PCIe 3.0 and USB 3.1, respectively.

Furthermore, 64b/67b is used in Interlaken protocol [5] that was invented by Cisco Systems and Cortina Systems in 2006. The encoding scheme is almost the same as 64b/66b, but there is an additional 1-bit to indicate whether the 64-bit data must be inverted or not. This encoding makes it possible to bound the RD within $+/-96$, which cannot be done by the 64b/66b. However, to guarantee the RD bounds, it has larger bit-overhead 4.687%, 1.562% larger than the 64b/66b.

In 1983, Bosch [6] developed a serial communication protocol, called Controller Area Network or CAN Bus. This protocol uses a dynamic handling of communication frames. For example, suppose that the RL bound is five. If the transmitted data have the same bits whose length is longer or equal to five, then the inverted bit will be inserted and transmitted. This technique is called bit stuffing. The bit stuffing simply guarantees the RL bounds. In 2015 J. Saade [7] proposed a new line coding that can control RL and RD by using bit stuffing and inverting data with a bit indicator. Their technique reduces the bit-overhead very effectively. For example, the bit-overhead to achieve the same RL and RD bounds as 8b/10b is reduced to 17.4%.

III. Existing techniques

This section introduces several important techniques used for handling the RL and RD bounds. Let *Max_RL* and *Max_RD* denote the given maximum bound of the RL and maximum absolute bound of RD.

A. Controlling RL

A standard technique to manage the RL bounds is *bit stuffing* [6]. In this technique, when a new bit x is given, the RL up to x is updated such that it is is incremented if x is equal to the previous bit, and otherwise, it is reset to 1. If it reaches *Max_RL*, then $\overline{x}$ is inserted after x, and RL is reset to 1. Simple examples are shown in Figure 1 for *Max_RL*$= 5$. In these examples, the hatched bits in "Raw data" are x, and the inserted bits are hatched in "Encoded data". "RL" represents the RL for "Encoded data" up to the corresponding bit position. Note that in the case (b), the "0" bit insertion is actually not needed, because the next bit is "0". However, if this "0" bit insertion is skipped depending on the next bit, decoding the encoded bit sequences becomes complicated. Hence, when the RL reaches *Max_RL*, $\overline{x}$ is always inserted. This procedure is shown in Algorithm 1. Note that *RL* is a global variable that contains the RL value updated by UPDATE_RL(x) procedure.

Raw data	0	1	1	1	1	1		1	0
Encoded data	0	1	1	1	1	1	***0***	1	0
RL	1	1	2	3	4	5	1	1	1

(a)

Raw data	0	1	1	1	1	1		0	0
Encoded data	0	1	1	1	1	1	***0***	0	0
RL	1	1	2	3	4	5	1	2	3

(b)

Fig. 1. Examples of the RL handling for *Max_RL*$= 5$.

Algorithm 1 RL handling

```
1: procedure APPLY_RL_HANDLING(x) ▷ RL is a global var.
2:     TRANSMIT(x);
3:     UPDATE_RL(x);
4:     if (RL == Max_RL) then
5:         TRANSMIT(x̄);
6:         RESET_RL();
7:     end if
8: end procedure
```

In [7], another technique called *modified bit stuffing* is used to manage the RL bounds. In this technique, when a bit insertion is needed, two bits $\overline{x}x$ are inserted. An advantage of this technique is that it does not affect the RD value. Thus, it can be applied independently of the RD handling. It, however causes a large bit-overhead.

B. Controlling RD

For controlling RD, the absolute value of the RD should be decreased when necessary. [7] proposes a low-overhead technique, which we call *S-bit inversion*. Their technique can be summarized as follows under the assumption that S is even and an S-bit long FIFO is available. An input bit stream comes through the FIFO. Suppose that a bit x is obtained from the output of the FIFO, and the FIFO is shifted with a new bit in the input of the FIFO. Let $RD(x)$ denote the RD of the encoded data up to x, and RD_{FIFO} denote the RD of the bits contained in the FIFO. Furthermore, let *RD_TH* (RD Threshold) be *Max_RD*$-S/2$, and it is greater than $S/2$.

- If $|RD(x)| <$ *RD_TH*, then x is just transmitted.
- If $RD(x) =$ *RD_TH*,
  - if $RD_{\text{FIFO}} > 0$, then x, the inverted contents of the FIFO, and an indication bit "1" are transmitted (in this order).
  - if $RD_{\text{FIFO}} < 0$, then x, the (non-inverted) contents of the FIFO, and an indication bit "0" are transmitted.

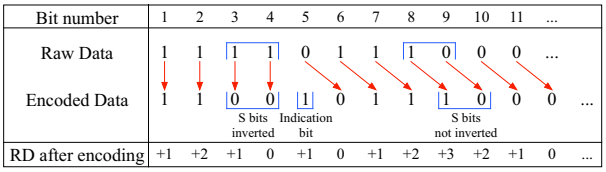

Fig. 2. An Example [7] with *Max_RD*= 3, *RD_TH*= 2 and $S = 2$.

  - if $RD_{\text{FIFO}} = 0$, then x and the (non-inverted) contents of the FIFO are transmitted (with no indication bit).
- If $RD(x) = -RD\_TH$,
  - if $RD_{\text{FIFO}} > 0$, then x, the (non-inverted) contents of the FIFO, and an indication bit "0" are transmitted.
  - if $RD_{\text{FIFO}} < 0$, then x, the inverted contents of the FIFO, and an indication bit "1" are transmitted.
  - if $RD_{\text{FIFO}} = 0$, then x and the (non-inverted) contents of the FIFO are transmitted (with no indication bit).

That is, this RD handling is taken when the absolute value of the RD reaches *RD_TH*. It first checks if the S-bit block in the FIFO, which is denoted by *S-block* henceforth, will increase or decrease the RD value. In the former case ($RD_{\text{FIFO}} > 0$), if the current RD value is positive, the S-block is inverted to decrease the RD value, and otherwise, no inversion is needed. The latter case ($RD_{\text{FIFO}} < 0$) can be handled similarly. In these cases, an indication bit "0" or "1" is inserted after the S-block to indicate the inversion/non-inversion. If the S-block does not change the RD value ($RD_{\text{FIFO}} = 0$), the inversion is not applied, and no indication bit is inserted, because this case can be detected in the receiver side. An example of this technique is shown in Figure 2.

The following two observations are important with respect to the correctness of this RD handling. For simplicity, we assume that "S-block" also includes the indication bit if it is added.

[OB1] The absolute values of the RD go up to at most *Max_RD* inside the S-block. The worst case happens when $RD_{\text{FIFO}} = 0$. For example, if $RD(x) = RD\_TH$, and "1" continues for $S/2$ times in the beginning of the S-block, the RD becomes largest, which is $RD\_TH+S/2$. From the assumption $RD\_TH=Max\_RD-S/2$, it is equal to *Max_RD*.

[OB2] The absolute values of the RD at the end of the S-block are at most *RD_TH*. For example, if $RD(x) = -RD\_TH$, and "1" continues in the entire S-block, the RD reaches $-RD\_TH+S$. If the indication bit is "1", it reaches $-RD\_TH+S+1$. From the assumption $RD\_TH > S/2$, it is equal to *RD_TH* or smaller.

Figure 3 shows how the RD values change around the S-block. The RD value at each bit position is inside the hatched range. From these observations, it is clear that the RD bounds are maintained using this technique.

IV. Proposed Approach

This section proposes a new low bit-overhead line coding based on the dynamic-frame length that can control both the RL and RD, and shows one FPGA-based implementation.

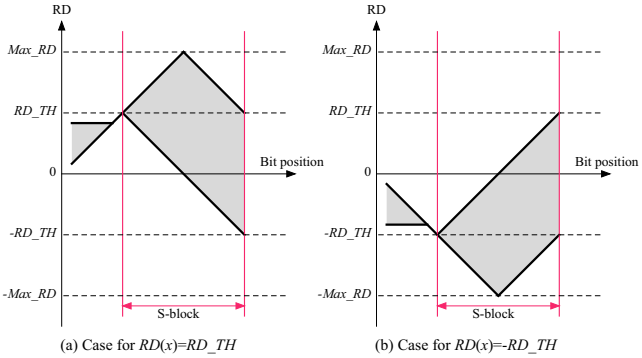

Fig. 3. Chart to show how the RD changes around the S-block.

A. Combining the RL and RD handlings

If both the RD and RL bounds should be controlled simultaneously, a simple serial application of the bit stuffing and the S-bit inversion techniques does not work even in any order as discussed below.

- Suppose that the bit stuffing is applied first. Then, a situation shown in Figure 4 can happen. In this situation, "0" is inserted by the bit stuffing, and it is inverted in the S-block. Thus, the RL value reached 5, which is over *Max_RL*. This is because the inverted bits in the S-block are not recognized when the RL handling is first performed. Thus, the RL value cannot be managed appropriately. Note that this happens, even if the modified bit stuffing is used.
- On the other hand, suppose that the S-bit inversion is applied first. Consider a situation shown in Figure 5. In this case, the S-bit inversion manages the RD value within its maximum (-5) (as shown in Figure 3 (b)), but the standard bit stuffing applied later further decreases

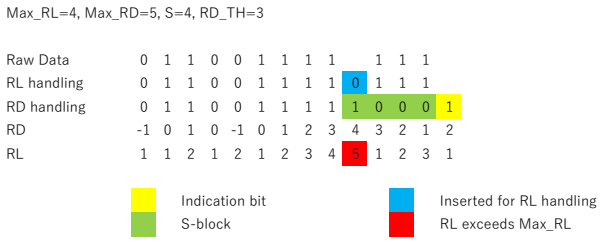

Fig. 4. An Example of RL-first handling with *RD_TH*= 3 and $S = 4$.

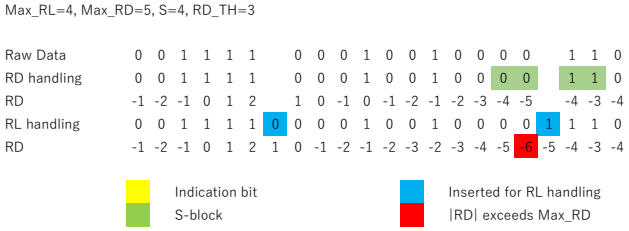

Fig. 5. An Example of RD-first handling with *RD_TH*= 3 and $S = 4$.

the RD values, which causes the violation of *Max_RD*. This is apparently because the bit stuffing does not care about the RD management. This problem can be solved by using the modified bit stuffing, which does not affect the RD values. Thus, [7] uses this approach. As a result, their method suffers from large bit-overhead due to the modified bit stuffing.

In order to reduce the high RL handling overhead, we propose a combined RL-RD handling approach. The main idea is to use APPLY_RL_HANDLING(x) procedure whenever bits are output by the S-bit inversion technique. Its pseudo code is shown in Algorithm 2. This is actually still not complete, but its idea is first explained below.

- Procedure UPDATE_RD() updates the RD value of the bit sequence sent so far, and a global variable RD contains its value, like UPDATE_RL(x).
- Procedure PREPARE_S_BLOCK(RD_{FIFO}) sets a flag S_{flag} for the S-bit inversion technique. That is, if it is "1", the S-block is inverted with the indication bit insertion of "1" at the end, if it is "-1", the S-block is untouched with the indication bit insertion of "0", and finally, if it is "0", the S-block is untouched with no indication bit insertion. These handlings are done at lines 14: and 17:.
- When exiting the S-block, the RD may be again equal to *RD_TH*. In this case, the S-block handling is immediately restarted as shown in lines from 19: to 21:.

In Algorithm 2, thanks to the combined approach, the S-block starts at the right position where the RD reaches $-$*RD_TH*. However, it does not work correctly. Figure 6 shows such an example. A problem occurs when the RD goes below $-$*RD_TH* at the end of the S-block. This is not the violation of *Max_RD* unlike the example in Figure 5. However, the second observation [OB2] mentioned in the previous subsection does not hold. In this example, from $RD_{\text{FIFO}} = 0$, the RD value at the end of the S-block is the same as the one before entering it, if no bit stuffing happens inside the S-block. Due to the effect to the RD by the bit stuffing, this situation happens. Thus, if the bit sequence as shown in Figure 3 (b) follows it, *Max_RD* violation will happen. This problem can be solved by using the modified bit stuffing only inside the S-block. But, it is wasteful if the above situation does not happen. We propose a little bit more sophisticated approach.

In our approach, when exiting the S-block, the RD value is checked, and if it absolute value exceeds *RD_TH*, "1" or "0" is inserted several times until the RD value becomes

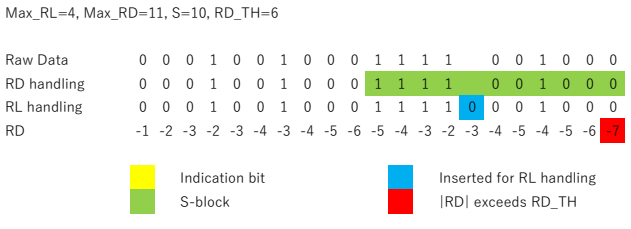

Fig. 6. An Example of incomplete RL-RD combined handling with *Max_RL*= 4, *RD_TH*= 6, and $S = 10$.

Algorithm 2 RL and RD handling (incomplete version)

```
1:  procedure RL-RD_COMBINED                ▷ RD is a global var.
2:      mode ← "outside_S_block";
3:      while (true) do
4:          x ← GET_DATA();
5:          if (mode is "outside_S_block") then
6:              APPLY_RL_HANDLING(x);
7:              UPDATE_RD();
8:              if (|RD| == RD_TH) then
9:                  RD_FIFO ← CAL_RD_S_BLOCK();
10:                 S_flag ←PREPARE_S_BLOCK(RD_FIFO);
11:                 mode ← "inside_S_block";
12:             end if
13:         else if (mode is "inside_S_block") then
14:             x ← INVERT(x) if (S_flag > 0);
15:             APPLY_RL_HANDLING(x);
16:             if (end of S block) then
17:                 INSERT_INDICATION_BIT(S_FLAG);
18:                 UPDATE_RD();
19:                 if (|RD| == RD_TH) then
20:                     RD_FIFO ← CAL_RD_S_BLOCK();
21:                     S_flag ←PREPARE_S_BLOCK(RD_FIFO);
22:                 else
23:                     mode ← "outside_S_block";
24:                 end if
25:             end if
26:         end if
27:     end while
28: end procedure
```

equal to *RD_TH*. We call it *S-block RD-adjustment*. This is implemented as shown in Algorithm 3, and this procedure S-BLOCK_RD-ADJUSTMENT is inserted between lines 18: and 19: in Algorithm 2. Note that during these insertions the bit stuffing may happen. Thus, in each iteration, the latest (updated) RD value is checked in line 8: of Algorithm 3. An example of our RL-RD combined approach is shown in Figure 7. As shown in this example, after S-BLOCK_RD-ADJUSTMENT procedure is applied, the RD handling immediately starts the S-block, because S-BLOCK_RD-ADJUSTMENT stops when the RD value becomes *RD_TH*. In the second S-block, the bit

Algorithm 3 RD adjustment at the end of S-block

```
1:  procedure S-BLOCK_RD-ADJUSTMENT ▷ RD is a global
    var.
2:      while (|RD| >RD_TH) do
3:          if (RD > 0) then
4:              APPLY_RL_HANDLING("0");
5:          else
6:              APPLY_RL_HANDLING("1");
7:          end if
8:          UPDATE_RD();
9:      end while
10: end procedure
```

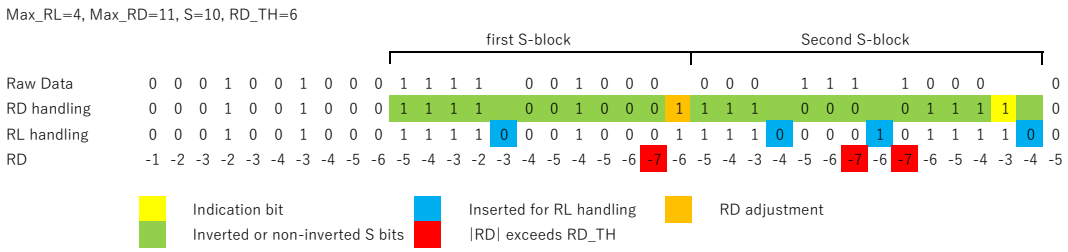

Fig. 7. An Example of complete RL-RD combined handling with *Max_RL*= 4, *RD_TH*= 6, and $S = 10$.

stuffing is applied three times, but no S-block RD-adjustment happened.

A remaining concern is if the first observation [OB1] holds in our algorithm. For example, when $RD(x)$=*RD_TH*, and "1" continues for $S/2$ times in the beginning of the S-block, the RD reaches *Max_RD*. If "1" is inserted by the bit stuffing, the RD exceeds *Max_RD*. However, since "1" continues, "1" is never inserted by the bit stuffing. In the last half of the S-block, "0" continues. So, "1" may be inserted by the bit stuffing. However, since the RD values are sufficiently below *Max_RD* already, it does not exceed *Max_RD*. Hence, [OB1] also holds. From these discussions, we can claim that our algorithm is correct.

B. Decoding algorithm

In order to decode the bit stream received at the receiver side, the stuffed bits should be removed first. This can be done by procedure DECODE_RL shown in Algorithm 4. Our decoding procedure DECODE_COMBINED shown in Algorithm 5 uses it whenever new bits are needed.

In DECODE_COMBINED procedure, the bit obtained by DECODE_RL procedure is just sent out to the receiver body (line 6:). If *RD* reaches *RD_TH*, the procedure goes into "inside_S_block" mode with initializing the FIFO. In this mode, the given bits are just put into the FIFO. The final bit of the S-block is processed in "process_indication_bit" mode. In this mode, the final bit is first added to the FIFO, and the RD value of the FIFO is checked. If it is non-zero, the next bit is the indication bit. Thus, another bit is gotten, and the inversion/non-inversion of the FIFO is performed according to the indication bit. Otherwise, there is no indication bit,

Algorithm 4 Decoding RL part

```
1:  procedure DECODE_RL                          ▷ RL is a global var.
2:      Prev_RL ← RL;
3:      x ←RECEIVE();
4:      UPDATE_RL(x);
5:      if (Prev_RL == Max_RL) then
6:          x ←RECEIVE();
7:          UPDATE_RL(x);
8:      end if
9:      return(x);
10: end procedure
```

Algorithm 5 Decoding RL-RD combined encoding

```
1:  procedure DECODE_COMBINED                    ▷ RD is a global var.
2:      mode ← "outside_S_block";
3:      while (true) do
4:          x ← DECODE_RL();
5:          if (mode is "outside_S_block") then
6:              PUT_DATA(x);                     ▷ x is sent to the receiver
7:              UPDATE_RD();
8:              if (|RD| == RD_TH) then
9:                  mode ← "inside_S_block";
10:                 FIFO← null;
11:             end if
12:         else if (mode is "inside_S_block") then
13:             APPEND_FIFO(x);
14:             if (one bit remaining in S block) then
15:                 mode ← "process_indication_bit";
16:             end if
17:         else if (mode is "process_indication_bit") then
18:             APPEND_FIFO(x);
19:             RD_FIFO ← CAL_RD(FIFO);
20:             if (RD_FIFO ≠ 0) then
21:                 x ← DECODE_RL();             ▷ x is an indication bit
22:                 if (x == 1) then
23:                     FIFO ← INVERT(FIFO);
24:                 end if
25:             end if
26:             PUT_DATA(FIFO);
27:             UPDATE_RD();
28:             while (|RD| >RD_TH) do
29:                 x ← DECODE_RL();             ▷ x is discarded
30:                 UPDATE_RD();
31:             end while
32:             if (|RD| == RD_TH) then
33:                 mode ← "inside_S_block";
34:                 FIFO ← null;
35:             else
36:                 mode ← "outside_S_block";
37:             end if
38:         end if
39:     end while
40: end procedure
```

and thus, DECODE_RL is not needed. In any case, the entire block in the FIFO is sent out to the receiver body. Then, the bits inserted by the S-block RD-adjustment are detected and discarded (lines 28: – lines 31:). Finally, it is checked if "inside_S_block" mode should be immediately started or not, and an appropriate mode is set.

C. Implementation

The proposed encoding and decoding methods can be implemented as shown in Figure 8.

Figure 8 (a) shows the encoder block diagram. The input streams are shifted to "FIFO" block, which is actually an S-bit shift-register. The output of the FIFO is connected to a four-input multiplexer. This four-input multiplexer is controlled by "Controller" block as follows.

- "FIFO" input is selected when non-inverted S-block is used.
- "$\sim$FIFO" input is obtained by inverting the FIFO output, and thus, is used when the inverted S-block is needed (*i.e.,* either $RD =$*RD_TH* and $RD_{\mathrm{FIFO}} > 0$ or $RD = -$*RD_TH* and $RD_{\mathrm{FIFO}} < 0$).
- "$\sim$TX" input is obtained by inverting the current bit "TX", and thus, is used when *RL=Max_RL* for the bit stuffing.
- "Indicator/RD_Adj" input is used when an indication bit is inserted. That is, when "Controller" block decides to invert or not to invert data in the S-block in case of $RD_{\mathrm{FIFO}} \neq 0$, this input is inserted after the last bit of the S-block is shifted out. This input is also used for the S-block RD-adjustment as mentioned in section IV-A. That is, when exiting from the S-block, $|RD| >$*RD_TH* can happen. In such a case, several "0" or "1" are inserted to reduce $|RD|$ until $|RD| =$*RD_TH*. Note that the bit stuffing might be needed during this handling.

"RD-counter" and "RL-counter" are used inside "Controller" block. "RL-counter" is counted up when the current bit is equal to the previous bit, and it is reset to 1 otherwise. "RD counter" is counted up when the current bit is equal to 1, and counted down when the current bit is equal to 0. "Clear" signal to "Controller" is used to reset "RL counter" to 1 and to reset "RD counter" to 0.

When bits need to be inserted for the bit stuffing, the S-bit inversion, or the S-block RD-adjustment, "FIFO" block and the input stream should be stopped. "Hold" signal from "Controller" is used for this purpose. "RD_{FIFO} Calculator" calculates RD_{FIFO}, which is actually represented by two bits, by counting the number of 1 in the FIFO and comparing it to $S/2$ as follows.

- If number of 1 is greater than $S/2$, then $RD_{\mathrm{FIFO}} =$ "+"
- If number of 1 is less than $S/2$, then $RD_{\mathrm{FIFO}} =$ "−"
- If number of 1 is equal to $S/2$, then $RD_{\mathrm{FIFO}} =$ "0"

This function is implemented as a combinational circuit.

Once RD_{FIFO} is calculated, it is used to perform the S-bit inversion and decide the indication bit.

Figure 8 (b) shows the decoder block diagram. It receives serial data streams sent from the transmitter, and decodes them by removing redundant bits inserted for the bit stuffing, the S-bit inversion, and the S-block RD-adjustment. "RD_{FIFO} Calculator", "RD counter", and "RL counter" also work similarly to those in the encoder. "Controller" stops "FIFO" shift register block and "Serial to Parallel" block when the current bits are redundant and should be discarded. "FIFO" block also includes the function to invert the whole bits in it, depending on "Indicator" bit. When the data is passed to "Serial to Parallel" block, it is ordered to a 8-bit parallel data, and is latched to "Data" output port.

The above blocks are actually implemented in an FPGA, and are simulated with ISim simulator to check their functions. Our FPGA implementation of the method [7] is also done for comparison. The target FPGA is XC7K325T, and the device utilization of the encoders and the decoders are shown in Table I (*Max_RL*$= 5$, *RD_TH*$= 6$, $S = 6$). This table also shows the maximum clock frequency of each block found in the timing reports of the synthesis phase. From this table, our method have slight overheads in the size and the performance, but they are not so bad.

The simulation waveforms of the encoder and decoder

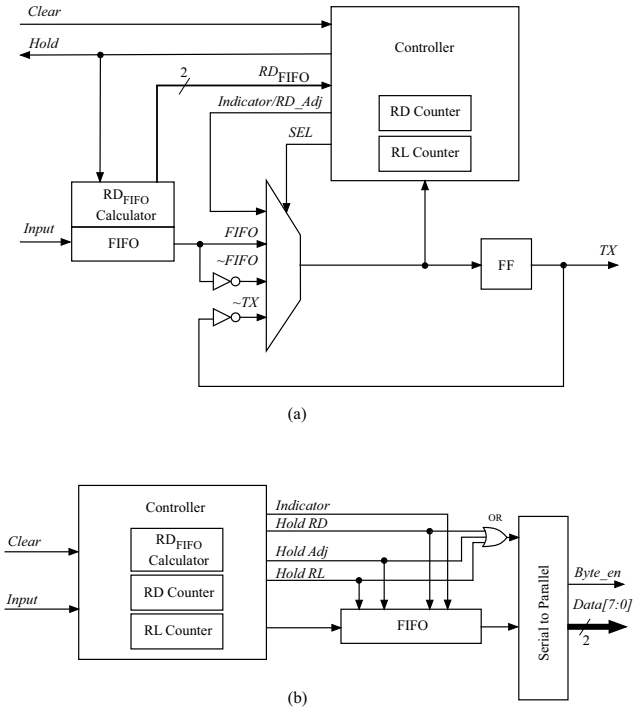

Fig. 8. (a) Encoder (b) Decoder block diagrams.

TABLE I
FPGA IMPLEMENTATION SUMMARY.

		Slice Reg.	Slice LUTs	Max Freq.
Encoder	Proposed	37	111	413 MHz
	[7]	47	74	478 MHz
Decoder	Proposed	54	68	551 MHz
	[7]	58	66	558 MHz

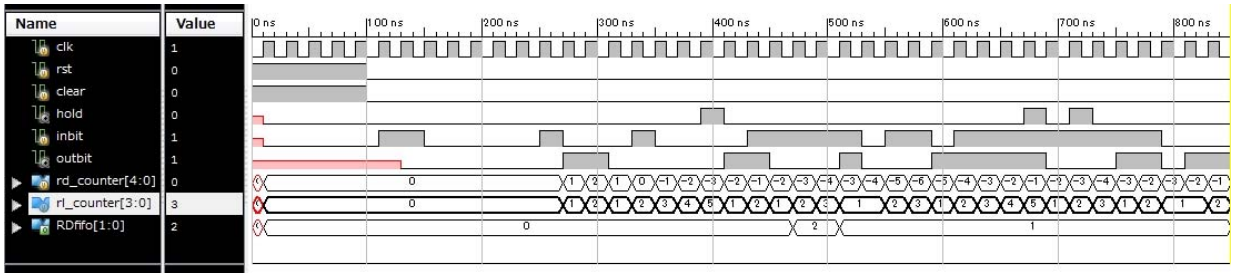

Fig. 9. Simulation waveforms of the encoder.

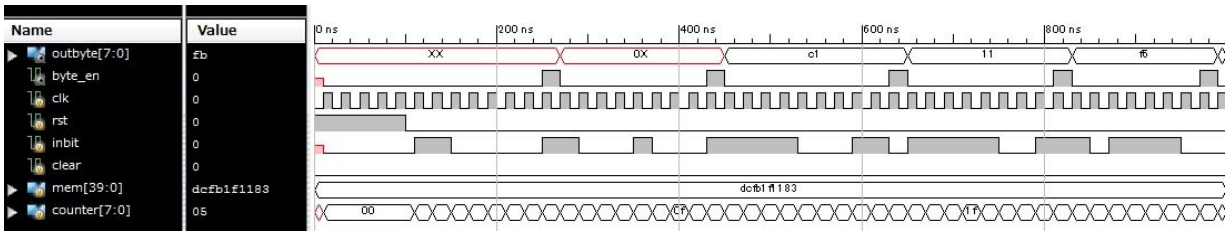

Fig. 10. Simulation waveforms of the decoder.

blocks are shown in Figure 9 and Figure 10. The input data stream for the encoder is "1100000100010001111 101101111111..." and that for the decoder is "1100000 11000100011111000110...".

V. Experimental Results

In order to evaluate the bit-overhead of the proposed method and compare them with those of the existing methods, we have performed intensive simulations by using Matlab for various line code parameters (*i.e.,* the RL bound, the absolute RD bound, and S). One simulation run uses random data stream of 400K bits, and average bit-overhead is computed as follows using 200 simulation runs for each method.

$$\text{Average bit-overhead} = \frac{\text{Average number of inserted bits}}{\text{Average raw data length}}$$

A. Comparing to general line coding methods

First, three typical line codings, 8b/10b, 16b/18b, and 64b/67b, are used for comparison, because the RD bounds are specified for them. Note that the others such as 64b/66b, 128b/130b, 128b/132b have unlimited RD values, and thus, those are not used for this comparison. Table II shows the parameters of these line codings. For applying the proposed method and the method [7], the same bounds are used as the original ones. The S parameters are chosen as 2, 12, and 64 for 8b/10b, 16b/18b, and 64b/67b.

TABLE II
Parameters of the line codings used for comparison.

Line coding	*Max_RL*	*Max_RD*	Bit-overhead
8b/10b	5	3	25%
16b/18b	42	26	12.5%
64b/67b	64	96	4.687%

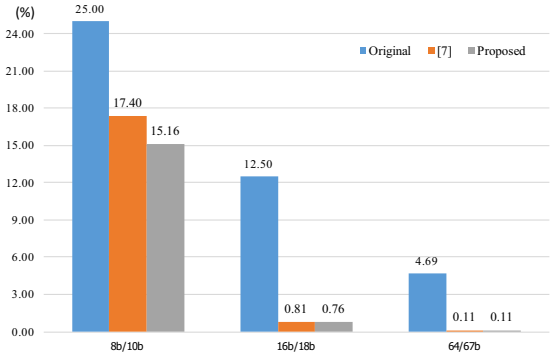

Fig. 11. Bit-overhead comparison to existing line coding methods.

Figure 11 compares the bit-overhead of each method, where those bit-overheads for the proposed method and [7] are obtained by simulation as mentioned above. Thanks to the S-bit inversion technique, the proposed method and [7] have very small bit-overheads when the RL and absolute RD bounds are relatively large. For a small RL bound such as 5, our method has smaller bit-overhead compared to [7], because the modified bit stuffing used in [7] worsens the bit-overhead. On the other hand, for larger RL bounds, the RL value rarely reach its bound, and thus, there are just a few chances of applying the (modified) bit stuffing. Hence, the bit-overheads of the two methods are very close. In 64b/67b case, the bit-overhead improvement by these method reaches about 98% ((4.687-0.11)/4.687=0.977).

B. Comparing to [7]

For further comparison of the proposed and [7] methods, the Matlab simulations are performed with various

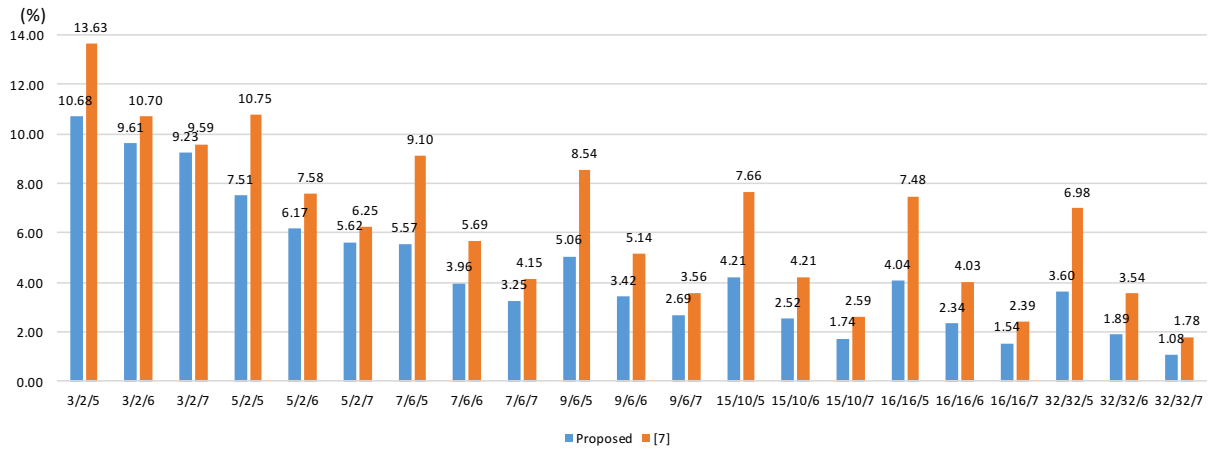

Fig. 12. Bit-overhead comparison to [7].

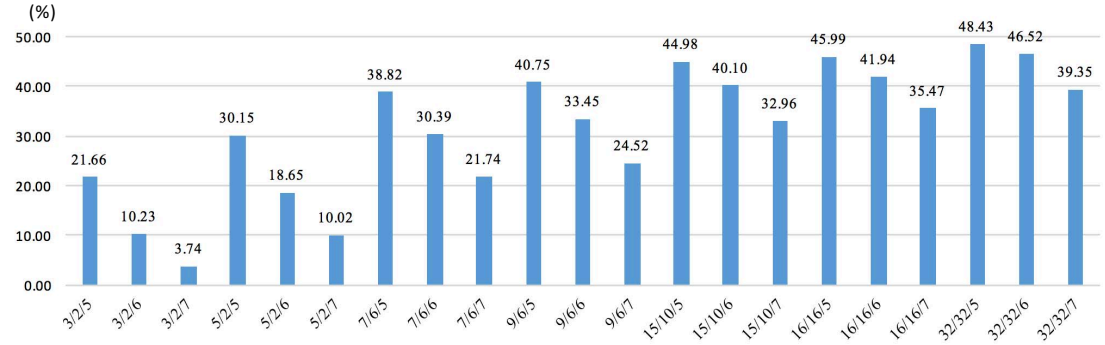

Fig. 13. Improvement of bit-overhead by the proposed method.

(rather randomly chosen) parameters. The bit-overheads for them are shown in Figure 12. In this figure, $x/y/z$ shows *RD_TH*/*S*/*Max_RL*. Figure 13 shows the improvement by our method for each parameter set, which is defined as

$$\frac{\text{(Bit-overhead of [7])} - \text{(Bit-overhead of ours)}}{\text{(Bit-overhead of [7])}}$$

These results show that the proposed method always has lower bit-overhead compared to [7]. Especially, for smaller RL bounds, our method has better results. About 48% bit-overhead improvement is obtained in "32/32/5" case.

VI. Conclusion

In this paper, we have proposed a new approach to a low bit-overhead line coding with both RD and RL limited. Its key idea is to combine the RD and RL handlings, instead of applying them in series. The detailed procedures of the proposed method as well as the sketch of its correctness proof are shown in this paper. The proposed method is further implemented in an FPGA, and the functions of the encoder and the decoder are verified by simulation. According to the synthesis timing reports, our encoder has a slight performance overhead compared to [7]. As for the bit-overheads, intensive Matlab simulation results show that our method always has lower bit-overheads than those of [7]. Our future work includes the reduction of the performance overhead of the proposed method. Also, it may be interesting to investigate how power consumption is reduced by using low bit-overhead encodings.

References

[1] A. Athavale and C. Christensen. High-speed serial io made simple, a designers' gguide, with FPGA applications. *Connectivity Solution, Edition 1.0, Xilinx*.

[2] A. X. Widmer and P. A. Franaszek. A DC-balanced, partitioned-block, 8b/10b transmission code. *IBM Journal of Research and Development*, 27(5):440–451, Sept 1983.

[3] A. Coles and D. Cunningham. Hp extended enterprise laboratory http://www.hpl.hp.com/techreports/98/hpl-98-168.pdf. *Low Overhead Block Coding for Multi-Gb/s Links*.

[4] R. J. Boehm. The SONET interface and network applications. In *Global Telecommunications Conference, 1988, and Exhibition. 'Communications for the Information Age.' Conference Record, GLOBECOM '88., IEEE*, pages 975–979 vol.2, Nov 1988.

[5] Interlaken protocol definition, revision 1.2. October 2008.

[6] BOSCH. CAN specification.

[7] J. Saade, A. Goulahsen, A. Picco, J. Huloux, and F. Petrot. Low overhead, DC-balanced and run length limited line coding. In *Signal and Power Integrity (SPI), 2015 IEEE 19th Workshop on*, pages 1–4, May 2015.