

AKADEMIA GÓRNICZO-HUTNICZA
WYDZIAŁ ELEKTROTECHNIKI, AUTOMATYKI, INFORMATYKI I ELEKTRONIKI
KIERUNEK MIKROELEKTRONIKA W TECHNICE I MEDYCYNIE



SYSTEMY DEDYKOWANE W UKŁADACH PROGRAMOWALNYCH

Projekt zaliczeniowy

Implementacja kodowania długości serii (RLE) w układzie z rodziną Zynq-7000

Agnieszka Kamień i Magdalena Rosół

Kraków, 2021-05-06

Spis treści

1 Historia zmian dokumentu	3
2 Opis projektu i algorytmu	4
2.1 Cel projektu	4
2.2 Opis algorytmu	4
2.3 Diagram algorytmu	4
3 Opis behawioralny algorytmu	7
3.1 Ustalenie architektury modułu	7
3.1.1 Moduł enkodera	7
3.1.2 Moduł dekodera	8
4 Kod w .v/.sv	9
4.0.1 Moduł enkodera	9
4.0.2 Moduł dekodera	10
5 Testy modułu behawioralnego	10
6 Opis syntezowalny algorytmu	11
6.1 Potokowa wersja modułu	11
6.2 Kod w .v/.sv	11
6.3 Testy modułu syntezowalnego	11
7 Inkorporacja modułu do większego systemu	12
7.1 Magistrala AXI	12
7.2 Sterownik	12
8 Symulacja	12
9 Uruchomienie systemu w układzie Zynq-7000	13

1 Historia zmian dokumentu

Wersja dokumentu	Data	Opis
1.0	22.04.2021	Pierwsza wersja dokumentu
2.0	29.04.2021	Opis pierwszej wersji modułu behawioralnego enkodera oraz dekodera.

Historię zmian edycji projektu można prześledzić w repozytorium: <https://github.com/agnieszkakam/RLE>.

2 Opis projektu i algorytmu

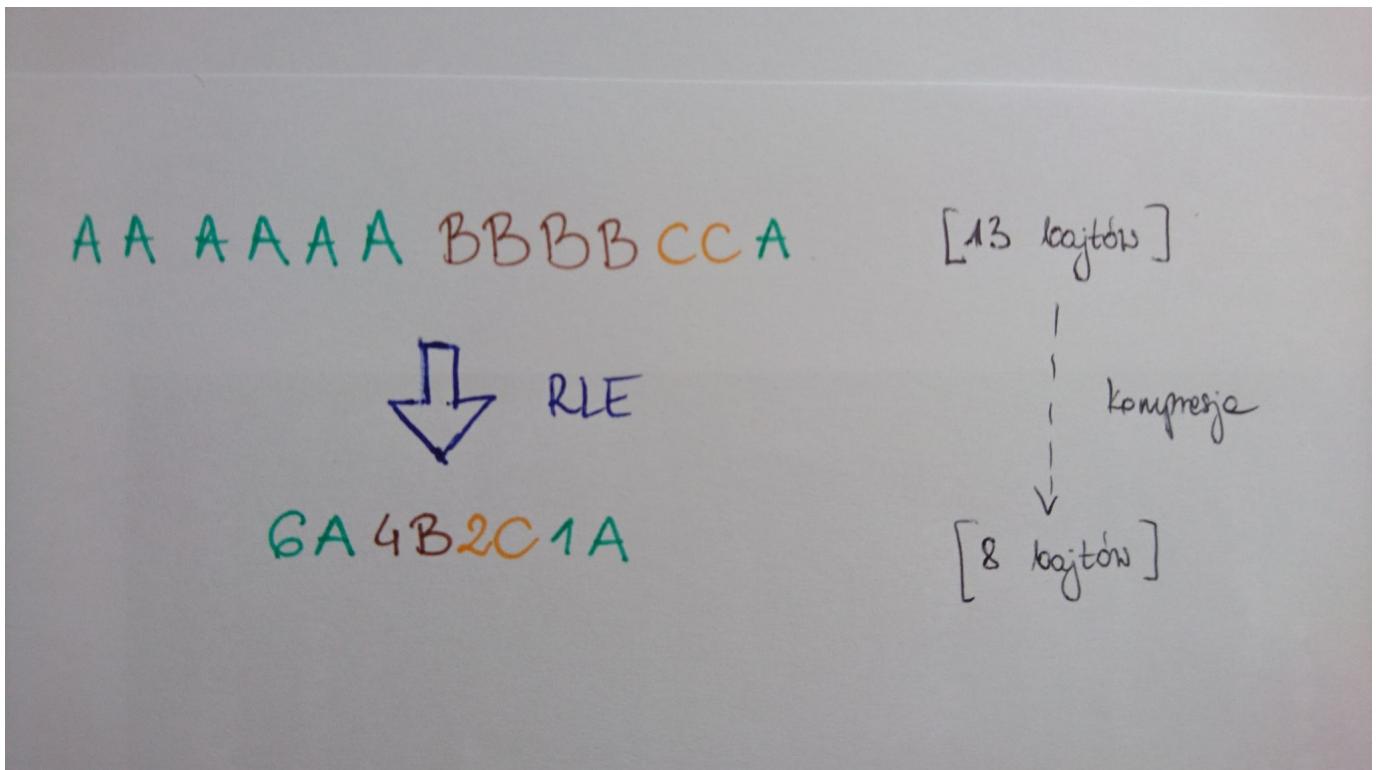
2.1 Cel projektu

Celem niniejszego projektu jest opracowanie modułów enkodera i dekodera realizujących kodowanie długości serii. Moduły te mają zostać napisane w języku opisu sprzętu Verilog bądź SystemVerilog, a docelowo powinny zostać uruchomione na płytce ZedBoard Zynq-7000 firmy Xilinx.

2.2 Opis algorytmu

Kodowanie długości serii (ang. Run-Length Encoding, RLE) jest formą bezstratnej kompresji danych. Oznacza to, że skompresowane dane mogą zostać z powrotem przekonwertowane do dokładnie takiej samej postaci jak w reprezentacji oryginalnej. Żadne informacje nie są tracone podczas kompresji - proces jest w pełni odwracalny.

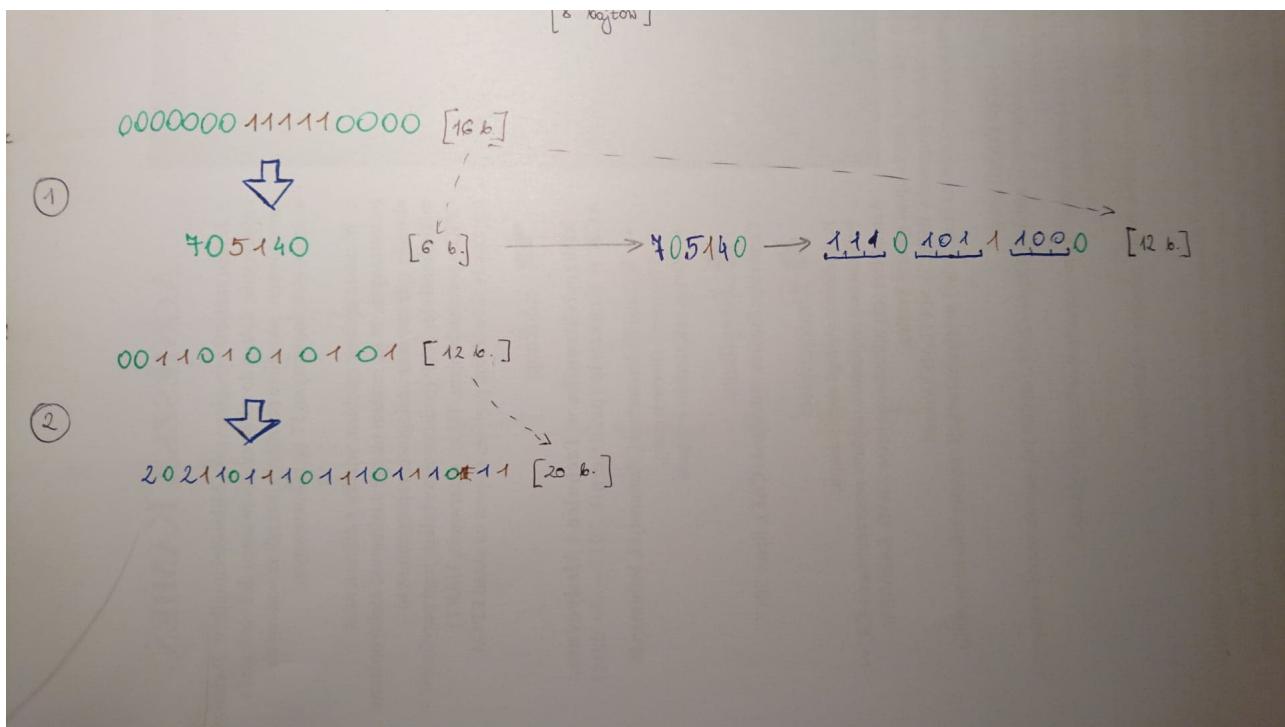
Zdecydowaną zaletą algorytmu jest łatwość jego implementacji oraz to, że nie wymaga dużej ilości zasobów CPU. Polega on na zastąpieniu ciągu znaków liczbą wystąpień danego znaku oraz jego symbolem (patrz rys. 1). Kompresja jest tym skuteczniejsza, im dane są bardziej powtarzalne - wielokrotnie powtarzający się „kolejno” bajt można zapisać w zaledwie dwóch bajtach. W najgorszym przypadku rozmiar danej może zwiększyć się aż dwukrotnie (np. podczas zapisu ciągu ABCD po kompresji otrzymuje się 1A1B1C1D).



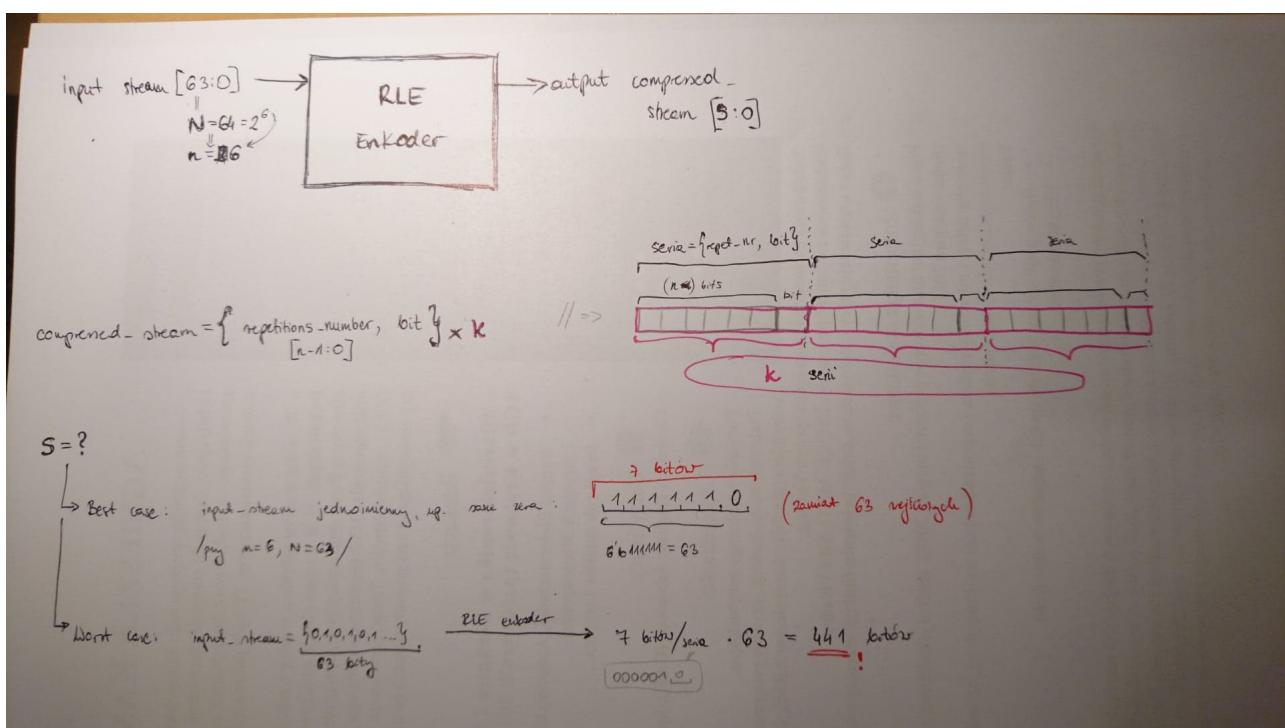
Rysunek 1. Przykład kodowania RLE

2.3 Diagram algorytmu

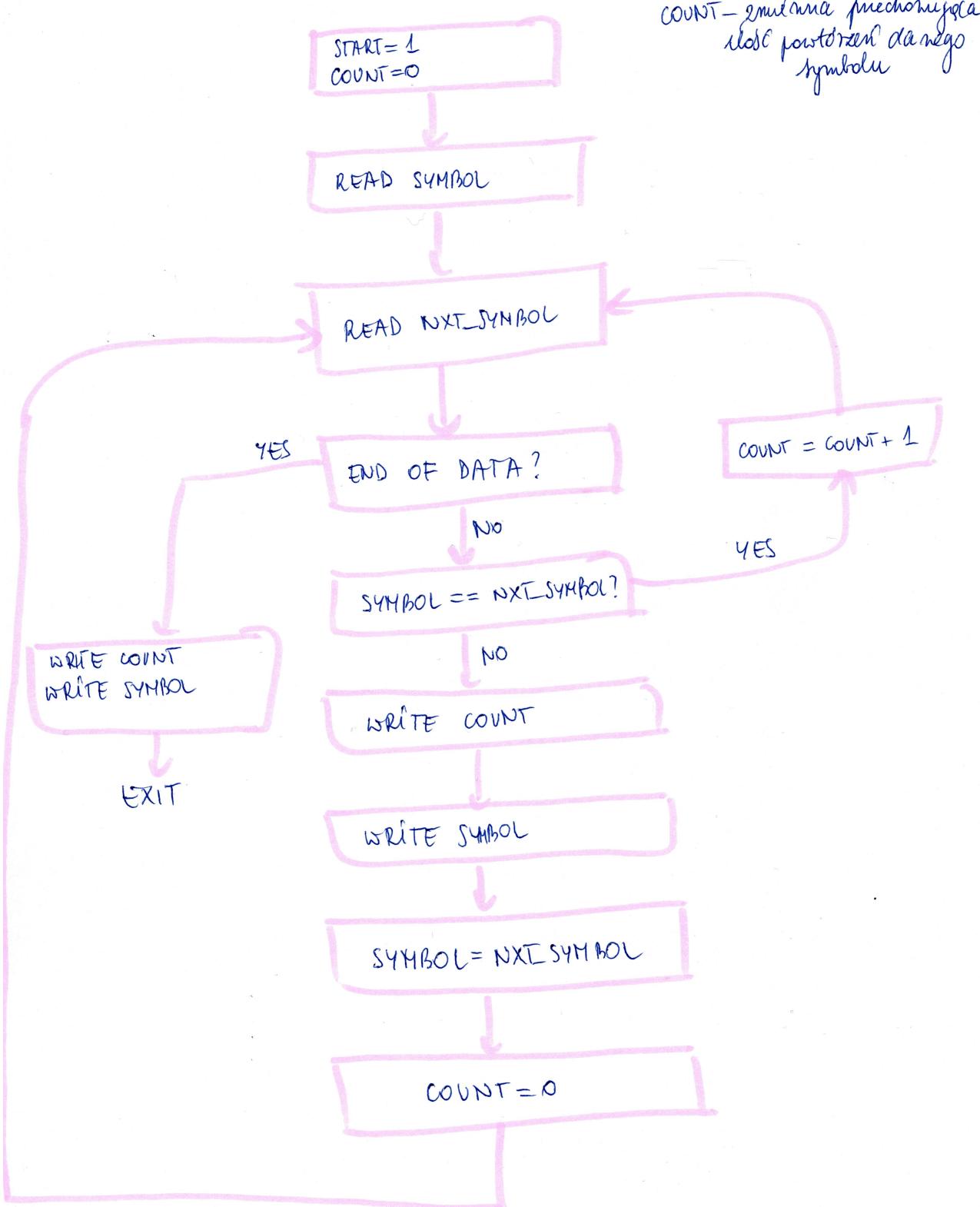
Jeżeli długość serii kodujemy na n bitach, to w sekwencji wejściowej możemy mieć 2^n znaków (zer lub jedynek). Wykorzystamy to, że nigdy nie kodujemy długości serii równej zero. Zatem symbol równy 0, możemy wykorzystać dla serii 16-znakowej.



Rysunek 2. RLE - przykład kodowania strumienia bitów



Rysunek 3. RLE Enkoder – pierwszy zamysł



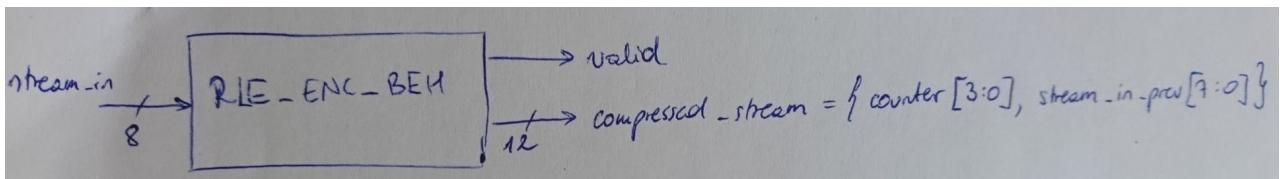
Rysunek 4. Schemat blokowy działania algorytmu

3 Opis behawioralny algorytmu

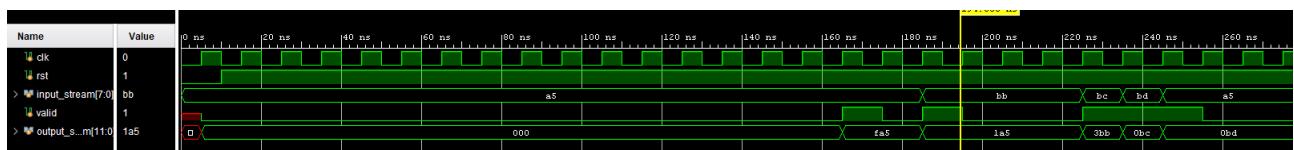
3.1 Ustalenie architektury modułu

3.1.1 Moduł enkodera

W pierwszym podejściu zaimplementowano moduł enkodera RLE bez parametrów. Przyjmuje on na wejście 8-bitowy strumień danych *stream_in*. Dopóki w kolejnych taktach zegara na wejściu enkodera pojawią się takie same sekwencje, wewnątrz modułu inkrementowany jest licznik, a wyjście *valid* znajduje się w stanie niskim. Gdy licznik się przepięni lub *stream_in* ulegnie zmianie, sygnał *valid* ustawiany jest w stan wysoki, a na wyjściu *compressed_stream* pojawią się dane zakodowane długością serii. W tej wersji modułu zastosowano licznik 4-bitowy, zatem jego zakres mieści się od wartości 0 do wartości 15. Po szesnastokrotnym wystąpieniu serii, na wyjściu pojawią się informacje o wystąpieniu takiej sekwencji, a licznik zaczyna na nowo liczyć od 0. Przykład takiego przebiegu zaprezentowano na rys. 6. Warto przypomnieć o przyjętej konwencji zliczania – by wykorzystać maksymalnie zakres licznika wszystkie wartości, które może on przyjąć, są wykorzystywane. Jednak wartości te są pomniejszone o jeden względem rzeczywistej liczby wystąpień serii. Dla zobrazowania: jednokrotne pojawienie się na wejściu enkodera strumienia 0xBC, poskutkuje wyjściem 0x0BC, a nie – jak można by się spodziewać – 0x1BC. Z kolei szesnastokrotne ($15 = h'F$) wystąpienie serii 0xCC, znajdzie odzwierciedlenie na wyjściu postaci 0xFCC.



Rysunek 5. Prototyp enkodera RLE



Rysunek 6. Przykładowy wynik symulacji modułu behawioralnego enkodera RLE

Kolejnym krokiem była parametryzacja modułu enkodera. Dla parametrów dobranych jak w wyżej opisanym module (bez parametryzacji) efekt symulacji był taki jak na rys. 6. Kod sparametryzowanego modułu umieszczono w podrozdziale 4.

Zdecydowano, że moduły RLE posłużą do kodowania sekwencji DNA. Ludzki genom składa się z kilku miliardów nukleotydów, które w sekwencji DNA oznaczone są przez litery 'a', 'c', 'g' oraz 't' od pierwszych liter ich nazw. Ponieważ inne litery nie powinny zostać wykorzystane w sekwencji DNA postanowiono dołożyć dodatkowe kodowanie nukleotydów, by skompresować ich rozmiar. Dzięki temu sekwencja wyjściowa będzie składała się z n -bitów, gdzie n jest sumą bitów licznika oraz dwóch bitów kodujących odpowiedni nukleotyd (A, D, C, T). Enkoder DNA-RLE na wejście przyjmuje dane o szerokości 8 bitów, odpowiadające wartościami liczbowym znaków ASCII: 'a', 'c', 'g', 't'.

By wyznaczyć najbardziej optymalną szerokość licznika enkodera, przeprowadzono testy na 5 zestawach danych. Analizowany zestaw danych był sekwencją DNA składającą się z 20 tysięcy nukleotydów, a testy wykonywano dla szerokości licznika z zakresu {1,2,3,4}. Najwyższy współczynnik kompresji za każdym razem osiągnięto dla licznika o szerokości 1 bita, wynosił on ok. 3,3.

II z kodowaniem; input DATA_WIDTH = 8 Sequence length = 20k																																
Test No.	CTR_WIDTH	VALID_CRC	Długość reaktywanych bitów (2+CTR_WIDTH) * VALID_CRC [w bitach]	Długość opakowania [bitów] DATA_WIDTH * Ry-length [w bitach]	Compresion Ratio $\left(\frac{100k}{Y}\right)$																											
1 (net 1)	1	16 041	48 123																													
	2	15 108	60 432																													
	3	15 049	75 245																													
	4	15 048	90 288																													
2	1	15 937	47 811																													
	2	14 965	59 860																													
	3	14 904	74 720																													
	4	14 903	89 418																													
3	1	16 033	48 117																													
	2	15 152	60 608																													
	3	15 109	95 545																													
	4	15 108	95 648																													
4	1	15 969	47 907																													
	2	15 018	60 072																													
	3	14 960	74 800																													
	4	14 960	89 760																													
5	1	16 009	48 027																													
	2	15 061	60 244																													
	3	15 011	95 077																													
	4	15 011	90 026																													
Wykładywanie: $a = 100'$ $c_0 = 101'$ $t = 100'$ $(DATA_WIDTH + CTR_WIDTH) * VALID_CRC$																																
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr> <td>1</td><td>1</td><td>044</td><td>368</td></tr> <tr> <td>2</td><td>2</td><td>114</td><td>408</td></tr> <tr> <td>3</td><td>3</td><td>165</td><td>536</td></tr> <tr> <td></td><td></td><td>180</td><td>546</td></tr> <tr> <td></td><td></td><td>143</td><td>533</td></tr> <tr> <td></td><td></td><td>140</td><td>570</td></tr> <tr> <td></td><td></td><td>163</td><td>966</td></tr> </table>					1	1	044	368	2	2	114	408	3	3	165	536			180	546			143	533			140	570			163	966
1	1	044	368																													
2	2	114	408																													
3	3	165	536																													
		180	546																													
		143	533																													
		140	570																													
		163	966																													

Rysunek 7. wsp. kompresji

3.1.2 Moduł dekodera

Kolejnym wymaganym w projekcie modułem był moduł dekodera. Moduł ten odpowiada za "odwracanie" działania enkodera – zamienia skompresowane dane z powrotem na strumień pojedynczych danych. Oznacza to, że pojawienie się na wejściu modułu 0x11A5 spowoduje pojawienie się na wyjściu 0x11 + 1 (jak opisywane w 3.1.1) symboli A5.

Moduł został sparametryzowany analogicznie do modułu enkodera, by zachować jednolitość danych.

Moduł przyjmuje na wejściu skompresowaną daną *compressed_stream_in*, w której występuje określona parametrami liczba bitów przeznaczona na daną oraz na licznik. Dla wyjaśnienia działania modułu przyjęto parametry 8 bitów dla danej oraz 6 bitów dla licznika. Oznacza to, że dana wejściowa 0x11A5 poda wartość danych równą 0xA5 oraz licznik równy 0x11. Wyjściem z modułu jest 8-bitowy strumień bitów danych *initial_stream*. Dla ww. danej wejściowej (0x11A5) wyjście powinno przyjąć postać A5, A5, A5, A5, ... aż do 0x11 + 1. By dane nie były wczytywane ciągle, co zaburzałoby prawidłowe działanie modułu zastosowana została flaga *rd_flag* oraz licznik opisywany wyżej. Gdy flaga wynosi 0 oznacza to, że moduł jest gotowy na przyjmowanie nowych danych wejściowych (nie przetwarza obecnie żadnej danej), natomiast gdy znajduje się w stanie wysokim – obecne dane są przetwarzane, moduł blokuje przyjmowanie nowych danych do czasu gdy licznik będzie wynosił zero. W każdym taktie zegara licznik jest dekrementowany, gdy osiągnie zero – flaga ustawiana jest na stan niski oraz wczytywana kolejna dana, z której kolejno brane są kolejne wartości danych oraz licznika.

Kod modułu umieszczono w podrozdziale 4.0.2.

Poniżej przedstawiono przykładowe działanie modułu:

Jak widać moduł działa prawidłowo – na wyjściu pojawia się odpowiednia liczba danych w stosunku do wyznaczonego licznika oraz dane są przetrzymywane aż do jego wyzerowania, co jest sygnaлизowane poprzez flagę *rd_flag*.

4 Kod w .v/.sv

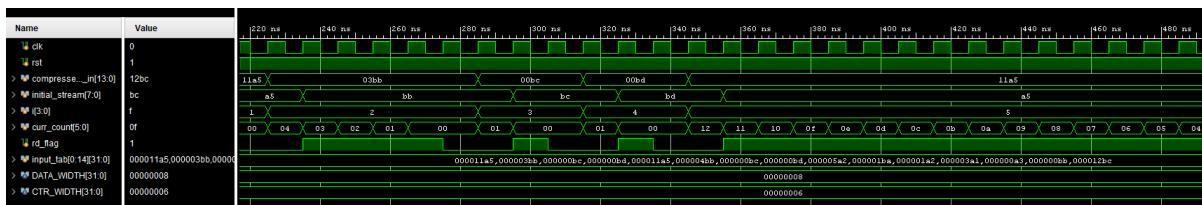
4.0.1 Moduł enkodera

```

1 module rle_encoder_beh
2 (
3     input clk,
4     input rst,
5     input [DATA_WIDTH - 1: 0] stream_in,
6     output reg [DATA_WIDTH + CTR_WIDTH - 1 :0] compressed_stream,
7     output reg valid
8 );
9
10 parameter DATA_WIDTH=8,
11      CTR_WIDTH=4;
12
13 localparam CTR_MAX = (1<<CTR_WIDTH) - 1;
14
15 reg [DATA_WIDTH-1:0] stream_in_prev;
16 reg [CTR_WIDTH:0] seq_counter = 0;
17
18 always @(posedge clk) begin
19     stream_in_prev <= stream_in;
20     if (!rst) begin
21         valid <= 1'b0;
22         compressed_stream <= 0;
23     end else begin
24         if (stream_in == stream_in_prev) begin
25             if (seq_counter != CTR_MAX) begin
26                 valid <= 1'b0;
27                 seq_counter <= seq_counter + 1;
28             end else begin
29                 valid <= 1'b1;
30                 compressed_stream <= {seq_counter[CTR_WIDTH-1:0], stream_in_prev};
31                 seq_counter <= 0;
32             end
33         end else begin
34             valid <= 1'b1;
35             compressed_stream <= {seq_counter[CTR_WIDTH-1:0], stream_in_prev};
36             seq_counter <= 0;
37         end
38     end
39 end
40
41 endmodule

```

Listing 1. Enkoder RLE



Rysunek 8. Przykładowy wynik symulacji modułu behawioralnego dekodera RLE

4.0.2 Moduł dekodera

```
1 module rle_decoder_beh
2 (
3     input clk,
4     input rst,
5     input [DATA_WIDTH + CTR_WIDTH - 1:0] compressed_stream_in,
6     output reg [DATA_WIDTH - 1:0] initial_stream,
7     output reg [CTR_WIDTH - 1:0] curr_count,
8     output reg rd_flag
9 );
10
11 parameter DATA_WIDTH=8,
12       CTR_WIDTH=6;
13
14 reg [7:0] curr_word;
15
16 always @(posedge clk) begin
17     if (!rst) begin
18         initial_stream <= 0;
19         curr_word <= 0;
20         curr_count <= 0;
21     end
22     else begin
23         if (rd_flag == 1'b0) begin
24             curr_count <= compressed_stream_in[DATA_WIDTH + CTR_WIDTH - 1:DATA_WIDTH] + 1'b1;
25             curr_word <= compressed_stream_in[DATA_WIDTH - 1:0];
26             rd_flag <= 1;
27         end
28         if (curr_count == 0) begin
29             rd_flag <= 0;
30         end
31         else begin
32             initial_stream <= curr_word;
33             curr_count <= curr_count - 1'b1;
34         end
35     end
36 end
37 endmodule
```

Listing 2. RLE decoder

5 Testy modułu behawioralnego

6 Opis syntezowalny algorytmu

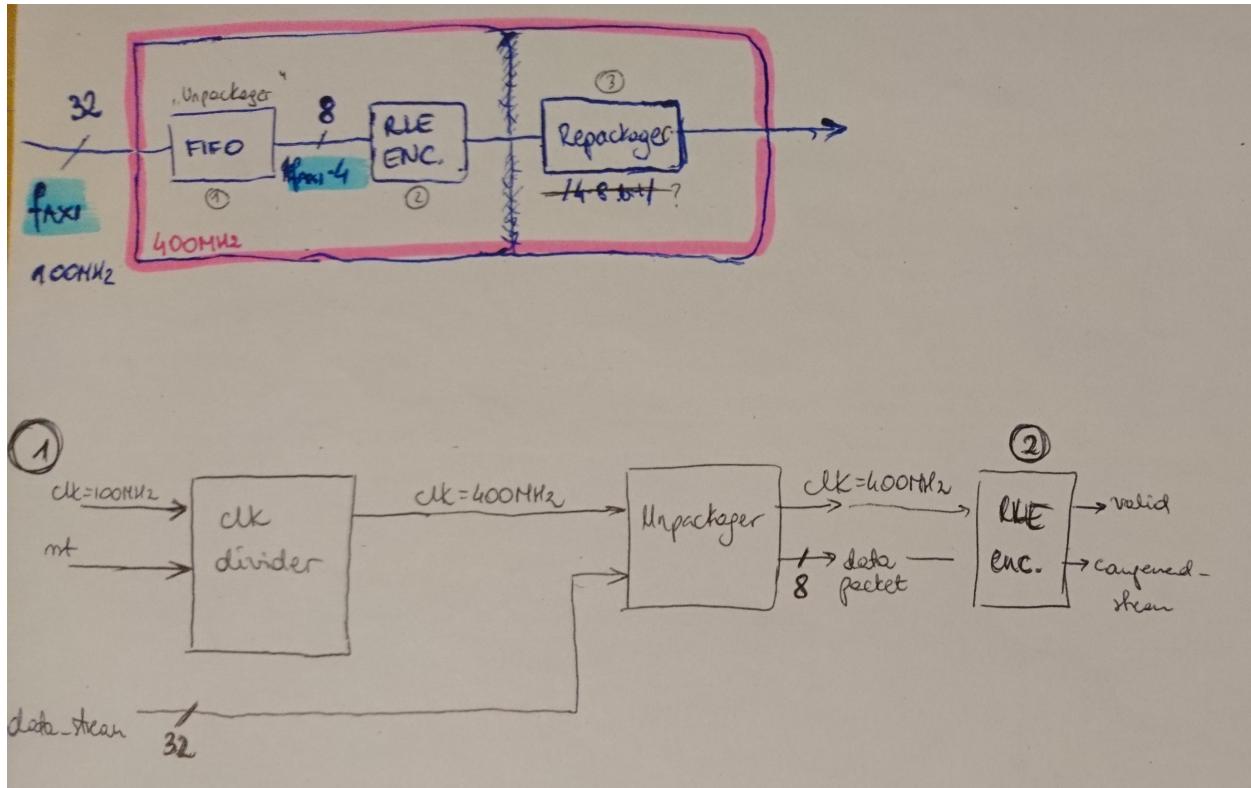
6.1 Potokowa wersja modułu

6.2 Kod w .v/.sv

6.3 Testy modułu syntezowalnego

7 Inkorporacja modułu do większego systemu

Dodatek modułu clk_divider (400 MHz).



Rysunek 9. system

7.1 Magistrala AXI

7.2 Sterownik

8 Symulacja

9 Uruchomienie systemu w układzie Zynq-7000