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 rodziny Zynq-7000

Agnieszka Kamień i Magdalena Rosół

Kraków, 2021-05-03

Spis treści

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

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

2 Opis projektu i algorytmu

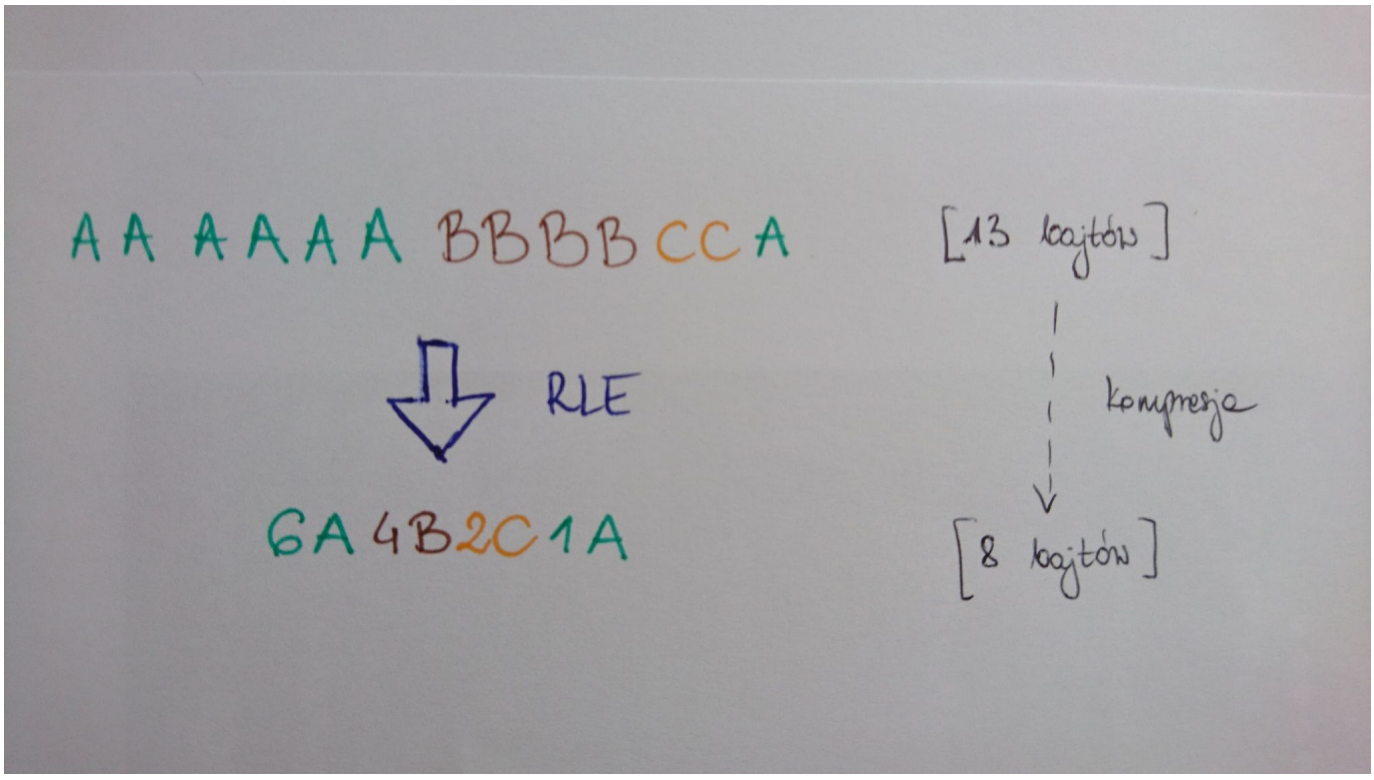
2.1 Cel projektu

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 Endcoding, 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.

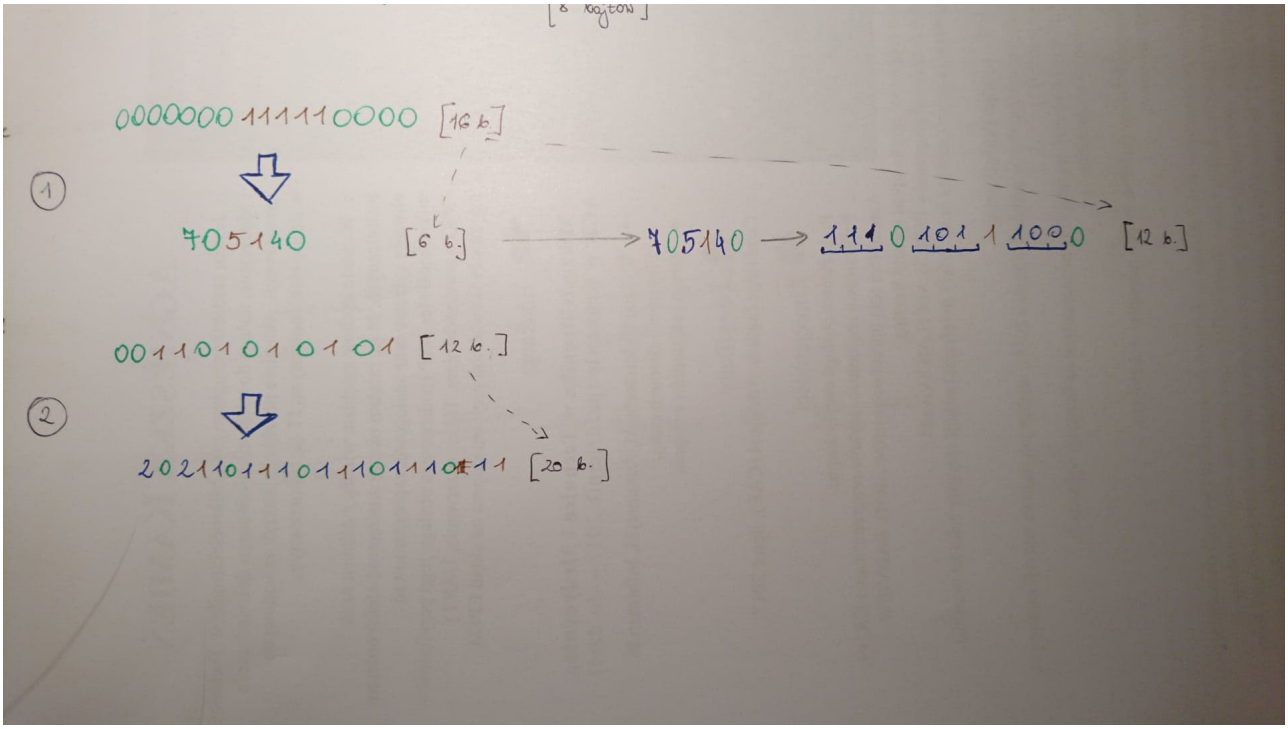
Zdecydowaną zaletą algorytmu jest łatwość jego implementacji oraz to, że nie wymaga dużej ilości zasobów CPU. Polega on na na zastąpieniu ciągu znaków liczbą wystąpień danego znaku oraz jego symbolem (patrz rys. 1). Kompresja jest tym skuteczniejsza, im dane są bardziej powtarzalne - wielokrotnie powtarzający 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).



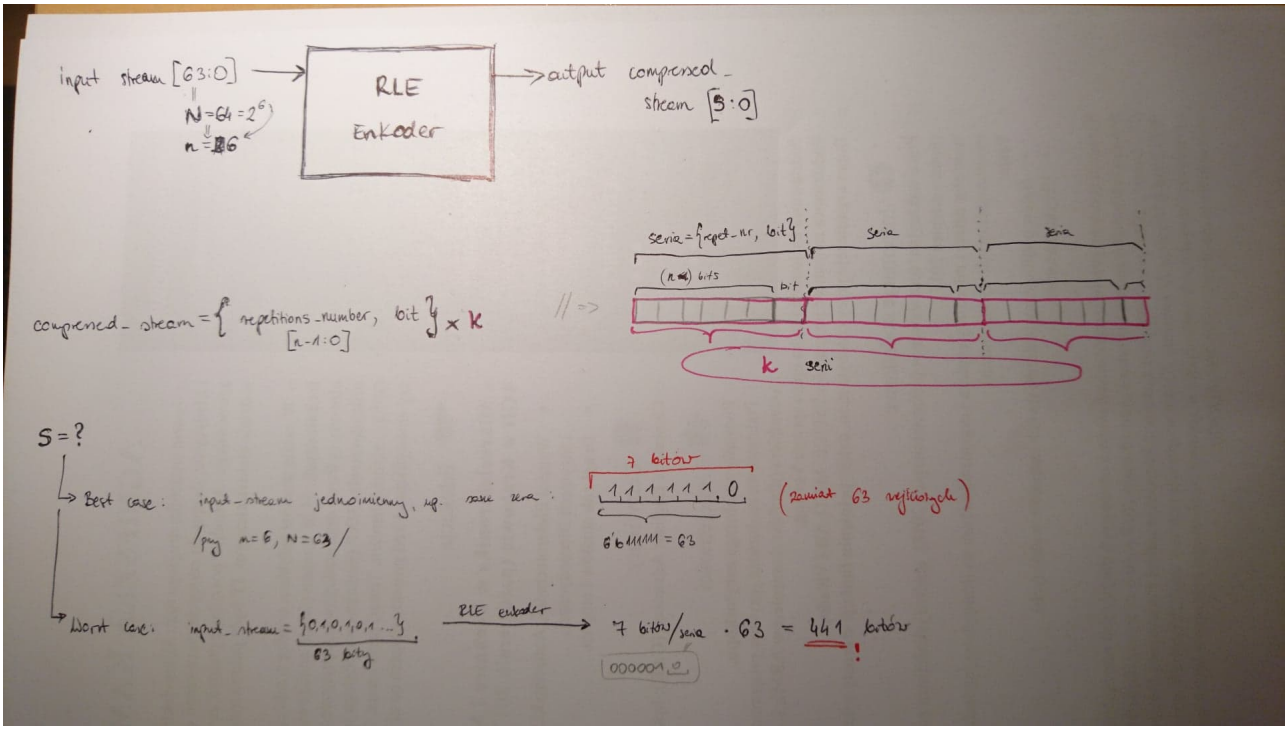
Rysunek 1. Przykład kodowania RLE

2.3 Diagram algorytmu

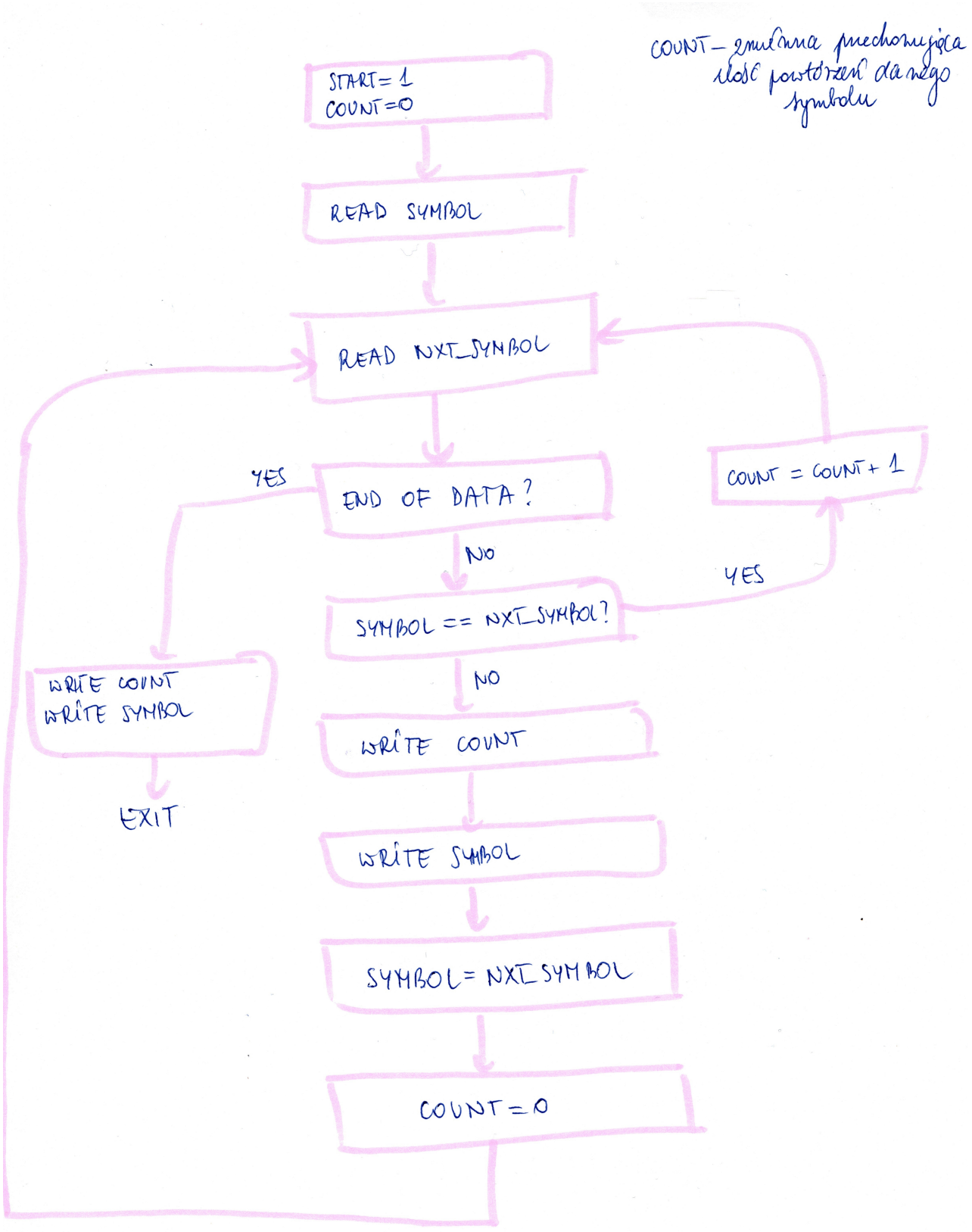
Jeżeli długość serii kodujemy na n bitach, to w sekwencji wejściowej możemy mieć 2^n znaków (zer lub jedynek). Wykorzystamy to, że nigdy nie kodujemy długości serii równej zero. Zatem symbol równy 0, możemy wykorzystać dla serii 16-znakowej.



Rysunek 2. RLE - przykład kodowania strumienia bitów



Rysunek 3. RLE Enkoder – pierwszy zamysł



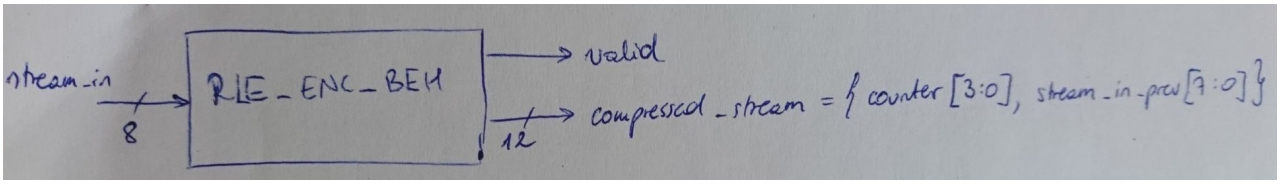
Rysunek 4. Schemat blokowy działania algorytmu

3 Opis behawioralny algorytmu

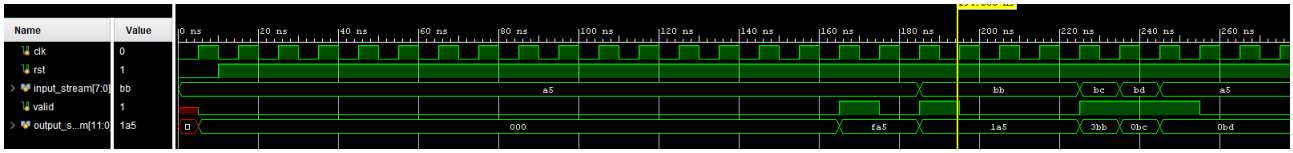
3.1 Ustalenie architektury modułu

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 pojawia się taki sam strumień, wewnątrz modułu inkrementowany jest licznik, a wyjście *valid* znajduje się w stanie niskim. Gdy licznik się przepełni lub *stream_in* ulegnie zmianie, sygnał *valid* ustawiany jest w stan wysoki, a na wyjściu *compressed_stream* pojawiają 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 pojawia się informacja 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 0x**0**BC, a nie – jak można by się spodziewać – 0x**1**BC. Z kolei szesnastokrotne (d'16=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

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.

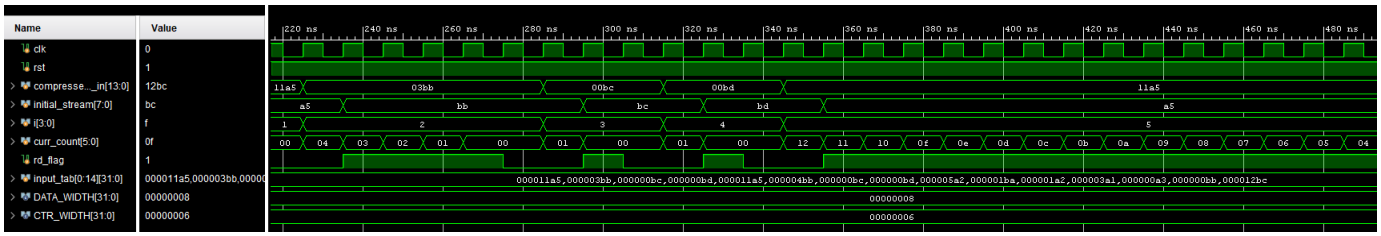
Moduł został sparametryzowany analogicznie do modułu enkodera, by zachować jednolitość danych.

Moduł przyjmuje na wejściu skompresowaną daną *compressed_stream_in*, w której występuje określona parametrami ilość 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 ona 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 takcie 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:



Rysunek 7. Przykładowy wynik symulacji modułu behawioralnego dekodera RLE

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 sygnalizowane poprzez flagę *rd_flag*.

4 Kod w .v/.sv

4.0.1 Moduł enkodera

```
module rle_encoder_beh
(
    input clk,
    input rst,
    input [DATA_WIDTH - 1: 0] stream_in,
    output reg [DATA_WIDTH + CTR_WIDTH - 1 :0] compressed_stream,
    output reg valid
);

parameter DATA_WIDTH=8,
          CTR_WIDTH=4;

localparam CTR_MAX = (1<<CTR_WIDTH) - 1;

reg [DATA_WIDTH-1:0] stream_in_prev;
reg [CTR_WIDTH:0] seq_counter = 0;

always @(posedge clk)  begin
    stream_in_prev <= stream_in;
    if (!rst) begin
        valid <= 1'b0;
        compressed_stream <= 0;
    end else begin
        if (stream_in == stream_in_prev) begin
            if (seq_counter != CTR_MAX) begin
                valid <= 1'b0;
                seq_counter <= seq_counter + 1;
```

```
            end else begin
                valid <= 1'b1;
                compressed_stream <= {seq_counter[CTR_WIDTH-1:0], stream_in_prev};
                seq_counter <= 0;
            end
        end else begin
            valid <= 1'b1;
            compressed_stream <= {seq_counter[CTR_WIDTH-1:0], stream_in_prev};
            seq_counter <= 0;
        end
    end
end

endmodule
```

Listing 1. RLE encoder

4.0.2 Moduł dekodera

```
module rle_decoder_beh
(
    input clk,
    input rst,
    input [DATA_WIDTH + CTR_WIDTH - 1:0] compressed_stream_in,
    output reg [DATA_WIDTH - 1:0] initial_stream,
    output reg [CTR_WIDTH - 1:0] curr_count,
    output reg rd_flag
);

parameter DATA_WIDTH=8,
          CTR_WIDTH=6;

reg [7:0] curr_word;

always @(posedge clk) begin
    if (!rst) begin
        initial_stream <= 0;
        curr_word <= 0;
        curr_count <= 0;
    end
    else begin
        if (rd_flag == 1'b0) begin
        curr_count <= compressed_stream_in[DATA_WIDTH + CTR_WIDTH - 1:DATA_WIDTH] + 1'b1;
        curr_word <= compressed_stream_in[DATA_WIDTH - 1:0];
        rd_flag <= 1;
        end
        if (curr_count == 0) begin
            rd_flag <= 0;
        end
        else begin
            initial_stream <= curr_word;
            curr_count <= curr_count - 1'b1;
        end
    end
end
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

7.1 Magistrala AXI

7.2 Sterownik

8 Symulacja

9 Uruchomienie systemu w układzie Zynq-7000