

## Dokumentacja końcowa PUF

### Temat:

Hashowanie danych za pomocą algorytmu SHA-256.

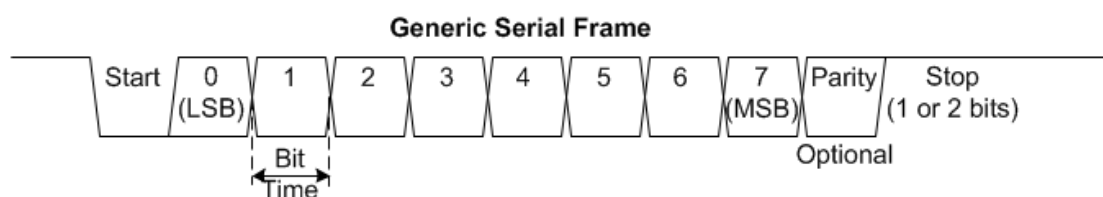
### Funkcjonalności:

1. Odbieranie danych do zakodowania poprzez UART.
2. Przetwarzanie otrzymywanych danych.
3. Wysyłanie wyliczonych hash-y poprzez UART.

### Analiza techniczna elementów systemu

- **Interfejs komunikacyjny**

Jako interfejs komunikacyjny wybrano UART, jest to asynchroniczny nadajnik (TX) – odbiornik (RX).



Rys. 1 Ramka danych UART

Wysyłana wiadomość zaczyna się od bitu „Startu”, który ma wartość logiczną równą zero (czyli stan niski), następnie wysyłane jest osiem bitów które są nośnikiem wiadomości i na końcu znajduje się „Stop”, który ma wartość logiczną odpowiadającą stanowi wysokiemu, może zajmować „długość” jednego bądź dwóch bitów (do projektu wybrano dwa bity).

Jeżeli nie jest wysyłana żadna wiadomość, to na „TX” pojawia się ciągle stan wysoki.

UART wykorzystuje do transmisji dwie linie:

- RX – linia służąca do odbierania wiadomości
- TX – linia służąca do nadawania wiadomości

Dane mogą być nadawane z różną prędkością transmisji („Baud rate”), najbardziej popularne to:

1200, 2400, 4800, 19200, 38400, 57600 i 115200.

Do projektu wybrana została prędkość: 19200 bps (bits per second).

- **Algorytm funkcji skrótu SHA-256**

Funkcja skrótu (funkcja haszująca) jest wykorzystywana do przyporządkowania dowolnie dużej liczbie klucza (hash-a) o stałym rozmiarze. Hash SHA-256 ma długość 256 bitów.

Realizacja algorytmu składa się z dwóch etapów:

1. Przygotowania.
2. Obliczania hash-a.

Etap 1.

Dopełnienie wiadomości,  $M$ , do wielokrotności 512 bitów:

- Dopisanie bitu „1” na końcu wiadomości.
- Dopisanie  $k$  bitów „0”, gdzie  $k$  jest najmniejszym dodatnim rozwiązaniem równania:

$$\begin{aligned} l + 1 + k &\equiv 448 \mod 512 \\ \Downarrow \\ k &= (448 - (l + 1)) \mod 512 \end{aligned}$$

- Dopisanie 64-bitowego bloku równemu długości wiadomości,  $l$ , w systemie binarnym.

Podział na  $N$  512-bitowych bloków,  $M^{(1)}, M^{(2)}, \dots, M^{(N)}$ . Ponadto w każdej iteracji algorytmu blok jest dzielony na szesnaście 32-bitowych słów:  $M_0^{(i)}, M_1^{(i)}, \dots, M_{15}^{(i)}$ .

Przypisanie początkowej wartości hash-owi,  $H^{(0)}$ . Składa się on z ośmiu 32-bitowych słów uzyskanych jako pierwsze 32 bity części ułamkowej pierwiastka stopnia drugiego z ośmiu kolejnych liczb pierwszych. W zapisie heksadecymalnym:

$$\begin{aligned} H_0^{(0)} &= 6a09e667 \\ H_1^{(0)} &= bb67ae85 \\ H_2^{(0)} &= 3c6ef372 \\ H_3^{(0)} &= a54ff53a \\ H_4^{(0)} &= 510e527f \\ H_5^{(0)} &= 9b05688c \\ H_6^{(0)} &= 1f83d9ab \\ H_7^{(0)} &= 5be0cd19 \end{aligned}$$

Etap 2.

Do obliczeń algorytm wykorzystuje następujące funkcje:

- $Ch(x, y, z) = (x \text{ and } y) \text{ xor } ((\text{not } x) \text{ and } z)$
- $Maj(x, y, z) = (x \text{ and } y) \text{ xor } (x \text{ and } z) \text{ xor } (y \text{ and } z)$
- $\Sigma_0(x) = ROTR_2(x) \text{ xor } ROTR_{13}(x) \text{ xor } ROTR_{22}(x)$
- $\Sigma_1(x) = ROTR_6(x) \text{ xor } ROTR_{11}(x) \text{ xor } ROTR_{25}(x)$
- $\sigma_0(x) = ROTR_7(x) \text{ xor } ROTR_{18}(x) \text{ xor } SHR_3(x)$
- $\sigma_1(x) = ROTR_{17}(x) \text{ xor } ROTR_{19}(x) \text{ xor } SHR_{10}(x)$
- $ROTR_n(x) = (x \gg n) \text{ or } (x \ll (w - n)),$   
 $w - \text{długość słowa } x \text{ w bitach (tutaj 32 - bity)}$
- $SHR_n(x) = x \gg n$

Każda operacja dodawania (+) przedstawiona w algorytmie jest wykonywana modulo  $2^{32}$ .

Zasoby wymagane przez algorytm w każdej iteracji:

1. Tablica 64 32-bitowych słów:  $W_0, W_1, \dots, W_{63}$ .
2. Osiem 32-bitowych zmiennych:  $a, b, \dots, h$ .
3. Hash składający się z 8 32-bitowych słów:  $H_0^{(i)}, H_1^{(i)}, \dots, H_7^{(i)}$ .
4. Dwie 32-bitowe zmienne:  $T_1$  i  $T_2$ .

W trakcie obliczeń wykorzystywane są również stałe,  $K_0, K_1, \dots, K_{63}$ , w zapisie heksadecymalnym:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

Uzyskuje się je z części ułamkowej pierwiastka stopnia trzeciego kolejnych liczb pierwszych zachowując pierwsze 32 bity.

Obliczanie hash-a przedstawia się następująco:

Pętla dla każdego bloku wiadomości,  $M^{(1)}, M^{(2)}, \dots, M^{(N)}$ , od  $i = 1$  do  $N$ :

{

1. Przygotowanie tablicy słów  $W_t$ :

$$W_t = \begin{cases} M_t^{(i)}, & \text{dla } 0 \leq t \leq 15 \\ \sigma_1(W_{t-2}) + W_{t-7} + \sigma_0(W_{t-15}) + W_{t-16}, & \text{dla } 16 \leq t \leq 63 \end{cases}$$

2. Zainicjowanie zmiennych:

$$a = H_0^{(i-1)}$$

$$b = H_1^{(i-1)}$$

$\vdots$

$$h = H_7^{(i-1)}$$

3. Pętla od  $t = 0$  do 63:

{

$$T_1 = h + \Sigma_1(e) + Ch(e, f, g) + K_t + W_t$$

$$T_2 = \Sigma_2(a) + Maj(a, b, c)$$

$$h = g$$

$$g = f$$

$$f = e$$

$$e = d + T_1$$

$$d = c$$

$$c = b$$

$$b = a$$

$$a = T_1 + T_2$$

}

4. Obliczenie  $i$ -tej wartości pośredniej hash-a:

$$H_0^{(i)} = a + H_0^{(i-1)}$$

$$H_1^{(i)} = b + H_1^{(i-1)}$$

$\vdots$

$$H_7^{(i)} = h + H_7^{(i-1)}$$

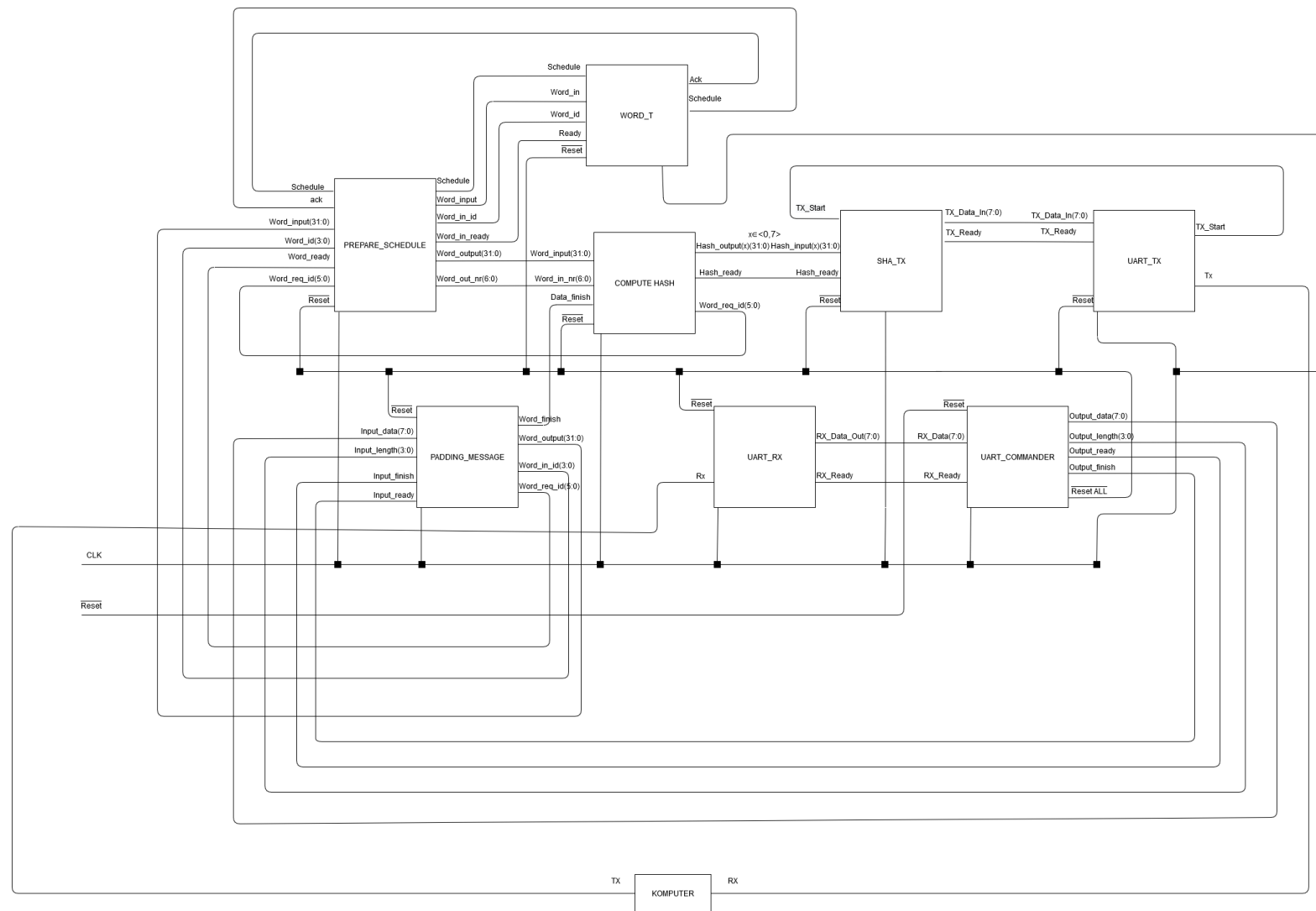
}

Po wykonaniu wszystkich iteracji  $N$  razy, otrzymany hash prezentuje się następująco:

$$H_0^{(N)}, H_1^{(N)}, H_2^{(N)}, H_3^{(N)}, H_4^{(N)}, H_5^{(N)}, H_6^{(N)}, H_7^{(N)}$$

Przedstawiono algorytm na podstawie: FIPS PUB 180-4

## Schemat blokowy



Rys. 2 Schemat blokowy uzyskanego układu, po prawej stronie każdego bloku znajdują się jego wyjścia, natomiast po lewej jego wejścia.

## Opis bloków

- MAIN: Główny blok łączący moduły i przesyłający między nimi sygnały.
- UART\_RX: Odbiornik UART wczytujący dane z zewnątrz.
- UART\_COMMANDER: Nadzoruje blok UART\_RX oraz interpretuje przychodzące dane. Rozróżnia komendy i przesyła dane do kolejnych bloków.
- SHA\_PADDING MESSAGE: Pobiera dane, składa je w 32-bitowe słowa i przekazuje dalej. Na końcu danych dodaje odpowiednie ciągi bitów wyrównujące treść do 512 bitów.
- SHA\_PREPARE\_SCHEDULE: Indeksuje przychodzące słowa, podaje je do wewnętrznego modułu WORD\_T i odbiera od niego wyliczone słowa. Obsługuje moduł SHA\_COMPUTE\_HASH podając mu słowa o wcześniej zadanym indeksie.
- WORD\_T: Tworzy tablicę słów z przychodzących danych i wylicza na ich podstawie kolejne. Informuje o liczbie wyliczonych i gotowych do odczytania słów.
- SHA\_COMPUTE\_HASH: Pobiera słowa o zadanym indeksie od poprzedniego modułu oraz realizuje główną część algorytmu, który oblicza funkcje skrótu (hash-e). Na koniec informuje czy obliczony hash jest gotowy do wysłania
- SHA\_TX: Blok odpowiadający za podział hash-a (zajmującego 256 bitów) po osiem bitów w celu przygotowania do wysłania przez UART (wykonano)
- UART\_TX: Nadajnik (TX) UART (zgodnie z opisem znajdującym się w zakładce „Interfejs komunikacyjny”) (wykonano)

## Opis pozostałych plików

- sha\_function: Funkcje wykorzystywane przez koder SHA-256 znajdujące się w pliku package
- constants: Deklaracja stałych wykorzystywanych przez koder SHA-256, się w pliku package

## Opis sygnałów

Clk – zegar o częstotliwości 12 MHz

Reset – reset bloków

Tx – transmisja do komputera

Rx – otrzymywanie danych

## Sha\_Tx – Uart\_Tx

TX\_Ready - określa czy nowe dane czekają na transmisję

TX\_Start – określa czy transmisja dotychczasowych danych została już zakończona

TX\_Data\_in – dane przeznaczone do transmisji

## **COMPUTE\_HASH - SHA\_TX**

Hash\_output – gotowy hash przeznaczony do transmisji

Hash\_ready – określa czy hash jest gotowy do transmisji

## **COMPUTE\_HASH - PREPARE\_SCHEDULE**

Word\_output(31:0) (wyjściowy PREPARE\_SCHEDULE), Word\_input(31:0) (wejściowy COMPUTE\_HASH) – trzydziestu dwu bitowe słowo

Word\_out\_nr(6:0) (wyjściowy PREPARE\_SCHEDULE), Word\_in\_nr(6:0) (wejściowy COMPUTE\_HASH) – określa ilość słów

Word\_req\_id(5:0) (wyjściowy COMPUTE\_HASH), Word\_req\_id(5:0) (wejściowy PREPARE\_SCHEDULE) - identyfikator żadanego słowa

## **WORD\_T - PREPARE\_SCHEDULE**

Schedule – rozkład ze wszystkimi słowami

Word\_input (wyjściowy PREPARE\_SCHEDULE), Word\_in (wejściowy WORD\_T) - trzydziestu dwu bitowe słowo

Word\_in\_id (wyjściowy PREPARE\_SCHEDULE), Word\_id (wejściowy WORD\_T) - identyfikator żadanego słowa

Ack – licznik gotowych słów

## **PREPARE\_SCHEDULE - PADDING\_MESSAGE**

Word\_input (wejściowy PREPARE\_SCHEDULE), Word\_output(31:0) (wyjściowy PADDING\_MESSAGE) - trzydziestu dwu bitowe słowo

Word\_in\_id (wyjściowy PADDING\_MESSAGE), Word\_id (wejściowy PREPARE\_SCHEDULE) - identyfikator żadanego słowa

Word\_req\_id(5:0) (wyjściowy PADDING\_MESSAGE), Word\_req\_id(5:0) (wejściowy PREPARE\_SCHEDULE) - identyfikator żadanego słowa

### **PADDING\_MESSAGE - UART\_COMMANDER**

Output\_data(7:0) (wyjście UART\_COMMANDER), Input\_data(7:0) (wejście PADDING\_MESSAGE) -

otrzymane dane do obliczenia hasha

Output\_length(3:0) (wyjście UART\_COMMANDER), Input\_length(3:0) (wejście PADDING\_MESSAGE) -

- długość danych liczona od najwyższego bitu

Output\_ready (wyjście UART\_COMMANDER), Input\_finish (wejście PADDING\_MESSAGE) – określa czy dane są gotowe do odczytu

Output\_finish (wyjście UART\_COMMANDER), Input\_ready (wejście PADDING\_MESSAGE) – określa czy wszystkie dane zostały przesłane do obliczania skrótu

### **UART\_COMMANDER - UART\_RX**

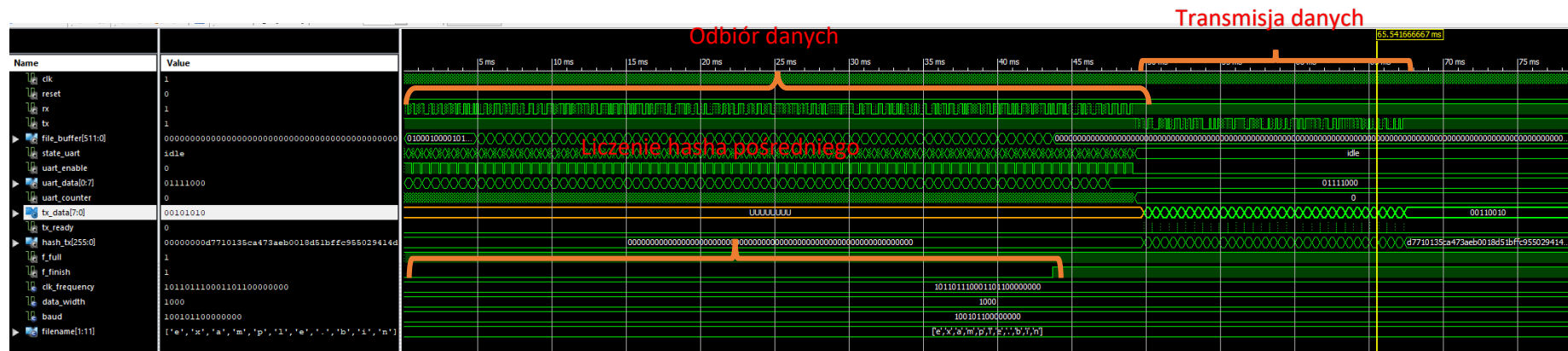
RX\_Data(7:0) (wejście UART\_COMMANDER), RX\_Data\_Out(7:0) (wyjście UART\_RX) -

otrzymane dane

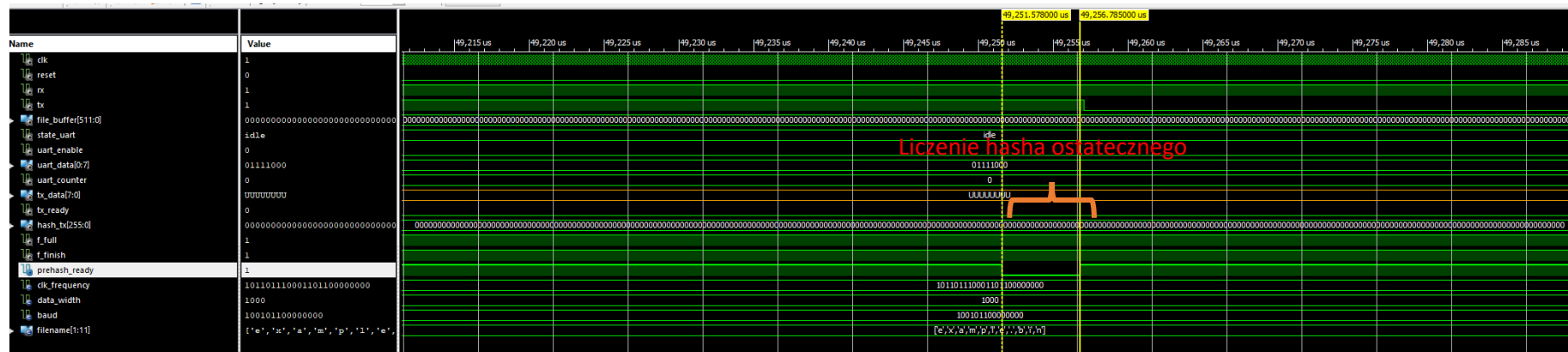
RX\_Ready (wejście UART\_COMMANDER), RX\_Ready (wyjście UART\_RX) - określa, czy dane są gotowe



## Wyniki symulacji



Rys. 3 Wynik symulacji układu



Rys. 4 Wynik symulacji układu

Czas trwania poszczególnych operacji:

Odbiór danych „RX” - 50 ms

Transmisja „TX” – 17,95 ms

Liczenie hasha pośredniego – 43,66 ms

Liczenie hasha ostatecznego – 5,207  $\mu$ s

Czas liczenia hasha pośredniego jest taki długi, ponieważ układ musi czekać aż powolny „RX” dostarczy dane do obliczania. Gdy całość danych zostanie odebrana, liczony jest hash ostateczny i tutaj widać szybkość układu liczącego funkcję skrótu. Zastosowanie szybszego interfejsu komunikacyjnego spowodowałoby znaczne przyspieszenie układu. Czas działania układu od początku odbierania danych do końca transmitowania danych trwa 67,95 ms, więc procesy komunikacyjne zajmują przeważającą większość czasu.

state\_uart - określa stan procesu Uart

file\_buffer - bufor do przechowywania fragmentów pliku wejściowego

uart\_enable - blokuje „wypychanie” danych do UART

uart\_data - przechowuje dane do transmisji

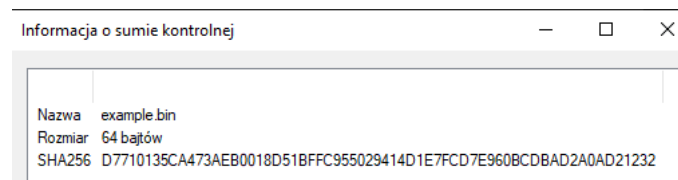
uart\_counter - zlicza przesłane bity

f\_full - flaga określająca, czy plik\_bufor jest pełny lub zakończony

f\_finish - flaga określająca, czy wszystkie dane zostały wysłane do modułu MAIN

prehash\_ready – określa czy hash jest w trakcie liczenia , jeżeli 0 jest liczony

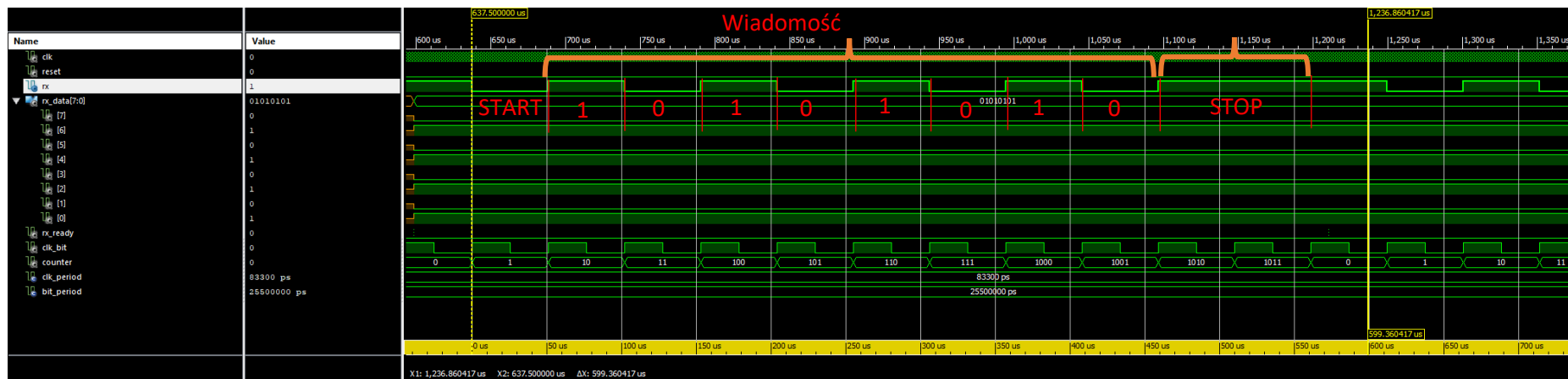
hash\_tx – bufor na wczytane dane z transmisji TX, za pomocą RX



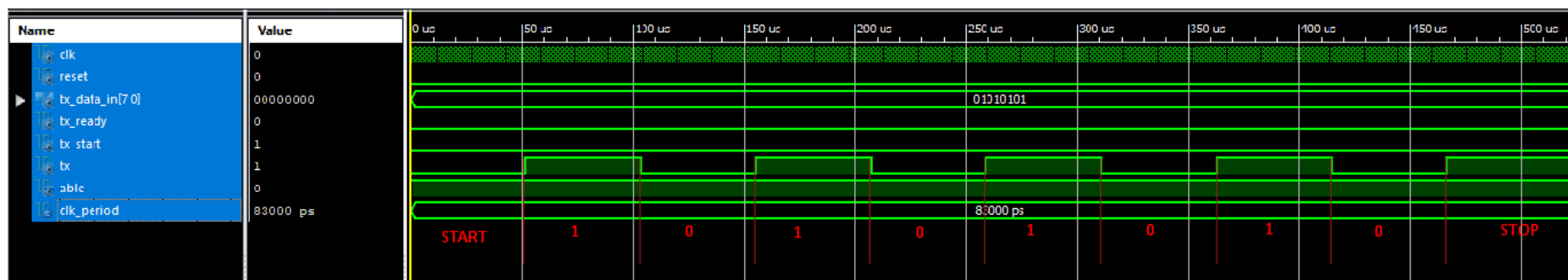
Rys. 5 Prawidłowy wynik kodowania

Z przeprowadzonego badania wynika, że układ działa prawidłowo „RX” poprawnie wczytuje informacje, a na „TX” uzyskujemy zgodny z oczekiwaniami wynik kodowania.

## Dodatkowe kryteria

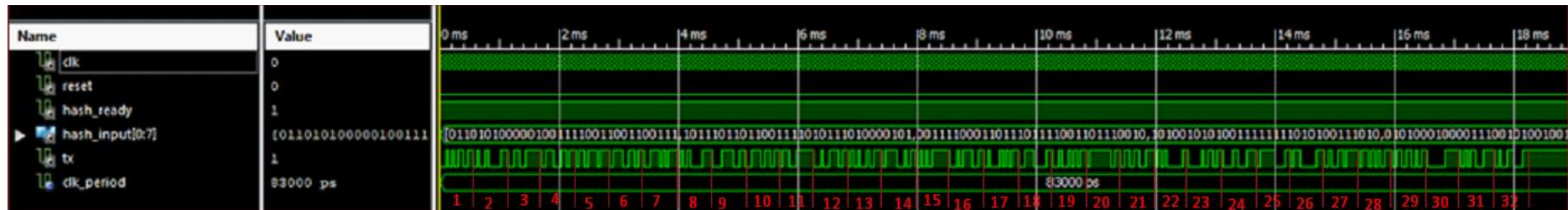


Rys. 6 Symulacja UART RX



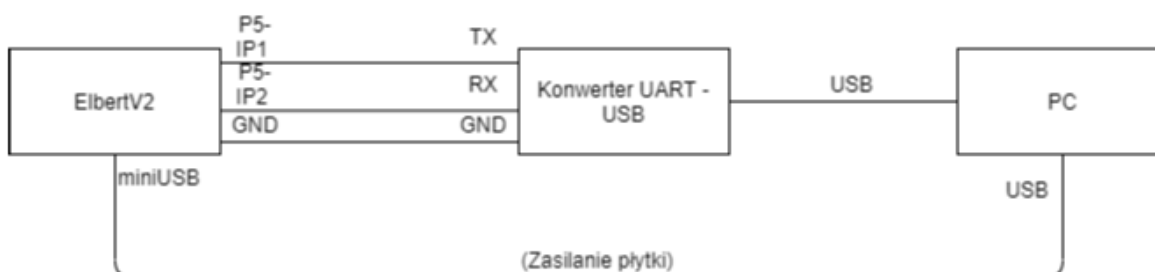
Rys. 7 Symulacja UART TX

Na magistrali UART szeregowo jest transmitowana wartość 170 (tx\_data\_in 10101010), na magistrali równoległej poprawnie otrzymano wysłaną wartość (tx). Czas nadawania jednego bitu wynosi 52  $\mu$ s, więc osiągnięto zakładany Baud rate 19200.

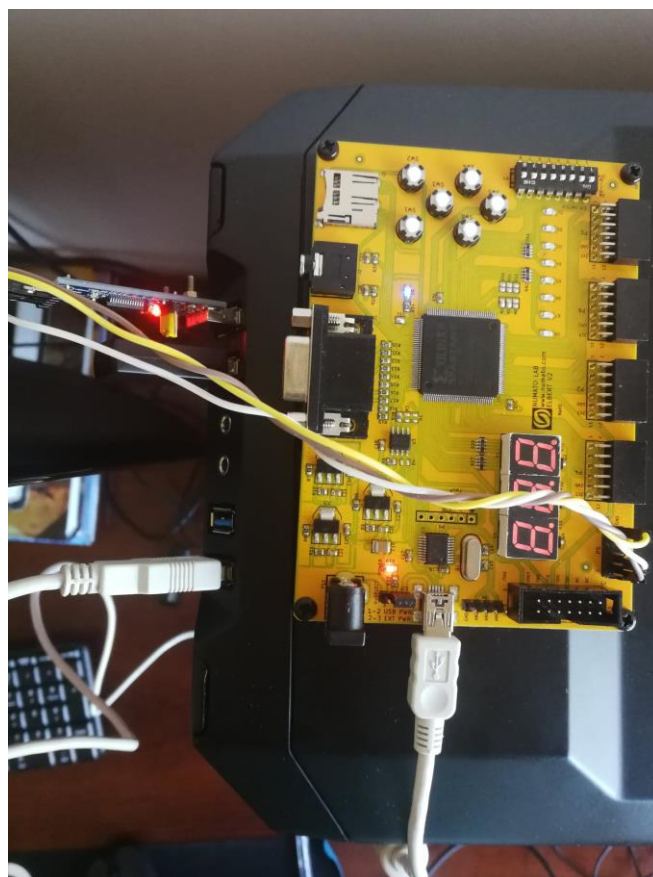


## Echo UART

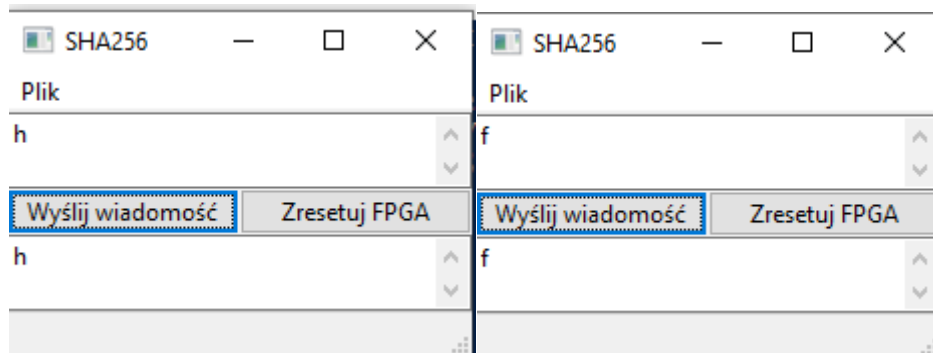
Podstawowy układ (układ „funkcji skrótu sha256”) okazał się zbyt wymagający dla płytki rozwojowej „FPGA Elbert v2 - Spartan 3A”. Aby zademonstrować działanie płytki rozwojowej FPGA, skonfigurowano dodatkowy układ który pobiera dane a następnie zwraca odrzyna wiadomość (przez UART).



Rys. 9 Schemat podłączenia płytki do PC, opis pinów (np. P5-IP1) jest zgodny z opisem znajdującym się na stronie <http://productdata.numato.com/assets/downloads/fpga/elbertv2/ElbertV2Sch.pdf>



Rys. 10 Badana płytka rozwojowa FPGA podłączona do komputera



*Rys. 11 Przykładowe uzyskane wyniki, górne pole tekstowe - wiadomości przeznaczonej do wysłania, dolne pole tekstowe – odszywana wiadomość zwrotna*

Z przeprowadzonego badania wynika, że układ działa prawidłowo, wiadomość wysłana z PC otrzymujemy z powrotem na wejściu komputera. Dane są transmitowane za pomocą interfejsu UART z zakładaną częstotliwością, 19200 Hz (na komputerze został ustawiony Baud rate 19200).

## **Finalne rezultaty**

Wykonano:

1. Odbieranie danych do zakodowania poprzez UART.
2. Przetwarzanie otrzymywanych danych (obliczanie hash-a).
3. Wysyłanie wyliczonych hash-y poprzez UART.
4. Symulacja całego układu.

Dodatkowe kryteria:

1. Symulacje: „UART\_RX”, „UART\_TX”, „Sha\_Tx”.
2. Zaprojektowanie układu ECHO UART.
3. Skonfigurowanie układu na płytce rozwojowej.

Nie wykonano:

1. Skonfigurowanie układu funkcji skrótu na płytce rozwojowej (układ „funkcji skrótu sha256” okazał się zbyt wymagający dla płytki rozwojowej „FPGA Elbert v2 - Spartan 3A”), zamiast tego skonfigurowanie układu „ECHO UART” na płytce rozwojowej.

## **Złożoności czasowa**

Bloki odpowiadające za obliczanie hash-a są bardzo szybkie, kiedy nie muszą czekać, aż wszystkie dane przeznaczone do zakodowania zostaną dostarczone, swoją pracę wykonują przez 5,207  $\mu$ s. Odbiór i transmisja danych przez UART trwa 67,95 ms, więc komunikacja zajmuje bardzo dużo czasu. Zastosowanie szybszego interfejsu komunikacyjnego (np. Ethernetu) znacznie przyspieszyłoby działanie układu.

## **Złożycie zasobów**

Zużycie zasobów dla układu funkcji skrótu jest zbyt duże dla płytki rozwojowej „FPGA Elbert v2 - Spartan 3A”, powodem tego są duże buforory danych wykorzystywane do liczenia hash-a. Aby zmniejszyć zużycie można wykorzystać pamięć RAM w celu przechowywania wspomnianych buforów.

```

Number of warnings: 10
Logic Utilization:
  Total Number Slice Registers: 3,315 out of 11,776 28%
    Number used as Flip Flops: 69
    Number used as Latches: 3,246
    Number of 4 input LUTs: 20,055 out of 11,776 170% (OVERMAPPED)
Logic Distribution:
  Number of occupied Slices: 10,158 out of 5,888 172% (OVERMAPPED)
    Number of Slices containing only related logic: 10,158 out of 10,158 100%
    Number of Slices containing unrelated logic: 0 out of 10,158 0%
    *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs: 20,219 out of 11,776 171% (OVERMAPPED)
    Number used as logic: 20,055
    Number used as a route-thru: 164

The Slice Logic Distribution report is not meaningful if the design is
over-mapped for a non-slice resource or if Placement fails.

Number of bonded IOBs: 4 out of 372 1%
Number of BUFGMUXs: 11 out of 24 45%
Number of MULT18X18SIOs: 16 out of 20 80%

```

*Rys. 12 Zużycie zasobów pyłki „FPGA Elbert v2 - Spartan 3A”*

Badany układ FPGA posiada zbyt małą ilość komórek LUT, „Slices” (które składają się z komórek LUT i przerzutników) dla badanego układu, aby to naprawić można zastosować zapis do pamięci RAM.

```

Slice Logic Utilization:
  Number of Slice Registers: 127 out of 93,120 1%
    Number used as Flip Flops: 52
    Number used as Latches: 75
    Number used as Latch-thrus: 0
    Number used as AND/OR logics: 0
  Number of Slice LUTs: 721 out of 46,560 1%
    Number used as logic: 719 out of 46,560 1%
      Number using O6 output only: 623
      Number using O5 output only: 16
      Number using O5 and O6: 80
      Number used as ROM: 0
    Number used as Memory: 0 out of 16,720 0%
    Number used exclusively as route-thrus: 2
      Number with same-slice register load: 0
      Number with same-slice carry load: 2
      Number with other load: 0

Slice Logic Distribution:
  Number of occupied Slices: 289 out of 11,640 2%
  Number of LUT Flip Flop pairs used: 731
    Number with an unused Flip Flop: 606 out of 731 82%
    Number with an unused LUT: 10 out of 731 1%
  Number of fully used LUT-FF pairs: 115 out of 731 15%
  Number of unique control sets: 16
  Number of slice register sites lost
  to control set restrictions: 89 out of 93,120 1%

```

*Rys. 13 Zużycie dla „Virtex6”*

Z przedstawionego badania wynika, że dla układów FPGA o większej liczbie zasobów, układ funkcji skrótu działa by bez problemu. Układ „Virtex6” to układy z innej półki cenowej, kosztuje około 20 tys. zł.



Slice Logic Utilization:				
Number of Slice Registers:	137 out of	4,800	2%	
Number used as Flip Flops:	52			
Number used as Latches:	85			
Number used as Latch-thrus:	0			
Number used as AND/OR logics:	0			
Number of Slice LUTs:	755 out of	2,400	31%	
Number used as logic:	749 out of	2,400	31%	
Number using O6 output only:	656			
Number using O5 output only:	16			
Number using O5 and O6:	77			
Number used as ROM:	0			
Number used as Memory:	0 out of	1,200	0%	
Number used exclusively as route-thrus:	6			
Number with same-slice register load:	4			
Number with same-slice carry load:	2			
Number with other load:	0			
Slice Logic Distribution:				
Number of occupied Slices:	283 out of	600	47%	
Number of MUXCYs used:	248 out of	1,200	20%	
Number of LUT Flip Flop pairs used:	758			
Number with an unused Flip Flop:	626 out of	758	82%	
Number with an unused LUT:	3 out of	758	1%	
Number of fully used LUT-FF pairs:	129 out of	758	17%	
Number of unique control sets:	16			
Number of slice register sites lost to control set restrictions:	95 out of	4,800	1%	

*Rys. 13 Zużycie dla „Spartan6”*

Dla układu FPGA Spartan6 (który jest znacznie tańszy od poprzednika) układ funkcji skrótu zadział by bez problemu.

**Podział prac**

	<b>Przemysław Jesinowicz</b>	<b>Eryk Wawrzyn</b>
<b>constants</b>	<b>x</b>	<b>x</b>
<b>elbertv2_pin</b>		<b>x</b>
<b>main</b>	<b>x</b>	<b>x</b>
<b>sha_compute_hash</b>	<b>x</b>	<b>x</b>
<b>sha_function</b>		<b>x</b>
<b>sha_padding_message</b>	<b>x</b>	
<b>sha_prepare_schedule</b>	<b>x</b>	
<b>sha_tx</b>		<b>x</b>
<b>sha_word_t</b>	<b>x</b>	<b>x</b>
<b>test_main</b>	<b>x</b>	
<b>test_sha_tx</b>		<b>x</b>
<b>test_uart_tx</b>		<b>x</b>
<b>uart_commander</b>	<b>x</b>	
<b>uart_rx</b>	<b>x</b>	
<b>uart_tx</b>		<b>x</b>
<b>test_uart_rx</b>	<b>x</b>	
<b>main (ECHO UART)</b>	<b>x</b>	<b>x</b>
<b>Aplikacja na komputer</b>		<b>x</b>
<b>Wgrywanie konfiguracji na płytke</b>	<b>x</b>	<b>x</b>
<b>Dokumentacja końcowa</b>	<b>x</b>	<b>x</b>
<b>Dokumentacja wstępna</b>	<b>x</b>	<b>x</b>

X – oznacza przydział do wykonanego zadania