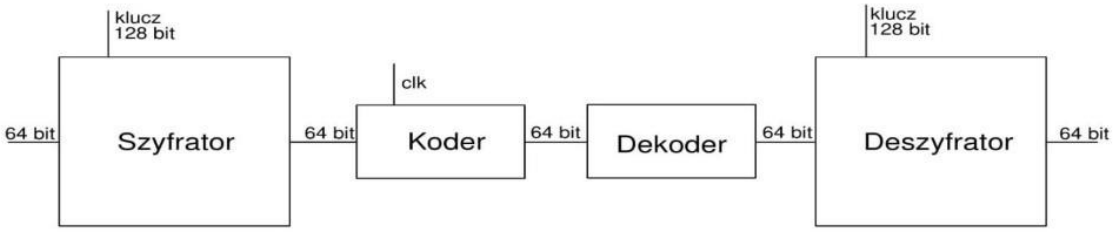
Projekt Szyfratora/Deszyfratora szyfru TEA wraz z Koderem/Dekoderem kodu Hamminga

Grupa projektowa: Michał Ryszka, Wojciech Kowalski, Mateusz Winnicki i Jakub Sokalski

Cel projektu: Zaprojektowanie i symulacja szyfratora/deszyfratora szyfru TEA wraz z koderem/dekoderem kodu Hamminga.



Założenia projektu: Projekt został zrealizowany przy pomocy języka **VHDL** i programu projektowego Quartus Prime 18.1, wcześniej został stworzony model za pomocą języka programowania **Python**.

Opis algorytmów i ich implementacja:

1. Kodowanie Hamminga

Kod Hamminga należy do grupy kodów blokowych, to znaczy, że każdorazowo kodowany jest blok wiadomości o określonej długości, zawierający oprócz bitów informacyjnych także bity kontrolne. W tym przypadku są to bity parzystości, wyliczane z odpowiednich bitów informacyjnych. Po odebraniu takiego bloku danych (dane plus bity kontrolne) możliwa jest jego walidacja pod kątem błędów.

Zalety kodu Hamminga:

- wykrywa i koryguje błędy polegające na przekłamaniu jednego bitu
- może wykrywać błędy podwójne
- dosyć prosty w implementacji

Wady kodu Hamminga:

- do niezawodnej transmisji wymagane odległości Hamminga miedzy słowami transmitowanymi i odbieranymi
- nie pozwala na korygowanie błędów podwójnych

Implementacja kodu Hamminga

W kodzie Hamminga bity kontrolne dodawane są na określonych pozycjach, ogólnie jest to bit pierwszy, a następne bity będą potęgami liczby 2. Dla kodu Hamminga (7,4) liczba bitów informacji wynosi 4 a liczba bitów kontrolnych 3, całkowita długość zakodowanego bloku wynosi 7. Bity kontrolne to bity parzystości wyznaczane dla odpowiednich bitów informacyjnych.

Bit position	1	2	3	4	5	6	7
	P_1	P_2	1	P_4	1	0	0

Bity parzystości wyznaczamy:

$$b_4 = b_5 + b_6 + b_7$$
$$b_2 = b_3 + b_6 + b_7$$
$$b_1 = b_3 + b_5 + b_7$$

Bit parzystości to bit kontrolny, który ma taką wartość aby ciąg złożony z niego i pewnego bitu informacji posiadał parzystą liczbę jedynek.

Przykład:

Wiadomość do zakodowania:
1010

Pozycja bitu:	p_1	p_2	p_3	p_4	p_5	p_6	p_7
wiadomość:			*1*		*0*	*1*	*0*
Bity kontrolne:	*1*	*1*		*1*			
Zakodowana wiadomość:	*1*	*1*	*1*	*1*	*0*	*1*	*0*

7 – całkowita liczba bitów w zadokowanej wiadomości (*n-bits*)
4 – liczba bitów informacyjnych (*k-bits*)
n – k – liczba bitów kontrolnych (*m-bits*)

2. **Szyfrowanie TEA**

Szyfr TEA (*Tiny Encryption Algorithm*) to jeden z najprostszych szyfrów blokowych, czyli takich które przekształcają informację w blokach o stałym rozmiarze na bloki zaszyfrowane. Cechuje się małą zajętością pamięci, dużą szybkością szyfrowania i wysoką odpornością na kryptoanalizę różnicową (jak do tej pory nie zarejestrowano udanej kryptoanalizy kodu). Przyjmuje on **dwie 32-bitowe liczby** i szyfruje je za pomocą **128-bitowego klucza** podzielonego na cztery części. Szyfrowanie polega na **32-krotnym** powtórzeniu następującego schematu:

- do sumy o początkowej wartości 0 dodaję się stałą równą 2654435769
- do pierwszej liczby dodaje się sumę modulo 2 trzech liczb:
    - sumy pierwszej części klucza i liczby drugiej przesuniętej bitowo o cztery w lewo
    - sumy liczby drugiej i stałej
    - sumy drugiej części klucza i liczby drugiej przesuniętej bitowo o pięć w prawo
- do drugiej liczby dodaje się sumę modulo 2 trzech liczb:
    - sumy trzeciej części klucza i liczby pierwszej przesuniętej bitowo o cztery w lewo
    - sumy liczby pierwszej i stałej
    - sumy czwartej części klucza i liczby pierwszej przesuniętej bitowo o pięć w prawo

Po wykonaniu wszystkich powtórzeń wiadomość jest zaszyfrowana i aby ją odszyfrować, należy postępować według podobnych kroków z odwróconą kolejnością schematu i zamiast dodawania do liczb i sumy, odejmuje się od nich.

Realizacja:

1. Python

Napisaliśmy program w Pythonie, przedstawiający działanie projektu. Składa się on z 5 plików .py

- **projekt_UCYF.py** - główny plik łączący ze sobą pliki zawierające szyfrator, koder, deszyfrator i dekoder.
- **koder.py** - plik realizujący koder za pomocą kodu Hamminga
- **dekoder.py** - plik realizujący dekoder za pomocą kodu Hamminga.
- **szyfrator.py** - plik realizujący szyfrator TEA
- **deszyfrator.py** - plik realizujący deszyfrator TEA

Na początku następuje kodowanie każdej litery na kod ASCII.

```
Prosze wprowadzic slowo:
ok
Nasze slowo w kodzie ASCII (decymalnie)  [111, 107]
```

Szyfrujemy parami kolejne litery. W tym przypadku jedyną parą będzie [111,107].

```
Rozpoczynam przetwarzanie ciagu:  [111, 107]


Wiadomosc po zaszyfrowaniu (w systemie dziesietnym)  [3584389021, 1778541706]
```

Algorytm szyfrujący i deszyfrujący do działania wymaga dwóch 32 bitowych wiadomości i 128 bitowego klucza. Szyfrowanie i deszyfrowanie polega na wykonaniu 32 powtórzeń w których na wektorach z danymi wykonywane są operacje dodawanie, przesunięcia bitowego i operacja XOR.

```
def szyfrator(v, k):
    y = c_uint32(v[0])
    z = c_uint32(v[1])
    sum = c_uint32(0)
    delta = 0x9e3779b9
    n = 32
    w = [0,0]

    while(n>0):

        sum.value += delta
        y.value += ( z.value << 4 ) + k[0] ^ z.value + sum.value ^ ( z.value >> 5 ) + k[1]
        z.value += ( y.value << 4 ) + k[2] ^ y.value + sum.value ^ ( y.value >> 5 ) + k[3]
        n -= 1

    w[0] = y.value
    w[1] = z.value
    print("Wiadomosc po zaszyfrowaniu (w systemie dziesietnym) ",w)

    return w
```

Realizacja kodu Hamminga

Kod Hamminga zrealizowany na tablicach. Najpierw wyliczamy ilość i pozycje bitów kontrolnych, następnie ich wartości i wstawiamy je w odpowiednie miejsca.

```
Tablica wstepna kodu Haminnga:  ['x1', 'x2', 1, 'x4', 1, 0, 1, 'x8', 0, 1, 0, 1, 1, 0, 1, 'x16', 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 'x32', 0, 1, 1, 1, 0, 1]
Obliczone bity kontrolne ulozone x1...xn:  [1, 0, 0, 0, 1, 0]
Po zakodowaniu:  [1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1]
```

```
Tablica wstepna kodu Haminnga:  ['x1', 'x2', 0, 'x4', 1, 1, 0, 'x8', 1, 0, 1, 0, 0, 0, 0, 'x16', 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 'x32', 0, 0, 1, 0, 1, 0]
Obliczone bity kontrolne ulozone x1...xn:  [1, 0, 0, 1, 1, 0]
Po zakodowaniu:  [1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0]
```

```
Wyraz po zakodowaniu:  [[1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1], [1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0]]


Slowo w bin po odkodowaniu:  [[1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1], [0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0]]
```

```python
# Główna czesc programu, Etap I wyznaczenie pozycji i ilosci bitow kontrolnych, Etap II obliczenie ich wartosci i wartosci bitu parzystosci
#Etap I
def dodaj_bity_kontrolne(tab_liter):
      wyraz = []
      for litery in tab_liter:
          slowo = litery

          x=int(math.log(len(slowo),2))+1          #wyliczam ilosc bitów kontrolnych
          tab_wstepna = []
          dlugosc = len(slowo)+x
          j=0
          z=1
          for i in range(1,dlugosc+1):
             flaga = 1
             z=1
             while z <= int(math.pow(2,x)):
               if i == z:
                    tab_wstepna.append(str('x'+str(z)))
                    flaga = 0
               z= 2*z
             if flaga:
                tab_wstepna.append(slowo[j])
                j += 1

          wyraz.append(tab_wstepna)


      return wyraz
```

```python
#Etap II

def kod_hamminga(wyraz):                          #Funkcja przyjumje całe wyrazy lub pojedyncze litery, cyfry i znaki w ASCII w formacie [[bit,bit,...,bit],[bit,bit,...,bit]...[bit,bit,...,bit]]
    tmp = dodaj_bity_kontrolne(wyraz)
    print("\n")
    kod = []
    it2=0                                         #Do testow
    for tab_wstepna in tmp:
        tab_pozycji = []                          #Tablica zawiera nr miejsca w którym znajdujacy się bit i ma wartosc 1
        bity_kontrolne = []
        tmp = []
        j=1
        for i in tab_wstepna:                     #W tej petli szukam w "wyrazie Haminnga" bitow o wartosci 1, do wyliczenia wartosci bitow kontrolnych
            if i == 0:
                tab_pozycji.append(j)
            j += 1

        liczba_x = len(tab_wstepna) - (tab_wstepna.count(1)+tab_wstepna.count(0))       #Ilosc bitow kontrolnych poki co w postaci X1..Xn

        for i in range(0,liczba_x):               #W tej pentli dla kazdego X1..Xn tworze tablice z wartosciami
            tmp.append(0)
            for j in tab_pozycji:

                tmp[i] = dec_bin(int(j))[len(dec_bin(int(j)))-i-1]+tmp[(i)]
```

```python
    for i in tmp:                                 #Tu wyliczam wartosc bitu kontrolnego
        if int(i)%2 == 0:
            bity_kontrolne.append(0)
        else:
            bity_kontrolne.append(1)

    it=0
    suma=0
    print("Tablica wstepna kodu Haminnga: ",tab_wstepna)                  # do testów
    print( "Obliczone bity kontrolne ulozone x1...xn: ",bity_kontrolne)   # do testow

    for i in range(0,len(tab_wstepna)):           #Przypisanie bitów kontrolnych do odpowienich miejsc
        if not(tab_wstepna[i] == 0 or tab_wstepna[i] == 1):
            tab_wstepna[i] = bity_kontrolne[it]
            it += 1


    kod.append(tab_wstepna)
    print("Po zakodowaniu: ",kod[it2])
    print("\n")
    it2 += 1
print("Wyraz po zakodowaniu: ",kod)
return kod
```

Odkodowywanie dekoderem polega wykryciu czy nie nastąpiło przekłamanie bitu, ewentualna poprawa wartości bitu jeśli jest przekłamany bit i usunięciu bitów kontrolnych.

Odszyfrowanie wiadomości na kod ASCII

```
Wiadomosc po odszyfrowaniu (w systemie dziesietnym)  [111, 107]
```

```
def deszyfrator(v, k):
    y = c_uint32(v[0])
    z = c_uint32(v[1])
    sum = c_uint32(0xc6ef3720)
    delta = 0x9e3779b9
    n = 32
    w = [0,0]

    while(n>0):
        z.value -= ( y.value << 4 ) + k[2] ^ y.value + sum.value ^ ( y.value >> 5 ) + k[3]
        y.value -= ( z.value << 4 ) + k[0] ^ z.value + sum.value ^ ( z.value >> 5 ) + k[1]
        sum.value -= delta
        n -= 1

    w[0] = y.value
    w[1] = z.value
    print("Wiadomosc po odszyfrowaniu (w systemie dziesietnym) ",w)
    print("\n")
    return w
```

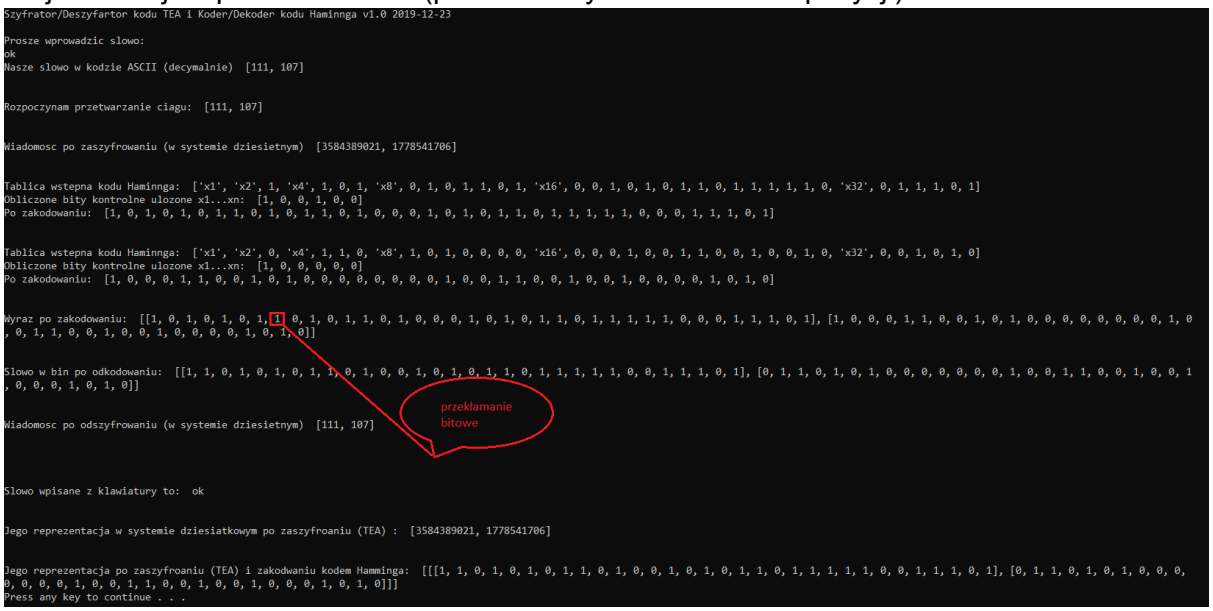
```
Slowo wpisane z klawiatury to:  ok


Jego reprezentacja w systemie dziesiatkowym po zaszyfroaniu (TEA) :  [3584389021, 1778541706]


Jego reprezentacja po zaszyfroaniu (TEA) i zakodwaniu kodem Hamminga:  [[[1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1], [0, 1, 1, 0, 1, 0, 1, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0]]]
```

Wynik działania programu

```
Szyfrator/Deszyfartor kodu TEA i Koder/Dekoder kodu Haminnga v1.0 2019-12-23

Prosze wprowadzic slowo:
ok
Nasze slowo w kodzie ASCII (decymalnie)  [111, 107]


Rozpoczynam przetwarzanie ciagu:  [111, 107]


Wiadomosc po zaszyfrowaniu (w systemie dziesietnym)  [3584389021, 1778541706]


Tablica wstepna kodu Haminnga:  ['x1', 'x2', 1, 'x4', 1, 0, 1, 'x8', 0, 1, 0, 1, 1, 0, 1, 'x16', 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 'x32', 0, 1, 1, 1, 0, 1]
Obliczone bity kontrolne ulozone x1...xn:  [1, 0, 0, 0, 1, 0]
Po zakodowaniu:  [1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1]


Tablica wstepna kodu Haminnga:  ['x1', 'x2', 0, 'x4', 1, 1, 0, 'x8', 1, 0, 1, 0, 0, 0, 0, 'x16', 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 'x32', 0, 0, 1, 0, 1, 0]
Obliczone bity kontrolne ulozone x1...xn:  [1, 0, 0, 1, 1, 0]
Po zakodowaniu:  [1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0]


Wyraz po zakodowaniu:  [[1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1], [1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0
, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0]]


Slowo w bin po odkodowaniu:  [[1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1], [0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1
, 0, 0, 0, 1, 0, 1, 0]]


Wiadomosc po odszyfrowaniu (w systemie dziesietnym)  [111, 107]



Slowo wpisane z klawiatury to:  ok


Jego reprezentacja w systemie dziesiatkowym po zaszyfroaniu (TEA) :  [3584389021, 1778541706]


Jego reprezentacja po zaszyfroaniu (TEA) i zakodwaniu kodem Hamminga:  [[[1, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1], [0, 1, 1, 0, 1, 0, 1, 0, 0, 0,
0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0]]]
Press any key to continue . . .
```

Warto zwrócić uwagę na fakt, że po wprowadzeniu sztucznego przekłamania bitowego przy zakodowaniu, odkodowana wiadomość pozostaje taka sama. Jest to niezaprzeczalna zaleta kodu Hamminga. (przy założeniu że przekłamaniu uległ jeden bit!)

To jest wersja z przekłamaniem (przekłamany został bit na 8. pozycji)



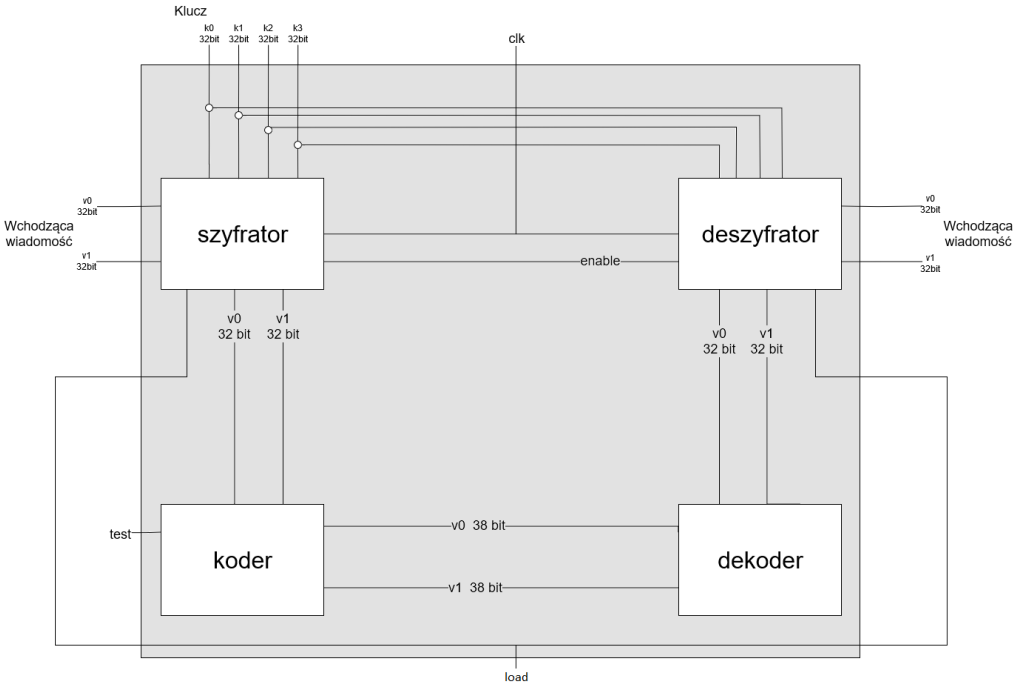
Zasymulowaliśmy sytuację w której jeden bit zostaje przekłamany, symulacja zakończyła się sukcesem dekoder z powodzeniem znalazł błąd i go naprawił. Na wyjściu otrzymaliśmy poprawnie odkodowaną wiadomość.

2. VHDL

W programie Quartus nasz projekt składa się z pięciu plików .vhd. Są to:

- **Projekt_UCYF.vhd** - główny plik łączący ze sobą pliki zawierające szyfrator, koder, deszyfrator i dekoder.
- **Hamming_encoder.vhd** - plik realizujący koder za pomocą kodu Hamminga
- **Hamming_decoder.vhd** - plik realizujący dekoder za pomocą kodu Hamminga.
- **szyfrator.vhd** - plik realizujący szyfrator TEA
- **deszyfrator.vhd** - plik realizujący deszyfrator TEA

Rysunek przedstawiający układ w projekcie:

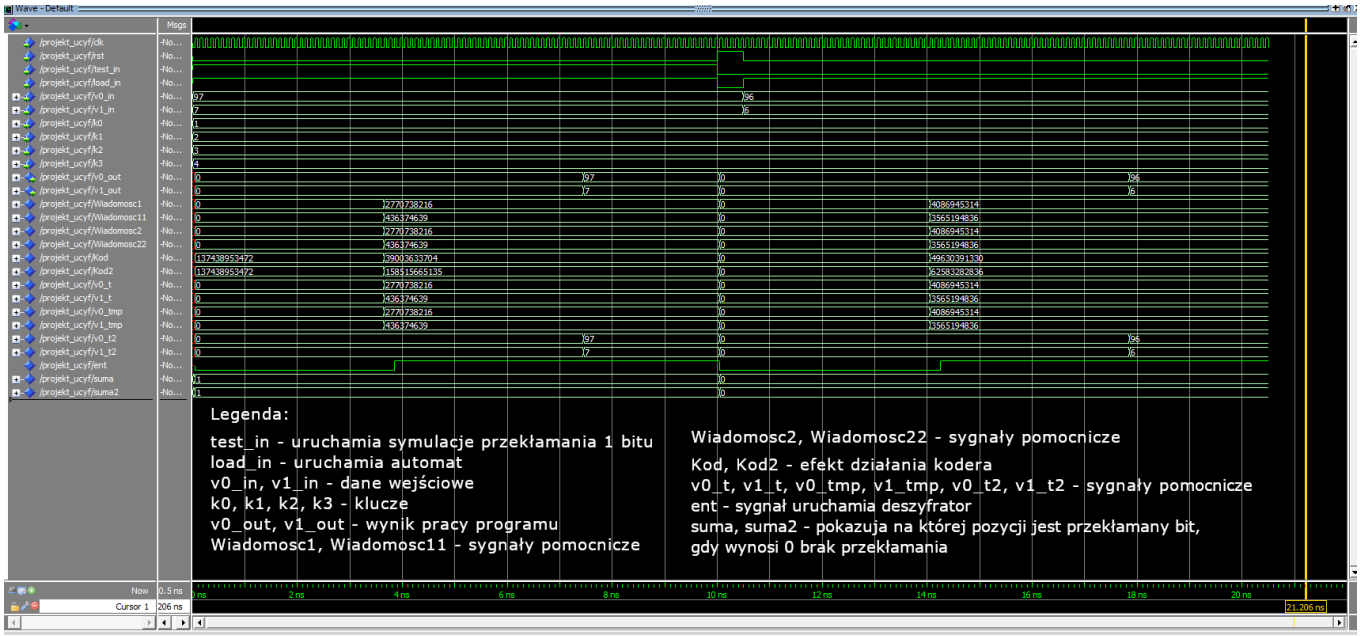


Szyfrator i deszyfrator TEA działają na podobnej zasadzie. Stworzyliśmy automat przyjmujący cztery możliwe stany:

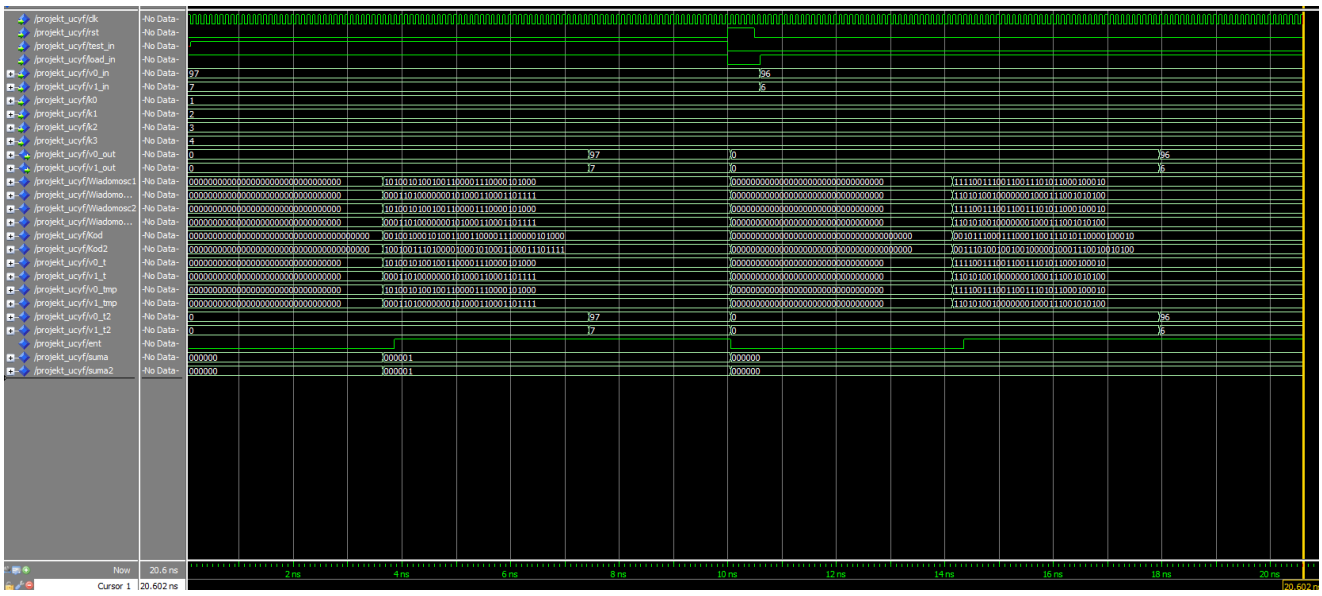
- **start** - stan początkowy, przypisuje startowe wartości zmiennym
- **licz** - stan, w którym wykonuje się jedna z pętli algorytmu TEA, zależnie od licznik powtarza się ten stan, albo przechodzi do stanu **zwroc**
- **zwroc** - stan, w którym przekazuje się zakodowane liczby na zewnątrz szyfratora lub deszyfratora
- **stop** - stan zatrzymania, w który wchodzi automat po zaszyfrowaniu wiadomości lub odszyfrowaniu

Uznaliśmy że najlepszym typem danych dla naszego projektu będzie typ “unsigned” ponieważ pozwala na wykonywanie wszystkich potrzebnych w naszym projekcie operacji

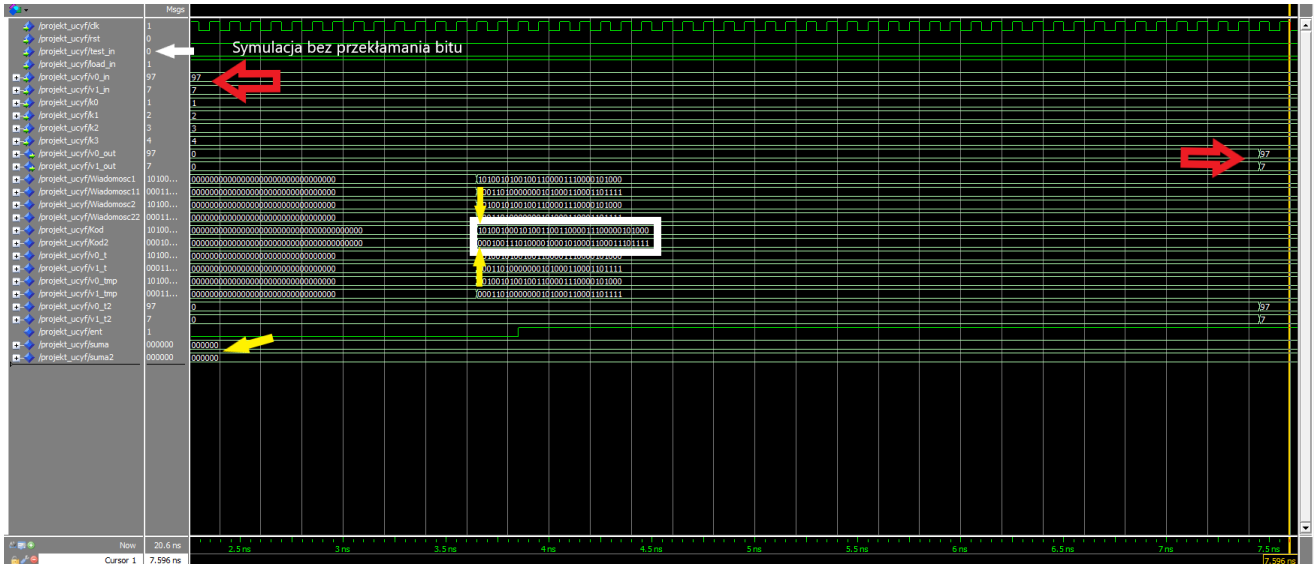
(dodawanie, odejmowanie, XOR i przesunięcia bitowe). Korzystaliśmy również ze zmiennych i pętli for do uzupełniania tablic w koderze i dekoderze. W szyfratorze i deszyfratorze zastosowaliśmy licznik i automat stanów. Nasz projekt łączy zarówno logikę sekwencyjną (szyfrator i deszyfrator) jak i kombinacyjną (dekoder, koder).



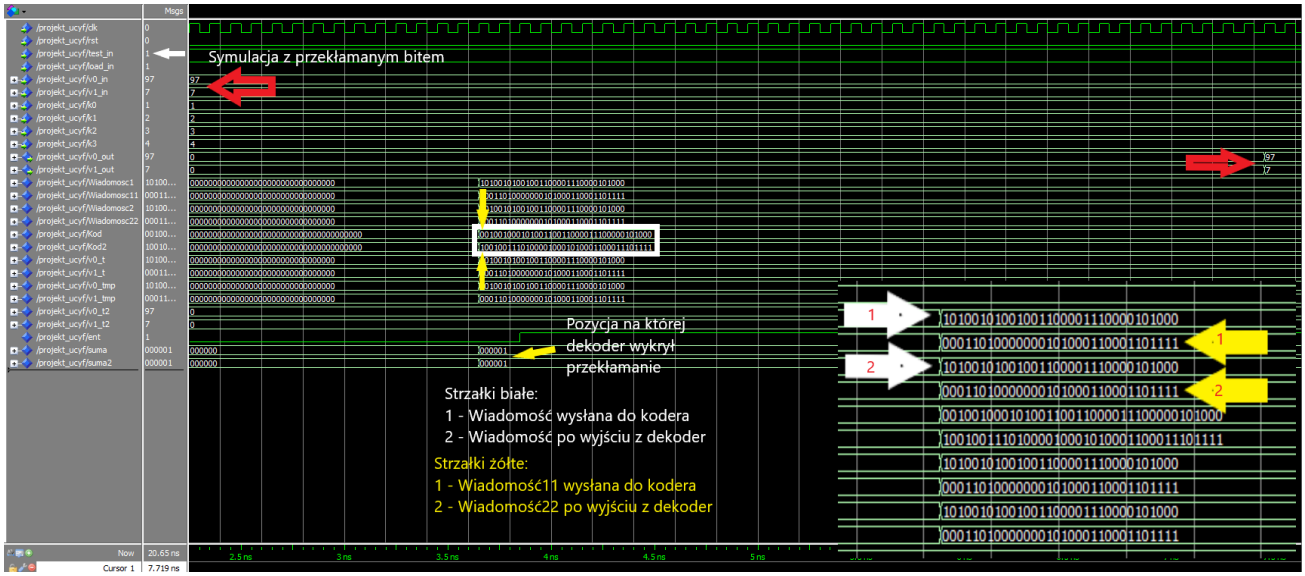
Zrzut ekranu prezentuje symulację wraz z Legendą opisującą poszczególne widoczne sygnały.



Powyższy zrzut prezentuję symulację wykonaną dla dwóch par liczb (97, 7) i dla (96,6). Dla pierwszej pary dodatkowo z testem przekłamania bitu.



Powyższa symulacja została przeprowadzona dla wejściowych danych: 97 i 7, bez symulowania bitu przekłamania. Jak zaznaczono czerwonymi strzałkami wiadomości na wyjściu i wejściu są identyczne. Potwierdza to prawidłowe działanie naszego projektu. Strzałka żółta wskazuje na sygnał który mówi nam na kórym bicie nastąpiło przekłamanie. Gdy wskazuje zero (jak w tym przypadku) oznacza to że nie nastąpiło przekłamanie bitu.



Powyższa symulacja została przeprowadzona dla wejściowych danych: 97 i 7, z symulowaniem bitu przekłamania. Jak zaznaczono czerwonymi strzałkami wiadomości na wyjściu i wejściu są identyczne. Potwierdza to prawidłowe działanie naszego projektu. Strzałka żółta wskazuje na sygnał który mówi nam na kórym bicie nastąpiło przekłamanie. W obecnym przypadku przekłamany jest bit numer 1. Dekoder wykrył tą nieprawidłowość i naprawił ją. Ja wskazują nam białe strzałki z numerami 1 i 2 wiadomość wchodząca do kodera i wiadomość wychodząca z dekodera są identyczne (jest to zaszyfrowana liczba 97), strzałki żółte wskazują to samo tylko dla zaszyfrowanej 7.

Wyniki i wnioski:

Na podstawie przeprowadzonych symulacji zarówno w Pythonie jak i w VHDL'u stwierdzamy że udało się nam poprawnie zaimplementować kodowanie Hamminga i szyfrowanie TEA w naszym projekcie. Nasza sprzętowa implementacja pozwala na szybkie szyfrowanie/deszyfrowanie i kodowanie/dekodowanie wiadomości o wielkości do 64 bitów. Poniżej przedstawiamy podstawowe dane techniczne naszego projektu:

Total logic elements: 1,218 (~1% zasobów)

Total registers: 655

Total pins: 260

F_{max}: 81,71 Mhz

Stosując wzór na przepustowość realizacji iteracyjnej

$$thruput_{iter} = f_{MAX} \cdot \frac{data\ size}{32}$$

otrzymujemy przepustowość szyfratora z koderem wynoszącą 163 Mbit/s. Jest to zadowalający wynik, jednak prawdopodobnie mając większe doświadczenie moglibyśmy zastosować inną realizację co zwiększyłoby z pewnością przepustowość, a być może i zmniejszyło ilość zużywanych przez nasz szyfrator zasobów.

3. Arduino (dodatkowo)

Dodatkowo w ramach nauki obsługi Arduino wykonaliśmy model naszego projektu na tej platformie. Wykorzystaliśmy kod napisany w Pythonie (został dostosowany do Arduino czyli zmieniony na C++). Model pozwala na wybranie cyfry z klawiatury. Następnie szyfruje ją, koduje i wysyłą przez diodę LED sygnał odczytuje fotorezystor. Odebrana wiadomość jest dekodowana, deszyfrowana i wyświetlana na ekranie LCD. Pod linkiem można zobaczyć działanie modelu:
https://www.youtube.com/watch?v=YpyHDQIto3s&t=15s