Politechnika
Wrocławska

Układy Cyfrowe i Systemy Wbudowane 2

Projekt

Temat:
Sprzętowa realizacja szyfratora i deszyfratora DES

Autorzy:

Marcin Klima 226022
Filip Mazur 226018

Prowadzący:
Dr. inż. Jarosław Sugier

Termin zajęć:
Wt. TN, $11^{00} - 14^{00}$

Wstęp

Temat naszego projektu brzmi *Sprzętowa realizacja szyfratora i deszyfratora DES.* Czym zatem jest owy enigmatyczny *DES*?

Data Encrytprion Standard Tak właśnie brzmi rozwinięcie skrótu. Jest to *symetryczny szyfr blokowy* stworzony przez korporację IBM [1]. Żeby w pełni zrozumieć czym jest temat naszego projektu niezbędne jest przedstawienie paru definicji:

Klucz szyfrujący/deszyfrujący jest to ciąg danych służących do szyfrowania wiadomości czytelnej w kryptogram (wiadomość zaszyfrowaną) za pomocą algorytmu szyfrowania. Klucz ten jest odpowiednio ustalany przez nadawcę w fazie szyfrowania. W analogicznie odwrotny sposób działa klucz deszyfrujący

Algorytm Symetryczny (algorytm z pojedynczym kluczem) jest typem algorytmu kryptograficznego, który do szyfrowania i deszyfrowania tekstu jawnego wykorzystuje te same klucze. [2]

Szyfr blokowy jest rodzajem szyfrowania symetrycznego (zdefiniowanego wyżej). Polega na szyfrowaniu bloku wejściowego (np. fragmentu pliku) na podstawie zadanego klucza, przekształcając go na blok wyjściowy o takiej samej długości w taki sposób, że niemożliwe jest odwrócenie tego przekształcenia bez posiadania klucza. [3]

Zasada działania algorytmu

DES jest szyfrem blokowym z blokami o długości 64 bitów. Do szyfrowania i deszyfrowania danych wykorzystywanych jest 56 bitów klucza, który zapisany jest w postaci 64 bitowego ciągu, w którym co 8 bit jest bitem kontrolnym i może służyć do kontroli parzystości. Algorytm szyfrowania danych jest następujący: na początku tekst jawny, który ma zostać zaszyfrowany, dzielony jest na bloki 64-bitowe. Następnie dla każdego bloku wykonywane są następujące operacje:

- dokonywana jest permutacja początkowa bloku przestawiająca bity w pewien określony sposób – nie zwiększa ona bezpieczeństwa algorytmu, a

jej początkowym celem było ułatwienie wprowadzania danych do maszyn szyfrujących używanych w czasach powstania szyfru

- blok wejściowy rozdzielany jest na dwie 32-bitowe części: lewą oraz prawą
- wykonywanych jest 16 cykli tych samych operacji, zwanych funkcjami Feistela, podczas których dane łączone są z kluczem. Operacje te wyglądają następująco:
  - bity klucza są przesuwane, a następnie wybieranych jest 48 z 56 bitów klucza
  - prawa część danych rozszerzana jest do 48-bitów za pomocą permutacji rozszerzonej
  - rozszerzona prawa połowa jest sumowana modulo 2 z wybranymi wcześniej (i przesuniętymi) 48 bitami klucza
  - zsumowane dane dzielone są na osiem 6-bitowych bloków i każdy blok podawany jest na wejście jednego z S-bloków (pierwszy 6-bitowy blok na wejście pierwszego S-bloku, drugi 6-bitowy blok na wejście drugiego S-bloku, itd.). Pierwszy i ostatni bit danych określa wiersz, a pozostałe bity kolumnę S-BOXa. Po wyznaczeniu miejsca w tabeli, odczytuje się wartość i zamienia na zapis dwójkowy. Wynikiem działania każdego S-bloku są 4 bity wyjściowe – tworzą one 32-bitowe wyjście S-bloków. Każdy S-Blok ma inną strukturę
  - wyjście S-bloków poddawane jest permutacji w P-blokach
  - bity tak przekształconego bloku sumowane są z bitami lewej połowy danych
  - tak zmieniony blok staje się nową prawą połową, poprzednia prawa połowa staje się natomiast lewą połową – cykl dobiega końca
- po wykonaniu 16 cykli operacji prawa i lewa połowa danych jest łączona w 64-bitowy blok
- dokonywana jest permutacja końcowa

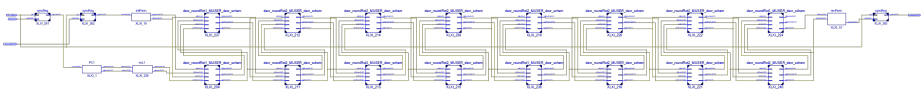
Deszyfrowanie polega na zastosowaniu tych samych operacji w odwrotnej kolejności (różni się od szyfrowania tylko wyborem podkluczy, który teraz odbywa się od końca). Naszym zadaniem projektowym jest napisanie sprzętowej implementacji owej metody szyfrowania.

Projekt

Nasz projekt podzielony został na dwa moduły:

- Szyfrator
- Deszyfrator

Szyfratora



Opis wejść układu

- Wejście zegarowe taktowane 50MHz, oznaczenie na schemacie `Clk_50MHz`
- Tekst jawny, 64-bity, oznaczenie na schemacie `plaintext(0:63)`
- Klucz, 64-bity, oznaczenie na schemacie `key(0:63)`

Opis wyjść układu

- Zaszyfrowana wiadomość, 64-bity, oznaczenie na schemacie `ciphertext(0:63)`

Opis symboli użytych w module

Symbol `syncReg`

```
entity syncReg is
Port ( regIn : in  STD_LOGIC_VECTOR (0 to 63);
clk : in  STD_LOGIC;
regOut : out  STD_LOGIC_VECTOR (0 to 63));
end syncReg;

architecture Behavioral of syncReg is

begin
```

```
process(clk)
begin
if rising_edge(clk) then
regOut <= regIn;
end if;
end process;

end Behavioral;
```

Symbol `initPerm`

```
entity initPerm  is
port(        myinput: in std_logic_vector(0 to 63);
rightoutput: out std_logic_vector(0 to 31);
leftoutput: out std_logic_vector(0 to 31));
end initPerm ;


architecture behavior of initPerm  is

type permArray is array(0 to 63) of integer range 0 to 63;

constant perm: permArray :=
(57,49,41,33,25,17, 9,1,
59,51,43,35,27,19,11,3,
61,53,45,37,29,21,13,5,
63,55,47,39,31,23,15,7,
56,48,40,32,24,16, 8,0,
58,50,42,34,26,18,10,2,
60,52,44,36,28,20,12,4,
62,54,46,38,30,22,14,6);

signal myArray : std_logic_vector(0 to 63);

begin

et0: for i in 0 to 63 generate
myArray( i ) <= myinput( perm( i ) );
end generate;
leftoutput <= myArray(0 to 31);
rightoutput <= myArray(32 to 63);

end behavior;
```

Moduł PC1

```
entity PC1  is
port(        myinput: in std_logic_vector(0 to 63);
rightoutput: out std_logic_vector(0 to 27);
leftoutput: out std_logic_vector(0 to 27));
end PC1 ;


architecture behavior of PC1  is

type permArray is array(0 to 55) of integer range 0 to 63;

constant perm: permArray :=
(56,48,40,32,24,16, 8,
0,57,49,41,33,25,17,
9, 1,58,50,42,34,26,
18,10, 2,59,51,43,35,
62,54,46,38,30,22,14,
6,61,53,45,37,29,21,
13, 5,60,52,44,36,28,
20,12, 4,27,19,11, 3);

signal myArray : std_logic_vector(0 to 55);

begin

et0: for i in 0 to 55 generate
myArray( i ) <= myinput( perm( i ) );
end generate;
leftoutput <= myArray(0 to 27);
rightoutput <= myArray(28 to 55);

end behavior;
```

Moduł roundRot1_MUSER_schem RYSUNEK

Opis wejść modułu

-

Sub-moduł expPerm

```
entity expPerm  is
port(        myinput: in std_logic_vector(0 to 31);
myoutput: out std_logic_vector(0 to 47));
```

```
end expPerm ;


architecture behavior of expPerm  is

type permArray is array(0 to 47) of integer range 0 to 47;

constant perm: permArray :=
(31, 0, 1, 2, 3, 4,
3, 4, 5, 6, 7, 8,
7, 8, 9,10,11,12,
11,12,13,14,15,16,
15,16,17,18,19,20,
19,20,21,22,23,24,
23,24,25,26,27,28,
27,28,29,30,31,0);

signal myArray : std_logic_vector(0 to 47);

begin

et0: for i in 0 to 47 generate
myArray( i ) <= myinput( perm( i ) );
end generate;
myoutput <= myArray(0 to 47);

end behavior;
```

Sub-moduł `xor48`

```
entity xor48 is
Port ( myinput : in  STD_LOGIC_VECTOR (0 to 47);
key : in  STD_LOGIC_VECTOR (0 to 47);
myoutput : out  STD_LOGIC_VECTOR (0 to 47));
end xor48;

architecture Behavioral of xor48 is

begin
myoutput <= key xor myinput;

end Behavioral;
```

Sub-moduł `sboxes`

```
entity sboxes is
port(        myinput: in std_logic_vector(0 to 47);
myoutput: out std_logic_vector(0 to 31));
end sboxes;

architecture behavior of sboxes is

component sbox_0
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_1
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_2
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_3
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_4
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_5
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_6
port(        myinput: in std_logic_vector(0 to 5);
myoutput: out std_logic_vector(0 to 3));
end component;

component sbox_7
port(        myinput: in std_logic_vector(0 to 5);
```

```
myoutput: out std_logic_vector(0 to 3));
end component;

begin

sb0: sbox_0  port map(
myinput=>myinput(0 to 5),
myoutput=>myoutput(0 to 3));

sb1: sbox_1 port map(
myinput=>myinput(6 to 11),
myoutput=>myoutput(4 to 7));

sb2: sbox_2 port map(
myinput=>myinput(12 to 17),
myoutput=>myoutput(8 to 11));

sb3: sbox_3 port map(
myinput=>myinput(18 to 23),
myoutput=>myoutput(12 to 15));

sb4: sbox_4 port map(
myinput=>myinput(24 to 29),
myoutput=>myoutput(16 to 19));

sb5: sbox_5 port map(
myinput=>myinput(30 to 35),
myoutput=>myoutput(20 to 23));

sb6: sbox_6 port map(
myinput=>myinput(36 to 41),
myoutput=>myoutput(24 to 27));

sb7: sbox_7 port map(
myinput=>myinput(42 to 47),
myoutput=>myoutput(28 to 31));
end;
```

Sub-moduł `pBlockPerm`

```
entity pblockPerm  is
port( myinput: in std_logic_vector(0 to 31);
myoutput: out std_logic_vector(0 to 31));
end pblockPerm ;
```

```
architecture behavior of pblockPerm  is

type permArray is array(0 to 31) of integer range 0 to 31;

constant perm: permArray :=
(15, 6,19,20,
28,11,27,16,
0,14,22,25,
4,17,30, 9,
1, 7,23,13,
31,26, 2, 8,
18,12,29, 5,
21,10, 3,24);

signal myArray : std_logic_vector(0 to 31);

begin
et0: for i in 0 to 31 generate
myArray( i ) <= myinput( perm( i ) );
end generate;

myoutput <= myArray(0 to 31);

end behavior;
```

Sub-moduł `SumLR`

```
entity sumLR is
port(        leftinput: in std_logic_vector(0 to 31);
rightinput: in std_logic_vector(0 to 31);
rightoutput: out std_logic_vector(0 to 31));
end sumLR;

architecture Behavioral of sumLR is

begin

rightoutput <= leftinput xor rightinput;

end Behavioral;
```

Sub-moduł `rotL1`

```
entity rotL1 is
port(        rightinput: in std_logic_vector(0 to 27);
leftinput: in std_logic_vector(0 to 27);
rightoutput: out std_logic_vector(0 to 27);
leftoutput: out std_logic_vector(0 to 27));
end rotL1;

architecture Behavioral of rotL1 is

signal leftKey : std_logic_vector(0 to 27);
signal rightKey : std_logic_vector(0 to 27);

begin

et0: for i in 0 to 26 generate
leftKey( i ) <= leftinput( i + 1 );
rightKey( i ) <= rightinput( i + 1 );
end generate;
leftKey( 27 ) <= leftinput( 0 );
rightKey( 27 ) <= rightinput( 0 );

leftoutput <= leftKey(0 to 27);
rightoutput <= rightKey(0 to 27);

end Behavioral;
```

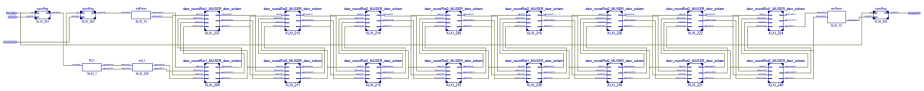
Moduł roundRot2_MUSER_scheme Schemat ten jest podobny, i korzysta z tych samych sub-modułów, co moduł `roundRot1_MUSER_scheme`, z wyjątkiem sub-modułu rotL2.

Sub-moduł `rotL2`

```
entity rotL2 is
port(        rightinput: in std_logic_vector(0 to 27);
leftinput: in std_logic_vector(0 to 27);
rightoutput: out std_logic_vector(0 to 27);
leftoutput: out std_logic_vector(0 to 27));
end rotL2;

architecture Behavioral of rotL2 is

signal leftKey : std_logic_vector(0 to 27);
signal rightKey : std_logic_vector(0 to 27);
```

```
begin

et0: for i in 0 to 25 generate
leftKey( i ) <= leftinput( i + 2 );
rightKey( i ) <= rightinput( i + 2 );
end generate;
leftKey( 27 ) <= leftinput( 1 );
rightKey( 27 ) <= rightinput( 1 );
leftKey( 26 ) <= leftinput( 0 );
rightKey( 26 ) <= rightinput( 0 );

leftoutput <= leftKey(0 to 27);
rightoutput <= rightKey(0 to 27);

end Behavioral;
```

Deszyfrator



Opis wejść układu

- Wejście zegarowe taktowane 50MHz, oznaczenie na schemacie `Clk_50MHz`
- Zaszyfrowana wiadomość, 64-bity, oznaczenie na schemacie `ciphertext(0:63)`
- Klucz szyfrujący, 64-bity, oznaczenie na schemacie `key(0:63)`

Opis modułów układu

Moduł `syncReg` Jest to ten sam moduł, którego użyliśmy w schemacie dla szyfratora.

Moduł `PC1` Jest to ten sam moduł, którego użyliśmy w schemacie dla szyfratora.

Moduł `initPerm` Jest to ten sam moduł, którego użyliśmy w schemacie dla szyfratora.

Moduł `rotL1` Jest to ten sam moduł, którego użyliśmy w schemacie dla szyfratora.

Moduł `decr_roundRot1_MUSER_decr_schem` Implementacje tego modułu prawie w ogóle nie różni się od użytego w szyfratorze modułu `roundRot1_MUSER_schem` z jedną różnicą – zamiast modułu rotL1 zastosowany jest tutaj sub-moduł rotR1, który przedstawiam poniżej

```
entity rotR1 is
port(        rightinput: in std_logic_vector(0 to 27);
leftinput: in std_logic_vector(0 to 27);
rightoutput: out std_logic_vector(0 to 27);
leftoutput: out std_logic_vector(0 to 27));
end rotR1;

architecture Behavioral of rotR1 is

signal leftKey : std_logic_vector(0 to 27);
signal rightKey : std_logic_vector(0 to 27);

begin

et0: for i in 1 to 27 generate
leftKey( i ) <= leftinput( i - 1 );
rightKey( i ) <= rightinput( i - 1 );
end generate;
leftKey( 0 ) <= leftinput( 27 );
rightKey( 0 ) <= rightinput( 27 );

leftoutput <= leftKey(0 to 27);
rightoutput <= rightKey(0 to 27);


end Behavioral;
```

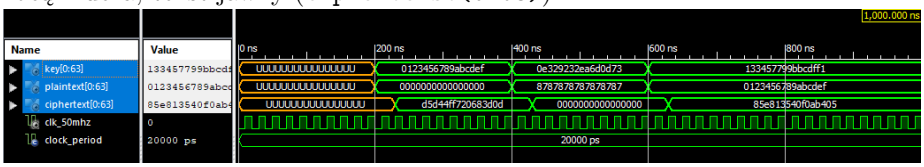
Moduł `decr_roundRot2_MUSER_decr_schem` Implementacje tego modułu prawie w ogóle nie różni się od użytego w szyfratorze modułu `roundRot1_MUSER_schem` z jedną różnicą – zamiast modułu rotL2 zastosowany jest tutaj sub-moduł rotR2, który przedstawiam poniżej

```
entity rotR2 is
port(        rightinput: in std_logic_vector(0 to 27);
leftinput: in std_logic_vector(0 to 27);
```

```
rightoutput: out std_logic_vector(0 to 27);
leftoutput: out std_logic_vector(0 to 27));
end rotR2;

architecture Behavioral of rotR2 is

signal leftKey : std_logic_vector(0 to 27);
signal rightKey : std_logic_vector(0 to 27);

begin

et0: for i in 2 to 27 generate
leftKey( i ) <= leftinput( i - 2 );
rightKey( i ) <= rightinput( i - 2 );
end generate;
leftKey( 1 ) <= leftinput( 27 );
rightKey( 1 ) <= rightinput( 27 );
leftKey( 0 ) <= leftinput( 26 );
rightKey( 0 ) <= rightinput( 26 );

leftoutput <= leftKey(0 to 27);
rightoutput <= rightKey(0 to 27);

end Behavioral;
```

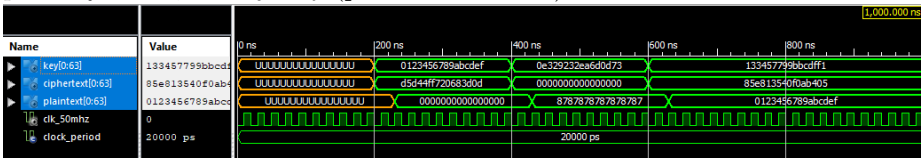
Przebiegi symulacji

Wejściem układu szyfrującego jest klucz (`key(0:63)`) oraz tekst jawny przeznaczony do zaszyfrowania (`plaintext(0:63)`). Wyjściem jest, zaszyfrowany za pomocą klucza, tekst jawny (`ciphertekst(0:63)`).



Rysunek 1: Przebieg symulacji dla układu szyfrującego DES

Wejściem układu deszyfrującego jest klucz (`key(0:63)`) oraz tekst zaszyfrowany do zaszyfrowania (`ciphertext(0:63)`). Wyjściem jest, odszyfrowany za pomocą klucza, tekst jawny (`plaintext(0:63)`).



Rysunek 2: Przebieg symulacji dla układu deszyfrującego DES

Implementacja

Wykorzystanie układu szyfratora

Liczba wykorzystancyh przerzutników: 184 na 9312 (1%)

Liczba wykorzystanych PIN'ów 4 wejściowych: 6272 na 9312 (67%)

Szczytowe użycie pamięci BRAM: 404MB

Wykorzystanie układu deszyfratora

Liczba wykorzystancyh przerzutników: 184 na 9312 (1%)

Liczba wykorzystanych PIN'ów 4 wejściowych: 6272 na 9312 (67%)

Szczytowe użycie pamięci BRAM: 401MB

Podsumowanie

Wnioski

Kryptografia i kryptoanaliza to bardzo trudne dziedziny i nawet tak prosta metoda szyfrowania, jaką jest DES, wymagał dużo pracy do zaimplementowania. Następnym możliwym rozszerzeniem projektu jest dodanie "skrzynki", która po podłączeniu do wejścia i wyjścia deszyfratora spróbuje złamać szyfr. Kolejnym pomysłem na rozszerzenie projektu jest dodanie podstawowej interakcji z użytkownikiem – np. wczytywanie danych z klawiatury oraz obsługa wyświetlacza.

Bibliografia

[1] Data Encryption Standard, https://pl.wikipedia.org/wiki/Data_Encryption_Standard

[2] Algorytm Symetryczny, https://pl.wikipedia.org/wiki/Algorytm_symetryczny

[3] Szyfr blokowy, https://pl.wikipedia.org/wiki/Szyfr_blokowy