STUDIA INFORMATICA 2000
Volume 21 Number 1 (39)

Adam ZIĘBIŃSKI, Jacek BAGIŃSKI, Krzysztof FAŁEK
Politechnika Śląska, Instytut Informatyki

IMPLEMENTACJE BLOKOWYCH ALGORYTMÓW KRYPTOGRAFICZNYCH W JĘZYKU VHDL

Streszczenie. W celu poszerzenia badań prowadzonych w Instytucie Informatyki, nad sprzętowym rozwiązaniem szyfratora, rozpoczęta została realizacja algorytmów IDEA i Blowfish. W artykule omówiono wyniki implementacji obydwu blokowych algorytmów kryptograficznych w środowisku Active-HDL firmy Aldec.

IMPLEMENTATION OF BLOCK CRYPTOGRAPHIC ALGORITHMS IN VHDL

Summary. Execution of the IDEA and the Blowfish algorithms has been started in order to broaden the research into the hardware solution of cipher conducted in the Institute of Computer Science. The outcomes of implementation of both block cryptographic algorithms in the environment of Active-HDL (of Aldec) are discussed in this article.

1. Wprowadzenie

W Instytucie Informatyki Politechniki Śląskiej realizowany jest projekt celowy obejmujący między innymi prace nad sprzętowym rozwiązaniem szyfratora i deszyfratora opartego na matrycach programowalnych FPGA [11]. W ramach realizowanych prac wykonywane są badania nad implementacją algorytmu DES w środowisku matryc programowalnych [12] z wykorzystaniem oprogramowania CAD/CAE [13][14]. W celu poszerzenia prowadzonych badań rozpoczęta została realizacja innych blokowych algorytmów kryptograficznych w środowisku matryc programowalnych. Do dalszych badań wybrano algorytm IDEA i Blowfish [3]. Obydwa algorytmy szyfrują i deszyfrują 64-bitowe bloki danych tak jak w przypadku algorytmu DES. Klucz wykorzystywany do szyfrowania

i deszyfrowania danych w algorytmie DES jest 64-bitowy. W przypadku algorytmu IDEA dane mogą być szyfrowane kluczem 128-bitowym. Natomiast algorytm Blowfish może szyfrować dane kluczem o długości do 448 bitów.

Efektem prowadzonych badań ma być powstanie:

- biblioteki elementów realizujących poszczególne funkcje obu algorytmów kryptograficznych w języku VHDL,
- elementów bibliotecznych realizujących funkcje szyfrowania i deszyfrowania zarówno algorytmem IDEA, jak i Blowfish.

2. Blokowe algorytmy kryptograficzne

W odróżnieniu od algorytmów strumieniowych przekształcających tekst jawny w szyfrogram bit po bicie. Algorytmy blokowe operują na blokach tekstu zwykle długości 64 bitów. Ta własność algorytmów blokowych sprawia, że zdecydowanie szybciej wykonują operacje szyfrowania i deszyfrowania niż algorytmy strumieniowe. Dodatkową zaletą blokowych algorytmów kryptograficznych jest wykonywanie operacji szyfrowania i deszyfrowania w ten sam sposób. Algorytmy blokowe są symetryczne i korzystają z tego samego klucza podczas szyfrowania i deszyfrowania. Jedyna różnica pomiędzy szyfrowaniem i deszyfrowaniem polega na generowaniu różnych podkluczy.

W ramach prowadzonych badań rozpoczęto implementację dwóch blokowych algorytmów kryptograficznych: IDEA i Blowfish w środowisku Active-HDL.

2.1. Opis algorytmu Blowfish – uwagi ogólne

Algorytm Blowfish jest algorytmem blokowym, zaprojektowanym przez Bruca Schneiera w 1993 r. [3]. Pozwala on na szyfrowanie bloków danych o długości 64 bitów, przy użyciu klucza o długości nie przekraczającej 448 bitów. Algorytm można podzielić na dwie podstawowe części:

- inicjalizującą,
- szyfrującą.

Podstawowymi elementami algorytmu są S-bloki i podklucze P [7]. Podklucze są 32-bitowymi elementami generowanymi na podstawie klucza w części inicjalizującej, oznaczane P_1 do P_{18} wykorzystywane są kolejno w każdym cyklu algorytmu. S-bloki są czterema 256-elementowymi tablicami, których 32-bitowe dane generowane są podobne jak podklucze w części inicjalizującej algorytmu. Służą one do wyznaczania wartości funkcji F.

2.1.1. Inicjalizacja algorytmu

Aby rozpocząć szyfrowanie bądź deszyfrowanie za pomocą algorytmu Blowfish, należy przeprowadzić jego inicjalizację, mającą na celu wygenerowanie odpowiednich wartości podkluczy P oraz S-bloków na podstawie klucza. W pierwszej fazie inicjalizacji podklucze P oraz wszystkie S-bloki są wypełniane kolejnymi cyframi liczby π w zapisie szesnastkowym.

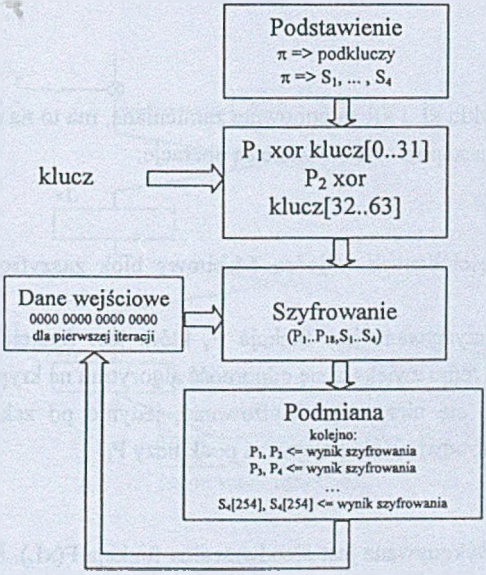
$P_1 := 243f6a88$

$P_2 := 85a308d3$

$P_3 := 13198a2e$

$P_4 := 03707344$ ...

Po podstawieniu wszystkich podkluczy i S-bloków wykonywana jest operacja xor na P_1 i pierwszych 32 bitach klucza szyfrującego, P_2 i kolejnych 32 bitach klucza itd. Po wyczerpaniu się bitów klucza następuje jego cykliczne powtarzanie aż do wykonania operacji na wszystkich podkluczach.



Rys. 1. Inicjalizacja algorytmu Blowfish
Fig. 1. Initialization of Blowfish algorithm

Tak przygotowanym zestawem wykonane zostaje szyfrowanie 64-bitowego bloku danych zawierającego same zera. Otrzymany zaszyfrowany 64-bitowy blok danych wyjściowych podzielony zostaje na dwie 32-bitowe części. Pierwsza część zastępuje podklucz P_1, natomiast druga P_2. Teraz następuje ponowne zaszyfrowanie wyniku pierwszej operacji szyfrowania przy użyciu nowych wartości P_1 i P_2. Otrzymany wynik przypisany zostaje podkluczom P_3

i P_4 oraz ponownie zawracany jest na wejście algorytmu szyfrującego. Proces ten jest powtarzany do momentu, aż podmienione zostaną wszystkie podklucze i S-bloki od S_1 do S_4. Inicjalizację obrazuje rys. 1.

Tak zainicjalizowane podklucze oraz S-bloki pozwalają na poprawne szyfrowanie algorytmem. Jeśli jednak algorytm ma spełniać rolę deszyfratora, należy jeszcze wykonać zamianę kolejności podkluczy: $P_1 := P_{18}$, $P_2 := P_{17} \ldots P_{18} := P_1$.

2.1.2. Szyfrowanie i deszyfrowanie przy pomocy algorytmu Blowfish

Szyfrowanie przy pomocy algorytmu składa się z szesnastu cykli, podczas których wykonywane są operacje permutacji oraz podstawień.

64-bitowy blok danych wejściowych zostaje podzielony na dwie 32-bitowe części xL i xR (rys. 2), następnie wykonywane są na nich kolejne cykle algorytmu. Podczas każdego cyklu wykonywane są następujące operacje (na 32-bitowych blokach danych oraz 32-bitowym podkluczu P_i,gdzie i jest numerem kolejnego cyklu):

$xL = xL \text{ xor } P_i$

$xR = F(xL) \text{ xor } xR$

zamiana xL i xR

Po wykonaniu szesnastego cyklu xL i xR są ponownie zamieniane, ma to na celu odwrócenie zamiany z ostatniego cyklu, następnie wykonywane są operacje:

$xR = xR \text{ xor } P_{17}$

$xL = xL \text{ xor } P_{18}$

i złączenie 32-bitowych części danych w jeden 64-bitowy blok zaszyfrowanych danych wyjściowych.

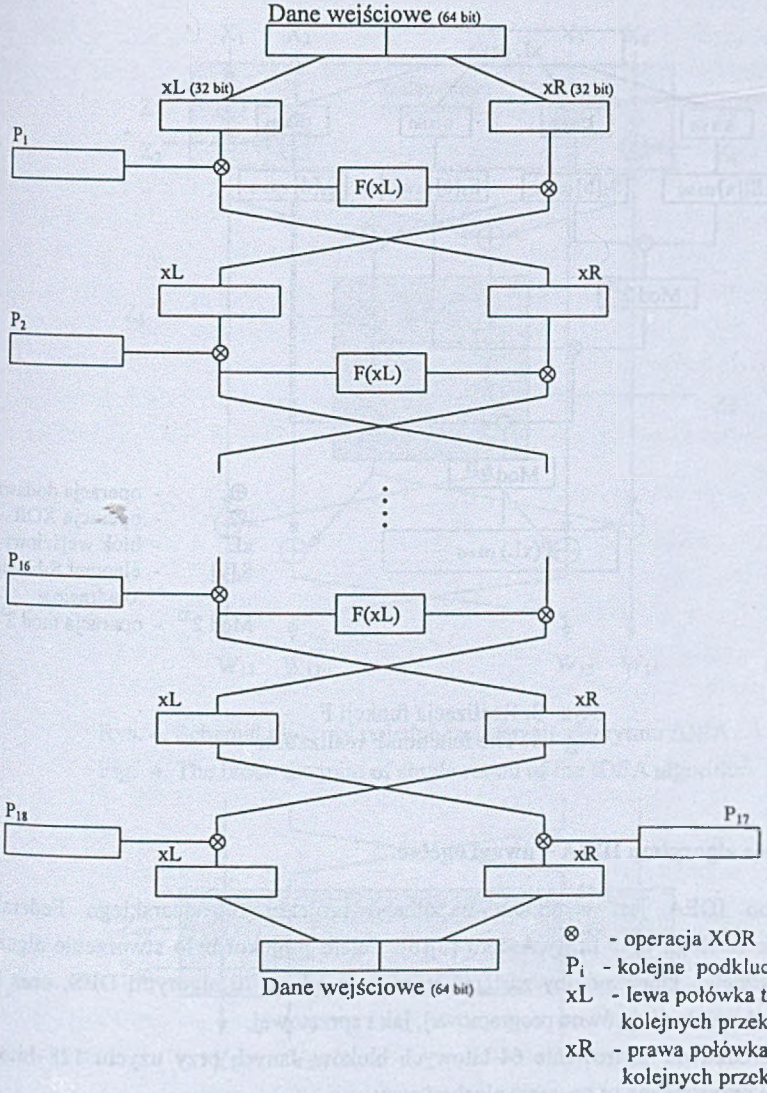
W każdym cyklu wykorzystywana jest funkcja *F*, która jest funkcją nieodwracalną, niezależną od iteracji. Dzięki temu zwiększa się odporność algorytmu na kryptoanalizę.

Deszyfrowanie nie różni się niczym od szyfrowania, jedynie po zakończeniu części inicjalizującej należy dokonać odwrócenia kolejności podkluczy P_i.

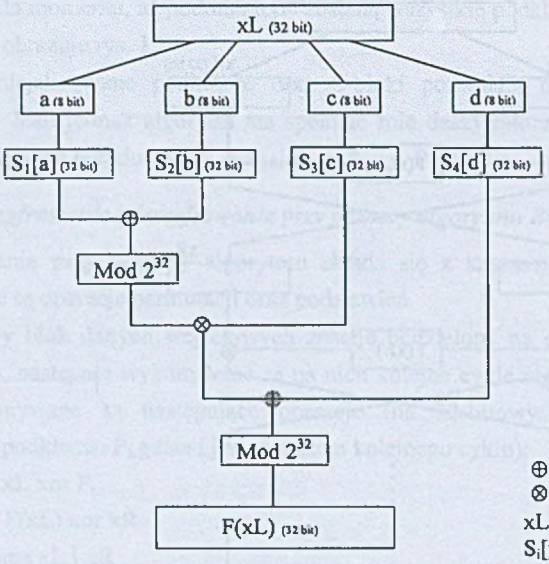
2.1.3. Funkcja F

Podczas każdego cyklu wykonywana jest nieodwracalna funkcja F(xL). Sposób działania funkcji przedstawiony jest na rys. 3. Wchodzący 32-bitowy blok danych zostaje podzielony na cztery 8-bitowe części: a, b, c, d. Następnie pobierane są 32-bitowe dane z S-bloków, odpowiednio $S_1[a]$, $S_2[b]$, $S_3[c]$, $S_4[d]$. (gdzie $S_2[126]$ oznacza 126 element drugiego S-bloku). Wynikiem zwracanym przez funkcję jest wynik operacji na kolejnych S-blokach:

$$F(xL) = ((((S_1[a] + S_2[b]) \bmod 2^{32}) \text{ xor } S_3[c]) + S_4[d]) \bmod 2^{32}$$



Rys. 2. Schemat blokowy algorytmu Blowfish
Fig. 2. The block diagram of Blowfish algorithm



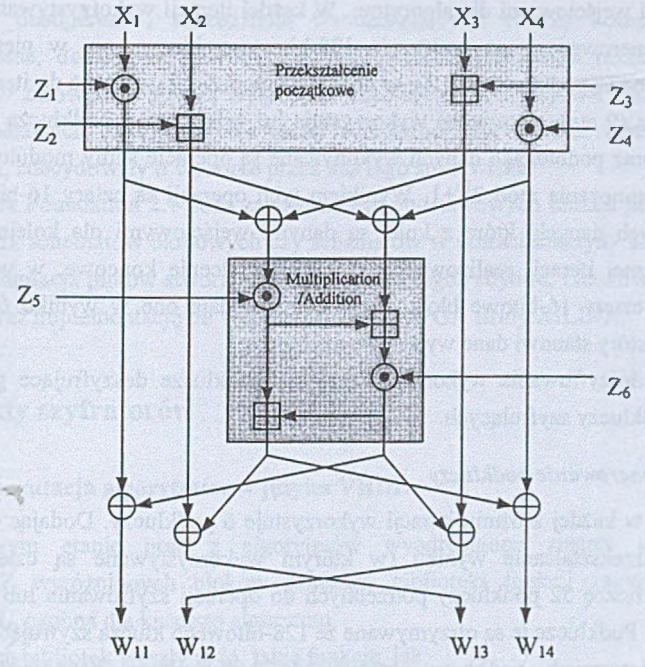
Rys. 3. Realizacja funkcji F
Fig. 3. The function F realization

2.2. Opis algorytmu IDEA – uwagi ogólne

Algorytm IDEA jest wynikiem wspólnego projektu Szwajcarskiego Federalnego Instytutu Technologii oraz firmy Ascom [5][6]. Celem projektu było stworzenie algorytmu kryptograficznego, który mógłby zastąpić stosowany od lat 70 algorytm DES, oraz byłby łatwy w implementacji zarówno programowej, jak i sprzętowej.

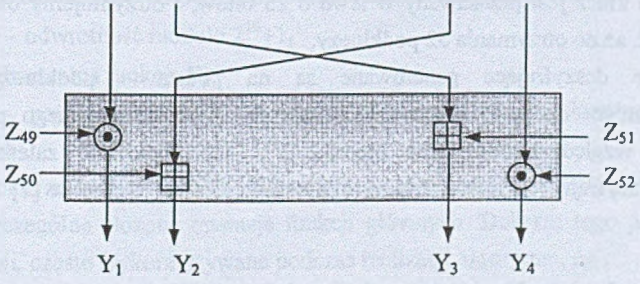
IDEA umożliwia szyfrowanie 64-bitowych bloków danych przy użyciu 128–bitowego klucza. Wykorzystywane są operacje algebraiczne:

- XOR logiczny (oznaczany dalej przez ⊕),
- dodawanie modulo 2^{16} (dodawanie z pominięciem przepełnienia oznaczane przez ⊞),
- mnożenie modulo $2^{16}+1$ (blok 16 zer odpowiada liczbie 2^{16}, operację tę oznaczamy przez ⊙) .



Rys. 4. Schemat blokowy pojedynczej iteracji algorytmu IDEA

Fig. 4. The block diagram of single round of the IDEA algorithm



Rys. 5. Schemat blokowy przekształcenia końcowego

Fig. 5. The block diagram of output transformation

2.2.1. *Szyfrowanie i deszyfrowanie*

Zarówno szyfrowanie, jak i deszyfrowanie odbywa się przy użyciu tego samego algorytmu. Składa się on z ośmiu cykli (iteracji) (rys. 4) oraz przekształcenia końcowego

(rys. 5). 64-bitowy blok danych jest dzielony na 16-bitowe podbloki X_1, X_2, X_3 oraz X_4, będące danymi wejściowymi dla algorytmu. W każdej iteracji wykorzystywanych jest sześć podkluczy, generowanych na podstawie 128-bitowego klucza, np. w pierwszej iteracji wykorzystywane są podklucze $Z_1...Z_6$, w drugiej iteracji $Z_7...Z_{12}$, itd. aż do iteracji nr 8. Etap przekształcania (9 etap algorytmu) wykorzystuje już tylko cztery podklucze ($Z_{49}...Z_{52}$). Na podkluczach oraz podblokach danych wykonywane są operacje sumy modulo 2, dodawania mod 2^{16} oraz mnożenia mod $2^{16}+1$. Wynikiem tych operacji są cztery 16-bitowe podbloki przekształconych danych, które z kolei są danymi wejściowymi dla kolejnej iteracji. Po wykonaniu ósmej iteracji realizowane jest przekształcenie końcowe, w wyniku którego otrzymujemy cztery 16-bitowe bloki. Po połączeniu dają one w wyniku 64-bitowy blok szyfrogramu, który stanowi dane wyjściowe szyfratora.

Podczas deszyfrowania wykorzystywane są podklucze deszyfrujące generowane na podstawie podkluczy szyfrujących.

2.2.2. Generowanie podkluczy

Algorytm w każdej z ośmiu iteracji wykorzystuje 6 podkluczy. Dodając do tego ostatni etap, czyli przekształcanie wyniku (w którym wykorzystywane są cztery podklucze) otrzymujemy liczbę 52 podkluczy potrzebnych do operacji szyfrowania lub deszyfrowania bloku danych. Podklucze te są otrzymywane ze 128-bitowego klucza szyfrującego.

Osiem pierwszych podkluczy szyfrowania otrzymujemy bezpośrednio z klucza szyfrowania po podzieleniu go na 16-bitowe fragmenty. Pierwszy podklucz oznaczany przez Z_1 równy jest 16 najbardziej znaczącym bitom, podklucz Z_2 następnym 16 bitom itd.

Następnie klucz jest przesuwany w lewo o 25 bitów, i otrzymujemy osiem następnych podkluczy itd. aż do otrzymania 52 podkluczy.

Podklucze deszyfrujące generowane są na podstawie podkluczy szyfrujących z wykorzystaniem operacji odwrotności modulo $2^{16}+1$ (oznaczanego przez Z_j^{-1}) oraz odwrotności względem dodawania modulo 2^{16} ($-Z_j$). Dokładne zależności pomiędzy podkluczami szyfrującymi a deszyfrującymi można znaleźć w literaturze [1].

3. Narzędzia wykorzystane do projektowania

Do realizacji obu projektów wykorzystujemy dwie aplikacje służące do wspomagania projektowania w środowisku matryc programowalnych, Active – HDL [13] oraz Xilinx Foundation 2.1 [14].

Active – HDL jest zintegrowanym środowiskiem przeznaczonym do tworzenia projektów opartych na języku opisu sprzętu VHDL. Bazując na koncepcji projektu, Active – HDL

pozwala sprawnie zarządzać jego składnikami. Jądrem systemu jest symulator VHDL-a, który wraz z debugerem i narzędziami wspomagającymi tworzy kompletny system pozwalający pisać, debugować i symulować kod VHDL. Umożliwia także wizualizację zmian sygnałów zarówno w postaci wykresów czasowych, jak i listy wartości sygnałów w kolejnych przedziałach czasowych. Te cechy symulatora, jak również ułatwiające pracę funkcje edytora, zdecydowały o wyborze przez nas tego środowiska.

Pakiet Xilinx Foundation 2.1 poza zapewnieniem podstawowych funkcji jak edycja kodu w języku VHDL, schematów blokowych czy schematów w postaci maszyny stanów, a także kompilacja i symulacja plików stworzonych za pomocą tegoż edytora, umożliwia dodatkowo syntezę kodu oraz implementację do wybranej matrycy FPGA firmy XILINX.

4. Projekty szyfratorów

4.1. Implementacja algorytmów w języku VHDL

W pierwszym etapie prac z algorytmów wyodrębnione zostały główne bloki funkcjonalne. Z wyróżnionych bloków powstały biblioteki funkcji zaimplementowane w języku VHDL, osobna dla każdego algorytmu.

W skład tych bibliotek weszły m.in. takie funkcje, jak:

- *mod_add* – dodawanie modulo 2^{16} (IDEA) lub modulo 2^{32} (Blowfish),
- *mod_mul* – mnożenie modulo $2^{16}+1$,
- *inv_add* – odwrotność względem dodawania modulo 2^{16},
- *inv_mod* – odwrotność modulo $2^{16}+1$,
- *F* – nieodwracalna funkcja F,
- *iteracja* – pojedyncza iteracja algorytmu (dotyczy zarówno algorytmu IDEA jak i Blowfish).

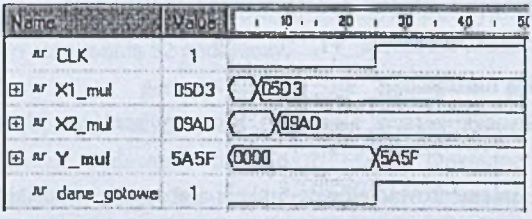
Jednak aby móc zaimplementować główne funkcje algorytmów, musiały powstać funkcje realizujące poszczególne złożone operacje funkcji głównych. Dobrym tego przykładem są funkcje konwersji, często wykorzystywane podczas realizacji algorytmu, np.:

- *bit_vec2int* – zamiana liczby typu bit_vector na liczbę typu integer,
- *int2bit_vec* – zamiana liczby typu integer na wektor bitów,
- *std2int* – zamiana liczby typu std_logic_vector na liczbę typu integer,
- *int2std* – zamiana liczby typu integer na std_logic_vector.

Funkcje konwersji zostały również zaimplementowane w języku VHDL i wchodzą w skład obu bibliotek.

Prostym przykładem funkcji bibliotecznej może być funkcja *mod_mul* (algorytm IDEA), której fragment zamieszczony jest poniżej. Funkcja wykonuje mnożenie modulo $2^{16}+1$ dwóch liczb, podanych jako parametry wejściowe funkcji w postaci wektorów 16-bitowych. Wewnątrz funkcji wektory zamieniane są na liczby typu integer i na nich wykonywane są działania. Rezultat działań konwertowany jest z powrotem na wektor 16 bitów i w tej postaci zwracany jest wynik funkcji.

```
function mod_mul...
   ...
   if X1_mul = "0000000000000000" then
      X1_pom := 65536;                    -- blok zer traktowany jak 2^16
   else
      X1_pom:=bit_vec2int(X1_mul);        -- konwersja: bit_vector→integer
   end if;
   if X2_mul = "0000000000000000" then
      X2_pom := 65536;
   else
      X2_pom:=bit_vec2int(X2_mul);
   end if;
   Y_pom := (X1_pom * X2_pom) mod 65537;
   -- konwersja wyniku: bit_vector→integer
   Y_mul := int2bit_vec(integer(Y_pom),16);
   ...
```

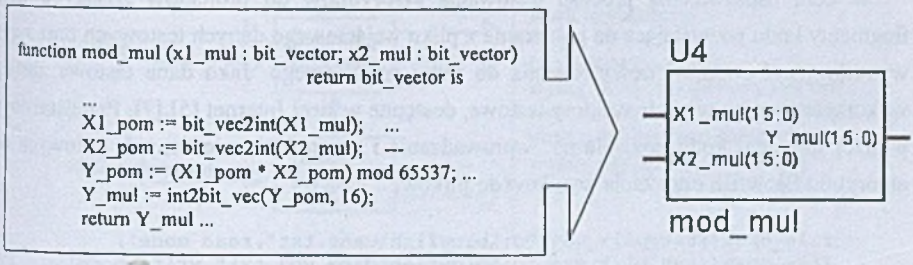


Time	Delta	CLK	X1_mul	X2_mul	Y_mul	dane_gotowe
0.000 ps	0	0	0000	0000	0000	0
5.000 ns	0	0	05D3	0000	0000	0
8.000 ns	0	0	05D3	09AD	0000	0
25.000 ns	0	1	05D3	09AD	0000	0
25.000 ns	1	1	05D3	09AD	5A5F	1

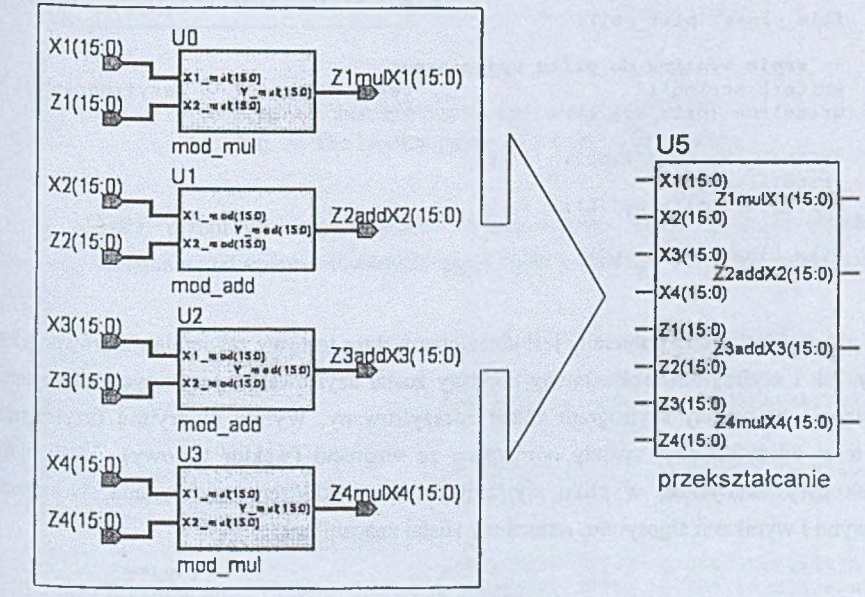
Rys. 6. Wyniki symulacji funkcji *mod_mul*
Fig. 6. Results of simulation of the *mod_mul* function

Wszystkie zaimplementowane funkcje zostały poddane symulacji logicznej w środowisku Active – HDL. Najprostszym przykładem takiej symulacji mogą być wykresy czasowe otrzymane w wyniku pojedynczej symulacji opisanej wyżej funkcji *mod_mul* (rys. 6).

W podobny sposób zostały przetestowane wszystkie elementarne funkcje, a następnie z funkcji tych powstały większe elementy (bloki funkcjonalne) algorytmu. W wyniku takich działań otrzymaliśmy projekt o strukturze hierarchicznej. Przykładem może tu być blok *mod_mul*, który oparty jest na funkcji o tej samej nazwie (rys. 7).



Rys. 7. Blok mod_mul
Fig. 7. The mod_mul block



Rys. 8. Przykład hierarchicznej struktury projektu
Fig. 8. The example of the hierachical structure of project

Blok ten jest jednym z elementów składowych bloku przekształcenia początkowego (rys. 8), który z kolei wchodzi w skład pojedynczej iteracji.

Tak otrzymane bloki funkcjonalne zostały połączone w kompletne algorytmy, które następnie zostały poddane testom w środowisku Active – HDL.

4.2. Końcowe testy algorytmów w środowisku Active - HDL

W celu usprawnienia procesu testowania algorytmów do projektów zostały dodane fragmenty kodu pozwalające na pobieranie z pliku wejściowego danych testowych oraz zapis wyników szyfrowania i deszyfrowania do pliku wyjściowego. Jako dane testowe zostały wykorzystane odpowiednie wektory testowe, dostępne w sieci Internet [5],[7]. Przedstawiony poniżej fragment kodu pozwala na wprowadzanie i wyprowadzanie danych testowych do algorytmu Blowfish oraz zapis wyników do plików.

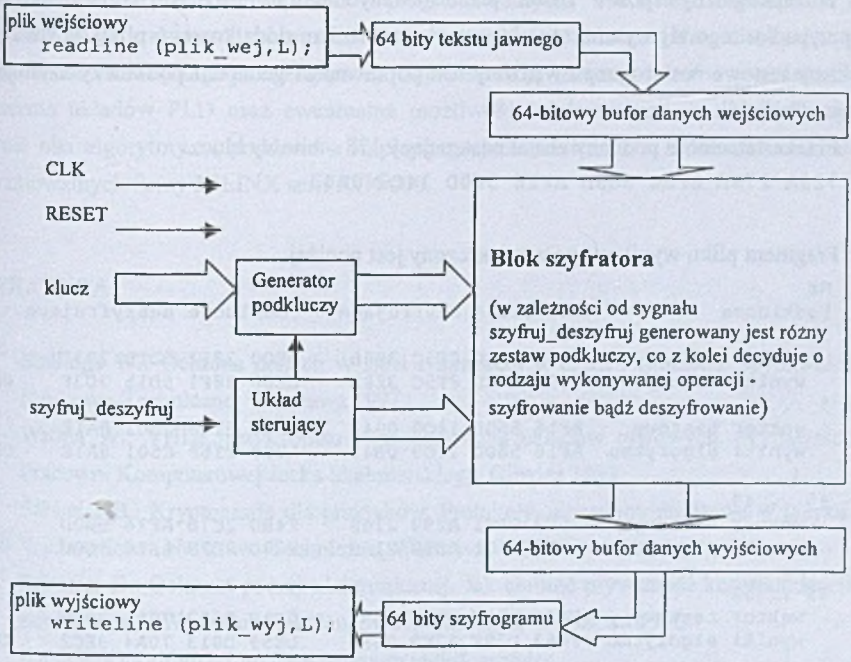
```
file_open(stat,plik_wej,"c:\blowfish\dane.txt",read_mode);
file_open(stat,plik_wyj,"c:\blowfish\dane_wyj.txt",write_mode);

-- czytanie danych z pliku wejsciowego
readline (plik_wej,L);
read (L,key);
     :
file_close (plik_wej);

-- zapis wynikow do pliku wyjsciowego
write(L,string'("                    tekst jawny        szyfrogram"));
writeline (plik_wyj,L);

write(L,string'("klucz:   "));
write(L, key);
writeline (plik_wyj,L);
     :
file_close (plik_wyj);
```

Z pliku wejściowego pobierany jest klucz oraz wektor testowy zawierający zarówno tekst jawny, jak i szyfrogram. Tekst jawny poddany został szyfrowaniu przy użyciu algorytmu, a następnie otrzymany szyfrogram został zdeszyfrowany. Wyniki algorytmu (szyfrogram oraz tekst odszyfrowany) zostały porównane ze wzorcem (wektor testowy). Jeżeli bloki danych były identyczne, w pliku wyjściowym, poza kluczem szyfrowania, wektorami testowymi i wynikami algorytmu, wstawiony został znacznik ok.



Rys. 9. Schemat blokowy testowania szyfratorów
Fig. 9. The block diagram of testing of the ciphers

W wyniku przeprowadzonych testów wygenerowany został plik potwierdzający poprawne działanie zaimplementowanych algorytmów, którego fragment przedstawiono poniżej.

```
                                tekst jawny       szyfrogram
klucz:          0000000000000000
wektor testowy:                 0000000000000000  4EF997456198DD78
wyniki algorytmu:               0000000000000000  4EF997456198DD78 ok

klucz:          0123456789ABCDEF
wektor testowy:                 1111111111111111  61F9C3802281B096
wyniki algorytmu:               1111111111111111  61F9C3802281B096 ok

klucz:          7CA110454A1A6E57
wektor testowy:                 01A1D6D039776742  59C68245EB05282B
wyniki algorytmu:               01A1D6D039776742  59C68245EB05282B ok

klucz:          43297FAD38E373FE
wektor testowy:                 762514B829BF486A  13F04154D69D1AE5
wyniki algorytmu:               762514B829BF486A  13F04154D69D1AE5 ok
```

W analogiczny sposób został przetestowany algorytm IDEA, jednak dodatkowo, w przypadku tego algorytmu, również z wykorzystaniem dodatkowego pliku zawierającego wektory testowe został przeprowadzony test poprawności generacji podkluczy szyfrujących i deszyfrujących.

Przekształceniom poddany został następujący 128 – bitowy klucz:

```
729A 27ED 8F5C 3E8B AF16 560D 14C9 0B43
```

Fragment pliku wynikowego zamieszczony jest poniżej.

```
Nr
Podklucza              Podklucze szyfrujące   Podklucze deszyfrujące
1 .. 4
  wektor testowy       729A 27ED 8F5C 3E8B    DE00 28F1 5D15 7D3F
  wyniki algorytmu     729A 27ED 8F5C 3E8B    DE00 28F1 5D15 7D3F    OK
5 .. 8
  wektor testowy       AF16 560D 14C9 0B43    A299 2168 C501 8A1E
  wyniki algorytmu     AF16 560D 14C9 0B43    A299 2168 C501 8A1E    OK
   ⋮                          ⋮                      ⋮
45 .. 48
  wektor testowy       75E2 CAC1 A299 2168    F4BD 2C7B AF16 560D
  wyniki algorytmu     75E2 CAC1 A299 2168    F4BD 2C7B AF16 560D    OK

49 .. 52
  wektor testowy       FB63 D70F A2EB C595    DE59 D813 70A4 3EC2
  wyniki algorytmu     FB63 D70F A2EB C595    DE59 D813 70A4 3EC2    OK.
```

W przypadku obydwu blokowych algorytmów kryptograficznych przeprowadzone testy wypadły pomyślnie, a otrzymane wyniki zgadzają się z danymi zawartymi w wektorach testowych. Można więc stwierdzić, że zarówno algorytm IDEA, jak i algorytm Blowfish zostały zaimplementowane w języku VHDL poprawnie i realizują funkcje szyfrowania i deszyfrowania danych.

5. Wnioski

Na obecnym etapie zostały zakończone prace związane z implementacją algorytmów w środowisku Active – HDL. Powstały biblioteki podstawowych elementów realizujących poszczególne funkcje algorytmów. Zarówno elementy biblioteczne, jak i działanie całych algorytmów zostały przetestowane w tym środowisku. Na podstawie otrzymanych wyników można stwierdzić, że implementacje poszczególnych bloków, jak i całych algorytmów zostały zrealizowane poprawnie.

Zarówno algorytm Blowfish, jak i IDEA zostały zaimplementowane w języku VHDL. W obu implementacjach nie zostały wykorzystane elementy biblioteczne związane

z którymkolwiek z producentów układów programowalnych. Wadą tego rozwiązania jest niewykorzystanie maksymalnych możliwości poszczególnych układów. Natomiast zaletą jest uniwersalność rozwiązania pozwalająca na implementację obu algorytmów niezależnie od producenta układów PLD oraz ewentualna możliwość implementacji w układach ASIC. Obecnie oba algorytmy są poddawane implementacji i testowaniu w środowisku układów programowalnych firmy XILINX serii XC4 000.

LITERATURA

1. Stallings W.: Ochrona danych w sieci i intersieci w teorii i praktyce. Wydawnictwa Naukowo-Techniczne, Warszawa 1997.
2. Wrona W.: VHDL język opisu i projektowania układów cyfrowych. Wydawnictwo Pracowni Komputerowej Jacka Skalmierskiego, Gliwice 1998.
3. Schneier B.: Kryptografia dla praktyków. Protokoły, algorytmy źródłowe w języku C. Wydawnictwa Naukowo-Techniczne, Warszawa 1995.
4. Schneier B.: Ochrona poczty elektronicznej. Jak chronić prywatność korespondencji w sieci Internet? Wydawnictwa Naukowo-Techniczne, Warszawa 1996.
5. IDEA™ Algorithm – http://www.ascom.ch/infosec/idea.
6. IDEA™ Algorithm, Short Description – http://www.it-sec.com.
7. The Blowfish Encryption Algorithm – http://www.counterpane.com/blowfish.html.
8. Foundation Express Application Note Supplement – http://www.xilinx.com.
9. Blowfish – http://fn2.freenet.edmonton.ab.ca/~jsavard/co0406.htm.
10. What on earth does "Propagating Cipher Block Chained 448-bit Blowfish" mean? – http://www.glaurung.demon.co.uk/checkpoint/crypto.html.
11. Projekt Celowy KBN nr 8T11C 026 98C/4258 „Bezpieczeństwo Systemów Komputerowych" - „Projekt rozwiązania sprzętowego szyfratora i deszyfratora opartego o matryce programowalne FPGA".
12. „Sprzętowa realizacja algorytmu szyfrującego „DES" w strukturę FPGA z wykorzystaniem aplikacji ACTIVE-CAD" - K. Trybicka-Francik, A. Ziębiński,, Materiały I Krajowej Konferencji Naukowej – Reprogramowalne Układy Cyfrowe, Szczecin 1998.
13. "Active-HDL, ver. 3.5", Aldec Inc., Henderson 1998.
14. "Xilinx Foundation Series, User's Guide vF2.1", XILINX, San Jose 1999.
15. "IEEE Standard VHDL Language AReference Manual", IEEE Std 1076-1993, IEEE, New York, June 1994.

Recenzent: Dr inż. Andrzej Białas

Wpłynęło do Redakcji 10 kwietnia 2000 r.

Abstract

The operation rules of the IDEA and the Blowfish [1][3][5][6][7] block cryptographic algorithms are presented in the first part of this paper. Next, the tools used during the formation of projects of the hardware implementation of these algorithms are briefly characterized. The main part of this article comprises description of another stages of the research on the implementation of the algorithms.

The functional blocks of the algorithms were isolated in the first stage of the research. The implementation of those blocks in VHDL and the formation of libraries of basic functions of the algorithms were the next stage. All elements of the libraries were tested in the Active-HDL environment [13]. The tests of whole algorithms were the last stage. The test vectors, which are accessible on the Internet pages [5], [7], were used here. The tests of the algorithms went well. The implementation of both algorithms in the environment of FPGA of Xilinx will be the next stage.