Галузь знань: Інформаційні технології

Напрям: Інженерія вбудованих систем

Шифр: ДСТУ

Апаратна реалізація алгоритму симетричного потокового перетворення ДСТУ 8845:2019 на програмованій логіковій інтегральній мікросхемі

2021 р.

АНОТАЦІЯ

В роботі на основі алгоритму симетричного потокового перетворення ДСТУ 8845:2019 реалізовано шифратор на базі програмованої логікової інтегральної мікросхеми (ПЛІС) із використанням мови опису апаратури Verilog HDL.

Розроблено та реалізовано на програмованій логіковій інтегральній мікросхемі множення двох многочленів по модулю $x^8 + x^4 + x^3 + x^2 + 1$ у вигляді передобчислених комірок модулів постійної запам'ятовуючої пам'яті, що уможливлює виконання операції MixColumn для функції нелінійної підстановки *T* за один такт. Виконання функції множення на α та α^{-1}в арифметиці поля Галуа $GF(2^{64})$ реалізовано на базі асинхронних постійних запам'ятовуючих пристроїв та комбінаційної логіки. Контроль режимів роботи регістру зсуву з лінійним оберненим зв'язком виконано на базі скінченого автомата. Керування вихідним ключовим потоком та елементами оберненого зв'язку виконано на базі комбінаційної схеми.

Робота складається зі вступу, двох розділів, висновків, переліку використаної літератури, додатків, загальним об'ємом 32 сторінок машинописного тексту, рисунків – 7, таблиць – 0, бібліографій – 14.

Ключові слова: алгоритм симетричного потокового перетворення, програмована логікова інтегральна мікросхема, апаратна реалізація.

Зміст

Вступ

Розвиток технологій призводить до того, що питання збереження та передачі даних постійно набуває все більшої актуальності. Відмови в роботі інформаційно-керуючих систем, втрата, знищення, підміна, перехоплення чи спотворення інформаційних потоків призводить до недопустимих втрат, а часто до катастрофічних наслідків. У сьогоденні автентичність та захищеність інформації є важливою складовою будь-якого процесу.

Масове зростання кількості приладів, котрі генерують дані щосекунди, приладів IoT ставить перед нами питання захисту інформації, котра генерується в реальному часі і котрою обмінюються як елементи однієї мережі так і елементи в глобального трафіку. Для вирішення даних проблем використовують потокове шифрування, зазвичай побітову операцію XOR між ключовим потоком і повідомленням. Проте для використання у сфері вбудованих систем використовують “полегшені” версії даних алгоритмів (lightware cryptography), які досить суттєво відрізняються від стандартних, “батьківських” варіацій. Для побудови потокових шифрів створено стандарт ISO/IEC 18033-4:2011, у якому й описано вихідні функції для різних потокових шифрів, та певні генератори псевдовипадкових чисел, які призначено для захисту інформації з обмеженим доступом, зокрема, для забезпечення конфіденційності інформації під час її обробки.

В даний час вимоги для криптоалгоритмів такого типу істотно зростають. Розширення каналів передачі даних, зростання їх швидкодії та розвиток квантових обчислень ставить під питання придатність більшості криптоалгоритмів, що вже використовуються.

Питання вдосконалення та розробки методів потокового симетричного шифрування є актуальною і важливою проблемою сьогодення. Особливо для вбудованих систем, де ціна та витрати енергії виходять на перший план, а обчислювальна потужність визначається характеристиками мікроконтролерів або програмованих логікових інтегральних схем. Такі системи знаходять все

ширше застосування при побудові промислових, споживчих, медичних, автомобільних та кіберфізичних систем.

Необхідність захисту інформації у вбудованих системах вимагає ефективну реалізацію криптоалгоритмів, за умови обмежень, які накладаються цими системами. Нами було обрано для реалізації криптоалгоритм “Струмок”, котрий виділяється високою швидкість формування ключового потоку (понад 10 Гбіт/с) та придатний для використання в постквантовому оточенні, при чому він добре підходить для використання в системах з обмеженими ресурсами.

Даний алгоритм показав високі результати при тестуванні швидкодії на різних платформах (UNIX, Windows, Android) [4] та є криптостійким (довжина ключа 512 та орієнтація на системи типу x64), проте його не було досліджено на базі програмованої логіки він не був реалізований.

Метою роботи є розробка апаратної реалізації алгоритму симетричного потокового перетворення ДСТУ 8845:2019 для вбудованих систем.

Об’єкт дослідження – процеси перетворення інформаційних повідомлень в програмованних логікових інтегральних схемах.

Предметом дослідження є апаратна реалізація алгоритму симетричного потокового перетворення ДСТУ 8845:2019.

I. Алгоритм симетричного потокового перетворення ДСТУ 8845:2019

1.1 Опис алгоритму

ДСТУ 8845:2019 — алгоритм симетричного потокового перетворення. В основі ДСТУ 8845:2019 лежить класична схема підсумовуючого генератора подібна генератору *SNOW* 2.0, *SNOW* 3.0 та *SNOW V*, які визначено в [13]. В основі ДСТУ 8845:2019 збережені всі базові операції шифрів сімейства *SNOW*, а раунд шифрування AES замінений на функцію нелінійної підстановки *T*, що реалізовує перестановку елементів скінченного поля $GF(2^{64})$ за допомогою компонентів національного стандарту симетричного крипто перетворення ДСТУ 7624:2014 [11].

ДСТУ 8845:2019 використовує 256-бітний вектор ініціалізації *IV* та 256-бітний або 512-бітний секретний ключ *K* і забезпечує високий та надвисокий рівень стійкості із врахуванням можливого застосування квантового криптографічного аналізу. При розробці алгоритму ДСТУ 8845:2019 орієнтувалися на сучасні 64-бітні обчислювальні системи, тому розмір слова обрано рівним 8 байт. запису байтів застосовують подання від старшого до молодшого. Генератор ключових потоків ДСТУ 8845:2019 у режимі генерації гами шифру схематично приведено на рис. 1.

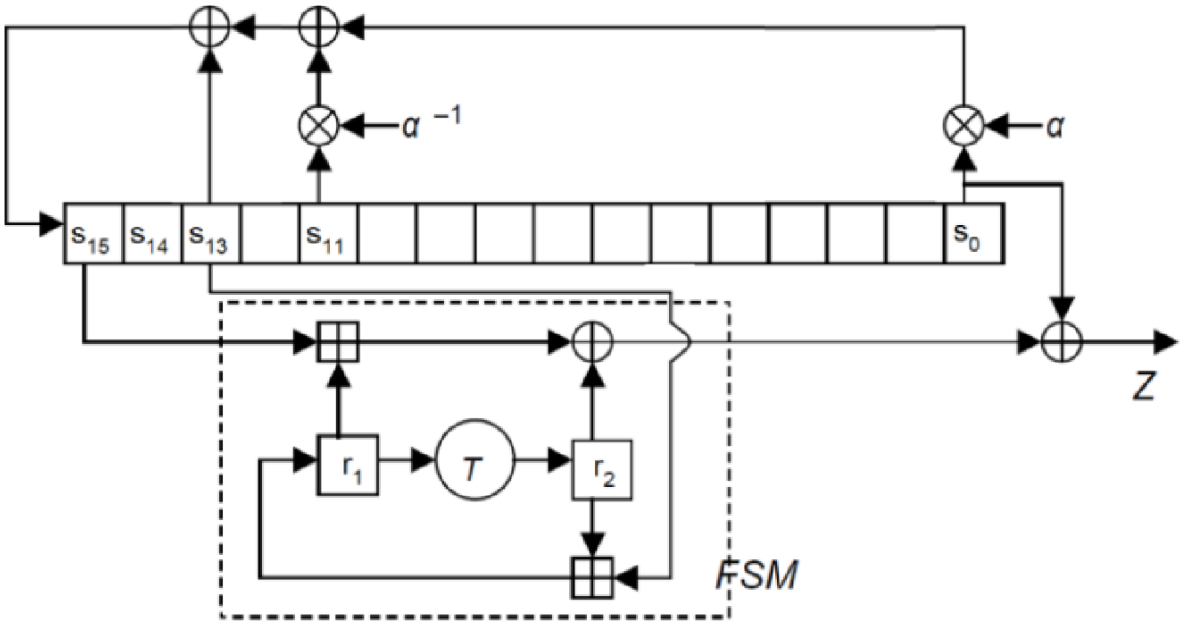

Рис. 1. Генератор ключових потоків ДСТУ 8845:2019 у режимі генерації гами шифру в довільний момент часу *i*

Як випливає з рис. 1 основними компонентами генератору є регістр зсуву з лінійним зворотнім зв'язком та скінчений автомат на базі якого виконується нелінійне перетворення *T*. Вхідні дані (ключ шифрування *K* та вектор ініціалізації *IV*) використовуються для ініціалізації змінної стану $S_i (i \geq 0)$, яка складається із двох компонент до складу яких входять[12]:

- 16 змінних $s^{(i)}$ – комірок регістра зсуву з лінійним зворотнім зв'язком: $s^{(i)} = (s_{15}^{(i)}, s_{14}^{(i)}, s_{13}^{(i)}, s_{12}^{(i)}, s_{11}^{(i)}, s_{10}^{(i)}, s_9^{(i)}, s_8^{(i)}, s_7^{(i)}, s_6^{(i)}, s_5^{(i)}, s_4^{(i)}, s_3^{(i)}, s_2^{(i)}, s_1^{(i)}, s_0^{(i)})$;
- Два регістри скінченного автомату $r^{(i)}$: $r^{(i)} = (r_2^{(i)}, r_1^{(i)})$.

На виході отримуємо ключовий потік (гамма), який формується з 8-байтних слів Z_i.

З рисунку 1 слідує , що відводи регістра зсуву з лінійним оберненим зв'язком побудовані за примітивним над полем $GF(2^{64})$ поліномом: $f(x) = x^{16} + x^{13} + \alpha^{-1}x^{11} + \alpha$, де α є коренем примітивного над полем $GF(2^8)$ поліному $gz(z) = z^8 + \beta^{170}z^7 + \beta^{166}z^6 + \beta^2 z^5 + \beta^{224}z^4 + \beta^{70}z^3 + \beta^2$.

Поле $GF(2^8)$ як і в ДСТУ 7624:2014 побудовано за примітивним на полем $GF(2)$ поліномом $p(y) = x^8 + x^4 + x^3 + x^2 + 1$, а коефіцієнти $g(z)$ подаються через ступінь примітивного елементу β поля $GF(2^8)$, тобто β корінь поліному $p(y)$. Тобто, у нас є вежа полів: $GF(2) \subset GF(2^8) \subset GF(2^{64}) \subset GF(2^{1024})$, де

- поле $GF(2^{1024})$ задається відводами зворотного зв'язку як фактор кільце $GF(2^{64})[x]/(f(x))$,
- поле $GF(2^{16424})$ задається як фактор кільце $GF(2^8)[z]/(g(z))$,
- поле $GF(2^{1024})$ задається як фактор кільце $GF(2)[y]/(p(y))$.

З вищезазначеного, слідує що період вихідної послідовності становить 2^{1024}.

Структурно в алгоритмі симетричного потокового перетворення ДСТУ 8845:2019 виділяють три основні функції [12]:

- функція ініціалізації, яка приймає в якості вхідних даних 256-бітний вектор ініціалізації *IV* та 256-бітний або 512-бітний секретний ключ *K*, і виробляє початкове значення змінної стану $S_0 = (s^{(0)}, r^{(0)})$;

- функція наступного стану *Next*, яка приймає на вхід зміну стану $S_i = (s^{(i)},\ r^{(i)})$ та виробляє наступне значення змінної стану $S_{i+1} = (s^{(i+1)},\ r^{(i+1)})$;
- функція ключового потоку *Strm*, що приймає на вході змінну стану $S_i = (s^{(i)},\ r^{(i)})$ та виробляє на виході 64-бітний ключовий потік Z^i.

Також функція *Next* може виконуватися в двох режимах, в залежності від способу виконання ітерації, як частина реалізації ініціалізації алгоритму ДСТУ 8845:2019 або як частина функції ключового потоку *Strm*.

ІІ. Апаратна реалізація алгоритму симетричного потокового перетворення ДСТУ 8845:2019

2.1 Функція ініціалізації внутрішнього стану *Init*

Алгоритм симетричного потокового перетворення ДСТУ 8845:2019 використовує структурні перетворення інформації, що присутні в сімействі шифрів SNOW та ДСТУ 7624:2014 при 256 або 512-ти бітному вхідному ключу K та 256-ти бітному вектору ініціалізації IV. Функція ініціалізації внутрішнього стану описується Функція ініціалізації внутрішнього стану *Init* описується наступним чином [12]:

Вхід: 256 або 512-бітний ключ K, 256-бітний вектор ініціалізації IV.

Вихід: початкове значення змінної стану $S_0 = (s^{(0)}, r^{(0)})$.

Ключ для 256-бітної та 512-бітної версії потокового шифру представляється у вигляді чотирьох 64-бітних слів $K = (K_3, K_2, K_1, K_0)$ або восьми 64-бітних слів $K = (K_7, K_6, K_5, K_4, K_3, K_2, K_1, K_0)$, де K_3 та K_7 , відповідно для 256 и 512 біт, найбільш значущі слова, а K_0 – найменш значущі. Вектор ініціалізації представляється у вигляді чотирьох 64-бітних слів $IV = (IV_3, IV_2, IV_1, IV_0)$, де IV_3 – найбільш значуще слово, а IV_0 – найменш значуще.

На вхід апаратно реалізованого модуля підключено 512, 256 бітну та 1 бітну шину, що містять ключ та розмір ключа. В залежності від обраного розміру ключа визначеного сигналом керування в 16 комірок регістру зсуву подаються наступні значення:

Для версії з 256-бітним ключем

$$s_{15}^{(-33)} = \neg K_0,\ s_{14}^{(-33)} = K_1,\ s_{13}^{(-33)} = \neg K_2,\ s_{12}^{(-33)} = K_3,\ s_{11}^{(-33)} = K_0,\ s_{10}^{(-33)}$$
$$= \neg K_1, s_9^{(-33)} = K_2, s_8^{(-33)} = K_3, s_7^{(-33)} = \neg K_0, s_6^{(-33)} = \neg K_1, s_5^{(-33)}$$
$$= K_2 \oplus IV_3, s_4^{(-33)} = K_3, s_3^{(-33)} = K_0 \oplus IV_2, s_2^{(-33)} = K_1 \oplus IV_1, s_1^{(-33)}$$
$$= K_2, s_0^{(-33)} = K_3 \oplus IV_0.$$

Для версії з 512-бітним ключем

$$s_{15}^{(-33)} = K_0, s_{14}^{(-33)} = \neg K_1, s_{13}^{(-33)} = K_2, s_{12}^{(-33)} = K_3, s_{11}^{(-33)} = \neg K_7, s_{10}^{(-33)}$$
$$= K_5, s_9^{(-33)} = \neg K_6, s_8^{(-33)} = K_4 \oplus IV_3, s_7^{(-33)} = \neg K_0, s_6^{(-33)}$$
$$= K_1, s_5^{(-33)} = K_2 \oplus IV_2, s_4^{(-33)} = K_3, s_3^{(-33)} = K_4 \oplus IV_1, s_2^{(-33)}$$
$$= K_5, s_1^{(-33)} = K_6, s_0^{(-33)} = K_7 \oplus IV_0.$$

Далі виконуються 32 ініціюючих такти без генерації ключового потоку, тобто чотири повних циклів. Формально це подається наступним чином:

$$S_{-1} = Next^{32}(S_{-33}, INIT),$$

що означає 32 ітерації з виконання функції *Next* у режимі ініціалізації *INIT*, $S_{-33} = (s^{(-33)}, r^{(-33)})$ – обраховані на попередньому кроці значення змінної стану. Розрахунок початкового значення змінної стану $S_0 = (s^{(0)}, r^{(0)})$ виконується за наступним правилом: $S_0 = Next(S_{-1})$, тобто шляхом виконання функції Next у звичайному режимі.

2.2 Функція наступного стану *Next*

Функція стану *Next* описується наступним чином. На вході: змінна стану $S_i = (s^{(i)}, r^{(i)})$, обраний режим (звичайний, або режим ініціалізації). На виході: Наступне значення змінної стану $S_{i+1} = \left(s^{(i+1)}, r^{(i+1)}\right)$. Спочатку виконується нелінійна підстановка для оновлення значення регістру $r_2^{(i+1)}$ скінченного автомату. Для цього розраховується значення функції T : $r_2^{(i+1)} = T\left(r^{(i)}\right)$. Далі оновлюється значення регістру $r_1^{(i+1)}$ скінченного автомату. Для цього розраховується значення $r_1^{(i+1)} = r_2^{(i+1)} +_{64} s_{13}^{(i)}$, де $+_{64}$ позначає операцію додавання цілих чисел за модулем 2^{64} . В результаті оновлюється значення 15 комірок регістру зсуву $s_j^{(i+1)} = s_{j+1}^{(i)}$ для всіх j = 0, 1, …, 14.

Оновлення значення 16-ї комірки, якщо встановлено звичайний режим функції *Next*, відбувається за правилом за правилом:

$$s_{15}^{(i+1)} = \left(s_0^{(i)} \otimes \alpha\right) \oplus \left(s_{11}^{(i)} \otimes \alpha^{-1}\right) \oplus s_{13}^{(i)}.$$

Якщо встановлено режим ініціалізації *INIT* функції *Next*, значення обчислюється за правилом:

$$s_{15}^{(i+1)} = FSM\left(s_{15}^{(i)}, r_1^{(i)}, s_2^{(i)}\right) \oplus \left(s_0^{(i)} \otimes \alpha\right) \oplus \left(s_{11}^{(i)} \otimes \alpha^{-1}\right) \oplus s_{13}^{(i)}.$$

Схематичне зображення генератора ключових потоків при виконанні функції Next у режимі ініціалізації приведено на рис. 2.

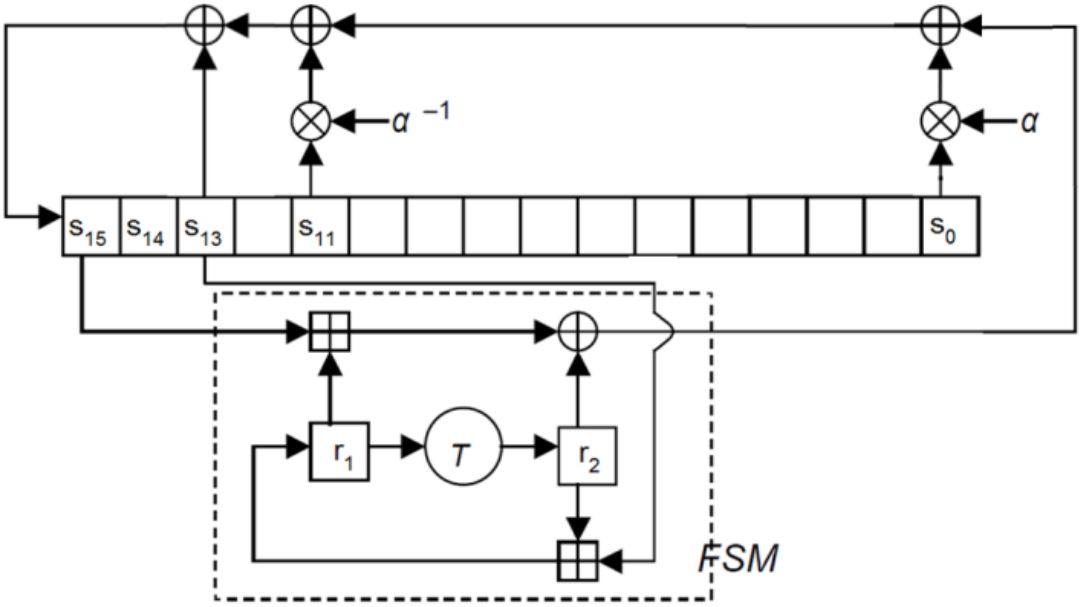

Рис. 2. Генератор ключових потоків ДСТУ 8845:2019 у режимі ініціалізації

Операції множення $\otimes$ на α та на α^{-1} та сутність функції FSM пояснюються далі.

Функція ключового потоку *Strm* визначається наступним чином:

$$Z_i = FSM(s_{15}^{(i)}, r_1^{(i)}, s_2^{(i)}) \oplus s_0^{(i)},$$

де Z_i 64-бітовий ключовий потік.

Функція скінченного автомату *FSM* визначається, як: $q = (x +_{64} y) \oplus z$ де *x*, *y* і *z* три 64-бітових рядка.

2.3 Функція нелінійної підстановки *T*

В ДСТУ 8845:2019 функція нелінійної підстановки *T* реалізує перестановку елементів скінченого поля GF(2^{64}) за допомогою операції *MixColumn* визначеної в ДСТУ 7624:2014 [8] та полягає в перетворені кожного елементу матриці стану за формулою:

$$w_{i,j} = (u >>> i) \otimes W_j \qquad (1)$$

де: $\otimes$ - скалярний добуток векторів, W_j – стовпець матриці стану,$v = (0x01, 0x01, 0x05, 0x01, 0x08, 0x06, 0x07, 0x04)$. Операція множення і додавання проводиться в кінцевому полі утвореному незвідним поліномом $Y(x) = x^8 + x^4 + x^3 + x^2 + 1$ або $0x11d$ в шістнадцятковому вигляді. Вектор v утворює циркулятивну матрицю МДР-коду і складається з послідовності байтових констант у шістнадцятковосу поданні, які інтерпретують як елементи поля $GF(2^8)$, при цьому циклічний зсув виконується відносно вектора над скінченим полем. Операцію *MixColumn* можна віуалізувати як:

$$C_j = \begin{bmatrix} 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 \\ 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 \\ 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 & 0x06 \\ 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 & 0x08 \\ 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 & 0x01 \\ 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 & 0x05 \\ 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 & 0x01 \\ 0x01 & 0x05 & 0x01 & 0x08 & 0x06 & 0x07 & 0x04 & 0x01 \end{bmatrix} \begin{bmatrix} w_{0,j} \\ w_{1,j} \\ w_{2,j} \\ w_{3,j} \\ w_{4,j} \\ w_{5,j} \\ w_{6,j} \\ w_{7,j} \end{bmatrix} \quad (2)$$

де C_j результуючий вектор. Множення байтів із стовпців матриці стану на вектор v можна реалізувати у вигляді комбінаційної схеми.

Множення будь-якого ненульового елемента поля $GF(2^8)$ на 2 можна реалізувати як логічний зсув бітів вліво на 1 позицію, в результаті чого буде отримано добуток розміром 9 біт. Далі необхідно отриманий добуток взяти по модулю $x^8 + x^4 + x^3 + x^2 + 1$. Дані операції можна записати наступним чином:

$$k_2(w_{i,j}) = w_{i,j} * 2 = \{w_{i,j}[6:0], 1'b0\} \oplus (8'h1d \wedge \{8\{w_{i,j}[7]\}\}) \quad (3)$$

де: $\wedge$ - операція кон'юнкції. Відношення (3) описує комбінаційну схему, що приведена на рис. 3.

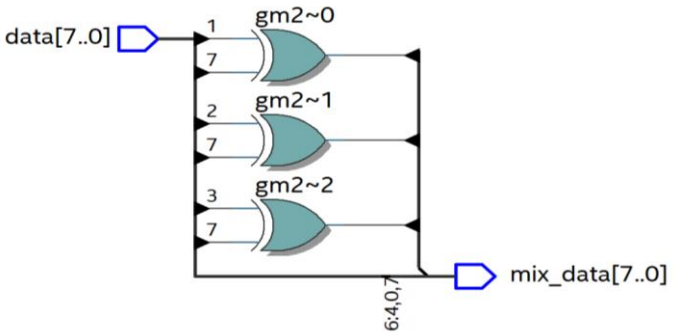

Рис. 3. Множення ненульового елемента поля $GF(2^8)$ на 2 по модулю $Y(x)$

Якщо ми можемо множити ненульового елемента поля $GF(2^8)$ на 2 по модулю $x^8 + x^4 + x^3 + x^2 + 1$ то ми можемо множити на будь-який інший ненульовий елемент поля $GF(2^8)$ більший за 2. Відповідно множення на 3 можна записати як:

$$k_3(w_{i,j}) = w_{i,j} * 3 = k_2(w_{i,j}) \oplus w_{i,j} = (w_{i,j} * 2) \oplus w_{i,j} = (\{w_{i,j}[6:0], 1'b0\} \oplus (8'h1d \wedge \{8\{w_{i,j}[7]\}\})) \oplus w_{i,j} \quad (4)$$

Відповідно комбінаційна схема, що описується (4) матиме вигляд:

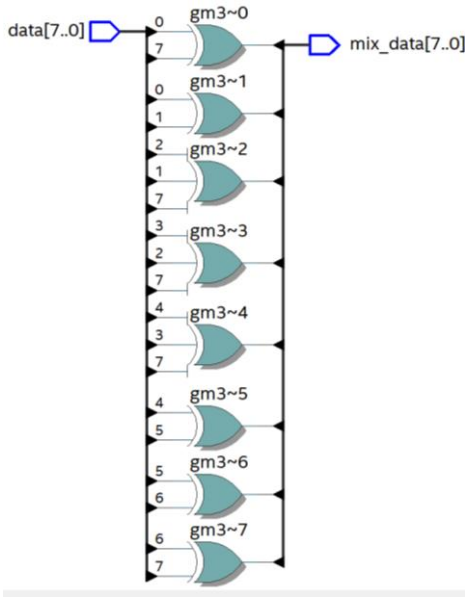

Рис. 4. Множення ненульового елемента поля $GF(2^8)$ на 3 по модулю $\gamma(x)$

Відповідно для інших коефіцієнтів із вектору v враховуючи (3) і (4) рівняння, що описують відповідні комбінаційні схеми, із врахуванням правил перетворень у $GF(2^8)$ матимуть вигляд:

$$k_4(w_{i,j}) = k_2(k_2(w_{i,j})) \quad (5)$$

$$k_5(w_{i,j}) = k_4(w_{i,j}) \oplus w_{i,j} \quad (6)$$

$$k_6(w_{i,j}) = k_2(k_3(w_{i,j})) \quad (7)$$

$$k_7(w_{i,j}) = k_6(w_{i,j}) \oplus w_{i,j} \quad (8)$$

$$k_8(w_{i,j}) = k_2(k_4(w_{i,j})) \quad (9)$$

Тоді згідно (1) враховуючи (3-9) рівняння (2) приймає наступний вигляд:

$C_j =$

$[w_{0,j} \oplus w_{1,j} \oplus k_5(w_{2,j}) \oplus w_{3,j} \oplus k_8(w_{4,j}) \oplus k_6(w_{5,j}) \oplus k_7(w_{6,j}) \oplus$

$k_4(w_{7,j})\ k_4(w_{0,j}) \oplus w_{1,j} \oplus w_{2,j} \oplus k_5(w_{3,j}) \oplus w_{4,j} \oplus k_8(w_{5,j}) \oplus k_6(w_{6,j}) \oplus$

$k_7(w_{7,j})\ k_7(w_{0,j}) \oplus k_4(w_{1,j}) \oplus w_{2,j} \oplus w_{3,j} \oplus k_5(w_{4,j}) \oplus w_{5,j} \oplus k_8(w_{6,j}) \oplus$
$k_6(w_{7,j})\ k_6(w_{0,j}) \oplus k_7(w_{1,j}) \oplus k_4(w_{2,j}) \oplus w_{3,j} \oplus w_{4,j} \oplus k_5(w_{5,j}) \oplus w_{6,j} \oplus$
$k_8(w_{7,j})\ k_8(w_{0,j}) \oplus k_6(w_{1,j}) \oplus k_7(w_{2,j}) \oplus k_4(w_{3,j}) \oplus w_{4,j} \oplus w_{5,j} \oplus k_5(w_{6,j}) \oplus$
$w_{7,j}\ w_{0,j} \oplus k_8(w_{1,j}) \oplus k_6(w_{2,j}) \oplus k_7(w_{3,j}) \oplus k_4(w_{4,j}) \oplus w_{5,j} \oplus w_{6,j} \oplus$
$k_5(w_{7,j})\ k_5(w_{0,j}) \oplus w_{1,j} \oplus k_8(w_{2,j}) \oplus k_6(w_{3,j}) \oplus k_7(w_{4,j}) \oplus k_4(w_{5,j}) \oplus w_{6,j} \oplus$
$w_{7,j}\ w_{0,j} \oplus k_5(w_{1,j}) \oplus w_{2,j} \oplus k_8(w_{3,j}) \oplus k_6(w_{4,j}) \oplus k_7(w_{5,j}) \oplus k_4(w_{6,j}) \oplus w_{7,j}\ ]$(10)

Таким чином ми отримали рівняння (10), що описує комбінаційну схему яка реалізовує перетворення *MixColumn* в ДСТУ 7624:2014. Також швидке обчислення вектору C_j можна реалізувати за правилом:

$$Q^T = T_0[w_0] \oplus T_1[w_1] \oplus T_2[w_2] \oplus T_3[w_3] \oplus T_4[w_4] \oplus T_5[w_5] \oplus T_6[w_6] \oplus T_7[w_7], \quad (11)$$

де

$$T_0[\alpha] = \begin{pmatrix} 01 \\ 04 \\ 07 \\ 06 \\ 08 \\ 01 \\ 05 \\ 01 \end{pmatrix} * \pi_0[\alpha], T_1[\alpha] = \begin{pmatrix} 01 \\ 01 \\ 04 \\ 07 \\ 06 \\ 08 \\ 01 \\ 05 \end{pmatrix} * \pi_1[\alpha],$$

$$T_2[\alpha] = \begin{pmatrix} 05 \\ 01 \\ 01 \\ 04 \\ 07 \\ 06 \\ 08 \\ 01 \end{pmatrix} * \pi_2[\alpha], T_3[\alpha] = \begin{pmatrix} 01 \\ 05 \\ 01 \\ 01 \\ 04 \\ 07 \\ 06 \\ 08 \end{pmatrix} * \pi_3[\alpha],$$

$$T_4[\alpha] = \begin{pmatrix} 08 \\ 01 \\ 05 \\ 01 \\ 01 \\ 04 \\ 07 \\ 06 \end{pmatrix} * \pi_0[\alpha], \qquad T_5[\alpha] = \begin{pmatrix} 06 \\ 08 \\ 01 \\ 05 \\ 01 \\ 01 \\ 04 \\ 07 \end{pmatrix} * \pi_1[\alpha],$$

$$T_6[\alpha] = \begin{pmatrix} 07 \\ 06 \\ 08 \\ 01 \\ 05 \\ 01 \\ 01 \\ 04 \end{pmatrix} * \pi_2[\alpha], \qquad T_7[\alpha] = \begin{pmatrix} 04 \\ 07 \\ 06 \\ 08 \\ 01 \\ 05 \\ 01 \\ 01 \end{pmatrix} * \pi_3[\alpha],$$

Модуль реалізації функції нелінійного перетворення приведено на рис. 5.

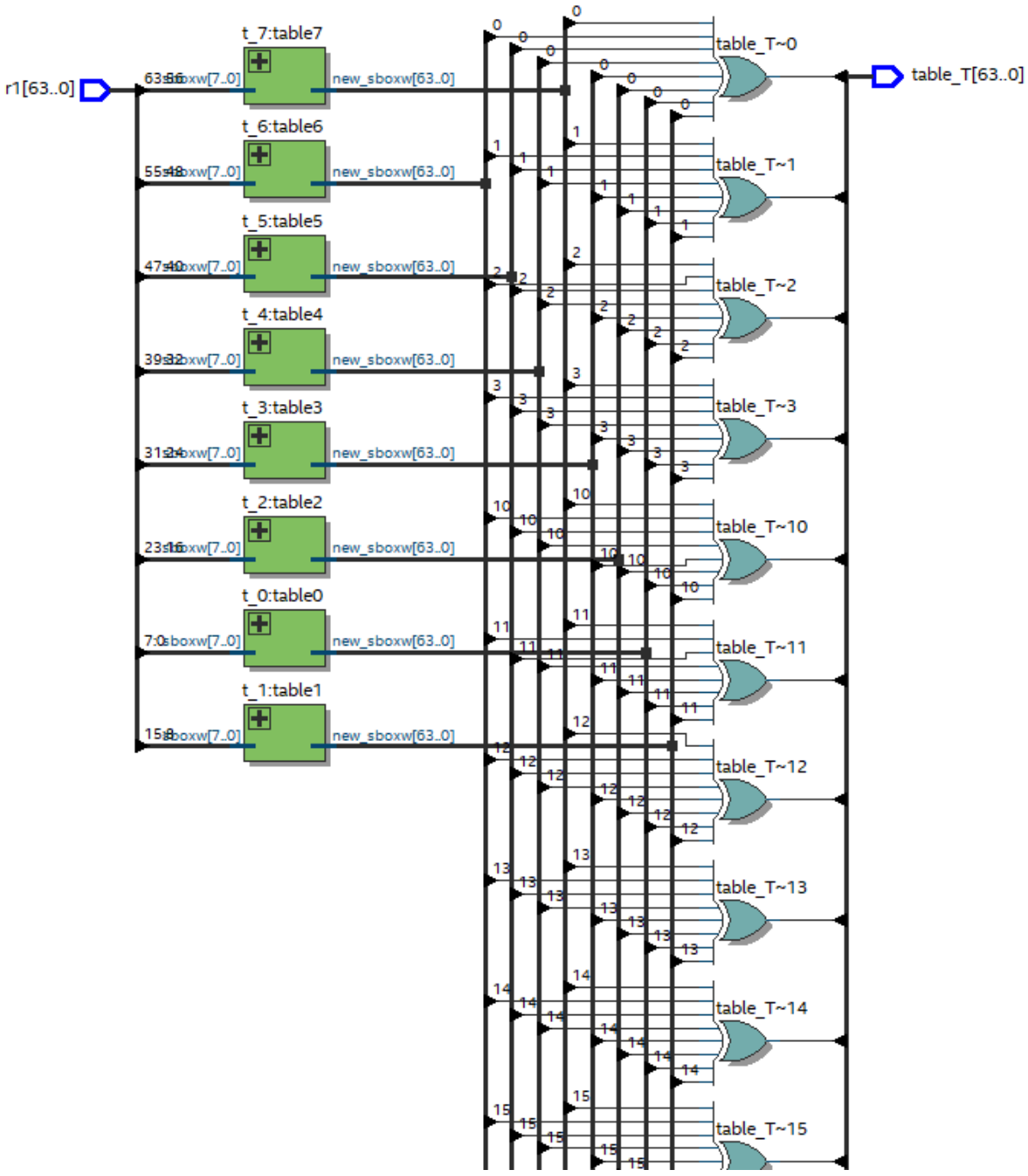

Рис. 5 Апаратно реалізований модуль функції нелінійного перетворення T

Застосування таблиць-констант $T_i[\alpha]$, i = 0,1,...,7 дозволяє значно зменшити кількість операцій, зокрема функція нелінійної підстановки обчислюється за сім операцій XOR над 64-бітовими рядками.

Для керування режимами роботи регістра зсуву із лінійним зв'язком використано скінчений автомат, графічне представлення якого приведено на рис. 6.

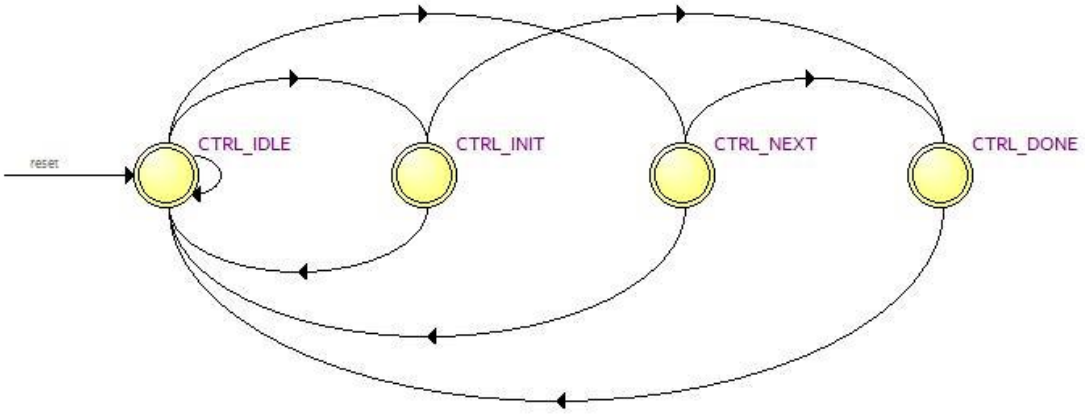

Рис. 6. СА керування регістром зсуву.

Модуль верхнього рівня апаратної реалізації алгоритму симетричного потокового перетворення ДСТУ 8845:2019 приведено на рис. 7. Verilog HDL код модулів розробленої апаратної реалізації алгоритму симетричного потокового перетворення ДСТУ 8845:2019 приведено в додатку.

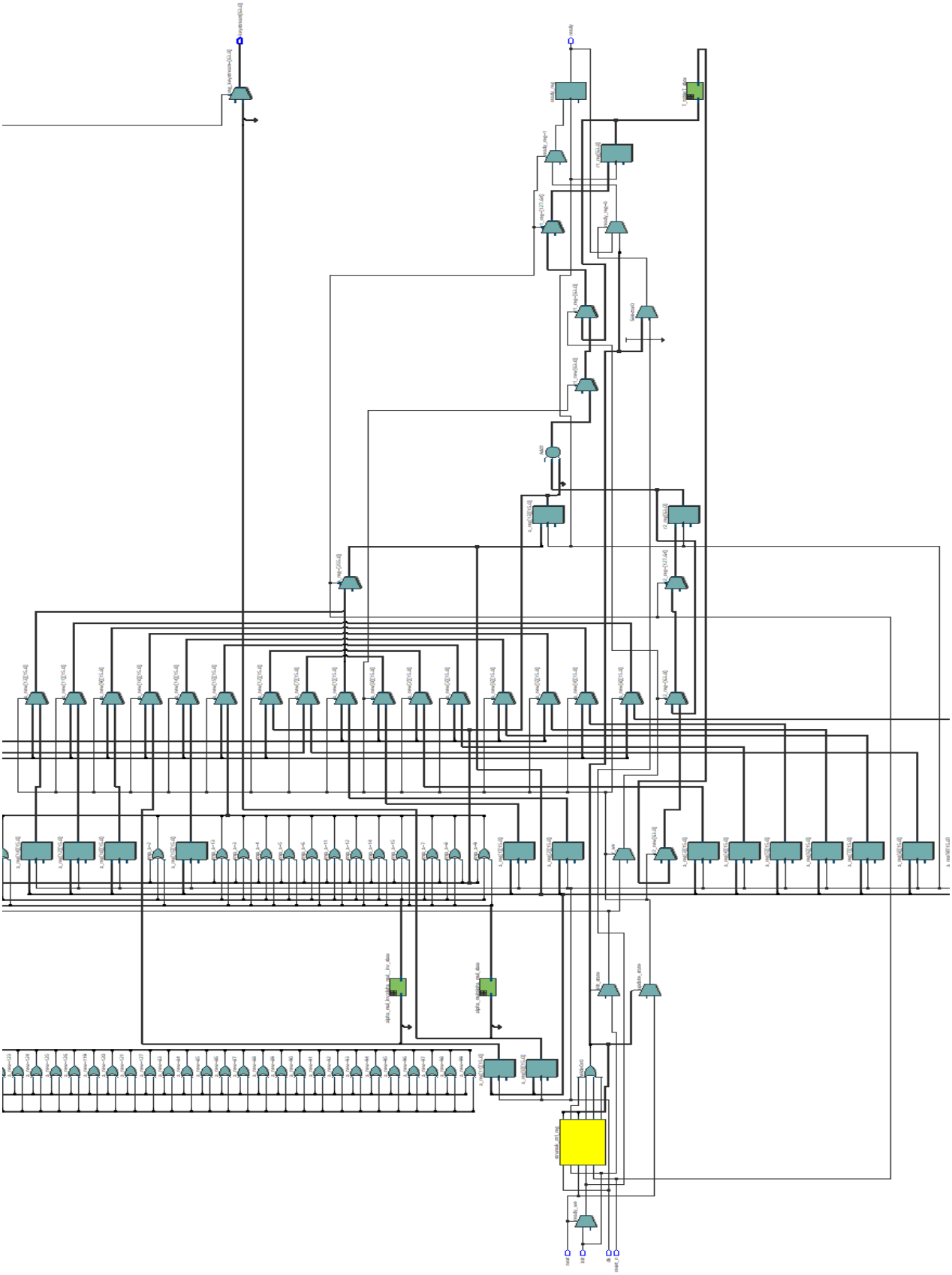

Рис. 7 Апаратна реалізація алгоритму симетричного потокового перетворення ДСТУ 8845:2019

Висновки

У результаті виконання отримано наступні результати:

В роботі на основі алгоритму симетричного потокового перетворення ДСТУ 8845:2019 реалізовано шифратор на базі програмованої логікової інтегральної мікросхеми (ПЛІС) із використанням мови опису апаратури Verilog HDL.

Розроблено та реалізовано на програмованій логіковій інтегральній мікросхемі множення двох многочленів по модулю $x^8 + x^4 + x^3 + x^2 + 1$ у вигляді передобчислених комірок модулів постійної запам'ятовуючої пам'яті, що уможливлює виконання операції MixColumn для функції нелінійної підстановки T за один такт. Виконання функції множення на α та α^{-1}в арифметиці поля Галуа $GF(2^{64})$ реалізовано на базі асинхронних постійних запам'ятовуючих пристроїв та комбінаційної логіки. Контроль режимів роботи регістру зсуву з лінійним оберненим зв'язком виконано на базі скінченого автомата. Керування вихідним ключовим потоком та елементами оберненого зв'язку виконано на базі комбінаційної схеми.

Список використаної літератури

1. A new SNOW stream cipher called SNOW-V / Patrik Ekdahl, Thomas Johansson, Alexander Maximov, Jing Yang

2. Дипломна робота на здобуття ступеня бакалавра на тему: Алгебраїчна атака на двійкові SNOW2.0-подібні потокові шифри / Овчарова М.А., Олексійчук А.М..// К.: 2019

3. Gorbenko I., Kuznetsov A., Lutsenko M. and Ivanenko D. The research of modern stream ciphers // 4th International Scientific-Practical Conference Problems of Infocommunications. Science and Technology (PIC S&T), Kharkov, 2017, pp. 207-210.

4. Дослідження кросплатформних реалізацій потокових симетричних шифрів / Кузнецов О.О., Фроленко В.О., Єрьомін Є.С., Іваненко Д.В. // Радіотехніка 2018. Випуск 193, ст. 94-106.

5. Совин Я.Р. Ефективна реалізація алгоритму блокового симетричного шифрування ДСТУ 7624:2014 («Калина») для 8/16/32-бітових вбудованих систем / Я.Р. Совин, В.І. Отенко, Є.Ф. Штефанюк // Сучасний захист інформації №2(30), 2017 - ст. 6-16.

6. A Survey of Lightweight Cryptography Implementations / [T. Eisenbarth, S. Kumar, C. Paar et al] // IEEE Design & Test of Computers – Special Issue on Secure ICs for Secure Embedded Computing. – 2007. – Vol. 24, Nr. 6. – pp. 522-533.

7. Performance Analysis of Contemporary Light-Weight Block Ciphers on 8-bit Microcontrollers / Rinne S., Eisenbarth T., Paar C. // ECRYPT Workshop Software Performance Enhancement for Encryption and Decryption. – 2007. – pp. 33-43.

8. ДСТУ 7624:2014. Інформаційні технології. Криптографічний захист інформації. Алгоритм симетричного блокового перетворення. – Введ. 01–07–2015. – К.: Мінекономрозвитку України, 2016.

9. Кузнецов А. Моделирование перспективного блочного шифра «Калина» / А.А. Кузнецов, Д.В. Иваненко, Е.П. Колованова // Прикладная радиоэлектроника. – Том 13. -№ 3. -2014, ст. 201-207.

10. Долгов В.І. Криптографічні властивості зменшеної версії шифра «Калина» / В.І. Долгов, Р.В. Олійников, А.Ю. Большаков, А.В. Григорьєв, Е.В. Дроботько // Прикладна радіоелектроніка: наук.-техн. журнал. – 2010. Том 9. №3. –С. 349-354

11. Математична структура потокового шифру «Струмок» / Кузнецов О.О., Горбенко І.Д., Горбенко Ю.І., Олексійчук А.М., Тимченко В.А. // Радіотехніка 2018. Випуск 193, ст. 17-27

12.ДСТУ 8845:2019. Інформаційні технології. Криптографічний захист інформації. Алгоритм симетричного потокового перетворення. – Введ. 01–10–2019. – К.: ДП «УкрНДНЦ», 2019.

13.ISO/IEC 18033-4:2011. Information technology – Security techniques – Encryption algorithms – Part 4: Stream ciphers. On-line]. Internet: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=54532 [Dec., 2012].

14. Kuznetsov O., M. Lutsenko and D. Ivanenko, "Strumok stream cipher: Specification and basic properties," 2016 Third International Scientific-Practical Conference Problems of Infocommunications Science and Technology (PIC S&T), Kharkiv, 2016, pp. 59-62.

Додатки

Verilog код ядра ДСТУ 7624:2014

```
module strumok_core(
                input wire           clk,
                input wire           reset_n,
                input wire           init,
                input wire           next,
                input wire [511 : 0]  key,
                input wire [255 : 0]  iv,
 input wire           key_length,
                output wire [63 : 0]  keystream,
                output wire           ready
               );
 //----------------------------------------------------------------
 // Internal constant and parameter definitions.
 //----------------------------------------------------------------

 localparam NO_UPDATE    = 3'h0;
 localparam INIT_UPDATE  = 3'h1;
 localparam SBOX_UPDATE  = 3'h2;
 localparam MAIN_UPDATE  = 3'h3;
 localparam FINAL_UPDATE = 3'h4;
 localparam CTRL_IDLE        = 3'h0;
 localparam CTRL_INIT        = 3'h1;
 localparam CTRL_NEXT        = 3'h2;
 localparam CTRL_FSM_UPDATE  = 3'h3;
 localparam CTRL_LFSR_UPDATE = 3'h4;
 localparam CTRL_OUTPUT      = 3'h5;
 localparam CTRL_DONE        = 3'h6;
 //----------------------------------------------------------------
```

```
// Registers including update variables and write enable.
//----------------------------------------------------------------
reg         ready_reg;
reg         ready_new;
reg         ready_we;

reg [15 : 00]  a_reg [0 : 63];
reg [15 : 00]  a_new [0 : 63];
reg          a_we;
reg         lfsr_we;

reg [63 : 00] r1_reg;
reg [63 : 00] r1_new;
reg [63 : 00] r2_reg;
reg [63 : 00] r2_new;
reg [63 : 00] reg_keystream;
reg         r_we;

reg [02 : 00] strumok_ctrl_reg;
reg [02 : 00] strumok_ctrl_new;
reg         strumok_ctrl_we;

reg [63 : 0] tmp0, tmp1, tmp2;
reg [64:0] r1_r15, r2_r13;
 reg  ready_next_init;
 reg  [63 : 00] tmp_a;
 reg  [05 : 00]  next_count;
//----------------------------------------------------------------
// Wires.
//----------------------------------------------------------------
```

```
wire [63 : 0] T_in, T_out, alpha_in, alpha_inv_in, r0_alpha, r11_alpha_inv;

reg  [63 : 0] r_15;
reg  [63 : 0] r_13;
reg  [63 : 0] r_11;
reg  [63 : 0] r_0;
 reg init_state;
reg update_state;
   //----------------------------------------------------------------
//Concurent connectivity for ports etc.
//----------------------------------------------------------------
  assign ready        = ready_reg;
assign T_in         = r1_reg;
assign alpha_in     = r_0;
assign alpha_inv_in = r_11;
assign keystream    = reg_keystream;

//----------------------------------------------------------------
//Module instations.
//----------------------------------------------------------------
T_table T_sbox(.r1(T_in),
               .table_T(T_out)
              );
alpha_mul alpha_mul_sbox(.sboxw(alpha_in),
               .alpha(r0_alpha)
              );
alpha_mul_inv alpha_mul__inv_sbox(.sboxw(alpha_inv_in),
               .alpha_mult_inv(r11_alpha_inv)
              );
//----------------------------------------------------------------
```

```
// reg_update
//----------------------------------------------------------------
always @ (posedge clk)
  begin: reg_update
    integer i;

    if (!reset_n)
      begin
        for (i = 0 ; i < 16 ; i = i + 1)
          begin
            a_reg[i] <= 64'h0;

          end

        r1_reg         <= 64'h0;
        r2_reg         <= 64'h0;
        ready_reg      <= 1'b1;
        strumok_ctrl_reg <= CTRL_IDLE;
      end
    else
      begin
        if (r_we)
          begin
            r1_reg <= r1_new;
            r2_reg <= r2_new;

          end

        if (a_we)
          begin
```

```
          for (i = 0; i < 16 ; i = i + 1)
            a_reg[i] <= a_new[i];
        end


      if (ready_we)
        ready_reg <= ready_new;

      if (strumok_ctrl_we)
        strumok_ctrl_reg <= strumok_ctrl_new;
    end
  end // reg_update


//----------------------------------------------------------------
// lfsr_logic
//----------------------------------------------------------------
always @*
  begin : lfsr_logic
    integer i;



    for (i = 0 ; i < 16 ; i = i + 1)
      begin
        a_new[i] = 64'h0;

      end

    r_15 <= a_reg[15];
```

```
            r_13 <= a_reg[13];
            r_11 <= a_reg[11];
            r_0  <= a_reg[00];

  tmp_a <= 64'h0;
  ready_next_init <= 1'h0;
            next_count <= 6'h0;


if (init_state)
              begin
                ready_next_init <= 1'h1;
                    if(key_length)
                          begin
                                a_new[15] = ~key[063 : 000];
                                a_new[14] =  key[127 : 064];
                                a_new[13] = ~key[191 : 128];
                                a_new[12] =  key[255 : 192];
                                a_new[11] =  key[063 : 000];
                                a_new[10] = ~key[127 : 064];
                                a_new[09] =  key[191 : 128];
                                a_new[08] =  key[255 : 192];
                                a_new[07] = ~key[063 : 000];
                                a_new[06] = ~key[127 : 064];
                                a_new[05] =  key[191 : 128]^iv[255 : 192];
                                a_new[04] =  key[255 : 192];
                                a_new[03] =  key[063 : 000]^iv[191 : 128];
                                a_new[02] =  key[127 : 064]^iv[127 : 064];
                                a_new[01] =  key[191 : 128];
                                a_new[00] =  key[127 : 064]^iv[063 : 000];
```

```
                a_we      =  1'h1;
            end
        else
            begin
                a_new[15] =  key[063 : 000];
                a_new[14] = ~key[127 : 064];
                a_new[13] =  key[191 : 128];
                a_new[12] =  key[255 : 192];
                a_new[11] = ~key[511 : 447];
                a_new[10] =  key[319 : 256];
                a_new[09] = ~key[383 : 320];
                a_new[08] =  key[319 : 256]^iv[255 : 192];
                a_new[07] = ~key[063 : 000];
                a_new[06] =  key[127 : 064];
                a_new[05] =  key[191 : 128]^iv[191 : 128];
                a_new[04] =  key[255 : 192];
                a_new[03] =  key[319 : 256]^iv[127 : 064];
                a_new[02] =  key[319 : 256];
                a_new[01] =  key[383 : 320];
                a_new[00] =  key[511 : 447]^iv[063 : 000];
                a_we      = 1'h1;
            end
end

    if (update_state)
        begin
            for (i = 0 ; i < 15 ; i = i + 1)
                a_new[i] = a_reg[(i + 1)];
            a_new[15] = tmp_a;
            a_we      = 1'h1;
```

```
                end


        if (ready_next_init)
                begin
                        next_count <= next_count + 1;
                end


        if (&next_count[4:0])
                begin
                ready_next_init <= 1'h0;
                        tmp_a <=  r0_alpha ^ r11_alpha_inv ^ r_13;
                        reg_keystream <= tmp2 ^ r_0;
                end
        else
                begin

                        tmp_a <=  tmp2 ^ r0_alpha ^ r11_alpha_inv ^ r_13;
                        reg_keystream <= 64'h0;
    end
  end

//-----------------------------------------------------------------
// fsm_logic
//-----------------------------------------------------------------
always @*
  begin : fsm_logic
   reg [63 : 0] tmp0, tmp1, tmp2;
         reg [64:0] r1_r15, r2_r13;
```

```
 r1_new <= 64'h0;
 r2_new <= 64'h0;
 r_we   <= 1'h0;


r1_r15 <= r1_reg + r_15;
       tmp0   <= r1_r15[63:0];

       r2_r13 <= r2_reg + r_13;
       tmp1   <= r2_r13[63:0];

       tmp2   <= tmp0^r2_reg;


 if (init_state)
   begin
     r1_new <= 64'h0;
     r2_new <= 64'h0;
     r_we   <= 1'h1;
   end

 if (update_state)
   begin
     r1_new <= tmp1;
     r2_new <= T_out;
     r_we   <= 1'h1;
   end
```

```
  end
//-----------------------------------------------------------------
// output_logic
//-----------------------------------------------------------------
always @*
  begin : output_logic
    if (init_state)
      begin
      end
    if (update_state)
      begin
      end
  end
//-----------------------------------------------------------------
// strumok_core_ctrl
//-----------------------------------------------------------------
always @*
  begin: strumok_core_ctrl
    ready_new      = 1'h0;
    ready_we       = 1'h0;
    init_state     = 1'h0;
    update_state   = 1'h0;
    strumok_ctrl_new = CTRL_IDLE;
    strumok_ctrl_we  = 1'h0;

    case(strumok_ctrl_reg)
      CTRL_IDLE:
        begin
          if (init)
            begin
```

```
        ready_new      = 1'h0;
        ready_we       = 1'h1;
                                  init_state     = 1'h1;
        strumok_ctrl_new = CTRL_INIT;
        strumok_ctrl_we  = 1'h1;
      end

    if (next)
      begin
                                update_state   = 1'h1;
        ready_new      = 1'h0;
        ready_we       = 1'h1;
        strumok_ctrl_new = CTRL_NEXT;
        strumok_ctrl_we  = 1'h1;
      end
  end


CTRL_INIT:
  begin
    strumok_ctrl_new = CTRL_DONE;
    strumok_ctrl_we  = 1'h1;
  end


CTRL_NEXT:
  begin
    strumok_ctrl_new = CTRL_DONE;
    strumok_ctrl_we  = 1'h1;
  end
```

```
/*
        CTRL_FSM_UPDATE:
          begin
            strumok_ctrl_new = CTRL_DONE;
            strumok_ctrl_we  = 1'h1;
          end
        CTRL_LFSR_UPDATE:
          begin
            strumok_ctrl_new = CTRL_DONE;
            strumok_ctrl_we  = 1'h1;
          end

*/
        CTRL_DONE:
          begin
            ready_new      = 1'h1;
            ready_we       = 1'h1;
            strumok_ctrl_new = CTRL_IDLE;
            strumok_ctrl_we  = 1'h1;
          end

        default:
          begin
          end
      endcase // case (strumok_ctrl_reg)
    end // strumok_core_ctrl
endmodule // strumok_core
// EOF strumok_core.v
```