Хмельницький національний університет
Факультет інформаційних технологій
Кафедра комп'ютерної інженерії та інформаційних систем

КВАЛІФІКАЦІЙНА РОБОТА МАГІСТРА

Галузь знань 12 – Інформаційні технології

Спеціальність 123 –Комп'ютерна інженерія

на тему «Метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA»

КвРКІП. 0180178.21.21 ПЗ

Виконав: студент 2 курсу, група КІ2м-21-1 ________ Юрчук А.О.
Підпис Ініціали, прізвище

Керівник к.т.н., доцент ________ Бобровнікова К.Ю.
Науковий ступінь, вчене звання Підпис Ініціали, прізвище

До захисту допускаю:
Зав. кафедри КІІС, д.т.н., проф.
Т.О. Говорущенко
11 05 2023 р.

Хмельницький, 2023

ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

Факультет Інформаційних технологій

Кафедра Комп'ютерної інженерії та інформаційних систем

Освітній рівень магістр

Галузь знань 12 Інформаційні технології

Спеціальність 123 Комп'ютерна інженерія

Освітня програма освітньо-наукова програма «Комп'ютерна інженерія та програмування»

ЗАТВЕРДЖУЮ

Зав. кафедри Т.О.Говорущенко

"01" 09 2022 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ МАГІСТРА

Юрчуку Андрію Олексійовичу

Прізвище, ім'я, по батькові студента

1. Тема проекту (роботи) Метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

Керівник проекту (роботи) Бобровнікова К.Ю., к.т.н., доцент

Прізвище, ім'я, по батькові, науковий ступінь, вчене звання

Затверджена наказом ректора університету від 09.01.2023 р. № 1

2. Строк подання студентом проекту (роботи) на кафедру 01.05.2023 р.

3. Вихідні дані до проекту (роботи) Завдання на дипломне проектування

4. Зміст пояснювальної записки (перелік питань, які потрібно розробити)

Аналіз відомих методів побудови архітектур для релізації aes шифрування

Модель програмно-апаратної архітектури для реалізації AES шифрування

Метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

Розроблення програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

5. Перелік графічного матеріалу (із зазначенням обов'язкових креслень)

6. Консультанти розділів кваліфікаційної роботи магістра

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
Нормоконтроль	Лисенко С.М., професор кафедри КІІС		
Антиплагіат	Нічепорук А.О., доцент кафедри КІІС		

7. Дата видачі завдання « 06 » 09 2022р.

КАЛЕНДАРНИЙ ПЛАН

№з/п	Назва етапів (розділів) кваліфікаційної роботи магістра	Термін виконання етапів проекту (роботи)	Примітка
1	Вибір напряму дослідження та узгодження тематики КвРМ з керівником	05.09.2022	виконано
2	Ознайомлення з предметною областю; формулювання мети та задач дослідження; визначення об'єкта та предмета дослідження	05.10.2022	виконано
3	Робота над розділом 1 – аналіз відомих моделей, методів за темою; постановка задачі	05.11.2022	виконано
4	Робота над розділом 2 – розробка моделей для вирішення поставленої задачі	05.12.2022	виконано
5	Робота над науковою статтею	05.01.2023	виконано
6	Робота над розділом 3 – розробка методів для вирішення поставленої задачі	15.02.2022	виконано
7	Робота над розділом 4 – проектування та розробка ПЗ для вирішення поставленої задачі, експериментальна частина	05.04.2023	виконано
8	Оформлення пояснювальної записки згідно вимог	15.04.2023	виконано
9	Попередній захист ДРМ	18.04.2023	виконано
10	Захист ДРМ на засіданні ЕК	До 10.05.2023	

Студент Підпис А.О. Юрчук Ініціали, прізвище

Керівник роботи К.Ю. Б[illegible]нікова

РЕФЕРАТ

Тема кваліфікаційної роботи магістра: Метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Автор роботи: Андрій Юрчук

Керівник роботи: Бобровнікова К.Ю.

Пояснювальна записка: 76 с., 13 рис., 4 табл., 2 дод., 80 джерел.

FPGA, АРХІТЕКТУРА, ПРОГРАМНО-АПАРАТНИЙ ЗАСІБ, АРХІТЕКТУР СИСТЕМИ, РЕАЛІЗАЦІЯ AES ШИФРУВАННЯ.

Об'єктом дослідження є апаратне AES шифрування на основі FPGA.

Предметом дослідження є метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Метою кваліфікаційної роботи магістра є підвищення швидкодії AES шифрування шляхом реалізації програмно-апаратної архітектури на основі FPGA.

Для розв'язання поставлених задач використовуються основні положення теорії комп'ютерних мереж та систем, системного аналізу, моделювання, методів аналізу даних, теорії математичної статистики, теорії дискретної математики.

Наукова новизна отриманих результатів:

– удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA, який на відміну від відомих використовує FPGA, і який забезпечує підвищення швидкодії AES шифрування.

– набули подальшого розвитку програмно-технічні засоби AES шифрування шляхом реалізації програмно-апаратної архітектури на основі FPGA.

В результаті виконаного наукового дослідження буде розроблено апаратно-програмні засоби AES шифрування шляхом реалізації програмно-апаратної архітектури на основі FPGA.

ЗМІСТ

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ 10

ВСТУП 11

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ПОБУДОВИ АРХІТЕКТУР ДЛЯ РЕЛІЗАЦІЇ AES ШИФРУВАННЯ 14

1.1 Дослідження високопродуктивних архітектур для релізації AES шифрування 14

1.2 Методи композитного поля S-Box 16

1.3 Технології композитного поля S-Box і конвеєрна підтримка 17

1.4 Мінімальні апаратні архітектури для AES 19

1.5 Спільне використання підструктури 20

1.6 Методи нормальної основи 21

1.7 Архітектури Galois/Counter Mode 21

1.8 Однокристальна криптографія на основі FPGA 24

1.4 Висновки а постановка задачі 27

2 МОДЕЛЬ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ 29

2.1 Математичні передумови побудови програмно-апаратної архітектури для реалізації AES шифрування 29

2.1.1 Скінченні поля 29

2.1.2 Основні поняття 29

2.1.3 Розширення скінченних полів 32

2.2 Складені поля. 33

2.3 Представлення базису 33

2.4 Стандартний базис 34

2.5 Нормальний базис 36

2.6 Подвійний базис 37

2.7 Ізоморфізм поля 38

2.8 Композитні поля, застосовані до AES 39

2.9 Висновки 46

3 МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA 48

3.1 Основи методу побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA 48

3.2 Архітектура системи 48

3.3 Апаратне проектування компонентів 48

3.3.1 Ключовий графік AES 49

3.3.2 S-box AES 57

3.3.3 GHASH 59

3.3.4 GCM 61

3.4 Організація системи 63

3.5 Висновки 72

4 РОЗРОБЛЕННЯ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA 74

4.1 Вибір типу архітектури та зразків проектування 74

4.2 Продуктивність апартного забезпечення 75

4.3 Проєктування програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA 77

4.4 Розробка системного програмного забезпечення 80

4.4.1 Процес збірки 80

4.4.2 Завантаження прошивки та виконання 83

4.5 Висновки 84

ВИСНОВКИ 85

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ 87

ДОДАТОК А Копія публікації 96

ДОДАТОК Б Презентація 107

ДОДАТОК В Лістинг програмного забезпечення 129

СКОРОЧЕННЯ ТА УМОВНІ ПОЗНАКИ

AES - Advanced Encryption Standard

FPGA - Програмована користувачем вентильна матриця

ВСТУП

Актуальність шифрування даних набуває все більшої важливості у світі, де технології стають все більш розповсюдженими та доступними.

Шифрування даних є важливою мірою для захисту конфіденційної інформації, такої як фінансові та медичні записи, особисті дані, бізнес-таємниці, інтелектуальна власність, тощо.

У сучасному світі, де деякі з найбільших компаній збирають та обробляють огромні обсяги особистих даних, забезпечення захисту цих даних стає важливішим, ніж будь-коли раніше.

Важливість шифрування даних ще більше зростає в контексті зростаючої кількості кібератак, які спрямовані на викрадення чутливої інформації та злому систем захисту даних.

Шифрування даних допомагає забезпечити конфіденційність, цілісність та доступність даних, знижуючи ризики їх несанкціонованого доступу та зміни. Також, шифрування даних може допомогти забезпечити відповідність різноманітним нормативним вимогам, які стосуються захисту особистих даних, таких як Загальний регламент про захист персональних даних (GDPR).

Таким чином, шифрування даних є невід'ємною складовою захисту інформації та захисту конфіденційної інформації в цифровому світі, де конфіденційність даних є все більш важливою для захисту прав та свобод людей.

AES (Advanced Encryption Standard) є одним з найбільш поширених та надійних алгоритмів симетричного шифрування. Алгоритм AES використовується для захисту даних в різних областях, таких як банківські операції, комунікації через Інтернет, зберігання даних на пристроях з пам'яттю, тощо.

AES (Advanced Encryption Standard) є одним з найбільш популярних симетричних алгоритмів шифрування, який використовується для захисту конфіденційної інформації в різних сферах, включаючи банківську, фінансову, комунікаційну, інформаційну безпеку та інші.

Ось деякі з користей застосування AES шифрування:

Конфіденційність: AES шифрування дозволяє зберегти конфіденційність даних шляхом перетворення їх у незрозумілий для сторонніх користувачів формат. Зашифровані дані можуть бути розшифровані тільки з використанням відповідного ключа.

Надійність: AES шифрування є одним з найбільш надійних і стійких до атак методів шифрування. Ключ AES складається з декількох раундів шифрування, які змінюють дані і перетворюють їх у незрозумілий формат.

Гнучкість: AES шифрування може бути використане в різних сценаріях застосування, таких як шифрування файлів, електронної пошти, мережевого трафіку та іншого.

Швидкість: AES шифрування є швидким і легким для використання, що дозволяє захистити велику кількість даних за короткий час.

Міжнародний стандарт: AES шифрування є міжнародним стандартом, прийнятим в усьому світі для захисту конфіденційної інформації. Це забезпечує сумісність між різними системами і дозволяє забезпечити захист даних в усіх країнах.

Актуальність роботи полягає в розробці методу побудови програмно-апаратної архітектури для реалізації AES шифрування на основі застосування FPGA.

Апаратна реалізація шифрування AES є дуже актуальною в наш час, коли є запит на обробку великої кількості даних, які потребують шифрування та збереження конфіденційності.

Отже, апаратна реалізація шифрування AES має велике значення для захисту конфіденційності та забезпечення безпеки даних, що є особливо важливим у світі, де інформація стає все більш цінною та цифрова безпека стає все більш актуальною.

Метою кваліфікаційної роботи магістра є підвищення швидкодії AES шифрування.

Поставлена мета досягається розв'язанням таких основних задач:

– дослідити методи синтезу апаратно-програмних засобів реалізації AES шифрування на основі FPGA;

– проаналізувати сучасні програмно-технічні засоби реалізації AES шифрування на основі FPGA

– розробити метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA;

– реалізувати метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Об'єктом дослідження є апаратне AES шифрування на основі FPGA.

Предметом дослідження є метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Наукова новизна отриманих результатів:

1. набув подальшого розвитку метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA, який на відміну від відомих використовує FPGA, і який забезпечує підвищення швидкодії AES шифрування.

2. набули подальшого розвитку програмно-технічні засоби AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA.

Практична цінність отриманих результатів.

В результаті виконаного наукового дослідження буде розроблено апаратно-програмні засоби AES шифрування шляхом реалізації програмно-апаратної архітектури на основі FPGA.

Для розв'язання поставлених задач використовуються основні положення теорії комп'ютерних мереж та систем, теорії архітектури комп'ютерних систем, системного аналізу, моделювання, методів аналізу даних, теорії математичної статистики, теорії дискретної математики.

За темою кваліфікаційної роботи магістра опублікована одна стаття у фаховому науковому виданні ВОТТП [1].

1 АНАЛІЗ ВІДОМИХ МЕТОДІВ ПОБУДОВИ АРХІТЕКТУР ДЛЯ РЕЛІЗАЦІЇ AES ШИФРУВАННЯ

1.1 Дослідження високопродуктивних архітектур для релізації AES шифрування

Існує велика кількість опублікованої літератури щодо високоефективних конструкцій або методів AES реалізації. З точки зору найвищого рівня, більшість високопродуктивних конструкцій конвеєрні або мають розгорнуту круглу структуру AES.

Один із найперших AES проектів, які заявляли про високу продуктивність, використовували структуру таблиці пошуку замінити цілий раунд AES [1].

Цей підхід вперше був запропонований в [1], які посилалися на ці апаратні LUTs як T таблиці.

Нещодавно T-таблиці стали називати T-боксами.

Як зазначено в [2], необхідними витратами на цю операцію є чотири таблиці пошуку T-box на 256 записів. Кожен запис T-box має довжину в одне слово або 4 байти, що дає загалом 8 кілобіт для всіх T-boxів, необхідних для одного раунду.

Підхід T-box був покращеним у порівнянні з попередньою високопродуктивною реалізацією тих самих авторів, які використовували aLUTдля S-Box [3].

Друге впровадження T-box було виконано [2]. Ця конкретна реалізація використовувала 32-бітний шлях даних і мав 128-бітний ключ із автономним розкладом ключів.

Як у всіх AES Реалізації T-box потребують великої пам’яті для досягнення максимальної продуктивності.

Реалізації T-box, які працюють на 128-бітових блоках, пропонують високу пропускну здатність.

Реалізації з використанням 32-розрядних блоків можуть бути економічнішими з точки зору використання ресурсів, але мають меншу пропускну

здатність [3].

Ця методологія T-box була застосована до ASIC за [4].

Загальною технікою покращення продуктивності ітерованого алгоритму є розгортання кількох раундів.

Це призводить до усунення накопиченої затримки від мультиплексорів і регістрів, які зазвичай керують циклічними циклами.

Результатом застосування цієї методики є дублювання обладнання для кожного розгорнутого раунду, що створює великий критичний шлях.

Чим більше критичний шлях, тим нижчою повинна бути тактова частота [5].

Конвеєрні реалізації були детально досліджені великою кількістю дослідників.

Конвеєрність можна розглядати як покращення стратегії розгортання. Ця техніка збільшує кількість даних, які можна обробляти одночасно, вставляючи регістри між незалежними апаратними модулями, що дозволяє обробляти непов’язані дані в кожному модулі.

Рівень паралелізму даних призводить до збільшення пропускної здатності для реалізації за рахунок затримки для окремого блоку.

Регістри, розміщені між модулями, змушують збільшити вартість ресурсів вище, ніж потрібно для стратегії розгорнутого циклу проектування.

Регістри можуть бути розміщені між раундами для конвеєрування між раундами та/або між окремими операціями раундів для конвеєрування всередині раундів. Конвеєрне облаштування всередині циклу також називають стратегією проектування підконвеєрів [5].

У конвеєрних конструкціях зазвичай використовують онлайн-розклад ключів.

У цьому типі розкладу ключів ключ розширюється паралельно з операціями шифрування або дешифрування, якщо це необхідно. Це усуває потребу в додатковій пам'яті розширення ключів.

Офлайн-розклад ключів попередньо обчислює всі круглі ключі до початку

операцій шифрування або дешифрування.

Хоча це гарантує, що всі круглі ключі будуть доступні, коли це необхідно в процесі шифрування або дешифрування, це також вимагає наявності пам'яті для зберігання кожного раундового ключа.

В [6] подано порівняння кількох конвеєрних реалізацій на пристрої Xilinx Virtex II, що відрізняються кількістю розгортання, розділенням круглої трансформації та технологією S-Box.

Результати цих різних реалізацій показали пряме впровадження S-Box уFPGAлогіка, яка дає неоптимальні результати за всіма вимірюваними показниками.

В [10] використано п'ятиетапний конвеєр для реалізації логіки шифрування з онлайн-розкладом ключів. S-Box був реалізований за допомогою FPGA.

1.2 Методи композитного поля S-Box

В [7] автор припускає, що розрахунок інверсії поля Галуа в перетворенні SubBytes можна було б звести до необхідних операцій у полях Галуа нижчого порядку.

Використовуючи цю техніку, можливо реалізувати схему для інверсії поля Галуа.

Перша робота щодо застосування цієї методології до AESS-Box був [8].

У дослідженні методологія композитного поля була використана для обчислення інверсії поля Галуа, а також MixColumns iAddRoundKeyоперації в GF((24)2).

Операції поля Галуа нижчого порядку значно зменшили кількість воріт арифметичних операцій кінцевого поля.

В [9] більш детально ілюструє, як ця техніка складеного поля використовується для реалізації S-Box.

1.3 Технології композитного поля S-Box і конвеєрна підтримка

Згідно [9] впровадження для anASIC, [33] застосував підхід композитного поля GF((24)2) до anAESреалізація шифрування на Xilinx Virtex IIПЛІС. Перетворення Sub- Bytes і MixColumns були зіставлені в складене поле.

Круглі константи, які використовуються в розкладі ключів, також були зіставлені в складене поле.

В [10] реалізовано AES шифрування за допомогою 128-бітного ключа та 128-бітного шляху даних.

Використовувалась міжраундна конвеєрна передача, яка потребувала унікального блоку пам’яті після операцій кожного відповідного розгорнутого раунду.

Додаткова велика пам'ять, необхідна для реєстрації пам'яті після кожного раунду стверджувати, що техніка композитного поля дозволить реалізувати таку конструкцію в менших пристроях за рахунок зменшення потреби в додатковій пам’яті S-Box.

Слідом за роботою в [11-15] показано, що оптимально використовувати методи складених полів лише в перетворенні SubBytes.

В [16] також представлено покращений розклад ключів, розроблений для конвеєрного підходу всередині раунду з використанням складеного поля GF((24)2).

Однією з головних цілей конвеєрної обробки є балансування затримки на кожному етапі.

Етапи конвеєра з найбільшою затримкою обмежуватимуть продуктивність усього конвеєра.

З цією метою створено вежу складених полів GF(((22)2)2). Використовуючи цю техніку, можна розбити обчислення для інверсії поля Галуа в перетворенні SubBytes на менші компоненти.

Ці менші одиниці дизайну ідеально підходять для конвеєрної реалізації, оскільки перетворення SubBytes має найбільшу затримку будь-якого окремого

етапу, коли реалізовано за допомогою таблиці пошуку.

В [17] реалізовано метод, який використовує цю стратегію, відомий як субконвеєрний дизайн із збалансованими етапами.

В [18] використано цю стратегію для створення високопродуктивної конвеєрної конструкції на Xilinx FPGA.

Ретельний аналіз 16 різних конструкцій GF(((22)2)2) проведено в [19], дає оптимальний вибір констант незвідних поліномів.

Як згадувалося раніше, було використано складене поле GF((24)2) у розрахунку S-Box. У дослідженні також обговорювалися три підходи до обчислення інверсії поля Галуа у цьому полі:

1) безперервне розкладання складеного поля на GF(((22)2)2);

2) з використанням алгоритму квадрата та множення;

3) обчислення комбінаційних рівнянь для кожного біта. Було виявлено, що пряме обчислення комбінаційних рівнянь для кожного біта призвело до зменшення довжини критичного шляху та загальної кількості вентилів.

В [19] описано затримку критичного шляху, виявлену в реалізації композитного поля S-Box, використовуючи одиниці попереднього обчислення. Перший представлений дизайн замінює інверсний компонент GF(24) і останні два множники GF(24) на 2 набори множників GF(24), по одному набору для кожного з чотирьох бітів у остаточному GF(28) S-Box вихідне значення. Вхідними даними для цих множників є 1) постійні обернені значення в GF(24) і 2) значення, обчислені під час другого етапу традиційного складеного поля S-Box. За рахунок удвічі більшого використання площі ця конструкція зменшує критичний шлях на 20%.

В [20] подано опис компоненту зворотного афінного перетворення на менші підперетворення GF(24), що виконуються на виході кожного з 32 множників GF(24). Автор стверджує, що якби було виконано внутрішньораундове укладання трубопроводу, для першого проекту знадобилося б лише 3 етапи трубопроводу та 2 етапи трубопроводу для другого проекту замість 5 етапів трубопроводу, які використовуються в традиційному підході.

1.4 Мінімальні апаратні архітектури для AES

Розглянемо методи, спрямовані на реалізацію компактного обладнання.

У такій реалізації пропускна здатність приноситься в жертву замість зниження вимог до обладнання [21].

Це відповідає меншій кількості транзисторів. В FPGA компактна апаратна архітектура використовує менше реконфігураційної логіки, ніж стандартна або високопродуктивна реалізація.

Ітерована кругла структура AES дозволяє створити компактну конструкцію, просто перебираючи обладнання, необхідне для реалізації одного раунду алгоритму.

Щоб мінімізувати вимоги до обладнання, деякі компактні реалізації виконують розклад ключів онлайн [22].

Використовуючи нове обладнання з більшими характеристиками BRAM, зміг реалізувати T-box лише в двохБРАМклітини.

Отже, ця реалізація використовувала невелику кількість реконфігурованої логіки, що призводить до більш високого співвідношення пропускної здатності до площі.

Проти такого підходує, який показав, що «апаратні реалізації MixColumns менші, ніж T-box, мають коротший критичний шлях і споживають менше енергії»[23, 24].

Ще один поширений прийом для зменшення вимог до AES полягає у використанні 8-бітних данихширина апата. Це призводить до значного зниження пропускної здатності.

В [25] використано меншу ширину шляху даних, щоб створити реалізацію низької області з різко зниженою пропускною здатністю. S-Box було дозволено перебувати всередині БРАМ.

В [26] використано метод для побудови ASIC перетворення MixColumns, яка

використовує новий метод, який обчислює один стовпець стану за сім тактових циклів. S-Box реалізовано за допомогою комбінаційної логіки.

в [27] також використовував цей підхід для ASIC з більш високою продуктивністю.

В [28] використано 8-бітний шлях даних для створення спеціального процесору інструкцій (ASIP) на Xilinx FPGA.

Додатковий FPGA для збереження площі досягається за допомогою спільного множника кінцевого поля в складеному полі SubBytes.

Всі AES операції, включаючи кроки MixColumns, виконуються з використанням ітеративної архітектури множення-накопичення. Використовуючи ці методи, в [29] подано використання меншої кількості реконфігураційної логіки, ніж будь-яка з попередніх реалізацій.

1.5 Спільне використання підструктури

Реалізація, яка підтримує як шифрування, так і дешифрування, здатна спільно використовувати підструктури між ними.

В [30] створено реалізацію низької області, яка використовує апаратне забезпечення перетворення MixColumns.

Відповідно до [31], подана концепція спільного використання підструктури MixColumns. Використовуючи цю ідею, перетворення InvMixColumns є просто перетворенням MixColumns з деякою попередньою обробкою.

Це знижує продуктивність алгоритму, але також зменшує вимоги до апаратного забезпечення.

У такому підході шифрування та дешифрування не можна використовувати паралельно.

Ключовий момент, який використовується в [32] було розроблено таким чином, що всі підключі були попередньо згенеровані та збережені в одному місці. Оскільки розклад ключів і модулі шифрування не використовувалися одночасно,

BRAM збережений S-Box ділився між двома за допомогою логіки перемикання.

Окрім спільного використання підструктури MixColumns, можна спільно використовувати апаратне забезпечення під час перетворення Sub- Bytes. Зокрема, інверсія поля Галуа ідентична між шифруванням і дешифруванням. Потрібна певна логіка перемикання, щоб створити шлях даних, який би будь-який

1) виконати зворотне афінне перетворення, потім інверсію поля Галуа (розшифрувати) або 2) виконати інверсію поля Галуа, потім афінне перетворення (шифрувати)[33, 34].

1.6 Методи нормальної основи

Іншим удосконаленням техніки композитного поля стала зміна її основи.[42] представлені елементи в нормальному базисі на кожному рівні у вежі полів GF(((22)2)2).

У дослідженні було досліджено 432 можливі декомпозиції, що дало найменшу апаратну реалізацію для цього конкретного композитного поля.

В [35] представлено використання методу, у якому не опускаються до найнижчого складеного поля, вибираючи натомість використання GF((24)2). Вартість цієї реалізації була порівнянна з витратами на попередню роботу.

В [36] реалізовано весь алгоритм AES у звичайній основі на anASIC.

Жоден із цих попередніх перерахованих методів нормальної бази не було реалізовано в FPGA.

1.7 Архітектури Galois/Counter Mode

Реалізації GCM досліджуються в цьому розділі виключно цільовимиAESяк алгоритм блокового шифрування. Ці проекти в першу чергу зосереджені на високій продуктивності, оскільки автентифіковане шифрування та дешифрування з GCM є відносно ефективними та розпаралелюваними [37].

В [38] подано конвеєрний ASIC реалізації GCM із пропускною здатністю 34 Гбіт/с за допомогою 128-бітного ключа.

Ітеративна реалізація AES використана для обчислення H, тоді як в основному використовувався круглий конвеєрний підхід GCM шлях шифрування даних. S-Box були реалізовані вLUTs.

Критичний шлях визначається на паралельному множнику скінченного поля Mastrovito, який використовується у функції GHASH.

ТрійкаASICGCMреалізації в [39] були завершені для всіх трьохAESрозміри ключів, і кожен мав меншу кількість воріт, ніж зазначено в [40].

В [41] пояснено це використанням складеного поля та абінарна діаграма рішень (BDD)методи впровадження S-Box і архітектурні зміни в GCM шлях даних.

Послідовна GCM архітектура оцінена з AES архітектурою, побудовані як 1) 4-ступенева конвеєрна петля та 2) кругла конвеєрна архітектура та паралельна GCM архітектура була побудована з чотирма ітеративними AES реалізації. Чотиритактовий множник GF(2128) використовується з послідовним 4-ступеневим AES реалізація конвеєрного циклу, яка потім забезпечує аналогічну пропускну здатність паралельного GCM реалізація на тій же тактовій частоті. Послідовна версія, однак, мала меншу кількість воріт.

Послідовний GCM з конвеєрною AES реалізацією досягла найвищої пропускної здатності з найкращим співвідношенням пропускної здатності до площі.

В [42] показано пропускну здатність, де використовується розпаралелювання функції GHASH, де вхідні дані чергуються в парних або непарних послідовностях.

Чотири паралельні компоненти GHASH використовуються разом із чотирма AES компонентами, де кожен конвеєрний.

Пропускна здатність перевищує 100 Гбіт/с із кращим співвідношенням пропускна здатність до площі.

Цю статистику співвідношення було покращено за рахунок спільного використання того самого планувальника ключів для всієї схеми.

В [43, 44] GHASH розбивається на чотири 128 біт на 32 біт конвеєрного цифрово-послідовного множення.

У поєднанні з конвеєром на 56 ступенів AES завдяки дослідженні досягається пропускна здатність 54,94 Гбіт/с із вищим співвідношенням пропускна здатність до площі, ніж бачили раніше.

Віртекс-4FPGAреалізаціїGCMбули досліджені в [45].

AES реалізації мали ширину шляху даних 128, 64, 32, 16 біт і підтримували всі три довжини ключа.

Оцінювалися реалізації S-Box: таблиці пошуку; підхід композитного поля, спадаючий у GF(24); і використання вбудованої BRAM.

Множник GF(2128) був реалізований як біт-паралель, цифра-послідовний і як гібрид, де складене поле GF((216)8) дозволяє 16-бітному множнику завершити множення.

У [46] виявлено, що гібридний множник був більшим і повільнішим, ніж цифровий послідовний підхідFPGA,ймовірно через обраний незвідний поліном для складеного поля. 128-бітний шлях данихGCMЯдро мало найвищу продуктивність понад 10 Гбіт/с. 16-розрядні версії datapath працювали зі швидкістю приблизно 2 Гбіт/с і використовували значно менше логіки, ніж версії з вищими datapath.

В [47] два високопродуктивні GCM проекти, націлені на Xilinx Virtex-4FPGAбули створені. Детальний аналіз складності за допомогою FPGA також подано примітиви.

Множник GF(2128) реалізований у бітовому паралельному режимі для продуктивності та з використанням алгоритму Карацуби для більш ефективного проектування.

AES реалізації націлені на 128-бітні ключі та конвеєрні з використанням підходу композитного поля для S-Box, щоб збалансувати час затримки з множником GHASH.

Окрема ітерація AES компонент використовується для обчислення H, але поділяє компонент планування ключа з основним конвеєрним шляхом шифру AES.

GCM що містить конвеєрний AES із множником на основі Karatsuba досяг 15 Гбіт/с, тоді як внутрішній раунд конвеєрний AES з бітовим паралельним множником досягла 20 Гбіт/с.

Поліпшення подано в [48], де використано 4-ступеневий конвеєрний множник на основі Карацуби-Офмана в компоненті GHASH.

Як зазначено в [49], цей підхід для мультиплікатора має меншу складність порівняно з підходом у [50],зменшення вимог до апаратної площі. BRAM, LUT і S-блоки, реалізовані на основі композитного поля, використовуються в різних реалізаціях.

В [51] досягнуто вищої максимальної робочої частоти, ніж їхня попередня робота в і пропускна здатність для покращення зрізу у три рази, порівнюючи реалізації Virtex-4 зAESпідтримує всі три ключі довжини.

На Virtex-4 найвища пропускна здатність становила 34 Гбіт/с; тоді як на Virtex-5 найвища досягнута пропускна здатність становила 39 Гбіт/с.

1.8 Однокристальна криптографія на основі FPGA

Концепція системи на одному чіпі не є новою. Розробники систем, маючи на меті скоротити ресурси, необхідні для впровадження цифрової системи, використовували ASIC реалізувати свої задуми.

Час і витрати, пов’язані зі створенням ASIC може значно перевищувати створення FPGA дизайн.

Вимоги до логічних ресурсів для реалізації всієї системи на FPGA збережений FPGA впровадження неможливо протягом деякого часу.

Крім того, відмовостійкі системи вимагають фізично відокремлених резервних компонентів.

В [52] подано аналіз надлишкових FPGA, де багато аналітиків безпеки вважали нерозв’язною проблемою.

Лише порівняно недавно FPGA виробники змогли виготовити інструменти,

які перевіряють таку конструкцію [53].

Розглянемо опису досліджень, проведених для створення безпеки FPGA примітивів та методи проектування, необхідні для створення безпечної апаратної реалізації.

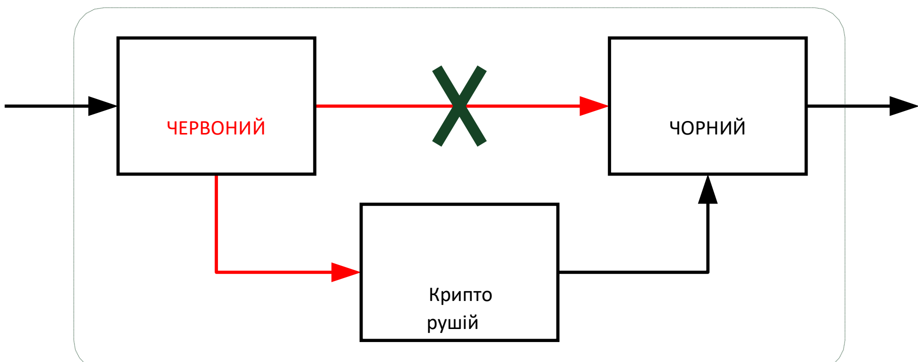


Рисунок 1.1: Фізично розділені чорна та червона підсистеми, які обмінюються даними лише за допомогою криптографічного механізму[54]

В [54] обговорено необхідні примітиви для надійно відокремленого дизайну в реконфігурованій системі.

Поняття рову та підйомного мосту вводяться для встановлення фізичної ізоляції та фізичних інтерфейсів, що піддаються статичній перевірці, відповідно, на одному реконфігурованому чіпі.

Представлена додаткова техніка, відома як реконфігураційне очищення, щоб переконатися, що інформація не залишається, яка може поставити під загрозу безпеку системи після часткової реконфігурації певного ядра.

У дослідженні ізольовані керни можуть використовувати лише маршрути, довжина яких не перевищує ширину рову.

Експеримент в [55] це усунення шестигранних і довгих ліній призвело до збільшення площі на 14,9% і збільшення затримки критичного шляху на 18,9% в середньому.

В [56] створено проект для перевірки практичності використання ровів, підйомних мостів і еталонних моніторів в реальній системі.

У Xilinx реалізовано два процесори MicroBlazeFPGA як червоно-чорна система.

У цій системі ресурси, в тому числі AES ядра шифрування, спільні на вбудованій периферійній шині (OPB).

Еталонний монітор використовується для управління доступом до кожного з периферійних пристроїв певним процесором.

Монітор може розташовуватися на ОПБ або між процесором і пам'яттю системи.

Монітор не можна обійти, і він повинен підтверджувати доступ до кожного периферійного пристрою.

Використовуючи монітор, можна спільно використовувати пристрій пам'яті між ядрами процесора.

Певний діапазон адрес пам'яті може бути дозволений для доступу до певного процесора.

Кожен процесор і еталонний монітор ізольовані ровом невикористаного CLB і має відповідні розвідні мости, по яких можуть проходити відповідні сигнали.

Оскільки кожен периферійний пристрій у прикладі системи є пристроєм із відображенням пам’яті, контрольний монітор керує доступом до діапазону пам’яті, що належить кожному периферійному пристрою.

Розмір рову безпосередньо впливає на кількість використаних CLB і еталонний монітор додає деякі незначні накладні витрати для кожного периферійного доступу.

І Altera, і Xilinx розробили програмні засоби та ПЛІС, які підтримують необхідні примітиви розділення.

В [57] описано структуру огорожі в Xilinx ПЛІС як набір невикористаних CLB, в якому може бути відсутня логіка чи маршрутизація. Цей блок використовується для створення ізольованих областей, через які доступ дозволено через макроси шини.

Використання більшої довжини прямо заборонено використання зав’язків,

якщо одна з можливих кінцевих точок зав'язків може бути в ізольованому регіоні.

Інструмент перевірки ізоляції (IVT) був розроблений для перевірки кожного ізольованого регіону [61].

Розділений дизайн (праворуч), який був заблокований на блоки в FPGA план поверху з відповідними інтерфейсами маршрутизації безпеки (зліва) [58].

Альтера ПЛІС дозволяє створювати захищену область, коли повністю оточують деяку логіку на кристалі.

Еквівалентна концепція макросу шини, що використовується в Xilinx ПЛІС є isi , що розміщені між захищеними регіонами.

SRI має кілька рівнів безпеки, які обмежують сигнали, які можуть передаватися між блоками.

У FPGA планувальник рівнів з відповідним ISI за допомогою інструментів Quartus II від Altera.

1.4 Висновки а постановка задачі

Таким чином, необхідним є підвищення швидкодії AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA.

Тому необхідним є розроблення методу побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Поставлена мета досягається розв'язанням таких основних задач:

1. дослідити методи синтезу апаратно-програмних засобів реалізації AES шифрування на основі FPGA;
2. проаналізувати сучасні програмно-технічні засоби реалізації AES шифрування на основі FPGA
3. розробити метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA;
4. реалізувати метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

2 МОДЕЛЬ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ

2.1 Математичні передумови побудови програмно-апаратної архітектури для реалізації AES шифрування

Розглянемо основні властивості та аспекти побудови програмно-апаратної архітектури для реалізації AES шифрування.

Для вирішення поставлених задача необхідним є залучення теорії скінченних полів, а також пов'язаної з ними арифметика полів.

Також необхідним є розгляд методів реалізації, зокрема опис маніпуляцій з двійковими даними у складених скінченних полях.

З огляду на це, розглянемо математику складених полів та ізоморфні відображення між представленням складеного поля і стандартним двійковим представленням.

2.1.1 Скінченні поля

Більшість обчислень в AES виконується над скінченним полем.

З цією метою зроблено спробу пояснити попередні та пов'язані з ним термінологічні поняття.

Тема скінченних полів дуже детально розглядається у [26] та [27].

Детальний опис композитних полів та ізоморфних відображень до композитних полів можна знайти в докторській дисертації [28] та [29].

2.1.2 Основні поняття

Для того, щоб визначити скінченне поле, необхідно визначити деяку термінологію та основні поняття.

Означення 1 [26]. Бінарною операцією над множиною S називається операція, яка відображає декартовий добуток, $S \times S$, двох елементів з S назад у множину S.

Означення 2 [26, 28]. Множина G, визначена за допомогою бінарної операції, називається групою, якщо вона має наступні властивості:

- Асоціативність бінарної операції: $(ab)c = a(bc), \forall\ a, b, c\ \in G$.
- Елемент ідентичності $e\ \in G$: $ae = ea = a$.
- Обернений елемент $a^{-1} \in G$ для будь-якого елемента $aa^{-1} = a^{-1}a = e$.
- G не є порожньою множиною.

Група називається абелевою, або комутативною, якщо $ab = ba, \forall\ a, b\ \in G$.

Близьким до поняття групи є поняття кільця.

На відміну від групи, обране кільце визначається двома бінарними операціями.

Важливо зазначити, що немає вимоги щодо чітко визначеної мультиплікативної інверсії для кожного елемента кільця.

Означення 3 [26]. Множина R, визначена за допомогою двох бінарних операцій: додавання(+) та множення(·), називається кільцем1 якщо вона підкоряється наступним властивостям:

- Асоціативність відносно кожної бінарної операції: $(ab)c = a(bc)$ та $a + (b + c) = (a + b) + c, \forall\ a, b, c\ \in R$.
- R – абелева група під доповненням з елементом тотожності 0.
- Обернений до доданка елемент $b\ \in R$ для будь-якого елемента $a \in R$: $a + b = b + a = 0$.
- Елемент тотожності для множення $1\ \in R$: $1 - a = a - 1 = a, \forall\ a\ \in R$.
- Дистрибутивність $\forall\ a, b, c\ \in R$: $a(b + c) = ab + ac = (b + c)a$.
- R не є порожньою множиною.

Використовуючи означення кільця, поле можна визначити з деякими додатковими властивостями.

Означення 4 [26]. Кільце R утворює поле, якщо воно підпорядковується наступним властивостям:

- Ненульові елементи R утворюють абелеву групу при множенні.
- Мультиплікативна тотожність елемента 1 не дорівнює адитивній тотожності елемента 0.

Кожне поле є інтегральною областю [26], що означає, що поле не має нульових дільників.

Це особливо корисно разом з визначенням операції ділення.

Це означає, що для кожного елемента поля існує єдиний обернений мультиплікатор.

Для роботи, виконаної в роботі, актуальними є розгляд лише скінченних полів.

Скінченне поле – це поле зі скінченною кількістю елементів, також відоме як поле Галуа.

Поле Галуа скорочено позначається як $GF(p^n)$ або $\mathbb{F}_{p^n}$, де n – натуральне число, а p – просте число. p^n вказує на порядок поля, а p – на характеристику поля [27].

Означення 5 [28]. Порядком поля називається кількість елементів у полі.

Означення 6 [28]. Характеристикою поля називається кількість додавань мультиплікативного елемента тотожності, необхідних для досягнення суми, еквівалентної адитивному елементу тотожності.

На додаток до обмеження уваги в цій дисертації полями Галуа, ці скінченні поля також будуть обмежені характеристикою двох.

Як зазначено в [28], кожен елемент буде вважатися власною адитивною інверсією.

Мотивуючим фактором тут є створення поля, яке можна легко представити булевими рівняннями, що визначають роботу цифрової логіки.

2.1.3 Розширення скінченних полів

Кільце цілих чисел позначається $\mathbb{Z}_n$, де n – модуль цілого числа.

Подібним чином можна створити кільце многочленів за модулем многочлена $f(x)$.

Таке кільце поліномів позначається $\frac{\mathbb{Z}_n[x]}{f(x)}$, де $f(x) \in \mathbb{Z}_n[x]$.

Важливо зазначити, що кільце цілих чисел $\mathbb{Z}_n$ стає скінченним полем, коли n є простим.

Застосовуючи ту саму логіку до кільця многочленів, можна дійти висновку, що для того, щоб $\frac{\mathbb{Z}_n[x]}{f(x)}$ було полем, мало того, що n має бути простим, так ще й модуль многочлена $f(x)$ має бути незвідним [29].

Такі незвідні поліноми існують для кожного скінченного поля з порядком простого степеня.

Теорема 1 [27, 26]. *Якщо поле F існує і містить кількість елементів, яка дорівнює степеню простого числа, то $\forall\, d \geq 1$, існує принаймні один незвідний многочлен степеня d над над полем F.*

У попередньому підрозділі було визначено $GF(p^n)$ як поле Галуа з модулем простого степеня. При $n = 1, GF(p)$ описує поле цілих чисел.

При $n > 1$ поле називається полем розширення $GF(p)$. І навпаки, $GF(p)$ є підполем, або основним полем, $GF(p^n)$.

Елемент поля розширення $GF(p^n)$ можна представити у вигляді полінома з коефіцієнтами у підполі.

Наприклад, многочлен $A(x) = a_{n-1}x^{n-1} + \cdots + a_0$ має коефіцієнти $a_i \in GF(p)$, де n – максимальний степінь незвідного многочлена $f(x)$ і $i = 0, 1, \ldots, n - 1$[29].

Максимальний степінь невизначеного x ніколи не буде більшим за $n - 1$, оскільки він зводиться до незвідного полінома $f(x)$ степеня n. p^n поліномів є

класами залишків за модулем $f(x)$ у $GF(p^n)$, з чого випливає, що алгоритми арифметики в полі будуть залежати від вибору $f(x)$[28].

2.2 Складені поля.

Окремим випадком полів розширення є складені поля, які позначаються як $GF((2^n)^m)$ для другої характеристики.

Означення 7 [28, 29]. Пара полів $\{GF(2^n),\ Q(y) = y^n + \sum_{i=0}^{n-1} q_i y^i\}$ і $\{GF((2^n)^m),\ P(x) = x^m + \sum_{i=0}^{m-1} p_i x^i\}$ називаються складеним полем, якщо. . .

- $GF(2^n)$ побудовано з $GF(2)$ до $Q(y)$.
- $GF(2^n)^m$ побудовано з $GF(2^n)$ до $P(x)$.

де $Q(y)$ та $P(x)$ – незвідні поліноми над $GF(2)$.

Якщо $k = nm$, то $GF(2^k)$ і $GF((2^n)^m)$ мають однаковий порядок.

Будь-які два поля з однаковим порядком ізоморфні до один одного [27].

Хоча ці два поля ізоморфні, алгоритми, що обчислюються над ними, можуть мати різну складність.

Зміна складності відбуватиметься по відношенню до визначених польових операцій: додавання і множення.

Подібно до того, як зміна в незвідному поліномі для поля розширення може призвести до зміни алгоритмічної складності, зміна в незвідних поліномах $Q(y)$ або $P(x)$ також може призвести до такої зміни для складеного поля.

У більш загальному сенсі, значення, вибрані для n і m, спричинять цю зміну через їхній вплив на незвідні поліноми поля [28].

2.3 Представлення базису

Хоча існує багато різних представлень базису, в прикладних дослідженнях криптографії найчастіше зустрічаються три:

- стандартний базис;
- нормальний базис;
- подвійний базис.

Вибір конкретного представлення базису визначає складність скінченної польової арифметики і будь-яких побудованих ізоморфізмів.

Буде показано представлення польових елементів і зроблено огляд для кожного з трьох базисів.

Базис скінченного поля – це множина елементів, які при лінійному об'єднанні утворюють кожен елемент скінченного поля.

Загальний базис $GF(p^m)$ показано у рівнянні 3.1, де a^i – коефіцієнт, x^i – базисний вектор, а $A(x)$ – елемент скінченного поля.

Кількість базисних елементів m прямо пропорційна степеню простого цілого числа що визначає основне поле:

$$A(x) = a_0x_0 + a_1x_1 + a_2x_2 + \ldots + a_{m-1}x_{m-1}, \text{де } a_i \in GF(q)\,. \tag{2.1}$$

2.4 Стандартний базис

Якщо скінченне розширення поля $\frac{GF(2^m)[x]}{R(x)}$ утворюється над $GF(2)$, то стандартний базис для розширення над ґрунтовим полем $GF(2)$ показано в рівнянні 3.2 за допомогою кореня x з $R(x)$.

Цей примітивний корінь x називається генератором.

$$\{1, x, x^2, \ldots, x^{m-1}\} \tag{2.2}$$

Кожен вектор у цьому базисі лінійно незалежний від усіх інших базисів. m елементів поля $GF(2^m)$ подано у вигляді поліномів, як показано у рівнянні 2.3. Кожен a_i є елементом базисного поля $GF(2)$.

Як і у випадку з будь-яким базисом, лінійна комбінація m базисних елементів дає кожен із скінченних елементів поля [28]:

$$A(x) = a_0 + a_1x + a_2x^2 + \ldots + a_{m-1}x^{m-1}, \text{ де } a_i \in GF(2) \quad (2.3)$$

Стандартний базис також відомий як поліноміальний базис або канонічний базис, особливо стосовно скінченних розширень полів.

Походження назви «поліноміальний базис» очевидне через прямий зв'язок між базисом і представленням елемента у вигляді полінома [28].

Арифметичні операції над скінченними полями з використанням стандартного базису виконуються за модулем полінома редукції поля.

Кожен коефіцієнт невизначеного в представленні елемента поля також зменшується за модулем основного поля.

Використовуючи поле характеристики два, додавання еквівалентне відніманню.

Це проста операція виключного АБО з коефіцієнтами в еквівалентних степенях невизначеного.

Множення та піднесення до квадрату є складнішими операціями з безліччю різних реалізацій зі специфічними перевагами.

Множення у стандартному базисі, у своїй найелементарнішій формі, виконується тими ж методами, що і звичайне множення многочленів.

Операція піднесення до квадрата включає в себе деякі операції зсуву, за якими слідують множення і операція виключного АБО.

2.5 Нормальний базис

Якщо над $GF(2)$ утворено таке саме скінченне розширення поля $\frac{GF(2^m)[x]}{R(x)}$, то нормальний базис для поля розширення над основним полем $GF(2)$ показано у рівнянні 2.4.

$$\{x, x^q, x^{q^2}, x^{q^3}, \ldots, x^{q^{m-1}}\}. \quad (2.4)$$

У цій множині примітивний елемент x підноситься до зростаючого степеня характеристики поля, позначеного q.

Знову ж таки, x також відомий як генератор для скінченного поля, оскільки він породжує всі елементи мультиплікативної групи.

Як зазначалося раніше, в цій дисертації розглядаються лише поля з характеристикою 2. У цьому випадку $q = 2$.

Кожен скінченний елемент поля визначається коефіцієнтами примітивних елементів поля, як показано в рівнянні 2.5.

Знову ж таки, кожен коефіцієнт зменшується на модуль поля ґрунту.

$$A(x) = a_0 x + a_1 x^{2^1} + a_2 x^{2^2} + \ldots + a_{m-1} x^{2^{m-1}}, \text{ де } a_i \in GF(q) \quad (2.5)$$

Як і в стандартному базисі, додавання в нормальному базисі є простим виключним АБО коефіцієнтів відповідного примітивного елемента.

На відміну від стандартного базису, множення є більш складним.

Перевагою нормального базису є те, що піднесення до квадрату є простим циклічним зсувом коефіцієнтів примітивних елементів.

2.6 Подвійний базис

У стандартному базисі елементи мали послідовну генераторну потужність.

У звичайному базисі елементи були послідовними генераторними експонентами.

Для будь-якого базису можна створити подвійний базис [30, 28]. Використовуючи елементи базису $\{x_0, x_1, \ldots, x_{m-1}\}$ з загального прикладу, нехай h – ненульова лінійна функція від $GF(q^m)$ до $GF(q)$.

Використовуючи цей базис, його подвійний базис можна визначити як $\{\gamma_0, \gamma_1, \ldots, \gamma_{m-1}\}$, де

$$h(x_i \gamma_i) = \begin{cases} 1 \text{ якщо } i = j \\ 0 \text{ якщо } i \neq j \end{cases}, \text{ де } 0 \leq i \text{ і } j \leq m - 1. \quad (2.6)$$

У [28] вказано лінійну функцію h від добутку базисних елементів, тоді як у [30] не вказано конкретну лінійну функцію.

Згідно з [30], перетворення з подвійним базисом є проблемою імпорту та експорту, коли необхідно ефективно змінити базис.

Методи з подвійним базисом дозволяють здійснювати перетворення, які є більш ефективними, ніж множення матриць, коли матриця є занадто великою для обмеженого середовища [30].

У певних реалізаціях, як, наприклад, у [31], підхід з подвійним базисом може призвести до меншої площі апаратного забезпечення і потенційно забезпечити високу продуктивність для польових операцій, оскільки розмір поля збільшується.

2.7 Ізоморфізм поля

Особливий інтерес у цій дисертації становить відображення між двійковим стандартним представленням і композиційним польовим представленням скінченних польових елементів.

Більше того, ці ізоморфізми дозволяють виконувати операції, які спочатку виконувались у $GF(2^8)$, у полях нижчого порядку, таких як $GF(2^4)$ або $GF(2^2)$. Задокументовано декілька методів побудови матриць перетворень.

Ізоморфізм – це бієктивне відображення, де функція відображення та обернена до неї є гомоморфізмами.

Простіше кажучи, це відображення між двома векторними просторами з двома умовами:

1) функція відображення є однозначною;

2) відображення зберігає структуру.

У контексті векторного простору вимога збереження структури означає, що відображення зберігає обидві польові операції: додавання і множення.

Векторні простори однакової розмірності ізоморфні.

Скінченне поле – це векторний простір над своїми підполями, який ізоморфний іншому скінченному полю того ж порядку.

Наприклад, розширенням скінченного поля $GF(p^k)$ є векторний простір розмірності k над $GF(p)$.

Розглянуте розширення буде ізоморфним до іншого скінченного поля з порядком p^k.

Для того, щоб побудувати ізоморфізм від стандартного представлення до представлення складеного поля, необхідно побудувати відображення між базисними елементами $GF(2^k)$ та $GF((2^n)^m)$ для $k = nm$.

Використовуючи позначення для незвідних многочленів, нехай α – примітивний корінь $P(x)$, а β – примітивний корінь незвідного многочлена $R(z)$ зі степенем k та коефіцієнтами у $GF(2)$.

Іншими словами, $R(x)$ є незвідним многочленом $GF(2^k)$ у двійковому стандартному поданні.

Відображення створюється з використанням значення t, яке задовольняє наступним умовам з [28]:

$$T\beta^i = \alpha^{it}, \text{де } i = 0, 1, \ldots, k - 1 . \quad (2.7)$$

Рівняння 2.7 – не єдина умова, якій має задовольняти значення t. Як було сказано на початку цього підрозділу, структура повинна зберігатися з ізоморфізмом.

Для збереження множення α^t має бути коренем з $R(z)$ по модулю незвідних поліномів $Q(y)$ і $P(x)$, як показано в рівнянні 2.8 [28].

$$R(\alpha^t) = 0 . \quad (2.8)$$

Знаходження значення t можна здійснити кількома способами, найочевиднішим з яких є перебір методом грубої сили [28].

Складність ізоморфізму безпосередньо пов'язана зі складними польовими примітивними поліномами.

2.8 Композитні поля, застосовані до AES

Використовуючи визначення AES S-Box та побудову композитного поля, можна побудувати S-Box, використовуючи поля Галуа нижчого порядку.

AES S-Box виконує інверсію поля Галуа та афінне перетворення.

Для реалізації обчислень інверсії поля Галуа використовується підхід композитного поля.

Щоб мати змогу використовувати підхід складеного поля для реалізації S-Box, необхідно згенерувати ізоморфне відображення між полем Галуа AES S-Box і бажаним складеним полем.

Рисунок 2.1 ілюструє три основні обчислення інверсії в полі Галуа $GF(2^8)$ з використанням методу композитного поля.

Оскільки афінне перетворення є нічим іншим, як лінійним перетворенням, за яким слідує трансляція, воно може бути інтегроване з ізоморфним відображенням, щоб зменшити кількість необхідних обчислень [32, 33].

Ці відображення будуються за допомогою описаних вище методів.

Розглянемо процес отримання інверсії $GF((2^4)^2)$.

Як було зазначено, елементи $GF(2^8)$ можуть бути виражені у вигляді полінома першого степеня з коефіцієнтами в $GF(2^4)$, $bx + c$, якщо задано незвідний поліном $x^2 + Ax + B$.

Саме при виконанні обчислень з цими коефіцієнтами може бути реалізовано скорочення необхідної логіки.

В [15] стверджується, що в $GF(2^4)$ існують незвідні многочлени, де $A = 1$ і B –константа, яка не порушує незвідність многочлена.

Використовуючи незвідний многочлен $P_2(x) = x^2 + x + \lambda$, обернену до $bx + c$ можна обчислити за допомогою коефіцієнтів у $GF(2^4)$.

Кроки цієї процедури адаптовано з [15] та [32].

Це обернення еквівалентне розв'язанню для $A(x)$ і $B(x)$ такого рівняння з [32]:

$$A(x)P_2(x) + B(x)S(x) = 1 \, mod \, P_2(x) \quad (2.9)$$

У цьому рівнянні $S(x) = bx + c$.

Таким чином, $B(x)$ є мультиплікативною оберненою до $S(x)$.

Рисунок 2.1 – Діаграма високого рівня, що показує три основні компоненти в обчисленні композитного поля S-Box, без урахування афінного перетворення [5] константа тут і далі позначається як λ

Коли незвідний многочлен $P_2(x)$ ділиться на $S(x)$, результат може бути виражений як добуток між $S(x)$ і частковим многочленом $Q(x)$, що помножений на многочлен $R(x)$, що залишився.

$$P_2(x) = Q(x)S(x) + R(x) \tag{2.10}$$

Використання довгого ділення для обчислення $Q(x)$ та $R(x)$ дає наступний результат:

$$Q(x) = b^{-1}x + (1 + b^{-1}c)b^{-1} \tag{2.11}$$

$$R(x) = \lambda + (1 + b^{-1}c)b^{-1}c \tag{2.12}$$

Підставивши рівняння 2.11 та 2.12 назад у рівняння 2.10, отримаємо рівняння 2.13:

$$P_2(x) = (b^{-1}x + b^{-1}(1 + b^{-1}c))S(x) + (\lambda + b^{-1}c(1 + b^{-1}c)) \tag{2.13}$$

Щоб вилучити від'ємні експоненти з рівняння 2.13, b^2 розподіляється по обидва боки рівняння.

$$b^2 P_2(x) = (bx + (b + c))S(x) + (b^2\lambda + bc + c^2) \tag{2.14}$$

$$b^2 P_2(x) = (bx + (b + c))S(x) + (b^2\lambda + c(b + c)) \tag{2.15}$$

Для того, щоб мати рівняння, яке можна порівняти з рівнянням 2.9, постійний член повинен бути еквівалентним одиниці.

Розподіл $(b^2\lambda + c(b + c))^{-1}$ на обидві частини рівняння досягає цієї мети.

$$(b^2\lambda + c(b + c))^{-1} \cdot b^2P_2(x) = (b^2\lambda + c(b + c))^{-1} \cdot (bx + (b + c))S(x) + (b^2\lambda + bc + c^2) \tag{2.16}$$

$$(b^2\lambda + c(b + c))^{-1} \cdot b^2P_2(x) = (b^2\lambda + c(b + c))^{-1} \cdot (bx + (b + c))S(x) + 1 \quad (2.17)$$

$$(b^2\lambda + c(b + c))^{-1} \cdot b^2P_2(x) + (b^2\lambda + c(b + c))^{-1} \cdot (bx + (b + c))S(x) = 1 \quad (2.18)$$

Між рівняннями 2.17 і 2.18 член $S(x)$ змінює сторону рівняння.

Знак тут не змінюється, оскільки це поле з характеристикою два.

Порівняння рівняння 2.18 з початковим рівнянням 2.9 показує, що коефіцієнт при $S(x)$ у рівнянні 2.18 є шуканою оберненою мультиплікативною величиною, $S^{-1}(x)$ або $(bx + c)^{-1}$:

$$S^{-1}(x) = (bx + c)^{-1} = (b^2\lambda + c(b + c))^{-1} \cdot (bx + (b + c)) \quad (2.19)$$

Безпосереднє відображення рівняння 2.19 на апаратні логічні компоненти показано на рисунку 2.2.

На цьому рисунку показано високорівневий вигляд взаємодії між компонентами складеного поля S-Box, що беруть безпосередню участь в інверсії поля Галуа.

Ця схема не є повною реалізацією S-Box, оскільки вона не включає афінне перетворення.

Блоки, позначені $\delta(x)$ та $\delta^{-1}(x)$, є ізоморфізмом та оберненим ізоморфізмом відповідно.

Компоненти на рисунку 2.2 між двома компонентами ізоморфізму належать до $GF(2^4)$.

У межах цієї компоненти слід зазначити, що інверсія поля Галуа все ще залишається потрібною.

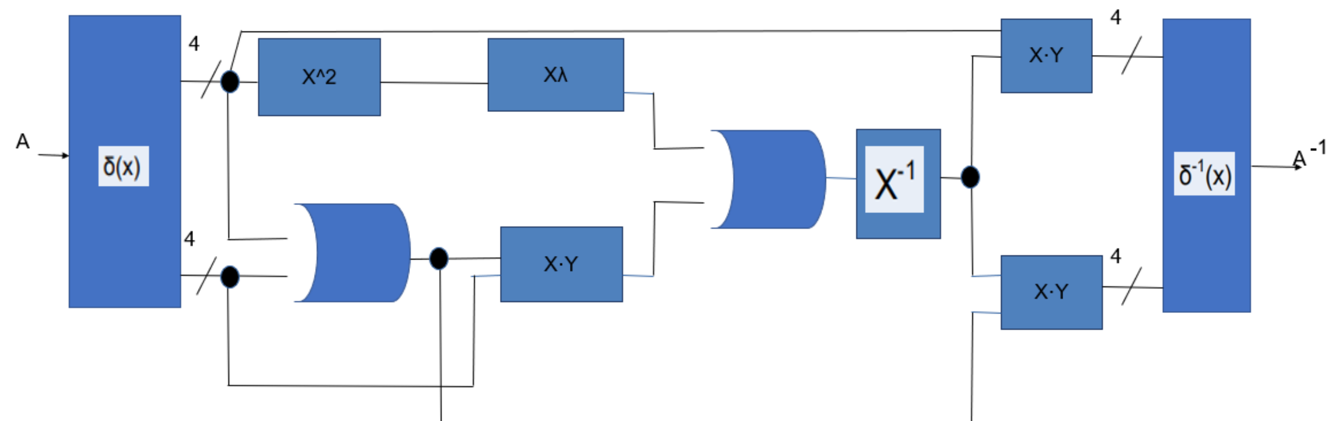


Рисунок 3.2 – Блок-схема, що показує окремі апаратні компоненти в композитній польовій реалізації S-Box [5]

Цей компонент, однак, може використовувати простий LUT, оскільки він потребує лише 16 елементів замість початкових 256 елементів, необхідних для повної реалізації S-Box в LUT.

Інші підходи, такі як продовження декомпозиції поля Галуа або піднесення до квадрата і множення, також можуть бути використані для вказаного компонента [32].

Безумовно, неперервна декомпозиція можлива для компонент у $GF(2^4)$, але вона може бути невигідною через алгоритмічну складність одного підходу порівняно з іншим на даній платформі.

Побудова $GF(2^4)$ з використанням складеного поля $GF((2)^2)$ вимагає визначення ще одного незвідного полінома поля, $P_1(x)$.

Константа, необхідна для того, щоб цей поліном був незвідним, визначена як φ.

$$P_1(x) = x^2 + x + \varphi \quad (2.20)$$

Використовуючи s як добуток a та b у $GF(2^4)$, операція множення виконується так, як показано нижче:

$$s = a \cdot b \tag{2.21}$$

Як і раніше, коли елементи в $GF(2^8)$ подано у вигляді рівняння першого степеня з коефіцієнтами в $GF(2^4)$, елементи в $GF(2^4)$ можна подати у вигляді рівняння першого степеня з коефіцієнтами в $GF(2)$.

Таке рівняння показано нижче, де старші біти позначено індексом H, а молодші біти – індексом L.

$$s_H x + s_L = (a_H x + a_L)(b_H x + b_L) \tag{2.22}$$

Цей добуток можна оцінити, а потім зменшити за допомогою полінома поля, як показано нижче:

$$s_H x + s_L = (a_H b_H)x^2 + (a_H b_L + a_L b_H)x + a_L b_L \tag{2.23}$$

$$s_H x + s_L = (a_H b_H)(x + \varphi) + (a_H b_L + a_L b_H)x + a_L b_L \tag{2.24}$$

$$s_H x + s_L = (a_H b_H + a_H b_L + a_L b_H)x + (a_H b_H \varphi + a_L b_L) \tag{2.25}$$

Рівняння 2.25 визначає множник $GF((2)^2)$, логічна апаратна реалізація якого показана на рисунку 2.3. Цей множник $GF((2)^2)$ можна розкласти на $GF(2)$, за бажанням.

Складність алгоритму цих складених полів безпосередньо пов'язана з вибором експонент при побудові складеного поля, через їх вплив на незвідні поліноми.

В роботі [35] проаналізовано побудови складених полів, які використовують всі значення λ і φ, що не призводять до зведеного поля полінома, що дає оптимальні значення для констант.

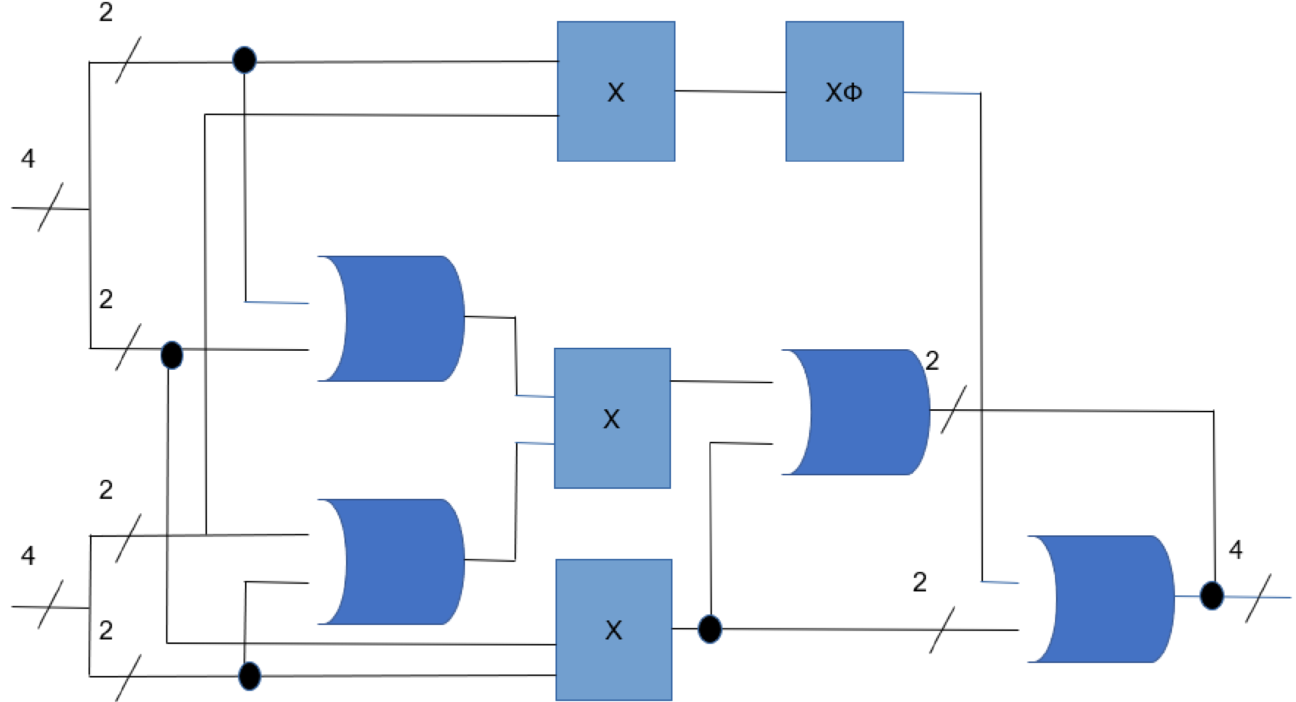


Рисунок 3.3 – Блок-схема, що показує окремі апаратні компоненти в $GF((2)^2)$ мультиплікатора [5]

2.9 Висновки

В розділі подано опис математичних передумов побудови програмно-апаратної архітектури для реалізації AES шифрування.

З цією метою представлено такі поняття, які застосовані до розроблення методу шифрування:

- скінченні поля;
- розширення скінченних полів;
- складені поля;
- представлення базису;
- стандартний базис;
- нормальний базис;
- подвійний базис;
- ізоморфізм поля;

- композитні поля.

3 МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

3.1 Основи методу побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

3.2 Архітектура системи

З метою вирішення поставленої задачі було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Розглянемо кроки методу та основні аспекти функціонування кожного апаратного компонента.

Після того, як дизайн компонентів визначено, організація систем спрямована на високу продуктивність, малу площу та баланс між цими двома показниками. Дизайн систем спочатку обговорюється на загальному рівні, спільному для кожної цільової метрики.

Також представлено інтерфейс і програмне забезпечення для верифікації, після чого слідує розділ, що описує роботу, виконану за допомогою інструментів безпеки електронного проектування і автоматизації (EDA), наданих корпорацією Altera.

3.3 Апаратне проектування компонентів

У цьому розділі описано проектування апаратних компонентів найнижчого рівня у дослідженні.

3.3.1 Ключовий графік AES

Перший графік ключів, який буде реалізовано в дослідженні, було обрано для автономного режиму з шириною тракту даних 128 біт.

Розклад ключів AES розроблено таким чином, що кожне 32-бітне слово залежить від попереднього 32-бітового слова, що виключає можливість паралельної генерації слів.

Для створення 128-бітного шляху даних необхідно згенерувати чотири 32-бітних слова протягом одного такту.

Таким чином, більш пізні слова в раунді розкладу ключів будуть проходити більше комбінаційної логіки, ніж попередні слова, перш ніж будуть зареєстровані в кінцевому підсумку.

Цей ключовий розклад було змодельовано на основі 128-бітової схеми передачі даних, представленої в [56].

Ця схема показана на рисунку 3.1. Оскільки проект, представлений в [56], був орієнтований на підхід ASIC, він був модифікований, щоб краще відповідати архітектурі цільової ПЛІС. Цю модифіковану схему показано на рисунку 3.2.

У цій модифікованій версії розкладу ключів приватний ключ зберігається безпосередньо в регістрах, позначених як $t0 - t7$.

Це зменшує необхідну кількість регістрів на вісім.

Автомат, який керує розкладом ключів, залишає входи мультиплексора на регістрах переключеними таким чином, щоб до них можна було записувати дані, поки автомат не працює.

Дозволи регістрів – це гарячі кодовані входи на верхньому рівні розкладу клавіш, які повинні бути дозволені, коли записується відповідне ключове слово.

Ці сигнали складаються за схемою АБО з сигналами дозволу регістрів, що використовуються під час звичайних операцій розширення ключів.

Оскільки вхідні регістри було вилучено, було також вилучено чотири двовхідні 32-розрядні мультиплексори, які були входами до верхніх видалених регістрів.

Розклад ключів у [56] також не міг би забезпечити цілий 128-бітний блок кожного раунду розкладу ключів для всіх довжин ключів, як це було зображено на рисунку.

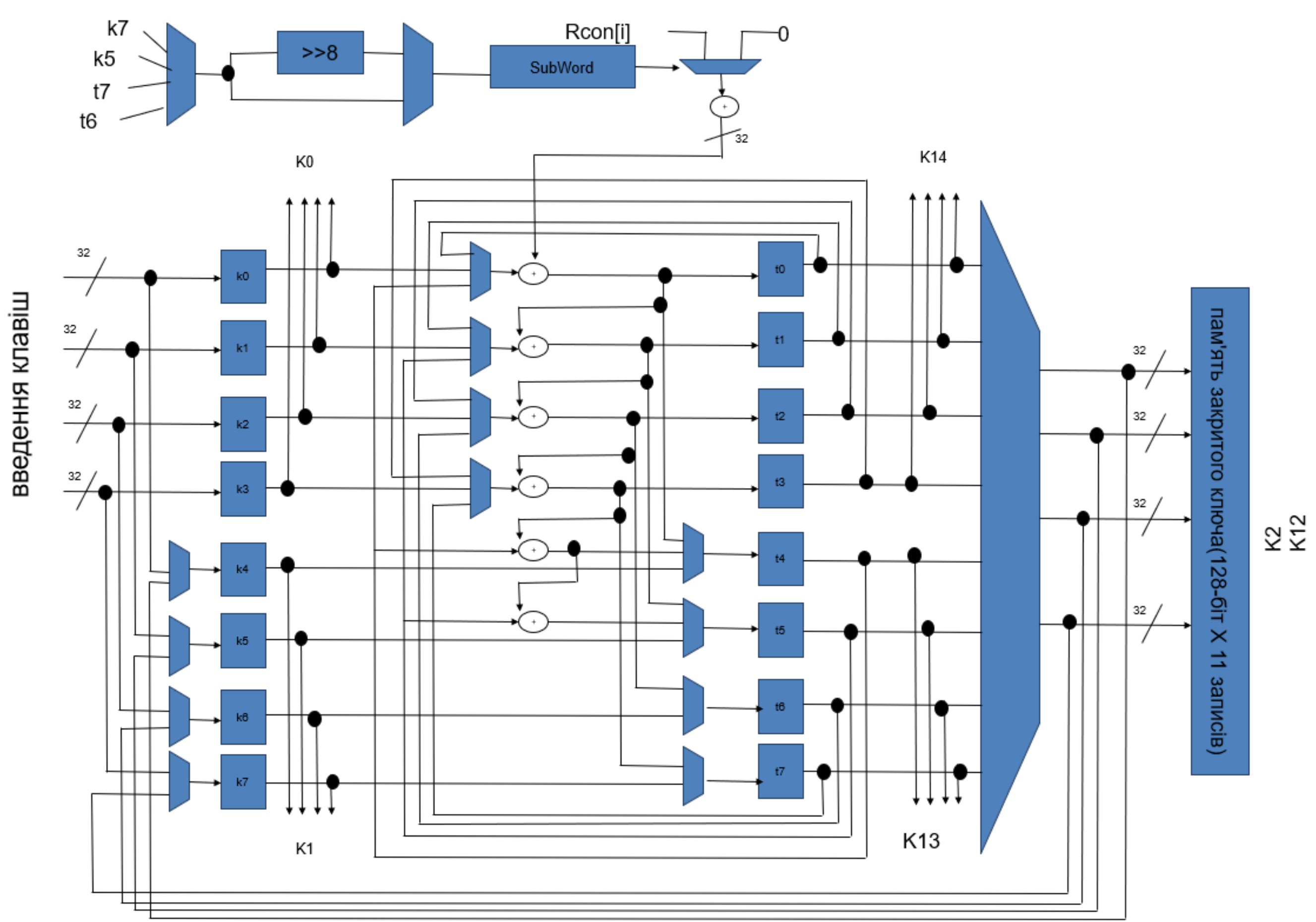


Рисунок 3.1 – Розклад ключів траєкторії даних шириною 128 біт з [56]

Частково це було пов'язано з наявністю входів мультиплексора з круговою функцією.

У цій модифікованій версії лише три входи необхідні для мультиплексора раундових функцій, що дозволить генерувати чотири 32-бітних слова за кожен цикл.

Модифікований графік ключів має лише один 32-бітовий вхід, щоб відповідати архітектурі GCM, і виводить чотири 32-бітових слова до розширеної пам'яті ключів кожного циклу.

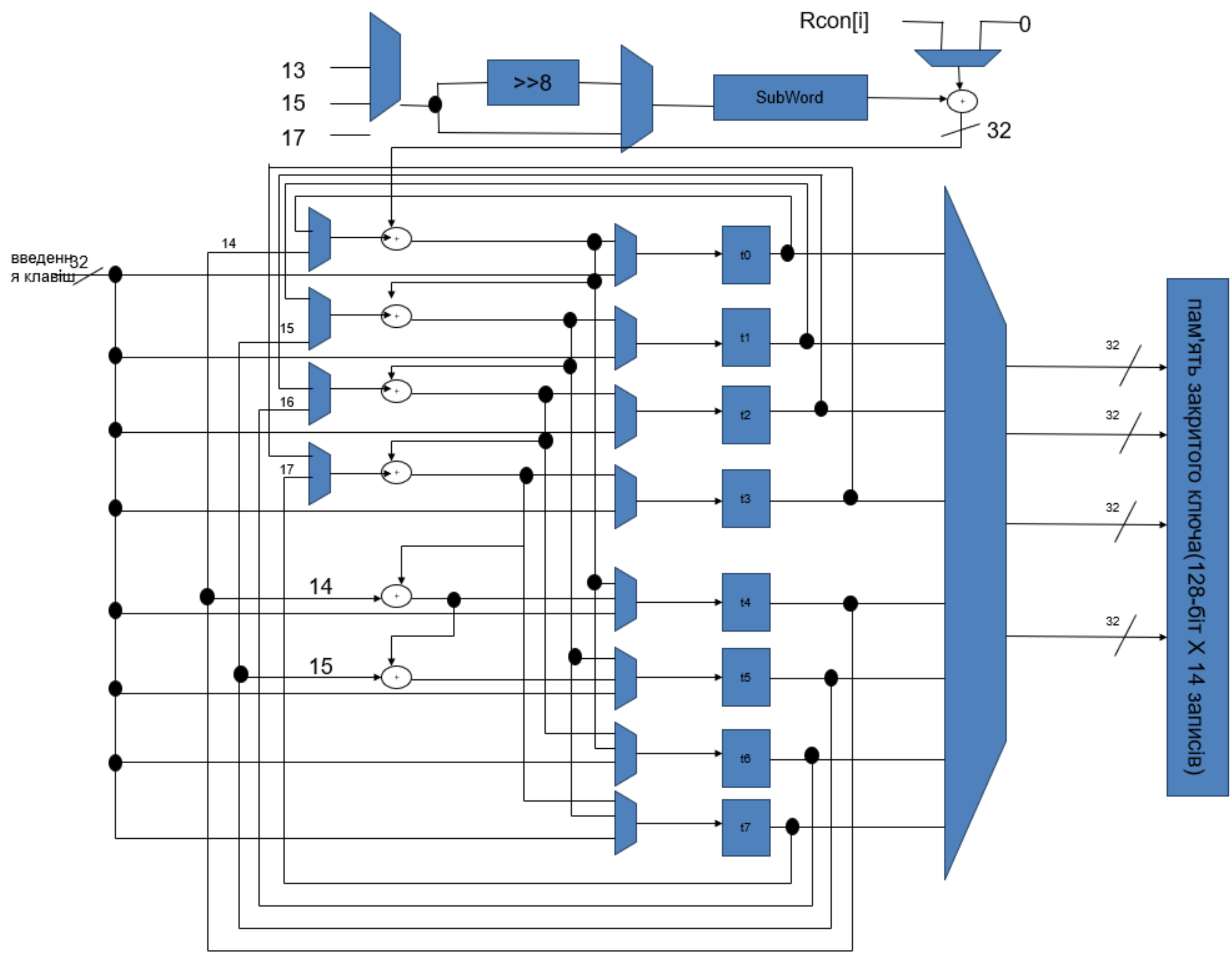


Рисунок 3.2 –Розклад ключів тракту даних шириною 128 біт з [56], який було модифіковано, щоб краще відповідати архітектурі цільової ПЛІС

Компонент розширеної пам'яті ключів залишається абстрагованим від розкладу ключів, щоб архітектуру можна було легко змінити у майбутньому.

Модифікована схема розкладу ключів, показана на рисунку 3.3, спочатку була розроблена для підтримки комбінаційних обчислень S-Box.

Було розроблено другу версію, яка дозволяє використовувати S-Box на основі пам'яті.

Ця версія з пам'яттю вимагає додаткового циклу ініціалізації для всіх трьох довжин ключів.

Три 32-розрядні входи додано до мультиплексора круглої функції, щоб дозволити зчитування даних до того, як вони зазвичай реєструються.

Це змінює порядок компонентів, знайдених на основному шляху даних через розклад ключів.

Мультиплексор круглої функції та мультиплексор підслова, замість того, щоб завжди бути першими компонентами під час циклу круглої функції, тепер є останніми компонентами.

Це не впливає на довжину критичного шляху через ключовий розклад.

Друга реалізація ключового розкладу була обрана для того, щоб мати 32-бітну ширину шляху даних.

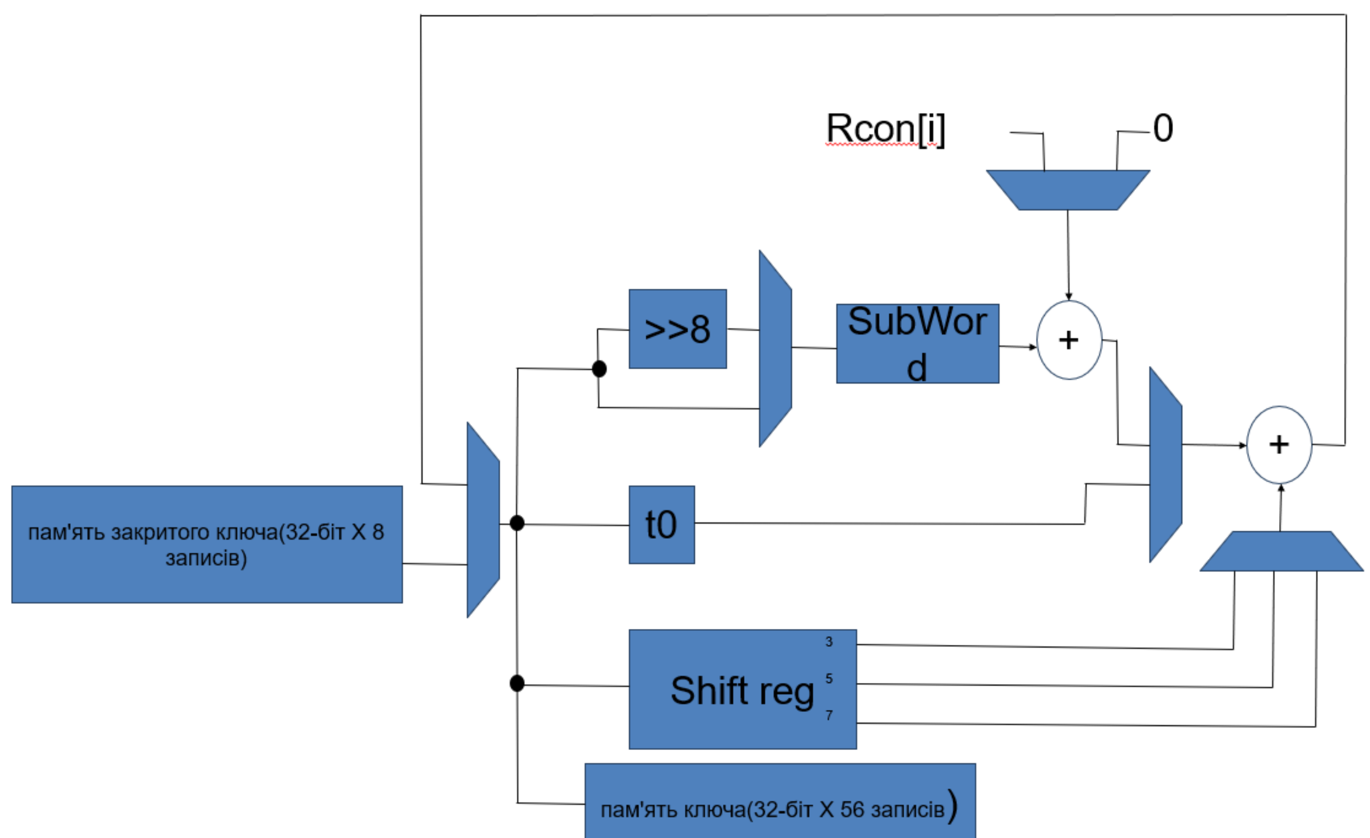


Рисунок 3.3 –Розклад ключів тракту даних шириною 32 біти, який було модифіковано з [51], щоб краще відповідати архітектурі цільової ПЛІС

Така ширина реалізації краще узгоджується з алгоритмом розкладу ключів AES, оскільки за один раунд розкладу ключів можна згенерувати повністю ціле слово.

Залежності між 32-бітними словами в розкладі ключів проілюстровано на рисунку 3.4 та рисунку 3.5.

Розклад ключів шириною 32 біти було змодельовано на основі реалізації в [51].

Ця реалізація була модифікована для кращого використання ресурсів цільової ПЛІС.

Модифіковану версію цього розкладу показано на рисунку 3.3.

Оригінальна версія орієнтована на довжину ключа 128 біт.

Ця модифікована версія підтримує всі три довжини ключів AES.

Тут використовується восьмиелементний регістр зсуву з двома додатковими відгалуженнями, які забезпечують доступ до четвертого і шостого елементів відповідно.

Зверніть увагу, що на рисунку ці елементи індексуються з нуля, тобто вони зсунуті на одиницю.

Залежно від обраної довжини ключа, мультиплексор вибирає відповідний вихід зсувного регістра на основі компонента SubWord.

У цій версії єдиний 32-бітний круглий регістр переміщується після мультиплексора, який раніше слідував за ним.

Шлях даних для цієї реалізації з комбінованим компонентом SubWord є довшим.

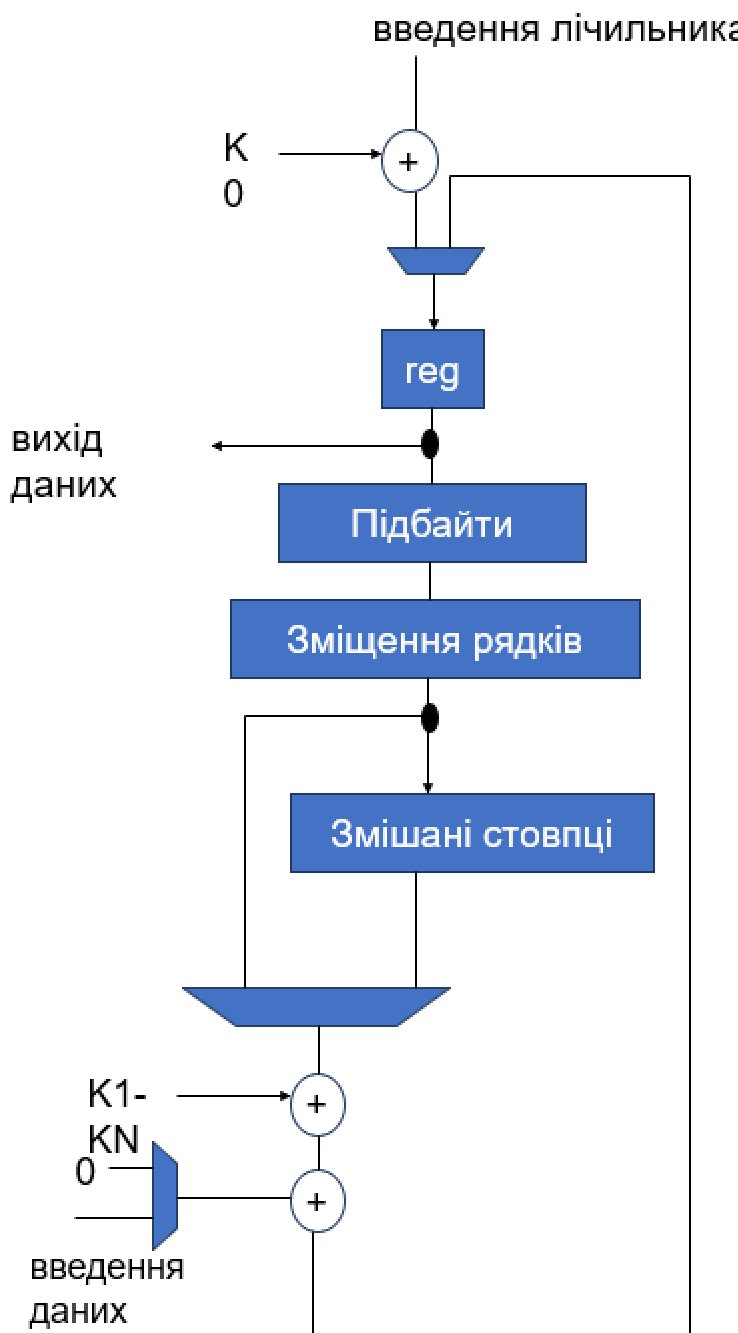


Рисунок 3.4 – Шлях даних шифрування AES шириною 128 біт, модифікований з [59]

5.1.2 Шифрування AES

Відповідно до двох ключових реалізацій розкладу, було розроблено два основні компоненти шифрування AES.

В дослідженні обидві реалізації використовували комбінований компонент MixColumns.

Перша реалізація шифрування використовувала 128-бітний шлях даних і базувалася на методі простого циклу, описаному в [59].

Оскільки робота була орієнтована на підхід ASIC, було зроблено кілька модифікацій для кращого використання ПЛІС, на яку орієнтована ця робота.

Ця модифікована версія показана на рисунку 5.4.

128-бітний раундовий регістр було переміщено з місця після фінального XOR у раунді до місця, розташованого одразу після вхідного мультиплексора.

У цьому місці раундовий регістр може також реєструвати початкове XOR з першим ключем раунду.

У попередньому розташуванні регістрів раундів розширена пам'ять ключів була б необхідна для забезпечення як першого ключа раунду, так і другого ключа раунду в межах одного циклу.

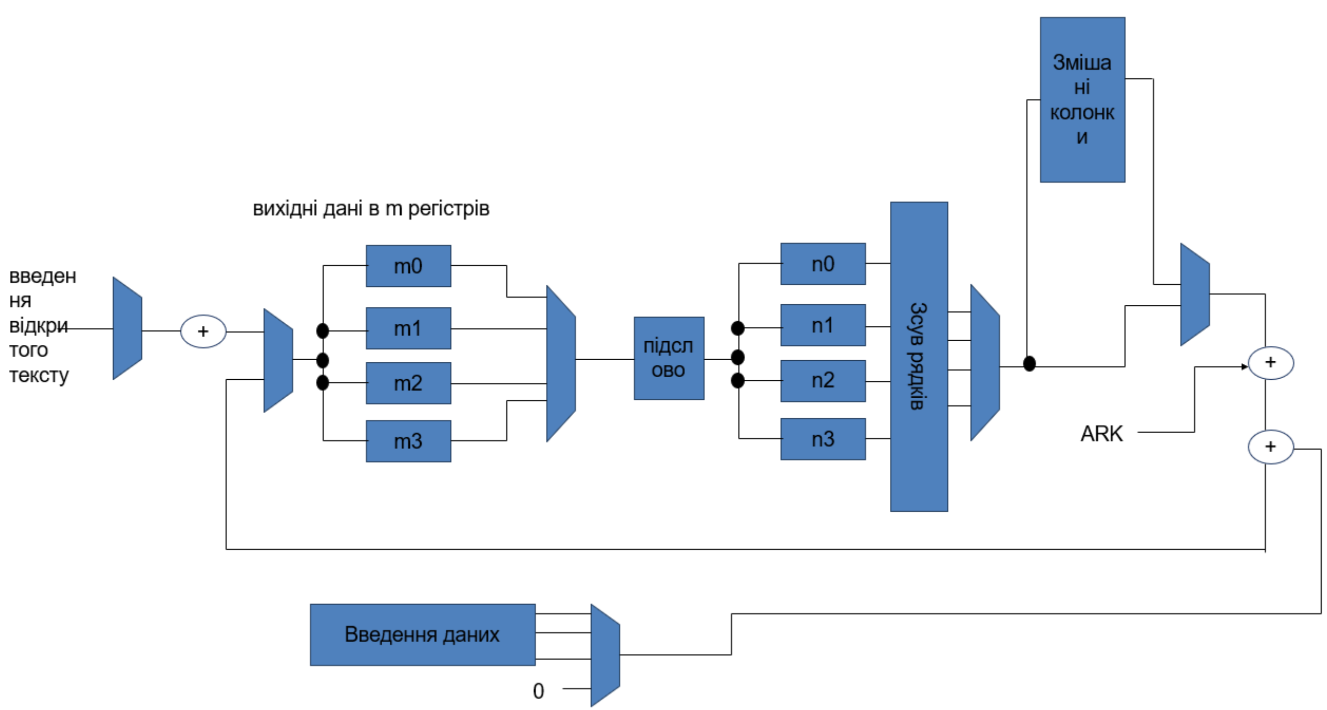


Рисунок 3.5 – Шлях даних шифрування AES шириною 32 біти

ПЛІС підтримувати справжній двопортовий режим роботи пам'яті, але інший вхід розширеної пам'яті ключів для справжнього двопортового режиму виділено під розклад ключів.

Таким чином, щоб усунути необхідність додавання ще одного 128-розрядного регістра в схему або мультиплексор для сигналів керування справжньою двопортовою пам'яттю, круглий регістр було просто переміщено.

Це також зменшує довжину критичного шляху через реалізацію шифрування, коли використовується комбінована реалізація S-Box.

У [59] додатковий 128-бітний вхід подається на компонент, який мультиплексується з 128 нульовими бітами.

Вихід мультиплексора подається на вентиль XOR.

Під час звичайних ітераційних раундів використовуються всі нульові входи схеми.

Після завершення остаточного ARK мультиплексор перемикається на XOR зашифрованих даних з вхідними даними.

Це робиться для того, щоб полегшити реалізацію функції шифрування в режимі роботи лічильника.

Це кінцеве значення потім реєструється в круглому регістрі.

У версії шифрування, яка використовує реалізацію SubBytes на основі пам'яті, круглий регістр вилучається з основного круглого шляху даних.

Круглий регістр все ще залишається в самій схемі, але використовується для реєстрації кінцевого значення вихідних даних.

32-розрядна широка версія шифрування AES базується на ітераційному підході до слів і показана на рисунку 3.6.

Вимоги до підкомпонентів зменшено порівняно з 128-бітною версією, особливо у функції SubBytes round, де замість шістнадцяти потрібно лише чотири S-Box'и.

Вісім 32-бітних регістрів використовуються під час операцій з циклами роботи.

Перша група з чотирьох регістрів використовується між раундами, тоді як друга група регістрів використовується для вирівнювання даних перед їх обробкою операцією раунду ShiftRows.

Це пов'язано із залежністю, яка існує між 32-бітними словами у цій операції системи.

Після того, як дані вирівняні, операція ShiftRows round – це просто перестановка сигналів на рівні байтів.

Мультиплексори використовуються для вибору відповідного 32-бітового слова для решти операцій циклу.

Ця 32-розрядна реалізація також має 128-розрядний додатковий вхід для роботи в режимі лічильника.

У цьому випадку мультиплексор на цьому вході перемикається між усіма чотирма 32-бітними словами на цьому 128-бітному вході на додаток до нульового входу.

3.3.2 S-box AES

Для кожної схеми шифрування і розкладу ключів необхідно вибрати відповідну реалізацію S-боксу для роботи з підбайтами або підсловами.

У роботі було досліджено чотири реалізації S-Box:

1) на основі однопортової пам'яті M9K;

2) на основі двопортової пам'яті M9K;

3) на основі LUT;

4) на основі складеного поля.

Реалізація на основі складеного поля використовує комбінаційну логіку для обчислення вихідного значення S-Box, тоді як інші три реалізації посилаються на попередньо обчислене значення.

S-Box – це не більше ніж інверсія поля Галуа та афінне перетворення, що виконується над одним байтом.

Загалом, це 2^8 можливих значень, які можна попередньо обчислити.

Для використання LUT для реалізації S-Box необхідно виділити 256 однобайтових елементів.

Ці великі LUT утворюються шляхом об'єднання чотирьох вхідних LUT з логічних елементів (LE) цільової ПЛІС.

Вихід LUT доступний після одного циклу.

Реалізації на основі пам'яті створюються таким же чином, за винятком того, що замість логічних елементів LE використовуються елементи пам'яті M9K ПЛІС. Кожен M9K забезпечує 8192 біт пам'яті.

Екземпляри пам'яті з одним портом мають один вхід адреси, один вхід даних і один вихід даних.

Екземпляри пам'яті з двома портами мають два входи адреси, два входи даних і два виходи даних.

Економія коштів при використанні S-Box на основі справжньої двопортової пам'яті вдвічі менша, ніж при використанні однопортової пам'яті, оскільки потрібно вдвічі менше мікросхем M9K, для реалізації такої ж кількості S-Box.

Для використання в дослідженні було обрано композитне поле S-Box, щоб представити альтернативу підходам пам'яті та LUT.

Залежно від використання, підхід складеного поля може бути використаний для досягнення високої продуктивності або малої площі.

У роботі компоненти S-Box не є конвеєрними.

Іншими словами, між окремими компонентами в межах компонента S-Box немає регістрів.

Це було зроблено для того, щоб дослідити зменшену логічну вартість реалізації складеного поля.

Структура складеного поля S-Box показана на рисунку 3.2.

Трубопровідне з'єднання призвело б до значно коротшого критичного шляху через компонент, але збільшило б використання його ресурсів.

3.3.3 GHASH

Реалізації GHASH, розглянуті в дослідженні, відрізняються за способом застосування множника $GF(2^{128})$.

Розглянуто побітовий послідовний підхід, повністю паралельний підхід та різноманітні підходи послідовного множення та додавання.

Повністю паралельний підхід обчислює результат за один такт, але має дуже великий критичний шлях.

На відміну від повного паралельного підходу, побітовий послідовний підхід має набагато менший критичний шлях, але потребує 128 тактів для обчислення результату.

Підходи послідовного множення та додавання зменшують кількість тактів, що зустрічаються в побітовому послідовному підході, за рахунок множення в ступені двох чисел бітів за один такт.

Через залежність даних у цих множеннях від поля Галуа, ці обчислення не виконуються паралельно.

Таким чином, чим більша кількість бітів множиться за один такт, тим глибший критичний шлях множника.

Всі реалізації множника $GF(2^{128})$ були створені на основі одного загального екземпляру.

Як мінімум, загальний множник працював у режимі послідовного перемноження бітів.

Як максимум, загальний множник реалізовував повністю паралельний підхід обчислень.

Ця реалізація була заснована на описі послідовного підходу до множення та додавання, представленому в [57].

Бажана кількість тактів до завершення безпосередньо відповідає ширині послідовно перемножуваних бітів системи у зменшеному множнику обраного поля Галуа.

Діаграма високого рівня, що ілюструє окремі компоненти МДГ та їхні взаємозв'язки подано на рисунку 3.6.

Наприклад, реалізація, яка виконується за 16 тактів, використовує множник $GF(2^8)$ X $GF(2^{128})$.

У кожному раунді функції GHASH декілька байтів з константи H множаться на 128-бітне поточне значення GHASH.

Зменшення виконується на основі примітивного полінома GHASH: $x^{128} + x^7 + x^2 + x + 1$. У 128-бітному регістрі зберігається значення часткового результату, доки не буде оцінено кінцевий результат.

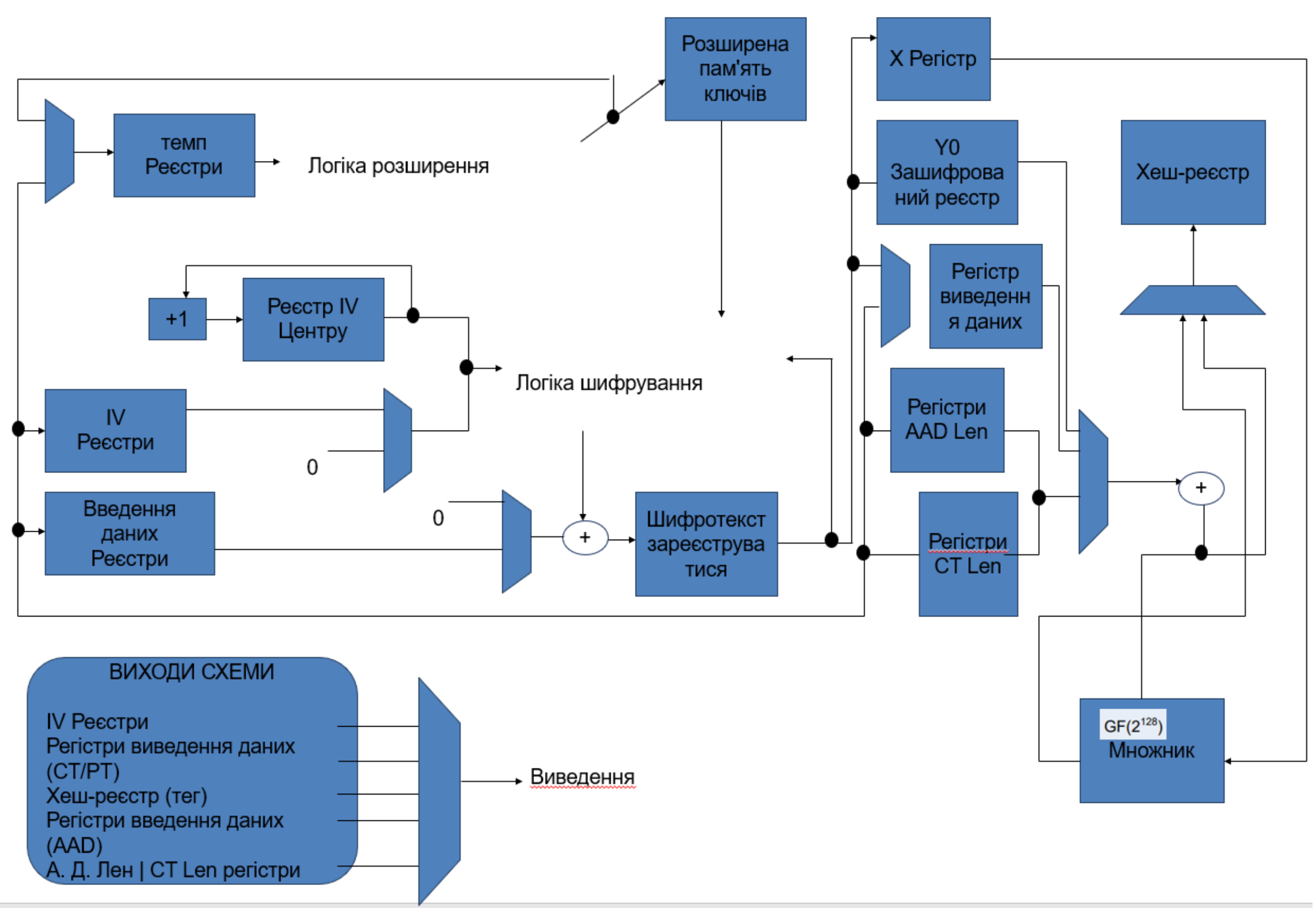


Рисунок 3.6 – Діаграма високого рівня, що ілюструє окремі компоненти МДГ та їхні взаємозв'язки

Після цього кінцевий результат записується у 128-бітний регістр.

Значення, що зберігаються в цьому регістрі, завжди будуть кінцевим результатом послідовних операцій множення та додавання.

3.3.4 GCM

Найвищий рівень алгоритму GCM відповідає за контроль функціональності всіх попередньо розроблених підкомпонентів.

Регістри, які отримують вхідні дані від входу даних верхнього рівня рушія GCM, записуються 32-бітними словами.

Усередині рушія операції обробляються з 32-бітною або 128-бітною шириною, як це визначено конкретним компонентом.

Усі реалізації GCM у даному дослідженні використовують 96-розрядний IV, як рекомендовано стандартом NIST [9] з метою підвищення ефективності [55].

Регістри довжини AAD та CT мають по 64 біти.

Ці регістри було додано для того, щоб процесор міг повідомляти рушій про довжину даних, які було оброблено.

Рушій GCM був розроблений для підтримки даних довжиною, що дорівнює розміру блоку AES.

Це перекладає операції додавання даних на процесори, що є тривіальною операцією.

Ці довжини необхідні для генерації кінцевого тегу.

Четвертий і приватний ключ зберігаються на верхньому рівні розробленого рушія.

Особистий ключ зберігається у молодших бітах розширеної пам'яті отриманих ключів.

Розклад ключів AES виконує цю копію за задумом.

Керуючий автомат, зображений на рисунку 3.7, було розроблено таким чином, що при зміні закритого ключа здійснюється виконання перерахунку і для значення Y_0.

Ця функція була додана для того, щоб отримати додаткову економію за рахунок відсутності необхідності викликати другу операцію, щоб просто обчислити це значення, коли IV записується після зміни приватного ключа.

Якщо IV не буде змінено за допомогою приватного ключа, то економія все одно відбудеться, оскільки Y_0 буде перераховано з новим ключем.

Система була розроблена таким чином, що IV може бути змінений самостійно, що потім оновить Y_0.

Хоча це не обов'язково, рушій GCM надає доступ для читання до регістрів довжини AAD і CT. Це може бути корисним у сценарії налагодження. Це не є обов'язковим, оскільки GCM є онлайн-алгоритмом.

Цей рушій був розроблений для підтримки всіх трьох довжин ключів AES і GCM в режимах шифрування і розшифрування.

На рисунку 3.6 показано сигнал, активний лише в режимі розшифрування, а жирним чорним кольором – сигнал, активний лише в режимі шифрування вхідних даних.

Довжина ключа AES вказується рушію за допомогою одного гарячого кодованого сигналу.

Режим шифрування вказується за допомогою одного сигналу, який має високий логічний рівень в режимі шифрування і низький логічний рівень в режимі дешифрування.

Внутрішній контрольний регістр постійно зчитується автоматом для визначення наступної інформації про стан.

На входи цього регістра керування подаються зовнішні строб-сигнали. Ці сигнали відповідають операціям, доступним у рушії:

1) запуск зміни ключа,

2) запуск зміни IV,

3) запуск хешування aad,

4) запуск шифрування/розшифрування CT/PT та хешування,

5) та запуск генерації фінальних тегів TAG.

Було створено додатковий регістр для повідомлення керуючому автомату про те, що останній компонент завершив операцію розшифрування.

Високорівнева мета цієї дипломної роботи полягала в тому, щоб зробити акцент на модульності. Додавання цього регістру дозволяє змінювати різні реалізації AES і GHASH без необхідності модифікувати машину стану GCM верхнього рівня, щоб визначити, яка з них споживає більше циклів.

Це важливо, оскільки компонент шифрування AES і компонент GHASH можуть працювати паралельно під час операції розшифрування.

Кожен компонент механізму повідомляє керуючому автомату про завершення роботи за допомогою прапорця «done».

Якби ці стани відстежувалися безпосередньо, то знадобилося б три додаткових стани, якщо вони завершили роботу одночасно або якщо один з них завершився раніше за інший.

3.4 Організація системи

На найвищому рівні система складається з двох мікропроцесорів Nios II та двох рушіїв GCM.

Процесори Nios II були розроблені зі спеціальним компонентом, який просто виводить сигнали шини Avalon як канал.

Потім ці сигнали шини Avalon були з'єднані з іншим компонентом, який реєстрував кожен з сигналів.

Сигнал waitrequest оброблявся як особливий випадок, оскільки він сигналізує процесору, що рушій хоче, щоб він зачекав.

Ці зареєстровані сигнали потім були виходами підсистеми процесора (чорного або червоного кольору).

Таким чином, реєстровий інтерфейс обгортає провідники процесора Nios II. Регістровий інтерфейс також був розміщений навколо кожного з сигналів рушіїв GCM. Потім обидва регістрових інтерфейси були з'єднані.

Регістрові інтерфейси знаходяться на найвищому рівні проектної ієрархії в розділі підсистеми.

Таким чином, будь-який сигнал, що проходить між процесорним розділом і розділом GCM-рушія, реєструється на його виході і знову на відповідному вході на цільовому компоненті.

Цей регістровий інтерфейс було додано на основі рекомендації [67], в якій вказано, що регістри мінімізують затримки на міжрозділових шляхах і запобігають необхідності оптимізації міжграничної логіки.

Модуль GCM спочатку був розроблений для підтримки 128-бітових входів, але був модифікований для підтримки 32-бітових входів.

Ця модифікація була необхідна через максимальну ширину шини даних процесора Nios II.

Оскільки ця система була призначена для використання з Nios II, це була необхідна модифікація.

Якби цю зміну не було зроблено, на верхньому шарі обгортки GCM було б потрібно більше логіки для правильної обробки даних, що надходять до рушія.

Єдиною визначеною вимогою в цій області було те, що рушій повинен взаємодіяти з процесором Nios II, тому 128-бітовий вхід і вихід з системи був непотрібним.

Безпосередньо під інтерфейсом обгортки регістрів у рушії GCM знаходиться компонент інтерфейсу GCM.

Роль цього модуля полягає у перетворенні всіх сигналів, що проходять між процесорами та рушіями, у формати, які кожен з них очікує.

Одними з найважливіших функцій цього компонента є регістри керування, конфігурації та стану.

Регістр стану може бути прочитаний будь-яким процесором і просто реєструє, коли кожне із завдань рушіїв GCM завершено.

У ньому вдвічі більше бітів, ніж операцій GCM, оскільки і чорний, і червоний процесори мають власні біти, які вказують на те, яке завдання завершено.

Ці біти мають бути очищені відповідним процесором.

Наприклад, чорний процесор не може очистити біти регістру стану червоного процесора.

Мета очищення біта – повідомити іншому процесору, що поточний процесор закінчив роботу з даними, які відповідають певній операції.

Знову ж таки, всі ці біти можуть бути прочитані будь-яким центральним процесором.

Таким чином, кожен процесор може прочитати, коли операція, на яку він очікує, завершена, і коли інший процесор закінчив виконання операцій, пов'язаних з конкретною операцією GCM.

Регістр керування дозволяє процесору запустити операцію GCM, а регістр конфігурації встановлює довжину ключа AES і режим роботи шифрування або дешифрування.

Регістр конфігурації може бути встановлений тільки центральним процесором.

На інші біти регістрів керування та стану накладено обмеження, щоб чорний процесор не міг до них записувати.

Це залежить від поточної конфігурації рушія.

У таблиці 3.1 показано, яку адресу може або не може записувати або читати даний процесор за певних умов.

Запис до регістра з неправильного процесора не має жодного фактичного ефекту.

Читання з регістра, недоступного для певного процесора, призведе до нульового результату. Карта реєстру GCM подано в таблиці 3.1.

Таблиця 3.1 – Карта реєстру GCM

Назва регістра	Кодування		Декодування	
	Червоний	Чорний	Червоний	Чорний
Статус	читання-запис	читання-запис	читання-запис	читання-запис
config	читання-запис	читання	читання-запис	читання
control	запис	-	запис	запис
key_()	запис	-	запис	-
key_1	запис	-	запис	-
key 22	запис	-	запис	-
key_3	запис	-	запис	-
key_4	запис	-	запис	-
key_5	запис	-	запис	-
key_6	запис	-	запис	-
key_7	запис	-	запис	-
iv_()	читання-запис	читання	читання	читання-запис
iv_1	читання-запис	читання	читання	читання-запис
iv_2	читання-запис	читання	читання	читання-запис
aad_0	читання-запис	читання	читання	запис
aad_1	читання-запис	читання	читання	запис
aad_2	читання-запис	читання	читання	запис
aad_3	читання-запис	читання	читання	запис
pt_()	читання-запис	-	читання	-

Кінець таблиці 3.1 – Карта реєстру GCM

pt_1	читання-запис	-	читання	-
pt2	читання-запис	-	читання	-
pt_3	читання-запис	-	читання	-
ct_O	читання	читання	читання	читання-запис
ct_1	читання	читання	читання	читання-запис
ct_2	читання	читання	читання	читання-запис
ct_3	читання	читання	читання	читання-запис
tag-0	читання	читання	читання	читання
tag_1	читання	читання	читання	читання
tag_2	читання	читання	читання	читання
tag_3	читання	читання	читання	читання
aad_len_O	читання-запис	читання	читання	читання-запис
aad_len_1	читання-запис	читання	читання	читання-запис
ct_len_0	читання-запис	читання	читання	читання-запис
ct_len_1	читання-запис	читання	читання	читання-запис
ver_tag_()	-	-	читання	запис
ver_tag_1	-	-	читання	запис
ver_tag_2	-	-	читання	запис
ver_tag_3	-	-	читання	запис

5.2.1 Високопродуктивна архітектура

Високопродуктивна реалізація в дослідженні націлена на 128-бітовий канал даних AES.

Реалізація шифрування AES базується на круговій ітераційній схемі, де круглі ключі зчитуються з пам'яті на кристалі.

Ці раундові ключі поміщаються в пам'ять на кристалі за 128-бітною схемою автономних ключів.

Пам'ять на кристалі організовано таким чином, що вона адресується 128-бітними блоками.

Це дозволяє прочитати або записати цілий круглий ключ за один виконаний цикл.

Множник GHASH реалізовано повністю паралельно.

5.2.2 Дизайн малої області

Реалізація на малій ділянці в дослідженні була зосереджена на використанні 32-бітового шляху даних для розкладу ключів AES і компонентів шифрування. Розклад ключів базувався на 32-бітній ітераційній версії.

Реалізація 32-розрядного шифрування AES є простою ітераційною схемою апаратури.

Круглі ключі зберігаються в пам'яті на кристалі, що адресується 32-бітними словами.

Перемножувач $GF(2^{128})$ у компоненті GHASH реалізовано побітово-послідовним способом: множення та накопичення одного біта вхідних даних на 128-бітне значення H послідовно протягом 128 тактів.

Високорівнева архітектура відрізняється від рисунку 5.6 додатковою логікою, необхідною компоненту GHASH для накопичення часткових добутків.

5.2.3 Збалансований дизайн продуктивності

Реалізація, спрямована на баланс між високою продуктивністю та малою площею, використовує той самий 128-бітний графік ключів AES на шляху передачі даних та компоненти шифрування, що й високопродуктивна реалізація.

Різниця полягає в компоненті GHASH, яка використовує підхід послідовного множення та додавання з [58].

Результат обчислюється за 16 тактів за допомогою операцій послідовного множення та додавання з використанням одного 128-бітного операнду та одного байтового операнду.

Високорівнева архітектура рушія GCM відрізняється від зображеної на рисунку 3.7.

GHASH-компонент, де додатковий регістр і кілька мультиплексорів необхідні для обчислення часткового добутку.

Залежно від бажаних атрибутів реалізації та цільової архітектури, на якій ця система буде розміщена, можна зробити ряд покращень.

На рівні підкомпонентів можна реалізувати спільне використання S-Box між розкладом ключів AES та реалізаціями шифрування.

Під час нормальної роботи GCM малоймовірно, що IV або приватний ключ будуть часто змінюватися.

У цьому випадку розклад ключів буде простоювати переважну більшість часу. Розклад ключів для всіх варіантів ширини шляху даних використовує чотири компоненти S-Box SubWord. Це можна було б використати у 32-бітній широкій версії функції шифрування AES.

Так само, як і спільне використання підструктури S-Box, розклад ключів шириною 32 біти можна використовувати разом з компонентом шифрування AES шириною 128 біт.

Знову ж таки, розклад ключів не викликається часто, і компроміс може бути зроблений для економії логічних ресурсів за рахунок більш повільного розкладу ключів при зміні IV або приватного ключа.

Компонент розширеної пам'яті ключів потребуватиме додаткової логіки керування, щоб мати можливість записувати 32-розрядні слова і при цьому надавати 128-розрядні круглі ключі механізму шифрування.

Варто зазначити, що у всіх реалізаціях, розглянутих у дослідженні, більшість логічних ресурсів на ПЛІС було використано компонентом GHASH.

Таким чином, використання спільної підструктури між S-Box'ами може дати лише помірну економію.

Реалізація 8-бітної ширини каналу даних для компонентів розкладу ключів і шифрування може забезпечити більш значну економію площі порівняно з підходом шириною 128 біт.

Об'єднання цих 8-бітних компонентів AES з послідовною реалізацією GHASH створить схему, яка використовує дуже мало логічних ресурсів і має схожу пропускну здатність як для операцій GHASH, так і для операцій шифрування.

Під час операції шифрування виникала б дуже велика затримка, оскільки GHASH має слідувати за шифруванням послідовно через існуючу залежність від даних.

Така схема була б корисною в ситуації, коли використовується дуже маленька, недорога, захищена ПЛІС і продуктивність не є проблемою.

Це також може бути кращою реалізацією для ПЛІС, де необхідно реалізувати надмірність. Можна було б розмістити багато таких рушіїв GCM на одній мікросхемі.

На високому рівні можна використовувати структури даних типу FIFO (first in, first out) для створення черги даних для рушія, які потім можуть бути передані в конвеєр під час операцій шифрування.

Додатково або окремо можна розглянути можливість прямого введення даних у рушій ГПШ.

У цьому випадку процесори керували б потоком даних по всій схемі, але не пропускали б дані безпосередньо через себе.

Це усунуло б будь-яку програмну обробку даних, дозволивши їм повністю проходити через апаратне забезпечення.

Рисунок 3.7 – Машина станів верхнього рівня GCM

Такий підхід має значні переваги у продуктивності, але такий тип реалізації потребує більш ретельного вивчення, щоб переконатися, що він пройде перевірку на безпеку.

Більш простим покращенням могло б бути видалення надлишкових регістрів, які існують на верхньому рівні компоненту GCM.

Це призвело б до економії логічних ресурсів, але могло б зменшити модульність системи.

Для зменшення ресурсів, що використовуються, необхідним є використання регістрів зсуву для розширеної пам'яті ключів.

Це усуне потребу в лічильниках і відповідних лініях керування сигналами адреси пам'яті. Ця методика може вимагати менше елементів пам'яті від ПЛІС, залежно від їх організації та використання в проекті.

Щоб створити більш автономну версію GCM, довжину даних можна було б реєструвати всередині самого рушія.

В даний час довжина даних повинна бути кратною розміру блоку AES, а загальна довжина даних повинна надаватися рушію через два 64-бітних регістри.

У реалізації, заповнення та маскування може виконуватися процесорами. Якщо буде прийнято рекомендацію передавати дані безпосередньо до рушія, а не через процесори, це заповнення може бути вже не таким тривіальним.

На той час може стати доцільним реєструвати довжину даних внутрішньо і додавати зсув для маскування даних, якщо вона не кратна розміру блоку AES.

3.5 Висновки

З метою вирішення поставленої задачі було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

У розділі було розроблено систему з розділенням червоного/чорного з незалежними реалізаціями шифрування та дешифрування GCM з використанням AES для передачі автентифікованої та зашифрованої інформації між двома

процесорами Nios II.

Реалізації цієї системи були оцінені на ПЛІС Cyclone V на основі метрик високої продуктивності, низького використання ресурсів та балансу між ними.

Було проведено аналіз використання ресурсів, отримано підтвердження факту підвищення швидкодії AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA.

4 РОЗРОБЛЕННЯ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

4.1 Вибір типу архітектури та зразків проектування

Розглянемо три напрямки аналізу та реалізації запропонованого методу розроблення програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA : 1) верифікація, 2) продуктивність і 3) аналіз безпеки.

Таким чином, необхідно провести аналіз верифікації та встановити чи результати роботи системи є правильними.

Також необхідно проаналізувати рівень продуктивності реалізованих засобів та пропускної здатності апаратних механізмів GCM.

Також необхідно здійснити аналіз безпеки та рівень впливу функцій безпеки ПЛІС на реалізацію системи.

Верифікація апаратних модулів проводилась за допомогою тестових стендів VHDL та роботи в ПЛІС за допомогою мікропроцесорів м'якого ядра, що використовувались в якості стимулів. Після того, як апаратний бітовий потік було скомпільовано, програмне забезпечення було скомпільовано відповідно до цього апаратного забезпечення. Фактична верифікація була виконана з використанням тестових векторів, наданих у [20].

Процесору були передані дані тестового вектора у вигляді статичних констант. Тестування було розбито на функції C для кожного тестового вектора. Дані конфігурації та закритого ключа спочатку використовувалися для налаштування обох рушіїв GCM, потім дані у відкритому вигляді надсилалися до рушія шифрування GCM. Чорний мікропроцесор отримав ці зашифровані дані від механізму шифрування і передав їх назад до механізму розшифрування. Механізм розшифрування був ініціалізований червоним процесором з тією ж конфігурацією, що і механізм шифрування. Якщо дані, отримані червоним процесором від рушія дешифрування, збігаються з даними, які спочатку були надіслані рушію

шифрування, тест вважається успішним. Успішність або неуспішність кожного тесту записується у стандартний вивід.

Таблиця 4.1 – Статистика продуктивності одного GCM

Мета	біти	Архітект. AES S-Box	Архітектура GHASH	Частота GCM Max, MHz	Пропускна здатність 128-bit Pckt, Mbps	Пропускна здатність 2K, Mbps
Мала площа	32	C-Field	Bit serial	69.3	31.8	59.9
	32	M9K	Bit serial	121	54.6	112
Висока продуктивність	128	C-Field	Full parallel	70.4	461	539
	128	M9K	Full parallel	75.9	501	601
Збалансована продуктивність	128	C-Field	16-Sequen.	66.3	222	398
	128	M9K	16-Sequen.	98.1	305	701

4.2 Продуктивність апартного забезпечення

Результати продуктивності для кожної конфігурації системи наведено у таблиці 6.1. Розмір пакету – це обсяг зашифрованих даних, які обробляються механізмом GCM перед тим, як генерується тег автентифікації. Функціональність була перевірена на належну обробку AAD, але обробка AAD не була включена до тестів продуктивності. Архітектура та периферія двох процесорів Nios II/ів були узгоджені в кожному тесті.

Як і очікувалося, глибокий критичний шлях, знайдений у реалізації композитного поля S-Box, обмежив максимальну частоту необхідного GCM-рушія.

Можливою перевагою цього підходу є низькі вимоги до M9K, як показано в таблиці 4.2.

Реалізації GCM, що використовують переваги справжньої двопортової пам'яті на кристалі, є набагато більш ресурсоефективними, враховуючи переважання блоків пам'яті M9K на цільовій ПЛІС.

Крім того, пропускна здатність і тактова частота були вищими для реалізацій з використанням S-Box на основі пам'яті.

Реалізація на основі пам'яті зі збалансованою продуктивністю мала найвищу пропускну здатність при більшому розмірі пакетів.

Це пояснюється вищою частотою і тим, що високопродуктивна реалізація мала еквівалентний компонент шифрування AES, який продовжував виконувати свої обчислення, поки паралельний компонент GHASH простоював.

Архітектура апаратного забезпечення має мітки: один і два позначено червоний і чорний розділи підсистеми відповідно. Мітки три і чотири позначають розділи механізму шифрування і дешифрування GCM відповідно.

Темні заштриховані блоки – це використані ресурси, за винятком кордонів захищених регіонів.

Це огорожі невикористаної логіки, де не дозволено жодних маршрутних з'єднань.

Невеликі світло заштриховані області, що не містять логіки між захищеними областями, є SRI, які дозволяють маршрутизацію між областями.

Смуги, що проходять вертикально по всій ПЛІС, є елементами пам'яті M9K та вбудованими помножувачами.

Важливо відзначити, що нерозподілений простір є місцем, де глобальні сигнали вводяться в глобальну структуру маршрутизації для розподілу по всій ПЛІС.

Таблиця 4.2 – Статистика використання ресурсів

Мета	біти	Архітект. AES S-Box	Архітектура GHASH
Мала площа	32	C-Field	порозрядна
	32	M9K	порозрядна
Висока продуктивність	128	C-Field	Повний паралелізм
	128	M9K	Повний паралелізм
Збалансована продуктивність	128	C-Field	16-бітна послідовність
	128	M9K	16-бітна послідовність

4.3 Проєктування програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

У дослідженні проекти спочатку були функціонально перевірені та остаточні проекти були синтезовані для пристрою Cyclone V.

Хоча реалізоване апаратне забезпечення є функціонально еквівалентним, для задоволення вимог безпеки на пристрої LS були необхідні додаткові обчислювальні ресурси.

Таблиця 4.2 ілюструє загальну кількість використаних LE і M9K із загальної кількості, виділеної для конкретного захищеного регіону.

Разом розмір і розташування захищених регіонів 1 визначають кількість ресурсів, доступних для регіону 2 встановлюють час внутрішньосхемної і зовнішньої передачі даних в регіони і з регіонів 3 обмежують доступні шляхи маршрутизації між захищеними розділами.

Кожен захищений регіон також є логічним розділом сам по собі, що запобігає оптимізації, яка могла б знаходитсь на кордонах розмежування [67].

Таким чином, ці захищені області сильно залежать від конструкції та платформи. Для того, щоб було легше робити висновки, розміри захищених областей були фіксованими для конкретного цільового застосування.

Для високопродуктивної реалізації накладні витрати включають 8912 LE, 36 M9K і 20 вбудованих мультиплікаторів, які повинні залишатися невикористаними для формування кордонів між захищеними регіонами.

SRI може містити лише маршрутизацію, тому 3120 LE, 12 M9K та 8 вбудованих множників, що містяться в SRI, виділяються, але не завжди використовуються.

Беручи до уваги ці цифри, залишається 20576 LE, 35 M9K і 20 вбудованих множників вільними для використання в якості глобальної логіки.

Рисунок 4.2 – Плоский графік, що показує потік даних між підсистемами

Додатковою проблемою при виборі розміру захищеного регіону є маршрутизація.

У [2] зазначено, що мінімальний розмір захищеного регіону становить 8x8 LABs, інакше існує ймовірність того, що маршрутизація повинна буде виходити за межі регіону для з'єднання з кожною LAB.

Ширина SRI також рекомендується 12 LAB, коли сигнали проходять вгору або вниз, і 17 LAB, коли сигнали проходять вліво або вправо в межах обраної FPGA.

Розділи шифрування та дешифрування пропускають 78 з'єднань по вертикалі, що вимагає ширини 8 LAB.

Червоний і чорний розділи мають по 186 з'єднань з JTAG-концентратором Altera, що вимагає 16 LAB для вертикальних з'єднань.

Остаточний розмір був обраний 10 LAB для вертикальних і 5 LAB для горизонтальних з'єднань, що дає 205 можливих з'єднань.

Цей мінімальний рекомендований розмір області суттєво впливає на надмірну висоту червоної та чорних перегородок.

Банки вводу/виводу, доступні для глобальної логіки, обмежені в захищеному дизайні, оскільки банки вводу/виводу не можуть бути спільними для захищених регіонів [2].

У дослідженні один вільний банк був використаний на глобальному рівні для забезпечення системного годинника та входів скидання.

За винятком сигналів інтерфейсу Altera JTAG, інші банки вводу/виводу не використовувались.

Обмеження захищеної маршрутизації унеможливили повний доступ до банків вводу/виводу 7 і 6, і лише частковий доступ до виводів на банках 8 і 5.

Проектування з урахуванням вищезгаданих обмежень призвело до неефективного використання ресурсів ПЛІС у захищених регіонах.

Еталонний дизайн для цього проекту використовував JTAG-з'єднання Altera для вводу та виводу даних до систем.

Це обмежило планування таким чином, що червоні та чорні підсистемні розділи повинні були підключатися до сигналів JTAG.

У практичному застосуванні розробник призначив би виводи таким чином, щоб дані могли проходити через ПЛІС, усуваючи необхідність виділення надмірної логіки для схемотехнічної маршрутизації.

Реалізація GCM зі збалансованою продуктивністю, що використовує справжню двопортову пам'ять, забезпечує найбільш ефективне використання пам'яті і логіку, забезпечуючи при цьому високу пропускну здатність.

4.4 Розробка системного програмного забезпечення

У цьому розділі детально описано організацію та процес створення програмного забезпечення, що працює на вбудованих процесорах Nios II.

Наведено опис сценаріїв автоматизації, які були розроблені для створення, збірки та завантаження програмного забезпечення.

4.3.1 Організація

На високому рівні програмне забезпечення організовано у три основні проекти:

1) blk_proc,

2) red_proc

3) lib_gcm.

Проекти blk_proc і red_proc містять код, який стосується лише чорного і червоного процесорів відповідно. Каталог lib_ gcm містить спільний код, який використовується обома процесорами для взаємодії з відповідним рушієм GCM.

4.4.1 Процес збірки

Бітовий потік програмного забезпечення створюється шляхом виконання серії скриптів. Ці скрипти описані нижче:

create-this-bsp_<red/black>.sh

Створює двійкові файли пакету підтримки плати (BSP) для процесора Nios II.

create-this-app_<red/black>.sh Створює бінарні файли програми для процесора Nios II.

create-this-lib.sh Спроба створити статично зв'язану бібліотеку для функцій рушія GCM.

Скрипти BSP та створення додатків були модифіковані з версій, наданих у програмному забезпеченні Altera.

Всі три скрипти використовують відносне розташування для адресації файлів і каталогів.

Таким чином, необхідно, щоб програмне забезпечення було організовано у структурі каталогів.

Крім того, скриптам потрібен доступ до файлів, створених під час апаратної компіляції.

Ці скрипти очікують, що файли апаратного забезпечення знаходяться на один рівень вище розташування самих скриптів у каталозі з назвою «hw altera».

Скрипти генерації BSP передають відповідні файли і каталоги до «nios2-bsp». Потім ця програма створює ucosii_net_zipfs BSP, необхідний для використання процесора Nios з м'яким ядром.

Сценарії генерації додатків передають відповідні файли і каталоги до «nios2-app-generate-makefile», який рекурсивно генерує Makefile, необхідний для створення коду програми для конкретного BSP.

Після того, як цей Makefile створено, скрипт виконує команду «make» для новоствореного Makefile.

Сценарій створення статичної бібліотеки програмного забезпечення GCM не був використаний у цьому проекті.

Це пов'язано зі способом створення статичної бібліотеки інструментами збірки Nios.

Ці інструменти вимагають, щоб статична бібліотека була створена для певної архітектури процесора та набору функцій.

У дослідженні такий рівень абстрагування від кодової бази GCM був просто не потрібен. Найкращою практикою в цьому випадку було гарантувати, що обидва процесори використовують однаковий код для взаємодії з рушіями GCM. Це спрощує взаємодію з рушієм і дозволяє вносити зміни в базовий код апаратної взаємодії рушія GCM без обов'язкової зміни програмного інтерфейсу, представленого вбудованому прикладному коду мікропроцесора.

З точки зору безпеки, чорний процесор має доступ до того ж коду взаємодії з рушієм GCM, що і червоний процесор.

Це не є проблемою, оскільки апаратна конфігурація не дозволяє чорному процесору досягти того ж рівня контролю, що й червоному. Знову ж таки, це лише код апаратної взаємодії.

У цьому проекті було використано сценарії командного інтерпретатора, а не доступне інтерактивне середовище розробки (IDE) Nios, головним чином для спрощення процесу збірки.

У звичайному циклі розробки програмного забезпечення для вбудованих процесорів з використанням Nios IDE, IDE має бути відкрито і використано для побудови BSP і коду програми, завантаження бітового потоку, його виконання і виконання будь-якого налагодження, бажаного користувачем.

У цьому проекті було створено багато різних апаратних конструкцій, кожна з яких вимагала власного програмного середовища для верифікаційного тестування.

Високорівнева обгортка апаратного інтерфейсу GCM для процесора Nios II також не змінювалася між апаратними проектами.

Таким чином, програмне забезпечення було узгодженим між усіма різними апаратними проектами.

Набагато швидше було просто виконати ці серії скриптів для перевірки функціональності системи, ніж створювати цілий новий проект розробки просто для тестування того самого програмного забезпечення на різному обладнанні.

4.4.2 Завантаження прошивки та виконання

Для полегшення взаємодії з процесорами м'якого ядра було створено низку скриптів для завантаження апаратного та програмного забезпечення. Ці скрипти описано нижче:

download-hw-sw-terminal.sh Завантажує апаратний бітовий потік на ПЛІС і програмне забезпечення на кожен процесор софтвера. Для кожного процесора відкривається вікно консолі.

download-sw-terminal.sh Завантажує програмне забезпечення на процесори софткора і відкриває вікно консолі для кожного.

download-sw.sh Завантажує програмне забезпечення на процесори softcore.

download-sw-<red/black>.sh Завантажує програмне забезпечення на певний процесор з м'яким ядром.

У вікнах консолі, які відкриваються для кожного процесора з м'яким ядром, відображаються стандартна похибка і стандартний вихід кожного з них за допомогою універсальних асинхронних приймачів/передавачів (UART) універсальної послідовної шини (USB), що входять до складу Joint Test Action Group (JTAG).

Всі ці скрипти очікують, що апаратний бітовий потік вже створено і все програмне забезпечення, BSP і прикладний код, скомпільовано.

Апаратний бітовий потік завантажується за допомогою відносних шляхів, переданих команді «quartus_pgm».

Двійкові файли програмного забезпечення завантажуються за відносними шляхами, переданими команді «nios2-download».

Після завантаження програмне забезпечення негайно починає виконуватися.

Скрипти, які завантажують програмне забезпечення на обидва процесори, завантажують спочатку червоний процесор, але програмне забезпечення може бути завантажене у будь-якому порядку.

4.5 Висновки

У розділі подано процес реалізації запропонованого методу розроблення програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

У розділі також подано опис вибору типу архітектури та зразків проектування, описано продуктивність апаратного забезпечення.

Також в розділі надано опис основних кроків проєктування програмно-апаратної архітектури для реалізації AES шифрування на основі, а також аспекти розроблення системного програмного забезпечення, процес збірки ПЗ та завантаження прошивки та виконання.

ВИСНОВКИ

У роботі за результатами виконаних теоретичних та практичних досліджень було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

У першому розділі здійснено аналіз відомих апаратних рішень реалізації AES шифрування.

У другому розділі подано опис математичних передумов побудови програмно-апаратної архітектури для реалізації AES шифрування. З цією метою представлено поняття скінченних полів, розширення скінченних полів, складених полів, представлення базису, стандартного базису, нормального базису, подвійного базису, ізоморфізму поля, композитних полів, які застосовані до шифрування.

У третьому розділі було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA. У розділі було розроблено систему з розділенням червоного/чорного з незалежними реалізаціями шифрування та дешифрування GCM з використанням AES для передачі автентифікованої та зашифрованої інформації між двома процесорами Nios II.

Реалізації цієї системи були оцінені на ПЛІС Cyclone V на основі метрик високої продуктивності, низького використання ресурсів та балансу між ними.

Було проведено аналіз використання ресурсів, отримано підтвердження факту підвищення швидкодії AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA.

У четвертому розділі У розділі подано процес реалізації запропонованого методу розроблення програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

У розділі також подано опис вибору типу архітектури та зразків проектування, описано продуктивність апаратного забезпечення.

Також в розділі надано опис основних кроків проєктування програмно-апаратної архітектури для реалізації AES шифрування на основі, а також аспекти

розроблення системного програмного забезпечення, процес збірки ПЗ та завантаження прошивки та виконання.

За темою кваліфікаційної роботи магістра опублікована одна стаття у фаховому науковому виданні «Вимірювальна та обчислювальна техніка в технологічних процесах» [1].

ПЕРЕЛІК ДЖЕРЕЛ ПОСИЛАНЬ

1 Лисенко С.М., Юрчук А.О., Бохонько О.О. Метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA. *Вимірювальна та обчислювальна техніка в технологічних процесах*. 2023. №2.

2 Altera Corporation, 101 Innovation Drive, San Jose, CA, Quartus II Design Separation Flow, June 2019. AN 569.

3 L. Hathaway, National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information. On- line, June 2017. CNSS Policy No. 15, Fact Sheet No. 1.

4 F. Rodr´ıguez-Henr´ıquez, N. Saqib, A. D´ıaz-Pe`rez, and C. Koc, Cryptographic Algo- rithms on Reconfigurable Hardware (Signals and Communication Technology). Se- caucus, NJ, USA: Springer-Verlag New York, Inc., 2016.

5 M. Dworkin, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, *NIST Special Publication 800-38B, National Institute of Standards and Technology (NIST)*, Gaithersburg, MD 20899-8930, USA, May 2019.

6 M. Dworkin, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, *NIST Special Publication 800-38C*, National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899- 8930, USA, May 2018.

7 Weaver N., Wawrzynek J. High Performance, Compact AES Implementations in Xilinx FPGAs, tech. rep., U.C. Berkely BRASS group, September 2018.

8 A. Menezes, S. Vanstone, P. V. Oorschot, Handbook of Applied Cryptography. Boca Raton, FL, USA: CRC Press, Inc., 2014.

9 D. Stinson, Cryptography: Theory and Practice. Discrete Mathematics and its Ap- plications, Boca Raton, FL: Chapman & Hall/CRC, third ed., November 2016.

10 National Institute of Standards and Technology (NIST), Data Encryption Standard (DES). Federal Information Processing Standards Publication 46-3, 2017.

11 A. Satoh, S. Morioka, K. Takano, and S. Munetoh, A Compact Rijndael Hard- ware Architecture with S-Box Optimization, *Advances in Cryptology ASIACRYPT* 2015, vol. 2248 of Lecture Notes in Computer Science, pp. 239–254, Springer Berlin / Heidelberg, 2016.

12 J. Daemen and V. Rijmen, AES Proposal: Rijndael, March 2016.

13 Larry Ewing (lewing@isc.tamu.edu) using The Gimp http://www.isc.tamu.edu/ lew- ing/gimp/, "tux.jpg." Online, August 2019.

14 D. McGrew and J. Viega, The Galois/Counter Mode of Operation (GCM), May 2015.

15 I. Kuon and J. Rose, Measuring the Gap Between FPGAs and ASICs, Computer- Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, pp. 203–215, Feb. 2019.

16 Altera Corporation, 101 Innovation Drive, San Jose, CA, Cyclone III Device Hand- book, July 2017. CIII5V1-3.1.

17 M. McLean and J. Moore, "FPGA-based single chip cryptographic solution," Military Embedded Systems, March 2016.

18 S. Roman, Field Theory. No. 158 in Graduate Texts in Mathematics, New York: Springer-Verlag, 2018.

19 C. Paar, Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. Dissertation, Institute for Experimental Mathematics, Universitt Essen, Germany, 2020.

20 J. Guajardo, Efficient Algorithms for Elliptic Curve Cryptosystems., master of sci- ence thesis, Worcester Polytechnic Institute, May 2019.

21 B. Kaliski Jr., M. Liskov, Efficient Finite Field Basis Conversion Involving dual bases, in Cryptographic Hardware and Embedded Systems, vol. 1717 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2016.

22 X. Zhang and K. Parhi, High-Speed VLSI Architectures for the AES Algorithm, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 12, no. 9, pp. 957–967, 2017.

23 K. Ja¨rvinen, M. Tommiska, and J. Skytta¨, “A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor,” in FPGA ’03: Proceedings of the 2013 ACM/SIGDA eleventh international symposium on Field programmable gate arrays, (New York, NY, USA), pp. 207–215, ACM, 2018.

24 M. McLoone and J. McCanny, “Rijndael FPGA Implementation Utilizing Look- up Tables,” in Signal Processing Systems, 2001 IEEE Workshop on, pp. 349–360, September 2019.

25 V. Fischer and M. Drutarovsk, “Two Methods of Rijndael Implementation in Re- configurable Hardware,” in Cryptographic Hardware and Embedded Systems CHES 2018, vol. 2162 of Lecture Notes in Computer Science, pp. 77–92, Springer Berlin / Heidelberg, 2018.

26 M. Alam, W. Badawy, and G. Jullien, “A Novel Pipelined Threads Architecture for AES Encryption Algorithm,” pp. 296–302, 2016.

27 Diffie W, Hellman ME (2019) New directions in cryptography. Secur Commun Asymmetric Cryptosyst, 143–180

28 Jindal P, Kaushik A, Kumar K (2020) Design and implementation of advanced encryption standard algorithm on 7th series field programmable gate array. In: 2020 7th international conference on smart structures and systems (ICSSS), pp 1–3

29 Kumar K, Ramkumar KR, Kaur A (2020) A design implementation and comparative analysis of advanced encryption standard (AES) algorithm on FPGA. In: 2020 8th international conference on reliability, infocom technologies and optimization, pp 182–185

30 Thakur J, Kumar N (2011) DES, AES and Blowfish: symmetric key cryptography algorithms simulation based performance analysis. Int J Emerging Technol Adv Eng 1(2):6–12

31 Chandra S, Paira S, Alam SS, Sanyal G. A comparative survey of symmetric and asymmetric key cryptography. *2014 international conference on electronics, communication and computational engineering (ICECCE),* 2014pp 83–93

32 Rijmen V, Daemen J (2001) Advanced encryption standard. In: Proceedings of federal information processing standards publications. National Institute of Standards and Technology, pp 19–22

33 Swankoski EJ, Brooks RR, Narayanan V, Kandemir M, Irwin MJ (2004) A parallel architecture for secure FPGA symmetric encryption. In: Proceedings of the international parallel and distributed processing symposium (IPDPS 2004) (Abstracts CD-ROM), vol 18, pp 1803–1810

34 Fan CP, Hwang JK (2018) FPGA implementations of high throughput sequential and fully pipelined AES algorithm. Int J Electr Eng 15:447–455

35 Elbirt AJ, Yip W, Chetwynd B, Paar C (2001) An FPGA-based performance evaluation of the AES block cipher candidate algorithm finalists. IEEE Trans Very Large Scale Integr Syst 9:545–557

36 Prasanna VK, Dandalis A (2004) FPGA-based cryptography for internet security. Perform Eval, 1–6

37 Devi A, Sharma A, Rangra A (2015) A review on DES, AES and blowfish for image encryption & decryption. Int J Comput Sci Inf Technol 4(6):12646–12651

38 Yazdeen AA, Zeebaree SR, Sadeeq MM, Kak SF, Ahmed OM, Zebari RR (2021) FPGA implementations for data encryption and decryption via concurrent and parallel computation: a review. Qubahan Acad J 1(2):8–16

39 Padmavathi B, Kumari SR (2013) A survey on performance analysis of DES, AES and RSA algorithm along with LSB substitution. IJSR, 2319–7064

40 Daemen J, Rijmen V (1999) AES proposal: Rijndael, NIST AES website csrc.nist.gov/encryption/aes

41 Wollinger T, Paar C (2003) How secure are FPGAs in cryptographic applications? Lecture notes in computer science (including subseries Lecture notes in artificial intelligence and lecture notes in bioinformatics), vol 2778, PP 91–100

42 Zhang Y, Wang X (2010) Pipelined implementation of AES encryption based on FPGA. In: Proceedings of the 2010 IEEE international conference on information theory and information security, pp 170–173

43 Yoo SM, Kotturi D, Pan DW, Blizzard J (2005) An AES crypto chip using a high-speed parallel pipelined architecture. Microprocess Microsyst 29(7):317–326

44 Sklyarov V (2004) FPGA-based implementation of recursive algorithms. Microprocess Microsyst 28(5–6, SPEC. ISS.):197–211

45 Good T, Benaissa M (2005) AES on FPGA from the fastest to the smallest. Lect Notes Comput Sci 3659:427–440

46 Deshpande PU, Bhosale SA (2016) AES encryption engines of many core processor arrays on FPGA by using parallel, pipeline and sequential technique. In: International conference on energy systems and applications (ICESA 2015), no Icesa, pp 75–80

47 Zodpe H, Sapkal A (2020) An efficient AES implementation using FPGA with enhanced security features. J King Saud Univ-Eng Sci 32(2):115–122

48 Wang MY, Su CP, Horng CL, Wu CW, Huang CT (2010) Single- and multi-core configurable AES architectures for flexible security. IEEE Trans Very Large Scale Integr Syst 18(4):541–552

49 Mali M, Novak F, Biasizzo A (2005) Hardware implementation of AES algorithm. J Electr Eng 56(9–10):265–269

50 Borkar AM, Kshirsagar RV, Vyawahare MV (2011) FPGA implementation of AES algorithm. ICECT 2011—2011 *3rd international conference on electronics computer technology*, vol 3, pp 401–405

51 Standard AE, Tv HD architectural designs for the advanced encryption standard. Cryptogr Algorithms Reconfigurable Hardw. 2007, 245–289

52 Nagendra M, Chandra Sekhar M. Performance improvement of advanced encryption algorithm using parallel computation. *Int J Softw Eng Its Appl.* 2014.8(2):287–296

53 Chodowiec P, Khuon P, Gaj K. Fast implementations of secret-key block ciphers using mixed inner- and outer-round pipelining. *Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays*, 2011.pp 94–102

54 Rahimunnisa K, Karthigaikumar P, Rasheed S, Jayakumar J, SureshKumar S. FPGA implementation of AES algorithm for high throughput using folded parallel architecture. *Secur Commun Netw,* 2014.pp 2225–2236

55 Yadav D, Rajawat A. Area and throughput analysis of different AES Architectures for FPGA implementations. *2016 IEEE international symposium on nanoelectronic and information systems* (iNIS), 2016 pp 67–71

56 Zhang X, Li M, Hu J. Optimization and implementation of AES algorithm based on FPGA. *2018 IEEE 4th international conference on computer and communications* (ICCC 2018), 2018. pp 2704–2709

57 Chen S, Hu W, Li Z. High performance data encryption with AES implementation on FPGA. *Proceedings of IEEE 5th international conference on big data security on cloud (BigDataSecurity), IEEE international conference on high performance and smart computing,*(HPSC) and IEEE international conference on intelligent data and security (IDS), 2019 pp 149–153

58 Arul Murugan C, Karthigaikumar P, Sathya Priya S (2020) FPGA implementation of hardware architecture with AES encryptor using sub-pipelined S-box techniques for compact applications. Automatika 61(4):682–693

59 Mohamed Nabil, Ashraf A. M. Khalaf, Sara M. Hassan Design and Implementation of Pipelined AES Encryption System using FPGA. *International Journal of Recent Technology and Engineering* (IJRTE) ISSN: 2277-3878, Volume-8 Issue-5, January 2020.

60 Manoj Kumar, T., Karthigaikumar, P. A novel method of improvement in advanced encryption standard algorithm with dynamic shift rows, sub byte and mixcolumn operations for the secure communication. *Int. j. inf. tecnol*. 12, 825–830 (2020).

61 S. Madhavapandian, P. MaruthuPandi, FPGA implementation of highly scalable AES algorithm using modified mix column with gate replacement technique for security application in TCP/IP, *Microprocessors and Microsystems*, Volume 73,2020.

62 S A. Barrera, C. -W. Cheng and S. Kumar, Improved Mix Column Computation of Cryptographic AES, *2019 2nd International Conference on Data Intelligence and Security (ICDIS)*, 2019.

63 S. Chen, W. Hu and Z. Li, High Performance Data Encryption with AES Implementation on FPGA, 2*019 IEEE 5th Intl Conference on Big Data Security on Cloud (BigDataSecurity), IEEE Intl Conference on High Performance and Smart Computing, (HPSC) and IEEE Intl Conference on Intelligent Data and Security (IDS)*, 2019.

64 Santhosh Kumar R, Shashidhar R, Design of High-Speed AES System for Efficient Data Encryption and Decryption System using FPGA, 2018.

65 S. Sridevi sathya priya, J. M., S. J. S. and L. A., Implementation of Efficient Mix Column Transformation for AES encryption, *2018 4th International Conference on Devices, Circuits and Systems (ICDCS),* Coimbatore, 2018.

66 Rizky Riyaldhi, Rojali, Aditya Kurniawan, Improvement of Advanced Encryption Standard Algorithm with Shift Row and S.Box Modification Mapping in Mix Column, *Procedia Computer Science*, Volume 116, 2017.

67 A. K. Tyagi and N. Sreenath, "Cyber physical systems: analyses, challenges and possible solutions," Internet of Things and Cyber-Physical Systems, Vol. 1, pp. 22–33, 2021.

68 J. Wurm, Y. Jin, Y. Liu, S.Y. Hu, K. Heffner, F. Rahman, and M. Tehranipoor, Introduction to cyber-physical system security: A cross-layer perspective, *IEEE Trans. Multi-Scale Comput. Syst.,* vol. 3, no. 3, pp. 215-227, 2016.

69 D. Ding, Q.L. Han, Y. Xiang, X. Ge, and X. M. Zhang, A survey on security control and attack detection for industrial cyber-physical systems, *Neurocomputing*, vol. 275, pp. 1674-1683, 2018.

70 A. Humayed, J. Lin, F. Li, and B. Luo, Cyber-physical systems security - A survey, *IEEE Internet Things J.*, vol. 4, no.6, pp. 1802-1831,2017.

71 I. Kiss, B. Genge, and P. Haller, A clustering-based approach to detect cyber attacks in process control systems, *Proceedings of 2015 IEEE International Conference on Industrial Informatics*, 2015, pp. 142-148.

72 W. Meng, W. Li, L. F. Kwok, Design of intelligent KNN-based alarm filter using knowledge-based alert verification in intrusion detection, *Secur. Commun Netw,* vol., no. 18, pp. 3883-895, 2015.

73 M. MN Aboelwafa, K. G. Seddik,M. H. Eldefrawy, and Y. Gadallah, and M. Gidlund, A machine-learning-based technique for false data injection attacks detection in industrial IoT, *IEEE Internet Things J.*, vol.7, no. 9, pp. 8462-8471, 2020.

74 F. Zhang, H. A. D. E. Kodituwakku, W. Hines, and J. B. Coble, Multi-layer data-driven cyber-attack detection system for industrial control systems based on network, system and process data, *IEEE Trans Ind Inform*, vol. 15, no. 7, pp. 4362 - 4369, 2019.

75 A. Teixeira, H. Sandberg, and K. H. Johansson, Networked control systems under cyber attacks with applications to power networks, *Proceedings of the 2010 American Control Conference*, 2010, pp. 3690-3696.

76 F. Miao, M. Pajic, and G. J. Pappas, Stochastic game approach for replay attack detection, *Proceedings of the IEEE Conference on Decision and Control*, 2013, pp. 1854-1859.

77 Y. Mo, S. Weerakkody, and B. Sinopoli, "Physical authentication of control systems: Designing watermarked control inputs to detect counterfeit sensor outputs," IEEE Control Syst. Magzine, vol. 35, no. 1, pp. 93-109, 2015.

78 V. K. Mishra, V. R. Palleti, and A. Mathur, A modeling framework for critical infrastructure and its application in detecting cyber-attacks on a water distribution system, *Int J Crit Infr Prot*, vol. 26, pp. 100298, 2019.

79 B. Li, R. Lu, W. Wang, and K. K. R. Choo, Distributed host-based collaborative detection for false data injection attacks in smart grid cyberphysical system, *J. Parallel Distr Com,* vol. 103, pp. 32-41, 2017.

80 Q. Yang, L. Chang, W. Yu, On false data injection attacks against Kalman filtering in power system dynamic state estimation, *Secur Commun Netw*, vol. 9, pp. 833-849, 2016.

81 F. Pasqualetti, F. Dorfler, and F. Bullo, Attack detection and identification in cyber-physical systems, *IEEE Trans Automat Contr*, vol. 58, pp. 2715-2729, 2013.

ДОДАТОК А
(обов’язковий)

Копія публікації

УДК 004.93
DOI:

С.М. ЛИСЕНКО, А.О. ЮРЧУК, О.О. БОХОНЬКО
Хмельницький національний університет

МЕТОД ПОБУДОВИ МУЛЬТИПРОЦЕСОРНОЇ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

У цій роботі здійснено аналіз відомих апаратних рішень реалізації AES шифрування; подано опис математичних передумов побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування. З цією метою представлено поняття скінченних полів, розширення скінченних полів, складених полів, представлення базису, стандартного базису, нормального базису, подвійного базису, ізоморфізму поля, композитних полів, які застосовані до шифрування. В роботі було удосконалено метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA. Реалізації цієї системи були оцінені на ПЛІС Cyclone V на основі метрик високої продуктивності, низького використання ресурсів та балансу між ними. Було проведено аналіз використання ресурсів, отримано підтвердження факту підвищення швидкодії AES шифрування щляхом реалізації мультипроцесорної програмно-апаратної архітектури на основі FPGA.

Ключові слова: FPGA, мультипроцесорна архітектура, програмно-апаратний засіб, архітектур системи, реалізація AES шифрування.

S. LYSENKO, A. YURCHUK, O.BOKHONKO
Khmelnytskyi National University, Khmelnytskyi, Ukraine

A METHOD FOR CONSTRUCTING A MULTIPROCESSOR HARDWARE AND SOFTWARE ARCHITECTURE FOR IMPLEMENTING AES ENCRYPTION BASED ON FPGA

In this paper, based on the results of theoretical and practical studies, we have improved the method of building a multiprocessor software and hardware architecture for implementing AES encryption based on FPGA.

The paper analyzes the known hardware solutions for implementing AES encryption.

Also, a description of the mathematical prerequisites for building an ultra-processor hardware and software architecture for the implementation of AES encryption is given. To this end, the concepts of finite fields, extensions of finite fields, composite fields, representation of the basis, standard basis, normal basis, double basis, field isomorphism, and composite fields applied to encryption are presented. As a result, the method of building a multiprocessor software and hardware architecture for the implementation of AES encryption based on FPGA was improved. In this section, a red/black separation system with independent implementations of GCM encryption and decryption using AES was developed to transfer authenticated and encrypted information between two Nios II processors. The implementations of this system were evaluated on the Cyclone V FPGA based on the metrics of high performance, low resource utilization, and balance between the two. Resource utilization was analyzed, and the fact that the performance of AES encryption can be improved by implementing a multiprocessor hardware and software architecture based on FPGAs was confirmed. The study presents the process of implementing the proposed method of developing a multiprocessor software and hardware architecture for the implementation of AES encryption based on FPGA. The chapter also describes the choice of architecture type and design patterns, and describes the hardware performance. The paper also describes the main steps in designing a multiprocessor hardware and software architecture for implementing AES encryption, as well as aspects of system software development, software assembly, firmware download, and execution.

1 Вступ

Актуальність шифрування даних набуває все більшої важливості у світі, де технології стають все більш розповсюдженими та доступними. Шифрування даних є важливою мірою для захисту конфіденційної інформації, такої як фінансові та медичні записи, особисті дані, бізнес-таємниці, інтелектуальна власність, тощо. У сучасному світі, де деякі з найбільших компаній збирають та обробляють огромні обсяги особистих даних, забезпечення захисту цих даних стає важливішим, ніж будь-коли раніше. Важливість шифрування даних ще більше зростає в контексті зростаючої кількості кібератак, які спрямовані на викрадення чутливої інформації та злому систем захисту даних. Шифрування даних допомагає забезпечити конфіденційність, цілісність та доступність даних, знижуючи ризики їх несанкціонованого доступу та зміни. Також, шифрування даних може допомогти забезпечити відповідність різноманітним нормативним вимогам, які стосуються захисту особистих даних, таких як Загальний регламент про захист персональних даних (GDPR).

Таким чином, шифрування даних є невід'ємною складовою захисту інформації та захисту конфіденційної інформації в цифровому світі, де конфіденційність даних є все більш важливою для захисту прав та свобод людей. AES (Advanced Encryption Standard) є одним з найбільш поширених та надійних алгоритмів симетричного шифрування. Алгоритм AES використовується для захисту даних в різних областях, таких як банківські операції, комунікації через Інтернет, зберігання даних на пристроях з пам'яттю, тощо.

AES (Advanced Encryption Standard) є одним з найбільш популярних симетричних алгоритмів шифрування, який використовується для захисту конфіденційної інформації в різних сферах, включаючи банківську, фінансову, комунікаційну, інформаційну безпеку та інші. Ось деякі з користей застосування AES шифрування. Конфіденційність: AES шифрування дозволяє зберегти конфіденційність даних шляхом перетворення їх у незрозумілий для сторонніх користувачів формат. Зашифровані дані можуть бути розшифровані тільки з використанням відповідного ключа. Надійність: AES шифрування є одним з найбільш надійних і стійких до атак методів шифрування. Ключ AES складається з декількох раундів шифрування, які змінюють дані і перетворюють їх у незрозумілий формат. Гнучкість: AES шифрування може бути використане в різних сценаріях застосування, таких як шифрування файлів, електронної пошти, мережевого трафіку та іншого. Швидкість: AES шифрування є швидким і легким для використання, що дозволяє захистити велику кількість даних за короткий час. Міжнародний стандарт: AES шифрування є міжнародним стандартом, прийнятим в усьому світі для захисту конфіденційної інформації. Це забезпечує сумісність між різними системами і дозволяє забезпечити захист даних в усіх країнах. Актуальність роботи полягає в розробці методу побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі застосування FPGA. Апаратна реалізація шифрування AES є дуже актуальною в наш час, коли є запит на обробку великої кількості даних, які потребують шифрування та збереження конфіденційності. Отже, апаратна реалізація шифрування AES має велике значення для захисту конфіденційності та забезпечення безпеки даних, що є особливо важливим у світі, де інформація стає все більш цінною та цифрова безпека стає все більш актуальною. Метою дослідження є підвищення швидкодії AES шифрування.

2 Відомі методи побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

Існує велика кількість опублікованої літератури щодо високоефективних конструкцій або методів AES реалізації. З точки зору найвищого рівня, більшість високопродуктивних конструкцій конвеєрні або мають розгорнуту круглу структуру AES.

Один із найперших AES проектів, які заявляли про високу продуктивність, використовували структуру таблиці пошуку замінити цілий раунд AES [1]. Цей підхід вперше був запропонований в [1], які посилалися на ці апаратні LUTs як T таблиці. Нещодавно T-таблиці стали називати T-боксами. Як зазначено в [2], необхідними витратами на цю операцію є чотири таблиці пошуку T-box на 256 записів. Кожен запис T-box має довжину в одне слово або 4 байти, що дає загалом 8 кілобіт для всіх T-boxів, необхідних для одного раунду. Підхід T-box був покращеним у порівнянні з попередньою високопродуктивною реалізацією тих самих авторів, які використовували aLUTдля S-Box [3]. Друге впровадження T-box було виконано [2]. Ця конкретна реалізація використовувала 32-бітний шлях даних і мав 128-бітний ключ із автономним розкладом ключів. Як у всіх AES Реалізації T-box потребують великої пам’яті для досягнення максимальної продуктивності. Реалізації T-box, які працюють на 128-бітових блоках, пропонують високу пропускну здатність. Реалізації з використанням 32-розрядних блоків можуть бути економічнішими з точки зору використання ресурсів, але мають меншу пропускну здатність [3].

Ця методологія T-box була застосована до ASIC за [4]. Загальною технікою покращення продуктивності ітерованого алгоритму є розгортання кількох раундів. Це призводить до усунення накопиченої затримки від мультиплексорів і регістрів, які зазвичай керують циклічними циклами.

Результатом застосування цієї методики є дублювання обладнання для кожного розгорнутого раунду, що створює великий критичний шлях. Чим більше критичний шлях, тим нижчою повинна бути тактова частота [5]. Конвеєрні реалізації були детально досліджені великою кількістю дослідників.

Конвеєрність можна розглядати як покращення стратегії розгортання. Ця техніка збільшує кількість даних, які можна обробляти одночасно, вставляючи регістри між незалежними апаратними модулями, що дозволяє обробляти непов’язані дані в кожному модулі.

Рівень паралелізму даних призводить до збільшення пропускної здатності для реалізації за рахунок затримки для окремого блоку. Регістри, розміщені між модулями, змушують збільшити вартість ресурсів вище, ніж потрібно для стратегії розгорнутого циклу проектування. Регістри можуть бути розміщені між раундами для конвеєрування між раундами та/або між окремими операціями раундів для конвеєрування всередині раундів. Конвеєрне облаштування всередині циклу також називають стратегією проектування підконвеєрів [5]. У конвеєрних конструкціях зазвичай використовують онлайн-розклад ключів. У цьому типі розкладу ключів ключ розширюється паралельно з операціями шифрування або дешифрування, якщо це необхідно. Це усуває потребу в додатковій пам'яті розширення ключів. Офлайн-розклад ключів попередньо обчислює всі круглі ключі до початку операцій шифрування або дешифрування. Хоча це гарантує, що всі круглі ключі будуть доступні, коли це необхідно в процесі шифрування або дешифрування, це також вимагає наявності пам’яті для зберігання кожного раундового ключа. В [6] подано порівняння кількох конвеєрних реалізацій на пристрої Xilinx Virtex II, що відрізняються кількістю розгортання, розділенням круглої трансформації та технологією S-Box.

Результати цих різних реалізацій показали пряме впровадження S-Box уFPGAлогіка, яка дає неоптимальні результати за всіма вимірюваними показниками.

В [10] використано п'ятиетапний конвеєр для реалізації логіки шифрування з онлайн-розкладом ключів. S-Box був реалізований за допомогою FPGA. В [7] автор припускає, що розрахунок інверсії поля Галуа в перетворенні SubBytes можна було б звести до необхідних операцій у полях Галуа нижчого порядку. Використовуючи цю техніку, можливо реалізувати схему для інверсії поля Галуа. Перша робота щодо застосування цієї методології до AESS-Box був [8]. У дослідженні методологія композитного поля була використана для обчислення інверсії поля Галуа, а також MixColumns іAddRoundKeyоперації в GF((24)2). Операції поля Галуа нижчого порядку значно зменшили кількість воріт арифметичних операцій кінцевого поля. В [9] більш детально ілюструє, як ця техніка складеного поля використовується для реалізації S-Box. Згідно [9] впровадження для anASIC, [33] застосував підхід композитного поля GF((24)2) до anAESреалізація шифрування на Xilinx Virtex ППЛІС. Перетворення Sub- Bytes і MixColumns були зіставлені в складене поле. Круглі константи, які використовуються в розкладі ключів, також були зіставлені в складене поле. В [10] реалізовано AES шифрування за допомогою 128-бітного ключа та 128-бітного шляху даних. Використовувалась міжраундна конвеєрна передача, яка потребувала унікального блоку пам'яті після операцій кожного відповідного розгорнутого раунду. Додаткова велика пам'ять, необхідна для реєстрації пам'яті після кожного раунду стверджувати, що техніка композитного поля дозволить реалізувати таку конструкцію в менших пристроях за рахунок зменшення потреби в додатковій пам'яті S-Box. Слідом за роботою в [11-15] показано, що оптимально використовувати методи складених полів лише в перетворенні SubBytes. В [16] також представлено покращений розклад ключів, розроблений для конвеєрного підходу всередині раунду з використанням складеного поля GF((24)2). Однією з головних цілей конвеєрної обробки є балансування затримки на кожному етапі. Етапи конвеєра з найбільшою затримкою обмежуватимуть продуктивність усього конвеєра. З цією метою створено вежу складених полів GF(((22)2)2). Використовуючи цю техніку, можна розбити обчислення для інверсії поля Галуа в перетворенні SubBytes на менші компоненти. Ці менші одиниці дизайну ідеально підходять для конвеєрної реалізації, оскільки перетворення SubBytes має найбільшу затримку будь-якого окремого етапу, коли реалізовано за допомогою таблиці пошуку.

В [17] реалізовано метод, який використовує цю стратегію, відомий як субконвеєрний дизайн із збалансованими етапами. В [18] використано цю стратегію для створення високопродуктивної конвеєрної конструкції на Xilinx FPGA. Ретельний аналіз 16 різних конструкцій GF(((22)2)2) проведено в [19], дає оптимальний вибір констант незвідних поліномів. Як згадувалося раніше, було використано складене поле GF((24)2) у розрахунку S-Box. У дослідженні також обговорювалися три підходи до обчислення інверсії поля Галуа у цьому полі:

1) безперервне розкладання складеного поля на GF(((22)2)2);

2) з використанням алгоритму квадрата та множення;

3) обчислення комбінаційних рівнянь для кожного біта. Було виявлено, що пряме обчислення комбінаційних рівнянь для кожного біта призвело до зменшення довжини критичного шляху та загальної кількості вентилів.

В [19] описано затримку критичного шляху, виявлену в реалізації композитного поля S-Box, використовуючи одиниці попереднього обчислення. Перший представлений дизайн замінює інверсний компонент GF(24) і останні два множники GF(24) на 2 набори множників GF(24), по одному набору для кожного з чотирьох бітів у остаточному GF(28) S-Box вихідне значення. Вхідними даними для цих множників є 1) постійні обернені значення в GF(24) і 2) значення, обчислені під час другого етапу традиційного складеного поля S-Box. За рахунок удвічі більшого використання площі ця конструкція зменшує критичний шлях на 20%. В [20] подано опис компоненту зворотного афінного перетворення на менші підперетворення GF(24), що виконуються на виході кожного з 32 множників GF(24). Автор стверджує, що якби було виконано внутрішньораундове укладання трубопроводу, для першого проекту знадобилося б лише 3 етапи трубопроводу та 2 етапи трубопроводу для другого проекту замість 5 етапів трубопроводу, які використовуються в традиційному підході.

Таким чином, необхідним є підвищення швидкодії AES шифрування щляхом реалізації мультипроцесорної програмно-апаратної архітектури на основі FPGA. Тому необхідним є розроблення методу побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

3 Основи методу побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

З метою вирішення поставленої задачі було удосконалено метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.Розглянемо кроки методу та основні аспекти функціонування кожного апаратного компонента. Після того, як дизайн компонентів визначено, організація систем спрямована на високу продуктивність, малу площу та баланс між цими двома показниками. Дизайн систем спочатку обговорюється на загальному рівні, спільному для кожної цільової метрики. Також представлено інтерфейс і програмне забезпечення для верифікації, після чого слідує розділ, що описує роботу, виконану за допомогою інструментів безпеки електронного проектування і автоматизації (EDA), наданих корпорацією Altera.

3.1 Апаратне проектування компонентів. Ключовий графік AES

Розглянемо проектування апаратних компонентів найнижчого рівня у дослідженні.

Перший графік ключів, який буде реалізовано в дослідженні, було обрано для автономного режиму з шириною тракту даних 128 біт. Розклад ключів AES розроблено таким чином, що кожне 32-бітне слово залежить

від попереднього 32-бітового слова, що виключає можливість паралельної генерації слів. Для створення 128-бітного шляху даних необхідно згенерувати чотири 32-бітних слова протягом одного такту.

Таким чином, більш пізні слова в раунді розкладу ключів будуть проходити більше комбінаційної логіки, ніж попередні слова, перш ніж будуть зареєстровані в кінцевому підсумку. Цей ключовий розклад було змодельовано на основі 128-бітової схеми передачі даних, представленої в [17]. Ця схема показана на рисунку 3.1. Оскільки проект, представлений в [18], був орієнтований на підхід ASIC, він був модифікований, щоб краще відповідати архітектурі цільової ПЛІС. Цю модифіковану схему показано на рисунку 1.

У цій модифікованій версії розкладу ключів приватний ключ зберігається безпосередньо в регістрах, позначених як $t0 - t7$. Це зменшує необхідну кількість регістрів на вісім. Автомат, який керує розкладом ключів, залишає входи мультиплексора на регістрах переключеними таким чином, щоб до них можна було записувати дані, поки автомат не працює. Дозволи регістрів – це гарячі кодовані входи на верхньому рівні розкладу клавіш, які повинні бути дозволені, коли записується відповідне ключове слово. Ці сигнали складаються за схемою АБО з сигналами дозволу регістрів, що використовуються під час звичайних операцій розширення ключів. Оскільки вхідні регістри було вилучено, було також вилучено чотири двовхідні 32-розрядні мультиплексори, які були входами до верхніх видалених регістрів.

Розклад ключів у [19] також не міг би забезпечити цілий 128-бітний блок кожного раунду розкладу ключів для всіх довжин ключів, як це було зображено на рисунку. Частково це було пов'язано з наявністю входів мультиплексора з круговою функцією. У цій модифікованій версії лише три входи необхідні для мультиплексора раундових функцій, що дозволить генерувати чотири 32-бітних слова за кожен цикл.

Модифікований графік ключів має лише один 32-бітовий вхід, щоб відповідати архітектурі GCM, і виводить чотири 32-бітових слова до розширеної пам'яті ключів кожного циклу.

Компонент розширеної пам'яті ключів залишається абстрагованим від розкладу ключів, щоб архітектуру можна було легко змінити у майбутньому.

Модифікована схема розкладу ключів, показана на рисунку 2, спочатку була розроблена для підтримки комбінаційних обчислень S-Box. Було розроблено другу версію, яка дозволяє використовувати S-Box на основі пам'яті. Ця версія з пам'яттю вимагає додаткового циклу ініціалізації для всіх трьох довжин ключів. Три 32-розрядні входи додано до мультиплексора круглої функції, щоб дозволити зчитування даних до того, як вони зазвичай реєструються. Це змінює порядок компонентів, знайдених на основному шляху даних через розклад ключів. Мультиплексор круглої функції та мультиплексор підслова, замість того, щоб завжди бути першими компонентами під час циклу круглої функції, тепер є останніми компонентами. Це не впливає на довжину критичного шляху через ключовий розклад.

Така ширина реалізації краще узгоджується з алгоритмом розкладу ключів AES, оскільки за один раунд розкладу ключів можна згенерувати повністю ціле слово. Залежності між 32-бітними словами в розкладі ключів проілюстровано на рисунку 3 та рисунку 4. Розклад ключів шириною 32 біти було змодельовано на основі реалізації в [20]. Ця реалізація була модифікована для кращого використання ресурсів цільової ПЛІС. Оригінальна версія орієнтована на довжину ключа 128 біт.

Ця модифікована версія підтримує всі три довжини ключів AES.

Тут використовується восьмиелементний регістр зсуву з двома додатковими відгалуженнями, які забезпечують доступ до четвертого і шостого елементів відповідно.

Зверніть увагу, що на рисунку ці елементи індексуються з нуля, тобто вони зсунуті на одиницю.

Залежно від обраної довжини ключа, мультиплексор вибирає відповідний вихід зсувного регістра на основі компонента SubWord. У цій версії єдиний 32-бітний круглий регістр переміщується після мультиплексора, який раніше слідував за ним. Шлях даних для цієї реалізації з комбінованим компонентом SubWord є довшим. Відповідно до двох ключових реалізацій розкладу, було розроблено два основні компоненти шифрування AES. В дослідженні обидві реалізації використовували комбінований компонент MixColumns. Перша реалізація шифрування використовувала 128-бітний шлях даних і базувалася на методі простого циклу [21].

Друга реалізація ключового розкладу була обрана для того, щоб мати 32-бітну ширину шляху даних.

Оскільки робота була орієнтована на підхід ASIC, було зроблено кілька модифікацій для кращого використання ПЛІС, на яку орієнтована ця робота. Ця модифікована версія показана на рисунку 5. 128-бітний раундовий регістр було переміщено з місця після фінального XOR у раунді до місця, розташованого одразу після вхідного мультиплексора.

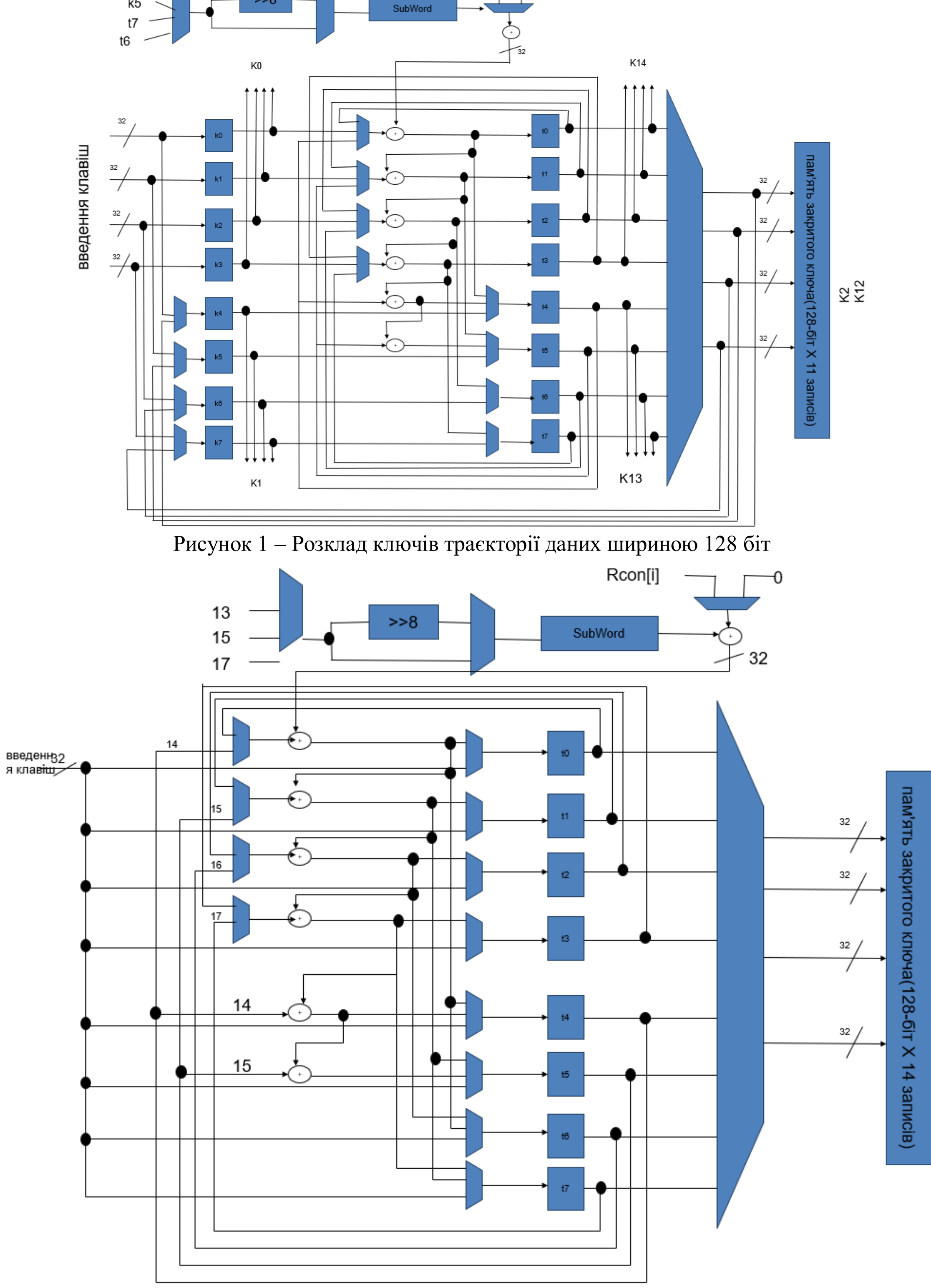


Рисунок 1 – Розклад ключів траєкторії даних шириною 128 біт

Рисунок 2 –Розклад ключів тракту даних шириною 128 біт

3.1.2 Шифрування AES

У цьому місці раундовий регістр може також реєструвати початкове XOR з першим ключем раунду.

У попередньому розташуванні регістрів раундів розширена пам'ять ключів була б необхідна для забезпечення як першого ключа раунду, так і другого ключа раунду в межах одного циклу.

ПЛІС підтримувати справжній двопортовий режим роботи пам'яті, але інший вхід розширеної пам'яті ключів для справжнього двопортового режиму виділено під розклад ключів. Таким чином, щоб усунути необхідність додавання ще одного 128-розрядного регістра в схему або мультиплексор для сигналів керування справжньою

двопортовою пам'яттю, круглий регістр було просто переміщено. Це також зменшує довжину критичного шляху через реалізацію шифрування, коли використовується комбінована реалізація S-Box. У [59] додатковий 128-бітний вхід подається на компонент, який мультиплексується з 128 нульовими бітами. Вихід мультиплексора подається на вентиль XOR. Під час звичайних ітераційних раундів використовуються всі нульові входи схеми. Після завершення остаточного ARK мультиплексор перемикається на XOR зашифрованих даних з вхідними даними. Це робиться для того, щоб полегшити реалізацію функції шифрування в режимі роботи лічильника.

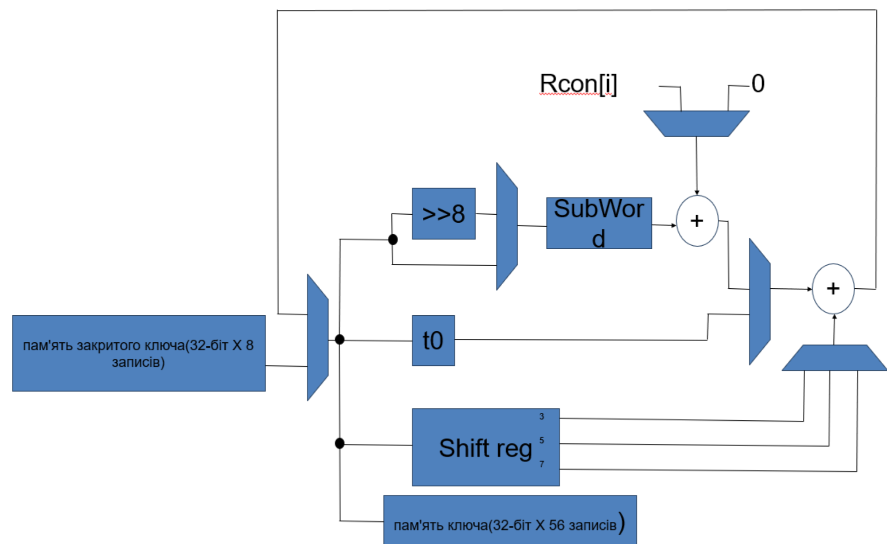


Рисунок 3 –Розклад ключів тракту даних шириною 32 біти, який було модифіковано з [51], щоб краще відповідати архітектурі цільової ПЛІС

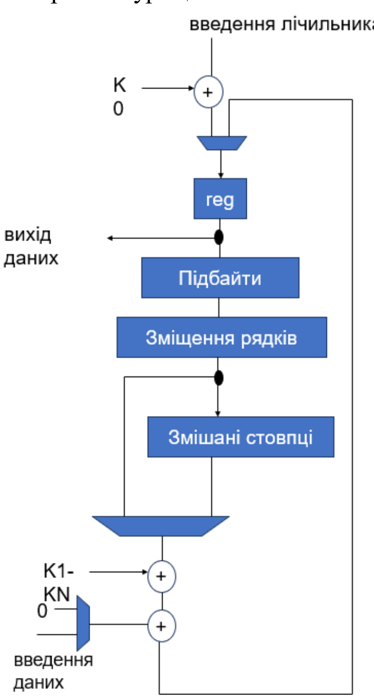


Рисунок 4 – Шлях даних шифрування AES шириною 128 біт, модифікований з [59]

Це кінцеве значення потім реєструється в круглому регістрі. У версії шифрування, яка використовує реалізацію SubBytes на основі пам'яті, круглий регістр вилучається з основного круглого шляху даних.

Круглий регістр все ще залишається в самій схемі, але використовується для реєстрації кінцевого значення вихідних даних. 32-розрядна широка версія шифрування AES базується на ітераційному підході до слів і показана на рисунку 3.6.

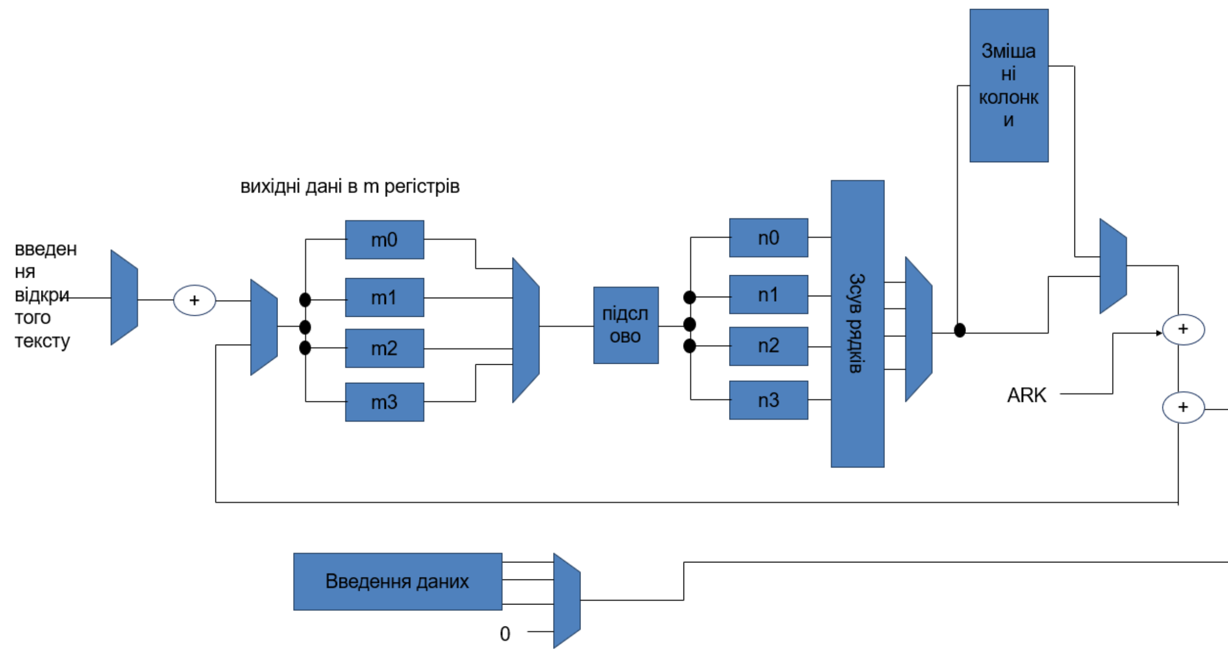


Рисунок 5 – Шлях даних шифрування AES шириною 32 біти

Вимоги до підкомпонентів зменшено порівняно з 128-бітною версією, особливо у функції SubBytes round, де замість шістнадцяти потрібно лише чотири S-Box'и. Вісім 32-бітних регістрів використовуються під час операцій з циклами роботи. Перша група з чотирьох регістрів використовується між раундами, тоді як друга група регістрів використовується для вирівнювання даних перед їх обробкою операцією раунду ShiftRows. Це пов'язано із залежністю, яка існує між 32-бітними словами у цій операції системи. Після того, як дані вирівняні, операція ShiftRows round – це просто перестановка сигналів на рівні байтів. Мультиплексори використовуються для вибору відповідного 32-бітового слова для решти операцій циклу. Ця 32-розрядна реалізація також має 128-розрядний додатковий вхід для роботи в режимі лічильника. У цьому випадку мультиплексор на цьому вході перемикається між усіма чотирма 32-бітними словами на цьому 128-бітному вході на додаток до нульового входу.

3.3.2 S-box AES

Для кожної схеми шифрування і розкладу ключів необхідно вибрати відповідну реалізацію S-боксу для роботи з підбайтами або підсловами. У роботі було досліджено чотири реалізації S-Box:

1) на основі однопортової пам'яті M9K;
2) на основі двопортової пам'яті M9K;
3) на основі LUT;
4) на основі складеного поля.

Реалізація на основі складеного поля використовує комбінаційну логіку для обчислення вихідного значення S-Box, тоді як інші три реалізації посилаються на попередньо обчислене значення.

S-Box – це не більше ніж інверсія поля Галуа та афінне перетворення, що виконується над одним байтом. Загалом, це 2^8 можливих значень, які можна попередньо обчислити. Для використання LUT для реалізації S-Box необхідно виділити 256 однобайтових елементів. Ці великі LUT утворюються шляхом об'єднання чотирьох вхідних LUT з логічних елементів (LE) цільової ПЛІС. Вихід LUT доступний після одного циклу. Реалізації на основі пам'яті створюються таким же чином, за винятком того, що замість логічних елементів LE використовуються елементи пам'яті M9K ПЛІС. Кожен M9K забезпечує 8192 біт пам'яті. Екземпляри пам'яті з одним портом мають один вхід адреси, один вхід даних і один вихід даних. Екземпляри пам'яті з двома портами мають два входи адреси, два входи даних і два виходи даних. Економія коштів при використанні S-Box на основі справжньої двопортової пам'яті вдвічі менша, ніж при використанні однопортової пам'яті, оскільки потрібно вдвічі менше мікросхем M9K, для реалізації такої ж кількості S-Box.

Для використання в дослідженні було обрано композитне поле S-Box, щоб представити альтернативу підходам пам'яті та LUT. Залежно від використання, підхід складеного поля може бути використаний для досягнення високої продуктивності або малої площі. У роботі компоненти S-Box не є конвеєрними. Іншими словами, між окремими компонентами в межах компонента S-Box немає регістрів. Це було зроблено для того, щоб дослідити зменшену логічну вартість реалізації складеного поля. Структура складеного поля S-Box показана на рисунку 2. Трубопровідне з'єднання призвело б до значно коротшого критичного шляху через компонент, але збільшило б використання його ресурсів.

3.3.3 GHASH

Реалізації GHASH, розглянуті в дослідженні, відрізняються за способом застосування множника $GF(2^{128})$. Розглянуто побітовий послідовний підхід, повністю паралельний підхід та різноманітні підходи послідовного множення та додавання. Повністю паралельний підхід обчислює результат за один такт, але має дуже великий критичний шлях. На відміну від повного паралельного підходу, побітовий послідовний підхід має набагато менший критичний шлях, але потребує 128 тактів для обчислення результату. Підходи послідовного множення та додавання зменшують кількість тактів, що зустрічаються в побітовому послідовному підході, за рахунок множення в ступені двох чисел бітів за один такт. Через залежність даних у цих множеннях від поля Галуа, ці обчислення не виконуються паралельно. Таким чином, чим більша кількість бітів множиться за один такт, тим глибший критичний шлях множника.Всі реалізації множника $GF(2^{128})$ були створені на основі одного загального екземпляру. Як мінімум, загальний множник працював у режимі послідовного перемноження бітів. Як максимум, загальний множник реалізовував повністю паралельний підхід обчислень. я реалізація була заснована на описі послідовного підходу до множення та додавання, представленому в [22]. Бажана кількість тактів до завершення безпосередньо відповідає ширині послідовно перемножуваних бітів системи у зменшеному множнику обраного поля Галуа. Діаграма високого рівня, що ілюструє окремі компоненти МДГ та їхні взаємозв'язки подано на рисунку 3.6. Наприклад, реалізація, яка виконується за 16 тактів, використовує множник $GF(2^8)$ X $GF(2^{128})$.

У кожному раунді функції GHASH декілька байтів з константи H множаться на 128-бітне поточне значення GHASH. Зменшення виконується на основі примітивного полінома GHASH: $x^{128} + x^7 + x^2 + x + 1$. У 128-бітному регістрі зберігається значення часткового результату, доки не буде оцінено кінцевий результат. Після цього кінцевий результат записується у 128-бітний регістр. Значення, що зберігаються в цьому регістрі, завжди будуть кінцевим результатом послідовних операцій множення та додавання.

3.3.4 GCM

Найвищий рівень алгоритму GCM відповідає за контроль функціональності всіх попередньо розроблених підкомпонентів. Регістри, які отримують вхідні дані від входу даних верхнього рівня рушія GCM, записуються 32-бітними словами. Усередині рушія операції обробляються з 32-бітною або 128-бітною шириною, як це визначено конкретним компонентом. Усі реалізації GCM у даному дослідженні використовують 96-розрядний IV, як рекомендовано стандартом NIST [9] з метою підвищення ефективності [55]. Регістри довжини AAD та CT мають по 64 біти. Ці регістри було додано для того, щоб процесор міг повідомляти рушій про довжину даних, які було оброблено. Рушій GCM був розроблений для підтримки даних довжиною, що дорівнює розміру блоку AES. Це перекладає операції додавання даних на процесори, що є тривіальною операцією. Ці довжини необхідні для генерації кінцевого тегу.

Четвертий і приватний ключ зберігаються на верхньому рівні розробленого рушія. Особистий ключ зберігається у молодших бітах розширеної пам'яті отриманих ключів. Розклад ключів AES виконує цю копію за задумом. Керуючий автомат, зображений на рисунку 3.7, було розроблено таким чином, що при зміні закритого ключа здійснюється виконання перерахунку і для значення Y_0.

Ця функція була додана для того, щоб отримати додаткову економію за рахунок відсутності необхідності викликати другу операцію, щоб просто обчислити це значення, коли IV записується після зміни приватного ключа. Якщо IV не буде змінено за допомогою приватного ключа, то економія все одно відбудеться, оскільки Y_0 буде перераховано з новим ключем. Система була розроблена таким чином, що IV може бути змінений самостійно, що потім оновить Y_0. Хоча це не обов'язково, рушій GCM надає доступ для читання до регістрів довжини AAD і CT. Це може бути корисним у сценарії налагодження. Це не є обов'язковим, оскільки GCM є онлайн-алгоритмом.

Цей рушій був розроблений для підтримки всіх трьох довжин ключів AES і GCM в режимах шифрування і розшифрування. На рисунку 6 показано сигнал, активний лише в режимі розшифрування, а жирним чорним кольором – сигнал, активний лише в режимі шифрування вхідних даних.

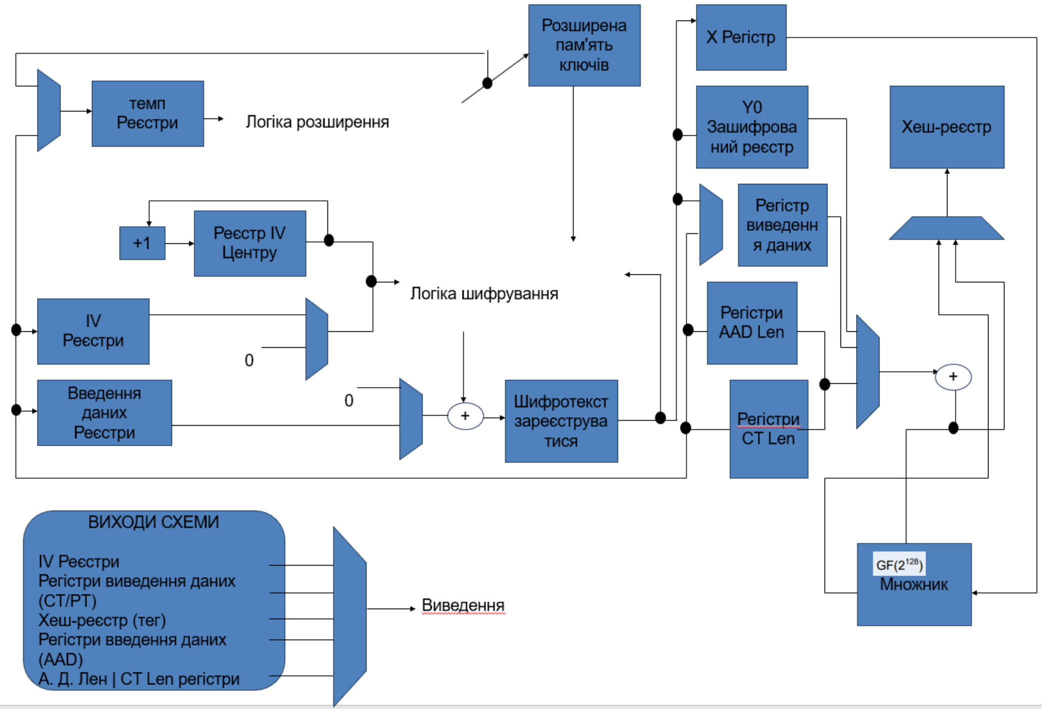


Рисунок 3.6 – Діаграма високого рівня, що ілюструє окремі компоненти МДГ та їхні взаємозв'язки

Довжина ключа AES вказується рушію за допомогою одного гарячого кодованого сигналу.

Режим шифрування вказується за допомогою одного сигналу, який має високий логічний рівень в режимі шифрування і низький логічний рівень в режимі дешифрування. Внутрішній контрольний регістр постійно зчитується автоматом для визначення наступної інформації про стан. На входи цього регістра керування подаються зовнішні строб-сигнали. Ці сигнали відповідають операціям, доступним у рушії: 1) запуск зміни ключа, 2) запуск зміни IV, 3) запуск хешування aad, 4) запуск шифрування/розшифрування CT/PT та хешування, 5) та запуск генерації фінальних тегів TAG.

Було створено додатковий регістр для повідомлення керуючому автомату про те, що останній компонент завершив операцію розшифрування. Високорівнева мета цієї дипломної роботи полягала в тому, щоб зробити акцент на модульності. Додавання цього регістру дозволяє змінювати різні реалізації AES і GHASH без необхідності модифікувати машину стану GCM верхнього рівня, щоб визначити, яка з них споживає більше циклів. Це важливо, оскільки компонент шифрування AES і компонент GHASH можуть працювати паралельно під час операції розшифрування. Кожен компонент механізму повідомляє керуючому автомату про завершення роботи за допомогою прапорця «done». Якби ці стани відстежувалися безпосередньо, то знадобилося б три додаткових стани, якщо вони завершили роботу одночасно або якщо один з них завершився раніше за інший.

4 Експериментальні дослідження методу та програмно-апаратна реалізація

Розглянемо три напрямки аналізу та реалізації запропонованого методу розроблення мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA : 1) верифікація, 2) продуктивність і 3) аналіз безпеки. Таким чином, необхідно провести аналіз верифікації та встановити чи результати роботи системи є правильними. Також необхідно проаналізувати рівень продуктивності реалізованих засобів та пропускної здатності апаратних механізмів GCM. Також необхідно здійснити аналіз безпеки та рівень впливу функцій безпеки ПЛІС на реалізацію системи. Верифікація апаратних модулів проводилась за допомогою тестових стендів VHDL та роботи в ПЛІС за допомогою мікропроцесорів м'якого ядра, що використовувались в якості стимулів. Після того, як апаратний бітовий потік було скомпільовано, програмне забезпечення було скомпільовано відповідно до цього апаратного забезпечення. Фактична верифікація була виконана з використанням тестових векторів, наданих у [20].

Процесору були передані дані тестового вектора у вигляді статичних констант. Тестування було розбито на функції C для кожного тестового вектора. Дані конфігурації та закритого ключа спочатку використовувалися для налаштування обох рушіїв GCM, потім дані у відкритому вигляді надсилалися до рушія шифрування GCM. Чорний мікропроцесор отримав ці зашифровані дані від механізму шифрування і передав їх назад до механізму розшифрування. Механізм розшифрування був ініціалізований червоним процесором з тією ж конфігурацією, що і механізм шифрування. Якщо дані, отримані червоним процесором від рушія дешифрування, збігаються з даними, які спочатку були надіслані рушію шифрування, тест вважається успішним. Успішність або неуспішність кожного тесту записується у стандартний вивід.

4.2 Продуктивність апаратного забезпечення

Результати продуктивності для кожної конфігурації системи наведено у таблиці 1. Розмір пакету – це обсяг зашифрованих даних, які обробляються механізмом GCM перед тим, як генерується тег автентифікації. Функціональність була перевірена на належну обробку AAD, але обробка AAD не була включена до тестів продуктивності.

Таблиця 1 – Статистика продуктивності одного GCM

	біти	Архітект. AES S-Box	Архітектура GHASH	Частота GCM Max, MHz	Пропускна здатність 128-bit Pckt, Mbps	Пропускна здатність 2K, Mbps
Мала площа	32 32	C-Field M9K	Bit serial Bit serial	69.3 121	31.8 54.6	59.9 112
Висока продуктивність	128 128	C-Field M9K	Full parallel Full parallel	70.4 75.9	461 501	539 601
Збалансована продуктивність	128 128	C-Field M9K	16-Sequen. 16-Sequen.	66.3 98.1	222 305	398 701

Архітектура та периферія двох процесорів Nios II/ів були узгоджені в кожному тесті. Як і очікувалося, глибокий критичний шлях, знайдений у реалізації композитного поля S-Box, обмежив максимальну частоту необхідного GCM-рушія. Можливою перевагою цього підходу є низькі вимоги до M9K, як показано в таблиці 2. Реалізації GCM, що використовують переваги справжньої двопортової пам'яті на кристалі, є набагато більш ресурсоефективними, враховуючи переважання блоків пам'яті M9K на цільовій ПЛІС. Крім того, пропускна здатність і тактова частота були вищими для реалізацій з використанням S-Box на основі пам'яті. Реалізація на основі пам'яті зі збалансованою продуктивністю мала найвищу пропускну здатність при більшому розмірі пакетів. Це пояснюється вищою частотою і тим, що високопродуктивна реалізація мала еквівалентний компонент шифрування AES, який продовжував виконувати свої обчислення, поки паралельний компонент GHASH простоював. Архітектура апаратного забезпечення має мітки: один і два позначено червоний і чорний розділи підсистеми відповідно. Мітки три і чотири позначають розділи механізму шифрування і дешифрування GCM відповідно. Темні заштриховані блоки – це використані ресурси, за винятком кордонів захищених регіонів. Це огорожі невикористаної логіки, де не дозволено жодних маршрутних з'єднань. Невеликі світло заштриховані області, що не містять логіки між захищеними областями, є SRI, які дозволяють маршрутизацію між областями. Смуги, що проходять вертикально по всій ПЛІС, є елементами пам'яті M9K та вбудованими помножувачами. Важливо відзначити, що нерозподілений простір є місцем, де глобальні сигнали вводяться в глобальну структуру маршрутизації для розподілу по всій ПЛІС.

4. Проєктування мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

У дослідженні проекти спочатку були функціонально перевірені та остаточні проекти були синтезовані для пристрою Cyclone V. Хоча реалізоване апаратне забезпечення є функціонально еквівалентним, для задоволення вимог безпеки на пристрої LS були необхідні додаткові обчислювальні ресурси. Таблиця 2 ілюструє загальну кількість використаних LE і M9K із загальної кількості, виділеної для конкретного захищеного регіону. Разом розмір і розташування захищених регіонів 1 визначають кількість ресурсів, доступних для регіону 2 встановлюють час внутрішньосхемної і зовнішньої передачі даних в регіони і з регіонів 3 обмежують доступні шляхи маршрутизації між захищеними розділами. Кожен захищений регіон також є логічним розділом сам по собі, що запобігає оптимізації, яка могла б знаходитсь на кордонах розмежування [67]. Таким чином, ці захищені області сильно залежать від конструкції та платформи. Для того, щоб було легше робити висновки, розміри захищених областей були фіксованими для конкретного цільового застосування.

Таблиця 2 – Статистика використання ресурсів

Мета	біти	Архітект. AES S-Box	Архітектура GHASH
Мала площа	32	C-Field	порозрядна
	32	M9K	порозрядна
Висока продуктивність	128	C-Field	Повний паралелізм
	128	M9K	Повний паралелізм
Збалансована продуктивність	128	C-Field	16-бітна послідовність
	128	M9K	16-бітна послідовність

Висновки

З метою вирішення поставленої задачі було удосконалено метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

У розділі було розроблено систему з розділенням червоного/чорного з незалежними реалізаціями шифрування та дешифрування GCM з використанням AES для передачі автентифікованої та зашифрованої інформації між двома процесорами Nios II.

Реалізації цієї системи були оцінені на ПЛІС Cyclone V на основі метрик високої продуктивності, низького використання ресурсів та балансу між ними.

Було проведено аналіз використання ресурсів, отримано підтвердження факту підвищення швидкодії AES шифрування щляхом реалізації мультипроцесорної програмно-апаратної архітектури на основі FPGA..

Література

1. Altera Corporation, 101 Innovation Drive, San Jose, CA, Quartus II Design Separation Flow, June 2019. AN 569.
2. L. Hathaway, National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information. On- line, June 2017. CNSS Policy No. 15, Fact Sheet No. 1.
3. F. Rodr´ıguez-Henr´ıquez, N. Saqib, A. D´ıaz-Pe`rez, and C. Koc, Cryptographic Algo- rithms on Reconfigurable Hardware (Signals and Communication Technology). Se- caucus, NJ, USA: Springer-Verlag New York, Inc., 2016.
4. M. Dworkin, Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication, *NIST Special Publication 800-38B, National Institute of Standards and Technology (NIST)*, Gaithersburg, MD 20899-8930, USA, May 2019.
5. M. Dworkin, Recommendation for Block Cipher Modes of Operation: The CCM Mode for Authentication and Confidentiality, *NIST Special Publication 800-38C*, National Institute of Standards and Technology (NIST), Gaithersburg, MD 20899- 8930, USA, May 2018.
6. N. Weaver and J. Wawrzynek, “High Performance, Compact AES Implementations in Xilinx FPGAs,” tech. rep., U.C. Berkely BRASS group, September 2018.
7. A. Menezes, S. Vanstone, and P. V. Oorschot, Handbook of Applied Cryptography. Boca Raton, FL, USA: CRC Press, Inc., 2014.
8. D. Stinson, Cryptography: Theory and Practice. Discrete Mathematics and its Ap- plications, Boca Raton, FL: Chapman & Hall/CRC, third ed., November 2016.
9. National Institute of Standards and Technology (NIST), “Data Encryption Standard (DES).” Federal Information Processing Standards Publication 46-3, 2017.
10. A. Satoh, S. Morioka, K. Takano, and S. Munetoh, “A Compact Rijndael Hard- ware Architecture with S-Box Optimization,” in Advances in Cryptology ASIACRYPT 2015, vol. 2248 of Lecture Notes in Computer Science, pp. 239–254, Springer Berlin / Heidelberg, 2016.
11. J. Daemen and V. Rijmen, “AES Proposal: Rijndael,” March 2016.
12. Larry Ewing (lewing@isc.tamu.edu) using The Gimp http://www.isc.tamu.edu/ lew- ing/gimp/, “tux.jpg.” Online, August 2019.
13. D. McGrew and J. Viega, “The Galois/Counter Mode of Operation (GCM),” May 2015.
14. I. Kuon and J. Rose, “Measuring the Gap Between FPGAs and ASICs,” Computer- Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 26, pp. 203–215, Feb. 2019.
15. Altera Corporation, 101 Innovation Drive, San Jose, CA, Cyclone III Device Hand- book, July 2017. CIII5V1-3.1.
16. M. McLean and J. Moore, FPGA-based single chip cryptographic solution,” Military Embedded Systems, March 2016.
17. S. Roman, Field Theory. No. 158 in Graduate Texts in Mathematics, New York: Springer-Verlag, 2018.
18. C. Paar, Efficient VLSI Architectures for Bit-Parallel Computation in Galois Fields. Dissertation, Institute for Experimental Mathematics, Universitt Essen, Germany, 2020.
19. J. Guajardo, Efficient Algorithms for Elliptic Curve Cryptosystems., master of sci- ence thesis, Worcester Polytechnic Institute, May 2019.
20. B. Kaliski Jr and M. Liskov, Efficient Finite Field Basis Conversion Involving dual bases, in Cryptographic Hardware and Embedded Systems, vol. 1717 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 2016.
21. X. Zhang and K. Parhi, High-Speed VLSI Architectures for the AES Algorithm, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 12, no. 9, pp. 957–967, 2017.
22. K. Ja¨rvinen, M. Tommiska, and J. Skytta¨, A Fully Pipelined Memoryless 17.8 Gbps AES-128 Encryptor, *FPGA ’03: Proceedings of the 2013 ACM/SIGDA eleventh international symposium on Field programmable gate arrays*, (New York, NY, USA), pp. 207–215, ACM, 2018.

ДОДАТОК Б
(обов'язковий)

Презентація

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
Кафедра комп'ютерної інженерії та інформаційних систем

Юрчук Андрій

МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Науковий керівник – д.т.н. проф.
Боровнікова.К.Ю

Хмельницький - 2023

МЕТА І ЗАДАЧІ ДОСЛІДЖЕННЯ

Метою роботи підвищення швидкодії AES шифрування шляхом реалізації мультипроцесорної програмно-апаратної архітектури на основі FPGA.

Об'єкт дослідження – апаратне AES шифрування на основі FPGA.

Предмет дослідження – метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGE.

МЕТА І ЗАДАЧІ ДОСЛІДЖЕННЯ

Поставлена мета досягається розв'язанням таких основних **задач**:

1. дослідити методи синтезу апаратно-програмних засобів реалізації AES шифрування на основі FPGA;
2. проаналізувати сучасні програмно-технічні засоби реалізації AES шифрування на основі FPGA
3. розробити модель Модель програмно-апаратної архітектури для реалізації AES шифрування;
4. розробити метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA;
5. реалізувати метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA

ПЕРЕДБАЧУВАНА НАУКОВА НОВИЗНА ОТРИМАНИХ РЕЗУЛЬТАТІВ

1. Удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA, який на відміну від відомих використовує FPGA, і який забезпечує підвищення швидкодії AES шифрування.
2. Набули подальшого розвитку програмно-технічні засоби AES шифрування шляхом реалізації програмно-апаратної архітектури на основі FPGA.

АКТУАЛЬНІСТЬ ДОСЛІДЖЕННЯ

Актуальність шифрування даних набуває все більшої важливості у світі, де технології стають все більш розповсюдженими та доступними.

Шифрування даних є важливою мірою для захисту конфіденційної інформації, такої як фінансові та медичні записи, особисті дані, бізнес-таємниці, інтелектуальна власність, тощо.

У сучасному світі, де деякі з найбільших компаній збирають та обробляють огромні обсяги особистих даних, забезпечення захисту цих даних стає важливішим, ніж будь-коли раніше.

Важливість шифрування даних ще більше зростає в контексті зростаючої кількості кібератак, які спрямовані на викрадення чутливої інформації та злому систем захисту даних.

МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

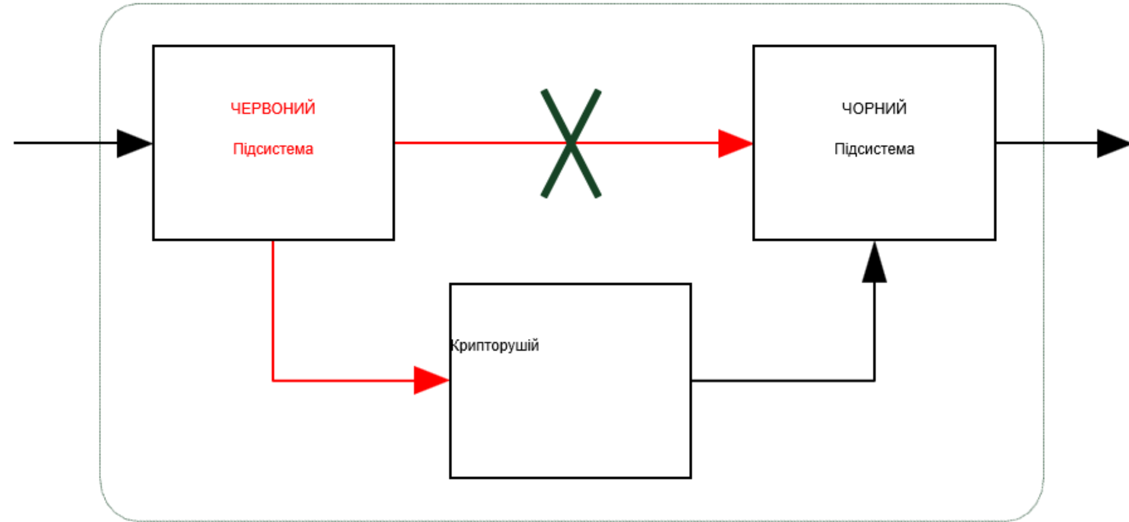
З метою вирішення поставленої задачі було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

Після того, як дизайн компонентів визначено, організація систем спрямована на високу продуктивність, малу площу та баланс між цими двома показниками. Дизайн систем спочатку обговорюється на загальному рівні, спільному для кожної цільової метрики.

Також представлено інтерфейс і програмне забезпечення для верифікації, після чого слідує розділ, що описує роботу, виконану за допомогою інструментів безпеки електронного проектування і автоматизації (EDA), наданих корпорацією Altera.

МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Фізично розділені чорна та червона підсистеми, які обмінюються даними лише за допомогою криптографічного механізму

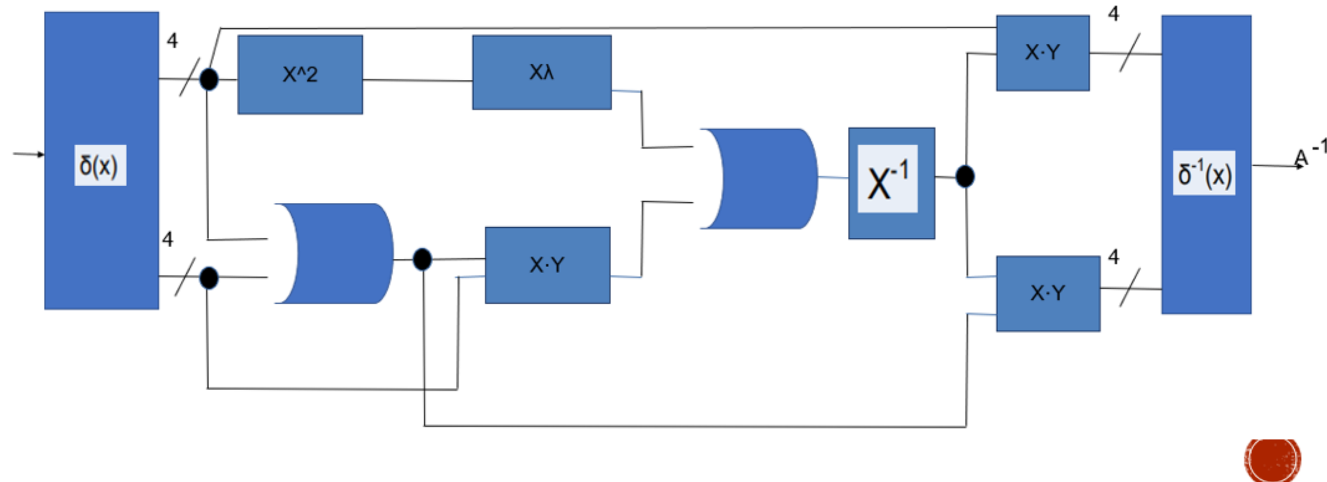


МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Діаграма високого рівня, що показує три основні компоненти в обчисленні композитного поля S-Box

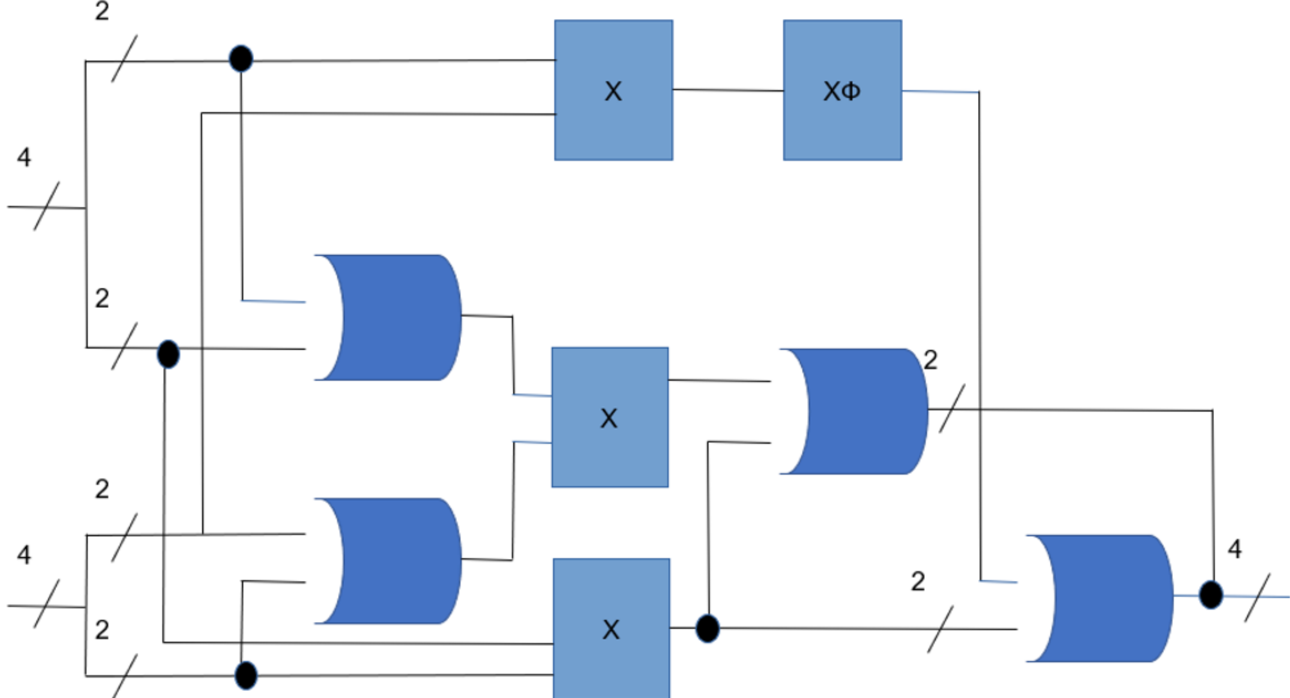
МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Блок-схема, що показує окремі апаратні компоненти в композитній польовій реалізації S-Box

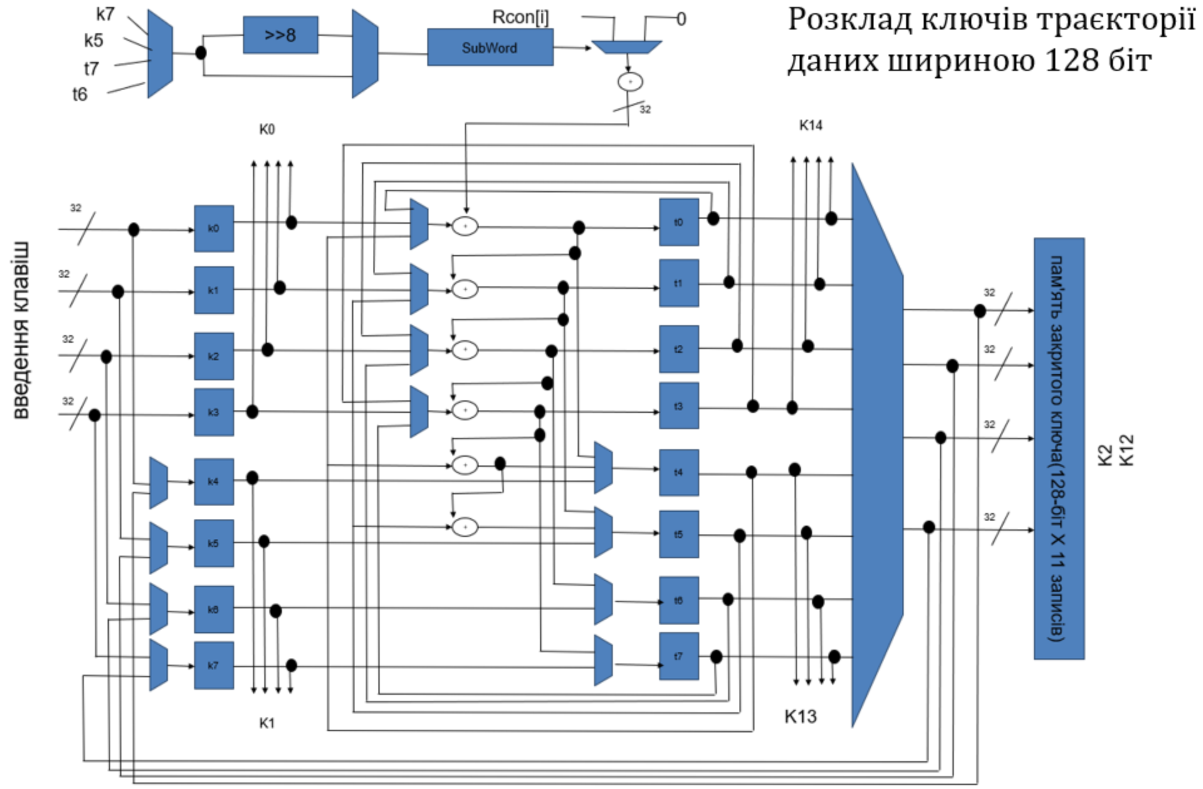


МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Блок-схема, що показує окремі апаратні компоненти

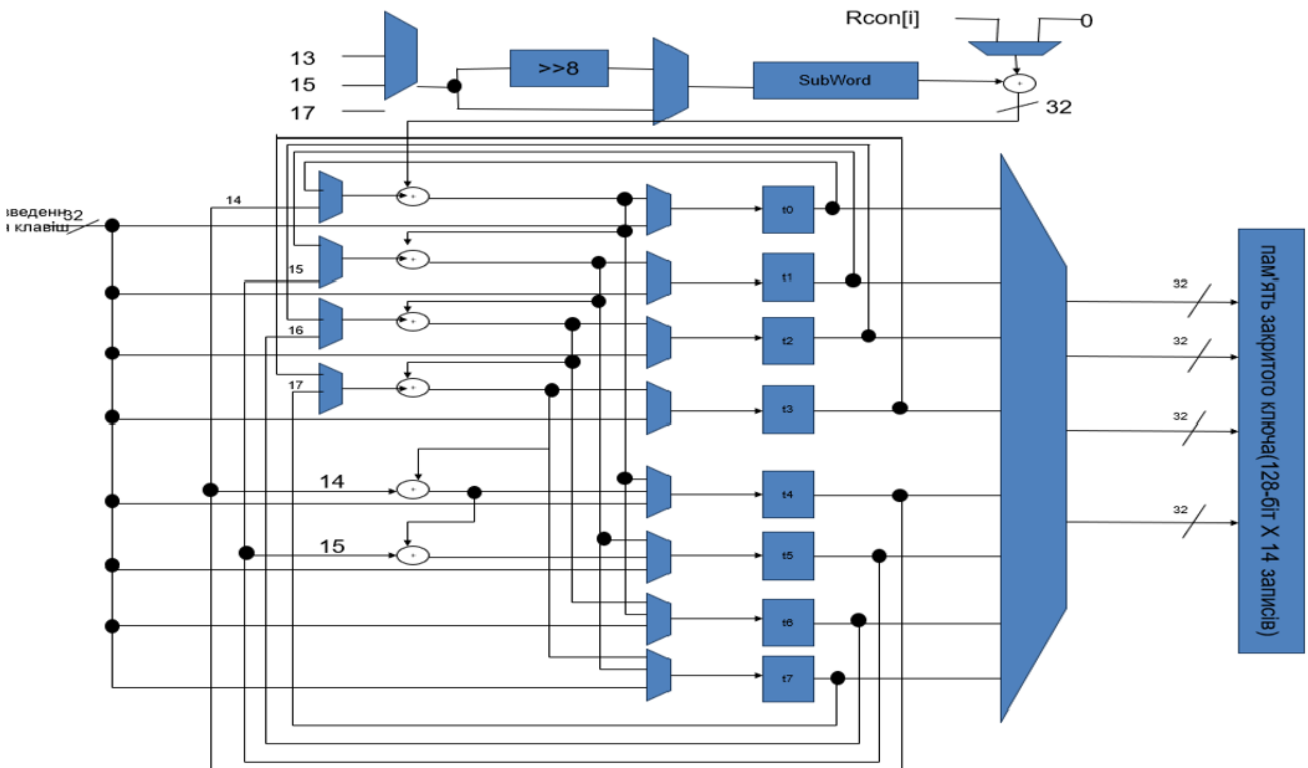


МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA



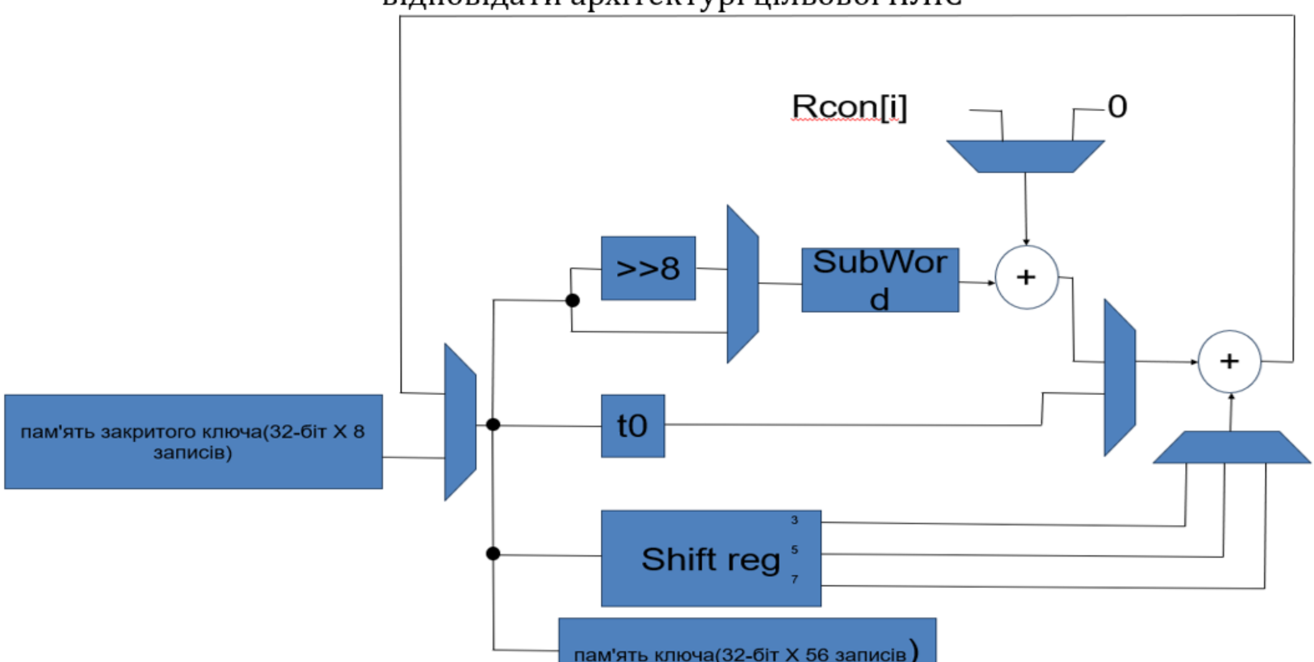
МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОІ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Розклад ключів тракту даних шириною 128 біт , який було модифіковано, щоб краще відповідати архітектурі цільової ПЛІС



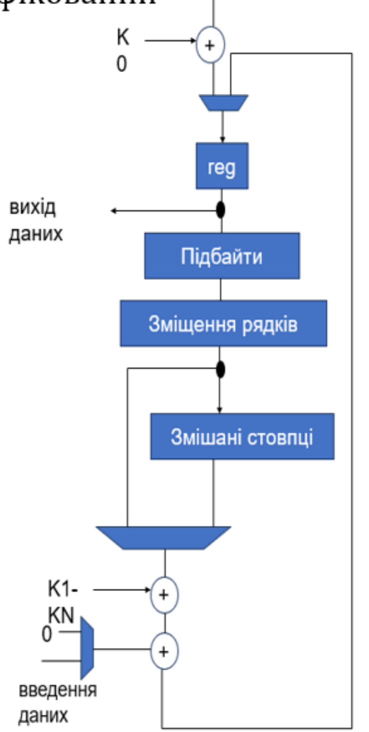
МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Розклад ключів тракту даних шириною 32 біти, який було модифіковано, щоб краще відповідати архітектурі цільової ПЛІС

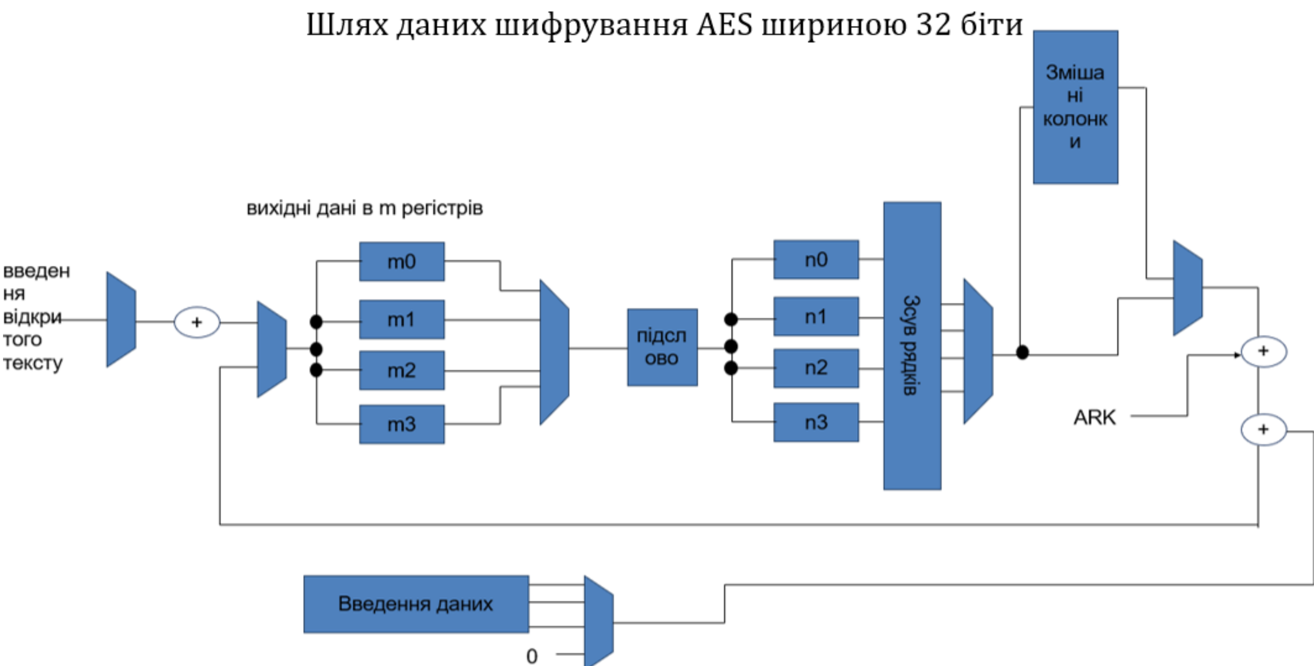


МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Шлях даних шифрування AES шириною 128 біт, модифікований

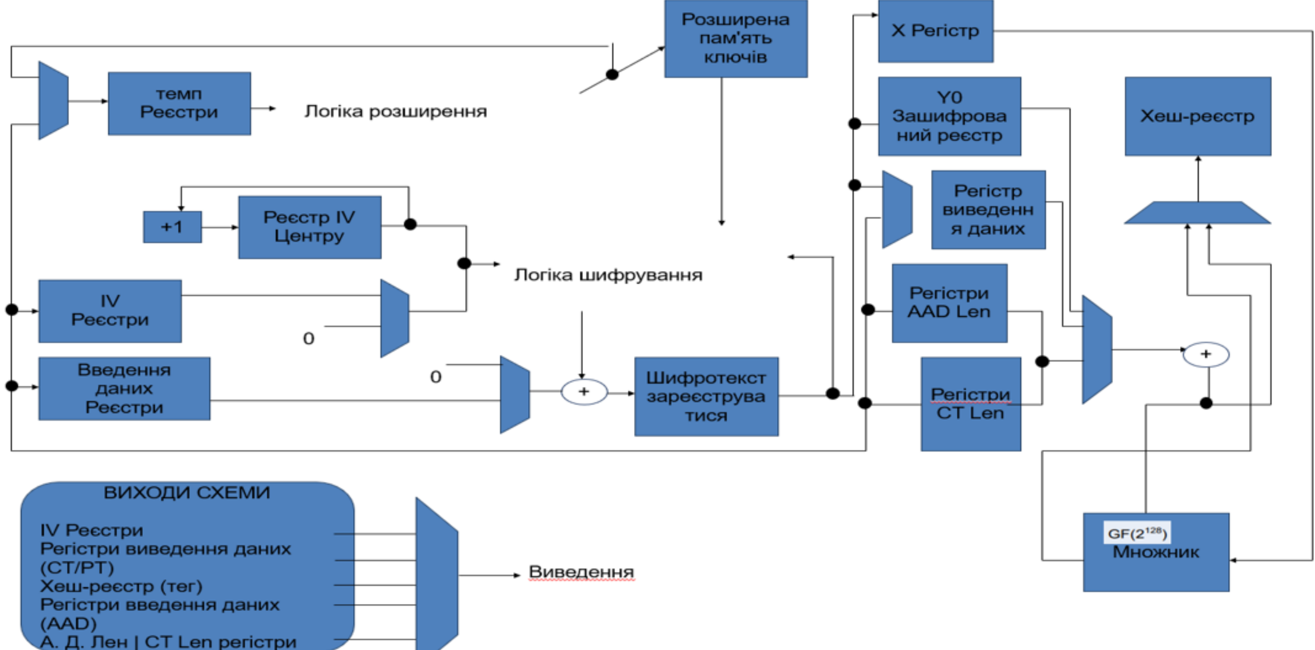


МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA



МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

діаграма високого рівня, що ілюструє окремі компоненти МДГ та їхні взаємозв'язки



МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Статистика продуктивності одного GCM

Мета	біти	Архітект.AES S-Box	Архітектура GHASH	Частота GCM Max, MHz	Пропускна здатність 128-bit Pckt, Mbps	Пропускна здатність 2K, Mbps
Мала площа	3232	C-FieldM9K	Bit serialBit serial	69.3121	31.854.6	59.9112
Висока продуктивність	128128	C-FieldM9K	Full parallelFull parallel	70.475.9	461501	539601
Збалансована продуктивність	128128	C-FieldM9K	16-Sequen.16-Sequen.	66.398.1	222305	398701

ПРОЄКТУВАННЯ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Статистика використання ресурсів .

Мета	біти	Архітект.AES S-Box	Архітектура GHASH
Мала площа	32	C-Field	порозрядна
	32	M9K	порозрядна
Висока продуктивність	128	C-Field	Повний паралелізм
	128	M9K	Повний паралелізм
Збалансована продуктивність	128	C-Field	16-бітна послідовність
	128	M9K	16-бітна послідовність

МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯРЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGA

Плоский графік, що показує потік даних між підсистемами

ВИСНОВКИ

Таким чином, в 1 розділі необхідним є підвищення швидкодії AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA. Тому необхідним є розроблення методу побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

ВИСНОВКИ

- В 2 розділі подано опис математичних передумов побудови програмно-апаратної архітектури для реалізації AES шифрування.
- З цією метою представлено такі поняття, які застосовані до розроблення методу шифрування:
- -скінченні поля;
- -розширення скінченних полів;
- -складені поля;
- -представлення базису;
- -стандартний базис;
- -нормальний базис;
- -подвійний базис;
- -ізоморфізм поля;
- -композитні поля.

ВИСНОВКИ

- У 3 розділі було розроблено систему з розділенням червоного/чорного з незалежними реалізаціями шифрування та дешифрування GCM з використанням AES для передачі автентифікованої та зашифрованої інформації між двома процесорами Nios II.
- З метою вирішення поставленої задачі було удосконалено метод побудови програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.
- Реалізації цієї системи були оцінені на ПЛІС Cyclone V на основі метрик високої продуктивності, низького використання ресурсів та балансу між ними.
- Було проведено аналіз використання ресурсів, отримано підтвердження факту підвищення швидкодії AES шифрування щляхом реалізації програмно-апаратної архітектури на основі FPGA.

ВИСНОВКИ

- У 4 розділі подано процес реалізації запропонованого методу розроблення програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.
- У розділі також подано опис вибору типу архітектури та зразків проектування, описано продуктивність апаратного забезпечення.
- Також в розділі надано опис основних кроків проєктування програмно-апаратної архітектури для реалізації AES шифрування на основі, а також аспекти розроблення системного програмного забезпечення, процес збірки ПЗ та завантаження прошивки та виконання.

ДОДАТОК В
(обов'язковий)

Лістинг програмного забезпечення

```
library ieee;
use ieee.std_logic_1164.all;

entity aes_enc is
	port (
		clk : in std_logic;
		rst : in std_logic;
		key : in std_logic_vector(127 downto 0);
		plaintext : in std_logic_vector(127 downto 0);
		ciphertext : out std_logic_vector(127 downto 0);
		done : out std_logic
	);
end aes_enc;

architecture behavioral of aes_enc is
	signal reg_input : std_logic_vector(127 downto 0);
	signal reg_output : std_logic_vector(127 downto 0);
	signal subbox_input : std_logic_vector(127 downto 0);
	signal subbox_output : std_logic_vector(127 downto 0);
	signal shiftrows_output : std_logic_vector(127 downto 0);
	signal mixcol_output : std_logic_vector(127 downto 0);
	signal feedback : std_logic_vector(127 downto 0);
	signal round_key : std_logic_vector(127 downto 0);
	signal round_const : std_logic_vector(7 downto 0);
	signal sel : std_logic;
begin
	reg_input <= plaintext when rst = '0' else feedback;
	reg_inst : entity work.reg
		generic map(
			size => 128
		)
		port map(
			clk => clk,
			d   => reg_input,
			q   => reg_output
		);
	-- Encryption body
	add_round_key_inst : entity work.add_round_key
		port map(
			input1 => reg_output,
			input2 => round_key,
			output => subbox_input
		);
	sub_byte_inst : entity work.sub_byte
		port map(
			input_data  => subbox_input,
			output_data => subbox_output

		);
	shift_rows_inst : entity work.shift_rows
		port map(
```

```
                input  => subbox_output,
                output => shiftrows_output
            );
        mix_columns_inst : entity work.mix_columns
            port map(
                input_data  => shiftrows_output,
                output_data => mixcol_output
            );
        feedback <= mixcol_output when sel = '0' else shiftrows_output;
        ciphertext <= subbox_input;
        -- Controller
        controller_inst : entity work.controller
            port map(
                clk          => clk,
                rst          => rst,
                rconst        => round_const,
                is_final_round => sel,
                done          => done
            );
        -- Keyschedule
        key_schedule_inst : entity work.key_schedule
            port map(
                clk        => clk,
                rst        => rst,
                key        => key,
                round_const => round_const,
                round_key   => round_key
            );
end architecture behavioral;
```

Footer
© 2023 GitHub, Inc.
Footer navigation

- Terms
- Privacy
- Security
- Status
- Docs
- Contact GitHub
- Pricing
- API
- Training
- Blog
- About

```
library ieee;
use ieee.std_logic_1164.all;

entity add_round_key is
        port (
            input1 : in std_logic_vector(127 downto 0);
            input2 : in std_logic_vector(127 downto 0);
            output : out std_logic_vector(127 downto 0)
        );
end add_round_key;
```

```
architecture rtl of add_round_key is

begin
	output <= input1 xor input2;
end architecture rtl;


library ieee;
use ieee.std_logic_1164.all;

entity controller is
	port (
		clk : in std_logic;
		rst : in std_logic;
		rconst : out std_logic_vector(7 downto 0);
		is_final_round : out std_logic;
		done : out std_logic
	);
end controller;

architecture behavioral of controller is
	signal reg_input : std_logic_vector(7 downto 0);
	signal reg_output : std_logic_vector(7 downto 0);
	signal feedback : std_logic_vector(7 downto 0);
begin
	reg_input <= x"01" when rst = '0' else feedback;
	reg_inst : entity work.reg
		generic map(
			size => 8
		)
		port map(
			clk => clk,
			d   => reg_input,
			q   => reg_output
		);
	--	register_with_reset : process(clk) is
	--	begin
	--		if rising_edge(clk) then
	--			if (rst = '0') then
	--				reg_output <= x"01";
	--			else
	--				reg_output <= feedback;
	--			end if;
	--		end if;
	--   end process register_with_reset;

	gfmult_by2_inst : entity work.gfmult_by2
		port map(
			input_byte  => reg_output,
			output_byte => feedback
		);
	rconst <= reg_output;
	is_final_round <= '1' when reg_output = x"36" else '0';
	done <= '1' when reg_output = x"6c" else '0';
end architecture behavioral;
```

© 2023 GitHub, Inc.
Footer navigation

Terms
Privacy
Securi

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
	generic (size : positive);
	port (
		clk : in std_logic;
		d : in std_logic_vector(size - 1 downto 0);
		q : out std_logic_vector(size - 1 downto 0)
	);
end reg;

architecture behavioral of reg is
	signal current_stata, next_state : std_logic_vector(size - 1 downto 0);
begin
	next_state <= d;
	p1 : process(clk) is
	begin
		if (clk'event and clk = '1') then
			current_stata <= next_state;
		end if;
	end process p1;
	q <= current_stata;
end architecture behavioral;



library ieee;
use ieee.std_logic_1164.all;

entity sbox is
	port (
		input_byte : in std_logic_vector(7 downto 0);
		output_byte : out std_logic_vector(7 downto 0)
	);
end sbox;

architecture behavioral of sbox is
```

```
begin
	lut : process (input_byte) is
	begin
		case input_byte is
			when x"00" => output_byte <= x"63";
			when x"01" => output_byte <= x"7c";
			when x"02" => output_byte <= x"77";
			when x"03" => output_byte <= x"7b";
			when x"04" => output_byte <= x"f2";
			when x"05" => output_byte <= x"6b";
			when x"06" => output_byte <= x"6f";
			when x"07" => output_byte <= x"c5";
			when x"08" => output_byte <= x"30";
			when x"09" => output_byte <= x"01";
			when x"0a" => output_byte <= x"67";
			when x"0b" => output_byte <= x"2b";
			when x"0c" => output_byte <= x"fe";
			when x"0d" => output_byte <= x"d7";
			when x"0e" => output_byte <= x"ab";
			when x"0f" => output_byte <= x"76";
			when x"10" => output_byte <= x"ca";
			when x"11" => output_byte <= x"82";
			when x"12" => output_byte <= x"c9";
			when x"13" => output_byte <= x"7d";
			when x"14" => output_byte <= x"fa";
			when x"15" => output_byte <= x"59";
			when x"16" => output_byte <= x"47";
			when x"17" => output_byte <= x"f0";
			when x"18" => output_byte <= x"ad";
			when x"19" => output_byte <= x"d4";
			when x"1a" => output_byte <= x"a2";
			when x"1b" => output_byte <= x"af";
			when x"1c" => output_byte <= x"9c";
			when x"1d" => output_byte <= x"a4";
			when x"1e" => output_byte <= x"72";
			when x"1f" => output_byte <= x"c0";
			when x"20" => output_byte <= x"b7";
			when x"21" => output_byte <= x"fd";
			when x"22" => output_byte <= x"93";
			when x"23" => output_byte <= x"26";
			when x"24" => output_byte <= x"36";
			when x"25" => output_byte <= x"3f";
			when x"26" => output_byte <= x"f7";
			when x"27" => output_byte <= x"cc";
```

```
when x"28" => output_byte <= x"34";
when x"29" => output_byte <= x"a5";
when x"2a" => output_byte <= x"e5";
when x"2b" => output_byte <= x"f1";
when x"2c" => output_byte <= x"71";
when x"2d" => output_byte <= x"d8";
when x"2e" => output_byte <= x"31";
when x"2f" => output_byte <= x"15";
when x"30" => output_byte <= x"04";
when x"31" => output_byte <= x"c7";
when x"32" => output_byte <= x"23";
when x"33" => output_byte <= x"c3";
when x"34" => output_byte <= x"18";
when x"35" => output_byte <= x"96";
when x"36" => output_byte <= x"05";
when x"37" => output_byte <= x"9a";
when x"38" => output_byte <= x"07";
when x"39" => output_byte <= x"12";
when x"3a" => output_byte <= x"80";
when x"3b" => output_byte <= x"e2";
when x"3c" => output_byte <= x"eb";
when x"3d" => output_byte <= x"27";
when x"3e" => output_byte <= x"b2";
when x"3f" => output_byte <= x"75";
when x"40" => output_byte <= x"09";
when x"41" => output_byte <= x"83";
when x"42" => output_byte <= x"2c";
when x"43" => output_byte <= x"1a";
when x"44" => output_byte <= x"1b";
when x"45" => output_byte <= x"6e";
when x"46" => output_byte <= x"5a";
when x"47" => output_byte <= x"a0";
when x"48" => output_byte <= x"52";
when x"49" => output_byte <= x"3b";
when x"4a" => output_byte <= x"d6";
when x"4b" => output_byte <= x"b3";
when x"4c" => output_byte <= x"29";
when x"4d" => output_byte <= x"e3";
when x"4e" => output_byte <= x"2f";
when x"4f" => output_byte <= x"84";
when x"50" => output_byte <= x"53";
when x"51" => output_byte <= x"d1";
when x"52" => output_byte <= x"00";
when x"53" => output_byte <= x"ed";
when x"54" => output_byte <= x"20";
```

```
when x"55" => output_byte <= x"fc";
when x"56" => output_byte <= x"b1";
when x"57" => output_byte <= x"5b";
when x"58" => output_byte <= x"6a";
when x"59" => output_byte <= x"cb";
when x"5a" => output_byte <= x"be";
when x"5b" => output_byte <= x"39";
when x"5c" => output_byte <= x"4a";
when x"5d" => output_byte <= x"4c";
when x"5e" => output_byte <= x"58";
when x"5f" => output_byte <= x"cf";
when x"60" => output_byte <= x"d0";
when x"61" => output_byte <= x"ef";
when x"62" => output_byte <= x"aa";
when x"63" => output_byte <= x"fb";
when x"64" => output_byte <= x"43";
when x"65" => output_byte <= x"4d";
when x"66" => output_byte <= x"33";
when x"67" => output_byte <= x"85";
when x"68" => output_byte <= x"45";
when x"69" => output_byte <= x"f9";
when x"6a" => output_byte <= x"02";
when x"6b" => output_byte <= x"7f";
when x"6c" => output_byte <= x"50";
when x"6d" => output_byte <= x"3c";
when x"6e" => output_byte <= x"9f";
when x"6f" => output_byte <= x"a8";
when x"70" => output_byte <= x"51";
when x"71" => output_byte <= x"a3";
when x"72" => output_byte <= x"40";
when x"73" => output_byte <= x"8f";
when x"74" => output_byte <= x"92";
when x"75" => output_byte <= x"9d";
when x"76" => output_byte <= x"38";
when x"77" => output_byte <= x"f5";
when x"78" => output_byte <= x"bc";
when x"79" => output_byte <= x"b6";
when x"7a" => output_byte <= x"da";
when x"7b" => output_byte <= x"21";
when x"7c" => output_byte <= x"10";
when x"7d" => output_byte <= x"ff";
when x"7e" => output_byte <= x"f3";
when x"7f" => output_byte <= x"d2";
when x"80" => output_byte <= x"cd";
when x"81" => output_byte <= x"0c";
```

```
when x"82" => output_byte <= x"13";
when x"83" => output_byte <= x"ec";
when x"84" => output_byte <= x"5f";
when x"85" => output_byte <= x"97";
when x"86" => output_byte <= x"44";
when x"87" => output_byte <= x"17";
when x"88" => output_byte <= x"c4";
when x"89" => output_byte <= x"a7";
when x"8a" => output_byte <= x"7e";
when x"8b" => output_byte <= x"3d";
when x"8c" => output_byte <= x"64";
when x"8d" => output_byte <= x"5d";
when x"8e" => output_byte <= x"19";
when x"8f" => output_byte <= x"73";
when x"90" => output_byte <= x"60";
when x"91" => output_byte <= x"81";
when x"92" => output_byte <= x"4f";
when x"93" => output_byte <= x"dc";
when x"94" => output_byte <= x"22";
when x"95" => output_byte <= x"2a";
when x"96" => output_byte <= x"90";
when x"97" => output_byte <= x"88";
when x"98" => output_byte <= x"46";
when x"99" => output_byte <= x"ee";
when x"9a" => output_byte <= x"b8";
when x"9b" => output_byte <= x"14";
when x"9c" => output_byte <= x"de";
when x"9d" => output_byte <= x"5e";
when x"9e" => output_byte <= x"0b";
when x"9f" => output_byte <= x"db";
when x"a0" => output_byte <= x"e0";
when x"a1" => output_byte <= x"32";
when x"a2" => output_byte <= x"3a";
when x"a3" => output_byte <= x"0a";
when x"a4" => output_byte <= x"49";
when x"a5" => output_byte <= x"06";
when x"a6" => output_byte <= x"24";
when x"a7" => output_byte <= x"5c";
when x"a8" => output_byte <= x"c2";
when x"a9" => output_byte <= x"d3";
when x"aa" => output_byte <= x"ac";
when x"ab" => output_byte <= x"62";
when x"ac" => output_byte <= x"91";
when x"ad" => output_byte <= x"95";
when x"ae" => output_byte <= x"e4";
```

```
when x"af" => output_byte <= x"79";
when x"b0" => output_byte <= x"e7";
when x"b1" => output_byte <= x"c8";
when x"b2" => output_byte <= x"37";
when x"b3" => output_byte <= x"6d";
when x"b4" => output_byte <= x"8d";
when x"b5" => output_byte <= x"d5";
when x"b6" => output_byte <= x"4e";
when x"b7" => output_byte <= x"a9";
when x"b8" => output_byte <= x"6c";
when x"b9" => output_byte <= x"56";
when x"ba" => output_byte <= x"f4";
when x"bb" => output_byte <= x"ea";
when x"bc" => output_byte <= x"65";
when x"bd" => output_byte <= x"7a";
when x"be" => output_byte <= x"ae";
when x"bf" => output_byte <= x"08";
when x"c0" => output_byte <= x"ba";
when x"c1" => output_byte <= x"78";
when x"c2" => output_byte <= x"25";
when x"c3" => output_byte <= x"2e";
when x"c4" => output_byte <= x"1c";
when x"c5" => output_byte <= x"a6";
when x"c6" => output_byte <= x"b4";
when x"c7" => output_byte <= x"c6";
when x"c8" => output_byte <= x"e8";
when x"c9" => output_byte <= x"dd";
when x"ca" => output_byte <= x"74";
when x"cb" => output_byte <= x"1f";
when x"cc" => output_byte <= x"4b";
when x"cd" => output_byte <= x"bd";
when x"ce" => output_byte <= x"8b";
when x"cf" => output_byte <= x"8a";
when x"d0" => output_byte <= x"70";
when x"d1" => output_byte <= x"3e";
when x"d2" => output_byte <= x"b5";
when x"d3" => output_byte <= x"66";
when x"d4" => output_byte <= x"48";
when x"d5" => output_byte <= x"03";
when x"d6" => output_byte <= x"f6";
when x"d7" => output_byte <= x"0e";
when x"d8" => output_byte <= x"61";
when x"d9" => output_byte <= x"35";
when x"da" => output_byte <= x"57";
when x"db" => output_byte <= x"b9";
```

```
                when x"dc" => output_byte <= x"86";
                when x"dd" => output_byte <= x"c1";
                when x"de" => output_byte <= x"1d";
                when x"df" => output_byte <= x"9e";
                when x"e0" => output_byte <= x"e1";
                when x"e1" => output_byte <= x"f8";
                when x"e2" => output_byte <= x"98";
                when x"e3" => output_byte <= x"11";
                when x"e4" => output_byte <= x"69";
                when x"e5" => output_byte <= x"d9";
                when x"e6" => output_byte <= x"8e";
                when x"e7" => output_byte <= x"94";
                when x"e8" => output_byte <= x"9b";
                when x"e9" => output_byte <= x"1e";
                when x"ea" => output_byte <= x"87";
                when x"eb" => output_byte <= x"e9";
                when x"ec" => output_byte <= x"ce";
                when x"ed" => output_byte <= x"55";
                when x"ee" => output_byte <= x"28";
                when x"ef" => output_byte <= x"df";
                when x"f0" => output_byte <= x"8c";
                when x"f1" => output_byte <= x"a1";
                when x"f2" => output_byte <= x"89";
                when x"f3" => output_byte <= x"0d";
                when x"f4" => output_byte <= x"bf";
                when x"f5" => output_byte <= x"e6";
                when x"f6" => output_byte <= x"42";
                when x"f7" => output_byte <= x"68";
                when x"f8" => output_byte <= x"41";
                when x"f9" => output_byte <= x"99";
                when x"fa" => output_byte <= x"2d";
                when x"fb" => output_byte <= x"0f";
                when x"fc" => output_byte <= x"b0";
                when x"fd" => output_byte <= x"54";
                when x"fe" => output_byte <= x"bb";
                when x"ff" => output_byte <= x"16";
                when others => null; -- GHDL complains without this statement
            end case;

        end process lut;

end architecture behavioral;
```

```
library ieee;
use ieee.std_logic_1164.all;

entity reg is
	generic (size : positive);
	port (
		clk : in std_logic;
		d : in std_logic_vector(size - 1 downto 0);
		q : out std_logic_vector(size - 1 downto 0)
	);
end reg;

architecture behavioral of reg is
	signal current_stata, next_state : std_logic_vector(size - 1 downto 0);
begin
	next_state <= d;
	p1 : process(clk) is
	begin
		if (clk'event and clk = '1') then
			current_stata <= next_state;
		end if;
	end process p1;
	q <= current_stata;
end architecture behavioral;


library ieee;
use ieee.std_logic_1164.all;

entity aes_dec is
	port (
		clk : in std_logic;
		rst : in std_logic;
		dec_key : in std_logic_vector(127 downto 0);
		ciphertext : in std_logic_vector(127 downto 0);
		plaintext : out std_logic_vector(127 downto 0);
		done : out std_logic
	);
end aes_dec;

architecture rtl of aes_dec is
	signal mux_output : std_logic_vector(127 downto 0);
	signal reg_output : std_logic_vector(127 downto 0);
	signal inv_mixcol_input : std_logic_vector(127 downto 0);
	signal inv_mixcol_output : std_logic_vector(127 downto 0);
```

```
        signal invsr_input : std_logic_vector(127 downto 0);
        signal invsb_input : std_logic_vector(127 downto 0);
        signal feedback : std_logic_vector(127 downto 0);
        signal round_key : std_logic_vector(127 downto 0);
        signal round_const : std_logic_vector(7 downto 0);
        signal is_first_round : std_logic;
begin
        -- Decryption body
        mux_output <= ciphertext when rst = '0' else feedback;
        reg_inst : entity work.reg
                generic map(
                        size => 128
                )
                port map(
                        clk => clk,
                        d   => mux_output,
                        q   => reg_output
                );

        add_round_key_inst : entity work.add_round_key
                port map(
                        input1 => reg_output,
                        input2 => round_key,
                        output => inv_mixcol_input
                );
        inv_mix_columns_inst : entity work.inv_mix_columns
                port map(
                        input_data  => inv_mixcol_input,
                        output_data => inv_mixcol_output
                );
        invsr_input <= inv_mixcol_input when is_first_round = '1' else inv_mixcol_output;
        inv_shift_rows_inst : entity work.inv_shift_rows
                port map(
                        input  => invsr_input,
                        output => invsb_input
                );

        inv_sub_byte_inst : entity work.inv_sub_byte
                port map(
                        input_data  => invsb_input,
                        output_data => feedback
                );
         -- Keysched
ule
         key_schedule_inst : entity work.key_schedule
```

```
		port map(
			clk        => clk,
			rst        => rst,
			key        => dec_key,
			round_const => round_const,
			round_key   => round_key
		);
	-- Controller
	controller_inst : entity work.controller
		port map(
			clk          => clk,
			rst          => rst,
			rconst       => round_const,
			is_first_round => is_first_round,
			done         => done
		);
	plaintext <= inv_mixcol_input;
end architecture rtl;




library ieee;
use ieee.std_logic_1164.all;

entity sbox is
	port (
		input_byte : in std_logic_vector(7 downto 0);
		output_byte : out std_logic_vector(7 downto 0)
	);
end sbox;

architecture behavioral of sbox is

begin
	lut : process (input_byte) is
	begin
		case input_byte is
			when x"00" => output_byte <= x"63";
			when x"01" => output_byte <= x"7c";
			when x"02" => output_byte <= x"77";
			when x"03" => output_byte <= x"7b";
			when x"04" => output_byte <= x"f2";
			when x"05" => output_byte <= x"6b";
			when x"06" => output_byte <= x"6f";
```

```
when x"07" => output_byte <= x"c5";
when x"08" => output_byte <= x"30";
when x"09" => output_byte <= x"01";
when x"0a" => output_byte <= x"67";
when x"0b" => output_byte <= x"2b";
when x"0c" => output_byte <= x"fe";
when x"0d" => output_byte <= x"d7";
when x"0e" => output_byte <= x"ab";
when x"0f" => output_byte <= x"76";
when x"10" => output_byte <= x"ca";
when x"11" => output_byte <= x"82";
when x"12" => output_byte <= x"c9";
when x"13" => output_byte <= x"7d";
when x"14" => output_byte <= x"fa";
when x"15" => output_byte <= x"59";
when x"16" => output_byte <= x"47";
when x"17" => output_byte <= x"f0";
when x"18" => output_byte <= x"ad";
when x"19" => output_byte <= x"d4";
when x"1a" => output_byte <= x"a2";
when x"1b" => output_byte <= x"af";
when x"1c" => output_byte <= x"9c";
when x"1d" => output_byte <= x"a4";
when x"1e" => output_byte <= x"72";
when x"1f" => output_byte <= x"c0";
when x"20" => output_byte <= x"b7";
when x"21" => output_byte <= x"fd";
when x"22" => output_byte <= x"93";
when x"23" => output_byte <= x"26";
when x"24" => output_byte <= x"36";
when x"25" => output_byte <= x"3f";
when x"26" => output_byte <= x"f7";
when x"27" => output_byte <= x"cc";
when x"28" => output_byte <= x"34";
when x"29" => output_byte <= x"a5";
when x"2a" => output_byte <= x"e5";
when x"2b" => output_byte <= x"f1";
when x"2c" => output_byte <= x"71";
when x"2d" => output_byte <= x"d8";
when x"2e" => output_byte <= x"31";
when x"2f" => output_byte <= x"15";
when x"30" => output_byte <= x"04";
when x"31" => output_byte <= x"c7";
when x"32" => output_byte <= x"23";
when x"33" => output_byte <= x"c3";
```

```
when x"34" => output_byte <= x"18";
when x"35" => output_byte <= x"96";
when x"36" => output_byte <= x"05";
when x"37" => output_byte <= x"9a";
when x"38" => output_byte <= x"07";
when x"39" => output_byte <= x"12";
when x"3a" => output_byte <= x"80";
when x"3b" => output_byte <= x"e2";
when x"3c" => output_byte <= x"eb";
when x"3d" => output_byte <= x"27";
when x"3e" => output_byte <= x"b2";
when x"3f" => output_byte <= x"75";
when x"40" => output_byte <= x"09";
when x"41" => output_byte <= x"83";
when x"42" => output_byte <= x"2c";
when x"43" => output_byte <= x"1a";
when x"44" => output_byte <= x"1b";
when x"45" => output_byte <= x"6e";
when x"46" => output_byte <= x"5a";
when x"47" => output_byte <= x"a0";
when x"48" => output_byte <= x"52";
when x"49" => output_byte <= x"3b";
when x"4a" => output_byte <= x"d6";
when x"4b" => output_byte <= x"b3";
when x"4c" => output_byte <= x"29";
when x"4d" => output_byte <= x"e3";
when x"4e" => output_byte <= x"2f";
when x"4f" => output_byte <= x"84";
when x"50" => output_byte <= x"53";
when x"51" => output_byte <= x"d1";
when x"52" => output_byte <= x"00";
when x"53" => output_byte <= x"ed";
when x"54" => output_byte <= x"20";
when x"55" => output_byte <= x"fc";
when x"56" => output_byte <= x"b1";
when x"57" => output_byte <= x"5b";
when x"58" => output_byte <= x"6a";
when x"59" => output_byte <= x"cb";
when x"5a" => output_byte <= x"be";
when x"5b" => output_byte <= x"39";
when x"5c" => output_byte <= x"4a";
when x"5d" => output_byte <= x"4c";
when x"5e" => output_byte <= x"58";
when x"5f" => output_byte <= x"cf";
when x"60" => output_byte <= x"d0";
```

```
when x"61" => output_byte <= x"ef";
when x"62" => output_byte <= x"aa";
when x"63" => output_byte <= x"fb";
when x"64" => output_byte <= x"43";
when x"65" => output_byte <= x"4d";
when x"66" => output_byte <= x"33";
when x"67" => output_byte <= x"85";
when x"68" => output_byte <= x"45";
when x"69" => output_byte <= x"f9";
when x"6a" => output_byte <= x"02";
when x"6b" => output_byte <= x"7f";
when x"6c" => output_byte <= x"50";
when x"6d" => output_byte <= x"3c";
when x"6e" => output_byte <= x"9f";
when x"6f" => output_byte <= x"a8";
when x"70" => output_byte <= x"51";
when x"71" => output_byte <= x"a3";
when x"72" => output_byte <= x"40";
when x"73" => output_byte <= x"8f";
when x"74" => output_byte <= x"92";
when x"75" => output_byte <= x"9d";
when x"76" => output_byte <= x"38";
when x"77" => output_byte <= x"f5";
when x"78" => output_byte <= x"bc";
when x"79" => output_byte <= x"b6";
when x"7a" => output_byte <= x"da";
when x"7b" => output_byte <= x"21";
when x"7c" => output_byte <= x"10";
when x"7d" => output_byte <= x"ff";
when x"7e" => output_byte <= x"f3";
when x"7f" => output_byte <= x"d2";
when x"80" => output_byte <= x"cd";
when x"81" => output_byte <= x"0c";
when x"82" => output_byte <= x"13";
when x"83" => output_byte <= x"ec";
when x"84" => output_byte <= x"5f";
when x"85" => output_byte <= x"97";
when x"86" => output_byte <= x"44";
when x"87" => output_byte <= x"17";
when x"88" => output_byte <= x"c4";
when x"89" => output_byte <= x"a7";
when x"8a" => output_byte <= x"7e";
when x"8b" => output_byte <= x"3d";
when x"8c" => output_byte <= x"64";
when x"8d" => output_byte <= x"5d";
```

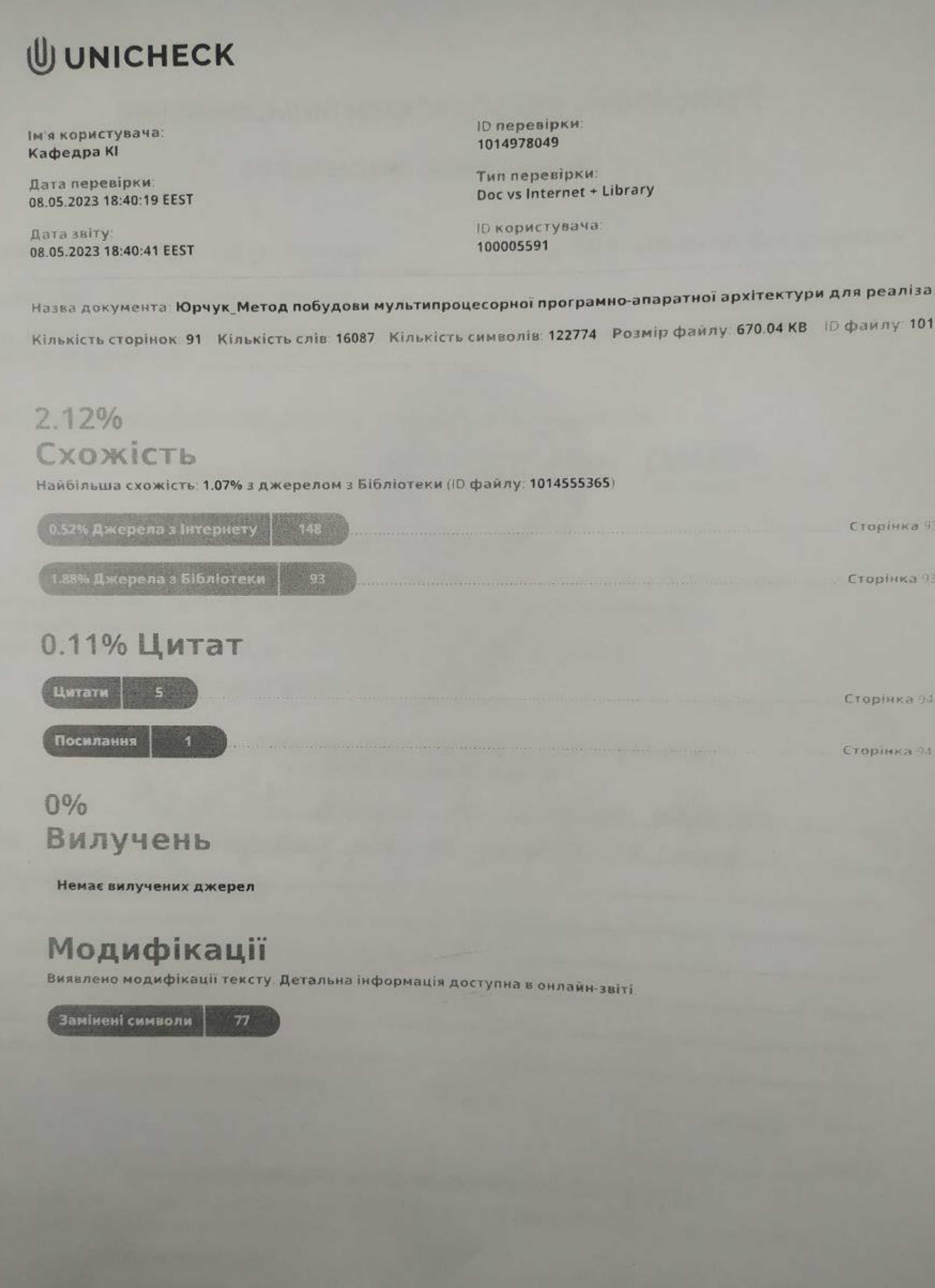
```
when x"8e" => output_byte <= x"19";
when x"8f" => output_byte <= x"73";
when x"90" => output_byte <= x"60";
when x"91" => output_byte <= x"81";
when x"92" => output_byte <= x"4f";
when x"93" => output_byte <= x"dc";
when x"94" => output_byte <= x"22";
when x"95" => output_byte <= x"2a";
when x"96" => output_byte <= x"90";
when x"97" => output_byte <= x"88";
when x"98" => output_byte <= x"46";
when x"99" => output_byte <= x"ee";
when x"9a" => output_byte <= x"b8";
when x"9b" => output_byte <= x"14";
when x"9c" => output_byte <= x"de";
when x"9d" => output_byte <= x"5e";
when x"9e" => output_byte <= x"0b";
when x"9f" => output_byte <= x"db";
when x"a0" => output_byte <= x"e0";
when x"a1" => output_byte <= x"32";
when x"a2" => output_byte <= x"3a";
when x"a3" => output_byte <= x"0a";
when x"a4" => output_byte <= x"49";
when x"a5" => output_byte <= x"06";
when x"a6" => output_byte <= x"24";
when x"a7" => output_byte <= x"5c";
when x"a8" => output_byte <= x"c2";
when x"a9" => output_byte <= x"d3";
when x"aa" => output_byte <= x"ac";
when x"ab" => output_byte <= x"62";
when x"ac" => output_byte <= x"91";
when x"ad" => output_byte <= x"95";
when x"ae" => output_byte <= x"e4";
when x"af" => output_byte <= x"79";
when x"b0" => output_byte <= x"e7";
when x"b1" => output_byte <= x"c8";
when x"b2" => output_byte <= x"37";
when x"b3" => output_byte <= x"6d";
when x"b4" => output_byte <= x"8d";
when x"b5" => output_byte <= x"d5";
when x"b6" => output_byte <= x"4e";
when x"b7" => output_byte <= x"a9";
when x"b8" => output_byte <= x"6c";
when x"b9" => output_byte <= x"56";
when x"ba" => output_byte <= x"f4";
```

```
when x"bb" => output_byte <= x"ea";
when x"bc" => output_byte <= x"65";
when x"bd" => output_byte <= x"7a";
when x"be" => output_byte <= x"ae";
when x"bf" => output_byte <= x"08";
when x"c0" => output_byte <= x"ba";
when x"c1" => output_byte <= x"78";
when x"c2" => output_byte <= x"25";
when x"c3" => output_byte <= x"2e";
when x"c4" => output_byte <= x"1c";
when x"c5" => output_byte <= x"a6";
when x"c6" => output_byte <= x"b4";
when x"c7" => output_byte <= x"c6";
when x"c8" => output_byte <= x"e8";
when x"c9" => output_byte <= x"dd";
when x"ca" => output_byte <= x"74";
when x"cb" => output_byte <= x"1f";
when x"cc" => output_byte <= x"4b";
when x"cd" => output_byte <= x"bd";
when x"ce" => output_byte <= x"8b";
when x"cf" => output_byte <= x"8a";
when x"d0" => output_byte <= x"70";
when x"d1" => output_byte <= x"3e";
when x"d2" => output_byte <= x"b5";
when x"d3" => output_byte <= x"66";
when x"d4" => output_byte <= x"48";
when x"d5" => output_byte <= x"03";
when x"d6" => output_byte <= x"f6";
when x"d7" => output_byte <= x"0e";
when x"d8" => output_byte <= x"61";
when x"d9" => output_byte <= x"35";
when x"da" => output_byte <= x"57";
when x"db" => output_byte <= x"b9";
when x"dc" => output_byte <= x"86";
when x"dd" => output_byte <= x"c1";
when x"de" => output_byte <= x"1d";
when x"df" => output_byte <= x"9e";
when x"e0" => output_byte <= x"e1";
when x"e1" => output_byte <= x"f8";
when x"e2" => output_byte <= x"98";
when x"e3" => output_byte <= x"11";
when x"e4" => output_byte <= x"69";
when x"e5" => output_byte <= x"d9";
when x"e6" => output_byte <= x"8e";
when x"e7" => output_byte <= x"94";
```

```
                when x"e8" => output_byte <= x"9b";
                when x"e9" => output_byte <= x"1e";
                when x"ea" => output_byte <= x"87";
                when x"eb" => output_byte <= x"e9";
                when x"ec" => output_byte <= x"ce";
                when x"ed" => output_byte <= x"55";
                when x"ee" => output_byte <= x"28";
                when x"ef" => output_byte <= x"df";
                when x"f0" => output_byte <= x"8c";
                when x"f1" => output_byte <= x"a1";
                when x"f2" => output_byte <= x"89";
                when x"f3" => output_byte <= x"0d";
                when x"f4" => output_byte <= x"bf";
                when x"f5" => output_byte <= x"e6";
                when x"f6" => output_byte <= x"42";
                when x"f7" => output_byte <= x"68";
                when x"f8" => output_byte <= x"41";
                when x"f9" => output_byte <= x"99";
                when x"fa" => output_byte <= x"2d";
                when x"fb" => output_byte <= x"0f";
                when x"fc" => output_byte <= x"b0";
                when x"fd" => output_byte <= x"54";
                when x"fe" => output_byte <= x"bb";
                when x"ff" => output_byte <= x"16";
                when others => null; -- GHDL complains without this statement
            end case;

        end process lut;

end architecture behavioral;
```


UNICHECK
Ім'я користувача:
Кафедра КІ
Дата перевірки:
08.05.2023 18:40:19 EEST
Дата звіту:
08.05.2023 18:40:41 EEST
ID перевірки:
1014978049
Тип перевірки:
Doc vs Internet + Library
ID користувача:
100005591
Назва документа: Юрчук_Метод побудови мультипроцесорної програмно-апаратної архітектури для реаліза...
Кількість сторінок: 91 Кількість слів: 16087 Кількість символів: 122774 Розмір файлу: 670.04 KB ID файлу: 1014
2.12%
Схожість
Найбільша схожість: 1.07% з джерелом з Бібліотеки (ID файлу: 1014555365)
0.52% Джерела з Інтернету 148
Сторінка 93
1.88% Джерела з Бібліотеки 93
Сторінка 93
0.11% Цитат
Цитати 5
Сторінка 94
Посилання 1
Сторінка 94
0%
Вилучень
Немає вилучених джерел
Модифікації
Виявлено модифікації тексту. Детальна інформація доступна в онлайн-звіті
Замінені символи 77

11.05.23, 10:56 Юрчук.html

Mon May 08 17:45:14 EEST 2023, Медзатий Дмитро Миколайович, Хмельницький національний університет, Х

Anti-Plagiarism v-15.257

Максимальне співпадіння з одним документом 0.0%

Словники перевірки: en_US, ru_RU, ua_UA. **Помилок в документах: 12%**

ID: 113089 Назва: МКР Метод побудови мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA Додано в БД: 2023-05-08 Автора: Юрчук А.О. Керівники: Бобровнікова К.Ю. Консультанти: Опоненти:	Документ		Сумарний збіг по Базі Даних	
	Символи	Лексеми	Символи	Лексеми
	99208	868	492 (0%)	8 (1%)

Джерело плагіату

ID	Опис	Наявність плагіату в документі	
		Символи	Лексеми

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ХМЕЛЬНИЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ

РЕЦЕНЗІЯ НА КВАЛІФІКАЦІЙНУ РОБОТУ

Здобувач: Юрчук Андрій Олексійович

Тема: МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGE

Спеціальність: 123 «Комп'ютерна інженерія»

Обсяг кваліфікаційної роботи:

Кількість листів креслень – ; кількість сторінок записки 76

1. Короткий зміст роботи та прийнятих рішень У роботі проведенні дослідження AES шифрування на основі FPGA та метод побудови прогроамно-апаратної архітектури

2. Висновок про відповідність роботи дипломному завданню Кваліфікаційна робота магістра відповідає виданому завданню, як у теоретичній , так і в проктичній її частині

3. Характеристика виконання кожного розділу, ступінь використання останніх досягнень науки і техніки і передових методів роботи: У першому розділі проведенно огляд відомих методів побудови архітектур для реалізації AES . У другому розділі математичны предумови побудови. У третьому розділі запропоновано метод побудови архітектур програмно-апаратної для AES шифрування на основі FPGA. У четвертому розділі запропуновано розроблення програмно-апаратної архітектури для AES шифрування на основі FPGE.

4. Позитивні сторони роботи: В результаті виконаного наукового дослідження буде розромбленно програмно апаратні засоби AES

5. Негативні сторони роботи: В роботі присутні певні логічні помилки щодо опису моделей програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

5. Негативні сторони роботи: В роботі присутні помилки щодо опису мультипроцесорної програмно-апаратної архітектури для реалізації AES шифрування на основі FPGA.

6. Оцінка графічного оформлення та пояснювальної записки роботи: –

7. Відгук про роботу в цілому: В загальному робота виконана на невисокому рівні.

8. Інші зауваження: –

9. Оцінка кваліфікаційної роботи:

Розглянувши позитивні та негативні сторони представленої кваліфікаційної роботи вважаю, що робота заслуговує оцінки «задовільно» 3.00 (Е)

Рецензент (прізвище, ім'я, по батькові, посада, місце роботи) д.т.н.. професор, Мартинюк В.В., завідувач кафедри

"10 травня" 2023р.

Завідувачу кафедри КІІС
д-р.техн.наук, проф. Говорущенко Т. О.

Юрчук Андрій Олексійович
ПІБ здобувача вищої освіти

ФІТ, 2 курсу, групи КІ2м-21-1

ЗАЯВА

З правилами чинного Положення «Про дотримання академічної доброчесності в Хмельницькому національному університеті» від 01.07.2022, згідно з яким виявлення плагіату є підставою для відмови в допуску кваліфікаційної роботи до захисту та застосування заходів дисциплінарної та академічної відповідальності, ознайомлений (а). Про використання програмно-технічних засобів для перевірки кваліфікаційних робіт здобувачів вищої освіти на плагіат оповіщений (а) та надаю свою згоду на обробку та збереження університетом моєї роботи в інституційному репозитарії університету.

Також надаю університету право на передачу моєї роботи для обробки та збереження в базах даних програмно-технічних засобів (Unicheck та Anti-Plagiarism) та використання роботи для виявлення плагіату в інших роботах, які перевіряються програмно-технічними засобами та користувачами, що мають доступ до цих програмно-технічних засобів, виключно в обмежених цілях для виявлення плагіату в текстах робіт.

Робота для перевірки університетом надається в друкованому та електронному варіанті. Електронна версія моєї роботи збігається (ідентична) з друкованою.

22 квітня 2023
дата

підпис

РІШЕННЯ ЕКСПЕРНОЇ КОМІСІЇ

КАФЕДРИ КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ ТА ІНФОМАЦІЙНИХ СИСТЕМ

ПРО ДОПУСК КВАЛІФІКАЦІЙНОЇ РОБОТИ ДО ЗАХИСТУ

Підтверджуємо ознайомлення з результатом звіту подібності щодо роботи, генерованого системою виявлення текстових збігів/ідентичності/схожості:

Назва: МЕТОД ПОБУДОВИ ПРОГРАМНО-АПАРАТНОЇ АРХІТЕКТУРИ ДЛЯ РЕАЛІЗАЦІЇ AES ШИФРУВАННЯ НА ОСНОВІ FPGE

Автор: Юрчук Андрій Олексійович

Спеціальність: 123 – Компютерна інженерія

Освітня програма: освітньо-наукова

Науковий керівник: Бобровнікова Кіра Юрівна

Після аналізу звіту подібності зроблено такий висновок:

№	Висновок	Позначка про відповідність
1	Запозичення, виявлені в роботі, є законними і не є плагіатом. Робота приймається до захисту.	відповідає
2	Виявлені запозичення не є плагіатом, розміщені в розділах, які не описують безпосередньо авторське дослідження, але кількість цитат перевищує обсяг, виправданий поставленою метою роботи Робота приймається до захисту, але має бути відкоригована. Відкоригований варіант має бути поданий на кафедру за 2 дні до захисту, разом із заявою щодо самостійності виконання письмової роботи та ідентичності друкованої та електронної версії роботи	
3	Виявлені запозичення не є плагіатом, але частково розміщені в розділах, які описують безпосередньо авторське дослідження, а кількість цитат перевищує обсяг, виправданий поставленою метою роботи. В зв'язку з цим мета роботи та поставлені завдання не були досягнені. Робота може бути допущена до захисту (наступного року) після того як буде відкоригована та допрацьована і успішно пройде повторну перевірку на академічний плагіат.	
4	Робота містить навмисні текстові спотворення, передбачувані спроби укриття запозичень або інші прояви академічного плагіату. Робота містить фабрикацію або фальсифікацію даних. Робота не допускається до захисту.	

Підтвердження:

Запозичення, виявлені в роботі, є законними і не є плагіатом, оскільки:

1) Усі фрагментні, або мають належним чином оформленні посилання;

2) Окремі миявленні ззбіги єзагальновживаними фразами або виразами, про що свідчать посилання системи на збіг з джерелами на один фрагмент речення ;

3) Всі зафіксовання системою ознаки модифікації тексту відносяться до комбінування латинських символів з украънськими скороченнями індексів в формулах , що не є модифікацією тексту.

Сумарний обсяг всіх запозичень, визначений системою виявлення збігів/ ідентичності/схожості Unicheck, складає 2.12% і адресується до 148 першоджерела; та системою Anti-Plagiarism складає 24%, що, з урахуванням наведених обгрунтувань, відповідає характеру наукового дослідження і свідчить на користь кваліфікаційної роботи.

Керівник роботи ____________ К.Ю.Бобровнікова

Гарант ОП ____________ О. С. Савенко

Завідувач кафедри КІІС ____________ Т. О. Говорущенко