

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

МОСКОВСКИЙ ИНСТИТУТ ЭЛЕКТРОНИКИ И МАТЕМАТИКИ
им. А.Н. ТИХОНОВА

Двойнишников Николай Евгеньевич, группа БИВ165

РАЗРАБОТКА КРИПТОГРАФИЧЕСКОЙ СИСТЕМЫ НА ПЛИС

Выпускная квалификационная работа по направлению подготовки
09.03.01. Информатика и вычислительная техника
студента образовательной программы
«Информатика и вычислительная техника»

Студент

/ Двойнишников Н.Е.

подпись

И.О. Фамилия

Руководитель

к.т.н., доцент ДКИ МИЭМ НИУ ВШЭ
Романов А.Ю.

должность, звание, И.О. Фамилия

Консультант

должность, звание, И.О. Фамилия

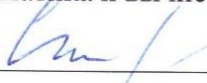
Рецензент

должность, звание, И.О. Фамилия

Москва, 2020

«УТВЕРЖДАЮ»

Академический руководитель
образовательной программы
«Информатика и вычислительная техника»


Ю.И. Гудков
« 20 » 12 20 19 г.

**ЗАДАНИЕ
на выпускную квалификационную работу бакалавра**

студенту группы БИВ-165 Двойнишникову Николаю Евгеньевичу

1. Тема работы

Разработка криптографической системы на ПЛИС.
Development of a FPGA cryptographic system.

2. Требования к работе

2.1. Необходимо реализовать модель криптографической системы с использованием языка описания аппаратуры (HDL). Разработанная HDL модель криптографической системы должна обрабатывать постоянный входной поток данных по заданному алгоритму шифрования.

2.2. Выполненная работа должна быть задокументирована в соответствии с ГОСТ. Документация должна состоять из описания работы и полученных результатов. В работе должна присутствовать исследовательская и практическая части. Практическая часть работы должна быть воспроизводима в соответствии с описанием и документацией, все разработанные коды и датасеты – задокументированы и помещены в репозиторий GitHub. Рекомендуются в работе опираться на данные из открытых источников, пользоваться доступным ПО и аппаратным обеспечением УЛ САПР, ДКИ МИЭМ.

2.3. Для выполнения аппаратной реализации в ПЛИС использовать язык описания аппаратуры Verilog.

3. Содержание работы

3.1. Обзор существующих алгоритмов легковесного шифрования. Обоснование выбора алгоритма шифрования Hummingbird для реализации на ПЛИС.

3.2. Описание математической модели алгоритма шифрования Hummingbird.

3.3. Разработка HDL модели криптографической системы.



- 3.4. Разработка тестового окружения для RTL-симуляции и верификации криптографической системы.
- 3.5. Тестирование работы реализованной модели и анализ полученных результатов.
- 3.6. Составление документации по ВКР.

4. Сроки выполнения этапов работы

Проект ВКР представляется студентом в срок до:	«10» февраля 2020 г.
Первый вариант ВКР представляется студентом в срок до:	«26» апреля 2020 г.
Итоговый вариант ВКР представляется студентом руководителю до загрузки работы в систему «Антиплагиат» в срок до:	«20» мая 2020 г.

Задание выдано «12» декабря 2019 г.

Задание принято к исполнению «12» декабря 2019 г.

 подпись руководителя	А.Ю. Романов И.О. Фамилия
 подпись студента	Н.Е. Двоишников И.О. Фамилия

АННОТАЦИЯ

Алгоритмы легковесного шифрования являются новым направлением в развитии алгоритмов шифрования с закрытым ключом. Такая потребность возникла в результате появления большого количества устройств с небольшой вычислительной мощностью. Поэтому возникла необходимость в алгоритмах, способных обеспечить достаточный уровень безопасности, при минимальном использовании ресурсов. В работе представлена реализация алгоритма легковесного шифрования Hummingbird, разобран алгоритм шифрования и проведено моделирование на способность шифровать и дешифровать данные корректно. Проведен анализ полученных результатов и сравнение их с имеющимися аналогами.

Работа состоит из 30 страниц, 17 источников, 9 рисунков и 4 таблиц.

ABSTRACT

Lightweight encryption algorithms are a new development in the realm of private key encryption algorithms. The demand for it arose with the propagation of devices with relatively low computational power. Thus, new algorithms were needed that could ensure an adequate level of security while using minimal resources. This work presents a practical application of the Hummingbird lightweight encryption solution, describes its precise algorithm, and models its capacity to cipher and decipher data correctly. The resulting data is analyzed and compared to the existing alternatives as well.

The work contains 30 pages, 17 sources, 9 pictures, and 4 tables.

ОГЛАВЛЕНИЕ

1 Обзор и анализ предметной области	8
1.1 Обзор литературы	8
1.2 Аналитические выводы.....	11
2 Цели и задачи.....	12
2.1 Описание методов решения	12
3 Разработка элементов системы шифрования.....	14
3.1 Структура процесса шифрования и дешифрования	14
3.2 Таблицы замены S-boxes	16
3.3 Блок шифрования.....	17
3.4 Процесс инициализации системы	18
3.5 Обновление регистров внутреннего состояния.....	19
3.6 Управляющее устройство.....	20
3.7 Сборка модели и управление	21
3.8 Принцип взаимодействия шифратора и дешифратора.....	21
4 Тестирование модели.....	24
4.1 Моделирование с использованием встроенных утилит среды Quartus.....	24
4.2 Прототипирование на ПЛИС	25
5 Анализ полученных результатов	27

ВВЕДЕНИЕ

В мире наблюдается тенденция на увеличение количества устройств для Интернета вещей [1] и возникает потребность в удешевлении каждого такого устройства. Не маловажным является также энергопотребление. Многие устройства работают от съемных аккумуляторов и батареек, что требует облегчение алгоритма шифрования в плане энергозатрат. С другой стороны, существенно увеличивается объем передаваемых данных, который необходимо оградить от 3-их лиц. Ставшие уже классическими алгоритмы шифрования развивались по пути увеличения стойкости шифрования, и в стремлении улучшить данный показатель, происходило планомерное увеличение требуемых ресурсов для их реализации. В конечном счете это приводит к удорожанию конечного устройства. Такие алгоритмы хоть и являются образцами надежности, но слишком дороги в реализации.

Решением на появление новой проблемы стало новое направление в криптографии - Lightweight cryptography. Основная задача данного направления, это нахождение баланса между ресурсоемкостью и безопасностью.

Для сравнения: наиболее скоростная реализация алгоритма AES демонстрирует скорость до 70 Гбит/сек [4]. Такая реализация использует конвейерную архитектуру процессора и требует более 250,000 GE. В то же время наиболее компактная реализация этого алгоритма требует порядка 2,400 GE [5]. Разница в используемых элементах отличается примерно в 100 раз.

В данной работе будет реализован на языке Verilog и протестирован на ПЛИС один из алгоритмов легковесного шифрования – Hummingbird. Проведен его анализ и сравнение с другими аналогами.

1 Обзор и анализ предметной области

1.1 Обзор литературы

В статье [12] описываются предпосылки появления и требования предъявляемые к алгоритмам легковесного шифрования. В последние годы приложения и устройства Интернета вещей стремительно развиваются благодаря новым технологиям, таким как RFID-метки и беспроводные сенсорные сети (WSN) [2]. RFID позволяет маркировать каждое отдельное устройство, служащее одним из механизмов в IoT. Благодаря WSN, каждый "объект", т.е. люди, устройства и т.д. становятся объектом беспроводной сети и могут общаться между собой.

У таких больших сетей есть необходимость в защите передаваемой информации, и ряд других проблем, таких как конфиденциальность, кроссплатформенность, аутентификация, легкость исполнения (мало затратное), стандартизация и т.п.

Легкость исполнения: уникальная особенность безопасности данных в IoT [7]. Появилась из-за ограниченности устройств в вычислительных мощностях и электрической емкости. В IoT используется множество разных устройств, многие из них проектируются максимально простыми и не рассчитанными на серьезные и энергозатратные вычисления [17]. Данная проблема послужила возникновению такого направления криптографии, как легковесная криптография.

При выборе или разработке алгоритма легковесного шифрования, всегда приходится находить баланс между тремя главными параметрами [14]: надежностью, производительностью и ценой (рис. 1).

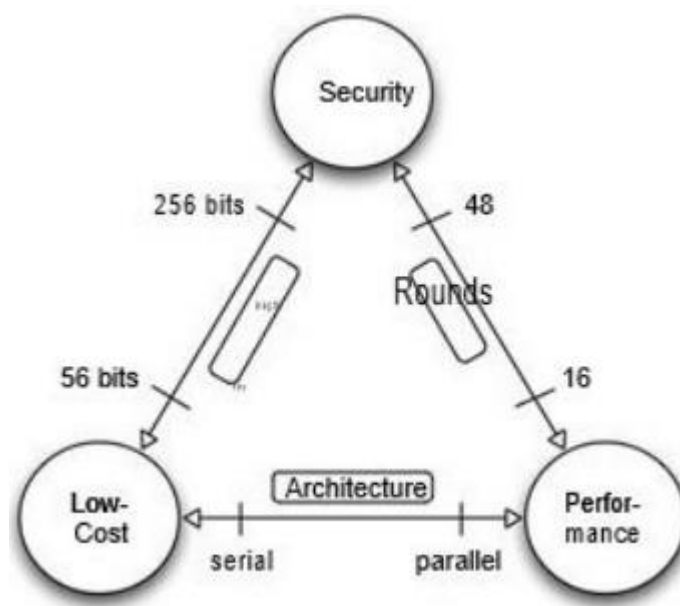


Рисунок 1 – Схематическое взаимодействия надежности, производительности и цены

Данная необходимость возникает из-за главных требований к шифрованию: оно должно обеспечивать безопасность передаваемых данных (надежность). Классические алгоритмы шифрования, такие как DES, AES стремятся к максимальной безопасности передаваемых данных. В специфике создания алгоритмов присутствуют противоречия: чем меньше мы можем затратить ресурсов для реализации алгоритма, тем выше вероятность, что такой алгоритм будет не способен защитить данные от атак, и в следствие будет бесполезен.

Таким образом можно сформулировать главную задачу легковесной криптографии: обеспечить достаточную степень защиты информации, затратив при этом минимально возможное количество используемых ресурсов и энергопотребления.

Уже сформированы некоторые подходы к решению данной проблемы:

- уменьшение (до разумных пределов) размеров основных параметров алгоритма: блока шифруемых данных, ключа шифрования и внутреннего состояния алгоритма.
- компенсация вынужденной потери стойкости алгоритмов за счет проектирования на основе хорошо изученных и широко применяемых операций, осуществляющих элементарные линейные/нелинейные преобразования.

- уменьшение размеров данных, используемых в конкретных операциях. Например, таблицы замены: таблице, заменяющей 8-битовые фрагменты данных, необходимо 256 байт, но такую таблицу можно составить из комбинации двух 4-битовых таблиц, требующих всего 32 байта в сумме.
- использование «дешевых» с точки зрения ресурсоемкости, но эффективных преобразований, таких как управляемые битовые перестановки, сдвиговые регистры и пр.
- применение преобразований, в отношении которых возможны варианты реализации в зависимости от ресурсов конкретного шифратора (например, уменьшение требований к памяти, но в ущерб скорости шифрования, или наоборот).

В настоящее время уже существует множество алгоритмов легковесного шифрования. Каждый из них имеет свои особенности, как правило алгоритм выбирается под конкретные условия. Поэтому нет универсального решения, ведь иногда можно ослабить стойкость и выбрать более компактный алгоритм. В других случаях к стойкости предъявляются большие требования, поэтому самый компактный алгоритм может уже не подойти.

Многие требования, предъявляемые к алгоритмам, предназначенным к использованию в низкоресурсных устройствах, были закреплены в рамках международного стандарта ISO/IEC FDIS 29192 – Information technology – Security techniques – Lightweight cryptography [6].

Так же стоит заметить, что в целом алгоритмы делятся на две большие группы: для аппаратной реализации и для программной. Данное различие обусловлено как структурой самого алгоритма, так и набором операций, преобразований и используемой памятью.

Для выбора подходящего алгоритма был проведен сравнительный анализ нескольких вариантов: алгоритм KATAN, алгоритм PRESENT [20] и алгоритм Hummingbird

1.2 Аналитические выводы

Из анализа литературы следует, что разработка криптографической системы с использованием алгоритмов легковесного шифрования является актуальной, так как данная область знаний появилась относительно недавно и в настоящее время активно развивается. Главными параметрами анализа полученных результатов будет являться стойкость шифрования к внешним угрозам и общая компактность системы. Реализация системы на ПЛИС позволит получить аппаратную реализацию для использования в устройствах с небольшой вычислительной мощностью.

Для разработки модели выбран алгоритм легковесного шифрования Hummingbird так как он спроектирован для аппаратной реализации, использует простые и дешевые в реализации операции хог, циклический сдвиг, сложение и вычитание по модулю и табличную замену. Отличительной особенностью данного алгоритма от рассмотренных аналогов является то, что обработка данных происходит потоком, это добавляет данному алгоритму стойкость. Не мало важным является то, что алгоритм достаточно подробно описан и использует хорошо изученные операции в криптоанализе.

2 Цели и задачи

Целью данной работы является создание криптографической системы на языке Verilog, в основе которой будет лежать алгоритм легковесного шифрования Hummingbird. Основное описание алгоритма взято из статьи [9] Система должна корректно производить шифрование и дешифрование данных. Должен быть описан алгоритм взаимодействия с устройством. Полученные результаты должны быть проверены на количество используемых логических элементов, быстродействие и произведено сравнение с аналогами. Так же необходимо провести прототипирование на ПЛИС, в нашем случае на De1-SoC.

2.1 Описание методов решения

Требуется выполнить следующие этапы работы над проектом:

1. Разработка элементов, используемых в алгоритме Hummingbird (модули шифрования и дешифрования, модуль управления внутренними состояниями, элементы управления);
2. Тестирование всех компонентов на соответствие ожидаемым выходным сигналам;
3. Сбор всех компонентов в полную модель и ее отладка.
4. Сбор статистики работы системы с различными параметрами ключей и шифруемых данных. Анализ результатов.

Модель разработана на языке Verilog. Ее симуляция выполняется в среде Modelsim. Для анализа полученных результатов будут использоваться встроенные утилиты среды разработки Quartus. В первую очередь будет реализован алгоритм шифрования и дешифрования без обновления внутренних состояний, так как это является основой разработки. Далее будет добавлено управляющее устройство с возможностью потокового шифрования входных данных. Взаимодействие между управляющим устройством и модулями шифрования происходит при помощи изменения регистров состояний и управляющих сигналов. Процесс обработки данных происходит по следующему алгоритму: происходит инициализация системы, которая занимает 5 тактов. Далее устанавливается состояние на ожидание шифрования, и с приходом управляющего сигнала, сообщаящего нашей системе, что готовы данные для шифрования, последовательно происходят процессы

шифрования и обновления данных. Как только управляющий сигнал устанавливается в 0, происходит остановка шифрования и ожидание начала процесса дешифрования. Результаты работы системы возвращаются обратно пользователю в общей шине в моменты их получения системой.

Для корректного шифрования модели необходимо указать ключ шифрования и начальное состояние системы.

Схематическое представление криптографической системы представлено на рисунке 2.

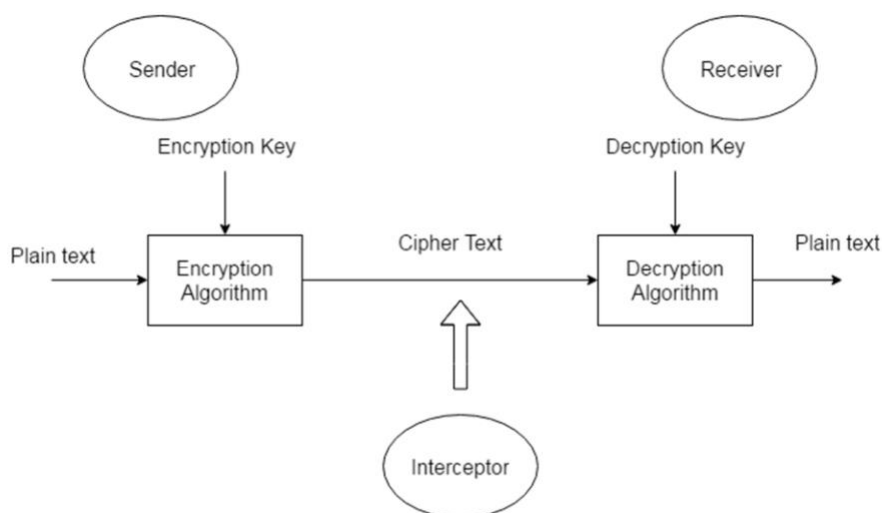


Рисунок 2 – Схематическое представление криптографической системы

3 Разработка элементов системы шифрования

3.1 Структура процесса шифрования и дешифрования

Для шифрования данных будет использоваться алгоритм легковесного шифрования Hummingbird. Данный алгоритм обрабатывает поток данных блоками по 16 бит, использует ключ величиной в 256 бит и имея 4 регистра внутреннего состояния RS по 16 бит. Размер ключа позволяет добиться адекватного уровня защиты информации. Для понимания диаграмм, описывающих структуру алгоритма, необходимо ввести список используемых элементов. Данный список приведен в таблице 1.

Таблица 1

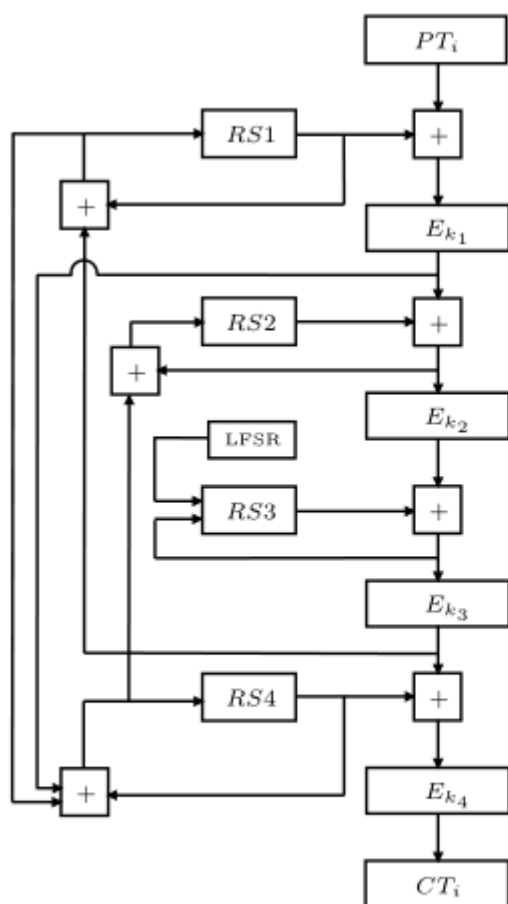
Обозначения, используемые в описании алгоритма [9]

PT_i	the i -th plaintext block, $i = 1, 2, \dots, n$
CT_i	the i -th ciphertext block, $i = 1, 2, \dots, n$
K	the 256-bit secret key
$E_K(\cdot)$	the encryption function of Hummingbird with 256-bit secret key K
$D_K(\cdot)$	the decryption function of Hummingbird with 256-bit secret key K
k_i	the 64-bit subkey used in the i -th block cipher, $i = 1, 2, 3, 4$, such that $K = k_1 \ k_2 \ k_3 \ k_4$
$E_{k_i}(\cdot)$	a block cipher encryption algorithm with 16-bit input, 64-bit key k_i , and 16-bit output, i.e., $E_{k_i} : \{0, 1\}^{16} \times \{0, 1\}^{64} \rightarrow \{0, 1\}^{16}, i = 1, 2, 3, 4$
$D_{k_i}(\cdot)$	a block cipher decryption algorithm with 16-bit input, 64-bit key k_i , and 16-bit output, i.e., $D_{k_i} : \{0, 1\}^{16} \times \{0, 1\}^{64} \rightarrow \{0, 1\}^{16}, i = 1, 2, 3, 4$
RS_i	the i -th 16-bit internal state register, $i = 1, 2, 3, 4$
LFSR	a 16-stage Linear Feedback Shift Register with the characteristic polynomial $f(x) = x^{16} + x^{15} + x^{12} + x^{10} + x^7 + x^3 + 1$
\boxplus	modulo 2^{16} addition operator
\boxminus	modulo 2^{16} subtraction operator
\oplus	exclusive-or (XOR) operator
$m \ll l$	left circular shift operator, which rotates all bits of m to the left by l bits, as if the left and the right ends of m were joined.
$K_j^{(i)}$	the j -th 16-bit key used in the i -th block cipher, $j = 1, 2, 3, 4$, such that $k_i = K_1^{(i)} \ K_2^{(i)} \ K_3^{(i)} \ K_4^{(i)}$
S_i	the i -th 4-bit to 4-bit S-box used in the block cipher, $S_i : \mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4, i = 1, 2, 3, 4$
NONCE $_i$	the i -th nonce which is a 16-bit random number, $i = 1, 2, 3, 4$

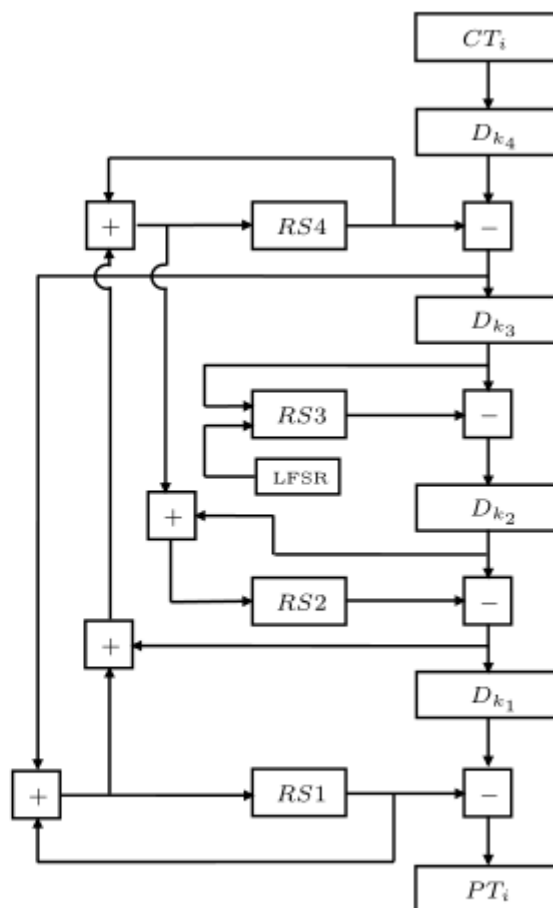
Структура алгоритма шифрования, представляет собой комбинацию 4 блоков шифрования $E_{k_1}, E_{k_2}, E_{k_3}$ и E_{k_4} которые производят шифрование блоков информации по 16 бит. В структуре так же имеются 4 регистра внутренних состояний RS1, RS2, RS3 и RS4, которые складываются по модулю с результатами шифрования и обновляются после каждого пройденного блока. Секретный ключ K делится на 4

малых ключа k_1 , k_2 , k_3 и k_4 по 64 бит, каждый из которых используется в соответствующем блоке шифрования E_k . 16 битный блок исходного текста PT поданный на вход алгоритма сначала складывается по модулю 16 с содержимым первого регистра состояния $RS1$. Полученное значение поступает на вход следующего блока шифрования. Данная процедура аналогично повторяется 4 раза и результат шифрования блока E_{k_4} подается на выход устройства. Полученный результат является блоком шифртекста CT . После этого произойдет обновление регистров внутреннего состояния по определенному алгоритму, с использованием результатов промежуточного шифрования и значения LFSR.

Процесс дешифрования происходит симметрично процессу шифрования, в обратном направлении. Структура процессов приведена на рисунке 3.



(a) Encryption Process



(b) Decryption Process

Рисунок 3 - Структура процесса шифрования(а) и дешифрования(б)

3.2 Таблицы замены S-boxes

Для дальнейшего описания и разработки алгоритма шифрования, необходимо разработать таблицы замены. Для уменьшения используемой памяти необходимо составить 4 таблицы для замены 4 битных блоков. Для обратного преобразования будут использоваться соответствующие таблицы. Используемые значения S-box приведены в таблице 2.

Таблица 2

Значения S-box используемые в модели

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_1(x)$	8	6	5	F	1	C	A	9	E	B	2	4	7	0	D	3
$S_2(x)$	0	7	E	1	5	B	8	2	3	A	D	6	F	C	4	9
$S_3(x)$	2	E	F	5	C	1	9	A	B	4	6	8	0	7	3	D
$S_4(x)$	0	7	3	4	C	1	A	F	D	E	6	B	2	8	9	5

Для еще большего уменьшения используемых логических элементов алгоритмом Hummingbird в аппаратной реализации можно заменить четыре используемых S-box на один, который будет использовать циклично четыре раза подряд. Данное изменение увеличит время обработки каждого блока и снизит надежность шифрования, но позволит сделать реализацию более компактной.

Аппаратная реализация данного элемента представляет собой комбинацию мультиплексоров.

3.3 Блок шифрования

Используемые 4 блока шифрования E_k имеют идентичную структуру. В структуре блока шифрования используются 3 основных операции: наложение ключа операцией xor, табличная замена и линейное преобразование. Первая операция представляет собой 4 такта наложения части ключа, после этого происходит табличная замена и далее линейное преобразование. К каждому модулю шифрования подведена шина с 64 битным ключом. Внутри модуля ключ разделяется на 4 малых ключа по 16 бит. Именно с данными ключами происходит наложение исходных данных. При табличной замене блок данных разделяется на 4 шины по 4 бита и происходит табличная замена с помощью модуля S-box. Полученное значение соединяется и над ним производится операция линейного преобразования $L(m)$, которая представляет собой циклический сдвиг и наложение. Структура данного преобразования:

$$L(m) = m \text{ xor } (m \ll 6) \text{ xor } (m \ll 10)$$

где $m = (m_0, m_1, \dots, m_{15})$ это 16 битный блок данных.

После выполнения цикла преобразований происходит дополнительный этап наложения ключа и табличной замены. Структура блока шифрования представлена на рисунке 4.

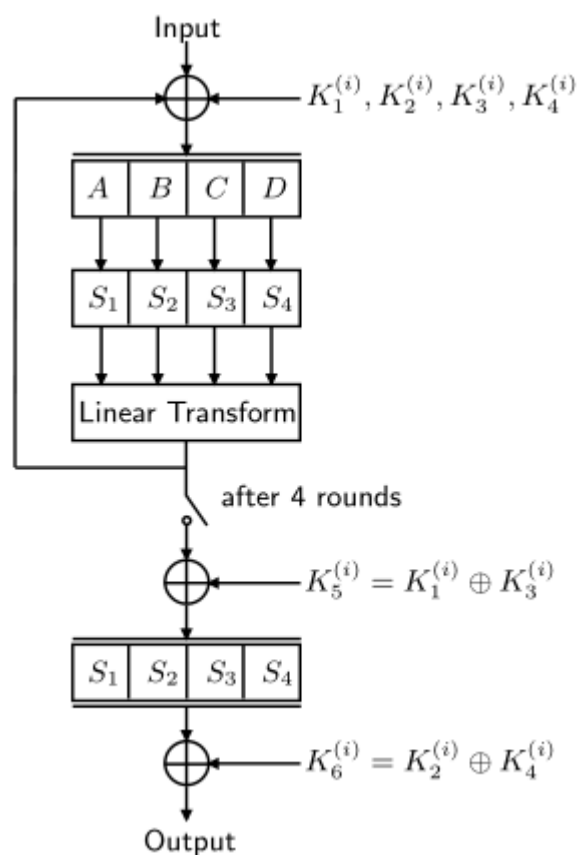


Рисунок 4 – блок-диаграмма процесса шифрования

Данный процесс шифрования можно представить в виде последовательности преобразований, в таком случае данный процесс будет представлять собой комбинационную схему и может выполняться асинхронно. Данное изменение позволит увеличить скорость обработки каждого отдельного блока информации, но реализация такого сценария будет менее компактной.

3.4 Процесс инициализации системы

Для начала работы системы необходимо произвести инициализацию регистров внутреннего состояния. Для алгоритма Hummingbird предусмотрен специальный процесс, после прохождения которого система может начинать шифрование данных. На первом такте инициализации системы в регистры заносятся значения NONCE. Данные числа представляют из себя случайные четыре числа. На вход алгоритма шифрования подается значение равное $RS1 + RS3$. Далее проходит четыре такта обновления значений регистров. После завершения обновлений, в регистр LFSR записывается значение выражения:

$$LFSR = TV3 \mid 0x1000$$

По окончании 5 такта система установлена и готова к работе. Схема инициализации приведена на рисунке 5.

Nonce Initialization:

$$RS1_0 = \text{NONCE}_0$$

$$RS2_0 = \text{NONCE}_1$$

$$RS3_0 = \text{NONCE}_2$$

$$RS4_0 = \text{NONCE}_3$$

Four Rounds Encryption:

for $t = 0$ **to** 3 **do**

$$V12_t = E_{k_1} ((RS1_t \oplus RS3_t) \oplus RS1_t)$$

$$V23_t = E_{k_2} (V12_t \oplus RS2_t)$$

$$V34_t = E_{k_3} (V23_t \oplus RS3_t)$$

$$TV_t = E_{k_4} (V34_t \oplus RS4_t)$$

$$RS1_{t+1} = RS1_t \oplus TV_t$$

$$RS2_{t+1} = RS2_t \oplus V12_t$$

$$RS3_{t+1} = RS3_t \oplus V23_t$$

$$RS4_{t+1} = RS4_t \oplus V34_t$$

end for

LFSR Initialization:

$$\text{LFSR} = TV_3 \mid 0 \times 1000$$

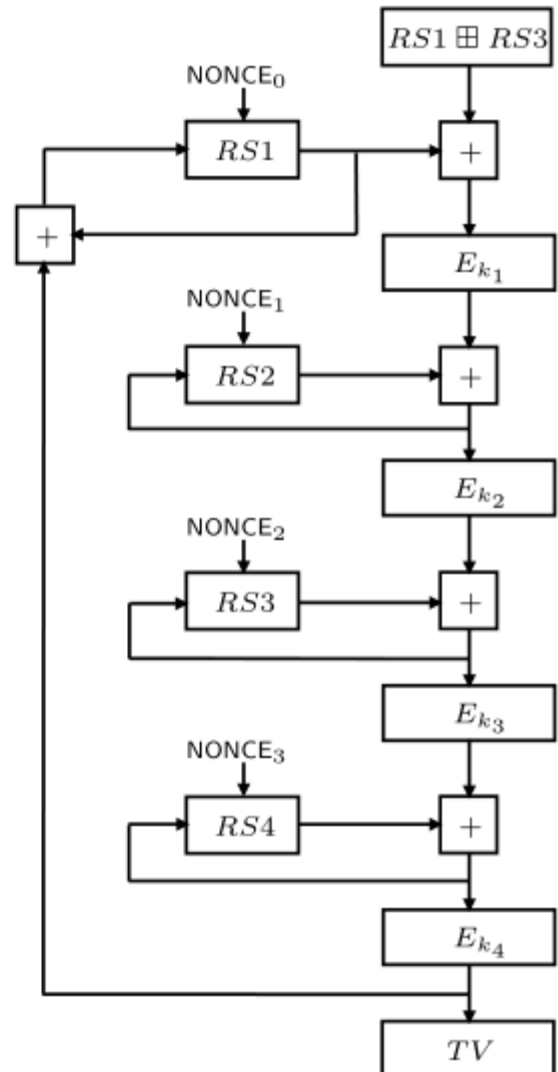


Рисунок 5 – блок-диаграмма процесса инициализации

3.5 Обновление регистров внутреннего состояния

После прохождения инициализации начинается процесс шифрования. Алгоритм Hummingbird производит обновление регистров состояния на каждом такте шифрования и дешифрования. Каждый регистр имеет собственное уравнение, зависящее от текущего состояния регистра и данных, полученных на этапе шифрования. Более подробно уравнения преобразования можно посмотреть в таблице 3. Стоит отметить, что процесс обновления регистров состояния для шифрования и дешифрования не отличаются.

Шифрование/дешифрование и процедура обновления регистров состояния.

Процесс шифрования	Процесс дешифрования
$V12_t = E_{k_1}(PT_i \boxplus RS1_t)$	$V34_t = D_{k_4}(CT_i) \boxplus RS4_t$
$V23_t = E_{k_2}(V12_t \boxplus RS2_t)$	$V23_t = D_{k_3}(V12_t) \boxplus RS3_t$
$V34_t = E_{k_3}(V23_t \boxplus RS3_t)$	$V12_t = D_{k_2}(V23_t) \boxplus RS2_t$
$CT_i = E_{k_4}(V34_t \boxplus RS4_t)$	$PT_i = D_{k_1}(V34_t) \boxplus RS1_t$
Обновление регистров состояния	
$LFSR_{t+1} \leftarrow LFSR_t$ $RS1_{t+1} = RS1_t \boxplus V34_t$ $RS3_{t+1} = RS3_t \boxplus V23_t \boxplus LFSR_{t+1}$ $RS4_{t+1} = RS4_t \boxplus V12_t \boxplus RS1_{t+1}$ $RS2_{t+1} = RS2_t \boxplus V12_t \boxplus RS4_{t+1}$	

3.6 Управляющее устройство

Для управления фазами алгоритма необходимо продумать модель управляющего устройства. Структура представлена в виде автомата Мили [16]. В начальное состояние S0 автомат приходит по сигналу reset. В этом состоянии начинается процесс инициализации регистров внутреннего состояния. На вход шифратора подается комбинация RS1+RS3, а сами регистры устанавливаются в состояния NONCE. На следующем такте происходит переход в следующее состояние S1. В данном случае происходит 4 раунда обновления регистров состояния, описанные в пункте 3.4. По их окончании автомат переходит в следующее состояние S2, в котором устанавливает значение регистра LFSR и ожидает начало шифрования. За начало процесса шифрования отвечает сигнал data_rdy. По положительному фронту тактового сигнала и значению data_rdy = 1, автомат переходит в следующее состояние S3. В состоянии S4 происходит процесс шифрования, на вход шифратора подается значение с входной шины input_data. Каждый такт происходит шифрование 1 блока данных, то есть по 16 бит. Так же каждый такт происходит обновление регистров состояния. Условием окончания шифрования и переходом в следующее состояние - режим ожидания, является приход сигнала data_rdy = 0. В состоянии S4 автомат ожидает продолжения

процесса шифрования по сигналу `data_rdy = 1`, если необходимо продолжить процесс и перейти в состояние S3, или сигнала `reset`, который начнет подготовку к шифрованию нового потока данных перейдя в состояние S0.

3.7 Сборка модели и управление

Необходимо объединить разработанные нами блоки в единую модель. Для этого первым делом разделим наше устройство на две части. Первая часть шифратор, который будет представлять из себя комбинационную схему. Данная часть модуля работает без тактового сигнала. Вторая часть — это управляющее устройство, которое будет работать по тактовому сигналу, и по положительному фронту которого будет происходить проверка условий на переход в другую фазу и обновление регистров состояний.

Для управления моделью предназначены следующие шины данных: тактовый сигнал `clk`, сигнал сброса `reset`, шина входных данных `data_in`, сигнал готовности данных для шифрования `data_rdy`.

Первым делом, по линии `reset` приходит сигнал 0, который сбрасывает модель в состояния начала инициализации. После этого происходит инициализация, которая продолжается 5 тактов, и устанавливается режим ожидания. Далее подаются данные на вход устройства и по сигналу `data_rdy` начинается процесс шифрования. Продолжается он до момента получения сигнала `data_rdy = 0`.

Перед началом работы устройства необходимо задать значения ключа шифрования и значений NONCE, чтобы корректно провести процесс инициализации и шифрования. Для этого выведены специальные одноименные шины, по которым необходимо подать нужные значения.

3.8 Принцип взаимодействия шифратора и дешифратора

Стоит отметить, что в реализации данного алгоритма алгоритм шифрования должен быть расположен на стороне отправителя, а алгоритм дешифрования на стороне считывателя. В нашем случае два данных алгоритма будут располагаться внутри одного устройства. По данной причине мы сразу будем подавать значения, полученные от шифратора и подавать на вход дешифратора. Обновление регистров

состояния будет происходить так же в один такт и будет использоваться общий ключ.

Но стоит понимать, что для реализации данного алгоритма на разных устройствах необходимо выполнить дополнительные операции аутентификации. Это необходимо для получения информации об метке или считывающем устройстве, на основании которых можно определить, является ли данное устройство искомым. Данная операция позволяет не отправлять данные посторонним пользователям и экономит время. Так же является одним из признаков того, что полученные данные могут быть расшифрованы корректно.

Для алгоритма Hummingbird есть специальный протокол, в котором описан алгоритм аутентификации. Процесс проходит в 3 этапа. На каждом из этапов устройства обмениваются информацией: блоками для шифрования и результатами операций. Для примера рассмотрим алгоритм подтверждения метки, представленный на рисунке 6.

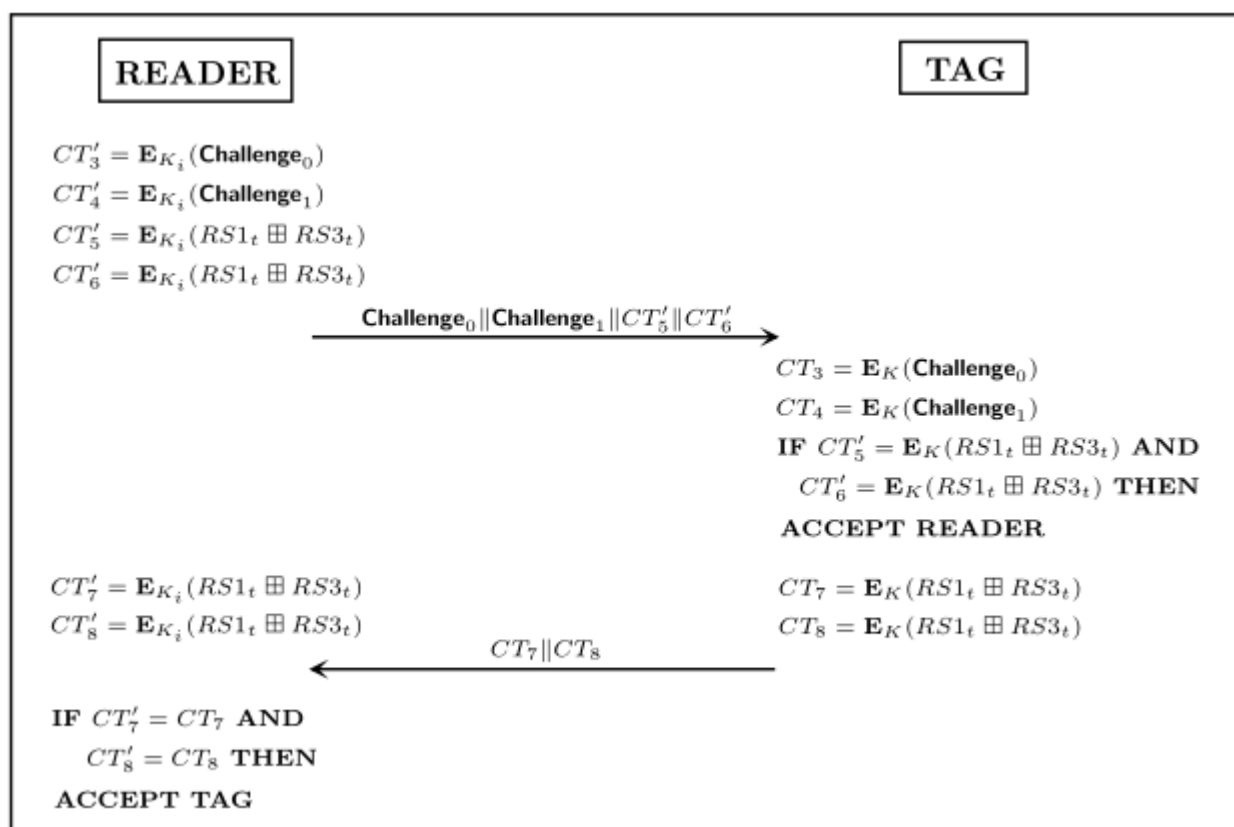


Рисунок 6 – процесс аутентификации алгоритма Hummingbird

Считыватель берет два случайных числа Challenge и производит их шифрование. Так же на основании текущего состояния системы производится

шифрование суммы двух регистров, и отправляется на метку. Метка, получив сообщение производит шифрование чисел Challenge. После установления нового состояния системы метка производит шифрование суммы двух регистров. Если результаты шифрования совпадают со значением, полученным от считывателя, то метка подтверждает данный считыватель. После этого метка формирует новое сообщение, в котором так же шифрует два числа на основании текущего состояния системы и посылает их считывателю. Считыватель в свою очередь так же производит шифрование и сравнивает их с полученными значениями от метки. При условии совпадения значений считыватель подтверждает данную метку. После прохождения данной процедуры метка и считыватель могут производить обмен данными зная, что данные будут корректно расшифрованы.

В данной процедуре используются принцип шифрования с открытым ключом [8], производится проверка на совпадение секретных ключей и начального состояния. Данная операция выполняется таким образом в целях обеспечения безопасности системы. Линии связи считывателя и метки являются открытыми, то есть к ним имеют доступ все желающие. Поэтому данные алгоритмы аутентификации позволяют настроить связь между устройствами без передачи секретных данных (ключа шифрования и начального состояния системы) по незащищенным линиям связи.

4 Тестирование модели

4.1 Моделирование с использованием встроенных утилит среды Quartus

Для тестирования модели проводилось моделирование нескольких ситуаций [11]: полное шифрование и дешифрование, шифрование и дешифрование с разными ключами, и с разными начальными состояниями. Данные ситуации позволят понять насколько чувствительный алгоритм к значению ключа и начальных данных.

Для первой ситуации укажем общий ключ, одинаковые данные для инициализации и сравним значения, получаемые на выходе шифратора и дешифратора. Данные, которые получены с шифратора передаются на вход дешифратора, таким образом вход данных на шифратор и выход дешифратора должны иметь одинаковые значения. Данные представлены в шестнадцатеричном виде.

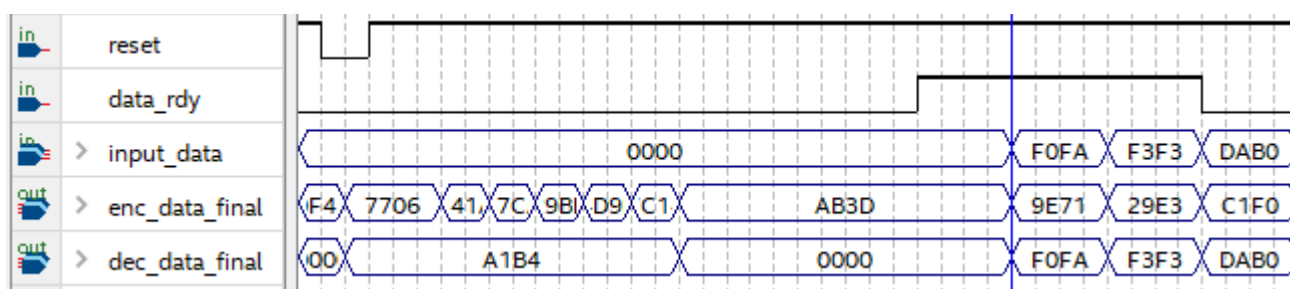


Рисунок 7 – график работы шифратора и дешифратора

Как видно по рисунку 7 данные совпадают, значит процесс шифрования происходит корректно. Теперь проверим чувствительность алгоритма к ключу, для этого шифратору и дешифратору укажем разные значения ключа и посмотрим на их работу. Значения ключа изменим в одном из бит.

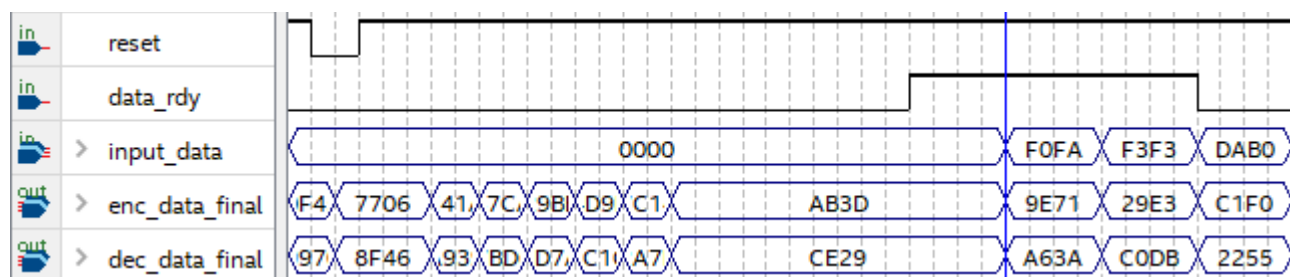


Рисунок 8 – график работы шифратора и дешифратора с разными ключами

Как видно по рисунку 8, незначительное изменение значения ключа привело к неправильной дешифрации данных. Благодаря структуре алгоритма изменение

ключа всего лишь в одном бите позволило существенно изменить полученные от дешифратора данные.

Далее протестируем работу модели, при разных начальных состояниях системы. Данное обстоятельство так же, как и неверный ключ, должно привести к невозможности корректно расшифровать данные.

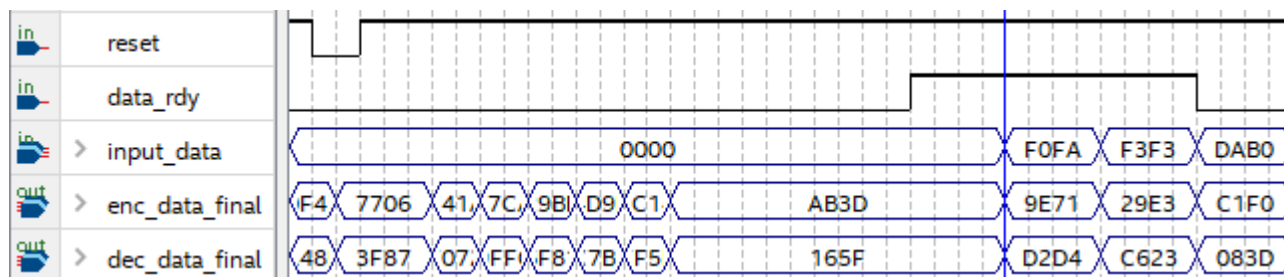


Рисунок 9 – график работы шифратора и дешифратора с разными начальными состояниями

Как видно на рисунке 9, изменение начального состояния системы для дешифратора (шифратор и дешифратор имеют разные начальные состояния) привело к неправильной дешифрации данных. Ввиду структуры алгоритма изменение начального состояния всего лишь в одном из регистров привело к изменению полученных от дешифратора данных.

4.2 Прототипирование на ПЛИС

Прототипирование будет производиться при помощи среды разработки Quartus на плате De1-SoC.

Скомпилируем проект и протестируем его работу на ПЛИС. В качестве примера будем шифровать один блок данных. Сравним входные данные на шифраторе со значением выходной шины дешифратора.

Ввод данных о ключе будет производиться из вне, так же, как и значения о начальном состоянии системы. При нажатии на KEY[0] будет происходить сброс и начнется инициализация. Данные для шифрования хранятся в блоке RAM памяти и подаются на вход каждый такт, с появления условия data_rdy. Сигнал data_rdy подается пользователем по нажатию на кнопку KEY[1]. Для корректного срабатывания кнопок они подключены к модулю криптографической системы через модуль debouncer, необходимый для предотвращениядребезга сигнала. Информация

о работе системы выводится на имеющиеся индикаторы: информация об окончании шифрования и дешифрования на диоды LEDR, а данные выводятся по блокам на семисегментные индикаторы в шестнадцатеричном формате.

Моделирование прошло успешно. В ходе прототипирования были получены результаты аналогичные представленным в пункте 4.1. Из этого можно полагать, что наше устройство способно работать на частоте 50MHz, что является стандартной тактовой частотой для нашего ПЛИС.

5 Анализ полученных результатов

Для анализа полученных результатов использовалась среда разработки Quartus. В ходе прототипирования модели на ПЛИС были получены данные о количестве используемых элементов. Для корректного сравнения данных с аналогами необходимо узнать количество используемых элементов для реализации шифрования. В нашей реализации мы получили 297 логических элементов (далее - ЛЕ). В зависимости от потребностей, данное значение можно уменьшить за счет реализации алгоритма шифрования с использованием тактового сигнала, и исполнения блоков шифрования и замены в цикле. Данное изменение позволит сократить количество ЛЕ приблизительно до 125 ЛЕ. В данной работе это производиться не будет, т.к. данное действие существенно увеличивает время обработки блока данных. Так же это действие ведет к изменению структуры управляющего устройства и изменению алгоритма управления модулем.

На ПЛИС De1-SoC имеется источник тактовых сигналов с частотой 50MHz. Наш алгоритм при прототипировании показал возможность работать и корректно обрабатывать данные на данной частоте. Причем каждый такт он обрабатывает один блок. Таким образом получаем пропускную способность нашей модели в приблизительно 762Mbps. Эффективность данного алгоритма в пересчете на пропускную способность на один ЛЕ составляет приблизительно 2.44.

Сравним данную реализацию с аналогами. Данные приведены в таблице 4.

Таблица 4

Данные реализации алгоритмов аналогов [15]

Cipher	Key Size	Block Size	FPGA Device	Total Occupied Slices	Max. Freq. (MHz)	Throughput (Mbps)	Efficiency (Mbps/# Slices)
PRESENT	80	64	Spartan-3 XC3S400-5	176	258	516	2:93
	128	64		202	254	508	2:51
PRESENT	80	64	Spartan-3E XC3S500	271	—	—	—
XTEA	128	64	Spartan-3 XC3S50-5	254	62:6	36	0:14
			Virtex-5 XC5VLX85-3	9; 647	332:2	20; 645	2:14
ICEBERG	128	64	Virtex-2	631	—	1; 016	1:61
SEA	126	126	Virtex-2 XC2V4000	424	145	156	0:368
AES	128	128	Spartan-2 XC2S30-6	522	60	166	0:32
AES			Spartan-3 XC3S2000-5	17; 425	196:1	25; 107	1:44
			Spartan-2 XC2S15-6	264	67	2:2	0:01
AES			Spartan-2 XC2V40-6	1; 214	123	358	0:29
AES			Spartan-3	1; 800	150	1700	0:9

Наш алгоритм шифрует данные блоками по 16 бит, используя 256 битный ключ. Моделирование было проведено на ПЛИС Cyclone V 5CSEMA5F31C6. Общее количество использованных элементов равно 297. Сравнивая с другими алгоритмами, можно утверждать, что данная реализация является достаточно удачной. Как мы видим наш алгоритм имеет средние показатели по количеству ЛЕ, и работает на достаточно небольшой частоте, что не мешает нам получать приличную пропускную способность. Данный результат возможен благодаря обработке блока данных за один такт.

Так же стоит заметить, что наш алгоритм более эффективный при обработке данных большой длины, т. к. в структуре алгоритма заложен процесс подготовки к шифрованию. Таким образом при желании зашифровать 2 байта информации, мы потратим 5 тактов на подготовку и 1 такт на шифрование. Это стоит учитывать при реализации алгоритма в структуре устройства.

При изменении алгоритма для уменьшения количества используемых ЛЕ, придется вводить тактовый сигнал в процесс шифрования и выполнять ряд

операций циклически. Данное изменение увеличит время выполнения как самого шифрования, так и процесса инициализации. Изменение в структуре может уменьшить пропускную способность в 4 или 16 раз, в зависимости от количества введенных циклов обработки.

ЗАКЛЮЧЕНИЕ

В результате работы разработана HDL модель криптографической системы с использованием алгоритма легковесного шифрования Hummingbird. Он может обрабатывать блоки данных по 16 бит с использованием ключа шифрования величиной 256 бит.

Проведено тестирование системы. Прототип корректно выполняет операции шифрования и дешифрования данных. Так же проведены тесты на чувствительность алгоритма к изменению ключа шифрования и изменения начального состояния. При неверных значениях ключа или начального состояния, исходные данные и расшифрованные не совпадают. Значит данные шифруются согласно алгоритму.

Проведен анализ полученных результатов. Полученная реализация алгоритма имеет право на существование. По количеству ЛЕ реализованный алгоритм сопоставим с аналогами. За счет потоковой обработки данных пропускная способность алгоритма получилась даже немного больше несмотря на малую величину блока данных.

Как было сказано в начале работы, нет универсального алгоритма. Данная модель подходит для систем, требующих достаточно высокую пропускную способность при средних затратах на используемые ЛЕ.

Данная модель может быть изменена под разные требования к скорости выполнения, компактности и надежности шифрования. При желании сделать алгоритм более компактным необходимо изменить структуру шифрования данных, что приведет к уменьшению пропускной способности в разы (от 4 до 16 раз).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Mayer M. Innovation at Google: the physics of data, [Электронный ресурс] PARC Forum, 2009. Режим доступа: www.slideshare.net/PARCIInc/innovation-at-google-the-physics-of-data
2. Большие и умные: в каких отраслях пригодится интернет вещей, [Электронный ресурс], Режим доступа: <https://www.rbc.ru/trends/industry/5dd3bea79a7947ee743c5fd7>
3. Paar C. Light-Weight Cryptography for Ubiquitous Computing//Securing Cyberspace Workshop IV: Special purpose hardware for cryptography –Attacks and Applications. (Los Angeles, December 4, 2006) University of California.
4. Hodjat A., Verbauwhede I. Minimum Area Cost for a 30 to 70 Gbits/s AES Processor. In IEEE Computer Society Annual Symposium on VLSI (ISVLSI 2004), 2004. 498–502 с.
5. Moradi A., Poschmann A., Ling S., Paar C., Wang H. Pushing the Limits: A Very Compact and a Threshold Implementation of AES”, EUROCRYPT 2011, LNCS 6632, 69–88 с.
6. ISO/IEC 29192-1:2(en) [Электронный ресурс], Режим доступа: <https://www.iso.org/obp/ui/#iso:std:iso-iec:29192:-1:ed-1:v1:en>
7. Aleksey Z. Lightweight cryptography (Part 1)// Вопросы кибербезопасности. 2015. (9). С. 26–43.
8. Elxsi T. Public Key Cryptography. 2009.
9. Engels D. [и др.]. Ultra-lightweight cryptography for low-cost RFID tags: Hummingbird algorithm and protocol, 2009.
10. Kumar N.C. [и др.]. Lightweight cryptography for distributed PKI based MANETS // International Journal of Computer Networks and Communications. 2018. № 2 (10). С. 69–83.
11. М. Р. The Verification Process / Р. М., 2008. 62–77 с.
12. Mahmoud R. [и др.]. Internet of things (IoT) security: Current status, challenges and prospective measures 2015. 336–341 с.

244.

14. Tharmat A. Assignment of Network Management On «Lightweight Cryptography». 2016.

15. Usman M. [и др.]. SIT: A Lightweight Encryption Algorithm for Secure Internet of Things // International Journal of Advanced Computer Science and Applications. 2017. № 1 (8). С. 1–10.

16. Yalla P., Kaps J.P. Lightweight cryptography for FPGAs 2009. 225–230 с.

17. Е.А. И., Е.А. Т. Анализ алгоритмов шифрования малоресурсной криптографии в контексте интернета вещей // TECHNICAL SCIENCES. 2019. (3). С. 182–186.

ПРИЛОЖЕНИЕ

```
module final_humming(
input          clk,
input          reset,
input          data_rdy,
input          [ 15:0] input_data,
input          [256:0] key,
input          [ 15:0] NONCE0,
input          [ 15:0] NONCE1,
input          [ 15:0] NONCE2,
input          [ 15:0] NONCE3,
output         [ 15:0] enc_data_final,
output         [ 15:0] dec_data_final,
output         enc_complete,
output         dec_complete);

reg [15:0] data_in;

wire [15:0] rs1;
wire [15:0] rs2;
wire [15:0] rs3;
wire [15:0] rs4;
wire [15:0] enc1_out;
wire [15:0] enc2_out;
wire [15:0] enc3_out;

wire [15:0] dec3_in;
wire [15:0] dec3_out;
wire [15:0] dec2_out;
wire [15:0] dec1_in;
wire rs_rdy;

wire [15:0] enc_data_out;
wire [15:0] dec_data_in;
wire [15:0] dec_data_out;

assign enc_data_final=(enc_complete)?enc_data_out:16'hx;
assign dec_data_final=(dec_complete)?dec_data_out:16'h0;
assign dec_data_in = enc_data_out;

encryption inst_enc( data_in,
                     key,
                     rs1,
                     rs2,
                     rs3,
                     rs4,
                     enc1_out,
                     enc2_out,
                     enc3_out,
                     enc_data_out);

decryption inst_dec( dec_data_in,
                     key,
                     rs1,
```

```

        rs2,
        rs3,
        rs4,
        dec3_in,
        dec3_out,
        dec2_out,
        dec1_in,
        dec_data_out);

initial_process inst_init(clk,
        reset,
        data_rdy,
        NONCE0,
        NONCE1,
        NONCE2,
        NONCE3,
        enc1_out,
        enc2_out,
        enc3_out,
        enc_data_out,
        dec3_in,
        dec3_out,
        dec2_out,
        dec1_in,
        rs1,
        rs2,
        rs3,
        rs4,
        enc_complete,
        dec_complete,
        rs_rdy);

always @(posedge clk)
begin
    if(~reset)
    begin
        data_in <= NONCE0 + NONCE2;
    end
    else if(rs_rdy)
    begin
        data_in <= input_data;
    end
end

endmodule

module encryption(
input  [15:0] datain,
input  [255:0]key,
input  [15:0] rs1,
input  [15:0] rs2,
input  [15:0] rs3,
input  [15:0] rs4,
output [15:0] enc1_out,
output [15:0] enc2_out,
output [15:0] enc3_out,
output [15:0] dataout);

```

```

wire [15:0] enc1_in;
wire [15:0] enc2_in;
wire [15:0] enc3_in;
wire [15:0] enc4_in;
wire [15:0] enc4_out;

assign enc1_in = datain + rs1;
assign enc2_in = enc1_out + rs2;
assign enc3_in = enc2_out + rs3;
assign enc4_in = enc3_out + rs4;

encryption_function inst1 (enc1_in, key[ 63:0 ], enc1_out);
encryption_function inst2 (enc2_in, key[127:64 ], enc2_out);
encryption_function inst3 (enc3_in, key[191:128], enc3_out);
encryption_function inst4 (enc4_in, key[255:192], enc4_out);

assign dataout = enc4_out;

endmodule

```

```

module decryption(
input [ 15:0] datain,
input [255:0]key,
input [ 15:0] rs1,
input [ 15:0] rs2,
input [ 15:0] rs3,
input [ 15:0] rs4,
output [ 15:0] dec3_in,
output [ 15:0] dec3_out,
output [ 15:0] dec2_out,
output [ 15:0] dec1_in,
output [ 15:0] dataout);

wire [15:0] dec4_out;
wire [15:0] dec2_in;
wire [15:0] dec1_out;

assign dec3_in = dec4_out - rs4;
assign dec2_in = dec3_out - rs3;
assign dec1_in = dec2_out - rs2;

decryption_function inst4 (datain, key[255:192], dec4_out);
decryption_function inst3 (dec3_in, key[191:128], dec3_out);
decryption_function inst2 (dec2_in, key[127:64 ], dec2_out);
decryption_function inst1 (dec1_in, key[ 64:1 ], dec1_out);

assign dataout = dec1_out - rs1;

endmodule

```

```

module initial_process(
input                                     clk,
input                                     reset,
input                                     data_rdy,
input [15:0] nonce0,
input [15:0] nonce1,

```

```

input                [15:0] nonce2,
input                [15:0] nonce3,
input                [15:0] enc1_out,
input                [15:0] enc2_out,
input                [15:0] enc3_out,
input                [15:0] enc_data_out,
input                [15:0] dec3_in,
input                [15:0] dec3_out,
input                [15:0] dec2_out,
input                [15:0] dec1_in,
output reg [15:0] rs1_out,
output reg [15:0] rs2_out,
output reg [15:0] rs3_out,
output reg [15:0] rs4_out,
output                                encComplete,
output                                decComplete,
output reg                                rs_rdy);

```

```

reg [15:0] LFSR;
reg [2:0] cnt;
reg enc_complete;
reg dec_complete;
reg [15:0] RS1;
reg [15:0] RS2;
reg [15:0] RS3;
reg [15:0] RS4;

```

```

assign encComplete = enc_complete;
assign decComplete = dec_complete;

```

```

always @(posedge clk)
begin
if(~reset)
begin
cnt = 0;
rs_rdy <= 0;
end
else
case(cnt)
'b0: begin
enc_complete <= 0;
dec_complete <= 0;
rs1_out <= nonce0;
rs2_out <= nonce1;
rs3_out <= nonce2;
rs4_out <= nonce3;
cnt <= cnt + 1;
end
'b1: begin
rs1_out <= rs1_out + enc_data_out;
rs2_out <= enc1_out + rs2_out;
rs3_out <= enc2_out + rs3_out;
rs4_out <= enc3_out + rs4_out;
cnt <= cnt + 1;
end
'b10: begin
rs1_out <= rs1_out + enc_data_out;

```

```

        rs2_out <= enc1_out + rs2_out;
        rs3_out <= enc2_out + rs3_out;
        rs4_out <= enc3_out + rs4_out;
        cnt <= cnt + 1;
    end
'b11: begin
    rs1_out <= rs1_out + enc_data_out;
    rs2_out <= enc1_out + rs2_out;
    rs3_out <= enc2_out + rs3_out;
    rs4_out <= enc3_out + rs4_out;
    cnt <= cnt + 1;

    end
'b100:begin
    rs1_out <= rs1_out + enc_data_out;
    rs2_out <= enc1_out + rs2_out;
    rs3_out <= enc2_out + rs3_out;
    rs4_out <= enc3_out + rs4_out;
    cnt <= cnt + 1;
    rs_rdy <= 1;

    end
'b101:begin
    cnt <= cnt + data_rdy;
    enc_complete <= data_rdy;
    dec_complete <= data_rdy;

    end
'b110:begin
    if(data_rdy)
    begin
        rs1_out <= rs1_out + enc3_out;
        rs2_out <= rs2_out + enc1_out + rs4_out + enc1_out + rs1_out + enc3_out;
        rs3_out <= rs3_out + enc2_out + LFSR;
        rs4_out <= rs4_out + enc1_out + rs1_out + enc3_out;
    end
    cnt <= cnt + !data_rdy;
    enc_complete <= data_rdy;
    dec_complete <= data_rdy;

    end
'b111:begin
    cnt <= cnt - data_rdy;
    enc_complete <= data_rdy;
    dec_complete <= data_rdy;
    end
default: cnt <= 4'b000;
endcase
end

endmodule

module encryption_function(
input  wire  [15:0] data_in,
input  wire  [63:0] key_in,
output wire  [15:0] data_out);

wire [15:0] key1_in, key2_in, key3_in, key4_in;
wire [15:0] xor1, xor2, xor3, xor4, xor5;
wire [15:0] S_box1, S_box2, S_box3, S_box4, S_box5;
wire [15:0] L_transf1, L_transf2, L_transf3, L_transf4;

```

```

assign key1_in = key_in[15:0];
assign key2_in = key_in[31:16];
assign key3_in = key_in[47:32];
assign key4_in = key_in[63:48];

assign xor1 = data_in ^ key1_in;

S1_box_enc s11 (xor1[15:12],S_box1[15:12]);
S2_box_enc s12 (xor1[11:8], S_box1[11:8]);
S3_box_enc s13 (xor1[ 7:4], S_box1[ 7:4]);
S4_box_enc s14 (xor1[ 3:0], S_box1[ 3:0]);

linear_transform_enc lt1 (S_box1, L_transf1);

assign xor2 = L_transf1 ^ key2_in;

S1_box_enc s21 (xor2[15:12],S_box2[15:12]);
S2_box_enc s22 (xor2[11:8], S_box2[11:8]);
S3_box_enc s23 (xor2[ 7:4], S_box2[ 7:4]);
S4_box_enc s24 (xor2[ 3:0], S_box2[ 3:0]);

linear_transform_enc lt2 (S_box2, L_transf2);

assign xor3 = L_transf2 ^ key3_in;

S1_box_enc s31 (xor3[15:12],S_box3[15:12]);
S2_box_enc s32 (xor3[11:8], S_box3[11:8]);
S3_box_enc s33 (xor3[ 7:4], S_box3[ 7:4]);
S4_box_enc s34 (xor3[ 3:0], S_box3[ 3:0]);

linear_transform_enc lt3 (S_box3, L_transf3);

assign xor4 = L_transf3 ^ key4_in;

S1_box_enc s41 (xor4[15:12],S_box4[15:12]);
S2_box_enc s42 (xor4[11:8], S_box4[11:8]);
S3_box_enc s43 (xor4[ 7:4], S_box4[ 7:4]);
S4_box_enc s44 (xor4[ 3:0], S_box4[ 3:0]);

linear_transform_enc lt4 (S_box4, L_transf4);

assign xor5 = L_transf4 ^ (key1_in ^ key3_in);

S1_box_enc s51 (xor5[15:12],S_box5[15:12]);
S2_box_enc s52 (xor5[11:8], S_box5[11:8]);
S3_box_enc s53 (xor5[ 7:4], S_box5[ 7:4]);
S4_box_enc s54 (xor5[ 3:0], S_box5[ 3:0]);

assign data_out = S_box5 ^ (key2_in ^ key4_in);

endmodule

module decryption_function(
input  wire  [15:0]  data_in,
input  wire  [63:0]  key_in,
output          [15:0]  data_out);

```

```

wire [15:0] key1_in, key2_in, key3_in, key4_in;
wire [15:0] xor1, xor2, xor3, xor4, xor5;
wire [15:0] S_box1, S_box2, S_box3, S_box4, S_box5;
wire [15:0] L_transf1, L_transf2, L_transf3, L_transf4;

assign key1_in = key_in[15:0];
assign key2_in = key_in[31:16];
assign key3_in = key_in[47:32];
assign key4_in = key_in[63:48];

assign xor1 = data_in ^ (key2_in ^ key4_in);

S1_box_dec s51 (xor1[15:12], S_box1[15:12]);
S2_box_dec s52 (xor1[11:8], S_box1[11:8]);
S3_box_dec s53 (xor1[7:4], S_box1[7:4]);
S4_box_dec s54 (xor1[3:0], S_box1[3:0]);

assign xor2 = S_box1 ^ (key1_in ^ key3_in);

linear_transform_dec lt1 (xor2, L_transf1);

S1_box_dec s41 (L_transf1[15:12], S_box2[15:12]);
S2_box_dec s42 (L_transf1[11:8], S_box2[11:8]);
S3_box_dec s43 (L_transf1[7:4], S_box2[7:4]);
S4_box_dec s44 (L_transf1[3:0], S_box2[3:0]);

assign xor3 = S_box2 ^ key4_in;

linear_transform_dec lt2 (xor3, L_transf2);

S1_box_dec s31 (L_transf2[15:12], S_box3[15:12]);
S2_box_dec s32 (L_transf2[11:8], S_box3[11:8]);
S3_box_dec s33 (L_transf2[7:4], S_box3[7:4]);
S4_box_dec s34 (L_transf2[3:0], S_box3[3:0]);

assign xor4 = S_box3 ^ key3_in;

linear_transform_dec lt3 (xor4, L_transf3);

S1_box_dec s21 (L_transf3[15:12], S_box4[15:12]);
S2_box_dec s22 (L_transf3[11:8], S_box4[11:8]);
S3_box_dec s23 (L_transf3[7:4], S_box4[7:4]);
S4_box_dec s24 (L_transf3[3:0], S_box4[3:0]);

assign xor5 = S_box4 ^ key2_in;

linear_transform_dec lt4 (xor5, L_transf4);

S1_box_dec s11 (L_transf4[15:12], S_box5[15:12]);
S2_box_dec s12 (L_transf4[11:8], S_box5[11:8]);
S3_box_dec s13 (L_transf4[7:4], S_box5[7:4]);
S4_box_dec s14 (L_transf4[3:0], S_box5[3:0]);

assign data_out = S_box5 ^ key1_in;

endmodule

```

```

module linear_transform_enc(
input  wire  [15:0] m,
output          [15:0] L);

wire [15:0] m6, m10;

assign m6 [15:0] = {m[9:0],m[15:10]};
assign m10[15:0] = {m[5:0],m[15:6]};

assign L = m ^ m6 ^ m10;

endmodule

module linear_transform_dec(
input  wire  [15:0] m,
output          [15:0] L);

wire [15:0] m2, m4, m12, m14;

assign m2 [15:0] = {m[13:0],m[15:14]};
assign m4 [15:0] = {m[11:0],m[15:12]};
assign m12 [15:0] = {m[3:0],m[15:4]};
assign m14 [15:0] = {m[1:0],m[15:2]};

assign L = m ^ m2 ^ m4 ^ m12 ^ m14;

endmodule

module S1_box_enc(

        input          [3:0]  in,
        output reg      [3:0]  out
);

always @*
    case(in)
        'h0: out = 'h8;
        'h1: out = 'h6;
        'h2: out = 'h5;
        'h3: out = 'hf;
        'h4: out = 'h1;
        'h5: out = 'hc;
        'h6: out = 'ha;
        'h7: out = 'h9;
        'h8: out = 'he;
        'h9: out = 'hb;
        'ha: out = 'h2;
        'hb: out = 'h4;
        'hc: out = 'h7;
        'hd: out = 'h0;
        'he: out = 'hd;
        'hf: out = 'h3;
    endcase

endmodule
//-----

```



```

module S2_box_enc(
    input          [3:0] in,
    output reg     [3:0] out
);

always @*
    case(in)
        'h0: out = 'h0;
        'h1: out = 'h7;
        'h2: out = 'he;
        'h3: out = 'h1;
        'h4: out = 'h5;
        'h5: out = 'hb;
        'h6: out = 'h8;
        'h7: out = 'h2;
        'h8: out = 'h3;
        'h9: out = 'ha;
        'ha: out = 'hd;
        'hb: out = 'h6;
        'hc: out = 'hf;
        'hd: out = 'hc;
        'he: out = 'h4;
        'hf: out = 'h9;
    endcase

endmodule
//-----
module S3_box_enc(
    input          [3:0] in,
    output reg     [3:0] out
);

always @*
    case(in)
        'h0: out = 'h2;
        'h1: out = 'he;
        'h2: out = 'hf;
        'h3: out = 'h5;
        'h4: out = 'hc;
        'h5: out = 'h1;
        'h6: out = 'h9;
        'h7: out = 'ha;
        'h8: out = 'hb;
        'h9: out = 'h4;
        'ha: out = 'h6;
        'hb: out = 'h8;
        'hc: out = 'h0;
        'hd: out = 'h7;
        'he: out = 'h3;
        'hf: out = 'hd;
    endcase

endmodule
//-----
module S4_box_enc(

```

```

        input      [3:0] in,
        output reg [3:0] out
    );

    always @*
        case(in)
            'h0: out = 'h0;
            'h1: out = 'h7;
            'h2: out = 'h3;
            'h3: out = 'h4;
            'h4: out = 'hc;
            'h5: out = 'h1;
            'h6: out = 'ha;
            'h7: out = 'hf;
            'h8: out = 'hd;
            'h9: out = 'he;
            'ha: out = 'h6;
            'hb: out = 'hb;
            'hc: out = 'h2;
            'hd: out = 'h8;
            'he: out = 'h9;
            'hf: out = 'h5;
            endcase

    endmodule
//-----
module S1_box_dec(

        input      [3:0] in,
        output reg [3:0] out
    );

    always @*
        case(in)
            'h0: out = 'hd;
            'h1: out = 'h4;
            'h2: out = 'ha;
            'h3: out = 'hf;
            'h4: out = 'hb;
            'h5: out = 'h2;
            'h6: out = 'h1;
            'h7: out = 'hc;
            'h8: out = 'h0;
            'h9: out = 'h7;
            'ha: out = 'h6;
            'hb: out = 'h9;
            'hc: out = 'h5;
            'hd: out = 'he;
            'he: out = 'h8;
            'hf: out = 'h3;
            endcase

    endmodule
//-----
module S2_box_dec(

```

```

        input          [3:0] in,
        output reg     [3:0] out
    );

```

```

always @*
    case(in)
        'h0: out = 'h0;
        'h1: out = 'h3;
        'h2: out = 'h7;
        'h3: out = 'h8;
        'h4: out = 'he;
        'h5: out = 'h4;
        'h6: out = 'hb;
        'h7: out = 'h1;
        'h8: out = 'h6;
        'h9: out = 'hf;
        'ha: out = 'h9;
        'hb: out = 'h5;
        'hc: out = 'hd;
        'hd: out = 'ha;
        'he: out = 'h2;
        'hf: out = 'hc;
    endcase

```

```

endmodule
//-----
module S3_box_dec(

```

```

        input          [3:0] in,
        output reg     [3:0] out
    );

```

```

always @*
    case(in)
        'h0: out = 'hc;
        'h1: out = 'h5;
        'h2: out = 'h0;
        'h3: out = 'he;
        'h4: out = 'h9;
        'h5: out = 'h3;
        'h6: out = 'ha;
        'h7: out = 'hd;
        'h8: out = 'hb;
        'h9: out = 'h6;
        'ha: out = 'h7;
        'hb: out = 'h8;
        'hc: out = 'h4;
        'hd: out = 'hf;
        'he: out = 'h1;
        'hf: out = 'h2;
    endcase

```

```

endmodule
//-----
module S4_box_dec(

```

```

        input          [3:0] in,

```

```

        output reg      [3:0]  out
    );

    always @*
        case(in)
            'h0: out = 'h0;
            'h1: out = 'h5;
            'h2: out = 'hc;
            'h3: out = 'h2;
            'h4: out = 'h3;
            'h5: out = 'hf;
            'h6: out = 'ha;
            'h7: out = 'h1;
            'h8: out = 'hd;
            'h9: out = 'he;
            'ha: out = 'h6;
            'hb: out = 'hb;
            'hc: out = 'h4;
            'hd: out = 'h8;
            'he: out = 'h9;
            'hf: out = 'h7;
        endcase

endmodule

```