|  |
| --- |
| Министерство образования Республики Беларусь |
|  |
| Учреждение образования |
| БелорусскиЙ государственный университет |
| информатики и радиоэлектроники |
|  |
| Факультет компьютерного проектирования |
|  |
| Кафедра электронных вычислительных средств |
|  |
| |  | | --- | | *К защите допустить*: | | Зав. кафедрой ЭВС | | \_\_\_\_\_\_\_\_\_\_А. А. Петровский | |
| Пояснительная записка |
| к дипломному проекту |
| на тему: |
|  |
|  |
| **ПОСЛЕДОВАТЕЛЬНЫЙ ПРОЦЕССОР АЛГОРИТМА ШИФРОВАНИЯ AES НА БАЗЕ FPGA** |
|  |
| БГУИР ДП 1-40 02 02 01 054 ПЗ |
|  |
| |  |  |  | | --- | --- | --- | | Студент |  | А. С. Шашков | | Руководитель |  | В. Б. Клюс | | Консультанты: |  |  | | *от кафедры ЭВС* |  | В. Б. Клюс | | *по экономической части* |  | Л. И. Старова | | *по охране труда* |  | О. С. Потоцкая | | Нормоконтролер |  | В. П. Луговский | | Рецензент |  |  | |  |  |  | |  |  |  | |

Минск 2012

**РЕФЕРАТ**

УДК 004.3

Шашков, А. С. Последовательный процессор алгоритма шифрования AES на базе FPGA : дипл. проект по спец. «Электронные вычислительные средства» / А. С. Шашков. – Минск : БГУИР, 2012. – 164 с.

В дипломном проекте был разработан процессор зашифрования и расшифрования алгоритма AES для различных ПЛИС фирмы Xilinx. Изначально были спроектированы несколько модификаций процессора, осуществляющего только режим зашифрования. Так, были получены модификации 11-тактового процессора зашифрования, 10-тактового процессора зашифрования, процессора зашифрования на базе Т-таблиц, а также модификации данных процессоров с использованием блочной и распределённой памяти. На базе двух лучших модификаций процессора зашифрования были разработаны две модификации процессора, осуществляющего как процедуру зашифрования, так и процедуру расшифрования.

Для лучших по показателю быстродействия процессоров зашифрования и зашифрования/расшифрования была проведена процедура размещения и трассировки для кристаллов Xilinx Virtex 5, 6 и 7, а также для Spartan 6. Лучший полученный процессор зашифрования имеет пропускную способность свыше 4 гигабит в секунду, а лучший разработанный процессор зашифрования и расшифрования имеет пропускную способность свыше 3 гигабит в секунду для кристаллов Virtex 5,6,7. Полученная пропускная способность позволяет данным разработкам получить применение в быстродействующих системах передачи и хранения данных.

Все представленные в проекте модификации процессоров могут представлять определённый интерес в зависимости от специфики конкретного приложения.

Ил.- 68, табл.- 9, прилож.- 4, список лит.- 7 назв.;

графическая часть – 6 л. формата А1

Ключевые слова: процессор, шифрование, AES, ПЛИС, FPGA.

**СОДЕРЖАНИЕ**

Введение7

1 Краткое описание алгоритма 8

1.1 Общее описание процесса зашифрования 8

1.2 SubBytes 8

1.3 ShiftRows10

1.4 MixColumns10

1.5 AddRoundKey11

1.6 Общее описание процесса расшифрования 13

1.7 InvSubBytes 14

1.8 InvShiftRows 15

1.9 InvMixColumns15

2 Обзор известных технических решений по теме проекта 17

3 Схемотехническое проектирование20

3.1 Анализ возможных архитектурных решений 20

3.2 Разработка структурной схемы30

3.3 Разработка функциональной схемы и реализация на ПЛИС процессора

зашифрования 32

3.3.1 11-тактовый процессор зашифрования с прямой реализацией функции

MixColumns 33

3.3.2 10-тактовый процессор зашифрования с прямой реализацией

функции MixColumns и отдельным блоком AddRoundKey для

обработки нулевого раунда38

3.3.3 10-тактовый процессор зашифрования с прямой реализацией

функции MixColumns и отдельным блоком AddRoundKey для

обработки десятого раунда 39

3.3.4 11-тактовый процессор зашифрования на архитектуре с

использованием Т-таблиц на базе однопортовой ROM-памяти 40

3.3.5 11-тактовый процессор зашифрования на архитектуре с

использованием Т-таблиц на базе синхронной двухпортовой

ROM-памяти 41

3.3.6 Исходное описание различных модификаций процессоров

зашифрования 43

3.3.7 Реализация модификаций процессора зашифрования на базе ПЛИС

фирмы Xilinx 44

3.3.8 Тестирование и анализ характеристик разработанных модификаций

процессора зашифрования 46

3.3.9 11-тактовый процессор зашифрования с прямой реализацией

функции MixColumns и синхронной памятью раундовых ключей51

3.3.10 Исходное описание, размещение и трассировка 11-тактового

процессора зашифрования с прямой реализацией функции

MixColumns и синхронной памятью раундовых ключей 53

3.4 Разработка функциональной схемы и реализация на ПЛИС процессора

зашифрования и расшифрования 56

3.4.1 Разработка модуля шифрования и расшифрования 56

3.4.2 Разработка функциональной схемы процессора зашифрования и

расшифрования с асинхронной памятью раундовых ключей 58

3.4.3 Разработка функциональной схемы процессора зашифрования и

расшифрования с синхронной памятью раундовых ключей59

3.4.4 Исходное описание, размещение и трассировка модификаций

процессора зашифрования и расшифрования 62

3.4.5 Анализ характеристик разработанных модификаций процессора

зашифрования и расшифрования 63

4 Технико-экономическое обоснование проекта 68

4.1 Краткая характеристика проведенной работы 68

4.2 Расчёт объёма и трудоёмкости программного обеспечения 68

4.3 Расчёт себестоимости и отпускной цены72

5 Охрана труда79

5.1 Проектирование и расчёт искусственного освещения при разработке

электронных систем 79

Заключение 85

Список использованных источников86

Приложение А (обязательное) Графические материалы 87

Приложение Б (обязательное) Исходные описания разработанных

модификаций процессоров96

Приложение В (обязательное) Отчёты Xilinx ISE 146

Приложение Г (обязательное) Результаты тестирования 154

Ведомость дипломного проекта 164

**ВВЕДЕНИЕ**

Проблема конфиденциальности информации появилась задолго до появления современных средств передачи информации. Эта проблема обусловила появление криптографии, одной из старейших наук, история который насчитывает несколько тысяч лет. Современная криптография - это наука о методах обеспечения конфиденциальности (невозможности прочтения информации посторонним) и аутентичности (целостности и подлинности авторства, а также невозможности отказа от авторства) информации.

Теперь, в современных условиях, когда огромные потоки информации могут относительно свободно и быстро распространяться по компьютерным сетям, могут легко копироваться на быстродействующие и очень компактные носители информации – проблема конфиденциальности стоит особенно остро. Так, было разработано множество алгоритмов шифрования данных при помощи ключа, рассчитанных на применение в компьютерных системах. Компьютерные алгоритмы шифрования отличаются тем, что помимо высокой криптографической стойкости они должны иметь структуру, приемлемую для реализации на интегральных схемах или в качестве прикладных программ для компьютерных систем.

Приемлемость алгоритма с точки зрения аппаратной реализации определяется такими требованиями, как минимальные суммарные аппаратные затраты, максимальное быстродействие, а также минимальное энергопотребление. Один из алгоритмов, удачно сочетающих все эти требования – Advanced Encryption Standard (или AES). В данном дипломном проекте была осуществлена попытка оптимальной реализации этого алгоритма на итерационной структуре для ПЛИС по критерию максимальной производительности при минимальных затратах ресурсов.

**1 КРАТКОЕ ОПИСАНИЕ АЛГОРИТМА**

В этом подразделе будет кратко описан алгоритм шифрования AES FIPS PUB 197 [1] (со 128-битным ключом) в виде, достаточном для понимания сути алгоритма с точки зрения проектирования вычислительной аппаратуры.

**1.1 Общее описание процесса зашифрования**

Входной блок данных разбивается на байты in0..in15; каждый байт соответствующим образом помещается в регистр State (смотрите рисунок 1.1). Далее содержимое регистра State шаг за шагом преобразуется в соответствующем порядке четырьмя следующими функциями: SubBytes(State), ShiftRows(State), MixColumns(State), AddRoundKey(State, RoundKey). Кратко рассмотрим содержание этих функций.

Полный процесс зашифрования может быть показан с помощью блок-схемы, показанной на рисунке 1.2.

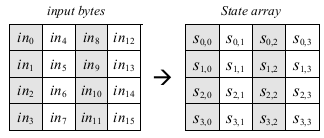


Рисунок 1.1 – Загрузка регистра State

**1.2 SubBytes**

Функция SubBytes (рисунок 1.3) заключается в однозначной замене каждого байта регистра State согласно таблице замен S-Box (рисунок 1.4). XY обозначает соответственно первую и второю части байта, подлежащего замене (здесь и далее для удобства будут использованы шестнадцатеричные числа). Данная таблица замен может быть получена с помощью вычисляющей комбинационной схемы, но этот метод заведомо медленнее прямого табличного (хоть и позволяет сэкономить ресурсы), поэтому в данном проекте такой метод не рассматривается.

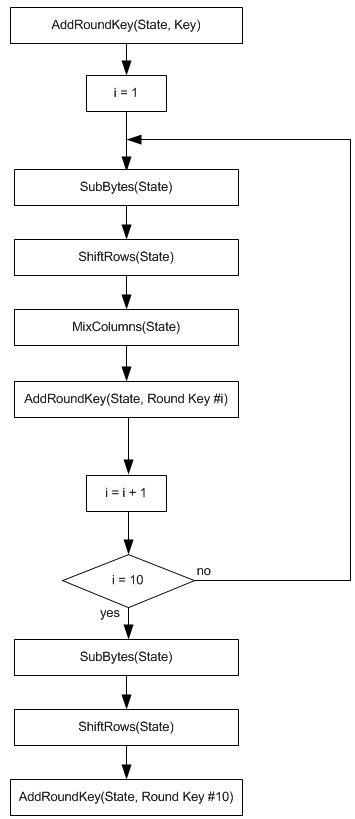


Рисунок 1.2 – Блок-схема алгоритма процесса зашифрования

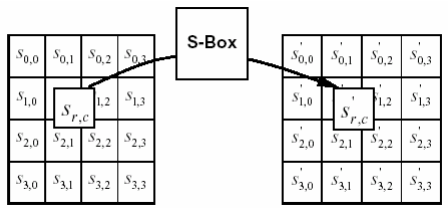


Рисунок 1.3 – Функция SubBytes

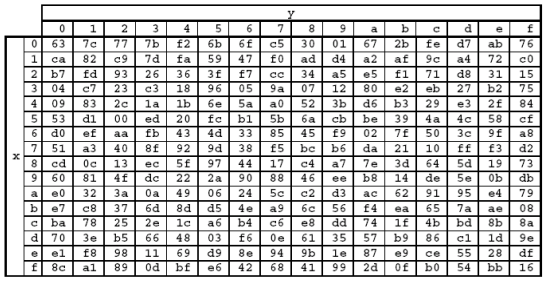


Рисунок 1.4 – Таблица замен S-Box

**1.3 ShiftRows**

Функция ShiftRows (рисунок 1.5) циклически сдвигает первый ряд регистра State на один байт, второй ряд на два байта и третий ряд, соответственно, на 3 байта.

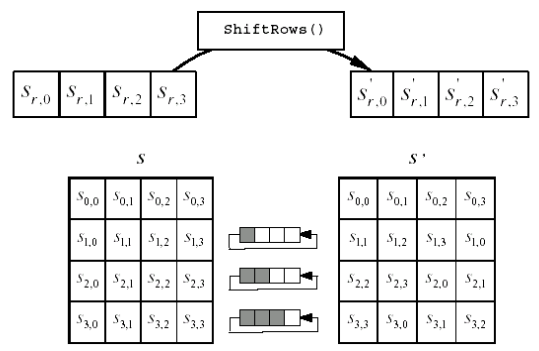


Рисунок 1.5 – Функция ShiftRows

**1.4 MixColumns**

Функция MixColumns (рисунок 1.6) – это замена каждого столбца регистра State на новый столбец, вычисление которого может быть описано следующими операциями в математическом поле GF(28), определение которого дано в стандарте FIPS PUB 197 [1]:

|  |  |
| --- | --- |
|  | (1.1) |

Далее в записке будет показано, каким образом выражения 1.1 будут преобразованы к удобному (для аппаратной реализации) виду при помощи функции Xtime(байт), также описанной в стандарте FIPS PUB 197. Функция Xtime довольно проста и сводится к следующим элементарным действиям:

1. Проверить старший бит входного байта;
2. Логически сдвинуть входной байт на один разряд влево;
3. Если в пункте 1 старший байт был равен единице, осуществить операцию побитового XOR числа 0x1B с результатом, полученным после пункта 2.

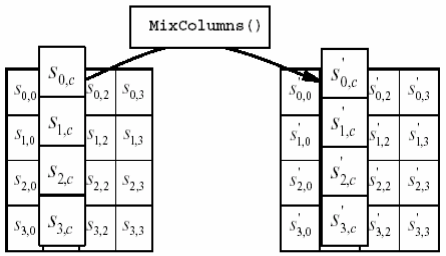


Рисунок 1.6 – Функция MixColumns

**1.5 AddRoundKey**

Функция AddRoundKey является простой операцией побитового XOR содержимого регистра State и соответствующего для конкретного раунда ключа (RoundKey). Функция изображена на рисунке 1.7.

Рассмотрим, как получаются раундовые ключи. Все раундовые ключи, как и входной (начальный) ключ, имеют размерность 128 бит. Первый раундовый ключ получается путём проведения ряда операций над входным ключом. Второй раундовый ключ получается из первого путём осуществления тех же самых операций и в том же самом порядке; остальные ключи получаются из предыдущих ключей по аналогии. Теперь рассмотрим эти операции. Для графического пояснения воспользуемся изображениями из [2].

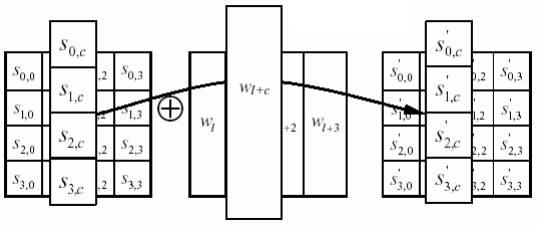


Рисунок 1.7 – Функция AddRoundKey

1. Последний столбец исходного ключа циклически сдвигается вверх (операция RotWord, рисунок 1.8).

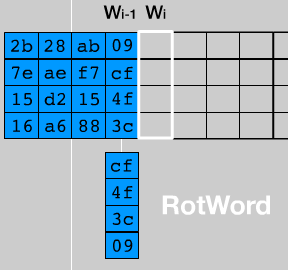


Рисунок 1.8 – Операция RotWord

1. Все байты сдвинутого столбца заменяются соответствующими байтами из таблицы замен S-Box (рисунок 1.9, рисунок 1.4).

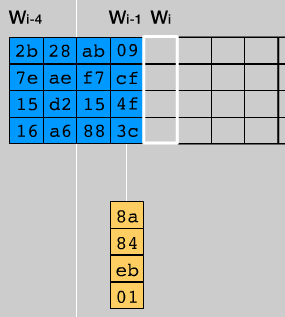


Рисунок 1.9 – Операция RotWord

1. Полученный результат складывается с содержимым первого столбца исходного ключа, а также с содержимым соответствующего столбца таблицы Rcon (рисунок 1.10, рисунок 1.11). Под сложением тут и далее понимается операция побитового XOR. Таблица Rcon содержит по одному столбцу для каждого соответствующего раундового ключа: первый столбец для первого ключа, второй для второго и т.д., всего 10 столбцов для 10 раундовых ключей. Результат этой операции и есть первый столбец получаемого раундового ключа.

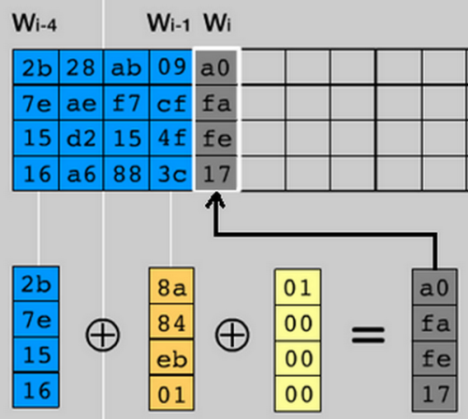


Рисунок 1.10 – Получение первого столбца

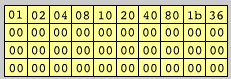


Рисунок 1.11 – Таблица Rcon

1. Второй столбец получается как сумма первого столбца получаемого ключа, и сумма второго столбца исходного ключа. Третий столбец получается аналогично, только складывается третий столбец исходного и второй столбец получаемого ключей. Четвёртый столбец получается суммированием четвёртого столбца исходного ключа и третьего столбца получаемого ключа (рисунок 1.12, 1.13, 1.14).

Аналогичным образом при инициализации процессора требуется получить последовательно все 10 раундовых ключей.

**1.6 Общее описание процесса расшифрования**

Процесс расшифрования может быть показан с помощью блок-схемы алгоритма, показанной на рисунке 1.15. От процесса зашифрования эта схема отличается также обратным порядком подачи раундовых ключей

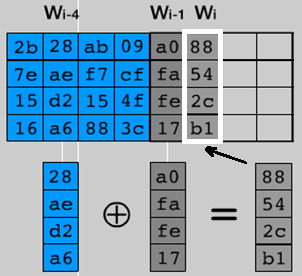


Рисунок 1.12 – Получение второго столбца

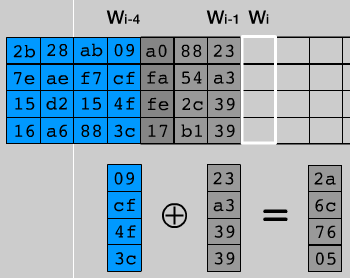


Рисунок 1.13 – Получение четвёртого столбца

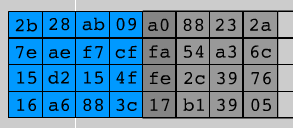


Рисунок 1.14 – Исходный и получаемый ключи

и инверсными версиями функций SubBytes(State), ShiftRows(State), MixColumns(State): InvSubBytes(State), InvShiftRows(State), InvMixColumns(State) соответственно. Рассмотрим эти функции.

**1.7 InvSubBytes**

InvSubBytes отличается от SubBytes только таблицей замены: в ней используется таблица Inverse S-Box вместо таблицы S-Box. Таблица Inverse S-Box показана на рисунке 1.16.

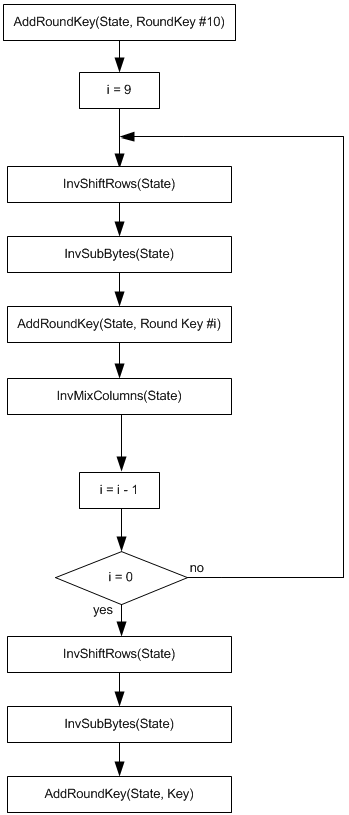


Рисунок 1.15 – Блок-схема алгоритма процесса зашифрования

**1.8 InvShiftRows**

InvShiftRows отличается от ShiftRows только направлением осуществления сдвигов. Функция InvShiftRows показана на рисунке 1.17.

**1.9 InvMixColumns**

InvMixColumns отличается от MixColumns только способом вычисления столбца-замены. Новый столбец вычисляется выражениями (1.2).

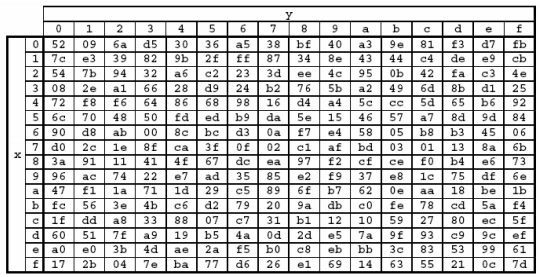


Рисунок 1.16 – Таблица Inverse S-Box

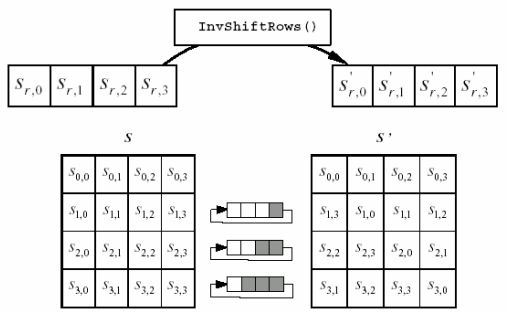


Рисунок 1.17 – Функция InvShiftRows

|  |  |
| --- | --- |
|  | (1.2) |

Так же, как и для MixColumns, выражения 1.2 для InvMixColumns приводятся к виду, удобному для аппаратной реализации при помощи функции Xtime, что также будет показано далее в пояснительной записке.

**2 ОБЗОР ИЗВЕСТНЫХ ТЕХНИЧЕСКИХ РЕШЕНИЙ ПО ТЕМЕ**

**ПРОЕКТА**

Важным этапом перед началом разработки является анализ известных решений, аналогичных разрабатываемому устройству.

В первую очередь рассмотрим разработку процессора зашифрования фирмы Helion Technology [3]. Данный процессор зашифрования построен на итерационной структуре и имеет аппаратный блок генерации раундовых ключей для 128-разрядного входного ключа. Характеристика данного процессора для кристаллов ПЛИС Virtex 5, 6 и Spartan 6 представлена в таблице 2.1 (данная разработка не задействует блочную память) [3].

Таблица 2.1 – Заявленные характеристики процессора зашифрования фирмы Helion Technology для различный кристаллов ПЛИС фирмы Xilinx

|  |  |  |  |
| --- | --- | --- | --- |
|  | Spartan 6 (-2) | Virtex 5 (-3) | Virtex 6 (-3) |
| Процессор зашифрования фирмы Helion Technology [3] | 332 слайса,  162 МГц,  1885 Мбит/с | 342 слайса,  363 МГц,  4224Мбит/с | 331 слайс,  450 МГц,  5236 Мбит/с |

Процессор зашифрования фирмы Helion Technology выдаёт результаты каждый 11ый такт. Более подробные характеристики данного процессора, а также его внутренняя структура не разглашаются.

Также в документе фирмы Helion Technology [3] указано, что данная фирма предоставляет процессоры зашифрования и расшифрования, но характеристики таких процессоров для разных семейств FPGA не указаны.

Другая разработка, близкая по характеристикам к разработке фирмы Helion Technology представлена в статье «Implementation of the AES-128 on Virtex-5 FPGAs» [4]. Данный процессор зашифрования построен на итерационном ядре, а также содержит блок получения раундовых ключей, причём полученный ключи помещаются в RAM-память. Характеристики данного процессора представлены в таблице 2.2. Укрупнённая структура ядра зашифрования данной разработки показана на рисунке 2.1.

Ввиду того, что в данном проекте нужно разработать процессор, осуществляющий как зашифрование, так и расшифрование, интересно узнать, какое соотношение по занимаемым ресурсам и быстродействию будут иметь процессор зашифрования и процессор зашифрования и расшифрования. Такое исследование представлено в статье «FPGA Implementations of S-box vs. T-box iterative architectures of AES» [5].

Таблица 2.2 – Заявленные характеристики процессора зашифрования, описанного в статье «Implementation of the AES-128 on Virtex-5 FPGAs» [4]

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Максимальная частота, МГц | Слайсы, штук | Объём BRAM, КБ | Энергопотребление, мВт | Быстродействие, Мбит/с |
| 350 | 400 | 0 | 617 | 4100 |

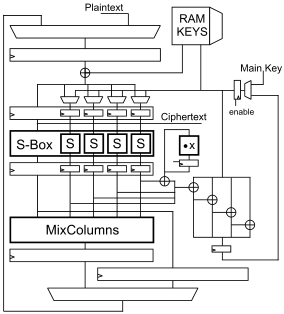


Рисунок 2.1 – Структура ядра зашифрования процессора зашифрования, описанного в статье «Implementation of the AES-128 on Virtex-5 FPGAs» [4]

Структура ядра зашифрования и расшифрования, предложенная в данной статье показана на рисунке 2.2. На рисунке 2.3 отображены полученные результаты. Как видно, представленный в статьей процессор зашифрования и расшифрования оказался медленнее процессора зашифрования на 35%, при этом занимая более чем в 2 раза больше слайсов.

Другие возможные архитектурные решения рассмотрим подробнее в разделе «3.1 Анализ возможных архитектурных решений».

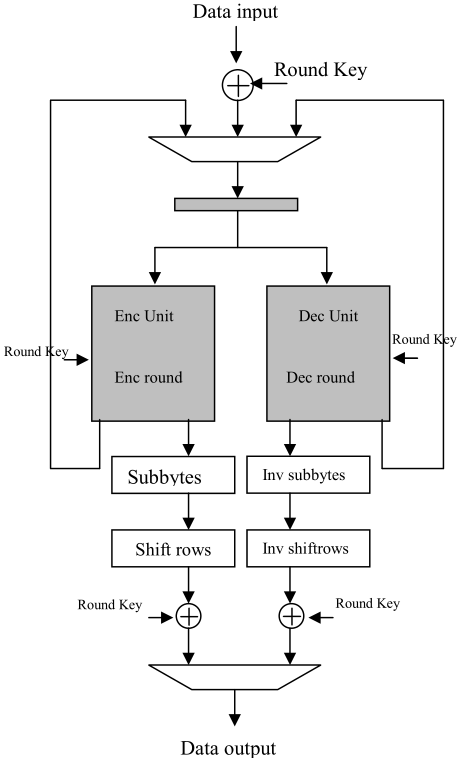


Рисунок 2.2 – Структура ядра зашифрования и расшифрования процессора зашифрования, описанного в статье «FPGA Implementations of S-box vs. T-box iterative architectures of AES» [5]

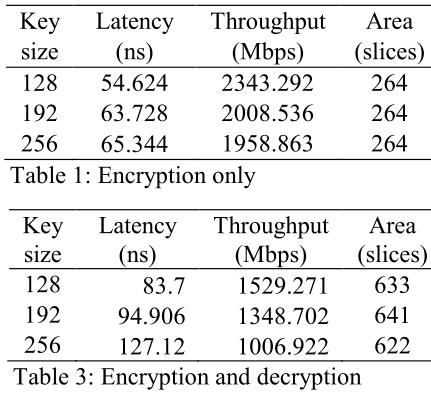


Рисунок 2.3 – Заявленные характеристики процессоров зашифрования и зашифрования и расшифрования, описанных в статье [5]

**3 СХЕМОТЕХНИЧЕСКОЕ ПРОЕКТИРОВАНИЕ**

**3.1 Анализ возможных архитектурных решений**

Ввиду особенностей алгоритма шифрования AES, многие функциональные блоки процессора могут быть реализованы различными схемотехническими способами, обеспечивающими в итоге одинаковую логическую функциональность.

Начнём анализ с блока генерации ключей. Ключи можно генерировать двумя способами: предварительно, либо одновременно с процессом шифрования. Однако для процесса расшифрования ключи требуется подавать в обратном порядке, таким образом генерировать ключи для процесса расшифрования одновременно с процессом расшифрования невозможно. Так как в проекте требуется реализовать и процесс расшифрования, выберем метод предварительной генерации ключей, который подойдёт как для режима шифрования, так и для режима расшифрования.

В техническом задании указано, что проектируемый процессор должен иметь итерационную структуру ядра. Под итерационной структурой ядра понимается такая его структура, которая обеспечивает последовательную и разделённую во времени обработку блоков входных данных на одном и том же (возможно, реконфигурируемом) ядре, представленном комбинационной схемой. Алгоритм AES является итерационным алгоритмом, поэтому построение такого типа ядра является одним из наиболее очевидных решений, позволяющих очень существенно сократить необходимые для построения такого процессора ресурсы, а также обеспечить существенно более низкое энергопотребление.

Возможно несколько различных способов построения итерационного ядра. В первую очередь рассмотрим вариант, который обеспечивает обработку одного раунда за один такт. Такое комбинационное ядро будет получать из входного регистра блок данных, а также соответствующий ему раундовый ключ. Выходной блок будет записан в тот же самый регистр, из которого был принят входной блок. Таким образом, после 11 итераций обработки будет получен готовый выходной блок, который можно снять прямо с входного/выходного регистра.

Алгоритм AES обладает таким важным свойством, как параллелизм при обработке одного раунда. Это свойство исходит из того, что, не считая функции ShiftRows, которая фактически перемешивает столбцы входного блока данных, остальные функции, входящие в состав комбинационной логики обработки раунда, проводятся исключительно над столбцами (подблоками данных по 4 байта) и результат обработки этих подблоков никак не зависит от других подблоков. Всё это значит, что итерационное ядро может обрабатывать один раунд на одном и том же 32-разрядном мини-ядре раздельно во времени за четыре такта, причём расходы на комбинационную логику непосредственной обработки раунда сократятся в четыре раза. Однако такое ядро будет обрабатывать входные блоки как минимум в 4 раза медленнее, чем 128-разрядное ядро, обрабатывающее один раунд за один такт. Также, введение подраундов обработки раундов приведёт к усложнению управляющей логики, а также к введению дополнительных мультиплексоров, что ещё сильнее уменьшит максимальную тактовую частоту. Поэтому, ввиду требования технического задания о максимально возможной производительности процессора, откажемся от 4-тактового раундового ядра.

Вариантом структуры ядра с раздельной обработкой подраундов может быть ядро, обрабатывающее два из четырёх подблоков за такт. Такая структура будет иметь в два раза большую производительность, и будет занимать в два раза больше ресурсов по отношению к структуре с четырьмя подраундами. Так же, как и четырёхраундовая структура, такая структура имеет важный недостаток – существенно более низкую производительность, поэтому данная структура не представляет большого интереса.

Итерационная структура является структурой, противопоставляемой развёрнутой или конвейерной. Под развёрнутой архитектурой понимается ядро полностью обрабатывающее весь блок данных за один такт на длинном ядре, состоящем из каскада повторяющихся секций, где каждая секция отвечает за обработку очередного раунда. При этом задержка до установления выходного сигнала в развёрнутом ядре возрастёт пропорционально количеству секций раундов – в примерно в 10 раз. Если между такими секциями поставить регистры, получится конвейерная структура. Конвейерная структура, разбитая на секции с помощью регистров, будет иметь задержку между регистрами, очень близкую к задержке в итерационном ядре. Это позволит такому конвейерному ядру работать на примерно той же частоте, что и итерационное ядро, но при этом выдавать результат шифрования очередного входного блока не через каждые n тактов, а в каждом следующем такте после обработки первого входного блока данных. Однако как развёрнутое ядро, так и конвейерное ядро занимают примерно в 10 раз больше ресурсов, а производительность конвейерного ядра излишне велика для множества возможных применений процессоров шифрования AES.

Одним из возможных вариантов развёрнутого ядра может быть ядро, состоящие из k раундовых блоков, каскадом стоящих друг за другом. Такое ядро может обрабатывать k раундов за такт. Но задержка до установления выходного сигнала также возрастёт в k раз, что даст в итоге производительность, близкую к производительности ядра с обычной итерационной структурой, и при этом ядро будет занимать в k раз больше ресурсов. Поэтому выбор такой структуры ядра не имеет большого смысла.

Обратным вариантом к развёрнутой структуре ядра может быть ядро, разделённое напополам регистром. Особенностью такого ядра является то, что комбинационная схема ядра делится на две части, при этом задержка между промежуточным регистром и входным или выходным регистром также уменьшается в два раза. Это позволяет увеличить тактовую частоту такого ядра примерно в два раза, с учётом возможных потерь на включения промежуточного регистра. Такое ядро имеет производительность близкую к обычному итерационному ядру, но при этом может работать на вдвое большей тактовой частоте. Дальнейшее разбиение подчастей на части позволит так же без увеличения итоговой производительности увеличивать тактовую частоту ядра. Это может быть использовано, если требуется согласовать ядро процессора шифрования по частоте с внешними блоками, использующими такое ядро. На практике такое разбиение имеет большой смысл только для конвейерных структур, поэтому откажемся от структуры с разделённым ядром.

Проанализировав возможные варианты построения итерационного ядра, остановимся на ядре, обеспечивающем обработку одного раунда за один такт.

Теперь рассмотрим, как может быть реализовано такое ядро.

Рассмотрим, как могут быть реализованы подблоки ядра: ShiftRows, SubBytes, MixColumns, AddRoundKey, InvShiftRows, InvSubBytes, InvMixColumns.

Рассмотрим функции ShiftRows и InvShiftRows. Данные операции осуществляют циклические сдвиги рядов регистра State. Аппаратно это функция наиболее рационально реализуется простейшей операцией перестановки входных сигналов и подачей их на выход, без каких-либо дополнительных преобразований. Таким образом, данная функция не задействует никаких ресурсов, кроме ресурсов связей.

Рассмотрим функции SubBytes и InvSubBytes. Эти функции заменяют каждый из 16 байт входного блока данных новыми данными, согласно таблицам замены S-Box и Inverse S-Box. Таблицы замен можно реализовать на комбинационной логике, вычисляющей байты-замены. Но наиболее производительным решением будет являться ROM-память. Блок данных State составляет 16 байт, значит, нужно 16 одинаковых параллельных модулей ROM памяти, по одному модулю на каждый байт. ROM-блоки могут быть реализованы на FPGA двумя способами: на распределённой памяти (на базе LUT), либо на базе блочной памяти. Далее в работе интересно рассмотреть оба варианта.

Рассмотрим функцию AddRoundKey. Данная функция представляет собой комбинационную схему, которая суммирует (побитовая операция XOR) входной блок данных и соответствующий ему раундовый ключ. Функция AddRoundKey является простейшей операцией над целым 128-битным блоком данных и не имеет альтернативных реализаций.

Рассмотрим функцию MixColumns. Это сложная функция, которая заменяет каждый столбец регистра State по сложному закону, описанному следующими выражениями 3.1:

|  |  |
| --- | --- |
|  | (3.1) |

Как описывается в стандарте AES, данные выражения приводятся к удобному для аппаратной реализации виду при помощи функции Xtime (смотрите раздел 1.4 MixColumns).

Проанализируем содержание выражений. Некоторые слагаемые представляют собой произведения в пространстве GF(28). Функция Xtime в этом пространстве представляется как произведение {02} • {XX}:

|  |  |
| --- | --- |
|  | (3.2) |

Воспользуемся свойством дистрибутивности умножения в поле GF(28) и преобразуем оставшееся произведение {03}•{*XX*}:

|  |  |
| --- | --- |
|  | (3.3) |

Таким образом, с помощью данных преобразований мы полностью исключили из заданных выражений операции умножения, оставив только функцию Xtime (которая реализуется простой комбинационной схемой) и функцию сложения (побитового XOR) – также простейшую операцию:

|  |  |
| --- | --- |
|  | (3.4) |
|  | (3.5) |
|  | (3.6) |
|  | (3.7) |

Далее рассмотрим функцию InvMixColumns. Данная функция аналогична функции MixColumns, но использует другие, алгоритмически более сложные выражения:

|  |  |
| --- | --- |
|  | (3.8) |

Преобразуем все умножения, содержащиеся в данных выражениях через функцию Xtime.

Воспользуемся следующими соотношениями, пояснёнными в стандарте AES:

|  |  |
| --- | --- |
|  | (3.9) |
|  | (3.10) |
|  | (3.11) |

Тогда:

|  |  |
| --- | --- |
|  | (3.12) |
|  | (3.13) |

|  |  |
| --- | --- |
|  | (3.14) |
|  | (3.15) |

Перепишем исходные выражения в виде, подходящем для аппаратной реализации:

|  |  |
| --- | --- |
|  | (3.16) |
|  | (3.17) |
|  | (3.18) |
|  | (3.19) |

В отличие от выражений функции MixColumns, выражения функции InvMixColumns используют целый каскад из функций Xtime (рисунок 3.1):



Рисунок 3.1 – Каскад из функций Xtime

Таким образом, мы получили функции, прямо описывающие комбинационные схемы, реализующие функции MixColumns и InvMixColumns. Как видно, эти функции являются довольно громоздкими комбинационными схемами. Другими словами это значит, что данные функции делают существенный вклад в задержку выдачи выходного сигнала ядра. Ввиду этого факта будет интересно рассмотреть вариант реализации этих функций на базе ROM-памяти, в которую записанные заранее рассчитанные значения функций.

Вариант реализации функции MixColumns на базе ROM-памяти получил в тематической литературе название T-Box, или T-таблиц. Ввиду того, что, согласно стандарту AES, функции ShiftRows и SubBytes (а также, InvShiftRows и InvSubBytes) могут быть переставлены местами без функционального изменения алгоритма, мы можем получить реализацию, в которой таблицы замен SubBytes и функция MixColumns стоят друг за другом в комбинационной схеме без искажения конечного результата. Вместо каскада двух разных ROM-блоков SubBytes и MixColumns разумно эти блоки заранее рассчитать вместе, как единый блок.

Как видно из алгоритма, как блок SubBytes, так и блок MixColumns независимо осуществляют операции над одинаковыми столбцами (4 байта) блока State. Таким образом, воспользовавшись матричным определением функции MixColumns, сформулируем функцию, осуществляющую преобразование SubBytes и MixColumns над одним столбцом входного блока State:

|  |  |
| --- | --- |
|  | (3.20) |

где b0-b3 представляют собой выходной столбец блока, a0-a3 – входной столбец перед функцией SubBytes, а SB – функцию SubBytes.

Из 3.20 получим выражение 3.21:

|  |  |
| --- | --- |
|  | (3.21) |

Заменим части выражения 3.21 на соответствующие Т-таблицы:

|  |  |
| --- | --- |
|  | (3.22) |

Запишем выражения для данных Т-таблиц:

|  |  |
| --- | --- |
|  | (3.23) |
|  | (3.24) |
|  | (3.25) |
|  | (3.26) |

Таким образом, мы получили выражение 3.22, которое с помощью содержимого таблиц, описанного выражениями 3.23-3.26, выбирает по одному байту данных из каждой таблицы для каждого bi, затем суммирует эти четыре байта по модулю два (операция побитового XOR), чтобы получить конечное значения байта bi. Выражение 3.22 является прямым описанием комбинационной схемы, реализующей совместную функцию SubBytes и MixColumns.

Каждая Т-таблица получает на вход один байт входного блока, и этот байт является адресом для выборки данных из памяти. Для любого входного байта в таблице содержится запись из четырёх байтов. Можно заметить, что каждая из таблиц содержит по две одинаковые строки SB(a). Значит, нет необходимости хранить эту строку дважды в одной таблице. Это позволяет сократить память до трёх байт на запись, что позволяет сократить объём памяти каждой Т-таблицы на 25%. Также можно увидеть, что все Т-таблицы содержат данные одного и того же типа: SB(a), 02xSB(a) и 03xSB(a). Это значит, что можно использовать любую из таблиц в качестве базовой, получая другие таблицы простой перестановкой связей в комбинационной схеме. Это значительно уменьшает трудоёмкость составления Т-таблиц на языке описания аппаратуры.

Однако в десятом раунде обработки входного слова функция MixColumns исключается из пути обработки данных. Тогда, на выход функции необходимо подавать только результат обработки входного блока функцией SubBytes. Результат обработки функцией SubBytes для каждого байта содержится в явном виде в Т-таблицах: SB(a). Таким образом, в случае 10го раунда строки, отвечающие за содержимое функции SB(a) должны непосредственно направляться на выход функции T-Box.

Ввиду того, что Т-таблицы представляют собой ROM-память, очевидны две возможные реализации таких таблиц: на базе распределённой памяти, а также на базе блочной памяти.

Очень удобным способом сокращения ресурсов блочной памяти является использование двухпортовой блочной памяти с синхронным доступом. Такая память доступна в любом из современных чипов FPGA фирмы Xilinx. На базе одного блока такой памяти можно функционально реализовать сразу две ROM-таблицы. Ввиду того, что Т-таблицы одинаковы по содержимому, можно хранить лишь одну копию таблицы в каждом блоке двухпортовой памяти, в которой будет два независимых порта чтения, функционирующих внешне как два раздельных блока ROM-памяти. Таким образом, вместо четырёх ROM-таблиц на столбец данных понадобится всего две таблицы. Суммарно для всего 128-битного блока данных State понадобится всего 8 таких двухпортовых блоков, вместо 16 однопортовых.

Существенным отличием при использовании двухпортовой памяти является то, что такая память может изменять выходные данные только по сигналу синхронизации. Это вносит некоторые изменения в общую архитектуру ядра. Так как выходы синхронной памяти неизменны между сигналами синхронизации, такая память может одновременно служить регистром данных, необходимым для хранения промежуточных данных после обработки предыдущего раунда. Получается, что в выходной регистр данные заносятся только в последнем раунде, а в самом ядре обратная связь реализуется без участия этого выходного регистра, что отличает данную реализацию от архитектур без использования синхронной ROM-памяти.

Далее в проекте будет сделано сравнение всех выше описанных способов реализации ядра на базе Т-таблиц.

Особенностью алгоритма AES является то, что перед обработкой в первом раунде, входной блок предварительно пропускается через функцию AddRoundKey совместно с нулевым или десятым раундовым ключом. Эта операция составляет весь нулевой раунд обработки входного блока данных. Другие же раунды существенно сложнее нулевого и содержат полный набор сложных функций: SubBytes, MixColumns или их инверсные варианты – наряду с функцией AddRoundKey. Такая особенность сразу предлагает несколько вариантов реализации ядра. В одном случае возможно использование общего ядра, которое в нулевом раунде мультиплексирует путь данных таким образом, чтобы оставить только функцию AddRoundKey. Такой метод позволяет сократить суммарные ресурсы. В другом случае возможны несколько вариантов реализации ядра, при котором блок AddRoundKey дублируется на входе в ядро. Это позволяет рассчитывать нулевой раунд ещё до поступления блока данных в ядро и одновременно выполнять обработку предыдущего раунда на ядре. При таком методе выходные блоки данных после первого выходного блока могут быть считаны каждые 10 тактов, вместо 11 тактов без использования дополнительной логики, требуемой для реализации дополнительного блока AddRoundKey. Ещё одним способом реализации процесса шифрования 10 тактов может быть дополнительный модуль AddRoundKey, включённый в ядро непосредственно перед выходным регистром и после функции SubBytes. При такой схеме обработка 9го и 10го раундов возможны в одном последнем такте, что в даёт суммарно 10 тактов на обработку всего входного блока данных.

**3.2 Разработка структурной схемы**

Теперь разработаем структурную схему для всех возможных реализаций архитектур, предложенных к рассмотрению в предыдущем разделе.

Покажем структурную схему процессора шифрования на базе итерационного ядра (рисунок 3.2):



Рисунок 3.2 – Структурная схема процессора шифрования и расшифрования

Определим внешние связи процессора шифрования. Внешние связи состоят из входной 128-битной шины INPUT\_TEXT и выходной 128-битной шины OUTPUT\_TEXT, а также из следующих внешних входных и выходных сигналов управления: сигнал сброса RESET, сигнал выбора режима ENCR/DECR, сигнал готовности входных данных (входного ключа либо входного блока) DATA\_WRITE, сигнал готовности генерации всех раундовых ключей и готовности выходных данных AVAILABLE. Одна и та же шина INPUT\_TEXT используется как для приёма ключа, так и для приёма входного блока данных, что обусловлено тем, что приём ключа и приём блока данных не могут происходить одновременно. Сигнал AVAILABLE также имеет двойное назначение в виду разделения во времени процедуры генерации раундовых ключей и процедуры шифрования и зашифрования. Так, выданный сигнал AVAILABLE сигнализирует о готовности всех раундовых ключей после процедуры генерации ключей, а после процедуры шифрования и расшифрования сигнал AVAILABLE обозначает готовность выходного блока данных. Сигнал DATA\_WRITE по аналогии начинает как процесс ввода ключа в модуль генерации ключей, так и процесс ввода блока данных в модуль зашифрования и расшифрования. Так как процессор является синхронным, он также имеет вход для задающего тактового сигнала (CLK).

Работа процессора начинается с процесса инициализации памяти раундовых ключей. После включения устройства по приходу сигнала DATA\_WRITE модуль управления запускает работу модуля генерации ключей. В следующем такте после фиксации сигнала DATA\_WRITE модуль генерации ключей по шине INPUT\_TEXT снимает значение ключа шифрования; запускается счётчик, который в последующих тактах управляет правильной генерацией раундовых ключей. Сгенерированные ключи, а также исходный ключ помещаются в соответствующие ячейки модуля памяти. По завершении генерации и записи всех ключей модулем управления выдаётся внешний сигнал AVAILABLE. После этого процессор может начинать непосредственно процесс шифрования или расшифрования.

Входной однобитный сигнал ENCR/DECR выбирает режим работы процессора шифрования: зашифрование (1) или расшифрование (0); этот сигнал должен быть неизменным в течение всего цикла шифрования одного блока входных данных.

По приходу внешнего сигнала готовности блока входных данных DATA\_WRITE модуль управления направляет данные с шины INPUT\_TEXT в модуль зашифрования и расшифрования, а также запускает счётчик и модуль состояния и вывода.

Модуль зашифрования и расшифрования обрабатывает содержимое модуля состояния и вывода и направляет результат на выход устройства. Запуск модуля состояния и вывода означает, что обработанные модулем зашифрования и расшифрования данные теперь могут обновить содержимое выходного регистра модуля зашифрования/расшифрования по сигналу CLK. Нужно отметить, что для некоторых структур ядра в случае первого и последнего раундов обработки данных модуль зашифрования и расшифрования не задействует часть внутренних функций; выбор необходимых функций для текущего раунда выбирается сигналом счётчика.

Счётчик также последовательно выбирает соответствующие ключи, расположенные в модуле памяти, а модуль памяти направляет эти ключи непосредственно в модуль зашифрования и расшифрования. Важно отметить, что в режиме зашифрования ключи подаются в прямом порядке, а в режиме расшифрования – в обратном. Это необходимо учесть при проектировании функции выбора адреса элемента памяти счётчиком.

Сигнал CLK обеспечивает разделение этапов работы некоторых модулей процессора во времени, а именно: модуля генерации ключей, записи в модуль памяти, смены состояния счётчика, смены состояния модуля управления, обновления состояния выходного регистра модуля зашифрования и расшифрования.

В конце процесса шифрования готовый блок данных (128 бит) сопровождается сигналом готовности результата AVAILABLE, который также является стробирующим сигналом для внешнего устройства, принимающего выходные данные. Одновременно с процессом снятия выходных данных внешнее устройство может удерживать сигнал готовности следующего блока входных данных (DATA\_WRITE). Таким образом, процесс снятия выходных данных и процесс подачи новых входных данных в процессор шифрования будут совмещены во времени.

Показанная на рисунке 3.2 структурная схема описывает работу процессора зашифрования и расшифрования. Однако в целях обеспечения последовательности разработки начнём проектирование с процессора, который осуществляет только зашифрование, а затем модифицируем его архитектуру, добавив в лучшую из версий процессора зашифрования блоки расшифрования.

Структурная схема процессора зашифрования будет повторять структурную схему процессора зашифрования и расшифрования за исключением отсутствия в процессоре зашифрования сигнала выбора режима зашифрования или расшифрования ENCR/DECR.

**3.3 Разработка функциональной схемы и реализация на ПЛИС**

**процессора зашифрования**

Начнём функциональное проектирование процессора с версии, реализующей только зашифрование. Построим все предложенные варианты функциональных схем процессора на процессоре зашифрования, затем выберем наилучшую из реализаций, на базе которой сделаем процессор, реализующий как зашифрование, так и расшифрование.

**3.3.1 11-тактовый процессор зашифрования с прямой реализацией**

**функции MixColumns**

Укрупнённая функциональная схема процессора зашифрования, реализующая прямую комбинационную реализацию функции MixColumns согласно формулам 3.1-3.7 представлена на рисунке 3.3.

Управляющим модулем устройства является автомат управления FSM. Покажем блок-схему алгоритма его работы на рисунке 3.4.

Блок распределённой памяти раундовых ключей RoundKeyRAM имеет один адресный вход, который определяет текущую ячейку памяти. Данные из ячейки памяти считываются и подаются на выход асинхронно согласно адресу на входе; если входной сигнал разрешения записи WE активен, запись данных с входа INPUT в текущую ячейку разрешается по приходу фронта тактирующего сигнала.

Процессор шифрования работает в двух режимах: режим инициализации памяти раундовых ключей и режим зашифрования. В начале работы процессора счётчик Counter\_to\_10 сигналом RESET сбрасывается в состояние 0. Сигнал сброса RESET также выполняет и функцию сброса всех раундовых ключей (при необходимости смены ключа). Счётчик Counter\_to\_10 выполняет в процессоре сразу несколько функций. Во-первых, счётчик определяет номер раунда, для которого генерируется ключ (в блоке KeySchedule), а также определяет адрес ячейки памяти, в которую нужно записать сгенерированный ключ. Во-вторых, он подключён к блоку зашифрования, который отключает соответствующие блоки в нулевом и десятом раундах. В-третьих, в режиме непосредственной работы состояние этого счётчика также является адресом, который выбирает из памяти нужный раундовый ключ непосредственно на вход модуля зашифрования.

Состояние i\_0 автомата FSM – состояние ожидания запуска. Автомат переходит в следующее состояние i\_1 по приходу сигнала DATA\_WRITE. По фронту тактового импульса при активном сигнале RESET управляющий автомат всегда переходит в состояние i\_0 из любого другого состояния.

Состояние i\_1 автомата FSM является состоянием загрузки ключа шифрования в регистр инициализации памяти раундовых ключей. Сигнал control\_enable\_reg является стробирующим сигналом для регистра reg\_128\_en; посредством сигнала control\_write\_zero\_key мультиплексор

|  |  |
| --- | --- |
|  | Рисунок 3.3 – Укрупнённая функциональная схема 11-тактового процессора зашифрования с прямой реализацией функции MixColumns |



Рисунок 3.4 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET) для раздела 3.3.1

MUX\_128\_2 переключается в режим подачи данных (ключа шифрования) с шины данных на вход регистра. В следующем такте (по фронту) входные данные установятся на выходе этого регистра, после чего будут неизменны до прихода следующего фронта тактового импульса.

Состояние i\_2 автомата FSM является состоянием генерации и записи раундовых ключей в память KeyScheduleRAM. Сигнал control\_write\_ram разрешает запись содержимого регистра reg\_128\_en в память по приходу фронта тактового импульса. Одновременно сигнал с регистра поступает в модуль KeySchedule; на выходе модуля устанавливается значение следующего раундового ключа; далее сигнал поступает на вход регистра через мультиплексор (т.к. теперь control\_write\_zero\_key = 0). Напомним, что для генерации раундового ключа и задания адреса записи в память требуется информация от счётчика, поэтому сигналом control\_start\_counter обеспечивается увеличение значения счётчика на единицу по фронту каждого тактового импульса. Таким образом, по приходу фронта следующего тактового импульса произойдут следующие события: текущее содержимое регистра запишется в память по адресу, указываемому счётчиком; новый раундовый ключ, находящийся на входе регистра запишется в этот регистр, а также увеличится значение счётчика; сигнал с выхода регистра пройдёт через модуль KeySchedule и значение нового раундового ключа установится на входе регистра. Этот процесс будет циклически повторяться, пока значение счётчика не будет равно 10, что будет означать окончание инициализации и запись последнего ключа в следующем такте; в этом же такте сигнал control\_data\_available установится в ‘1’. В следующем такте автомат перейдёт в состояние s\_0, а на выходе устройства установится единичный сигнал AVAILABLE, который сообщит внешнему устройству-приёмнику, что процесс генерации раундовых ключей окончен, и можно осуществлять загрузку входных блоков данных для последующего их зашифрования. Счётчик также перейдёт в следующее состояние 0 (счётчик циклический, после значения 10 переходит в 0; таким образом, не требуется дополнительный сигнал сброса счётчика).

Состояние s\_0 автомата FSM – состояние ожидания запуска процесса зашифрования. Автомат переходит в следующее состояние s\_0\_2 по приходу сигнала DATA\_WRITE. Входные данные при этом по фронту следующего такта записываются во входной регистр.

Состояние s\_0\_2 обозначает собой переход процессора в режим начала непосредственной работы, а именно – режим ввода блока данных из входного регистра в ядро зашифрования. В этом состоянии автомат выдаёт сигнал control\_input, который выбирает вход IN\_1 мультиплексора (режим ввода начальных данных), а также подаёт сигнал control\_start\_counter, который включает как счётчик, так и регистр состояния и вывода reg\_128\_en на выходе процессора. Сигнал control\_initialized = 1 в данном состоянии, что обозначает возможность записи результата обработки нулевого раунда ядром зашифрования в выходной регистр ядра reg\_128\_en. В следующем такте в выходной регистр ядра запишется конечное значение обработки входного блока для нулевого раунда (счётчик равен 0). Также в следующем такте значение счётчика увеличится, а автомат перейдёт в состояние s\_1.

Состояние s\_1 – это состояние циклической обработки данных выходного регистра ядра (через обратную связь). В каждом последующем такте счётчик выбирает нужные для правильной работы блоки обработки данных в модуле шифрования, а промежуточный результат шифрования, записанный в выходной регистр, подаётся на вход обратной связи блока шифрования. Когда счётчик будет иметь значение 10, автомат выдаст сигнал control\_data\_available. В следующем такте результат зашифрования (или расшифрования) запишется в выходной регистр ядра, а на выходе регистра, принимающего сигнал control\_data\_available, установится единичный сигнал, который позволит содержимому выходного регистра ядра быть записанным в выходной регистр устройства reg\_128\_en. Выходной регистр устройства меняет своё содержимое только один раз за полный цикл зашифрования и введён в устройство с целью обеспечения возможности корректной проверки выходного сигнала.

Также, если в состоянии s\_1 при счётчике равном 10 сигнал DATA\_WRITE = ‘1’, в следующем такте автомат перейдёт в состояние s\_0\_2, и новый блок данных сразу подастся в блок зашифрования. Если сигнал DATA\_WRITE = ‘0’, то устройство перейдёт в состояния ожидания s\_0 и будет ожидать новый блок данных, который будет сопровождаться сигналом сигнал DATA\_WRITE = ‘1’. Таким образом, устройство сможет не только выдать готовые данные, но и принять новый блок данных при активном сигнале сигнал DATA\_WRITE = ‘1’. Это возможно благодаря тому, что в состоянии s\_0\_2 мультиплексором разрывается обратная связь от регистра состояния, и ничего не препятствует корректной обработке нового блока данных.

Опишем подробнее структуру ядра зашифрования. В нулевом раунде сигнал автомата control\_input мультиплексирует путь входных данных через мультиплексор MUX\_Input сразу на блок AddRoundKey, после которого обработанные данные поступают на вход выходного регистра. Таким образом, нулевой раунд обрабатывается на такой структуре один такт. Суммарно на обработку входного блока данных уходит 11 тактов, причём в 11ом такте можно совмещать ввод нового блока с выводом предыдущего обработанного блока. Свойством последнего (десятого) раунда является выключаемый в этом раунде блок MixColumns. Мультиплексор MUX\_Sel обеспечивает такое выключение блока MixColumns из пути прохождения обработки данных.

**3.3.2 10-тактовый процессор зашифрования с прямой реализацией**

**функции MixColumns и отдельным блоком AddRoundKey для**

**обработки нулевого раунда**

Данная структура представляет собой модификацию описанного в разделе 3.3.1 процессора зашифрования. Блок-схема алгоритма работы автомата для новой модификации процессора зашифрования показана в Приложении А на рисунке А.1. Полная функциональная схема новой модификации процессора представлена в Приложении А на рисунке А.2. В данном процессоре подверглись модификации блок генерации ключей, блок зашифрования, а счётчик в данной модификации считает от нуля до 9; соответствующим модификациям подвергся и управляющий автомат.

Опишем новую структуру блока генерации ключей. В данной структуре одна и та же линия связи соединяет как выходной регистр для блока KeySchedule, так и модуль памяти RoundKeyRAM. При такой связи данные на этой линии по тому же фронту тактового импульса записываются как в регистр, так и в память. Таким образом, уже в состоянии автомата i\_1 (состояние ввода ключа в модуль генерации раундовых ключей) можно подавать сигналы control\_write\_ram (разрешение записи в память) и control\_start\_counter (сигнал запуска счётчика). Уже в следующем такте после состояния i\_1 автомат перейдёт в состояние i\_2, за первый такт которого сгенерируется и запишется в память уже следующий раундовый ключ.

Получается, что по сравнению с предыдущей версией такого процессора изменяются управляющие сигналы состояния автомата i\_1.

Ядро зашифрования в этой модификации реализовано практически без изменений. Однако ввиду того, что в нулевом раунде для обработки блока данных необходим только модуль AddRoundKey, модуль AddRoundKey (вместе с выходным регистром) вынесен в отдельное микроядро для реализации только нулевого раунда. Выход регистра этого микроядра подключён на вход к основной части ядра. Значит, основное ядро (которое теперь занимается только раундами 1-10) может обойтись теперь только одним единственным мультиплексором MUX\_Sel, который отключает блок MixColumns в последнем раунде шифрования.

Микроядро, реализующее обработку нулевого раунда, всегда получает на один из входов нулевой раундовый ключ (входной ключ). Это значит, что для одновременной обработки 10го раунда предыдущего блока данных и нулевого раунда следующего блока данных необходимо иметь доступ одновременно к 10му и нулевому ключу. В данном процессоре использовано решение, при котором нулевой ключ хранится в отдельном регистре, а ключи с 1го по 10ый – записываются в блок памяти KeyScheduleRAM.

Также, для данного процессора нужен счётчик, имеющий 10 различных состояний: от 0 до 9.

**3.3.3 10-тактовый процессор зашифрования с прямой реализацией**

**функции MixColumns и отдельным блоком AddRoundKey для**

**обработки десятого раунда**

Данная версия процессора зашифрования является модификацией процессора зашифрования, описанного в разделе 3.3.1. Полная функциональная схема нового процессора представлена в Приложении А на рисунке А.3. Блок-схема алгоритма работы автомата для новой модификации процессора зашифрования показана в Приложении А на рисунке А.4. Так, изменению подверглось ядро зашифрования, когда как модуль генерации ключей остался почти без изменений. Соответствующим образом изменились и состояния автомата управления. Также, ввиду сокращения числа тактов, необходимых для проведения всех раундов преобразования до 10, в данной модификации используется счётчик от 0 до 9, вместо счётчика от 0 до 10.

Так, входные данные, подающиеся через блок MUX\_Input, поступают сначала на блок AddRoundKey, в который подаётся нулевой ключ. Далее данные обрабатываются блоками ShiftRows, SubBytes и MixColumns – блоками, не зависящими от раундовых ключей. Таким образом, перед выходом регистра reg\_128 устанавливается промежуточный блок данных, представляющий собой процесс полной обработки нулевого раунда и частичной обработки первого раунда, без учёта операции AddRoundKey с первым ключом. Однако в последующем такте на начальный блок AddRoundKey будет подаваться уже первый ключ, таким образом, после первого блока AddRoundKey в ядре установится сигнал, соответствующий обработке после первого раунда. Перед выходом регистра обратной связи reg\_128 установится уже частично обработанный второй раунд. Так будет повторяться до девятого (последнего) такта обработки. На девятом такте требуется выключать блок MixColumns из пути прохождения данных (этот такт соответствует десятому раунду). Получается, что в этом такте на выходе начального блока AddRoundKey устанавливается результат обработки после 9 раундов. После блока ShiftRows устанавливается результат обработки после 9го такта без учёта операции AddRoundKey с десятым раундовым ключом. Поэтому, подключим к выходу блока ShiftRows дополнительный блок AddRoundKey, на который будет подаваться десятый раундовый ключ. На девятом такте обработки включим выходной регистр reg\_128\_en, который по приходу 10го тактового импульса запишет конечный результат, снятый с выходного блока ядра AddRoundKey. Таким образом, за счёт дополнительного блока AddRoundKey 9ый и 10ый раунды процесса зашифрования получаются в одном последнем такте, и результат на выходе процессора шифрования можно получать каждые 10 тактов.

Как видно по описанию алгоритма работы ядра, в данном процессоре необходимо применять счётчик с 10 различными состояниями: от 0 до 9. При этом необходимо хранить 11 раундовых ключей. Также, 10ый (и последний) раундовый ключ необходимо подавать в ядро одновременно с 9ым раундовым ключом. Эта проблема решена следующим образом.

Блок генерации ключей первые 9 раундов работает в таком же режиме, что и блок генерации ключей из раздела 3.3.1. После обработки девятого раундового ключа, по приходу фронта тактового импульса перед регистром хранения промежуточных ключей устанавливается последний, десятый раундовый ключ. По приходу тактового импульса он запишется в регистре промежуточных ключей. Далее процесс генерации ключей останавливается, и десятый ключ не записывается из данного регистра в память. С этого момента состояние регистра будет неизменным, а значит, 10ый раундовый ключ всегда будет храниться в нём. Подключим выход этого регистра к выходному модулю AddRoundKey ядра зашифрования. Таким образом, в последнем такте шифрования в ядро будут подаваться одновременно 10ый раундовый ключ (с регистра промежуточных ключей) и 9ый ключ (выбираемый счётчиком из памяти KeyScheduleRAM). Также, на один такт сократится процесс генерации ключей.

**3.3.4 11-тактовый процессор зашифрования на архитектуре с**

**использованием Т-таблиц на базе однопортовой ROM-памяти**

Данная реализация также базируется на архитектуре, описанной в разделе 3.3.1. Полная функциональная схема этого процессора представлена в Приложении А на рисунке А.5 (изменённый блок выделен штрихпунктирной линией).

Единственным отличием данной архитектуры является способ реализации функций SubBytes и MixColumns в ядре зашифрования. Согласно разделу «3.1 Анализ возможных архитектурных решений», эти функции реализованы в едином блоке T-Box Module. Реализация части этого блока (для одного столбца блока данных State) показана на рисунке 3.5. Так как операции проводимые функциями SubBytes и MixColumns являются операциями, последовательно проводимыми не над целым блоком в 16 байт, а только над его частью (столбец данных, состоящий из четырёх байт), то получается, что функция T-Box Module состоит из четырёх параллельных модулей, полностью идентичных по структуре, где каждый из модулей принимает на вход только свою группу входных байтов (столбец состояния State). На чертеже как раз и показан один такой модуль.

Итак, каждый из четырёх байтов входного столбца поступает в схеме на собственный блок памяти T-Table. На выходе каждого из модулей памяти, согласно входному байту, выдаётся по 3 байта данных. Далее, согласно формулам 3.22-3.26 эти выходные данные поступают на входы четырёх отдельно стоящих четырёхвходовых модулей побитовой операции XOR. Выход каждого из модулей XOR является соответствующим байтом выходного слова после операций SubBytes и MixColumns.

Однако в десятом раунде обработки входного слова функция MixColumns исключается из пути обработки данных. Тогда, на выход функции необходимо подавать только результат обработки входного блока функцией SubBytes. Результат обработки функцией SubBytes для каждого байта содержится в явном виде в Т-таблицах. Используя четыре выходных мультиплексора, мы можем управлять результатом выходной функции: если текущее показание счётчика равно «10», значит, на выходы подаются непосредственно байты из Т-таблиц, соответствующие только операции SubBytes.

**3.3.5 11-тактовый процессор зашифрования на архитектуре с**

**использованием Т-таблиц на базе синхронной двухпортовой**

**ROM-памяти**

Как было сказано в разделе «3.1 Анализ возможных архитектурных решений», применение двухпортовой памяти влечёт собой необходимый ряд изменений в архитектуре процессора, описанного в разделе 3.3.5.

Существенным отличием при использовании двухпортовой памяти является то, что такая память может изменять выходные данные только по сигналу синхронизации. Это вносит некоторые изменения в общую



Рисунок 3.5 – Функциональная схема части блока T-Box Module в версии с однопортовой памятью, обрабатывающая 4 байта входных данных

архитектуру ядра. Так как выходы синхронной памяти неизменны между сигналами синхронизации, такая память может одновременно служить регистром данных, необходимым для хранения промежуточных данных после обработки предыдущего раунда. Получается, что в выходной регистр данные заносятся только в последнем раунде, а в самом ядре обратная связь реализуется без участия этого выходного регистра, что отличает данную реализацию от архитектур без использования синхронной памяти.

Полная функциональная схема процессора с использованием синхронной двухпортовой представлена в Приложении А на рисунке А.6.

Как видно, в отличие от варианта с асинхронной однопортовой памятью, обратная связь в ядре зашифрования теперь проходит без регистра, напрямую на вход ядра. Функцию регистра хранения промежуточных данных теперь выполняет сама синхронная память, а конечный результат поступает в выходной регистр только в последнем, 11ом такте зашифрования. Небольшому изменению подвергся управляющий автомат – в данной модификации за ненадобностью был исключён сигнал control\_initialized; Также после сигнала control available теперь стоит только один регистр.

Модуль обработки одного столбца входных данных функцией T-Box Module в варианте, исполненном на двухпортовой синхронной памяти, показан в Приложении А на рисунке А.7.

Как видно из рисунка, данный модуль функционально не изменился по сравнению с версией данного модуля для асинхронной памяти. Функционально память работает как 4 независимых блока памяти, хотя физически присутствует всего два модуля памяти с двумя независимыми портами для чтения на каждом. Также напомним, что выходы такой двухпортовой памяти меняют состояние только по приходу фронта тактового импульса, и в течение такта остаются неизменными вне зависимости от изменений входных данных между двумя следующими друг за другом фронтами тактовых импульсов.

**3.3.6 Исходное описание различных модификаций процессоров**

**зашифрования**

Согласно техническому заданию для описания проектируемого процессора зашифрования в данном проекте был выбран язык VHDL.

Исходный текст описаний на языке описания VHDL представлен в Приложении Б.

Головным модулем каждой из модификаций процессора зашифрования является модуль Main.vhd. Для любой из модификаций в модуле Main.vhd приводится полное описание управляющего автомата FSM, а также все взаимосвязи для укрупнённых блоков, как показано на рисунках, описывающих их функциональные схемы. Остальные модули, имеющие такие же названия, как и на указанных выше рисунках, описаны индивидуально.

**3.3.7 Реализация модификаций процессора зашифрования на базе**

**ПЛИС фирмы Xilinx**

Размещение и трассировка всех модификаций проекта согласно техническому заданию проводилась в среде проектирования Xilinx ISE 13.4. Для обеспечения удобства сравнения полученных результатов как между описанными модификациями данного проекта, так и для обеспечения удобства сравнения со сторонними аналогами был выбран кристалл семейства Xilinx Virtex 5 xc5vlx30-3ff676 (показатель скорости -3). Данный кристалл содержит достаточное количество выводов для любой из представленных модификаций, а также содержит достаточно большое количество ресурсов, избыточное для возможности размещения данных проектов. Далее в проекте будет найдена наилучшая модификация, после чего найденная модификация будет размещена и растрассирована также в кристаллах семейств Virtex 6, 7, а также Spartan 6.

Стоит напомнить, что разрабатываемый процессор шифрования не предполагается использовать в качестве отдельного законченного устройства, поэтому также интерес представляет информация не только о количестве занимаемых ресурсов, а также о количестве остающихся ресурсов для реализации других, возможно, гораздо более сложных блоков, требующих предварительного расшифрования вводимых данных.

Отчёты, выданные средой проектирования Xilinx ISE 13.4 представлены в Приложении В. Также, в Приложении В представлены отчёты об энергопотреблении полученных устройств.

В таблице 3.1 приведём краткий отчёт о быстродействии и занимаемых ресурсах по описанным в разделах 3.3.1-3.3.5 процессорам зашифрования.

Отметим, что модификации 3.3.4-3.3.5 были размещены и растрассированы в двух разных версиях: T-Box Module на блочной памяти, а также T-Box Module на распределённой памяти (посредством LUT).

Определим пропускную способность процессоров зашифрования по формуле (3.27), где N = 11 для 11-тактовых модификаций, 10 – для 10-тактовых:

|  |  |
| --- | --- |
|  | (3.27) |

Таблица 3.1 – Отчёт о быстродействии и занимаемых ресурсах по описанным в разделах 3.3.1-3.3.5 процессорам для Virtex 5 xc5vlx30-ff676-3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Версия процессора | Макси-мальная частота, МГц | Слай-сы, штук | Объём BRAM, КБ | Энерго-потреб-ление, мВт | Пропуск-ная способ-ность, Мбит/с |
| 11-тактовый процессор зашифрования с прямой реализацией функции MixColumns (раздел 3.3.1) | 359 | 372 | 0 | 617 | 4189 |
| 10-тактовый процессор зашифрования с прямой реализацией функции MixColumns и отдельным блоком AddRoundKey для обработки нулевого раунда (раздел 3.3.2) | 273 | 524 | 0 | 614 | 3501 |
| 10-тактовый процессор зашифрования с прямой реализацией функции MixColumns и отдельным блоком AddRoundKey для обработки десятого раунда (раздел 3.3.3) | 245 | 365 | 0 | 568 | 3140 |
| 11-тактовый процессор зашифрования на архитектуре с использованием Т-таблиц на базе однопортовой ROM-памяти на блочной памяти (раздел 3.3.4) | 216 | 276 | 324 | 584 | 2520 |

Продолжение таблицы 3.1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Версия процессора | Макси-мальная частота, МГц | Слай-сы, штук | Объём BRAM, КБ | Энерго-потреб-ление, мВт | Пропуск-ная способ-ность, Мбит/с |
| 11-тактовый процессор зашифрования на архитектуре с использованием Т-таблиц на базе однопортовой ROM-памяти на распределённой памяти (раздел 3.3.4) | 336 | 617 | 0 | 716 | 3919 |
| 11-тактовый процессор зашифрования на архитектуре с использованием Т-таблиц на базе синхронной двухпортовой ROM-памяти на блочной памяти (раздел 3.3.5) | 215 | 165 | 324 | 574 | 2512 |
| 11-тактовый процессор зашифрования с использованием Т-таблиц на базе синхронной двухпортовой ROM-памяти на распределённой памяти (раздел 3.3.5) | 222 | 971 | 0 | 811 | 2586 |

**3.3.8 Тестирование и анализ характеристик разработанных**

**модификаций процессора зашифрования**

Сначала подтвердим работоспособность всех описанных модификаций в режиме симуляции в программе ModelSim после размещения и трассировки, приведём описание теста и проверим, выдают ли данные модификации процессора зашифрования ожидаемый результат на частотах, гарантированных в отчётах среды Xilinx ISE.

Временная диаграмма входных тестовых сигналов и ожидаемых выходных сигналов представлена на рисунке 3.6.

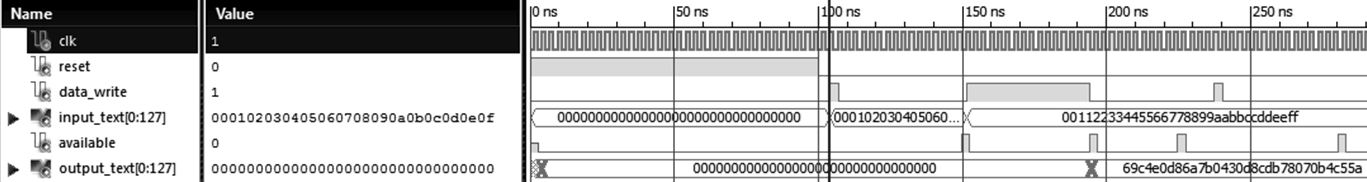


Рисунок 3.6 – Временная диаграмма входных тестовых сигналов и выходных ожидаемых сигналов для процессора зашифрования

Опишем входные сигналы, показанные на рисунке 3.6. Сначала в течение 100 нс в процессор подаётся сигнал RESET, который обеспечивает правильный сброс кристалла FPGA. Далее после сброса на входной шине INPUT\_TEXT устанавливается сигнал, соответствующий ключу, которым будут шифроваться входные данные (тестовый ключ, приведённый в стандарте AES FIPS PUB 197: 0x000102030405060708090a0b0c0d0e0f). Одновременно с ключом шифрования в процессор в течение одного такта поступает активный сигнал DATA\_WRITE, который сообщает процессору, что входные данные (а именно, ключ шифрования) готовы быть приняты во входной регистр в следующем такте. Процессор в свою очередь должен после этого начать генерацию раундовых ключей и по окончании этой процедуры должен выдать сигнал AVAILABLE.

Приняв сигнал AVAILABLE внешнее устройство должно установить на входной шине INPUT\_TEXT новое значение блока данных, подлежащих зашифрованию (тестовый входной блок данных, приведённый в стандарте AES FIPS PUB 197: 0x00112233445566778899aabbccddeeff). Одновременно с установлением входного блока данных внешнее устройство выдаёт сигнал DATA\_WRITE, который позволяет процессору зашифрования записать входной блок данных во внутренний регистр и начать процесс зашифрования. По окончанию процесса зашифрования процессор должен выставит на один такт в активное состояния вывод AVAILABLE, и в этом же такте на выходном регистре процессора должно установиться правильное значение зашифрованного блока (шина OUTPUT\_TEXT): 0x69c4e0d86a7b0430d8cdb78070b4c55a.

Как видно на рисунке 3.6, входной сигнал DATA\_WRITE может быть установлен всё время, пока внешнее устройство требует зашифрования входного блока, подаваемого на входную шину процессора зашифрования (сигнал DATA\_WRITE анализируется процессором только в определённые интервалы времени). Предполагается, что внешнее устройство будет подавать новые ключи каждые 11 тактов (каждые 10 тактов для 10-тактовой версии процессора), чтобы процессор, находясь на последней стадии обработки предыдущего блока данных, мог в этом же такте принять новый блок данный, и уже в следующем такте начать его обрабатывать.

Таким образом, сигнал DATA\_WRITE удерживается на входе процессора время, необходимое для принятия двух одинаковых блоков входных данных, и процессор должен выдать после очередного сигнала AVAILABLE такой же блок выходных данных: 0x69c4e0d86a7b0430d8cdb78070b4c55a. После выдачи нового блока сигнал DATA\_WRITE устанавливается в низкий уровень, после чего процессор должен перейти в состояние ожидания подачи нового блока данных, который будет сопровождаться очередным сигналом DATA\_WRITE.

Через некоторое время пребывания процессора зашифрования в состоянии ожидания снова подаётся сигнал DATA-WRITE, а на входной шине установлен тот же самый входной блок данных 0x00112233445566778899aabbccddeeff. Значит, устройство должно начать процесс зашифрования и в конце этого процесса выдать активный сигнал AVAILABLE = ‘1’ вместе с выходным сигналом шины OUTPUT\_TEXT 0x69c4e0d86a7b0430d8cdb78070b4c55a.

Описание данного теста на языке VHDL представлено в Приложении Б.

Теперь перейдём к приведению результатов тестирования после размещения и трассировки модификаций процессоров зашифрования, описанных в разделах 3.3.1-3.3.5. Соответствующие временные диаграммы приведены на рисунке 3.7 для 11-тактового процессора зашифрования с прямой реализацией функции MixColumns; остальные результаты приведены в Приложении Г на рисунках Г.1-Г.6. Как видно, все модификации успешно проходят тесты, выдавая ожидаемые результаты.

Теперь, когда показана работоспособность данных модификаций на максимальных частотах гарантированных отчётами Xilinx ISE, можно проанализировать данные этих отчётов о быстродействии, критических путях, энергопотреблении, а также сформировать на базе этих данных рекомендации для возможных модификаций.

В первую очередь рассмотрим модификации процессора зашифрования на базе Т-таблиц. В случае использования однопортовой ROM-памяти для Т-таблиц быстрее оказалась модификация на базе распределённой памяти. Увеличение быстродействия относительно модификации с использованием блочной памяти составило 56% при увеличении количества слайсов на 224%.

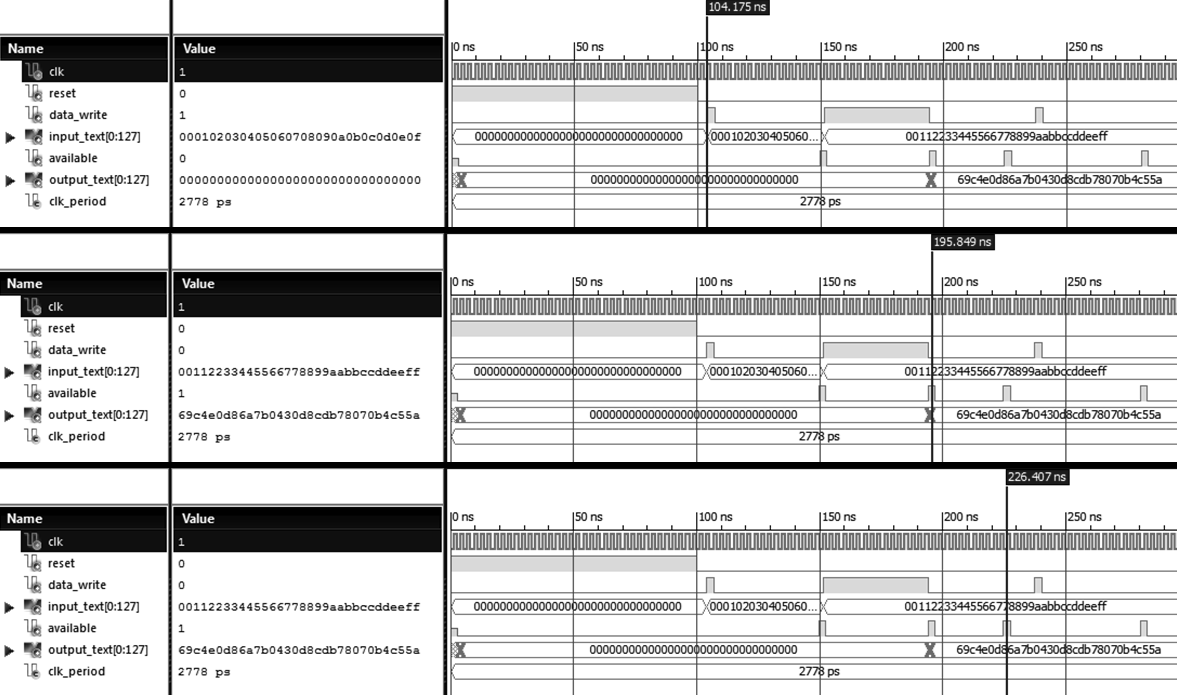


Рисунок 3.7 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns

Двухпортовая модификация на базе распределённой памяти также оказалась быстрее двухпортовой модификации на блочной памяти, однако прирост в быстродействии оказался незначителен, причём данная модификация занимает почти в 6 раз больше ресурсов. С другой стороны двухпортовая модификация занимает всего 165 слайсов, что является рекордным показателем по слайсам для всех приведённых модификаций. Таким образом, при наличии свободной блочной памяти (суммарно блоков на 324 КБ) и достаточного быстродействия такая реализация может быть применена в условиях жёсткой экономии слайсов.

Можно сделать вывод, что самой эффективной модификацией на распределённой памяти является модификация с однопортовой памятью. В случае использования блочной памяти лучшие результаты эффективности показала модификация на базе двухпортовой памяти.

Однако использование Т-таблиц на данной ПЛИС Virtex 5 даже в самой быстрой модификации с использованием однопортовой памяти на распределённой памяти не обеспечило лучших показателей как по абсолютному быстродействию, так и по быстродействию, приходящемуся на один слайс затраченных ресурсов. Судя по отчётам Post-PAR Static Timing Report, для всех модификаций на базе Т-таблиц все максимальные критические пути оказались в самом ядре шифрования, причём большой вклад в эти задержки вносила как раз распределённая память, занимающая несколько уровней. Таким образом, ввиду большого объёма необходимой памяти, память была построена в несколько уровней логики, что значительно замедлило быстродействие всех модификаций на базе Т-таблиц на распределённой памяти, а также существенно увеличило количество занимаемых слайсов.

Обе 10-тактовые модификации (в версиях с дополнительным блоком для нулевого раунда и дополнительным блоком для десятого раунда) также не оказались самыми быстрыми. Не смотря на теоретический прирост в быстродействии за счёт выдачи результата каждые 10 тактов (вместо выдачи результата каждые 11 тактов), общее усложнение схемы привело существенному снижению максимально возможной тактовой частоты, что привело к тому, что итоговое быстродействие существенно снизилось относительно 11-тактовой модификации описанной в пункте 3.3.1.

Из двух 10-тактовых модификаций лучшие результаты по быстродействию показала модификация с дополнительным модулем AddRoundKey для нулевого раунда. Однако прирост в быстродействии на 12% дался ценой большего на 44% количества затраченных ресурсов.

Самой быстрой и самой ресурсоэффективной из представленных модификаций оказалась наиболее простая 11-тактовая модификация с прямой реализацией функции MixColumns, описанная в разделе 3.3.1. При максимальной частоте в 359,971МГц такой процессор зашифрования имеет быстродействие в 4189 Мбит в секунду, при затраченных ресурсах в 372 слайса. Эта модификация процессора не только показала лучшие результаты среди представленных в разделах 3.3.1-3.3.5 модификаций, но и оказалась очень близка по всем параметрам к самой быстрой из всех аналогов версии ядра зашифрования, представленной фирмой Helion Technology [3]. Так, версия процессора зашифрования фирмы Helion Technology имеет максимальную частоту в 363 МГц (быстродействие – 4224 Мбит/с), что на всего 1% быстрее самого быстрого процессора зашифрования, представленного в данном проекте. Версия от Helion Technology занимает 342 слайса, в то время как сделанная в данном проекте аналогичная версия занимает 372 слайса, что на 30 слайсов больше. Однако следует учесть, что на выходе построенного устройства с целью обеспечения удобства тестирования выходных сигналов был поставлен избыточный с точки зрения функционирования ядра выходной регистр на 128 бит. Такой регистр как раз занимает не менее 32 слайсов, и если его не учитывать при подсчёте общего необходимого количества слайсов, то мы выходим на результат лучший или равный результату, полученному инженерами фирмы Helion Technology для аналогичной разработки для кристалла Virtex 5 (-3).

Исследование отчётов о критических путях лучшей модификации процессора зашифрования (11-тактового процессора зашифрования с прямой реализацией функции MixColumns) показало, что самые долгие критические пути содержаться не только в модуле зашифрования, как ожидалось, но и в гораздо более простом модуле генерации раундовых ключей. Также важно отметить, что около 70% от всех задержек на критических путях составила задержка связей межсоединений. Исходя из этого, возникло предположение, что, возможно, пространственное разнесение модуля генерации ключей и модуля зашифрования могло бы сократить максимальные критические пути как в модуле зашифрования, так и в модуле генерации ключей. Одним из решений этой задачи может быть использование памяти раундовых ключей, выдающих раундовые ключи в ядро синхронно, вместо предложенного ранее варианта асинхронной выдачи данных ключей. Таким образом, путь прохождения сигнала от памяти до модуля зашифрования будет разделён регистром на две части, что позволит расположить модуль зашифрования дальше от модуля памяти ключей и модуля генерации ключей при сохранении малой задержки прохождения сигнала от памяти к модулю зашифрования. Это в свою очередь может позволить снизить длину критических путей в самом модуле зашифрования. Построим такую модификацию и посмотрим, не улучшился ли результат.

**3.3.9 11-тактовый процессор зашифрования с прямой реализацией**

**функции MixColumns и синхронной памятью раундовых ключей**

Функциональная схема такого процессора будет внешне повторять схему, показанную на рисунке 3.4. Однако следует понимать, что память для данной модификации должна выдавать результаты синхронно, что влечёт за собой ряд необходимых изменений в последовательности работы управляющего автомата (смотрите рисунок 3.8), а также в модуле зашифрования.

Во-первых, это значит, что счётчик должен считать с опережением на один такт, так как, например, первый раундовый ключ будет выдан в ядро не сразу, а только по приходу очередного фронта тактового импульса.



Рисунок 3.8 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET)

Получается, что уже после регистрации сигнала DATA\_WRITE автомат должен разрешить работу счётчика (в состоянии s\_0 счётчик равен 0, что указывает на выдачу нулевого ключа в следующем такте). Так, в следующем раунде автомат перейдёт в состояние s\_0\_2, а счётчик будет указывать уже на первый раундовый ключ в памяти, хотя в данном такте сам модуль памяти раундовых ключей всё ещё выдаёт нулевой раундовый ключ.

Во-вторых, при таком режиме работы в последнем такте работы, сразу перед выдачей готового результата в выходной регистр модуля зашифрования счётчик будет равен не «10», а «0». Это значит, что для выключения блока MixColumns в последнем раунде мультиплексор MUX\_Sel должен переключаться на линию «ROUND\_NUMBER = 10», когда сигнал счётчика будет равен 0 (вместо 10), так как теперь десятому раунду обработку соответствует текущее значение счётчика, равное нулю.

Теперь можем перейти к описанию и проверке работы данной модификации процессора.

**3.3.10 Исходное описание, размещение и трассировка 11-тактового**

**процессора зашифрования с прямой реализацией функции**

**MixColumns и синхронной памятью раундовых ключей**

Относительно исходной модификации, описанной в разделе 3.3.1, в новое модификации изменились лишь блоки Main и KeyScheduleRAM. Исходное описание новой модификации приведено в Приложении Б.

Как и для ранее описанных модификаций, данная модификация была размещена и растрассирована также на ресурсах FPGA Xilinx Virtex 5 xc5vlx30-3ff676 (показатель скорости -3). Полученные результаты занесены в таблицу 3.2, в эту же таблицу для сравнения занесены данные полученные для исходной модификации процессора, описанного в разделе 3.3.1. В эту же таблицу занесены результаты для новой модификации при использовании блочной памяти для памяти раундовых ключей и для ROM-памяти S-Box.

Результаты тестирования после размещения и трассировки приведены на рисунке 3.9 для версии с распределённой памятью; для версии с блочной памятью результаты тестирования размещены в Приложении Г на рисунке Г.7.

Теперь проанализируем полученные результаты (подробные отчёты Xilinx ISE представлены в Приложении В). Относительно модификации с асинхронной памятью в новой модификации с синхронной памятью на распределённой памяти удалось незначительно поднять частоту до 360,101 МГц, что позволило получить в результате полное быстродействие равное 4190 Мбит/с. Количество ресурсов, занимаемых данных процессором, незначительно увеличилось. Более существенно возросло энергопотребление (на 28%), что объясняется возросшей динамической нагрузкой за счёт добавленных на выход распределённой памяти раундовых ключей регистров.

Таблица 3.2 – Отчёт о быстродействии и занимаемых ресурсах по описанным в разделах 3.3.1 и 3.3.9 процессорам для Virtex 5 xc5vlx30-ff676-3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Версия процессора | Макси-мальная частота, МГц | Слай-сы, штук | Объём BRAM, КБ | Энергопо-требление, мВт | Быстро-действие, Мбит/с |
| 11-тактовый процессор зашифрования с прямой реализацией функции MixColumns (раздел 3.3.1) | 359 | 372 | 0 | 617 | 4189 |
| 11-тактовый процессор зашифрования с прямой реализации функции MixColumns и синхронной памятью раундовых ключей *на распределённой памяти* (раздел 3.3.9 ) | 360 | 386 | 0 | 790 | 4190 |
| 11-тактовый процессор зашифрования с прямой реализации функции MixColumns и синхронной памятью раундовых ключей *на блочной памяти* (раздел 3.3.9) | 219 | 249 | 324 | 540 | 2555 |



Рисунок 3.9 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью раундовых ключей *на распределённой памяти*

Анализ отчёта о критических путях показал результаты, схожие с предыдущей модификацией на базе асинхронной памяти: самые долгие критические пути содержаться не только в модуле зашифрования, но и в модуле генерации раундовых ключей. Также важно отметить, что около 70% от всех задержек на почти всех критических путях составила задержка связей межсоединений. Большая часть критических путей имеет очень близкие задержки, что не даёт возможности существенно повысить быстродействие

Модификация с использованием блочной памяти для памяти раундовых ключей и для ROM-памяти S-Box оказалась существенно медленнее версии с распределённой памятью (на 40%). Однако эта модификация обеспечивает быстродействие, сопоставимое с версиями процессоров зашифрования на базе Т-таблиц, и при этом уступает по количеству занимаемых ресурсов только версии процессора на базе Т-таблиц с использованием блочной двухпортовой памяти.

Таким образом, можно сделать вывод о том, что новая модификация процессора на распределённой памяти не оказалась существенно лучше предыдущей, а по показателю энергопотребления вообще существенно уступает. Однако при разработке процессора шифрования/зашифрования прирост по быстродействию может существенно возрасти ввиду усложнения модуля зашифрования/расшифрования относительно модуля зашифрования, поэтому в качестве заготовок для проектирования процессора зашифрования/расшифрования возьмём обе лучшие модификации: с синхронной памятью и с асинхронной памятью.

Теперь можем перейти к проектированию процессора, реализующего как функцию зашифрования, так и функцию расшифрования.

**3.4 Разработка функциональной схемы и реализация на ПЛИС**

**процессора зашифрования и расшифрования**

**3.4.1 Разработка модуля шифрования и расшифрования**

Сначала покажем, как устроена главная часть процессора зашифрования/расшифрования – модуль зашифрования и расшифрования, представленный сложной комбинационной схемой. Покажем внутреннее строение ядра зашифрования и расшифрования на рисунке 3.10. Отметим, что данный модуль остаётся неизменным как для модификации процессора зашифрования/расшифрования с синхронной памятью, так и для модификации с асинхронной памятью.

Ядро шифрования и расшифрования – главный операционный модуль процессора шифрования. Ядро построено на основе ядра процессора зашифрования, описанного в разделе 3.3.1. Этот модуль работает в двух режимах – зашифрование и расшифрование согласно алгоритмам, показанным в разделе 1. Ввиду того, что алгоритмы зашифрования и расшифрования используют различные функции для обработки, модуль составлен как две параллельные линии (одна для расшифрования, другая для зашифрования), а каждая линия имеет разветвления в виду того, что не все раунды как расшифрования, так и зашифрования функционально-одинаковые. Мультиплексоры MUX\_Sel исключают модули MixColumns и InvMixColumns, если текущий раунд – №10. Мультиплексор MUX\_Input инициализирует подачу входного блока в модуль шифрования. Модуль Mux\_Output обеспечивает выбор нужной линии для снятия промежуточных результатов, а также обеспечивает корректное завершение нулевого раунда в случае, если ENCR\_DECR = 0.

Отметим, что функции MixColumns и InvMixColumns также реализованы по формулам, описанным для этих функций в разделе 3.1 (Т-таблицы не используются). Функции SubBytes и InvSubBytes реализуются на распределённой памяти как 16 параллельных одинаковых ROM-блоков на распределённой памяти.

|  |  |
| --- | --- |
|  | Рисунок 3.10 – Укрупнённая функциональная схема модуля зашифрования и расшифрования |

**3.4.2 Разработка функциональной схемы процессора зашифрования и**

**расшифрования с асинхронной памятью раундовых ключей**

Теперь мы можем приступить к описанию полной схемы. Полная схема процессора зашифрования/расшифрования с асинхронной памятью раундовых ключей представлена в Приложении А на рисунке А.8. Прямоугольник, нарисованный на данной схеме жирной линией, является модулем зашифрования/расшифрования, который подробно показан на рисунке 3.10 (смотрите раздел 3.4.1).

Блок-схема управляющего автомата повторяет блок схему управляющего автомата для аналогичного процессора зашифрования, описанного в разделе 3.3.1, за одним единственным исключением: в состоянии s\_0 (состояние ожидания сигнала DATA\_WRITE) дополнительно введённый управляющий сигнал control\_set\_encr\_decr\_mode устанавливается в единицу, что обозначает разрешение изменения состояния регистра ENCR/DECR (если ENCR/DECR = ‘1’ – режим зашифрования, = ‘0’ – режим расшифрования). Таким образом, процессор может изменять свой режим работы (зашифрование или расшифрование) только в состоянии ожидания s\_0. Полная блок-схема алгоритма работы автомата для нового процессора зашифрования и расшифрования показана в Приложении А на рисунке А.9.

Также в процессор введён новый модуль MUX\_Inv, который отвечает за правильную генерацию адреса для памяти раундовых ключей. Так как в случае режима расшифрования (ENCR/DECR = ‘0’) необходимо обеспечить выдачу раундовых ключей в обратном порядке, устройство MUX\_Inv обеспечивает логическую инверсию адреса («10» вместо «0», «9» вместо «1» и т.д.), если значение входного сигнала ENCR/DECR равно нулю.

Кроме выше указанных изменений, данный процессор зашифрования в остальном полностью повторяет алгоритм работы процессора зашифрования, описанного в разделе 3.3.1.

**3.4.3 Разработка функциональной схемы процессора зашифрования и**

**расшифрования с синхронной памятью раундовых ключей**

Данная модификация процессора расшифрования базируется на версии процессора зашифрования с синхронной памятью раундовых ключей, описанной в разделе 3.3.9. Полная схема новой модификации процессора зашифрования/расшифрования показана на рисунке 3.11. Прямоугольник, нарисованный на данной схеме жирной линией, является модулем зашифрования/расшифрования, который подробно показан на рисунке 3.10 (смотрите раздел 3.4.1).

Для обеспечения нормальной работы в данный процессор по отношению к процессору зашифрования (раздел 3.3.9) были внесены некоторые изменения в управляющий автомат и полную функциональную схему.

Так, по аналогии с процессором, описанным в предыдущем разделе, для правильно выдачи адресов в модуль памяти раундовых ключей в режиме расшифрования в процессор введён модуль MUX\_Inv.

В управляющий автомат были введены новые сигналы control\_enable\_core\_output\_reg (отвечает за разрешение записи в выходной регистр, следующий сразу за ядром зашифрования/расшифрования), а также сигнал control\_set\_encr\_decr\_mode, который, по аналогии с процессором из прошлого раздела, задаёт в состоянии s\_0 автомата режим зашифрования или расшифрования. В данной модификации сигнал автомата control\_initialized выполняет несколько другую функцию: он указывает в состояниях s\_0 – s\_1 о том, что устройство прошло этап получения всех раундовых ключей и разрешает модулю MUX\_Inv осуществлять логическую инверсию адреса для памяти раундовых ключей в режиме расшифрования.

Полная блок-схема алгоритма работы автомата для нового процессора зашифрования и расшифрования показана на рисунке 3.12.

|  |  |
| --- | --- |
|  | Рисунок 3.11 – Укрупнённая функциональная схема процессора зашифрования/расшифрования с  синхронной памятью раундовых ключей |



Рисунок 3.12 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET) для процессора зашифрования/расшифрования с синхронной памятью ключей

**3.4.4 Исходное описание, размещение и трассировка модификаций**

**процессора зашифрования и расшифрования**

Исходное описание полученных модификаций процессоров зашифрования и расшифрования приведено в Приложении Б.

Как и для ранее описанных версий процессора зашифрования, данная модификация была сначала размещена и растрассирована на ресурсах FPGA Xilinx Virtex 5 xc5vlx30-3ff676 (показатель скорости -3). Полученные результаты занесены в таблицу 3.3. Более подробные отчёты приведены в Приложении В.

Таблица 3.3 – Отчёт о быстродействии и занимаемых ресурсах для процессоров зашифрования и расшифрования на Virtex 5 xc5vlx30-ff676-3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Макси-мальная частота, МГц | Слайсы, штук | Объём BRAM, КБ | Энергопо-требление, мВт | Пропускная способ-ность, Мбит/с |
| Процессор зашифрования и расшифрования с асинхронной памятью ключей (раздел 3.4.2) | 221 | 712 | 0 | 715 | 2573 |
| Процессор зашифрования и расшифрования с синхронной памятью ключей (раздел 3.4.3) | 258 | 778 | 0 | 890 | 3001 |

Временная диаграмма входных тестовых сигналов и ожидаемых выходных сигналов представлена на рисунке 3.13.

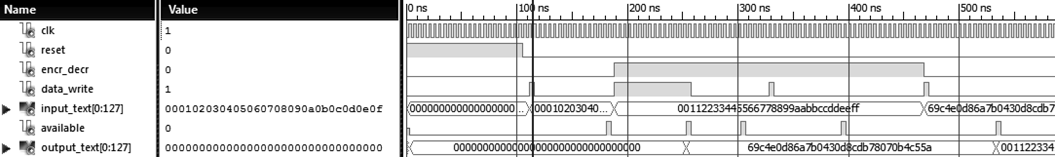


Рисунок 3.13 – Временная диаграмма входных тестовых сигналов и выходных ожидаемых сигналов для процессора зашифрования и расшифрования

Временная диаграмма входных тестовых сигналов мало отличается от диаграммы, описанной в разделе 3.3.8 для тестирования процессоров зашифрования.

Так, вся первая часть данной временной диаграммы осталась прежней, однако вместе с подачей на вход процессора нового блока для последующей обработки на входе устройства устанавливается сигнал ENCR/DECR = ‘1’, что обозначает задание режима зашифрования для этого блока данных. На всём временном отрезке, пока сигнал ENCR/DECR = ‘1’ выходные сигналы тестируемого процессора должны повторять выходные сигналы, описанные на рисунке 3.6.

Для тестирования режима расшифрования сразу после перехода сигнала ENCR/DECR в ‘0’ сигнал DATA\_WRITE становится активным на один такт, а на входной шине устанавливается блок данных 0x69c4e0d86a7b0430d8cdb78070b4c55a. Тестируемый процессор должен установиться в режим расшифрования и вскоре выдать сигнал AVAILABALE = ‘1’, сопровождаемый расшифрованным блоком данных (0x00112233445566778899aabbccddeeff).

Описание данного теста на языке VHDL представлено в Приложении Б.

Успешные результаты тестирования после размещения и трассировки приведены на рисунке 3.14 для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей; диаграмма для процессора зашифрования и расшифрования с асинхронной памятью раундовых ключей представлена в Приложении Г на рисунке Г.8.

**3.4.5 Анализ характеристик разработанных модификаций процессора**

**зашифрования и расшифрования**

Как видно из таблицы 3.3, большей производительностью обладает процессор зашифрования и расшифрования в версии с синхронной памятью, причём прирост в быстродействии составил целых 17%. В то же время аналогичные версии процессоров зашифрования отличались менее чем на 1% по показателю быстродействия. Аналогично, представленная версия процессора зашифрования/расшифрования с синхронной памятью при максимально возможном быстродействии занимает несколько больше ресурсов (на 9%), а также имеет на 24% большее энергопотребление. Таким образом, предположение о возможности ускорения процессора за счёт перехода к синхронной памяти ключей оказалось верным.

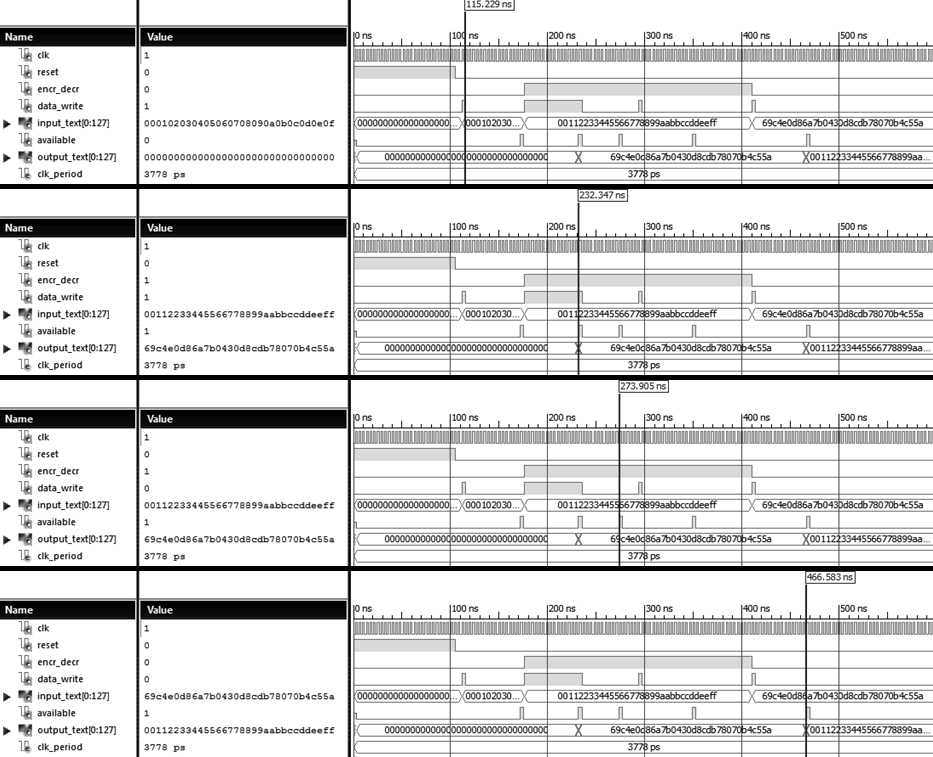
****

Рисунок 3.14 – Временная диаграмма для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей

Анализ критических путей обеих версий процессора зашифрования и расшифрования показал похожие результаты: большая часть критических задержек приходится на пути, проходящие через модуль зашифрования и расшифрования, причём более 70% задержек приходится на задержки линий связи. В то же время для версии с синхронной памятью ключей задержки внутри модуля шифрования существенно меньше.

Согласно техническому заданию, требуется получить данные о характеристиках полученных процессоров не только на Virtex 5, но и на Virtex 6, 7 и Spartan 6. Разместим и растрассируем на указанных кристаллах ПЛИС процессоры зашифрования и зашифрования/расшифрования с синхронной памятью и проанализируем полученные результаты с максимальным быстродействием (смотрите таблицу 3.4, а также отчёты Xilinx ISE в Приложении В). При полученных результатах работоспособность была проверена при помощи тестов, приведённых на рисунках 3.6 и 3.13, и результатом данных тестов стали диаграммы, повторяющие диаграммы 3.7 и 3.14 (смотрите рисунки Г.9-Г.14 Приложения Г).

Для кристалла Virtex 5 снижение быстродействия процессора зашифрования/расшифрования относительно процессора зашифрования составило 28%, причём число занимаемых ресурсов при этом возросло в 2 раза.

Для кристалла Virtex 6 снижение быстродействия процессора зашифрования/расшифрования относительно процессора зашифрования составило 31%, причём число занимаемых ресурсов при этом возросло только в 1,6 раза. Однако важно отметить, что версия зашифрования для Virtex 6 занимает на 12% больше слайсов, чем аналогичная версия для Virtex 5. Как видно, даже при наличии у семейств Virtex 4 и 6 почти одинаковой структуры слайсов программы размещения и трассировки для этих семейств дают несколько разные результаты для одного и того же VHDL-описания при подобных настройках.

Для кристалла Virtex 7 снижение быстродействия процессора зашифрования/расшифрования относительно процессора зашифрования составило 28%, причём число занимаемых ресурсов при этом возросло только в 1,36 раза. Также нужно отметить, что для варианта с максимальным полученным быстродействием как для версии процессора зашифрования, так для процессора зашифрования/расшифрования занимают существенно больше слайсов, чем для кристаллов Virtex 5 и 6.

Для кристалла Spartan 6 быстродействие процессора шифрования/зашифрования составило 67% от быстродействия подобного процессора зашифрования при возросшем числе занимаемых слайсов на 46%. При этом по занимаемым ресурсам версии для Spartan 6 сопоставимы с аналогичными результатами, полученными для Virtex 6. Это объясняется родством данных семейств, а значит и родством средств размещения и трассировки для них.

Энергопотребление для кристалла Spartan 6 оказалось существенно меньшим, чем для других представленных кристаллов ПЛИС. Энергопотребление для кристалла Virtex 6 наоборот оказалось значительно более высоким, чем даже для Virtex 7.

Сравним полученные результаты для процессора зашифрования с синхронной памятью ключей с сопоставимыми разработками фирмы Helion Technology [3]. Полученные характеристики для сравнения приведём в таблице 3.5.

Таблица 3.4 – Отчёт о быстродействии и занимаемых ресурсах для процессоров зашифрования/расшифрования с синхронной памятью ключей

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Тип процессора | Кристалл ПЛИС | Макси-маль-ная  частота, МГц | Слайсы, штук | Энерго-потре-бление,  мВт | Пропус-кная способ-ность, Мбит/с |
| Процессор зашифрования | Virtex 5 (-3)  xc5vlx30-ff676-3 | 360 | 386 | 790 | 4190 |
| Virtex 6 (-3)  xc6vlx75t-ff784-3 | 425 | 432 | 1780 | 4941 |
| Virtex 7 (-3)  xc7vx330t-ffg1157-3 | 445 | 628 | 371 | 5176 |
| Spartan 6 (-2)  xc6slx45-fgg676-2 | 223 | 447 | 171 | 2594 |
| Процессор зашифрования и расшифрования | Virtex 5 (-3)  xc5vlx30-ff676-3 | 258 | 778 | 890 | 3001 |
| Virtex 6 (-3)  xc6vlx75t-ff784-3 | 292 | 694 | 1865 | 3395 |
| Virtex 7 (-3)  xc7vx330t-ffg1157-3 | 318 | 854 | 455 | 3706 |
| Spartan 6 (-2)  xc6slx45-fgg676-2 | 150 | 652 | 225 | 1745 |

Как было написано ранее в записке, для кристалла Virtex 5 (3) разработка, представленная в данном проекте очень близка по всем характеристикам к аналогичной разработке фирмы Helion Technology (менее 1% отставания представленной разработки от разработки фирмы Helion Technology). Несколько большее (на 44 слайса) количество ресурсов представленной в данном проекте разработки процессора зашифрования объясняется наличием у последнего дополнительного выходного 128-разрядного регистра, введённого в тестовых целях.

Для кристалла Virtex 6 результаты получились несколько хуже. При большем количестве занимаемых слайсов представленный в данном проекте процессор зашифрования оказался медленнее на 6%.

Для кристалла Spartan 6 характеристики разработанного процессора зашифрования оказались существенно выше по параметру быстродействия (на 38%). При этом полученный процессор занимает, за вычетом ресурсов для дополнительного выходного регистра, примерно на 23% больше слайсов.

Таблица 3.5 – Сравнение характеристик процессора зашифрования с синхронной памятью ключей и аналогичной разработкой фирмы Helion Technology для различных кристаллов ПЛИС

|  |  |  |  |
| --- | --- | --- | --- |
| Версия процессора | Spartan 6 (-2) | Virtex 5 (-3) | Virtex 6 (-3) |
| Процессор зашифрования с синхронной памятью ключей (раздел 3.3.9) | 447 слайсов,  223 МГц,  2594 Мбит/с | 386 слайсов,  360 МГц,  4190 Мбит/с | 432 слайса,  425 МГц,  4941 Мбит/с |
| Процессор зашифрования фирмы Helion Technology [3] | 332 слайса,  162 МГц,  1885 Мбит/с | 342 слайса,  363 МГц,  4224Мбит/с | 331 слайс,  450 МГц,  5236 Мбит/с |

Таким образом, в данном проекте не удалось превзойти аналогичную разработку фирмы Helion Technology [3] по всем параметрам, однако удалось получить очень близкие результаты для кристалла Virtex 5, сопоставимые результаты для кристалла Virtex 6 и существенно лучшие по показателю быстродействия результаты для кристалла Spartan 6.

В техническом задании к данному проекту указано, что требуется построить процессор зашифрования и расшифрования с максимально возможным быстродействием. В качестве такого процессора был выбран разработанный процессор зашифрования и расшифрования с синхронной памятью ключей, описанный в разделе 3.4.3. Структурная схема данного процессора, повторяющая по смыслу рисунок 3.2, представлена на чертеже ГУИР.431282.054 Э1. Функциональная схема данного процессора, повторяющая по смыслу рисунок 3.27, представлена на чертеже ГУИР.431282.054 Э2. Схема алгоритма работы данного процессора представлена на чертеже ГУИР.431282.054 ПД. Результаты проектирования также показаны на плакате ГУИР.431282.054 ПЛ.1, а результаты тестирования показаны на плакате ГУИР.431282.054 ПЛ.2.

**4 ТЕХНИКО-ЭКОНОМИЧЕСКОЕ ОБОСНОВАНИЕ ПРОЕКТА**

**4.1 Краткая характеристика проведенной работы**

В ходе проектирования процессора шифрования алгоритма AES для ПЛИС было разработано несколько альтернативных архитектур, из которых была выбрана наилучшая по критерию производительности. Полученный процессор шифрования способен осуществлять операции зашифрования и расшифрования с размером ключа 128 бит с максимальным быстродействием свыше 3 гигабит в секунду для кристаллов семейств Xilinx Virtex 5, 6, 7. Для полученных архитектур согласно техническому заданию было получено исходное описание на языке VHDL.

Полученный модуль шифрования может быть использован, в том числе, в следующих областях цифровой техники:

– переносные накопители данных;

– цифровое теле- и радиовещание;

– беспроводная связь (Wi-Fi, сотовая связь);

– технологии передачи данных (локальные сети, модемы).

Поскольку разработанный модуль шифрования не предназначен для использования в качестве законченного устройства, а рассчитан на применения в сложных цифровых устройствах в качестве преобразователя входных и выходных данных, то не известно в какой конкретной технической системе данный модуль будет применяться. Значит, выполним расчёт только себестоимости и отпускной цены программного обеспечения (ПО), разработанного модуля шифрования.

**4.2 Расчёт объёма и трудоёмкости программного обеспечения**

Общий объём (VО) программного продукта определяем путём подсчёта количества строк исходного кода (Lines of Code, LOC) в полученном исходном описании модуля шифрования алгоритма AES. В качестве строки исходного кода принимались строки, содержащие один оператор языка (логические строки), строки, содержащие объявления сигналов, шин, регистров, а также строки, в которых объявляются экземпляры модулей. Строки, содержащие комментарии и пустые строки не учитывались при подсчёте числа строк исходного кода описания проекта.

В результате анализа всех файлов описания проекта общая сумма строк кода составила VО = 2262 LOC. Данная величина принимается в качестве уточнённого объёма Vу = 2262 LOC.

Определим категорию сложности разработки ПО. Разработанное описание модуля шифрования обладает следующими характеристиками: обеспечение существенного распараллеливания вычислений, режим работы в реальном времени, обеспечение переносимости. По содержанию описанных характеристик, отнесём разработанное ПО к первой категории.

По уточнённому объёму ПО Vу и категории сложности [6] определяются нормативная и общая трудоёмкость ПО. Нормативная трудоёмкость составляет Тн = 73 чел./дн.

Так как разработка описания модуля шифрования – сложный проект с длительным временем разработки, то условно разделим этот период на следующие этапы:

– техническое задание (ТЗ);

– эскизный проект (ЭП);

– технический проект (ТП);

– рабочий проект (РП).

Нормативная трудоёмкость рассчитывает для каждой стадии Тн*i* и вычисляется по формуле (4.1):

|  |  |
| --- | --- |
|  | (4.1) |

где *d*ст*i* – удельный вес трудоёмкости i-й стадии разработки ПО в общей

трудоёмкости ПО. Удельные веса трудоёмкости *d*стi приведены в

таблице 4.1.

Трудоёмкость каждой стадии определяется на основе нормативной трудоёмкости с учётом новизны, степени использования стандартных модулей ПО и удельного веса трудоёмкости каждой стадии в общей трудоёмкости ПО по выражению (4.2):

|  |  |
| --- | --- |
|  | (4.2) |

где – уточнённая трудоёмкость разработки ПО на i-ой стадии;

– коэффициент, учитывающий сложность ПО, вводится на всех

стадиях;

– поправочный коэффициент, учитывающий степень использования

стандартных модулей, принимаем равным единице, вводится только

на стадии рабочего проекта;

– коэффициент, учитывающий степень новизны ПО, вводится на всех

стадиях (принимаем равным 0,9 для категории Б).

Коэффициент сложности ПО равен единице с прибавленным значением, определяемым наличием ряда дополнительных характеристик [6], усложняющих проект. Рассматриваемый проект, как уже было определено, обладает тремя такими характеристиками, поэтому добавочное значение составит 0,18, и, в итоге, значение коэффициента сложности будет равно 1,18.

Определим по формуле (4.1) значения трудоёмкости каждой стадии разработки:

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

Вычислим значения общей трудоёмкости для каждой стадии:

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

Общая трудоёмкость рассчитывается с учётом распределения трудоёмкости по стадиям и рассчитывается по формуле (4.3):

|  |  |
| --- | --- |
|  | (4.3) |

где *n* – количество стадий разработки.

Подставив в формулу (4.3) полученные значения общей трудоёмкости для каждой из стадий получим:

|  |  |
| --- | --- |
|  |  |

Ввиду того, что срок разработки ПО Тр оговорен техническим заданием на дипломный проект и составляет четыре месяца или 0,33 года, то необходимо определить число исполнителей проекта по формуле (4.4):

|  |  |
| --- | --- |
|  | (4.4) |

где – эффективный фонд рабочего времени одного работника в течение

года, который составляет 236 дней [6].

Из (4.2) получим:

|  |  |
| --- | --- |
|  |  |

Итоговую численность исполнителей Чр принимаем равной 1 человеку.

Сроки разработки каждой стадии находятся по формуле (4.5):

|  |  |
| --- | --- |
|  | (4.5) |

Вычисляем сроки разработки для каждой из стадий:

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

|  |  |
| --- | --- |
|  |  |

Результаты расчётов трудоёмкости разработки ПО, числа исполнителей и сроков разработки приведены в таблице 4.1.

Таблица 4.1 – Результаты расчётов трудоёмкости разработки ПО, численности исполнителей и сроков разработки по стадиям

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Показатели | Стадии | | | | Итого |
| ТЗ | ЭП | ТП | РП |
| Коэффициенты удельных весов трудоёмкости стадии разработки ПО (*d*ст) | 0,09 | 0,07 | 0,07 | 0,77 | 1,0 |
| Распределение нормативной трудоёмкости ПО (Тн) по стадиям, чел./дн. | 6,57 | 5,11 | 5,11 | 56,21 | 73 |
| Коэффициент сложности ПО (Кс) | 1,18 | 1,18 | 1,18 | 1,18 | - |
| Коэффициент, учитывающий использования стандартных модулей (Кт) | - | - | - | 1 | - |
| Коэффициент, учитывающий новизну ПО (Кн) | 0,9 | 0,9 | 0,9 | 0,9 | - |
| Общая трудоёмкость ПО (Ту), чел./дн. | 6,98 | 5,43 | 5,43 | 59,70 | 77,53 |
| Численность исполнителей (Чрi), чел | 1 | 1 | 1 | 1 | 1 |
| Сроки разработки, лет | 0,03 | 0,023 | 0,023 | 0,254 | 0,33 |

**4.3 Расчёт себестоимости и отпускной цены**

Основной статьёй расходов на создание ПО является заработная плата разработчиков (исполнителей) проекта. Оплата труда осуществляется на основе Единой тарифной сетки (ЕТС) Республики Беларусь, в которой приведены тарифные разряды и тарифные коэффициенты.

Месячная тарифная ставка Тм каждого исполнителя определяется по формуле (4.6):

|  |  |
| --- | --- |
|  | (4.6) |

где – месячная тарифная ставка 1-го разряда, составляющая на данный

момент 200 тыс. белорусских рублей;

– тарифный коэффициент, соответствующий установленному

тарифному разряду.

В расчётах заработной платы исполнителей примем значение тарифного разряда равного 10, что соответствует категории специалиста с высшим образованием. Для принятого тарифного разряда значение тарифного коэффициента Тк = 2,48. Подставив принятые значения в формулу (4.6) получим:

|  |  |
| --- | --- |
|  |  |

Часовая тарифная ставка рассчитывается по формуле (4.7):

|  |  |
| --- | --- |
|  | (4.7) |

где – среднемесячная норма рабочего времени при 4-часовой рабочей

неделе (в часах).

Принимая в качестве величины среднемесячной нормы рабочего времени = 170 часов, получим:

|  |  |
| --- | --- |
|  |  |

Основная заработная плата исполнителя рассчитывается по формуле (4.8):

|  |  |
| --- | --- |
|  | (4.8) |

где *n* – количество исполнителей, занятых разработкой ПО;

– часовая тарифная ставка i-го исполнителя, руб.;

– количество работы в день, часы;

– плановый фонд рабочего времени i-го исполнителя, дни;

К – коэффициент премирования.

В данном проекте число исполнителей *n* равно одному человеку, плановый фонд рабочего времени Фп = 67 дней, коэффициент премирования принимается равным К = 1,3. Отсюда имеем:

|  |  |
| --- | --- |
|  |  |

Дополнительная заработная плата рассчитывается по формуле (4.9):

|  |  |
| --- | --- |
|  | (4.9) |

где Нд – норматив дополнительной заработной платы, принимается

равным 10%.

Подставляя полученные значения находим:

|  |  |
| --- | --- |
|  |  |

Отчисления в фонд социальной защиты населения и обязательного медицинского страхования рассчитываются по формуле (4.10):

|  |  |
| --- | --- |
|  | (4.10) |

где – норматив отчислений в фонд социальной защиты и обязательного

медицинского страхования, принимается равным 35%.

Тогда из (4.10) находим:

|  |  |
| --- | --- |
|  |  |

Расходы по статье «Материалы» определяются по формуле (4.11):

|  |  |
| --- | --- |
|  | (4.11) |

где – норма расходов на материалы, принимается равной 3%.

Тогда по формуле (4.11) находим:

|  |  |
| --- | --- |
|  |  |

Расходы по статье «Машинное время» вычисляются по формуле (4.12):

|  |  |
| --- | --- |
|  | (4.12) |

где – себестоимость одного машино-часа, руб.;

– норматив расхода машинного времени на отладку 100 строк

исходного кода, машино-часы (согласно [6] примем равным 3,6

с учётом применения ПО для отладки).

В качестве цены одного машино-часа Цм примем величину стоимости одного киловатт-часа для непромышленных бюджетных организаций 770,6 рублей. Отсюда находим:

|  |  |
| --- | --- |
|  |  |

Расходы по статье «Научные командировки» рассчитываются по формуле (4.13):

|  |  |
| --- | --- |
|  | (4.13) |

где – норматив расходов на командировки, принимаем равным 30%.

Тогда по выражению (4.13) получим:

|  |  |
| --- | --- |
|  |  |

Расходы по статье «Прочие затраты» вычисляются по формуле (4.14):

|  |  |
| --- | --- |
|  | (4.14) |

где – норматив прочих затрат, принимаем равным 20%.

Тогда получим:

|  |  |
| --- | --- |
|  |  |

Расходы по статье «Накладные расходы» вычисляются по формуле (4.15):

|  |  |
| --- | --- |
|  | (4.15) |

где – норматив накладных расходов, принимаем равным 100%.

Тогда получим:

|  |  |
| --- | --- |
|  |  |

Полная себестоимость разработки ПО вычисляется по формуле (4.16):

|  |  |
| --- | --- |
|  | (4.16) |

Подставляя рассчитанные ранее значения находим:

|  |  |
| --- | --- |
|  |  |

Прогнозируемая прибыль от реализации ПО вычисляется по формуле (4.17):

|  |  |
| --- | --- |
|  | (4.17) |

где – уровень рентабельности ПО, принимаем равным 15%.

Тогда получим:

|  |  |
| --- | --- |
|  |  |

Прогнозируемая цена ПО без учёта налогов вычисляется по формуле (4.18):

|  |  |
| --- | --- |
|  | (4.18) |

Тогда, подставляя полученные значения, получим:

|  |  |
| --- | --- |
|  |  |

Рассчитаем налог на добавленную стоимость по формуле (4.19):

|  |  |
| --- | --- |
|  | (4.19) |

где – норматив НДС, принимаем равным 20%.

Тогда, подставляя полученные значения, получим:

|  |  |
| --- | --- |
|  |  |

Прогнозируемая отпускная цена ПО модуля процессора алгоритма AES вычисляется по формуле (4.20):

|  |  |
| --- | --- |
|  | (4.20) |

Отсюда находим:

|  |  |
| --- | --- |
|  | (4.20) |

Таким образом, в результате расчёта отпускной цены разработки ПО разработанного модуля процессора шифрования алгоритма AES для систем обработки данных составила 8545 тыс. руб. при заложенной рентабельности в 15%.

Результаты расчёта статей затрат при разработке ПО, а также расчёт его отпускной цены приведён в таблице 4.2.

Таблица 4.2 – Расчёт основных статей затрат при разработке ПО

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Наименование статей | Услов-ное обозна-чение | Норматив | Расчётная формула | Сумма, тыс. руб. |
| Основная заработная плата исполнителей |  | - |  | 2033,016 |
| Дополнительная заработная плата исполнителей |  | Нд =10% |  | 203,302 |
| Отчисления в фонд социальной защиты населения и обязательного медицинского страхования |  | Нсз =35% |  | 782,711 |
| Материалы и комплектующие | М | Нм = 3% |  | 60,990 |
| Машинное время |  | Нмв = 3,6 |  | 62,751 |
| Расходы на научные командировки |  | Нрнк =30% |  | 609,905 |
| Прочие прямые расходы |  | Нпз = 20% |  | 406,603 |
| Накладные расходы |  | Ннр=100% |  | 2033,016 |
| Полная себестоимость |  | - |  | 6192,294 |
| Прогнозируемая прибыль |  | Урп =15% |  | 928,844 |
| Прогнозируемая цена без налогов |  | - |  | 7121,138 |
| НДС (при отсутствии льгот) |  | Ндс =20% |  | 1424,228 |
| Прогнозируемая отпускная цена |  | - |  | 8545,366 |

**5 ОХРАНА ТРУДА**

**5.1 Проектирование и расчёт искусственного освещения при разработке**

**электронных систем**

В данном проекте проводится разработка процессора алгоритма шифрования AES на базе FPGA. Такая разработка включает работу с ЭВМ в средах автоматизированного проектирования, работу с бумажными документами и чертежами, а также, возможно, работу с отладочными электронными устройствами. В таких условиях разработчик подвергается воздействию многих неблагоприятных факторов, одними из самых важных из которых являются отсутствие или недостаток естественного света, недостаточность освещения рабочей зоны.

Проведём расчёт общего равномерного искусственного освещения для рабочего кабинета разработчика электронных систем при условии отсутствия естественного освещения. Работы такого типа можно отнести к работам средней точности разряда IV б.

Согласно методике, предложенной в [7], рассчитаем равномерное искусственное освещение для офисного помещения (рабочего кабинета). План производственного (офисного) помещения показан на рисунке 5.1. Длина помещения – 5,56 метра, ширина помещения – 3,37 метра, высота – 2,9 метра. Содержание в воздушной среде рабочей зоны пыли, дыма и копоти составляет менее 1 мг/м3 (нормальные воздушные условия). Коэффициенты отражения от потолка (белая краска) – 70%; от стен (краска кремового цвета) – 50%, от рабочей поверхности (коричневого цвета) – 30%.



Рисунок 5.1 – План производственного помещения

Для помещения с нормальными воздушными условиями выберем светильник типа ЛПО46, подходящий для общего освещения в производственных и офисных помещениях с нормальными условиями. Длина светильника равна 1235 мм, тип крепления – потолочный.

Определим расчётную высоту подвеса светильника hп в соответствии с рисунком 5.2 по формуле (5.1):

|  |  |
| --- | --- |
|  | (5.1) |
|  |  |

где *H* – высота помещения, м;

*h*п – высота освещаемой поверхности (0,8 м);

*h*с – высота свеса светильника от потолка (для потолочных светильников

равно нулю).

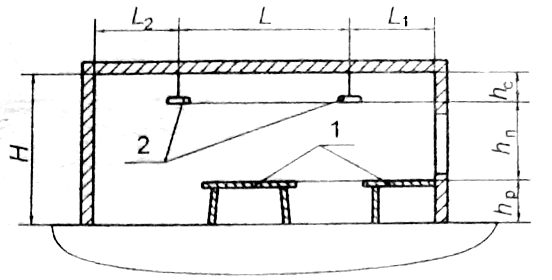


Рисунок 5.2 – Разрез помещения:

1 – рабочие поверхности; 2 – светильники

Тогда получим:

|  |  |
| --- | --- |
|  |  |
|  |  |

Определим оптимальные расстояния между рядами светильников L по формуле (5.2):

|  |  |
| --- | --- |
|  | (5.2) |

где λ – оптимальное отношение расстояний между светильниками ЛПО46 к

высоте их подвеса согласно [7] примем равным 1,6.

Тогда получим:

|  |  |
| --- | --- |
|  |  |
|  |  |

Рассчитаем расстояния от стен до крайних рядов L1 (с учётом того, что рабочие места находятся у стен) по формуле (5.3):

|  |  |
| --- | --- |
|  | (5.3) |

Рассчитаем расстояния от стен до крайних рядов L2 (с учётом того, что проходы расположены у стен) по формуле (5.4):

|  |  |
| --- | --- |
|  | (5.4) |

Тогда получим:

|  |  |
| --- | --- |
|  |  |
|  |  |

Оценим, сколько рядов можно разместить в помещении по формуле (5.5):

|  |  |
| --- | --- |
|  | (5.5) |

где *b* – ширина помещения, м;

*np* – количество рядов светильников в помещении.

Тогда для помещения шириной в 3,37 метра получим наибольшее число рядов из выражения (подставляя L1):

|  |  |
| --- | --- |
|  |  |

Таким образом, светильники располагаем в один ряд, рабочие места у стен.

Определим количество светильников в ряду по формуле (5.6):

|  |  |
| --- | --- |
|  | (5.6) |

где *nсв* – количество светильников в ряду;

*a* – длина помещения, м;

*Lсв* – длина светильника, м.

Тогда для помещения длиной в 5,56 метра получим:

|  |  |
| --- | --- |
|  |  |

Таким образом, в одном ряду можно разместить 3 светильника.

По формуле (5.7) рассчитаем индекс помещения:

|  |  |
| --- | --- |
|  | (5.7) |

где *i* – индекс освещения;

*a* и *b* – длина и ширина помещения, м.

Тогда для данного помещения получим:

|  |  |
| --- | --- |
|  |  |

Согласно [7], данному индексу помещения, равному 0,999, при заданных цветах поверхности стен, потолка и цвете рабочей поверхности соответствует коэффициент использования светового потока лампы η, равный 50%.

Рассчитаем световой поток лампы. Для этого определим нормативную минимальную освещённость. Из [7] находим нормируемую минимальную освещённость *E*min, для работ средней точности класса IV б равную 200 лк.

Световой поток получим из формулы (5.8):

|  |  |
| --- | --- |
|  | (5.8) |

где *F* – световой поток одной лампы, лм;

*E*min – нормативная минимальная освещённость, лк;

*S* – освещаемая площадь помещения, м2;

*N* – число светильников на плане помещения;

*n* – число ламп в светильнике (2 лампы для ЛПО46);

*η* – коэффициент использования светового потока (в долях);

*Kз* – коэффициент запаса, для заданных условий среды согласно [7]

принимаем равным 1,5;

*Z* – коэффициент неравномерности минимальной освещённости

(согласно [7], для люминесцентных ламп принимаем равным 1,1).

Тогда для заданного помещения получим:

|  |  |
| --- | --- |
|  |  |

По полученному в результате расчёта требуемому световому потоку выбираем ближайшую стандартную лампу. Для заданного светильника ЛПО46 выберем лампу ЛД 36-7, дающую световой поток 2300 лм и имеющую длину 1213,6 мм. Однако сначала проверим, достаточное ли было выбрано количество ламп, если каждая лампа имеет световой поток *F*станд, равный 2300 лм. По формуле (5.9) проверим отклонение Δ светового потока лампы (допускается отклонение светового потока на -10%...+20%):

|  |  |
| --- | --- |
|  | (5.9) |

Подставив в (5.7) значения *F*станд и *F* получим:

|  |  |
| --- | --- |
|  |  |

Полученное значение отклонения Δ составило +11,6% и не выходит за рамки допустимых пределов.

Для большей равномерности искусственного освещения с учётом того, что светильники располагаются в один ряд, отдалим ряд светильников от стены на максимальное расстояние по формуле (5.3): Покажем размещение светильников для обеспечения равномерного общего освещения для работ по разработке электронных систем на плане помещения на рисунке 5.3.



Рисунок 5.3 – Размещение светильников на плане помещения

Таким образом, был проведён расчёт производственного искусственного освещения (в условиях отсутствия естественного освещения), подходящего для работ по разработке электронных систем. Достаточное число светильников ЛПО46 для обеспечения нормального освещения в данном офисном помещении равно трём, причём каждый из светильников имеет по две совместимые с таким светильником лампы типа ЛД 36-7 (дневного света), дающие световой поток по 2300 лм. При этом предполагается, что рабочие места должны быть расположены у стен.

**ЗАКЛЮЧЕНИЕ**

В дипломном проекте был разработан процессор зашифрования и расшифрования алгоритма AES для различных ПЛИС фирмы Xilinx.

Изначально были спроектированы несколько модификаций процессора, осуществляющего только режим зашифрования. Так, были получены модификации 11-тактового процессора зашифрования, 10-тактового процессора зашифрования, процессора зашифрования на базе Т-таблиц, 11-тактового процессора зашифрования с синхронной памятью раундовых ключей, а также модификации данных процессоров с использованием блочной памяти. По результатам процедуры размещения и трассировки на кристалле Virtex 5 была найдены две самые быстрые и эффективные по соотношению «производительность на затраченные ресурсы» версии процессора зашифрования: 11-тактовая модификация процессора и 11-тактовая модификация процессора зашифрования с синхронной памятью ключей.

На базе двух этих модификаций были разработаны две модификации процессора, осуществляющего как процедуру зашифрования, так и процедуру расшифрования. По результатам размещения и трассировки на кристалле Virtex 5 было установлено, что наибольшей производительностью и эффективностью по соотношению «производительность на затраченные ресурсы» обладает версия процессора зашифрования и расшифрования на базе 11-тактового процессора зашифрования с синхронной памятью ключей. Эта модификация и была выбрана в качестве результата проектирования процессора зашифрования и расшифрования с максимальной производительностью при минимальных ресурсах.

Для лучших по показателю быстродействия процессоров зашифрования и зашифрования/расшифрования была проведена процедура размещения и трассировки для кристаллов Xilinx Virtex 5, 6 и 7, а также для Spartan 6. Лучший полученный процессор зашифрования имеет пропускную способность свыше 4 гигабит в секунду, а лучший разработанный процессор зашифрования и расшифрования имеет пропускную способность свыше 3 гигабит в секунду для кристаллов Virtex 5,6,7. Полученная пропускная способность позволяет данным разработкам получить применение в быстродействующих системах передачи и хранения данных. Полученные характеристики сравнимы с характеристиками аналогичных разработок.

Также можно заключить, что все представленные в проекте модификации процессоров могут представлять определённый интерес в зависимости от специфики конкретного приложения.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Advanced Encryption Standard (AES) (FIPS PUB 197) [Электронный ресурс] : Federal Information Processing Standard / National Institute of Standards and Technology. – Электронные данные. – Режим доступа : http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[2] Rijndael Flash Animation [Электронный ресурс] : Adobe Flash File / Boston College, Computer Science Department. – Электронные данные. – Режим доступа : http://www.cs.bc.edu/~straubin/cs381-05/blockciphers/rijndael\_ingles2004.swf.

[3] High Performance AES (Rijndael) cores for Xilinx FPGA [Электронный ресурс] : Datasheet / Helion Technology. – Электронные данные. – Режим доступа : www.heliontech.com/downloads/aes\_xilinx\_helioncore.pdf.

[4] Implementation of the AES-128 on Virtex-5 FPGAs [Электронный ресурс] : Article / Philippe Bulens, Francois-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, Gael Rouvroy – Электронные данные. – Режим доступа : perso.uclouvain.be/fstandae/publis/53.pdf.

[5] FPGA Implementations of S-box vs. T-box iterative architectures of AES [Электронный ресурс] : Article / Bhupathi Kakarlapudi, Nitin Alabur – Электронные данные. – Режим доступа : teal.gmu.edu/courses/ECE746/project/reports\_2008/AES\_T-box\_report.pdf.

[6] Палицын, В. А. Технико-экономическое обоснование дипломных проектов : метод. пособие для студ. всех спец. БГУИР. В 4-х ч. Ч.4 : Проекты программного обеспечения / В. А. Палицын. - Минск : БГУИР, 2006. – 76 с.

[7] Щербина, Н. В. Охрана труда. Проектирование и расчёт производственного освещения : метод. пособие / Н. В. Щербина, Д. А. Мельниченко, А. В. Копыток. - Минск : БГУИР, 2009. – 36 с.

**ПРИЛОЖЕНИЕ A**

**(обязательное)**

**Графические материалы**



Рисунок А.1 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET) для раздела 3.3.2

|  |  |
| --- | --- |
|  | Рисунок А.2 – Укрупнённая функциональная схема 10-тактового процессора зашифрования с прямой реализацией  функции MixColumns и отдельным блоком AddRoundKey для обработки нулевого раунда |
|  | Рисунок А.3 – Укрупнённая функциональная схема 10-тактового процессора зашифрования с прямой реализацией  функции MixColumns и отдельным блоком AddRoundKey для обработки десятого раунда |



Рисунок А.4 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET) для раздела 3.3.3

|  |  |
| --- | --- |
|  | Рисунок А.5 – Укрупнённая функциональная схема 11-тактового процессора зашифрования  на архитектуре с использованием Т-таблиц на базе однопортовой ROM-памяти |
|  | Рисунок А.6 – Упрощённая функциональная схема 11-тактового процессора зашифрования  на архитектуре с использованием Т-таблиц на базе двухпортовой ROM-памяти |



Рисунок А.7 – Функциональная схема части блока T-Box Module в версии с двухпортовой памятью, обрабатывающая 4 байта входных данных

|  |  |
| --- | --- |
|  | Рисунок А.8 – Укрупнённая функциональная схема процессора зашифрования и расшифрования с асинхронной  памятью раундовых ключей |



Рисунок А.9 – Блок-схема алгоритма работы автомата FSM (без учёта сигнала RESET) для процессора зашифрования и расшифрования с асинхронной памятью ключей

**ПРИЛОЖЕНИЕ Б**

**(обязательное)**

**Исходные описания разработанных модификаций процессоров**

**1 Описание процессора зашифрования и расшифрования с**

**синхронной памятью раундовых ключей**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Main is

port ( CLK : in std\_logic;

RESET : in STD\_LOGIC;

ENCR\_DECR : in STD\_LOGIC;

DATA\_WRITE : in STD\_LOGIC;

INPUT\_TEXT : in std\_logic\_vector(0 to 127);

AVAILABLE : out STD\_LOGIC;

OUTPUT\_TEXT : out std\_logic\_vector(0 to 127)

);

end Main;

architecture Main\_architecture of Main is

type state\_type is (i\_0, i\_1, i\_2, s\_0, s\_0\_2, s\_1);

signal state, next\_state : state\_type;

signal counter, a1 : STD\_LOGIC\_VECTOR (3 downto 0);

component RoundEncryptDecrypt

port ( ROUND\_KEY : in STD\_LOGIC\_VECTOR(0 to 127);

ROUND\_NUMBER : in STD\_LOGIC\_VECTOR(3 downto 0);

INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

FB\_INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

INIT\_INPUT : in STD\_LOGIC;

ENCR\_DECR : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component KeyScheduleRAM

port ( CLK : in std\_logic;

WE : in std\_logic;

ADDRESS : STD\_LOGIC\_VECTOR (3 downto 0);

INPUT : in std\_logic\_vector(0 to 127);

OUTPUT : out std\_logic\_vector(0 to 127)

);

end component;

component MUX\_Inv

port ( R : in STD\_LOGIC\_VECTOR (3 downto 0);

SEL : in STD\_LOGIC;

ENCR\_DECR : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR (3 downto 0) );

end component;

component Reg\_128\_EN

port( CLK : in std\_logic;

EN : in std\_logic;

D : in std\_logic\_vector(0 to 127);

Q : out std\_logic\_vector(0 to 127)

);

end component;

component MUX\_128\_2

port ( IN\_0 : in STD\_LOGIC\_VECTOR(0 to 127);

IN\_1 : in STD\_LOGIC\_VECTOR(0 to 127);

SEL : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component Counter\_to\_10

port ( CLK : in STD\_LOGIC;

CLR : in STD\_LOGIC;

CE : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0)

);

end component;

component KeySchedule

port ( prev\_key : in STD\_LOGIC\_VECTOR(0 to 127);

round\_number : in STD\_LOGIC\_VECTOR (3 downto 0);

round\_key : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

signal ENCR\_DECR\_reg, reg\_RESET, reg\_DATA\_WRITE, tmp\_AVAILABLE,

enable\_output\_reg, enable\_core\_output\_reg,

enable\_input\_reg : STD\_LOGIC := '0';

signal rk\_to\_re\_d, input\_reg\_output, w2, w3:

STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal s1, s2, s3: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal control\_enable\_reg: STD\_LOGIC := '0';

signal control\_start\_counter: STD\_LOGIC := '0';

signal control\_write\_zero\_key: STD\_LOGIC := '0';

signal control\_write\_ram: STD\_LOGIC := '0';

signal control\_initialized: STD\_LOGIC := '0';

signal control\_enable\_core\_output\_reg: STD\_LOGIC := '0';

signal control\_input: STD\_LOGIC := '0';

signal control\_set\_encr\_decr\_mode: STD\_LOGIC := '0';

signal control\_data\_available: STD\_LOGIC := '0';

begin

RES: process(CLK)

begin

if (CLK'event and CLK = '1') then

reg\_RESET <= RESET;

end if;

end process;

E\_D: process(CLK)

begin

if (CLK'event and CLK = '1') then

if control\_set\_encr\_decr\_mode = '1' then

ENCR\_DECR\_reg <= ENCR\_DECR;

end if;

end if;

end process;

D\_W: process(CLK)

begin

if (CLK'event and CLK = '1') then

reg\_DATA\_WRITE <= DATA\_WRITE;

end if;

end process;

enable\_core\_output\_reg <= control\_enable\_core\_output\_reg;

enable\_output\_reg <= tmp\_AVAILABLE;

enable\_input\_reg <= reg\_DATA\_WRITE;

tmp\_AVAIL: process(CLK)

begin

if (CLK'event and CLK = '1') then

tmp\_AVAILABLE <= control\_data\_available;

end if;

end process;

AVAIL: process(CLK)

begin

if (CLK'event and CLK = '1') then

AVAILABLE <= tmp\_AVAILABLE;

end if;

end process;

main1: KeyScheduleRAM port map (

CLK => CLK,

WE => control\_write\_ram,

ADDRESS => a1,

INPUT => s1,

OUTPUT => rk\_to\_re\_d

);

main1\_2: MUX\_Inv port map (

R => counter,

SEL => control\_initialized,

ENCR\_DECR => ENCR\_DECR\_reg,

OUTPUT => a1

);

input\_reg: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_input\_reg,

D => INPUT\_TEXT,

Q => input\_reg\_output

);

main2: RoundEncryptDecrypt port map (

ROUND\_KEY => rk\_to\_re\_d,

ROUND\_NUMBER => counter,

INPUT => input\_reg\_output,

FB\_INPUT => w3,

INIT\_INPUT => control\_input,

ENCR\_DECR => ENCR\_DECR\_reg,

OUTPUT => w2

);

main3: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_core\_output\_reg,

D => w2,

Q => w3

);

main4: Counter\_to\_10 port map (

CLK => CLK,

CLR => reg\_RESET,

CE => control\_start\_counter,

Q => counter

);

wrk0: KeySchedule port map (

prev\_key => s1,

round\_number => counter,

round\_key => s2

);

wrk1: MUX\_128\_2 port map (

IN\_0 => s2,

IN\_1 => input\_reg\_output,

SEL => control\_write\_zero\_key,

OUTPUT => s3

);

wrk2: Reg\_128\_EN port map (

CLK => CLK,

EN => control\_enable\_reg,

D => s3,

Q => s1

);

ouput\_reg: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_output\_reg,

D => w3,

Q => OUTPUT\_TEXT

);

FSM: process (state, reg\_DATA\_WRITE, counter)

begin

control\_enable\_reg <= '0';

control\_start\_counter <= '0';

control\_write\_zero\_key <= '0';

control\_write\_ram <= '0';

control\_initialized <= '0';

control\_enable\_core\_output\_reg <= '0';

control\_set\_encr\_decr\_mode <= '0';

control\_input <= '0';

control\_data\_available <= '0';

case (state) is

when i\_0 =>

if reg\_DATA\_WRITE = '1' then

next\_state <= i\_1;

else

next\_state <= i\_0;

end if;

when i\_1 =>

control\_enable\_reg <= '1';

control\_write\_zero\_key <= '1';

next\_state <= i\_2;

when i\_2 =>

control\_write\_ram <= '1';

control\_start\_counter <= '1';

if counter = "1010" then

control\_data\_available <= '1';

next\_state <= s\_0;

else

control\_enable\_reg <= '1';

next\_state <= i\_2;

end if;

when s\_0 =>

control\_set\_encr\_decr\_mode <= '1';

control\_initialized <= '1';

if reg\_DATA\_WRITE = '1' then

control\_start\_counter <= '1';

next\_state <= s\_0\_2;

else

next\_state <= s\_0;

end if;

when s\_0\_2 =>

control\_initialized <= '1';

control\_enable\_core\_output\_reg <= '1';

control\_input <= '1';

control\_start\_counter <= '1';

next\_state <= s\_1;

when s\_1 =>

control\_initialized <= '1';

control\_enable\_core\_output\_reg <= '1';

if counter = "0000" then

control\_data\_available <= '1';

if reg\_DATA\_WRITE = '1' then

control\_start\_counter <= '1';

next\_state <= s\_0\_2;

else

next\_state <= s\_0;

end if;

else

control\_start\_counter <= '1';

next\_state <= s\_1;

end if;

end case;

end process;

SYNC\_FSM: process(CLK)

begin

if rising\_edge(CLK) then

if reg\_RESET = '1' then

state <= i\_0;

else

state <= next\_state;

end if;

end if;

end process;

end Main\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity RoundEncryptDecrypt is

port ( ROUND\_KEY : in STD\_LOGIC\_VECTOR(0 to 127);

ROUND\_NUMBER : in STD\_LOGIC\_VECTOR(3 downto 0);

INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

FB\_INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

INIT\_INPUT : in STD\_LOGIC;

ENCR\_DECR : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end RoundEncryptDecrypt;

architecture RoundEncryptDecrypt\_architecture of RoundEncryptDecrypt is

component SubBytes

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component ShiftRows

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component MixColumns

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component InvSubBytes

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component InvShiftRows

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component InvMixColumns

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component AddRoundKey

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

round\_key : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

signal b0, b1, b2, b3: STD\_LOGIC\_VECTOR(0 to 127)

:= (others => '0');

signal d0, d1, d2, d3: STD\_LOGIC\_VECTOR(0 to 127)

:= (others => '0');

signal m1, m2, m3: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal tmp: STD\_LOGIC;

begin

en\_1: SubBytes port map (

input => FB\_INPUT,

output => b0

);

en\_2: ShiftRows port map (

input => b0,

output => b1

);

en\_3: MixColumns port map (

input => b1,

output => b2

);

en\_4: process(ROUND\_NUMBER, b1, b2)

begin

case ROUND\_NUMBER is

when "0000" => m1 <= b1;

when others => m1 <= b2;

end case;

end process;

en\_5: process(INIT\_INPUT, INPUT, m1)

begin

case INIT\_INPUT is

when '1' => m2 <= INPUT;

when others => m2 <= m1;

end case;

end process;

en\_6: AddRoundKey port map (

input => m2,

round\_key => ROUND\_KEY,

output => b3

);

de\_1: InvShiftRows port map (

input => FB\_INPUT,

output => d0

);

de\_2: InvSubBytes port map (

input => d0,

output => d1

);

de\_3: AddRoundKey port map (

input => d1,

round\_key => ROUND\_KEY,

output => d2 );

de\_4: InvMixColumns port map (

input => d2,

output => d3 );

de\_5: process(ROUND\_NUMBER, d2, d3)

begin

case ROUND\_NUMBER is

when "0000" => m3 <= d2;

when others => m3 <= d3;

end case;

end process;

tmp\_sel: process(ENCR\_DECR, ROUND\_NUMBER)

begin

if (ROUND\_NUMBER = "0001") or (ENCR\_DECR = '1') then

tmp <= '1';

else

tmp <= '0';

end if;

end process;

select\_output: process(tmp, b3, m3)

begin

if tmp = '1' then

OUTPUT <= b3;

else

OUTPUT <= m3;

end if;

end process;

end RoundEncryptDecrypt\_architecture;

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity KeyScheduleRAM is

port ( CLK : in std\_logic;

WE : in std\_logic;

ADDRESS : STD\_LOGIC\_VECTOR (3 downto 0);

INPUT : in std\_logic\_vector(0 to 127);

OUTPUT : out std\_logic\_vector(0 to 127)

);

end KeyScheduleRAM;

architecture KeyScheduleRAM\_architecture of KeyScheduleRAM is

type ram\_type is array (0 to 15) of std\_logic\_vector (0 to 127);

signal RAM : ram\_type;

begin

process (CLK)

begin

if (CLK'event and CLK = '1') then

if (WE = '1') then

RAM(conv\_integer(ADDRESS)) <= INPUT;

end if;

OUTPUT <= RAM(conv\_integer(ADDRESS));

end if;

end process;

end KeyScheduleRAM\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity MUX\_Inv is

port ( R : in STD\_LOGIC\_VECTOR (3 downto 0);

SEL : in STD\_LOGIC;

ENCR\_DECR : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR (3 downto 0)

);

end MUX\_Inv;

architecture MUX\_Inv\_architecture of MUX\_Inv is

begin

process (SEL, ENCR\_DECR, R)

begin

if ( ENCR\_DECR = '0' and SEL = '1' ) then

case R is

when "0000" => OUTPUT <= "1010";

when "0001" => OUTPUT <= "1001";

when "0010" => OUTPUT <= "1000";

when "0011" => OUTPUT <= "0111";

when "0100" => OUTPUT <= "0110";

when "0101" => OUTPUT <= "0101";

when "0110" => OUTPUT <= "0100";

when "0111" => OUTPUT <= "0011";

when "1000" => OUTPUT <= "0010";

when "1001" => OUTPUT <= "0001";

when "1010" => OUTPUT <= "0000";

when others => OUTPUT <= "1010";

end case;

else

OUTPUT <= R;

end if;

end process;

end MUX\_Inv\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Reg\_128\_EN is

port( CLK : in std\_logic;

EN : in std\_logic;

D : in std\_logic\_vector(0 to 127);

Q : out std\_logic\_vector(0 to 127)

);

end Reg\_128\_EN;

architecture Reg\_128\_EN\_architecture of Reg\_128\_EN is

signal tmp: std\_logic\_vector(0 to 127);

begin

process (CLK)

begin

if (CLK'event and CLK='1') then

if EN = '1' then

tmp <= D;

end if;

end if;

end process;

Q <= tmp;

end Reg\_128\_EN\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity MUX\_128\_2 is

port ( IN\_0 : in STD\_LOGIC\_VECTOR(0 to 127);

IN\_1 : in STD\_LOGIC\_VECTOR(0 to 127);

SEL : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end MUX\_128\_2;

architecture MUX\_128\_2\_architecture of MUX\_128\_2 is

begin

process(SEL, IN\_0, IN\_1)

begin

case SEL is

when '1' => OUTPUT <= IN\_1;

when '0' => OUTPUT <= IN\_0;

when others => OUTPUT <= IN\_0;

end case;

end process;

end MUX\_128\_2\_architecture;

library ieee;

use ieee.std\_logic\_1164.all;

use ieee.std\_logic\_unsigned.all;

entity Counter\_to\_10 is

port ( CLK : in STD\_LOGIC;

CLR : in STD\_LOGIC;

CE : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0)

);

end Counter\_to\_10;

architecture Counter\_to\_10\_architecture of Counter\_to\_10 is

signal tmp: std\_logic\_vector(3 downto 0);

begin

process (CLK, CLR)

begin

if rising\_edge(CLK) then

if (CLR = '1') then

tmp <= "0000";

elsif CE = '1' then

if tmp = "1010" then

tmp <= "0000";

else

tmp <= tmp + 1;

end if;

end if;

end if;

end process;

Q <= tmp;

end Counter\_to\_10\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity KeySchedule is

Port ( prev\_key : in STD\_LOGIC\_VECTOR(0 to 127);

round\_number : in STD\_LOGIC\_VECTOR (3 downto 0);

round\_key : out STD\_LOGIC\_VECTOR(0 to 127)

);

end KeySchedule;

architecture KeySchedule\_architecture of KeySchedule is

component S\_Box

Port ( input : in STD\_LOGIC\_VECTOR(7 downto 0);

output : out STD\_LOGIC\_VECTOR(7 downto 0)

);

end component;

component Rcon\_zero\_row

Port ( input : STD\_LOGIC\_VECTOR (3 downto 0);

output : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end component;

signal RotWord: STD\_LOGIC\_VECTOR(0 to 31) := (others => '0');

signal SubBts: STD\_LOGIC\_VECTOR(0 to 31) := (others => '0');

signal Rcon\_column\_1st\_element: STD\_LOGIC\_VECTOR(0 to 7) := (others => '0');

signal rk: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

for\_Rcon: Rcon\_zero\_row port map (

input => round\_number,

output => Rcon\_column\_1st\_element

);

RotWord <= prev\_key(104 to 127) & prev\_key(96 to 103);

generate\_for\_1st\_column:

for i in 0 to 3 generate

begin

for\_each\_byte\_in\_1st\_column:

S\_Box port map (

input => RotWord(8\*i to 8\*i+7),

output => SubBts(8\*i to 8\*i+7) );

end generate;

rk(0 to 31) <= prev\_key(0 to 31) xor

SubBts xor

(Rcon\_column\_1st\_element & X"000000");

rk(32 to 63) <= prev\_key(32 to 63) xor rk(0 to 31);

rk(64 to 95) <= prev\_key(64 to 95) xor rk(32 to 63);

rk(96 to 127) <= prev\_key(96 to 127) xor rk(64 to 95);

round\_key <= rk;

end KeySchedule\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Rcon\_zero\_row is

Port ( input : STD\_LOGIC\_VECTOR (3 downto 0);

output : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end Rcon\_zero\_row;

architecture Rcon\_zero\_row\_architecture of Rcon\_zero\_row is

begin

substitute: process(input)

begin

case input is

when "0000" => output <= X"01";

when "0001" => output <= X"02";

when "0010" => output <= X"04";

when "0011" => output <= X"08";

when "0100" => output <= X"10";

when "0101" => output <= X"20";

when "0110" => output <= X"40";

when "0111" => output <= X"80";

when "1000" => output <= X"1B";

when "1001" => output <= X"36";

when others => output <= X"01";

end case;

end process;

end Rcon\_zero\_row\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity S\_Box is

Port ( input : in STD\_LOGIC\_VECTOR (7 downto 0);

output : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end S\_Box;

architecture S\_Box\_architecture of S\_Box is

begin

substitute: process(input)

begin

case input is

when X"00" => output <= X"63";

when X"01" => output <= X"7C";

when X"02" => output <= X"77";

when X"03" => output <= X"7B";

when X"04" => output <= X"F2";

when X"05" => output <= X"6B";

when X"06" => output <= X"6F";

when X"07" => output <= X"C5";

when X"08" => output <= X"30";

when X"09" => output <= X"01";

when X"0A" => output <= X"67";

when X"0B" => output <= X"2B";

when X"0C" => output <= X"FE";

when X"0D" => output <= X"D7";

when X"0E" => output <= X"AB";

when X"0F" => output <= X"76";

when X"10" => output <= X"CA";

when X"11" => output <= X"82";

when X"12" => output <= X"C9";

when X"13" => output <= X"7D";

when X"14" => output <= X"FA";

when X"15" => output <= X"59";

when X"16" => output <= X"47";

when X"17" => output <= X"F0";

when X"18" => output <= X"AD";

when X"19" => output <= X"D4";

when X"1A" => output <= X"A2";

when X"1B" => output <= X"AF";

when X"1C" => output <= X"9C";

when X"1D" => output <= X"A4";

when X"1E" => output <= X"72";

when X"1F" => output <= X"C0";

when X"20" => output <= X"B7";

when X"21" => output <= X"FD";

when X"22" => output <= X"93";

when X"23" => output <= X"26";

when X"24" => output <= X"36";

when X"25" => output <= X"3F";

when X"26" => output <= X"F7";

when X"27" => output <= X"CC";

when X"28" => output <= X"34";

when X"29" => output <= X"A5";

when X"2A" => output <= X"E5";

when X"2B" => output <= X"F1";

when X"2C" => output <= X"71";

when X"2D" => output <= X"D8";

when X"2E" => output <= X"31";

when X"2F" => output <= X"15";

when X"30" => output <= X"04";

when X"31" => output <= X"C7";

when X"32" => output <= X"23";

when X"33" => output <= X"C3";

when X"34" => output <= X"18";

when X"35" => output <= X"96";

when X"36" => output <= X"05";

when X"37" => output <= X"9A";

when X"38" => output <= X"07";

when X"39" => output <= X"12";

when X"3A" => output <= X"80";

when X"3B" => output <= X"E2";

when X"3C" => output <= X"EB";

when X"3D" => output <= X"27";

when X"3E" => output <= X"B2";

when X"3F" => output <= X"75";

when X"40" => output <= X"09";

when X"41" => output <= X"83";

when X"42" => output <= X"2C";

when X"43" => output <= X"1A";

when X"44" => output <= X"1B";

when X"45" => output <= X"6E";

when X"46" => output <= X"5A";

when X"47" => output <= X"A0";

when X"48" => output <= X"52";

when X"49" => output <= X"3B";

when X"4A" => output <= X"D6";

when X"4B" => output <= X"B3";

when X"4C" => output <= X"29";

when X"4D" => output <= X"E3";

when X"4E" => output <= X"2F";

when X"4F" => output <= X"84";

when X"50" => output <= X"53";

when X"51" => output <= X"D1";

when X"52" => output <= X"00";

when X"53" => output <= X"ED";

when X"54" => output <= X"20";

when X"55" => output <= X"FC";

when X"56" => output <= X"B1";

when X"57" => output <= X"5B";

when X"58" => output <= X"6A";

when X"59" => output <= X"CB";

when X"5A" => output <= X"BE";

when X"5B" => output <= X"39";

when X"5C" => output <= X"4A";

when X"5D" => output <= X"4C";

when X"5E" => output <= X"58";

when X"5F" => output <= X"CF";

when X"60" => output <= X"D0";

when X"61" => output <= X"EF";

when X"62" => output <= X"AA";

when X"63" => output <= X"FB";

when X"64" => output <= X"43";

when X"65" => output <= X"4D";

when X"66" => output <= X"33";

when X"67" => output <= X"85";

when X"68" => output <= X"45";

when X"69" => output <= X"F9";

when X"6A" => output <= X"02";

when X"6B" => output <= X"7F";

when X"6C" => output <= X"50";

when X"6D" => output <= X"3C";

when X"6E" => output <= X"9F";

when X"6F" => output <= X"A8";

when X"70" => output <= X"51";

when X"71" => output <= X"A3";

when X"72" => output <= X"40";

when X"73" => output <= X"8F";

when X"74" => output <= X"92";

when X"75" => output <= X"9D";

when X"76" => output <= X"38";

when X"77" => output <= X"F5";

when X"78" => output <= X"BC";

when X"79" => output <= X"B6";

when X"7A" => output <= X"DA";

when X"7B" => output <= X"21";

when X"7C" => output <= X"10";

when X"7D" => output <= X"FF";

when X"7E" => output <= X"F3";

when X"7F" => output <= X"D2";

when X"80" => output <= X"CD";

when X"81" => output <= X"0C";

when X"82" => output <= X"13";

when X"83" => output <= X"EC";

when X"84" => output <= X"5F";

when X"85" => output <= X"97";

when X"86" => output <= X"44";

when X"87" => output <= X"17";

when X"88" => output <= X"C4";

when X"89" => output <= X"A7";

when X"8A" => output <= X"7E";

when X"8B" => output <= X"3D";

when X"8C" => output <= X"64";

when X"8D" => output <= X"5D";

when X"8E" => output <= X"19";

when X"8F" => output <= X"73";

when X"90" => output <= X"60";

when X"91" => output <= X"81";

when X"92" => output <= X"4F";

when X"93" => output <= X"DC";

when X"94" => output <= X"22";

when X"95" => output <= X"2A";

when X"96" => output <= X"90";

when X"97" => output <= X"88";

when X"98" => output <= X"46";

when X"99" => output <= X"EE";

when X"9A" => output <= X"B8";

when X"9B" => output <= X"14";

when X"9C" => output <= X"DE";

when X"9D" => output <= X"5E";

when X"9E" => output <= X"0B";

when X"9F" => output <= X"DB";

when X"A0" => output <= X"E0";

when X"A1" => output <= X"32";

when X"A2" => output <= X"3A";

when X"A3" => output <= X"0A";

when X"A4" => output <= X"49";

when X"A5" => output <= X"06";

when X"A6" => output <= X"24";

when X"A7" => output <= X"5C";

when X"A8" => output <= X"C2";

when X"A9" => output <= X"D3";

when X"AA" => output <= X"AC";

when X"AB" => output <= X"62";

when X"AC" => output <= X"91";

when X"AD" => output <= X"95";

when X"AE" => output <= X"E4";

when X"AF" => output <= X"79";

when X"B0" => output <= X"E7";

when X"B1" => output <= X"C8";

when X"B2" => output <= X"37";

when X"B3" => output <= X"6D";

when X"B4" => output <= X"8D";

when X"B5" => output <= X"D5";

when X"B6" => output <= X"4E";

when X"B7" => output <= X"A9";

when X"B8" => output <= X"6C";

when X"B9" => output <= X"56";

when X"BA" => output <= X"F4";

when X"BB" => output <= X"EA";

when X"BC" => output <= X"65";

when X"BD" => output <= X"7A";

when X"BE" => output <= X"AE";

when X"BF" => output <= X"08";

when X"C0" => output <= X"BA";

when X"C1" => output <= X"78";

when X"C2" => output <= X"25";

when X"C3" => output <= X"2E";

when X"C4" => output <= X"1C";

when X"C5" => output <= X"A6";

when X"C6" => output <= X"B4";

when X"C7" => output <= X"C6";

when X"C8" => output <= X"E8";

when X"C9" => output <= X"DD";

when X"CA" => output <= X"74";

when X"CB" => output <= X"1F";

when X"CC" => output <= X"4B";

when X"CD" => output <= X"BD";

when X"CE" => output <= X"8B";

when X"CF" => output <= X"8A";

when X"D0" => output <= X"70";

when X"D1" => output <= X"3E";

when X"D2" => output <= X"B5";

when X"D3" => output <= X"66";

when X"D4" => output <= X"48";

when X"D5" => output <= X"03";

when X"D6" => output <= X"F6";

when X"D7" => output <= X"0E";

when X"D8" => output <= X"61";

when X"D9" => output <= X"35";

when X"DA" => output <= X"57";

when X"DB" => output <= X"B9";

when X"DC" => output <= X"86";

when X"DD" => output <= X"C1";

when X"DE" => output <= X"1D";

when X"DF" => output <= X"9E";

when X"E0" => output <= X"E1";

when X"E1" => output <= X"F8";

when X"E2" => output <= X"98";

when X"E3" => output <= X"11";

when X"E4" => output <= X"69";

when X"E5" => output <= X"D9";

when X"E6" => output <= X"8E";

when X"E7" => output <= X"94";

when X"E8" => output <= X"9B";

when X"E9" => output <= X"1E";

when X"EA" => output <= X"87";

when X"EB" => output <= X"E9";

when X"EC" => output <= X"CE";

when X"ED" => output <= X"55";

when X"EE" => output <= X"28";

when X"EF" => output <= X"DF";

when X"F0" => output <= X"8C";

when X"F1" => output <= X"A1";

when X"F2" => output <= X"89";

when X"F3" => output <= X"0D";

when X"F4" => output <= X"BF";

when X"F5" => output <= X"E6";

when X"F6" => output <= X"42";

when X"F7" => output <= X"68";

when X"F8" => output <= X"41";

when X"F9" => output <= X"99";

when X"FA" => output <= X"2D";

when X"FB" => output <= X"0F";

when X"FC" => output <= X"B0";

when X"FD" => output <= X"54";

when X"FE" => output <= X"BB";

when X"FF" => output <= X"16";

when others => null;

end case;

end process;

end S\_Box\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity AddRoundKey is

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

round\_key : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127) );

end AddRoundKey;

architecture AddRoundKey\_architecture of AddRoundKey is

begin

output <= input xor round\_key;

end AddRoundKey\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity ShiftRows is

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end ShiftRows;

architecture ShiftRows\_architecture of ShiftRows is

begin

output(0 to 7) <= input(0 to 7);

output(8 to 15) <= input(40 to 47);

output(16 to 23) <= input(80 to 87);

output(24 to 31) <= input(120 to 127);

output(32 to 39) <= input(32 to 39);

output(40 to 47) <= input(72 to 79);

output(48 to 55) <= input(112 to 119);

output(56 to 63) <= input(24 to 31);

output(64 to 71) <= input(64 to 71);

output(72 to 79) <= input(104 to 111);

output(80 to 87) <= input(16 to 23);

output(88 to 95) <= input(56 to 63);

output(96 to 103) <= input(96 to 103);

output(104 to 111) <= input(8 to 15);

output(112 to 119) <= input(48 to 55);

output(120 to 127) <= input(88 to 95);

end ShiftRows\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity InvShiftRows is

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end InvShiftRows;

architecture InvShiftRows\_architecture of InvShiftRows is

begin

output(0 to 7) <= input(0 to 7);

output(8 to 15) <= input(104 to 111);

output(16 to 23) <= input(80 to 87);

output(24 to 31) <= input(56 to 63);

output(32 to 39) <= input(32 to 39);

output(40 to 47) <= input(8 to 15);

output(48 to 55) <= input(112 to 119);

output(56 to 63) <= input(88 to 95);

output(64 to 71) <= input(64 to 71);

output(72 to 79) <= input(40 to 47);

output(80 to 87) <= input(16 to 23);

output(88 to 95) <= input(120 to 127);

output(96 to 103) <= input(96 to 103);

output(104 to 111) <= input(72 to 79);

output(112 to 119) <= input(48 to 55);

output(120 to 127) <= input(24 to 31);

end InvShiftRows\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity SubBytes is

port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end SubBytes;

architecture SubBytes\_architecture of SubBytes is

component S\_Box

Port ( input : in STD\_LOGIC\_VECTOR(7 downto 0);

output : out STD\_LOGIC\_VECTOR(7 downto 0)

);

end component;

signal w: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

generate\_for\_each\_byte:

for i in 0 to 15 generate

begin

substitute: S\_Box port map (

input => input(8\*i to 8\*i+7),

output => w(8\*i to 8\*i+7)

);

end generate;

output <= w;

end SubBytes\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity InvSubBytes is

port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end InvSubBytes;

architecture InvSubBytes\_architecture of InvSubBytes is

component Inv\_S\_Box

Port ( input : in STD\_LOGIC\_VECTOR(7 downto 0);

output : out STD\_LOGIC\_VECTOR(7 downto 0)

);

end component;

signal w: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

generate\_for\_each\_byte:

for i in 0 to 15 generate

begin

substitute: Inv\_S\_Box port map (

input => input(8\*i to 8\*i+7),

output => w(8\*i to 8\*i+7)

);

end generate;

output <= w;

end InvSubBytes\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Inv\_S\_Box is

Port ( input : in STD\_LOGIC\_VECTOR (7 downto 0);

output : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end Inv\_S\_Box;

architecture Inv\_S\_Box\_architecture of Inv\_S\_Box is

begin

substitute: process(input)

begin

case input is

when X"00" => output <= X"52";

when X"01" => output <= X"09";

when X"02" => output <= X"6A";

when X"03" => output <= X"D5";

when X"04" => output <= X"30";

when X"05" => output <= X"36";

when X"06" => output <= X"A5";

when X"07" => output <= X"38";

when X"08" => output <= X"BF";

when X"09" => output <= X"40";

when X"0A" => output <= X"A3";

when X"0B" => output <= X"9E";

when X"0C" => output <= X"81";

when X"0D" => output <= X"F3";

when X"0E" => output <= X"D7";

when X"0F" => output <= X"FB";

when X"10" => output <= X"7C";

when X"11" => output <= X"E3";

when X"12" => output <= X"39";

when X"13" => output <= X"82";

when X"14" => output <= X"9B";

when X"15" => output <= X"2F";

when X"16" => output <= X"FF";

when X"17" => output <= X"87";

when X"18" => output <= X"34";

when X"19" => output <= X"8E";

when X"1A" => output <= X"43";

when X"1B" => output <= X"44";

when X"1C" => output <= X"C4";

when X"1D" => output <= X"DE";

when X"1E" => output <= X"E9";

when X"1F" => output <= X"CB";

when X"20" => output <= X"54";

when X"21" => output <= X"7B";

when X"22" => output <= X"94";

when X"23" => output <= X"32";

when X"24" => output <= X"A6";

when X"25" => output <= X"C2";

when X"26" => output <= X"23";

when X"27" => output <= X"3D";

when X"28" => output <= X"EE";

when X"29" => output <= X"4C";

when X"2A" => output <= X"95";

when X"2B" => output <= X"0B";

when X"2C" => output <= X"42";

when X"2D" => output <= X"FA";

when X"2E" => output <= X"C3";

when X"2F" => output <= X"4E";

when X"30" => output <= X"08";

when X"31" => output <= X"2E";

when X"32" => output <= X"A1";

when X"33" => output <= X"66";

when X"34" => output <= X"28";

when X"35" => output <= X"D9";

when X"36" => output <= X"24";

when X"37" => output <= X"B2";

when X"38" => output <= X"76";

when X"39" => output <= X"5B";

when X"3A" => output <= X"A2";

when X"3B" => output <= X"49";

when X"3C" => output <= X"6D";

when X"3D" => output <= X"8B";

when X"3E" => output <= X"D1";

when X"3F" => output <= X"25";

when X"40" => output <= X"72";

when X"41" => output <= X"F8";

when X"42" => output <= X"F6";

when X"43" => output <= X"64";

when X"44" => output <= X"86";

when X"45" => output <= X"68";

when X"46" => output <= X"98";

when X"47" => output <= X"16";

when X"48" => output <= X"D4";

when X"49" => output <= X"A4";

when X"4A" => output <= X"5C";

when X"4B" => output <= X"CC";

when X"4C" => output <= X"5D";

when X"4D" => output <= X"65";

when X"4E" => output <= X"B6";

when X"4F" => output <= X"92";

when X"50" => output <= X"6C";

when X"51" => output <= X"70";

when X"52" => output <= X"48";

when X"53" => output <= X"50";

when X"54" => output <= X"FD";

when X"55" => output <= X"ED";

when X"56" => output <= X"B9";

when X"57" => output <= X"DA";

when X"58" => output <= X"5E";

when X"59" => output <= X"15";

when X"5A" => output <= X"46";

when X"5B" => output <= X"57";

when X"5C" => output <= X"A7";

when X"5D" => output <= X"8D";

when X"5E" => output <= X"9D";

when X"5F" => output <= X"84";

when X"60" => output <= X"90";

when X"61" => output <= X"D8";

when X"62" => output <= X"AB";

when X"63" => output <= X"00";

when X"64" => output <= X"8C";

when X"65" => output <= X"BC";

when X"66" => output <= X"D3";

when X"67" => output <= X"0A";

when X"68" => output <= X"F7";

when X"69" => output <= X"E4";

when X"6A" => output <= X"58";

when X"6B" => output <= X"05";

when X"6C" => output <= X"B8";

when X"6D" => output <= X"B3";

when X"6E" => output <= X"45";

when X"6F" => output <= X"06";

when X"70" => output <= X"D0";

when X"71" => output <= X"2C";

when X"72" => output <= X"1E";

when X"73" => output <= X"8F";

when X"74" => output <= X"CA";

when X"75" => output <= X"3F";

when X"76" => output <= X"0F";

when X"77" => output <= X"02";

when X"78" => output <= X"C1";

when X"79" => output <= X"AF";

when X"7A" => output <= X"BD";

when X"7B" => output <= X"03";

when X"7C" => output <= X"01";

when X"7D" => output <= X"13";

when X"7E" => output <= X"8A";

when X"7F" => output <= X"6B";

when X"80" => output <= X"3A";

when X"81" => output <= X"91";

when X"82" => output <= X"11";

when X"83" => output <= X"41";

when X"84" => output <= X"4F";

when X"85" => output <= X"67";

when X"86" => output <= X"DC";

when X"87" => output <= X"EA";

when X"88" => output <= X"97";

when X"89" => output <= X"F2";

when X"8A" => output <= X"CF";

when X"8B" => output <= X"CE";

when X"8C" => output <= X"F0";

when X"8D" => output <= X"B4";

when X"8E" => output <= X"E6";

when X"8F" => output <= X"73";

when X"90" => output <= X"96";

when X"91" => output <= X"AC";

when X"92" => output <= X"74";

when X"93" => output <= X"22";

when X"94" => output <= X"E7";

when X"95" => output <= X"AD";

when X"96" => output <= X"35";

when X"97" => output <= X"85";

when X"98" => output <= X"E2";

when X"99" => output <= X"F9";

when X"9A" => output <= X"37";

when X"9B" => output <= X"E8";

when X"9C" => output <= X"1C";

when X"9D" => output <= X"75";

when X"9E" => output <= X"DF";

when X"9F" => output <= X"6E";

when X"A0" => output <= X"47";

when X"A1" => output <= X"F1";

when X"A2" => output <= X"1A";

when X"A3" => output <= X"71";

when X"A4" => output <= X"1D";

when X"A5" => output <= X"29";

when X"A6" => output <= X"C5";

when X"A7" => output <= X"89";

when X"A8" => output <= X"6F";

when X"A9" => output <= X"B7";

when X"AA" => output <= X"62";

when X"AB" => output <= X"0E";

when X"AC" => output <= X"AA";

when X"AD" => output <= X"18";

when X"AE" => output <= X"BE";

when X"AF" => output <= X"1B";

when X"B0" => output <= X"FC";

when X"B1" => output <= X"56";

when X"B2" => output <= X"3E";

when X"B3" => output <= X"4B";

when X"B4" => output <= X"C6";

when X"B5" => output <= X"D2";

when X"B6" => output <= X"79";

when X"B7" => output <= X"20";

when X"B8" => output <= X"9A";

when X"B9" => output <= X"DB";

when X"BA" => output <= X"C0";

when X"BB" => output <= X"FE";

when X"BC" => output <= X"78";

when X"BD" => output <= X"CD";

when X"BE" => output <= X"5A";

when X"BF" => output <= X"F4";

when X"C0" => output <= X"1F";

when X"C1" => output <= X"DD";

when X"C2" => output <= X"A8";

when X"C3" => output <= X"33";

when X"C4" => output <= X"88";

when X"C5" => output <= X"07";

when X"C6" => output <= X"C7";

when X"C7" => output <= X"31";

when X"C8" => output <= X"B1";

when X"C9" => output <= X"12";

when X"CA" => output <= X"10";

when X"CB" => output <= X"59";

when X"CC" => output <= X"27";

when X"CD" => output <= X"80";

when X"CE" => output <= X"EC";

when X"CF" => output <= X"5F";

when X"D0" => output <= X"60";

when X"D1" => output <= X"51";

when X"D2" => output <= X"7F";

when X"D3" => output <= X"A9";

when X"D4" => output <= X"19";

when X"D5" => output <= X"B5";

when X"D6" => output <= X"4A";

when X"D7" => output <= X"0D";

when X"D8" => output <= X"2D";

when X"D9" => output <= X"E5";

when X"DA" => output <= X"7A";

when X"DB" => output <= X"9F";

when X"DC" => output <= X"93";

when X"DD" => output <= X"C9";

when X"DE" => output <= X"9C";

when X"DF" => output <= X"EF";

when X"E0" => output <= X"A0";

when X"E1" => output <= X"E0";

when X"E2" => output <= X"3B";

when X"E3" => output <= X"4D";

when X"E4" => output <= X"AE";

when X"E5" => output <= X"2A";

when X"E6" => output <= X"F5";

when X"E7" => output <= X"B0";

when X"E8" => output <= X"C8";

when X"E9" => output <= X"EB";

when X"EA" => output <= X"BB";

when X"EB" => output <= X"3C";

when X"EC" => output <= X"83";

when X"ED" => output <= X"53";

when X"EE" => output <= X"99";

when X"EF" => output <= X"61";

when X"F0" => output <= X"17";

when X"F1" => output <= X"2B";

when X"F2" => output <= X"04";

when X"F3" => output <= X"7E";

when X"F4" => output <= X"BA";

when X"F5" => output <= X"77";

when X"F6" => output <= X"D6";

when X"F7" => output <= X"26";

when X"F8" => output <= X"E1";

when X"F9" => output <= X"69";

when X"FA" => output <= X"14";

when X"FB" => output <= X"63";

when X"FC" => output <= X"55";

when X"FD" => output <= X"21";

when X"FE" => output <= X"0C";

when X"FF" => output <= X"7D";

when others => null;

end case;

end process;

end Inv\_S\_Box\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity MixColumns is

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127) );

end MixColumns;

architecture MixColumns\_architecture of MixColumns is

component Xtime

Port(

input : in STD\_LOGIC\_VECTOR(7 downto 0);

output : out STD\_LOGIC\_VECTOR(7 downto 0)

);

end component;

signal xtm, w: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

generate\_xtm:

for i in 0 to 15 generate

begin

for\_each\_byte: Xtime port map (

input => input(8\*i to 8\*i+7),

output => xtm(8\*i to 8\*i+7)

);

end generate;

generate\_new\_s\_0\_c:

for i in 0 to 3 generate

begin

w(32\*i to 32\*i+7) <= xtm(32\*i to 32\*i+7) xor

xtm(32\*i+8 to 32\*i+15) xor

input(32\*i+8 to 32\*i+15) xor

input(32\*i+16 to 32\*i+23) xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_1\_c:

for i in 0 to 3 generate

begin

w(32\*i+8 to 32\*i+15) <= xtm(32\*i+8 to 32\*i+15) xor

xtm(32\*i+16 to 32\*i+23) xor

input(32\*i to 32\*i+7) xor

input(32\*i+16 to 32\*i+23) xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_2\_c:

for i in 0 to 3 generate

begin

w(32\*i+16 to 32\*i+23) <= xtm(32\*i+16 to 32\*i+23) xor

xtm(32\*i+24 to 32\*i+31) xor

input(32\*i to 32\*i+7) xor

input(32\*i+8 to 32\*i+15) xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_3\_c:

for i in 0 to 3 generate

begin

w(32\*i+24 to 32\*i+31) <= xtm(32\*i to 32\*i+7) xor

xtm(32\*i+24 to 32\*i+31) xor

input(32\*i to 32\*i+7) xor

input(32\*i+8 to 32\*i+15) xor

input(32\*i+16 to 32\*i+23);

end generate;

output <= w;

end MixColumns\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Xtime is

Port ( input : in STD\_LOGIC\_VECTOR (7 downto 0);

output : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end Xtime;

architecture Xtime\_architecture of Xtime is

signal l\_shift\_input: STD\_LOGIC\_VECTOR (7 downto 0)

:= (others => '0');

signal cond\_x1b\_xor: STD\_LOGIC\_VECTOR (7 downto 0)

:= (others => '0');

begin

l\_shift\_input <= input(6 downto 0) & '0';

select\_output: process(input, l\_shift\_input)

begin

if input(7) = '0' then

cond\_x1b\_xor <= l\_shift\_input;

else

cond\_x1b\_xor <= l\_shift\_input xor X"1B";

end if;

end process;

output <= cond\_x1b\_xor;

end Xtime\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity InvMixColumns is

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end InvMixColumns;

architecture InvMixColumns\_architecture of InvMixColumns is

component Xtime\_3times

Port ( input : in STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x1 : out STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x2 : out STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x3 : out STD\_LOGIC\_VECTOR (7 downto 0)

);

end component;

signal xtm\_1, xtm\_2, xtm\_3, w: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

generate\_xtm\_1\_to\_3:

for i in 0 to 15 generate

begin

for\_each\_byte: Xtime\_3times port map (

input => input(8\*i to 8\*i+7),

xt\_x1 => xtm\_1(8\*i to 8\*i+7),

xt\_x2 => xtm\_2(8\*i to 8\*i+7),

xt\_x3 => xtm\_3(8\*i to 8\*i+7)

);

end generate;

generate\_new\_s\_0\_c:

for i in 0 to 3 generate

begin

w(32\*i to 32\*i+7) <= xtm\_1(32\*i to 32\*i+7) xor

xtm\_2(32\*i to 32\*i+7) xor

xtm\_3(32\*i to 32\*i+7) xor

xtm\_1(32\*i+8 to 32\*i+15) xor

xtm\_3(32\*i+8 to 32\*i+15) xor

input(32\*i+8 to 32\*i+15) xor

xtm\_2(32\*i+16 to 32\*i+23) xor

xtm\_3(32\*i+16 to 32\*i+23) xor

input(32\*i+16 to 32\*i+23) xor

xtm\_3(32\*i+24 to 32\*i+31) xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_1\_c:

for i in 0 to 3 generate

begin

w(32\*i+8 to 32\*i+15) <= xtm\_3(32\*i to 32\*i+7) xor

input(32\*i to 32\*i+7) xor

xtm\_1(32\*i+8 to 32\*i+15) xor

xtm\_2(32\*i+8 to 32\*i+15) xor

xtm\_3(32\*i+8 to 32\*i+15) xor

xtm\_1(32\*i+16 to 32\*i+23) xor

xtm\_3(32\*i+16 to 32\*i+23) xor

input(32\*i+16 to 32\*i+23) xor

xtm\_2(32\*i+24 to 32\*i+31) xor

xtm\_3(32\*i+24 to 32\*i+31) xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_2\_c:

for i in 0 to 3 generate

begin

w(32\*i+16 to 32\*i+23) <= xtm\_2(32\*i to 32\*i+7) xor xtm\_3(32\*i to 32\*i+7) xor

input(32\*i to 32\*i+7) xor

xtm\_3(32\*i+8 to 32\*i+15) xor

input(32\*i+8 to 32\*i+15) xor xtm\_1(32\*i+16 to 32\*i+23) xor

xtm\_2(32\*i+16 to 32\*i+23) xor

xtm\_3(32\*i+16 to 32\*i+23) xor xtm\_1(32\*i+24 to 32\*i+31)xor

xtm\_3(32\*i+24 to 32\*i+31)xor

input(32\*i+24 to 32\*i+31);

end generate;

generate\_new\_s\_3\_c:

for i in 0 to 3 generate

begin

w(32\*i+24 to 32\*i+31) <= xtm\_1(32\*i to 32\*i+7) xor

xtm\_3(32\*i to 32\*i+7) xor input(32\*i to 32\*i+7) xor

xtm\_2(32\*i+8 to 32\*i+15) xor xtm\_3(32\*i+8 to 32\*i+15) xor input(32\*i+8 to 32\*i+15) xor

xtm\_3(32\*i+16 to 32\*i+23) xor input(32\*i+16 to 32\*i+23) xor

xtm\_1(32\*i+24 to 32\*i+31) xor xtm\_2(32\*i+24 to 32\*i+31) xor xtm\_3(32\*i+24 to 32\*i+31);

end generate;

output <= w;

end InvMixColumns\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Xtime\_3times is

Port ( input : in STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x1 : out STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x2 : out STD\_LOGIC\_VECTOR (7 downto 0);

xt\_x3 : out STD\_LOGIC\_VECTOR (7 downto 0) );

end Xtime\_3times;

architecture Xtime\_3times\_architecture of Xtime\_3times is

component Xtime

Port( input : in STD\_LOGIC\_VECTOR(7 downto 0);

output : out STD\_LOGIC\_VECTOR(7 downto 0) );

end component;

signal w1, w2, w3: STD\_LOGIC\_VECTOR (7 downto 0)

:= (others => '0');

begin

serial\_1: Xtime port map ( input => input,

output => w1 );

serial\_2: Xtime port map ( input => w1,

output => w2 );

serial\_3: Xtime port map ( input => w2,

output => w3 );

xt\_x1 <= w1;

xt\_x2 <= w2;

xt\_x3 <= w3;

end Xtime\_3times\_architecture;

**2 Описание 11-тактового процессора зашифрования с прямой**

**рализацией функции MixColumns**

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Main is

port ( CLK : in std\_logic;

RESET : in STD\_LOGIC;

DATA\_WRITE : in STD\_LOGIC;

INPUT\_TEXT : in std\_logic\_vector(0 to 127);

AVAILABLE : out STD\_LOGIC;

OUTPUT\_TEXT : out std\_logic\_vector(0 to 127)

);

end Main;

architecture Main\_architecture of Main is

type state\_type is (i\_0, i\_1, i\_2, s\_0, s\_0\_2, s\_1);

signal state, next\_state : state\_type;

signal counter: STD\_LOGIC\_VECTOR (3 downto 0);

component RoundEncryptDecrypt

port ( ROUND\_KEY : in STD\_LOGIC\_VECTOR(0 to 127);

ROUND\_NUMBER : in STD\_LOGIC\_VECTOR(3 downto 0);

INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

FB\_INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

INIT\_INPUT : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127) );

end component;

component KeyScheduleRAM

port ( CLK : in std\_logic;

WE : in std\_logic;

ADDRESS : STD\_LOGIC\_VECTOR (3 downto 0);

INPUT : in std\_logic\_vector(0 to 127);

OUTPUT : out std\_logic\_vector(0 to 127) );

end component;

component Reg\_128\_EN

port( CLK : in std\_logic;

EN : in std\_logic;

D : in std\_logic\_vector(0 to 127);

Q : out std\_logic\_vector(0 to 127) );

end component;

component MUX\_128\_2

port ( IN\_0 : in STD\_LOGIC\_VECTOR(0 to 127);

IN\_1 : in STD\_LOGIC\_VECTOR(0 to 127);

SEL : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component Counter\_to\_10

port ( CLK : in STD\_LOGIC;

CLR : in STD\_LOGIC;

CE : in STD\_LOGIC;

Q : out STD\_LOGIC\_VECTOR (3 downto 0)

);

end component;

component KeySchedule

port ( prev\_key : in STD\_LOGIC\_VECTOR(0 to 127);

round\_number : in STD\_LOGIC\_VECTOR (3 downto 0);

round\_key : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

signal reg\_DATA\_WRITE, RESET\_reg, tmp\_AVAILABLE,

enable\_output\_reg, enable\_core\_output\_reg,

enable\_input\_reg : STD\_LOGIC := '0';

signal rk\_to\_re\_d, input\_reg\_output, w2, w3:

STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal s1, s2, s3: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal control\_enable\_reg: STD\_LOGIC := '0';

signal control\_start\_counter: STD\_LOGIC := '0';

signal control\_write\_zero\_key: STD\_LOGIC := '0';

signal control\_write\_ram: STD\_LOGIC := '0';

signal control\_initialized: STD\_LOGIC := '0';

signal control\_input: STD\_LOGIC := '0';

signal control\_data\_available: STD\_LOGIC := '0';

begin

RES: process(CLK)

begin

if (CLK'event and CLK = '1') then

RESET\_reg <= RESET;

end if;

end process;

D\_W: process(CLK)

begin

if (CLK'event and CLK = '1') then

reg\_DATA\_WRITE <= DATA\_WRITE;

end if;

end process;

enable\_core\_output\_reg <= control\_initialized;

enable\_output\_reg <= tmp\_AVAILABLE;

enable\_input\_reg <= reg\_DATA\_WRITE;

tmp\_AVAIL: process(CLK)

begin

if (CLK'event and CLK = '1') then

tmp\_AVAILABLE <= control\_data\_available;

end if;

end process;

AVAIL: process(CLK)

begin

if (CLK'event and CLK = '1') then

AVAILABLE <= tmp\_AVAILABLE;

end if;

end process;

main1: KeyScheduleRAM port map (

CLK => CLK,

WE => control\_write\_ram,

ADDRESS => counter,

INPUT => s1,

OUTPUT => rk\_to\_re\_d

);

input\_reg: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_input\_reg,

D => INPUT\_TEXT,

Q => input\_reg\_output

);

main2: RoundEncryptDecrypt port map (

ROUND\_KEY => rk\_to\_re\_d,

ROUND\_NUMBER => counter,

INPUT => input\_reg\_output,

FB\_INPUT => w3,

INIT\_INPUT => control\_input,

OUTPUT => w2

);

main3: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_core\_output\_reg,

D => w2,

Q => w3

);

main4: Counter\_to\_10 port map (

CLK => CLK,

CLR => RESET\_reg,

CE => control\_start\_counter,

Q => counter

);

wrk0: KeySchedule port map (

prev\_key => s1,

round\_number => counter,

round\_key => s2

);

wrk1: MUX\_128\_2 port map (

IN\_0 => s2,

IN\_1 => input\_reg\_output,

SEL => control\_write\_zero\_key,

OUTPUT => s3

);

wrk2: Reg\_128\_EN port map (

CLK => CLK,

EN => control\_enable\_reg,

D => s3,

Q => s1

);

ouput\_reg: Reg\_128\_EN port map (

CLK => CLK,

EN => enable\_output\_reg,

D => w3,

Q => OUTPUT\_TEXT

);

FSM: process (state, reg\_DATA\_WRITE, counter)

begin

control\_enable\_reg <= '0';

control\_start\_counter <= '0';

control\_write\_zero\_key <= '0';

control\_write\_ram <= '0';

control\_initialized <= '0';

control\_input <= '0';

control\_data\_available <= '0';

case (state) is

when i\_0 =>

if reg\_DATA\_WRITE = '1' then

next\_state <= i\_1;

else

next\_state <= i\_0;

end if;

when i\_1 =>

control\_enable\_reg <= '1';

control\_write\_zero\_key <= '1';

next\_state <= i\_2;

when i\_2 =>

control\_write\_ram <= '1';

control\_start\_counter <= '1';

if counter = "1010" then

control\_data\_available <= '1';

next\_state <= s\_0;

else

control\_enable\_reg <= '1';

next\_state <= i\_2;

end if;

when s\_0 =>

if reg\_DATA\_WRITE = '1' then

control\_start\_counter <= '1';

next\_state <= s\_0\_2;

else

next\_state <= s\_0;

end if;

when s\_0\_2 =>

control\_initialized <= '1';

control\_input <= '1';

control\_start\_counter <= '1';

next\_state <= s\_1;

when s\_1 =>

control\_initialized <= '1';

if counter = "0000" then

control\_data\_available <= '1';

if reg\_DATA\_WRITE = '1' then

control\_start\_counter <= '1';

next\_state <= s\_0\_2;

else

next\_state <= s\_0;

end if;

else

control\_start\_counter <= '1';

next\_state <= s\_1;

end if;

end case;

end process;

SYNC\_FSM: process(CLK)

begin

if rising\_edge(CLK) then

if RESET\_reg = '1' then

state <= i\_0;

else

state <= next\_state;

end if;

end if;

end process;

end Main\_architecture;

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity RoundEncryptDecrypt is

port ( ROUND\_KEY : in STD\_LOGIC\_VECTOR(0 to 127);

ROUND\_NUMBER : in STD\_LOGIC\_VECTOR(3 downto 0);

INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

FB\_INPUT : in STD\_LOGIC\_VECTOR(0 to 127);

INIT\_INPUT : in STD\_LOGIC;

OUTPUT : out STD\_LOGIC\_VECTOR(0 to 127)

);

end RoundEncryptDecrypt;

architecture RoundEncryptDecrypt\_architecture of RoundEncryptDecrypt is

component SubBytes

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component ShiftRows

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component MixColumns

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

component AddRoundKey

Port ( input : in STD\_LOGIC\_VECTOR(0 to 127);

round\_key : in STD\_LOGIC\_VECTOR(0 to 127);

output : out STD\_LOGIC\_VECTOR(0 to 127)

);

end component;

signal b0, b1, b2: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

signal m1, m2: STD\_LOGIC\_VECTOR(0 to 127) := (others => '0');

begin

en\_1: SubBytes port map (

input => FB\_INPUT,

output => b0

);

en\_2: ShiftRows port map (

input => b0,

output => b1

);

en\_3: MixColumns port map (

input => b1,

output => b2

);

en\_4: process(ROUND\_NUMBER, b1, b2)

begin

case ROUND\_NUMBER is

when "0000" => m1 <= b1;

when others => m1 <= b2;

end case;

end process;

en\_5: process(INIT\_INPUT, INPUT, m1)

begin

case INIT\_INPUT is

when '1' => m2 <= INPUT;

when others => m2 <= m1;

end case;

end process;

en\_6: AddRoundKey port map (

input => m2,

round\_key => ROUND\_KEY,

output => OUTPUT

);

end RoundEncryptDecrypt\_architecture;

**3 Описание теста для процессора зашифрования**

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

ENTITY main\_test IS

END main\_test;

ARCHITECTURE behavior OF main\_test IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT Main

PORT(

CLK : IN std\_logic;

RESET : IN std\_logic;

DATA\_WRITE : IN std\_logic;

INPUT\_TEXT : IN std\_logic\_vector(0 to 127);

AVAILABLE : OUT std\_logic;

OUTPUT\_TEXT : OUT std\_logic\_vector(0 to 127)

);

END COMPONENT;

--Inputs

signal CLK : std\_logic := '0';

signal RESET : std\_logic := '0';

signal DATA\_WRITE : std\_logic := '0';

signal INPUT\_TEXT : std\_logic\_vector(0 to 127) := (others => '0');

--Outputs

signal AVAILABLE : std\_logic;

signal OUTPUT\_TEXT : std\_logic\_vector(0 to 127);

-- Clock period definitions

constant CLK\_period : time := 3 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)

uut: Main PORT MAP (

CLK => CLK,

RESET => RESET,

DATA\_WRITE => DATA\_WRITE,

INPUT\_TEXT => INPUT\_TEXT,

AVAILABLE => AVAILABLE,

OUTPUT\_TEXT => OUTPUT\_TEXT

);

-- Clock process definitions

CLK\_process :process

begin

CLK <= '0';

wait for CLK\_period/2;

CLK <= '1';

wait for CLK\_period/2;

end process;

-- Stimulus process

stim\_proc: process

begin

RESET <= '1';

wait for 100 ns;

RESET <= '0';

wait for CLK\_period\*3;

wait until (CLK'event and CLK = '1');

DATA\_WRITE <= '1';

INPUT\_TEXT <= X"000102030405060708090a0b0c0d0e0f";

wait for CLK\_period;

DATA\_WRITE <= '0';

wait until AVAILABLE = '1';

wait for CLK\_period\*1;

wait until (CLK'event and CLK = '1');

INPUT\_TEXT <= X"00112233445566778899aabbccddeeff";

DATA\_WRITE <= '1';

wait on AVAILABLE;

wait on AVAILABLE;

wait until (CLK'event and CLK = '1');

DATA\_WRITE <= '0';

wait for CLK\_period\*15;

wait until (CLK'event and CLK = '1');

DATA\_WRITE <= '1';

wait for CLK\_period\*1;

DATA\_WRITE <= '0';

wait;

end process;

END;

**4 Описание теста для процессора зашифрования и расшифрования**

LIBRARY ieee;

USE ieee.std\_logic\_1164.ALL;

ENTITY main\_test IS

END main\_test;

ARCHITECTURE behavior OF main\_test IS

-- Component Declaration for the Unit Under Test (UUT)

COMPONENT Main

PORT(

CLK : IN std\_logic;

RESET : IN std\_logic;

ENCR\_DECR : in STD\_LOGIC;

DATA\_WRITE : IN std\_logic;

INPUT\_TEXT : IN std\_logic\_vector(0 to 127);

AVAILABLE : OUT std\_logic;

OUTPUT\_TEXT : OUT std\_logic\_vector(0 to 127)

);

END COMPONENT;

--Inputs

signal CLK : std\_logic := '0';

signal RESET : std\_logic := '0';

signal ENCR\_DECR : std\_logic := '0';

signal DATA\_WRITE : std\_logic := '0';

signal INPUT\_TEXT : std\_logic\_vector(0 to 127) := (others => '0');

--Outputs

signal AVAILABLE : std\_logic;

signal OUTPUT\_TEXT : std\_logic\_vector(0 to 127);

-- Clock period definitions

constant CLK\_period : time := 3 ns;

BEGIN

-- Instantiate the Unit Under Test (UUT)

uut: Main PORT MAP (

CLK => CLK,

RESET => RESET,

ENCR\_DECR => ENCR\_DECR,

DATA\_WRITE => DATA\_WRITE,

INPUT\_TEXT => INPUT\_TEXT,

AVAILABLE => AVAILABLE,

OUTPUT\_TEXT => OUTPUT\_TEXT

);

-- Clock process definitions

CLK\_process :process

begin

CLK <= '0';

wait for CLK\_period/2;

CLK <= '1';

wait for CLK\_period/2;

end process;

-- Stimulus process

stim\_proc: process

begin

RESET <= '1';

wait for 100 ns;

RESET <= '0';

wait for CLK\_period;

wait until (CLK'event and CLK = '1');

wait for CLK\_period/20;

DATA\_WRITE <= '1';

INPUT\_TEXT <= X"000102030405060708090a0b0c0d0e0f";

wait for CLK\_period;

DATA\_WRITE <= '0';

wait until AVAILABLE = '1';

wait for CLK\_period\*1;

wait until (CLK'event and CLK = '1');

wait for CLK\_period/20;

ENCR\_DECR <= '1';

INPUT\_TEXT <= X"00112233445566778899aabbccddeeff";

DATA\_WRITE <= '1';

wait on AVAILABLE;

wait on AVAILABLE;

DATA\_WRITE <= '0';

wait for CLK\_period\*15;

wait until (CLK'event and CLK = '1');

wait for CLK\_period/20;

DATA\_WRITE <= '1';

wait for CLK\_period\*1;

DATA\_WRITE <= '0';

wait on AVAILABLE;

wait on AVAILABLE;

wait for CLK\_period\*15;

wait until (CLK'event and CLK = '1');

wait for CLK\_period/20;

DATA\_WRITE <= '1';

INPUT\_TEXT <= X"69c4e0d86a7b0430d8cdb78070b4c55a";

ENCR\_DECR <= '0';

wait for CLK\_period\*1;

DATA\_WRITE <= '0';

wait;

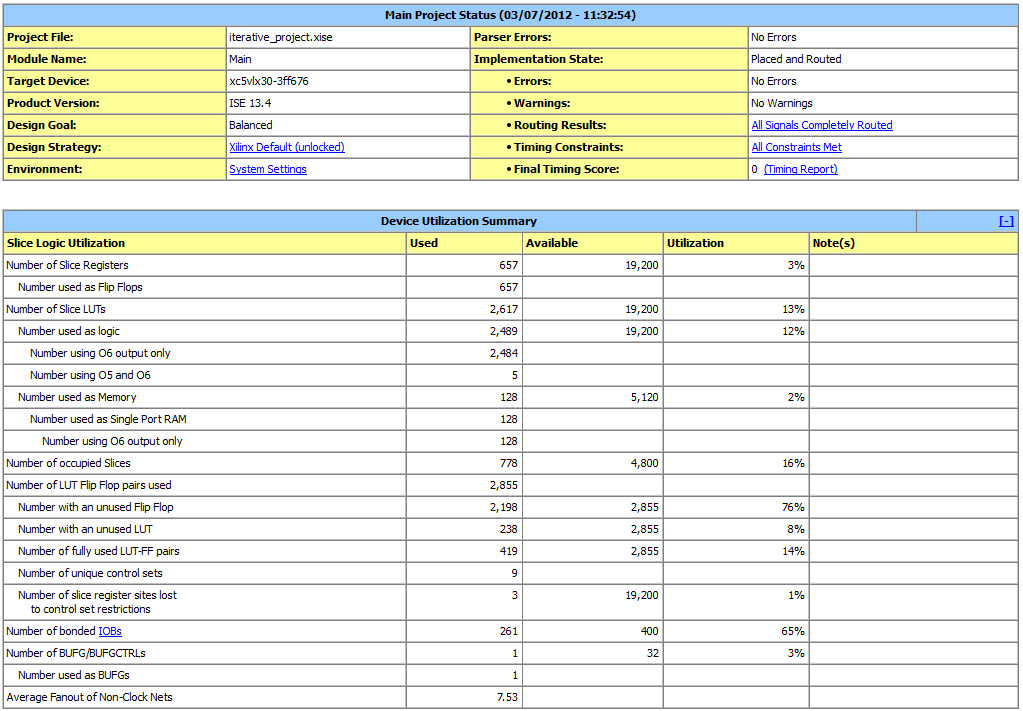
end process;

END;

**ПРИЛОЖЕНИЕ В**

**(обязательное)**

**Отчёты Xilinx ISE**

****

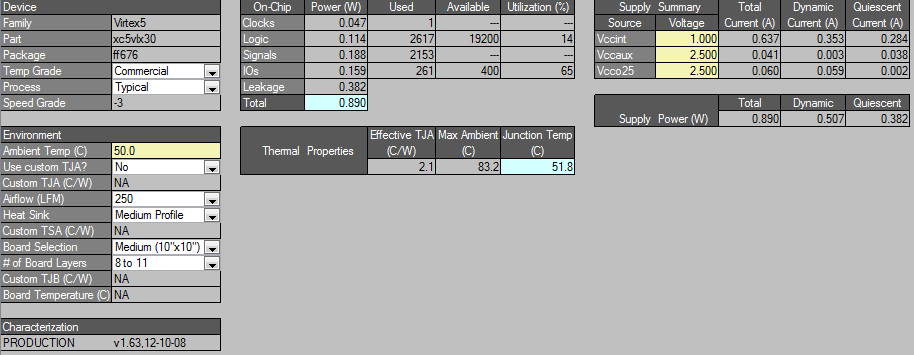
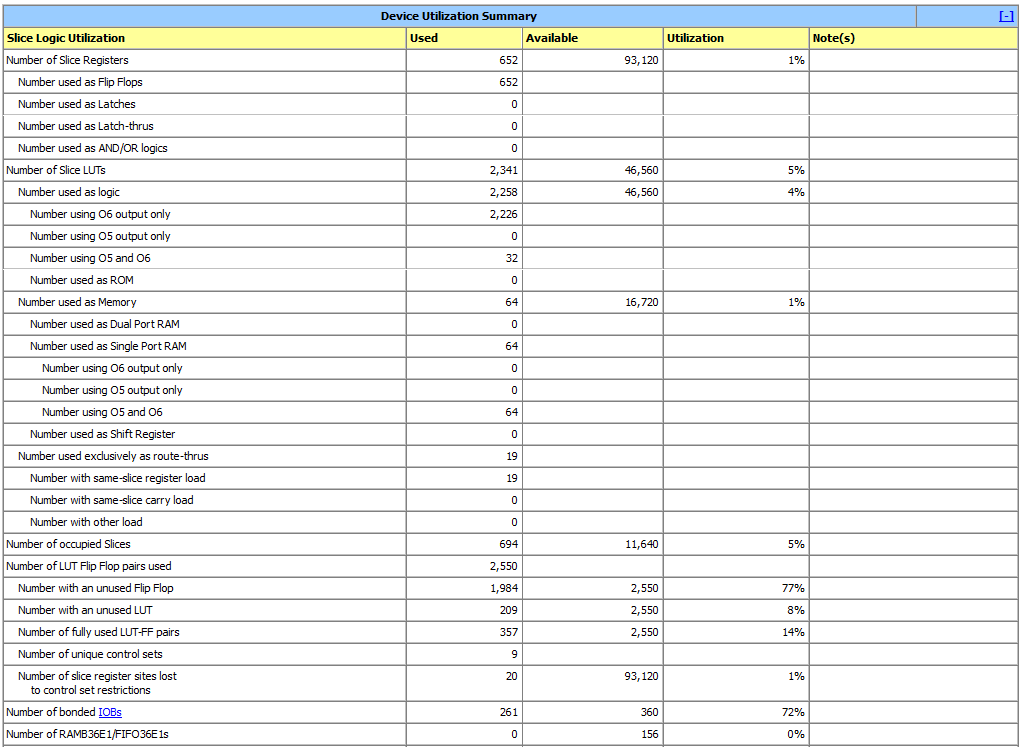
****

Рисунок В.1 – Отчёты Xilinx ISE для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для

Virtex 5 xc5vlx30-ff676-3 (-3)

****

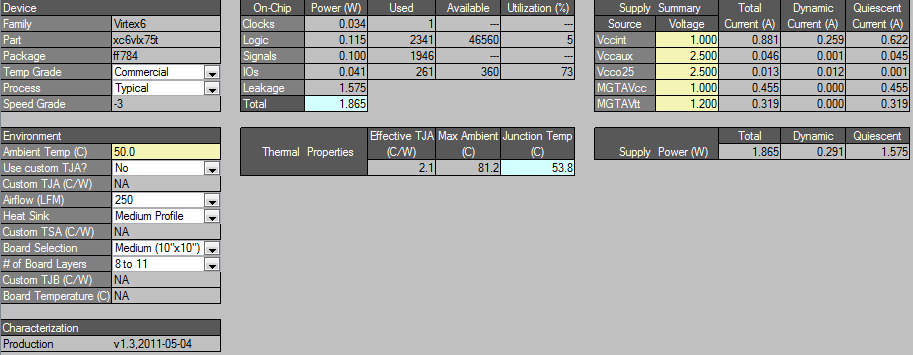
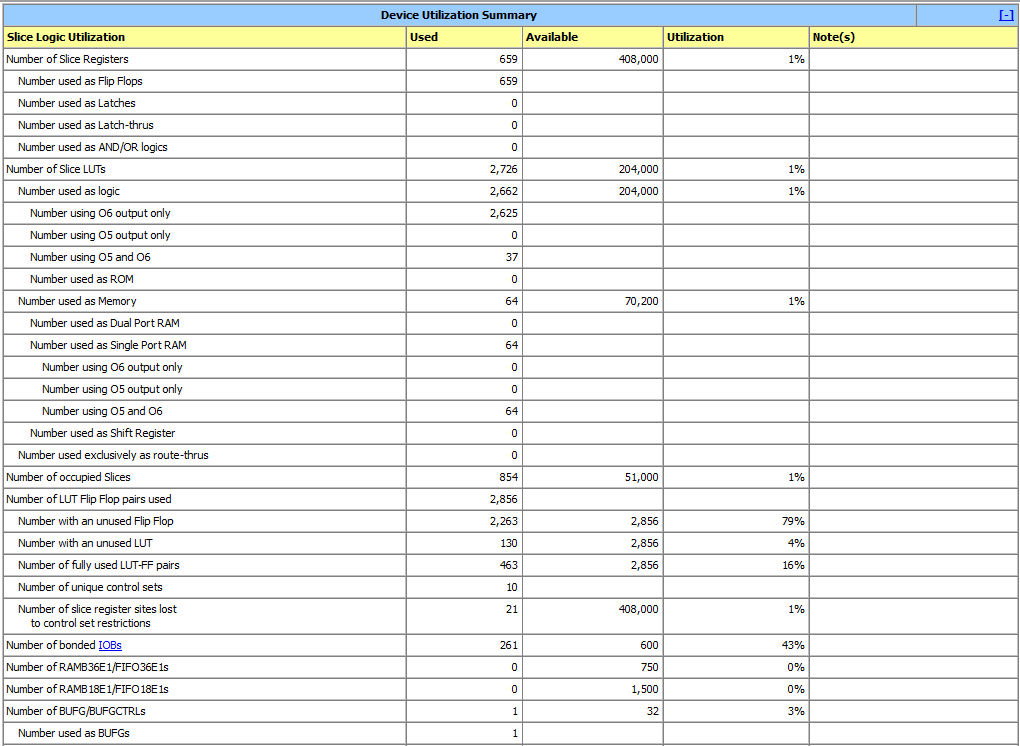
****

Рисунок В.2 – Отчёты Xilinx ISE для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для

Virtex 6 xc6vlx75t-ff784-3 (-3)

****

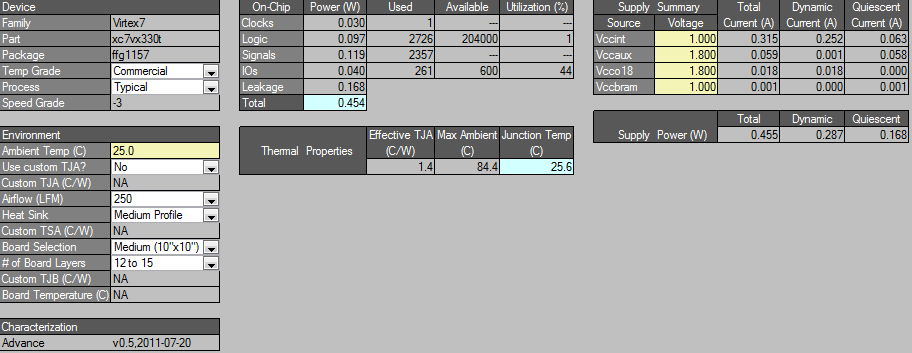
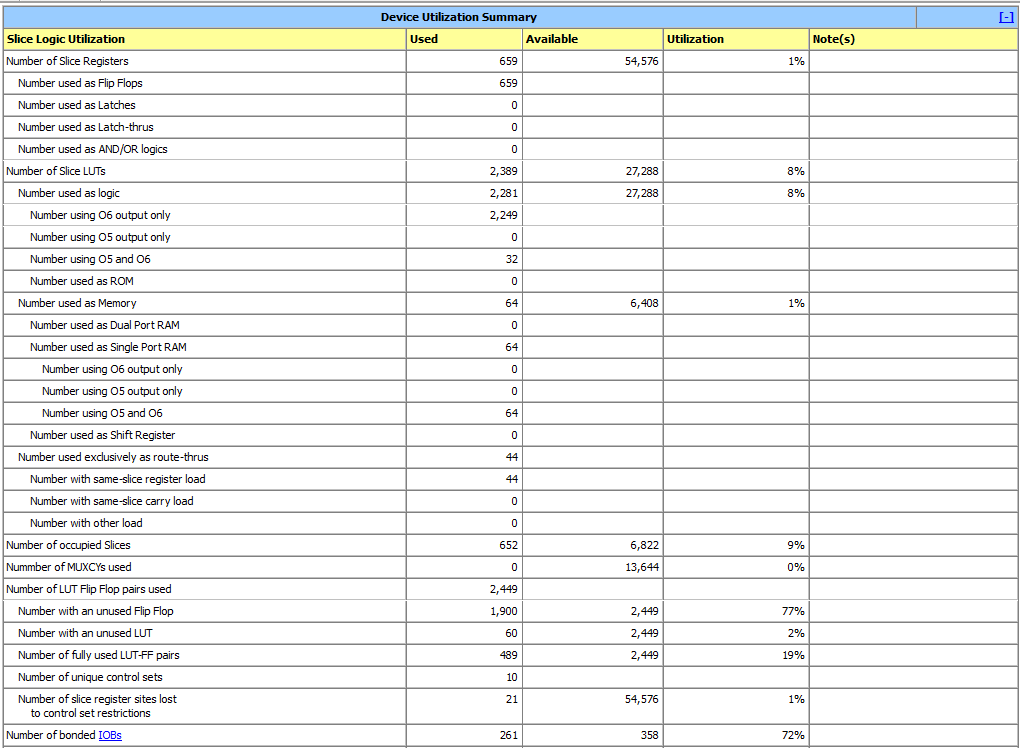
****

Рисунок В.3 – Отчёты Xilinx ISE для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для

Virtex 7 xc7vx330t-ffg1157-3 (-3)

****

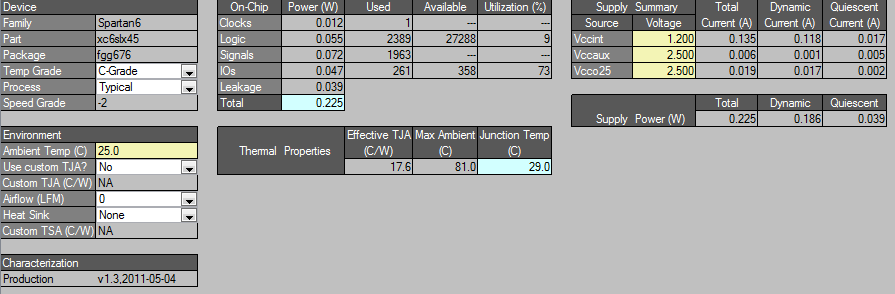
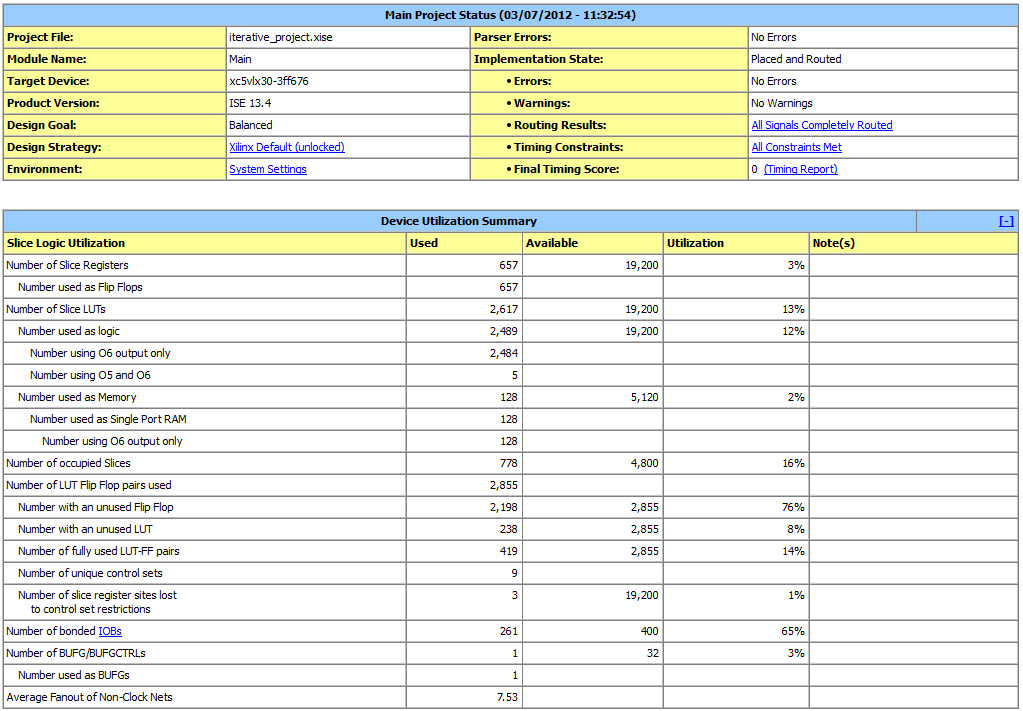
****

Рисунок В.4 – Отчёты Xilinx ISE для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для

Spartan 6 xc6slx45-fgg676-2 (-2)

****

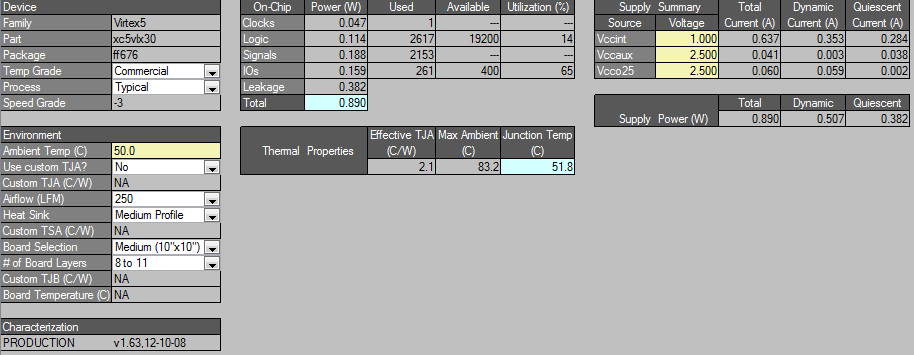
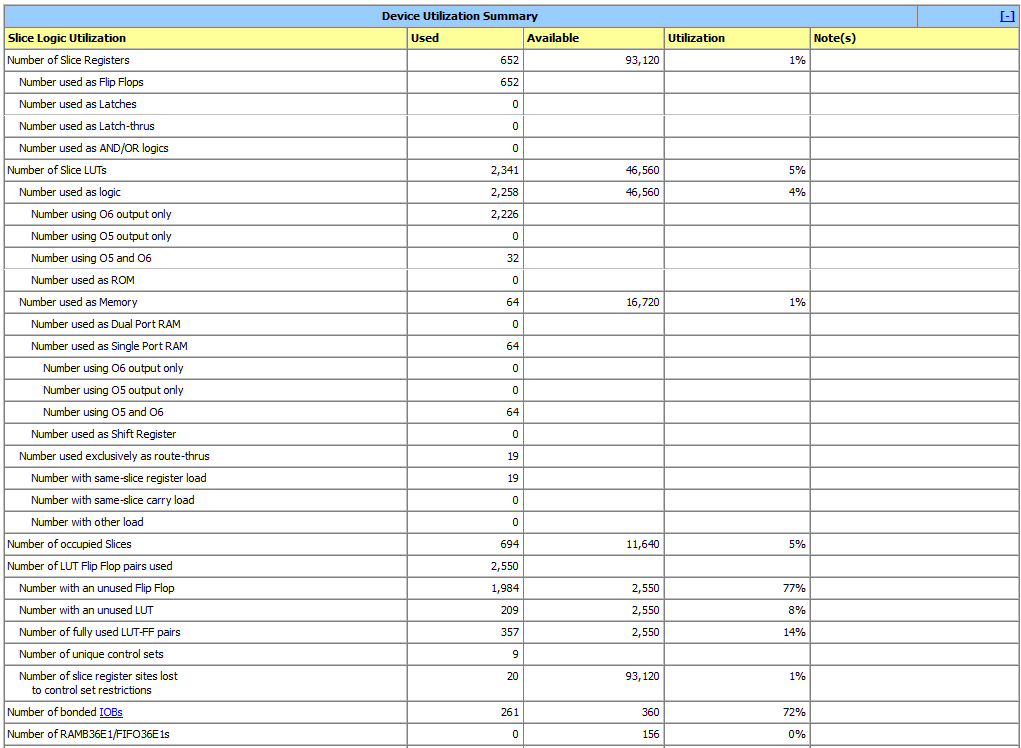
****

Рисунок В.5 – Отчёты Xilinx ISE для 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью круговых ключей для Virtex 5 xc5vlx30-ff676-3 (-3)

****

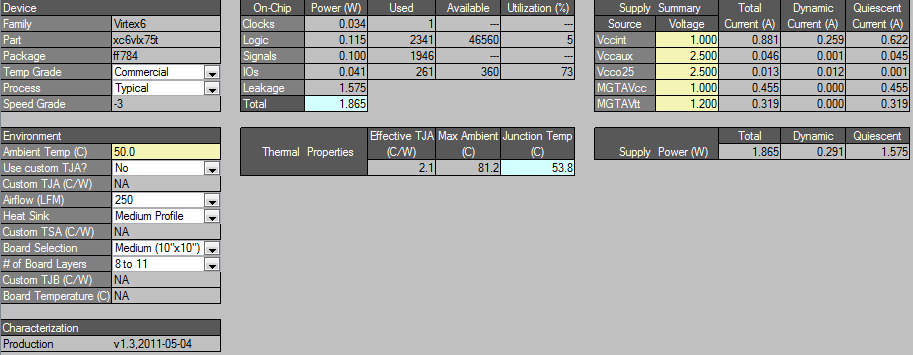
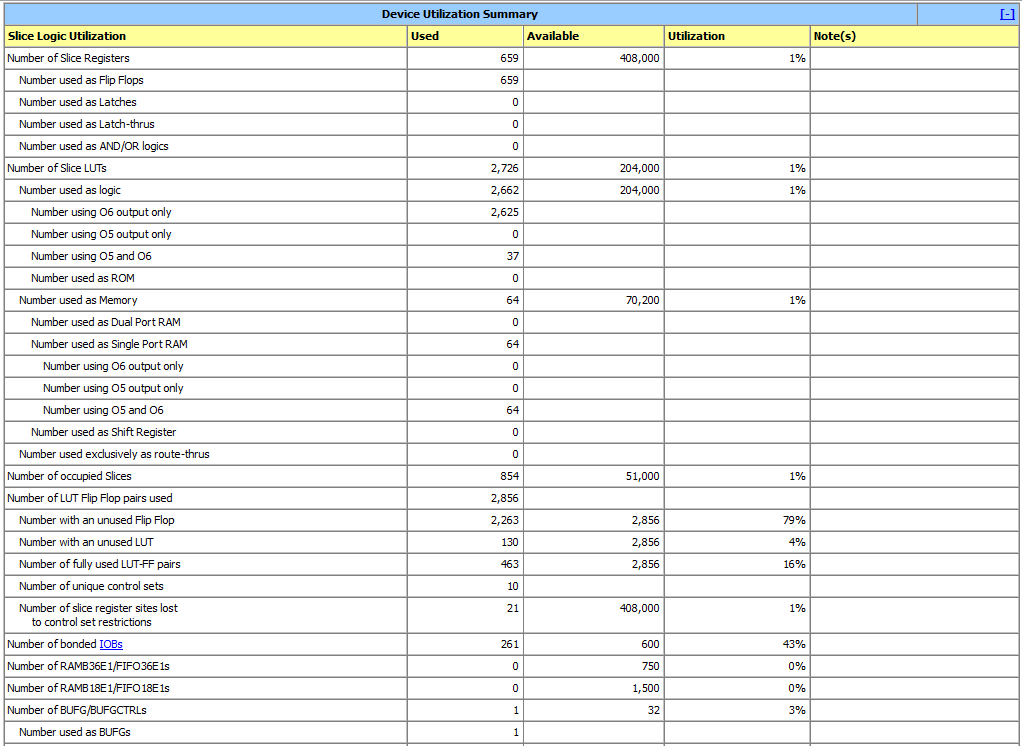
****

Рисунок В.6 – Отчёты Xilinx ISE для 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью круговых ключей для Virtex 6 xc6vlx75t-ff784-3 (-3)

****

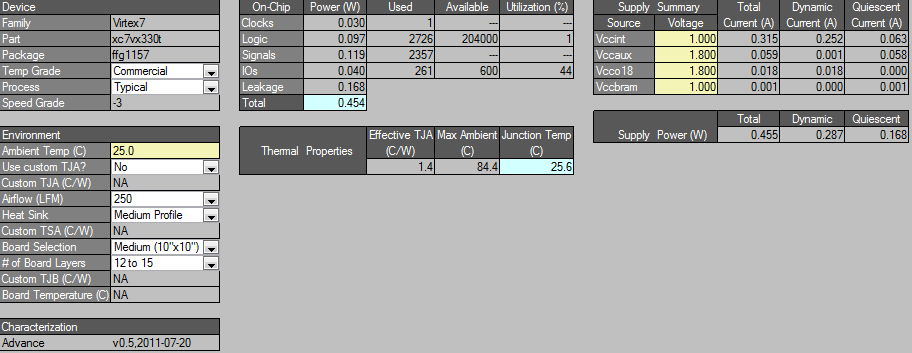
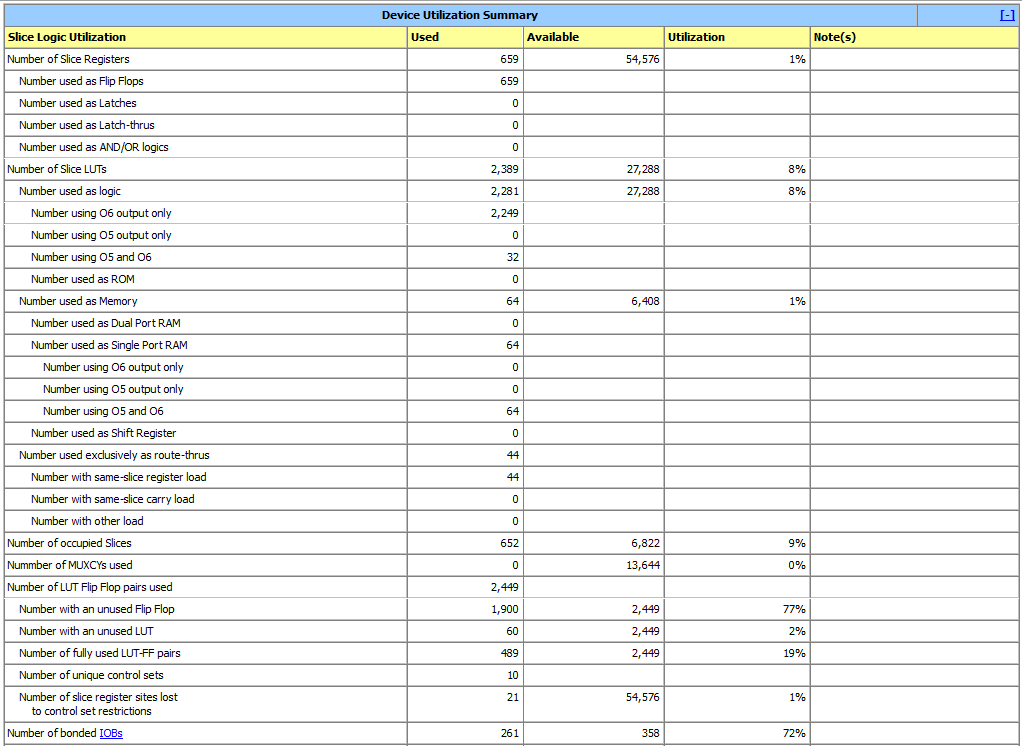
****

Рисунок В.7 – Отчёты Xilinx ISE для 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью круговых ключей для Virtex 7 xc7vx330t-ffg1157-3 (-3)

****

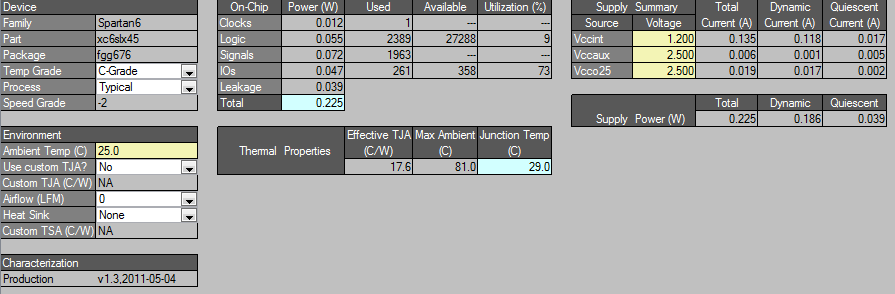
****

Рисунок В.8 – Отчёты Xilinx ISE для 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью круговых ключей для Spartan 6 xc6slx45-fgg676-2 (-2)

**ПРИЛОЖЕНИЕ Г**

**(обязательное)**

**Результаты тестирования**



Рисунок Г.1 – Временные диаграммы теста 10-тактового процессора зашифрования с прямой реализацией функции MixColumns и отдельным блоком AddRoundKey для обработки нулевого раунда для Virtex 5 (-3)

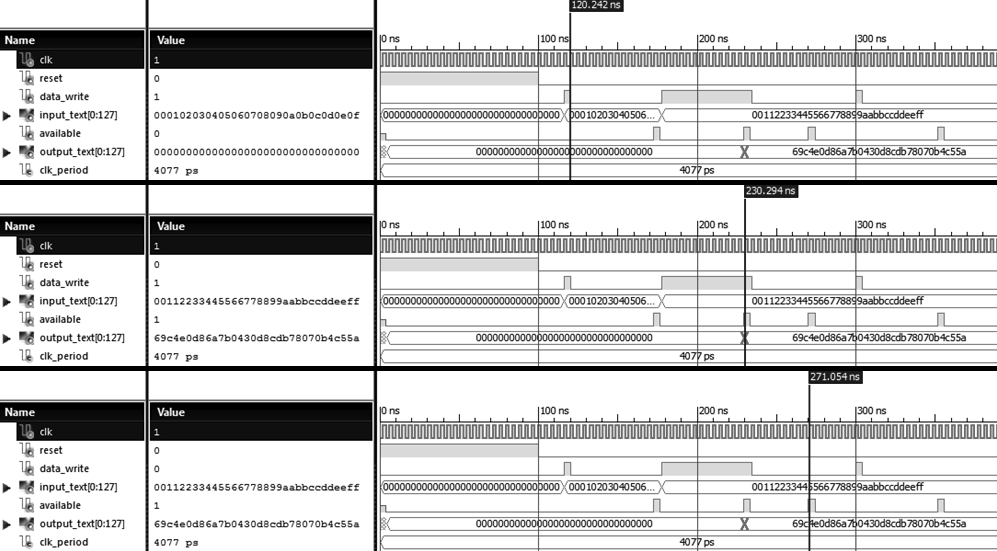


Рисунок Г.2 – Временные диаграммы теста 10-тактового процессора зашифрования с прямой реализацией функции MixColumns и отдельным блоком AddRoundKey для обработки десятого раунда для Virtex 5 (-3)

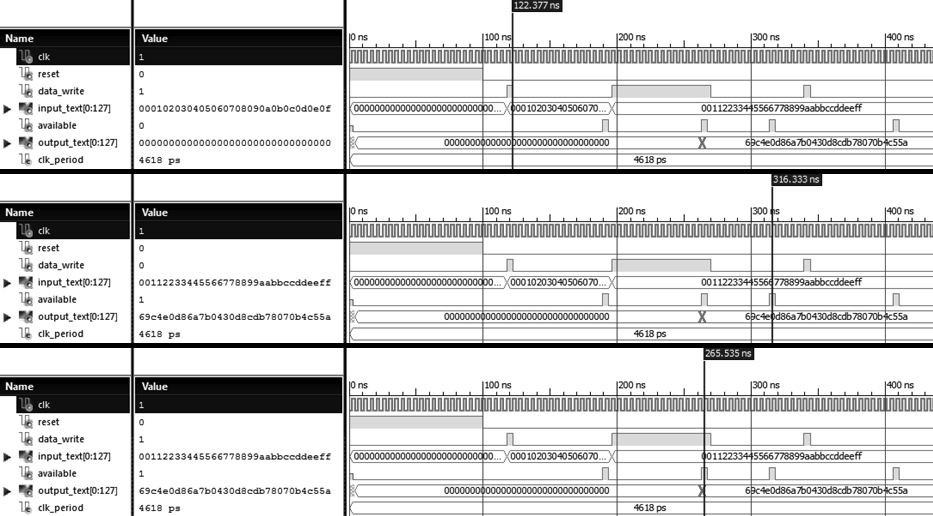


Рисунок Г.3 – Временные диаграммы теста 11-тактового процессора зашифрования на архитектуре с использованием Т-таблиц на базе однопортовой ROM-памяти *на блочной памяти* для Virtex 5 (-3)



Рисунок Г.4 – Временные диаграммы теста 11-тактового процессора зашифрования на архитектуре с использованием Т-таблиц на базе однопортовой ROM-памяти *на распределённой памяти* для Virtex 5 (-3)

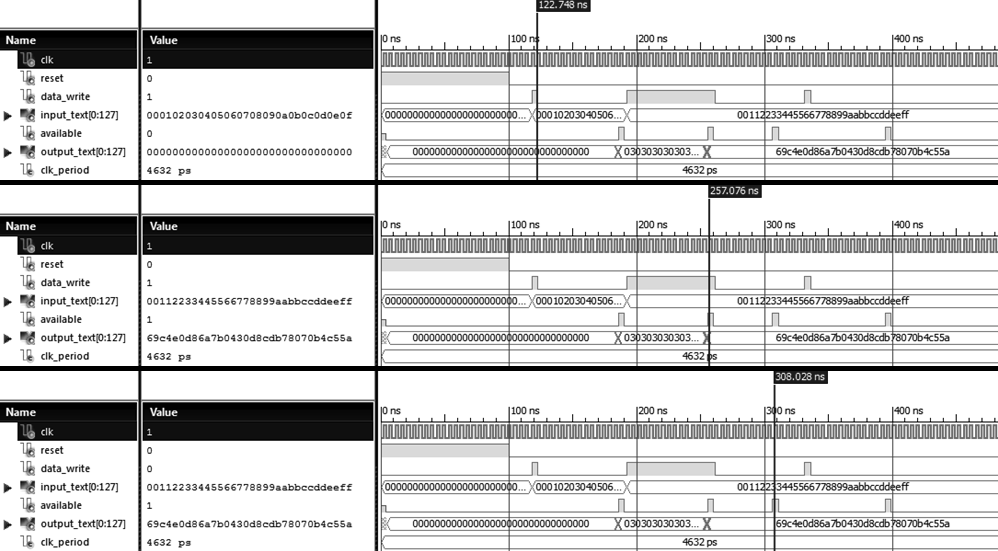


Рисунок Г.5 – Временные диаграммы теста 11-тактового процессора зашифрования на архитектуре с использованием Т-таблиц на базе двухпортовой ROM-памяти *на блочной памяти* для Virtex 5 (-3)

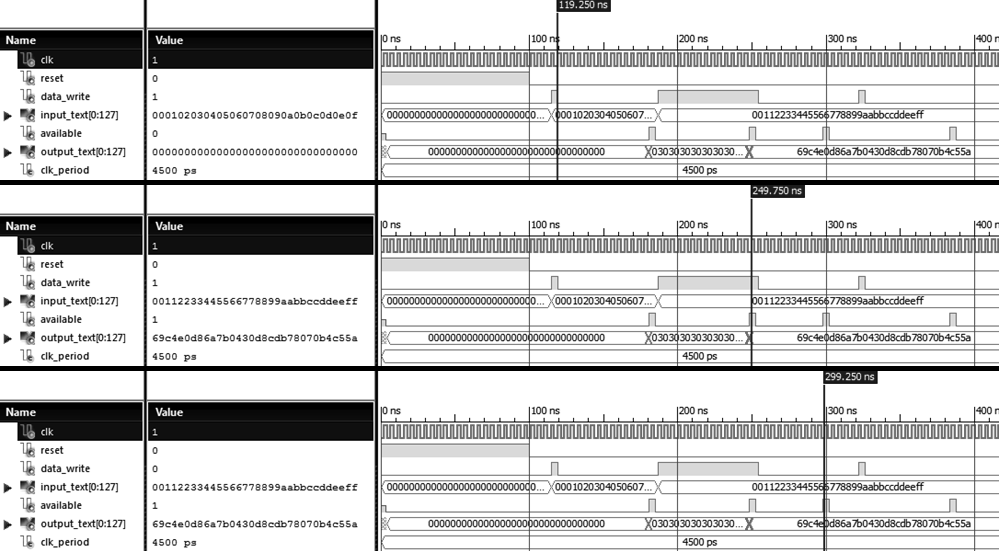


Рисунок Г.6 – Временные диаграммы теста 11-тактового процессора зашифрования на архитектуре с использованием Т-таблиц на базе двухпортовой ROM-памяти *на распределённой памяти* для Virtex 5 (-3)



Рисунок Г.7 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью раундовых ключей *на блочной памяти* для Virtex 5 (-3)

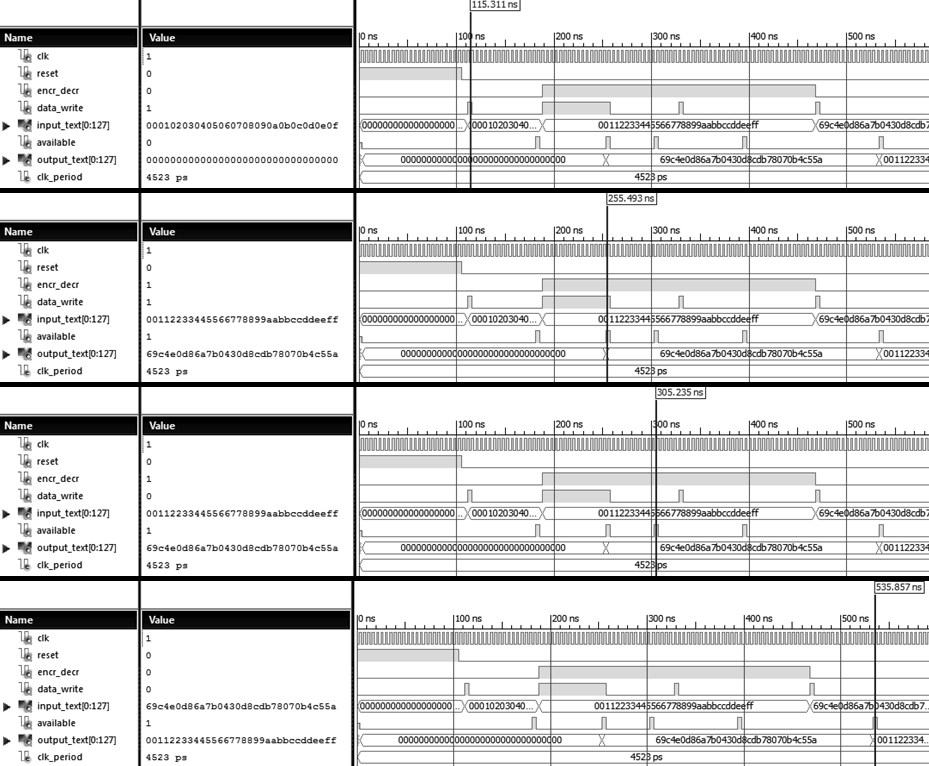
****

Рисунок Г.8 – Временная диаграмма для процессора зашифрования и расшифрования с асинхронной памятью раундовых ключей для Virtex 5 (-3)

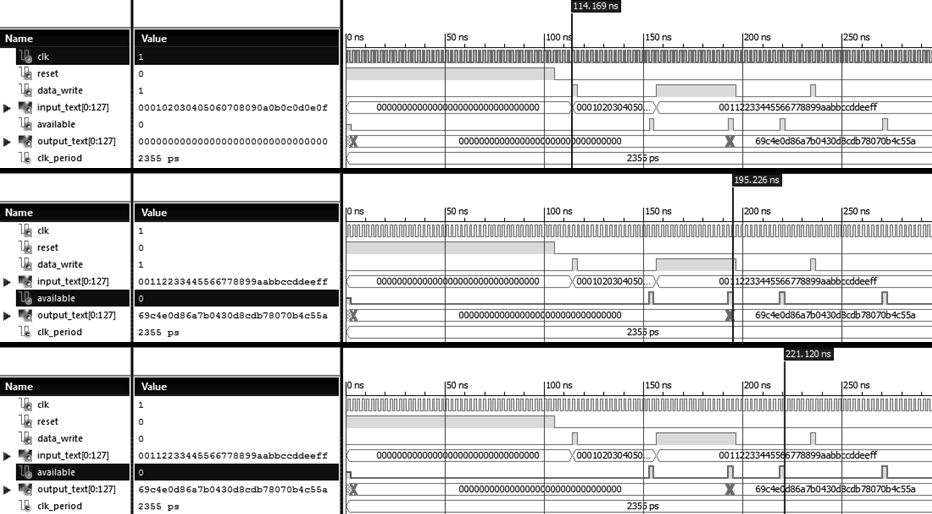


Рисунок Г.9 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью раундовых ключей *на распределённой памяти* для Virtex 6 (-3)



Рисунок Г.10 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью раундовых ключей *на распределённой памяти* для Virtex 7 (-3)

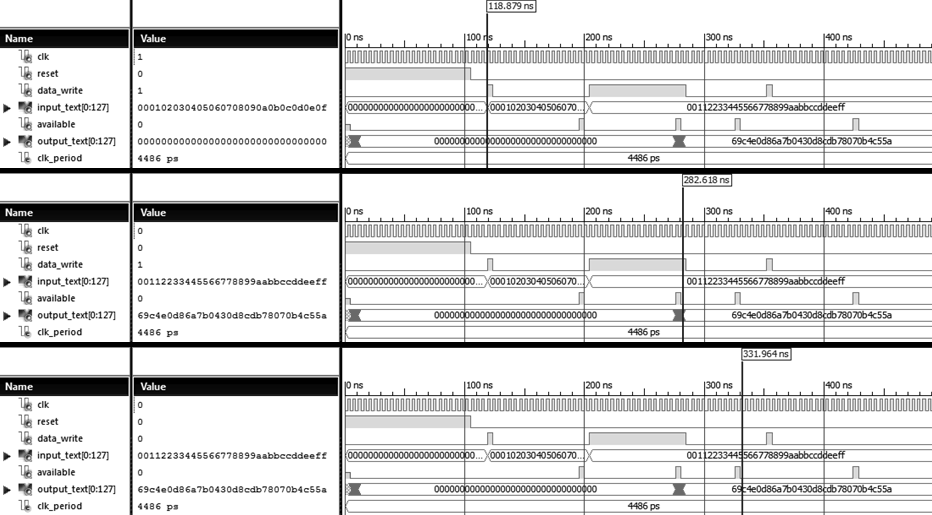


Рисунок Г.11 – Временные диаграммы теста 11-тактового процессора зашифрования с прямой реализацией функции MixColumns и синхронной памятью раундовых ключей *на распределённой памяти* для Spartan 6 (-2)

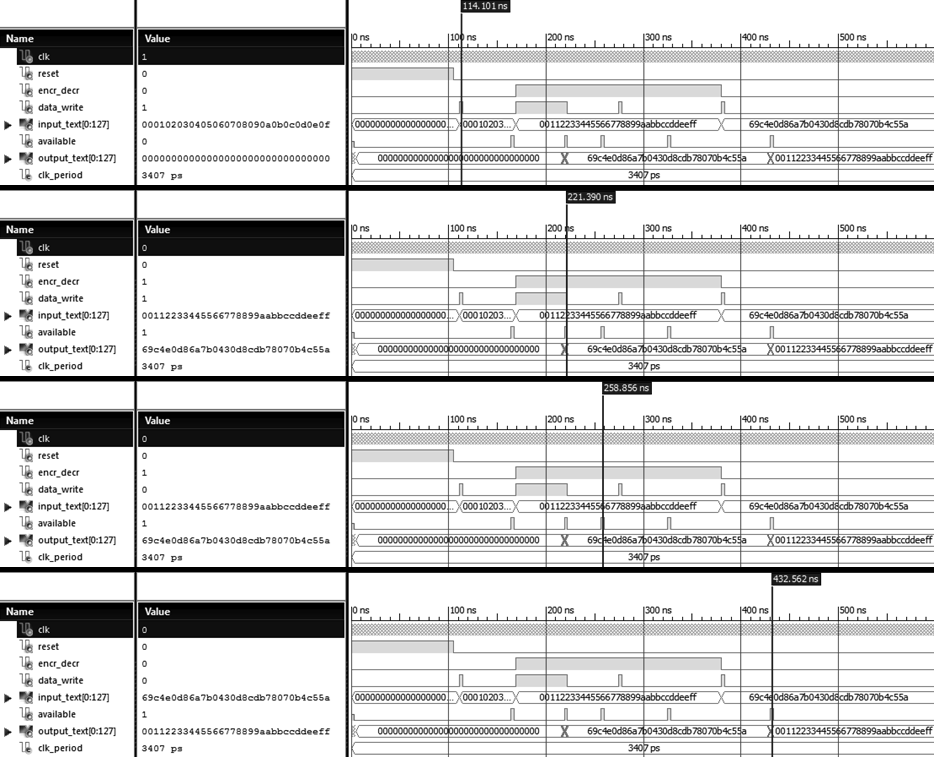
****

Рисунок Г.12 – Временная диаграмма для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для Virtex 6 (-3)

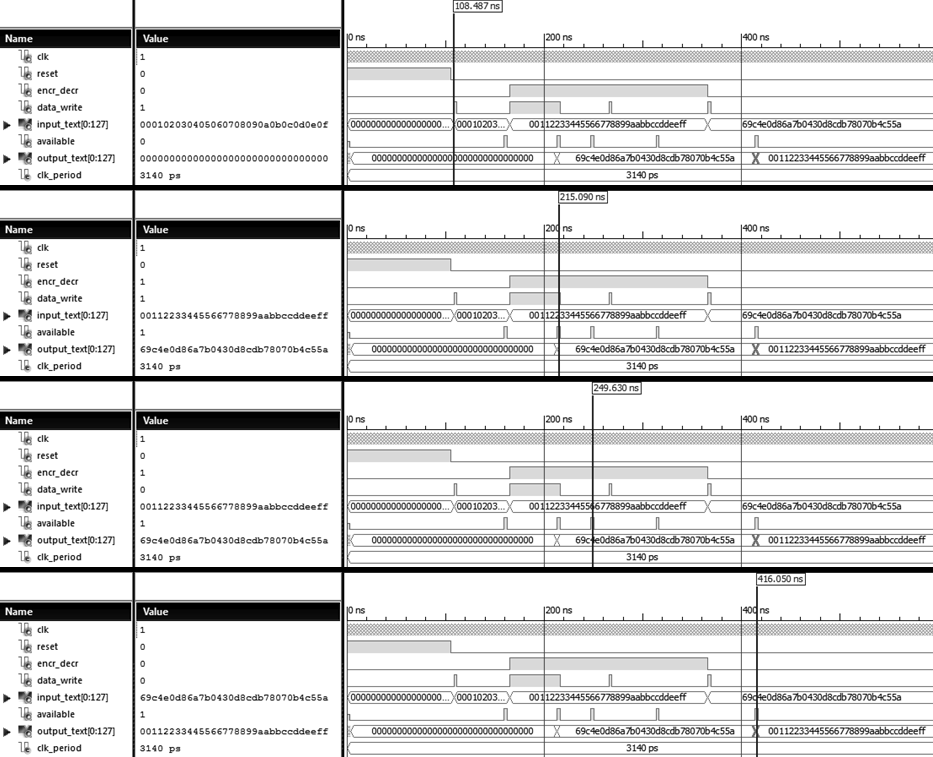
****

Рисунок Г.13 – Временная диаграмма для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для Virtex 7 (-3)

****

Рисунок Г.14 – Временная диаграмма для процессора зашифрования и расшифрования с синхронной памятью раундовых ключей для Spartan 6 (-2)