Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
«Сыктывкарский государственный университет имени Питирима Сорокина»
Институт точных наук и информационных технологий
Кафедра информационной безопасности

Допустить к защите
Зав. кафедрой информационной безопасности,
к. ф.-м. н., доцент
_____ Л. С. Носов
«____»_____ 2017 года

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Разработка аппаратного средства шифрования на основе ПЛИС.

Научный руководитель:
к. ф.-м. н., доцент _____ Л. С. Носов
«____»_____ 2017 г.

Исполнитель:
Студент 143 группы _____ Е. Ю. Пипуныров
«____»_____ 2017 г.

Сыктывкар 2017

Содержание

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей работе применяются следующие сокращения и определения с соответсвующими значениями:

HDL — hardware describe language;

RAM — random access memory;

Random access memory — вид памяти, в котором можно единовременно получить доступ к любой ячейке;

Read only memory — вид памяти, к которой возможен только доступ на чтение;

RISC-архитектура— архитектура процессора, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим;

RISK — reduced instruction set computer;

ROM — read only memory;

UART — universal asynchronous receiver-transmitter.

Universal asynchronous receiver-transmitter — устройство, преобразующее передаваемые данные в последовательный вид так, чтобы было возможно передать их по одной физической цифровой линии другому аналогичному устройству.

АЛУ — арифметико-логическое устройство;

Арифметико-логическое устройство — блок процессора, выполнений арифметические и логические преобрания над данными;

Архитектура ЦПУ — набор типов данных, операций и регистров, доступных для программирования;

АС — аппаратное средство;

Базовый алгоритм шифрования — блочный шифр, реализующий обратимое отображение блоков открытого текста в блоки шифртекста;

ВКР — выпускнаая квалификационная работа;

ЗИ — защита информации;

Интерфейс — совокупность средств, методов и правил взаимодействия между элементами системы;

Ключ шифрования — изменяемый параметр в виде последовательности символов, определяющий криптографическое преобразование;

КМЗИ — криптографические методы ЗИ;

Контроллер — блок процессора, генерирующее сигналы управления трактом данных процессора;

ЛЭ — логические элементы;

Микроархитектура ЦПУ — способ, которым данная архитектура реализована в процессоре;

ПЛИС — программируемые логические интегральные схемы;

Потоковый шифратор — устройство, в котором каждый символ открытого текста преобразуется в символ шифрованного текста в зависимости не только от используемого ключа, но и от его расположения в потоке открытого текста, в котором нет необходимости в дополнении текста до определённой длины;

САПР — система автоматического проектирования;

Система автоматического проектирования — система. предназначенная для автоматизации процесса проектирования и разработки;

Тракт данных — часть микроархитектуры, осуществляющая преобразования данных;

ЦПУ — центральное процессорное устройство;

ВВЕДЕНИЕ

Постоянное развитие технологий приводит к повсеместному применению электронных устройств и постоянному обмену информацией между ними. Утечка данной информации может послужить источником угроз приватности и даже общественной безопасности. Важность передаваемой информации стимулирует массовое применение средств защиты. Самым распространённым средством защиты передачи данных является криптографический метод. Большинство систем используют широко распространённые, стандартизованные протокола передачи данных, однако иногда возникает необходимость обеспечить безопасность передачи данных, циркулирующих в нестандартных системах, для которых не существует "шаблонных" готовых решений. Разработка систем защиты для нестандартных решений "с нуля" может отнять много времени, сил, денег и других ресурсов. Помимо того, криптографические методы защиты информации (КМЗИ) хоть и повышают защищённость данных, однако, требуют большого количества дополнительных вычислений, нагружающих аппаратуру. Стандартные микропроцессорные устройства имеют общий, неспециализированный набор команд, использование которых снижает скорость обработки данных.

В данных условиях было решено разработать вариант самостоятельного аппаратного средства (АС) защиты информации (ЗИ). К данному устройству предъявляются следующие требования:

- Способность к интеграции с любыми цифровыми интерфейсами;
- Возможность изменения структуры устройства на аппаратном уровне, адаптирования устройства к различным факторам;
- Желательна возможность более быстрого изменения алгоритма работы устройства посредством его программирования.

Учитывая данные требования, возможным решением может стать разработка АС на основе программируемых логических интегральных схем (ПЛИС) посредством описания его на языке описания аппаратуры (HDL). Дабы была возможность программирования устройства, а так же для увеличения гибкости его применения, было решено выполнить данное устройство в виде процессорного ядра MIPS-подобной архитектуры с расширенным трактом данных для оптимизации криптографических вычислений.

Объектом исследования выпускной квалификационной работы (ВКР) является создание аппаратного средства шифрования данных, удовлетворяющего вышеописанным требованиям.

Предметом исследования ВКР является процесс создания микропроцессорного ядра и оптимизация его для реализации алгоритма шифрования на аппаратном уровне.

Целью ВКР является создание процессорного ядра с MIPS-подобной архитектурой и расширенным трактом данных, оптимизированным для реализации выбранного криптографического алгоритма.

Постановка задачи

Для достижения поставленной цели необходимо решить следующие задачи:

- Изучить ПЛИС и HDL;
- Разработать архитектуру процессорного ядра;
- Выбрать алгоритм шифрования для реализации;
- Разработать микроархитектуру процессорного ядра;
- Произвести оптимизацию процессорного ядра;
- Реализовать шифрование тестового канала передачи данных для демонстрации возможностей устройства.

1 ИНСТРУМЕНТЫ РЕАЛИЗАЦИИ

1.1 ПЛИС

На сегодняшний день при проектировании цифровых устройств никто не использует дискретные элементы. Данный подход трудозатратен, дорогостоящ и неэффективен. Вместо этого применяют матрицы логических элементов.

ПЛИС представляет собой матрицу конфигурируемых логических элементов (ЛЭ), которые также называются конфигурируемыми логическими блоками. Каждый ЛЭ можно сконфигурировать для выполнения функций некоторой комбинационной или последовательной схемы. На Рисунке 1 приведена обобщённая структура ПЛИС. ЛЭ окружены элементами ввода/вывода, которые предназначены для организации обмена информацией между FPGA и прочими компонентами системы. Элементы ввода/вывода соединяют входы и выходы логических элементов с контактами корпуса микросхемы. Логические элементы могут быть соединены между собой и с эле-

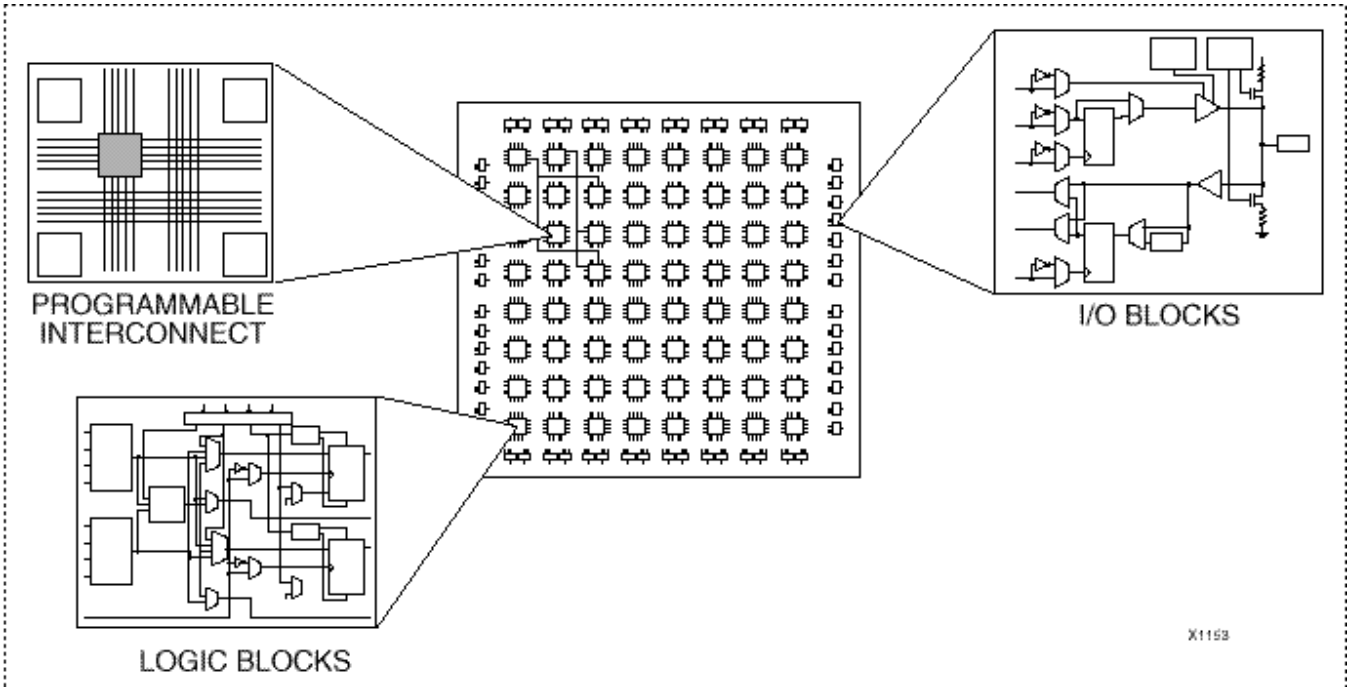

Рисунок 1 — Пример структуры ПЛИС

ментами ввода/вывода с помощью программируемых каналов трассировки[3].

На сегодняшний день рынок ПЛИС в основном представлен двумя компаниями-производителями: Xilinx и Altera. Обе компании зарекомендовали себя отличными производителями, в результате чего рынок поделён примерно пополам.

1.2 HDL

В 1990-е годы разработчики обнаружили, что их производительность труда резко возрастала, если они работали на более высоком уровне абстракции, определяя только логическую функцию и предоставляя создание оптимизированных логических элементов системе автоматического проектирования и разработки (САПР). Для этой цели стали использовать языки описания аппаратуры или HDL. Два основных языка описания аппаратуры — Verilog и VHDL. Verilog и VHDL построены на похожих принципах, но их синтаксис весьма различается. В сравнении с Verilog VHDL является более громоздким и многословным. Вдобавок, на сегодняшний день, наблюдается тенденция к преобладанию Verilog.

На обоих языках можно полностью описать любую электронную систему. Две основные цели HDL — логическая симуляция и синтез[2].

Симуляция предназанчена для снижения затрат на тестирование системы. Тестирование системы в лаборатории весьма трудоёмко. Исследовать причины ошибок в лаборатории может быть очень сложно, так как наблюдать можно только сигналы, подключенные к контактам чипа, а то, что происходит внутри чипа, напрямую наблюдать невозможно. Исправление ошибок уже после того, как система была выпущена, может быть очень дорого. Например, исправление одной ошибки в новейших интегральных микросхемах стоит больше миллиона долларов и занимать несколько месяцев.

Во время симуляции на входы модуля подаются некоторые воздействия и проверяются выходы, чтобы убедиться, что модуль функционирует корректно, так же, можно просмотреть состояние всех внутренних элементов. С другой стороны, нужно учитывать, что данные значения являются дишь прогнозируемыми и могут несколько различаться с реальными. Для приближения к реальности, производителями HDL и САПР придумана возможность задания дополнительных параметров симуляции. При правильном задании всех параметров и правильном проектировании устройства, результаты симуляции практически всегда совпадают с реальными.

Логический синтез преобразует код на HDL в нетлист, описывающий цифровую аппаратуру (т.е. логические элементы и соединяющие их проводники). Логический синтезатор может выполнять оптимизацию для сокращения количества необходимых элементов. Нетлист может быть текстовым файлом или нарисован в виде схемы, чтобы было легче визуализировать систему.

В конце концов, нетлист можно преобразовать в бинарную "прошивку" и загрузить её на ПЛИС.

1.3 Выбор метода реализации задач

Основной задачей данной работы является создание микропроцессорного ядра посредством описания его на HDL, оптимизация его для криптографического алгоритма и демонстрация его возможностей посредством шифрования тестового канала передачи данных. Инструментами для решения основной задачи на основании вышеизложенных фактов были выбраны язык описания аппаратуры Verilog и тестовая ПЛИС Xilinx Spartan-3AN FPGA Starter Kit Board. Для демонстрации было решено реализовать шифрование данных, передаваемых по малопопулярному на сегодняшний день интерфейсу RS-232.

2 АРХИТЕКТУРА ПРОЦЕССОРНОГО ЯДРА

В качестве основы для разрабатываемой архитектуры используется архитектура MIPS, а потому, используется понятие "MIPS-подобная архитектура". Архитектура MIPS является простой reduced instruction set computer (RISC) архитектурой.

От архитектуры MIPS были унаследованы следующие основные особенности:

- Форматы инструкций;
- Режимы адресации;

Помимо того, были унаследованы основные команды и дополнены командами, оптимизированными для реализации алгоритма ГОСТ Р 34.12 с размером блока 64 бита. В следующих частях этой главы рассматриваются вышеописанные особенности разрабатываемой архитектуры.

2.1 Типы инструкций

В архитектуре MIPS в качестве компромисса между простотой и универсальностью используются три формата инструкций: типа R, типа I и типа J. Небольшое количество форматов обеспечивает определенное единообразие между всеми тремя типами и, как следствие, более простую аппаратную реализацию. При этом разные форматы позволяют учитывать различные потребности инструкций, как, например, необходимость хранить большие константы внутри инструкций.

2.1.1 Инструкции типа R

Название типа R является сокращением от "Register-type" — регистрового типа. Инструкции типа R используют три регистра в качестве операндов: два регистра-источника и один регистр-назначение. На Рисунке 2 показан машинный формат команды типа R. 32-битная команда состоит из шести полей: `op`, `rs`, `rt`, `rd`, `shamt` и `funct`. Каждое поле состоит из пяти или шести бит[2].

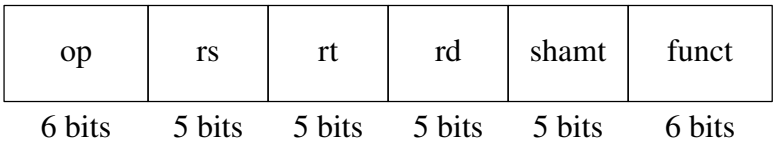

Рисунок 2 — Инструкция типа R

Операция, выполняемая командой, закодирована двумя полями: полем `op` (также называемым `opcode` или кодом операции) и полем `func` (также называемым функцией). У всех команд типа R поле `opcode` равно нулю. Операция, выполняемая этими командами, определяется исключительно полем `funct`. Например, поля `opcode` и `funct` у инструкции `add` равны 0 и 32

соответственно. Аналогично, у команды `sub` поля `opcode` и `funct` равны 0 и 34.

Операнды закодированы тремя полями: `rs, rt` и `rd` . Поля содержат номера регистров. Пятое поле, `shamt`, является сокращением от shift amount используется только для операций сдвига. В таких командах двоичное значение, хранимое в 5-битном поле `shamt`, задаёт величину сдвига. У всех остальных команд типа R поле `shamt` равно 0.

2.1.2 Инструкции типа I

Название типа I является сокращением от immediate-type или непосредственного типа. Инструкции типа I используют в качестве операндов два регистра и один непосредственный операнд (константу).

На Рисунке 3 показан формат машинной команды типа I. 32-битная команда состоит из четырёх полей: `op, rs, rt` и `imm`. Первые три поля `(op, rs` и `rt)` аналогичны таким же полям в командах типа R. Поле `imm` (сокр. от immediate) содержит 16-битную константу. 16-битные константы перед использованием в операциях будут расширены до 32 бит посредством знако-

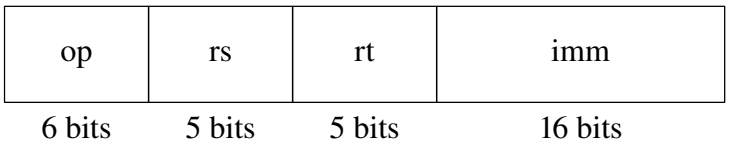

Рисунок 3 — Инструкция типа I

вого расширения либо дополнения нулями[2].

Операция определяется исключительно полем `opcode`. Операнды заданы в трёх полях: `rs, rt` и `imm`. Поля `rs` и imm всегда используются как операнды-источники. Поле `rt` в некоторых командах (например, `addi` и `lw`) содержит номер регистра-назначения, в других (например, `sw`) — номер регистра-источника.

2.1.3 Инструкции типа J

Название типа J является сокращением от английского слова jump — прыжок. Этот формат используется только для инструкций безусловного перехода и ветвления. Как и другие команды, команды типа J начинаются с 6-битного поля кода операции `opcode`. Также, этот формат инструкций содержит 26-битный операнд `addr`. Константное значение `addr` используется для указания адреса перехода[2].

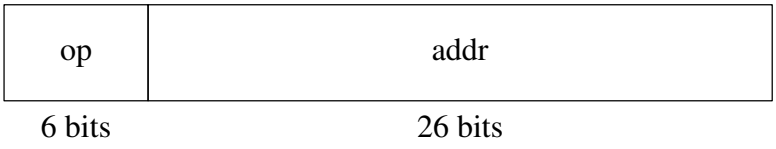

Рисунок 4 — Инструкция типа J

2.2 Режимы адресации

В архитектуре MIPS используются пять режимов адресации: регистровый, непосредственный, базовый, относительно счётчика команд и псевдопрямой. Первые три режима (регистровый, непосредственный и базовый) определяют способы чтения и записи операндов. Последние два (режим адресации относительно счетчика команд и псевдопрямой режим) определяют способы записи счётчика команд[2].

2.2.1 Регистровая адресация

При регистровой адресации регистры используются для всех операндов-источников и операндов-назначений (иными словами — для всех операндов и результата). Все инструкции типа R используют именно такой режим адресации[2].

2.2.2 Непосредственная адресация

При непосредственной адресации в качестве операндов наряду с регистрами используют 16-битные константы (непосредственные операнды). Этот режим адресации используют некоторые инструкции типа I, такие как сложение с константой `(addi)` и загрузка константы в старшие 16 бит регистра `(lui)`[2].

2.2.3 Базовая адресация

Инструкции для доступа в память, такие как загрузка слова `(lw)` и сохранение слова `(sw)`, используют базовую адресацию. Эффективный адрес операнда в памяти вычисляется путём сложения базового адреса в регистре `rs` и 16-битного смещения с расширенным знаком, являющегося непосредственным операндом[2].

2.2.4 Адресация относительно счётчика команд

Инструкции условного перехода, или ветвления, используют адресацию относительно счётчика команд для определения нового значения счётчика команд в том случае, если нужно осуществить переход. Смещение со знаком прибавляется к счётчику команд (PC) для определения нового значения PC, поэтому тот адрес, куда будет осуществлен переход, называют адресом

относительно счётчика команд[2].

2.2.5 Псевдопрямая адресация

При прямой адресации адрес перехода задаётся внутри инструкции. Инструкции безусловного перехода `j` и `jal` в идеале могли бы использовать прямую адресацию для определения 32-битного целевого адреса перехода, указывающего адрес инструкции, которая будет выполнена следующей. К сожалению, в формате инструкций типа J нет достаточного количества бит для того, чтобы задать полный 32-битный адрес перехода. Шесть старших бит инструкции занимает код операции (поле `opcode`), поэтому для адреса перехода остаётся только 26 бит. К счастью, два младших бита адреса перехода всегда должны быть равны нулю, потому что все инструкции выровнены по словам. Следующие 26 бит адреса перехода берутся из поля `addr` инструкции. Четыре старших бита адреса перехода берутся из четырёх старших бит значения `PC + 4`. Такой способ адресации называется псевдопрямым[2].

2.3 Инструкции

При составлении итогового набора инструкций, основными целями являлись минималистичность, простота и максимальная применяемость. Всего в конечном наборе инструкций присутствует 26 инструкций.

При описании инструкций используются следующие обозначения:

- `[reg]` — содержимое регистра;
- `imm` — непосредственный операнд в инструкциях типа I;
- `addr` — адрес в инструкциях типа J;
- `signext`— непосредственный операнд после знакового расширения;
- `zeroext`— непосредственный операнд после расширения нулями;
- `BTA`— целевой адрес ветвления;
- `JTA`— целевой адрес перехода;
- `PC`— значение счётчика команда;
- `label`— текстовая метка для адреса инструкции;
- `addr`— адрес в режиме адресации относительно счётчика команд;
- `[addr]`— содержимое памяти по адресу `addr`.
- `KEY[n]`— значение n-го 32-битного поля ключа `KEY` .
- `SUB[$n]`— значение таблицы SUB для регистра `$n`.

2.3.1 Арифметические

В итоговой архитектуре представлены следующие арифметические инструкции:

- `add $rd, $rs, $rt` — инструкция сложения со знаковым расширением: `[$rd] = [$rs] + [$rt];`
- `sub $rd, $rs, $rt` — инструкция вычитания со знаковым расширением: `[$rd] = [$rs] - [$rt];`
- `addu $rd, $rs, $rt` — инструкция сложения без расширения знака: `[$rd] = [$rs] + [$rt];`
- `subu $rd, $rs, $rt` — инструкция вычитания без расширения знака: `[$rd] = [$rs] - [$rt];`
- `addi $rd, $rs, imm` — инструкция сложения с непосредственным операндом со знаковым расширением: `[$rd] = [$rs] + imm;`
- `subi $rd, $rs, imm` — инструкция вычитания непосредственного операнда со знаковым расширением: `[$rd] = [$rs] - imm;`

2.3.2 Логические

В итоговой архитектуре представлены следующие логические инструкции:

- `and $rd, $rs, $rt` — инструкция логического И: `[$rd] = [$rs] and [$rt];`
- `andi $rd, $rs, imm` — инструкция логического И с непосредственным операндом: `[$rd] = [$rs] and imm;`
- `nand $rd, $rs, $rt` — инструкция логического НЕ И: `[$rd] = not ([$rs] and [$rt]);`
- `or $rd, $rs, $rt` — инструкция логического ИЛИ: `[$rd] = [$rs] or [$rt];`
- `ori $rd, $rs, imm` — инструкция логического ИЛИ с непосредственным операндом: `[$rd] = [$rs] or imm;`
- `nor $rd, $rs, $rt` — инструкция логического НЕ ИЛИ: `[$rd] = not ([$rs] or [$rt]);`
- `xor $rd, $rs, $rt` — инструкция логического ИСКЛЮЧАЮЩЕГО ИЛИ: `[$rd] = [$rs] xor [$rt];`
- `slt $rd, $rs, $rt` — инструкция "установить, если меньше": `[$rd] = ([$rs] < [$rt])? 1 : 0;`
- `slti $rd, $rs, imm` — инструкция "установить, если меньше непосредственного операнда": `[$rd] = ([$rs] < imm)? 1 : 0;`

- `sll $rd, $rs, shamt` — инструкция логического сдвига влево: `[$rd] = [$rs] << shamt;`
- `srl $rd, $rs, shamt` — инструкция логического сдвига вправо: `[$rd] = [$rs] >> shamt;`

2.3.3 Переходы и ветвления

В итоговой архитектуре представлены следующие инструкции перехода и ветвления:

- `beq $rs, $rt, label` — ветвление, если равно: `([$rd] = [$rs])? PC = BTA;`
- `bne $rs, $rt, label` — ветвление, если не равно: `([$rd] != [$rs])? PC = BTA;`
- `j label` — безусловный переход: `PC = JTA;`
- `jal label` — безусловный переход с возвратом: `PC = JTA, $ra = PC + 4;`
- `jr $rs` — безусловный переход по регистру: `PC = [$rs];`

2.3.4 Передача данных

В итоговой архитектуре представлены следующие инструкции передачи данных:

- `lw $rt, imm($rs)` — загрузка слова из памяти: `$rt = [addr];`
- `sw $rt, imm($rs)` — загрузка слова в память: `[addr] = [$rt];`
- `lui $rt, imm` — загрузка непосредственного операнда в верхние 16 бит регисра и дополнение 0 нижних: `$rt = {imm, 16'b0};`

2.3.5 Дополнительные инструкции для оптимизации криптоалгоритма

В итоговую архитектуру были добавлены следующие инструкции для оптимизации криптографического алгоритма.

- `slc $rd, $rs, shamt` — инструкция циклического сдвига влево: `[$rd] = {[$rs][(sizeof($rs) - shamt - 1) : 0], [$rs][sizeof($rs) - 1 : sizeof($rs) - shamt};`
- `src $rd, $rs, shamt` — инструкция циклического сдвига вправо: `[$rd] = {[$rs][(shamt - 1) : 0], [$rs][sizeof($rs) - 1 : shamt};`
- `kxor $rd, $rs, shamt` — инструкция ИСКЛЮЧАЮЩЕГО ИЛИ с ключом: `$rd = $rs xor KEY[shamt];`
- `cs $rt, $rs` — инструкция подстановки из таблицы подстановок: `$rs = SUB[$rs].`

3 АЛГОРИТМ ШИФРОВАНИЯ

В качестве алгоритма шифрования для реализации выбран алгоритм ГОСТ 34.12-2015 с длиной блока 64 бита или «Магма». Данный алгоритм представляет из себя простую сеть Фейстеля, состоящую из 32 раундов. Так как для демонстрации было принято реализовать потоковый шифратор, было решено реализовать данный алгоритм в режиме режиме гаммирования с обратной связью по выходу. Подробности работы данного алгоритма и режима рассмотрены ниже.

3.1 ГОСТ 34.12-2015. Алгоритм «Магма»

3.1.1 Преобразования

В начале блок входного текста разбивается на две половины. Основная часть всех преобразований выполняется над половиной, содержащей младшие 32 бита.

Первым преобразованием является сложение с ключом по модулю 2^{32}.

Затем, полученный 32-битный блок разбивается на блоки по 4 бита, каждый из которых используется в качестве индекса в одной из 8 таблиц подстановок. Выходные значения конкатенируются и получается новый 32-битный блок. В рамках ГОСТ 34.12-2015 данное преобразование

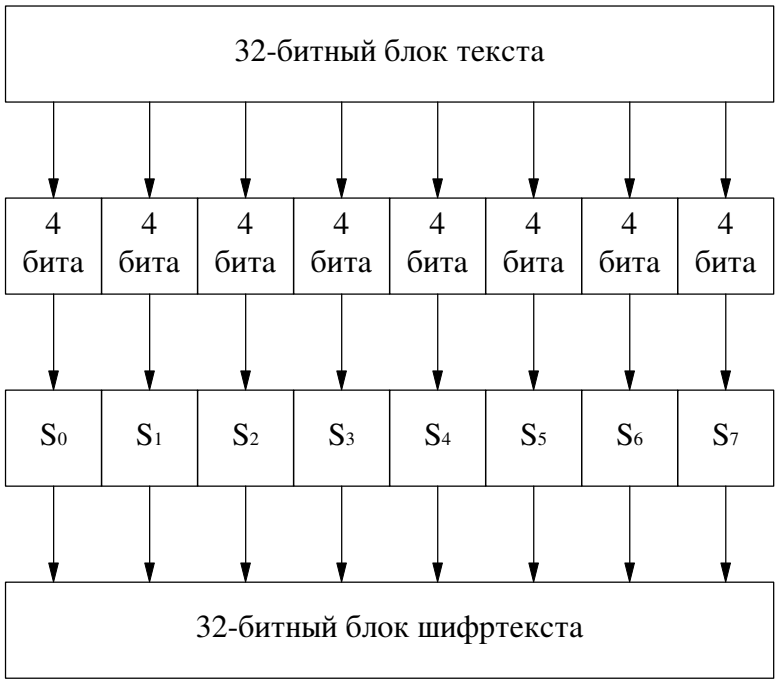

Рисунок 5 — Преобразование *t*

названо *t*-преобразование. Третьим преобразованием является циклический сдвиг 32-битного блока в сторону старших разрядов на 11 бит[10].

Вышеописанные преобразования в документе ГОСТ-а объединяюстя в преобразование g,

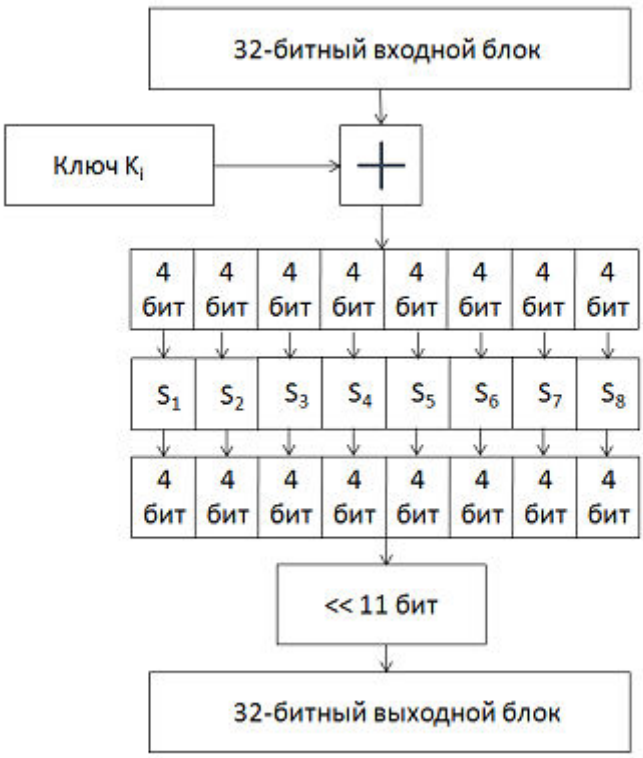

Рисунок 6 — Преобразование g

которая в итоге приобретает вид, изображённый на Рисунке 6.

После g-преобразования младшие 32 бита складываются по модулю 2 со старшими. Получившийся 32-битный блок поступает на место младших 32 бит итогового блока, а младшие 32 бита исходного — на место старших 32 бит итогового блока.

Вышеописанные преобразования в документе ГОСТ-а объединяются в преобразование G, равное одному раунду. Такой раунд повторяется 31 раз. В последнем этапе выход f-преобразования поступает на место старших 32-бит итогового блока, а младшие 32 бита выхода предпоследнего этапа — на место младших 32 итогового.

Полностью схема преобразований показана на Рисунке 7.

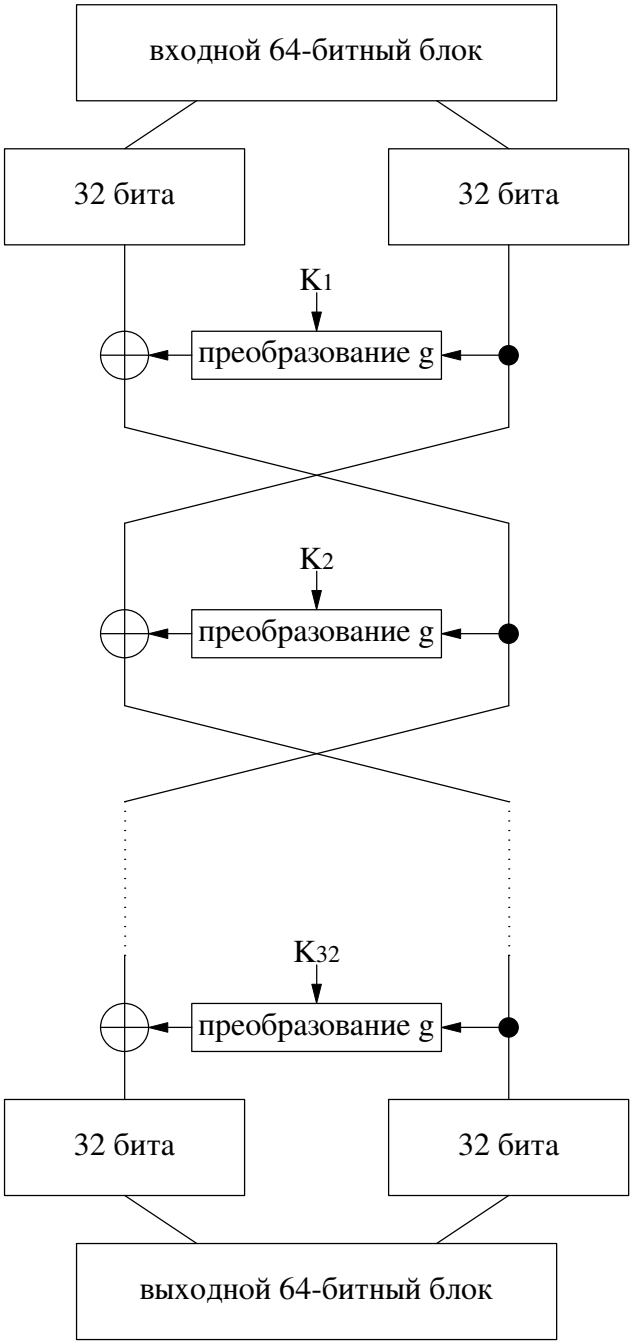

Рисунок 7 — Полная схема преобразований

3.1.2 Выработка раундовых ключей

Для генерации подключей исходный 256-битный ключ разбивается на восемь 32-битных блоков: $K_1 - K_8$:

$$K_1 = k_{255} \| - \| k_{224};$$

$$K_2 = k_{223}\| - \|k_{192};$$

$$K_3 = k_{191}\| - \|k_{160};$$

$$K_4 = k_{159}\| - \|k_{128};$$

$$K_5 = k_{127}\| - \|k_{96};$$

$$K_6 = k_{95}\| - \|k_{64};$$

$$K_7 = k_{63}\| - \|k_{32};$$

$$K_8 = k_{31}\| - \|k_0;$$

Ключи $K_9 - K_{24}$ являются циклическим повторением ключей $K_1 - K_8$. Ключи $K_{25} - K_{32}$ являются ключами $K_8 - K_1$.

3.1.3 Шифрование и расшифрование

В итоге, шифрование одного 64-битного блока данных имеет следующий вид:

$$E_{K_1-K_{32}}(a) = G^*[K_{32}]G[K_{31}] - G[K_1](a_{1,}a_0),$$

где $a = a_1\|a_0$

Расшифрование отличается только порядком применения ключей:

$$D_{K_1-K_{32}}(a) = G^*[K_1]G[K_2] - G[K_{32}](a_{1,}a_0),$$

где $a = a_1\|a_0$

3.2 Режим гаммирования с обратной связью по выходу

3.2.1 Общие положения

Режим имеет следующие основные параметры:

s — размер вырабатываемой гаммы шифра;

n — размер блока базового алгоритма шифрования;

$m = z \cdot n$ — размер регистра сдвига;

IV — синхропосылка.

Зашифрование в режиме гаммирования с обратной связью по выходу заключается в по компонентном сложении открытого текста с гаммой шифра, которая вырабатывается блоками длины s. При вычислении очередного блока гаммы выполняется зашифрование n разрядов

регистра сдвига с большими номерами базовым алгоритмом блочного шифрования. Затем заполнение регистра сдвигается на n бит в сторону разрядов с большими номерами, при этом в разряды с меньшими номерами записывается полученный выход базового алгоритма блочного-шифрования. Блок гаммы вычисляется путём усечения выхода базового алгоритма блочного шифрования.

При использовании режима гаммирования с обратной связью по выходу не накладывается ограничений на длину входного блока открытого текста, а потому на его основе возможно построить потоковый шифратор.

3.2.2 Зашифрование

Открытый текст P представляется в виде $P = P_1 \| P_2 \| \ldots \| P_q$, Блоки шифртекста вычисляются по следующему правилу:

$$R_1 = IV,$$

$$\begin{cases} (Y_i = e_k(MSB_n(R_i)) \\ C_i = P_i \oplus T_s(Y_i), \qquad i = 1, 2, \ldots, q-1, \\ R_{i+1} = LSB_{m-n}(R_i) \| Y_i, \end{cases}$$

$$Y_q = e_k(MSB_n(R_q))$$

$$C_q = P_q \oplus T_r(Y_q),$$

Результирующий шифртекст имеет вид:

$$C = C_1 \| C_2 \| \ldots \| C_q.$$

Зашифрование в режиме гаммирования с обратной связью по выходу проиллюстрировано на Рисунке 8.

3.2.3 Расшифрование

Расшифрование абсолютно идентично шифрованию, за исключением того, что с вырабатываемой гаммой складывается не открытый текст, а шифртекст. Данная особенность делает шифратор и дешифратор абсолютно идентичными.

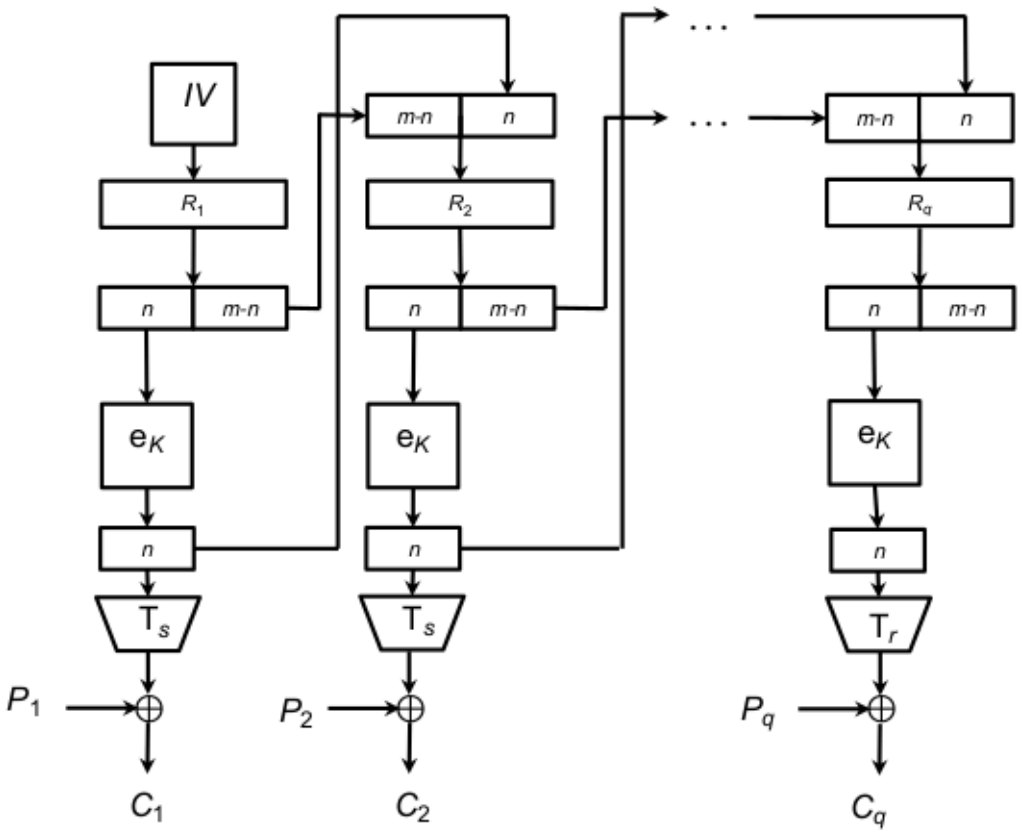

Рисунок 8 — Схема зашифрования в режиме гаммирования с обратной связью по выходу

4 РЕАЛИЗАЦИЯ

В данной главе описываются разработанное ядро, способ его оптимизации для реализации выбранного алгоритма и оснащение его интерфейсами для шифрования выбранного тракта передачи данных в качестве презентации его возможностей.

4.1 Создание процессорного ядра

Компьютерная архитектура определяется набором команд и архитектурным состоянием. Архитектурное состояние процессора MIPS определяется содержимым счётчика команд (program counter) и 32 видимых программисту регистра, поэтому любой процессор, реализующий MIPS-подобную архитектуру, вне зависимости от его микроархитектуры обязан иметь счётчик команд и ровно 32 регистра. Зная текущее архитектурное состояние, процессор точно знает, какую операцию и над какими данными надо выполнить для получения нового архитектурного состояния.

Рассмотрим структуру, показанную на Рисунке 9 и модуль верхнего уровня для разработанного процессорного ядра.

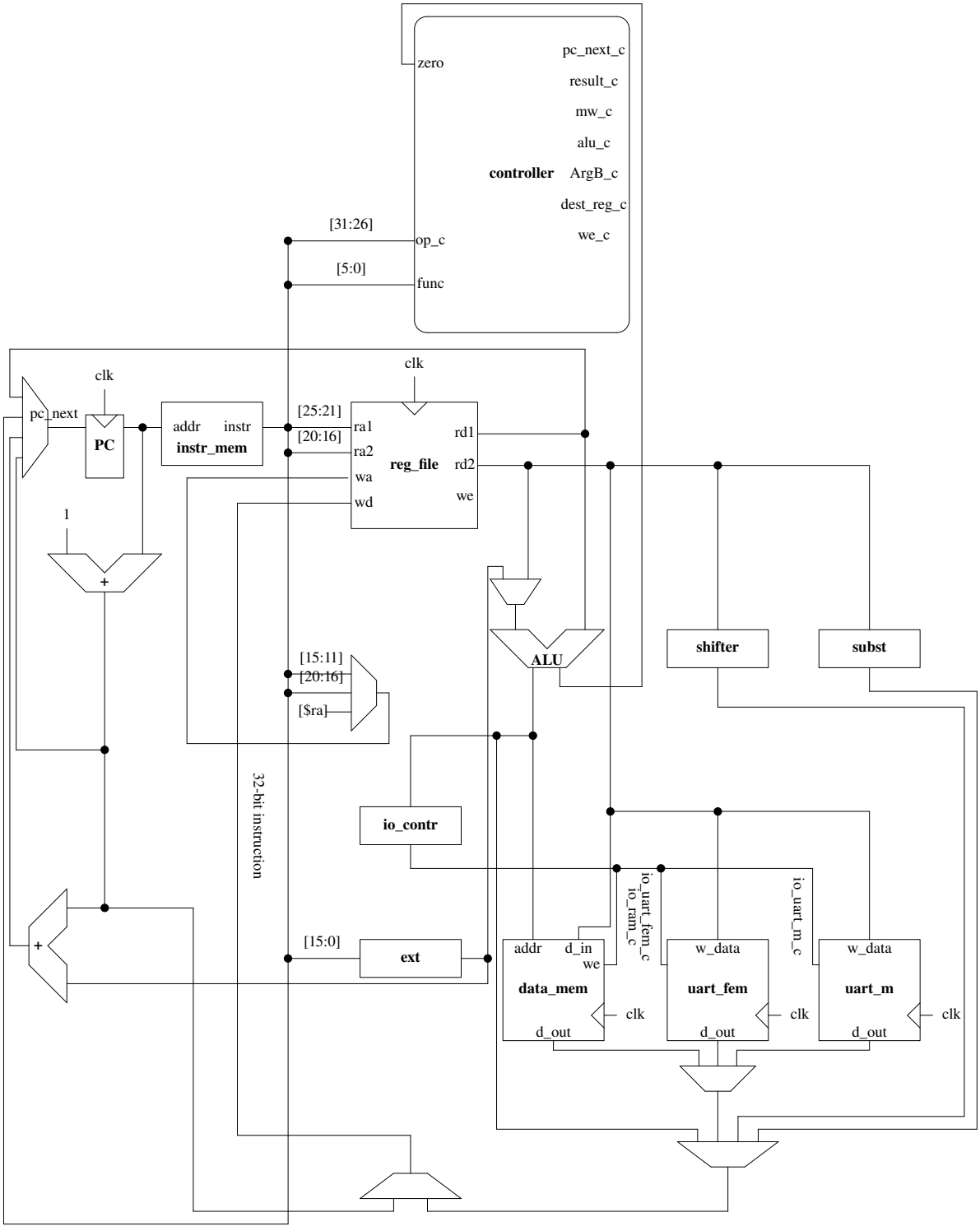

Рисунок 9 — Общая схема ядра

В файле cpu.v описана общая структура ядра.

```
module cpu
(
        input wire     clk_in,
        input wire     reset,

        input wire     rx_m,
        output wire    tx_m,
        input wire     rx_fem,
        output wire    tx_fem,

        output wire    led0,
        output wire    led1,
        output wire    led2,
        output wire    led3,
        output wire    led4,
        output wire    led5,
        output wire    led6,
        output wire    led7
        );

        wire [31:0]    instr, pc_val, mem_addr,
        mem_write, mem_read, ram_out;
        wire [5:0]     op_c, funct;
        wire           zero, argB_c, we_c, wd_c, io_ram_c,
        io_uart_fem_c, io_uart_m_c, clk;
        wire [1:0]     dest_reg_c, ext_c, io_read_c;
        wire [2:0]     pc_next_c, result_c;
        wire [3:0]     us, alu_c;
        wire [7:0]     leds, uart_fem_r, uart_m_r;
        wire [31:0]    bus, tmr_ctrl;
        wire           counter, tmr_overflow;
        wire [15:0]    tmr_cntr;

        reg            counter_reg = 1'b1;

        always @ (posedge clk_in)
                counter_reg <= counter;

        assign counter = counter_reg + 1'b1;
        assign clk = counter;
```

```
contr          contr_u ( .op_c(op_c), .funct(funct),
.zero(zero), .argB_c(argB_c), .dest_reg_c(dest_reg_c),
.result_c(result_c),.we_c(we_c), .ext_c(ext_c), .wd_c(wd_c),
.sh_d_c(sh_d_c), .pc_next_c(pc_next_c), .alu_c(alu_c));

datapath       datapath_u ( .clk(clk), .reset(reset),
.instr(instr), .pc_val(pc_val), .mem_addr(mem_addr),
.mem_write(mem_write), .mem_read(mem_read), .op_c(op_c),
.funct(funct), .zero(zero), .argB_c(argB_c),
.dest_reg_c(dest_reg_c), .result_c(result_c), .we_c(we_c),
.ext_c(ext_c),.wd_c(wd_c),  .sh_d_c(sh_d_c),
.pc_next_c(pc_next_c), .alu_c(alu_c), .us(us),
.tmr_ctrl(tmr_ctrl), .tmr_cntr(tmr_cntr),
.tmr_overflow(tmr_overflow), .leds(leds), .bus(bus));

instr_mem      instr_mem (.addr(pc_val[10:0]), .data(instr));

timer          tmr (.clk(clk), .reset(~tmr_ctrl[31]),
.scale(tmr_ctrl[30:16]), .period(tmr_ctrl[15:0]),
.cntr(tmr_cntr), .overflow(tmr_overflow));

io_contr       io_c (.addr(mem_addr), .op_c(op_c),
.io_ram_c(io_ram_c), .io_uart_fem_c(io_uart_fem_c),
.io_uart_m_c(io_uart_m_c), .io_read_c(io_read_c));

ram            data_mem (.clk(clk), .we(io_ram_c),
.addr(mem_addr[11:0]), .d_in(mem_write), .d_out(ram_out));

uart           uart_fem (.clk(clk), .reset(reset),
.rd_uart(io_read_c[0]), .wr_uart(io_uart_fem_c), .rx(rx_fem),
.w_data(mem_write[7:0]), .tx_full(us[1]), .rx_empty(us[0]),
.tx(tx_fem), .r_data(uart_fem_r));

uart           uart_m (.clk(clk), .reset(reset),
.rd_uart(io_read_c[1]), .wr_uart(io_uart_m_c), .rx(rx_m),
.w_data(mem_write[7:0]), .tx_full(us[3]), .rx_empty(us[2]),
.tx(tx_m), .r_data(uart_m_r));

mux3to1        mux_io_read (.in0(ram_out),
.in1({24'b0, uart_fem_r}), .in2({24'b0, uart_m_r}),
.ctrl(io_read_c), .out(mem_read));
```

```
        assign {led0, led1, led2, led3, led4, led5, led6, led7} = leds;

endmodule
```

Файл CPU.v

В структуре итогового процессорного ядра можно выделить 3 основных модуля:

1. Регистровый файл;
2. Арифметико-логическое устройство;
3. Контроллер.

В рамках данной части будут рассмотрены эти три модуля. Файлы описания остальной части ядра приведены в приложении.

Файл reg_file.v содержит описание регистрового файла, хранящего все 32 видимых регистра.

```
`include "./include/registers.v"

module reg_file(
        input wire              clk,
        input wire              we,//for write enable

        input wire [4:0]        ra1,//
        input wire [4:0]        ra2,//for read  addresses
        input wire [4:0]        wa,//for write address


        output wire [31:0]      rd1,// for read data
        output wire [31:0]      rd2,//
        input wire [31:0]       wd,//for write data

        input wire [3:0]        us,//for uart state
        input wire [15:0]       tmr_cntr,//for timer counter
        output wire [31:0]      tmr_ctrl,//for timer controll

        output wire [7:0]       leds
);

        reg [31:0] rf [31:0];
```

```
        initial
                begin
                $readmemh("reg_init.data", rf, 0, 31);
                end

        always @(posedge clk)
                if (we)
                        rf[wa] <= wd;

        assign rd1 = (ra1 == `US)? {28'b0, us} :
        ((ra1 == `TMR)? {16'b0, tmr_cntr} : rf[ra1]);
        assign rd2 = (ra2 == `US)? {28'b0, us} :
        ((ra1 == `TMR)? {16'b0, tmr_cntr} : rf[ra2]);

        assign tmr_ctrl = rf[`TCON];

        assign leds = rf [`V0][7:0];

endmodule
```

Файл reg_file.v

Регистровый файл выполнен в виде блока RAM. На вход поступают 3 адреса — два адреса регистров-источников операндов (`rd1, rd2`), значения которых подаются на выходы `rd1, rd2` и один адрес регистра-назначения результата (`wa`), которому присваивается значение, поданое на вход `wd`, при условии, что был послан управляющий сигнал `we`. Помимо того, в регистровый файл отображаются состояния периферийных устройств. Данная особенность будет рассмотрена подробнее позже.

Файл alu.v содержит описание арифметико-логического устройства (АЛУ).

```
module alu(
        input wire [3:0]      mode,
        input wire [31:0]     A,
        input wire [31:0]     B,
        output reg [31:0]     C,
        output reg            zero
    );

localparam //functions
         //logical
```

```
and_f = 4'b0000,
or_f = 4'b0001,
xor_f = 4'b0010,
nor_f = 4'b0011,
slt_f = 4'b0100,
nand_f = 4'b0101,

//arithmetic
add_f = 4'b1000,
subtr_f = 4'b1001;

always @ *
begin
zero = 0;
      case(mode)
      and_f:
            C = A & B;

      nand_f:
            C = ~(A & B);

      or_f:
            C = A | B;

      xor_f:
            C = A ^ B;

      nor_f:
            C = ~(A | B);

      slt_f:
            C = (A < B) ? 32'h00000001 : 32'h00000000;

      subtr_f:
            C = A - B;

      add_f:
            C = A + B;

      default:
            C = A;
```

```
            endcase

            if (C == 0)
                        zero = 1'b1;
        end

endmodule
```

Файл alu.v

На входы A и B подаются значения операндов, на вход mode — код выполняемой инструкции, результат подаётся на выход C. Стоит так же отметить наличие флага zero, выставляемого при равенстве результата нулю.

Файл contr.v содержит описание контроллера.

```
`include "./include/funct_codes.v"
module contr(
        input wire          zero,
        input wire [5:0]    op_c,
        input wire [5:0]    funct,

        output wire         argB_c,
        output wire [1:0]   dest_reg_c,
        output wire         we_c,
        output wire [1:0]   ext_c,
        output wire         wd_c,
        output wire         sh_d_c,
        output wire [2:0]   pc_next_c,
        output wire [2:0]   result_c,
        output wire [3:0]   alu_c
        );

        wire [2:0]    aluop;
        wire          jr_c;

        maindec       main_dec(.funct(funct), .op_c(op_c),
        .argB_c(argB_c), .dest_reg_c(dest_reg_c),
        .we_c(we_c), .result_c(result_c), .beq(beq),
        .bne(bne), .j_c(j_c), .jr_c(jr_c), .ext_c(ext_c),
        .wd_c(wd_c), .sh_d_c(sh_d_c), .aluop(aluop));
```

```
        aludec          alu_dec(.funct(funct),
        .aluop(aluop), .alu_c(alu_c) );

        assign branch_c =(~zero & bne) | (zero & beq);
        assign pc_next_c = {jr_c, j_c, branch_c};

endmodule
```

Файл contr.v

Контроллер разделён на две части: основной декодер и декодер АЛУ. Основной декодер преобразует кода операции и функции в управляющие сигналы. Декодер АЛУ преобразует код функции и сигнал `aluop`, поступающий с основного декодера в управляющие сигналы АЛУ.

4.2 Оптимизация ядра для реализации криптографического алгоритма

Основной упор в процессе оптимизации был сделан на максимальное снижение взаимодействия с памятью на время выполнения алгоритма шифрования. Причиной этому является то, что хоть архитектура процессора и является гарвардской, а сам процессор однотактным, взаимодействие с памятью всё равно является трудозатратным и ресурсоёмким процессом.

4.2.1 Выработка раундовых ключей

Во-первых, в тракт данных был добавлен модуль, реализующий генерацию раундовых ключей. Описание данного модуля содержит файл `rk_gen.v` (raund key generator).

```
module rk_gen
(
        input wire [4:0]      raund,
        output wire [31:0]    raundkey
);
        reg [31:0] key [2:0];
        wire [2:0] keynum;

        initial
        begin
                key[0] = 32'hffeeddcc;
                key[1] = 32'hbbaa9988;
                key[2] = 32'h77665544;
                key[3] = 32'h33221100;
                key[4] = 32'hf0f1f2f3;
```

```
            key[5] = 32'hf4f5f6f7;
            key[6] = 32'hf8f9fafb;
            key[7] = 32'hfcfdfeff;
        end

        assign keynum = (raund[4:3] == 2'b11)? raund[2:0] : ~raund[2:0];

        assign raundkey = key[keynum];

endmodule
```

Файл rk_gen.v

Номер раунда поступает на вход `raund`. 256-битный ключ реализован в виде блока ROM. Для вычисления номера подключа из номера раунда, использовалось одно замеченное свойство: если начать нумерацию раундов не с 1, а с 0, то номером ключей для первых 24 (0-23) раундов является значение последних 3 бит номера раунда. Для последних же 8 раундов (24-31) для получения номера раундового ключа, достаточно произвести инвертацию последних 3 бит. Данная операция описана в следующей строке:

```
assign keynum = (raund[4:3] == 2'b11)? raund[2:0] : ~raund[2:0];
```

Выход данного модуля через мультиплексор подключён ко входу A АЛУ. На вход B подаётся значение 32-битного блока. АЛУ производит операцию сложения данных значений по модулю 2. Таким образом, полностью реализовано сложение с раундовым ключом за 1 рабочий такт процессора.

4.2.2 Реализация таблицы подстановок

Во-вторых, был добавлен модуль, реализующий таблицу подстановок на аппаратном уровне. Описание данного модуля содержит файл `subst.v` (look up table).

```
module subst
(
        input wire [31:0]     in,
        output wire [31:0]    out
);
        reg  [3:0] sub0 [15:0];
        reg  [3:0] sub1 [15:0];
        reg  [3:0] sub2 [15:0];
```

```
reg  [3:0] sub3 [15:0];
reg  [3:0] sub4 [15:0];
reg  [3:0] sub5 [15:0];
reg  [3:0] sub6 [15:0];
reg  [3:0] sub7 [15:0];

wire [3:0] out0, out1, out2, out3, out4, out5, out6, out7;

initial
begin
sub0[0] = 4'hc; sub0[1] = 4'h4; sub0[2] = 4'h6; sub0[3] = 4'h2;
sub0[4] = 4'ha; sub0[5] = 4'h5; sub0[6] = 4'hb; sub0[7] = 4'h9;
sub0[8] = 4'he; sub0[9] = 4'h8; sub0[10] = 4'hd; sub0[11] = 4'h7;
sub0[12]= 4'h0; sub0[13] = 4'h3; sub0[14] = 4'hf; sub0[15] = 4'h1;

sub1[0] = 4'h6; sub1[1] = 4'h8; sub1[2] = 4'h2; sub1[3] = 4'h3;
sub1[4] = 4'h9; sub1[5] = 4'ha; sub1[6] = 4'h5; sub1[7] = 4'hc;
sub1[8] = 4'h1; sub1[9] = 4'he; sub1[10] = 4'h4; sub1[11] = 4'h7;
sub1[12]= 4'hb; sub1[13] = 4'hd; sub1[14] = 4'h0; sub1[15] = 4'hf;

sub2[0] = 4'hb; sub2[1] = 4'h3; sub2[2] = 4'h5; sub2[3] = 4'h8;
sub2[4] = 4'h2; sub2[5] = 4'hf; sub2[6] = 4'ha; sub2[7] = 4'hd;
sub2[8] = 4'he; sub2[9] = 4'h1; sub2[10] = 4'h7; sub2[11] = 4'h4;
sub2[12]= 4'hc; sub2[13] = 4'h9; sub2[14] = 4'h6; sub2[15] = 4'h0;

sub3[0] = 4'hc; sub3[1] = 4'h8; sub3[2] = 4'h2; sub3[3] = 4'h1;
sub3[4] = 4'hd; sub3[5] = 4'h4; sub3[6] = 4'hf; sub3[7] = 4'h6;
sub3[8] = 4'h7; sub3[9] = 4'h0; sub3[10] = 4'ha; sub3[11] = 4'h5;
sub3[12]= 4'h3; sub3[13] = 4'he; sub3[14] = 4'h9; sub3[15] = 4'hb;

sub4[0] = 4'h7; sub4[1] = 4'hf; sub4[2] = 4'h5; sub4[3] = 4'ha;
sub4[4] = 4'h8; sub4[5] = 4'h1; sub4[6] = 4'h6; sub4[7] = 4'hd;
sub4[8] = 4'h0; sub4[9] = 4'h9; sub4[10] = 4'h3; sub4[11] = 4'he;
sub4[12]= 4'hb; sub4[13] = 4'h4; sub4[14] = 4'h2; sub4[15] = 4'hc;

sub5[0] = 4'h5; sub5[1] = 4'hd; sub5[2] = 4'hf; sub5[3] = 4'h6;
sub5[4] = 4'h9; sub5[5] = 4'hc; sub5[6] = 4'hc; sub5[7] = 4'ha;
sub5[8] = 4'hb; sub5[9] = 4'h7; sub5[10] = 4'h8; sub5[11] = 4'h1;
sub5[12]= 4'h4; sub5[13] = 4'h3; sub5[14] = 4'he; sub5[15] = 4'h0;

sub6[0] = 4'h8; sub6[1] = 4'he; sub6[2] = 4'h2; sub6[3] = 4'h5;
```

```
        sub6[4] = 4'h6; sub6[5] = 4'h9; sub6[6] = 4'h1; sub6[7] = 4'hc;
        sub6[8] = 4'hf; sub6[9] = 4'h4; sub6[10] = 4'hb; sub6[11] = 4'h0;
        sub6[12]= 4'hd; sub6[13] = 4'ha; sub6[14] = 4'h3; sub6[15] = 4'h7;

        sub7[0] = 4'h1; sub7[1] = 4'h7; sub7[2] = 4'he; sub7[3] = 4'hd;
        sub7[4] = 4'h0; sub7[5] = 4'h5; sub7[6] = 4'h8; sub7[7] = 4'h3;
        sub7[8] = 4'h4; sub7[9] = 4'hf; sub7[10] = 4'ha; sub7[11] = 4'h6;
        sub7[12]= 4'h9; sub7[13] = 4'hc; sub7[14] = 4'hb; sub7[15] = 4'h2;
        end

        assign out0 = sub0[in[31:28]];
        assign out1 = sub1[in[27:24]];
        assign out2 = sub2[in[23:20]];
        assign out3 = sub3[in[19:16]];
        assign out4 = sub4[in[15:12]];
        assign out5 = sub5[in[11:8]];
        assign out6 = sub6[in[7:4]];
        assign out7 = sub7[in[3:0]];

        assign out = {out0, out1, out2, out3, out4, out5, out6, out7};
endmodule
```

Файл subst.v

Сама таблица реализована в виде 8 блоков ROM (`sub0 – sub7`). На вход `in` подаётся значение блока, для которого производится операция замены. Данное значение разбивается на 4-битные блоки, которые в дальнейшем используются в качестве адреса соответствующего им блока ROM. Полученные значения от каждого блока конкатенируются и подаются на выход `out`.

4.2.3 Аппаратная реализация сдвиговых операций

В третьих, было расширено устройство сдвига посредством добавления в него возможности циклического сдвига. Данный вид сдвига не входит в базовую архитектуру MIPS-процессоа. Описание данного модуля содержит файл `shifter.v` .

```
module shifter (
        input wire [1:0]      sh_c,
        input wire [4:0]      shamt,
        input wire [31:0]     in,
```

```
    output wire[31:0]     out
    );

    wire [31:0] ls0, rs0, ls1, rs1, ls2, rs2, ls3, rs3, ls4, rs4;

    assign rs0 = shamt[0] ? {in[0], in[31:1]} : in;
    assign ls0 = shamt[0] ? {in[30:1], in[31]} : in;

    assign rs1 = shamt[1] ? {rs0[1:0], rs0[31:2]} : rs0;
    assign ls1 = shamt[1] ? {ls0[29:0], ls0[31:30]} : ls0;

    assign rs2 = shamt[2] ? {rs1[3:0], rs1[31:4]} : rs1;
    assign ls2 = shamt[2] ? {ls1[27:0], ls1[31:28]} : ls1;

    assign rs3 = shamt[3] ? {rs2[7:0], rs2[31:8]} : rs2;
    assign ls3 = shamt[3] ? {ls2[23:0], ls2[31:24]} : ls2;

    assign src = shamt[4] ? {rs3[15:0], rs3[31:16]} : rs3;
    assign slc = shamt[4] ? {ls3[15:0], ls3[31:16]} : ls3;

    assign out = sh_c[1] ? (sh_c[0]? slc : src) :
    (sh_c[0]? (in << shamt) : (in >> shamt));

endmodule
```

Файл shifter.v

В данном модуле циклический сдвиг реализован в поэтапном виде, т.е. каждый последующий этап реализует больший сдвиг, чем предыдущий. Данная схема наиболее оптимальна при сдвиге большого количества бит.

4.2.4 Итог

Таким образом, доля инструкций обмена данными с памятью во время выполнения основного алгоритма шифрования сведена к минимуму. Полная раелизация одного раунда основного алгоритма МАГМА занимает всего рабочих тактов процессора. В главе, посвящённой разработке ассемблера, приведён код реализации алгоритма шифрования на специально созданном языке ассемблера.

4.3 Оснащение процессора периферийными интерфейсами

В итоговый процессор была включена поддержка двух периферийных интерфейсов: RS-232 DB9 и DE9. Передача данных поверх этого интерфейса происходит по протоколу UART. Конечное устройство производит шифрование и дешифрование данных, передаваемых по данным интерфейсам. Данные интерфейсы были выбраны по причине их изначального присутствия на тестовой плате, относительной простоты их реализации, а так же потому, что ни данные интерфейсы, а тем более, ни шифраторы данных интерфейсов не пользуются популярностью.

Хотя компания Xilinx и распространяет готовый интерфейс для своих плат, было решено создать свою реализацию для достижения переносимости кода на платы других производителей и интеграции с остальной микроархитектурой устройства.

Описание модуля верхнего уровня интерфейса содержится в файле `uart.v`. Описания для разъёмов DB9 и DE9 абсолютно идентичны, а потому, нет смысла их дублировать.

```
module uart
      #(
      parameter
      DBIT =         8,
      SB_TICK =      16,
      DVSR =         81,
      DVSR_BIT =     7,
      FIFO_W =       2
      )
      (
      input wire            clk, reset,
      input wire            rd_uart, wr_uart, rx,
      input wire [7:0]      w_data,
      output wire           tx_full, rx_empty, tx,
      output wire [7:0]     r_data
      );

      wire          tick, rx_done_tick, tx_donr_tick;
      wire          tx_empty, tx_not_empty;
      wire [7:0]    tx_fifo_out, rx_data_out;

      tick_gen      #(.M(DVSR), .N(DVSR_BIT)) tg (.clk(clk),
      .reset(reset), .tick(tick));

      uart_rx       #(.DBIT(DBIT), .SB_TICK(SB_TICK))
      receiver (.clk(clk), .reset(reset), .rx(rx), .s_tick(tick),
```

```
        .rx_done_tick(rx_done_tick), .dout(rx_data_out) );

        fifo_r          #(.B(DBIT), .W(FIFO_W))
        fifo_rx (.clk(clk), .reset(reset), .rd(rd_uart),
        .wr(rx_done_tick), .w_data(rx_data_out),
        .empty(rx_empty), .r_data(r_data));

        fifo_t          #(.B(DBIT), .W(FIFO_W))
        fifo_tx (.clk(clk), .reset(reset), .rd(tx_done_tick),
        .wr(wr_uart), .w_data(w_data), .empty(tx_empty),
        .full(tx_full), .r_data(tx_fifo_out));

        uart_tx         #(.DBIT(DBIT), .SB_TICK(SB_TICK))
        transmitter (.clk(clk), .reset(reset), .tx_start(tx_not_empty),
        .s_tick(tick), .din(tx_fifo_out),
        .tx_done_tick(tx_done_tick), .tx(tx) );

        assign tx_not_empty = ~tx_empty;

endmodule
```

Файл uart.v

Интерфейс состоит из конфигурируемого генератора тактового сигнала `tick_gen`, приёмника `uart_rx`, передатчика `uart_tx` и соответствующих FIFO-регистров `fifo_r` и `fifo_t`.

Оба интерфейса связаны с регистром `$us` (uart state) в памяти. Данный регистр содержит информацию о наличии принятых данных в регистре приёмника и заполненности регистра передатчика. В случае переполнения регистра приёмника, происходит соответствующее прерывание.

4.4 Реализация программируемости устройства

Последней желаемой характеристикой устройства является его программируемость. В данной работе программируемость реализована несколько нестандартным методом. Причины тому описаны ниже.

Во-первых, созданное процессорное ядро имеет гарвардскую архитектуру, что означает, что память команд и память данных физически разделены друг от друга, а память команд не изменяется в процессе работы устройства.

Во-вторых, устройство реализовано на основе ПЛИС, что значит, что его структуру можно изменить в любой момент.

В результате, память команд в данном случае можно реализовать в виде ROM-памяти. Программируемость же означает реконфигурацию памяти команд в соответствии с неким программным кодом во время реконфигурации всей ПЛИС. Первым преимуществом данного подхода является простота его реализации. Вторым преимуществом является то, что при синтезе нетлиста происходит оптимизация тракта данных синтезатором, который отбрасывает незадействованную часть логики. Так, например при реализации шифратора, в конечном нетлисте не будет присутствовать таймер, описание которого, однако, присутствует в составе проекта.

Для программирования устройства выбран язык ассемблера MIPS. Для преобразования языка ассемблера в коды инструкций, был написан скрипт на языке AWK. Программные файлы данного преобразователя находятся в каталоге `asm`. Для создания более простого интерфейса преобразователя был написан shell-скрипт под названием `asm.sh`. Память программ описана в файле `instr_mem.v`.

```
`include "./include/registers.v"
`include "./include/funct_codes.v"

module instr_mem (
        input wire [10:0] addr,
        output wire [31:0] data
        );

        reg [31:0] mem [511:0];

        initial
                begin
                $readmemh("instr.data", mem, 0, 511);
                end

        assign data = mem[addr[10:2]];

endmodule
```

Файл instr_mem.v

Директива `$readmemh("instr.data", mem, 0, 511)` инициализирует память команд содержимым файла `instr.data`. Подразумевается, что последний содержит вывод преобразователя.

В случае шифратора, файл ассемблерного кода выглядит следующим образом:

```
        ori $s0, $s0, 31

        lui $t1, 0xfedc
        ori $t1, $t1, 0xba98

        lui $t0, 0x7654
        ori $t0, $t0, 0x3210

        lui $t3, 0xfedc
        ori $t3, $t3, 0xba98

        lui $t2, 0x7654
        ori $t2, $t2, 0x3210

init_male_encryption:
        andi $t4, $t4, 0x0

init_male_raund:
        ori $t5, $t0, 0x0000
        ckx $t0, $t4, $t0
        cs $t0, $t0
        slc $t0, $t0, 11
        xor $t0, $t0, $t1
        ori $t1, $t5, 0x0000
        addi $t4, $t4, 0x0001
        bne $t4, $s0, init_male_raund

        ori $t5, $t0, 0x0000
        ckx $t0, $t4, $t0
        cs $t0, $t0
        slc $t0, $t0, 11
        xor $t1, $t0, $t1
        ori $t0, $t5, 0x0000

init_female_encryption:
```

```
        andi $t4, $t4, 0x0

init_female_raund:
        ori $t6, $t2, 0x0000
        ckx $t2, $t4, $t2
        cs $t2, $t2
        slc $t2, $t2, 11
        xor $t2, $t2, $t3
        ori $t3, $t6, 0x0000
        addi $t4, $t4, 0x0001
        bne $t4, $s0, init_female_raund

        ori $t6, $t2, 0x0000
        ckx $t2, $t4, $t2
        cs $t2, $t2
        slc $t2, $t2, 11
        xor $t3, $t2, $t3
        ori $t2, $t6, 0x0000

        j uart_load


male_encryption:
        andi $t4, $t4, 0x0

male_raund:
        ori $t5, $t0, 0x0000
        ckx $t0, $t4, $t0
        cs $t0, $t0
        slc $t0, $t0, 11
        xor $t0, $t0, $t1
        ori $t1, $t5, 0x0000
        addi $t4, $t4, 0x0001
        bne $t4, $s0, male_raund

        ori $t5, $t0, 0x0000
        ckx $t0, $t4, $t0
        cs $t0, $t0
        slc $t0, $t0, 11
        xor $t1, $t0, $t1
        ori $t0, $t5, 0x0000
```

```
        j uart_load

female_encryption:
        andi $t4, $t4, 0x0

female_raund:
        ori $t6, $t2, 0x0000
        ckx $t2, $t4, $t2
        cs $t2, $t2
        slc $t2, $t2, 11
        xor $t2, $t2, $t3
        ori $t3, $t6, 0x0000
        addi $t4, $t4, 0x0001
        bne $t4, $s0, female_raund

        ori $t6, $t2, 0x0000
        ckx $t2, $t4, $t2
        cs $t2, $t2
        slc $t2, $t2, 11
        xor $t3, $t2, $t3
        ori $t2, $t6, 0x0000

        j uart_load

uart_load:
        andi $t7, $us, 0x0004
        beq $t7, $0, male_received

        andi $t7, $us, 0x0001
        beq $t7, $0, female_received

        j uart_load

male_received:
        lw $t8, 0xfff8($0)
        xor $t8, $t0, $t8
        sw $t8, 0xfffc($0)
        j male_encryption

female_received:
        lw $t8, 0xfffc($0)
```

```
xor $t8, $t2, $t8
sw $t8, 0xfff8($0)
j female_encryption
```

Файл cipher.asm

Параметр z режима работы шифра выбран равным единице, параметр n — восьми (см. стр. 16 гл. 2). В начале происходит инициализация — в сдвиговые регистры загружается синхропосылка и шифруется стандартным алгоритмом шифрования. Затем осуществляется безусловный переход к метке `uart_load`. Из регистра $us извлекается информация о состоянии UART-контроллеров. Если какой-либо из них не пуст, то из него загружается полученный байт, складывается по модулю 2 с соответствующим сдвиговым регистром и сохраняется в буфер другого контроллера. Затем происходит переход к стандартному алгоритму шифрования соответствующего регистра. Для обоих сдвиговых регистров код стандартного алгоритма шифрования дублирован, а не реализован в виде отдельной функции. Это сделано для достижения дополнительной оптимизации.

4.5 Характеристики итогового устройства

Общие характеристики конечного устройства приведены в таблице.

Таблица 1 — Характеристики итогового устройства

Характеристика	Значание
Тактовая частота	25 МГц (при реализации на Spartan3AN)
Тактов на инструкцию	1
Разраядность	32 бита
Алгоритм шифрования	ГОСТ 34.12 с длиной блока 64 бита "МАГМА"
Таймер	16-битный, конфигурируемый
Интерфейсы	LED x8, RS-232-DE9, RS232-DB9

На основании вышеизложенных характеристик, можно точно измерить возможные скорости шифрования устройства:

$$V_{ш} = f / N_{и} = 25000000/302 = 82781 \text{ (Блок/сек)}$$

$$1 \text{ бит} \le B \le 64 \text{ бита}$$

$$82781 \text{ бит} \le V_{ш} * B \le 662251 \text{ байт}$$

f – тактоваячастота;

$N_{и}$ – количество инструкци для шифрования одного блока;

B – размер одного блока;

$V_{ш}$ – количество шифруемых в секунду блоков.

Дальнейшие изменения скорости работы могут быть связаны со скоростью работы интерфейса.

4.6 Сравнение с возможными альтернативами устройства

Естественно, как уже отмечалось ранее, у получившегося устройства имеется ряд альтернатив, среди которых можно выделить три основные группы:

1. Готовые криптоконтроллеры;
2. Готовые контроллеры, содержащие криптосопроцессоры;
3. Софт-процессоры с оптимизированными модулями для криптографических вычислений.

Каждая группа обладает рядом преимуществ и недостатков по сравнению с остальными.

Действительно близкими по смыслу и функционалу к представленному устройству являются представители третьей группы. Существует ряд свободно распространяемых софт-процессоров, например, CryptoBlaze компании Xilinx. Данные устройства почти всегда являются оптимизированными для плат определённого производителя, что является как их положительной, так и отрицательной стороной, ведь при переносе на другие платы, чаще всего возникают проблемы.

Первые две группы являют собой готовые аппаратные блоки. Разделение на две группы производится толко по типам внутренней структуры, которая рассматриваться подробнее не будет, а потому в дальнейшем они будут объединены в одну. Представители данной группы обладают значительно большей производительностью по сравнению с софт-процессорами, так как являются готовыми отптимизированными аппаратными блоками, а софт-процессоры — по факту, лишь совокупностью запрограммированных ячеек. Так, например, контроллеры STM32F4xx компании STMicroelectronics способны работать на частоте до 168МГц, используя 32-битный тракт данных, что в результате приводит к скорости шифрования в разы большей, чем у получившегося устройства. Однако, софт-процессоры имеют гораздо болшую гибкость по сравнению с готовыми аппаратными блоками. Так, например, при необходимости реализовать алгоритм «МАГМА» на базе контроллера из серии STM32F4xx, пришлось бы реализовывать его на программном уровне, что заняло бы не меньше времени, чем описание оптимизированного модуля на Verilog или VHDL для софт процессора, а работало бы, возможно, несколько хуже и с затратой несколько больших ресурсов. Наконец, намного большие проблемы возникли бы, при необходимости реализовать шифрацию данных, передаваемых через специфичный канал передачи.

Учитывая, что достижение максимальной гибкости и переносимости является основной целью данной работы, можно сделать вывод о том, что полученное устройство является достойной

альтернативой существующим решениям стоящей проблемы.

ЗАКЛЮЧЕНИЕ

В ходе ВКР был решён ряд задач.

Была изучена необходимая для решения дальнейших задач теоретическая база.

Была составлена архитектура процессорного ядра, содержащая все необходимые средства для решения поставленной цели.

В качестве основного алгоритма шифрования был выбран алгоритм ГОСТ 32.12-2015 с длиной блока 64 бита или «МАГМА». Для реализации на его основе шифратора, использовался режим работы гаммирования с обратной связью по выходу.

Была разработана и описана микроархитектура процессорного ядра, реализующего составленную MIPS-подобную архитектуру на языке Verilog.

Для оптимизации выполнения выбранного криптоалгоритма разработанным микропроцессором, был расширен его тракт данных посредством введения в него модулей, реализующих следующие операции, присущие выбранному алгоритму:

1. Выработка раундовых ключей;
2. Операция подстановок;
3. Циклический сдвиг.

Был реализован и интегрирован с основным ядром интерфейс RS-232 и реализован потоковый шифратор протокола UART, используемого для передачи данных по этому интерфейсу для демонстрации возможностей полученного устройства.

Таким образом, было спроектировано и описано на языке Verilog процессорное ядро с трактом данных, оптимизированным для криптографических вычислений алгоритма ГОСТ 34.12-2015 с длиной блока 64 бита «МАГМА». На основе данного ядра было создано устройство потокового шифрования данных. Устройство реализовано и протестировано на плате Xilinx Spartan 3-AN. Устройство удовлетворяет всем предъявленным требованиям, оно показало неплохие результаты работы, сравнимые с аналогами. Явными его преимуществами являются простота адаптации к любым ситуациям и открытость и переносимость описания микроархитектуры.

СПИСОК ИСПОЛЬЗОВАННЫХ ЛИТЕРАТУРЫ И ИСТОЧНИКОВ

1. CryptoBlaze: 8-Bit Security Microcontrolleri. — v1.0. — XILINX, 2003. — 9 с.

2. David M. Harris and Sarah L. Harris. Digital Design and Computer Architecture. — 1-е изд. — Boston:Morgan Kaufman, 2007. — 570 с.

3. P. Pal Chaudhuri. Computer organisation and design. — 3-е изд. — Delhi:PHI Learning, 2014. — 897 с.

4. Pong P. Chu. FPGA prototyping by Verilog examples Xilinx Spartan-3 Version. — New Jersey:John Wiley & Sons, 2008. — 488 с.

5. Spartan-3A/3AN FPGA Starter Kit Board User Guidei. — v. 1.1. — XILINX, 2008. — 140 с.

6. The Institute of Electrical and Electronics Engineers. IEEE 1364-2001 IEEE Standard Verilog Hardware Description Language. — 3 Park Avenue, New York, NY 10016-5997, USA:The Institute of Electrical and Electronics Engineers, 2001i. — 778 с.

7. The Institute of Electrical and Electronics Engineers. IEEE Std 1076, 2000 Edition IEEE Standard VHDL Language Reference Manual. — 3 Park Avenue, New York, NY 10016-5997, USA: The Institute of Electrical and Electronics Engineers, 2000. — 299 с.

8. Самоделов А. Криптография в отдельном блоке: криптографический сопроцессор семейства STM32F4xx: [Электронный ресурс] // Официальный сайт компании "Компэл": URL: http://www.compel.ru/lib/ne/2012/6/4-kriptografiya-v-otdelnom-bloke-kriptograficheskiy-so-protsessor-semeystva-stm32f4xx. (дата обращения 03.12.2016).

9. Федеральное агентство по техническому регулированию и метрологии. ГОСТ Р 34.13-2015 Информационная технология КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА ИНФОРМАЦИИ Режимы работы блочных шифрові. — М.:Стандартинформ,2015. — 42 с.

10. Федеральное агентство по техническому регулированию и метрологии. ГОСТ Р 34.12-2015 Информационная технология КРИПТОГРАФИЧЕСКАЯ ЗАЩИТА ИНФОРМАЦИИ Блочные шифры. — М.:Стандартинформ,2015. — 25 с.

ПРИЛОЖЕНИЯ

Приложение А

СПИСОК ФАЙЛОВ НА CD-ДИСКЕ

1. Spartan-3A/3AN FPGA Starter Kit Board User Guide.
2. Файл adder.v;
3. Файл aludec.v;
4. Файл alu.v;
5. Файл contr.v;
6. Файл cpu.v;
7. Файл datapath.v;
8. Файл instr_mem.v;
9. Файл io_contr.v;
10. Файл kxor.v;
11. Файл maindec.v;
12. Файл mux2to1.v;
13. Файл mux3to1.v;
14. Файл mux4to1.v;
15. Файл PC.v;
16. Файл pc_val_mux.v;
17. Файл ram.v;
18. Файл reg_file.v;
19. Файл registers.v;
20. Файл rk_gen.v;
21. Файл func_codes.v;
22. Файл shifter.v;
23. Файл sign_ext.v;
24. Файл sll2.v;
25. Файл subst.v;
26. Файл timer.v;
27. Файл asm.sh;
28. Файл final.awk;
29. Файл init.awk;

30. Файл oplist.awk;

31. Файл cipher.asm.