**Министерство образования и науки Российской Федерации**

**федеральное государственное автономное образовательное учреждение высшего образования**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»**

**(Университет ИТМО)**

О Т Ч Ё Т

**о выполнении курсовой работы**

Тема задания: Анализ алгоритмов и методов описания аппаратуры для реализации аппаратного расширителя процессора SCR1, производящего операцию извлечения логарифма из целого числа.

Выполнил: студент группы P41193

Хлыстун Д.С.

Проверил: Березина Екатерина Михайловна

Санкт-Петербург

2020

**СОДЕРЖАНИЕ**

[1 ПОИСК АЛГОРИТМОВ ВЫЧИСЛЕНИЯ ЛОГАРИФМА ЛЮБОЙ СТЕПЕНИ ИЗ ЧИСЛА И ИХ АНАЛИЗ С ЦЕЛЬЮ ВЫЯВИТЬ ОПТИМАЛЬНЫЕ ВАРИАНТЫ ДЛЯ РЕАЛИЗАЦИИ АППАРАТНОГО МОДУЛЯ ВЫЧИСЛЕНИЯ ЛОГАРИФМА ИЗ ЧИСЛА ТИПА INT 3](#_Toc44897260)

[**1.1** **Первый метод** 3](#_Toc44897261)

[**1.2** **Второй метод** 8](#_Toc44897262)

[2 РАЗРАБОТКА АППАРАТНОГО МОДУЛЯ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ В СРЕДЕ VIVADO HLS. 12](#_Toc44897263)

[**2.1 Разработка аппаратного блока на языке программирования Си** 12](#_Toc44897264)

[**2.2 Анализ реализованного варианта.** 12](#_Toc44897265)

[3 UNIT-ТЕСТЫ В VIVADO 14](#_Toc44897266)

[**3.1 Первое тестирование** 14](#_Toc44897267)

[**3.2 Второе тестирование** 16](#_Toc44897268)

[**3.3 Третье тестирование** 19](#_Toc44897269)

[4 ВНЕСЕНИЕ ИЗМЕНЕНИЙ В ПРОЕКТ ПРОЦЕССОРА SCR1 ДЛЯ ДОБАВЛЕНИЯ ВЫБРАННОГО РАНЕЕ МОДУЛЯ 22](#_Toc44897270)

[5 ТЕСТИРОВАНИЕ ДОБАВЛЕННОЙ КОМАНДЫ В СОСТАВЕ SCR1 25](#_Toc44897271)

[**5.1 Оценка производительности** 25](#_Toc44897272)

[**5.2 Unit-тест в составе SCR1** 25](#_Toc44897273)

[ЗАКЛЮЧЕНИЕ 26](#_Toc44897274)

[ПРИЛОЖЕНИЕ 1 27](#_Toc44897275)

# 1 ПОИСК АЛГОРИТМОВ ВЫЧИСЛЕНИЯ ЛОГАРИФМА ЛЮБОЙ СТЕПЕНИ ИЗ ЧИСЛА И ИХ АНАЛИЗ С ЦЕЛЬЮ ВЫЯВИТЬ ОПТИМАЛЬНЫЕ ВАРИАНТЫ ДЛЯ РЕАЛИЗАЦИИ АППАРАТНОГО МОДУЛЯ ВЫЧИСЛЕНИЯ ЛОГАРИФМА ИЗ ЧИСЛА ТИПА INT

**Логарифм** положительного числа  по основанию   (обозначается ) — это показатель степени, в которую надо возвести , чтобы получить .

## **Первый метод**

Вычисление логарифмов довольно распространённая операция в цифровой обработке сигналов. Чаще приходится считать только свёртки (умножение с накоплением) и амплитуды с фазами. Как правило для вычисления логарифмов на FPGA применяется алгоритм CORDIC в гиперболическом варианте, требующий только таблицы и простых арифметических операций. Однако это не всегда бывает удобно, особенно если проект большой, кристалл маленький и начинаются проблемы с оптимизацией. В приведенном далее алгоритме не используется таблиц, но есть умножение, точнее возведение в квадрат. И поскольку схемотехнически возведение в квадрат проще общего случая умножения, возможно этот алгоритм представляет интерес для специализированных микросхем, хотя для FPGA разницы конечно нет.

Пусть есть число **X**. Требуется найти число **Y** такое, чтобы  .  
Положим так же что **X** лежит в интервале от 1 до 2. Это не слишком ограничивает общность, поскольку **X** всегда можно перевести в этот интервал умножением или делением на степень двойки. Для **Y** это будет означать добавление или вычитание целого числа, что делается легко. Итак, **X** лежит в интервале от 1 до 2. Тогда **Y** будет лежать в интервале от 0 до 1. Запишем **Y** как бесконечную двоичную дробь:

Коэффициенты  в этой записи есть ни что иное, как просто биты двоичного представления числа **Y**. Причём поскольку **Y** меньше 1, очевидно что .

Возведём первое уравнение в квадрат:   и как и ранее, запишем двоичное представление числа **2Y**. Очевидно, что

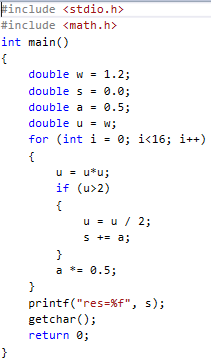
Т.е. биты    остались теми же, просто сдвинулись степени двойки. Бита  в представлении нет, потому что он равен нулю. Возможны два случая:

В первом случае в качестве нового значения **X** примем  , во втором-

В итоге задача свелась к прежней. Новое **X** опять лежит в интервале от 1 до 2, новое **Y** от 0 до 1. Но я узнал один бит результата. Делая такие же шаги в дальнейшем, можно получить сколько угодно битов **Y**.

После этого необходимо рассмотреть целочисленную реализацию.  
Пусть N-разрядные двоичные без знаковые числа, отображают интервал [0, 1]. Для удобства буду считать единицей число , а не , и соответственно двойкой число .

На листинге 1.1.1 представлен код программы такого алгоритма, написанный на языке С.



Листинг 1.1.1- код программы для первого алгоритма

Исследовав данный алгоритм, выявляется его серьезный недостаток- он может рассчитывать логарифм только лишь для значений, расположенных на интервале от [1;2].

В качестве примера посчитаем Как известно, любой логарифм от 1 равен 0. Удостоверимся в этом при помощи калькулятора(рис 1.1.2).

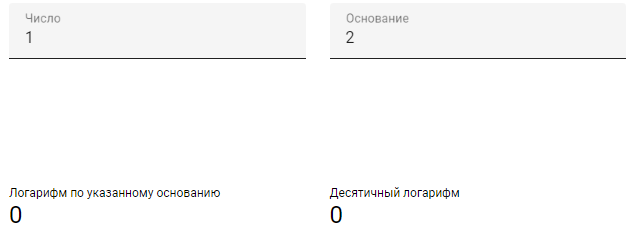


Рис. 1.1.2- расчет при помощи калькулятора.

Теперь посмотрим на результат вычисления программы (рис. 1.1.3).

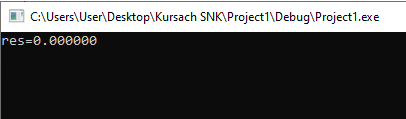


Рис. 1.1.3 – результат работы программы

Теперь вычислим .

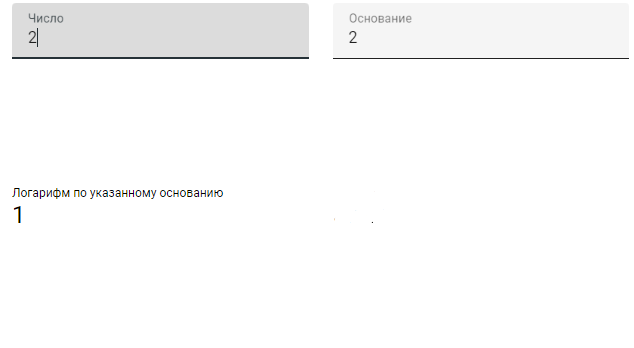


Рис. 1.1.4- расчет при помощи калькулятора.

И опять же вычислим при помощи программы (рис. 1.1.4).

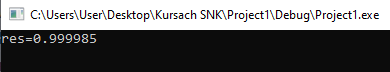


Рис. 1.1.4 –вычисление при помощи программы.

И, на всякий случай, проверим любое промежуточное значение, например- 1.5 (рис. 1.1.5).

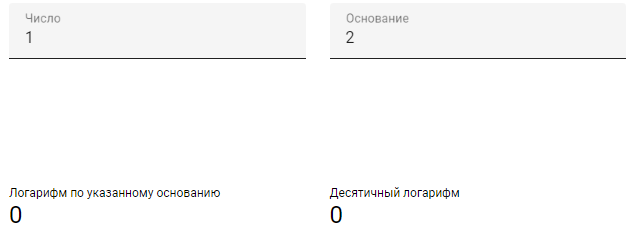


Рис. 1.1.5- расчет при помощи калькулятора.

Теперь посмотрим на результат вычисления программы (рис. 1.1.6).

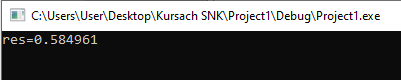


Рис. 1.1.6 – результат работы программы

Ну и в подтверждение слов, что данный метод не сможет посчитать логарифм из числа, которое больше двух, приведены еще несколько рисунков.

Вычислим логарифм из числа 2.1 (рис. 1.1.7).

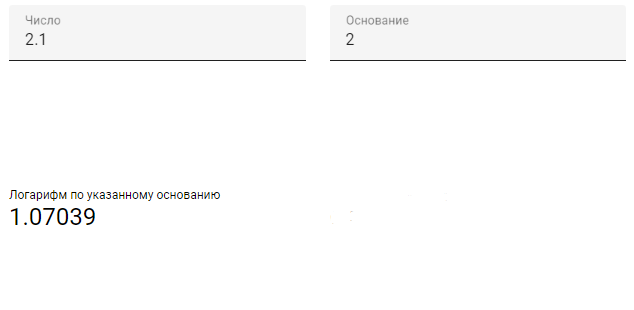


Рис. 1.1.7- расчет при помощи калькулятора.

Ну и то же самое вычисление при помощи программы (рис. 1.1.8).

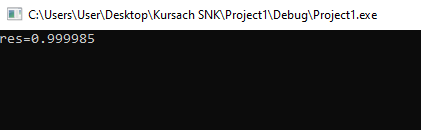


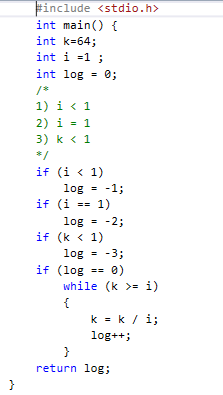
Рис. 1.1.8 – результат работы программы

Как можно заметить, присутствует погрешность, соответственно можно сделать вывод, что алгоритм работает неверно и внедрять его куда-либо не стоит.

Ранее в разделе были приведены

## **Второй метод**

Второй метод был придуман непосредственно мной, отталкиваясь от всем известного определения логарифма. Логарифм- это показатель степени, а если возвести число в определённую степень -это перемножить его само на себя несколько раз. То есть, чтобы взять логарифм по основанию от какого-либо числа, нужно это число делить на основание логарифма до тех пор, пока не перестанет происходить деление нацело. При этом, при каждом «удачном» делении на единицу будет наращиваться счетчик, значение которого в конечном итоге и будет результатом извлечения логарифма. Ниже приведен листинг программы для данного алгоритма (листинг 1.2.1).



Листинг 1.2.1 - код программы для второго алгоритма

К- это число, из которого нужно извлечь логарифм, i- основание логарифма, ну и непосредственно log- счетчик, который будет инкрементироваться, и, непосредственно, являться ответом.

Разберем работу алгоритма на примере кода программы из листинга.

Сделаем пару тестовых вычислений.

Например посчитаем приведенный выше при помощи калькулятора и программы. Калькулятор выдал такой результат (рис. 1.2.2).

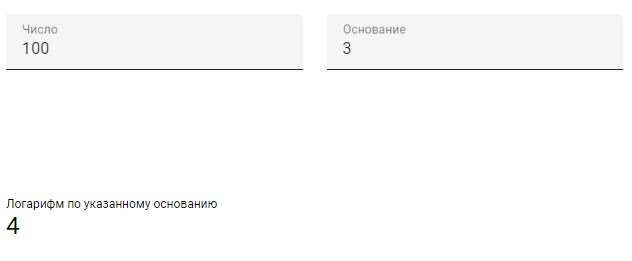


Рис. 1.2.2- расчет при помощи калькулятора.

Результат работы программы (рис. 1.2.3):

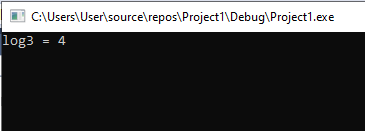


Рис. 1.2.3- результат работы программы

Ну и проверим еще какой-нибудь логарифм. Например, . Что выдаст калькулятор (рис. 1.2.4).

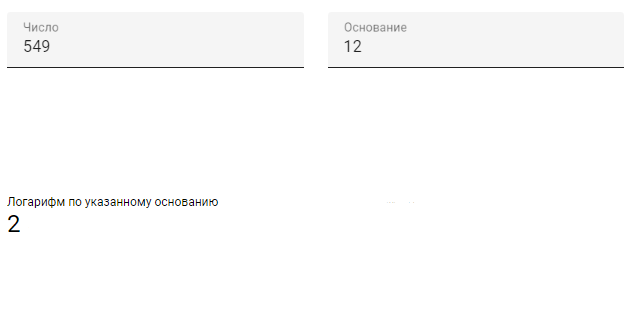


Рис. 1.2.2- расчет при помощи калькулятора.

Посмотрим на результат работы программы (рис.1.2.5).

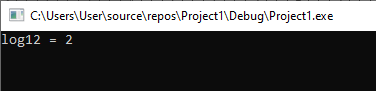


Рис. 1.2.5- результат работы программы

Так же стоит отметить, что в данном алгоритме реализованы проверки существования логарифма: проверяется, не отрицательное ли основание логарифма, не равно ли оно 0 и не меньше ли единицы непосредственно само число.

В ходе поиска идей для проверок логарифма был рассмотрен вариант с возвращением функции значения -1, если выполняется хоть одно условие не существования логарифма. Данный способ распространен, прост в реализации, и достаточно достоверный. Однако человеку, который не разбирается в области математики и хотел бы просто вычислить логарифм, нужна конкретика, чтобы понимать, какие именно введенные им данные некорректны.

Поэтому было реализовано следующее:

* в случае, если основание логарифма отрицательное- возвращается код ошибки -1;
* если основание логарифма равно 1, то возвращается код ошибки

-2

* и, если непосредственно само число меньше 1, то возвращается код ошибки -3.

Исходя из всего вышесказанного, можно удостовериться, что второй алгоритм работает, прост в реализации и возвращает код ошибки в случае ввода некорректных данных.

Исходя из этих критериев, второй алгоритм является лучшим и будет реализован далее.

# 2 РАЗРАБОТКА АППАРАТНОГО МОДУЛЯ НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ В СРЕДЕ VIVADO HLS.

## **2.1 Разработка аппаратного блока на языке программирования Си**

Для разработки посредством синтеза высокого уровня использована система автоматизированного проектирования Vivado HLS. Она предназначена для создания цифровых устройств с применением языков высокого уровня.

Согласно алгоритму, был составлен программный код на языке программирования Си (приложение 1). Он представляет из себя простой код на языке программирования высокого уровня, но был преобразован средой Vivado HLS в ходе синтеза в набор модулей \*.v (рисунок 2.1.). Код основного модуля представлен в приложении 2.

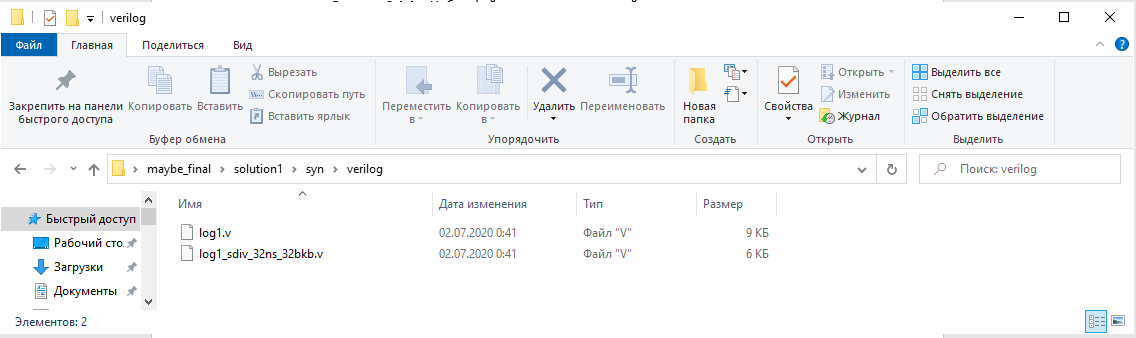


Рисунок 2.1.1 – Набор файлов, сгенерированный высокоуровневым синтезом

## **2.2 Анализ реализованного варианта.**

Для анализа реализованного аппаратного модуля использована среда разработки Vivado.

Для выполнения анализа модуля необходимо выполнить такие действия:

* добавить в проект тестирования сгенерированные средой Vivado HLS модули \*.v;
* запустить симуляцию (Run Simulation), при удачном выполнении которой будет сгенерирована осциллограмма;
* непосредственно, анализ самой осциллограммы.

Для тестового запуска (рисунок 2.2) были заданы такие параметры:

* основание логарифма (i) =12;
* само число, из которого нужно вычислить логарифм по основанию 12 (k) = 15010 = 9616.

Воспользовавшись калькулятором, можно убедиться, что , что и можно увидеть на осциллограмме – в ap\_ret [31:0] располагается 2.

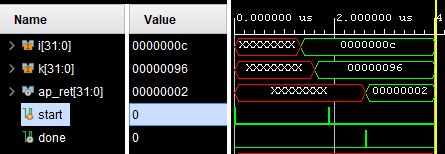


Рис.2.2 - результат работы модуля

# 3 UNIT-ТЕСТЫ В VIVADO

## **3.1 Первое тестирование**

Данное тестирование заключается проверке вычисления квадратного корня модулем следующих чисел:

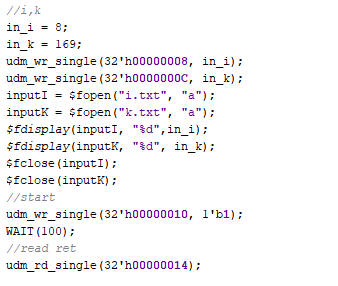
Само число:

* 0x08A136;
* 0x092A573;
* 0x08AEEE6;
* 0x02A67556;
* 0x351932;
* 0x0;
* 0x1;
* 0x2;
* 0x3;
* 0x4;
* 0xA9.

Основание логарифма:

* 0x58;
* 0xАВ;
* 0x5;
* 0x61;
* 0x43;
* 0x7;
* 0x2;
* 0x7;
* 0x3;
* 0x9;
* 0x8.

Для это используется отрезок код, представленный в листинге 3.1.1. Данный листинг включает отправку в модуль числа 0xA9, взведение бита начала работы модуля, а также считывания результата. Все действия также протоколируются в файлы со значениями чисел. Аналогичным образом тестируются остальные выше перечисленные значения (полный листинг занимает большой объем).



Листинг 3.1.1 – Отрезок кода для симуляции

На рисунке 3.1.2 представлены результаты вычислений, которые были записаны в файлы в ходе симуляции.

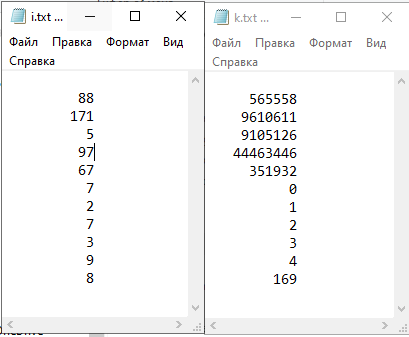


Рисунок 3.1.2 - Результаты выполнения симуляции

Полученные результаты помещены в таблицу Excel и сравнены с вычислением встроенной функции извлечения квадратного корня (рисунок 3.1.3).

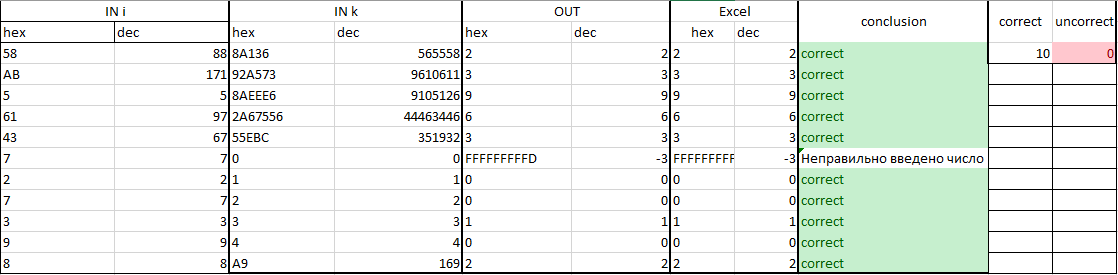
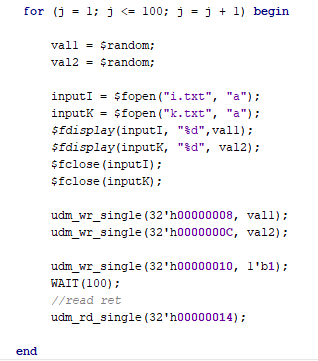


Рисунок 3.1.3 – Результаты первого тестирования

Данный тест показал, что в 10 случаях результаты вычислений модуля сходятся с результатами вычисления калькулятора и в одном случае возвращается код ошибки -3 (неправильно введено число, из которого нужно извлечь логарифм).

## **3.2 Второе тестирование**

Данное тестирование заключается в случайном определении входных данных в ходе симуляции. Для этого была использована функция $random языка описания аппаратуры Verilog в цикле из 100 итераций (листинг 3.2.1). На каждой итерации определяются новые основание логарифма и само число, после чего происходит отправка всех необходимых данных в модуль. Аналогично предыдущим тестам, все промежуточные значения и операции протоколируются в соответствующие файлы.



Листинг 3.2.1 – Код для симуляции

Значения, полученные функцией $random в ходе симуляции, были записаны в файл, после чего скопированы в таблицу Excel для дальнейшего сравнения, результаты которого представлены на рисунках 3.2.2, 3.2.3 и 3.2.4.

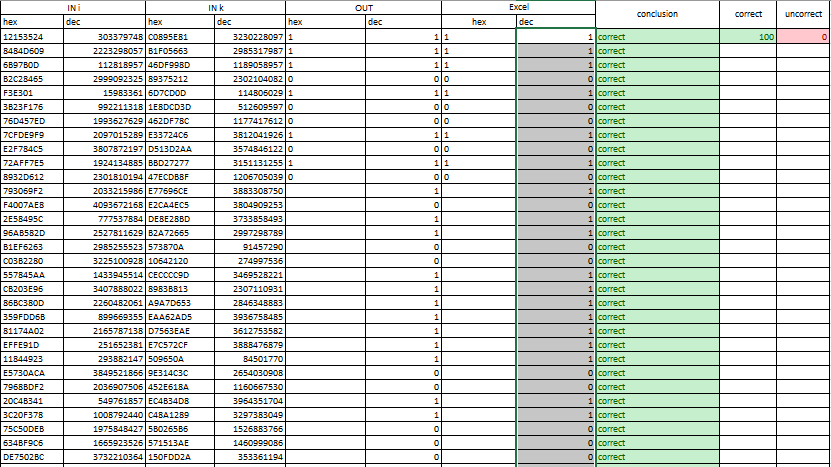


Рисунок 3.2.2 – Первый фрагмент сравнения

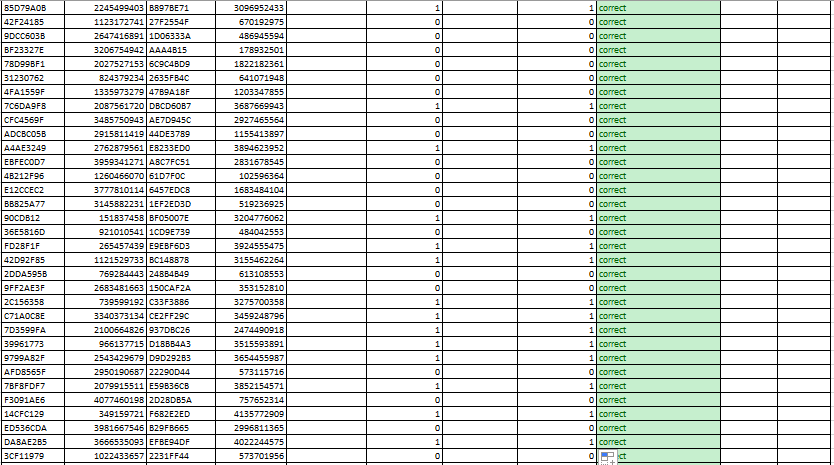


Рисунок 3.2.3 – Второй фрагмент сравнения

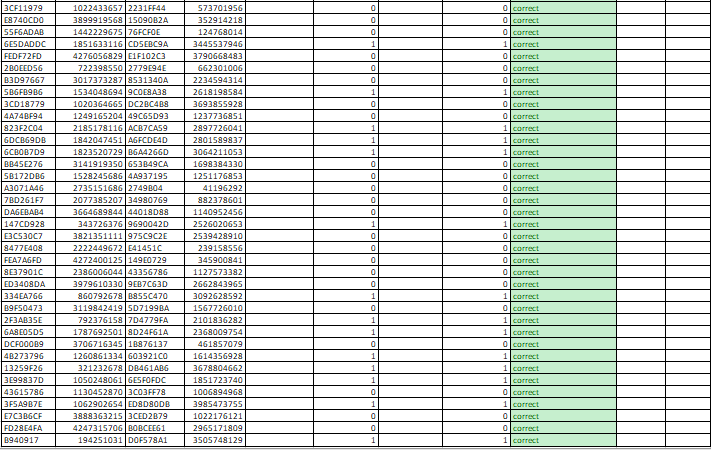
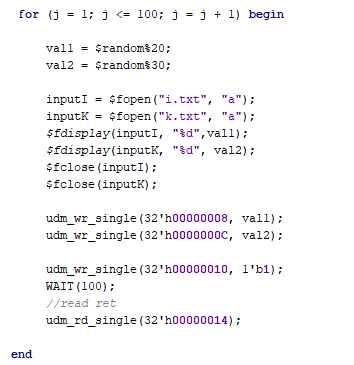


Рисунок 3.2.4 – Третий фрагмент сравнения

Проанализировав полученные данные, можно прийти к выводу, что в виду больших чисел как в основании логарифма, так и непосредственно и самих числах, которые нужно прологарифмировать, принято решение ограничить диапазон для функции $random.

## **3.3 Третье тестирование**

Ограничив диапазон для функции $random (листинг 3.3.1), получили следующие значения (рисунок 3.3.2-3.3.4).



Листинг 3.3.1- ограничение диапазона функции $random

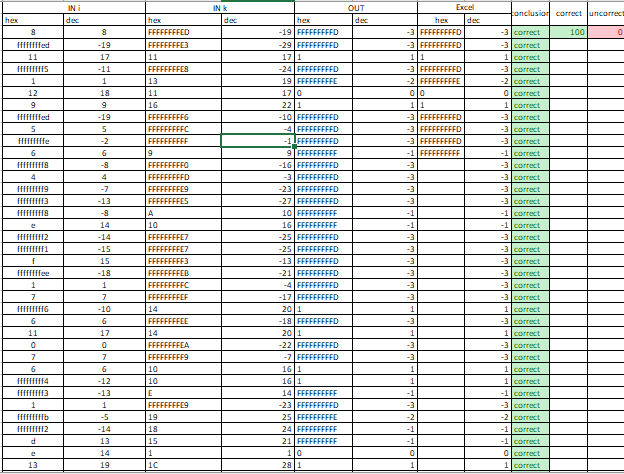


Рисунок 3.3.2 – Первый фрагмент сравнения

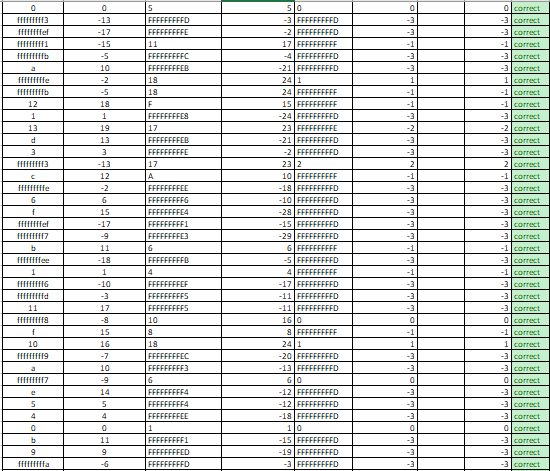


Рисунок 3.2.3 –Второй фрагмент сравнения

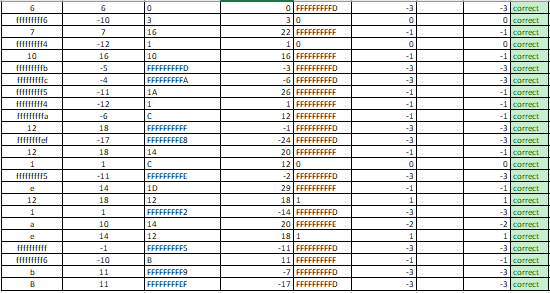


Рисунок 3.2.4 – Третий фрагмент сравнения

Проанализировав полученные данные, можно прийти к выводу, что данный модуль работает правильно- все его результаты сошлись с результатами калькулятора, а там, где были введены некорректные данные, модуль выдал соответствующий код ошибки:

* -1, когда основание логарифма было выбрано отрицательным;
* -2, если выбранное основание логарифма было равным 1;
* и, если непосредственно само число меньше 1, то возвращался код ошибки -3.

# 4 ВНЕСЕНИЕ ИЗМЕНЕНИЙ В ПРОЕКТ ПРОЦЕССОРА SCR1 ДЛЯ ДОБАВЛЕНИЯ ВЫБРАННОГО РАНЕЕ МОДУЛЯ

В начале необходимо составить 32-битное машинное представление для добавляемой инструкции. Исходя из набора инструкций, представленном в спецификации [7], поля в машинном представлении инструкции могут быть распределены, как показано на рисунке 4.1.

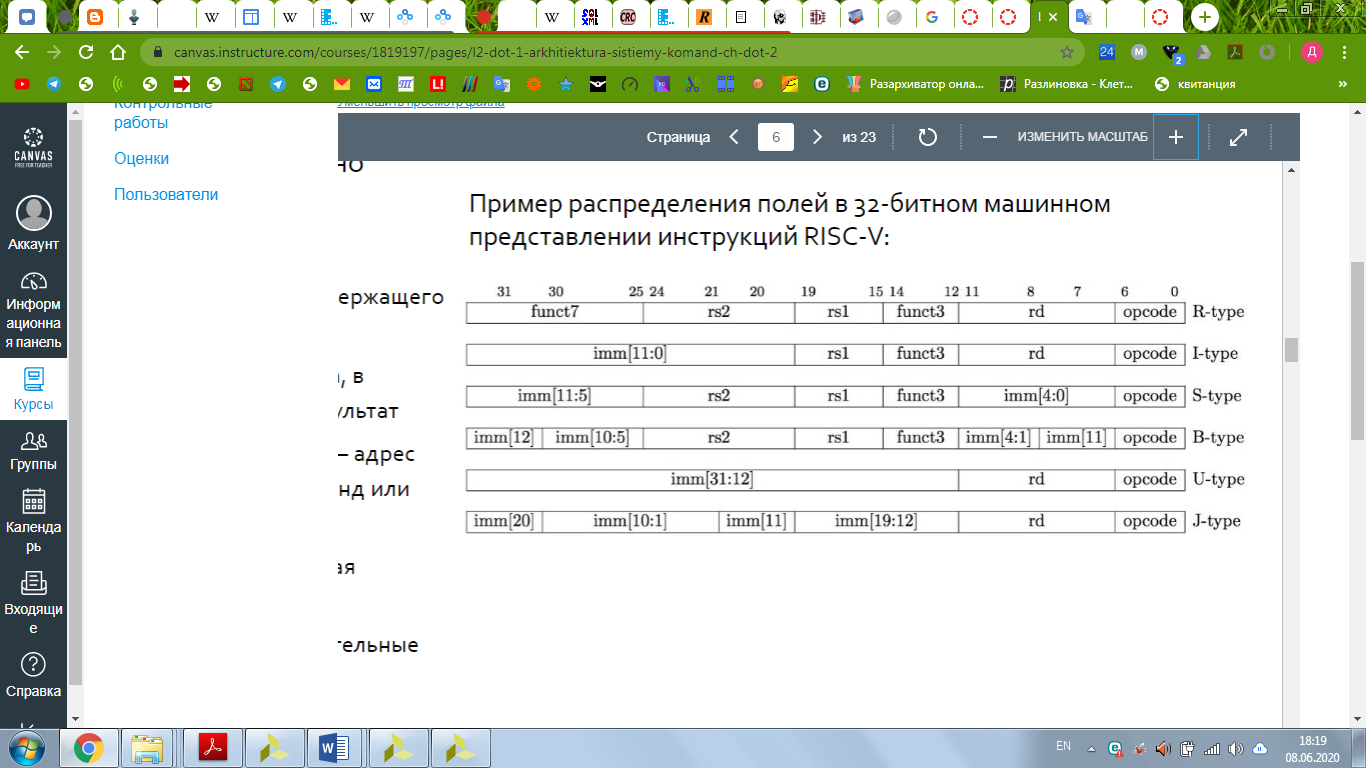


Рисунок 4.1 – Пример распределения полей в 32-битном машинном представлении инструкций

Поле добавляемой инструкции funct7 будет содержать код .

После funct3 будет содержать код .

Для того, чтоб процессор мог обработать добавленную инструкцию, необходимо внести изменения в декодере, в файле проекта scr1\_riscv\_isa\_decoding.svh (рисунок 4.2).

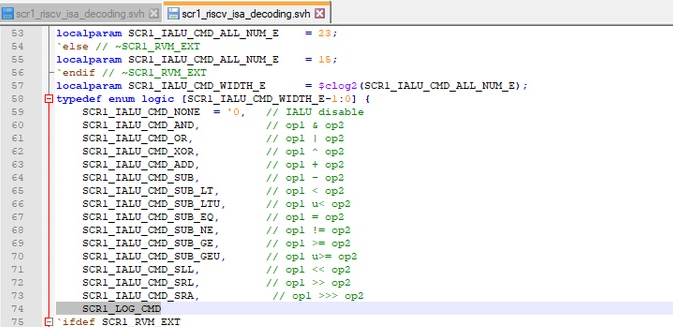


Рисунок 4.2- добавление сигнала, обозначающего операцию

Далее требовалось внести изменения в файл scr1\_pipe\_idu.sv (рисунок 4.3), чтобы сигнал операции заносился в ialu через соответствующие funct7 и funct3.

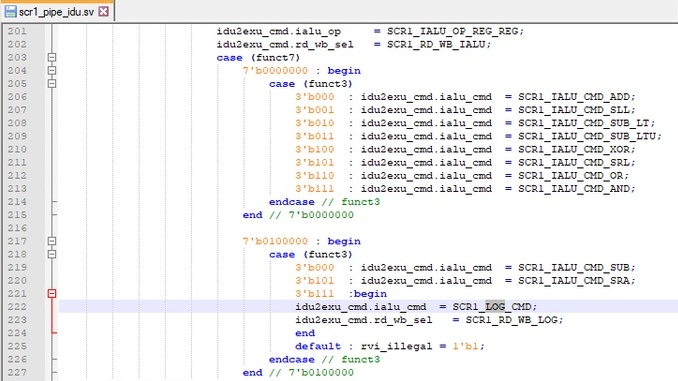


Рисунок 4.3 – Изменения в файле scr1\_pipe\_idu.sv

Следующим шагом были внесения изменений в файл scr1\_pipe\_exu.sv (рисунок 4.4). Этими изменениями я добавил в проект SCR1 непосредственно свой модуль и подключил сигналы к нему.

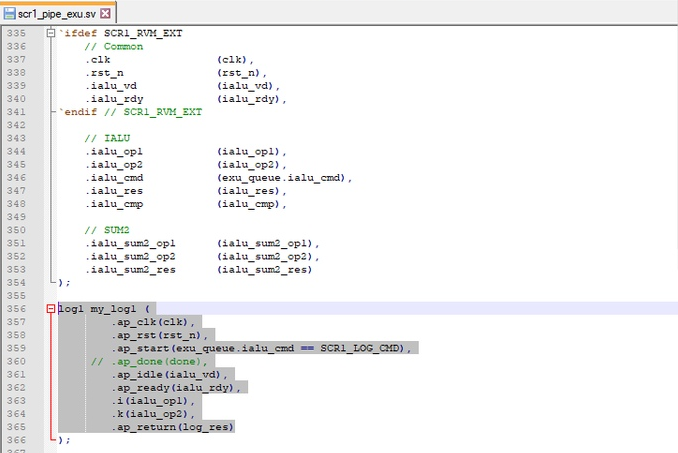


Рисунок 4.4 – Изменения в файле scr1\_pipe\_exu.sv

Следующим изменением все в том же файле записал результат работы моего модуля (рисунок 4.5).

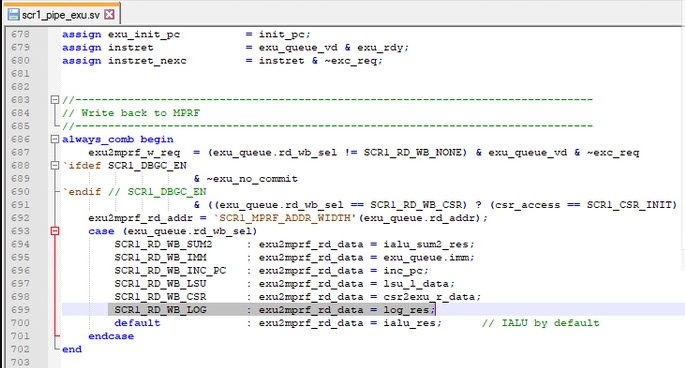


Рисунок 4.4 – запись результата работы модуля

# 5 ТЕСТИРОВАНИЕ ДОБАВЛЕННОЙ КОМАНДЫ В СОСТАВЕ SCR1

## **5.1 Оценка производительности**

## **5.2 Unit-тест в составе SCR1**

Для проверки команды необходимо составить тест, который проверит вычисление квадратного корня из следующих значений:

* основание логарифма I =12;
* само число k = 150.

Согласно спецификации SCR1 [10] проект адаптирован на проверку – содержит реализованные конструкции тестирования инструкций. В данном случае необходимо использовать конструкцию-макрос   
 TEST\_LOG(1, .word 0x4020F1B3, 2, 12, 150).

Был выполнен запуск симуляции и получен провальный результат выполнения теста (рис. 5.2.1).

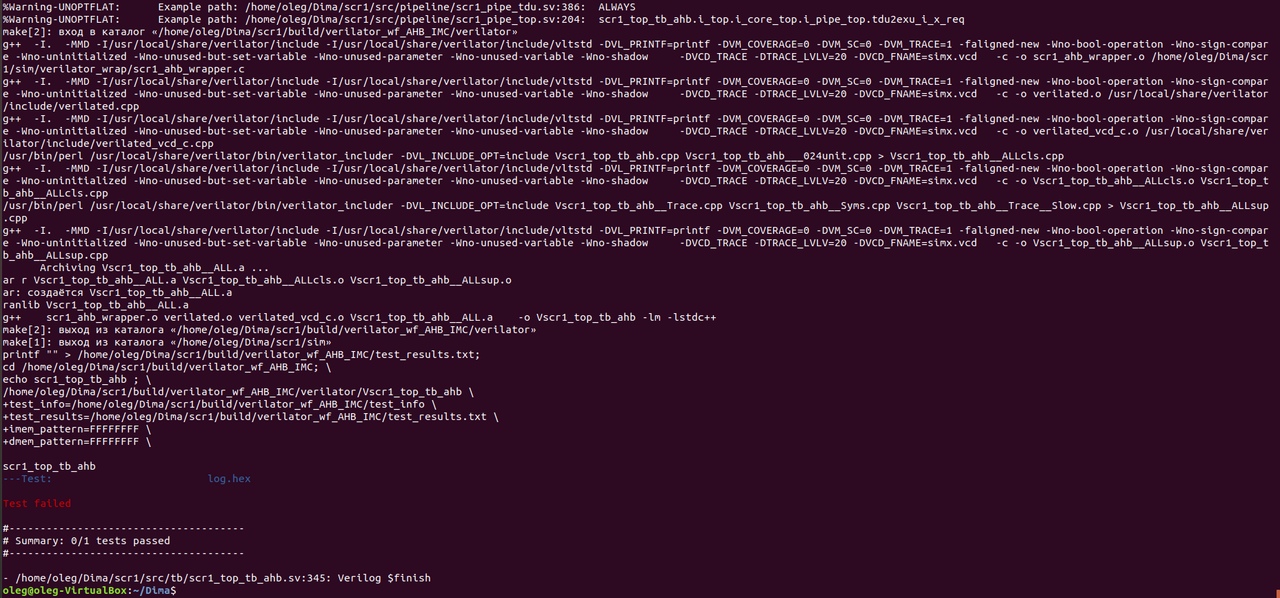


Рисунок 5.2.1 – Результат тестирования в симуляторе Verilator

# ЗАКЛЮЧЕНИЕ

В ходе работы были рассмотрены алгоритмы и методы описания аппаратуры для реализации аппаратного расширителя процессора SCR1, производящего операцию извлечения квадратного корня из целого числа.

В первом разделе рассмотрены алгоритмы вычисления логарифма из целого числа, приведены примеры расчета и определены лучший вариант алгоритма для реализации аппаратного модуля.

После выбора алгоритма был реализован аппаратный модуль с помощью синтеза высокого уровня в программе Vivado HLS.

После модуль был добавлен в проект SCR1, а также произведены соответствующие изменения микроархитектуры процессора. Модуль в составе проекта был протестирован в симуляторе Verilator.

# ПРИЛОЖЕНИЕ 1

**ЛИСТИНГ КОДА ТЕСТА ДЛЯ СИМУЛЯЦИИ В VERILATOR**

#define LOG\_OPCODE 0x40007033

#define RD\_SHIFT 7

#define RD\_MASK (0xF << RD\_SHIFT)

#define RS1\_SHIFT 15

#define RS1\_MASK (0xF << RS1\_SHIFT)

/\* rd, rs1, rs2 - integer registers num 0-31 \*/

#define INST\_LOG(rd, rs1, rs2) \

.word (LOG\_OPCODE | ((rd) << RS1\_SHIFT & RS1\_MASK) | \

((rs1) << RS1\_SHIFT & RS1\_MASK))

//TEST\_CASE(1, x2, 4, INST\_ADD(2, 1, 1));

#define TEST\_LOG( testnum, inst, result, val1, val2 ) \

TEST\_CASE( testnum, x3, result, \

li x1, MASK\_XLEN(val1); \

li x2, MASK\_XLEN(val2); \

inst; \

)

RVTEST\_RV32U

RVTEST\_CODE\_BEGIN

#-------------------------------------------------------------

# Arithmetic tests begin

#-------------------------------------------------------------

TEST\_LOG(1, .word 0x4020F1B3, 2, 12, 150);

TEST\_PASSFAIL

RVTEST\_CODE\_END

.data

RVTEST\_DATA\_BEGIN

TEST\_DATA

RVTEST\_DATA\_END