**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 квадратного корня из целого числа, оптимизированного для максимальной производительности.

Выполнил: студент группы P41193

Симак А.С.

Проверил: Березина Екатерина Михайловна

Санкт-Петербург

2020

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 4](#_Toc44613044)

[1 ПОИСК АЛГОРИТМОВ ВЫЧИСЛЕНИЯ КВАДРАТНОГО КОРНЯ ИЗ ЧИСЛА И ИХ АНАЛИЗ С ЦЕЛЬЮ ВЫЯВИТЬ ОПТИМАЛЬНЫЕ ВАРИАНТЫ ДЛЯ РЕАЛИЗАЦИИ АППАРАТНОГО МОДУЛЯ ВЫЧИСЛЕНИЯ КВАДРАТНОГО КОРНЯ ИЗ ЧИСЛА ТИПА INT 5](#_Toc44613045)

[1.1 Вычитание 5](#_Toc44613046)

[1.2 Итерационная формула Герона 6](#_Toc44613047)

[1.3 Битовые участки числа в двоичном представлении 7](#_Toc44613048)

[2 РАЗРАБОТКА НЕСКОЛЬКИХ АППАРАТНЫХ МОДУЛЕЙ НА ЯЗЫКЕ ОПИСАНИЯ АППАРАТУРЫ VERILOG И ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ В СРЕДЕ VIVADO HLS. 9](#_Toc44613049)

[2.1 Разработка аппаратного блока на языке программирования Си 9](#_Toc44613050)

[2.2 Разработка аппаратного блока на языке описания аппаратуры Verilog 10](#_Toc44613051)

[2.3 Анализ реализованных вариантов для выявления оптимального по быстродействию. 11](#_Toc44613052)

[3 UNIT-ТЕСТЫ В VIVADO 13](#_Toc44613053)

[3.1 Первое тестирование 13](#_Toc44613054)

[3.2 Второе тестирование 16](#_Toc44613055)

[3.3 Третье тестирование 18](#_Toc44613056)

[4 ВНЕСЕНИЕ ИЗМЕНЕНИЙ В ПРОЕКТ ПРОЦЕССОРА SCR1 ДЛЯ ДОБАВЛЕНИЯ ВЫБРАННОГО РАНЕЕ МОДУЛЯ 21](#_Toc44613057)

[5 ТЕСТИРОВАНИЕ ДОБАВЛЕННОЙ КОМАНДЫ В СОСТАВЕ SCR1 24](#_Toc44613058)

[5.1 Тестирование производительности 24](#_Toc44613059)

[5.2 Unit-тест в составе SCR1 26](#_Toc44613060)

[ЗАКЛЮЧЕНИЕ 29](#_Toc44613061)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 30](#_Toc44613062)

[ПРИЛОЖЕНИЕ 1 31](#_Toc44613063)

[ПРИЛОЖЕНИЕ 2 32](#_Toc44613064)

[ПРИЛОЖЕНИЕ 3 43](#_Toc44613065)

[ПРИЛОЖЕНИЕ 4 44](#_Toc44613066)

[ПРИЛОЖЕНИЕ 5 46](#_Toc44613067)

# ВВЕДЕНИЕ

Целью данной работы является анализ алгоритмов и методов описания аппаратуры для реализации аппаратного расширителя процессора SCR1, производящего операцию извлечения квадратного корня из целого числа, оптимизированного для максимальной производительности.

В первом разделе будет рассмотрено несколько алгоритмов вычисления квадратного корня и представлены соответствующие примеры расчета для дальнейшего выявления оптимальных вариантов для реализации аппаратного модуля вычисления квадратного корня из числа типа INT.

Второй раздел посвящен разработке аппаратных модулей извлечения квадратного корня. Для реализаций будут использованы 2 рассмотренных в прошлом разделе алгоритма. Разработка модуля посредством синтеза высокого уровня на языке программирования Си будет произведена на основе итерационной формулы Герона. Разработка на языке описания аппаратуры Verilog будет произведена на основе третьего рассмотренного алгоритма.

В третьем разделе описаны действия и изменения для добавления аппаратного расширителя в проект процессора SCR1.

В четвертом разделе произведено тестирование работоспособности аппаратного расширителя набора команд.

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

**Описание этапа:**

В данном разделе будет рассмотрено несколько алгоритмов вычисления квадратного корня (источники литературы № 1, 2, 3) и представлены соответствующие примеры расчета для дальнейшего выявления оптимальных вариантов для реализации аппаратного модуля вычисления квадратного корня из числа типа INT.

## 1.1 Вычитание

Для квадратов чисел верны следующие равенства [1]:

* 1 = ;
* 1 + 3 = ;
* 1 + 3 + 5 = ;
* и так далее.

То есть, узнать целую часть квадратного корня числа можно, вычитая из него все нечётные числа по порядку, пока остаток не станет меньше следующего вычитаемого числа или равен нулю, и посчитав количество выполненных действий. Например, для числа 9 квадратный корень будет вычислен следующим образом:

1. Отнимем от исходного числа первое нечетное число: 9 − 1 = 8
2. Отнимем от получившегося на прошлой итерации результата следующее нечетное число: 8 − 3 = 5
3. Выполним действие, аналогичное прошлому шагу: 5 − 5 = 0

Для исследуемого числа количество итераций по алгоритму равно 3, следовательно, согласно данному алгоритму квадратный корень из 9 равен 3.

Алгоритм является простым для понимания и реализации, но обладает следующими недостатками:

* производительность аппаратного блока будет падать при увеличении числа, что обусловлено ростом количества итераций;
* алгоритм является итеративным, что усложняет его аппаратную реализацию.

## 1.2 Итерационная формула **Герона**

Итерационная формула (1.1) Герона имеет следующий вид [2]:

, (1.1)

где — фиксированное положительное число, а — любое положительное число. Итерации по представленной формуле (1.1) выполняются до тех пор, пока абсолютное значение разности результатов не будет меньше либо равно заданной точности.

Рассмотрим пример (таблица 1.1). В качестве числа для извлечения квадратного корня возьмем 25, первым предположением будет число 3, а точность равна 0.001.

Таблица 1.1 – Пример вычисления квадратного корня из числа 25, используя итерационную формулу Герона

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  Итерации |  |  | Результат |  |
| 1 | 3 |  | 5.67 | 2.67 |
| 2 | 5.67 |  | 5.04 | 0.63 |

Продолжение таблицы 1.1

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 3 | 5.04 |  | 5 | 0.04 |
| 4 | 5 |  | 5 | 0 |

На последней итерации был получен результат 5, что соответствует значению квадратного корня из числа 25.

Данный алгоритм является также простым для понимания и реализации, а также более оптимальным по сравнению с ранее рассмотренным. Несмотря на то, что он также является итерационным, количество шагов значительно меньше даже для больших чисел, что дает бо́льшую производительность. Он также обладает существенным недостатком – необходимо реализовать дополнительно операцию деления, которая не является базовой в языке описания аппаратуры Verilog. В таком случае, целесообразно использовать синтез высокого уровня на языке программирования Си в среде разработки Vivado HLS.

## 1.3 Битовые участки числа в двоичном представлении

Данный алгоритм основан на операциях с битами числа в двоичном представлении [3] и будет рассмотрен на примере числа 169. Так как результирующее число операции извлечения квадратного корня будет иметь в 2 раза меньше бит, чем у исходного, запишем число в двоичной форме и, начиная со старшего разряда, выделим участки, кратные двум битам (рис. 1.1). Для частного случая числа 169 количество разрядов – 8 бит. Соответственно, результат извлечения квадратного корня будет занимать 4 бита, а количество выделенных участков также будет равно 4.



Рисунок 1.1 – Выделение участков

1. Возьмем первый участок числа из двух бит, равный . Число, квадрат которого будет меньше или равен первому участку, равно ( = . Так как < , запишем в искомое число .
2. Дальше берем второй участок, равный . В искомом слове необходимо дописать еще одну : . = < . Следовательно, искомое число остается равным .
3. Третий участок: . Дописываем в конец искомого числа : . Проверяем: = > . Так как число получилось больше, чем на третьем участке, вместо в конец искомого числа поставим . Тогда искомое число станет равным .
4. Четвертый участок равен . В конец искомого числа ставим : . = < . Следовательно, искомое число равно , что соответствует квадратному корню из числа 169.

Данный алгоритм сложен в понимании и требует длительного времени реализации, но полностью подходит для разработки аппаратного блока на языке описания аппаратуры Verilog. Выполнение операций происходит с битами числа, что позволяет выполнить расчет всех бит результирующего числа параллельно. Это является преимуществом в плане производительности. Все операции в алгоритме являются базовыми для языка описания аппаратуры Verilog.

# 2 РАЗРАБОТКА НЕСКОЛЬКИХ АППАРАТНЫХ МОДУЛЕЙ НА ЯЗЫКЕ ОПИСАНИЯ АППАРАТУРЫ VERILOG И ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ В СРЕДЕ VIVADO HLS.

**Описание этапа:**

Данный раздел посвящен разработке аппаратных модулей извлечения квадратного корня. Для реализаций будут использованы 2 рассмотренных в прошлом разделе алгоритма. Разработка модуля посредством синтеза высокого уровня на языке программирования Си будет произведена на основе итерационной формулы Герона. Разработка на языке описания аппаратуры Verilog будет произведена на основе третьего рассмотренного алгоритма. Использованы источники литературы № 4, 5, 6.

## 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 [4].

Согласно алгоритму, был составлен программный код на языке программирования Си [5] (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 и скриптов \*.tcl (рисунок 2.1). Код основного модуля представлен в приложении 2. Можно заметить, что код является очень габаритным, то есть будет занимать большую площадь на микросхеме. Также код содержит включение других модулей и требует включение дополнительных компонентов микросхемы с помощью скриптов, сгенерированных средой разработки. Это все усложняет внедрение модуля в готовый проект со сложной архитектурой и микроархитектурой.

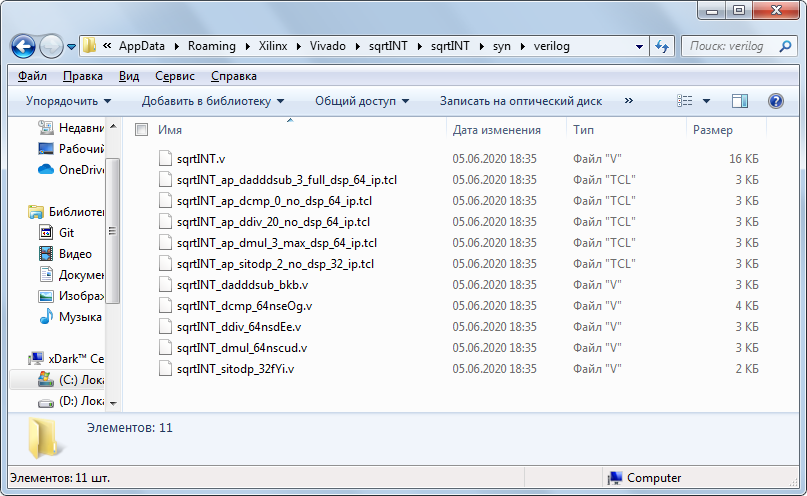


Рисунок 2.1 – Набор файлов, сгенерированный высокоуровневым синтезом

## 2.2 Разработка аппаратного блока на языке описания аппаратуры Verilog

Для разработки аппаратного блока на языке описания аппаратуры Verilog[6] была использована среда разработки Vivado.

Согласно третьему рассмотренному алгоритму, был составлен программный код модуля извлечения квадратного корня, который представлен в приложении 3. Как и было сказано ранее, алгоритм позволяет выполнить расчет всех бит числа параллельно, что и реализовано в коде. Данный модуль требует намного меньше логических элементов, следовательно, занимаемая площадь на микросхеме заметно меньше, чем при прошлой реализации. Также из-за коротких логических путей модуль выполняет вычисление квадратного корня из любого числа за 1 такт.

## 2.3 Анализ реализованных вариантов для выявления оптимального по быстродействию.

Для сравнения быстродействия реализованных аппаратных модулей использована среда разработки Vivado.

Сначала протестируем модуль, реализованный по второму алгоритму. Необходимо добавить в проект тестирования сгенерированные средой Vivado HLS модули \*.v, а также запустить скрипты \*.tcl в консоли “Tcl Console” среды Vivado. После данных манипуляций следует запустить симуляцию, по окончании которой будет сгенерирована осциллограмма. Она является наглядным представлением правильной работы модуля, а также его производительности – предоставляет время выполнения извлечения квадратного корня исследуемым аппаратным блоком.

На рисунке 2.2 представлена осциллограмма, полученная после выполнения симуляции с модулем, разработанным по второму алгоритму посредством синтеза высокого уровня в среде Vivado HLS. В качестве числа для тестирования используется 169, квадратный корень которого равен 13.

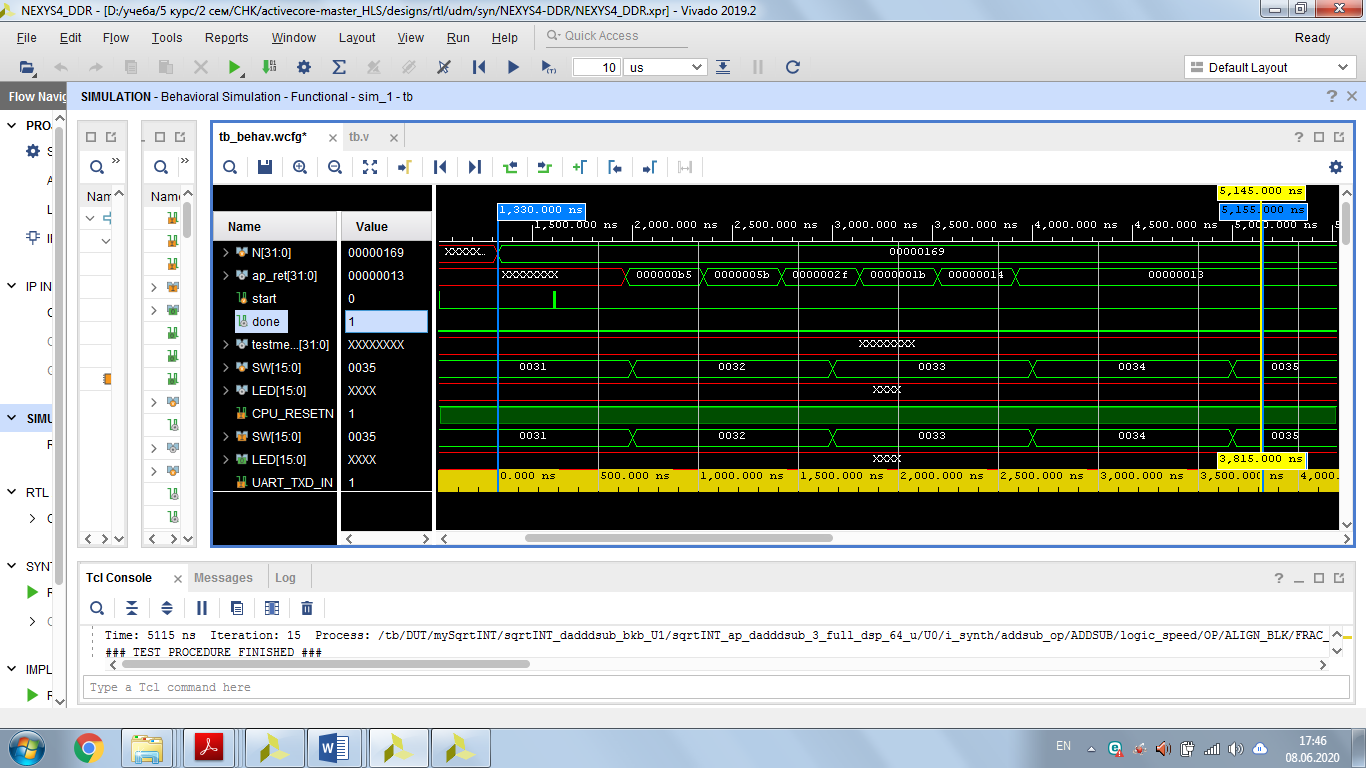


Рисунок 2.2 – Осциллограмма после выполнения симуляции с модулем по второму алгоритму

На осциллограмме можно наблюдать, что модуль извлекает квадратный корень верно: в N[31:0] располагается исходное число 169; в ap\_ret[31:0] содержится число 13 в момент взведения флага о завершении вычисления “done”. Также осциллограмма показывает, что выполнение извлечения корня от момента прихода входных данных (числа, из которого извлекается корень) до окончания заняло длительное время – 3,815.000 ns.

На рисунке 2.3 представлен осциллограмма, полученная после симуляции с вторым модулем извлечения квадратного корня. Выходными данными так же выступает число 169.

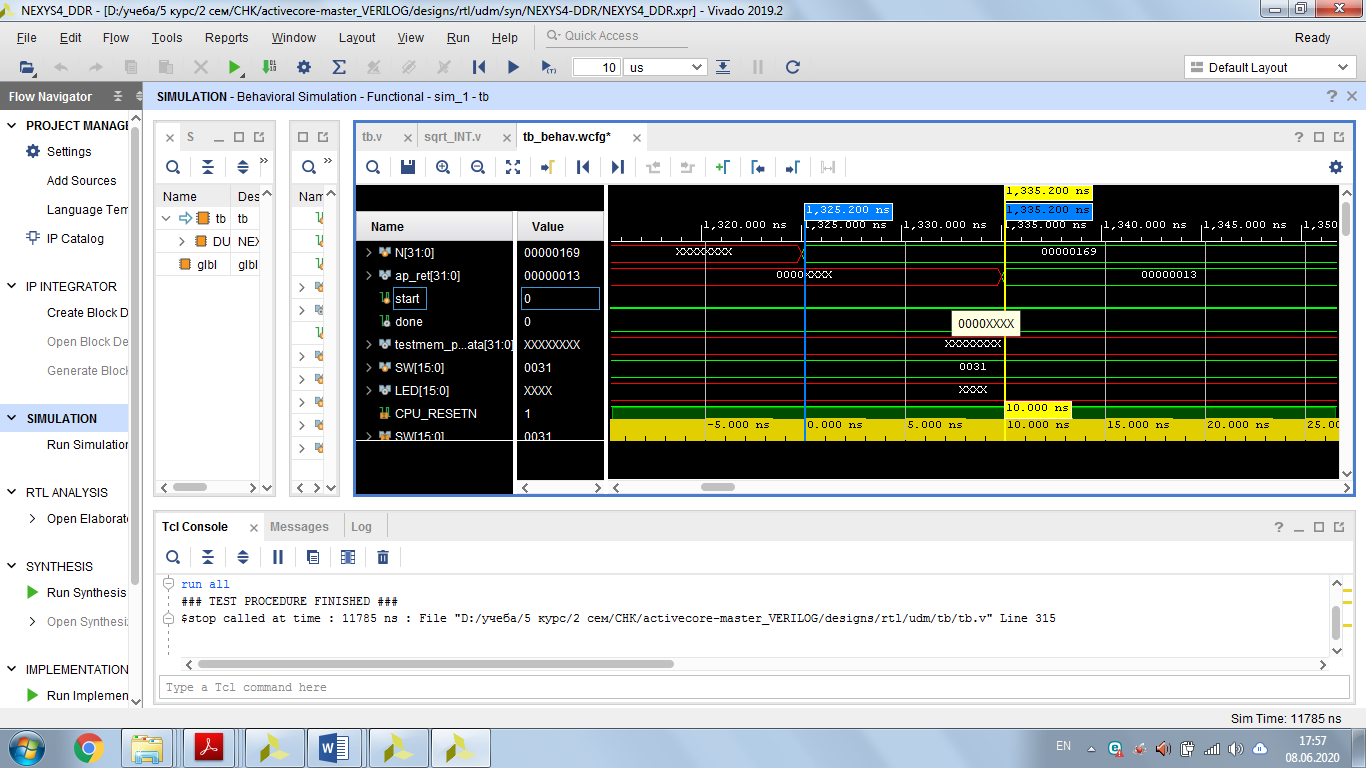


Рисунок 2.3 – Осциллограмма после выполнения симуляции с модулем по третьему алгоритму

На осциллограмме можно наблюдать, что модуль извлекает квадратный корень верно: в N[31:0] располагается исходное число 169; в ap\_ret[31:0] содержится число 13 спустя 10 ns после поступления данных. Длительность выполнение извлечения квадратного корня данным модулем намного меньше, чем рассмотренным ранее, следовательно, целесообразно будет использовать его в качестве аппаратного расширителя набора команд процессора SCR1.

# 3 UNIT-ТЕСТЫ В VIVADO

**Описание этапа:**

Данный раздел посвящен тестированию выбранного модуля, реализованного на языке описания аппаратуры Verilog. Оно заключается в сравнении результатов выполнения функции извлечения квадратного корня программы для работы с электронными таблицами Excel c результатами вычисления в модуле. Соответственно, само сравнение резултатов проходило в Excel.

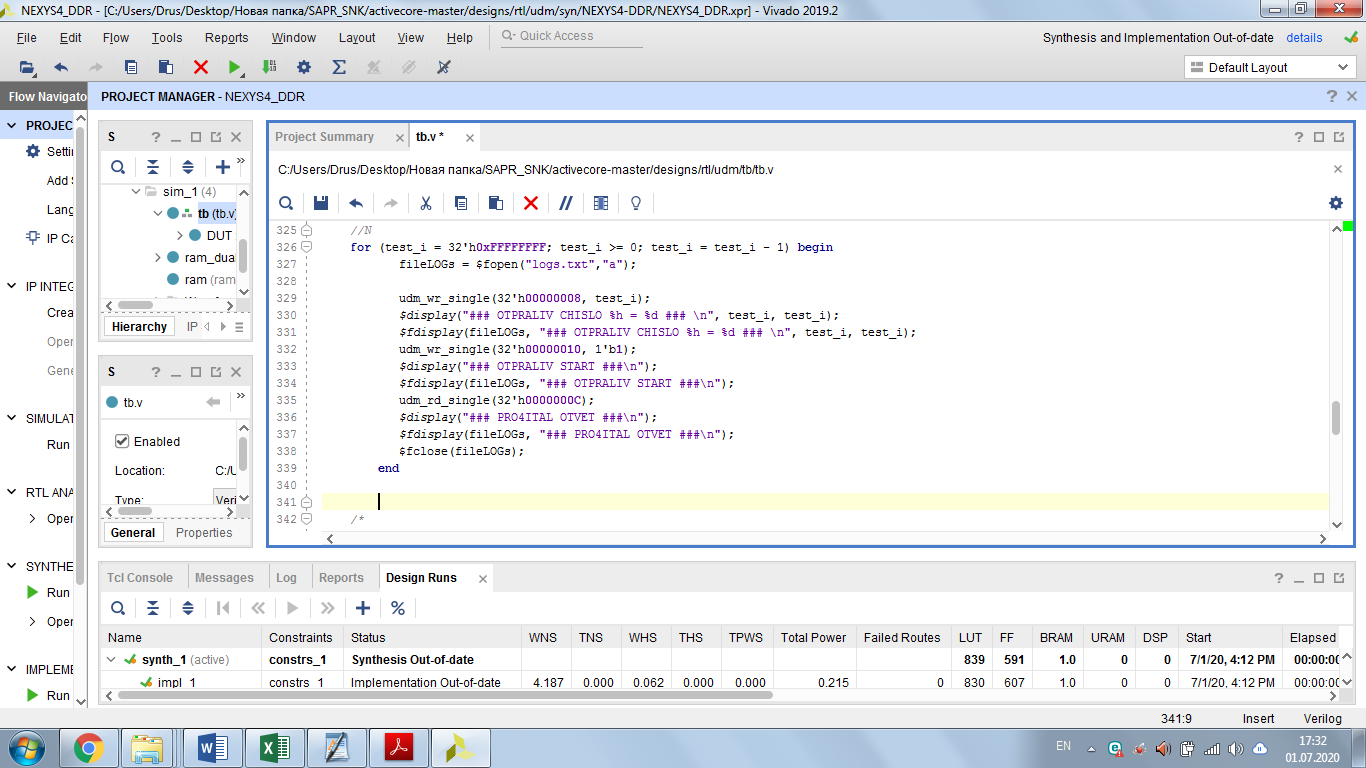
Сначала была произведена попытка тестирования всех целочисленных значений от 0xFFFFFFFF до 0x0. Данное тестирование является нецелесообразным (обоснование представлено в соответствующем подразделе). Из-за этого были выбраны другие подходы к тестированию.

Второе тестирование заключается в отправке на вход модуля максимального допустимого числа, т.е. 0xFFFFFFFF, чисел немного меньше него, числа 0x0, чисел немного больше него и контрольного значения (0xA9).

Входные данные для третьего тестирования были определены с помощью функции $random языка Verilog в ходе симуляции.

## 3.1 Первое тестирование

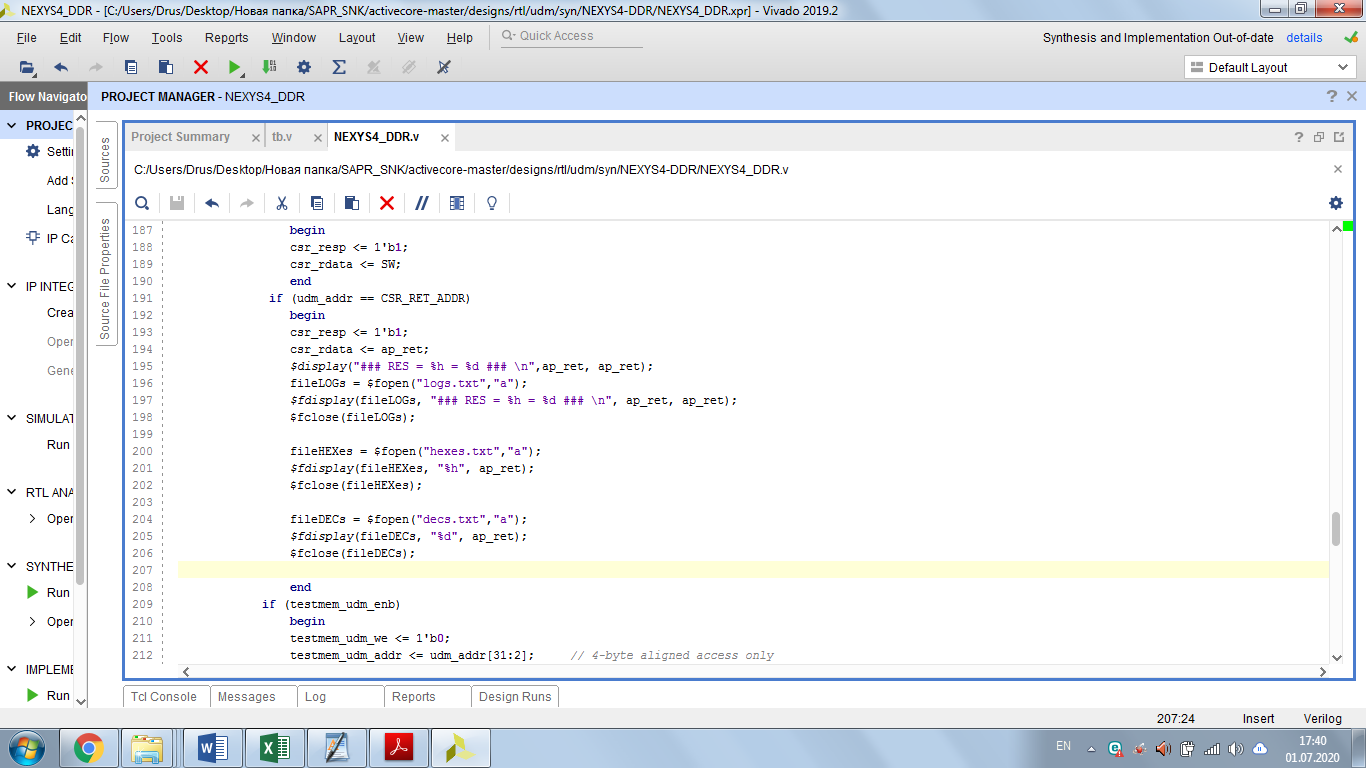
Данное тестирование заключается в переборе всех целочисленных значений от 0xFFFFFFFF до 0x0. Для это в коде теста для симуляции используется код, представленный в листинге 3.1.



Листинг 3.1 – Код для симуляции

Используется цикл for от 0xFFFFFFFF до 0x0. В каждой итерации на вход модуля отправляется значение test\_i, содержащее число, из которого извлекается корень. После чего взводится бит запуска модуля. Напоследок взводится бит для чтения результата из модуля. Все действия протоколируются в файлы, которые указаны в листинге.

При чтении результата также производится его запись в десятичном и шестнадцатеричном представлениях в соответствующие файлы (листинг 3.2) для дальнейшей вставки их в таблицу Excel.



Листинг 3.2 – Запись результатов вычисления модулем

В ходе тестирования было выявлено, что существует ряд недостатков при его реализации:

* скоротечность симуляции напрямую зависит от производительности машины, на которой производится тестирование, что также влияет на следующий недостаток;
* требуется слишком большой промежуток времени для проведения симуляции со всеми значениями (за двое суток не было протестировано и половины значений);
* табличный процессор Excel не позволяет включить в таблицу все целочисленные значения.

Сравнение было произведено для того набора чисел, которые были получены спустя 2 дня симуляции в Excel’е. Частичные результаты тестирования представлены на рисунках 3.1, 3.2.

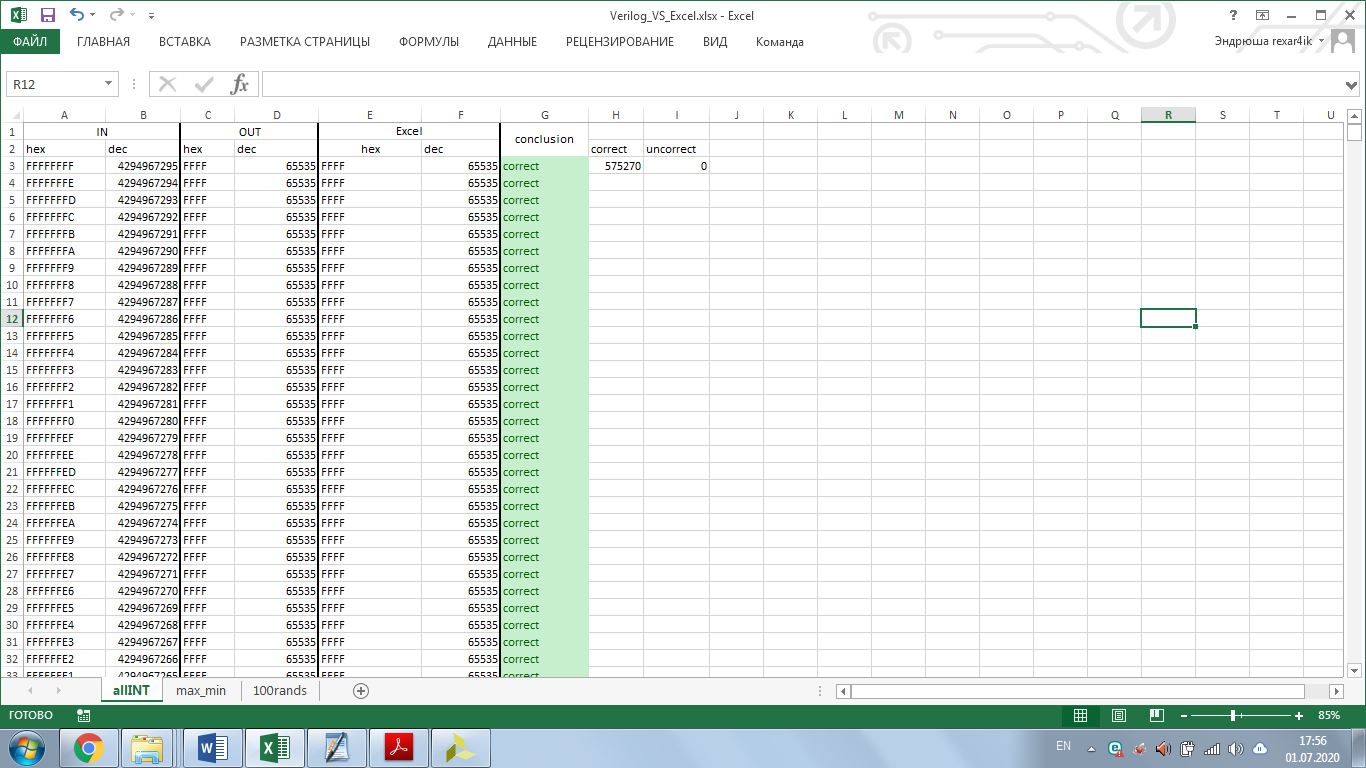


Рисунок 3.1 – Часть результатов тестирования

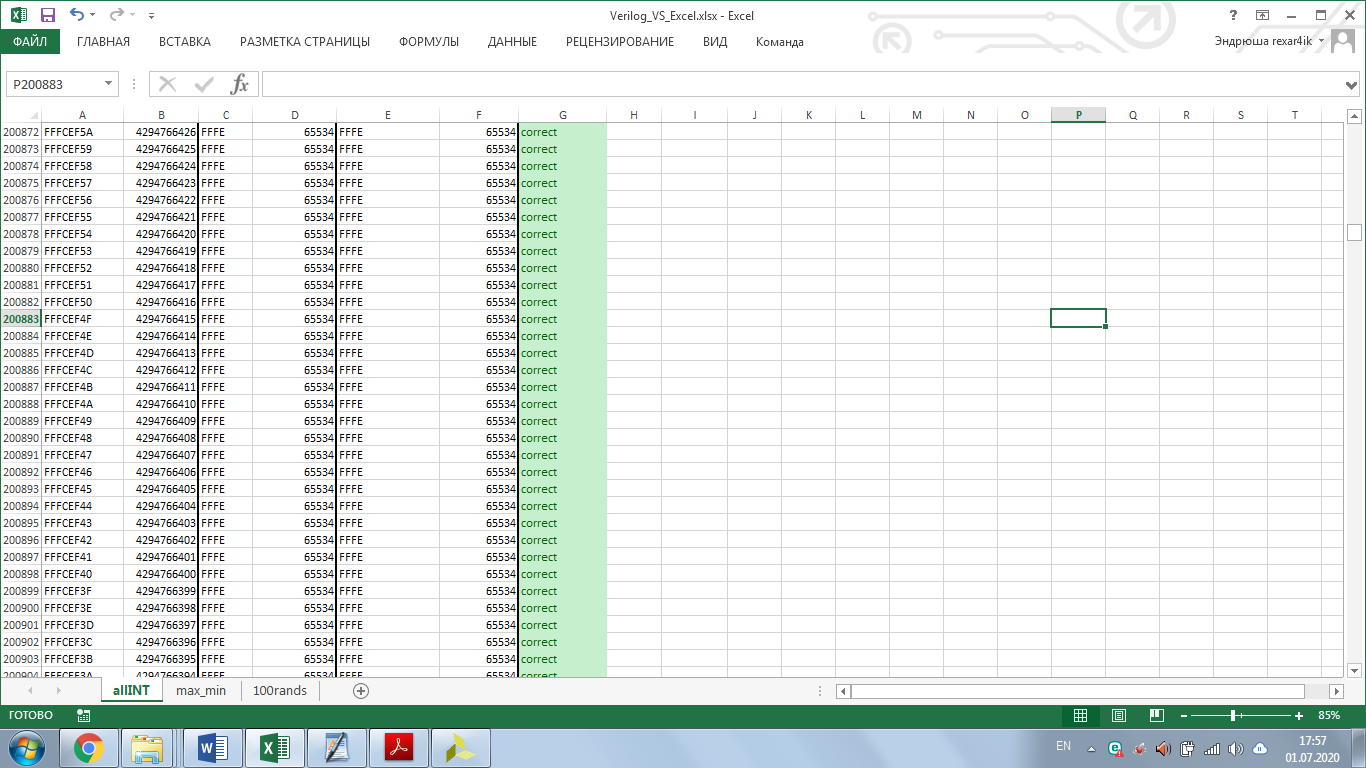


Рисунок 3.2 – Часть результатов тестирования

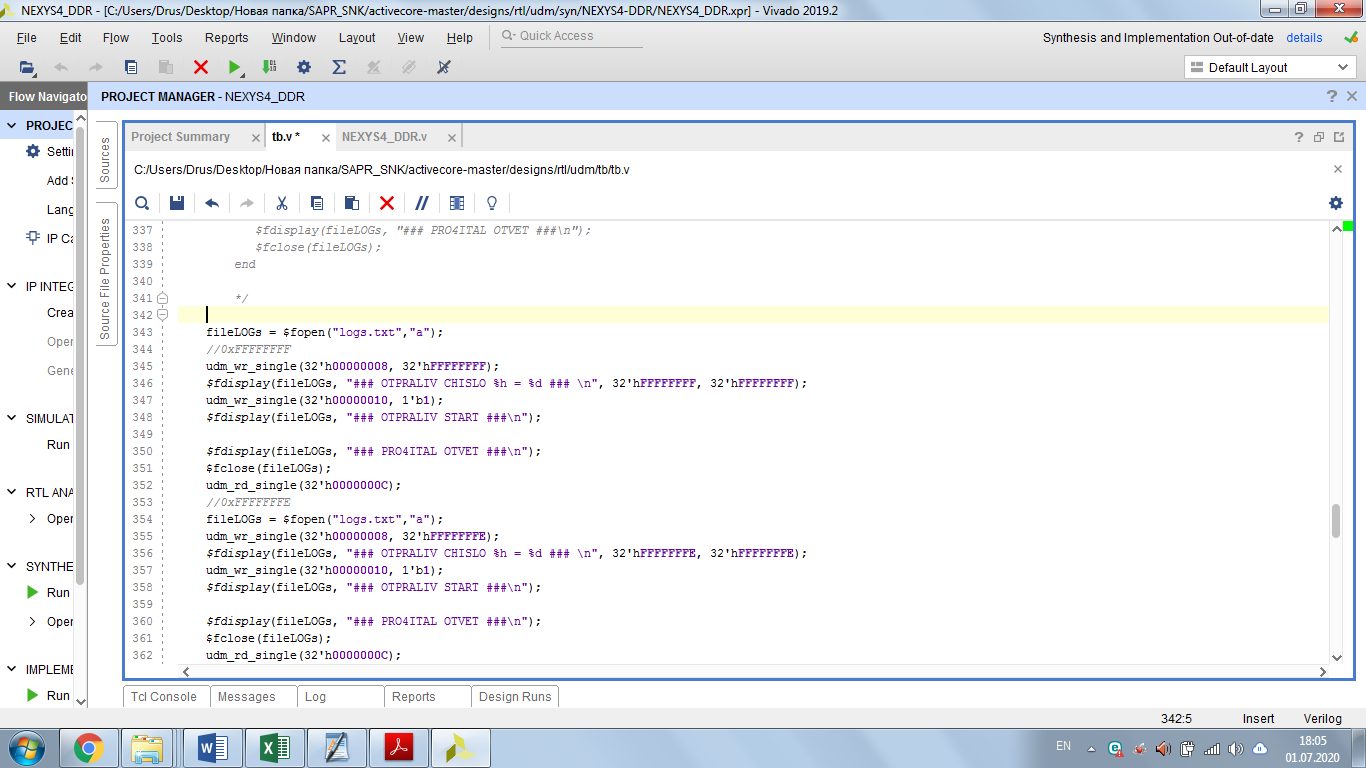
Исходя из ранее перечисленных недостатков тестирования следует произвести другие тестирования.

## 3.2 Второе тестирование

Данное тестирование заключается проверке вычисления квадратного корня модулем следующих чисел:

* 0xFFFFFFFF;
* 0xFFFFFFFE;
* 0xFFFFFFFD;
* 0xFFFFFFFC;
* 0xFFFFFFFB;
* 0x0;
* 0x1;
* 0x2;
* 0x3;
* 0x4;
* 0xA9.

Для это в коде теста для симуляции используется отрезок кода, представленный в листинге 3.3. Данный листинг включает отправку в модуль числа 0xFFFFFFFF, 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). Результаты вычислений также протоколируются, как и в прошлом тесте (листинг 3.2).



Листинг 3.3 – Отрезок кода для симуляции

На рисунке 3.3 представлены результаты вычислений, которые были записаны в файлы в ходе симуляции.

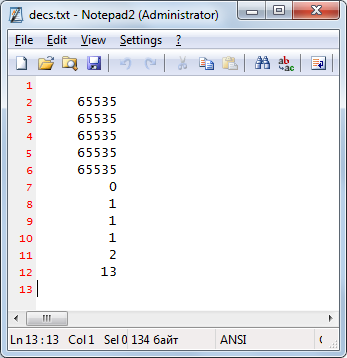
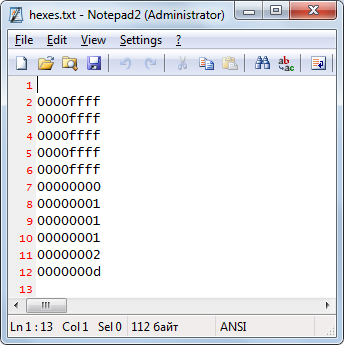


Рисунок 3.3 Результаты выполнения симуляции

Полученные результаты помещены в таблицу Excel и сравнены с вычислением встроенной функции извлечения квадратного корня (рисунок 3.4).

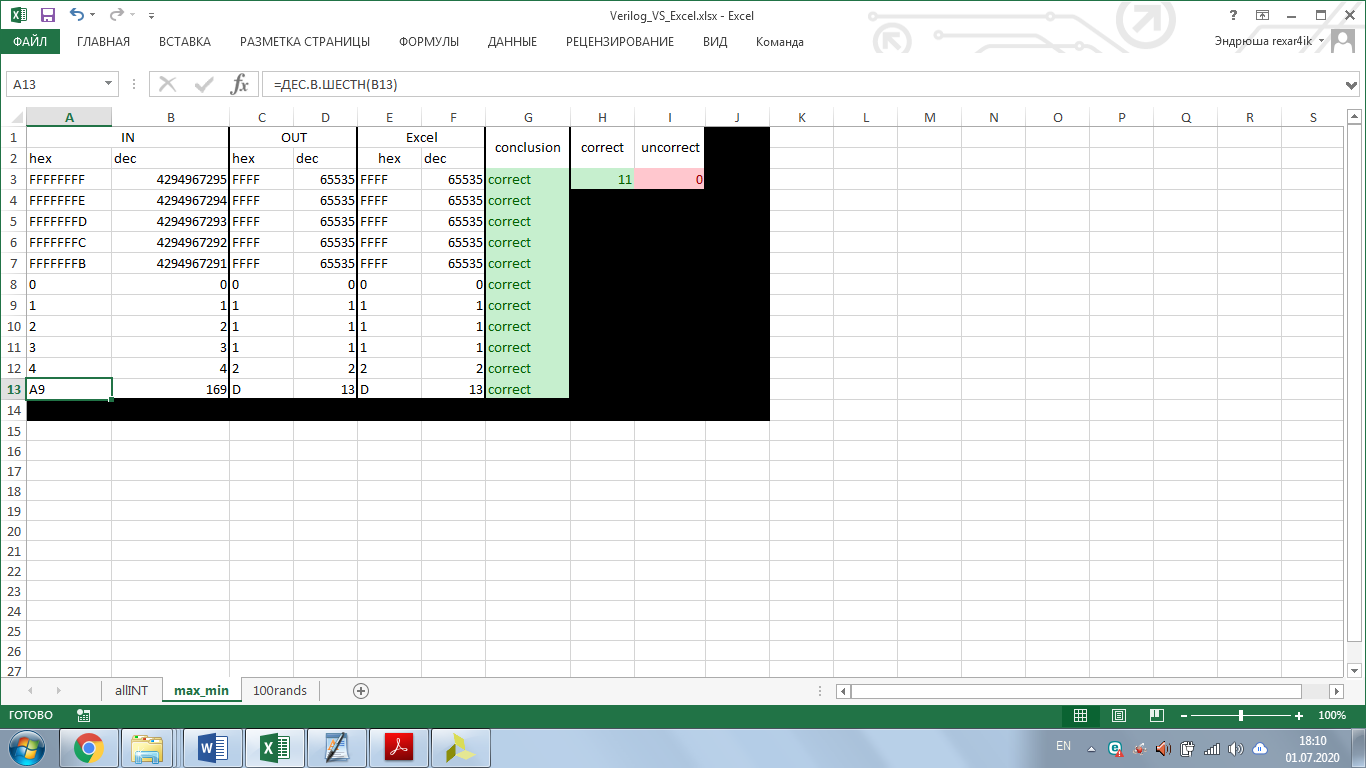
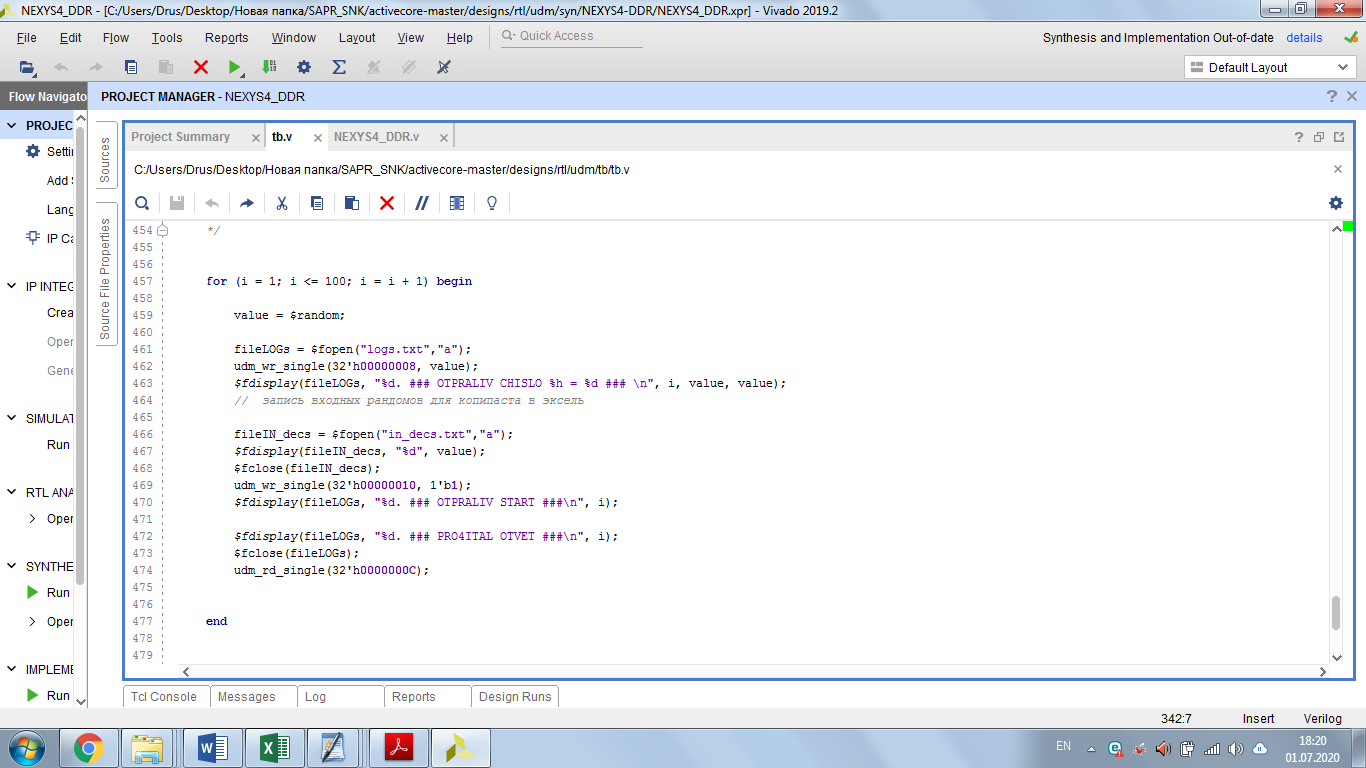


Рисунок 3.4 – Результаты второго тестирования

Данный тест показал, что результаты вычислений модуля сходятся с результатами вычисления встроенной функции извлечения корня Excel.

## 3.3 Третье 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 3.4). На каждой итерации определяется новое входное значение, после чего происходит отправка всех необходимых данных в модуль. Аналогично предыдущим тестам, все промежуточные значения и операции протоколируются в соответствующие файлы.



Листинг 3.4 – Код для симуляции

Значения, полученные функцией $random в ходе симуляции, были записаны в файл in\_decs.txt, после чего скопированы в таблицу Excel для дальнейшего сравнения, результаты которого представлены на рисунках 3.5, 36 и 3.7.

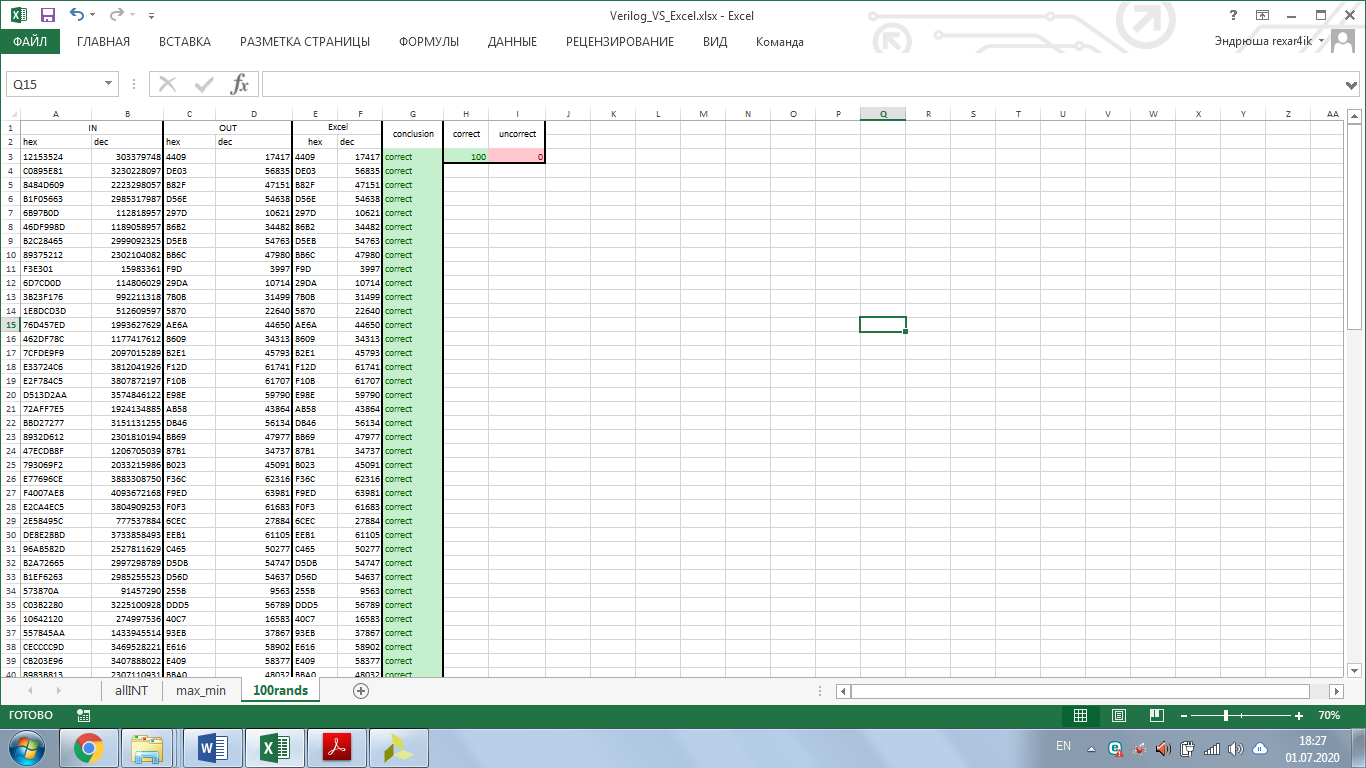


Рисунок 3.5 – Первый фрагмент сравнения

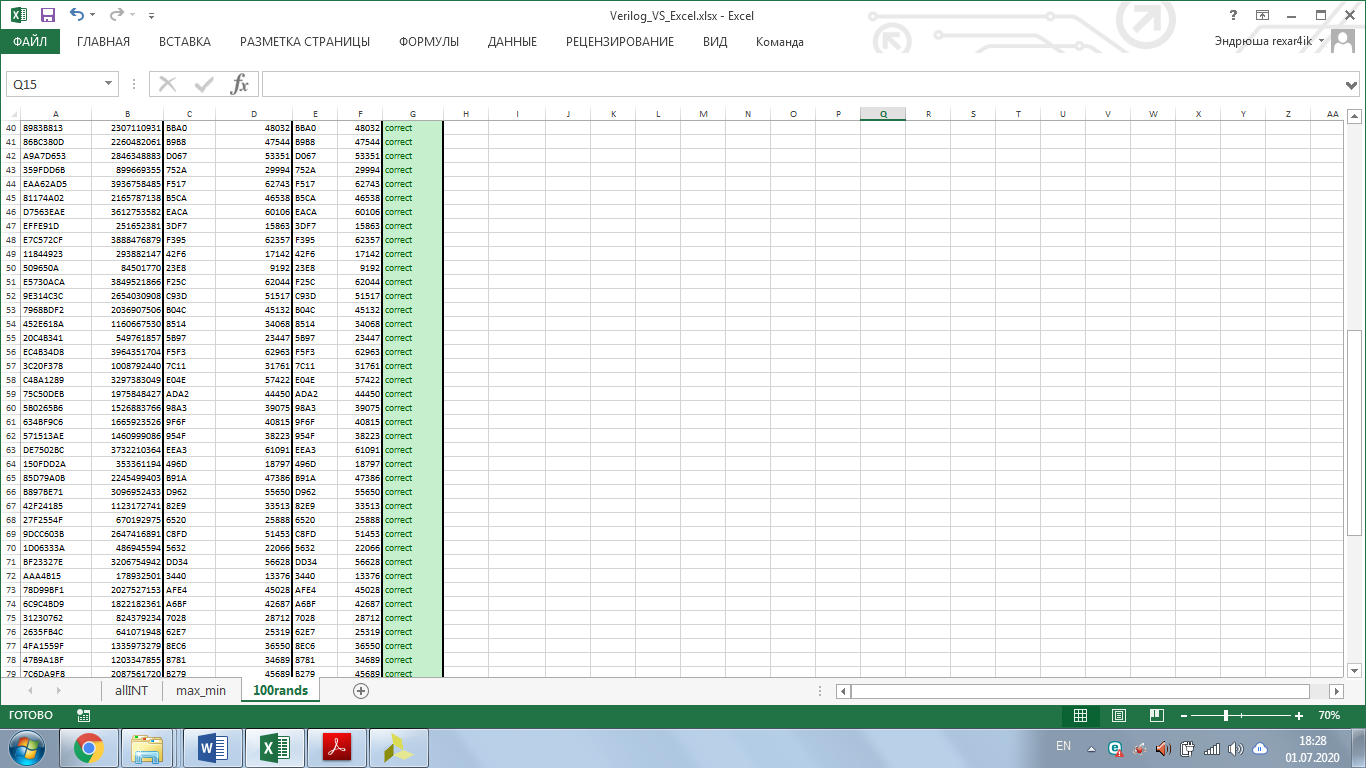


Рисунок 3.6 – Второй фрагмент сравнения

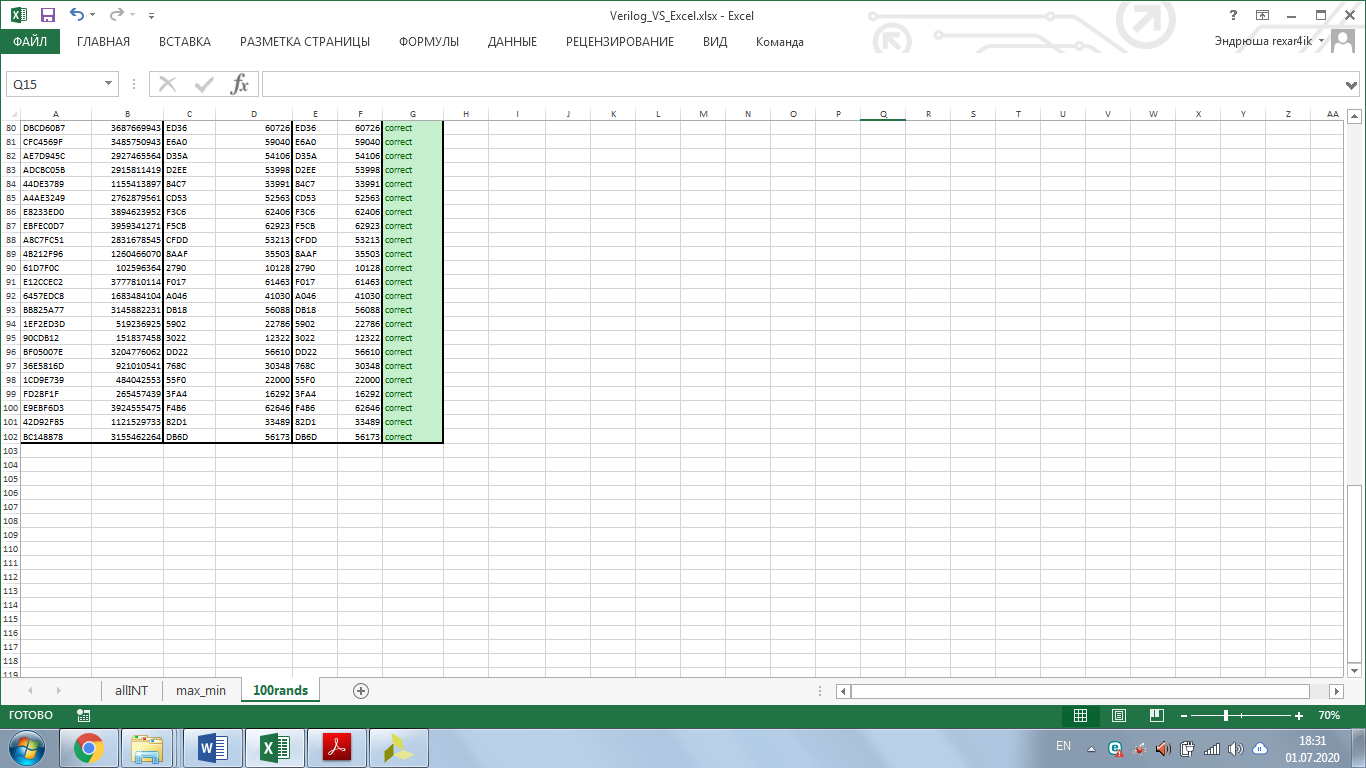


Рисунок 3.7 – Третий фрагмент сравнения

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

**Описание этапа:**

Раздел посвящен добавлению выбранного в прошлом разделе модуля в проект процессора SCR1. Описаны действия и изменения. Использованы спецификации архитектуры RISC-V и процессора SCR1 (источники литературы № 7, 8) и книга о языках описания аппаратуры (источник № 9).

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 3.1.

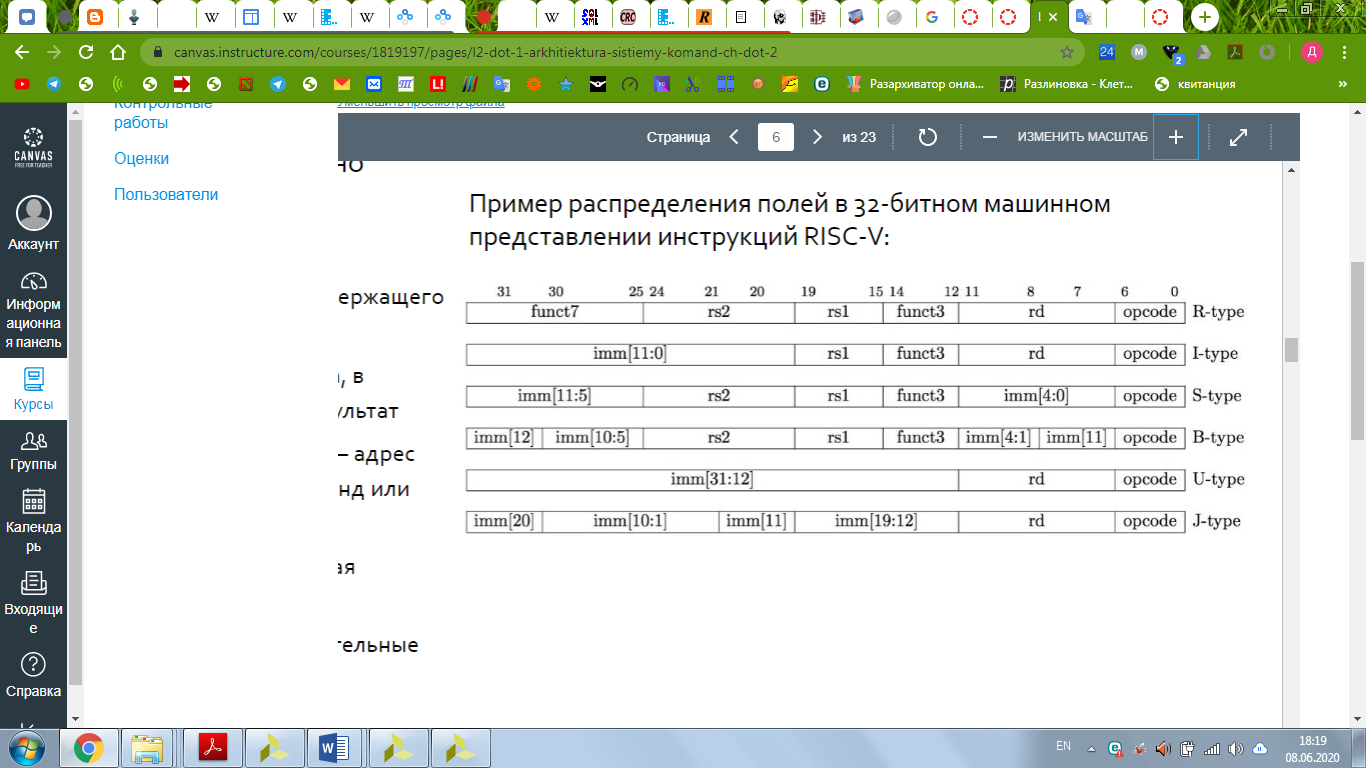


Рисунок 3.1 – Пример распределения полей в 32-битном машинном представлении инструкций

Для команды извлечения квадратного корня использован формат, подобный команде SLLI (рисунок 3.2).

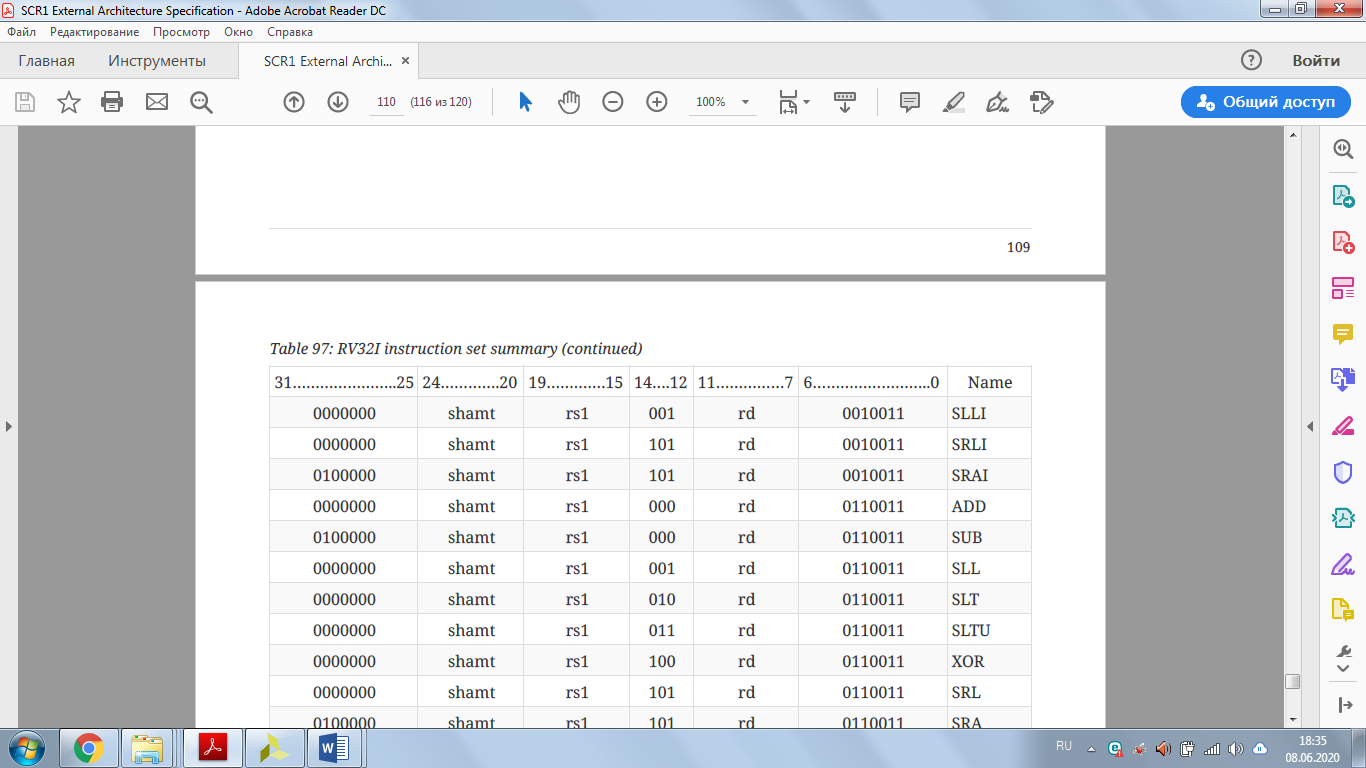


Рисунок 3.2 – Распределение полей в 32-битном машинном представлении инструкции SLLI

Согласно спецификации проекта SCR1[8], тип операции определяет совокупность 3-х полей:

* opcode;
* funct7;
* funct3.

Поле добавляемой инструкции opcode будет содержать код , соответствующий операции с константой, как определено в спецификации [8].

После funct3 будет содержать код , а поле opcode – незанятый набор бит

Для того, чтоб процессор мог обработать добавленную инструкцию, необходимо внести изменения в декодере, который содержится в файле проекта scr1\_pipe\_idu.sv. Необходимо в ветке выбора opcode “SCR1\_OPCODE\_OP\_IMM” в ветке выбора по полю funct3 “001” добавить ветку в оператор “case” [9], соответствующую полю funct7. Изменение представлено на рисунке 3.3.

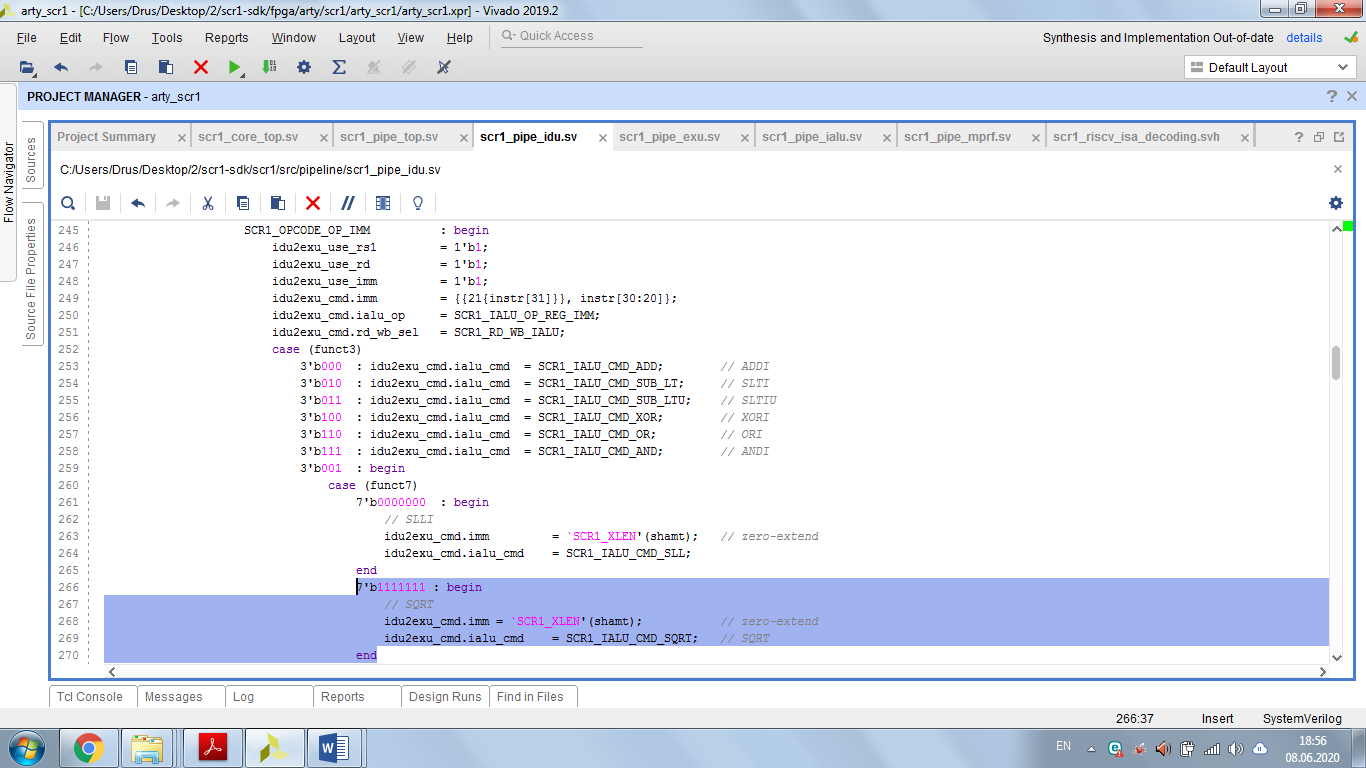


Рисунок 3.3 – Изменения в декодере проекта SCR1

Для добавления самого модуля в вычислительное устройство процессора необходимо произвести изменения в файле scr1\_pipe\_ialu.sv, который содержит логику для произведения арифметических и логических операций. В него необходимо добавить локальные сигналы для модуля (рис. 3.4), код модуля (рис. 3.5) и сигнал записи результата вычисления модуля в выходной порт вычислительного устройства (рис. 3.6).

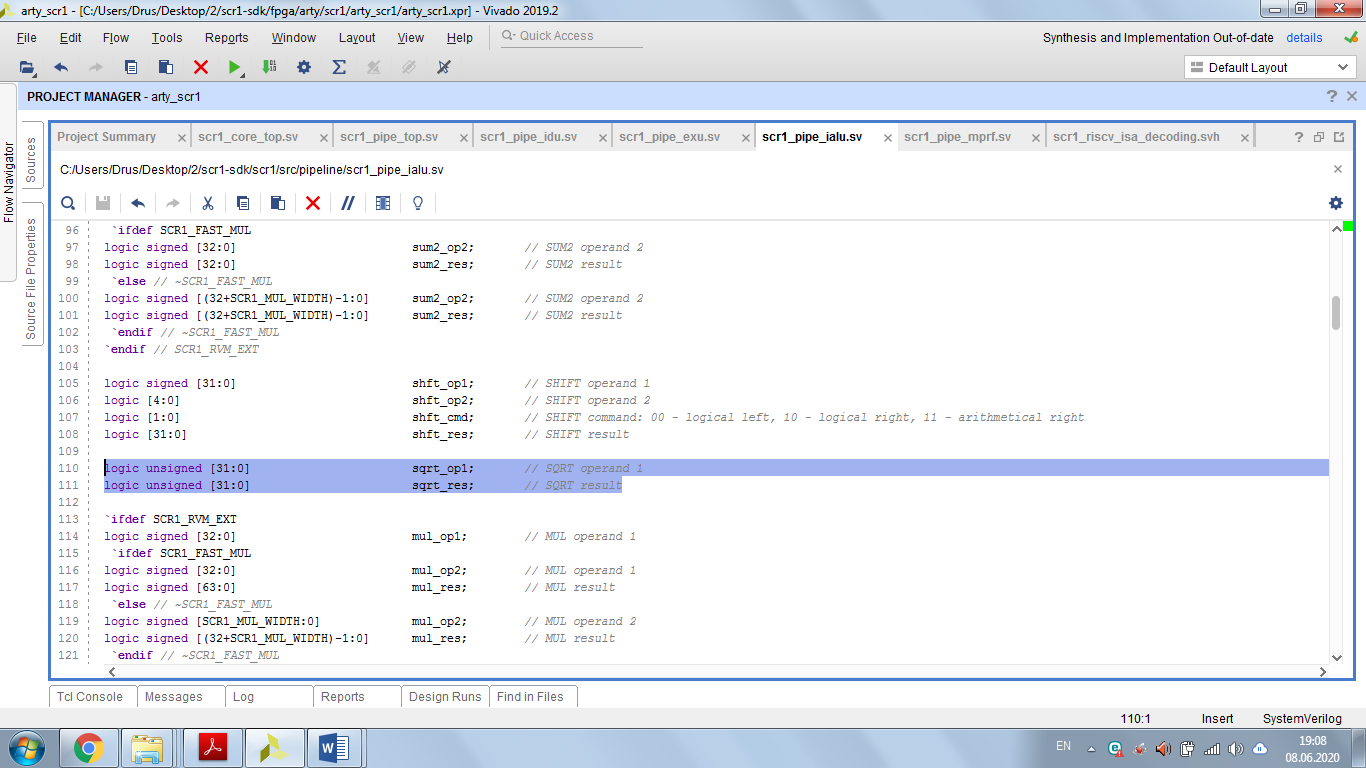


Рисунок 3.4 – Добавление локальных сигналов

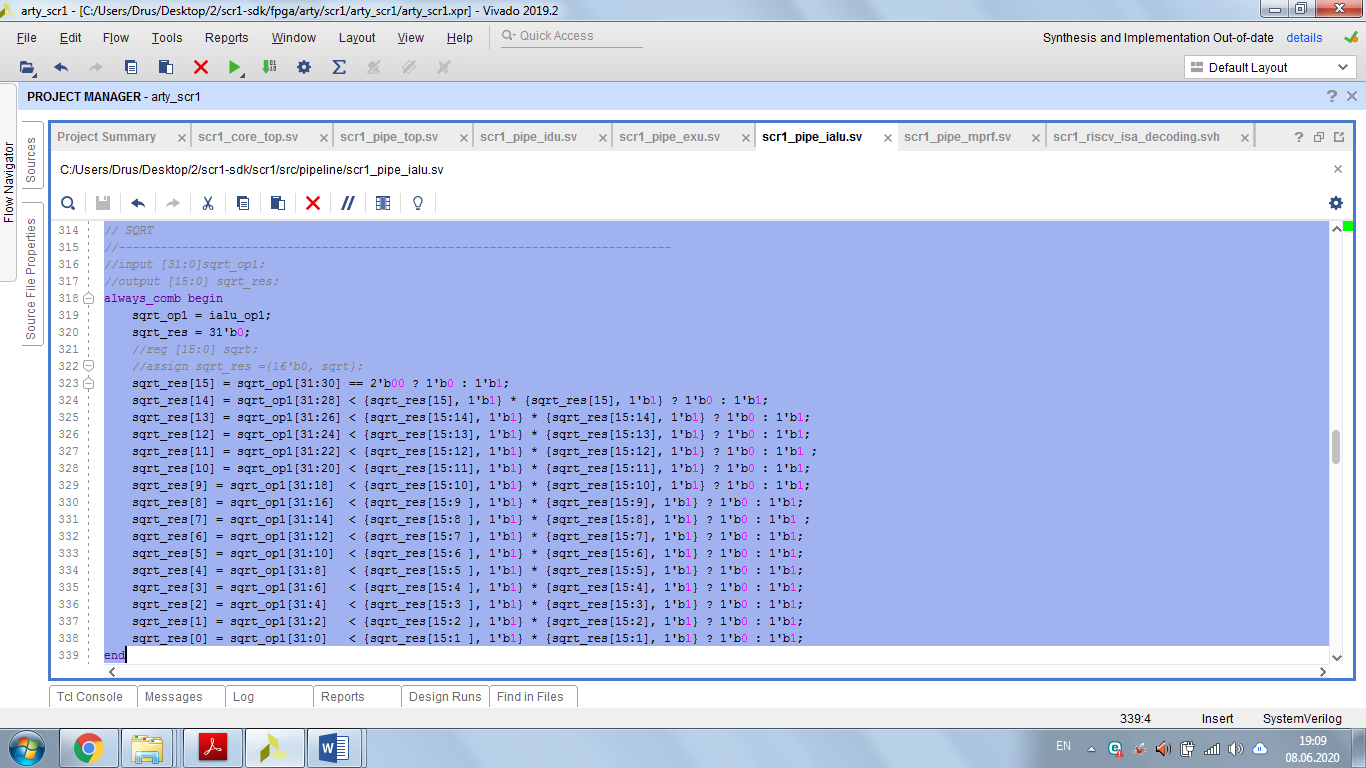
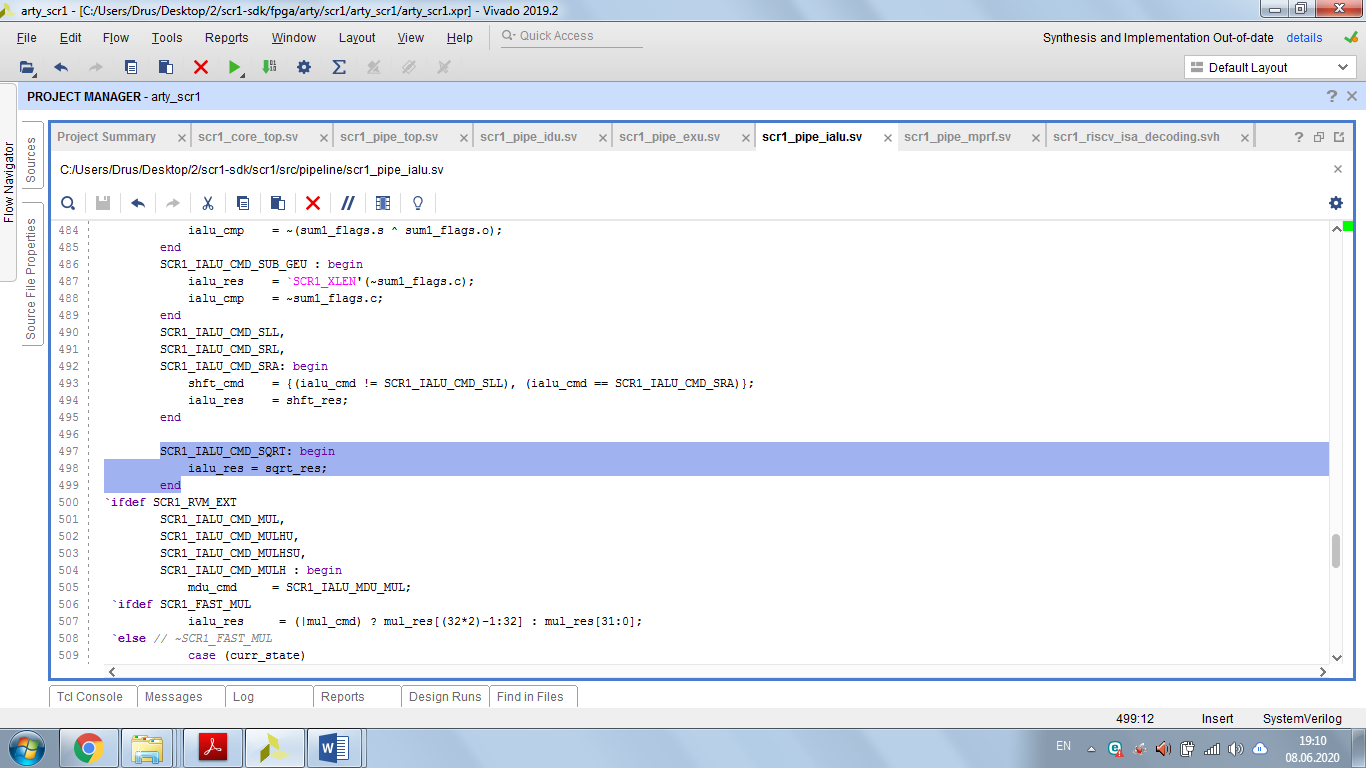


Рисунок 3.5 – Добавление модуля



3.6 – Добавление сигнала записи результата

# 5 ТЕСТИРОВАНИЕ ДОБАВЛЕННОЙ КОМАНДЫ В СОСТАВЕ SCR1

**Описание этапа:**

Данный раздел посвящен проверке добавленной команды в составе проекта SCR1 (источник литературы № 10).

В начале была проверена производительность. Для данного теста использована среда разработки Vivado и проект scr1-sdk.

После этого была проведена проверка работоспособности команды программным кодом. Для тестирования использован симулятор Verilator.

## 5.1 Оценка производительности

Перед проверкой работоспособности команды необходимо произвести оценку производительности проекта с добавленным модулем. Она производится посредством логического синтеза для программируемой логической интегральной схемы Artix-7 и заключается в проверке поддержки частоты по умолчанию для проекта SCR1 без аппаратного расширителя. Чтобы убедиться в том, что не произойдет падения производительности при заданной частоте, необходимо запустить полный flow и проверить параметр WNS.

Положительный WNS, он же SETUP SLACK, говорит о том, что отсутствуют TIMING VIOLATIONS, потому что SETUP SLACK – это не что иное, как разница между “требуемым временем на исполнение комбинаторной логики” () и “временем прибытия управляющего сигнала” (). Соответственно, если WNS (SETUP SLACK) является положительным, то это означает, что самый длинный путь проходит – отсутствуют TIMING VIOLATIONS (). Если , значит, есть TIMING VIOLATIONS.

На рисунке 5.1 продемонстрирован результат запуска имплементации.

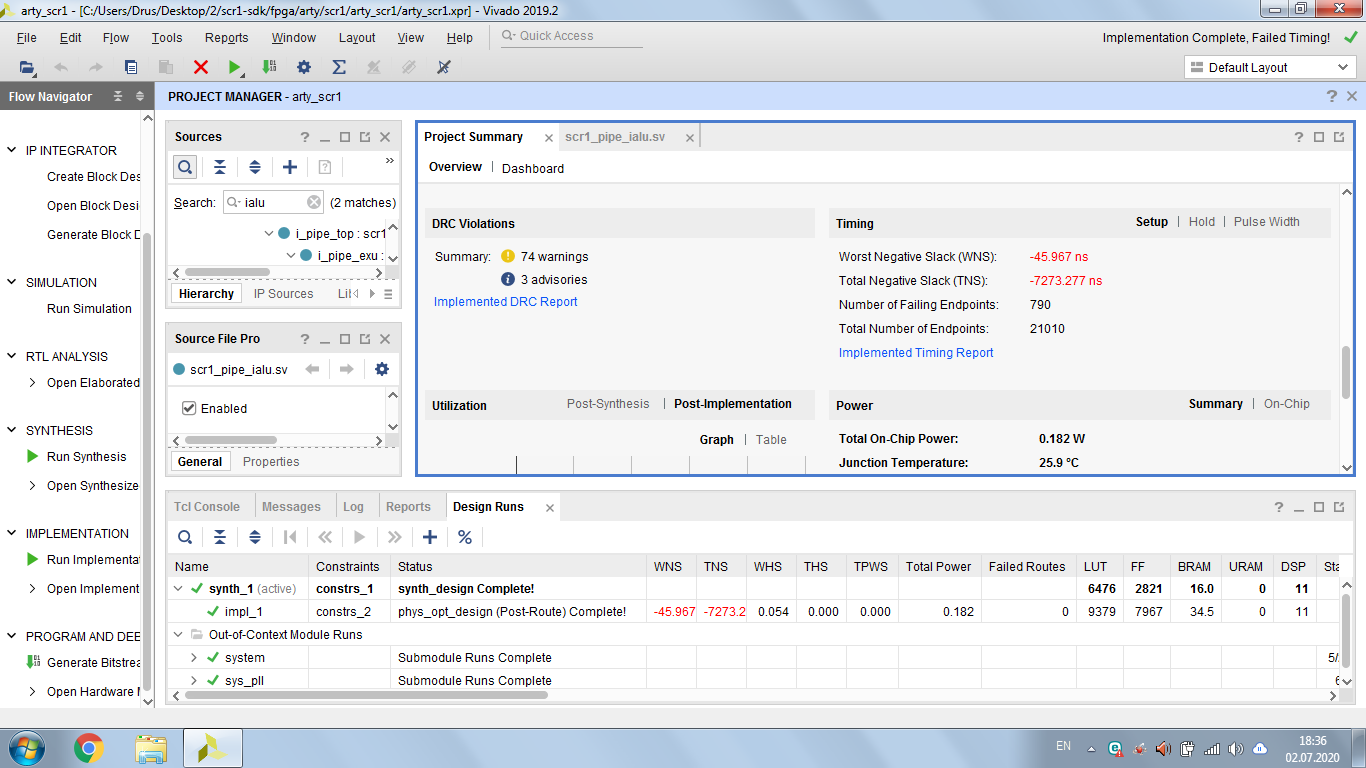


Рисунок 5.1 – Временные показатели производительности

Исходя из полученных результатов временных показателей можно сделать вывод, что присутствуют TIMING VIOLATIONS, так как WNS является отрицательным. Следовательно, добавленный модуль содержит слишком длинный путь комбинаторной логики для частоты по умолчанию для проекта SCR1 без аппаратного расширителя, равной 25 Mhz.

Одним из путей преодоления данной проблемы является понижение частоты (увеличение периода). Но при этом упадет общая производительность устройства. В связи с этим необходимо разделить выполнение процедуры извлечения квадратного корня на 4 стадии. Разделение реализовывается путём добавления управляющего устройства, которое будет “включать” на каждом такте одну из частей модуля. При этом необходимо добавить оператор “case” непосредственно в модуль, ветками которого будут состояния, генерируемые управляющим устройством. Также необходимо добавить сигнал, сигнализирующий о готовности результата выполнения модуля. Листинг новой реализации модуля продемонстрирован в приложении 4.

На рисунке 5.2 продемонстрирован результат запуска имплементации с новой реализацией вычисления квадратного корня.

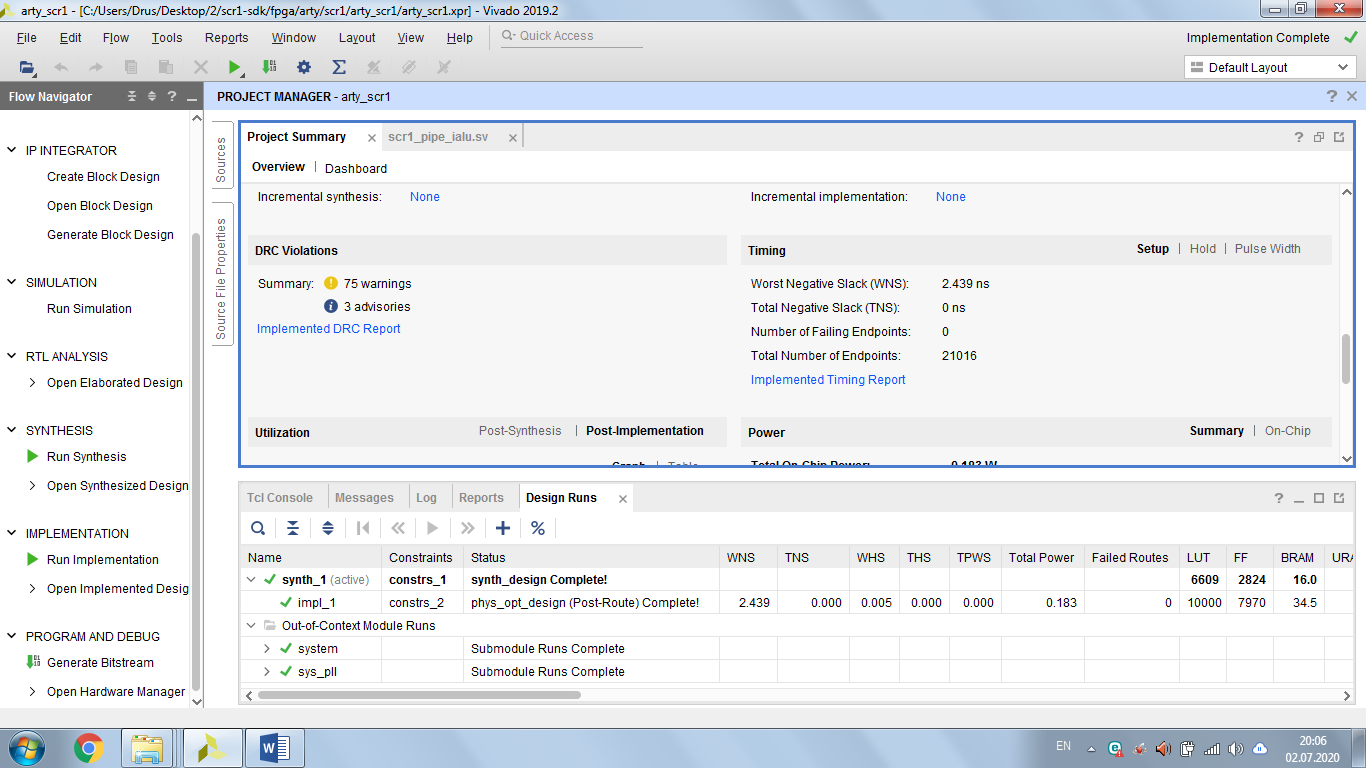


Рисунок 5.2 Временные показатели производительности

Полученные результаты свидетельствуют о том, что отсутствуют TIMING VIOLATIONS, так как WNS является положительным. Соответственно, система поддерживает частоту 25 MHz, которая является частотой по умолчанию для ядра SCR1 без аппаратного расширителя.

## 5.2 Unit-тест в составе SCR1

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

* 0xFFFFFFFF – максимально допустимое значение системы;
* 0xFFFFFFFE;
* 0xFFFFFFFD;
* 0x0;
* 0x1;
* 0x2;
* 0x3;
* 0x4;
* 0xA9.

Согласно спецификации SCR1 [10] проект адаптирован на проверку – содержит реализованные конструкции тестирования инструкций. В данном случае необходимо использовать конструкцию-макрос   
TEST\_IMM\_OP\_SQRT (testnum, inst, result, val1), операндами которой являются:

* testnum – номер теста;
* instr – машинное представление тестируемой инструкции;
* result – ожидаемый результат выполнения инструкции;
* val1 – входное значение.

Также макросами-конструкциями TEST\_SQRT\_DEST\_BYPASS и TEST\_SQRT\_SCR1\_BYPASS протестировано поведение команды в пайплайне. Их операнды аналогичны предыдущему макросу за исключением операнда nop\_cycles, которым задается количество конструкций “nop”.

Код теста представлен в приложении 5. В начале теста подключены библиотеки, содержащие нужные для тестирования макросы. Также реализован макрос, который генерирует машинное представление добавленной команды INST\_SQRT(rd, rs1).

Был выполнен запуск симуляции и получен успешный результат выполнения теста (рис. 5.3).

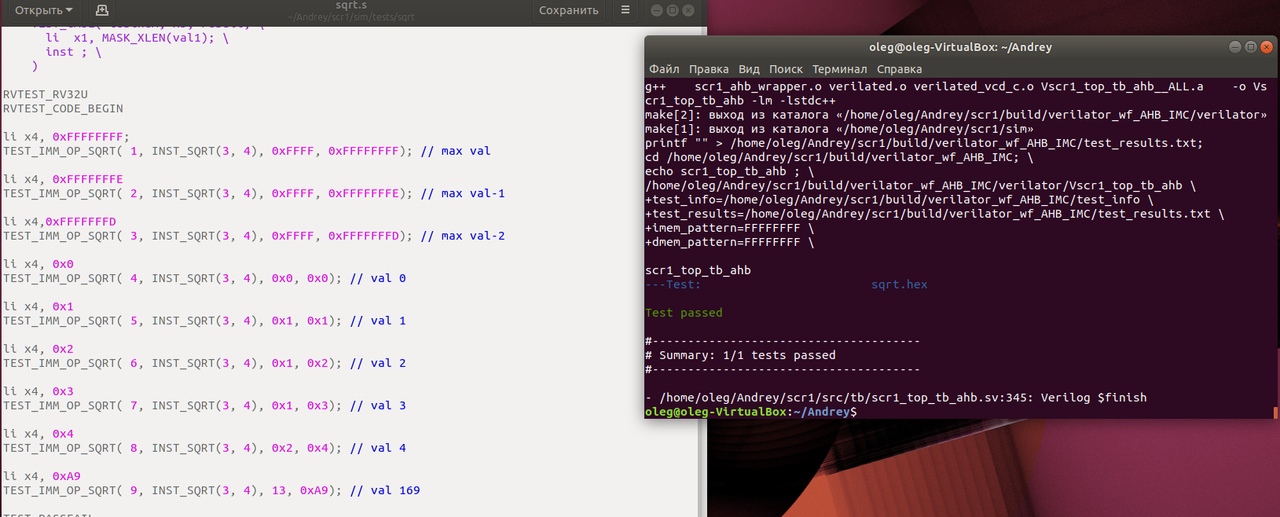


Рисунок 5.3 – Результат тестирования в симуляторе Verilator

# 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 квадратного корня из целого числа, приведены примеры расчета и определены оптимальные варианты для реализации аппаратных модулей.

После выбора алгоритмов по ним были реализованы аппаратные модули по двум методам описания аппаратуры: синтез высокого уровня и разработка на языке описания аппаратуры Verilog. Хотя Синтез высокого уровня значительно упрощает разработку, он не дает таких хороших результатов в производительности и оптимальном использовании микросхемы, как непосредственная разработка модулей на языке Verilog.

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

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Прямостанов, С. М. Методы извлечения квадратного корня / С. М. Прямостанов, Л. В. Лысогорова. — Текст : непосредственный // Юный ученый. — 2017. — № 2.2 (11.2). — С. 76-77.

2. Encyclopaedia of the History of Science, Technology, and Medicine in Non-Western Cultures /Helaine Selin. — Amsterdam: Springer Netherlands, 2008. — 2415с.

3. FPGAch [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: http://fpgach.blogspot.com/, свободный

4. XILINX [Электронный ресурс] /. — Электрон. текстовые дан. — Режим доступа: https://www.xilinx.com/, свободный

5. Страуструп Бьерн /Бьярне Страуструп. — Москва: Вильямс, 2016. — 1328с.

6. Соловьев, В.В. Основы языка проектирования цифровой аппаратуры Verilog /В.В. Соловьев. — Москва: Горячая Линия - Телеком, 2014. — 208с.

7. The RISC-V Instruction Set Manual /Andrew Waterman, Yunsup Lee, Rimas Avizienis, David Patterson and etc.. — Berkeley: 2016. — 79с.

8. SCR1 External Architecture Specification /. — 2019. — 114с.

9. Тарасов, И.Е. ПЛИС Xilinx. Языки описания аппаратуры VHDL и Verilog, САПР, приемы проектирования /И.Е. Тарасов. — Москва: Горячая Линия - Телеком, 2019. — 538с.

10. SCR1 User Manual /. — 2019. — 16с.

# ПРИЛОЖЕНИЕ 1

**ЛИСТИНГ КОДА МОДУЛЯ ПО ФОРМУЛЕ ГЕРОНА НА ЯЗЫКЕ ПРОГРАММИРОВАНИЯ СИ**

#include <math.h>

unsigned int sqrtINT(int n)

{

double xn, xn\_1 = 1;

double result;

do

{

xn = xn\_1;

xn\_1 = (xn + n / xn) / 2;

}

while ( fabs(xn - xn\_1) > 0.000000001 );

return (unsigned int)xn\_1;

}

# ПРИЛОЖЕНИЕ 2

**ЛИСТИНГ КОДА МОДУЛЯ ПО ФОРМУЛЕ ГЕРОНА НА VERILOG**

module sqrtINT (

ap\_clk,

ap\_rst,

ap\_start,

ap\_done,

ap\_idle,

ap\_ready,

n,

ap\_return

);

parameter ap\_ST\_fsm\_state1 = 43'd1;

parameter ap\_ST\_fsm\_state2 = 43'd2;

parameter ap\_ST\_fsm\_state3 = 43'd4;

parameter ap\_ST\_fsm\_state4 = 43'd8;

parameter ap\_ST\_fsm\_state5 = 43'd16;

parameter ap\_ST\_fsm\_state6 = 43'd32;

parameter ap\_ST\_fsm\_state7 = 43'd64;

parameter ap\_ST\_fsm\_state8 = 43'd128;

parameter ap\_ST\_fsm\_state9 = 43'd256;

parameter ap\_ST\_fsm\_state10 = 43'd512;

parameter ap\_ST\_fsm\_state11 = 43'd1024;

parameter ap\_ST\_fsm\_state12 = 43'd2048;

parameter ap\_ST\_fsm\_state13 = 43'd4096;

parameter ap\_ST\_fsm\_state14 = 43'd8192;

parameter ap\_ST\_fsm\_state15 = 43'd16384;

parameter ap\_ST\_fsm\_state16 = 43'd32768;

parameter ap\_ST\_fsm\_state17 = 43'd65536;

parameter ap\_ST\_fsm\_state18 = 43'd131072;

parameter ap\_ST\_fsm\_state19 = 43'd262144;

parameter ap\_ST\_fsm\_state20 = 43'd524288;

parameter ap\_ST\_fsm\_state21 = 43'd1048576;

parameter ap\_ST\_fsm\_state22 = 43'd2097152;

parameter ap\_ST\_fsm\_state23 = 43'd4194304;

parameter ap\_ST\_fsm\_state24 = 43'd8388608;

parameter ap\_ST\_fsm\_state25 = 43'd16777216;

parameter ap\_ST\_fsm\_state26 = 43'd33554432;

parameter ap\_ST\_fsm\_state27 = 43'd67108864;

parameter ap\_ST\_fsm\_state28 = 43'd134217728;

parameter ap\_ST\_fsm\_state29 = 43'd268435456;

parameter ap\_ST\_fsm\_state30 = 43'd536870912;

parameter ap\_ST\_fsm\_state31 = 43'd1073741824;

parameter ap\_ST\_fsm\_state32 = 43'd2147483648;

parameter ap\_ST\_fsm\_state33 = 43'd4294967296;

parameter ap\_ST\_fsm\_state34 = 43'd8589934592;

parameter ap\_ST\_fsm\_state35 = 43'd17179869184;

parameter ap\_ST\_fsm\_state36 = 43'd34359738368;

parameter ap\_ST\_fsm\_state37 = 43'd68719476736;

parameter ap\_ST\_fsm\_state38 = 43'd137438953472;

parameter ap\_ST\_fsm\_state39 = 43'd274877906944;

parameter ap\_ST\_fsm\_state40 = 43'd549755813888;

parameter ap\_ST\_fsm\_state41 = 43'd1099511627776;

parameter ap\_ST\_fsm\_state42 = 43'd2199023255552;

parameter ap\_ST\_fsm\_state43 = 43'd4398046511104;

input ap\_clk;

input ap\_rst;

input ap\_start;

output ap\_done;

output ap\_idle;

output ap\_ready;

input [31:0] n;

output [31:0] ap\_return;

reg ap\_done;

reg ap\_idle;

reg ap\_ready;

(\* fsm\_encoding = "none" \*) reg [42:0] ap\_CS\_fsm;

wire ap\_CS\_fsm\_state1;

wire [63:0] grp\_fu\_70\_p2;

reg [63:0] reg\_94;

wire ap\_CS\_fsm\_state31;

wire ap\_CS\_fsm\_state41;

wire [63:0] grp\_fu\_90\_p1;

reg [63:0] tmp\_reg\_282;

wire ap\_CS\_fsm\_state4;

wire [63:0] grp\_fu\_80\_p2;

reg [63:0] tmp\_1\_reg\_287;

wire ap\_CS\_fsm\_state26;

wire [63:0] grp\_fu\_75\_p2;

reg [63:0] xn\_1\_reg\_292;

wire ap\_CS\_fsm\_state36;

wire ap\_CS\_fsm\_state42;

wire [0:0] icmp\_ln12\_fu\_134\_p2;

reg [0:0] icmp\_ln12\_reg\_304;

wire [0:0] icmp\_ln12\_1\_fu\_140\_p2;

reg [0:0] icmp\_ln12\_1\_reg\_309;

reg [63:0] xn\_reg\_58;

wire ap\_CS\_fsm\_state43;

wire [0:0] and\_ln12\_fu\_150\_p2;

reg [63:0] grp\_fu\_70\_p1;

wire ap\_CS\_fsm\_state27;

wire ap\_CS\_fsm\_state37;

wire ap\_CS\_fsm\_state32;

wire ap\_CS\_fsm\_state5;

wire [63:0] grp\_fu\_85\_p0;

wire [63:0] p\_Val2\_s\_fu\_99\_p1;

wire [62:0] trunc\_ln368\_fu\_103\_p1;

wire [63:0] p\_Result\_s\_fu\_111\_p3;

wire [10:0] tmp\_5\_fu\_124\_p4;

wire [51:0] trunc\_ln12\_fu\_107\_p1;

wire [0:0] or\_ln12\_fu\_146\_p2;

wire [0:0] grp\_fu\_85\_p2;

wire [63:0] p\_Val2\_1\_fu\_156\_p1;

wire [51:0] tmp\_V\_1\_fu\_169\_p1;

wire [53:0] mantissa\_V\_fu\_173\_p4;

wire [10:0] tmp\_V\_fu\_159\_p4;

wire [11:0] zext\_ln502\_fu\_187\_p1;

wire [11:0] add\_ln502\_fu\_191\_p2;

wire [10:0] sub\_ln1311\_fu\_205\_p2;

wire [0:0] isNeg\_fu\_197\_p3;

wire signed [11:0] sext\_ln1311\_fu\_211\_p1;

wire [11:0] ush\_fu\_215\_p3;

wire signed [31:0] sext\_ln1311\_1\_fu\_223\_p1;

wire [53:0] zext\_ln1285\_fu\_231\_p1;

wire [136:0] zext\_ln682\_fu\_183\_p1;

wire [136:0] zext\_ln1287\_fu\_227\_p1;

wire [53:0] r\_V\_fu\_235\_p2;

wire [0:0] tmp\_4\_fu\_247\_p3;

wire [136:0] r\_V\_1\_fu\_241\_p2;

wire [31:0] zext\_ln662\_fu\_255\_p1;

wire [31:0] tmp\_8\_fu\_259\_p4;

reg [1:0] grp\_fu\_70\_opcode;

reg [42:0] ap\_NS\_fsm;

// power-on initialization

initial begin

#0 ap\_CS\_fsm = 43'd1;

end

sqrtINT\_dadddsub\_bkb #(

.ID( 1 ),

.NUM\_STAGE( 5 ),

.din0\_WIDTH( 64 ),

.din1\_WIDTH( 64 ),

.dout\_WIDTH( 64 ))

sqrtINT\_dadddsub\_bkb\_U1(

.clk(ap\_clk),

.reset(ap\_rst),

.din0(xn\_reg\_58),

.din1(grp\_fu\_70\_p1),

.opcode(grp\_fu\_70\_opcode),

.ce(1'b1),

.dout(grp\_fu\_70\_p2)

);

sqrtINT\_dmul\_64nscud #(

.ID( 1 ),

.NUM\_STAGE( 5 ),

.din0\_WIDTH( 64 ),

.din1\_WIDTH( 64 ),

.dout\_WIDTH( 64 ))

sqrtINT\_dmul\_64nscud\_U2(

.clk(ap\_clk),

.reset(ap\_rst),

.din0(reg\_94),

.din1(64'd4602678819172646912),

.ce(1'b1),

.dout(grp\_fu\_75\_p2)

);

sqrtINT\_ddiv\_64nsdEe #(

.ID( 1 ),

.NUM\_STAGE( 22 ),

.din0\_WIDTH( 64 ),

.din1\_WIDTH( 64 ),

.dout\_WIDTH( 64 ))

sqrtINT\_ddiv\_64nsdEe\_U3(

.clk(ap\_clk),

.reset(ap\_rst),

.din0(tmp\_reg\_282),

.din1(xn\_reg\_58),

.ce(1'b1),

.dout(grp\_fu\_80\_p2)

);

sqrtINT\_dcmp\_64nseOg #(

.ID( 1 ),

.NUM\_STAGE( 2 ),

.din0\_WIDTH( 64 ),

.din1\_WIDTH( 64 ),

.dout\_WIDTH( 1 ))

sqrtINT\_dcmp\_64nseOg\_U4(

.clk(ap\_clk),

.reset(ap\_rst),

.din0(grp\_fu\_85\_p0),

.din1(64'd4472406533629990549),

.ce(1'b1),

.opcode(5'd2),

.dout(grp\_fu\_85\_p2)

);

sqrtINT\_sitodp\_32fYi #(

.ID( 1 ),

.NUM\_STAGE( 4 ),

.din0\_WIDTH( 32 ),

.dout\_WIDTH( 64 ))

sqrtINT\_sitodp\_32fYi\_U5(

.clk(ap\_clk),

.reset(ap\_rst),

.din0(n),

.ce(1'b1),

.dout(grp\_fu\_90\_p1)

);

always @ (posedge ap\_clk) begin

if (ap\_rst == 1'b1) begin

ap\_CS\_fsm <= ap\_ST\_fsm\_state1;

end else begin

ap\_CS\_fsm <= ap\_NS\_fsm;

end

end

always @ (posedge ap\_clk) begin

if (((1'b1 == ap\_CS\_fsm\_state43) & (1'd1 == and\_ln12\_fu\_150\_p2))) begin

xn\_reg\_58 <= xn\_1\_reg\_292;

end else if ((1'b1 == ap\_CS\_fsm\_state4)) begin

xn\_reg\_58 <= 64'd4607182418800017408;

end

end

always @ (posedge ap\_clk) begin

if ((1'b1 == ap\_CS\_fsm\_state42)) begin

icmp\_ln12\_1\_reg\_309 <= icmp\_ln12\_1\_fu\_140\_p2;

icmp\_ln12\_reg\_304 <= icmp\_ln12\_fu\_134\_p2;

end

end

always @ (posedge ap\_clk) begin

if (((1'b1 == ap\_CS\_fsm\_state41) | (1'b1 == ap\_CS\_fsm\_state31))) begin

reg\_94 <= grp\_fu\_70\_p2;

end

end

always @ (posedge ap\_clk) begin

if ((1'b1 == ap\_CS\_fsm\_state26)) begin

tmp\_1\_reg\_287 <= grp\_fu\_80\_p2;

end

end

always @ (posedge ap\_clk) begin

if ((1'b1 == ap\_CS\_fsm\_state4)) begin

tmp\_reg\_282 <= grp\_fu\_90\_p1;

end

end

always @ (posedge ap\_clk) begin

if ((1'b1 == ap\_CS\_fsm\_state36)) begin

xn\_1\_reg\_292 <= grp\_fu\_75\_p2;

end

end

always @ (\*) begin

if (((1'b1 == ap\_CS\_fsm\_state43) & (1'd0 == and\_ln12\_fu\_150\_p2))) begin

ap\_done = 1'b1;

end else begin

ap\_done = 1'b0;

end

end

always @ (\*) begin

if (((ap\_start == 1'b0) & (1'b1 == ap\_CS\_fsm\_state1))) begin

ap\_idle = 1'b1;

end else begin

ap\_idle = 1'b0;

end

end

always @ (\*) begin

if (((1'b1 == ap\_CS\_fsm\_state43) & (1'd0 == and\_ln12\_fu\_150\_p2))) begin

ap\_ready = 1'b1;

end else begin

ap\_ready = 1'b0;

end

end

always @ (\*) begin

if ((1'b1 == ap\_CS\_fsm\_state37)) begin

grp\_fu\_70\_opcode = 2'd1;

end else if ((1'b1 == ap\_CS\_fsm\_state27)) begin

grp\_fu\_70\_opcode = 2'd0;

end else begin

grp\_fu\_70\_opcode = 'bx;

end

end

always @ (\*) begin

if ((1'b1 == ap\_CS\_fsm\_state37)) begin

grp\_fu\_70\_p1 = xn\_1\_reg\_292;

end else if ((1'b1 == ap\_CS\_fsm\_state27)) begin

grp\_fu\_70\_p1 = tmp\_1\_reg\_287;

end else begin

grp\_fu\_70\_p1 = 'bx;

end

end

always @ (\*) begin

case (ap\_CS\_fsm)

ap\_ST\_fsm\_state1 : begin

if (((ap\_start == 1'b1) & (1'b1 == ap\_CS\_fsm\_state1))) begin

ap\_NS\_fsm = ap\_ST\_fsm\_state2;

end else begin

ap\_NS\_fsm = ap\_ST\_fsm\_state1;

end

end

ap\_ST\_fsm\_state2 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state3;

end

ap\_ST\_fsm\_state3 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state4;

end

ap\_ST\_fsm\_state4 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state5;

end

ap\_ST\_fsm\_state5 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state6;

end

ap\_ST\_fsm\_state6 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state7;

end

ap\_ST\_fsm\_state7 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state8;

end

ap\_ST\_fsm\_state8 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state9;

end

ap\_ST\_fsm\_state9 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state10;

end

ap\_ST\_fsm\_state10 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state11;

end

ap\_ST\_fsm\_state11 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state12;

end

ap\_ST\_fsm\_state12 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state13;

end

ap\_ST\_fsm\_state13 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state14;

end

ap\_ST\_fsm\_state14 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state15;

end

ap\_ST\_fsm\_state15 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state16;

end

ap\_ST\_fsm\_state16 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state17;

end

ap\_ST\_fsm\_state17 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state18;

end

ap\_ST\_fsm\_state18 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state19;

end

ap\_ST\_fsm\_state19 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state20;

end

ap\_ST\_fsm\_state20 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state21;

end

ap\_ST\_fsm\_state21 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state22;

end

ap\_ST\_fsm\_state22 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state23;

end

ap\_ST\_fsm\_state23 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state24;

end

ap\_ST\_fsm\_state24 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state25;

end

ap\_ST\_fsm\_state25 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state26;

end

ap\_ST\_fsm\_state26 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state27;

end

ap\_ST\_fsm\_state27 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state28;

end

ap\_ST\_fsm\_state28 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state29;

end

ap\_ST\_fsm\_state29 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state30;

end

ap\_ST\_fsm\_state30 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state31;

end

ap\_ST\_fsm\_state31 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state32;

end

ap\_ST\_fsm\_state32 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state33;

end

ap\_ST\_fsm\_state33 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state34;

end

ap\_ST\_fsm\_state34 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state35;

end

ap\_ST\_fsm\_state35 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state36;

end

ap\_ST\_fsm\_state36 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state37;

end

ap\_ST\_fsm\_state37 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state38;

end

ap\_ST\_fsm\_state38 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state39;

end

ap\_ST\_fsm\_state39 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state40;

end

ap\_ST\_fsm\_state40 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state41;

end

ap\_ST\_fsm\_state41 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state42;

end

ap\_ST\_fsm\_state42 : begin

ap\_NS\_fsm = ap\_ST\_fsm\_state43;

end

ap\_ST\_fsm\_state43 : begin

if (((1'b1 == ap\_CS\_fsm\_state43) & (1'd0 == and\_ln12\_fu\_150\_p2))) begin

ap\_NS\_fsm = ap\_ST\_fsm\_state1;

end else begin

ap\_NS\_fsm = ap\_ST\_fsm\_state5;

end

end

default : begin

ap\_NS\_fsm = 'bx;

end

endcase

end

assign add\_ln502\_fu\_191\_p2 = ($signed(12'd3073) + $signed(zext\_ln502\_fu\_187\_p1));

assign and\_ln12\_fu\_150\_p2 = (or\_ln12\_fu\_146\_p2 & grp\_fu\_85\_p2);

assign ap\_CS\_fsm\_state1 = ap\_CS\_fsm[32'd0];

assign ap\_CS\_fsm\_state26 = ap\_CS\_fsm[32'd25];

assign ap\_CS\_fsm\_state27 = ap\_CS\_fsm[32'd26];

assign ap\_CS\_fsm\_state31 = ap\_CS\_fsm[32'd30];

assign ap\_CS\_fsm\_state32 = ap\_CS\_fsm[32'd31];

assign ap\_CS\_fsm\_state36 = ap\_CS\_fsm[32'd35];

assign ap\_CS\_fsm\_state37 = ap\_CS\_fsm[32'd36];

assign ap\_CS\_fsm\_state4 = ap\_CS\_fsm[32'd3];

assign ap\_CS\_fsm\_state41 = ap\_CS\_fsm[32'd40];

assign ap\_CS\_fsm\_state42 = ap\_CS\_fsm[32'd41];

assign ap\_CS\_fsm\_state43 = ap\_CS\_fsm[32'd42];

assign ap\_CS\_fsm\_state5 = ap\_CS\_fsm[32'd4];

assign ap\_return = ((isNeg\_fu\_197\_p3[0:0] === 1'b1) ? zext\_ln662\_fu\_255\_p1 : tmp\_8\_fu\_259\_p4);

assign grp\_fu\_85\_p0 = p\_Result\_s\_fu\_111\_p3;

assign icmp\_ln12\_1\_fu\_140\_p2 = ((trunc\_ln12\_fu\_107\_p1 == 52'd0) ? 1'b1 : 1'b0);

assign icmp\_ln12\_fu\_134\_p2 = ((tmp\_5\_fu\_124\_p4 != 11'd2047) ? 1'b1 : 1'b0);

assign isNeg\_fu\_197\_p3 = add\_ln502\_fu\_191\_p2[32'd11];

assign mantissa\_V\_fu\_173\_p4 = {{{{1'd1}, {tmp\_V\_1\_fu\_169\_p1}}}, {1'd0}};

assign or\_ln12\_fu\_146\_p2 = (icmp\_ln12\_reg\_304 | icmp\_ln12\_1\_reg\_309);

assign p\_Result\_s\_fu\_111\_p3 = {{1'd0}, {trunc\_ln368\_fu\_103\_p1}};

assign p\_Val2\_1\_fu\_156\_p1 = xn\_1\_reg\_292;

assign p\_Val2\_s\_fu\_99\_p1 = reg\_94;

assign r\_V\_1\_fu\_241\_p2 = zext\_ln682\_fu\_183\_p1 << zext\_ln1287\_fu\_227\_p1;

assign r\_V\_fu\_235\_p2 = mantissa\_V\_fu\_173\_p4 >> zext\_ln1285\_fu\_231\_p1;

assign sext\_ln1311\_1\_fu\_223\_p1 = $signed(ush\_fu\_215\_p3);

assign sext\_ln1311\_fu\_211\_p1 = $signed(sub\_ln1311\_fu\_205\_p2);

assign sub\_ln1311\_fu\_205\_p2 = (11'd1023 - tmp\_V\_fu\_159\_p4);

assign tmp\_4\_fu\_247\_p3 = r\_V\_fu\_235\_p2[32'd53];

assign tmp\_5\_fu\_124\_p4 = {{p\_Val2\_s\_fu\_99\_p1[62:52]}};

assign tmp\_8\_fu\_259\_p4 = {{r\_V\_1\_fu\_241\_p2[84:53]}};

assign tmp\_V\_1\_fu\_169\_p1 = p\_Val2\_1\_fu\_156\_p1[51:0];

assign tmp\_V\_fu\_159\_p4 = {{p\_Val2\_1\_fu\_156\_p1[62:52]}};

assign trunc\_ln12\_fu\_107\_p1 = p\_Val2\_s\_fu\_99\_p1[51:0];

assign trunc\_ln368\_fu\_103\_p1 = p\_Val2\_s\_fu\_99\_p1[62:0];

assign ush\_fu\_215\_p3 = ((isNeg\_fu\_197\_p3[0:0] === 1'b1) ? sext\_ln1311\_fu\_211\_p1 : add\_ln502\_fu\_191\_p2);

assign zext\_ln1285\_fu\_231\_p1 = $unsigned(sext\_ln1311\_1\_fu\_223\_p1);

assign zext\_ln1287\_fu\_227\_p1 = $unsigned(sext\_ln1311\_1\_fu\_223\_p1);

assign zext\_ln502\_fu\_187\_p1 = tmp\_V\_fu\_159\_p4;

assign zext\_ln662\_fu\_255\_p1 = tmp\_4\_fu\_247\_p3;

assign zext\_ln682\_fu\_183\_p1 = mantissa\_V\_fu\_173\_p4;

endmodule //sqrtINT

# ПРИЛОЖЕНИЕ 3

**ЛИСТИНГ КОДА МОДУЛЯ ПО АЛГОРИТМУ РАБОТЫ С БИТАМИ НА VERILOG**

module sqrt\_INT(

clk,

rst,

start,

din,

res,

done

);

input clk;

input rst;

input start;

input [31:0]din;

output [31:0] res;

reg [15:0] sqrt;

assign res = { 16’b0,sqrt };

always @(posedge clk , posedge start) begin

if (din == 8'hXXXXXXXX)

done <=1'b0;

else begin

sqrt[15] = din[31:30] == 2'b00 ? 1'b0 : 1'b1;

sqrt[14] = din[31:28] < {sqrt[15], 1'b1} \* {sqrt[15], 1'b1} ? 1'b0 : 1'b1;

sqrt[13] = din[31:26] < {sqrt[15:14], 1'b1} \* {sqrt[15:14], 1'b1} ? 1'b0 : 1'b1;

sqrt[12] = din[31:24] < {sqrt[15:13], 1'b1} \* {sqrt[15:13], 1'b1} ? 1'b0 : 1'b1;

sqrt[11] = din[31:22] < {sqrt[15:12], 1'b1} \* {sqrt[15:12], 1'b1} ? 1'b0 : 1'b1 ;

sqrt[10] = din[31:20] < {sqrt[15:11], 1'b1} \* {sqrt[15:11], 1'b1} ? 1'b0 : 1'b1;

sqrt[9] = din[31:18] < {sqrt[15:10], 1'b1} \* {sqrt[15:10], 1'b1} ? 1'b0 : 1'b1;

sqrt[8] = din[31:16] < {sqrt[15:9 ], 1'b1} \* {sqrt[15:9], 1'b1} ? 1'b0 : 1'b1;

sqrt[7] = din[31:14] < {sqrt[15:8 ], 1'b1} \* {sqrt[15:8], 1'b1} ? 1'b0 : 1'b1 ;

sqrt[6] = din[31:12] < {sqrt[15:7 ], 1'b1} \* {sqrt[15:7], 1'b1} ? 1'b0 : 1'b1;

sqrt[5] = din[31:10] < {sqrt[15:6 ], 1'b1} \* {sqrt[15:6], 1'b1} ? 1'b0 : 1'b1;

sqrt[4] = din[31:8] < {sqrt[15:5 ], 1'b1} \* {sqrt[15:5], 1'b1} ? 1'b0 : 1'b1;

sqrt[3] = din[31:6] < {sqrt[15:4 ], 1'b1} \* {sqrt[15:4], 1'b1} ? 1'b0 : 1'b1;

sqrt[2] = din[31:4] < {sqrt[15:3 ], 1'b1} \* {sqrt[15:3], 1'b1} ? 1'b0 : 1'b1;

sqrt[1] = din[31:2] < {sqrt[15:2 ], 1'b1} \* {sqrt[15:2], 1'b1} ? 1'b0 : 1'b1;

sqrt[0] = din[31:0] < {sqrt[15:1 ], 1'b1} \* {sqrt[15:1], 1'b1} ? 1'b0 : 1'b1;

end

end

endmodule

# ПРИЛОЖЕНИЕ 4

**ЛИСТИНГ КОДА МНОГОТАКТНОГО МОДУЛЯ ПО АЛГОРИТМУ РАБОТЫ С БИТАМИ НА VERILOG**

always\_comb begin

case (sqrt\_state)

2'b00: begin

sqrt\_op1 = ialu\_op1;

sqrt\_res = 31'b0;

sqrt\_res[15] = sqrt\_op1[31:30] == 2'b00 ? 1'b0 : 1'b1;

sqrt\_res[14] = sqrt\_op1[31:28] < {sqrt\_res[15], 1'b1} \* {sqrt\_res[15], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[13] = sqrt\_op1[31:26] < {sqrt\_res[15:14], 1'b1} \* {sqrt\_res[15:14], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[12] = sqrt\_op1[31:24] < {sqrt\_res[15:13], 1'b1} \* {sqrt\_res[15:13], 1'b1} ? 1'b0 : 1'b1;

sqrt\_rdy = 1'b0;

end

2'b01: begin

sqrt\_res[11] = sqrt\_op1[31:22] < {sqrt\_res[15:12], 1'b1} \* {sqrt\_res[15:12], 1'b1} ? 1'b0 : 1'b1 ;

sqrt\_res[10] = sqrt\_op1[31:20] < {sqrt\_res[15:11], 1'b1} \* {sqrt\_res[15:11], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[9] = sqrt\_op1[31:18] < {sqrt\_res[15:10], 1'b1} \* {sqrt\_res[15:10], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[8] = sqrt\_op1[31:16] < {sqrt\_res[15:9 ], 1'b1} \* {sqrt\_res[15:9], 1'b1} ? 1'b0 : 1'b1;

sqrt\_rdy = 1'b0;

end

2'b10: begin

sqrt\_res[7] = sqrt\_op1[31:14] < {sqrt\_res[15:8 ], 1'b1} \* {sqrt\_res[15:8], 1'b1} ? 1'b0 : 1'b1 ;

sqrt\_res[6] = sqrt\_op1[31:12] < {sqrt\_res[15:7 ], 1'b1} \* {sqrt\_res[15:7], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[5] = sqrt\_op1[31:10] < {sqrt\_res[15:6 ], 1'b1} \* {sqrt\_res[15:6], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[4] = sqrt\_op1[31:8] < {sqrt\_res[15:5 ], 1'b1} \* {sqrt\_res[15:5], 1'b1} ? 1'b0 : 1'b1;

sqrt\_rdy = 1'b0;

end

2'b11: begin

sqrt\_res[3] = sqrt\_op1[31:6] < {sqrt\_res[15:4 ], 1'b1} \* {sqrt\_res[15:4], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[2] = sqrt\_op1[31:4] < {sqrt\_res[15:3 ], 1'b1} \* {sqrt\_res[15:3], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[1] = sqrt\_op1[31:2] < {sqrt\_res[15:2 ], 1'b1} \* {sqrt\_res[15:2], 1'b1} ? 1'b0 : 1'b1;

sqrt\_res[0] = sqrt\_op1[31:0] < {sqrt\_res[15:1 ], 1'b1} \* {sqrt\_res[15:1], 1'b1} ? 1'b0 : 1'b1;

sqrt\_rdy = 1'b1;

end

default: sqrt\_rdy = 1'b0;

endcase

end

///FSM SQRT

always\_ff @(posedge clk, negedge rst\_n) begin

if (~rst\_n) begin

sqrt\_state <= 2'b00;

end

else

if (sqrt\_req) begin

case (sqrt\_state)

2'b00: sqrt\_state <= 2'b01;

2'b01: sqrt\_state <= 2'b10;

2'b10: sqrt\_state <= 2'b11;

default: sqrt\_state <= 2'b00;

endcase

end else begin

sqrt\_state <= 2'b00;

end

end

# ПРИЛОЖЕНИЕ 5

**ЛИСТИНГ КОДА ТЕСТА ДЛЯ СИМУЛЯЦИИ В VERILATOR**

#include "riscv\_test.h"

#include "riscv\_macros.h"

#include "test\_macros.h"

#define SQRT\_OPCODE 0xFE001013

#define RD\_SHIFT 7

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

#define RS1\_SHIFT 15

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

#define INST\_SQRT(rd, rs1) \

.word (SQRT\_OPCODE | ((rd) << RD\_SHIFT & RD\_MASK) | ((rs1) << RS1\_SHIFT & RS1\_MASK))

#define TEST\_IMM\_OP\_SQRT( testnum, inst, result, val1) \

TEST\_CASE( testnum, x3, result, \

li x1, MASK\_XLEN(val1); \

inst ; \

)

#define TEST\_INSERT\_NOPS\_0

#define TEST\_INSERT\_NOPS\_1 nop; TEST\_INSERT\_NOPS\_0

#define TEST\_INSERT\_NOPS\_2 nop; TEST\_INSERT\_NOPS\_1

#define TEST\_SQRT\_DEST\_BYPASS( testnum, nop\_cycles, inst, result, val1 ) \

TEST\_CASE( testnum, x3, result, \

li x6, 0; \

1: li x1, MASK\_XLEN(val1); \

inst;\

TEST\_INSERT\_NOPS\_ ## nop\_cycles \

addi x28, x3, 0; \

addi x6, x6, 1; \

li x7, 2; \

bne x6, x7, 1b \

)

#define TEST\_SQRT\_SCR1\_BYPASS( testnum, nop\_cycles, inst, result, val1 ) \

TEST\_CASE( testnum, x3, result, \

li x6, 0; \

1: li x1, MASK\_XLEN(val1); \

TEST\_INSERT\_NOPS\_ ## nop\_cycles \

inst ; \

addi x6, x6, 1; \

li x7, 2; \

bne x6, x7, 1b \

)

RVTEST\_RV32U

RVTEST\_CODE\_BEGIN

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

# Arithmetic tests begin

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

li x4, 0xFFFFFFFF;

TEST\_IMM\_OP\_SQRT( 1, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF); // max val

li x4, 0xFFFFFFFE

TEST\_IMM\_OP\_SQRT( 2, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFE); // max val-1

li x4,0xFFFFFFFD

TEST\_IMM\_OP\_SQRT( 3, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFD); // max val-2

li x4, 0x0

TEST\_IMM\_OP\_SQRT( 4, INST\_SQRT(3, 4), 0x0, 0x0); // val 0

li x4, 0x1

TEST\_IMM\_OP\_SQRT( 5, INST\_SQRT(3, 4), 0x1, 0x1); // val 1

li x4, 0x2

TEST\_IMM\_OP\_SQRT( 6, INST\_SQRT(3, 4), 0x1, 0x2); // val 2

li x4, 0x3

TEST\_IMM\_OP\_SQRT( 7, INST\_SQRT(3, 4), 0x1, 0x3); // val 3

li x4, 0x4

TEST\_IMM\_OP\_SQRT( 8, INST\_SQRT(3, 4), 0x2, 0x4); // val 4

li x4, 0xA9

TEST\_IMM\_OP\_SQRT( 9, INST\_SQRT(3, 4), 13, 0xA9); // val 169

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

# Arithmetic tests end

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

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

# Bypassing tests begin

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

li x4, 0xFFFFFFFF;

TEST\_SQRT\_DEST\_BYPASS( 10, 0, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

TEST\_SQRT\_DEST\_BYPASS( 11, 1, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

TEST\_SQRT\_DEST\_BYPASS( 12, 2, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

TEST\_SQRT\_SCR1\_BYPASS( 13, 0, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

TEST\_SQRT\_SCR1\_BYPASS( 14, 1, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

TEST\_SQRT\_SCR1\_BYPASS( 15, 2, INST\_SQRT(3, 4), 0xFFFF, 0xFFFFFFFF );

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

# Bypassing tests end

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

TEST\_PASSFAIL

RVTEST\_CODE\_END

.data

RVTEST\_DATA\_BEGIN

TEST\_DATA

RVTEST\_DATA\_END