Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“Национальный исследовательский университет ИТМО”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Разработка высокоскоростного интерфейса аппаратного ускорителя для системной динамики

Автор Чижиков Иван Валерьевич _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность) 09.04.01
(код, наименование)
Технологии компьютерного моделирования

Квалификация магистр
(бакалавр, магистр)*

Руководитель Перл И. А., доцент, к.т.н _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

Соруководитель Пенской А. В., к.т.н. _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

Вуз-партнер Университет ИТМО
(официальное название вуза)

Санкт-Петербург, 2020 г.

Обучающийся__Чижиков Иван Валерьевич_____
(ФИО полностью)

Группа__P42152_____Факультет/институт/кластер__факультет ПИиКТ_____

Направленность (профиль), специализация _____
_____

Консультант (ы):

а) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

б) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

ВКР принята “____”_____20 ____г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты “____”_____20 ____г.

Секретарь ГЭК _____ _____
(ФИО) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения ___ _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО"

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся: Чижиков Иван Валерьевич
(ФИО)

Наименование темы ВКР: Разработка высокоскоростного интерфейса аппаратного ускорителя для системной динамики

Наименование организации, где выполнена ВКР: Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1. Цель исследования: Разработать высокоскоростной интерфейс взаимодействия между вычислительной платформой NITTA, способной эффективно рассчитывать модели системной динамики, и конечным пользователем использующем операционную систему Linux.
2. Задачи, решаемые в ВКР: выбор и обоснование высокоскоростного интерфейса передачи данных для вычислительной платформы NITTA, разработка схемотехнического модуля на языке описания аппаратуры обеспечивающего передачу данных вычислительной платформе NITTA через выбранный интерфейс, разработка схемотехнического модуля, позволяющего изменять исполняемый NITTA алгоритм, разработка драйвера Linux обеспечивающий взаимодействие вычислительной платформы NITTA с ПЛИС по средствам выбранного интерфейса, разработка пользовательской библиотеки для удобной работы пользователя с драйвером.
3. Число источников, использованных при составлении обзора: 12
4. Полное число источников, использованных в работе: 20
5. В том числе источников по годам:

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
6	0	1	11	0	2

6. Использование информационных ресурсов Internet: 18
7. Использование современных пакетов компьютерных программ и технологий

Пакеты компьютерных программ и технологий	**Раздел работы**
Microsoft Word 2016	1-4
Intel Quartus Prime	3
Platform Designer	3
Register Transfer Level Viewer	3
ModelSim	3, 4
Pin Planner	3
Intel Quartus Prime Programmer	3
Vim	4
Make	4
gcc	4

8. Краткая характеристика полученных результатов: разработан интерфейс передачи данных между компьютером с операционной системой на основе ядра Linux и вычислительной платформы NITTA используя PCI Express, разработан протокол изменения исполняемого алгоритма вычислительной платформы NITTA без необходимости реконфигурирования, разработан PCI Express драйвер операционной системы на основе ядра Linux, разработана пользовательская библиотека к драйверу. Прошивка занимает на программируемой логической интегральной схеме 10844 логических элемента, 6345 регистров и 32896 бит памяти.
9. Полученные гранты при выполнении работы: нет
10. Наличие публикаций и выступлений на конференциях по теме выпускной работы: нет

Обучающийся_____ _____
(ФИО) (подпись)

Руководитель ВКР_____ _____
(ФИО) (подпись)

“_____”_____20___г.

Ministry of Science and Higher Education of the Russian Federation
FEDERAL STATE AUTONOMOUS EDUCATIONAL INSTITUTION OF HIGHER EDUCATION
"NATIONAL RESEARCH UNIVERSITY ITMO"

ANNOTATION

GRADUAL QUALIFICATION WORK

Student: Ivan Chizhikov
WRC topic title: Development of a high-speed hardware interface accelerator for system dynamics
Name of the organization where the WRC is performed: ITMO University

CHARACTERISTIC OF FINAL QUALIFICATION WORK

1. Research objective: Developed by a high-speed interface between the computing platform and NITTA, capable of efficiently calculating system dynamics models and for end users using the Linux operating system.
2. Tasks solved in WRC: selection and justification of a high-speed transmission interface data for the NITTA computing platform, development of a circuit module on hardware description language providing data transfer to a computing platform NITTA through the selected interface, the development of a circuit module allowing modify the executable NITTA algorithm, the development of the Linux driver provides interaction of the NITTA computing platform with FPGA by means of the selected interface, development of a user library for convenient user work with the driver.
3. Number of sources used in compiling the review: 12
4. The total number of sources used in the work: 20
5. Including sources by year:

Domestic			**Of foreign**		
Last 5 years old	From 5 to 10 years	More 10 years	Last 5 years	From 5 to 10 years	More 10 years
6	0	1	11	0	2

6. Use of Internet information resources: 18
7. Using modern software packages and technologies:

Computer software and technology packages	**Work section**
Microsoft Word 2016	1-4
Intel Quartus Prime	3
Platform designer	3
Register Transfer Level Viewer	3
ModelSim	3, 4
Pin planner	3
Intel Quartus Prime Programmer	3
Vim	4
Make	4
gcc	4

8. Brief description of the results: a data transfer interface is developed between a computer with an operating system based on the Linux kernel and a computing NITTA platform using PCI Express, developed a protocol for changing the executable NITTA computing platform algorithm without the need for reconfiguration, developed PCI Express driver operating system based on the Linux kernel, developed user library to the driver.
9. Grants received upon completion of work: no
10. Availability of publications and speeches at conferences on the theme of final work: no
Student _____ _____
Head of WRC _____ _____

“_____” _____ 20 ___

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

"НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО"

УТВЕРЖДАЮ

Руководитель ОП

_____ _____

(Фамилия, И.О.) (подпись)

«____» «_____» 20____ г.

З А Д А Н И Е
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Обучающийся Чижиков Иван Валерьевич

(ФИО полностью)

Группа P42152 **Факультет/институт/кластер** факультет ПИиКТ

Квалификация магистр

(магистр, инженер, бакалавр)**

Направление подготовки 09.04.01 – Информатика и вычислительная техника

(код, название направления подготовки)

Направленность (профиль) образовательной программы Технологии компьютерного моделирования

Специализация Основы моделирования сложных систем

Тема ВКР Разработка высокоскоростного интерфейса аппаратного ускорителя для системной динамики

Руководитель Перл Иван Андреевич, доцент ФПИиКТ, к.т.н., ординарный доцент

(ФИО полностью, место работы, должность, ученая степень, ученое звание)

2 Срок сдачи студентом законченной работы до «_____» «_____» 20_____г.

3 Техническое задание и исходные данные к работе Разработать высокоскоростной интерфейс передачи данных между процессором NITTA и приложением системной динамики для операционной системы Linux.

Исходными данными является схемотехническое описание вычислительной платформы NITTA на языке описания аппаратуры Verilog.

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

1) Выбрать и обосновать высокоскоростной интерфейс передачи данных для вычислительной платформы NITTA.

2) Разработать схемотехнический модуль на языке описания аппаратуры обеспечивающий передачу данных вычислительной платформе NITTA через выбранный интерфейс.

3) Разработать схемотехнический модуль, позволяющий изменять исполняемый NITTA алгоритм.

4) Разработать драйвер операционной системы Linux обеспечивающий взаимодействие вычислительной платформы NITTA с ПЛИС по средствам выбранного интерфейса.

5) Разработать пользовательскую библиотеку для удобной работы пользователя с драйвером.

5 Перечень графического материала (с указанием обязательного материала)

6 Исходные материалы и пособия

Дэвид М. Харрис и Сара Л. Харрис Цифровая схемотехника и архитектура компьютера

Издательство Morgan Kaufman English Edition 2013

7 Дата выдачи задания «____» «_____» 20____г.

Руководитель ВКР_____
(подпись)

Задание принял к исполнению_____ «____» «_____» 20____г.
(подпись)

Оглавление

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

ПЛИС – Программируемая логическая интегральная схема.

САПР – Система автоматизированного проектирования.

NITTA – NISC/TTA, специализированный реконфигурируемый вычислитель для которого ведется разработка.

САПР NITTA – САПР для создания реконфигурируемых вычислителей реального времени на ПЛИС.

SPI – Serial Peripheral Interface, последовательный периферийный интерфейс.

I2C – Inter-Integrated Circuit, последовательная шина данных для связи интегральных схем.

Lua – язык программирования высокого уровня.

Linux – ядро некоторого количества операционных систем.

PCI Express – Peripheral Component Interconnect Express, один из высокоскоростных интерфейсов передачи данных.

Makefile – файл с набором инструкций для программы make.

make – программа упрощающая процесс компиляции проектов.

NISC – Non instruction set computing, вычисление без набора команд.

TTA – Transport triggered architecture, транспортно ориентированная архитектура.

CISC – Complex instruction set computer, вычислитель с полным набором команд.

RISC – Reduced instruction set computer, вычислитель с сокращенным набором команд.

XMILE – XML Modeling Interchange Language, язык описания моделей системной динамики основанный на XML.

XML – Extensible Markup Language, расширяемый язык разметки.

DMA – Direct memory access, прямой доступ к памяти.

AMD – Advanced micro devices, производитель интегральной микросхемной электроники.

Intel – производитель интегральной микросхемной электроники.

Altera – дочерняя компания Intel.

Cyclone – ряд ПЛИС компании Altera.

ID – идентификатор.

Vendor ID – уникальный идентификатор производителя устройства.

Device ID – уникальный идентификатор модели устройства.

VHDL, Verilog, SystemVerilog – языки описания аппаратуры.

FIFO – first in, first out, способ организации взаимодействия с памятью, очередь.

BAR – Base Address Register, базовый адресный регистр.

Qsys – инструмент системной интеграции в программе Quartus Prime.

Quartus Prime – программа для создания прошивок для ПЛИС.

ВВЕДЕНИЕ

Актуальность темы исследования

Системная динамика все чаще применяется для предсказания поведения систем в различных ситуациях, а сложность моделей постоянно возрастает, что требует больших вычислительных мощностей, также их использование позволяет улучшить точность расчета. Сейчас различные вычисления производятся либо на процессорах общего назначения, либо на видеокартах. В некоторых случаях разрабатываются интегральные схемы специального назначения, но они дороги в проектировании и производстве, особенно при мелкосерийных масштабах. Проблема дороговизны производства была решена при помощи программируемых логических интегральных схем (ПЛИС), т.к. они имеют изменяемую логику. Их можно использовать для различных задач и, соответственно, можно производить одну и ту же модель для решения различных задач, что уменьшает затраты на проектирование (в меньшей степени) и производство. Однако, сам процесс описания внутренней логики ПЛИС является достаточно сложным для конечного пользователя так как производится на специализированных языках описания аппаратуры и имеет множество особенностей по сравнению с общепринятыми языками программирования. Для решения этой проблемы разрабатывается система автоматизированного проектирования NITTA предназначенная для упрощения конфигурирования ПЛИС конечным пользователем путем компиляции программ написанных на языке Lua в некоторый код, интерпретируемый процессором NITTA в набор команд, которые могут исполняться на ПЛИС.

Степень разработанности темы исследования

Ранее уже были разработаны интерфейсы на основе SPI и I2C для проекта NITTA, но скорость передачи данных по ним не является достаточной

для сколь бы то ни было существенных задач, таких как расчет моделей системной динамики.

Цели исследования

Разработать высокоскоростной интерфейс взаимодействия между вычислительной платформой NITTA, способной эффективно рассчитывать модели системной динамики, и конечным пользователем использующем операционную систему Linux.

Задачи исследования

1) Выбрать и обосновать высокоскоростной интерфейс передачи данных для вычислительной платформы NITTA.

2) Разработать схемотехнический модуль на языке описания аппаратуры обеспечивающий передачу данных вычислительной платформе NITTA через выбранный интерфейс.

3) Разработать схемотехнический модуль, позволяющий изменять исполняемый NITTA алгоритм.

4) Разработать драйвер операционной системы Linux обеспечивающий взаимодействие вычислительной платформы NITTA с ПЛИС по средствам выбранного интерфейса.

5) Разработать пользовательскую библиотеку для удобной работы пользователя с драйвером.

Научная новизна

В представленной работе разработан высокоскоростной интерфейс передачи данных для нового семейства процессоров, который позволяет не только передавать данные для вычислительного алгоритма, но и задавать сам алгоритм без реконфигурирования ПЛИС.

Теоретическая и практическая значимость работы

Данная работа будет полезна людям, пытающимся разработать собственный механизм взаимодействия ПЛИС с пользователем используя PCI Express, особенно потому что на русском языке не так много информации об

этом. Результаты данной работы будут использоваться непосредственно в самом проекте NITTA как основной способ передачи данных и для загрузки вычислительного алгоритма.

Степень достоверности и апробация результатов

Разработанные схемотехнические модули были протестированы в тестовом окружении, эмулирующем ПЛИС. Проект запускался на испытательном стенде, состоящем из компьютера с операционной системой Linux, ПЛИС Altera Cyclone IV EP4CGX75CF23I7 на плате расширения с интерфейсом PCI Express X4.

Объем и структура работы

Работа содержит четыре главы. Первая глава посвящена обзору исследуемой области и выбору интерфейса передачи данных. Вторая глава посвящена описанию организации памяти в проекте и разработанных протоколов передачи данных. В третьей главе описываются разработанные схемотехнические модули для ПЛИС. Четвертая глава посвящена разработке драйвера операционной системы Linux, пользовательской библиотеки и тестированию проекта. Также работа содержит приложения А, Б, В. Приложение А содержит описание разработанных схемотехнических модулей на языках описания архитектуры. Приложение Б содержит код драйвера Linux, Makefile и скрипт позволяющие удобным образом собирать и устанавливать драйвер в ядро операционной системы. Приложение В содержит код пользовательской библиотеки и Makefile для удобства ее сборки.

ГЛАВА 1. ОПИСАНИЕ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Системная динамика

Системная динамика — один из способов имитационного моделирования поведения системы посредством интегрирования множества взаимосвязанных функций по времени с некоторым шагом. Данная концепция была предложена инженером Массачусетского технологического института Джеем Райт Форрестером (Jay Wright Forrester) в публикации Industrial dynamics 1961г. про «исследование информационных обратных связей в промышленной деятельности с целью показать, как организационная структура, усиления (в политиках) и задержки (в принятии решений и действиях) взаимодействуют, влияя на успешность предприятия». После чего получила широкое распространение в экономике, управлении предприятием, сельском хозяйстве, экологии, социальных науках, сфере обслуживания и других [1, 2].

Сиситемодинамическая модель состоит из параметров и характеристик описываемой системы, называемых стоками и описаний их взаимодействия между собой называемых потоками. Вычисление осуществляется итеративно, где на каждом шаге в зависимости от специфики характеристики для вычисления ее значения на следующем промежутке времени берется связанные с ней значения начальных параметров, значений на предыдущей итерации или начальных значений и входных данных вычисляется новое значение этой характеристики. На каждой итерации вычисляются значения всех характеристик.

В своей книге Основы системного анализа [3] Горохов А. В. утверждает следующее.

Важной составляющей имитационного моделирования является диалоговое общение исследователя, разработчика, проектировщика в

процессе моделирования с комплексом, реализующим модель, что позволяет в наибольшей степени использовать опыт и интуицию специалистов в изучении реальных сложных систем. Это необходимо для контроля текущих результатов и способности корректировки развития моделируемой ситуации с целью получения новых знаний о характере изучаемых процессов, а также для обучения специалистов работе с новыми системами.

С другой стороны, системно-динамическое моделирование имеет существенное ограничение - ограниченную точность моделирования и невозможность априорной оценки (косвенную характеристику точности может дать анализ чувствительности модели к изменениям отдельных параметров исследуемых систем). Кроме того, разработка хорошей системно-динамической модели часто обходится дороже создания аналитических моделей и требует больших временных затрат. Тем не менее, системная динамика является одним из наиболее широко используемых методов при решении задач синтеза и анализа сложных систем.

Основными преимуществами системодинамического подхода перед другими видами моделирования являются:

— сложные слабоформализованные ситуации, в которых невозможно применение аналитических методов или они настолько сложны и трудоемки, что динамическое моделирование дает более простой способ решения проблемы;

— моделирование поведения систем в ситуациях, которые ранее не встречались; в данном случае имитация служит для предварительной проверки новых стратегий управления системой перед проведением эксперимента на реальном объекте;

— моделирование ситуаций, наблюдение которых осложнено большой длительностью их развития или наоборот, то есть когда необходимо контролировать развитие ситуации путем ускорения или замедления явлений в ходе имитации.

Наиболее частыми задачами, решаемыми при помощи системной динамики, являются:

— исследование сложных систем с целью выявления причинно-следственных связей;

— прогнозирование последствия изменения стратегий управления сложной системой;

— обучение специалистов работе со сложными природнотехническими комплексами [3].

Сложность расчета моделей системной динамики зависит от количества параметров и сложности вычисляемых математических выражений, а также от шага интегрирования. Поэтому экономически эффективным и удобным для конечного пользователя способом расчета сложных системно-динамических моделей является использование специализированных вычислительных систем. Наиболее целесообразным способом взаимодействия с такими системами является удаленный доступ по средствам Интернет. Сейчас в Университете ИТМО ведется разработка облачной вычислительной системы sdCloud предназначенной для этих целей. С ее помощью пользователи со всего мира имеют возможность вычислять свои модели на специализированных серверах, хранить и делиться ими с другими пользователями [4].

Однако, использование мощных серверов все еще не является наиболее эффективным способом вычисления. Поэтому ведется работа над использованием программируемых логических интегральных схем с реконфигурируемой вычислительной платформой NITTA. Именно за счет возможности подстройки внутренней архитектуры под требуемые задачи планируется достичь большей эффективности вычислений.

1.2 Вычислительная платформа NITTA

Проект NITTA основан на идее объединения концепций вычислений без набора команд (Non instruction set computing, NISC) и транспортно-ориентированной архитектуры (Transport triggered architecture, TTA). Для понимания работы проекта в целом следует рассмотреть эти концепции отдельно.

NISC предполагает статически запланированный набор управляющих групп сигналов. Статически запланированный означает, что вся последовательность сигналов становится известной на этапе компиляции проекта. Эти сигналы являются простыми запросами к управляемым элементам без необходимости интерпретации команд как в архитектурах вычислитель с полным набором команд (Complex instruction set computer, CISC) и вычислитель с сокращенным набором команд (Reduced instruction set computer, RISC). Преимуществами данной архитектуры над CISC и RISC являются:

— простой управляющий контроллер, без аппаратного планировщика и декодера команд;

— более эффективное использование ресурсов аппаратуры;

— прост в проектировании.

Использование же TTA, смысл которой заключается в прямой передаче данных между вычислительными модулями, исключая буферные блоки, позволяет получить еще большую эффективность выполнения алгоритма [5].

В концепции NISC набор сменяемых групп управляющих сигналов описывает выполнение некоторого алгоритма или его этап полностью. Это означает, что алгоритм не обязательно задавать исключительно на этапе компиляции, а можно менять между циклами, если алгоритм не является агрегирующим прошлые результаты или дальнейшее использование текущего

алгоритма не запланировано. Безусловно, новый алгоритм должен соответствовать текущей конфигурации ПЛИС.

Изменение алгоритма вычислителя без необходимости реконфигурирования ПЛИС на данный момент не реализовано в NITTA. Такая возможность в некоторых случаях могла бы позволить намного быстрее начать выполнение нового алгоритма, так как реконфигурация является достаточно продолжительным процессом.

Примером применения такой возможности может быть возможность быстро начать перерасчет системодинамической модели с новыми значениями констант.

Реализация этой концепции осуществляется на программируемой логической интегральной схеме при помощи специализированных языков описания аппаратуры. Использование ПЛИС позволяет подстроить эту архитектуру под определенную задачу, что должно обеспечить прирост в скорости работы и энергоэффективности при выполнении алгоритма.

Для удобной и быстрой работы с вычислителем ведется разработка системы автоматизированного проектирования имеющей многоуровневую структуру:

— прикладной уровень, на котором осуществляется интерпретация модели системной динамики на языке XMILE или прикладного алгоритма на языке высокого уровня Lua в код промежуточного уровня;

— промежуточный уровень, описывает шаг моделирования как набор синхронных потоков данных;

— внутренний уровень, описывает требуемые вычислительные блоки, а также порядок передачи данных между ними;

— уровень реализации, состоит из схемотехнических модулей, представляющих внутренний уровень на языке описания аппаратуры для конкретной программируемой логической интегральной схемы.

Такая структура позволяет автоматизировать большинство процессов необходимых для конфигурирования вычислителя, позволяя конечному пользователю не задумываться о ее реализации, однако в случае необходимости остается возможность для ручной настройки под специфичные требования на любом из уровней.

Модели системной динамики могут быть хорошо описаны с использованием вычислительной платформы NITTA ввиду следующих особенностей:

— объем вычислений для каждой итерации детерминирован, а значит, может выполняться в реальном времени;

— алгоритм не имеет регулярной структуры, что препятствует эффективному использованию графических процессоров;

— значения разных стоков могут рассчитываться параллельно, что позволяет говорить о перспективности использования ПЛИС [6].

Все вышесказанные преимущества позволяют эффективно выполнять разнообразные вычислительные алгоритмы, в том числе и расчет системно-динамических моделей.

1.3 Интерфейсы передачи данных

Для выполнения любых задач необходимы начальные данные, которые эффективно должны быть предоставлены вычислительной платформе. Для этого начальные данные должны получаться от пользователя. Для этой цели существует большое количество различных интерфейсов передачи данных между компьютером и внешним устройством. Для вычислительной платформы NITTA уже имеются интерфейсы передачи данных такие как последовательный периферийный интерфейс (Serial Peripheral Interface, SPI) и межинтегральная схема (Inter-Integrated Circuit, I^2C), однако, скорость передачи по ним не является достаточно высокой для требуемых задач.

Так как количество начальных данных для моделей системной динамики может быть достаточно большим в случаях, когда система имеет большое количество начальных параметров, когда необходимо загружать некоторую статистику за определенный промежуток времени, или же необходимо загружать новый набор данных для каждого этапа вычисления, в том числе и данные процессов происходящих в реальном времени, например, от устройств Интернета вещей, таким образом полученные данные будут наиболее актуальными и дадут более точный прогноз, особенно в ближнесрочной перспективе [7].

Например, SPI интерфейс предполагает по одному контакту для передачи данных в каждом направлении с заданной частотой, IIC также для передачи данных использует два контакта, что сильно ограничивает скорость передачи данных. Для обеспечения требуемой пропускной способности рассмотрим наиболее популярные высокоскоростные интерфейсы передачи данных: RapidIO, PCI, PCI Express, QuickPath Interconnect и HyperTransport. В таблице 1 представлены максимальные пропускные способности этих шин.

Таблица 1 Высокоскоростные интерфейсы передачи данных

Название	Скорость передачи
RapidIO LP-LVDS	8,53 Гбит/с
PCI 3.0	2 Гбит/с
PCI Express 3.0 (x16)	126,03 Гбит/с
QuickPath Interconnect 64x 3,2 ГГц	204,8 Гбит/с
HyperTransport 3.1 32/3,2 ГГц	409,6 Гбит/с

RapidIO – интерфейс передачи данных разработанный Mercury Computer Systems и Motorola используется для соединения внутри печатных плат, а также нескольких плат между собой. Обладает следующими особенностями:

— равноправные абоненты, где каждый может инициировать обмен данных;

— возможность отправки сообщений пакетами;

— чтение и запись используя прямой доступ памяти direct memory access (DMA), без использования процессора;

— сетевая организация взаимодействующих элементов;

— обработка неупорядоченной передачи данных;

— имеет широкое адресное пространство 48 и 64 бита;

— требуется поддержка управления несколькими транзакциями одновременно [8].

PCI – интерфейс передачи данных предназначенный для соединения компонентов встроенного контроллера между собой, подключения периферийных устройств к компьютерным системам и памяти. Изначально разрабатывался для передачи видеосигнала, но после получил широкую популярность для различных задач. Особенности:

— низкая задержка для доступа к случайному участку памяти, около 60 наносекунд;

— поддерживает регистры с информацией об устройстве;

— поддержка прямого доступа к памяти (DMA);

— поддерживает адресное пространство как 32, так и 64 бита;

— имеется несколько адресных пространств, которые могут выполнять различные функции [9].

PCI Express – это высокопроизводительная универсальная шина ввода/вывода, предназначенная для широкого спектра вычислительных и коммуникационных платформ созданная на базе PCI, имеет такую же программную модель, но обладает рядом преимуществ. Особенности:

— имеет тип соединения точка-точка, что исключает конкуренцию устройств на компьютерной шине;

— возможность управления питанием;

— является полнодуплексным интерфейсом;

— возможность вставки устройства с PCI Express в уже работающую систему;

— различные совместимые друг с другом форм-факторы;

— возможность создания отчетов об ошибках;

— возможность обработки ошибок;

— поддержка различных напряжений на подключаемых устройствах;

— обеспечение целостности данных с накладными расходами на передачу около 1,5% [10].

QuickPath Interconnect – последовательная шина данных разработанная Intel для соединения процессоров в многопроцессорных системах и процессора с чипсетом. Достоинствами этой шины являются:

— одна из наибольших среди существующих скоростей передача информации;

— соединение типа точка-точка;

— возможность использования общей памяти для нескольких элементов [11].

HyperTransport – это современная высокопроизводительная масштабируемая технология межточечного соединения с высокой пропускной способностью, основанная на пакетах, в виде HT линий, HT-разъемов и HT-кабелей, которые соединяют процессоры друг к другу, процессоры к сопроцессорам и процессоры для ввода-вывода и периферийные контроллеры. Ее преимущества:

— является самой быстрой из существующих шин передачи данных;

— поддерживает высокий спектр частот;

— большой спектр возможной ширины линий;

— можно организовать как последовательную и как параллельную шину;

— является открытой, что позволяет различным производителям использовать ее в своих целях [12].

Шина HyperTransport является наиболее предпочтительной шиной для использования ее в передаче данных, однако она разработана компанией AMD, а разработка проекта ведется преимущественно на ПЛИС компании Altera, которая в свою очередь принадлежит Intel. Intel является прямым конкурентом AMD и ни она, ни ее дочерние компании не используют шины HyperTransport. Другой высокоскоростной шиной является QuickPath Interconnect, но она не применяется для подключения периферийных устройств к компьютеру, коим и является плата с находящейся на ней ПЛИС.

Таким образом для данного проекта был выбран интерфейс передачи данных PCI Express версии 3.0. Хоть и существует уже версия 4.0, но она еще особо распространена на современных материнских платах и готовых плата с установленными на них ПЛИС. Ширина шины для проекта была выбрана x4 ввиду имеющейся платы с поддержкой PCI Express, но специфика интерфейса предполагает гибкость в плане выбора ширины шины и изменение ее на другую не представляется сложным.

1.4 Интерфейс PCI Express

Рассмотрим детальнее интерфейс используемый в этом проекте.

Каждый контакт для передачи данных использует низковольтную дифференциальную передачу сигнала (LVDS).

Контакты для x4 конфигурации:

— RX0, RX1, RX2, RX3 – сигналы приема данных;

— TX0, TX1, TX2, TX3 – сигналы передачи данных;

— REFCLK+, REFCLK- – сигнала опорной частоты 100 МГц;

— RST – сигнала сброса карты;

— сигналы питания: +3.3V, +12V, +3.3Vaux, GND;

— сигналы шины SMBus – SMB_CLK, SMB_DATA;

— сигналы интерфейса JTAG – TCLK, TDI, TDO, TMS, TRST#.

PCI Express устройство отображается в памяти устройства как набор регистров определяющих конфигурационное пространство. Регистры Vendor ID, Device ID, Command, Status, Revision ID, Class Code, Header Type являются обязательными. Их назначения:

— Vendor ID – идентификатор производителя устройства, назначается разработчиком интерфейса;

— Device ID – идентификатор типа устройства;

— Command – предназначен для управления устройством, может быть как считан, так и записан;

— Status – необходим для определения состояния и свойств устройства;

— Revision ID – версия устройства, в случае, если для одного и того же устройства есть несколько версий требующих различного поведения;

— Class Code – необходим для определения основной функции устройства и интерфейс устройства;

— Header Type – определяет назначения последующих регистров.

Среди необязательных, но широко используемых регистров следует выделить регистры базовых адресных пространств Base Address Registers (BAR). Они определяют память устройства, с которой можно взаимодействовать. Таких регистров может быть от нуля до шести, соответственно устройство может иметь шесть независимых областей памяти. Каждый такой регистр может представлять область памяти с шириной адреса в 32 бита. Также имеется возможность объединения двух соседних регистров, если необходима ширина адреса до 64 бит. Размер памяти в каждом адресе определяется устройством и может быть до 4 байт.

Конфигурационное пространство интерфейса в памяти представлено на рисунке 1.

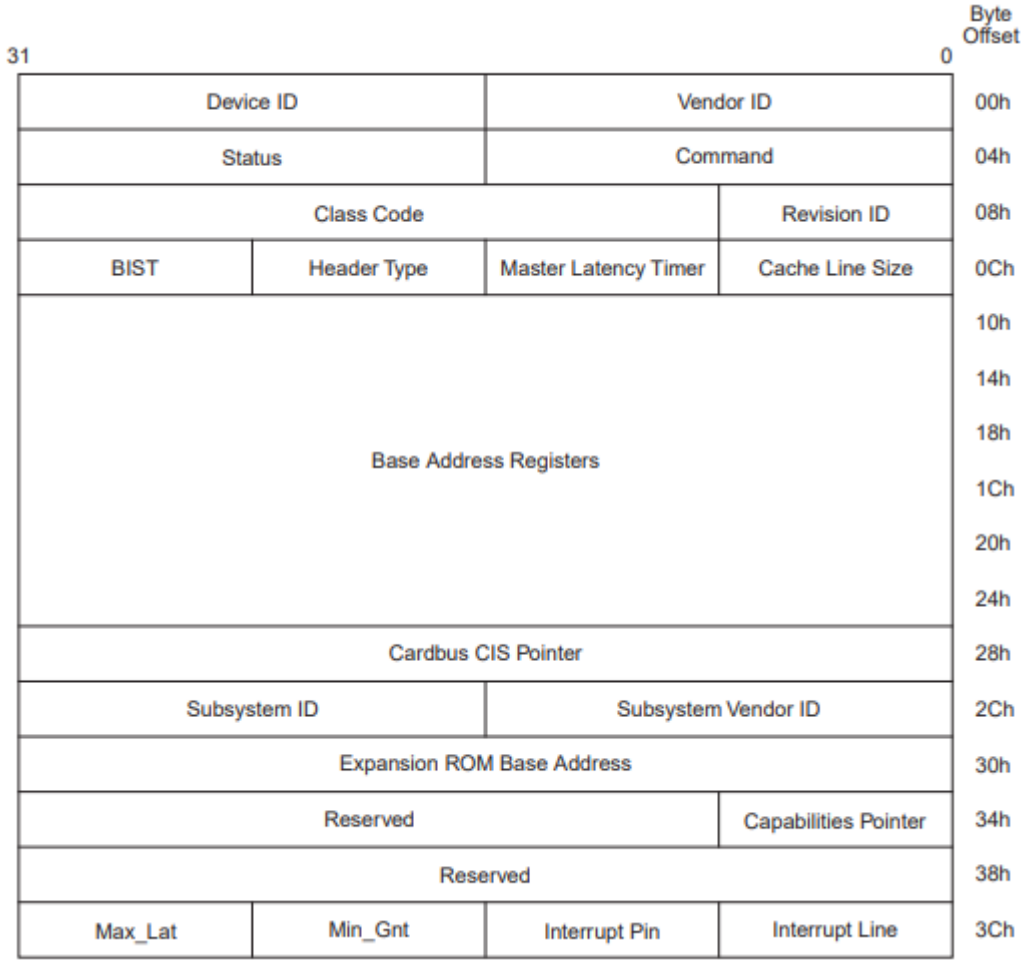

Рисунок 1 – Конфигурационное пространство PCI Express в памяти [10]

1.5 Постановка задачи

В рамках данного проекта требуется обеспечить высокоскоростной обмен данными между компьютером и ПЛИС с работающей в нем вычислительной платформой NITTA по средствам PCI Express. Для этого необходимо

1) На языках описания архитектуры разработать ряд модулей обеспечивающий получение данных по PCI Express и их хранение до надобности вычислителем.

2) Разработать протокол, позволяющий изменить действующий алгоритм, выполняемый NITTA в рамках текущей конфигурации ПЛИС. Это позволит выполнить расчет другого алгоритма на текущей микроархитектуре в случаях, когда сам алгоритм не сильно отличается от алгоритма, для которого изначально конфигурировалась вычислительная платформа.

3) Модифицировать имеющиеся блоки вычислителя таким образом, чтобы можно было остановить процесс вычисления и изменить данные текущего алгоритма, а также разработать модули, обеспечивающие получение, хранение и пересылку данных о новом алгоритме.

4) На стороне компьютера необходимо разработать драйвер для операционной системы на основе ядра Linux способный обеспечить передачу данных по PCI Express.

5) Реализовать в драйвере возможность загрузки алгоритма в вычислительную платформу NITTA.

ГЛАВА 2. АРХИТЕКТУРА ПРЕДЛАГАЕМОГО РЕШЕНИЯ

Ввиду того что задача состоит в создании интерфейса взаимодействия между двумя сильно различающимися устройствами, то и реализация взаимодействия на них будет сильно отличаться. Для ПЛИС на которой реализована вычислительная платформа NITTA необходимо разработать ряд схемотехнических модулей на языках описания оборудования таких как VHDL, Verilog и SystemVerilog. А именно блок передачи данных для исполняемого алгоритма и схемотехнический модуль, отвечающий за программирование NITTA (изменение исполняемого алгоритма на новый). Для компьютера или сервера под управлением операционной системы на основе ядра Linux предлагается разработать драйвер PCI Express, а также пользовательскую библиотеку, предназначенную для включения в САПР NITTA, чтобы разработчики или пользователи имели удобный способ взаимодействия с вычислителем.

На рисунке 2 изображена схема взаимодействия разрабатываемых элементов с процессором NITTA.

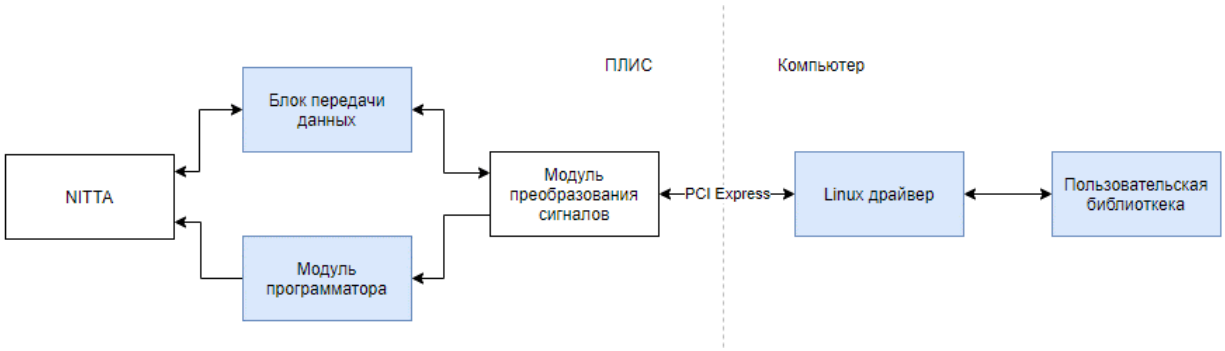

Рисунок 2 - Схема взаимодействия разрабатываемых элементов (выделены голубым) с процессором NITTA

2.1 Структура проекта NITTA

Для данного проекта нас интересует только та часть вычислительной платформы NITTA, которая реализуется на стороне ПЛИС. Она состоит из набора модулей, написанных на языке описания аппаратуры, а именно модули, отвечающие за:

— управление ходом исполнения алгоритма: `pu_simple_control`;

— различные операции по обработке данных: `pu_accum`, `pu_div`, `pu_fram`, `pu_multiplier`, `pu_shift`;

— передачу данных из ПЛИС и в нее: `pu_slave_spi`.

Эти модули являются частью связующего модуля, который подключен к модулю верхнего уровня, осуществляющему связь внутренней логики ПЛИС и ее физических контактов. Каждого вида модулей может быть использовано различное количество для текущей архитектуры ПЛИС или не быть вовсе. Также там имеются шина команд и шина данных.

Во время компиляции проекта в специальную область памяти помещается код требуемого алгоритма. После того как в модуль `pu_simple_control` придет сигнал `signal_cycle_start` разрешающий модулю начать исполнение на шину команд устанавливается первая команда алгоритма. После этого каждый такт процессора сдвиговый регистр увеличивает свое значение на единицу, а на шину команд (control_bus) устанавливается новая команда, соответствующая индексу регистра.

Тактирующий сигнал получается от частотного генератора, расположенного на плате, сигнал которого приводится к необходимой частоте при помощи модуля фазовой автоподстройки частоты (Phase Locked Loop, PLL).

Также на этапе компиляции другие модули, такие как pu_fram получают начальные значения в память и для всех модулей задаются параметры согласно требованиям алгоритма.

Для остальных модулей предусмотрено различное поведение в зависимости от значений на шине команд. Эти модули выполняют какие-либо операции, а результаты своих вычислений выставляют на шину данных (data_bus), которой могут пользоваться все модули. Загрузка и выгрузка данных осуществляется через интерфейс SPI, который подключен к модулю `pu_slave_spi`. Загружаемые данные рассчитаны только на один такт вычислительного процесса. На рисунке 3 изображена типовая схема соединения модулей в проекте NITTA.

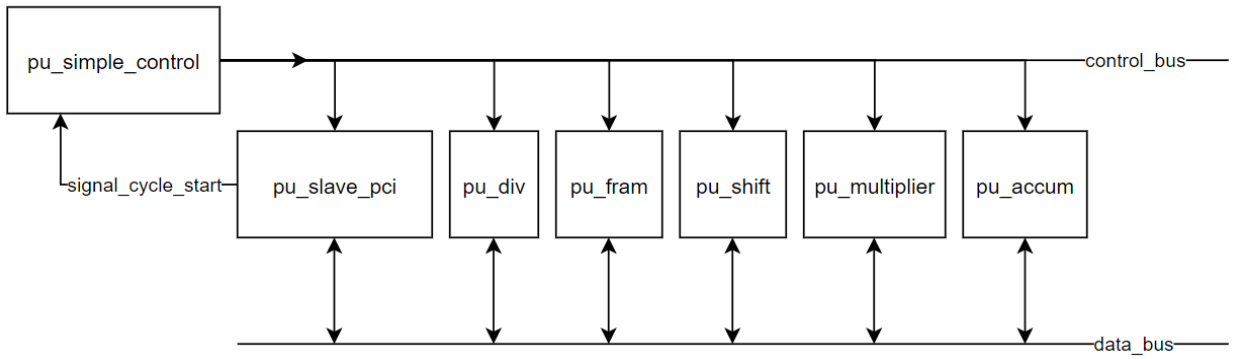

Рисунок 3 – схема работы вычислителя NITTA

2.2 Модификация вычислительной платформы NITTA

Для того чтобы обеспечить корректную работу вычислительной платформы NITTA с поддержкой PCI Express на ПЛИС необходимо изменить модуль верхнего уровня таким образом, чтобы он учитывал подключение контактов PCI Express интерфейса, настроить место подключения этих контактов к выводам микросхемы, а также необходим схемотехнический модуль преобразования сигналов с шины PCI Express в блоки данных, которые можно будет обрабатывать и хранить в памяти.

Для организации передачи данных необходимо разработать модуль, к которому NITTA сможет обращаться наряду с другими модулями. Который также будет принимать блоки данных от PCI Express. Ввиду того что PCI Express передает закодированные данные по своей шине, то необходим также модуль, не являющийся частью вычислителя NITTA занимающийся преобразованием сигналов шины в интерфейс, оперирующий блоками данных.

Также необходимо организовать надежное и эффективное хранение полученных данных до того момента, пока не придет запрос на их получение. Целесообразно чтобы количество вмещаемых данных было намного больше чем требуется на один вычислительный цикл чтобы уменьшить задержки, возникающие при обработке сигналов и передаче данных.

Для возможности изменения текущего алгоритма не меняя конфигурацию ПЛИС необходимо разработать модуль, который будет получать данные от пользователя через интерфейс PCI Express и модуль преобразования сигналов в данные и сохранять новый алгоритм в свою память. После подтверждения от пользователя, что алгоритм готов к загрузке и в момент простоя вычислителя (когда алгоритм не выполнятся и нет доступных данных для исполнения текущим алгоритмом) начнется смена текущего алгоритма блокируя обычный режим вычислителя.

Алгоритм состоит не только из группы наборов сигналов для модуля `pu_simple_control`, но и другие модули могут содержать некоторые данные, которые необходимо задать до начала выполнения алгоритма, такие как константы или начальное состояние памяти.

На шину данных необходимо выставлять по очереди команды нового алгоритма, а на шину команд специальные данные, сообщающие требуемым модулям, что текущие данные на шине предназначены для них. После конца прошивки начнет работу новый алгоритм. На рисунке 4 изображена диаграмма последовательности с предполагаемым процессом изменения алгоритма.

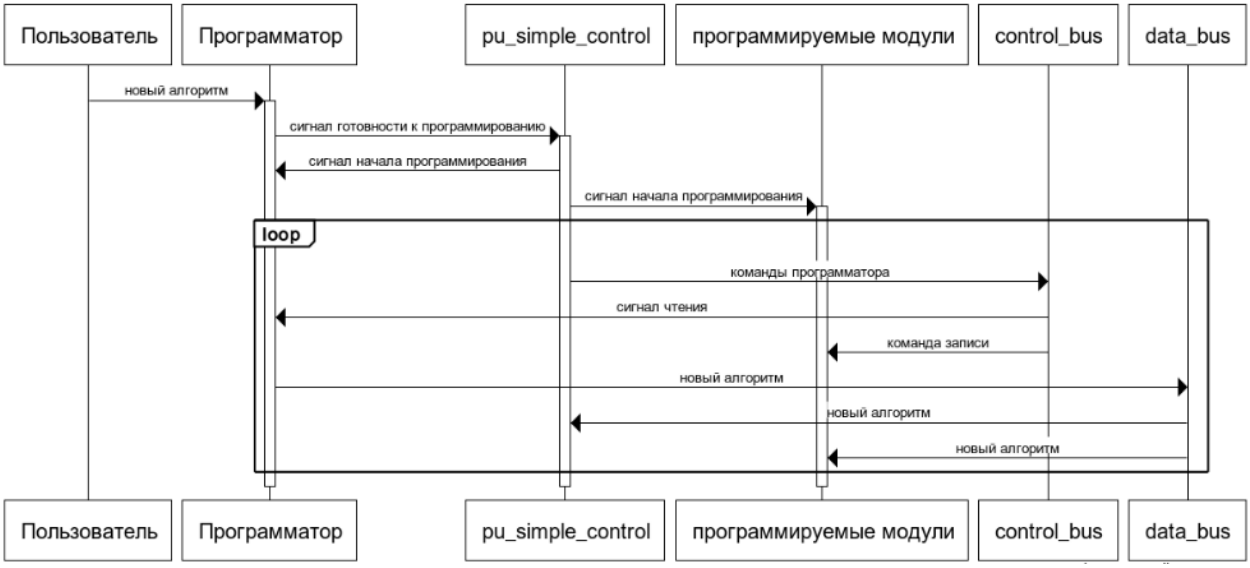

Рисунок 4 – Диаграмма последовательности для процесса изменения алгоритма вычислительной платформы

2.3 Организация памяти в разрабатываемых модулях

Организация хранения данных является важной частью любого проекта. Ввиду того, что проект состоит из нескольких частей, каждая из которых работает на собственной частоте и должна иметь доступ к данным другой, вопрос организации памяти и способов доступа к ней стоит наиболее остро.

Рассмотрим организацию памяти в модуле, содержащем входные данные для вычислителя.

Есть множество способов возможной организации памяти в ПЛИС подходящих для этой задачи. Рассмотрим детальнее следующие способы.

Так как необходимо контролировать корректность данных передаваемых процессору NITTA то необходимо исключить возможность одновременного доступа к памяти и процессора NITTA и модуля, который принимает данные от PCI Express. Поэтому одним из вариантов является выделение двух независимых областей памяти, в одну область должны записываться данные из PCI Express, когда буфер заполняется или когда пользователь сообщает о том, что необходимо начать вычисления. Данные из PCI Express начинают записываться в другую область памяти, а NITTA может брать данные из первой области с гарантией, что они не изменятся. Также необходимо считать количество записанных данных, чтобы не вылезти за границы доступных данных.

Записывать результирующие данные необходимо также в две независимые области памяти, которые также следует менять при заполнении и уведомлять об этом пользователя. На рисунке 5 изображена схема такой организации доступа к памяти.

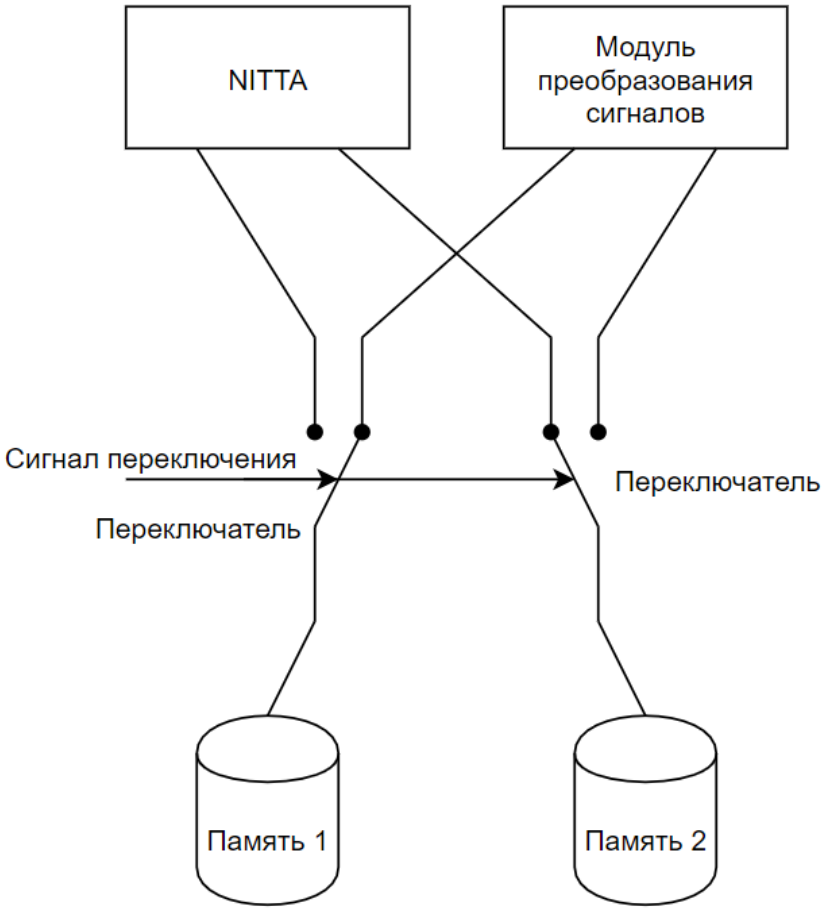

Рисунок 5 – Схема организации памяти с двумя независимыми областями

Второй рассматриваемый вариант это две очереди. Одна очередь в которую данные будут поступать из PCI Express и доставаться из нее процессором NITTA, другая же для того, чтобы в нее складывались результирующие данные, которые после могут быть получены пользователем через интерфейс PCI Express. Для этого необходимо однозначно контролировать процесс заполнения очередей, а также процесс получения данных из них.

Этот вариант более предпочтительный, так как не требует подсчет записанных данных на стороне пользователя и требует меньшего количества выделенных областей памяти. Пользователь же всегда сможет узнать сколько есть доступных данных с результатами чтобы забрать их и сколько есть свободного места в очереди, в которую помещаются данные необходимые алгоритму. На рисунке 6 изображена схема организации доступа к памяти с использованием очередей.

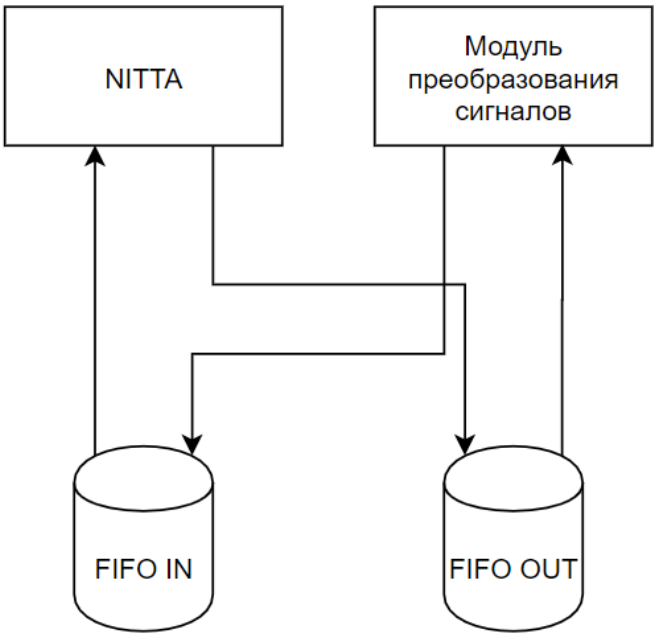

Рисунок 6 – Схема организации памяти с очередями

Для модуля необходимого для изменения вычислительного алгоритма предлагается использовать одну область памяти, в которую будет записываться новый алгоритм для микроархитектуры уже расположенной внутри ПЛИС, а процессор NITTA сможет только читать эту область. Необходимо обеспечить контроль за доступ к этой памяти. А именно запрещать ее изменение, когда вычислительная платформа программируется и изменять сигнал о готовности алгоритма к прошивке, когда хоть один блок данных изменился.

Память в ПЛИС может синтезироваться как сложная регистровая схема или как специализированный блок оперативного запоминающего устройства Random-access memory (RAM). Второй вариант более предпочтительный, так как в ПЛИС зачастую заложены участки с такой структурой. Это позволяет ускорить процесс синтеза схемотехнического описания микроархитектуры и использовать меньше функциональных элементов, что обеспечит лучшее быстродействие ПЛИС, а в случае надобности использовать эти элементы для других целей. На рисунке 7 изображена схема организации общей памяти с переключаемым доступом.

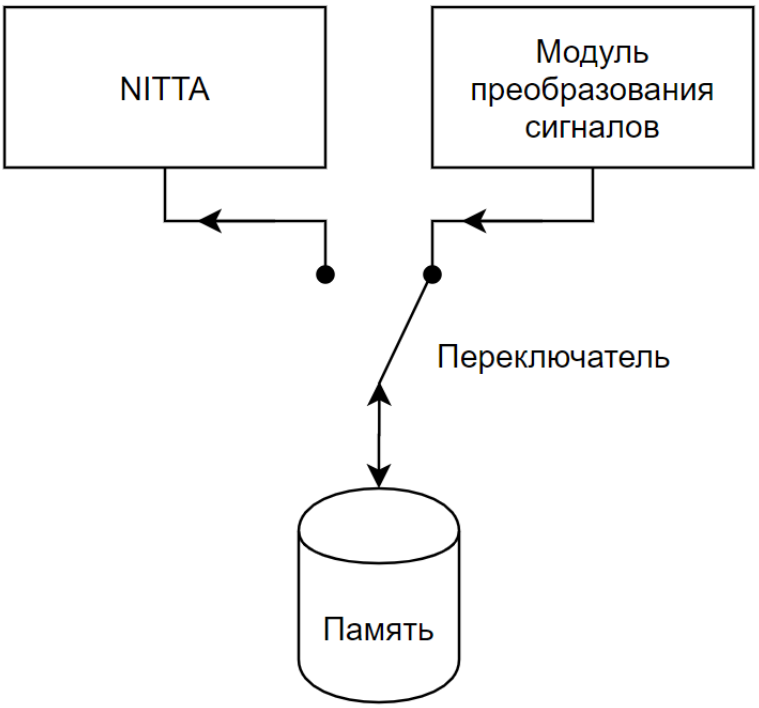

Рисунок 7 – Схема организации памяти с переключаемым доступом

Учитывая вышесказанное нам необходимо добиться того чтобы вся описываемая память синтезировалась в RAM. Какие с этим могут быть трудности будет рассмотрено в следующей главе.

2.4 Особенности реализации драйвера Linux

Драйвера в операционных системах на базе ядра Linux могут быть в пользовательском пространстве или пространстве ядра.

Особенности драйверов пользовательского пространства:

— простота отладки;

— возможность использования сторонних библиотек;

— авария не влияет на состояние ядра;

Особенности драйверов пространства ядра:

— поддержка только встроенных библиотек Linux;

— при аварии возможно повреждение ядра;

— более высокая скорость работы;

— возможность контроля доступа к устройству;

В рамках данной работы разрабатывается второй вариант, который также будет называться модулем ядра. Выбранный вариант обеспечивает большее быстродействие системы и предоставляет больше возможностей по контролю протекающих процессов, что особенно важно, т.к. NITTA является системой работающей в режиме реального времени.

Также драйвера необходимы для обеспечения контролируемого доступа к устройству пользователями из пользовательского пространства. Для этих целей драйвером в системе создается файл обращаясь к которому пользователь может взаимодействовать с устройством, потому его также можно называть устройством. Существуют такие файлы-устройства двух типов: символьные и блочные.

Символьные устройства предоставляют пользователю ряд функций, которые обрабатывают данные пользователя в пространстве ядра. Имеют стандартные функции чтения, записи, открытия и закрытия устройства, а также общую функцию управления вводом/выводом ioctl. Обрабатывают данные получаемые от пользователя последовательно.

Блочные устройства предназначены в основном для обеспечения доступа к памяти. Они оперируют блоками данных фиксированной длины и зачастую размер такого блока слишком большой для нашей задачи.

Поэтому в данном проекте драйвером создается символьное устройство, а большинство функций, обеспечивающих взаимодействие с NITTA вызываются через ioctl.

Другой задачей драйвера является обеспечение взаимодействия ядра операционной системы со сторонним устройством по средствам PCI Express. Правильно сконфигурированное устройство отображается в память ядра своим конфигурационным пространством. Драйвер должен определить устройство, для которого он предназначен по Vendor ID и Device ID. После чего необходимо выделить память, которая будет ассоциирована с памятью физического устройства для каждого базового адресного пространства, которое используется в устройстве. После этого создать символьное устройство.

Для данного проекта оптимальным будет использование двух независимых адресных пространств. BAR0 использовать для осуществления передачи данных для текущего алгоритма, а BAR1 для передачи и подтверждения алгоритма, которым следует перепрограммировать вычислитель.

Драйвер должен через функции символьного устройства предоставлять соблюдение протокола взаимодействия с вычислительной платформой NITTA. Также необходимо учесть возможность удаления модуля ядра из пространства ядра. Для этого перед удалением модуля ядра необходимо освободить выделенную память и удалить символьное устройство.

Для удобства взаимодействия пользователя с драйвером необходимо разработать пользовательскую библиотеку, где все функции драйвера будут просты в использовании для конечного пользователя или разработчика САПР

NITTA. На рисунке 8 изображена диаграмма последовательности для драйвера Linux.

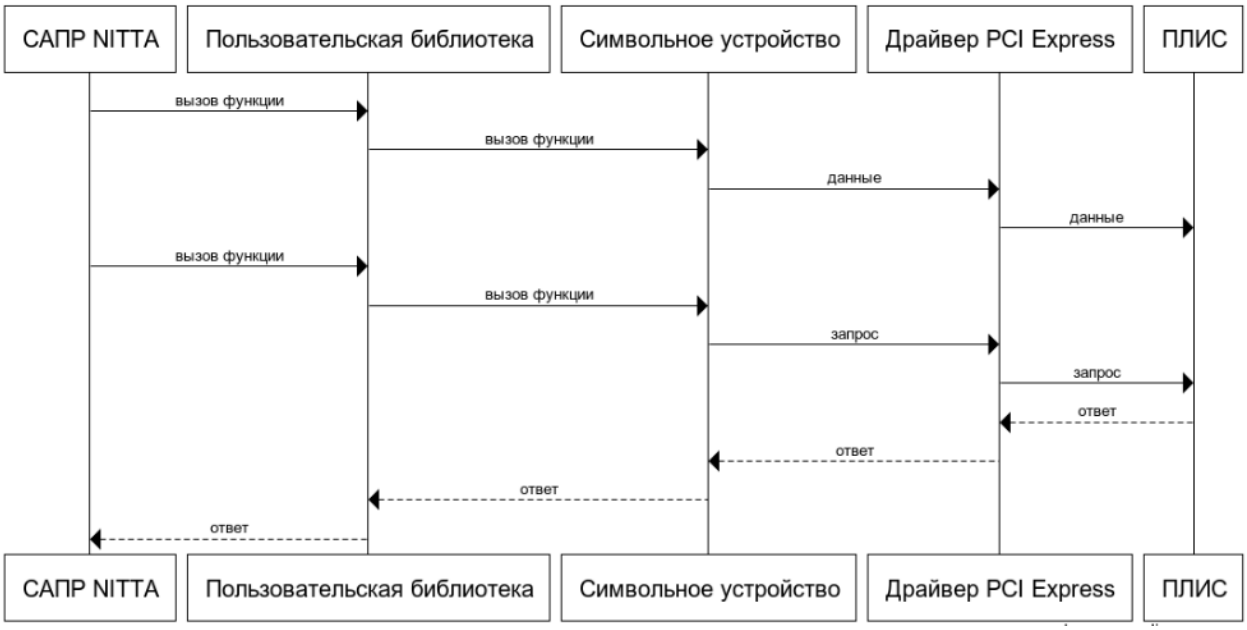

Рисунок 8 – Диаграмма последовательности для драйвера Linux

Пользователю необходимо отправлять данные в ПЛИС, которая сообщает удачно ли загружены данные и в каком количестве. После чего вычислительная платформа производит вычисления, забирая данные из очереди и помещает выходные данные в другую очередь. Очередь с результатами должна периодически или перед запросом на получение результата опрашиваться пользователем.

2.5 Используемые инструменты

Проект состоит из двух частей. Одна из них работает на стороне ПЛИС и написана на языках описания аппаратуры таких как VHDL, Verilog и SystemVerilog, а именно набор схемотехнических модулей, отвечающих за организацию памяти, взаимодействие PCI Express и NITTA интерфейсов и организации передачи данных. Вторая же работает на компьютере под управлением операционной системы на основе ядра Linux и является одним из модулей ядра содержащим часть, отвечающую за взаимодействие операционной системы с PCI Express и часть, отвечающую за взаимодействие пользователя с пространством ядра.

Для работы над частью, работающей на ПЛИС применялась система автоматизированного проектирования от компании Intel Quartus Prime для описания и компиляции схемотехнических модулей на языках описания архитектуры VHDL, Verilog и SystemVerilog. Для удобства соединения многошинных модулей по средствам интерфейсов использовалось программное обеспечение Platform Designer, для удобства визуализации разработанного модуля использовался Register Transfer Level Viewer (RTL Viewer). Для отладки проекта использовалось программное обеспечение моделирующее поведение ПЛИС на компьютере ModelSim, для назначения сигналов модуля верхнего уровня контактам на ПЛИС использовался Pin Planner, для прошивания ПЛИС использовался программный модуль Intel Quartus Prime Programmer.

Для разработки части проекта отвечающей за взаимодействие компьютера с ПЛИС в качестве языка программирования применялся C90, также были использованы заголовочные файлы ядра Linux. Программа Make и gcc для компиляции модуля ядра.

2.6 Выводы

В рамках данной главы были рассмотрены следующие этапы разработки:

1) Была проанализирована структура вычислительной платформы NITTA. Обозначены этапы взаимодействия пользователя с вычислителем подлежащие разработке в рамках проекта.

2) Выявлены места, которые требуют изменения для возможности внедрения нового интерфейса передачи данных.

3) Произведен выбор способа организации памяти в проекте для разрабатываемых модулей и правил доступа к ней.

4) Выполнен выбор с обоснованием типа драйвера операционной системы Linux.

5) Описан способ взаимодействия пользователя с драйвером по средствам символьного устройства.

6) Описаны инструменты, использованные в процессе работы над проектом.

ГЛАВА 3. РЕАЛИЗАЦИЯ ИНТЕРФЕЙСА В ПЛИС

3.1 Преобразование входных сигналов PCI Express

Преобразование сигналов с контактов интерфейса PCI Express в блоки данных является большой и сложной задачей, но компанией Intel был разработан сложный функциональный блок IP Compiler for PCI Express, предназначенный для преобразования входных сигналов в интерфейс данных Avalon Memory-Mapped. На рисунке 9 изображены настройки этого функционального блока, используемого в проекте [13].

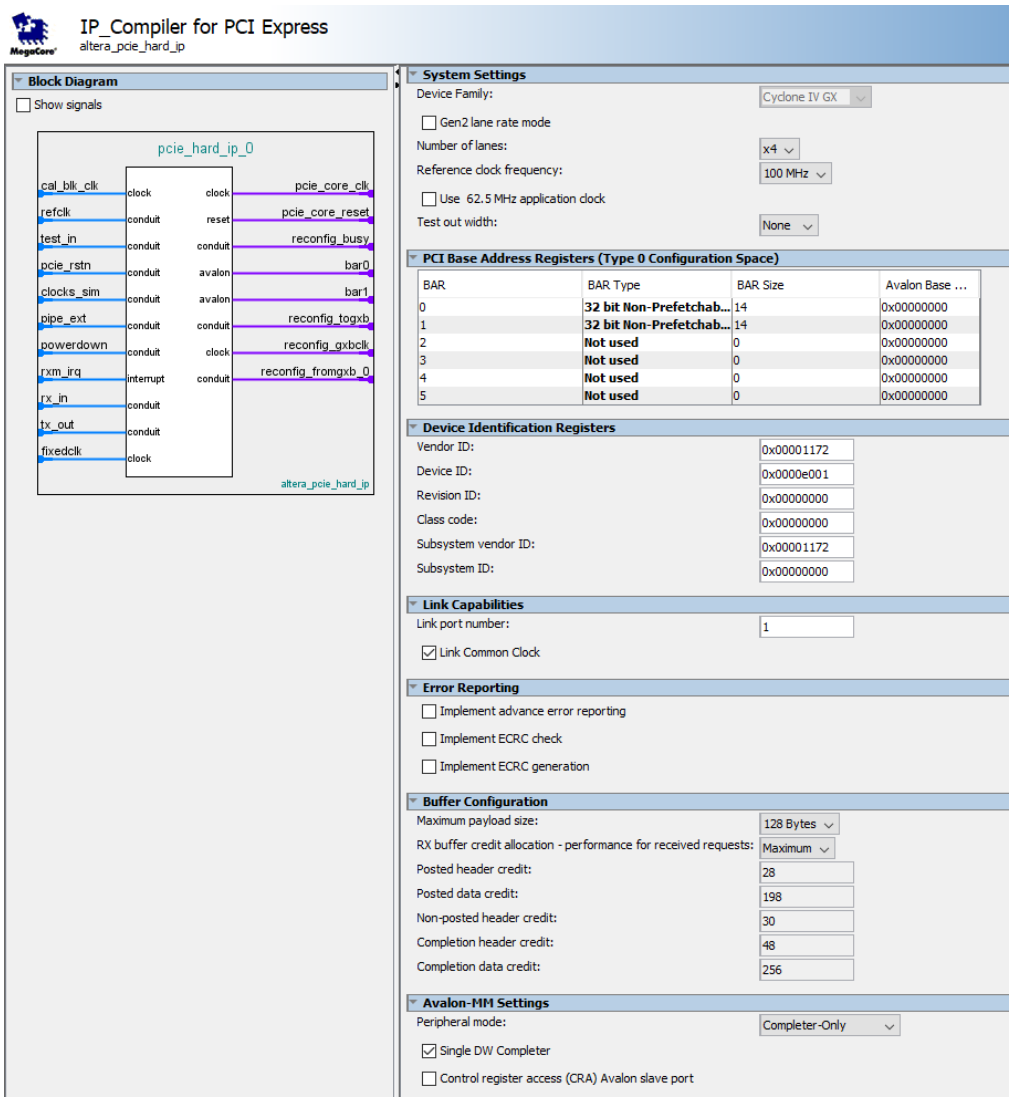

Рисунок 9 - настройки IP Compiler for PCI Express

3.2 Модель передачи данных

Для текущего проекта был разработан схемотехнический модуль `pu_slave_pci` отвечающий за взаимодействие NITTA с выше описанным модулем. Для этого разработанный модуль реализует Avalon Memory Mapped интерфейс. Рассмотрим его шины детальнее:

— `address[31:0]` – адрес шириной до 32 бит по которому идет чтение или запись;

— `read` – сигнал того, что сейчас идет чтение;

— `waitrequest` – запрос ожидания, выставляется если модуль не может сейчас ответить на запрос Master;

— `write` – сигнал того, что сейчас идет запись;

— `readdatavalid` – подтверждает, что на шине `readdata` в данный момент установлены действительные данные;

— `readdata[31:0]` – шина шириной 32 бита для данных которые считывает пользователь;

— `writedata[31:0]` – шина шириной 32 бита на которой располагаются данные, которые пришли в ПЛИС от пользователя;

— `byteenable[3:0]` – применяется если необходимо взаимодействие только с определенными байтами на шинах `readdata` и `writedata`.

Также IP Compiler for PCI Express предполагает использование частоты на которой работает PCI Express интерфейс, а именно 100 МГц.

На рисунке 10 изображена диаграмма сигналов Avalon Memory Mapped интерфейса.

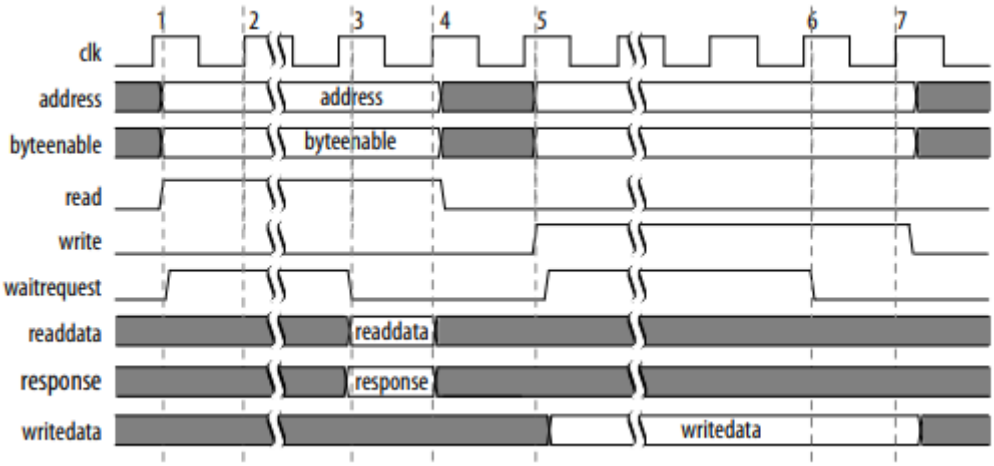

Рисунок 10 – Диаграмма сигналов Avalon Memory Mapped

Теперь рассмотрим интерфейс, используемый в NITTA.

Для NITTA также важным показателем является частота работы процессора, кроме нее интерфейс представлен следующими сигналами и шинами:

— `signal_wr` – устанавливается если NITTA хочет записать данные;

— `data_in[31:0]` – шина шириной 32 бита с данными которые хочет записать NITTA;

— `signal_oe` – сигнал устанавливается если NITTA хочет считать данные;

— `data_out[31:0]` – шина шириной 32 бита, на которую следует устанавливать данные;

— `flag_stop` – сигнал передаваемый процессору NITTA о том, что созданы условия для корректной работы.

Также есть особенность в работе сигналов `signal_wr` и `signal_oe`, если установлены одновременно оба сигнала, то следует передать процессору NITTA следующий набор данных, если же выставлен в единицу только `signal_oe`, то следует передать процессору NITTA данные которые уже передавались в прошлый раз.

3.3 Организация передачи данных в NITTA

Рассмотрим подробнее как работает модуль pu_slave_pci. Он обеспечивает обработку и хранение данных пришедших от пользователя через PCI Express относящихся только к нулевому базовому адресному пространству. А также отдает данные вычислительному циклу, когда это требуется.

Ввиду того, что для хранения данных используются циклические очереди, пользователь на этапе отправки данных не знает адреса куда следует поместить текущие данные, поэтому распределением данных по адресам памяти занимается схемотехнический модуль очереди. А пользователь отправляет все данные по одному и тому же адресу.

Соответственно другие адреса могут быть выделены под различные команды, например, очистка памяти и вызов сброса вычислителя программным образом.

Получение выходных данных осуществляется операцией чтения по тому же адресу. Для того, чтобы пользователь мог определить то, сколько данных он еще может передать до того, как заполнится очередь для входных данных или узнать сколько данных доступно в очереди с результатами работы алгоритма, пользователь может выполнить операцию чтения по выделенным для этих целей адресам.

Для реализации интерфейса NITTA, а именно особенностей чтения был добавлен буфер на чтение из очереди с входными данными. Сами же процессы чтения осуществляются по специальным сигналам разрешения, которые формируются под конкретные возможные ситуации.

Также необходимо ограничивать длительность сигналов чтения и записи одним тактом для того, чтобы указатель не перепрыгивал на последующие значения раньше, чем это необходимо.

Сигнал сообщающий процессору NITTA о том, что можно приступать к выполнению алгоритма состоит из двух сигналов, каждый из которых должен сообщать логическую единицу:

— в очереди с начальными данными достаточно данных для одного вычислительного цикла;

— в очереди с результатами достаточно места для записи результатов вычислительного цикла.

Теперь рассмотрим особенности работы очередей. Из-за того, что NITTA и PCI Express работают на различных тактовых частотах необходимо исключить ситуации, когда значение адреса для записи или чтения не будут стабильными. Для того чтобы избежать возможных метастабильных состояний принято использовать счетчик Грея [14, 15].

Очередь может корректно передать данные только в такт, следующий за тактом обращения к ней. Для PCI Express это не является проблемой т.к. длительность одной операции чтения или записи длится дольше одного такта. Процессор NITTA же должен учитывать это и посылать запрос к данным за один такт до их фактической надобности, но это не является проблемой т.к. все операции на весь вычислительный цикл определяются заранее.

На рисунках 11 и 12 изображены блок-схемы работы разработанного модуля.

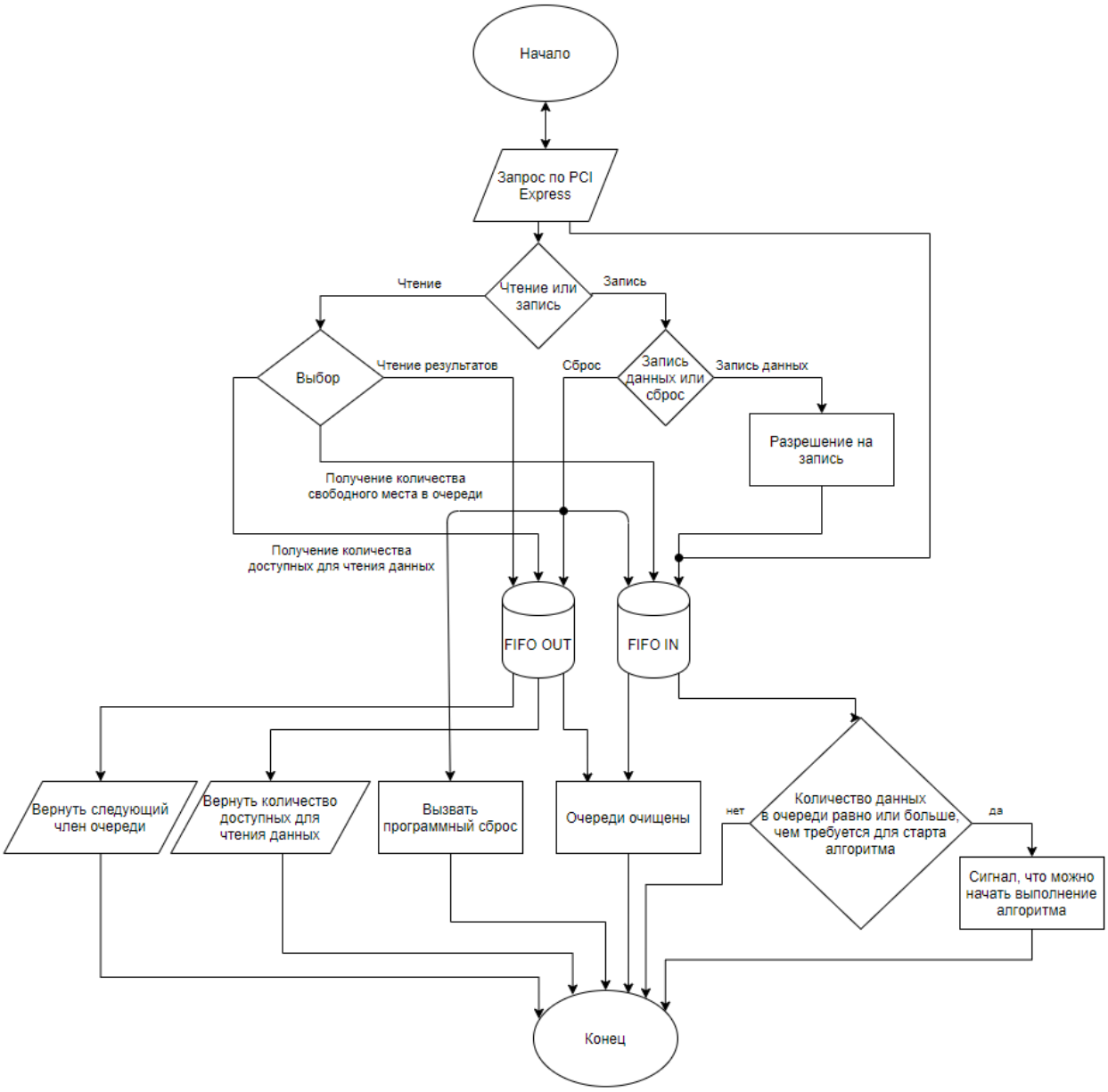

Рисунок 11 – Блок-схема взаимодействия модуля pu_slave_pci с IP Compiler for PCI Express

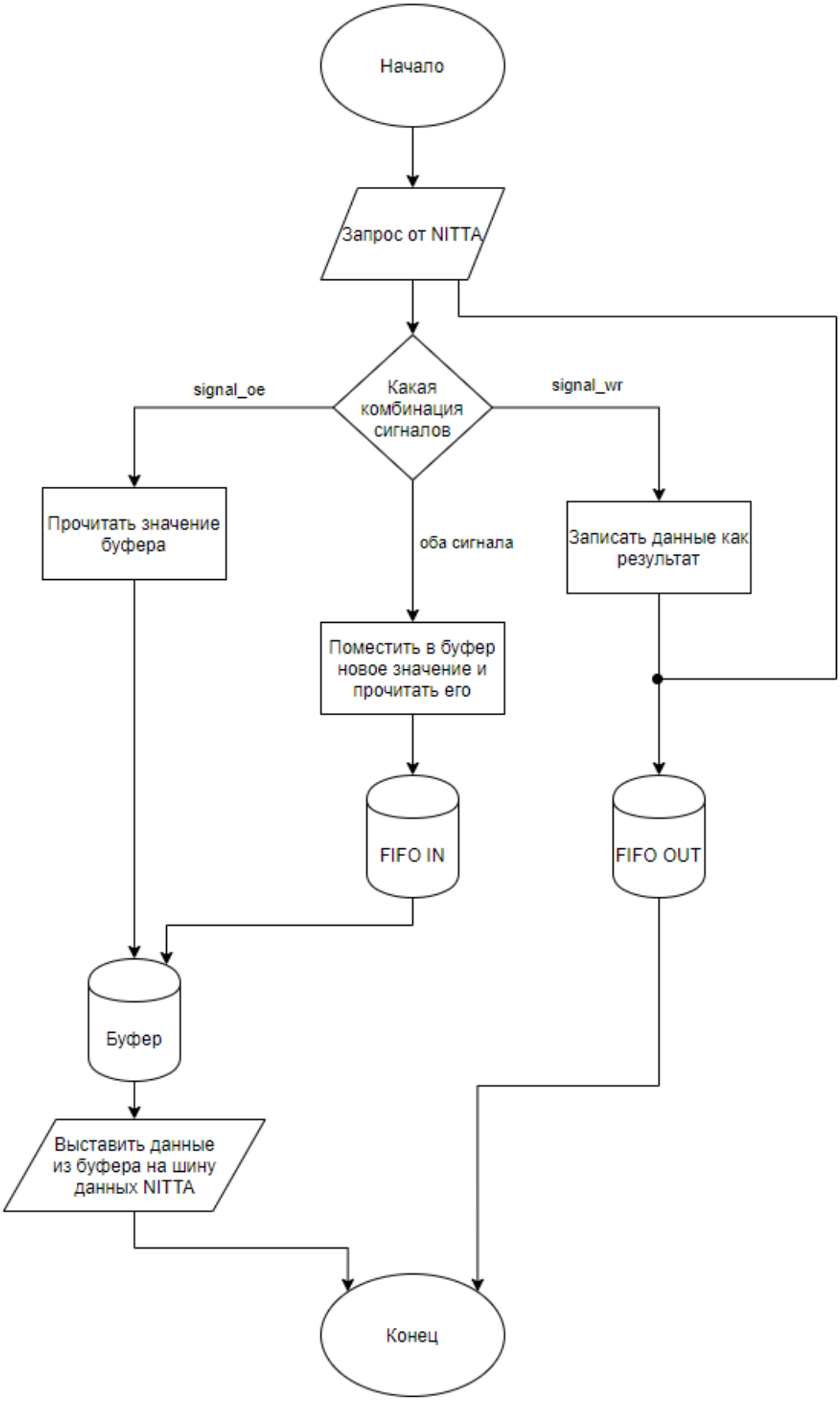

Рисунок 12 – Блок-схема взаимодействия модуля pu_slave_pci с NITTA

3.4 Конфигурация NITTA для загрузки алгоритмов

Для того чтобы передавать процессору NITTA новый алгоритм исполнения необходимо модифицировать модули, которые имеют какие-либо области памяти или параметры, зависящие от алгоритма.

Наибольшим изменениям необходимо подвергнуть модуль pu_simple_control. Данный модуль должен иметь еще одну область памяти с командами, назовем ее памятью программатора, отвечающими за изменение алгоритма. Смена текущей команды на следующую осуществляется сдвиговый регистр.

Когда приходит команда что следует начать перепрограммирование и процессор находится в состоянии простоя выставляется сигнал о том, что идет перепрограммирование и начинается исполнение команд программатора.

Нулевая команда отвечает за получение количества команд в новом алгоритме. Первая за получение и запись новых команд нового алгоритма в память, старого алгоритма. Далее каждая четная команда получает количество элементов в следующем модуле, данные которого принадлежат изменению.

Длительность всех четных команд длится один такт, а длительность нечетных согласно числу, полученному во время исполнения предыдущей команды.

3.5 Схемотехнический модуль передачи алгоритма в NITTA

Схемотехнический модуль необходимый для изменения исполняемого алгоритма `pu_slave_pci_programmer` принимает сигналы интерфейса Avalon Memory-Mapped от второго адресного пространства IP Compiler for PCI Express.

Пользователь должен знать о текущей конфигурации ПЛИС для создания корректного нового алгоритма. Этот алгоритм состоит из команд, которые необходимо записать в память модуля `pu_simple_control`, а также данные для модулей, обладающих буферами, которые необходимо заполнить до выполнения алгоритма. Для каждого такого модуля выделяется область в памяти, в которую пользователь записывает данные соответствующие каждому модулю. Также необходимо записать в специальные регистры количество записанных данных для каждого модуля. После этого необходимо отправить сигнал, сообщающий о завершении загрузки прошивки.

Сигнал завершения работы необходимо провести из области тактового сигнала PCI Express в область тактового сигнала процессора NITTA. Теперь до тех пор, пока данный алгоритм не запрограммирует вычислительную платформу или пользователь не захочет изменить алгоритм модуль pu_simple_control будет получать сигнал о готовности к перепрограммированию. Во время перепрограммирования данные пришедшие от пользователя не повлияют на данные программируемого алгоритма.

Новый алгоритм записывается в двухпортовую память, где один порт тактируется сигналом PCI Express, а второй частотой NITTA. Контроль за одновременным доступом к памяти осуществляется алгоритмически. Память может читаться NITTA только в том случае, если пользователь подтвердил алгоритм и больше его не меняет.

Когда от NITTA приходит сигнал чтения на шину данных выставляется сначала информация о том, сколько данных будет содержать следующий блок. В это же время осуществляется чтение первого блока из памяти так как на то чтобы выставить данные на шину требуется один такт. Таким образом передаются все данные, после чего алгоритм помечается загруженным и не требующим больше программирования.

Любое изменение алгоритма сбрасывает значение о том, что этот алгоритм уже загружен.

На рисунках 13 и 14 изображены блок-схемы работы схемотехнического модуля загрузки нового алгоритма для NITTA.

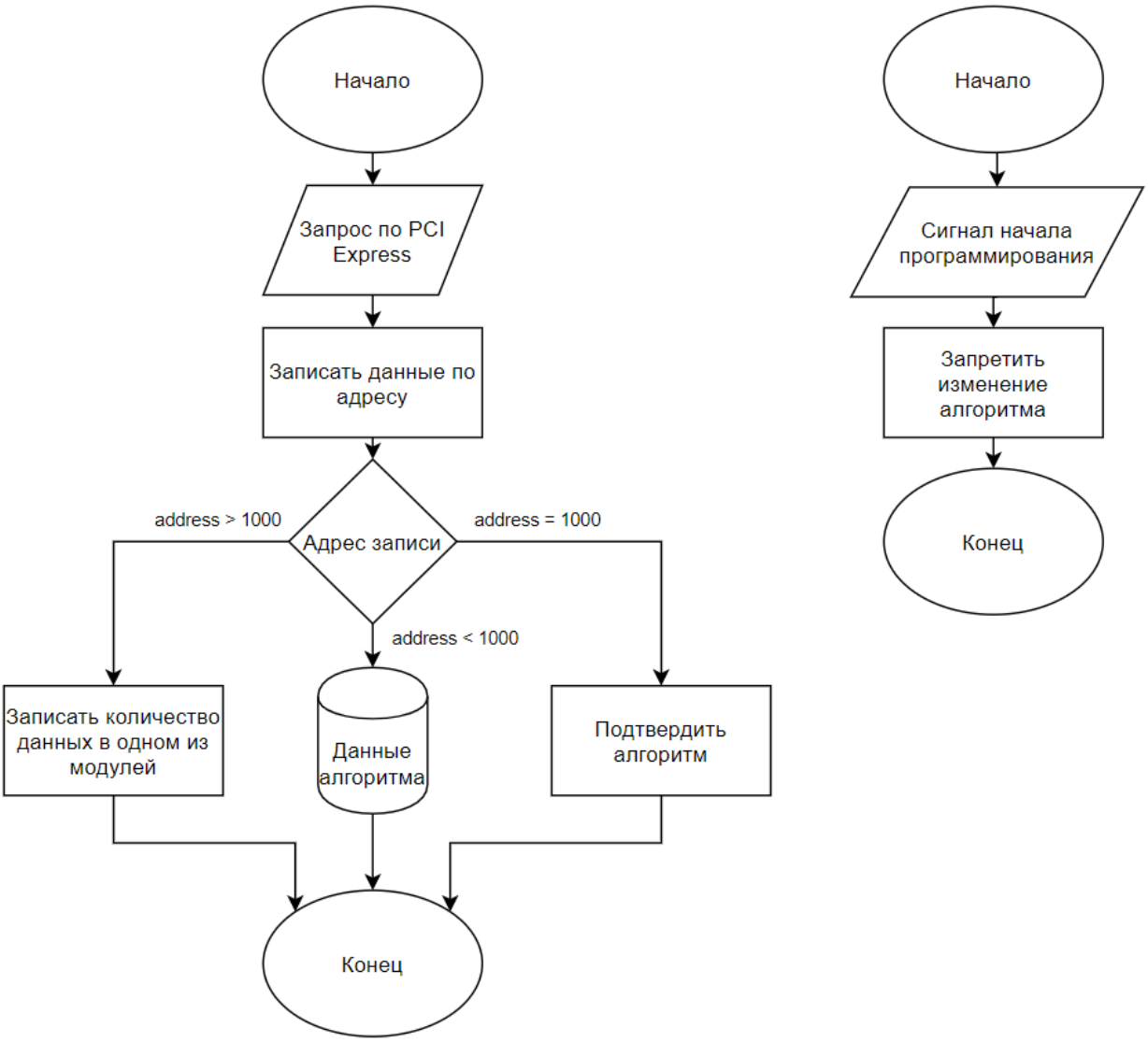

Рисунок 13 – Блок-схема модуля pu_slave_pci_programmer для получения алгоритма

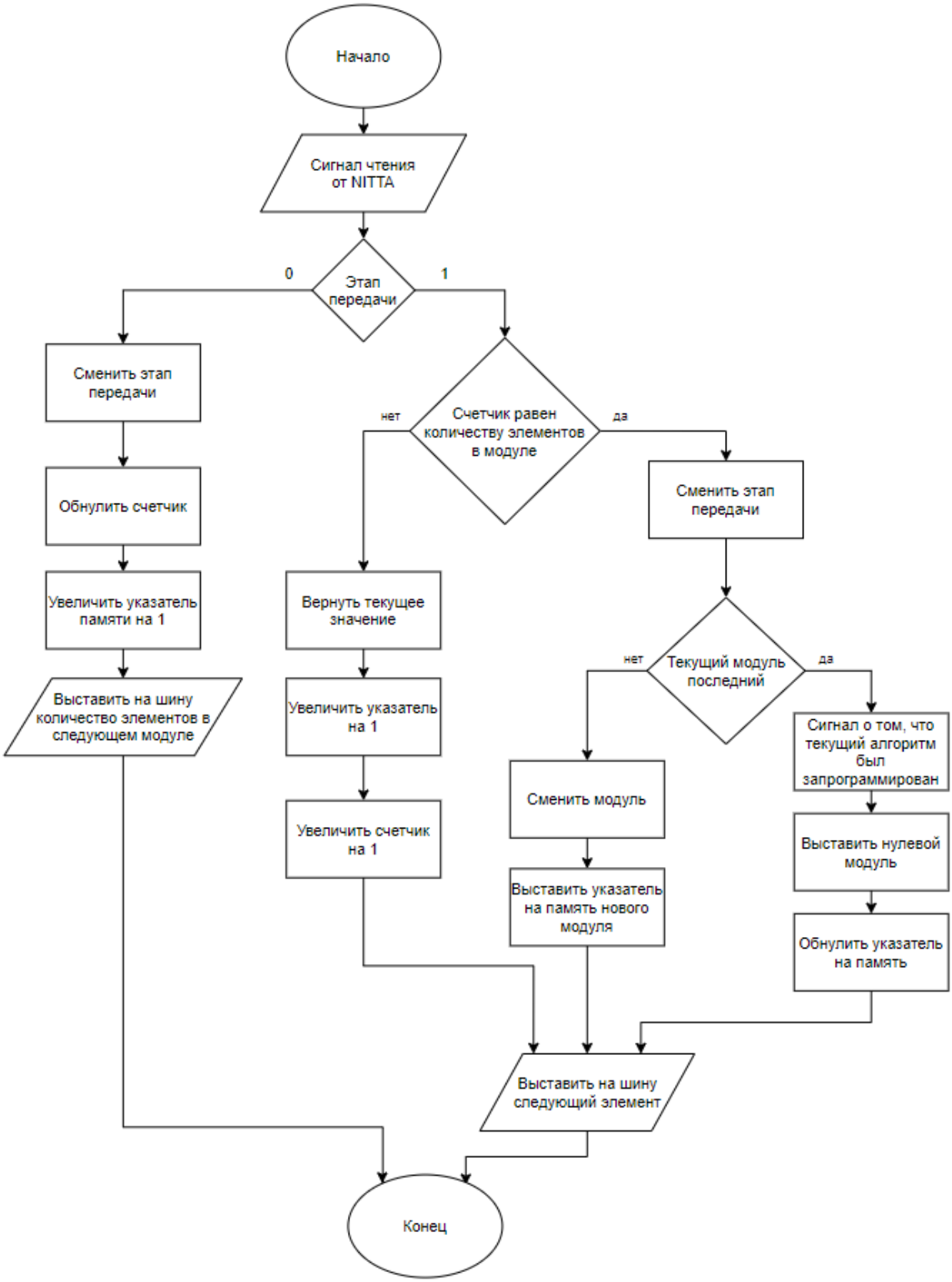

Рисунок 14 – Блок-схема модуля pu_slave_pci_programmer для загрузки алгоритма в NITTA

3.6 Загрузка проекта ПЛИС

IP Compiler for PCI Express имеет большое количество сигналов. Нахождение и использование сигналов может занимать большое количество времени. Для упрощения соединения модулей компанией Intel разработано программное обеспечение Platform Designer, которое позволяет соединять различные модули используя интерфейсы. Это упрощает создание и поддержку сложных систем. В данном проекте таким образом соединяются модули IP Compiler for PCI Express, PLL, main_net и главный тактовый сигнал. На рисунке 15 изображена схема соединения этих модулей в Platform Designer.

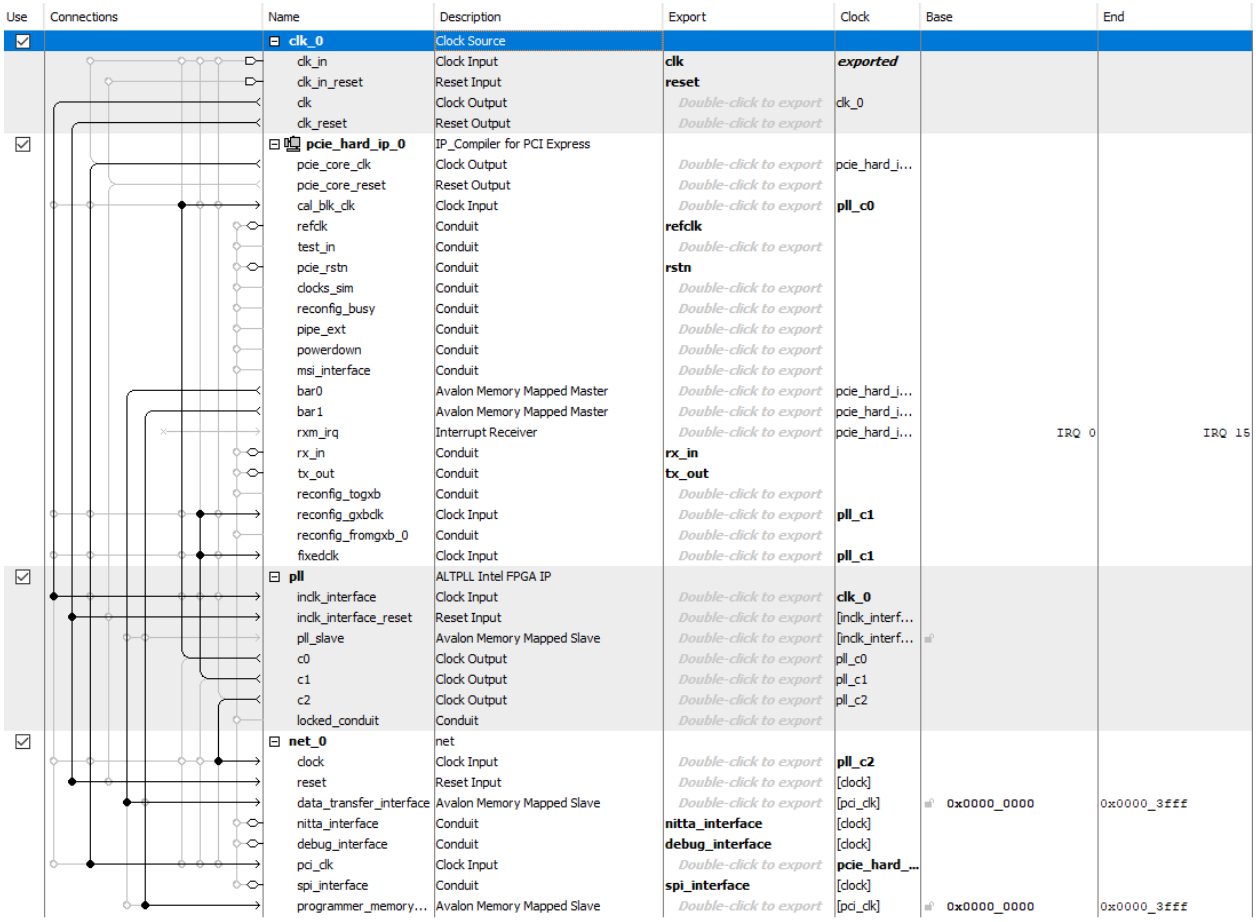

Рисунок 15 – схема соединения модулей в Platform Designer

Platform Designer также создает модуль, только его расширение *.qsys, т.к. в проекте такой модуль всего один, то и будем называть его qsys модуль.

Этот модуль необходимо подключить к модулю верхнего уровня, его код приведен в приложении А в файле top.vhd. В нем только подключаются сигналы с платы к модулю qsys [16].

Для загрузки проекта внутрь ПЛИС необходимо выбрать версию устройства, а также включить функцию генерацию файла прошивки с расширением *.pof, указав чип памяти, используемый на плате. После чего необходимо назначить сигналы модуля верхнего уровня конкретным контактам на плате, используя программное обеспечение Pin Planner. На рисунке 16 закрашенными изображены контакты, подключенные к top.vhd, а на рисунке 17 то, какие выходные сигналы модуля верхнего уровня подключены к каким именно контактам ПЛИС, а также тип подключения.

Далее необходимо:

1) Выполнить компиляцию проекта.
2) подключить к плате с ПЛИС и компьютеру с проектом JTAG кабель с USB-Blaster Altera.
3) Установить JTAG драйвер на компьютер.
4) Открыть Programmer.
5) Выбрать режим Active Serial Programming.
6) В качестве устройства выбрать USB-Blaster.
7) Добавить созданный *.pof файл.
8) Включить для файла поле Program/Configure
9) Нажать кнопку Start.
10) Дождаться завершения прошивки.
11) Перезагрузить плату.

После чего плата начнет работать с новой конфигурацией.

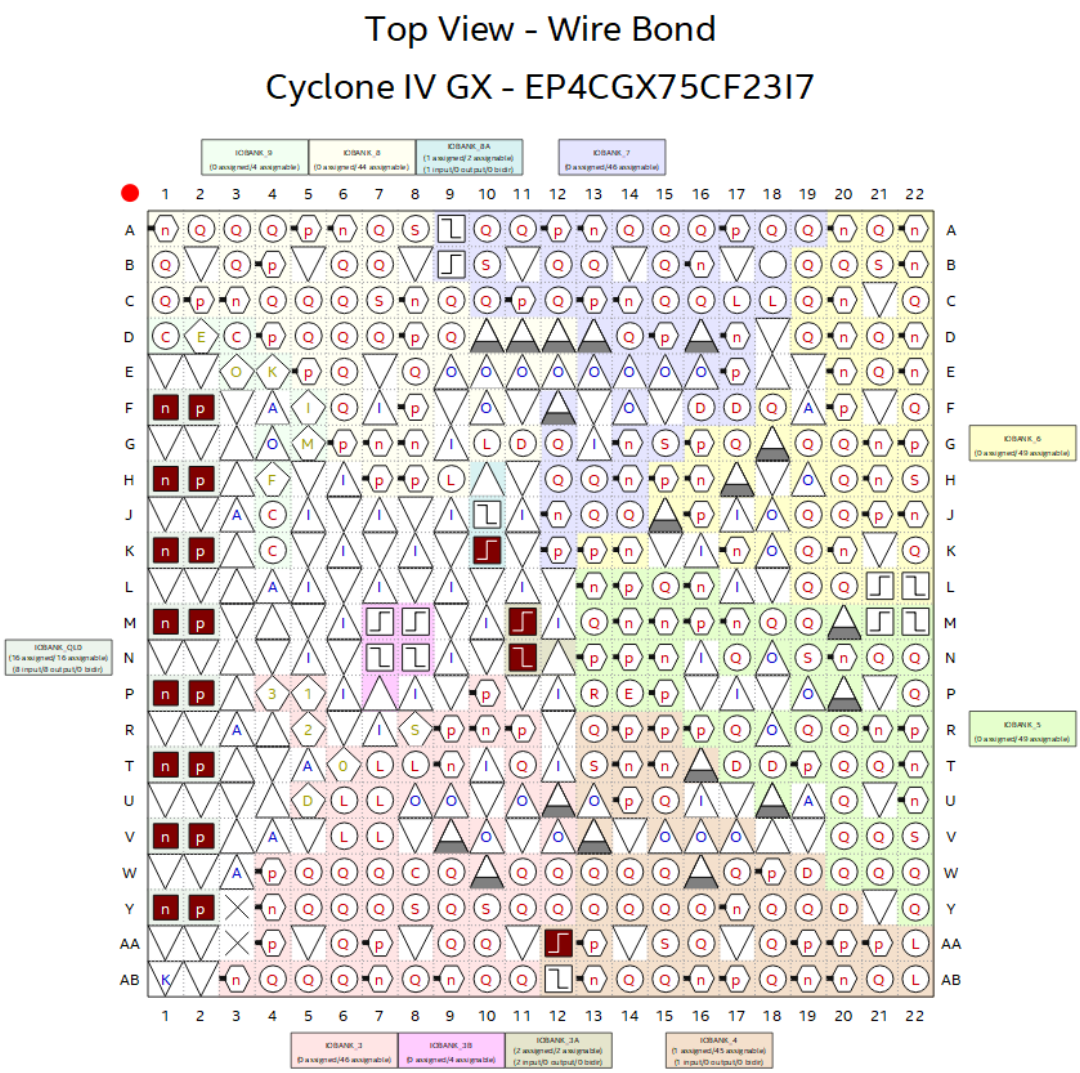

Рисунок 16 – Подключенные контакты к ПЛИС

Node Name	Direction	Location	I/O Bank	VREF Group	Fitter Location	I/O Standard	Current Strength	Differential Pair
HCLK	Input	PIN_AA12	4	B4_N2	PIN_AA12	2.5 V	16mA (default)	
PCIE_REFCLK	Input	PIN_M11	3A	B3_N0	PIN_M11	HCSL	Maximu...fault)	PCIE_REFCLK(n)
PCIE_REFCLK(n)	Input	PIN_N11	3A	B3_N0	PIN_N11	HCSL	Maximu...fault)	PCIE_REFCLK
PCIE_RX0	Input	PIN_Y2	QL0		PIN_Y2	1.5-V PCML	Maximu...fault)	PCIE_RX0(n)
PCIE_RX0(n)	Input	PIN_Y1	QL0		PIN_Y1	1.5-V PCML	Maximu...fault)	PCIE_RX0
PCIE_RX1	Input	PIN_T2	QL0		PIN_T2	1.5-V PCML	Maximu...fault)	PCIE_RX1(n)
PCIE_RX1(n)	Input	PIN_T1	QL0		PIN_T1	1.5-V PCML	Maximu...fault)	PCIE_RX1
PCIE_RX2	Input	PIN_M2	QL0		PIN_M2	1.5-V PCML	Maximu...fault)	PCIE_RX2(n)
PCIE_RX2(n)	Input	PIN_M1	QL0		PIN_M1	1.5-V PCML	Maximu...fault)	PCIE_RX2
PCIE_RX3	Input	PIN_H2	QL0		PIN_H2	1.5-V PCML	Maximu...fault)	PCIE_RX3(n)
PCIE_RX3(n)	Input	PIN_H1	QL0		PIN_H1	1.5-V PCML	Maximu...fault)	PCIE_RX3
PCIE_TX0	Output	PIN_V2	QL0		PIN_V2	1.5-V PCML	Maximu...fault)	PCIE_TX0(n)
PCIE_TX0(n)	Output	PIN_V1	QL0		PIN_V1	1.5-V PCML	Maximu...fault)	PCIE_TX0
PCIE_TX1	Output	PIN_P2	QL0		PIN_P2	1.5-V PCML	Maximu...fault)	PCIE_TX1(n)
PCIE_TX1(n)	Output	PIN_P1	QL0		PIN_P1	1.5-V PCML	Maximu...fault)	PCIE_TX1
PCIE_TX2	Output	PIN_K2	QL0		PIN_K2	1.5-V PCML	Maximu...fault)	PCIE_TX2(n)
PCIE_TX2(n)	Output	PIN_K1	QL0		PIN_K1	1.5-V PCML	Maximu...fault)	PCIE_TX2
PCIE_TX3	Output	PIN_F2	QL0		PIN_F2	1.5-V PCML	Maximu...fault)	PCIE_TX3(n)
PCIE_TX3(n)	Output	PIN_F1	QL0		PIN_F1	1.5-V PCML	Maximu...fault)	PCIE_TX3
PCIE_nRST	Input	PIN_K10	8A	B8_N0	PIN_K10	2.5 V	16mA (default)	

Рисунок 17 – Сигналы модуля верхнего уровня, подключенные к ПЛИС

Код всех схемотехнических модулей, разработанных для данного проекта приведен в приложении А.

3.7 Выводы

В рамках данной главы были рассмотрены следующие этапы разработки:

1) Обосновано использование сложного функционального блока IP Compiler for PCI Express для преобразования сигналов PCI Express в интерфейс Avalon Memory-Mapped. Показаны настройки этого блока, используемые в проекте.

2) Описана используемая модель передачи данных. Рассмотрены сигналы интерфейса Avalon Memory-Mapped и интерфейса передачи данных, используемого в NITTA.

3) Описан разработанный схемотехнический модуль передачи данных для текущего вычислительного алгоритма NITTA. Изображены блок-схемы его работы.

4) Описано как была изменена вычислительная платформа NITTA для возможности изменения исполняемого алгоритма.

5) Описан разработанный схемотехнический модуль для изменения исполняемого вычислителем алгоритма без реконфигурации ПЛИС. Изображены блок-схемы его работы.

6) Описаны этапы, которые следует выполнить для успешной загрузки модифицированного вычислителя NITTA на ПЛИС.

ГЛАВА 4. ПОДДЕРЖКА ИНТЕРФЕЙСА В КОМПЬЮТЕРЕ

4.1 Описание работы драйвера PCI Express для Linux

При создании модуля ядра мы должны учесть, что каждый модуль должен иметь методы инициализации `module_init` и выхода `module_exit` описанные в библиотеке ядра `module.h`. В методе инициализации мы вызываем функцию из библиотеки ядра `pci.h` `pci_register_driver`, которой следует передать структуру `pci_driver`, также описанную в той же библиотеке со следующими полями:

— `name` – имя драйвера;

— `id_table` – таблица с типом устройства и списком Vendor ID и Device ID устройств, к которым должен применяться этот драйвер;

— `probe` – функция, вызываемая во время подключения драйвера к устройству;

— `remove` – функция, вызываемая при удалении устройства из компьютера или при удалении самого модуля из системы.

В функции `probe` функции необходимо опросить устройство на его доступность, на доступность его памяти, получить все доступные базовые адресные регистры, зарезервировать области памяти, узнать начало и объем памяти для каждого базового пространства, которая была спроецирована в оперативную память, выделить необходимое количество памяти, сделать память устройства видимой для драйвера и получить на нее указатель. Далее создать символьное устройство.

В функции `remove` необходимо проверить есть ли ссылка на память базовых адресных пространств и если есть, то ее следует освободить, далее освободить память, выделенную драйверу, уничтожить все ранее созданные символьные устройства, освободить все PCI ресурсы, сообщить системе, что мы больше не используем PCI устройство [17].

4.2 Описание работы символьного устройства

Символьное устройство позволяет связать модуль ядра с пользовательским пространством и обеспечить простой доступ к памяти устройства.

Создание нашего устройства выполняется в функции `create_char_devs`, вызываемой при подключении драйвера к устройству. Для этого сначала необходимо инициализировать структуру `file_operations`, она имеет следующую сигнатуру:

— `owner` – владелец, ссылка на текущий модуль;

— `open` – функция, вызываемая при открытии символьного устройства;

— `release` – функция, вызываемая при закрытии символьного устройства;

— `unlocked_ioctl` – функция специального взаимодействия с символьным устройством;

— `read` – функция чтения из символьного устройства;

— `write` – функция записи в символьное устройство.

В функции `create_char_devs` сначала необходимо зарегистрировать номера для создаваемых символьных устройств с привязкой к имени драйвера. После чего необходимо получить старший номер для полученных адресов. Необходимо создать структуру `class`, которая представляет абстракцию низкоуровневой реализации устройств в виде вызовов высокоуровневых функций. Для этого необходимо вызвать функцию class_create, передав ей указатель на текущий модуль и имя драйвера. После этого необходимо структуре сообщить функцию, отвечающую за настройку прав доступа к устройству. Далее функция `cdev_init` инициализирует структуру `file_operations` для создаваемого символьного устройства. После следует вызвать функцию `cdev_add` добавляющую устройство в

пространство ядра. В конце следует вызвать функцию `device_create` отображающую символьное устройство в директории `/dev`, где обычно располагаются различные устройства.

Теперь рассмотрим реализацию функций, поддерживаемых создаваемым символьным устройством.

В функции open выделяем память под структуру описывающую память драйвера и инициализируем ее поля и сохраняем указатель на структуру как приватные данные открытого файла для последующего доступа к ней.

В функции release освобождаем выделенную память и удаляем ссылку на нее [18].

Функции, вызываемые в ioctl

Для взаимодействия с ПЛИС было разработано несколько функций:

— `get_count_to_read_from_nitta;`

— `read_from_nitta;`

— `write_to_nitta;`

— `clear_nitta;`

— `download_algorithm.`

`get_count_to_read_from_nitta` посылает запрос на чтение по адресу выделенному для PCI Express в памяти компьютера, который в ПЛИС будет преобразован в нулевой адрес плюс смещение. В качестве результата ПЛИС и функция возвращают количество данных доступных для чтения.

`read_from_nitta` получает от пользователя ссылку на структуру содержащую массив и количество элементов в нем после чего вызывает функцию `get_count_to_read_from_nitta`, получая количество доступных данных для чтения из очереди с результатами сравнивая величину массива и количество доступных данных, выбирая из них минимальное. После читает из очереди с результатами необходимое количество элементов в цикле записывая их в пользовательский массив, возвращает количество прочитанных значений.

`write_to_nitta` получает от пользователя ссылку на такую же структуру, как и в функции `read_from_nitta`, читает из ПЛИС количество свободных мест в очереди для входных значений, сравнивает это число с количеством элементов в массиве также выбирая меньшее, после чего записывает в цикле элементы массива в очередь для входных значений в ПЛИС.

`clear_nitta` записывает число, на деле не важно какое, в выделенный адрес памяти ПЛИС, вызывая обнуление счетчиков обеих очередей и вызывая программный сброс в ПЛИС.

Загрузка нового алгоритма для NITTA осуществляется через ioctl функцию `download_algorithm` принимающую в качестве аргумента указатель на структуру содержащую количество заполняемых элементов, указатель на массив с количеством блоков данных для каждого элемента и указатель на массив содержащий все данные для всех модулей в том же порядке, в котором данные будут программироваться [19].

Код модуля ядра Linux находится в приложении Б.

Вызов всех описанных функций осуществляется через вызов IOCTL функции символьного устройства, которая получает от пользователя дескриптор устройства, уникальное число, которое должно соответствовать некоторому уникальному числу и данные пользователя. Для создания такого числа в библиотеке `linux/ioctl.h` предусмотрены специальные макросы, вызов которых был добавлен в заголовочный файл `chardev.h`, также там присутствует объявление структур, используемых в разработанных функциях. Пользовательская версия заголовочного файла содержит те же самые макросы и объявления структуры. В драйвере внутри функции `nitta_dev_ioctl` при помощи оператора `switch` находится соответствие функции с макросом и выполняется требуемая операция.

4.3 Пользовательская библиотека

Пользовательская библиотека разработана для удобства взаимодействия с разработанным драйвером и имеет следующие функции:

— `open_dev;`

— `close_dev;`

— `read_from_nitta;`

— `write_to_nitta;`

— `clear_nitta;`

— `get_count_to_read;`

— `download_algorithm;`

— `add_to_algorithm.`

Также там присутствует структура `nitta_algorithm`, пользователь должен в своем коде объявить экземпляр ее типа.

Имена некоторых функций совпадают с именами `ioctl` функций модуля ядра и позволяют удобным способом, не используя дополнительные структуры и макросы. Назначение остальных функций изложено далее.

`open_dev` – открывает символьное устройство `nitta` и сохраняет его дескриптор.

`close_dev` – закрывает дескриптор символьного устройства `nitta`.

`add_to_algorithm` – необходим для заполнения созданного экземпляра структуры. Ее необходимо вызвать для записи каждого блока алгоритма в том же порядке, в котором алгоритм должен программироваться.

Код драйвера находится в приложении В.

4.4 Установка драйвера Linux

На компьютере с установленной системой на базе ядра Linux(испытания проводились на операционной системе Ubuntu) необходимо скачать заголовочные файлы ядра текущей версии операционной системы. Это можно сделать, выполнив в терминале следующую команду:

```
sudo apt-get install linux-headers-$(uname –r)
```

Также необходимо установить программу make. Это можно сделать командами:

```
sudo apt update
sudo apt install make
```

Поместить все файлы из приложения Б в один каталог. В файле `pcidriver-main.c` необходимо в строчках `#DEFINE VENDOR_ID` и `#DEFINE DEVICE_ID` заменить шестнадцатеричные числа на соответствующие Vendor ID и Device ID устройства с которым будет проверяться работоспособность драйвера. Далее, находясь в каталоге с файлами вызвать исполнение скрипта `install.sh`.

Для компиляции пользовательской библиотеки необходимо поместить файлы из приложения В в один каталог и выполнить команду make.

В свою программу пользователь должен поместить в каталог своей программы поместить файл `nitta_pci.so` и заголовочные файлы `nitta_pci.h` и `nitta_chardev.h`. После чего может использовать функции библиотеки.

4.5 Результаты работы

На рисунке 18 изображена диаграмма работы тестового модуля эмулирующего работу проекта в ModelSim. В данный момент проект последовательно числа по два. В тестовом модуле сначала эмулируется передача через PCI Express чисел от нуля до девяти, после чего начинается чтение результатов. Полученные результаты соответствуют ожидаемым значениям [20].

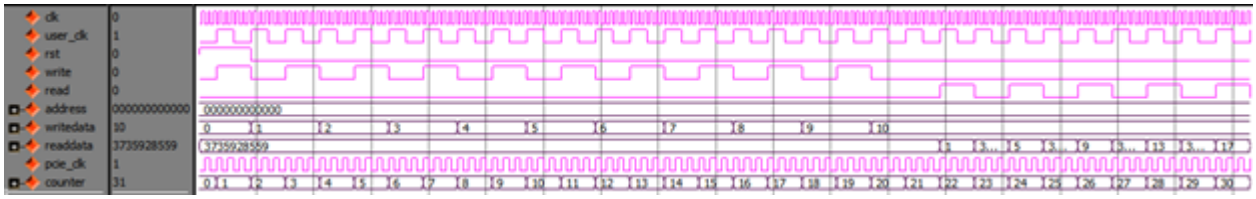

Рисунок 18 – Диаграмма работы тестового модуля проекта

На рисунке 19 приведено изображение экрана где сначала в ПЛИС передается два числа, после чего получается результат. В качестве тестового алгоритма вычислительная платформа складывает два числа и прибавляет к результату единицу. Результат выводится в десятичном и шестнадцатеричных форматах через пробел.

```
~/test ./test
30
5
write 2 or less nums to nitta
Count from nitta: 2%
~/test ./read
read 1 or less nums from nitta
count: 1
ans: 36 24
```

Рисунок 19 – Изображение экрана с примером работы проекта

На рисунке 20 изображена диаграмма работы модуля перепрограммирования вычислительной платформы NITTA в тестовом окружении.

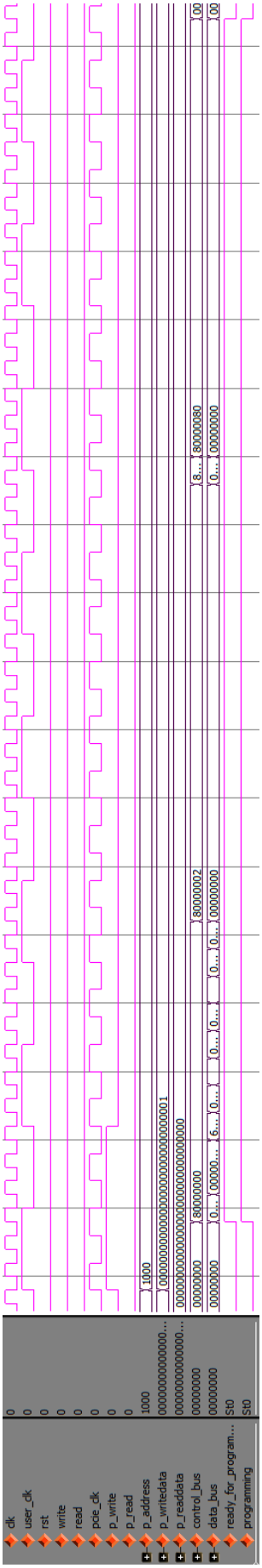

Рисунок 20 – Диаграмма работы процесса изменения алгоритма NITTA

4.6 Выводы

В рамках данной главы были рассмотрены следующие этапы разработки:

1) Описана основа модуля ядра Linux для проекта. Расписаны функции, разработанные для возможности определения драйвером требуемого устройства и подключения к нему.

2) Описана работа символьного устройства. Расписаны разработанные функции, реализующие протоколы взаимодействия пользователя с ПЛИС.

3) Описаны функции пользовательской библиотеки, позволяющие простым образом обращаться пользователю к символьному устройства драйвер.

4) Расписаны этапы, которые следует выполнить для успешного использования разработанного драйвера и пользовательской библиотеки.

5) Приведены результаты тестирования разработанного проекта в тестовом окружении, эмулирующем ПЛИС и на испытательном стенде.

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был разработан высокоскоростной интерфейс передачи данных для вычислительной платформы NITTA. А именно:

— разработаны модули передачи данных для вычислительной платформы NITTA;

— разработаны модули для изменения алгоритма исполняемого NITTA без реконфигурации ПЛИС;

— разработаны тестовые модули для проверки логической верности разработанных модулей;

— разработан PCI Express драйвер для операционной системы на основе ядра Linux;

— разработана пользовательская библиотека для удобства взаимодействия пользователя с драйвером.

Проект тестировался на плате с версией PCI Express x4, однако, есть и версии с 16 контактами для каждого направления. Для того чтобы использовать PCI Express с более широкой шиной необходимо только немного изменить параметры IP Compiler for PCI Express.

Для более эффективной работы с возможностью изменения алгоритма вычислительной платформы можно снабдить модуль прошивки регистрами с информацией о конфигурации, чтобы пользователь мог опросить ПЛИС перед созданием алгоритма загрузкой алгоритма.

Разработанный интерфейс обеспечивает более быстрый и удобный способ обмена данных пользователя с вычислительной платформой NITTA. Что призвано обеспечить достаточную пропускную способность для объемов данных, которыми необходимо манипулировать при расчете моделей системной динамики.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Forrester, J. W., “Industrial Dynamics - After the first Decade”. Management Science Vol. 14 No. 7, Март, 1968.
2. Лычкина Н. Н. Ретроспектива и перспектива системной динамики. Анализ динамики развития // Бизнес-информатика. 2009. №3. URL: https://cyberleninka.ru/article/n/retrospektiva-i-perspektiva-sistemnoy-dinamiki-analiz-dinamiki-razvitiya (дата обращения: 08.05.2020).
3. Горохов, А. В. Основы системного анализа: учебное пособие для вузов / А. В. Горохов. — Москва: Издательство Юрайт, 2019. — 140 с. — (Университеты России). — ISBN 978-5-534-09459-6.
4. Способы расчета моделей системной динамики в облачной инфраструктуре URL: https://openbooks.itmo.ru/ru/file/5083/5083.pdf (дата обращения 03.05.20).
5. Пенской А.В. Разработка и исследование архитектурных стилей проектирования уровневой организации встроенных систем : канд. техн. наук : 05.13.12 / Пенской А.В. - Санкт-Петербург, 2016. - 169 с. URL: https://isu.ifmo.ru/index/0EF1389C59C61A76286892961DA96781 (дата обращения 08.05.2020).
6. Пенской А.В. Проектирование вычислительной платформы для моделирования динамических систем. XLVII научная и учебно-методическая конференция Университета ИТМО. 2018 URL: https://elibrary.ru/download/elibrary_41375494_13531777.pdf (дата обращения 08.05.2020).
7. И. А. Перл, М. М. Петрова, А. А. Мулюкин, О. В. Каленова Исполнение моделей системной динамики на основе непрерывного потока входных данных // Программные продукты и системы. 2018. №2. URL: https://cyberleninka.ru/article/n/ispolnenie-modeley-sistemnoy-dinamiki-

na-osnove-nepreryvnogo-potoka-vhodnyh-dannyh (дата обращения: 08.05.2020).

8. RapidIO Interconnect Specification URL: http://www.rapidio.org/wp-content/uploads/2018/06/RapidIO-Specification-4-1.pdf (дата обращения 09.05.20).
9. PCI Local Bus Specification Revision 3.0 URL: http://fpga-faq.narod.ru/PCI_Rev_30.pdf (дата обращения 09.05.20).
10. PCI Express Base Specification Revision 3.0 URL: http://www.lttconn.com/res/lttconn/pdres/201402/20140218105502619.pdf (дата обращения 09.05.20).
11. An Introduction to the Intel QuickPath Interconnect URL: https://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf (дата обращения 09.05.20).
12. B. Holden, J. Trodden, D. Anderson HyperTransport 3.1 InterconnectTechnology URL: https://www.mindshare.com/files/ebooks/HyperTransport%203.1%20Interconnect%20Technology.pdf (дата обращения 09.05.20).
13. IP Compiler for PCI Express User Guide URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_pci_express.pdf (дата обращения 19.04.20).
14. Базовые принципы построения FIFO URL: https://marsohod.org/11-blog/195-fifobasics (дата обращения 19.04.20).
15. Asynchronous FIFO URL: - http://www.asic-world.com/examples/verilog/asyn_fifo.html (дата обращения 19.04.20)
16. Qsys System Design Tutorial URL: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tt/tt_qsys_intro.pdf (дата обращения 19.04.20).
17. linux/pci.h URL: https://github.com/torvalds/linux/blob/master/include/linux/pci.h (дата обращения 19.04.20)

18. linux/cdev.h URL: https://github.com/torvalds/linux/blob/master/include/linux/cdev.h (дата обращения 19.04.20)
19. linux/ioctl.h URL: https://github.com/torvalds/linux/blob/master/arch/alpha/include/uapi/asm/ioctl.h (дата обращения 19.04.20).
20. ModelSim User's Manual URL: https://www.microsemi.com/document-portal/doc_view/131619-modelsim-user (дата обращения 19.04.20).

ПРИЛОЖЕНИЕ А

```
Файл pu_slave_pci.sv
`timescale 1ps/1ps
module pu_slave_pci
      #(    parameter   ADDRESS_WIDTH = 12,
                                    DATA_WIDTH = 32,
                                    DATA_DEPTH = (1 << ADDRESS_WIDTH),
                                    ATTR_WIDTH = 4,
                                    NUMBER_OF_RECEIVE = 2,
                                    NUMBER_OF_SEND = 1,
                                    FIFO_SIZE = 8)
(
      input pcie_clk,
      input reset,
      // AMMI
      input                         write,
      input                         read,
      input [ADDRESS_WIDTH-1:0]     address,
      input [DATA_WIDTH-1:0]        writedata,
      output reg [DATA_WIDTH-1:0]   readdata
      // NITTA interface
    , input                         signal_cycle_begin
    , input                         signal_in_cycle
    , input                         signal_cycle_end
    , input                         nitta_clk
    , input                         signal_wr
    , input    [DATA_WIDTH-1:0]     data_in
    , input    [ATTR_WIDTH-1:0]     attr_in
    , input                         signal_oe
    , output   [DATA_WIDTH-1:0]     data_out
    , output   [ATTR_WIDTH-1:0]     attr_out
    , output reg                    flag_stop
    , output                        software_reset
);
logic [DATA_WIDTH-1:0] buffer;
// fifo in
logic empty_in;
logic read_en_in;
logic full_in;
logic write_en_in;
logic [7:0] count_in;
logic reset_in;
logic [DATA_WIDTH-1:0] pcie_data_in;
logic [DATA_WIDTH-1:0] nitta_data_out;
aFifo #(DATA_WIDTH, FIFO_SIZE) fifo_in(nitta_data_out, empty_in, read_en_in,
nitta_clk, pcie_data_in, full_in, write_en_in, pcie_clk, count_in, reset_in);
// end fifo in
// fifo out
logic empty_out;
logic read_en_out;
logic full_out;
logic write_en_out;
logic [7:0] count_out;
logic [DATA_WIDTH-1:0] pcie_data_out;
logic [DATA_WIDTH-1:0] nitta_data_in;
logic read_en_in_pcie;
aFifo #(DATA_WIDTH, FIFO_SIZE) fifo_out(pcie_data_out, empty_out,
read_en_out, pcie_clk, nitta_data_in, full_out, write_en_out, nitta_clk,
count_out, reset_out);
// end fifo out
```

```
assign read_en_in = (signal_wr & signal_oe) | read_en_in_pcie;
assign write_en_out = signal_wr & ~signal_oe;
assign nitta_data_in = data_in;

assign buffer = (signal_oe && ~signal_wr) ? buffer : nitta_data_out;
logic prew_signal_oe = 0;
assign data_out = prew_signal_oe ? buffer : 0;
always_ff @(posedge nitta_clk)
      prew_signal_oe <= signal_oe;
assign software_reset = (write == 1 && address == 1);
assign reset_in = software_reset;
assign reset_out = software_reset;
// test
assign read_en_in_pcie = (address == 4 && read);
always_comb
begin
      if(read)
      begin
            if(address == 0/* & empty_out == 0*/)
                  readdata = pcie_data_out;
            else  if(address == 1)
                  readdata = (1 << FIFO_SIZE) - count_in;
            else if(address == 2)
                  readdata = count_out;
            else if(address == 4)
                  readdata = nitta_data_out;
            else if(address == 5)
                  readdata = 0;
            else
                  readdata = 32'hDEAD_DEAD;
      end else
            readdata = 32'hDEAD_BEEF;
end
assign pcie_data_in = writedata;
logic write_prew;
logic read_prew;
always_ff @(posedge pcie_clk)
begin
      write_prew <= write;
      read_prew <= read;
end
assign write_en_in = ~write_prew & address == 0 & write;
assign read_en_out = ~read_prew & address == 0 & read & empty_out == 0;
wire fifo_ready_for_send_data;
wire fifo_ready_for_read_data;
assign fifo_ready_for_send_data = (count_out + NUMBER_OF_SEND) < (1 <<
FIFO_SIZE);
assign fifo_ready_for_read_data = count_in >= NUMBER_OF_RECEIVE;
assign flag_stop = fifo_ready_for_read_data & fifo_ready_for_send_data;
assign attr_out = 0;
endmodule
```

Файл afifo.sv

```
//=========================================
// Function : Asynchronous FIFO (w/ 2 asynchronous clocks).
// Coder    : Alex Claros F.
// Date     : 15/May/2005.
// Notes    : This implementation is based on the article
//            'Asynchronous FIFO in Virtex-II FPGAs'
//            writen by Peter Alfke. This TechXclusive
//            article can be downloaded from the
//            Xilinx website. It has some minor modifications.
//=========================================
`timescale 1ns/1ns
module aFifo
  #(parameter    DATA_WIDTH    = 8,
                 ADDRESS_WIDTH = 4,
                 FIFO_DEPTH    = (1 << ADDRESS_WIDTH))
     //Reading port
    (output reg  [DATA_WIDTH-1:0]        Data_out
    ,output reg                          Empty_out,//Empty_all,
     input wire                          ReadEn_in,
     input wire                          RClk,
     //Writing port.
     input wire  [DATA_WIDTH-1:0]        Data_in,
     output reg                          Full_out,
     input wire                          WriteEn_in,
     input wire                          WClk,
       // Count elements
       output logic [ADDRESS_WIDTH-1:0]       Count,
     input wire                          Clear_in);
    /////Internal connections & variables//////
    reg   [DATA_WIDTH-1:0]               Mem [FIFO_DEPTH-1:0];
    logic  [ADDRESS_WIDTH-1:0]           pNextWordToWrite;
    logic  [ADDRESS_WIDTH-1:0]           pNextWordToRead;
    wire                                 EqualAddresses;
    wire                                 NextWriteAddressEn,
NextReadAddressEn;
    wire                                 Set_Status, Rst_Status;
    reg                                 Status;
    wire                                PresetFull, PresetEmpty;
    logic [ADDRESS_WIDTH-1:0] top;
    logic [ADDRESS_WIDTH-1:0] tail;
    // Data ports logic: (Uses a dual-port RAM).
       always_ff @(posedge RClk)
            if(ReadEn_in & !Empty_out)
                  Data_out <= Mem[pNextWordToRead];
            else
                  Data_out <= Data_out;
    //'Data_in' logic:
    always @ ( posedge WClk)
        if (WriteEn_in & !Full_out)
            Mem[pNextWordToWrite] <= Data_in;
    //Fifo addresses support logic:
    //'Next Addresses' enable logic:
    assign NextWriteAddressEn = WriteEn_in & ~Full_out;
    assign NextReadAddressEn  = ReadEn_in & ~Empty_out;//(ReadEn_in |
Buff_front)  & ~Empty_out;
    initial begin
          Data_out <= 0; // для первого способа //modelsim с этой строчкой не
работает, а ПЛИС работает
        Full_out <= 0;
        Empty_out <= 1;
        Status <= 0;
```

```
    end
    //Addreses (Gray counters) logic:
    GrayCounter #(ADDRESS_WIDTH) GrayCounter_pWr
       (.GrayCount_out(pNextWordToWrite),
        .Bin_out(top),
        .Enable_in(NextWriteAddressEn),
        .Clear_in(Clear_in),
        .Clk(WClk)
       );
    GrayCounter #(ADDRESS_WIDTH) GrayCounter_pRd
       (.GrayCount_out(pNextWordToRead),
        .Bin_out(tail),
        .Enable_in(NextReadAddressEn),
        .Clear_in(Clear_in),
        .Clk(RClk)
       );
    always_comb
        Count = (top > tail)? top - tail :top - tail + FIFO_DEPTH;
    //'EqualAddresses' logic:
    assign EqualAddresses = (pNextWordToWrite == pNextWordToRead);
    //'Quadrant selectors' logic:
    assign Set_Status = (pNextWordToWrite[ADDRESS_WIDTH-2] ~^
pNextWordToRead[ADDRESS_WIDTH-1]) &
                        (pNextWordToWrite[ADDRESS_WIDTH-1] ^
pNextWordToRead[ADDRESS_WIDTH-2]);
    assign Rst_Status = (pNextWordToWrite[ADDRESS_WIDTH-2] ^
pNextWordToRead[ADDRESS_WIDTH-1]) &
                        (pNextWordToWrite[ADDRESS_WIDTH-1] ~^
pNextWordToRead[ADDRESS_WIDTH-2]);
    //'Status' latch logic:
    always @ (Set_Status, Rst_Status, Clear_in) //D Latch w/ Asynchronous
Clear & Preset.
        if (Rst_Status | Clear_in)
            Status = 0;  //Going 'Empty'.
        else if (Set_Status)
            Status = 1;  //Going 'Full'.
    //'Full_out' logic for the writing port:
    assign PresetFull = Status & EqualAddresses;  //'Full' Fifo.
    always @ (posedge WClk, posedge PresetFull) //D Flip-Flop w/ Asynchronous
Preset.
        if (PresetFull)
            Full_out <= 1;
        else
            Full_out <= 0;
    //'Empty_out' logic for the reading port:
    assign PresetEmpty = ~Status & EqualAddresses;  //'Empty' Fifo.
    always @ (posedge RClk, posedge PresetEmpty)  //D Flip-Flop w/
Asynchronous Preset.
        if (PresetEmpty)
            Empty_out <= 1;
        else
            Empty_out <= 0;
endmodule
```

Файл: GrayCounter.sv

```
//=======================================
// Function : Code Gray counter.
// Coder    : Alex Claros F.
// Date     : 15/May/2005.
//=======================================
`timescale 1ns/1ps
module GrayCounter
   #(parameter   COUNTER_WIDTH = 4)
    (output reg  [COUNTER_WIDTH-1:0]    GrayCount_out,  //'Gray' code count
output.
     output logic [COUNTER_WIDTH-1:0]   Bin_out,
     input wire                         Enable_in,  //Count enable.
     input wire                         Clear_in,   //Count reset.
     input wire                         Clk);
    /////////Internal connections & variables//////
    reg    [COUNTER_WIDTH-1:0]          BinaryCount = 1;
    /////////Code///////////////////////
    initial begin
        GrayCount_out <= 0;
    end
    always @ (posedge Clk)
        if (Clear_in) begin
            BinaryCount   <= {COUNTER_WIDTH{1'b 0}} + 1;  //Gray count begins
@ '1' with
            GrayCount_out <= {COUNTER_WIDTH{1'b 0}};      // first
'Enable_in'.
        end
        else if (Enable_in) begin
            BinaryCount   <= BinaryCount + 1;
            GrayCount_out <= {BinaryCount[COUNTER_WIDTH-1],
                              BinaryCount[COUNTER_WIDTH-2:0] ^
BinaryCount[COUNTER_WIDTH-1:1]};
        end
       assign Bin_out = BinaryCount - 1;
endmodule
```

```
модифицированный файл main_net.v
module main_net #
      ( parameter DATA_WIDTH = 32
      , parameter ATTR_WIDTH = 4
      , parameter ADDRESS_WIDTH = 12
      )
    ( input                     clk
    , input                     reset
    , input                     is_drop_allow
    , output                    flag_cycle_begin
    , output                    flag_in_cycle
    , output                    flag_cycle_end
    , input mosi
    , input sclk
    , input cs
    , output miso
    , output              [7:0] debug_status
    , output              [7:0] debug_bus1
    , output              [7:0] debug_bus2
    // avalon memory-mapped interface
    , input                         write
    , input                         read
    , input [ADDRESS_WIDTH-1:0]     address
    , input [DATA_WIDTH-1:0]        writedata
    , output[DATA_WIDTH-1:0]        readdata
    // pci
    , input                         pcie_clk
    // avalon memory-mapped interface for programmer
    , input                         p_write
    , input                         p_read
    , input [ADDRESS_WIDTH-1:0]     p_address
    , input [DATA_WIDTH-1:0]        p_writedata
    , output[DATA_WIDTH-1:0]        p_readdata
    );
parameter MICROCODE_WIDTH = 32;
wire ready_for_programming;
wire programming;
assign programming = 0;
wire rst;
wire software_reset;
assign rst = reset | software_reset;
// Sub module_ instances
wire [MICROCODE_WIDTH-1:0] control_bus;
wire [DATA_WIDTH-1:0] data_bus;
wire [ATTR_WIDTH-1:0] attr_bus;
wire start, stop;
wire cycle;
wire [7:0] debug_pc;
assign debug_status = 0;//{ flag_cycle_begin, flag_in_cycle, flag_cycle_end,
debug_pc[4:0] };
assign debug_bus1 = data_bus[7:0];
assign debug_bus2 = data_bus[31:24] | data_bus[23:16] | data_bus[15:8] |
data_bus[7:0];
pu_simple_control #
        ( .MICROCODE_WIDTH( MICROCODE_WIDTH )
        , .PROGRAM_DUMP( "main_net/1_main.dump" )
        , .BASE_MEMORY_SIZE( 15 ) // 0 - address for nop microcode
        , .MICROCODE_DEPTH( 7 )
        ) control_unit
    ( .clk( clk )
    , .rst( rst )
    , .signal_cycle_start( 1'b0 || stop )
```

```
    , .signals_out( control_bus )
    , .flag_cycle_begin( flag_cycle_begin )
    , .flag_in_cycle( flag_in_cycle )
    , .flag_cycle_end( flag_cycle_end )
    , .programming( programming )
    , .ready_for_programming( 1'b0 || ready_for_programming )
    , .data_in( data_bus )
    );
wire [DATA_WIDTH-1:0] accum_data_out;
wire [ATTR_WIDTH-1:0] accum_attr_out;
pu_accum #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        ) accum
    ( .clk( clk )
    , .rst( rst )
    , .signal_resetAcc( control_bus[18] )
    , .signal_load( control_bus[19] )
    , .signal_neg( control_bus[20] )
    , .signal_oe( control_bus[21] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .data_out( accum_data_out )
    , .attr_out( accum_attr_out )
    );
wire [DATA_WIDTH-1:0] div_data_out;
wire [ATTR_WIDTH-1:0] div_attr_out;
pu_div #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .INVALID( 0 )
        , .PIPELINE( 4 )
        , .SCALING_FACTOR_POWER( 0 )
        , .MOCK_DIV( 1'b1 )
        ) div
    ( .clk( clk )
    , .rst( rst )
    , .signal_wr( control_bus[25] )
    , .signal_wr_sel( control_bus[26] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .signal_oe( control_bus[27] )
    , .signal_oe_sel( control_bus[28] )
    , .data_out( div_data_out )
    , .attr_out( div_attr_out )
    );
wire [DATA_WIDTH-1:0] fram1_data_out;
wire [ATTR_WIDTH-1:0] fram1_attr_out;
pu_fram #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .RAM_SIZE( 16 )
        , .FRAM_DUMP( "main_net/1_fram1.dump" )
        ) fram1
    ( .clk( clk )
    , .signal_addr( { control_bus[2], control_bus[3], control_bus[4],
control_bus[5] } )
    , .signal_wr( control_bus[1] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .signal_oe( control_bus[0] )
    , .data_out( fram1_data_out )
```

```
    , .attr_out( fram1_attr_out )
    , .programming( programming )
    );
wire [DATA_WIDTH-1:0] fram2_data_out;
wire [ATTR_WIDTH-1:0] fram2_attr_out;
pu_fram #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .RAM_SIZE( 16 )
        , .FRAM_DUMP( "main_net/1_fram2.dump" )
        ) fram2
    ( .clk( clk )
    , .signal_addr( { control_bus[8], control_bus[9], control_bus[10],
control_bus[11]} )
    , .signal_wr( control_bus[7] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .signal_oe( control_bus[6] )
    , .data_out( fram2_data_out )
    , .attr_out( fram2_attr_out )
    , .programming( programming )
    );
wire [DATA_WIDTH-1:0] mul_data_out;
wire [ATTR_WIDTH-1:0] mul_attr_out;
pu_multiplier #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .SCALING_FACTOR_POWER( 0 )
        , .INVALID( 0 )
        ) mul
    ( .clk( clk )
    , .rst( rst )
    , .signal_wr( control_bus[22] )
    , .signal_sel( control_bus[23] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .signal_oe( control_bus[24] )
    , .data_out( mul_data_out )
    , .attr_out( mul_attr_out )
    );
wire [DATA_WIDTH-1:0] shift_data_out;
wire [ATTR_WIDTH-1:0] shift_attr_out;
pu_shift #
        ( .DATA_WIDTH( 32 )
        , .ATTR_WIDTH( ATTR_WIDTH )
        ) shift
    ( .clk( clk )
    , .signal_work( control_bus[12] ), .signal_direction( control_bus[13] )
    , .signal_mode( control_bus[14] ), .signal_step( control_bus[15] )
    , .signal_init( control_bus[16] ), .signal_oe( control_bus[17] )
    , .data_in( data_bus )
    , .attr_in( attr_bus )
    , .data_out( shift_data_out )
    , .attr_out( shift_attr_out )
    );
wire [DATA_WIDTH-1:0] spi_data_out;
wire [ATTR_WIDTH-1:0] spi_attr_out;
pu_slave_pci #
            ( .ADDRESS_WIDTH( ADDRESS_WIDTH )
            , .DATA_WIDTH( DATA_WIDTH )
            , .DATA_DEPTH ( 1 << ADDRESS_WIDTH )
            , .ATTR_WIDTH ( ATTR_WIDTH )
```

```
            , .NUMBER_OF_RECEIVE( 2 )
            , .NUMBER_OF_SEND( 1 )
            , .FIFO_SIZE( 8 )
            ) pci
        ( // pci
          .pcie_clk(pcie_clk)
        , .reset(rst)
        , .write(write)
        , .read(read)
        , .address(address)
        , .writedata(writedata)
        , .readdata(readdata)
        // nitta
        , .signal_cycle_begin( flag_cycle_begin )
    , .signal_in_cycle( flag_in_cycle )
    , .signal_cycle_end( flag_cycle_end )
        , .nitta_clk( clk )
        , .signal_wr( control_bus[29] )
        , .data_in( data_bus ), .attr_in( attr_bus )
        , .signal_oe( control_bus[30] )
    , .data_out( spi_data_out ), .attr_out( spi_attr_out )
    , .flag_stop( stop )
        , .software_reset(software_reset)
        );
wire [DATA_WIDTH-1:0] programmer_data_out;
wire [ATTR_WIDTH-1:0] programmer_attr_out;
pu_slave_pci_programmer #
            ( .ADDRESS_WIDTH( ADDRESS_WIDTH )
            , .DATA_WIDTH( DATA_WIDTH )
            , .DATA_DEPTH ( 1 << ADDRESS_WIDTH )
            , .ATTR_WIDTH ( ATTR_WIDTH )
            , .COUNT_ELEMENTS( 3 )
            ) programmer
        ( // pci
          .pcie_clk(pcie_clk)
        , .reset(rst)
        , .write(p_write)
        , .read(p_read)
        , .address(p_address)
        , .writedata(p_writedata)
        , .readdata(p_readdata)
        // nitta
        , .signal_cycle_begin( flag_cycle_begin )
        , .signal_in_cycle( flag_in_cycle )
        , .signal_cycle_end( flag_cycle_end )
        , .nitta_clk( clk )
        , .data_in( data_bus ), .attr_in( attr_bus )
        , .signal_oe( control_bus[31] )
        , .data_out( programmer_data_out ), .attr_out( programmer_attr_out )
        , .programming(programming)
        , .ready_for_programming( ready_for_programming )
        );
assign data_bus = accum_data_out | div_data_out | fram1_data_out |
fram2_data_out | mul_data_out | shift_data_out | spi_data_out |
programmer_data_out;
assign attr_bus = accum_attr_out | div_attr_out | fram1_attr_out |
fram2_attr_out | mul_attr_out | shift_attr_out | spi_attr_out |
programmer_attr_out;
endmodule
```

```
модифицированный файл pu_simple_control.v
module pu_simple_control #
        ( parameter MICROCODE_WIDTH = 16
        , parameter BASE_MEMORY_SIZE = 200
        , parameter PROGRAM_DUMP = "dump.list"
        , parameter MICROCODE_DEPTH = $clog2( BASE_MEMORY_SIZE )//7
        , parameter MEMORY_SIZE = (1 << MICROCODE_DEPTH)
        , parameter DATA_WIDTH = 32
        , parameter PROGRAMMER_MEMORY_SIZE = 7
        )
    ( input wire                          clk
    , input wire                          rst
    , input wire                          signal_cycle_start
    , output wire                         flag_cycle_begin
    , output wire                         flag_in_cycle
    , output wire                         flag_cycle_end
    , output wire [MICROCODE_WIDTH-1:0]   signals_out
    , output wire                         programming
    , input  wire                         ready_for_programming
    , input  wire [DATA_WIDTH - 1 : 0]    data_in
    );
reg [MICROCODE_WIDTH-1:0]                 program_memory[MEMORY_SIZE-1:0];
reg [MICROCODE_WIDTH-1:0]                 programmer_memory
[PROGRAMMER_MEMORY_SIZE - 1 : 0];
initial begin
      programmer_memory[0] = 32'h8000_0000;
      programmer_memory[1] = 32'h8000_0000;
      programmer_memory[2] = 32'h8000_0000;
      programmer_memory[3] = 32'h8000_0002;
      programmer_memory[4] = 32'h8000_0000;
      programmer_memory[5] = 32'h8000_0080;
      programmer_memory[6] = 0;
end
//0 get count pu_simple_control
//1 get all data for pu_simple_control
//2 get count pu_fram1
//3 get all data for pu_fram1
//4 get count pu_fram2
//5 get all data for pu_fram2
//6 off programming
//set software_reset
reg [MICROCODE_DEPTH-1:0]                       pc;
reg [MICROCODE_DEPTH-1 : 0]                     pp = 0;
reg [MEMORY_SIZE - 1 : 0]                       counter = 0;
reg [MICROCODE_DEPTH - 1 : 0] curr_program_depth = BASE_MEMORY_SIZE;
reg [MICROCODE_DEPTH - 1 : 0] fram_count;
assign programming = ~flag_in_cycle & ready_for_programming;
initial $readmemh(PROGRAM_DUMP, program_memory, 0, BASE_MEMORY_SIZE-1);
always @(posedge clk) begin
       if     ( pp == PROGRAMMER_MEMORY_SIZE - 1 )
                    pp <= 0;
    else if       ( programming ) begin
            if ( pp[0] == 0 )
                                          pp <= pp + 1;
            else if ( pp == 1 & counter >= curr_program_depth - 1 )
         pp <= pp + 1;
            else if ( pp[0] == 1 & counter >= fram_count - 1 )
              pp <= pp + 1;
       end else
    if      ( rst )                                          pc <= 0;
    else if ( pc == 0 && signal_cycle_start )                pc <= 1;
    else if ( pc >= curr_program_depth - 1 && !signal_cycle_start ) pc <= 0;
```

```
    else if ( pc >= curr_program_depth - 1 &&  signal_cycle_start ) pc <= 1;
    else if ( pc > 0 )                                              pc <= pc + 1;
end
always @(posedge clk)
begin
      if(programming)
            if(pp[0] == 0)
                  counter <= 0;
            else
                  counter <= counter + 1;
      else
            counter <= 0;
end
always @(posedge clk)
begin
      if(programming)
            if(pp == 0)
                  curr_program_depth <= data_in;
            else if(pp == 1) begin
                  if(counter <= curr_program_depth - 1 )
                        program_memory[counter] <= data_in;
            end else if (pp[0] == 0)
                  fram_count <= data_in;
end
assign signals_out = programming ? programmer_memory[pp] : rst ?
program_memory[0] : program_memory[pc];
assign flag_cycle_begin = pc == 1;
assign flag_in_cycle    = pc != 0;
assign flag_cycle_end   = pc == curr_program_depth - 1;
endmodule
```

модифицированный файл pu_fram.v

```
module pu_fram
  #( parameter RAM_SIZE   = 16
   , parameter DATA_WIDTH = 32
   , parameter ATTR_WIDTH = 4
   , parameter ADDR_WIDTH = $clog2( RAM_SIZE )
   , parameter FRAM_DUMP = "NOT_DEFINED"
   )
  ( input  wire                  clk
  , input  wire [ADDR_WIDTH-1:0] signal_addr
  , input  wire                  signal_wr
  , input  wire [DATA_WIDTH-1:0] data_in
  , input  wire [ATTR_WIDTH-1:0] attr_in
  , input  wire                  signal_oe
  , output reg  [DATA_WIDTH-1:0] data_out
  , output reg  [ATTR_WIDTH-1:0] attr_out
  , input                         programming
  );
reg [DATA_WIDTH-1:0] data;
reg [ATTR_WIDTH-1:0] attr;
reg [ADDR_WIDTH-1:0] addr;
reg                  wr;
always @(posedge clk)
      if(~programming)
      begin
            addr <= signal_addr;
            wr   <= signal_wr;
            data <= data_in;
            attr <= attr_in;
      end
reg [DATA_WIDTH+ATTR_WIDTH-1:0] bank [RAM_SIZE-1:0];
initial $readmemh(FRAM_DUMP, bank, 0, RAM_SIZE-1);
reg [ADDR_WIDTH-1:0] counter;
always @( posedge clk )
      if(programming & signal_wr)
            counter <= counter + 1;
      else
            counter <= 0;
always @(posedge clk)
  if (programming & signal_wr) bank[counter] <= {4'h0, data_in};
  else if ( wr ) bank[addr] <= { attr, data };
always @(posedge clk)
  if ( ~signal_oe ) { attr_out, data_out } <= 0;
  else              { attr_out, data_out } <= bank[ signal_addr ];
endmodule
```

```
файл pu_slave_pci_programmer.sv
module pu_slave_pci_programmer
      #(    parameter   ADDRESS_WIDTH = 12,
                                    DATA_WIDTH = 32,
                                    DATA_DEPTH = (1 << ADDRESS_WIDTH),
                                    ATTR_WIDTH = 4,
                                    COUNT_ELEMENTS = 3)
(     input pcie_clk,
      input reset,
      // hard ip interface
      input write,
      input read,
      input [ADDRESS_WIDTH-1:0] address,
      input [DATA_WIDTH-1:0] writedata,
      output reg [DATA_WIDTH-1:0] readdata
      // NITTA interface
    , input                      signal_cycle_begin
    , input                      signal_in_cycle
    , input                      signal_cycle_end
    , input                     nitta_clk
    , input                      signal_wr
    , input    [DATA_WIDTH-1:0] data_in
    , input    [ATTR_WIDTH-1:0] attr_in
    , input                      signal_oe
    , output   [DATA_WIDTH-1:0] data_out
    , output   [ATTR_WIDTH-1:0] attr_out
    , input                      programming
    , output                     ready_for_programming
);
logic [ADDRESS_WIDTH - 1 : 0] address_pointer = 0;
logic [DATA_WIDTH - 1 : 0] count_in_element [COUNT_ELEMENTS - 1 : 0];
logic transfer_state = 0;
int pe = 0;
logic algorithm_confirmed = 0;
logic algorithm_confirmed_nitta_clk = 0;
always_ff@( posedge nitta_clk ) begin
      algorithm_confirmed_nitta_clk <= algorithm_confirmed;
end
int i;
always_ff @(posedge pcie_clk)
begin
      if(write & ~programming)
            if(address < 300) begin
                  algorithm_confirmed <= 0;
            end
            else if (address == 1000)
                  algorithm_confirmed <= 1;
            else if (address == 1001 & writedata <= 100)         // Размер
памяти для pu_simple_control
                  count_in_element[0] <= writedata;
            else if (address == 1002 & writedata <= 16)          // Размер
памяти для fram1
                  count_in_element[1] <= writedata;
            else if (address == 1003 & writedata <= 16)          // Размер
памяти для fram2
                  count_in_element[2] <= writedata;
end
logic confirmed_prew;
logic not_confirmed_front;
assign not_confirmed_front = ~algorithm_confirmed & confirmed_prew;
always_ff @ ( posedge nitta_clk )
      confirmed_prew <= algorithm_confirmed;
```

```
int counter = 0;
logic programmed = 0;
always_ff @(posedge nitta_clk)
begin
      if(~algorithm_confirmed_nitta_clk)
            programmed <= 0;
      else
            if(signal_oe) begin
                  programmed <= 0;
                  if(~transfer_state) begin
                        transfer_state <= ~transfer_state;
                        address_pointer <= address_pointer + 1;
                        counter <= 0;
                  end else if (counter < count_in_element[pe] - 1) begin
                        address_pointer <= address_pointer + 1;
                        counter <= counter + 1;
                  end else begin
                        transfer_state <= ~transfer_state;
                        if (pe >= COUNT_ELEMENTS - 1) begin
                              programmed <= 1;
                              pe <= 0;
                              address_pointer <= 0;
                        end else begin
                              address_pointer <= (pe + 1) * 100;
                              pe <= pe + 1;
                        end   end   end   end
wire wr;
assign wr = address < 300 & write;
wire [9 - 1 : 0] addr;
assign addr = address < 300 ? address : 0;
wire [DATA_WIDTH - 1 : 0] nitaa_data_out;
bram_tdp #
      ( .DATA( DATA_WIDTH )
   , .ADDR( 9 )
      ) mem
      ( // Port A
      .a_clk( pcie_clk )
   , .a_wr( wr )
   , .a_addr( addr )
   , .a_din( writedata )
   , .a_dout( readdata )
    // Port B
   , .b_clk( nitta_clk )
   , .b_addr( address_pointer )
   , .b_dout( nitaa_data_out ));
assign data_out = signal_oe ? transfer_state ? nitaa_data_out:
count_in_element[pe]: 0;
assign attr_out = 0;
assign ready_for_programming = algorithm_confirmed_nitta_clk & ~programmed;
endmodule
```

```
файл bram_tdp.sv
module bram_tdp #(
    parameter DATA = 72,
    parameter ADDR = 10
) (
    // Port A
    input   wire                a_clk,
    input   wire                a_wr,
    input   wire    [ADDR-1:0]  a_addr,
    input   wire    [DATA-1:0]  a_din,
    output  reg     [DATA-1:0]  a_dout,
    // Port B
    input   wire                b_clk,
    input   wire                b_wr,
    input   wire    [ADDR-1:0]  b_addr,
    input   wire    [DATA-1:0]  b_din,
    output  reg     [DATA-1:0]  b_dout
);
reg [DATA-1:0] mem [(2**ADDR)-1:0]; // Shared memory
int i;
initial begin
      for(i = 0; i < 2**ADDR; i = i + 1)
           mem[i] <= 0; end
// Port A
always @(posedge a_clk) begin
    a_dout      <= mem[a_addr];
    if(a_wr) begin
        a_dout      <= a_din;
        mem[a_addr] <= a_din;
    end end
// Port B
always @(posedge b_clk) begin
    b_dout      <= mem[b_addr];
    if(b_wr) begin
        b_dout      <= b_din;
        mem[b_addr] <= b_din; end end
    endmodule
```

файл top.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
library altera;
use altera.altera_syn_attributes.all;
entity top
 is
      port
      (
            HCLK        : IN STD_LOGIC;
            PCIE_nRST   : IN STD_LOGIC;
            PCIE_REFCLK : IN STD_LOGIC;
            PCIE_RX0    : IN STD_LOGIC;
            PCIE_RX1    : IN STD_LOGIC;
            PCIE_RX2    : IN STD_LOGIC;
            PCIE_RX3    : IN STD_LOGIC;
            PCIE_TX0    : OUT STD_LOGIC;
            PCIE_TX1    : OUT STD_LOGIC;
            PCIE_TX2    : OUT STD_LOGIC;
            PCIE_TX3    : OUT STD_LOGIC
      );
end top;
architecture ARCH of top is
      signal clk_200MHz : std_logic;
      signal clk_5kHz : std_logic;
      signal rst : std_logic;
      signal clk : std_logic;
      component nitta_qsys is
            port (
                  clk_clk             : in  std_logic := 'X'; -- clk
                  refclk_export       : in  std_logic := 'X'; -- export
                  reset_reset_n       : in  std_logic := 'X'; -- reset_n
                  rstn_export         : in  std_logic := 'X'; -- export
                  rx_in_rx_datain_0   : in  std_logic := 'X'; -- rx_datain_0
                  rx_in_rx_datain_1   : in  std_logic := 'X'; -- rx_datain_1
                  rx_in_rx_datain_2   : in  std_logic := 'X'; -- rx_datain_2
                  rx_in_rx_datain_3   : in  std_logic := 'X'; -- rx_datain_3
                  tx_out_tx_dataout_0 : out std_logic;        -- tx_dataout_0
                  tx_out_tx_dataout_1 : out std_logic;        -- tx_dataout_1
                  tx_out_tx_dataout_2 : out std_logic;        -- tx_dataout_2
                  tx_out_tx_dataout_3 : out std_logic         -- tx_dataout_3
            );
      end component nitta_qsys;
begin
      u0 : component nitta_qsys
            port map (
                  clk_clk             => HCLK,
                  refclk_export       => PCIE_REFCLK,
                  reset_reset_n       => '1',
                  rstn_export         => PCIE_nRST,
                  rx_in_rx_datain_0   => PCIE_RX0,
                  rx_in_rx_datain_1   => PCIE_RX1,
                  rx_in_rx_datain_2   => PCIE_RX2,
                  rx_in_rx_datain_3   => PCIE_RX3,
                  tx_out_tx_dataout_0 => PCIE_TX0,
                  tx_out_tx_dataout_1 => PCIE_TX1,
                  tx_out_tx_dataout_2 => PCIE_TX2,
                  tx_out_tx_dataout_3 => PCIE_TX3
            );
end;
```

ПРИЛОЖЕНИЕ Б

файл pcidriver-main.h

```
#pragma once
#include <linux/types.h>
struct nitta_driver_priv {
        u8 __iomem *hwmem;
        u8 __iomem *phwmem;
};
extern unsigned long mmio_start, mmio_len, p_mmio_start, p_mmio_len;
```

файл pcidriver-main.c

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/pci.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include <asm/uaccess.h> //put_user
#include "chardev.h"
//1172:e001
#define VENDOR_ID 0X1172
#define DEVICE_ID 0XE001
unsigned long mmio_start, mmio_len;
unsigned long p_mmio_start, p_mmio_len;
static struct pci_device_id nitta_driver_id_table[] = {
        { PCI_DEVICE(VENDOR_ID, DEVICE_ID)},
        {0, }
};
MODULE_DEVICE_TABLE(pci, nitta_driver_id_table);
static int nitta_driver_probe(struct pci_dev *pdev, const struct
pci_device_id *ent);
static void nitta_driver_remove(struct pci_dev *pdev);
static struct pci_driver nitta_driver = {
        .name   = "nitta_pci_driver",
        .id_table       = nitta_driver_id_table,
        .probe  = nitta_driver_probe,
        .remove = nitta_driver_remove
};
static int __init nitta_pci_driver_init(void)
{
        int out;
        printk(KERN_ALERT "Initialization\n");
        out = pci_register_driver(&nitta_driver);
        printk(KERN_ALERT "REGISTER: %d\n", out);
        return out;
}
static void __exit nitta_pci_driver_exit(void)
{
        pci_unregister_driver(&nitta_driver);
        printk(KERN_ALERT "Unregistered\n");
}
static int nitta_driver_probe(struct pci_dev *pdev, const struct
pci_device_id *ent)
{
        int bar, err;
        struct nitta_driver_priv *drv_priv;
        printk(KERN_ALERT "Probe function avaible!\n");
```

```
        err = pci_enable_device(pdev);
        printk(KERN_ALERT "ENABLE: %d\n", err);
        err = pci_enable_device_mem(pdev);
        bar = pci_select_bars(pdev, IORESOURCE_MEM);
        printk(KERN_ALERT "ERR: %d\n", err);
        printk(KERN_ALERT "bar: %d\n", bar);
        err = pci_request_selected_regions(pdev, bar, "NITTA PCI driver");
        printk(KERN_ALERT "ERR2: %d\n", err);
        pci_set_master(pdev);
        mmio_start = pci_resource_start(pdev, 0);
        mmio_len = pci_resource_len(pdev, 0);
        p_mmio_start = pci_resource_start(pdev, 1);
        p_mmio_len = pci_resource_len(pdev, 1);
        printk(KERN_ALERT "p_mmio_start: %x\np_mmio_len: %x\n",
                        p_mmio_start, p_mmio_len);
        drv_priv = kzalloc(sizeof(struct nitta_driver_priv), GFP_KERNEL);
        drv_priv->hwmem = ioremap(mmio_start, mmio_len);
        drm_priv->phwmem = ioremap(p_mmio_start, p_mmio_len);
        pci_set_drvdata(pdev, drv_priv);
        u16 bar0;
        pci_read_config_word(pdev, 18, &bar0);
        printk(KERN_ALERT "err: %d\nbar: %d\nmmio_start: %x\nmmio_len:
%x\nhwmem: %x\nbar0: %4x\n",
                err, bar, mmio_start, mmio_len, drv_priv->hwmem, bar0);
        create_char_devs(drv_priv);
        return 0;
}
static void nitta_driver_remove(struct pci_dev *pdev)
{
        struct nitta_driver_priv *drv_priv = pci_get_drvdata(pdev);
        if(drv_priv)
        {
                if(drv_priv->hwmem)
                {
                        iounmap(drv_priv->hwmem);
                }
                kfree(drv_priv);
        }
        destroy_char_devs();
        pci_release_regions(pdev);
        pci_disable_device(pdev);
}
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Chizhikov Ivan");
MODULE_DESCRIPTION("NITTA PCI module");
module_init(nitta_pci_driver_init);
module_exit(nitta_pci_driver_exit);
```

файл chardev.h

```
#ifndef NITTA_PCIE_CHARDEV
#define NITTA_PCIE_CHARDEV
#include <linux/ioctl.h>
#include "pcidriver-main.h"
#define NITTA_IOC_MAGIC 'a'
#define CLEAR_IOCTL _IOWR(NITTA_IOC_MAGIC,'a', unsigned long)
#define READ_FROM_NITTA _IOWR(NITTA_IOC_MAGIC, 'b', unsigned long)
#define WRITE_TO_NITTA _IOWR(NITTA_IOC_MAGIC, 'c', unsigned long)
#define GET_COUNT_TO_READ _IOWR(NITTA_IOC_MAGIC, 'd', unsigned long)
#define DOWNLOAD_ALGHORITHM _IOWR(NITTA_IOC_MAGIC, 'e', unsigned long)
int create_char_devs(struct nitta_driver_priv* drv);
int destroy_char_devs(void);
struct nitta_data
{
    uint32_t count;
    uint32_t *data;
};
struct nitta_algorithm
{
    struct nitta_data *modules;
    uint32_t count;
};
#endif /*NITTA_PCIE_CHARDEV*/
```

файл chardev.c

```
#include <linux/pci.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/kernel.h>
#include <linux/slab.h>
#include <linux/uaccess.h>
#include <linux/fs.h>
#include "chardev.h"
#include "pcidriver-main.h"
#define MAX_DEV 1
// Unique device number
static int dev_major = 0;
// Character Device Functions
static int nitta_dev_open(struct inode *inode, struct file *file);
static int nitta_dev_release(struct inode *inode, struct file *file);
static long nitta_dev_ioctl(struct file *file, unsigned int cmd, unsigned
long arg);
static ssize_t nitta_dev_read(struct file *file, char __user *buf, size_t
count, loff_t *offset);
static ssize_t nitta_dev_write(struct file *file, const char __user *buf,
size_t count, loff_t *offset);
static const struct file_operations nitta_dev_fops = {
        .owner = THIS_MODULE,
        .open = nitta_dev_open,
        .release = nitta_dev_release,
        .unlocked_ioctl = nitta_dev_ioctl,
        .read = nitta_dev_read,
        .write = nitta_dev_write
};
struct nitta_device_data {
        struct device* nitta_dev;
        struct cdev cdev;
};
// Class for creating a device in /sys
static struct class *nitta_devclass = NULL;
static struct nitta_device_data nitta_dev_data[MAX_DEV];
```

```
static struct nitta_driver_priv* drv_access = NULL;
static char *nitta_devnode(struct device *dev, umode_t *mode)
{
        if (!mode)
                return NULL;
        *mode = 0666;
        return NULL;
}
static int class_init(void)
{
        nitta_devclass = class_create(THIS_MODULE, "nitta_dev");
        /*if (IS_ERR(tty_class))
                return PTR_ERR(tty_class);*/
        nitta_devclass-> devnode = nitta_devnode;
        return 0;
}
int create_char_devs(struct nitta_driver_priv* drv)
        {
        int err, i;
        dev_t dev;
        // Memory allocation for devices
        err = alloc_chrdev_region(&dev, 0, MAX_DEV, "nitta_dev");
        // Assign major identifier for the device
        dev_major = MAJOR(dev);
        // Register a class called nitta_dev in sysfs
        class_init();
        // Creating devices
        for(i = 0; i < MAX_DEV; i++)
        {
                // A new device is identified with the file_operations
structure specified
                cdev_init(&nitta_dev_data[i].cdev, &nitta_dev_fops);
                nitta_dev_data[i].cdev.owner = THIS_MODULE;
                // Add a new character device to the kernel
                cdev_add(&nitta_dev_data[i].cdev, MKDEV(dev_major, i), 1);
                // create device /dev/nitta_dev-<i>
                nitta_dev_data[i].nitta_dev = device_create(nitta_devclass,
NULL, MKDEV(dev_major, i), NULL, "nitta%d", i);
        }

        drv_access = drv;
        return 0;
}
struct nitta_device_private {
        uint8_t chnum;
        struct nitta_driver_priv* drv;
};
static int nitta_dev_open(struct inode *inode, struct file *file)
{
        struct nitta_device_private* dev_priv;
        // Get the minor value from the current file system node
        unsigned int minor = iminor(inode);
        dev_priv = kzalloc(sizeof(struct nitta_device_private), GFP_KERNEL);
        dev_priv->drv = drv_access;
        dev_priv->chnum = minor;
        file->private_data = dev_priv;
        return 0;
}
static ssize_t nitta_dev_read(struct file *file, char __user *buf, size_t
count, loff_t *offset)
{
        printk(KERN_ALERT "MSG_Read\n");
```

```
        size_t i;
        u32 word;
        u8 byte;
        if(*offset >= mmio_len)
                return 0;
        if(*offset + count > mmio_len)
                count = mmio_len - *offset;
        struct nitta_device_private *dev_priv = file->private_data;
        u8 __iomem *adr = dev_priv->drv->hwmem;
        printk(KERN_ALERT "adr: %x, count: %d, offset*: %d\n", adr, count,
*offset);
        for(i = 0; i < count; i+=4)
        {
                word = ioread32((u32 *)(adr + i + *offset));
                byte = (u8)word;
                printk(KERN_ALERT "adr: %x\n", adr + i + *offset);
                printk(KERN_ALERT "READ: %X, %u, %s, %p\n", word, word, word,
word);
                {
                if(copy_to_user(buf + i/4, &byte, 1))
                        return -EFAULT;
                }
        }
        if (*offset == 0)
        {
        *offset += count;
        return count;
        }
        else
        return 0;
}
static ssize_t nitta_dev_write(struct file *file, const char __user *buf,
size_t count, loff_t *offset)
{
        printk(KERN_ALERT "MSG_Write\n");
        size_t i;
        size_t j;
        u8 byte;
        u32 word;
        if(*offset >= mmio_len)
                return 0;
        if(*offset + count > mmio_len)
                count = mmio_len - *offset;
        struct nitta_device_private *dev_priv = file->private_data;
        u8 __iomem *adr = dev_priv->drv->hwmem;
        for(i = 0; i < count; i++)
        {
                if(copy_from_user(&byte, buf +  i, 1))
                        return -EFAULT;
                word = (u32)byte;
                iowrite32(word, adr + *offset + i*4);
                printk(KERN_ALERT "word: %u, adr: %x\n", word, adr + *offset +
i*4);
                printk(KERN_ALERT "Записано в устройство: %u, %s adr, off, i -
> %x, %x, %d\n", word, word, adr, *offset, i);
        }
        *offset += count * 4;
        return count;
}
static long download_algorithm(u8 __iomem *adr, unsigned long arg)
{
        struct nitta_algorithm alg;
```

```
        __copy_from_user(&alg, arg, sizeof(struct nitta_algorithm));
        int i;
        int j;
        struct nitta_data data;
        uint32_t item;
        for(i = 0; i < alg.count; i++)
        {
        __copy_from_user(&data, arg, alg.modules + (i * sizeof(struct
nitta_data)));
                for(j = 0; j < data.count; j++)
                {
                        __get_user(item, data.data + j);
                        iowrite32(item, (u32 *) adr + (i * 100) + j);
                }
                iowrite32(data.count, (u32 *) adr + 1001 + i);
        }
        iowrite32(1, (u32 *) adr + 1000);
}
static long read_from_nitta(u8 __iomem *adr, unsigned long arg)
{
        uint32_t count = ioread32((u32 *)adr + 2);
        printk(KERN_ALERT "count: %u", count);
        printk(KERN_ALERT "enother string");
        struct nitta_data data;
        __copy_from_user(&data, arg, sizeof(struct nitta_data));
        if(count > data.count)
                count = data.count;
        int i;
        for(i = 0; i < count; i++)
        {
                __put_user(ioread32((u32 *)adr), data.data + i);
        }
        return count;
}
static long write_to_nitta(u8 __iomem *adr, unsigned long arg )
{
        uint32_t count = ioread32((u32 *)adr + 1);
        printk(KERN_ALERT "write count: %u", count);
        struct nitta_data data;
        __copy_from_user(&data, arg, sizeof(struct nitta_data));
        if(data.count < count)
                count = data.count;
        size_t i;
        uint32_t item;
        uint32_t count_in;
        for(i = 0; i < count; i++)
        {
                __get_user(item, data.data + i);
                printk(KERN_ALERT "data item: %u", item);
                count_in = ioread32((u32 *)adr + 1);
                printk(KERN_ALERT "before current count in: %u", count_in);
                iowrite32( item, (u32 *)adr );
                count_in = ioread32((u32 *)adr + 1);
                printk(KERN_ALERT "after current count in: %u", count_in);
        }
        return count;
}
static long get_count_to_read_from_nitta(u8 __iomem *adr)
{
        uint32_t count = ioread32((u32 *)adr + 2);
        printk(KERN_ALERT "count from nitta: %u", count);
        return count;
```

```
}
static long clear_nitta(u8 __iomem *adr)
{
        iowrite32(0, (u32 *)adr + 1);
        printk(KERN_ALERT "nitta clear");
        return 0;
}
static long nitta_dev_ioctl(struct file *file, unsigned int cmd, unsigned
long arg)
{
        printk(KERN_ALERT "ioctl here: cmd - %u, arg - %u", cmd, arg);
        struct nitta_device_private *dev_priv = file->private_data;
        u8 __iomem *adr = dev_priv->drv->hwmem;
        u8 __iomem *padr = dev_priv->drv->phwmem;
        int i;
        switch (cmd)
        {
        case CLEAR_IOCTL:
                return clear_nitta(adr);
        case READ_FROM_NITTA:
                return read_from_nitta(adr, arg);
        case WRITE_TO_NITTA:
                return write_to_nitta(adr, arg);
        case GET_COUNT_TO_READ:
                return get_count_to_read_from_nitta(adr);
        case DOWNLOAD_ALGHORITHM:
                return download_algorithm(padr, arg);
        default:
                return -1;
        }
        return 0;
}
static int nitta_dev_release(struct inode *inode, struct file *file)
{
        struct nitta_device_private* priv = file->private_data;
        kfree(priv);
        priv = NULL;
        return 0;
}
int destroy_char_devs(void)
{
        int i;
        for(i = 0; i < MAX_DEV; i++)
        {
                device_destroy(nitta_devclass, MKDEV(dev_major, i));
        }
        class_unregister(nitta_devclass);
        class_destroy(nitta_devclass);
        unregister_chrdev_region(MKDEV(dev_major, 0), MINORMASK);
        return 0;
}
```

файл Makefile

```
BINARY  := nitta_pcie_device
KVER    := $(shell uname -r)
KDIR    := /lib/modules/$(KVER)/build
C_FLAGS := -Wall
OBJECTS := \
               pcidriver-main.o \
               chardev.o \

ccflags-y += $(C_FLAGS)

$(BINARY)-y := $(OBJECTS)

obj-m += $(BINARY).o

all:
        make -C $(KDIR) M=$(PWD) clean
        make -C $(KDIR) M=$(PWD) modules

clean:
        make -C $(KDIR) M=$(PWD) clean

build:
        make -C $(KDIR) M=$(PWD) modules
install:
        insmod $(shell pwd)/$(BINARY).ko
uninstall:
        rmmod $(BINARY)
```

файл install.sh

```
make clean; sudo make uninstall; make build; sudo make install; make clean
lsmod | grep nitta
```

ПРИЛОЖЕНИЕ В

файл nitta_chardev.h

```
#ifndef NITTA_PCIE_USER_CHARDEV
#define NITTA_PCIE_USER_CHARDEV
#define NITTA_IOC_MAGIC 'a'
#define CLEAR_IOCTL _IOWR(NITTA_IOC_MAGIC,'a', unsigned long)
#define READ_FROM_NITTA _IOWR(NITTA_IOC_MAGIC, 'b', unsigned long)
#define WRITE_TO_NITTA _IOWR(NITTA_IOC_MAGIC, 'c', unsigned long)
#define GET_COUNT_TO_READ _IOWR(NITTA_IOC_MAGIC, 'd', unsigned long)
#define DOWNLOAD_ALGHORITHM _IOWR(NITTA_IOC_MAGIC, 'e', unsigned long)
struct nitta_data
{
    uint32_t count;
    uint32_t *data;
};
#endif
```

файл nitta_pci.h

```
#ifndef NITTA_PCIE_USER
#define NITTA_PCIE_USER
#include <stdint.h>
#include "nitta_chardev.h"
struct nitta_algorithm
{
    struct nitta_data *modules;
    uint32_t count;
};
int open_dev();
int close_dev();
int read_from_nitta(uint32_t *arr, const int count);
int write_to_nitta(const uint32_t *arr, const int count);
int clear_nitta();
int get_count_to_read();
int download_algorithm(struct nitta_algorithm *alg);
int add_to_algorithm(struct nitta_algorithm *alg, const uint32_t *arr, const
int count);
#endif
```

файл nitta_pci.c

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <stdlib.h>
#include "nitta_pci.h"
int nitta_dev = -1;
int open_dev()
{
    nitta_dev = open("/dev/nitta", 0);
    return nitta_dev;
}
int close_dev()
{
    return close(nitta_dev);
}
int read_from_nitta(uint32_t *arr, const int count)
```

```
{
    if(nitta_dev > 0)
    {
        struct nitta_data data_struct =
        {
            .count = count,
            .data = arr
        };
        return ioctl(nitta_dev, READ_FROM_NITTA, data_struct);
    }
    return -1;
}
int write_to_nitta(const uint32_t *arr, const int count)
{
    if(nitta_dev > 0)
    {
        struct nitta_data data_struct =
        {
            .count = count,
            .data = arr
        };
        return ioctl(nitta_dev, WRITE_TO_NITTA, data_struct);
    }
    return -1;
}
int get_count_to_read()
{
    if(nitta_dev > 0)
    {
        return ioctl(nitta_dev, GET_COUNT_TO_READ, 0);
    }
    return -1;
}
int clear_nitta()
{
    if(nitta_dev > 0)
    {
        return ioctl(nitta_dev, CLEAR_IOCTL, 0);
    }
    return -1;
}
int download_algorithm(struct nitta_algorithm *alg)
{   if(nitta_dev > 0)
    {
        return ioctl(nitta_dev, DOWNLOAD_ALGHORITHM, alg);
    }
    return -1;
}
int add_to_algorithm(struct nitta_algorithm *alg, const uint32_t *arr, const
int count)
{
    struct nitta_data *data = calloc(alg->count + 1, sizeof(struct
nitta_data));
    if(data == NULL)
    {
        return -1;
    }
    for(int i = 0; i < alg->count; i++) {
```

```
        data[i] = alg->modules[i];
    }
    data[alg->count].count = count;
    data[alg->count].data = arr;
    alg->modules = data;
    alg->count = alg->count + 1;
    return 0;
}
```

файл Makefile

```
SHARED_LIB := nitta_pci.so
C_FLAGS    := -Wall -fPIC
LD_FLAGS   := -shared
OBJECTS    := nitta_pci.c

all:
      $(CC) $(C_FLAGS) $(OBJECTS) -o $(SHARED_LIB) $(LD_FLAGS)
clean:
      rm $(SHARED_LIB)
```