Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

МЕХАНИЗМЫ ВВОДА-ВЫВОДА РЕКОНФИГУРИРУЕМОЙ ВЫЧИСЛИТЕЛЬНОЙ ПЛАТФОРМЫ РЕАЛЬНОГО ВРЕМЕНИ

Автор Емельянов Дмитрий Вячеславович _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность) 09.04.01
(код, наименование)
”Информатика и вычислительная техника”

Квалификация магистр
(бакалавр, магистр)*

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

К защите допустить

Руководитель ОП Платунов А.Е. профессор _____
(Фамилия, И.О., ученое звание, степень) (Подпись)

“_____” _____ 2019 г.

Санкт-Петербург, 2019 г.

Оглавление

Введение

Актуальность темы исследования. В настоящее время наблюдается рост потребности во встраиваемых вычислительных система, которые удовлетворяют требованиям работы в режиме реального времени, это ставит перед разработчиками задачи по совершенствованию методов и подходов в проектировании встраиваемых вычислительных систем [1]. Часто основным вычислительным устройством является процессор общего назначения, который позволяет решать большую часть типовых вычислительных задач, но не всегда эффективно позволяет реализовать прикладной алгоритм по заранее заданной математической модели с требуемой производительностью. Для решения подобных прикладных задач используются программируемые логические интегральные схемы (ПЛИС) с применением реконфигурируемых вычислительных систем [2], а также системы на кристалле (СнК).

Степень теоретической разработанности темы. По теме работы существует большое количество различных САПР HLS, имеющих готовые IP ядра для работы с приёмом и передачей данных между устройствами, это, например, Vivado HLS или Intel FPGA Compiler.

Целью работы является разработка подсистемы ввода-вывода для реконфигурируемой вычислительной платформы реального времени NITTA.

Для достижения поставленной в работе цели необходимо решить следующие задачи:

- Исследование направлений применение вычислительной платформы NITTA и определения критериев выбора и требований к выбранным интерфейсам передачи данных.
- Разработка вычислительных блоков на основе последовательных интерфейсов передачи данных SPI и I2C.
- Обеспечение поддержки вычислительных блоков в составе САПР, заключающееся в создании функциональных моделей вычислительных блоков.

- Разработка тестового окружения для верификации вычислительных блоков в составе вычислительной платформы NITTA.

Область исследования.

Областью исследования является системы ввода-вывода и реализация в вычислительных системах на базе ПЛИС, относящиеся к классу реконфигурируемых вычислительных систем.

Объект исследования. Объектом исследования является подсистема ввода-вывода вычислительной платформы NITTA.

Предмет исследования. Предметом исследования является подход в проектирования встраиваемых вычислительных систем на основе реконфигурируемой вычислительной платформы реального времени с применением высокоуровневого синтеза HLS.

Теоретическая и методологическая основа исследования.

Теоретическую и методологическую основу исследования составили научные труды в области проектирования встраиваемых вычислительных систем на основе применения высокоуровневого синтеза.

Научная новизна исследования. Научная новизна работы заключается в маршруте проектирования, предложенных и реализованных описаниях для протокола прикладного уровня, предложенных средств и методах функциональной верификации и позволяет протестировать функционал разрабатываемой системы с меньшими издержками при разработке.

Вычислительная платформа NITTA применяется для задач акселерации вычислительных задач, где основной целью вычислений является быстрое выполнение решения вычислительной задачи.

Практическая значимость исследования. Результаты данного исследования могут быть применены в системах с недетерминированной окружающей средой, для взаимодействия с объектами управления, а также использоваться в направлении взаимодействия сигнальный процессор с вычислительной платформой NITTA.

Апробация результатов исследования.

Результаты работы используются для задач HIL и PIL тестирования, на примере реализации ПИД – регулятора в проекте Реконфигурируемая вычислительная платформа реального времени NITTA. Результаты исследования были представлены на VIII Конгресс молодых ученых (2019) и XLVIII научная и учебно-методическая конференция Университета ИТМО (2019).

Объём и структура работы.

Работа содержит три главы, первая глава посвящена обзору вычислителей специального назначения и использованию высокоуровневого синтеза в задачах проектирования для ПЛИС, вторая глава посвящена описанию подсистемы ввода-вывода для вычислителей специального назначения на базе ПЛИС, третья глава посвящена разработке и апробации разработанного подсистемы ввода-вывода.

Глава 1. Средства разработки специализированных вычислителей

Применение ПЛИС с использованием инструментальных средств разработки представляют собой отдельный подход в проектировании реконфигурируемых вычислительных платформ [3]. Это выражается в создании требуемой реализации прикладного алгоритма на ПЛИС под задачу с заданными требованиями к системе. Отличие от традиционной модели проектирования состоит в возможности реализации архитектуры вычислительной платформы в зависимости от технических требований, предъявляемых платформе для решения поставленной задачи.

Средства разработки специализированных вычислителей представляет собой использование готовых продуктов, представленных на рынке, так и прикладных программ, ориентированных на решение проектирования реконфигурируемых вычислительных платформ. Система автоматизированного проектирования (САПР) включает все этапы разработки вычислительной платформы на базе использования ПЛИС, включая процесс синтеза, имплементации, моделирования и генерации прошивочного файла. Одной из таких средств разработки является САПР NITTA, которая делится на два направления разработки реконфигурируемой вычислительной платформы:

1) Описание организации вычислительного процесса, которое включает генерацию программного обеспечения, в соответствии с алгоритмом описанном на языке высокого уровня и заданной микроархитектуры вычислительной платформы в эквивалентные модули на языке описания аппаратуры (RTL, Register-Transfer Level) [4].
2) Описание анализа и синтеза сгенерированной микроархитектуры вычислителя и генерация прошивки для ПЛИС. Вычислительная платформа основана на базе программируемой логической интегральной схемы от фирмы Altera семейства Cyclone IV. В качестве платформ использовались платы De-Nano и отладочная плата с PCI Express.

Фирма Altera разработала программную среду Quartus с огромным количеством разнообразных пакетов, используемых при написании выпускной квалификационной работы (ВКР), такие как, среда моделирования ModelSim, готовые IP ядра. Quartus выполняет задачи синтеза и имплементации и позволяет использовать инструмент Timing Analyzer для нахождения внутренних связей внутри ПЛИС, имеющий задержки по передачи данных от точки к точке, и устранениях их с помощью перепроектирования схемы или задания временных ограничений для конкретного участка схемы.

1.1 Реконфигурируемые вычислительные платформы

В настоящее время для решения специализированных задач применяются вычислительные системы на основе процессора общего назначения. Подобные вычислительные системы имеют ограниченные возможности в решение узконаправленных задач из-за последовательности работы устройства и наличия определенного набора команд, на примере архитектур ARM, MIPS, RISC-V. Это накладывает ограничения на межпроцессорное взаимодействие и обмен данными между потоками и процессами, при этом позволяя выполнить прикладной алгоритм с ограниченными характеристиками системы [7]. Основная причина малой производительности выполнения алгоритма - это несоответствие архитектуры вычислительного устройства с информационной структурой решаемых задач. Применение реконфигурируемых вычислительных систем позволяет динамически перестраивать архитектуру процессора под решаемую задачу [5]. С использованием ПЛИС можно создавать вычислительные системы на основе соотношения текущая производительность и пиковая производительность. В качестве основополагающей базы для ПЛИС является создание базовых модулей, объеденных в общую структуру, позволяющие ускорить процесс проектирования и тестирования моделей объекта управления. Тестирование объекта управления при использовании сокращает время цикла разработки, за

счет изменения программной составляющей микрокода без необходимости повторного синтеза аппаратной составляющей вычислителя.

PIL тестирование подразумевает замену объекта управления в системе реконфигурируемой вычислительной платформой [8]. Взаимодействие с объектом управления происходит с использованием портов ввода-вывода. HIL тестирование подразумевает замену систему управления в системе реконфигурируемой вычислительной платформой, реализация прикладного алгоритма управления на реконфигурируемой платформе реального времени позволяет подключиться к существующему объекту управления через интерфейсы передачи данных SPI и I2C. По результатам прототипирования проводится анализ результатов и делается верификация о правильности работы алгоритма с реальным оборудованием, так и определения необходимые требования к целевой платформе для его реализации. Рассмотрим на примере регулировании температуры чашки с помощью ПИД регулятора, представленного на рисунке 1.1.

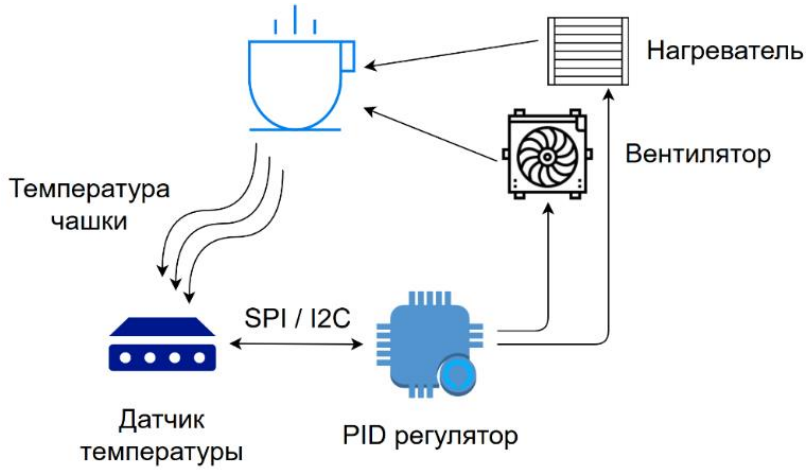

Рисунок 1.1 Стабилизация температуры чашки PID регулятором.

ПИД – регулятор вычисляет рассогласование от установленного значения температуры и реальной температуры и по полученному коэффициенту выполнит одно из двух действий, если коэффициент меньше нуля и имеет отрицательное значение, то нагревательный элемент выключается, при положительном значении включается нагревательный элемент. При HIL тестировании на реконфигурируемой вычислительной платформе реализуется

алгоритм работы объекта управления. На подсистемы ввода вывода заводятся входы и выводы реализуемой модели. В рамках текущей задачи, за ввод вывод отвечают интерфейс SPI или I2C, выполняющие роль входа и снимающий показания с датчика температуры, а выходом служит ШИМ сигнал. Поскольку модель работает в режиме жесткого реального времени, все её входы и выходы работают практически также, как входы и выходы реального объекта. Это позволяет обеспечить взаимодействие с вычислительной платформы с объектом управления и выполнить верификацию работы в условиях среды, максимально близких к рабочим. PIL тестирование позволяет реализовать алгоритм управления на реконфигурируемой вычислительной платформе реального времени для подключения к тестируемому объекту управления, рисунок 1.2. Управление объектом при этом идёт через те же интерфейсы, что будут использоваться при подключении рабочей системы.

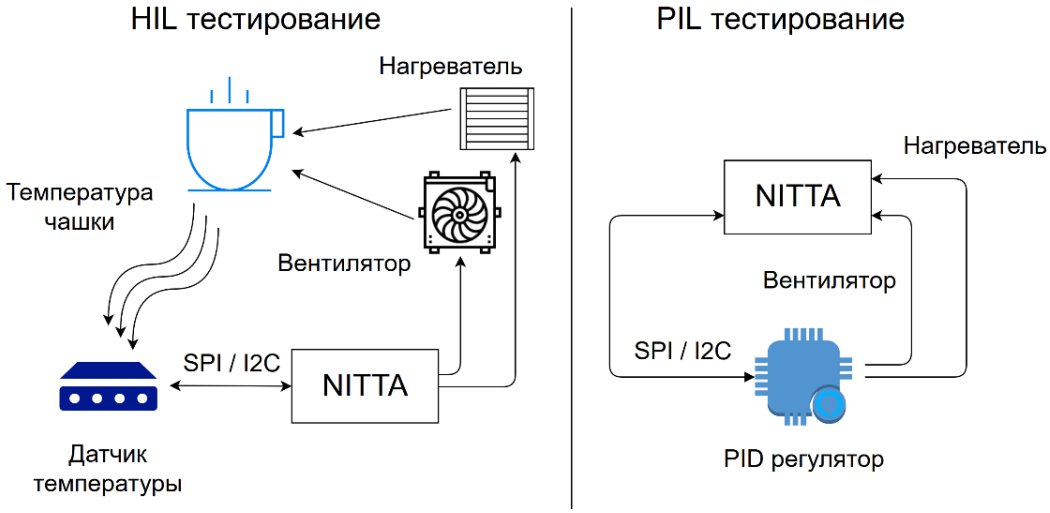

Рисунок 1.2 HIL и PIL тестирование для терморегуляции чашки

PIL тестирование применимо тогда, когда объект управления доступен для экспериментов, и при этом алгоритмы управления требуют тонкой настройки. Моделирование динамических процессов требуют затрат по времени исполнения и энергопотреблении. Использования реконфигурируемой вычислительной платформы позволяет ускорить процесс системно-динамического моделирования за счёт использования специализированных вычислителей, что в свою очередь накладывает ограничения на пропускную

способность интерфейсов передачи данных SPI и I2C, ограничивающие целостность передачи данных от вычислителя к управляющему устройству. Реализуется поддержка PCI Express, обеспечивающий требую пропускную способность для моделирования. HIL тестирование на реконфигурируемой вычислительной платформе позволяет избежать экспериментов на реальном оборудовании.

1.2 Подходы проектирования специализированных вычислителей

Проектирование специализированных вычислителей состоит из трех уровней организации вычислительного процесса: устройство ввода, устройство вывода и основной логики, выполняющий роль вычислителя. Вычислителем могут выступать как микропроцессора, микроконтроллеры, так и ПЛИС. Реализация алгоритмов на любом вычислителе занимает большое количество времени по отношению к времени передачи данных, поэтому основная нагрузка приходится на сам вычислитель и платформу на которой он реализован. При решении задач HIL и PIL тестирования вычислительная платформа ограничена временем исполнения прикладного алгоритма, это требует анализа различных вариантов архитектурной организации и целевой системы. Для создания реконфигурируемой вычислительных устройств подходит использование ПЛИС.

Спецификация реконфигурируемой вычислительной платформы состоит в требованиях работы в многозадачности вычисления в условиях реального времени. В разработках вычислительной аппаратуры широко используются компиляторы под конкретные процессорные ядра со своей системой команд, а также инструменты для высокоуровневого синтеза. Предоставленный набор программного обеспечения предоставляет для проектировщика специализированных вычислителей ограничений набор инструментов, на уровне системы команд процессора.

На практике используются четыре подхода к проектированию специализированных вычислителей [9].

1) Промышленные HDL дизайны.
2) Расширенные HDL дизайны.
3) Языки высокого уровня.
4) Проблемно-ориентированные языки.

В контексте реконфигурируемой вычислительной платформы NITTA используется подход в проектировании расширенного HDL дизайна. Данный подход позволяет абстрагироваться от описания аппаратуры на уровне регистровых передач, позволяет внедрять пользовательский функционал, реализовывать специализированную модель вычислений. По сравнению с другими подходами проектирования, данный подход не накладывает ограничений на использование и применение пользовательских моделей исполнения. Выполнение задач тестирования и расчета моделей системной динамики выполняются за счет взаимодействия специализированных единичных вычислительных блоков, обеспечивающей выполнение работы одной задачи. Такая архитектура позволяет добиться реконфигурируемой вычислительной платформы, высокой производительности и минимальными накладными расходами. В основе вычислительной платформы NITTA лежат такие принципы методологии проектирования вычислительных устройств, как: модель-ориентированная инженерия, высокоуровневый синтез и совместное проектирование. Вычислительная платформа NITTA представляет собой связку двух решений: NISC и TTA. NISC организую модель вычислительного процесса за счет использования микрокоманд (совокупности управляющих сигналов), TTA определяет подход к разработке вычислительной системы на основе использования отдельных вычислительных блоков, выполняющих одну простую задачу и взаимодействующих между собой напрямую, без использования промежуточной логики. Основой вычислительного процесса NITTA является обмен данными между вычислительными блоками, под управление микрокоманд.

1.2.1 Проектирование интегральных схем с использование HDL

Программируемые логические интегральные схемы представляют в упрощенном виде набор большого количества элементов в базисе 2И – НЕ. Процесс компиляции, применительно к ПЛИС, разделяется на три логических этапа. Первый этап отвечает за синтез схемы к абстрактному виду, когда компилятор минимизирует и оптимизирует схемы и по закону де Моргана приводит схему к базису 2И-НЕ, выстраивает логические связи и на выходе получаем эквивалентную схему, именуемую как Netlist. Второй этап имплементация схемы в структуру ПЛИС, где происходит размещение и разводка. Третий этап генерация прошивки, данный маршрут представлен на рисунке 1.3. Архитектуры ПЛИС, на примере Cyclone IV, в действительности содержит структуру намного сложнее. Структура ПЛИС состоит из крупных блоков, которые в свою очередь содержат несколько логических элементов. Логический элемент в своей структуре содержит динамически управляемый D-триггер и одноразрядный 4-х адресный ПЗУ, которая позволяет реализовать любую комбинационную схему на 4 входа, настраиваемая при помощи мультиплексоров. Все блоки соединяются цепями коммутации, которые позволяют соединять в различных комбинациях блоки и реализовать любую пользовательскую схему.

Так же микросхемы ПЛИС в своей архитектуре содержат готовые модули для работы с умножением, делением, и делителем частоты (PLL). На текущий момент существуют два типа ПЛИС, это FPGA и CPLD. FPGA характеризуется большой емкостью логических элементов, но меньшей производительностью. CPLD имеет противоположные характеристики FPGA, основное отличие — это внутренняя архитектура микросхем. ПЛИС позволяет использовать на своей микросхеме несколько независимых друг от друга пользовательских схем, что обеспечивает параллельность обработки данных на аппаратном уровне в отличие от микроконтроллеров, которые представляют собой последовательное устройство, выполняющее команды последовательно.

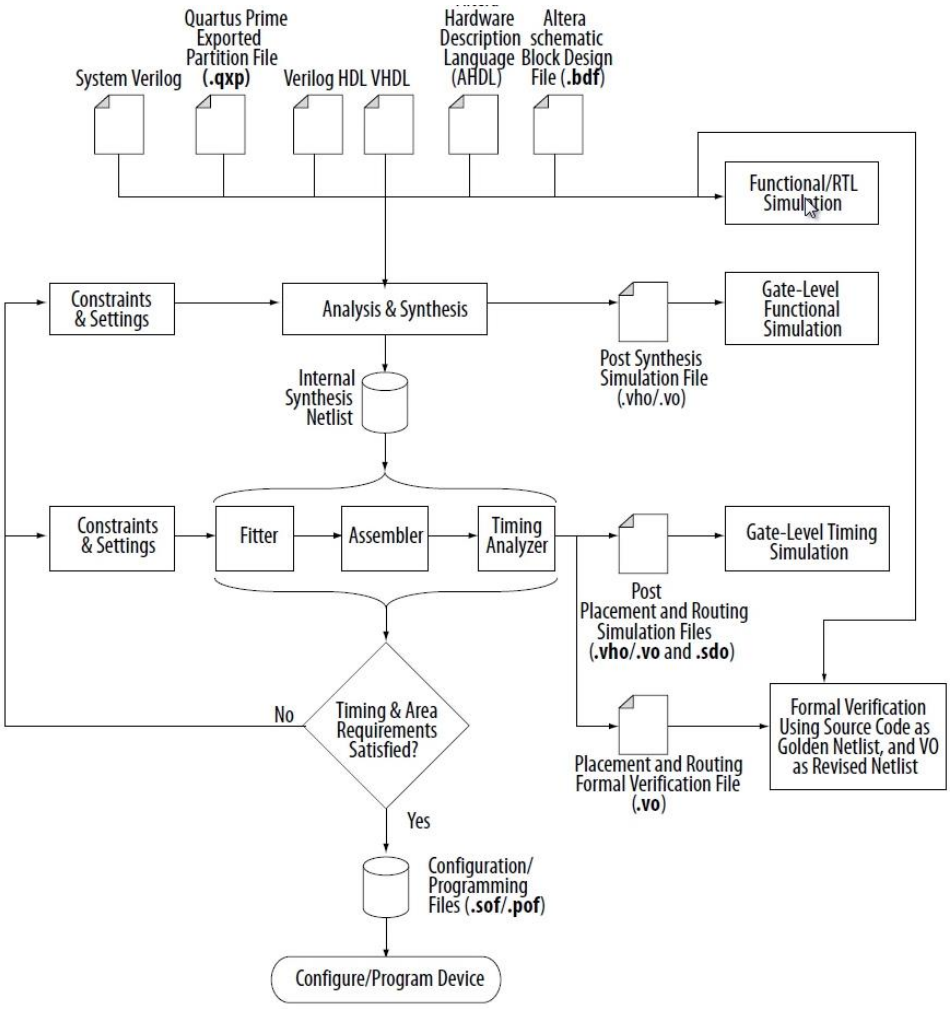

Рисунок 1.3 Маршрут проектирования под ПЛИС

Из базовых логических вентилей создаются триггеры, ячейки памяти, счетчики. Разработка интегральных схем позволяет детально прорабатывать каждый блок схемы и реализовать вычислительный процесс вплоть до такта. Данный подход накладывает ограничения на компетентность разработчика и требует знания цифровой схемотехники. Существует три способа ввода пользовательских схем: редактирование напрямую в интегральной схеме с использованием специализированных редакторов, графическое рисование схем и текстовое описание на языках проектирования схем. Два последних способа являются основными при проектировании пользовательских схем для ПЛИС. Графическое проектировании интегральных схем перестал применяться со временем, это связано с ростом сложности проекта, заниматься разработкой на уровне вентилей и их отладкой становится все сложнее. На данный момент для

разработки интегральных схем используются языки HDL (Hardware Description Language) VHDL и Verilog. Использование HDL языков позволяет решить две задачи:

1) Проектирование пользовательских схем на более высоком уровне абстракции, в сравнении с графическим подходом.
2) Моделирование проектируемой схемы для верификации требуемой логики работы.

Изначально данные языки использовались исключительно для отладки и тестирования схем, но со временем они позволили описывать поведение схем и их логическую структуру, что в последствии стало стандартом в области проектирования интегральных схем. Все технические решения в рамках проекта NITTA выполняются на языке описания аппаратуры Verilog. Verilog имеет два подмножества конструкций: синтезируемые конструкции и не синтезируемые конструкции. Синтезируемые подмножество конструкций используется для синтеза пользовательских схем в файл прошивку, загружаемую в ПЛИС, не синтезируемое подмножество используется для моделирования и отладки схем на начальном этапе проектирования. Структурно синтезируемое подмножество имеет три описания: структурное, потоковое, поведенческое [10].

Каждое описание предоставляет разработчику свой уровень абстракции над проектирование интегральных микросхем. Структурное описание структуру схемы на уровне вентилей, представляющий собой самый низкий уровень абстракции при описании интегральных схем. При аналогии его можно сравнить как транслятор исходного кода Ассемблер с языком программирования C. Потоковое и поведенческое используются чаще всего, и предоставляют боле высокий уровень абстракции. Предоставляет конструкции из языков программирования C и Pascal, позволяющие писать выражения и задавать алгоритм поведения схемы на конструкциях и операторах.

1.2.2 Проектирование ПЛИС с использованием HLS

С ростом сложности реализации прикладных алгоритмов управления на ПЛИС, растет сложность отладки и возможности дальнейшего поддержания и улучшения интегральных схем. ПЛИС позволяет реализовывать специализированные вычислители под конкретную задачу, использовать интерфейсы с унифицированным протоколом передачи данных. Стандартом для проектирования на ПЛИС на сегодняшний день являются языки описания аппаратуры Verilog и VHDL, что требует от разработчика знания цифровой схемотехники и навыков использования САПР под ПЛИС. С учетом разнообразия фирм, занимающихся разработкой интегральных схем, каждая компания разрабатывает свои программные обеспечения под свои семейства интегральных схем, внося дополнительные временные затраты и унификацию специализации для проектирования на ПЛИС. Направление, решающая ряд задач по проектированию на ПЛИС, является применение высококровного синтеза (HLS). HLS задействует функционал компиляции, транслируя описание прикладного алгоритма на языке высокого уровня с язык описания аппаратуры, с учетом использования целевой платформы.

HLS предоставляет доступ разработчику для воздействия на поведение имплементированной интегральной схемы, посредство за действия функционала САПР, с учетом требований к требуемой задаче. Это позволяет решить ряд задач, связанных с временными затратами на проектирование ПЛИС традиционными способами, снизить временные издержки, варьировать архитектурой вычислителя на поздних этапах разработки. Высокоуровневый синтез сокращает время тестирования за счет использования функционала непрерывной интеграции, достигаемого за счет быстрого моделирования, повторного исполнения тестового окружения и верификации интегральной схемы на уровне RTL. Применение САПР под ПЛИС несет в себе ряд факторов, которые скрыты и неявно поданы разработчику для отладки RTL, это:

- Сложность работы САПР.

- Непрозрачность работы САПР.
- Нестабильность САПР.

Это выражается в некорректном отображении прикладного алгоритма в RTL, что требует глубокие знания и понимание работы определенного САПР. Применение HLS накладывает ограничения на отображение прикладного алгоритма описанного на языке высокого уровня в описание RTL, на примере Vivado HLS для разработчика предлагается описывать прикладной алгоритм на C++. Vivado HLS ограничен подмножествами языка C++ и не поддерживает стандарты выше C++98 ISO/IEC 14882.

1.2.3 Реконфигурируемые вычислительные платформы

Разработка на ПЛИС позволяет за малый промежуток времени реализовать самые сложные вычислительные процессы, это достигается за счет изменение конфигурации прикладного алгоритма на ПЛИС. Реконфигурируемые вычислительны платформы используются для задач компьютерного зрения, цифровой обработки сигналов, коммуникации с другими устройствами и верификации.

В сравнение с традиционными вычислительными системами, применение ПЛИС содержит ряд преимуществ над традиционными вычислительными устройствами: обеспечение высоких показателей скорости работы за счет проектировании на RTL уровне, конфигурация в конкретный момент времени вычислительной платформы, удовлетворяющей требованиям задачи, в отличие от программной реализации на другом аппаратном обеспечении. ПЛИС может применяться для связи аппаратно-программных систем, где на аппаратном уровне реализуется универсальный прикладной алгоритм и по высокоскоростному интерфейсу передачи данных взаимодействует с программной частью. Применение реконфигурируемых вычислительных систем имеет ряд преимуществ перед традиционными способами выполнения прикладного алгоритма:

1) Скорость работы. ПЛИС поддерживают работу прикладного алгоритма на частотах от ГГц. Разработка ведется на RTL уровне, где разработчик способен изменять дизайн на уровне триггеров, что позволяет оптимизировать работу прикладного алгоритма на заданной частоте. Этот так же имеет недостаток, связанный с применением ПЛИС нового поколения, по сравнению с предшественниками.
2) Прозрачность проектирования. Реконфигурируемая вычислительная платформа позволяет избежать инкапсуляции системы, предоставляя пользователю данные и методы о системы, не скрывая детали реализации и позволяя гибко настраивать через графический интерфейс САПР.
3) Повторное использование IP ядер.
4) Структурно ПЛИС обладает двумя важными свойствами: параллельность и реконфигурируемость. Архитектура ПЛИС позволяет выполнять параллельное выполнение операций, что повышает производительности выполнения прикладного алгоритма. Так же производительность определяется увеличение тактовой частоты. Конфигурируемая вычислительная платформа работает в связке с HOST компьютером, обмениваясь данными по стандартным интерфейсам передачи данных: PCI, USB или RS-485.

Реконфигурируемая вычислительная платформа представляет собой набор интегральных схем ПЛИС расположенных на плате, чаще всего от одного производителя, и работает в связке с периферией: дополнительные модули памяти Flash, ППЗУ, SDRAM, датчики, интерфейсы передачи данных – USB, Ethernet, PCI, GPIO. Применение реконфигурируемых вычислительных платформ предполагает снизить нагрузку на HOST компьютере и перенести функционал прикладного алгоритма на специализированный вычислитель. Это позволяет за короткий промежуток времени реализовать целевой алгоритма и увеличить скорость обработки данных в разы.

1.3 Вычислительная платформа NITTA

В основе вычислительной платформе NITTA лежит принцип из модельно-ориентированной инженерии, заключающийся в понимании работы отдельных узлов системы, и как следствие модели вычислений всей системы как целого. Один центральных вычислительных блоков выполняет роль управление всем вычислительным процессом. Он представляет собой блок содержащий указатель на команду и буфер с микрокомандами. Микроархитектура объединяет две идеологии проектирования вычислительных систем, это Non Instruction Set Computing (NISC) и Transport Triggered Architecture (TTA) подход.

- NISC представляет собой подход, в котором не используется система команд ISA, как в традиционных процессорах общего назначения от Intel или AMD, а применяется набор горизонтальных команд, представленных как набор управляющих сигналов, где один управляющий сигнал отвечает, к примеру, только за запись в буфер обмена или считывании данных из блока ввода-вывода. На такой идеологии построен вычислительный процесс NITTA.
- TTA определяет подход взаимодействия между вычислительными блоками напрямую друг с другом, не используя промежуточные места хранения и обработки.

Основные особенности вычислительной платформы NIITA включают простую и понятную модель вычислений, взаимодействие вычислительных блоков по общей шине и изменяющие свое поведение под действие микрокоманд от управляющего блока. Платформа NITTA использует трансляцию модулей программы написанных на языке высокого уровня Lua в уровень регистровых передач через свой компилятор, написанный на функциональном языке программирования Haskell. Это позволяет отказаться от применения САПР от различных производителей и управлять процессом синтеза прикладного алгоритма на разных этапах проектирования.

Управление осуществляется через интерфейс графического интерфейса, разработанного на основе библиотеки React и развернутого на локальном домене. В качестве узлов системы выступают вычислительные блоки, между которыми происходят транзакции передачи данных и управляющих сигналов. В архитектуре NITTA вычислительные блоки делятся на следующие категории:

1) Вычислительные блоки, выполняющие роль управления моделью вычислительного процесса.
2) Вычислительные блоки, отвечающие за выполнение математических операций.
3) Вычислительные блоки, отвечающие за ввод-вывод вычислительной платформы.

Модель вычисления NITTA содержит один управляющий блок, в котором содержится список микрокоманд и счетчик команд, указывающий на выполнение текущей микрокоманды. Микрокоманды распространяются по шине данных и заводятся на вычислительные блоки, определенные в процессе синтеза прошивки. Шина данных представляет собой набор управляющих сигнальных линий управляющие отдельными вычислительными блоками. Вычислительный блок (Unit) выполняет одну простую задачу, сложение, вычитание, деление, блок ввода вывода, память, управляющие блоки. Каждый блок имеет стандартный интерфейс для взаимодействия с системой, это сигналы для чтения и записи, шины данных, сигнала сброса данных и тактовый сигнал. Так в зависимости от назначения блока появляются дополнительный интерфейс необходимый для корректной работы вычислительно блока, это может сигнал завершения вычислительного цикла, сигнал завершения приема-передачи данных через блоки ввода-вывода.

1.4 Постановка задачи

В настоящее время для решения прикладных задач используются процессоры общего назначения, микроконтроллеры и СБИС, обладающими рядом достоинств и недостатков. Одним из недостатков является платформа - зависимое исполнение прикладного алгоритма, накладывая издержки по времени и эффективности исполнения. Технология ПЛИС позволяет на основе своей архитектуры реализовывать реконфигурируемые и не платформ зависимые решения для задач акселерации вычислительных задач и HIL, и PIL тестирования. Центральным управляющим устройством может выступать вычислительная платформа NITTA, принимая по входным портам ввода-вывода данные с датчиков и отправляю управляющие сигналы на объекты управления, на основе алгоритма, заданного при проектировании. Поскольку модель работает в режиме жесткого реального времени, все её входы и выходы работают практически также, как входы и выходы реального объекта. Это позволяет подключить к машине реального времени натурный образец системы управления и испытать его в условиях, максимально близких к рабочим. Для решения поставленных задач необходим интерфейсы передачи данных с требуемой пропускной способностью, удовлетворяющим требованиям прикладного алгоритма. Для достижения поставленной цели разрабатывается библиотечные параметрические модули описанных на языке Verilog для обработки входных данных со стороны процессора NITTA и со стороны внешних устройств на основе интерфейсов SPI и I2C для задач HIL и PIL тестирования, где считывание и обмен данными протекают с периодом от секунды и системной динамикой, где скорость передачи данных сравнима с временем генерации полезной информации.

Глава 2. Подсистема ввода-вывода

ПЛИС позволяет реализовать стандартные интерфейсы передачи данных, благодаря наличию большого числа блоков ввода-вывода, настраиваемых в зависимости от интерфейса. Каждый вывод ПЛИС может быть настроен на вход, выход и двунаправленный вывод. Совокупность выводов образуют подсистему ввода-вывода, обеспечивающий связь между внешними устройствами и внутренней логической схемой ПЛИС. Подсистема ввода-вывода делиться на отдельные структурные блоки, именуемые как I/O Bank, где содержится отдельный набор вывод. Каждый вывод характеризуется по следующему ряду показателей:

- Node Name – название порта поименованного на языке описания аппаратуры.
- Direction – направление вывода на передачу данных, может быть сконфигурирован как вход, выход и двунаправленный вывод.
- Location – именованный вывод, характеризующий физическую ножку ПЛИС. Обозначается как PIN_C3, где “C” название порта, а “3” номер ножки ПЛИС.
- I/O Bank – номер Bank’а к которому относится вывод.
- I/O Standard – определяет уровень сигналов, при котором ПЛИС считывает логическую единицу и логический ноль, заданному стандарту.
- Current Strength – задает уровень тока, проходящего через вывод.

На текущий момент в ПЛИС используются два стандарта ввода-вывода, это стандарт, основанный на биполярных транзисторах (TTL) и технология КМОП, где используются полевые транзисторы. Технология на биполярных транзисторах работает с большими входными токами, по сравнению с технологией на полевых транзисторах, где входные токи значительно меньшими и является более экономичным решением. Процессорная топология ПЛИС позволяет реализовать фиксированную периферию, такую как: USB,

CAN, I2C, SD, UART, SPI, GPIO. Так же обеспечивает доступ к памяти, дисплеям через интерфейс VGA и реализовать параллельную шину данных PCI.

2.1 Архитектура ввода-вывода ПЛИС

Реконфигурация вычислительной платформы обеспечивается за счет логический ячеек, именуемых как запоминающее устройство, распределенных по все поверхности ПЛИС. В зависимости от семейства ПЛИС и производителя, число логических ячеей варьируется от 500 до 100 000 и больше. В основе любой интегральной микросхемы ПЛИС лежит набор из ключевых компонентов, из которых состоит микросхема ПЛИС, это группы логических элементов, межсоединения системы, блок тестирования TAP контроллер, выполненный в соответствии с стандартом IEEE Std. 1149.1-1990, и блок программирования. Матричная система интегральной схемы содержит вокруг себя набор выводов, объеденных в группы и выполняющие заданную роль. Функционально назначение вывод ПЛИС можно разделить по назначению: служебный вывод для тактового сигнала, Пользовательский программируемый ввод-вывод, вывод VCC и GND, служебные выводы для JTAG тестирования.

Физическая контактная ножка ПЛИС подключена через подтягивающие резисторы к буферу и триггеру. Подтягивающие резисторы обеспечивают снижение помех при переключениях и в случае нахождения ножки ПЛИС без соединения с другим физическим контактом, обеспечивает задание значения по умолчанию. Подтяжка к питанию соответствует логической единице (pull-up), подтяжка к нулю логическому нулю (pull-down). Рассматривая на примере используемой платы DE0-Nano, имеющей ПЛИС от фирмы Altera семейства Cyclone, то данная ПЛИС содержит двунаправленный буфер ввода-вывода и регистры для настройки скорости передачи. Блоки ввода-вывода расположены по периферии Cyclone IV. Каждый такой блок поддерживает режим ввода, вывода и двунаправленный вывод.

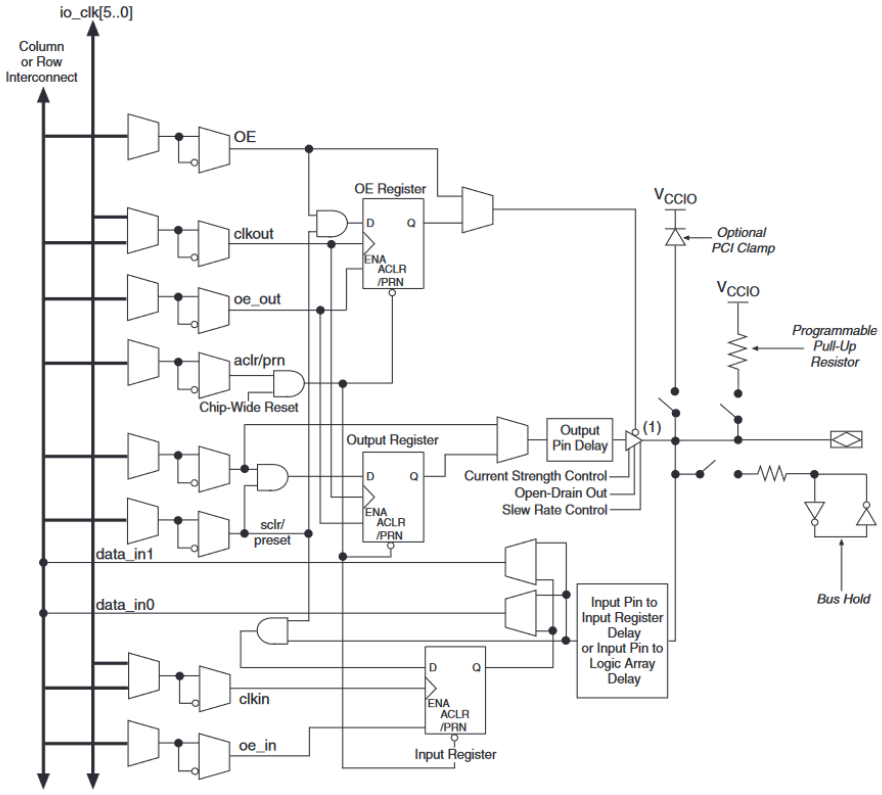

Рисунок 2.1 Структура элемента ввода-вывода ПЛИС семейства Cyclone

Источник: Cyclone FPGA Family Data Sheet, стр 50

На рисунке 3 показана структура Cyclone IV в конфигурации двунаправленного ввода-вывода. Как видно на структуре, элемент ввода-вывода имеет три регистра:

- OE Registor используется как сигнал разрешения входа.
- Output Registor обеспечивает меньшее время формирования сигналов на регистровом выходе.
- Input Register обеспечивает малое время установки.

Блоки ввода-вывода формируется в ряды или колонки и соединяются через шину к I/O Block Local Interconnect и LAB Local Interconnect. Со стороны ПЛИС идет ряд сигнальных линий, именуемых как Logic Array, управляющие блоком Data and Control Signal Selection, данная структура представлена на рисунке 4. После формируются управляющие сигналы для блоков ввода-вывода, где каждый блок IO содержит набор управляющих сигналов: oe, ce_in, ce_out, aclr / preset, sclr / preset, clk_in и clk_out. В IOE есть два пути для

комбинационного входного вывода для соединения с логическим массивом, где каждый путь имеет задержку.

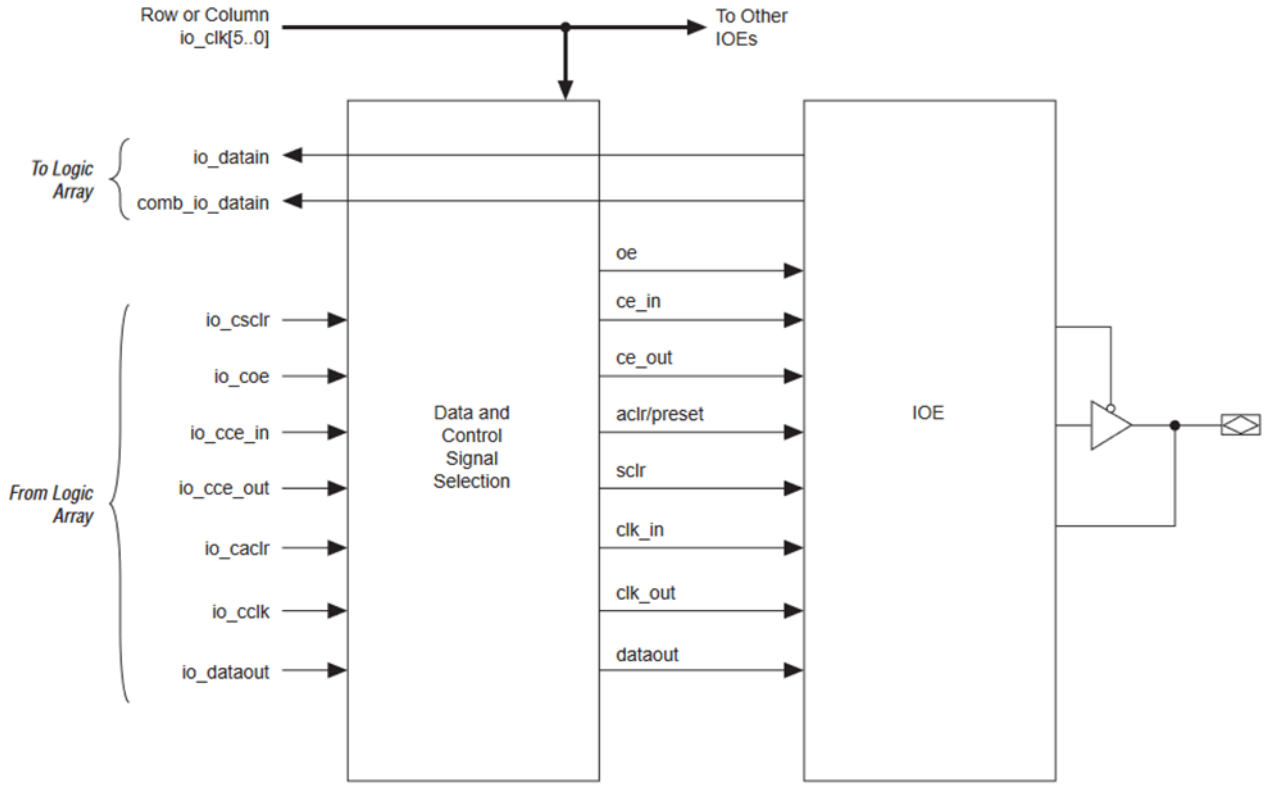

Рисунок 2.2 Путь управляющих сигналов через блок ввода-вывода [11]

Источник: Cyclone FPGA Family Data Sheet, стр 48

Это позволяет разработчику настраивать задержки от вывода до внутреннего регистра LE, который находится в двух разных областях устройства. Разработчик устанавливает две комбинаторные задержки на входе, выбирая разные задержки для двух разных путей в опции Quartus. Регистры IOE в устройствах Cyclone используют один и тот же источник для сброса или предустановки своего состояния. Разработчик может запрограммировать предустановку или сброс для каждого отдельного ввода-вывода и запрограммировать регистры на включение или отключение питания после завершения настройки. Если запрограммировано включение низкого уровня, асинхронный сброс может контролировать регистры. Если запрограммировано включение высокого уровня, асинхронная предустановка может управлять регистрами. Эта функция предотвращает случайную активацию входа активного низкого уровня другого устройства при включении питания. Если

один регистр в IOE использует предустановленный или очищенный сигнал, то все регистры в IOE должны использовать этот же сигнал, если им требуется предустановка или сброс. Кроме того, для регистратора IOE доступен сигнал синхронного сброса.

2.1.1 Синхронизация цикла передачи данных и цикла вычисления

Вычислительная платформа NITTA использует микрокоманды, для управление вычислительными блока, тем самым организуя вычислительный процесс. При работе процессора время на обработку и приёма-передачу данных может занимать один вычислительных цикл, или больше, это больше всего отражается при работе I2C интерфейса, где одновременно не могут приниматься и приниматься данные от управляющего устройства, что вносит дополнительные временные задержки для вычислительного цикла. Вычислительный цикл определяется тремя основными временными процессы: получение данных от управляющего контроллера, выполнение вычислений с полученными данными, и передача полученных результатов управляющему контроллеру. С точки зрения характеристик вычислительной платформы NITTA, выделяются следующие случаи синхронизации временных процессов.

1) С потерей и без потери полученных данных от управляющего контроллера. Когда время отправки результатов превышает время по длительности вычислительного цикла, за счет уменьшения частоты выборки, часть данных теряется. Что не критично в случае HIL и PIL тестирования, когда мы получаем температуру с датчика. Для случая работы вычислительной платформы NITTA в связке с микроконтроллером, где выполняется работа по обработки данных с CAN шины, это может повлечь снижение производительности системы и обеспечить некорректную работу встраиваемой системы. Поэтому снижение частоты выборки позволяет повысить производительность моделирования и обработки данных, только в некоторых ситуациях.

2) С пересечением вычислительных процессов. Разнесение вычислительных процессов во времени вносит жесткие требования к вычислительной платформе, для задач реального времени, когда передачи данных пересекается с вычислительным процессом NITTA. Это вносит временные задержки процессора, не влияя на производительности системы. Такой подход синхронизации усложняет процесс синтеза для платформы NITTA, за счет учета ограничений на вычислительный процесс и внесение дополнительных требований для управляющего контроллера. Данная задача решается генерацией JSON файла, в котором содержится информация о характеристиках вычислительной платформы NITTA, и учета данных характеристик управляющим контроллером.

Задача синхронизации вычислительных процессов решается следующим образом:

1) Определение границ цикла приема-передачи данных. Каждая подсистема-ввода вывода I2C, SPI, PCI имеет в своей реализации флаг flag_stop, который информирует процессор NITTA о завершение процесса приема-передачи. Определение начала приема-передачи для каждой подсистемы индивидуально, так SPI определяет начало передачи по спаду CS (Ship Select), а I2C по сигналу, когда SDA переходит в логический ноли при положительном значении SCL.
2) Определение машинного слова при приема-передачи. Определяется на этапе генерации прикладного алгоритма в язык описания аппаратуры. Имеет фиксированное значение, которое не изменяется во время выполнения вычислительного процесса. От данного параметра зависит как работа процессора NITTA, для работы с валидными данными, так и подсистема ввода-вывода для корректного взаимодействия с управляющим контролером.
3) Обеспечение целостности данных при приема-передачи. Использование подсчета контрольной суммы.

2.1.2 Обеспечение надежности передачи данных

Определившись с машинным словом встает вопрос о надежности передачи данных по подсистемам ввода-вывода. Из – за не идеальности контактного соединения, возникает дребезг контактов, что влечет искажение данных при передаче. На ПЛИС во избежание дребезга контактов используется фильтр, представляющий собой простой счетчик, при срабатывании которого, входная величина подается на выход фильтра и поступает на основную схему ПЛИС. Но данный подход позволяет сделать временное решение, так как на частотах сравнимых с управляющим контроллером, фильтр будет иметь на выходе постоянный логический ноль. Поэтому встает вопрос о надежности передачи данных между устройствами всех кадров, при этом передавая их в правильном порядке. В такой случай, для проверки надежности принятых данных используется приемником посылается информация о том, что принимающее устройство приняло данные успешно. Данный механизм можно наблюдать на протоколе передачи данных I2C, где протокол обязывает принимающую сторону отправлять в обратную сторону флаг подтверждения. Получив положительное значение, отправитель продолжает посылать данные. При получении негативного флага, приема-передача между устройствами прекращается. Протокол SPI данный механизм не поддерживает. В контексте данной проблемы для SPI используются добавление дополнительной информации в сообщение [12]:

1) В первом байте машинного слова содержится информация о статусе передачи данных с прошлого цикла. При этом данный байт так же подвержен ошибкам и изменению своих данных, поэтому данный байт содержит либо значение 0xFF соответствующее успешному приему данных, либо 0x00, что данные были приняты некорректно. Так же при передаче возможно ситуация полной потери пакета, для этого могут использоваться таймеры.

2) Использование машинного слова с контрольной суммой. Под контрольной суммой понимается группа контрольных бит, связанных с самим машинным словом. Контрольная сумма помещается в конец машинного слова, таким образом ошибка определяется путем суммирования всего машинного слова и сравнивается с контрольной суммой, если результат равен нулю, то сообщение было принято успешно без потери данных. Данный подход используется в протоколах IP.

2.1.3 Маршрут проектирования подсистем ввода-вывода

На заре развития компьютерных систем, разработчики для своих продуктов использовали свои интерфейсы передачи данных не регламентированных стандартом. Это дополнительные сложности для обеспечения взаимодействия с другими устройствами компаний. Со временем интерфейсы передачи данных были стандартизированы, либо методом де-факто, либо юридически. Проектирование какой-либо системы ввода вывода начинается с изучения особенности и тонкостей работы протокола передачи данных, без понимания принципов работы протокола, обеспечить корректное взаимодействие с другими устройствами не получится. Каждый протокол регламентируем интерфейс передачи данных, к примеру, SPI имеет 4 сигнальные линии: выбор устройства CS, сигнал синхронизации SCLK, и лини передачи данных между устройствами MOSI и MISO. I2C в свою очередь имеет только два логические линии, линия передачи данных SDA, работающий в обе стороны, и сигнал синхронизации SCL.

Каждый протокол имеет свои логические состояния: отправить байт, принять байт, отправить сигнал подтверждения, принять сигнал подтверждения, и т.д. Данная логическая структура позволяет для эффективной реализации и поддержки в будущем, реализовать подсистему ввода-вывода на основе конечного автомата (FSM). Такой подход позволяет отслеживать переходы между состояниями и настроить подсистему достаточно гибко. Так

же не маловажным фактором является то, что САПР Quartus поддерживает FSM и оптимизируем его.

При проектировании одним из основных инструментов разработчика является функциональное симуляция, которая позволяет отслеживать поведение схемы без использования отладочных плат. Подход в проектировании без промежуточного тестирования разработанной схем на отладочных платах, может повлечь трудности при запуске и настройке схем на ПЛИС. Это связано с тем, что для симуляции и синтеза схем используются разные правила в компиляции, которые не учитывают особенности друг друга. После отладки исходной схемы в симуляции и обеспечения необходимого поведения, начинается процесс компиляции. Компиляция делится на две фазы: анализ и синтез. После синтеза мы получаем Netlist, представляющий собой связь логических вентилей и триггеров, которые мы можем посмотреть в RTL Viewer. Процесс анализа и синтеза имеет ряд настроек. Далее идет процесс размещения схемы на кристалле и генерация прошивки. После получения файла прошивки для соблюдения временных характеристик схемы указываются временные ограничения на пути прохождения логических сигналов. Далее идет процесс проверки временных характеристик и размещения, если нас не удовлетворяют полученные характеристики, то мы меняем настройки и повторяем процедуру итерационного до тех пор, пока не будут получены требуемые характеристики. Далее генерируем файл прошивки sof/pof и прошиваем ПЛИС.

2.2 Реализация подсистемы ввода-вывода в САПР NITTA

Разработка и поддержка функционала в САПР представляет собой сложный маршрут проектирования, включающий ряд взаимозависимых задач направленные на реализацию того или иного функционала. Для правильной реализации подсистемы ввода-вывода в САПР NITTA требуется создание документации, его поддержка и сопровождение.

Состав работ направленных на реализацию подсистем ввода-вывода:

1) Разработка драйвера подсистемы ввода-вывода на языке Verilog.
2) Разработка модуля верхнего уровня, включающего разработанный драйвер подсистемы ввода-вывода, обеспечивающего интеграцию с архитектурой NITTA.
3) Обеспечение поддержки драйвера в составе САПР с реализации модели драйвера на языке Haskell.
4) Генерация аппаратной части в соответствии с заданной микроархитектурой NITTA.
5) Генерация программного обеспечения в соответствии с заданным прикладным алгоритмом.
6) Генерация файла, содержащего информацию о протоколе передачи данных используемых подсистем ввода-вывода.
7) Реализация функциональной модели для функций receive и send прикладного алгоритма.
8) Генерация тестового окружения для функционального моделирования.

От разработанной подсистемы ввода-вывода на предварительном этапе проектирования и разработки требуется выявить и отписать задачи, решаемые той или иной подсистемой ввода-вывода. В качестве решаемых задач рассматривается HIL и PIL тестирование с системной динамикой. Для этих задач требуется разработать поддержку в САПР NITTA последовательных интерфейсов передачи данных SPI и I2C. Это связано со спецификой недетерминированной окружающей среды HIL и PIL тестирования, что большинство используемых сенсоров использую в основном последовательные интерфейсы передачи данных. Для системной динамики основополагающую значимость подсистемы ввода-вывода играет, пропуская способность канала, обеспечивающее скоростную передачу данных от ведомого устройства, для этих целей был выбран параллельный интерфейс PCI Express.

Следующим этапом идет разработка выбранных подсистем ввода-вывода и создание предварительных требований к вычислительной платформе NITTA. Первоначальная разработка заключается в создании IP драйвера подсистемы ввода-вывода для ПЛИС, обеспечивающего основной функционал работы, такой как:

1) Создание и описание протокольной части подсистемы ввода-вывода.
2) Унификация протокольной части, обеспечивающей независимость взаимодействующих устройств с вычислительной платформой NITTA.
3) Получение данных от основной логики ПЛИС и передача данных ведущему устройству кратным заданному машинному слову.
4) Прием данных от ведущего устройства и передача полученных данных основной логике работы ПЛИС.

Разработав драйвер подсистемы ввода-вывода требуется обеспечить взаимодействие с вычислительной платформой NITTA. Для этого создается отдельный модуль, который реализует прием данных и передачу данных для прикладного алгоритма, реализуемого на вычислительной платформе NIITA. За счет различия частот работы управляющего контроллера и вычислительного устройства, в верхнем модуле подсистемы ввода-вывода обеспечивается буферизация данных.

После создания и успешного тестирования аппаратной составляющей подсистемы ввода-вывода требуется обеспечить поддержку в САПР NITTA, для этого создаёте функциональная модель, описанная на языке Haskell и включающая в себя функционал по интеграции экземпляра модуля верхнего уровня с параметрами в зависимости от прикладного алгоритма, генерацию тестового окружения и описание логики работы. После описания функциональной модели САПР NITTA генерирует на основе имеющихся подсистем ввода-вывода и требований прикладного алгоритма, на основе заданной микроархитектуры генерирует аппаратную составляющую для ПЛИС, в которой находится подсистема ввода-вывода и вычислительной

устройство NITTA. Для поддержания корректного взаимодействия с внешними устройствами генерируется файл протокола, содержащий правила, по которым происходит обмен с подсистемой ввода-вывода в составе вычислительной платформы NITTA.

Глава 3. Разработка интерфейсов для NITTA

3.1 Спецификация протокола SPI

SPI является стандартом, обеспечивающим коммуникацию между двумя и более устройствами. SPI содержит простую архитектурную реализацию, содержащую один сдвиговый регистр для передачи и приема данных, он прост в реализации и обеспечивает синхронную передачу данных за счет отдельной линии SCLK, гарнируемой ведущим устройством. SPI является широко используемым интерфейсом передачи данных, он используется в микроконтроллерах, системах на кристалле, ASIC-ах и большинстве датчиков. Для работы протокола используется 4 сигнальные линии:

1) Передача данных от ведущего устройства ведомому.
2) Передача данных от ведомого устройства ведущему.
3) Сигнал синхронизации.
4) Сигнал выбора ведомого.

В сравнении с последовательным интерфейсом передачи данных I2C, SPI имеет ряд достоинств и недостатков. Основное отличие интерфейса SPI от интерфейса I2C, что он позволяет осуществлять передачу данных в полнодуплексном режиме. SPI позволяет подключать любое количество устройств к шине, за счет использования индивидуальной выделенной сигнальной линии (CS) выбора ведомого устройства, что в сравнении с I2C имеются ограничения по количеству подключаемых устройств, как по количеству свободных адресов, так и по емкости линии передачи данных.

На примере работы интерфейса JTAG, когда через линии TDI и TDO микросхемы объединяются в последовательную линию передачи, образую единую цепочку BSR, так и интерфейс поддерживает последовательное соединение устройств через линии MISO и MOSI, это удобно, когда данные с одного ведомого устройства передаются напрямую второму ведомому устройству, минуя ведущее устройство.

3.2 Проектирование архитектуры вычислительного блока SPI

Вычислительный блок SPI обеспечивает взаимодействие вычислительной платформы NITTA с недетерминированным рабочим окружением [5]. Для обеспечения корректной работы и обеспечения поддержки модели SPI в САПР требуется реализовать следующие архитектурные решения: разработка драйвера последовательного интерфейса передачи данных SPI, в соответствии с де-факто стандартом, состоящим из ведущего и ведомого драйвера, и обеспечение взаимодействия драйвера SPI с вычислительной платформой NITTA. Ведомый драйвер SPI поддерживает один режим работы SPI, которая соответствует CPOL = 0 и CPHA = 1. По переднему фронту SCLK происходит передача бита данных из сдвигового регистра на линии MISO, по заднему фронту SCLK происходит смещение сдвигового регистра на 1 бит данных и запись в последнюю ячейку сдвигового регистра бита данных с линии MOSI. В соответствии с этой логикой работы был реализован конечный автомат, содержащий 3 активных состояния: состояние инициализации и старта приема-передачи данных, состояние обеспечивающее передачу бита данных на линию MISO, состояние приема данных с линии MOSI. Ведомый драйвер SPI содержит счетчик переданных бит байтного слова данных. Драйвер представляет собой параметризированный модуль, поддерживающий задание разрядности машинного слова и приема-передачу любого количества машинных слов. Ведущий драйвер SPI содержит экземпляр модуля ведомого драйвера SPI управляемого дополнительным конечным автоматом. Конечный автомат ведущего драйвера содержит 4 основных состояния, реализующего передачу данных в режиме CPOL = 0 и CPHA = 1, представлен на рисунке 3.1 и 3.2.

1) Состояние инициализации и старта приема-передачи данных.
2) Состояние формирования переднего фронта SCLK.
3) Состояние формирования спадающего фронта SCLK.
4) Состояние завершения приема-передачи данных.

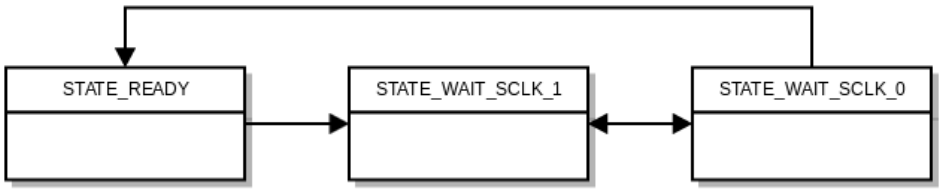

Рисунок 3.1 Структурная диаграммы драйвера SPI в режиме ведомого

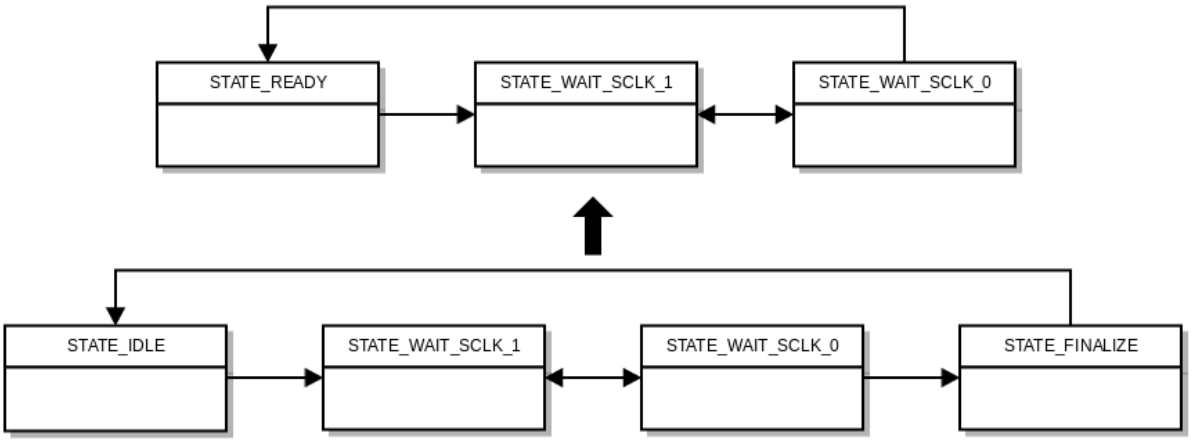

Рисунок 3.2 Структурная диаграммы драйвера SPI в режиме ведущего

Внутренняя обработка полученных данных по интерфейсу SPI осуществляется вычислительной платформой NITTA, которая имеет интерфейс для получения данных заданной разрядности машинного слова и выгрузка результата обратно в подсистему ввода-вывода, для этого необходимо учесть следующие требования к подсистеме ввода-вывода SPI со стороны NITTA:

1) Выгрузка результатов по управляющему сигналу signal_wr.
2) Получение данных от системы ввода-вывода в соответствии с работой прикладного алгоритма по сигналу signal_oe.

Вычислительная платформа NITTA работает в реальном времени и выполнение прикладного алгоритма осуществляется циклически. Одна итерация выполнения прикладного алгоритма считается как один вычислительный цикл, и как следствие обмен данными с подсистемой ввода-вывода осуществляется циклически, архитектура механизмов ввода-вывода представлена на рисунке 3.2.

Одна итерация цикла моделирования делится на фазы:

1) Получение данных от управляющего контроллера и передача их вычислительному циклу.
2) Выполнение прикладного алгоритма над данными, полученными по подсистеме ввода-вывода.
3) Отправка результатов вычислительного цикла управляющему контроллеру.

Для этого определим следующие требования к подсистеме ввода-вывода, для выполнения синхронизации цикла передачи данных и вычислительного цикла.

1. Определение начала и конца передачи данных. За начало передачи данных отвечает управляющий сигнал интерфейса SPI chip-select (CS). Переход из логической единицы в ноль формирует начало цикла передачи данных, переход из логического нуля в логическую единицу определяет конец цикла приема-передачи данных.
2. Формирование машинных слов заданной разрядности для вычислительного цикла.
3. Обеспечение целостности полученных данных и правильного порядка машинных слов от управляющего контроллера.

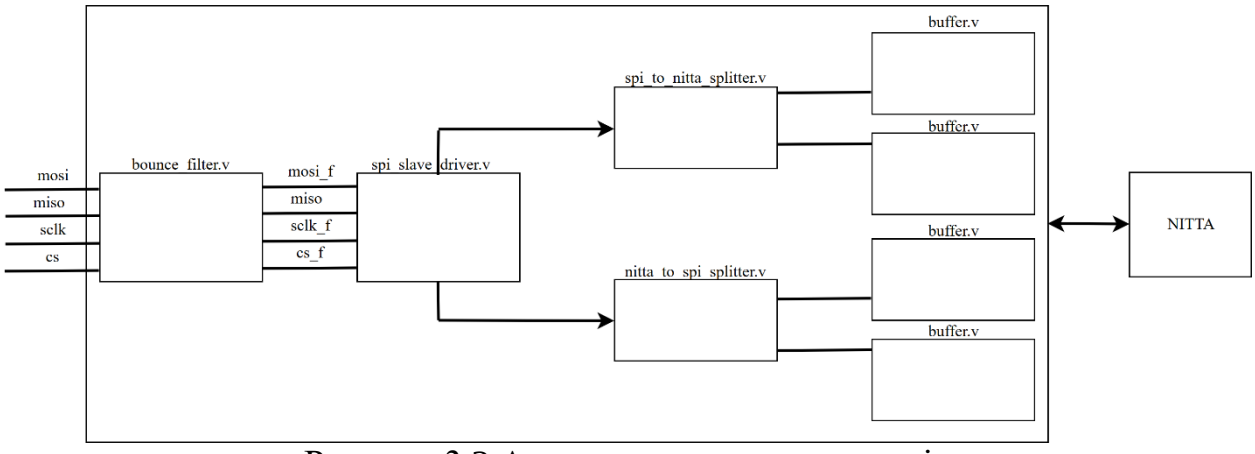

Рисунок 3.2 Архитектура модуля pu_spi

Для решения поставленных задач был разработан внешний параметризируемый модуль, включающий экземпляр драйвера SPI, модуль формирования машинного слова для вычислительного цикла из полученных

данных от драйвера SPI и буферизация данных на прием и передачу данных между циклом приема-передачи и вычислительным циклом.

Полученные данные от управляющего контроллера формируются в машинные слова заданной разрядности и сохраняются в буфер, по завершению цикла приема-передачи происходит переключение буферов через мультиплексор и данные из буфера по сигналу signal_oe поступают на вход вычислительного цикла NITTA. Получаем, что один буфер задействован на прием данных от управляющего устройства, второй для работы с вычислительным циклом NITTA. По завершению вычислительного цикла данные выгружаются в буфер подсистемы ввода-вывода SPI. При начале нового цикла приема-передачи, происходи переключение буферов на передачу и данные подаются на преобразователь машинного слова в байты, куда подаются на сдвиговый регистр SPI и подаются на линию MISO.

3.3 Разработка испытательного стенда для SPI

Один из этапов разработки подсистемы ввода-вывода для вычислительной платформы NITTA состоит в создании испытательного стенда для его отладки и тестирования. Испытательный стенд содержит средства измерения в виде логического анализатора и информационно-измерительной системы. Основная задача испытательного стенда состоит в генерации входных сигналов, подаваемых на реконфигурируемую вычислительную платформу NITTA и вызывающая обратную связь. Полученные результаты на испытательном стенде сравниваются с эталонными значениями для данного прикладного алгоритма, реализованного на вычислительной платформе NITTA. Но основе сравнения делается заключение о корректности выполнения тестируемого прикладного алгоритма.

Испытательный стенд состоит из:

1. Отладочная плата Raspberry Pi 3. Имеет несколько режимов работы в зависимости от вычислительной платформы NITTA. В режиме ведущего устройства передает данные для вычислительной

платформы NITTA, или получает в режиме ведомого, в соответствии с прикладным алгоритмом.

2. Отладочная плата на базе FPGA Altera Cyclone DE0-Nano. Реализует прикладной алгоритм на базе реконфигурируемой вычислительной платформе NIITA.
3. Логический анализатор CY7C68013A-56. Обеспечивает получение временных диаграмм передачи данных по интерфейсу SPI.

Испытательный стенд представлен на рисунке 3.3.

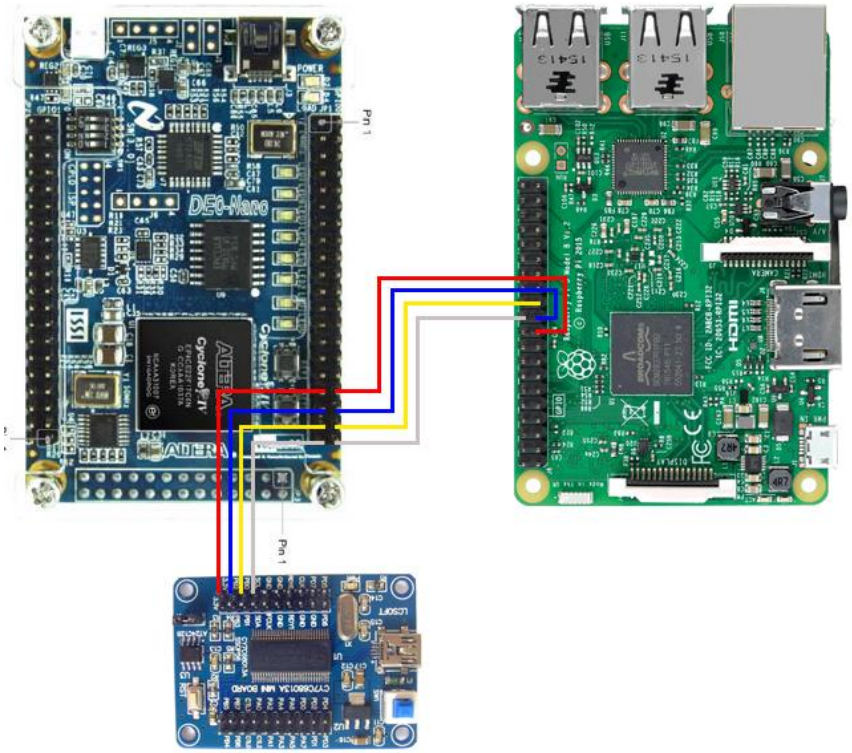

Рисунок 3.3 Испытательный стенд для тестирования SPI

Основные направления использования испытательного стенда:

1. Тестирование передачи данных между управляющий контроллером и вычислительной платформой данных заданной длины машинного слова и разрядности.
2. Исследование зависимости частоты SCLK SPI интерфейса управляющего контроллера от частоты работы вычислительной платформы.
3. Тестирование прикладного алгоритма, реализованного на вычислительной платформе NITTA.

4. Создание демонстрационного примера работы вычислительной платформе NITTA.
5. Разработка управляющего контроллера Raspberry Pi 3 велась на операционной системе ROS при интеграции с API BCM2835 ARM Peripherals, обеспечивает доступ к аппаратным интерфейсам SPI и I2C на Raspberry Pi 3.

3.4 Верификация и тестирование вычислительно блока SPI

Верификация цифровых блоков, разработанных для интегральных схем заключается в анализе результатов, полученных из САПР Quartus, применительно к ПЛИС фирмы Altera и исполнении тестов на функциональной симуляции с параллельным исполнение на физическом стенде. Для верификации необходимо определиться с ожидаемым поведением реализованного цифрового блока SPI с взаимодействием вычислительного цикла NITTA и внешнего управляющего контроллера. Далее требуется создать тестовое окружение для верификации вычислительного блока SPI. К разработанному вычислительному блоку SPI предъявляются следующие требования:

1. Выполнение этапа синтеза, имплементации и генерации прошивочного файла. Не нарушаются этап синтеза и трассировки наличием ограничений различных ПЛИС.
2. Прием-передача машинных слов заданной разрядности.
3. Длина принимаемого и машинного слова соответствует фиксированному значению, заданному как глобальный параметр вычислительной системы.
4. Количество принимаемых и отсылаемых данных имеет фиксированное значение за один цикл приема-передачи.
5. Последовательное выполнение вычислительного цикла и приема-передачи за одну итерацию цикла моделирования.

6. Работы вычислительного блока SPI в интеграции с вычислителем NITTA на частоте 200 MHz.

Первоначальная верификации заключается в создании функциональной модели для определения желаемого поведения на этапе проектирования цифрового блока SPI. Функциональная модель заключается в создании модуля на языке Verilog, который представляет собой тестируемый модуль и содержит экземпляр модуля вычислительного блока SPI. Тестируемый модель содержит ряд не синтезируемых инструкций Verilog игнорируемых компилятором, которые позволяют выводить отладочную информацию о текущем состоянии системы и определить временные вехи тестируемой системы, в которых полученное значение сравнивается эталонным для верификации промежуточных значений системы. Функциональная модель имеет свои преимущества и недостатки, за счет отличия правил компиляции проектируемых схем, результат функционального модулирования и физического не всегда совпадает с ожидаемым. Функционально моделирование позволяет верифицировать логику работы проектируемой схемы, но не позволяет смоделировать задержки распространения электрических сигналов по линиям передачи данных ПЛИС, что играет немаловажную роль при обеспечении требуемых характеристик работы интегральной схемы, результат представлен на рисунке 3.4.

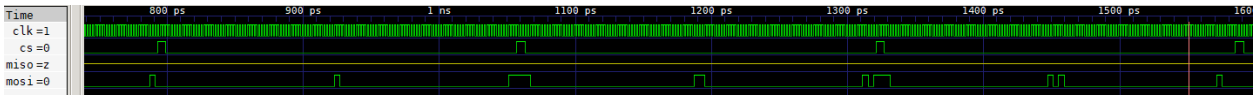

Рисунок 3.4 Результат моделирования генерации чисел Фибоначчи

Для верификации характеристик, предъявляемых к работе вычислительной платформе NITTA в режиме реального времени, использовалась информация о сгенерированной прошивке для ПЛИС средствами Quartus и проводилось тестирование на испытательном стенде. В качестве тестов для прикладного алгоритма использовалось генерация числовой последовательности Фибоначчи и суммирования входных данных.

Тестовая система состоит из управляющего контроллера Raspberry Pi 3 и DE0-Nano. Алгоритм работы Raspberry Pi 3 представляет собой реализацию на ROS с использование языка высокого уровня Python и обеспечивающее чтения и запись машинного слова устанавливаемого вычислительной платформой NITTA. Взаимодействие Raspberry Pi 3 осуществляется в зависимости от режима работы SPI вычислительной платформы NITTA.

Таблица 1. Характеристики вычислительной платформы NITTA для прикладного алгоритма генерации числовой последовательности Фибоначчи.

Показатель	Master	Slave	Суммирование
Занимаемая площадь	1,006/22,320	1,006/22,320	1,006/22,320
Частота работы	201,4 MHz	207,1 MHz	207,1 MHz

3.5 Спецификация I2C

Разработка встраиваемых систем содержит в себе общие структурные решения системы, которые подразумевают использование однокристального вычислительного устройства, на примере МК или ПЛИС, и внешнюю периферию для обработки данных и ее хранения. Для любой встраиваемой системы специфичным являются узлы, которые уникальны для такой встраиваемой системы. Из-за уникальности, на ранних стадиях развития компьютерных систем не удавалось обеспечить взаимодействие нужных устройства между собой, в связи с этим разрабатывались и внедрялись стандарты по передачи данных между устройствами. Один из таких интерфейсов стал Inter-Integrated Circuit (I2C), разработанный компанией Philips в 80-х годах прошлого столетия.

Первоначально спецификация I2C содержала размер адреса 7 бит и позволяла подключить 128 устройств к шине, но в 1992 году стандарт был пересмотрен и появилась возможность подключать 1024 устройства к одной шине. I2C использует две линии для передачи данных SCL и SDA, один является линией синхронизации, второй для передачи данных между

устройствами. Из-за наличия одной линии данных, в конкретный момент времени по шине можно передавать данные только в одном направлении, из-за этого он является полудуплексным интерфейсом передачи данных. Каждое устройство, которое имеет в своем составе интерфейс I2C имеет свой уникальный адрес, для драйвера I2C NITTA этот адрес равен 0x47. Такие устройства при передаче имеют два режиме работы и два состояния, быть ведущим или ведомым устройством, и осуществлять запись либо чтение. Так как ведущее устройство инициирует передачу данных, он же и является источник синхросигнала SCL.

По сравнению с интерфейсом SPI, I2C имеет только две линии передачи данных не зависимо от количества подключаемых к шине устройств, но это так же влечет за собой ошибки и проблемы при правильной работе устройств. Это связано по двум причинам, первое что необходимо для передачи данных по шине I2C – это подтягивающие резисторы к питанию, они должны содержаться только у одного устройства, при наличии подтягивающих резисторах на нескольких устройствам в параллельном сочетании суммарная сопротивление будет ниже и передача данных по шине будет невозможной. Вторая причина связана с ограничение по емкости на шине, каждое устройство вносит свою емкость и если значение в 400 пФ будет выше установленного, то передача данных по шине так же будет не возможной.

3.6 Проектирование архитектуры вычислительного блока I2C

Вычислительный блок I2C применятся для HIL и PIL тестирования, для этого требуется обеспечить взаимодействие вычислительной платформы NITTA с внешними датчиками, такими как BMP280. Перед разработкой стоит определится с требованиями, предъявляемыми к интерфейсу I2C. Проектируемый вычислительный блок входит в состав вычислительной платформы NITTA и недетерминированной окружающей среды, когда внешние подключаемые устройства по I2C в соответствии со своей спецификации

требуют свой алгоритм работы и последовательность действий для получения данных с внешнего устройства. Для этого требуется реализовать два отдельных модуля для ведущего и ведомого устройства I2C, с возможность чтения и записи неопределённого числа байт в машинном слове, это требование связано с тем, что ряд устройств требуют предварительной инициализации и настройки для корректной работы, и длина машинного слова не превышают двух байт данных, в свою очередь данные передаются размером в 4 байта. Последним требованием является внедрение и поддержание модели I2C в САПР NITTA, где требуемый размер передаваемых данных будет регламентироваться прикладным алгоритмом.

Ведомый драйвер I2C представляет собой логическую последовательность действий по приему и передачи данных. Первым этапом передачи данных между устройствами является передача от ведущего устройства 7-ми битный адрес ведомого устройства и бит на чтение или запись, где запись соответствует логическому нулю, а чтение логической единице. Следующим битом идет бит ACK, информирующий ведущее устройство о корректности принятых данных, в данном случае, если переданный адрес соответствует адресу ведомого устройства, то ACK имеет уровень логического нуля, в случае несоответствия, ACK имеет уровень логической единицы и приема-передача завершается.

Для случая успешного приема адреса устройства, продолжается приема-передача между устройствами размером в 1 байт и 9-ым битом ACK. Приема – передача завершается по инициативе ведущего устройства. В соответствии с данной логикой работы протокола I2C, реализация I2C представляет собой конечный автомат, состоящий из таких состояний, как:

1) Состояние инициализации данных и ожидание страта приема-передачи данных.
2) Состояние приема адреса, ведомого устройства.
3) Состояние приема сигнала подтверждения.
4) Состояние отправки сигнала подтверждения.

5) Состояние приема байта данных.
6) Состояние отправки байта данных.
7) Состояние завершения приема-передачи.

Ведомый драйвер I2C содержит два счетчика, необходимых для корректной приема-передачи данных. Первый счетчик отвечает за прием и передачу байта данных, второй отвечает за количество принятых и переданных байт данных. Драйвер обеспечивает задание числа переданных и принятых байт данных во время выполнения, но разрядность данных является фиксированным значением, так как I2C регламентируется передача данных фиксированного размера, кратному 1 байт. Передачи и прием данных для двух режимов работы I2C отличаются, это связано с необходимостью генерации собственного сигнала SCL, для этого реализуется отдельный драйвер I2C.

Ведущий драйвер I2C содержит аналогичный конечный автомат, отличающийся принципом получения и обработки полученных данных. Ведущий драйвер содержит в себе два ключевых функционала, необходимых для работы в режиме ведущего – это создание тактового генератора SCL и обработка данных с линии SDA. Работа тактового генератора представляет собой реализацию конечного автомата, содержащего следующий ряд состояний:

1) Инициализация старта приема-передачи.
2) Генерация логического нуля на линии SCL.
3) Генерация логической единице на линии SCL.

Управление тактовым генератором осуществляется со стороны вычислителя NITTA и окончание передачи данных со стороны обработки данных конечного автомата в ведущем драйвере I2C. Основной конечный автомат, выполняющий задачу по обработке данных с линии SDA содержит 6 состояний:

1) Состояние отправки адреса ведомому устройству.
2) Состояние принятия бита подтверждения со стороны ведомого устройства.

3) Состояние отправки бита подтверждения.
4) Состояние получения байта данных.
5) Состояние отправки байта данных.
6) Состояние окончания приема передачи данных.

Для получения данных из внутренней логики ПЛИС драйвером формируется i2c_write для передачи данных от сгенерированных данных ПЛИС к ведущему устройству, данный полученные от ведущего формируют сигнал i2c_read для передачи данных во внутреннюю логику ПЛИС. Внутренняя обработка осуществляется вычислительной платформой NITTA, для выполнения реализованного прикладного алгоритма. Обработка и взаимодействие вычислительной платформы NIITA осуществляется одинаково в сравнении с подсистемой SPI. Это позволило унифицировать процесс взаимодействия разработанных подсистем ввода-вывода с вычислительной платформой NITTA, не усложняя и не внося уникальных для данных подсистемы ввода-вывода механизмов обработки данных.

3.7 Разработка испытательного стенда для I2C

Для тестирования и верификации HIL и PIL тестирования требуется создать тестовое окружение, включающий все необходимые модули для создания испытательного стенда. Испытательный стенд состоит из:

1) Отладочная плата Raspberry Pi 3.
2) Отладочная плата на базе FPGA Altera Cyclone DE0-Nano.
3) Логический анализатор CY7C68013A-56.
4) Датчик температуры BMP280.
5) Нагревательный элемент.
6) Модуль реле.
7) Источник питания на 12В и 220В.

Основная задача испытательного стенда для I2C состоит в следующем:

1) Проверка гипотезы о применимости вычислительной платформы NITTA для HIL и PIL тестирования.
2) Проверка корректности приема-передачи данных между устройствами по интерфейсу I2C.
3) Проверки корректности моделирования системы с окружающей недетерминированной средой.

Испытательный стенд для HIL и PIL тестирования представлен на рисунке 3.5.

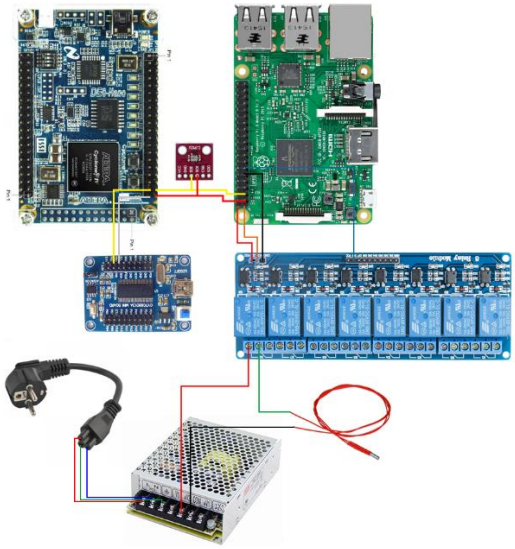

Рисунок 3.5 Испытательный стенд для тестирования I2C

Для упрощения тестирования в качестве управляющего контроллера выступает отладочная плата Raspberry Pi 3, которая по интерфейсу I2C подключается к датчику температуры BMP280. Это было сделано для того, чтобы основной прикладной алгоритм выполняемый на вычислительной платформе NITTA реализовывал исключительно функционал PID – регулятора, и не выполнял функции настройки датчика BPM280. Так же Raspberry Pi обеспечивает режим отладки и вывода информации о состоянии текущей системы для возможности визуальной отладки и понимания корректности работы прикладного алгоритма. К данной шине I2C так же подключается вычислительная платформа NITTA, которая выполняет роль моделирования PID – регулятора. Изначально Raspberry Pi получает от датчика BMP280 текущую температуру чашки, к которой он зафиксирован. Полученная температура передается вычислительной платформе NITTA, где результатом ее

работы является управление реле, к которому подключен нагреватель. Задача прикладного алгоритма PID регулятора состоит в стабилизации температуры в заданном значении. Из-за не идеальности окружающей среды, присутствия тепловых транспортных задержек с слабым остыванием датчика, стабилизации температуры несет сильный колебательный процесс как по времени, так и по диапазону изменения температуры.

3.8 Верификация и тестирование вычислительно блока I2C

Верификация вычислительного блока I2C состоит их 3-х этапов:

- Первый этап заключается в соблюдении требований на разработку вычислительного блока и проведении функционального моделирования в среде GTKWave.

- Второй этап заключается в верификации работы вычислительного блока в составе вычислительной платформы NITTA в ведомом и ведущем режиме и проведении функционального моделирования в среде GTKWave.

- Третий этап заключается в верификации работы вычислительного блока в составе прикладного алгоритма для задачи HIL и PIL тестирования с натурным тестированием.

Первые два этапа включают создание тестового окружения с использованием функциональной модели, для верификации и валидации поведенческой модели вычислительного блока I2C. Для этого на языке Verilog был разработаны тестовые модули, включающие проверку работы вычислительного блока I2C в двух режимах работы — ведущем и ведомом.

Ведущий режим вычислительного блока I2C должен соответствовать следующим требованиям:

1) Передача адреса ведомого устройства. При успешной инициализации ведомого устройства начинается цикл приема-передачи, при неудачном обнаружении цикл приема-передачи завершается с выставлением флага окончания цикла приема-передачи.

2) Передача данных заданной ширины машинного слова в одной итерации цикла приема-передачи. На каждый переданный байт получаем сигнал подтверждения, в случае неуспешной передачи данных завершаем цикл приема-передачи данных.
3) Прием данных заданной ширины машинного слова. На каждый полученный байт данных формируем сигнал подтверждения, в случае неудачного приема данных завершаем цикл приема-передачи.

На рисунке 3.6 и 3.7 показаны временные диаграммы для вычислительного блока I2C в режиме ведущего устройства, в который осуществляется запись и чтение машинного слова длиной в 4 байта.

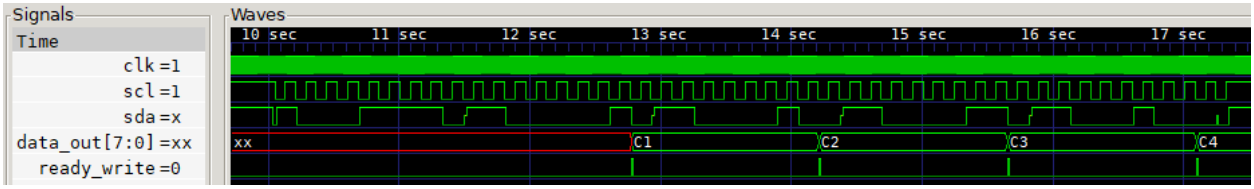

Рисунок 3.6 Прием 4 байта через I2C в режиме ведущего

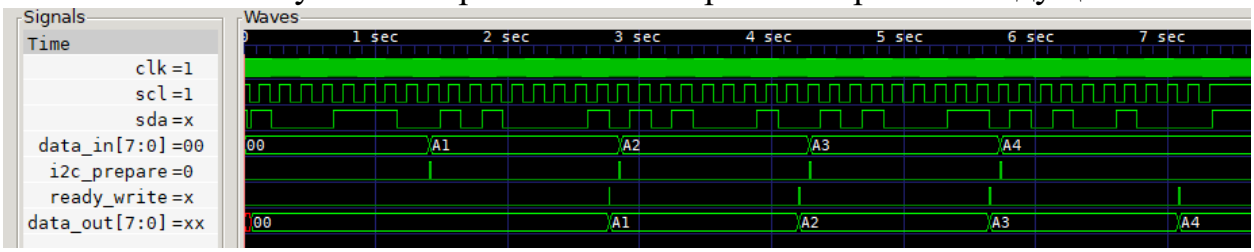

Рисунок 3.7 Передача 4 байта через I2C в режиме ведущего

Ведомый режим вычислительного блока I2C должен соответствовать следующим требованиям:

1) Прием адреса ведомого устройства. При успешном приеме адреса ведомого устройства формируем положительный сигнал подтверждения и начинаем приема-передачу данных, в случае несовпадения адресов, формируется отрицательный сигнал подтверждения и завершается приема-передача данных.
2) Передача данных фиксированной ширины машинного слова в одной итерации цикла приема-передачи. На каждый принятый байт получаем сигнал подтверждения, в случае неуспешного приема данных завершаем цикл приема-передачи данных.

3) Прием данных заданной ширины машинного слова. На каждый переданный байт данных получаем сигнал подтверждения, в случае неудачной передачи данных завершаем цикл приема-передачи.

На рисунке 3.8 и 3.9 показаны временные диаграммы для вычислительного блока I2C в режиме ведомого устройства, в который осуществляется запись и чтение машинного слова длиной в 8 байт.

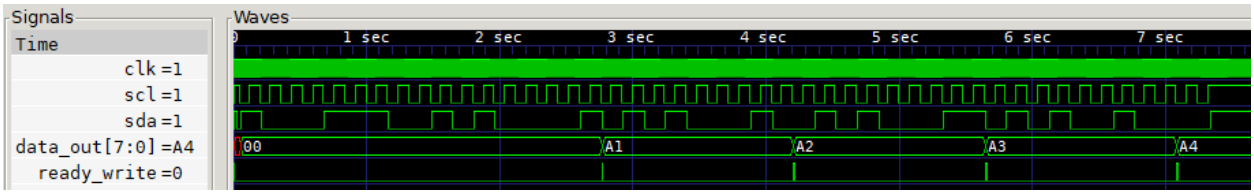

Рисунок 3.8 Прием 4 байта через I2C в режиме ведомого

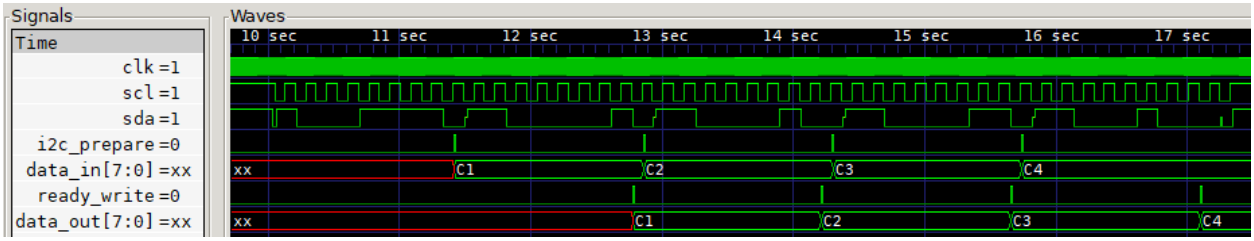

Рисунок 3.9 Передача 8 байт через I2C в режиме ведомого

Вычислительный блок I2C для прикладного алгоритма суммирования данных должен удовлетворять следующим требованиям:

1) Прием и формирование данных полученных из драйвера I2C в соответствии с заданной шириной машинного слова.

2) Получение данных из вычислительного цикла NITTA и сохранением в буфер обмена с последующим этапом формирования байт данных и передачей по интерфейсу I2C.

На рисунке 3.10 показана временная диаграмма для вычислительного блока I2C в составе работы прикладного алгоритма суммирования данных. В первом итерации цикла приема-передачи данных мы отправляем два значения – 4 и 3, на третьем цикле приема-передачи ведущее устройство получает результат 7.

Рисунок 3.10 Выполнение прикладного алгоритма суммирования

Для верификации интерфейсного блока I2C в составе прикладного алгоритма HIL и PID тестирования, тестовое окружение содержало модуль вычислительно блока I2C в режиме ведомого с интеграцией вычислительной платформы реального времени NITTA, реализующего прикладной алгоритм PID регулятора. На этапе верификации двух этапов использовались функциональное моделирование, что позволяло проанализировать функциональные диаграммы. По результатам функционального тестирования была выявлена корректность передаваемых данных и успешное выполнение тестирования, результаты тестирования сравнивались с эталонными значениями. После функционального моделирования проводилось натурное тестирование на испытательном стенде, суть которого заключается в стабилизации температуры на уровне 22 градусов под Цельсия, для этих целей использовался датчик BMP280 и нагревательный элемент на 40Вт. Управляющий контроллер Raspberry выполнял роль инициализации датчика температуры, управление внешним оборудованием, передачей температуры от датчика вычислительной платформе NITTA по интерфейсу I2C и приемом коэффициента ПИД обратно. Если полученный коэффициент имел отрицательное значение, то нагрев датчика отключался, имея положительное значение, происходи нагрев датчик. Тем самым получая колебательный эффект, который не заканчивается никогда, так как ПИД регулятор работает на основе ошибки. На рисунке 3.10 представлена запись в хронлогическом порядке, представляющее собой изменение температуры датчика под воздействие ПИД регулятора.

```
Data:    FF FE D0 FF
Temperature in Celsius: 22.004465
Data:    FF FE D0 FF
Temperature in Celsius: 22.004465
Data:    FF FE D0 FF
Temperature in Celsius: 22.004465
Data:          D2
Temperature in Celsius: 21.997288
Data:          D2
Temperature in Celsius: 21.990111
Data:    FF FE D0 FF
Temperature in Celsius: 22.004465
Data:          D2
Temperature in Celsius: 21.990111
```

Рисунок 3.10 Запись о событии изменения температуры

Поле Data характеризует полученные данные от NITTA, как видно по записи, имеется колебания температуры около 22 градусов под Цельсия. В ходе каждой итерации, на вычислительную платформу NITTA по интерфейсу I2C осуществлялась два раза запись температуры и один раз чтение ПИД коэффициента, пример одной итерации представлен на рисунке 3.11.

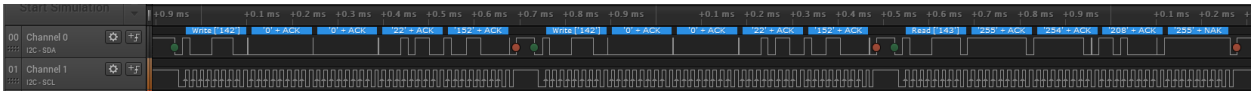

Рисунок 3.11 Приема-передача по I2C между Raspberri и NITTA

Рисунок 3.11 иллюстрирует механизмы ввода-вывода NITTA, на первом этапе осуществляется передача данных, второй этап выполняются вычисления ПИД регулятором, которые занимают порядка 50 тактов, третий этап выполняет выгрузку результаты из вычислительной платформы NITTA. За счет прямого контакта нагревательного элемента и датчика, удалось стабилизировать температуру с наименьшими транспортными задержками между средами, что позволило верифицировать и протестировать прикладной алгоритм ПИД регулятора, получающего и передающего данные по I2C с наименьшими временными затратами.

Заключение

Данная работа посвящена тематике исследования и разработки механизмов ввода-вывода для вычислительной платформы реального времени NITTA, обеспечивающий взаимодействие прикладного алгоритма с недетерминированной операционной средой. В ходе проведенного исследования были выполнены следующие задачи:

1) Были исследованы существующие интерфейсы передачи данных, и определены критерии выбора интерфейса передачи данных для реализации взаимодействия вычислительной платформы NITTA с внешними устройствами, с возможностью выполнения задач для области системной динамики и HIL, и PIL тестирования.
2) Были реализованы драйвера интерфейсов передачи данных SPI и I2C на основе выбранных критериев с целью практического подтверждения применения вычислительной платформы NITTA для HIL и PIL тестирования.
3) Были реализованы механизмы ввода-вывода обеспечивающие корректную интеграцию разработанных драйверов с вычислительной платформой NITTA.
4) Были исследованы возможности разработанных вычислительных блоков по требованиям применения к HIL и PIL тестированию, рассмотрены их ограничения. В частности, предложенное решение требует изменения прикладного алгоритма, выполняемого на NITTA, для возможности передачи данных различной размерности машинного слова, в зависимости от типа и назначения передаваемых и принимаемых данных. В соответствии с данными ограничениями были даны рекомендации по использованию и путь решения поставленных ограничений.

5) Был реализован испытательный стенд для HIL и PIL тестирования, обеспечивающего выполнение прикладного алгоритма ПИД регулятора.
6) Был реализован испытательный стенд для системной динамики.
7) Была внедрена операционная система ROS на управляющем контроллере Raspberry для интеграции и взаимодействия с вычислительной системой реального времени NITTA.

Определения, обозначения и сокращения

В настоящей пояснительной записке применяют следующие термины с соответствующими определениями:

ПЛИС - Программируемая логическая интегральная схема.

СнК - система на кристалле.

PCI - Peripheral Component Interconnect Express.

HIL - Hardware-in-the-loop simulation, Программно - аппаратное моделирование.

PIL - Purpose-in-Life Test, Процессорное тестирование.

SPI - Serial Peripheral Interface, последовательный периферийный интерфейс.

I2C - Inter-Integrated Circuit, последовательная шина данных для связи интегральных схем, использующая две двунаправленные линии связи.

САПР - Система автоматизированного проектирования.

RTL - Register-Transfer Level, Уровень регистровых передач.

DE0-Nano - Отладочная плата на базе FPGA Altera Cyclone.

Altera - крупнейшие разработчики ASIC, программируемых логических интегральных схем.

IP - Intellectual Property, готовые блоки для проектирования интегральных схем.

ModelSim – среда моделирования интегральных схем.

Timing Analyzer - временной анализатор прохождения сигналов в узлах интегральной схемы.

HDL - Hardware Description Language, язык описания аппаратуры.

Verilog - это язык описания аппаратуры, используемый для описания и моделирования электронных систем.

VHDL - это язык описания аппаратуры, используемый для описания и моделирования электронных систем.

HLS - High-level synthesis, Синтез высокого уровня.

NISC - One instruction set computer.

TTA - Transport triggered architecture.

PLL - Phase-Locked Loop, Фазовая автоподстройка частоты.

ПЗУ - Постоянное запоминающее устройство.

CPHA – Clock Phase, фаза синхронизации.

CPOL – Clock Polarity, полярность синхронизации.

CS – Chip Select, выбор устрйоства.

MOSI – Master Output Slave Input, линия передачи данных от ведущего устрйотсва.

MISO – Master Input Slave Output, линия передачи данных ведомому устройтству.

SCLK, CLK, SCK – сигнал синхронизации данных.

SCL – Serial CLock, последовательная линия синхронизации.

SDA – Serial Data, последовательная линия передачи данных.

Список литературы

1. А.В. Пенской, А.Е. Платунов, А.О. Ключев, Я.Г. Горбачев, Р.И. Яналов, Система высокоуровневого синтеза на основе гибридной реконфигурируемой микроархитектуры, Университет ИТМО, Санкт-Петербург 2019
2. Reiner Hartenstein, SE Curricula are Unqualified to Copewith theDataAvalanche, 2017
3. Артём Коновальчик, Применение специализированных вычислителей на основе ПЛИС для решения задач информационной безопасности, - 2013
4. Пенской А.В., Платунов А.Е., Яналов Р.И, Система высокоуровневого синтеза на основе гибридной NISC/TTA микроархитектуры, 2018
5. И.И. Левин, А.И. Дордопуло, Ю.И. Доронченко, М.К. Раскладкин, Реконфигурируемая вычислительная система на основе ПЛИС Virtex UltraScale с жидкостным охлаждением, 2016
6. Емельянов Д.В, Разработка вычислительного блока интерфейса SPI для вычислительной платформы реального времени, 2018
7. Н. Дмитренко, И.А. Каляев, И.И. Левин, Е.А. Семерников, Реконфигурируемые вычислительные системы для решения вычислительно трудоемких задач.
8. Отладочный стенд реального времени для систем управления SPEEDGOAT, - https://exponenta.ru/
9. А.А. Антонов, Проектирование микроархитектуры вычислителей на базе проблемно-ориентированных языков, 2017
10. Д.Е. Боркивец, А.И. Егоров, А.Г. Кузякин, Функциональное моделирование цифровых схем с помощью программ тестирования на языке C++, 2017
11. Cyclone FPGA Family Data Sheet, https://www.intel.com/
12. Харрис Д.М., Харрис С.Л. Цифровая схемотехника и архитектура компьютера. Издательство Morgan Kaufman, 2013 – 1632 с.

13. FPGA-based implementation of signal processing systems / R. Woods[et al.]. – – John Wiley & Sons, 2008.
14. Ha S. Decidable dataflow models for signal processing: Synchronous dataflow and its extensions / S. Ha, H. Oh // Handbook of Signal Processing Systems. — Springer, 2013. — P. 1083—1109.
15. М.В. Лапшин, Р.Р. Русаков, Выбор оптимальной платформы для реализации спецвычислителя.
16. Петров Е.Ю., АндреевА.С.. «Специализированные процессоры потоковой обработки». Вопросы радиоэлектроники, серия ЭВТ, вып. 2, с. 87-95.
17. Каляев А.В., Левин И.И. Модульно-наращиваемые многопроцессорные системы со структурно-процедурной организацией вычислений. – М. : Янус-К, 2003. – 380 с.
18. Коновальчик А. Высокопроизводительные вычислительные системы с реконфигурируемой архитектурой, построенной на ПЛИС // Современные технологии автоматизации. – 2013. – No 3
19. Hamzah R. A., Ibrahim H. Literature Survey on Stereo VisionDisparity Map Algorithms. Penang – Melaka: Hindawi Publishing,2015.
20. Н.В. Нестеренко, В.В. Ересько, Ю.С. Яковлев, Применение плис для построения вычислительных систем и их компонентов, - 2016
21. Каляев И.А. Реконфигурируемые вычислительные системы на основе полей ПЛИС [Электрон-ный ресурс]/ И.А. Каляев, И.И. Левин.– Новосибирск: НИИмногопроцессорных вычислительных систем Южного федерального университета, 12.10.2011.
22. ПЛИС-платформа в критических приложениях: гарантоспособные масштабируемые решения для информационных и управляющих систем АЭС / Е.С.Бахмач, А.А.Сиора, В.В.Скляр[и др.]//Радіоелектронні і комп’ютерні системи.–2008. –No6. –С.12–19

Приложение А. Фрагменты исходных кодов программного обеспечения

А.1 Описание RTL-модели вычислительного блока интерфейса SPI

```
module pu_slave_spi #
( parameter DATA_WIDTH     = 32
, parameter ATTR_WIDTH     = 4
, parameter SPI_DATA_WIDTH = 8
, parameter BUF_SIZE       = 6
, parameter BOUNCE_FILTER  = 4
, parameter INVALID        = 0
, parameter SIZE_WORDS     = 2
)
( input                      clk
, input                      rst
, input                      signal_cycle

, 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

, input                      cs
, input                      sclk
, input                      mosi
, output                     miso
);

reg send_buffer_sel;

wire send_buffer_wr[1:0];
wire send_buffer_oe[1:0];
wire [DATA_WIDTH-1:0] send_buffer_data_in[1:0];
wire [DATA_WIDTH-1:0] send_buffer_data_out[1:0];

generate
genvar i;
for ( i = 0; i < 2; i = i + 1 ) begin : send_buffer_i
buffer #
( .BUF_SIZE( BUF_SIZE )
, .DATA_WIDTH( DATA_WIDTH )
) send_buffer
( .clk( clk )
, .rst( rst || flag_stop )

, .wr( send_buffer_wr[i] )
, .data_in( send_buffer_data_in[i] )

, .oe( send_buffer_oe[i] )
, .data_out( send_buffer_data_out[i] )
```

```
                                    );
                                end
                            endgenerate

        assign send_buffer_wr[0] =  send_buffer_sel ? signal_wr : 1'h0;
        assign send_buffer_wr[1] = !send_buffer_sel ? signal_wr : 1'h0;
        assign send_buffer_data_in[0] =  send_buffer_sel ? data_in : 0;
        assign send_buffer_data_in[1] = !send_buffer_sel ? data_in : 0;

      assign send_buffer_oe[0] = !send_buffer_sel ? splitter_ready : 1'h0;
      assign send_buffer_oe[1] =  send_buffer_sel ? splitter_ready : 1'h0;
wire [DATA_WIDTH-1:0] nitta_to_splitter =  send_buffer_data_out[send_buffer_sel];

                            wire splitter_ready;
                wire [SPI_DATA_WIDTH-1:0] splitter_to_spi;
                            wire spi_prepare;
                        nitta_to_spi_splitter #
                          ( .DATA_WIDTH( DATA_WIDTH )
                          , .ATTR_WIDTH( ATTR_WIDTH )
                        , .SPI_DATA_WIDTH( SPI_DATA_WIDTH )
                              ) nitta_to_spi_splitter
                                 ( .clk( clk )
                             , .rst( rst || flag_stop )

                             , .spi_ready( spi_prepare )
                            , .to_spi( splitter_to_spi )

                          , .splitter_ready( splitter_ready )
                           , .from_nitta( nitta_to_splitter )
                                        );

                            wire spi_ready;
                wire [SPI_DATA_WIDTH-1:0] splitter_from_spi;
                            wire splitter_ready_sn;
                        wire [DATA_WIDTH-1:0] to_nitta;
                        spi_to_nitta_splitter #
                          ( .DATA_WIDTH( DATA_WIDTH )
                          , .ATTR_WIDTH( ATTR_WIDTH )
                        , .SPI_DATA_WIDTH( SPI_DATA_WIDTH )
                              ) spi_to_nitta_splitter
                                 ( .clk( clk )
                             , .rst( rst || flag_stop )
                             , .spi_ready( spi_ready )
                           , .from_spi( splitter_from_spi )
                       , .splitter_ready( splitter_ready_sn )
                             , .to_nitta( to_nitta )
                                        );

                          wire receive_buffer_wr[1:0];
                          wire receive_buffer_oe[1:0];
                          wire receive_buffer_fs[1:0];
                wire [DATA_WIDTH-1:0] receive_buffer_data_in[1:0];
                wire [DATA_WIDTH-1:0] receive_buffer_data_out[1:0];

                                generate
                                  genvar j;
                for ( j = 0; j < 2; j = j + 1 ) begin : receive_buffer_j
                                    buffer #
                                ( .DATA_WIDTH( DATA_WIDTH )
                                  , .BUF_SIZE( BUF_SIZE )
                                      ) receive_buffer
                                     ( .clk( clk )
                             , .rst( rst || receive_buffer_fs[j] )
```

```
                                , .wr( receive_buffer_wr[j] )
                              , .data_in( receive_buffer_data_in[j] )

                                , .oe_other( receive_buffer_oe[j] )
                            , .data_out_other( receive_buffer_data_out[j] )
                                                );
                                              end
                                           endgenerate

    assign receive_buffer_wr[0] =  send_buffer_sel ? splitter_ready_sn : 1'h0;
    assign receive_buffer_wr[1] = !send_buffer_sel ? splitter_ready_sn : 1'h0;
        assign receive_buffer_fs[0] = !send_buffer_sel ? flag_stop : 1'h0;
        assign receive_buffer_fs[1] =  send_buffer_sel ? flag_stop : 1'h0;
       assign receive_buffer_data_in[0] =  send_buffer_sel ? to_nitta : 0;
       assign receive_buffer_data_in[1] = !send_buffer_sel ? to_nitta : 0;
        assign receive_buffer_oe[0] = !send_buffer_sel ? signal_oe : 1'h0;
        assign receive_buffer_oe[1] =  send_buffer_sel ? signal_oe : 1'h0;

                                wire f_mosi, f_cs, f_sclk;

                                  pu_slave_spi_driver #
                                 ( .DATA_WIDTH( SPI_DATA_WIDTH )
                                           ) spi_driver
                                        ( .clk( clk )
                                        , .rst( rst )
                                 , .data_in( splitter_to_spi )
                                , .data_out( splitter_from_spi )
                                     , .ready( spi_ready )
                                   , .prepare( spi_prepare )
                                       , .mosi( f_mosi )
                                        , .miso( miso )
                                       , .sclk( f_sclk )
                                         , .cs( f_cs )
                                              );

bounce_filter #( .DIV(BOUNCE_FILTER) ) f_mosi_filter ( rst, clk, mosi, f_mosi );
bounce_filter #( .DIV(BOUNCE_FILTER) ) f_cs_filter   ( rst, clk, cs,   f_cs   );
bounce_filter #( .DIV(BOUNCE_FILTER) ) f_sclk_filter ( rst, clk, sclk, f_sclk );

                                   reg prev_f_cs;
                          always @( posedge clk ) prev_f_cs <= f_cs;

                                always @( posedge clk ) begin
                                 if ( rst ) send_buffer_sel <= 0;
              else if ( signal_cycle && f_cs ) send_buffer_sel <= !send_buffer_sel;
                                              end

                                always @( posedge clk ) begin
                                   if ( rst ) flag_stop <= 0;
                          else if ( !prev_f_cs && f_cs ) flag_stop <= 1;
                                      else flag_stop <= 0;
                                              end

                                   reg [1:0] count_word;
                                always @( posedge clk ) begin
                         if ( rst | flag_stop )        count_word <= 0;
                    else if ( splitter_ready_sn ) count_word <= count_word + 1;
                                              end

                                  reg capture_word;
                                always @(posedge clk) begin
                           if ( rst )            capture_word <= 0;
```

```
      else if ( flag_stop ) capture_word <= count_word != SIZE_WORDS;
                                    end

                        assign attr_out[3:1] = 3'b000;
                    assign attr_out[INVALID] = capture_word;

 assign data_out   = receive_buffer_oe[1] ? receive_buffer_data_out[1] :
                     receive_buffer_oe[0] ? receive_buffer_data_out[0] :
                           {DATA_WIDTH{1'b0}};

                                endmodule
```

A.2 Описание RTL-модели вычислительного блока интерфейса I2C

```
module pu_slave_i2c #
        ( parameter DATA_WIDTH     = 32
        , parameter ATTR_WIDTH     = 4
        , parameter I2C_DATA_WIDTH = 8
        , parameter BUF_SIZE       = 6
        , parameter BOUNCE_FILTER  = 4
        , parameter INVALID        = 0
        , parameter SIZE_WORDS     = 2
        )
    ( input                       clk
    , input                       rst
    , input                       signal_cycle

    , 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

    , input scl
    , inout sda

    , output D0
    , output D1
    , output D2
    , output D3
    , output D4
    , output D5
    , output D6
    , output D7
    );

assign D0 = sda;
assign D1 = scl;

reg send_buffer_sel;

wire send_buffer_wr[1:0];
wire send_buffer_oe[1:0];
wire [DATA_WIDTH-1:0] send_buffer_data_in[1:0];
wire [DATA_WIDTH-1:0] send_buffer_data_out[1:0];

generate
```

```
    genvar i;
    for ( i = 0; i < 2; i = i + 1 ) begin : send_buffer_i
        buffer #
                ( .BUF_SIZE( BUF_SIZE )
                , .DATA_WIDTH( DATA_WIDTH )
                ) send_buffer
            ( .clk( clk )
            , .rst( rst || flag_stop )

            , .wr( send_buffer_wr[i] )
            , .data_in( send_buffer_data_in[i] )

            , .oe( send_buffer_oe[i] )
            , .data_out( send_buffer_data_out[i] )
            );
    end
endgenerate

assign send_buffer_wr[0] =  send_buffer_sel ? signal_wr : 1'h0;
assign send_buffer_wr[1] = !send_buffer_sel ? signal_wr : 1'h0;
assign send_buffer_data_in[0] = send_buffer_sel ? data_in : 0;
assign send_buffer_data_in[1] = !send_buffer_sel ? data_in : 0;

assign send_buffer_oe[0] = send_buffer_sel ? splitter_ready : 1'h0;
assign send_buffer_oe[1] = !send_buffer_sel ? splitter_ready : 1'h0;
wire            [DATA_WIDTH-1:0]            nitta_to_splitter           =
send_buffer_data_out[send_buffer_sel];

wire splitter_ready;
wire [I2C_DATA_WIDTH-1:0] splitter_to_i2c;
nitta_to_i2c_splitter #
        ( .DATA_WIDTH( DATA_WIDTH )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .I2C_DATA_WIDTH( I2C_DATA_WIDTH )
        ) nitta_to_i2c_splitter
    ( .clk( clk )
    , .rst( rst || flag_stop )

    , .i2c_ready( i2c_prepare )
    , .to_i2c( splitter_to_i2c )

    , .splitter_ready( splitter_ready )
    , .from_nitta( nitta_to_splitter )
    );

wire [I2C_DATA_WIDTH-1:0] splitter_from_i2c;
wire splitter_ready_sn;
wire [DATA_WIDTH-1:0] to_nitta;
i2c_to_nitta_splitter #
        ( .DATA_WIDTH( DATA_WIDTH )
        , .ATTR_WIDTH( ATTR_WIDTH )
        , .I2C_DATA_WIDTH( I2C_DATA_WIDTH )
        ) i2c_to_nitta_splitter
    ( .clk( clk )
    , .rst( rst || flag_stop )
    , .i2c_ready( i2c_ready_write )
    , .from_i2c( splitter_from_i2c )
    , .splitter_ready( splitter_ready_sn )
    , .to_nitta( to_nitta )
    );

wire receive_buffer_wr[1:0];
wire receive_buffer_oe[1:0];
```

```
wire receive_buffer_fs[1:0];
wire [DATA_WIDTH-1:0] receive_buffer_data_in[1:0];
wire [DATA_WIDTH-1:0] receive_buffer_data_out[1:0];

generate
    genvar j;
    for ( j = 0; j < 2; j = j + 1 ) begin : receive_buffer_j
        buffer #
                ( .DATA_WIDTH( DATA_WIDTH )
                , .BUF_SIZE( BUF_SIZE )
                ) receive_buffer
            ( .clk( clk )
            , .rst( rst || receive_buffer_fs[j] )

            , .wr( receive_buffer_wr[j] )
            , .data_in( receive_buffer_data_in[j] )

            , .oe_other( receive_buffer_oe[j] )
            , .data_out_other( receive_buffer_data_out[j] )
            );
    end
endgenerate

assign receive_buffer_wr[0] =  send_buffer_sel ? splitter_ready_sn : 1'h0;
assign receive_buffer_wr[1] = !send_buffer_sel ? splitter_ready_sn : 1'h0;
assign receive_buffer_fs[0] = !send_buffer_sel ? flag_stop : 1'h0;
assign receive_buffer_fs[1] =  send_buffer_sel ? flag_stop : 1'h0;
assign receive_buffer_data_in[0] =  send_buffer_sel ? to_nitta : 0;
assign receive_buffer_data_in[1] = !send_buffer_sel ? to_nitta : 0;
assign receive_buffer_oe[0] = !send_buffer_sel ? signal_oe : 1'h0;
assign receive_buffer_oe[1] =  send_buffer_sel ? signal_oe : 1'h0;

wire f_scl;

pu_i2c_slave_driver #
  ( .I2C_DATA_WIDTH( I2C_DATA_WIDTH )
  , .DATA_WIDTH( DATA_WIDTH )
  , .ADDRES_DEVICE( 7'h47 )
  ) driver_slave
  ( .clk( clk )
  , .rst( rst )
  , .scl( f_scl )
  , .sda( sda )

  , .data_in( splitter_to_i2c )
  , .data_out( splitter_from_i2c )

  , .ready_write( i2c_ready_write )
  , .i2c_prepare( i2c_prepare )
  );

bounce_filter #( .DIV(BOUNCE_FILTER) ) f_scl_filter ( rst, clk, scl, f_scl
);

reg curr_sda;
always @(posedge clk) begin
  if (rst) begin
    curr_sda   <= 1'b1;
  end else begin
    curr_sda <= sda;
  end
end
```

```
reg wr;
always @(posedge clk) begin
  if (rst) begin
    wr <= 1'b0;
  end else begin
    if (i2c_ready_write) wr <= 0;
    if (i2c_prepare)     wr <= 1;
  end
end

always @(posedge clk) begin
  if (rst) begin
    flag_stop <= 1'b0;
  end else begin
    if (scl) begin
      if(curr_sda != sda && sda == 1'b1) begin
        flag_stop <= 1'b1;
      end else begin
        flag_stop <= 1'b0;
      end
    end
  end
end

always @( posedge clk ) begin
    if ( rst ) send_buffer_sel <= 0;
    else  if  (  signal_cycle  &&  scl  &&  sda  )  send_buffer_sel  <=
!send_buffer_sel;
end

reg [1:0] count_word;
always @( posedge clk ) begin
    if ( rst | flag_stop )        count_word <= 0;
    else if ( splitter_ready_sn ) count_word <= count_word + 1;
end

reg capture_word;
always @(posedge clk) begin
    if ( rst )            capture_word <= 0;
    else if ( flag_stop ) capture_word <= count_word != SIZE_WORDS;
end

assign attr_out[3:1] = 3'b000;
assign attr_out[INVALID] = capture_word;

assign data_out   = receive_buffer_oe[1] ? receive_buffer_data_out[1] :
                    receive_buffer_oe[0]  ?  receive_buffer_data_out[0]  :
{DATA_WIDTH{1'b0}};

endmodule
```