Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования

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

Факультет Компьютерных Технологий и Управления

Направление подготовки 230100 Специализация 230100.68.14

Квалификация (степень) магистр

Специальное звание магистр-инженер

Кафедра Вычислительной техники Группа 6114

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

на тему

Модульная система автоматического тестирования электронной аппаратуры

Автор магистерской диссертации Михайлов И. В.
(Фамилия, И., О.)

Научный руководитель Кустарев П.В.
(Фамилия, И., О.)

Руководитель магистерской программы Платунов А.Е.
(Фамилия, И., О.)

К з а щ и т е д о п у с т и т ь
Зав. кафедрой Алиев Т.И.
(Фамилия, И., О.)

“____” _____2014 г.

Санкт-Петербург, 2014 г.

Магистерская диссертация выполнена с оценкой _____

Дата защиты “____”_____2014 г.

Секретарь ГАК _____

Листов хранения _____

Чертежей хранения _____

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель и задачи проекта

Основная цель данного проекта — создание универсальной системы тестирования электронной аппаратуры, позволяющей проводить тестирование в автоматическом или полуавтоматическом режиме.

Задачи, решавшиеся в ходе реализации проекта:

- исследование существующих методов и языков описания тестов, выбор наиболее подходящего языка;
- разработка аппаратной архитектуры тестирующего комплекса, спецификация интерфейсов модулей системы, составление технических требований на реализацию модулей системы;
- разработка системы на кристалле основного модуля на базе ПЛИС;
- разработка технических требований на разработку программных средств загрузки тестовых программ и визуализации результатов.

Предметная область исследования

Предметная область данного исследования — информатика и вычислительная техника. Основная часть проекта представляет собой разработку системы на кристалле, также диссертация решает задачи из области встроенных систем и программной инженерии.

Практическая значимость проекта

Законченная система тестирования электронной аппаратуры, большая часть разработки которой описана в данной диссертации, была применена для внутренних нужд компании Stardex Oy. Также в будущем возможен выпуск данной системы для массовой продажи.

Объем и структура работы

Диссертационная работа содержит три основных главы.

В главе 1 дана постановка задачи магистерской диссертации и приведены характеристики проектируемой системы.

В главе 2 представлен проведенный анализ языков и методов тестирования. Даются критерии выбора языка тестирования, по этим критериям выбирается конкретный язык.

Глава 3 описывает разработанную систему тестирования в целом, дает описание отдельных ее модулей и связей между ними.

В главе 4 рассказывается о разработанной системе на кристалле исполнительного модуля. Каждый раздел этой главы содержит описание модуля (ядра) системы.

В приложениях даются исходные коды системы на кристалле.

ГЛАВА 1. ПОСТАНОВКА ЗАДАЧИ

1.1. Предметная область

На производствах электронного оборудования требуется проведение контроля качества выпускаемой продукции. Ручной контроль качества при помощи простейших приборов (мультиметра, осциллографа) возможен только в случае небольших партий, да и то дает много ошибок из-за человеческого фактора. Поэтому, особенно при наращивании производства до средних партий, встает задача автоматизации процесса контроля качества. В идеальном случае оператор линии контроля качества должен лишь подключить готовое изделие к специальному программируемому тестирующему стенду и дождаться результатов тестирования, но возможно и полуавтоматическое тестирование, когда оператор подключает изделие к стенду, но вручную задает воздействия и проверяет результат.

Кроме того, при разработке нового оборудования встает задача нагрузочного тестирования. Такая задача становится невозможной для выполнения вручную даже для одного экземпляра изделия. Требуется создание специального программируемого тестирующего стенда, выполняющего такое тестирование.

Данный проект призван решить задачу автоматизации тестирования продукции и контроля качества.

1.2. Постановка задачи

Целью проекта является создание системы тестирования электронного

оборудования.

Требования, предъявляемые к системе тестирования:

- Автономность и мобильность. Система не должна требовать подключения отдельного компьютера для использования.
- Наличие графического интерфейса пользователя и простота использования.
- Универсальность. Должна быть возможность тестирования разных классов сигналов: цифровых, аналоговых, смешанных, токовых петель, дифференциальных пар и т. д. Также должна быть возможность тестирования силовых цепей.

Параметры системы:

- Общее количество линий ввода-вывода — 30-50.
- Допустимый порядок временных погрешностей подачи и измерения цифровых сигналов — сотни наносекунд.

Задачи проекта:

- Выполнить исследование существующих языков тестирования, выбрать наиболее подходящий;
- Разработать архитектуру системы тестирования;
- Реализовать систему тестирования.

ГЛАВА 2. ИССЛЕДОВАНИЕ ЯЗЫКОВ И МЕТОДОВ ТЕСТИРОВАНИЯ

Язык тестирования — это проблемно-ориентированный язык программирования, предназначенный для описания тестов какой-либо системы (программной или аппаратной). Языки тестирования вместе с системами автоматического тестирования используются для автоматизации процесса тестирования.

Основное преимущество проблемно-ориентированного языка тестирования перед языками программирования общего назначения состоит в том, что проблемно-ориентированный язык позволяет написать набор тестов в максимально сжатые сроки в краткой и понятной форме. Это преимущество важно в связи с тем, что, как правило, бюджет и временные рамки для тестирования программного или аппаратного обеспечения сильно ограничены. Кроме того, проблемно-ориентированные языки тестирования, благодаря своей лаконичности, снижают вероятность человеческой ошибки при написании тестов. В связи с этим было принято решение рассматривать в данной работе только проблемно-ориентированные языки тестирования и не рассматривать языки программирования общего назначения.

В рамках данной работы был проведен анализ существующих языков тестирования электронной аппаратуры и программного обеспечения. Был выбран язык тестирования, наиболее подходящей для достижения поставленной цели.

2.1. Требования к языку тестирования

Основной принцип тестирования — установка предусловий теста и проверка ответа системы. Языки, не удовлетворяющие данному принципу, не рассматривались в ходе сравнения.

Критериями, по которым сравнивались языки, были (в порядке убывания значимости):

- Расширяемость языка и возможность его использования для тестирования аппаратных, а не только программных систем.
- Низкий порог вхождения и возможность написания тестов человеком с низкой квалификацией, так как работой с системой будут заниматься сотрудники отдела контроля качества, а не специалисты отдела разработки.
- Реальные результаты использования языка, наличие инструментов — для ускорения разработки тестирующей системы.
- Поддержка языком сложных функциональных конструкций (ветвления, рекурсии, циклов) для исполнения сложных тестов (например, нагрузочного тестирования).

2.2. Рассмотренные языки тестирования

ETSI TDL

ETSI TDL (test description language, язык описания тестов), разрабатываемый в Европейском институте по стандартизации в области телекоммуникаций (ETSI), является прототипом языка тестирования как

программного обеспечения, так и аппаратных продуктов. На 2013 год язык еще не является до конца стандартизированным, стандарт на синтаксис языка предполагается выпустить в 2015 году [1].

TestTalk

TestTalk — платформо- и инструментонезависимый язык, который позволяет тестировщикам описывать программные тесты проблемно-ориентированным образом [2], [3]. Для данной работы язык является слишком абстрактным, инструменты для работы с ним отсутствуют.

Cucumber

Cucumber — язык для ведения разработки через поведение (behavior-driven development) [4]. Язык является подмножеством языка программирования Ruby [5]. Имеет гибкий синтаксис, максимально приближенный к естественному человеческому языку. Синтаксис языка позволяет переводить ключевые слова на любые человеческие языки, а потому удобен для низкоквалифицированного разработчика тестовых программ. Существует возможность тестировать с его помощью программы, написанные не только на Ruby, но и на других языках программирования.

Test Template Framework и Z-нотация

Test Template Framework [6] — средство для тестирования на основе модели. Имеет под собой сложный математический аппарат Z-нотации ([7], [8]), в связи с чем не подходит для низкоквалифицированных разработчиков тестовых программ.

2.3. Сравнение рассмотренных языков тестирования

Стандарт на синтаксис языка ETSI TDL, рассмотренного в предыдущем параграфе, еще не создан, разработана лишь концепция. Поэтому данный язык не подходит для задачи проекта.

Сравним оставшиеся TestTalk, Cucumber и Test Template Framework, рассмотренные в предыдущем параграфе, по выбранным ранее критерием.

Расширяемость языка

Самую удобную и быструю расширяемость, благодаря использованию языка программирования Ruby как основы, показывает язык **Cucumber**. Возможность интернационализации, то есть использования ключевых слов на любых человеческих языках (а не только на английском) имеется сразу же после установки пакета языка. Возможность проводить тестирование аппаратуры появляется после создания программной прослойки, связывающей программный и аппаратный уровни.

Test Template Framework, являясь скорее математической моделью, чем языком программирования, так же показывает хорошую расширяемость.

TestTalk является платформой для создания проблемно-ориентированных языков тестирования, что тоже говорит о его хорошей расширяемости.

Порог вхождения

Интернационализация, простота синтаксиса и подробная документация делают **Cucumber** языком с наиболее низким порогом вхождения из рассмотренных.

Test Template Framework имеет самый высокий порог вхождения из-за сильного использования математического аппарата Z-нотации.

TestTalk, не являясь настолько математичным, как Test Template Framework, все равно обладает достаточно высоким порогом вхождения из-за почти полного отсутствия документации и сложного синтаксиса.

Наличие инструментов

Благодаря использованию языка программирования Ruby как основы для языка **Cucumber**, для этого языка становится доступным большое количество инструментов и библиотек, в том числе библиотек для взаимодействия с аппаратными интерфейсами компьютера.

Test Template Framework — это скорее математическая модель, чем полноценный язык программирования, поэтому выбор инструментов зависит от конкретной реализации этого средства описания тестов.

TestTalk является, по словам разработчиков [2], инструментонезависимым, и поэтому требует сложной конфигурации и доработки.

Поддержка сложных функциональных конструкций

По этому критерию лидирует **Test Template Framework**. С помощью этого средства можно создавать тестовые программы какой угодно сложности, но их создание требует глубокого изучения Z-нотации.

Язык **Cucumber** поддерживает всего три базовых оператора:

- Given – установка предусловий теста,

- When – задание входных воздействий,
- Then – проверка полученных результатов,

что делает его языком с наименьшими функциональными возможностями из перечисленных. С другой стороны, так как Cucumber является расширением Ruby, то становится возможным написание частей тестовых программ любой сложности на языке Ruby, что, однако, повышает порог вхождения и уменьшает читаемость тестовых программ.

TestTalk поддерживает большой набор различных конструкций, в том числе загрузку правил из файла, выдерживания пауз и т. д.

2.4. Выбранный язык тестирования

В результате сравнения языков и технологий было принято решение использовать язык Cucumber [4] в качестве языка описания тестов. Несмотря на малое количество поддерживаемых языковых конструкций, этот язык был выбран благодаря другим критериям: низкому порогу вхождения разработчика тестовых программ, наличию большого количества инструментов и простой расширяемости языка.

ГЛАВА 3. МОДУЛЬНАЯ ТЕСТИРУЮЩАЯ СИСТЕМА

Согласно постановке задачи, требуется разработать тестирующую систему, способную выдавать и принимать широкий класс сигналов (цифровые сигналы разных стандартов, аналоговые сигналы, токовые петли). Рассматривалась возможность построения монолитной системы, основная и единственная плата которой содержала бы процессор, исполняющий тестовые программы, и набор всевозможных периферийных интерфейсов. Но из-за требуемой универсальности такой вариант сложен в реализации и нецелесообразен. Проще сделать модульную систему, легко сменяемые периферийные модули которой содержали бы необходимые интерфейсы для тестирования конкретного оборудования.

Другими требованиями к системе являются простота использования и автономность (то есть, система не должна требовать подключения компьютера для работы). Сочетание этих двух требований приводит к необходимости поиска производительной и компактной вычислительной платформы для реализации графического интерфейса пользователя. К тому же, вычислительная платформа должна обеспечивать достаточное количество линий связи с периферийными модулями. Рассматривалась следующие варианты вычислительных платформ:

- Микроконтроллер с контроллером жидкокристаллического дисплея и сенсорного экрана. Главным достоинством этого варианта является его компактность. Основные недостатки — сложность в реализации графического интерфейса пользователя и низкая производительность.
- Одноплатный компьютер на базе производительного процессора с

ядром ARM. К таким компьютерам относятся, в частности, Raspberry Pi и BeagleBoard. Достоинствами такого решения являются относительно высокая производительность, компактность и быстрота разработки нужного программного обеспечения. Основной недостаток — необходимость пересборки и адаптации существующего программного обеспечения под аппаратную платформу, отличную от Intel x86.

- Материнская плата стандарта microATX с процессором Intel Atom или аналогичным. Главные достоинства этого варианта — высокая производительность и большое количество качественного программного обеспечения, простота его разработки. К недостаткам можно отнести малую компактность (требуется подключение внешнего носителя данных, громоздких кабелей питания) и отсутствие каких-либо цифровых интерфейсов, кроме USB и последовательных портов.

Ни один из рассмотренных вариантов не предоставляет возможности подключения 30-50 периферийных линий ввода-вывода, чего требует постановка задачи. По остальным характеристикам наиболее подходящим вариантом вычислительной платформы представляется одноплатный компьютер на базе процессора с ядром ARM. В качестве конкретного варианта было решено остановиться на компьютере BeagleBoard [9], как имеющем относительно высокую производительность и низкую цену.

Согласно постановке задачи, при подаче и измерении цифровых сигналов требуется соблюдение временных интервалов с точностью порядка сотен наносекунд, а так же требуется наличие как минимум 30 линий ввода-вывода. Для таких целей подходят или очень высокопроизводительные

микроконтроллеры, или программируемые логические интегральные схемы (ПЛИС). Систему реального времени со столь жесткими ограничениями по точности быстрее, а значит экономически выгоднее, построить на базе ПЛИС.

3.1. Разработанная архитектура

На рисунке 3.1 изображена разработанная архитектура тестирующей системы.

Система состоит из следующих аппаратных блоков:

- Компьютерный модуль (BeagleBoard). Содержит программный комплекс, предоставляющий пользователю интерфейс взаимодействия с тестирующей системой. Программный комплекс состоит из следующих частей:
  - Интерфейс пользователя. Позволяет пользователю выбирать тестовые программы, смотреть результаты и журналы тестирования.
  - Компилятор тестовых программ. Преобразует тестовые программы в машинный код специализированного процессора.
  - Загрузчик. Выполняет загрузку конфигурации ПЛИС на исполнительный модуль, а так же загрузку откомпилированных тестовых программ и получение результатов тестирования.

- Исполнительный модуль. Содержит систему на кристалле на базе ПЛИС (см. главу 4). Исполнительный модуль подключается к компьютерному при помощи интерфейса, описанного в параграфе 3.2.

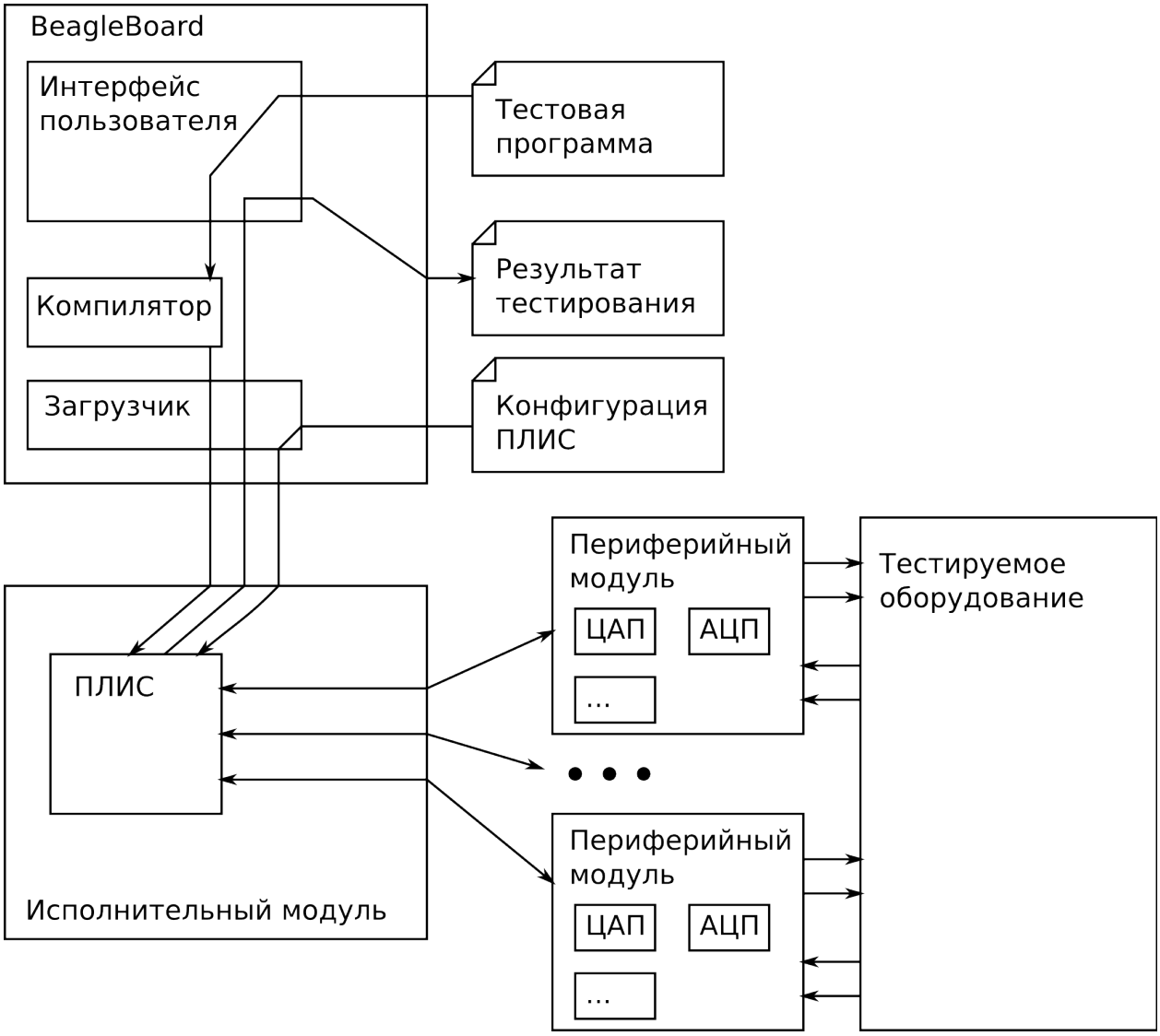

Рис 3.1: Архитектура тестирующей системы

- Периферийные модули. Обеспечивают преобразование сигналов в требуемый для тестируемого оборудования стандарт. Подключаются к исполнительному модулю по интерфейсам, описанным в параграфе 3.2.

3.2. Интерфейсы связи модулей системы

В рамках данной работы были разработаны интерфейсы связи модулей системы. Необходимо было разработать два интерфейса:

- Интерфейс связи компьютерного модуля (BeagleBoard) и исполнительного модуля и
- Интерфейс связи исполнительного модуля с периферийными.

Оба интерфейса имеют топологию "точка-точка".

Интерфейс связи компьютерного и исполнительного модулей ограничен разъемом интерфейса расширения BeagleBoard [9]. Этот интерфейс используется и для загрузки конфигурации ПЛИС, и для общения с системой на кристалле.

Интерфейсы связи исполнительного модуля с периферийными должны иметь лишь цифровые линии передачи данных. Благодаря универсальности ПЛИС линии передачи данных этих интерфейсов тоже можно сделать универсальными, то есть независимыми от аппаратных протоколов.

Интерфейс связи компьютерного модуля (BeagleBoard) и исполнительного модуля представлен в таблице 1. Полужирным шрифтом в таблице выделены сигналы, используемые для загрузки конфигурации ПЛИС на исполнительный модуль.

Интерфейс связи исполнительного модуля с периферийными представлен в таблице . Несмотря на то, что в текущей версии системы на кристалле модули SPI и I^2C не реализованы, сигналы этих интерфейсов выведены в спецификацию для упрощения дальнейшего расширения.

Для обоих интерфейсов используются стандартные 28-контактные разъемы (14 контактов в два ряда) с шагом 2.54 мм.

Вывод разъема	**Сигнал Beagleboard**	**Сигнал исполнительного модуля**
1	VIO_1V8	VIO1, VIO2
2	DC_5V	NC
3	io	BANK1-2 IO
4	io	BANK1-2 IO
5	io	BANK1-2 IO
6	UART2_TX	BANK1-2 IO
7	io	BANK1-2 IO
8	UART2_RX	BANK1-2 IO
9	io	BANK1-2 IO
10	io	BANK1-2 IO
11	**McSPI3_CS0/GPIO_135**	**nCONFIG**
12	**GPIO_158**	**nSTATUS**
13	io	BANK1-2 IO
14	io	BANK1-2 IO
15	**GPIO_133**	**CONF_DONE**
16	io	BANK1-2 IO
17	McSPI3_SOMI	BANK1-2 IO
18	io	BANK1-2 IO
19	**McSPI3_SIMO**	**DATA[0]**
20	io	BANK1-2 IO
21	**McSPI3_CLK**	**DCLK**
22	io	BANK1-2 IO
23	io	BANK1-2 IO
24	io	BANK1-2 IO
25	NC	NC
26	NC	NC
27	GND	GND
28	GND	GND

Таблица 1: Интерфейс связи компьютерного (BeagleBoard) и исполнительного модулей

Вывод разъема	Направление	Сигнал
1	От исполнительного модуля	3.3V
2	К исполнительному модулю	VIOx
3	От исполнительного модуля	MOSI1
4	К исполнительному модулю	MISO1
5	От исполнительного модуля	SCK1
6	От исполнительного модуля	SS1
7	От исполнительного модуля	MOSI2
8	К исполнительному модулю	MISO2
9	От исполнительного модуля	SCK2
10	От исполнительного модуля	SS2
11	Двунаправленный	SDA1
12	От исполнительного модуля	SCL1
13	Двунаправленный	SDA2
14	От исполнительного модуля	SCL2
15-26	Двунаправленный	IOx
27, 28	Общий	GND

Таблица 2: Интерфейс связи исполнительного и периферийного модуля

3.3. Выводы

В данной главе описана архитектура системы и сформулированы требования к ее модулям.

На процессорный модуль возлагается задача взаимодействия с пользователем через графический интерфейс и работа с исполнительным модулем согласно командам пользователя.

Система на кристалле исполнительного модуля должна исполнять тестовые программы в режиме реального времени.

Периферийные модули должны преобразовывать сигналы основного

модуля в сигналы тестируемого оборудования и иметь максимально простую конструкцию.

ГЛАВА 4. СИСТЕМА НА КРИСТАЛЛЕ

В этой главе описана система на кристалле, загружаемая на ПЛИС исполнительного модуля тестирующей системы.

4.1. Требования к системе на кристалле

Согласно предыдущей главе, система на кристалле должна исполнять тестовые программы в режиме реального времени. Для этих целей был разработан специализированный процессор. Для упрощения загрузки тестовых программ было решено использовать протокол Modbus, как имеющий простой физический уровень (последовательный порт), множество программных средств для взаимодействия с ним и открытую спецификацию. Для хранения тестовых программ требуется блок оперативной памяти, а для взаимодействия с периферийными модулями — порты ввода-вывода. Все эти модули должны быть объединены внутрикристальной шиной. Рассматривались шины AMBA и Wishbone. Наиболее документированной и простой оказалась шина Wishbone, поэтому выбор остановился на ней.

Так как выбранный язык тестирования поддерживает только простые конструкции и не поддерживает ветвления и циклов, то и специализированный процессор тоже может не поддерживать циклы и ветвления, а обязан поддерживать лишь проверки условий и задержки.

4.2. Архитектура системы на кристалле

На рисунке 4.1 изображена разработанная архитектура системы на кристалле.

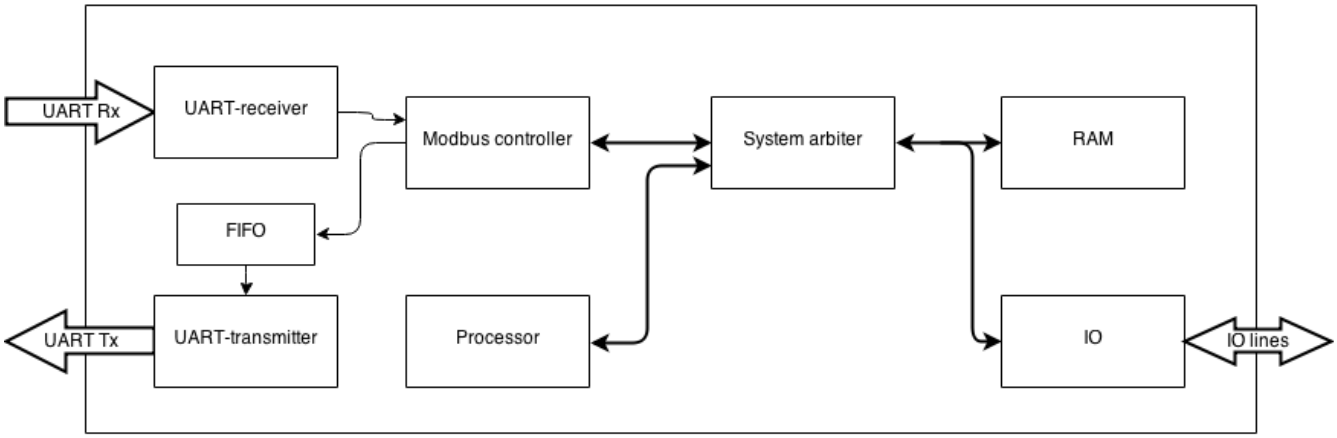

Рис 4.1: Архитектура системы на кристалле

Разработанная система на кристалле состоит из следующих модулей:

- UART-receiver – UART-приемник. Принимает данные по асинхронному последовательному интерфейсу и передает их контроллеру Modbus [10].
- UART-transmitter – UART-передатчик. Считывает данные из очереди и передает их по асинхронному последовательному интерфейсу.
- FIFO – циклический буфер. Служит для буферизации данных, передаваемых контроллером Modbus.
- Modbus controller – модуль, реализующий протокол Modbus. Является ведущим устройством шины Wishbone [11].
- Processor – процессор, выполняющий тестовые программы. Является ведущим устройством шины Wishbone.
- System arbiter – системный арбитр, управляющий доступом ведущих устройств к шине Wishbone.
- RAM – блок оперативной памяти. Является ведомым устройством

Wishbone.

- IO – блок контроллеров ввода-вывода. Является ведомым устройством Wishbone.

Модули соединены шиной Wishbone (жирные линии на рис. 4.1) и различными нестандартными шинами (тонкие линии).

4.3. UART-приемник

Интерфейс UART-приемника

На рис. 4.2 показан интерфейс модуля UART-приемника.

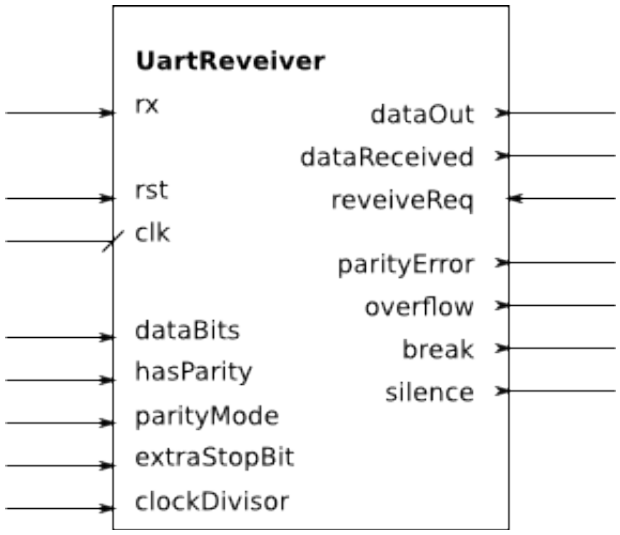

Рис 4.2: Интерфейс модуля UART-приемника

Модуль имеет следующие входные и выходные сигналы:

- clk – системный тактовый сигнал.
- rst – сигнал синхронного сброса.
- rx – линия приема данных.

- Группа конфигурационных сигналов:
  - dataBits – два бита – длина символа данных (Таблица 3).

Значение сигнала dataBits	Количество бит данных (без учета бита четности)
00	5
01	6
10	7
11	8

Таблица 3: Выбор длины символа в модуле UART-приемника

  - hasParity – один бит – наличие или отсутствие бита четности.
  - parityMode – два бита – режим проверки четности (Таблица 4).

Значение сигнала parityMode	Режим проверки четности	Пояснение
00	всегда ноль	бит четности должен быть равен нулю
01	нечетный	число всех бит, равных единице, включая бит четности, нечетно
10	четный	число всех бит, равных единице, включая бит четности, четно
11	всегда единица	бит четности должен быть равен единице

Таблица 4: Выбор режима проверки четности в модуле UART-приемника

  - extraStopBit – наличие дополнительного стопового бита. В случае нуля на этом входе принимается один стоповый бит, в случае единицы – два.
  - clockDivisor – делитель частоты системного тактового сигнала, определяющий скорость приема данных. Разрядность этого сигнала задается параметром CLOCK_DIVISOR_WIDTH. Частота приема данных (*baud-rate*) вычисляется по следующей формуле:

    f_uart = f_clk / (2 * clockDivisor + 2)

Сигналы этой группы защелкиваются во внутренние регистры по фронту тактового сигнала во время ожидания символа. Во время приема символа изменения сигналов игнорируются.

- dataOut – 9 бит – шина данных, на которой выставляется принятый символ. Данные выставляются по фронту тактового сигнала одновременно с сигналом dataReceived (см. рис. 4.3).
- dataReceived – 1 бит – флаг приема данных. Устанавливается по фронту тактового сигнала при успешном приеме символа. Сбрасывается по фронту тактового сигнала при наличии единицы на входе receiveReq (см. рис. 4.3).

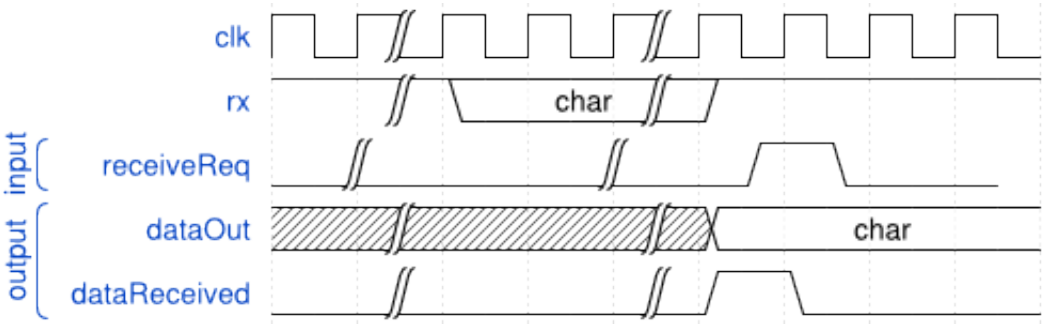

Рис 4.3: Временная диаграмма приема символа

- reveiveReq – 1 бит – сообщение модулю о том, что принятые данные были прочитаны. При наличии единицы на этом входе во время фронта тактового сигнала сигнал dataReceived сбрасывается в ноль (см. рис. 4.3).
- Группа информационных сигналов:
  - parityError – ошибка четности.
  - overflow – был принят новый символ, когда предыдущий еще не был

прочитан (см. рис. 4.4).

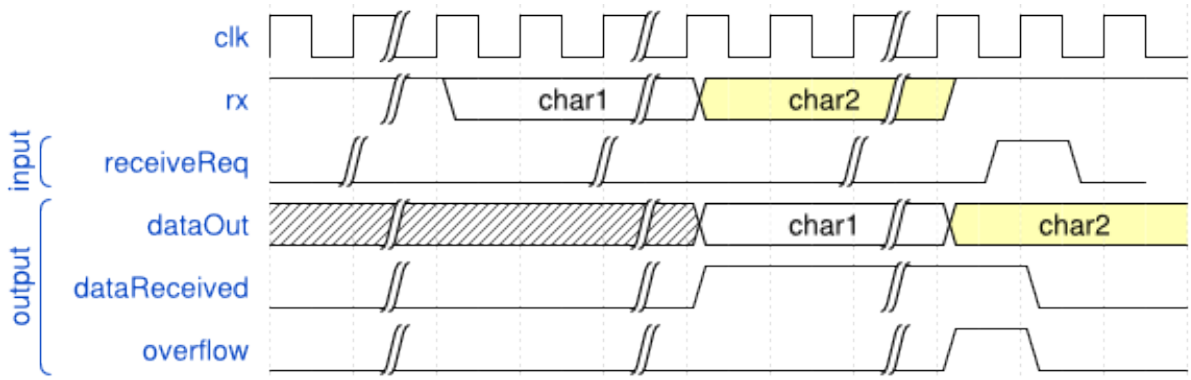

Рис 4.4: Временная диаграмма приема символа и переполнения внутреннего регистра UART-приемника

- break – ошибка приема: стоповый бит оказался равен 0, а не 1.
- silence – на линии не было данных в течение времени, соответствующем минимум трем символам.

Управляющий конечный автомат

UART-приемник управляется конечным автоматом. Диаграмма переходов автомата представленна на рис. 4.5.

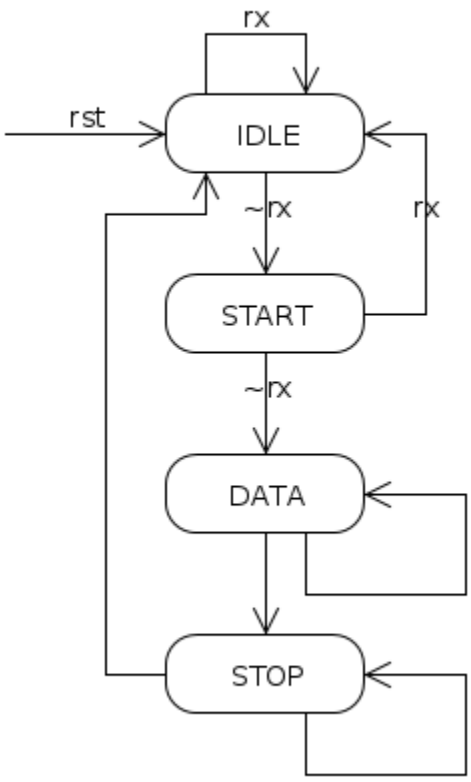

Рис 4.5: Диаграмма переходов конечного автомата UART-приемника

Переходы между состояниями (кроме переходов из состояния IDLE) происходят по фронту тактового сигнала при наличии единицы на линии uartClk, то есть в середине каждого принимаемого бита (рис. 4.6). Переход из состояния IDLE в состояние START происходит по фронту тактового сигнала при наличии нуля на линии rx. Дальнейшие переходы автомата происходят только тогда, когда значение счетчика равно значению делителя частоты. Счетчик сбрасывается в ноль, когда его значение достигает двукратного значения делителя.

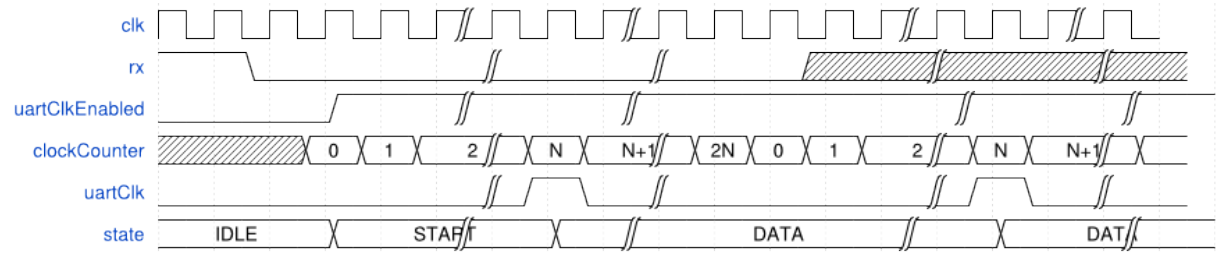

Рис 4.6: Начало приема данных UART-приемником. На диаграмме N соответствует значению clockDivisor.

Автомат имеет следующие состояния:

- IDLE – ожидание данных. Автомат переходит в это состояние по сигналу синхронного сброса rst и остается в нем, пока не обнаружит ноль на входе rx. При обнаружении нуля автомат переходит в состояние START, одновременно с чем запускается счетчик делителя частоты.
- START – прием стартового бита. Автомат переходит из этого состояния в состояние DATA при наличии нуля на rx и в IDLE при обнаружении на rx единицы. Так как переходы автомата происходят на середине приема бита, такая проверка позволяет отсечь помехи, то есть короткие, короче половины ожидаемого бита, нулевые импульсы.
- DATA – прием символа данных. Автомат находится в этом состоянии в течении стольких тактов, сколько нужно для приема всех бит символа данных. После этого происходит переход в состояние STOP.
- STOP – прием стоповых битов. Автомат находится в этом состоянии в течение одного или двух входных битов в зависимости от значения extraStopBit. Если были хоть один из ожидаемых стоповых битов оказался нулем, сигнал break приобретает значение 1. После приема

стоповых битов автомат переходит в состояние IDLE.

4.4. UART-передатчик

Интерфейс UART-передатчика

На рис. 4.7 показан интерфейс модуля UART-передатчика.

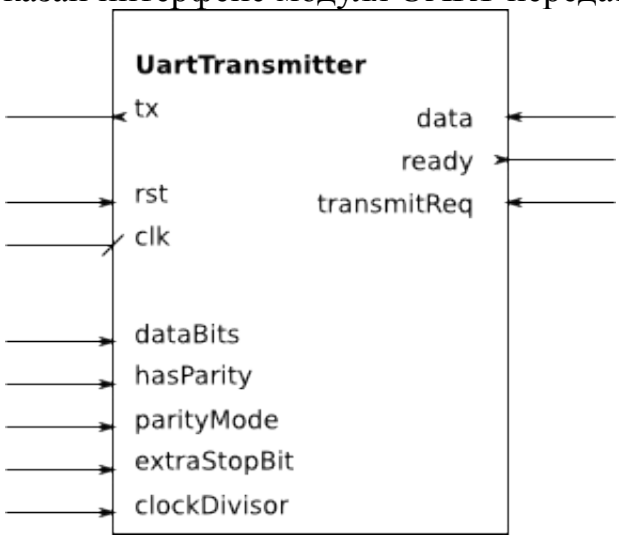

Рис 4.7: Интерфейс модуля UART-передатчика

Модуль имеет следующие входные и выходные сигналы:

- clk – системный тактовый сигнал.
- rst – сигнал синхронного сброса.
- tx – линия передачи данных.
- Группа конфигурационных сигналов:
  - dataBits – два бита – длина символа данных (Таблица 5).

Значение сигнала dataBits	Количество бит данных (без учета бита четности)
00	5
01	6
10	7
11	8

Таблица 5: Выбор длины символа в модуле UART-передатчика

- hasParity – один бит – наличие или отсутствие бита четности.
- parityMode – два бита – режим бита четности (Таблица 6).

Значение сигнала parityMode	Режим проверки четности	Пояснение
00	всегда ноль	бит четности равен нулю
01	нечетный	число всех бит, равных единице, включая бит четности, нечетно
10	четный	число всех бит, равных единице, включая бит четности, четно
11	всегда единица	бит четности равен единице

Таблица 6: Выбор режима бита четности в модуле UART-передатчика

- extraStopBit – наличие дополнительного стопового бита. В случае нуля на этом входе передается один стоповый бит, в случае единицы – два.
- clockDivisor – делитель частоты системного тактового сигнала, определяющий скорость передачи данных. Разрядность этого сигнала задается параметром CLOCK_DIVISOR_WIDTH. Частота передачи данных (*baud-rate*) вычисляется по следующей формуле:

  f_uart = f_clk / (2 * clockDivisor + 2)

Сигналы этой группы защелкиваются во внутренние регистры по фронту тактового сигнала при начале передачи символа. Во время передачи символа изменения этих сигналов игнорируются.

- data – 8 бит – шина данных, с которой принимается символ для передачи. Данные защелкиваются по фронту тактового сигнала при наличии единицы на входе transmitReq.
- ready – 1 бит – флаг готовности к передаче символа. Установлен в единицу, если модуль не передает данные и в ноль, если передает (рис. 4.8), то есть единица выставляется тогда и только тогда, когда управляющий автомат находится в состоянии IDLE.

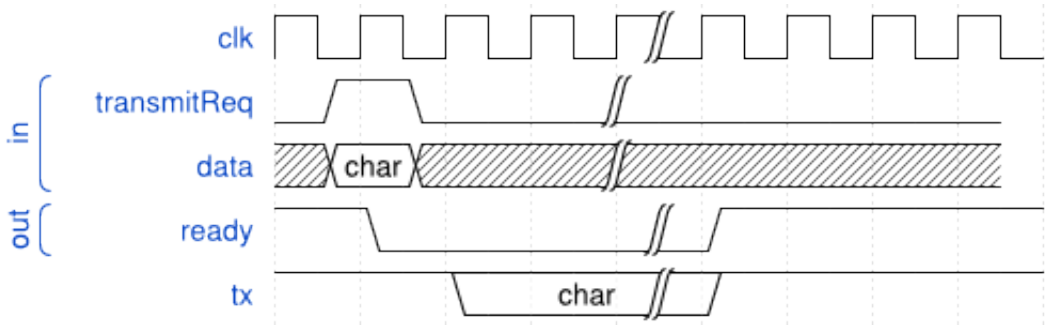

Рис 4.8: Временная диаграмма передачи символа

- transmitReq – 1 бит – сообщение модулю о том, что следует считать символ с шины данных и начать передачу. При наличии единицы на этом входе во время фронта тактового сигнала сигнал ready сбрасывается в ноль (см. рис. 4.8) и начинается передача данных.

Управляющий конечный автомат

UART-передатчик управляется конечным автоматом. Диаграмма переходов автомата представленна на рис. 4.9.

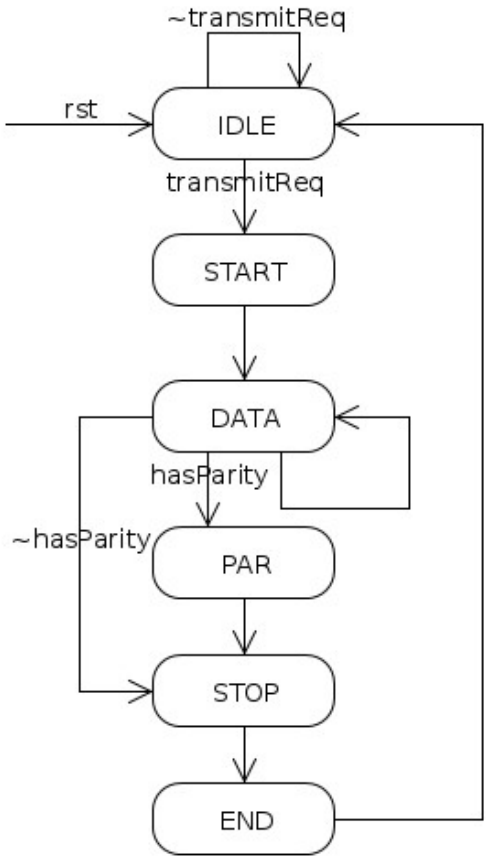

Рис 4.9: Диаграмма переходов конечного автомата UART-передатчика

Переходы между состояниями (кроме переходов из состояния IDLE) происходят по фронту тактового сигнала при наличии единицы на линии uartClk, то есть в начале каждого передаваемого бита. Переход из состояния IDLE в состояние START происходит по фронту тактового сигнала при наличии единицы на линии transmitReq. Дальнейшие переходы автомата

происходят только тогда, когда значение счетчика равно значению делителя частоты. Счетчик сбрасывается в ноль, когда его значение достигает двукратного значения делителя.

Автомат имеет следующие состояния:

- IDLE – ожидание данных. Автомат переходит в это состояние по сигналу синхронного сброса rst и остается в нем, пока не обнаружит единицу на входе transmitReq. При обнаружении единицы автомат переходит в состояние START, одновременно с чем запускается счетчик делителя частоты.
- START – передача стартового бита. В этом состоянии на выход tx подается значение 0. Из состояния START автомат безусловно переходит в состояние DATA.
- DATA – передача символа данных. Автомат находится в этом состоянии в течении стольких тактов, сколько нужно для передачи всех бит символа данных (кроме бита четности). После этого происходит переход в состояние STOP, если защелкнутое в начале передачи значение hasParity ноль, и в состояние PAR, если это значение – единица, то есть нужно передать бит четности.
- PAR – передача бита четности. На линию tx подается вычисленное значение бита четности.
- STOP – передача стоповых битов. Автомат находится в этом состоянии в течение одного или двух битов в зависимости от защелкнутого в начале передачи значения extraStopBit. После передачи стоповых битов

автомат переходит в состояние END.

- END – конец передачи. Автомат безусловно переходит в состояние IDLE.

4.5. FIFO-буфер

FIFO-буфер (*first in – first out, первым пришел – первым ушел*) – модуль для буферизации потоковых данных между высоко- и низкоскоростными модулями.

Интерфейс FIFO-буфера

На рис. 4.10 показан интерфейс модуля FIFO-буфера.

Рис 4.10: Интерфейс модуля FIFO-буфера

Модуль имеет следующие входные и выходные сигналы:

- clk – системный тактовый сигнал.
- rst – сигнал синхронного сброса.
- dataIn – шина записываемых данных. Разрядность задается параметром DATA_WIDTH.

- writeReq – запрос записи данных. Если внутренний буфер не переполнен, при наличии единицы на этом входе по фронту тактового сигнала осуществляется запись данных в конец очереди.
- writeAck – сигнал подтверждения записи данных. Устанавливается в единицу по фронту тактового сигнала в том и только в том случае, если на входе writeReq установлена единица и внутренний буфер не переполнен (рис. 4.11).
- full – сигнал заполненности внутреннего буфера (рис. 4.12).
- dataOut – шина считываемых данных. Разрядность задается параметром DATA_WIDTH.
- readReq – запрос чтения данных. Если внутренний буфер не пуст, при наличии единицы на этом входе по фронту тактового сигнала осуществляется отбрасывание головы очереди, а значение, бывшее в очереди первым, выставляется на шине dataOut.
- readAck – сигнал подтверждения чтения данных. Устанавливается в единицу по фронту тактового сигнала в том и только в том случае, если на входе readReq установлена единица и внутренний буфер не пуст (рис. 4.11).
- empty – сигнал пустоты внутреннего буфера (рис. 4.11).

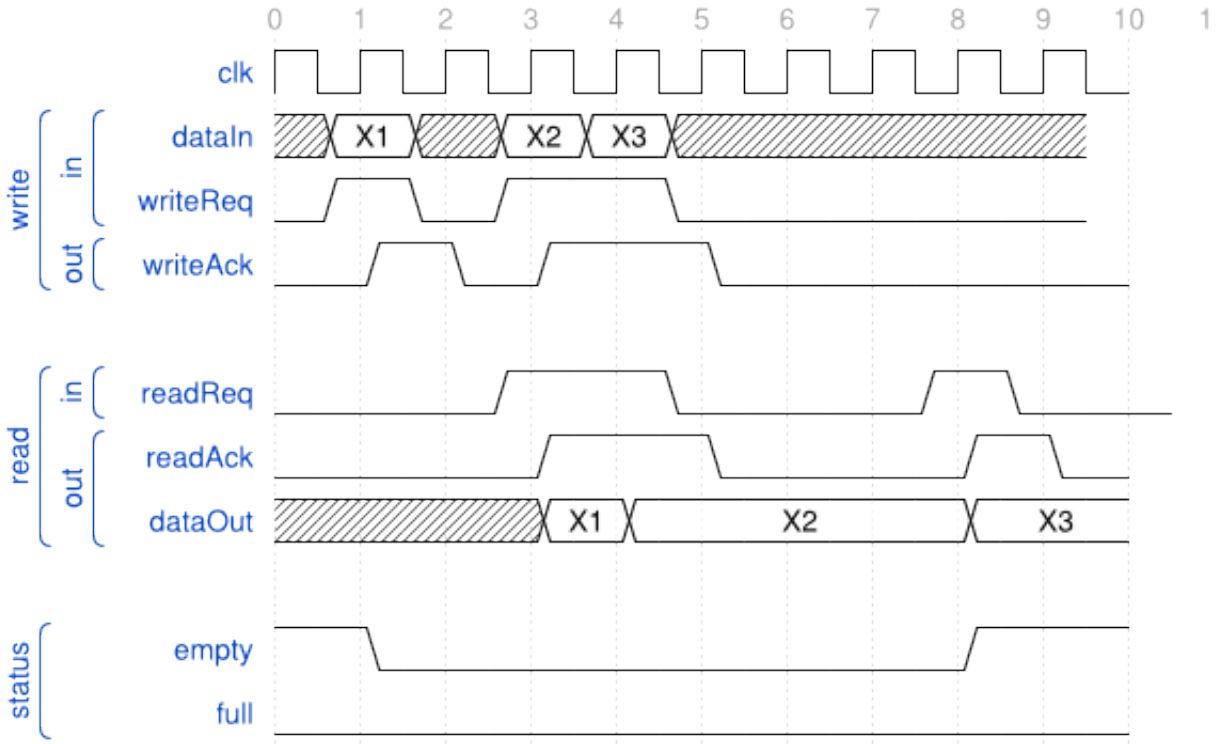

Рис 4.11: Временная диаграмма работы FIFO-буфера

На рисунке 4.11 показана временная диаграмма работы FIFO-буфера. Во время фронта 1 происходит запись в очередь значения X1, а во время фронтов 3 и 4 – запись значений X2 и X3. Одновременно на такте 3 происходит чтение первого значения из очереди, на такте 4 – второго, а на такте 8 – третьего. После записи первого значения сигнал empty принимает нулевое значение, а после чтения последнего – переходит обратно в единицу.

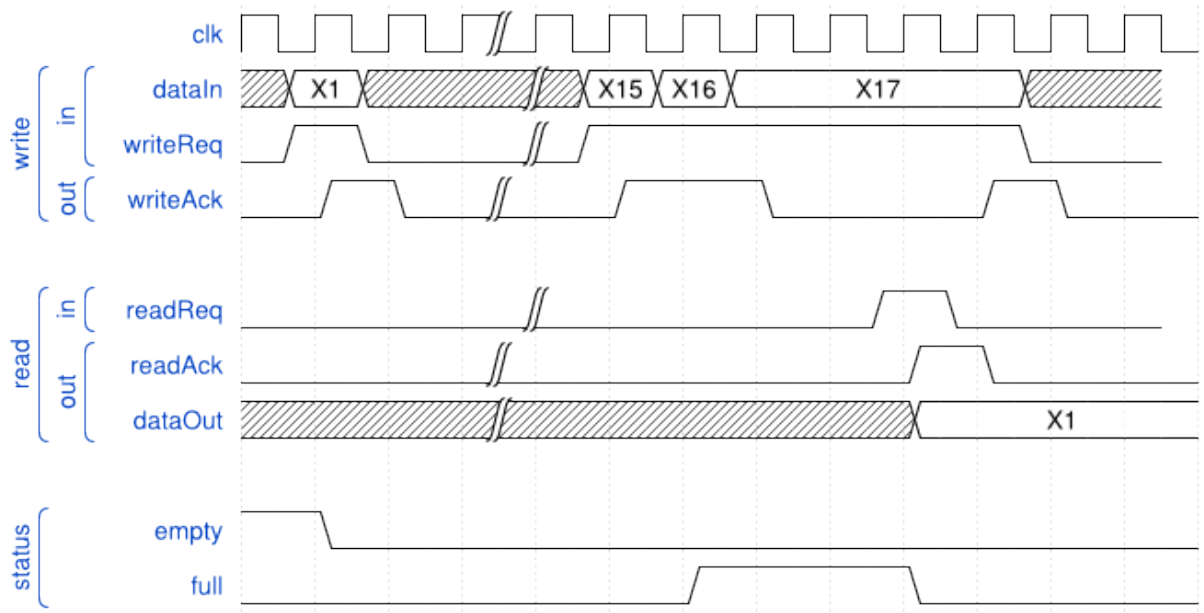

Рис 4.12: Переполнение FIFO-буфера

При переполнении очереди сигнал full устанавливается в единицу, а writeAck остается нулевым, пока из очереди не будут прочитаны данные (рис. 4.12). В данном примере предполагается, что размер очереди равен 16.

Организация FIFO-буфера

FIFO-буфер организован при помощи регистрового массива buffer и регистров getPtr и putPtr. Эти регистры хранят индексы головы и хвоста очереди соответственно, причем putPtr указывает на ячейку, которая еще пуста, и в которую будет записано следующее значение. Поэтому буфер считается пустым, если getPtr равно putPtr, и считается полным, если getPtr равно putPtr + 1.

4.6. Контроллер Modbus

Требования к контроллеру Modbus

Задачей системы на кристалле является исполнение тестовых программ в режиме реального времени. Поэтому основной задачей контроллера Modbus является загрузка этих программ во внутреннюю память системы на кристалле и проверка результатов их выполнения. Для выполнения этой задачи достаточно реализовать функции записи и чтения регистров.

Интерфейс Modbus-контроллера

Modbus-контроллер является ведущим устройством шины Wishbone и имеет следующий интерфейс (рис. 4.13):

Можно выделить следующие группы сигналов:

- Тактовый сигнал и сигнал сброса:
    - clk — системный тактовый сигнал, одновременно является тактовым сигналом шины Wishbone;
    - rst — сигнал синхронного сброса.
- Шина Wishbone:
    - wbAdrO — линия адреса. Имеет разрядность, устанавливаемую параметром ADDRESS_WIDTH;

◦ wbDatO — линия исходящих данных. Имеет разрядность,

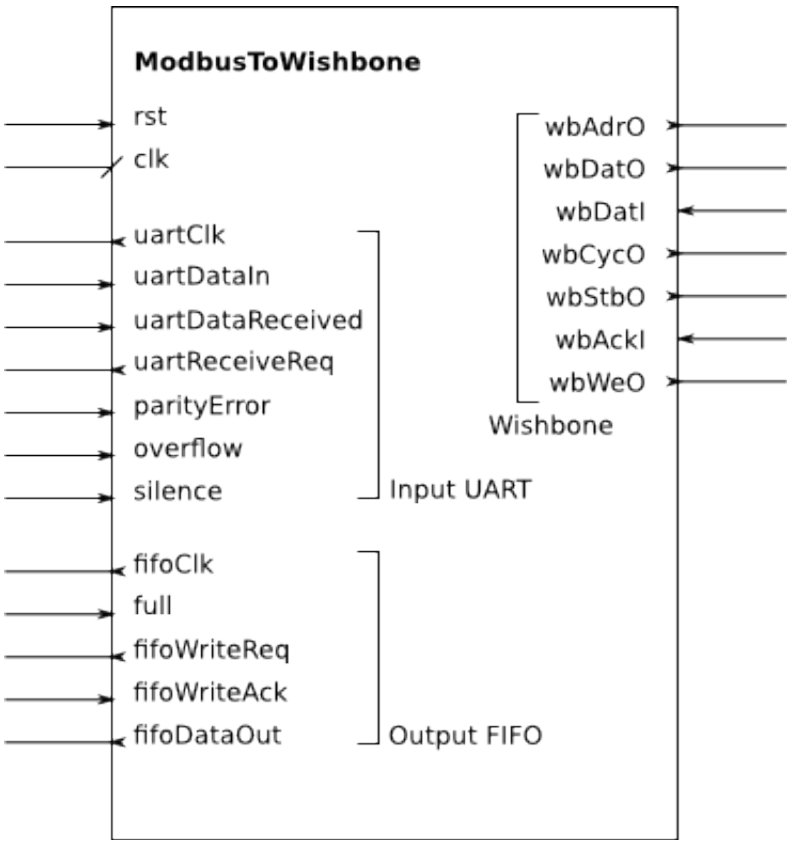

Рис 4.13: Интерфейс Modbus-контроллера

устанавливаемую параметром DATA_WIDTH;

◦ wbDatI — линия входящих данных. Имеет разрядность, устанавливаемую параметром DATA_WIDTH;

◦ wbCycO — индикация валидного цикла;

◦ wbStbO — индикация валидного цикла передачи данных;

◦ wbAckI — сигнал подтверждения. Высокий уровень означает, что текущий цикл успешно завершился;

◦ wbWeO — индикация того, является ли текущий цикл циклом

чтения или циклом записи.

- Шина приема данных через UART:
  - uartClk — тактовый сигнал для UART-приемника;
  - uartDataIn — шина входящих данных, имеет ширину девять бит;
  - uartDataReceived — индикация приема нового символа;
  - uartReceiveReq — подтверждение приема символа;
  - parityError — индикация ошибки четности;
  - overflow — индикация переполнения внутреннего буфера UART-приемника;
  - silence — индикация отсутствия данных в течении времени, соответствующего трем с половиной символам.

  Линии этой группы могут быть напрямую подключены к соответствующим линиям UART-приемника (см. стр. 27).
- Шина передачи данных через FIFO-буфер:
  - fifoClk — тактовый сигнал для FIFO-буфера;
  - full — индикация заполненности FIFO-буфера;
  - fifoWriteReq — запрос на запись данных в FIFO-буфер;
  - fifoWriteAck — подтверждение записи данных в FIFO-буфер;
  - fifoDataOut — шина исходящих данных, имеет ширину восемь бит.

Линии этой группы могут быть напрямую подключены к соответствующим линиям FIFO-буфера (см. стр. 38).

Поддерживаемые функции

Разработанным контроллером протокола Modbus поддерживаются следующие функции:

- Read Holding Registers – код функции 03 – чтение блока регистров устройства. Параметрами функции являются адрес первого регистра и количество регистров в блоке.
- Read Input Registers – код функции 04 – чтение блока входных регистров устройства. Параметрами функции являются адрес первого регистра и количество регистров в блоке.
- Write Multiple Registers – код функции 16 – запись блока регистров. Параметрами функции являются адрес первого регистра, количество регистров, в которые требуется запись, и значения, подлежащие записи в регистры.

Прием данных

Диаграмма переходов конечного автомата, отвечающего за прием данных, представлена на рисунке 4.14. Все переходы осуществляются по фронту тактового сигнала при выполнении следующих условий:

- При наличии логической единицы на входах rst или silence автомат переходит в состояние STATION_ADDRESS;

- В ином случае автомат переходит в следующее состояние только при логической единице на входе uartDataReceived.

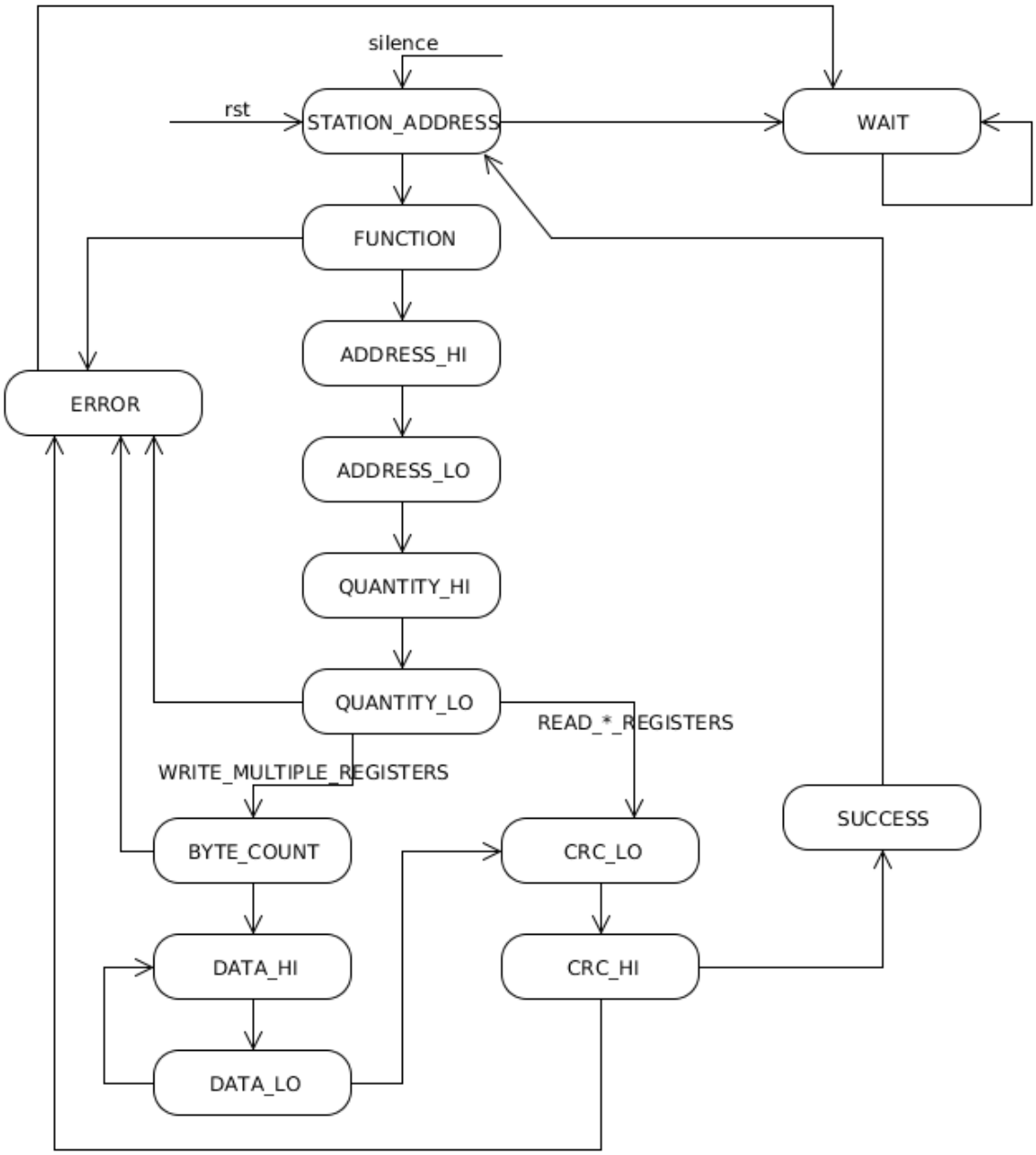

Рис 4.14: Диаграмма переходов конечного автомата, отвечающего за прием данных Modbus-контроллером

Автомат имеет следующие состояния:

- STATION_ADDRESS — прием адреса станции Modbus. Если принятый символ совпадает с адресом станции (заданным параметром MODBUS_STATION_ADDRESS), то происходит переход в состояние FUNCTION. Иначе происходит переход в состояние WAIT.
- WAIT — ожидания пакета данных. Автомат находится в этом состоянии до сброса или до появления логической единицы на входе silence (отсутствие данных на протяжении времени, соответствующего трем с половиной символам).
- FUNCTION — прием функции Modbus. Если принятый символ является кодом поддерживаемой функции (см. стр. 45), то автомат переходит в состояние ADDRESS_HI, иначе — в состояние ERROR.
- ADDRESS_HI — прием старшего байта адреса регистра. После приема старшего байта осуществляется безусловный переход в состояние ADDRESS_LO.
- ADDRESS_LO — прием младшего байта адреса регистра. После приема младшего байта осуществляется безусловный переход в состояние QUANTITY_HI.
- QUANTITY_HI — прием старшего байта количества регистров. После приема старшего байта осуществляется безусловный переход в состояние QUANTITY_LO.
- QUANTITY_LO — прием младшего байта количества регистров. После приема младшего байта осуществляется проверка корректности адреса

и количества регистров. Если что-либо из этого задано некорректно, автомат переходит в состояние ERROR, иначе в зависимости от текущей функции происходит переход в состояние BYTE_COUNT (функция 16, Write Multiple Registers) или в состояние CRC_LO (функции 03 и 04 — Read Holding Registers и Read Input Registers соответственно).

- BYTE_COUNT — прием количества байт пакета данных. После приема количества байт осуществляется безусловный переход в состояние DATA_HI.
- DATA_HI — прием старшего байта слова данных. После приема байта осуществляется безусловный переход в состояние DATA_LO.
- DATA_LO — прием младшего байта слова данных. После приема этого байта осуществляется переход в состояние CRC_LO, если это последнее слово в пакете, или в состояние DATA_HI, если слово не последнее.
- CRC_LO — прием младшего байта контрольной суммы. После приема младшего байта осуществляется безусловный переход в состояние CRC_HI.
- CRC_HI — прием старшего байта контрольной суммы. После приема старшего байта происходит проверка контрольной суммы, и если она верна, то автомат переходит в состояние SUCCESS, иначе — в состояние ERROR.
- SUCCESS — успешное завершение приема пакета. Автомат безусловно

переходит в состояние STATION_ADDRESS.

- ERROR — ошибка приема пакета. В это состояние автомат переходит в трех случаях: в случае несовпадения контрольной суммы, в случае неправильного адреса регистра и в случае неправильного количества регистров.

Связь принимающей и передающей частей модуля

Для сохранения принятых данных и связи принимающей и передающей частей модуля Modbus-контроллера были введены следующие регистры:

- modbusFunction — текущая функция.
- startAddress — принятый стартовый адрес блока регистров.
- quantity — принятое количество регистров в блоке.
- transactionBuffer — буфер для хранения данных, подлежащих записи в регистр.
- processRequest — флаг начала передачи. Защелкивается в единицу, когда конечный автомат, отвечающий за прием данных, находится в состоянии ERROR или SUCCESS. Сбрасывается в ноль, когда конечный автомат, отвечающий за прием данных, находится в состоянии WAIT.
- error — флаг наличия ошибки. Устанавливается в единицу при переходе конечного автомата, отвечающего за прием данных, в состояние ERROR.
- exceptionCode — код ошибки. Устанавливается в соответствии с ошибкой при переходе конечного автомата, отвечающего за прием

данных, в состояние ERROR.

Передача данных

Диаграмма переходов конечного автомата, отвечающего за передачу ответа, представлена на рисунке 4.15. Все переходы осуществляются по

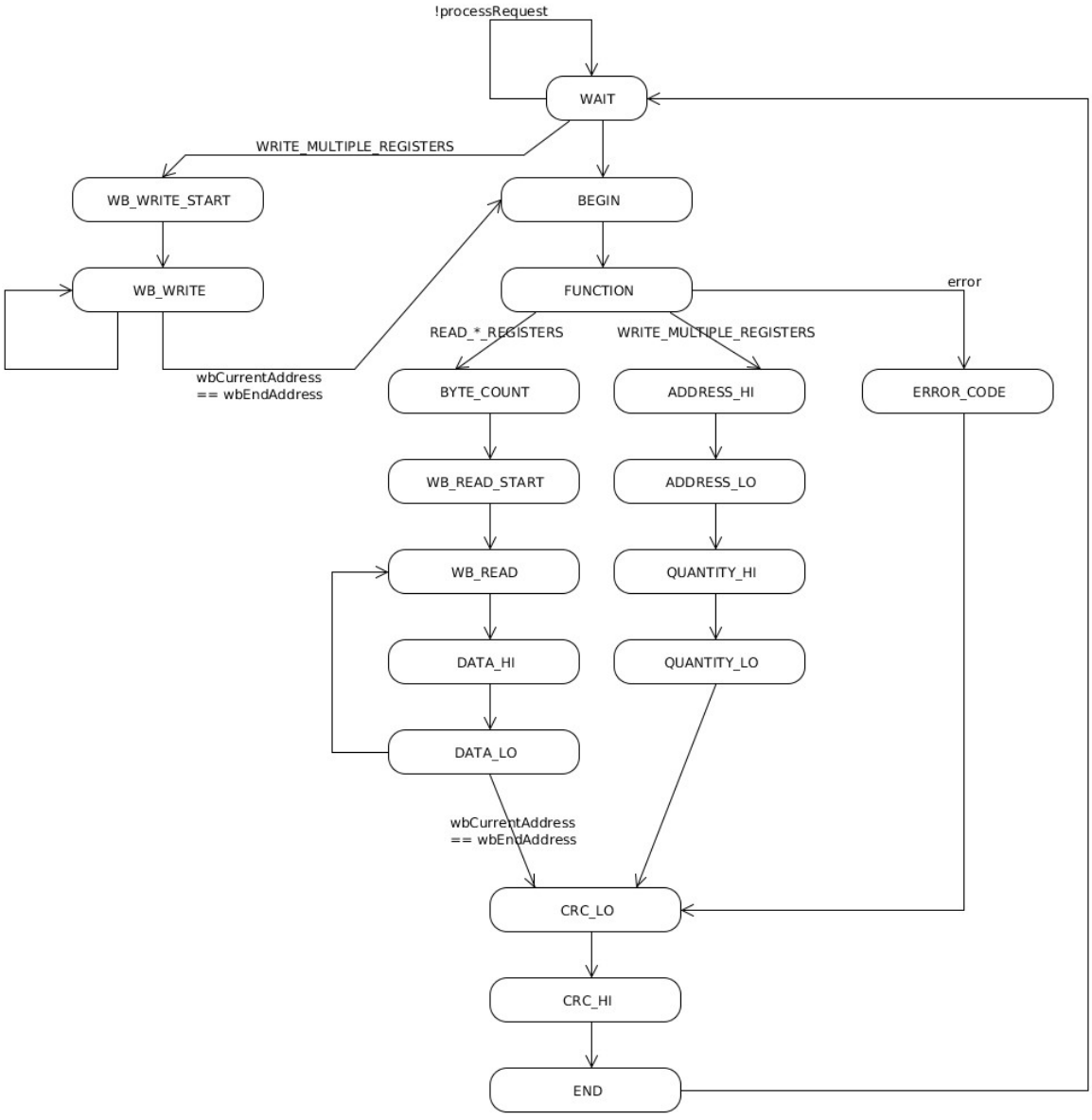

Рис 4.15: Диаграмма переходов конечного автомата, отвечающего за передачу данных Modbus-контроллером

фронту тактового сигнала.

Автомат имеет следующие состояния:

- WAIT — ожидание данных. Автомат переходит в это состояние по сигналу rst и остается в нем, пока в регистре processRequest находится ноль. При появлении единицы в регистре processRequest автомат переходит в состояние BEGIN, если текущая функция (значение в регистре modbusFunction) является функцией чтения (функции 03 и 04 — Read Holding Registers и Read Input Registers соответственно), и в состояние WB_WRITE_START — если текущая функция является функцией записи (функция 16, Write Multiple Registers).
- WB_WRITE_START — начало записи данных по шине Wishbone. В этом состоянии происходит установка регистра wbCurrentAddress в начальный адрес записи данных и указателя чтения буфера transactionBuffer в ноль. Из этого состояния происходит безусловный переход в состояние WB_WRITE.
- WB_WRITE — запись слова данных из буфера transactionBuffer на шину Wishbone. В этом состоянии инкрементируется регистр wbCurrentAddress и указатель чтения буфера, а на шине Wishbone сигналы wbCycO, wbStbO и wbWeO устанавливаются в единицу. Дальнейшие переходы происходят только при получении подтверждения окончания цикла wbAckI. При получении такого подтверждения автомат остается в текущем состоянии (и снова инкрементируются указатели), если записаны еще не все данные, и переходит в состояние BEGIN после последнего записанного слова

данных.

- BEGIN — начало передачи ответа. В этом состоянии в FIFO-буфер записывается адрес станции Modbus (заданный параметром MODBUS_STATION_ADDRESS). Далее происходит переход в состояние FUNCTION.
- FUNCTION — передача кода функции. Если ошибки не было (регистр error установлен в ноль), то в FIFO-буфер передается текущий код функции. При наличии ошибки (единица в регистре error) передается текущий код функции с установленным старшим битом. Далее автомат переходит в состояние ERROR_CODE при наличии ошибки и в состояние BYTE_COUNT (для функций чтения 03 и 04 — Read Holding Registers и Read Input Registers соответственно) или в состояние ADDRESS_HI (для функции записи 16, Write Multiple Registers).
- ADDRESS_HI — передача старшего байта стартового адреса блока регистров. В FIFO-буфер передается старший байт сохраненного (в регистре startAddress) стартового адреса блока регистров. После передачи происходит переход в состояние ADDRESS_LO.
- ADDRESS_LO — передача младшего байта стартового адреса блока регистров. В FIFO-буфер передается младший байт сохраненного (в регистре startAddress) стартового адреса блока регистров. После передачи происходит переход в состояние QUANTITY_HI.
- QUANTITY_HI — передача старшего байта количества регистров в блоке. В FIFO-буфер передается старший байт сохраненного (в

регистре quantity) количества регистров в блоке. После передачи происходит переход в состояние QUANTITY_LO.

- QUANTITY_LO — передача младшего байта количества регистров в блоке. В FIFO-буфер передается младший байт сохраненного (в регистре quantity) количества регистров в блоке. После передачи происходит переход в состояние CRC_LO.
- BYTE_COUNT — передача количества байт ответа. После передачи в FIFO-буфер количества байт происходит переход в состояние WB_READ_START.
- WB_READ_START — начало чтения данных по шине Wishbone. В этом состоянии происходит установка регистра wbCurrentAddress в начальный адрес чтения данных. Из этого состояния происходит безусловный переход в состояние WB_READ.
- WB_READ — чтение слова данных по шине Wishbone. В этом состоянии на шине Wishbone сигналы wbCycO и wbStbO устанавливаются в единицу, а wbWeO — в ноль. Дальнейшие переходы происходят только при получении подтверждения окончания цикла wbAckI. При получении такого подтверждения автомат переходит в состояние DATA_HI, а принятое слово данных защелкивается в регистр wbCurrentData.
- DATA_HI — передача старшего байта текущего слова данных. В FIFO-буфер передается старший байт регистра wbCurrentData (содержащего последнее прочитанное с шины Wishbone слово данных), далее

происходит переход в состояние DATA_LO.

- DATA_LO — передача младшего байта текущего слова данных. В FIFO-буфер передается младший байт регистра wbCurrentData (содержащего последнее прочитанное с шины Wishbone слово данных), регистр wbCurrentAddress инкрементируется, и осуществляется переход в состояние WB_READ, если были прочитаны и переданы еще не все данные, и в состояние CRC_LO, если было прочитано и передано все.
- CRC_LO — передача младшего байта контрольной суммы. Младший байт подсчитанной контрольной суммы (см. параграф "Модуль подсчета контрольной суммы" на стр. 54) передается в FIFO-буфер, а автомат переходит в состояние CRC_HI.
- CRC_HI — передача старшего байта контрольной суммы. Старший байт подсчитанной контрольной суммы (см. параграф "Модуль подсчета контрольной суммы" на стр. 54) передается в FIFO-буфер, а автомат переходит в состояние END.
- END — конец передачи. Автомат ждет подтверждения записи данных в FIFO-буфер и затем переходит в состояние WAIT.
- ERROR_CODE — передача кода ошибки. Текущий код ошибки, сохраненный в регистре exceptionCode, передается в FIFO-буфер. Автомат переходит в состояние CRC_LO.

Модуль подсчета контрольной суммы

Для подсчета контрольной суммы (CRC-16) принимаемых и передаваемых данных был использован модуль подсчета контрольной суммы

от OutputLogic. Модуль имеет интерфейс, показанный на рисунке 4.16.

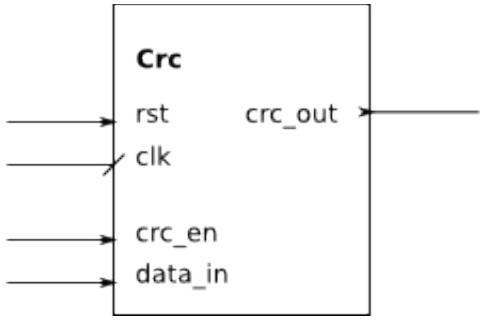

Рис 4.16: Интерфейс модуля подсчета контрольной суммы

Модуль имеет следующие сигналы:

- clk — тактовый сигнал. Подсчет контрольной суммы происходит по переднему фронту тактового сигнала.
- rst — синхронный сброс текущего результата.
- crc_en — активация подсчета контрольной суммы. Подсчет ведется только при наличии единицы на этом входе.
- data_in — входные данные. Имеют разрядность восемь бит.
- crc_out — результат контрольной суммы. Имеет разрядность 16 бит.

В модуле Modbus-контроллера создано два экземпляра модуля подсчета контрольной суммы. Один занимается подсчетом контрольной суммы принятых данных, другой — передаваемых.

Подсчет контрольной суммы принимаемых данных

На шину data_in модуля подсчета контрольной суммы принятых данных подается значение с шины uartDataIn. Сигнал rst устанавливается в единицу, когда принимающий автомат находится в состоянии STATION_ADDRESS.

Сигнал crc_en устанавливается в единицу только когда, когда uartReceiveReq установлен в единицу (происходит прием данных) и принимающий автомат находится в одном из состояний, требующих подсчет контрольной суммы. Эти состояния перечислены ниже:

- FUNCTION;
- ADDRESS_HI;
- ADDRESS_LO;
- QUANTITY_HI;
- QUANTITY_LO;
- BYTE_COUNT;
- DATA_HI;
- DATA_LO.

Временная диаграмма описанной выше логики работы показана на рисунке 4.17.

Подсчет контрольной суммы передаваемых данных

На шину data_in модуля подсчета контрольной суммы передаваемых данных подается значение с шины fifoDataOut. Сигнал rst устанавливается в единицу, когда передающий автомат находится в состоянии WAIT. Сигнал crc_en защелкивается по фронту тактового сигнала и подчиняется следующим правилам в зависимости от состояния передающего автомата:

- BEGIN — crc_en устанавливается в единицу;

- DATA_HI — в crc_en защелкивается значение сигнала wbAckI;
- WB_READ, WB_READ_START, WB_WRITE_START, WB_WRITE, CRC_LO, CRC_HI — crc_en устанавливается в ноль;
- в остальных состояниях в crc_en защелкивается значение сигнала fifoWriteAck.

Временная диаграмма, демонстрирующая описанную выше логику работы, показана на рисунке 4.18. На данном рисунке состояния C0 и C1 — это CRC_LO и CRC_HI соответственно.

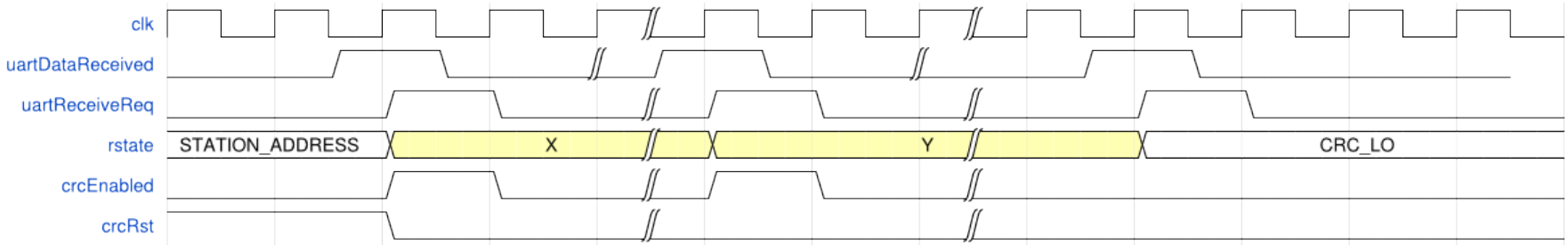

Рис 4.17: Временная диаграмма подсчета контрольной суммы принимаемых данных

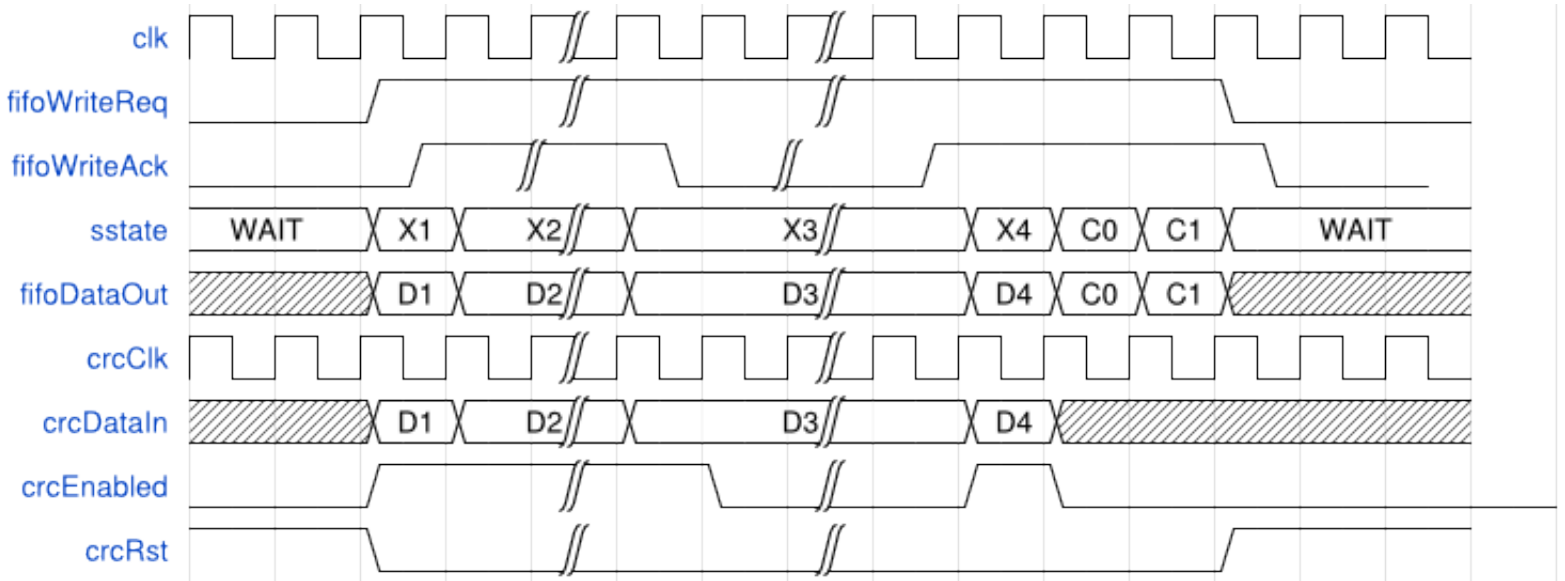

Рис 4.18: Временная диаграмма подсчета контрольной суммы передаваемых данных

4.7. Процессор

Основным вычислительным ядром построенной системы на кристалле является специфический процессор, выполняющий тестовые программы.

Требования к процессору

Процессор должен обладать следующими возможностями:

- Установка требуемых значений на линиях вывода;
- Проверку условий на линиях ввода;
- Выдержка временных задержек.

Ориентируясь на эти требования, был разработан процессор, описанный далее.

Интерфейс процессора

Модуль процессора имеет интерфейс, показанный на рис. 4.19.

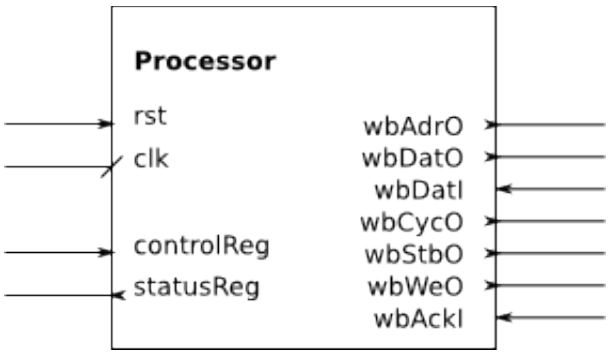

Рис 4.19: Интерфейс процессорного модуля

Модуль следующие сигналы:

- clk — системный тактовый сигнал, одновременно является тактовым

сигналом шины Wishbone.

- rst — сигнал синхронного сброса.
- wbAdrO — линия адреса Wishbone. Имеет разрядность16.
- wbDatO — линия исходящих данных Wishbone. Имеет разрядность 16.
- wbDatI — линия входящих данных Wishbone. Имеет разрядность 16.
- wbCycO — индикация валидного цикла Wishbone.
- wbStbO — индикация валидного цикла передачи данных Wishbone.
- wbAckI — сигнал подтверждения цикла Wishbone. Высокий уровень означает, что текущий цикл успешно завершился.
- wbWeO — индикация того, является ли текущий цикл Wishbone циклом чтения или циклом записи.
- controlReg — шина, значение с которой берется в качестве регистра управления (см. стр. 62).
- statusReg — значение регистра статуса процессора (см. стр. 62).

Система команд процессора

Система команд процессора представлена на таблице 7.

Команда	Слово 1	Слово 2	Слово 3	Слово 4	Слово 5
SET	0001	address	value	-	-
WAIT	0002	address	bottom	top	timeout
PAUSE	0003	timeout	-	-	-
WIN	0004	-	-	-	-
NOP	XXXX[1]	-	-	-	-

Таблица 7: Система команд процессора

Разберем подробнее каждую команду.

Команда SET служит для записи значения по заданному адресу. Первым словом команды идет код операции — 0001, далее адрес, затем значение.

Команда WAIT служит для ожидания появления значения из заданного диапазона по заданному адресу. Код операции данной команды — 0002. Параметры — адрес, диапазон значений и время, в течение которого требуется проверять значение по адресу. Как только значение по заданному адресу окажется лежащим в требуемом диапазоне, процессор переходит к исполнению следующей команды. Если же в течении времени, заданного параметром timeout, значение так и не попало в диапазон, то процессор останавливается и переходит в состояние неудачного завершения теста. Реальная задержка в тактах вычисляется как произведение операнда timeout на параметр TIMEOUT_CLOCK_DIVISOR процессорного модуля.

Команда PAUSE служит для приостановки процессора на заданное время. Единственный параметр timeout задает время задержки. Так же, как и в команде WAIT, реальная задержка в тактах вычисляется как произведение операнда timeout на параметр TIMEOUT_CLOCK_DIVISOR процессорного модуля.

1 Любой неизвестный код операции

Команда WIN служит для успешного завершения теста. После ее исполнения процессор останавливается и переходит в состояние успешного завершения теста.

Команда NOP (любой неизвестный код операции) пропускается.

Управляющий регистр

Работа процессора извне управляется значением управляющего регистра. Управляющий регистр имеет разрядность 16 бит и содержит следующие поля (таблица 8):

Бит	Назначение
0	Сброс процессора. Установка единицы в этот бит эквивалентна подачи единицы на вход rst процессорного модуля.
1-15	Зарезервировано для расширения функциональности.

Таблица 8: Управляющий регистр

Итак, чтобы перезапустить процессор после успешного или неуспешного завершения теста, нужно записать единицу, а затем ноль в младший бит управляющего регистра.

Регистр статуса

Текущее состояние процессора отражено в регистре статуса, значение которого выведено в порт statusReg. Регистр статуса имеет разрядность 16 бит и содержит следующие поля (таблица 9):

Бит	Назначение
0	Успешное завершение теста. Единица в этом бите означает, что тест успешно завершен.
1	Неуспешное завершение теста. Единица в этом бите означает, что тест завершен неуспешно.
2	Процессор работает. Единица в этом бите означает, что процессор не остановлен и тест не завершен.
3-15	Зарезервировано для расширения функциональности.

Таблица 9: Регистр статуса

Управляющий конечный автомат

Работа процессорного модуля контролируется конечным автоматом. Диаграмма переходов конечного автомата представлена на рисунке 4.20. Автомат имеет следующие состояния:

- HALT — останов процессора. Указатель инструкции сбрасывается в ноль, когда процессор находится в этом состоянии. Процессор переходит и находится в этом состоянии тогда, когда в младшем бите управляющего регистра или на входе rst находится единица. В ином случае автомат переходит в состояние FETCH_CMD_R.
- FETCH_CMD_R — запрос на чтение следующей команды. Автомат безусловно переходит в состояние FETCH_CMD_A.
- FETCH_CMD_A — чтение следующей команды. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее, в зависимости от прочитанного кода операции, автомат переходит в одно из следующих состояний:
  - FETCH_ADDR_A — для кодов операции 0001 и 0002 (команды SET

и WAIT соответственно);

- FETCH_TIME_A — для кода операции 0003 (команда PAUSE);
- SUCCESS — для кода операции 0004 (команда WIN);
- FETCH_CMD_R — для любого другого кода операции (неизвестная команда).

- FETCH_ADDR_A — чтение адреса. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее, в зависимости от прочитанного ранее кода операции, автомат переходит в одно из следующих состояний:
  - FETCH_VAL_A — для кода операции 0001 (команда SET);
  - FETCH_VAL_BOT_A — для кода операции 0002 (команда WAIT).
- FETCH_VAL_A — чтение значения для записи. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее происходит переход в состояние WRITE_REG_A.

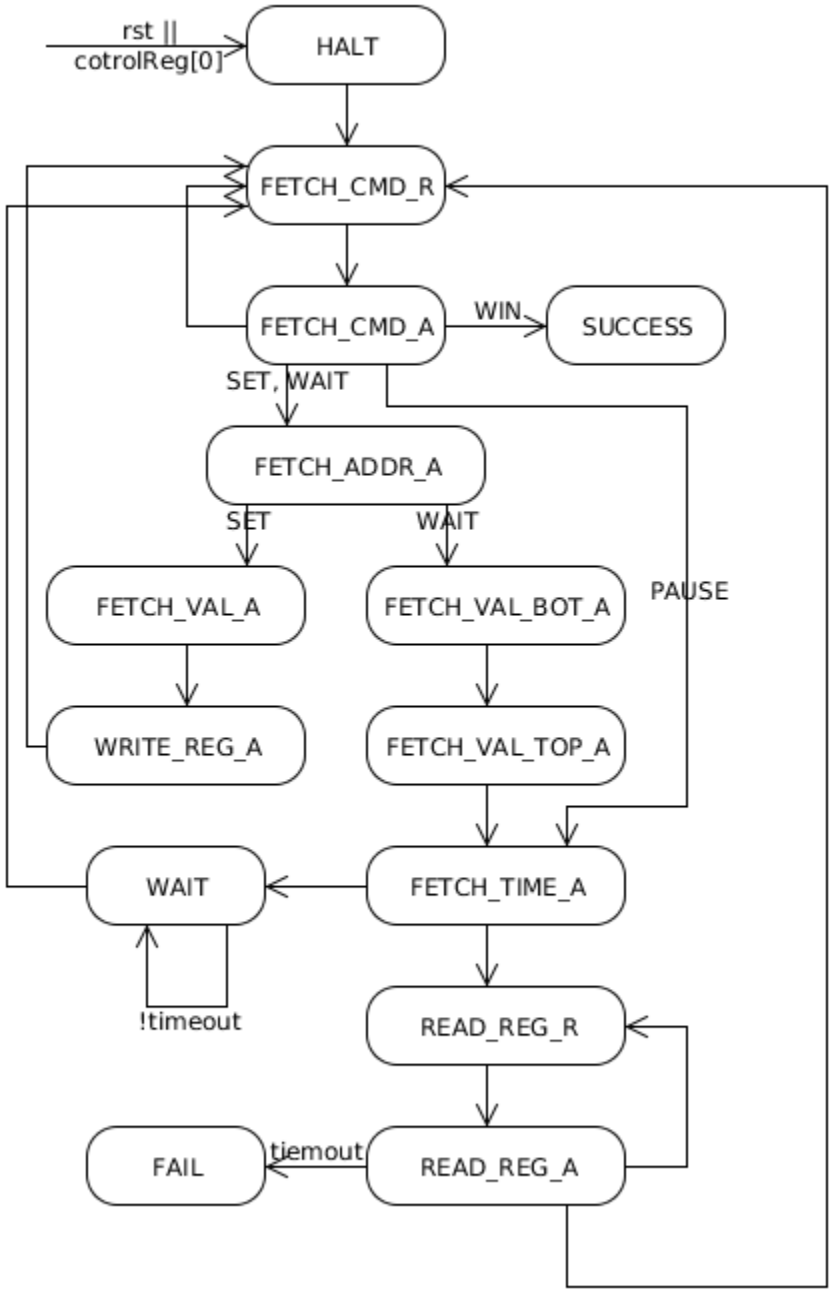

Рис 4.20: Диаграмма переходов управляющего конечного автомата процессора

- FETCH_VAL_BOT_A — чтение нижней границы проверяемого

значения. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее происходит переход в состояние FETCH_VAL_TOP_A.

- FETCH_VAL_TOP_A — чтение верхней границы проверяемого значения. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее происходит переход в состояние FETCH_TIME_A.
- FETCH_TIME_A — чтение времени задержки. На шине Wishbone инициируется цикл чтения данных. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее, в зависимости от прочитанного ранее кода операции, автомат переходит в одно из следующих состояний:
  - WAIT — для кода операции 0003 (команда PAUSE);
  - READ_REG_R — для кода операции 0002 (команда WAIT).
- WAIT — задержка. Автомат находится в этом состоянии, пока значение счетчика задержки меньше прочитанного ранее значение задержки. Далее происходит переход в состояние FETCH_CMD_R.
- WRITE_REG_A — запись значения по шине. В этом состоянии на шине Wishbone инициируется цикл записи, в котором адрес и данные — прочитанные ранее операнды команды SET. Автомат находится в этом

состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). Далее происходит переход в состояние FETCH_CMD_R.

- READ_REG_R — запрос на чтение регистра. Автомат переходит в состояние READ_REG_A.
- READ_REG_A — чтение регистра. На шине Wishbone инициируется цикл чтения данных, в котором адрес — ранее прочитанный операнд команды WAIT. Автомат находится в этом состоянии до успешного завершения цикла Wishbone (до появления единицы на линии wbAckI). После этого прочитанные данные сравниваются с прочитанным ранее диапазоном, а значение счетчика задержки — с прочитанным ранее значением задержки. В зависимости от этих условий происходит переход в следующие состояния:
  - FETCH_CMD_R — при попадании прочитанного значения в требуемый диапазон;
  - FAIL — при переполнении счетчика задержки;
  - READ_REG_R — в других случаях.
- FAIL — состояние, в которое переходит автомат при неуспешном завершении теста. Автомат находится в этом состоянии до появления единицы на входе rst или в младшем бите управляющего регистра.
- SUCCESS — состояние, в которое переходит автомат при успешном завершении теста. Автомат находится в этом состоянии до появления единицы на входе rst или в младшем бите управляющего регистра.

Состояния FETCH_CMD_R и READ_REG_R были введены для того, чтобы процессор отпускал шину между циклами чтения. В ином случае другим ведущим устройствам придется ждать освобождения шины неопределенное время.

4.8. Системный арбитр

Системный арбитр является устройством, согласующим работу нескольких ведущих устройств шины Wishbone. В качестве стратегии работы системного арбитра была выбрана циклическая стратегия, при которой доступ к шине предоставляется всем ведущим устройствам по очереди без выделения каких-либо приоритетов. Как только текущее активное устройство освобождает шину, управление передается следующему устройству, ожидающему начала цикла.

Интерфейс системного арбитра

Модуль системного арбитра имеет интерфейс, показанный на рисунке 4.21. Модуль имеет следующие параметры:

- MASTERS_WIDTH — двоичный логарифм количества подключаемых ведущих устройств;
- ADDRESS_WIDTH — ширина шин адреса;

- DATA_WIDTH — ширина шин данных.

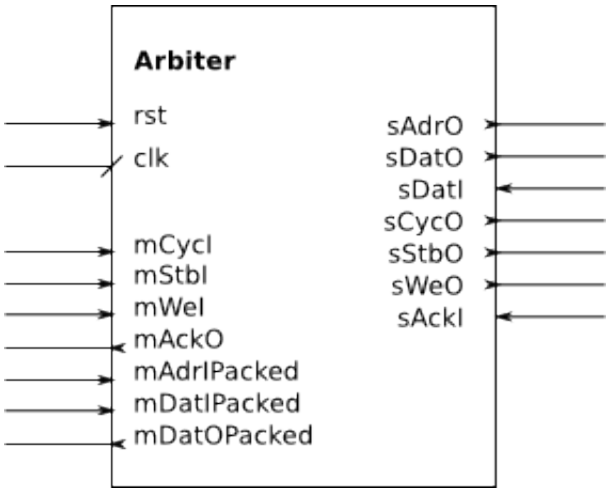

Рис 4.21: Интерфейс системного арбитра

Рассмотрим подробнее входные и выходные сигналы модуля:

- clk — системный тактовый сигнал. Является тактовым сигналом для всех шин Wishbone.
- rst — сигнал синхронного сброса.
- Группа сигналов для подключения ведущих устройств:
  - mCycI — сигналы индикации валидного цикла ведущих устройств. Шина имеет разрядность, соответствующую количеству ведущих устройств.
  - mStbI — сигналы индикации валидного цикла передачи данных ведущих устройств. Шина имеет разрядность, соответствующую количеству ведущих устройств.
  - mWeI — сигналы выбора запись/чтение ведущих устройств. Шина имеет разрядность, соответствующую количеству ведущих

устройств.

- mAckI — сигналы завершения цикла, подключаемые к ведущим устройствам. Шина имеет разрядность, соответствующую количеству ведущих устройств.
- mAdrIPacked — шины адреса ведущих устройств. Все шины, имеющие разрядность, задаваемую параметром ADDRESS_WIDTH, упакованы в одну шину. Получившаяся разрядность шины mAdrIPacked равна ADDRESS_WIDTH * (2^MASTERS_WIDTH). Пример подключения адресных шин показан на рисунке 4.22.

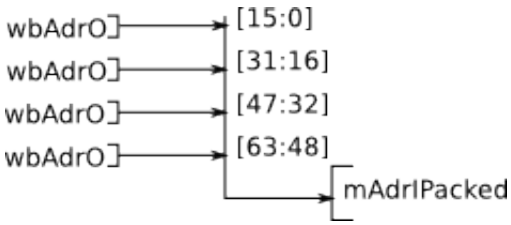

Рис 4.22: Упаковка шин на примере шины адреса. В данном примере ADDRESS_WIDTH=16, MASTERS_WIDTH=2

- mDatIPacked — исходящие шины данных ведущих устройств. Упакованы аналогично mAdrIPacked.
- mDatOPacked — входящие шины данных ведущих устройств. Упакованы аналогично mAdrIPacked и mDatIPacked.

- Группа сигналов для подключения ведомых устройств:
  - sAdrO — линия адреса Wishbone. Имеет разрядность, заданную параметром ADDRESS_WIDTH.
  - sDatO — линия исходящих данных Wishbone. Имеет разрядность,

заданную параметром DATA_WIDTH.

- sDatI — линия входящих данных Wishbone. Имеет разрядность, заданную параметром DATA_WIDTH.
- sCycO — индикация валидного цикла Wishbone.
- sStbO — индикация валидного цикла передачи данных Wishbone.
- sAckI — сигнал подтверждения цикла Wishbone. Высокий уровень означает, что текущий цикл успешно завершился.
- sWeO — индикация того, является ли текущий цикл Wishbone циклом чтения или циклом записи.

Управляющий конечный автомат

Работа системного арбитра управляется конечным автоматом. Диаграмма переходов этого автомата показана на рисунке 4.23.

Автомат имеет три состояния. Переходы между состояниями происходят по спаду тактового сигнала. Рассмотрим подробнее каждое состояние:

- RESET — состояние сброса. Автомат переходит в него при наличии единицы на входе rst и остается в нем до появления нуля на rst. В этом состоянии происходит сброс всех регистров (счетчика текущего ведущего устройства, защелкнутых входных сигналов). При появлении нуля на входе rst автомат переходит в состояние NEXT.

- NEXT — выбор следующего ведущего устройства. В этом состоянии инкрементируется счетчик текущего ведущего устройства до тех пор, пока на соответствующем входе mCycI не обнаружится единица (сигнал о начале цикла). После этого автомат переходит в состояние CYCLE.

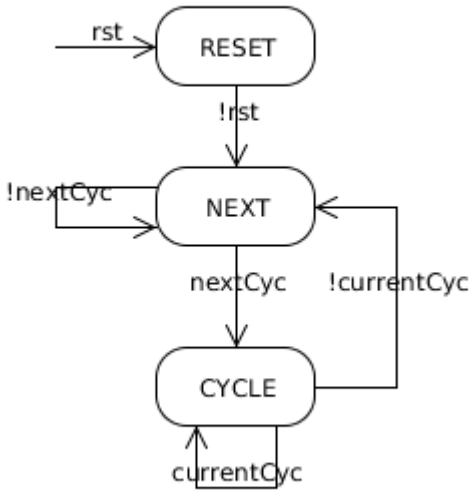

Рис 4.23: Диаграмма переходов конечного автомата, управляющего системным арбитром

- CYCLE — активный цикл. В этом состоянии на выходы sCycO, sStbO, sWeO, sAdrO и sDatO подаются сигналы с текущего активного ведущего устройства. Автомат находится в этом состоянии до тех пор, пока активное ведущее устройство держит шину, то есть подает логическую единицу на соответствующую линию mCycI. При невыполнении этого условия автомат переходит в состояние NEXT.

4.9. Блок памяти

Блок памяти представляет собой простой массив регистров. Данный модуль является ведомым устройством Wishbone.

4.10. Блок ввода-вывода

Разработанный модуль ввода-вывода представляет собой двунаправленный порт и является ведомым устройством шины Wishbone. Шина адреса Wishbone имеет разрядность один бит. Адрес 0 соответствует адресу чтения и записи данных в порт, адрес 1 — адресу записи режима работы порта. Каждый бит режима работы означает режим работы соответствующего бита порта. Если бит установлен в ноль, то данная линия работает на чтение, если в единицу — то на запись. Интерфейс блока ввода-вывода представлен на рисунке 4.24. Разрядность порта задается параметром DATA_WIDTH.

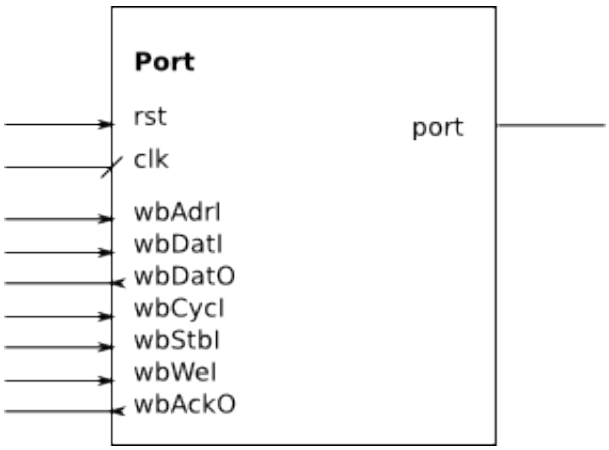

Рис 4.24: Интерфейс блока ввода-вывода

4.11. Организация адресного пространства

Так как шина адреса имеет разрядность 16 бит, для адресации доступно 65536 слов. Адресное пространство организовано следующим образом

(таблица 10):

Диапазон адресов	Назначение
0x0000-0x3FFF	Порты ввода-вывода
0x4000-0x7FFF	Регистры для хранения данных
0x8000-0xFFFF	Память программ

Таблица 10: Организация адресного пространства

Так как разъем периферийного модуля имеет 24 линии ввода-вывода, а разрядность шины данных составляет 16 бит, из диапазона портов ввода-вывода используется только шесть первых адресов, по два порта на каждый периферийный модуль.

ГЛАВА 5. ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Одной из задач проекта была разработка технических требований на разработку программного обеспечения.

5.1. Загрузчик конфигурации ПЛИС

Требуется разработать загрузчик конфигурации ПЛИС, работающий на компьютере BeagleBoard и загружающий конфигурацию ПЛИС по интерфейсу SPI разъема расширения.

Загрузчик должен принимать параметром командной строки имя файла конфигурации.

5.2. Компилятор тестовых программ

Требуется разработать на языке Ruby прослойку (модуль), выполняющий трансляцию тестовых программ на языке Cucumber в программы в машинных кодах специального процессора системы на кристалле.

Имя выходного файла должно являться параметром модуля.

5.3. Графический интерфейс пользователя

Требуется разработать графический интерфейс пользователя тестирующей системы, работающий на компьютере BeagleBoard. Интерфейс должен предоставлять следующие возможности:

- Хранение набора тестовых программ для каждого варианта

тестируемого оборудования.

- Вызов компилятора тестовых программ для преобразования их в инструкции процессора системы на кристалле.
- Загрузка откомпилированных тестовых программ на исполнительный модуль по протоколу Modbus, исполнение их и вывод результата.
- Хранение отчетов о тестировании.
- Ручное тестирование путем подачи произвольных импульсов и наблюдения произвольных сигналов.

Программа графического интерфейса пользователя является связующим звеном программного комплекса компьютерного модуля. Остальные компоненты комплекса (загрузчик конфигурации ПЛИС, компилятор тестовых программ) должны разрабатываться так, чтобы обеспечить их сопряжение с программой графического интерфейса.

ЗАКЛЮЧЕНИЕ

В результате данной работы была построена законченная система, позволяющая производить полуавтоматическое и автоматическое тестирование электронной аппаратуры. Для достижения этой цели были решены следующие задачи:

- Проанализированы современные языки и методы тестирования аппаратуры и программного обеспечения, выбран наиболее подходящий язык;
- Разработана архитектура тестирующей системы;
- Составлены технические задания на изготовление модулей системы;
- Разработана система на кристалле, загружаемая на исполнительный модуль;
- Составлены технические задания на разработку программного обеспечения интерфейса пользователя, компилятора тестовых программ и загрузчика системы на кристалле.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. THE ETSI TEST DESCRIPTION LANGUAGE (TDL). Results from the ETSI project STF 454. / A. Ulrich, G. Adamis, F. Kristoffersen, Ph. Makedonski, M.-F. Wendland, A. Wiles. – Режим доступа: http://ucaat.etsi.org/2014/presentations/Intro%20to%20ETSI%20TDL_Andreas%20Ulrich.pdf, свободный.
2. Liu, Chang. TestTalk: A Comprehensive Testing Language / Chang Liu. — Information & Computer Science University of California, Irvine — 7 p.
3. TestTalk, A Test Description Language: Write Once, Test by Anyone, Anytime, Anywhere, with Anything / Chang Liu , Debra J. Richardson.
4. Cucumber — making BDD fun [Электронный ресурс]. – Режим доступа: http://cukes.info/, свободный. – Загл. с экрана.
5. Ruby programming language [Электронный ресурс]. – Режим доступа: https://www.ruby-lang.org/, свободный. – Загл. с экрана.
6. A framework for specification-based testing / Stocks, Phil; Carrington, David. – IEEE Transactions on Software Engineering, 1996
7. Abrial, Jean-Raymond. Data Semantics / Jean-Raymond Abrial.
8. A Specification Language / Abrial, Jean-Raymond; Schuman, Stephen A; Meyer, Bertrand. – Cambridge University Press, 1980
9. BeagleBoard.org — BeagleBoard-xM [Электронный ресурс]. Режим доступа: http://beagleboard.org/Products/BeagleBoard-xM, свободный. – Загл. с экрана.

10. Modbus Specifications and Implementation Guides [Электронный ресурс]. Режим доступа: http://www.modbus.org/specs.php, свободный. – Загл. с экрана.

11. WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores [Электронный ресурс]. Режим доступа: http://cdn.opencores.org/downloads/wbspec_b3.pdf, свободный. – Загл. с экрана.

ПРИЛОЖЕНИЯ

Исходный код UART-приемника

```
module UartReceiver(
    input clk,
    input rst,
    input rx,
    input [1:0] dataBits, // data bits count = dataBits + 5
    input hasParity,
    input [1:0] parityMode, // 00 - space, 11 - mark, 10 - even, 01 - odd
    input extraStopBit,
     input [CLOCK_DIVISOR_WIDTH-1:0] clockDivisor, // f_uart = f_clk / (2 *
clockDivisor + 2)

    output reg [8:0] dataOut,
    output reg dataReceived,
    output reg parityError,
    output reg overflow,
    output reg break,
    output reg silence, // 3 or more characters of silence
    input receiveReq
);
    parameter CLOCK_DIVISOR_WIDTH=24;
    localparam STATE_IDLE = 3'd0;
    localparam STATE_START = 3'd1;
    localparam STATE_DATA = 3'd2;
    localparam STATE_STOP = 3'd3;

    reg [2:0] state = STATE_IDLE;

    reg [1:0] latchedDataBits = 0;
    reg latchedHasParity = 0;
    reg [1:0] latchedParityMode = 0;
    reg latchedExtraStopBit = 0;
    reg [CLOCK_DIVISOR_WIDTH-1:0] latchedClockDivisor = 0;

    /* Slow clock begin */
    reg [CLOCK_DIVISOR_WIDTH-1:0] clockCounter = 0;
    wire uartClkEnabled = state != STATE_IDLE;
    wire uartClk = clockCounter == latchedClockDivisor && uartClkEnabled;
    always @(negedge clk) begin
        if(rst) begin
            clockCounter <= 0;
        end else begin
            if(state == STATE_IDLE) begin
                clockCounter <= 0;
            end else begin
                if(!uartClkEnabled) begin
                    clockCounter <= 0;
```

```
                end else begin
                    if(clockCounter != latchedClockDivisor << 1)
                        clockCounter <= clockCounter + 1;
                    else begin
                        clockCounter <= 0;
                    end
                end
            end
        end
    end
    /* Slow clock end */

    /* Silence counter begin */
    reg [CLOCK_DIVISOR_WIDTH-1:0] silenceClockCounter = 0;
    wire silenceClk = silenceClockCounter >= clockDivisor << 1;
    reg [7:0] silenceCharsCounter = 0;
    initial begin
        silence = 1'b0;
    end
    always @(negedge clk) begin
        if(rst) begin
            silenceClockCounter <= 0;
        end else begin
            if(silenceClockCounter >= clockDivisor << 1) begin
                silenceClockCounter <= 0;
            end else begin
                silenceClockCounter <= silenceClockCounter + 1;
            end
        end
    end
    always @(posedge clk) begin
        if(rst || ~rx || state != STATE_IDLE) begin
            silenceCharsCounter <= 0;
            silence <= 1'b0;
        end else begin
            if(silenceClk) begin
                if(silenceCharsCounter == 33) begin
                    silence <= 1'b1;
                end else begin
                    silenceCharsCounter <= silenceCharsCounter + 1;
                    silence <= 1'b0;
                end
            end
        end
    end
    /* Silence counter end */

     wire [3:0] totalDataBits = 4'd5 + latchedDataBits + latchedHasParity -
4'd1;
    reg [3:0] dataCounter = 4'd0;
    reg [8:0] currentData = 9'd0;

    reg firstStopBitReceived = 1'b0;
```

```
    wire currentParityError;

    always @(posedge clk) begin
        if(rst) begin
            dataOut <= 9'b0;
            dataReceived <= 1'b0;
            parityError <= 1'b0;
            overflow <= 1'b0;
            break <= 1'b0;
            state <= STATE_IDLE;
        end else begin
            if(uartClk) begin
                case(state)
                    STATE_START:
                        if(rx)
                            state <= STATE_IDLE;
                        else begin
                            state <= STATE_DATA;
                            dataCounter <= 0;
                        end
                    STATE_DATA: begin
                            currentData[dataCounter] <= rx;
                            $display("@%t got bit %b", $time, rx);
                            if(dataCounter < totalDataBits) begin
                                dataCounter <= dataCounter + 4'd1;
                            end else begin
                                state <= STATE_STOP;
                                firstStopBitReceived <= 1'b0;
                            end
                        end
                    STATE_STOP:
                        if(rx) begin
                             if(!latchedExtraStopBit || firstStopBitReceived)
begin
                                state <= STATE_IDLE;
                                if(!receiveReq) begin
                                    if(dataReceived)
                                        overflow <= 1'b1;
                                    dataReceived <= 1'b1;
                                end
                                dataOut <= currentData;
                                parityError <= currentParityError;
                                break <= 1'b0;
                            end
                            firstStopBitReceived <= 1'b1;
                        end else begin
                            break <= 1'b1;
                            state <= STATE_IDLE;
                        end
                endcase
            end // uartClk
            if(state == STATE_IDLE) begin
                if(rx == 1'b0) begin
```

```
                    state <= STATE_START;
                    latchedDataBits <= dataBits;
                    latchedHasParity <= hasParity;
                    latchedParityMode <= parityMode;
                    latchedExtraStopBit <= extraStopBit;
                    latchedClockDivisor <= clockDivisor;
                end
            end
            if(receiveReq) begin
                dataReceived <= 1'b0;
                overflow <= 1'b0;
                break <= 1'b0;
            end
        end // rst
    end // always

    ParityChecker parityChecker(
        .data(currentData),
        .dataBits(latchedDataBits),
        .hasParity(latchedHasParity),
        .parityMode(latchedParityMode),
        .parityError(currentParityError)
    );
endmodule

module ParityChecker(
    input [8:0] data,
    input [1:0] dataBits, // data bits count = dataBits + 5
    input hasParity,
    input [1:0] parityMode,

    output reg parityError
);
    localparam PARITY_SPACE = 2'b00;
    localparam PARITY_ODD = 2'b01;
    localparam PARITY_EVEN = 2'b10;
    localparam PARITY_MARK = 2'b11;
    wire [8:0] dataMask = ~(~9'h0 << 5 + dataBits + hasParity);
    always @(data, dataBits, hasParity, parityMode, dataMask) begin
        if(!hasParity)
            parityError = 1'b0;
        else begin
            case(parityMode)
                PARITY_SPACE: parityError = data[5 + dataBits] == 1'b1;
                PARITY_MARK: parityError = data[5 + dataBits] == 1'b0;
                PARITY_EVEN: parityError = ^(data & dataMask);
                PARITY_ODD: parityError = ~^(data & dataMask);
            endcase
        end
    end
endmodule
```

Исходный код UART-передатчика

```
module UartTransmitter(
    input clk,
    input rst,
    output reg tx,
    input [1:0] dataBits, // data bits count = dataBits + 5
    input hasParity,
    input [1:0] parityMode, // 00 - space, 11 - mark, 10 - even, 01 - odd
    input extraStopBit,
     input [CLOCK_DIVISOR_WIDTH-1:0] clockDivisor,// f_uart = f_clk / (2 *
clockDivisor + 2)

    output ready,
    input [7:0] data,
    input transmitReq
);
    parameter CLOCK_DIVISOR_WIDTH=24;
    localparam STATE_IDLE = 3'd0;
    localparam STATE_START = 3'd1;
    localparam STATE_DATA = 3'd2;
    localparam STATE_PAR = 3'd3;
    localparam STATE_STOP = 3'd4;
    localparam STATE_END = 3'd5;

    initial tx = 1'b1;
    assign ready = state == STATE_IDLE;

    reg [2:0] state = STATE_IDLE;

    reg [CLOCK_DIVISOR_WIDTH-1:0] clockCounter = 0;
    wire uartClkEnabled = state != STATE_IDLE;
        wire uartClk = (clockCounter == latchedClockDivisor << 1) &&
uartClkEnabled;
    always @(negedge clk) begin
        if(rst) begin
            clockCounter <= 0;
        end else begin
            if(state == STATE_IDLE) begin
                clockCounter <= 0;
            end else begin
                if(!uartClkEnabled) begin
                    clockCounter <= 0;
                end else begin
                    if(clockCounter != latchedClockDivisor << 1)
                        clockCounter <= clockCounter + 1;
                    else begin
                        clockCounter <= 0;
                    end
                end
            end
        end
    end
```

```
wire parity;
UartTxParity uartTxParity(
    .data(latchedData),
    .dataBits(latchedDataBits),
    .parityMode(latchedParityMode),
    .parity(parity)
);
reg [7:0] latchedData = 8'd0;
reg [1:0] latchedDataBits = 2'd0;
reg latchedHasParity = 1'b0;
reg [1:0] latchedParityMode = 2'd0;
reg latchedExtraStopBit = 1'b0;
reg [CLOCK_DIVISOR_WIDTH-1:0] latchedClockDivisor = 0;

reg firstStopBitTransmitted = 1'b0;
reg [2:0] dataBitsRemaining = 3'd0;
always @(posedge clk) begin
    if(rst) begin
        tx <= 1'b1;
        state <= STATE_IDLE;
        latchedData <= 8'd0;
    end else begin
        if(state == STATE_IDLE && transmitReq) begin
            state <= STATE_START;
            latchedData <= data;
            latchedDataBits <= dataBits;
            latchedHasParity <= hasParity;
            latchedParityMode <= parityMode;
            latchedExtraStopBit <= extraStopBit;
            latchedClockDivisor <= clockDivisor;
        end
        if(uartClk) begin
            case(state)
                STATE_IDLE: tx <= 1'b1;
                STATE_START: begin
                    tx <= 1'b0;
                    state <= STATE_DATA;
                    dataBitsRemaining <= latchedDataBits + 3'd4;
                end
                STATE_DATA: begin
                    tx <= latchedData[0];
                    latchedData <= latchedData >> 1;
                    dataBitsRemaining <= dataBitsRemaining - 1;
                    if(dataBitsRemaining == 3'd0) begin
                        if(latchedHasParity)
                            state <= STATE_PAR;
                        else
                            state <= STATE_STOP;
                    end
                end
                STATE_PAR: begin
                    tx <= parity;
```

```
                        state <= STATE_STOP;
                    end
                    STATE_STOP: begin
                        tx <= 1'b1;
                        firstStopBitTransmitted <= 1'b1;
                        if(firstStopBitTransmitted || !latchedExtraStopBit)
                            state <= STATE_END;
                    end
                    STATE_END: begin
                        tx <= 1'b1;
                        state <= STATE_IDLE;
                    end
                    default: state <= STATE_IDLE;
                endcase
            end // uartClk
        end // rst
    end
endmodule

module UartTxParity(
    input [7:0] data,
    input [1:0] dataBits,
    input [1:0] parityMode,
    output reg parity
);
    wire [7:0] mask = ~((~8'h00) << (dataBits + 5));
    always @(*) begin
        case(parityMode)
            2'b00: parity = 1'b0; // space
            2'b01: parity = ~^(data & mask); // odd
            2'b10: parity = ^(data & mask); // even
            2'b11: parity = 1'b1; // mark
        endcase
    end
endmodule
```

Исходный код FIFO-буфера

```
module Fifo(
    input clk,
    input rst,

    output empty,
    output full,
    input readReq,
    output reg readAck,
    input writeReq,
    output reg writeAck,
    input [DATA_WIDTH-1:0] dataIn,
    output reg [DATA_WIDTH-1:0] dataOut
);
```

```
    parameter DATA_WIDTH = 16;
    parameter FIFO_LOG_LENGTH = 4;
    localparam FIFO_LENGTH = 1 << FIFO_LOG_LENGTH;

    reg [DATA_WIDTH-1:0] buffer[FIFO_LENGTH-1:0];
    reg [FIFO_LOG_LENGTH-1:0] getPtr = 0;
    reg [FIFO_LOG_LENGTH-1:0] putPtr = 0;
    initial dataOut = 0;
    assign empty = getPtr == putPtr;
    assign full = (putPtr + 1) == getPtr;
    initial readAck = 1'b0;
    initial writeAck = 1'b0;

    always @(posedge clk) begin
        if(rst) begin
            getPtr <= 0;
            putPtr <= 0;
        end else begin
            if(readReq) begin
                if(!empty) begin
                    getPtr <= getPtr + 1;
                    readAck <= 1'b1;
                    dataOut <= buffer[getPtr];
                end else
                    readAck <= 1'b0;
            end else begin
                readAck <= 1'b0;
            end
            if(writeReq) begin
                if(!full) begin
                    buffer[putPtr] <= dataIn;
                    putPtr <= putPtr + 1;
                    writeAck <= 1'b1;
                end else begin
                    writeAck <= 1'b0;
                end
            end else begin
                writeAck <= 1'b0;
            end
        end
    end
endmodule
```

Исходный код контроллера Modbus

```
/*
*/
module ModbusToWishbone(
    input clk,
    input rst,
    // Wishbone
```

```
    output [ADDRESS_WIDTH-1:0] wbAdrO,
    output [DATA_WIDTH-1:0] wbDatO,
    input [DATA_WIDTH-1:0] wbDatI,
    output reg wbCycO,
    output reg wbStbO,
    input wbAckI,
    output reg wbWeO,

    // Input UART
    output uartClk,
    input [8:0] uartDataIn,
    input uartDataReceived,
    input parityError,
    input overflow,
    input silence,
    output reg uartReceiveReq,

    // Output FIFO
    output fifoClk,
    input full,
    output reg fifoWriteReq,
    input fifoWriteAck,
    output reg [7:0] fifoDataOut
);
    parameter ADDRESS_WIDTH = 24;
    parameter DATA_WIDTH = 16;

    parameter MODBUS_STATION_ADDRESS = 8'h37;
    parameter OFFSET_INPUT_REGISTERS = 'hA00000;
    parameter QUANTITY_INPUT_REGISTERS = 'hffff;
    parameter OFFSET_HOLDING_REGISTERS = 'hA00000;
    parameter QUANTITY_HOLDING_REGISTERS = 'hffff;
    parameter OFFSET_FILES = 'hB00000;

    localparam FUN_READ_COILS = 8'h01;
    localparam FUN_READ_DISCRETE_INPUTS = 8'h02;
    localparam FUN_READ_HOLDING_REGISTERS = 8'h03;
    localparam FUN_READ_INPUT_REGISTERS = 8'h04;
    localparam FUN_WRITE_SINGLE_REGISTER = 8'h06;
    localparam FUN_WRITE_MULTIPLE_REGISTERS = 8'h10;
    localparam FUN_READ_FILE_RECORD = 8'h14;
    localparam FUN_WRITE_FILE_RECORD = 8'h15;

    assign uartClk = ~clk;
    assign fifoClk = ~clk;

    /* Receive begin */
    localparam RSTATE_STATION_ADDRESS = 'h0;
    localparam RSTATE_WAIT = 'h1;
    localparam RSTATE_FUNCTION = 'h2;
    localparam RSTATE_CRC_LO = 'h3;
    localparam RSTATE_CRC_HI = 'h4;
    localparam RSTATE_ADDRESS_LO = 'h5;
```

```
    localparam RSTATE_ADDRESS_HI = 'h6;
    localparam RSTATE_QUANTITY_LO = 'h7;
    localparam RSTATE_QUANTITY_HI = 'h8;
    localparam RSTATE_DATA_LO = 'h9;
    localparam RSTATE_DATA_HI = 'hA;
    localparam RSTATE_BYTE_COUNT = 'hB;
    localparam RSTATE_ERROR = 'hC;
    localparam RSTATE_SUCCESS = 'hD;

    reg isAddressValid;
    reg isTempQuantityValid;
    reg isQuantityValid;
    reg isTempByteCountValid;
    wire [15:0] tempQuantity = {quantityHi, uartDataIn[7:0]};
    wire [7:0] tempFunction = uartDataIn[7:0];
    wire [7:0] tempByteCount = uartDataIn[7:0];
    reg [7:0] modbusFunction = 8'd0;
    always @(*) begin
        case(modbusFunction)
            FUN_READ_COILS,
            FUN_READ_DISCRETE_INPUTS: begin
                        isTempQuantityValid = tempQuantity <= 16'h07d0 &&
tempQuantity >= 16'h0001;
                    isQuantityValid = quantity <= 16'h07d0 && quantity >=
16'h0001;
            end
            FUN_READ_HOLDING_REGISTERS,
            FUN_READ_INPUT_REGISTERS: begin
                        isTempQuantityValid = tempQuantity <= 16'h007d &&
tempQuantity >= 16'h0001;
                    isQuantityValid = quantity <= 16'h007d && quantity >=
16'h0001;
            end
            FUN_WRITE_MULTIPLE_REGISTERS: begin
                        isTempQuantityValid = tempQuantity <= 16'h007b &&
tempQuantity >= 16'h0001;
                    isQuantityValid = quantity <= 16'h007b && quantity >=
16'h0001;
            end

            default: begin
                isTempQuantityValid = 1'b0;
                isQuantityValid = 1'b0;
            end
        endcase

        case(modbusFunction)
            FUN_READ_COILS,
            FUN_READ_DISCRETE_INPUTS: begin
                isAddressValid = 1'b0;
            end
            FUN_READ_HOLDING_REGISTERS: begin
                        isAddressValid = startAddress + tempQuantity <=
```

```
QUANTITY_HOLDING_REGISTERS;
            end
            FUN_READ_INPUT_REGISTERS: begin
                            isAddressValid = startAddress + tempQuantity <=
QUANTITY_INPUT_REGISTERS;
            end
            FUN_WRITE_MULTIPLE_REGISTERS: begin
                            isAddressValid = startAddress + tempQuantity <=
QUANTITY_HOLDING_REGISTERS;
            end
            default: begin
                isAddressValid = 1'b0;
            end
        endcase

        case(modbusFunction)
            FUN_WRITE_MULTIPLE_REGISTERS:
                     isTempByteCountValid = tempByteCount == {quantity[6:0],
1'b0};
            default:
                isTempByteCountValid = 1'b0;
        endcase
    end // always

    /* uartReceiveReq begin */
    initial uartReceiveReq = 1'b0;
    always @(posedge clk) begin
        if(rst)
            uartReceiveReq <= 1'b0;
        else begin
            if(!silence) begin
                case(rstate)
                    RSTATE_STATION_ADDRESS: begin
                        if(sstate != SSTATE_WAIT)
                            uartReceiveReq <= 1'b0;
                        else
                            uartReceiveReq <= uartDataReceived;
                    end
                    RSTATE_SUCCESS,
                    RSTATE_ERROR,
                    RSTATE_WAIT: uartReceiveReq <= 1'b0;
                    default: uartReceiveReq <= uartDataReceived;
                endcase
            end
        end
    end
    /* uartReceiveReq end */

    reg [7:0] startAddressLo = 8'h0;
    reg [7:0] startAddressHi = 8'h0;
    wire [15:0] startAddress = { startAddressHi, startAddressLo };

    reg [7:0] quantityLo = 8'h0;
```

```
    reg [7:0] quantityHi = 8'h0;
    wire [15:0] quantity = { quantityHi, quantityLo };

    reg [7:0] currentDataHi = 8'd0;
    wire [15:0] currentData = {currentDataHi, uartDataIn[7:0]};

    /* processRequest begin */
    reg processRequest = 1'b0;
    always @(posedge clk) begin
        if(rst) begin
            processRequest <= 1'b0;
        end else begin
            if(processRequest && sstate == SSTATE_WAIT) begin
                processRequest <= 1'b0;
            end
            if(~silence && uartDataReceived && ~parityError) begin
                case(nextRstate)
                    RSTATE_ERROR,
                    RSTATE_SUCCESS: begin
                        processRequest <= 1'b1;
                    end
                endcase
            end
        end
    end
    /* processRequest end */

    always @(posedge clk) begin
        if(rst) begin
            currentDataHi <= 8'd0;
            transactionBufferWritePtr <= 7'd0;
        end else begin
            if(silence) begin
            end else begin
                if(uartDataReceived) begin
                    if(!parityError) begin
                        case(rstate)
                            RSTATE_FUNCTION: begin
                                modbusFunction <= uartDataIn[7:0];
                            end
                            RSTATE_CRC_LO: begin
                                expectedCrcLo <= uartDataIn[7:0];
                            end
                            RSTATE_CRC_HI: begin
                            end
                            RSTATE_ADDRESS_LO: begin
                                startAddressLo <= uartDataIn[7:0];
                            end
                            RSTATE_ADDRESS_HI: begin
                                startAddressHi <= uartDataIn[7:0];
                            end
                            RSTATE_QUANTITY_LO: begin
                                quantityLo <= uartDataIn[7:0];
```

```
                                case(modbusFunction)
                                    FUN_READ_HOLDING_REGISTERS,
                                    FUN_READ_INPUT_REGISTERS: begin
                                    end
                                    FUN_WRITE_MULTIPLE_REGISTERS: begin
                                        transactionBufferWritePtr <= 7'd0;
                                    end
                                    default: begin
                                    end
                                endcase
                            end
                            RSTATE_QUANTITY_HI: begin
                                quantityHi <= uartDataIn[7:0];
                            end
                            RSTATE_BYTE_COUNT: begin
                            end
                            RSTATE_DATA_HI: begin
                                currentDataHi <= uartDataIn[7:0];
                            end
                            RSTATE_DATA_LO: begin
                                 transactionBuffer[transactionBufferWritePtr]
<= currentData;
                                                        transactionBufferWritePtr  <=
transactionBufferWritePtr + 7'd1;
                            end
                        endcase
                    end // parityError
                end // uartDataReceived
            end // silence
        end
    end

    /* rstate begin */
    reg [7:0] rstate = RSTATE_STATION_ADDRESS;
    always @(posedge clk) begin
        rstate <= nextRstate;
    end
    /* rstate end */

    /* error begin */
    reg error = 1'b0;
    reg [7:0] exceptionCode = 8'h0;
    always @(posedge clk) begin
        if(rst) begin
            error <= 1'b0;
            exceptionCode <= 8'h0;
        end else begin
            error <= asyncError;
            exceptionCode <= asyncExceptionCode;
        end
    end
    /* error end */
```

```
    /* nextRstate begin */
    reg [7:0] nextRstate;
    reg asyncError;
    reg [7:0] asyncExceptionCode;
    always @(*) begin
        nextRstate = rstate;
        asyncError = error; // TODO сбрасывать в ноль
        asyncExceptionCode = exceptionCode;
        if(rst) begin
            nextRstate = RSTATE_STATION_ADDRESS;
        end else begin
            if(silence) begin
                nextRstate = RSTATE_STATION_ADDRESS;
            end else begin
                if(uartDataReceived) begin
                    if(parityError)
                        nextRstate = RSTATE_WAIT;
                    else begin
                        case(rstate)
                            RSTATE_STATION_ADDRESS: begin
                                if(sstate == SSTATE_WAIT) begin
                                                                if(uartDataIn[7:0]  ==
MODBUS_STATION_ADDRESS) begin
                                        nextRstate = RSTATE_FUNCTION;
                                    end else begin
                                        nextRstate = RSTATE_WAIT;
                                    end
                                end
                            end
                            RSTATE_FUNCTION: begin
                                case(tempFunction)
                                    FUN_READ_HOLDING_REGISTERS,
                                    FUN_READ_INPUT_REGISTERS,
                                    FUN_WRITE_MULTIPLE_REGISTERS: begin
                                        nextRstate = RSTATE_ADDRESS_HI;
                                    end
                                    default: begin
                                        nextRstate = RSTATE_ERROR;
                                        asyncError = 1'b1;
                                        asyncExceptionCode = 8'h1;
                                    end
                                endcase
                            end
                            RSTATE_CRC_LO: begin
                                nextRstate = RSTATE_CRC_HI;
                            end
                            RSTATE_CRC_HI: begin
                                nextRstate = RSTATE_SUCCESS;
                            end
                            RSTATE_ADDRESS_LO: begin
                                nextRstate = RSTATE_QUANTITY_HI;
                            end
                            RSTATE_ADDRESS_HI: begin
```

```
                                nextRstate = RSTATE_ADDRESS_LO;
                            end
                            RSTATE_QUANTITY_LO: begin
                                case(modbusFunction)
                                    FUN_READ_HOLDING_REGISTERS,
                                    FUN_READ_INPUT_REGISTERS: begin
                                                            if(isAddressValid  &&
isTempQuantityValid) begin
                                            nextRstate = RSTATE_CRC_LO;
                                        end else begin
                                            nextRstate = RSTATE_ERROR;
                                            asyncError = 1'b1;
                                            if(~isTempQuantityValid) begin
                                                asyncExceptionCode = 8'h03;
                                            end else begin
                                                asyncExceptionCode = 8'h02;
                                            end
                                        end
                                    end
                                    FUN_WRITE_MULTIPLE_REGISTERS: begin
                                        nextRstate = RSTATE_BYTE_COUNT;
                                    end
                                    default: begin
                                        nextRstate = RSTATE_WAIT;
                                    end
                                endcase
                            end
                            RSTATE_QUANTITY_HI: begin
                                nextRstate = RSTATE_QUANTITY_LO;
                            end
                            RSTATE_BYTE_COUNT: begin
                                case(modbusFunction)
                                    FUN_WRITE_MULTIPLE_REGISTERS: begin
                                         if(isAddressValid && isQuantityValid
&& isTempByteCountValid) begin
                                            nextRstate = RSTATE_DATA_HI;
                                        end else begin
                                            nextRstate = RSTATE_ERROR;
                                            asyncError = 1'b1;
                                                       if(~isQuantityValid ||
~isTempByteCountValid) begin
                                                asyncExceptionCode = 8'h03;
                                            end else begin
                                                asyncExceptionCode = 8'h02;
                                            end
                                        end
                                    end
                                    default: begin
                                        nextRstate = RSTATE_WAIT;
                                    end
                                endcase
                            end
                            RSTATE_DATA_HI: begin
```

```
                                nextRstate = RSTATE_DATA_LO;
                            end
                            RSTATE_DATA_LO: begin
                                if(transactionBufferWritePtr == quantity[6:0]
- 7'd1)
                                    nextRstate = RSTATE_CRC_LO;
                                else
                                    nextRstate = RSTATE_DATA_HI;
                            end
                            RSTATE_WAIT: ;
                            RSTATE_ERROR: begin
                                nextRstate = RSTATE_WAIT;
                            end
                            RSTATE_SUCCESS: begin
                                nextRstate = RSTATE_STATION_ADDRESS;
                            end
                            default: begin
                                $display("Unknown state %d", rstate);
                                nextRstate = RSTATE_WAIT;
                            end
                        endcase
                    end // parityError
                end // uartDataReceived
            end // silence
        end
    end
    /* nextRstate end */
    /* Receive end */

    /* Input CRC begin */
    wire [15:0] icrcOut;
    reg [7:0] expectedCrcLo = 8'h0;
    wire [15:0] expectedCrc = {uartDataIn[7:0], expectedCrcLo};
    wire icrcRst = rstate == RSTATE_STATION_ADDRESS || rstate == RSTATE_WAIT;
    reg icrcEnabled = 1'b0;
    reg [7:0] icrcData = 8'b0;
    Crc _crc(
        .data_in(icrcData),
        .crc_en(icrcEnabled),
        .crc_out(icrcOut),
        .rst(icrcRst),
        .clk(~clk)
    );
    always @(posedge clk) begin
        if(rst) begin
            icrcEnabled <= 1'b0;
        end else begin
            if(rstate != RSTATE_CRC_LO && rstate != RSTATE_CRC_HI) begin
                icrcEnabled <= uartDataReceived;
                icrcData <= uartDataIn[7:0];
            end else
                icrcEnabled <= 1'b0;
        end
```

```
    end
    /* Input CRC end */

    /**/
    reg [7:0] byteCount;
    always @(modbusFunction, quantity) begin
        case(modbusFunction)
            FUN_READ_HOLDING_REGISTERS,
            FUN_READ_INPUT_REGISTERS: byteCount = {quantity[6:0], 1'b0};
            default: byteCount = 8'd0;
        endcase
    end

    reg [ADDRESS_WIDTH-1:0] wbCurrentAddress = 0;
    reg [ADDRESS_WIDTH-1:0] wbStartAddress = 0;
    reg [ADDRESS_WIDTH-1:0] wbEndAddress = 0;
    always @(modbusFunction, quantity, startAddress) begin
        case(modbusFunction)
            FUN_READ_HOLDING_REGISTERS,
            FUN_WRITE_MULTIPLE_REGISTERS: begin
                wbStartAddress = OFFSET_HOLDING_REGISTERS + startAddress;
                    wbEndAddress = OFFSET_HOLDING_REGISTERS + startAddress +
quantity - 1;
            end
            FUN_READ_INPUT_REGISTERS: begin
                wbStartAddress = OFFSET_INPUT_REGISTERS + startAddress;
                      wbEndAddress = OFFSET_INPUT_REGISTERS + startAddress +
quantity - 1;
            end
            default: begin
                wbEndAddress = 0;
                wbStartAddress = 0;
            end
        endcase
    end
    /**/


    /* Transaction buffer begin*/
    reg [15:0] transactionBuffer [127:0];
    reg [6:0] transactionBufferWritePtr = 7'd0;
    reg [6:0] transactionBufferReadPtr = 7'd0;
    /* Transaction buffer end*/

    /* Send begin */
    localparam SSTATE_WAIT = 'h0;
    localparam SSTATE_STATION_ADDRESS = 'h1;
    localparam SSTATE_FUNCTION = 'h2;
    localparam SSTATE_ERROR_CODE = 'h3;
    localparam SSTATE_CRC_LO = 'h4;
    localparam SSTATE_CRC_HI = 'h5;
    localparam SSTATE_BEGIN = 'h6;
    localparam SSTATE_END = 'h7;
```

```
    localparam SSTATE_BYTE_COUNT = 'h8;
    localparam SSTATE_DATA_LO = 'ha;
    localparam SSTATE_WB_READ = 'hb;
    localparam SSTATE_DATA_HI = 'hc;
    localparam SSTATE_WB_WRITE_START = 'hd;
    localparam SSTATE_WB_WRITE = 'he;
    localparam SSTATE_ADDRESS_HI = 'hf;
    localparam SSTATE_ADDRESS_LO = 'h10;
    localparam SSTATE_QUANTITY_HI = 'h11;
    localparam SSTATE_QUANTITY_LO = 'h12;
    localparam SSTATE_WB_READ_START = 'h13;

    /* sstate begin */
    reg [7:0] sstate = SSTATE_WAIT;
    always @(posedge clk) begin
        sstate <= nextSstate;
    end
    /* sstate end */

    initial fifoDataOut = 8'h0;

    /* nextSstate begin */
    reg [7:0] nextSstate;
    always @(*) begin
        nextSstate = sstate;
        if(rst) begin
            nextSstate = SSTATE_WAIT;
        end else begin
            case(sstate)
                SSTATE_WAIT: begin
                    if(processRequest) begin
                        case(modbusFunction)
                            FUN_READ_HOLDING_REGISTERS,
                            FUN_READ_INPUT_REGISTERS:
                                nextSstate = SSTATE_BEGIN;
                            FUN_WRITE_MULTIPLE_REGISTERS: begin
                                nextSstate = SSTATE_WB_WRITE_START;
                            end
                        endcase
                    end
                end
                SSTATE_BEGIN: begin
                    nextSstate = SSTATE_FUNCTION;
                end
                SSTATE_CRC_LO: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_CRC_HI;
                    end
                end
                SSTATE_CRC_HI: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_END;
                    end
```

```
                end
                SSTATE_STATION_ADDRESS: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_FUNCTION;
                    end
                end
                SSTATE_FUNCTION: begin
                    if(fifoWriteAck) begin
                        if(error) begin
                            nextSstate = SSTATE_ERROR_CODE;
                        end else begin
                            case(modbusFunction)
                                FUN_READ_HOLDING_REGISTERS,
                                FUN_READ_INPUT_REGISTERS: begin
                                    nextSstate = SSTATE_BYTE_COUNT;
                                end
                                FUN_WRITE_MULTIPLE_REGISTERS: begin
                                    nextSstate = SSTATE_ADDRESS_HI;
                                end
                                default: begin
                                    nextSstate = SSTATE_WAIT;
                                end
                            endcase
                        end
                    end
                end
                SSTATE_ERROR_CODE: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_CRC_LO;
                    end
                end
                SSTATE_END: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_WAIT;
                    end
                end
                SSTATE_BYTE_COUNT: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_WB_READ_START;
                    end
                end
                SSTATE_WB_READ_START: begin
                    nextSstate = SSTATE_WB_READ;
                end
                SSTATE_WB_READ: begin
                    if(wbAckI) begin
                        nextSstate = SSTATE_DATA_HI;
                    end
                end
                SSTATE_DATA_HI: begin
                    if(fifoWriteAck) begin
                        nextSstate = SSTATE_DATA_LO;
                    end
```

```
                end
                SSTATE_DATA_LO: begin
                    if(fifoWriteAck) begin
                        if(wbCurrentAddress == wbEndAddress)
                            nextSstate = SSTATE_CRC_LO;
                        else
                            nextSstate = SSTATE_WB_READ;
                    end
                end
                SSTATE_WB_WRITE_START: begin
                    nextSstate = SSTATE_WB_WRITE;
                end
                SSTATE_WB_WRITE: begin
                    if(wbAckI) begin
                        if(wbCurrentAddress == wbEndAddress) begin
                            nextSstate = SSTATE_BEGIN;
                        end
                    end
                end
                SSTATE_ADDRESS_HI: begin
                    if(fifoWriteAck)
                        nextSstate = SSTATE_ADDRESS_LO;
                end
                SSTATE_ADDRESS_LO: begin
                    if(fifoWriteAck)
                        nextSstate = SSTATE_QUANTITY_HI;
                end
                SSTATE_QUANTITY_HI: begin
                    if(fifoWriteAck)
                        nextSstate = SSTATE_QUANTITY_LO;
                end
                SSTATE_QUANTITY_LO: begin
                    if(fifoWriteAck)
                        nextSstate = SSTATE_CRC_LO;
                end
                default: nextSstate = SSTATE_WAIT;
            endcase
        end // rst
    end
    /* nextSstate begin */

    /* wbCurrentAddress, wbCurrentData begin */
    reg [DATA_WIDTH-1:0] wbCurrentData = 0;
    always @(posedge clk) begin
        if(rst) begin
        end else begin
            if(sstate == SSTATE_WB_READ && wbAckI)
                wbCurrentData <= wbDatI;
            case(sstate)
                SSTATE_WB_READ_START: begin
                    wbCurrentAddress <= wbStartAddress;
                end
                SSTATE_DATA_LO: begin
```

```
                    wbCurrentAddress <= wbCurrentAddress + 1;
                end
                SSTATE_WB_WRITE_START: begin
                    wbCurrentAddress <= wbStartAddress;
                end
                SSTATE_WB_WRITE: begin
                    if(wbAckI)
                        wbCurrentAddress <= wbCurrentAddress + 1;
                end
            endcase
        end // rst
    end
    /* wbCurrentAddress, wbCurrentData end */
    always @(posedge clk) begin
        if(rst) begin
            fifoDataOut <= 8'h0;
        end else begin
            case(sstate)
                SSTATE_WAIT: begin
                    if(fifoWriteAck) begin
                        $display("fifoWriteAck during SSTATE_WAIT");
                    end
                end
                SSTATE_BEGIN: begin
                    fifoDataOut <= MODBUS_STATION_ADDRESS;
                end
                SSTATE_CRC_LO: begin
                    if(fifoWriteAck) begin
                        fifoDataOut <= ocrcOut[7:0];
                    end
                end
                SSTATE_CRC_HI: begin
                    if(fifoWriteAck) begin
                        fifoDataOut <= ocrcOut[15:8];
                    end
                end
                SSTATE_STATION_ADDRESS: begin
                    if(fifoWriteAck) begin
                        fifoDataOut <= MODBUS_STATION_ADDRESS;
                    end
                end
                SSTATE_FUNCTION: begin
                    if(fifoWriteAck) begin
                        if(error) begin
                            fifoDataOut <= { 1'b1, modbusFunction[6:0] };
                        end else begin
                            fifoDataOut <= modbusFunction;
                        end
                    end
                end
                SSTATE_ERROR_CODE: begin
                    if(fifoWriteAck) begin
                        fifoDataOut <= exceptionCode;
```

```
                end
            end
            SSTATE_BYTE_COUNT: begin
                if(fifoWriteAck) begin
                    fifoDataOut <= byteCount;
                    if(byteCount == 8'd0) begin
                        $display("Error: zero byte count");
                    end
                end
            end
            SSTATE_DATA_HI: begin
                fifoDataOut <= wbCurrentData[15:8];
            end
            SSTATE_DATA_LO: begin
                if(fifoWriteAck) begin
                    fifoDataOut <= wbCurrentData[7:0];
                end
            end
            SSTATE_ADDRESS_HI: begin
                fifoDataOut <= startAddressHi;
            end
            SSTATE_ADDRESS_LO: begin
                fifoDataOut <= startAddressLo;
            end
            SSTATE_QUANTITY_HI: begin
                fifoDataOut <= quantityHi;
            end
            SSTATE_QUANTITY_LO: begin
                fifoDataOut <= quantityLo;
            end
            default: ;
        endcase
    end // rst
end
/* Send end */


/* Output CRC begin */
wire [15:0] ocrcOut;
wire ocrcRst = sstate == SSTATE_WAIT;
Crc _crcOut(
    .data_in(fifoDataOut),
    .crc_en(ocrcEnabled),
    .crc_out(ocrcOut),
    .rst(ocrcRst),
    .clk(~clk)
);
/* Output CRC end */

/* ocrcEnabled begin */
reg ocrcEnabled = 1'b0;
always @(posedge clk) begin
    if(rst) begin
```

```
            ocrcEnabled <= 1'b0;
        end else begin
            case(sstate)
                SSTATE_BEGIN: ocrcEnabled <= 1'b1;
                SSTATE_DATA_HI: ocrcEnabled <= wbAckI;
                SSTATE_WB_READ,
                SSTATE_WB_READ_START,
                SSTATE_WB_WRITE_START,
                SSTATE_WB_WRITE,
                SSTATE_CRC_LO,
                SSTATE_CRC_HI: ocrcEnabled <= 1'b0;
                default: ocrcEnabled <= fifoWriteAck;
            endcase
        end
    end
    /* ocrcEnabled end */

    /* fifoWriteReq begin */
    initial fifoWriteReq = 1'b0;
    always @(posedge clk) begin
        if(rst) begin
            fifoWriteReq <= 1'b0;
        end else begin
            case(sstate)
                SSTATE_WAIT,
                SSTATE_WB_READ_START,
                SSTATE_WB_READ,
                SSTATE_WB_WRITE_START,
                SSTATE_WB_WRITE:
                    fifoWriteReq <= 1'b0;
                SSTATE_END:
                    fifoWriteReq <= ~fifoWriteAck;
                default:
                    fifoWriteReq <= 1'b1;
            endcase
        end
    end
    /* fifoWriteReq end */

    /* wishbone begin */
    assign wbDatO = transactionBuffer[transactionBufferReadPtr];
    assign wbAdrO = wbCurrentAddress;
    always @(*) begin
        wbCycO = 1'b0;
        wbStbO = 1'b0;
        wbWeO = 1'b0;
        if(~rst) begin
            case(sstate)
                SSTATE_WB_READ: begin
                    wbCycO = 1'b1;
                    wbStbO = 1'b1;
                    wbWeO = 1'b0;
                end
```

```
                SSTATE_WB_WRITE: begin
                    wbCycO = 1'b1;
                    wbStbO = 1'b1;
                    wbWeO = 1'b1;
                end
            endcase
        end
    end
    /* wishbone end */

    always @(posedge clk) begin
        if(rst) begin
            transactionBufferReadPtr <= 7'd0;
        end else begin
            case(sstate)
                SSTATE_WB_WRITE_START: begin
                    transactionBufferReadPtr <= 7'd0;
                end
                SSTATE_WB_WRITE: begin
                    if(wbAckI) begin
                         transactionBufferReadPtr <= transactionBufferReadPtr
+ 7'd1;
                    end
                end
            endcase
        end
    end
endmodule
```

Исходный код модуля подсчета контрольной суммы

```
//------------------------------------------------------------------------
--
// Copyright (C) 2009 OutputLogic.com
// This source file may be used and distributed without restriction
// provided that this copyright statement is not removed from the file
// and that any derivative work contains the original copyright notice
// and the associated disclaimer.
// THIS SOURCE FILE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS
// OR IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
// WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.
//------------------------------------------------------------------------
--
// CRC module for
//     data[7:0]
//     crc[15:0]=1+x^2+x^15+x^16;
//
module Crc(
      input [0:7] data_in,
      input        crc_en,
      output [0:15] crc_out,
```

```
      input        rst,
      input        clk);

      reg [0:15] lfsr_q,
                 lfsr_c;
      assign crc_out = lfsr_q;
      always @(*) begin
            lfsr_c[0] = lfsr_q[8] ^ lfsr_q[9] ^ lfsr_q[10] ^ lfsr_q[11] ^
lfsr_q[12] ^ lfsr_q[13] ^ lfsr_q[14] ^ lfsr_q[15] ^ data_in[0] ^ data_in[1] ^
data_in[2] ^ data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6] ^ data_in[7];
            lfsr_c[1] = lfsr_q[9] ^ lfsr_q[10] ^ lfsr_q[11] ^ lfsr_q[12] ^
lfsr_q[13] ^ lfsr_q[14] ^ lfsr_q[15] ^ data_in[1] ^ data_in[2] ^ data_in[3] ^
data_in[4] ^ data_in[5] ^ data_in[6] ^ data_in[7];
            lfsr_c[2] = lfsr_q[8] ^ lfsr_q[9] ^ data_in[0] ^ data_in[1];
            lfsr_c[3] = lfsr_q[9] ^ lfsr_q[10] ^ data_in[1] ^ data_in[2];
            lfsr_c[4] = lfsr_q[10] ^ lfsr_q[11] ^ data_in[2] ^ data_in[3];
            lfsr_c[5] = lfsr_q[11] ^ lfsr_q[12] ^ data_in[3] ^ data_in[4];
            lfsr_c[6] = lfsr_q[12] ^ lfsr_q[13] ^ data_in[4] ^ data_in[5];
            lfsr_c[7] = lfsr_q[13] ^ lfsr_q[14] ^ data_in[5] ^ data_in[6];
            lfsr_c[8] = lfsr_q[0] ^ lfsr_q[14] ^ lfsr_q[15] ^ data_in[6] ^
data_in[7];
            lfsr_c[9] = lfsr_q[1] ^ lfsr_q[15] ^ data_in[7];
            lfsr_c[10] = lfsr_q[2];
            lfsr_c[11] = lfsr_q[3];
            lfsr_c[12] = lfsr_q[4];
            lfsr_c[13] = lfsr_q[5];
            lfsr_c[14] = lfsr_q[6];
            lfsr_c[15] = lfsr_q[7] ^ lfsr_q[8] ^ lfsr_q[9] ^ lfsr_q[10] ^
lfsr_q[11] ^ lfsr_q[12] ^ lfsr_q[13] ^ lfsr_q[14] ^ lfsr_q[15] ^ data_in[0] ^
data_in[1] ^ data_in[2] ^ data_in[3] ^ data_in[4] ^ data_in[5] ^ data_in[6] ^
data_in[7];


      end // always

      always @(posedge clk) begin
            if(rst) begin
                  lfsr_q  <= {16{1'b1}};
            end
            else begin
                  lfsr_q  <= crc_en ? lfsr_c : lfsr_q;
            end
      end // always
endmodule // crc
```

Исходный код процессора

```
/* Команды:
    - SET записать значение в регистр:
        [1] [address] [value]
    - WAIT ждать значение в регистре и зафейлить тест, если не дождались:
```

```
        [2] [address] [bottom] [top] [timeout]
    - PAUSE выждать паузу:
        [3] [timeout]
    - WIN успешно завершить тест:
        [4]
*/
module Processor(
    input clk,
    input rst,

    // Wishbone
    output reg [ADDRESS_WIDTH-1:0] wbAdrO,
    output reg [15:0] wbDatO,
    input [15:0] wbDatI,
    output reg wbCycO,
    output reg wbStbO,
    output reg wbWeO,
    input wbAckI,

    input [15:0] controlReg,
    output [15:0] statusReg
);
    parameter TIMEOUT_CLOCK_DIVISOR = 1;
    parameter ADDRESS_WIDTH = 24;
    parameter PROGMEM_START = 'h10000;
    parameter PROGMEM_END   = 'h1FFFF;
    parameter REGMEM_START  = 'h00000;
    parameter REGMEM_END    = 'h0FFFF;

    localparam OPCODE_SET = 16'h0001;
    localparam OPCODE_WAIT = 16'h0002;
    localparam OPCODE_PAUSE = 16'h0003;
    localparam OPCODE_WIN = 16'h0004;

    function [15:0] instructionLength;
        input [15:0] opcode;
        begin
            case(opcode)
                OPCODE_SET: instructionLength = 16'd3;
                OPCODE_WAIT: instructionLength = 16'd5;
                OPCODE_PAUSE: instructionLength = 16'd2;
                OPCODE_WIN: instructionLength = 16'd1;
                default: instructionLength = 16'd1;
            endcase
        end
    endfunction

    /* controlReg begin */
    wire controlHalt = controlReg[0];
    /* controlReg end */

    /* statusReg begin */
    wire isRunning = state != STATE_HALT && !isFailed && !isSucceeded;
```

```
    wire isFailed = state == STATE_FAIL;
    wire isSucceeded = state == STATE_SUCCESS;
    assign statusReg = { 13'd0, isRunning, isFailed, isSucceeded };
    /* statusReg end */

    /* State machine begin */
    localparam STATE_HALT = 'h0;
    localparam STATE_FETCH_CMD_R = 'h1;
    localparam STATE_FETCH_CMD_A = 'h2;
    localparam STATE_FETCH_ADDR_A = 'h3;
    localparam STATE_FETCH_VAL_BOT_A = 'h4;
    localparam STATE_FETCH_VAL_TOP_A = 'h5;
    localparam STATE_FETCH_VAL_A = 'h6;
    localparam STATE_FETCH_TIME_A = 'h7;
    localparam STATE_WRITE_REG_A = 'h8;
    localparam STATE_READ_REG_A = 'h9;
    localparam STATE_WAIT = 'ha;
    localparam STATE_FAIL = 'hb;
    localparam STATE_SUCCESS = 'hc;
    localparam STATE_READ_REG_R = 'hd;

    reg [7:0] state = STATE_HALT;
    reg [7:0] nextState;

      wire [15:0] realOpcode = (state == STATE_FETCH_CMD_A) ? tempOpcode :
opcode;
    wire [15:0] tempOpcode = wbDatI;
    wire [15:0] tempReg = wbDatI;

      wire [15:0] realRegValue = (state == STATE_FETCH_VAL_A) ? wbDatI :
regValue;

    always @(posedge clk) begin
        if(rst || controlHalt) begin
            state <= STATE_HALT;
        end else begin
            state <= nextState;
        end
    end
    always @(*) begin
        if(rst || controlHalt) begin
            nextState = STATE_HALT;
        end else begin
            case(state)
                STATE_HALT: begin
                    nextState = STATE_FETCH_CMD_R;
                end
                STATE_FETCH_CMD_R: begin
                    nextState = STATE_FETCH_CMD_A;
                end
                STATE_FETCH_CMD_A: begin
                    if(wbAckI) begin
                        case(tempOpcode)
```

```
                            OPCODE_SET,
                            OPCODE_WAIT: begin
                                nextState = STATE_FETCH_ADDR_A;
                            end
                            OPCODE_PAUSE: begin
                                nextState = STATE_FETCH_TIME_A;
                            end
                            OPCODE_WIN: begin
                                nextState = STATE_SUCCESS;
                            end
                            default: begin
                                nextState = STATE_FETCH_CMD_R;
                            end
                        endcase
                    end else
                        nextState = state;
                end
                STATE_FETCH_ADDR_A: begin
                    if(wbAckI) begin
                        case(opcode)
                            OPCODE_SET: nextState = STATE_FETCH_VAL_A;
                            OPCODE_WAIT: nextState = STATE_FETCH_VAL_BOT_A;
                            default: begin
                                      $display("Automata error: bad opcode in
STATE_FETCH_ADDR_A");
                                nextState = STATE_HALT;
                            end
                        endcase
                    end else
                        nextState = state;
                end
                STATE_FETCH_VAL_A: begin
                    if(wbAckI) begin
                        nextState = STATE_WRITE_REG_A;
                    end else
                        nextState = state;
                end
                STATE_FETCH_VAL_BOT_A: begin
                    if(wbAckI) begin
                        nextState = STATE_FETCH_VAL_TOP_A;
                    end else
                        nextState = state;
                end
                STATE_FETCH_VAL_TOP_A: begin
                    if(wbAckI) begin
                        nextState = STATE_FETCH_TIME_A;
                    end else
                        nextState = state;
                end
                STATE_FETCH_TIME_A: begin
                    if(wbAckI) begin
                        case(opcode)
                            OPCODE_PAUSE: nextState = STATE_WAIT;
```

```
                            OPCODE_WAIT: nextState = STATE_READ_REG_R;
                            default: begin
                                     $display("Automata error: bad opcode in
STATE_FETCH_TIME_A");
                                nextState = STATE_HALT;
                            end
                        endcase
                    end else
                        nextState = state;
                end
                STATE_WRITE_REG_A: begin
                    if(wbAckI)
                        nextState = STATE_FETCH_CMD_R;
                    else
                        nextState = state;
                end
                STATE_READ_REG_R: begin
                    nextState = STATE_READ_REG_A;
                end
                STATE_READ_REG_A: begin
                    if(wbAckI) begin
                        if(tempReg >= regBottom && tempReg <= regTop) begin
                            nextState = STATE_FETCH_CMD_R;
                        end else begin
                            if(timeout) begin
                                nextState = STATE_FAIL;
                            end else
                                nextState = STATE_READ_REG_R;
                        end
                    end else
                        nextState = state;
                end
                STATE_WAIT: begin
                    if(timeout)
                        nextState = STATE_FETCH_CMD_R;
                    else
                        nextState = state;
                end
                default: nextState = state;
            endcase
        end
    end
    /* State machine end */

    /* Wishbone begin */
    initial begin
        wbCycO = 1'b0;
        wbStbO = 1'b0;
        wbWeO = 1'b0;
        wbDatO = 16'd0;
        wbAdrO = 0;
    end
    always @(posedge clk) begin
```

```
        if(rst || controlHalt) begin
            wbCycO <= 1'b0;
            wbStbO <= 1'b0;
            wbWeO <= 1'b0;
            wbDatO <= 16'd0;
            wbAdrO <= 0;
        end else begin
            // wbWeO begin
            case(nextState)
                STATE_WRITE_REG_A: begin
                    wbWeO <= 1'b1;
                end
                default: wbWeO <= 1'b0;
            endcase
            // wbWeO end

            // wbCycO and wbStbO begin
            case(nextState)
                STATE_FETCH_CMD_A,
                STATE_FETCH_ADDR_A,
                STATE_FETCH_VAL_A,
                STATE_FETCH_VAL_BOT_A,
                STATE_FETCH_VAL_TOP_A,
                STATE_FETCH_TIME_A,
                STATE_WRITE_REG_A,
                STATE_READ_REG_A: begin
                    wbCycO <= 1'b1;
                    wbStbO <= 1'b1;
                end
                default: begin
                    wbCycO <= 1'b0;
                    wbStbO <= 1'b0;
                end
            endcase
            // wbCycO and wbStbO end

            // wbAdrO and wbDatO begin
            case(nextState)
                STATE_FETCH_CMD_A: begin
                    wbAdrO <= instructionPointer;
                end
                STATE_FETCH_ADDR_A: begin
                    wbAdrO <= instructionPointer + 1;
                end
                STATE_FETCH_VAL_A: begin
                    wbAdrO <= currentInstructionPointer + 2;
                end
                STATE_FETCH_VAL_BOT_A: begin
                    wbAdrO <= currentInstructionPointer + 2;
                end
                STATE_FETCH_VAL_TOP_A: begin
                    wbAdrO <= currentInstructionPointer + 3;
                end
```

```
                STATE_FETCH_TIME_A: begin
                    case(realOpcode)
                          OPCODE_PAUSE: wbAdrO <= currentInstructionPointer +
1;
                        OPCODE_WAIT: wbAdrO <= currentInstructionPointer + 4;
                    endcase
                end
                STATE_WRITE_REG_A: begin
                    wbAdrO <= regAddress;
                    wbDatO <= realRegValue;
                end
                STATE_READ_REG_A: begin
                    wbAdrO <= regAddress;
                end
            endcase
            // wbAdrO and wbDatO end
        end
    end
    /* Wishbone end */

    /* instructionPointer begin */
    reg [ADDRESS_WIDTH-1:0] instructionPointer = PROGMEM_START;
    reg [ADDRESS_WIDTH-1:0] currentInstructionPointer = PROGMEM_START;
    always @(posedge clk) begin
        if(rst || controlHalt) begin
            instructionPointer <= PROGMEM_START;
            currentInstructionPointer <= PROGMEM_START;
        end else begin
            case(state)
                STATE_HALT: begin
                    instructionPointer <= PROGMEM_START;
                end
                STATE_FETCH_CMD_A: begin
                    if(wbAckI) begin
                                instructionPointer <= instructionPointer +
instructionLength(tempOpcode);
                        currentInstructionPointer <= instructionPointer;
                    end
                end
            endcase
        end
    end
    /* instructionPointer end */

    /* opcode begin */
    reg [15:0] opcode = 16'd0;
    always @(posedge clk) begin
        if(rst || controlHalt) begin
            opcode <= 16'd0;
        end else begin
            if(state == STATE_FETCH_CMD_A)
                opcode <= tempOpcode;
        end
```

```
    end
    /* opcode end */

    /* timeoutValue, regValue, regAddress, regBottom and regTop begin */
    reg [15:0] regBottom = 16'd0;
    reg [15:0] regTop = 16'd0;
    reg [15:0] regAddress = 16'd0;
    reg [15:0] timeoutValue = 16'd0;
    reg [15:0] regValue = 16'd0;
    always @(posedge clk) begin
        if(rst || controlHalt) begin
            regBottom <= 16'd0;
            regTop <= 16'd0;
            regAddress <= 16'd0;
            timeoutValue <= 16'd0;
            regValue <= 16'd0;
        end else begin
            if(state == STATE_FETCH_VAL_BOT_A && wbAckI)
                regBottom <= wbDatI;
            if(state == STATE_FETCH_VAL_TOP_A && wbAckI)
                regTop <= wbDatI;
            if(state == STATE_FETCH_ADDR_A && wbAckI)
                regAddress <= wbDatI;
            if(state == STATE_FETCH_TIME_A && wbAckI)
                timeoutValue <= wbDatI;
            if(state == STATE_FETCH_VAL_A && wbAckI)
                regValue <= wbDatI;
        end
    end
    /* timeoutValue, regAddress, regBottom and regTop end */

    /* timeout begin */
    wire timeout = timeoutCounter >= timeoutValue;
    reg [15:0] slowClockCounter = 16'd0;
    reg [15:0] timeoutCounter = 16'd0;
    always @(posedge clk) begin
        if(rst || controlHalt) begin
            slowClockCounter <= 16'd0;
        end else begin
            case(state)
                STATE_READ_REG_A,
                STATE_READ_REG_R,
                STATE_WAIT: begin
                    if(slowClockCounter == TIMEOUT_CLOCK_DIVISOR) begin
                        slowClockCounter <= 16'd0;
                        timeoutCounter <= timeoutCounter + 16'd1;
                    end else
                        slowClockCounter <= slowClockCounter + 16'd1;
                end
                default: begin
                    slowClockCounter <= 16'd0;
                    timeoutCounter <= 16'd0;
                end
```

```
            endcase
        end
    end
    /* timeout end */
endmodule
```

Исходный код системного арбитра

```
module Arbiter(
    input clk,
    input rst,

    input [MASTERS_COUNT-1:0] mCycI,
    input [MASTERS_COUNT-1:0] mStbI,
    input [MASTERS_COUNT-1:0] mWeI,
    output [MASTERS_COUNT-1:0] mAckO,
    input [ADDRESS_WIDTH*MASTERS_COUNT-1:0] mAdrIPacked,
    input [DATA_WIDTH*MASTERS_COUNT-1:0] mDatIPacked,
    output [DATA_WIDTH*MASTERS_COUNT-1:0] mDatOPacked,

    output reg sCycO,
    output reg sStbO,
    output reg sWeO,
    input sAckI,
    output [ADDRESS_WIDTH-1:0] sAdrO,
    input [DATA_WIDTH-1:0] sDatI,
    output [DATA_WIDTH-1:0] sDatO
);
    parameter MASTERS_WIDTH = 1;
    localparam MASTERS_COUNT = 1 << MASTERS_WIDTH;
    parameter ADDRESS_WIDTH = 32;
    parameter DATA_WIDTH = 32;

    wire [ADDRESS_WIDTH-1:0] mAdrI [MASTERS_COUNT-1:0];
    wire [DATA_WIDTH-1:0] mDatI [MASTERS_COUNT-1:0];
    wire [DATA_WIDTH-1:0] mDatO [MASTERS_COUNT-1:0];

    genvar i;
    generate
        for(i = 0; i < MASTERS_COUNT; i = i + 1) begin : genblock
                        assign  mAdrI[i]  =  mAdrIPacked[ADDRESS_WIDTH*(i+1)-
1:ADDRESS_WIDTH*i];
            assign mDatI[i] = mDatIPacked[DATA_WIDTH*(i+1)-1:DATA_WIDTH*i];
            assign mDatOPacked[DATA_WIDTH*(i+1)-1:DATA_WIDTH*i] = mDatO[i];
        end
    endgenerate

    reg [MASTERS_WIDTH-1:0] currentMaster = 0;

    wire currentStb = mStbI[currentMaster];
    wire currentCyc = mCycI[currentMaster];
```

```
    wire currentWe = mWeI[currentMaster];
    wire nextCyc = mCycI[currentMaster + {{(MASTERS_WIDTH-1){1'b0}}, 1'b1}];

    localparam STATE_RESET = 0;
    localparam STATE_NEXT = 1;
    localparam STATE_CYCLE = 2;
    reg [1:0] state = STATE_RESET;
    always @(negedge clk) begin
        if(rst) begin
            state <= STATE_RESET;
        end else begin
            case(state)
                STATE_RESET: begin
                    state <= STATE_NEXT;
                end
                STATE_NEXT: begin
                    if(nextCyc)
                        state <= STATE_CYCLE;
                end
                STATE_CYCLE: begin
                    if(!currentCyc)
                        state <= STATE_NEXT;
                end
                default: begin
                    state <= STATE_RESET;
                end
            endcase
        end
    end

    always @(negedge clk) begin
        if(rst) begin
            currentMaster <= 0;
        end else begin
            case(state)
                STATE_RESET: currentMaster <= 0;
                STATE_NEXT: currentMaster <= currentMaster + 1;
            endcase
        end
    end

    reg cycLatch = 1'b0;
    reg stbLatch = 1'b0;
    reg weLatch = 1'b0;
    always @(posedge clk) begin
        if(rst) begin
            cycLatch <= 1'b0;
            stbLatch <= 1'b0;
            weLatch <= 1'b0;
        end else begin
            case(state)
                STATE_RESET: begin
                    cycLatch <= 1'b0;
```

```
                    stbLatch <= 1'b0;
                    weLatch <= 1'b0;
                end
                STATE_CYCLE: begin
                    cycLatch <= currentCyc;
                    stbLatch <= currentStb;
                end
            endcase
        end
    end
    always @(*) begin
        case(state)
            STATE_CYCLE: begin
                sStbO = stbLatch & currentStb;
                sCycO = cycLatch & currentCyc;
                sWeO = weLatch & currentWe;
            end
            default: begin
                sStbO = 1'b0;
                sCycO = 1'b0;
                sWeO = 1'b0;
            end
        endcase
    end

    generate
        for(i = 0; i < MASTERS_COUNT; i = i + 1) begin : genblock1
             assign mAckO[i] = (state == STATE_CYCLE && currentMaster == i) ?
sAckI : 1'b0;
            assign mDatO[i] = sDatI;
        end
    endgenerate
    assign sDatO = mDatI[currentMaster];
    assign sAdrO = mAdrI[currentMaster];
endmodule
```

Исходный код модуля ввода-вывода

```
module Port(
    input clk,
    input rst,

    input wbStbI,
    input wbCycI,
    input wbWeI,
    output reg wbAckO,

    input [DATA_WIDTH-1:0] wbDatI,
    output reg [DATA_WIDTH-1:0] wbDatO,
    input wbAdrI, // 0 - data, 1 - mode (0 - read, 1 - write)
```

```
    inout [DATA_WIDTH-1:0] port
);
    parameter DATA_WIDTH = 16;

    initial wbAckO = 1'b0;
    initial wbDatO = {DATA_WIDTH{1'b0}};
    reg [DATA_WIDTH-1:0] mode = {DATA_WIDTH{1'b0}};
    reg [DATA_WIDTH-1:0] data = {DATA_WIDTH{1'b0}};

    genvar i;
    generate
        for(i = 0; i < DATA_WIDTH; i = i + 1) begin : genblock
            assign port[i] = mode[i] ? data[i] : 1'bz;
        end
    endgenerate

    wire portIn = port;

    always @(negedge clk) begin
        if(rst) begin
            wbAckO <= 1'b0;
            mode <= {DATA_WIDTH{1'b0}};
            data <= {DATA_WIDTH{1'b0}};
            wbDatO <= {DATA_WIDTH{1'b0}};
        end else begin
            if(wbStbI && wbCycI) begin
                wbAckO <= 1'b1;
                case(wbAdrI)
                    1'b0: begin
                        if(wbWeI)
                            data <= wbDatI;
                        wbDatO <= portIn;
                    end
                    1'b1: begin
                        if(wbWeI)
                            mode <= wbDatI;
                        wbDatO <= mode;
                    end
                endcase
            end else begin
                wbAckO <= 1'b0;
            end
        end
    end
endmodule

module InPort(
    input clk,
    input rst,

    input wbStbI,
    input wbCycI,
    input wbWeI,
```

```
    output reg wbAckO,

    output reg [DATA_WIDTH-1:0] wbDatO,

    input [DATA_WIDTH-1:0] port
);
    parameter DATA_WIDTH = 16;

    initial begin
        wbAckO = 1'b0;
        wbDatO = {DATA_WIDTH{1'b0}};
    end
    always @(negedge clk) begin
        if(rst) begin
            wbAckO <= 1'b0;
            wbDatO <= {DATA_WIDTH{1'b0}};
        end else begin
            wbAckO <= wbStbI & wbCycI;
            wbDatO <= port;
        end
    end
endmodule

module OutPort(
    input clk,
    input rst,

    input wbStbI,
    input wbCycI,
    input wbWeI,
    output reg wbAckO,

    output [DATA_WIDTH-1:0] wbDatO,
    input [DATA_WIDTH-1:0] wbDatI,

    output reg [DATA_WIDTH-1:0] port
);
    parameter DATA_WIDTH = 16;

    assign wbDatO = wbDatI;
    initial begin
        wbAckO = 1'b0;
        port = {DATA_WIDTH{1'b0}};
    end
    always @(negedge clk) begin
        if(rst) begin
            wbAckO <= 1'b0;
            port <= {DATA_WIDTH{1'b0}};
        end else begin
            wbAckO <= wbStbI & wbCycI;
            if(wbStbI & wbCycI & wbWeI)
                port <= wbDatI;
        end
```

```
    end
endmodule
```