

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет

Вычислительной техники

Кафедра

Вычислительная техника

Направление подготовки 09.03.01 «Информатика и вычислительная техника»

Профиль Вычислительные машины, комплексы, системы и сети

**ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ
РАБОТА БАКАЛАВРА**

на тему

**АППАРАТНЫЙ ПЛАНИРОВЩИК С ГЛОБАЛЬНОЙ
ОЧЕРЕДЬЮ ЗАДАЧ**

Студент Алексеева Алина Александровна
(подпись, дата) 17.06.19

Руководитель Бикташев Р.А.
(подпись, дата)

Нормоконтролёр Кучин А. В. 17.06.19
(подпись, дата)

Работа допущена к защите (протокол заседания кафедры от 31.05.19 № 35-09-19)

Заведующий кафедрой Митрохин М.А.
(подпись)

Работа защищена с отметкой отлично (протокол заседания ГЭК от 19.06.19 № 1)

Секретарь ГЭК Гурин Е. И.
(подпись)

Пенза, 2019

Федеральное государственное бюджетное образовательное учреждение
высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет
Вычислительной техники

Кафедра
Вычислительная
техника

«Утверждаю»
Заведующий кафедрой
М.А. Митрохин
«23» 04 2019 г.

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1. Студентка Алексеева Алина Александровна гр. 15BB1 факультета вычислительной техники.

2. Тема работы: «Аппаратный планировщик с глобальной очередью задач».

Тема утверждена приказом ПГУ № 554/о от "29" апреля 2019 г.

3. Руководитель работы к.т.н., профессор кафедры ВТ Бикташев Р.А.

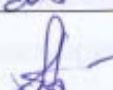
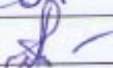
4. Задание на работу (назначение разработки, исходные данные и т.п.).

Назначение разработки – программа предназначена для реализации алгоритма планирования и диспетчеризации задач. Программа должна выполнять генерацию идентификаторов (ID) задач, имитацию работы процессоров многопроцессорной системы, а также выполнять функции планировщика задач, формирующего и поддерживающего очередь задач, и функции диспетчера задач, выполняющего выбор задач из очереди, запуск её на выполнение в любом свободном процессоре и удаление задач из очереди, получивших полное обслуживание.

5. Перечень подлежащих разработке вопросов.

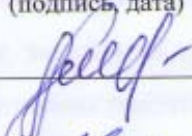
Требуется произвести анализ предметной области, разработать программные модули, которые будут обеспечить выполнение алгоритма диспетчеризации с глобальной очередью задач. Произвести отладку и тестирование, оформить выпускную квалификационную работу.

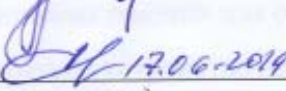
6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	Анализ предметной области	10	07.05.16	
2	Определение модулей программы	5	12.05.16	
3	Разработка основных функций	25	14.05.16	
4	Отладка и тестирование	20	28.05.16	
5	Оформление выпускной работы	30	04.06.16	

Дата выдачи задания 20 апреля 2019 г.

Руководитель бакалаврской работы  Бикташев Р.А.
(подпись, дата)

Задание к исполнению приняла студентка  Алексеева А.А.

Работу к защите допустить
Декан факультета  17.06.2019 Фионова Л.Р.
(подпись, дата)

РЕФЕРАТ

Бакалаврская работа содержит 64 листа, 14 рисунков, 10 источников, 2 приложения.

АППАРАТНЫЙ ПЛАНИРОВЩИК С ГЛОБАЛЬНОЙ ОЧЕРЕДЬЮ ЗАДАЧ.

Цель работы – разработка аппаратного планировщика с глобальной очередью задач, обеспечивающего высокую реакцию на запрос. Планировщик выполнен в виде программы на языке VHDL и обеспечивает возможность приема задач из общей очереди и распределения их по свободным процессорам, а также их синхронизацию методом «рандеву».

В результате проведенной работы создан проект, удовлетворяющий исходным требованиям. А именно, создан набор программных модулей на языке разработки аппаратуры, обеспечивающих работу аппаратного планировщика в виде сопроцессора в составе многопроцессорной системы. Созданные модули позволяют выполнять различные функции, моделирование аппаратного планировщика, имитацию работы процессоров и генерацию пакетов задач (идентификаторов задач - ID). Программа выполняет равномерное распределение задач между процессорами и их синхронизацию, а также имитацию наполнения пакетов для обработки.

Данная программа имитирует аппаратное устройство планирования задач в многопроцессорной ОС.

					ПГУ 2.090301.01.001 ПЗ			
Изм	Лист	№ докум.	Подпись	Дата				
Разработ	Алексеева А.А				Аппаратный планировщик с глобальной очередью задач Пояснительная записка	Лит	Лист	Листов
Проверил	Бикташев Р.А							
Н.Контр	Кучин А.В					ФВТ, каф ВТ гр 15вв1		
Утверд	Митрохин М.А							

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	6
1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ	8
1.1 Постановка задачи.....	8
1.3 Выбор решения.....	9
2. ПРОЕКТИРОВАНИЕ алгоритма.....	12
2.1. Блок управления очередью	17
2.2. Блок очереди задач.....	18
2.3. Блок выбора процессора.....	18
2.4. Блок синхронизации и назначения задач.....	19
3 ОПИСАНИЕ РАЗРАБОТКИ алгоритма.....	20
3.1 Выбор подхода для реализации алгоритма.....	20
3.1 Разработка модулей алгоритма.....	26
3. Формальное описание реализации алгоритма планировщика	29
4. Моделирование планировщика на языке VHDL и анализ результатов	35
4.1. Производительность диспетчера	36
Нормальный режим работы.	37
4.1.1. Перегрузка (все процессора заняты).	38
ЗАКЛЮЧЕНИЕ	39
5. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	40
6. ПРИЛОЖЕНИЕ А ЛИСТИНГИ ПРОГРАММЫ	41
7. ПРИЛОЖЕНИЕ В ПРЕЗЕНТАЦИЯ.....	58

ВВЕДЕНИЕ

Компьютерная промышленность начиная с первых дней своего существования постоянно стремилась к достижению максимальной вычислительной мощности. Компьютер мог выполнять дотрехсот операций в секунду, с легкостью превосходя производительность предшествующих ему вычислительных машин. Быстродействие современных ЭВМ в миллионы раз превышает возможности ПЭВМ, но это является стимулом к открытию новых технологий для повышения быстродействия вычислительных машин.

Любая сфера общества на данный момент очень тесно связано с использованием вычислительных машин, и чем дальше, тем больше, возникает потребность в более высокой скорости обработки информации за счет аппаратных ресурсов, используемых ЭВМ. Какими бы мощными ни становились компьютеры, их мощности все равно не хватает. В прошлом проблема всегда решалась за счет повышения тактовой частоты. К сожалению, этому повышению уже начинает препятствовать ряд фундаментальных ограничений.

Уже достаточно давно наметилась тенденция к переходу от программной реализации алгоритмов для операционной системы к аппаратной, которая вызвана стремлением к увеличению отношения производительности и понижении стоимости ЭВМ, которая обусловлена высоким развитием микроэлектроники.

Действительно, аппаратная реализация алгоритмов ОС позволяет уменьшить время выполнения необходимого алгоритма и, следовательно, повысить производительность ЭВМ, и сократить накладные расходы. Это обуславливается тем, что стоимость дополнительного оборудования для аппаратной реализации, благодаря прогрессу элементной базы, с каждым годом падает. Кроме того, аппаратная реализация алгоритмов позволяет в

определенной степени уменьшить трудности создания программного обеспечения, а также дает возможность улучшить совместимость различных версий ОС для компьютеров, построенных на разных платформах. Кроме того, в значительной степени понижается трудоемкость выполняемых операций по отладке низкого уровня ОС, тем самым повышая корректность всего программного обеспечения, благодаря лучшей формализации функций, реализованных аппаратным способом нежели программным.

Данная технология является достаточно актуальной на сегодняшний день, так как люди всегда будут заинтересованы в вопросе повышения быстродействия и достижения новых вычислительных мощностей за счет сокращения, как трудозатрат, так и расходов на разработку алгоритмов, поэтому данная тенденция продолжает развиваться, находя применение во множестве разных сфер.

В данной работе будет разработан алгоритм планировщика с глобальной очередью задач.

1. АНАЛИЗ ПРЕДМЕТНОЙ ОБЛАСТИ

1.1 Постановка задачи

Необходимо разработать аппаратный планировщик с глобальной очередью задач. Программа должна выполнять следующие функции:

- 1) Выборка задачи;
- 2) Назначение задачи в очередь;
- 3) Равномерное распределение задач для процессора;
- 4) Синхронизацию функционирования очереди задач с функционированием процессорного блока;
- 5) При возникновении прерывания, задача должна оставаться в той же очереди, куда и была назначена изначально;
- 6) После обработки, задача должна быть удалена из очереди.

Рассматриваемая программа должна быть разработана в виде набора программных модулей.

1.2 Актуальность работы

В настоящее время главные недостатки программного обеспечения, на котором базируются все традиционные операционные системы, могут быть устранены путем реализации функционала ядра аппаратными средствами. Данная предметная область появилась давно и на данный момент так же развивается.

Традиционные операционные системы имеют значительные накладные расходы на выполнение трудоемких функций и, в частности, на синхронизацию процессов, связанных с планированием задач, поскольку реализуются программным способом в пространстве ядра с применением механизмов критических секций, семафоров, рандеву и мониторов. Такие способы организации работы считаются достоверными и проверены временем, но требуют значительных временных затрат на выполнение

системных вызовов. Например, вхождение процесса в монитор и реализация очереди блокированных процессов, возникающей из-за конкуренции множества процессоров при доступе к планировщику, требуют выполнения программных прерываний, которые существенно увеличивают время ожидания выполнения процессов в системе и поэтому существенно снижают общую производительность многопроцессорной системы. Известно, что на выполнение планирования уходит до десятка тысяч процессорных тактов, что соответствует временным потерям в несколько микросекунд. Такое положение недопустимо для систем управляющих систем жесткого реального времени, в которых задачи должны быть выполнены строго в назначенный срок, иначе управляющая система может повести себя неадекватно или даже привести к катастрофе. Системные затраты времени можно значительно уменьшить применением аппаратной поддержки (сопроцессора) для выполнения функции, связанной с планированием задач, что приведет к увеличению коэффициента использования процессоров для выполнения приложений, а также сокращению времени ответа, улучшению надежности и безопасности систем управления. При этом функция планирования ложится на сопроцессор, реализованный аппаратно, освобождая центральные процессоры для выполнения приложений.

Поэтому на данный момент актуальной является задача создания аппаратного устройства (сопроцессора), позволяющего реализовать ускоренный алгоритм планирования и диспетчеризации задач.

1.3 Выбор решения

Для реализации поставленной задачи будет использован алгоритм распределения с глобальной очередью задач, который чаще всего применяется в многопроцессорных системах.

Для решения вопросов аппаратной реализации основных функций управления параллельной обработкой обычно используется формальный метод описания таких алгоритмов. В данной области самым популярным из формальных методов является язык, базирующийся на логике построения недетерминированных конечных автоматов (НДА), позволяющий описывать все реализуемые в алгоритме события в стандартной форме в виде систем канонических уравнений (НД SKU).

Формальное представление алгоритма в виде конечного автомата позволяет достаточно просто реализовать его, используя аппаратные средства. Для того, чтобы автоматизировать процесс построения аппаратного обеспечения алгоритма управления, использован язык VHDL, позволяющий проводить функциональное моделирование и перенос схемы на ПЛИС.

Для реализации работы алгоритма будет использован программный инструмент XilinxIse 14.7. Это программный инструмент, разработанный Xilinx для синтеза и анализа проектов HDL, позволяющий разработчику синтезировать проекты, выполнять анализ времени, анализировать диаграммы RTL и настраивать различные устройства с помощью программаторов.

Система является открытой для пользователей – поддерживаются различные форматы, различные семейства микросхем для проектирования различных устройств[1].

В качестве базы для реализации алгоритма работы будет использовано семейство микросхем Artix7, которое является правопреемником Spartan-6 и обладает по сравнению с ним целым рядом преимуществ, а именно, вдвое меньшее энергопотребление и тридцатипроцентное увеличение производительности.

Разработка алгоритма будет выполнена с использованием языка описания аппаратуры интегральных схем VHDL.

Язык проектирования VHDL является базовым языком при разработке аппаратуры современных вычислительных систем. Данный язык VHDL обеспечивает высокоуровневую абстракцию описания аппаратных средств благодаря наличию как множества predefined типов данных, так и возможности создавать пользовательские иерархически организованные типы данных на основе базовых, заложенных в языке. VHDL является формальной записью, предназначенной для описания функции и логической организации цифровой системы. Язык предназначен для моделирования преимущественно на вентильном уровне, уровне регистровых передач и корпусов микросхем, а также успешно используется и при синтезе устройств.

Используя описанное выше средство разработки и язык проектирования VHDL можно создать программу, реализующую алгоритм планировщика, который будет удовлетворять всем исходным требованиям.

2. ПРОЕКТИРОВАНИЕ АЛГОРИТМА

Проектирование алгоритма целесообразнее начать с составления общей схемы. Схема, отображающая самые основные действия разрабатываемых программных модулей, представлена ниже (Рисунок 1).

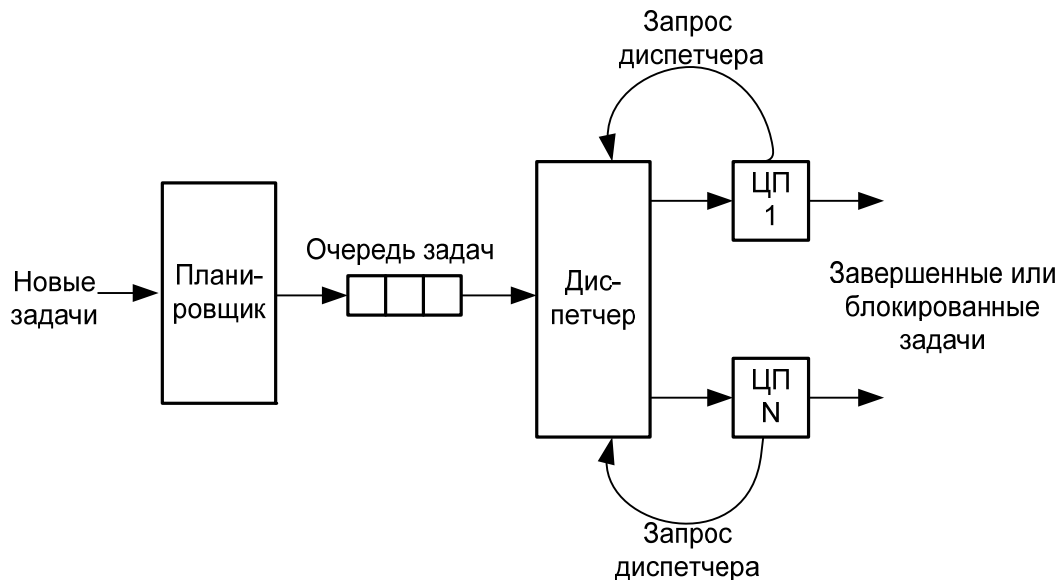


Рисунок 1. Общая схема алгоритма

После составления схемы будет продумана модель программы, произведено разбиение на основные модули и определен необходимый для работы функционал. После чего составлены примерные алгоритмы программных модулей, которые могут видоизменяться в процессе написания кода и его отладки.

Устройство аппаратной поддержки рассмотрено на примере многопроцессорной системы, состоящей из восьми процессоров, объединенных общей шиной. Возможны два варианта подключения диспетчера к процессору.

В первом варианте для передачи сигналов между процессором и диспетчером используется общая шина, что создает нежелательный дополнительный трафик на ней.

Во втором варианте в каждом процессорном узле реализован интерфейс для запроса и приёма идентификаторов задач от диспетчера.

Реализован второй подход, хотя он несколько усложняет процессорный блок, поскольку требует дополнительных выводов для вновь введенных сигналов. Общий вид системы представлен на рисунке ниже (Рисунок 2).

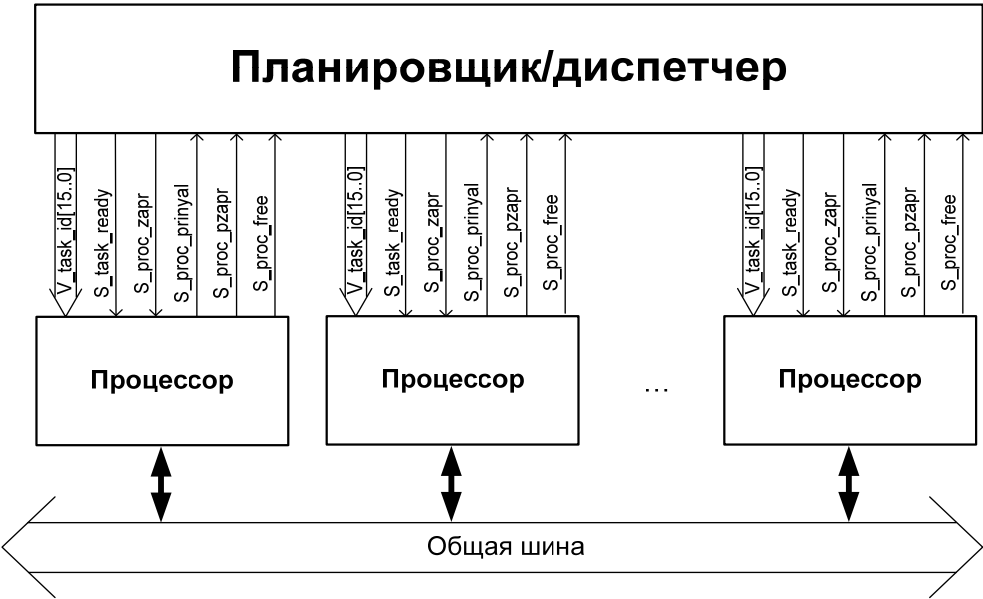


Рисунок 2. Общий вид многопроцессорной системы с аппаратной поддержкой планировщика задач.

К главным плюсам такой организации также можно отнести тот факт, что выход из строя одного из процессоров не влияет на работоспособность всей системы в целом. Если выйдет из строя один или несколько процессоров, то работоспособность многопроцессорной системы сохранится при небольшом снижении общей производительности.

Интерфейс взаимодействия процессора с диспетчером показан на рисунке ниже (Рисунок 3).



Рисунок 3.Интерфейс взаимодействия одного процессора с планировщиком/диспетчером.

«Клиентский» процесс развивается следующим образом. Если имеется некоторый процесс, ожидающийобслуживания, то диспетчер, при условии, что имеются свободные процессоры, устанавливает активный уровень сигнала «Запрос» (*S_proc_zapr*)у того процессора, который выбирается из числа свободных по заданному алгоритму. Процессор, получивший сигнал диспетчера о его выборе для обслуживания текущей задачи, выходит из спящего режима, и сообщаетдиспетчеру о том, что готов к приёму и выполнению задачи. Для этого процессор устанавливает активный уровень сигнала «Подтверждение запроса» (*S_proc_pzapr*), который воспринимает диспетчер. Кроме того, процессор снимает сигнал «Свободен» (*S_proc_free*), тем самым выходит из пула свободных. Процессор назначен, и далее диспетчер переходит к непосредственной передаче номера задачи процессору. Для этого он извлекает номер задачи из внутреннего регистра, выставляет его на шину (*V_task_id*), и передает в процессор сигнал «Процесс готов» (*S_task_ready*). Процессор воспринимает активный уровень, что гарантирует, достоверность данных на шине(*V_task_id*). Процессор фиксирует эти данные в своем внутреннем регистре, и когда его работа с шиной будет окончена, он выставляет сигнал «Процесс принят» (*S_proc_prinyal*).

«Серверный» процесс реализован следующим образом: свободный процессор, находится в спящем режиме. Сигнал «Свободен» (*S_proc_free*),

поступающий от процессора к диспетчеру принимает в этом случае активный уровень. Если в очереди имеются готовые задачи, выбирается обслуживающий процессор в соответствии с приоритетом, причем главный приоритет в данном случае присвоен процессору с номером 1. Выбранный процессор выходит из спящего режима, выбирает задачу из начала очереди (*S_fifo_read*), получает идентификатор задачи по шине (*V_task_id*), выставляет сигнал «Процесс принял» (*S_proc_prinyal*) и переходит в режим «Занято».

После этого планировщик может считать данные с шины и перейти к назначению следующей задачи. Процессор, получивший номер задачи переходит к её обслуживанию. Все сигналы интерфейса сняты. По завершению работы процессор известит диспетчер о том, что свободен, установлением активного уровня сигнала «Свободен» (*S_proc_free*), тем самым снова окажется в пуле свободных процессоров, и диспетчер вновь будет учитывать его при распределении задач.

Функционально устройство планирования задач представлено в виде четырех блоков (Рисунок 4):

- блок управления очередью задач;
- блок очереди задач *FIFO*;
- блок выбора процессора для обслуживания очередной задачи;
- блок синхронизации и назначения задач по процессорам.

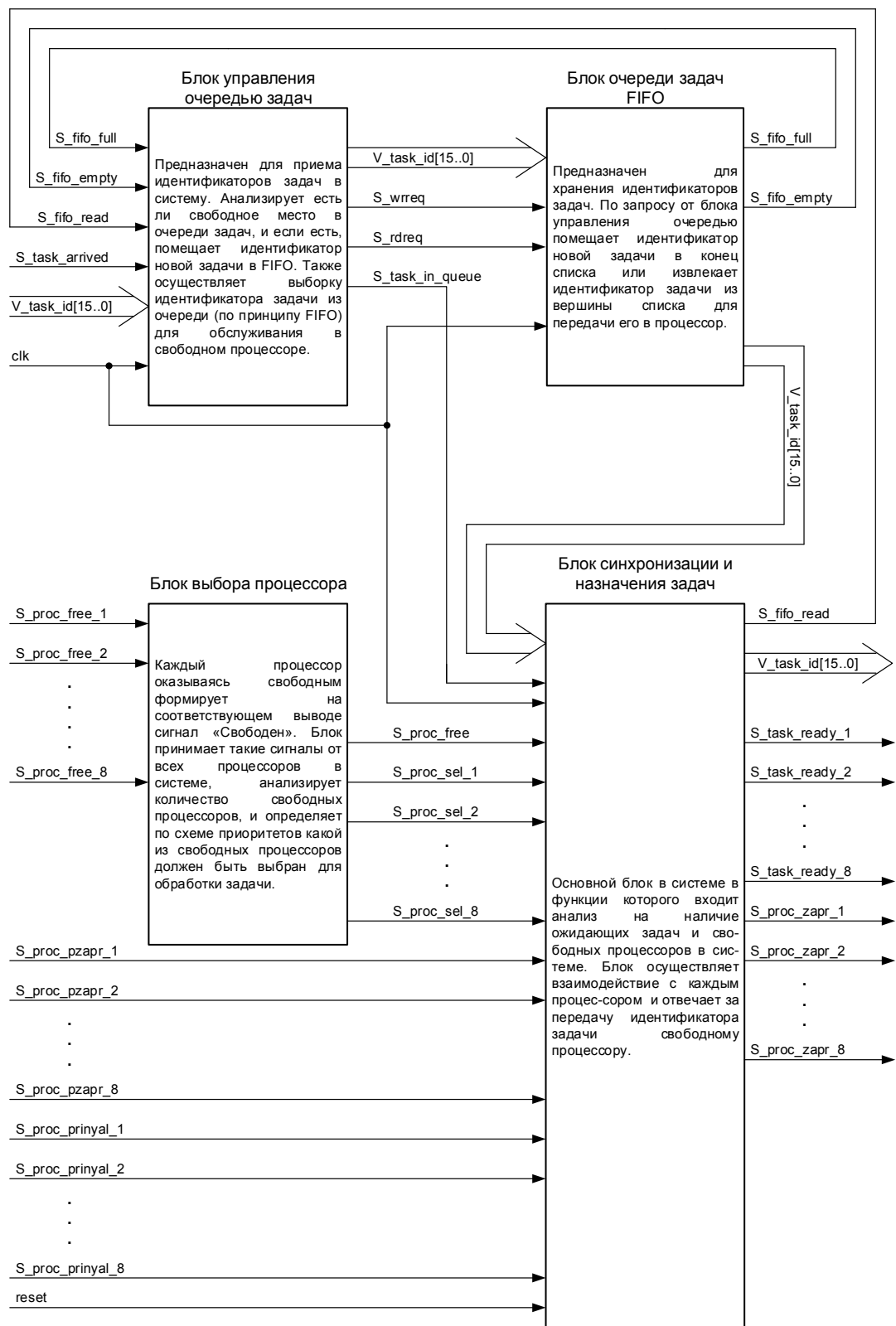


Рисунок 4. Структура устройства диспетчеризации задач.

2.1. Блок управления очередью

Блок управления очередью задач предназначен для приема идентификаторов задач в систему. Он производит анализ на наличие свободных мест в очереди задач, и далее, помещает идентификатор новой задачи в блок очереди *FIFO*. Также осуществляет выборку идентификатора задачи из очереди по принципу *FIFO*.

Таким образом, блок формирует очередь задач и отслеживает её состояние, используя для этого два сигнала. Активный уровень сигнала «*FIFO* заполнено» (*S_fifo_full*), который сообщает планировщику, что принимать новые задачи запрещено, так как нет свободных регистров для записи и хранения новых идентификаторов.

Ситуация с поступлением новых задач выглядит следующим образом. Если в очереди нет места для записи новых идентификаторов задач, то они планировщиком не воспринимаются. Второй сигнал – «*FIFO* пустое» (*S_fifo_empty*) указывает на то, что ни один идентификатор в очередь не поставлен. О наличии задачи в очереди свидетельствует сигнал *S_task_in_queue*. На его основе делается решение о том, выводить из спящего режима процессор или нет. Если очередь задач пуста, то блок синхронизации не обращается к процессорам. Сигналы «*FIFO* заполнено» (*S_fifo_full*) и «*FIFO* пустое» (*S_fifo_empty*) формируются и поступают в блок управления очередью задач из блока *FIFO*.

В блоке управления очередью задач имеется шина данных (*V_task_id*), по которой передаются идентификаторы задач. О том что, на шине присутствует новая процесс, планировщика извещает нарастающий фронт сигнала «Поступил процесс» (*S_task_arrived*). Если очередь не заполнена, то планировщик принимает данные с шины *V_task_id* и записывает их в очередь *FIFO*, в конец списка. Если очередь *FIFO* заполнена, данные на входной шине игнорируются. Запрос очередной задачи выполняет один из

процессоров, выбранный в качестве обслуживающего. Для этих целей используется сигнал «Запрос задачи» (*S_fifo_read_j*). На основании этого сигнала блок диспетчеризации формирует сигнал «Чтение *FIFO*» (*S_fifo_read*) и передает его блоку управления очередью задач.

Планировщик управляет блоком очереди задач *FIFO*, пользуясь сигналами *S_wrreq* и *S_rdreq*. *S_wrreq* используется для записи в очередь *FIFO*, а *S_rdreq* для чтения из неё.

2.2. Блок очереди задач

Блок очереди задач *FIFO* хранит поступающие идентификаторы задач. Идентификатор является частью контекста задачи, которое может состоять из нескольких полей, одно из которых указывает на адрес задачи в памяти многопроцессорной системы. По запросу от блока управления очередью заносит идентификатор новой задачи в конец списка или извлекает идентификатор запускаемой задачи из вершины списка для передачи его в процессор.

2.3. Блок выбора процессора

Блок выбора процессора принимает все сигналы «Свободен» (*S_proc_free*) от всех процессоров в системе. Он определяет, присутствуют ли в данный момент в системе свободные процессоры. Если в наличии имеется хотя бы один процессор, не занятый обслуживанием текущей задачи, то этот блок устанавливает активный уровень сигнала «Свободный процессор» (*S_proc_free*) и таким образом извещает блок управления назначением задач о том, что имеются не занятые процессоры.

2.4. Блок синхронизации и назначения задач

Блок синхронизации и назначения задач анализирует информацию о наличии ожидающих задач в очереди и свободных процессоров, назначаемых для обработки этих задач. Данный блок, используя механизм синхронизации «рандеву», осуществляет взаимодействие очереди задач с процессорами по описанному ранее интерфейсу. Блок выбора процессора формирует информацию о том, какой из свободных процессоров выбран для обслуживания текущей задачи. Идентификатор принятой на обслуживание задачи передается блоком очереди задач *FIFO* по шине *V_task_id*. Кроме того, блок управления очередью задач сообщает о состоянии очереди и осуществляет выборку идентификатора задач из блока очереди *FIFO* по требованию блока синхронизации и назначения задач.

3 ОПИСАНИЕ РАЗРАБОТКИ АЛГОРИТМА

3.1 Выбор подхода для реализации алгоритма

Существует два подхода к аппаратно-ориентированной реализации алгоритмов управления процессами и ресурсами в многопроцессорных (многоядерных) системах: микропрограммная и схемная.

Микропрограммный способ предполагает внесение обоснованно выбранных функций управления процессами и ресурсами в систему команд целевого процессора. В настоящее время микропрограммно реализуют неизменяемые функции ОС, которые способны работать в пользовательском пространстве и могут вызываться выполняющимся приложением. Микропрограммно реализуют примитивы типа «тестирование и блокировка», «выборка и инкремент», «выборка и добавление» и др. Они позволяют выполнять синхронизацию процессов методом активного ожидания (спин-блокировки) в пространстве пользователя. В некоторых языках программирования микропрограммно реализуют такие высокоуровневые механизмы синхронизации как, например, мониторы (в языке Java). Функции же управления задачами (планировщики, диспетчеры) чаще вызываются автоматически по таймеру и реализуются как процедура, выполняющаяся в пространстве ядра. Обычно такие функции требуют перенастройки для оптимизации вычислительного процесса под конкретный класс прикладных задач, что затрудняет использование микропрограммного подхода.

Схемная реализация заключается в перенесении компонент ядра операционной системы на уровень устройства управления или управляющего спецпроцессора. Схемная реализация позволяет полностью освободить ядро ОС от функций управления переключением задач и прерываниями, синхронизации процессов, обмена сообщениями и т.д. В результате API-функции, выполняемые в спецпроцессоре, значительно ускоряются, а

тщательная отладка аппаратуры современными средствами проектирования ПЛИС гарантирует надежность и безопасность операционной системы. Такой подход обеспечивает повышение производительности многопроцессорной системы за счет:

- 1) Быстрого выполнения функций ОС по управлению процессами и ресурсами в аппаратных средствах;
- 2) Освобождения центральных процессоров от функций ОС по управлению процессами и ресурсами, что позволяет полностью занять их только выполнением приложений.

Архитектура аппаратного ядра многопроцессорной операционной системы

Возможны три способа включения аппаратного ядра ОС в многопроцессорную систему: 1) через системную общую шину (Рисунок 5); 2) через отдельную (специальную, периферийную) шину (Рисунок 6), 3) с непосредственными связями (Рисунок 7).

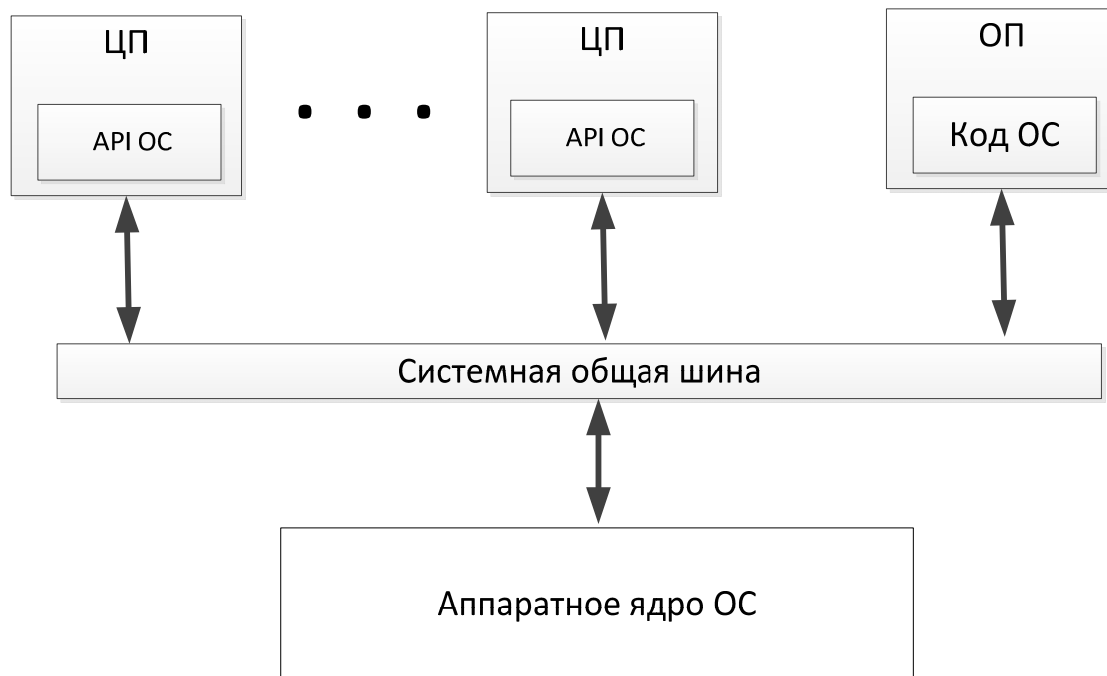


Рисунок 5. Включение аппаратного ядра через общую шину.

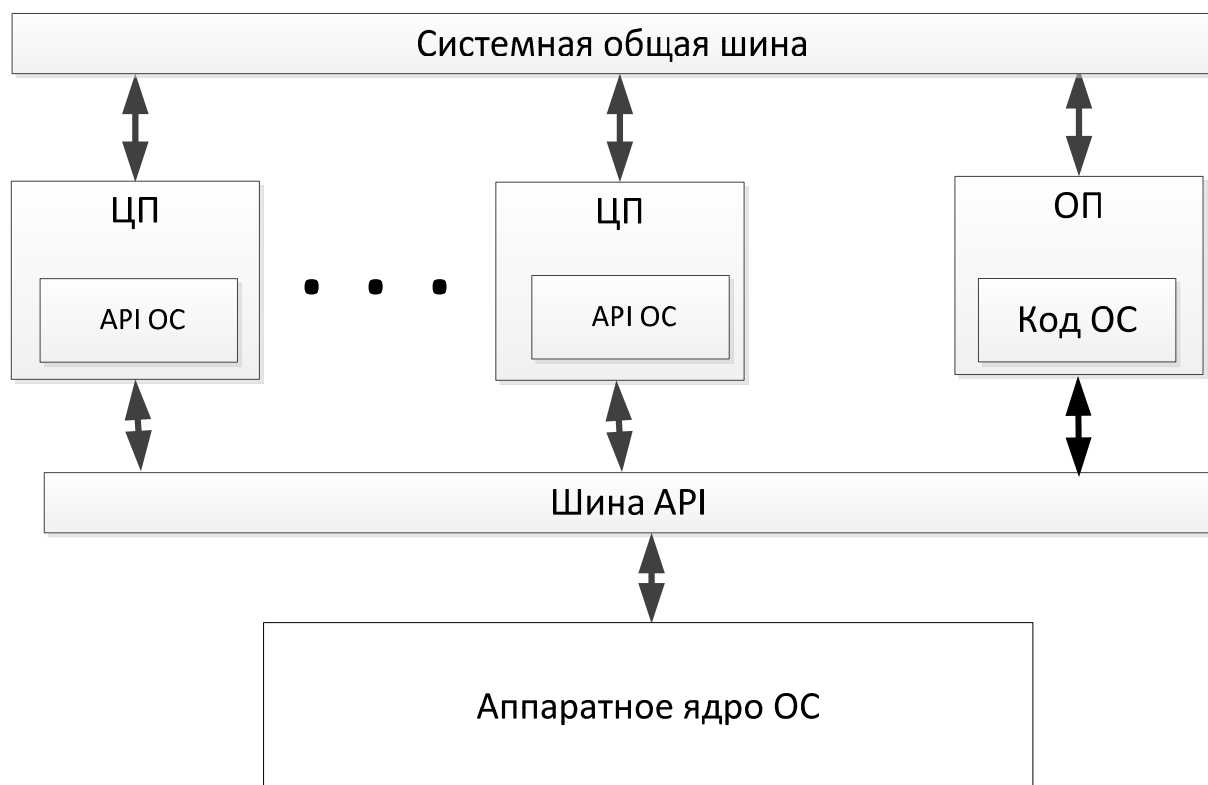


Рисунок 6. Включение аппаратного ядра через периферийную шину

б)

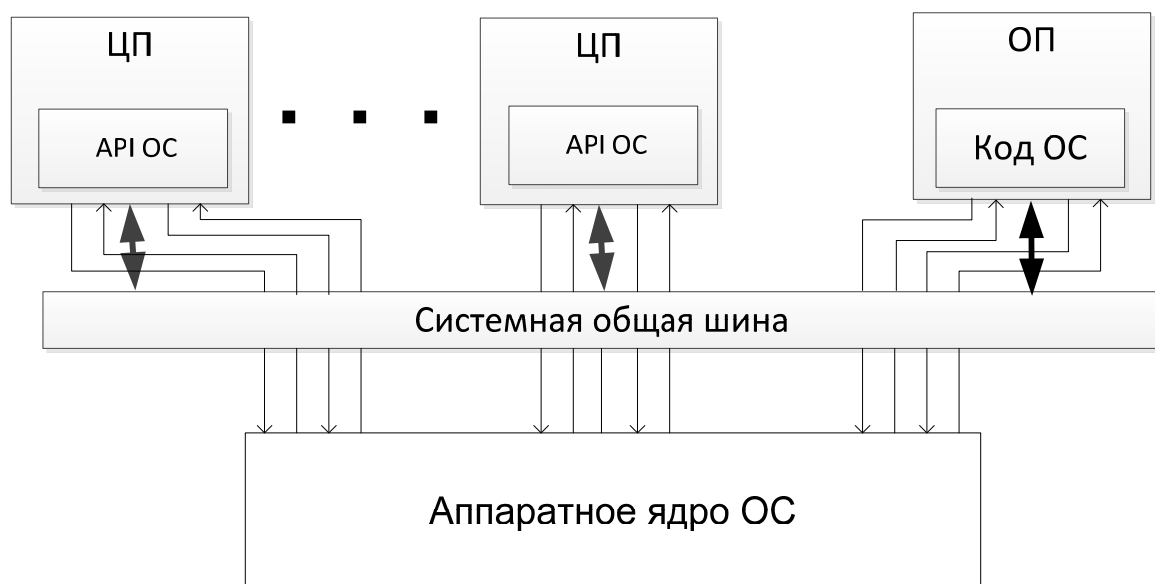


Рисунок 7. Включение аппаратного ядра с непосредственными связями

Первый вариант структурной схемы использует единственную общую шину для взаимодействия между процессорами, памятью системы, а также между API (Application Programming Interface) операционной системы (API ОС) и аппаратным ядром. API ОС является частью операционной системы, обеспечивающей программный интерфейс между выполняющимися приложениями через ядро ОС. API определяет функциональность, которую предоставляет программа операционной системы, при этом API позволяет абстрагироваться от того, как именно эта функциональность реализована. САМОЙ ВАЖНОЙ частью API операционных систем является множество системных вызовов. Положительной стороной такой архитектуры является то, что единый интерфейс дает возможность использования типовых средств (стандартного интерфейса) для включения аппаратного ядра в многопроцессорную систему. Однако она имеет недостаток, заключающийся в последовательном характере взаимодействия через общую шину, поскольку только два устройства могут связываться на любом промежутке времени. Это обстоятельство не обеспечивает параллелизм связей, что может создать высокий трафик, приводящий к перегрузке общей шины.

В архитектуре отдельной (специальной) шины взаимодействие процессоров с модулем аппаратного ядра производится по специальной шине API. В такой архитектуре системная шина не нагружена дополнительным трафиком от команд и данных операционной системы, что позволяет увеличить общую производительность многопроцессорной системы.

В третьем варианте между API процессора и аппаратным ядром ОС устанавливается непосредственная связь с помощью управляющих сигналов. Для этого в составе API каждого процессора включается два дополнительных блока. Первый - для декодирования команд, преобразующий команды системного вызова в сигналы управления и передачи их в аппаратное ядро. Второй – для приема сигналов, поступающих из ядра. Этот вариант наиболее производительный, поскольку несколько процессоров одновременно могут

взаимодействовать с ядром. Однако для его реализации необходимы дополнительные входы/выходы, что приводит к увеличению числа внешних выводов процессора.

Устройство управления процессами и ресурсами (УУПР) составляет аппаратное ядро операционной системы и состоит из трех основных частей: интерфейса ядра, блока управления процессами (БУП) и блока управления ресурсами (БУР). Интерфейс ядра состоит из логики, необходимой для декодирования инструкций АРІОС и выполнения всех необходимых обменных взаимодействий.

Первый вариант использует единственную общую шину для взаимодействия между блоками УУПР, а также между АРІ ОС и аппаратным ядром ОС. Положительной стороной является то, что единственный тип интерфейса позволяет использовать типовые средства для подключения аппаратного ядра в многопроцессорную систему. Такая архитектура имеет недостаток, вызванный последовательным характером взаимодействия через общую шину, что не способствует высокой производительности аппаратного ядра. Второй вариант реализации заключается в том, что взаимодействие внутренних модулей выполняется путем создания непосредственных связей. Такая архитектура связей имеет значительно более высокую пропускную способность, однако сложность её реализации несколько выше остальных.

Архитектура характеризуется совместным использованием модулей памяти и регистров, принадлежащих каждому блоку БУП и БУР. Идея состоит в том, что каждый блок владеет рядом регистров и возможно модулей памяти, в которые они пишут и читают данные, причем каждый блок также в состоянии считывать данные из регистров и памяти других блоков, поскольку они имеют единое адресное пространство. Таким образом, интерфейс каждого блока относительно простой, состоит только из регистров и логики доступа к памяти. Это обеспечивает единственную и довольно простую интерфейсную архитектуру, которую будет легче поддерживать, чем несколько различных

интерфейсов. API ОС может связаться с аппаратным ядром путем записи в соответствующие регистры. Все модули ядра могут связаться друг с другом, поскольку, регистры и сегменты памяти поддерживают несколько читателей и писателей. Архитектура общей памяти обеспечивает параллелизм в связях как между блоками ядра, так и между аппаратурой ядра и API ОС. В то же время блоки ядра могут получать доступ только к частям регистров и памяти каждого из других модулей, а API-интерфейс в состоянии получать доступ ко всей карте распределения памяти.

Главными модулями блока БУП (Рисунок 5): таймер, сторожевой таймер, диспетчер и планировщик задач. В его состав входит также таблица процессов, являющийся RAM-памятью для хранения дескрипторов готовых к выполнению процессов, которые содержат необходимые данные для управления сохранением и восстановлением контекстов.

БУР содержит в своём составе средства управления ресурсами (механизмы синхронизации процессов), которые обеспечивают взаимодействие параллельных процессов при доступе к общим (разделяемым) ресурсам, основанное на критических интервалах: мьютексы, бинарные и счетные семафоры, очереди сообщений. Этот блок может включать и другие механизмы синхронизации, например, флаги событий, почтовые ящики и т.д.

В последующих разделах будут рассмотрены вопросы структурной реализации аппаратного ядра многопроцессорных операционных систем с использованием средств функционального моделирования языке. При проектировании устройств аппаратной поддержки важен выбор математического аппарата для формализации, обеспечивающего простоту описания, верификации и структурной реализации алгоритмов управления процессами и ресурсами. Наиболее подходящим аппаратом, на наш взгляд, является язык логики недетерминированных автоматов. Использование этого языка позволило разработать НДА- модели алгоритмов взаимодействия параллельных процессов, основанные на критических интервалах (модуль

мьтексов), и модели очередей сообщений (модуль каналов). Из средств управления процессами разработаны модули диспетчера и частично планировщика задач, реализация которых показана в приведенных ниже разделах.

3.1 Разработка модулей алгоритма

Для удобства процесса разработки подсистема будет разделена на четыре модуля, один из которых будет разделен на четыре файла – первый, содержащий описание блока управления очередью задач, второй, содержащий IP Core файл, который проверяет параметры генерируемого ядра, третий, который содержит описание блока синхронизации задач и четвертый, который содержит описание блока выбора свободного процессора. Один из модулей – это модуль квазипроцессора, который имитирует обработку принятого пакета. Другой модуль содержит в себе счетчик, имитирующий наполнение пакетов.

В файле `queue_controller.vhd` будет реализован блок управления очередью задач, который предназначен для приема идентификаторов задач в систему. Он производит анализ на наличие свободных мест в очереди задач, если они есть, помещает идентификатор новой задачи в блок очереди *FIFO*. Также осуществляет выборку идентификатора задачи из очереди по принципу «первым пришел – первым обслужился» и помещает его в свободный процессор.

В файле `proc_selector.vhd` будет реализован блок выбора процессора, который определяет, присутствуют ли в данный момент в системе свободные процессоры.

В файле `distributor.vhd` будет реализован блок синхронизации и назначения задач, который анализирует информацию о наличии ожидающих

задач в очереди и свободных процессоров, назначаемых для обработки этих задач.

В файле `processor.vhd` будет реализован блок процессора. Кроме генератора задач дополнительно был разработан блок имитации работы процессора, который полностью поддерживает интерфейс «процессор-диспетчер».

В файле `task_spawner.vhd` будет реализован блок генерации задач. В процессе моделирования необходимо задавать диспетчеру идентификаторы задач, готовых к выполнению. Задачи поступают в систему постоянно, и их количество может быть очень большим.

Общий вид структурной схемы модели диспетчера задач в составе восьмипроцессорной системы представлен на рисунке ниже (Рисунок 8)

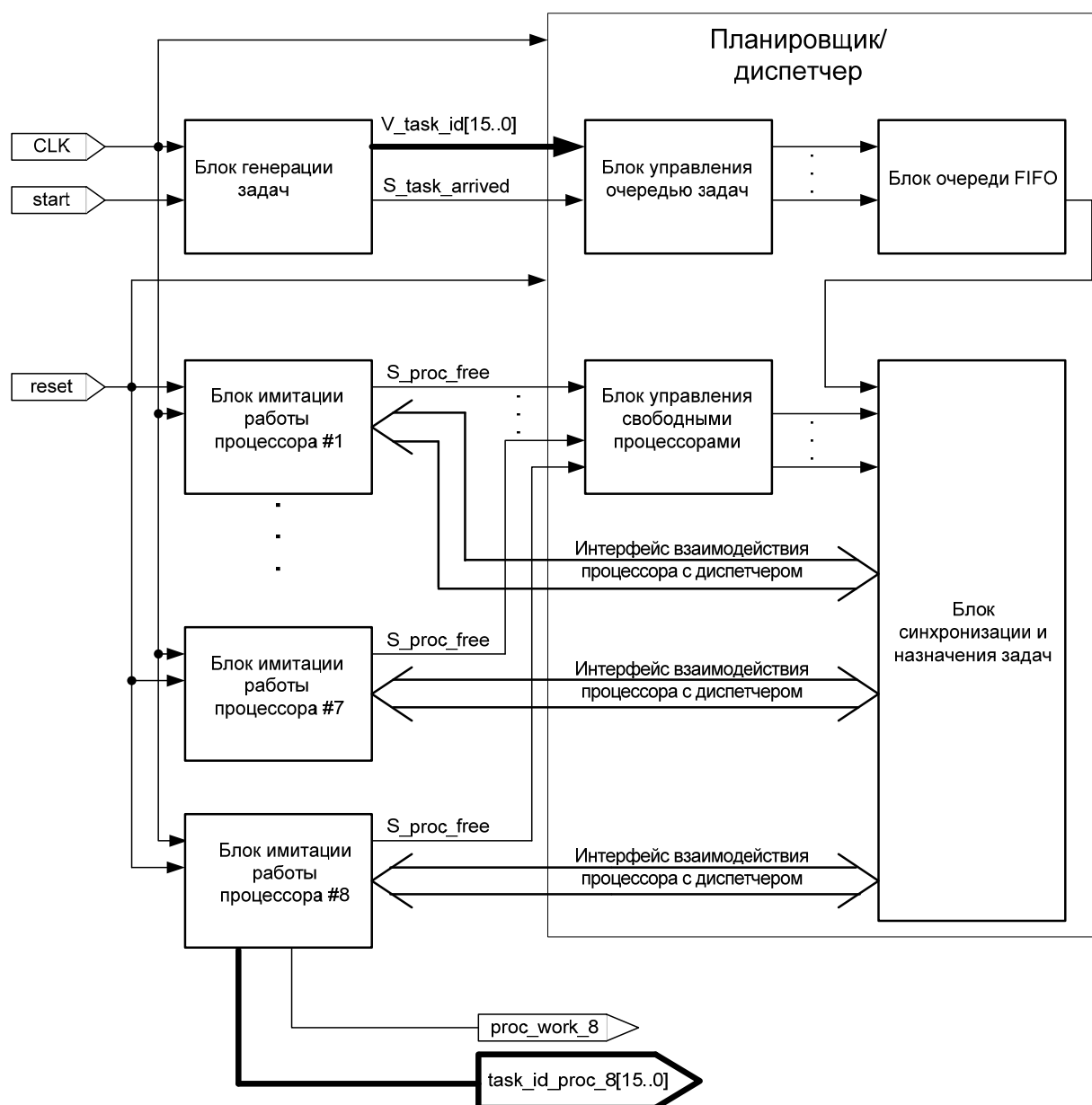


Рисунок 8 – Структурная схема модели деспетчера задач

При выполнении моделирования алгоритма практически все сигналы генерируются самостоятельно, только необходимо будет подать синхросигналы, от которого тактируются все блоки, сигналы начального сброса системы и запуска генератора задач.

3. ФОРМАЛЬНОЕ ОПИСАНИЕ РЕАЛИЗАЦИИ АЛГОРИТМА ПЛАНИРОВЩИКА

В самом общем виде алгоритм планирования задач выглядит следующим образом.

В начальном состоянии каждый включающийся в работу процессор выполняет операцию «готов» и переходит в спящий режим, в котором находится до тех пор, пока на обслуживание не поступит задача из очереди. Если все процессоры заняты обслуживанием, то вновь поступившая задача помещается в конец очереди, число мест в которой ограничено. Если вновь поступившая задача обнаружит, что очередь заполнена, она на обслуживание не принимается и покидает систему. Принятая на обслуживание задача находится в очереди до тех пор, пока не поступит на выполнение в процессор, при этом в очереди освобождается одно место. После выполнения очередной задачи планировщик/ диспетчер просматривает очередь, и если в ней имеются ожидающие, то он назначает на выполнение в свободный процессор ту, которая стоит в голове списка (по принципу *FIFO*). Если очередь пуста, то свободные процессоры переходят в режим ожидания. Если в процессорном пуле имеется несколько свободных процессоров, то производится приоритетный выбор одного из них для обслуживания очередной задачи.

В общем случае алгоритм планирования/диспетчеризации в явном виде связан с взаимодействием процессов. С одной стороны задачи (процессы, потоки), требующие своего выполнения, с другой стороны - обслуживающие их процессорные узлы. Эти действия необходимо синхронизировать таким образом, чтобы обеспечить так называемое «рандеву», когда j -й процессорный узел должен дожидаться поступления задачи, а i -й процесс – освобождения одного из процессоров, после чего она будет обслуживаться в течение некоторого времени.

Традиционный подход, связанный с таким обслуживанием аналогичен задаче «спящего парикмахера», который широко применялся для вычислительных систем с одним процессором. Для многопроцессорных систем такой подход должен быть расширен до задачи, которую можно назвать «работа парикмахерской». Эта процесс иллюстрирует отношения «клиент-сервер», которые имеют место между процессами в многопроцессорных вычислительных системах. Причем аналогом процессоров являются коллектив парикмахеров, а аналогом задач выступают клиенты парикмахерской.

Формализация алгоритмов управления параллельными процессами выполнена на основе использования механизма «рандеву». Весь алгоритм работы диспетчера содержит три части: клиентскую (постановка задачи в очередь), серверную (обслуживание процессорами) и «рандеву» (наличие задачи и готовность одного из процессоров к обслуживанию этой задачи). Для описания алгоритма введены основные частные события, представленные в таблице 1. В последней колонке представлено их соответствие сигналам на схеме устройства диспетчеризации задач.

Таблица 1. Отмеченная таблица событий

События	Описание частного события	Сигналы на схемах
S_I^t	Ожидание занесения задачи в очередь	$S_task_arrived$
S_{FQ}^t	В очереди имеются свободные места	$\overline{S_fifo_full}$
S_Q^t	Задача в очереди	$S_task_in_queue$
S_{ZPj}^t	Запрос j -го процессора диспетчером	$S_proc_zapr_j$
S_{GPj}^t	Задача готова к обслуживанию в j -м процессоре,	$S_task_ready_j$

	j -й процессор помещается в пул занятых	
S_{PZ}^{pj}	j -й процессор выдал сигнал подтверждение запроса	$S_{proc_pzapr_j}$

Таблица 1. Продолжение

S_S^{pj}	j -й процессор помещается в пул свободных	$S_{proc_free_j}$
S_S^p	В процессорном пуле имеются свободные процессоры	S_{proc_free}
S_{PR}^{pj}	Приоритет j -го процессора	Внутренний сигнал, на схеме не обозначен
S_{PT}^{pj}	j -й процессор задачу принял	$S_{proc_prinyal_j}$
S_{SL}^{pj}	j -й процессор выбран для обслуживания задачи	$S_{proc_sel_j}$
S_A^{pj}	j -й процессор выполняет обслуживание задачи	Аппаратно не реализовано
S_{OF}^t	Удалить задачу из очереди	Внутренний сигнал, на схеме не обозначен
S_{TO}^{pj}	Снять задачу с исполнения	Внутренний сигнал, на схеме не обозначен
S_E^{pj}	Выполнение задачи закончено	Внутренний сигнал, на схеме не обозначен
S_{RT}^t	Выдача результата выполнения текущей задачи	Внутренний сигнал, на схеме не обозначен
S_O^{pj}	j -й процессор находится в режиме ожидания	Внутренний сигнал, на схеме не обозначен

На основании словесно представленного алгоритма управления процессами и введенных событий, реализуемых в этом алгоритме, система канонических уравнений, описывающих эти события, будет иметь следующий вид: - для процесса «клиент», реализуемого диспетчером до момента randevu:

$$S_I^t(t+1) = x_z \vee \overline{S_I^t S_{FQ}^t}$$

$$S_Q^t(t+1) = S_I^t S_{FQ}^t \vee S_Q^t \overline{S_{SL}^{pj}};$$

$$S_{ZPj}^t(t+1) = S_Q^t S_{SL}^{pj} \vee S_{ZPj}^t \overline{S_{PZ}^{pj}};$$

$$S_{GPj}^t(t+1) = S_{ZPj}^t S_{PZ}^{pj} \vee S_{GPj}^t \overline{S_{PT}^{pj}},$$

где Xz - сигнал запроса задачи на вхождение в очередь.

- для процесса «сервер», реализуемого процессором до момента рандеву:

$$S_S^P(t+1) = (S_S^{p1} \vee S_S^{p2} \vee \dots \vee S_S^{pN}) \vee S_S^P \overline{S_{ZP}^t}$$

$$S_O^{pj}(t+1) = S_{ZP}^t \vee S_O^{pj} \overline{S_{SL}^{pj}}$$

$$S_{SL}^{pj}(t+1) = \bigvee_{\forall j \in N} S_O^{pj} S_{PR}^{pj}$$

$$S_{PZ}^{pj}(t+1) = S_{SL}^{pj} S_{ZPj}^t \vee S_{PZ}^{pj} \overline{S_{GPj}^t}; \quad (2)$$

$$S_{PT}^{pj}(t+1) = S_{PZ}^{pj} S_{GPj}^t.$$

Система канонических уравнений, описывающая события после рандеву:

$$S_A^{pj}(t+1) = S_{PT}^{pj} S_{GPj}^t \vee S_A^{pj} \overline{S_E^{pj}};$$

$$S_{TO}^{pj}(t+1) = S_A^{pj} S_E^{pj} \vee S_{TO}^{pj} \overline{S_{RT}^t};$$

$$S_{RT}^t(t+1) = S_A^{pj} S_E^{pj} \vee S_{RT}^t \overline{S_{TO}^{pj}};$$

$$S_{OF}^t(t+1) = S_{TO}^{pj} S_{RT}^t;$$

$$S_S^{pj}(t+1) = S_{TO}^{pj} S_{RT}^t \vee S_S^{pj} \overline{S_{GPj}^t}.$$

Уравнениям соответствует граф недетерминированного автомата (НДА). На Рисунке 9 Рисунок представлены графы выбора j -го процессора для обслуживания очередной задачи, клиентской и серверной частей алгоритма управления взаимодействующими процессами до момента «рандеву» (до оператора объединения $J(\&)$), и часть графа после «рандеву».

Следует иметь в виду, что в данном (аппаратно-ориентированном) алгоритме функция назначения задач выполняется диспетчером задач (клиентом). Он определяет наличие задачи в очереди, наличие свободных процессоров в пуле, формирует сигнал запроса процессора, дожидается ответный сигнал от процессорного блока (сервера), при получении которого возникает условие (рандеву), при котором задача направляется на выполнение в выбранный процессор.

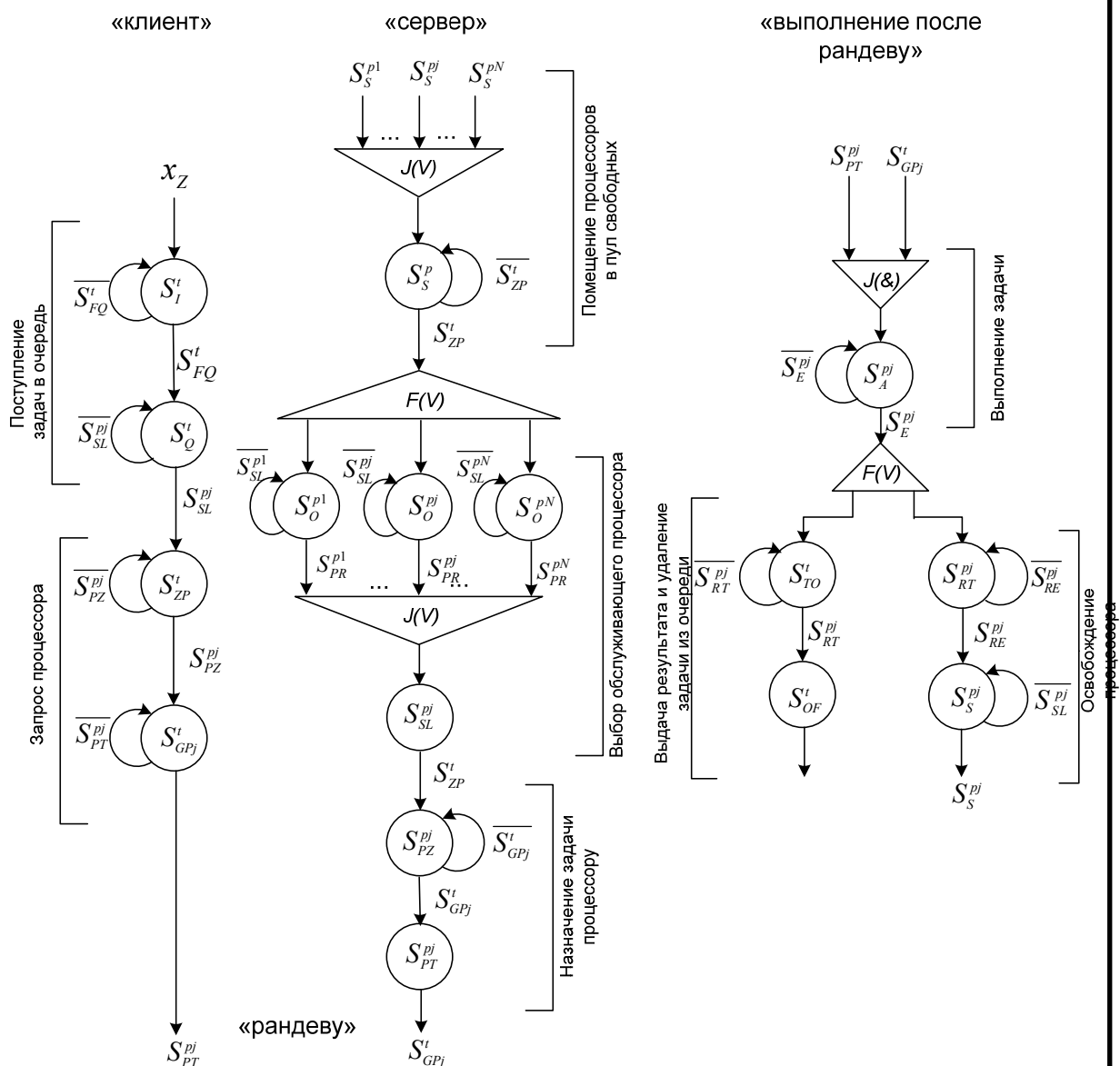


Рисунок 9. Граф НДА алгоритма планирования/диспетчеризации задач в многопроцессорной системе

Представленная система канонических уравнений и граф НДА алгоритма планирования/диспетчеризации с глобальной очередью задач отличается тем, что здесь имеет место процедура выбора процессора из всех находящихся в режиме ожидания. Следует иметь в виду, что функция $F(V)$ является разветвительной вершиной, за которой следует выполнение параллельных процессов по обеспечению состояния ожидания процессоров, входящих в пул свободных.

4. МОДЕЛИРОВАНИЕ ПЛАНИРОВЩИКА НА ЯЗЫКЕ VHDL И АНАЛИЗ РЕЗУЛЬТАТОВ

Модель устройства планировщика была реализована на языке *VHDL* в виде четырех программных модулей, описанных в разделе [2].

Первый модуль реализует логические функции первого блока – управление очередью задач. Второй модуль – это блок FIFO. Третий модуль – блок управления свободными процессорами, четвертый модуль – блок синхронизации.

Ввод схем и работа производилась в свободно распространяемой версии системы *Xilinx ISE WebPACK Design Suite 14.7*, моделирование производилось в той же системе.

Ниже рассмотрены ситуации, иллюстрирующие работу планировщика и вспомогательных компонентов.

На рисунке 10 показана работа основной части диспетчера — синхронизатора задач (компонент *distribution_unit.vhd*).

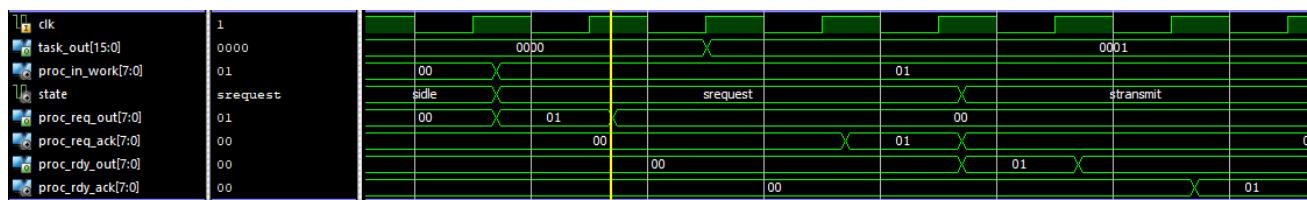


Рисунок 10. Работа синхронизатора.

На входе синхронизатора следующие сигналы:

- ПакетизFIFOtask_out (здесь: идентификатор задачи «0001»);
- Выбранный свободный процессор proc_in_work (здесь: первый).

Далее рассмотрим конечный автомат.

- 1) IDLE. Пребывающий в состоянии ожидания синхронизатор получает пакет и проверяет наличие свободных процессоров. В случае наличия такового осуществляется переход в состояние request.
- 2) REQUEST. Начинается процесс передачи идентификатора процессору:

- выставляется запрос на обработку пакета (proc_req_out);
- ожидается ответ (proc_req_ack). При получении ответа выполняется переход в состояние transmit.

3) TRANSMIT. Процессору выставляется строб валидности данных (proc_rdy_out), ожидается ответ (proc_rdy_ack). Автомат переходит в request (если есть необработанные задачи и свободные процессоры) или idle.

4.1. Производительность диспетчера

Задержка диспетчера при назначении задачи зависит от того, насколько быстро процессор будет присылать ответы на запросы (диспетчер не будет ожидать ответ дольше, чем указано в регистре timeout в описании синхронизатора).

Минимальное время обработки задачи составляет 7 тактов (Рисунок 11). Отправка запроса на согласование от диспетчера к процессору – такт. При получении запроса входные цепи процессора (триггер) создают задержку в один такт. На третий такт приходит ответ от процессора, который диспетчер воспринимает на четвертый такт (также после прохождения через входной триггер). В следующем такте диспетчер выдает строб готовности, который достигает конечного автомата процессора на шестом такте (через триггер). К седьмому такту диспетчер получает сообщение о том, что процессор успешно считал данные.

Оптимизация времени работы диспетчера может быть достигнута модификацией условия выдачи строба ready_out к процессору. Дальнейшая оптимизация может быть достигнута изменением способа формирования запроса req_out, выработкой сигнала proc_in_work комбинационной схемой, а не по синхросигналу (стабильность работы диспетчера при таком сценарии не проверялась).

Наиболее очевидной оптимизацией времени работы диспетчера является отказ от алгоритма согласования, используемого между диспетчером и процессором.

На Рисунке 11 показана работа процессоров proc0 и proc1 после получения ими задач от диспетчера.



Рисунок11. Результаты симуляции.

Регистр task_received содержит задачу, которую процессор будет обрабатывать.

Нормальный режим работы.

Разрядность счетчика 3 (максимальная задержка 7 тактов). По данным шины proc_free_in видно, что в обработке принимают участие только первые три-четыре процессора (т.к. обработка потока задач осуществляется достаточно быстро) (Рисунок 12).

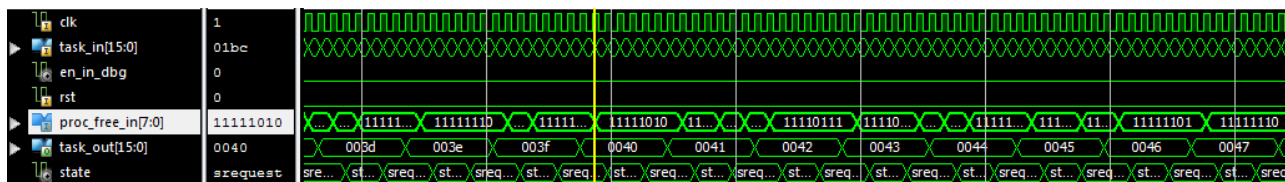


Рисунок 12. Нормальный режим работы.

4.1.1. Перегрузка (все процессора заняты).

Обратим внимание на регистр `proc_free_in` = 0. Это означает, что в данный момент времени все процессоры заняты. Перегрузка была достигнута путем установки в константу разрядность счетчика 5 бит (максимальная задержка 31 такт). Также усилен поток пакетов в глобальной очереди: 128 пакетов подряд. Отметим, что, если пакеты продолжают поступать, а процессоры и FIFO заняты, то приходящие пакеты не будут приняты на обработку (Рисунок 13).

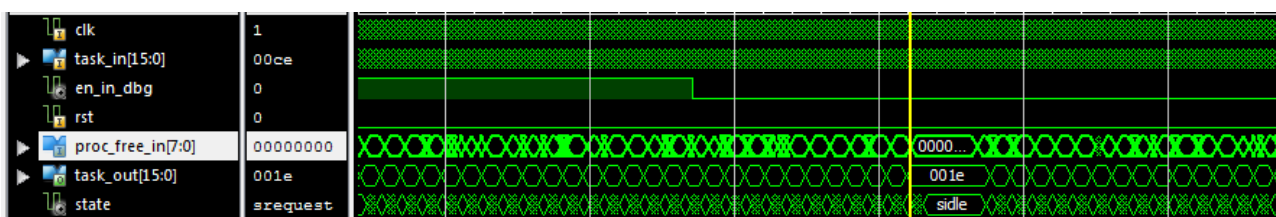


Рисунок 13. Перегрузка.

Если увеличить разрядность счетчика до 7 (максимальная задержка 127 тактов), то перегрузка станет заметнее: процессоры постоянно заняты, и лишь иногда один из них ненадолго освобождается (сигнал `proc_free_in`, Рисунок 11). Также на рисунке видна работа конечного автомата диспетчера. Можно заметить, что он переходит в состояние IDLE, когда все процессоры заняты.

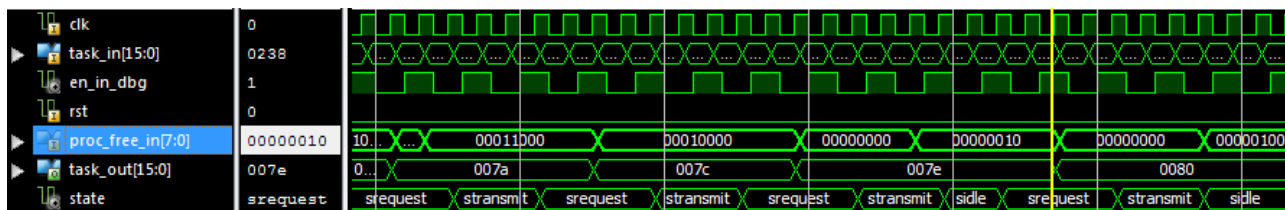


Рисунок 14. Сильная перегрузка.

ЗАКЛЮЧЕНИЕ

В результате выполнения представленной выпускной квалификационной работы был разработан аппаратный планировщик с глобальной очередью задач.

Данная технология является актуальной и широко применяется в настоящее время.

Модель устройства планировщика была реализована на языке VHDL в виде четырех программных модулей. Ввод схем и работа производились в свободно распространяемой версии системы Xilinx 14.7, моделирование производилось в той же системе. Для проведения моделирования использовались два дополнительных блока: блок генерации задач и блок имитации работы процессора. Модель была протестирована в различных режимах работы и показала высокие результаты по производительности. Полученное значение задержки назначения задачи при условии отсутствия очереди в процессорном узле не превышает одного такта.

Программа, описанная в этой выпускной квалификационной работе, удовлетворяет всем исходным требованиям и выполняет все заявленные функции.

Реализация аппаратного планировщика с глобальным разделением задач демонстрирует существенное повышение производительности многопроцессорной системы.

5. СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кнышев Д.А., Кузелин М.О. ПЛИС фирмы «XILINX»: описание структуры основных семейств. М.: Додэка-XX1,2001.
2. Кузелин М.О., Кнышев Д.А., Зотов В.Ю. Современные семейства ПЛИС фирмы Xilinx. М.: Горячая линия-Телеком,2004.
3. Мальцев П.П., Гарбузов Н.И., Шарапов А.П., Кнышев Д.А. Программируемые логические ИМС на КМОП-структурах и их применение. М.: Энергоатомиздат, 1998.
4. Тарасов И.Е. Разработка цифровых устройств на основе ПЛИС XILINX с применением языка VHDL. М.: Горячая линияТелеком, 2005.
5. The Programmable Logic Data Book. Xilinx Inc. 1999.
6. Бибило П. Н. Основы языка VHDL. М.: Солон-Р,2002.
7. Бибило П. Н. Синтез логических схем с использованием языка VHDL. М.: Солон-Р,2002.
8. Уэйкерли Дж. Ф. Проектирование цифровых устройств. В 2-х т. Т. 1 — 2. М.: Постмаркет, 2002.
9. Аппаратное обеспечение вычислительных систем. - М.: Маркет ДС, 2016. - 184 с.
10. Бикташев Р.А, Вашкевич Н.П, Меркурьев А.И. Аппаратная поддержка диспетчера с глобальной очередью задач в многопроцессорных системах. ПГУ, 2011-14 с.

6. ПРИЛОЖЕНИЕ А
ЛИСТИНГИ ПРОГРАММЫ

Файл test_bench.vhd

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_unsigned.all;

entity testbench is
end testbench;

architecture behavior of testbench is

    constant cTCQ           : time := 2 ns;
    constant clk_period     : time := 10 ns;

    signal clk               : std_logic;
    signal rst               : std_logic;

begin

    uut : entity WORK.L2_all( Behavioral )
    port map( clk           => clk,
              rst           => rst );

    clk_process : process
    begin
        clk <= '0';
        wait for clk_period / 2;
        clk <= '1';
        wait for clk_period / 2;
    end process;

    tb : process
    begin

        wait for 25 ns;
        rst <= '1' after cTCQ;
        wait for clk_period * 2;
        rst <= '0' after cTCQ;
        wait for 25 ns;

        wait;

    end process;

end;
```

Файл out.vhd

```

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;

entity L2_all is
    port (
        clk
        rst
    );
end L2_all;

architecture Behavioral of L2_all is

    -- queue scheduler <--> spawner
    signal task_spawner_to_scheduler : std_logic_vector( 15
downto 0 );
    signal enable_spawner_to_scheduler: std_logic;

    -- processor <--> scheduler
    signal task_to_proc
15 downto 0 );
    signal proc_free
7 downto 0 );
    signal task_rdy
std_logic_vector( 7 downto 0 );
    signal task_rdy_ack
7 downto 0 );
    signal task_req
std_logic_vector( 7 downto 0 );
    signal task_req_ack
7 downto 0 );

begin

    task_scheduler : entity WORK.L2_wrapper( Behavioral )
    port map( clk
        => clk,
        rst
        => rst,

        -- task spawner interface
        task_in
        =>
task_spawner_to_scheduler,
        task_en_in
        =>
enable_spawner_to_scheduler,

        -- processor interface
        proc_free_in
        => proc_free,

        task_out
        => task_to_proc,
        task_rdy_out
        => task_rdy,
        task_rdy_ack_in
        => task_rdy_ack,

```

```

        task_req_out      => task_req,
        task_req_ack_in   => task_req_ack );

processor_imitation : for i in 0 to 7 generate

    processor : entity WORK.processor( Behavioral )
    port map( clk          => clk,
              rst          => rst,

              -- scheduler interface
              task_in      =>

task_to_proc,

              proc_free_out      => proc_free(i),

              task_rdy_in        => task_rdy(i),
              task_rdy_ack_out   => task_rdy_ack(i),
              task_req_in        => task_req(i),
              task_req_ack_out   => task_req_ack(i)
    );

end generate;

id_spawner_imitation : entity WORK.task_spawner( Behavioral
)
port map( clk          => clk,
          rst          => rst,

          -- scheduler interface
          new_task_out  =>

task_spawner_to_scheduler,
          enable_out    =>
enable_spawner_to_scheduler );

end Behavioral;

```

Файл task_scheduler.vhd

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity L2_wrapper is
    Port(
        clk          : in  std_logic;
        rst          : in  std_logic;

        -- external interface
        task_in       : in  std_logic_vector(
15 downto 0 ); -- incoming tasks
        task_en_in    : in  std_logic;

        -- proc interface

```

```

proc_free_in      : in  std_logic_vector( 7
downto 0 );      -- '0' - busy, '1' - free

task_out          : out std_logic_vector( 15
downto 0 );
task_rdy_out      : out std_logic_vector( 7
downto 0 );      -- ready
task_rdy_ack_in   : in  std_logic_vector( 7
downto 0 );      -- pzapr
task_req_out      : out std_logic_vector( 7
downto 0 );      -- zapr
task_req_ack_in   : in  std_logic_vector( 7
downto 0 ));     -- prinyal

end L2_wrapper;

architecture Behavioral of L2_wrapper is

    -- queue controller <--> fifo
    signal task_queue_controller_to_fifo : std_logic_vector(
15 downto 0 );
    signal fifo_empty                    : std_logic;
    signal fifo_full                     : std_logic;
    signal fifo_write_request            : std_logic;
    signal fifo_read_request             : std_logic;

    -- queue controller <--> distributor
    signal fifo_read_from_distributor    : std_logic;
    signal tasks_in_queue                 : std_logic;

    -- fifo <--> distributor
    signal task_fifo_to_distributor      :
std_logic_vector( 15 downto 0 );

    -- selector <--> distributor
    signal proc_free                      : std_logic;
    signal proc_select                    :
std_logic_vector( 7 downto 0 );

begin

    queue_controller : entity WORK.task_queue_controller(
Behavioral )
    port map( clk          => clk,
               rst          => rst,

               -- external interface
               new_task      => task_in,
               new_task_en   => task_en_in,

               -- fifo interface

```

```

        fifo_empty      => fifo_empty,
        fifo_full       => fifo_full,
        task_to_fifo    =>
task_queue_controller_to_fifo,
        fifo_wr_en      => fifo_write_request,
        fifo_rd_en      => fifo_read_request,

        -- distributor interface
        distr_rd_req    => fifo_read_from_distributor,
        queue_busy      => tasks_in_queue );

    fifo : entity WORK.fifo_fwft
    port map( clk        => clk,
              rst        => rst,
              din        =>
task_queue_controller_to_fifo,
              wr_en      => fifo_write_request,
              rd_en      => fifo_read_request,
              dout       => task_fifo_to_distributor,
              full       => fifo_full,
              empty      => fifo_empty );

    proc_selector : entity WORK.processor_selector( Behavioral )
    port map( clk        => clk,
              rst        => rst,

              -- external interface
              proc_status_in => proc_free_in,

              -- selector <--> distributor
              proc_free_en  => proc_free,
              proc_selected => proc_select );

    distributor : entity WORK.distribution_unit( Behavioral )
    port map( clk        => clk,
              rst        => rst,

              -- fifo <--> distributor
              task_from_fifo    =>
task_fifo_to_distributor,

              -- selector <--> distributor
              any_proc_free    => proc_free,
              proc_selected    => proc_select,

              -- queue controller <--> distributor
              rd_req           =>
fifo_read_from_distributor,
              queue_busy      => tasks_in_queue,

              -- external

```

```

        task_out          => task_out,
        proc_rdy_out      => task_rdy_out,
        proc_rdy_ack_in   => task_rdy_ack_in,
        proc_req_out      => task_req_out,
        proc_req_ack_in   => task_req_ack_in
    );

end Behavioral;

```

Файл queue_controller.vhd

```

library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;

entity task_queue_controller is
    Port(
        clk          : in  std_logic;
        rst          : in  std_logic;

        -- external interface
        new_task      : in  std_logic_vector( 15 downto 0
    );
        new_task_en   : in  std_logic;

        -- fifo interface
        fifo_empty    : in  std_logic;
        fifo_full     : in  std_logic;
        task_to_fifo  : out std_logic_vector( 15 downto 0
    );
        fifo_wr_en    : out std_logic;
        fifo_rd_en    : out std_logic;

        -- distributor interface
        distr_rd_req   : in  std_logic;
        queue_busy    : out std_logic );

end task_queue_controller;

architecture Behavioral of task_queue_controller is

    constant cTCQ : time := 2 ns;

    signal task      : std_logic_vector( 15 downto 0 );
    signal task_en   : std_logic;

    signal f_wr_en : std_logic;
    signal f_rd_en : std_logic;

begin

    process( clk )

        -- input latch

```

```

begin
    if rising_edge( clk ) then
        task <= new_task after cTCQ;
        task_to_fifo <= task after cTCQ;
        task_en <= new_task_en after cTCQ;
    end if;
end process;

process( clk )
begin
    if rising_edge( clk ) then
        if rst = '1' then
            f_wr_en <= '0' after cTCQ;
        else
            if task_en = '1' then
                f_wr_en <= '1' after cTCQ;
            else
                f_wr_en <= '0' after cTCQ;
            end if;
        end if;
    end if;
end process;

fifo_wr_en <= f_wr_en;

process( clk )
begin
    if rising_edge( clk ) then
        if rst = '1' then
            f_rd_en <= '0' after cTCQ;
        else
            if fifo_empty = '0' and distr_rd_req = '1'
then
                f_rd_en <= '1' after cTCQ;
            else
                f_rd_en <= '0' after cTCQ;
            end if;
        end if;
    end if;
end process;

fifo_rd_en <= f_rd_en;

process( clk )
begin
    if rising_edge( clk ) then
        if rst = '1' then
            queue_busy <= '0' after cTCQ;
        else
            if fifo_empty = '0' then

```

```

        queue_busy      <= '1' after cTCQ;
    else
        queue_busy      <= '0' after cTCQ;
    end if;
end if;
end if;
end process;

end Behavioral;

Файл proc_selector.vhd
library IEEE;
    use IEEE.STD_LOGIC_1164.ALL;

entity processor_selector is
    port(
        clk              : in  std_logic;
        rst              : in  std_logic;

        -- external interface
        proc_status_in   : in  std_logic_vector( 7 downto 0
    );

        -- selector <--> distributor
        proc_free_en     : out std_logic;
        proc_selected    : out std_logic_vector( 7 downto 0
    ));

end processor_selector;

architecture Behavioral of processor_selector is

    constant cTCQ          : time := 2 ns;

    signal proc_status      : std_logic_vector( 7 downto 0 );
    signal foreground_proc  : std_logic_vector( 7 downto 0 );

begin

    process( clk )
        -- input latch
    begin
        if rising_edge( clk ) then
            proc_status <= proc_status_in after cTCQ;
        end if;
    end process;

    proc_free_en <= '1' when foreground_proc /= x"00" else '0';
    -- free processors available

    process( clk )
        -- free processor selector

```

```

begin
    if rising_edge( clk ) then
        if rst = '1' then
            foreground_proc <= ( others => '0' ) after
cTCQ;
        else
            for i in 7 downto 0 loop
                if proc_status(i) = '1' then
                    case i is
                        when 0 => foreground_proc <=
"00000001";
                        when 1 => foreground_proc <=
"00000010";
                        when 2 => foreground_proc <=
"00000100";
                        when 3 => foreground_proc <=
"00001000";
                        when 4 => foreground_proc <=
"00010000";
                        when 5 => foreground_proc <=
"00100000";
                        when 6 => foreground_proc <=
"01000000";
                        when 7 => foreground_proc <=
"10000000";
                        when others => foreground_proc
<= x"00";
                    end case;
                elsif proc_status = x"00" then
                    foreground_proc <= ( others => '0'
) after cTCQ;
                end if;
            end loop;
        end if;
    end if;
end process;

proc_selected <= foreground_proc;

end Behavioral;

```

Файл distributor.vhd

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity distribution_unit is
    port(
        clk          : in  std_logic;
        rst          : in  std_logic;

```

```

-- fifo <--> distributor
task_from_fifo : in  std_logic_vector( 15 downto 0
);

-- selector <--> distributor
any_proc_free  : in  std_logic;
proc_selected  : in  std_logic_vector( 7 downto 0
);

-- queue controller <--> distributor
rd_req          : out std_logic;
queue_busy      : in  std_logic;

-- external
task_out        : out std_logic_vector( 15 downto 0
);

proc_rdy_out    : out std_logic_vector( 7 downto 0
);
-- ready
proc_rdy_ack_in : in  std_logic_vector( 7
downto 0 );
-- prinyal
proc_req_out    : out std_logic_vector( 7 downto 0
);
-- zapr
proc_req_ack_in : in  std_logic_vector( 7
downto 0 ));
-- pzapr

end distribution_unit;

```

architecture Behavioral of distribution_unit is

```

type fsm_distribution is ( sIDLE, sREQUEST, sTRANSMIT );
signal state          : fsm_distribution;
constant cTCQ         : time := 2 ns;

signal proc_in_work    : std_logic_vector( 7 downto 0 );
signal proc_rdy_ack    : std_logic_vector( 7 downto 0 );
signal proc_req_ack    : std_logic_vector( 7 downto 0 );
signal timeout         : std_logic_vector( 7 downto 0 );

signal new_cycle       : std_logic;

begin

task_out <= task_from_fifo;
-- pin-to-pin

process( clk )
-- input latch
begin
if rising_edge( clk ) then
proc_rdy_ack <= proc_rdy_ack_in after cTCQ;
proc_req_ack <= proc_req_ack_in after cTCQ;

```

```

        end if;
    end process;

    new_cycle <= '1' when queue_busy = '1' and any_proc_free =
'1' else '0'; -- if there are data in the fifo and a free
processor; auxiliary signal

    process( clk )
        -- memory the selected processor
    begin
        if rising_edge( clk ) then
            if rst = '1' then
                proc_in_work <= ( others => '0' ) after cTCQ;
            else
                -- if there are data and process to work with
                if ( state = sIDLE or ( state = sTRANSMIT and
proc_rdy_ack = proc_in_work )) and new_cycle = '1' then
                    proc_in_work <= proc_selected after
cTCQ;

                    elsif state = sIDLE then
                        proc_in_work <= ( others => '0' ) after
cTCQ;

                    end if;
                end if;
            end if;
        end process;

        process( clk )
            -- finite state machine
        begin
            if rising_edge( clk ) then
                if rst = '1' then
                    state <= sIDLE after cTCQ;
                else
                    if timeout = x"ff" then
                        state <= sIDLE after cTCQ;
                    else
                        case state is

                            when sIDLE =>
                                if new_cycle = '1' then
                                    state <= sREQUEST after
cTCQ;

                                end if;

                                when sREQUEST =>
                                    if proc_req_ack = proc_in_work
then
                                        -- if processor is ready to get data
                                            state <= sTRANSMIT after
cTCQ;

                                        end if;

```

```

                                when sTRANSMIT =>
                                    if proc_rdy_ack = proc_in_work
then
                                -- if processor has received all the data
                                    if new_cycle = '1' then
                                        -- if there are another data and free
processor
                                                state <= sREQUEST
after cTCQ;
                                                else
                                                    state <= sIDLE after
cTCQ;
                                                    end if;
                                                end if;

                                when others =>
                                    state <= sIDLE after cTCQ;

                                end case;
                                end if;
                                end if;
                                end if;
                                end process;

                                process( clk )
                                    -- fsm timeout: if the fsm is too long in one
of the states - reset it!
                                begin
                                    if rising_edge( clk ) then
                                        if rst = '1' then
                                            timeout <= ( others => '0' ) after cTCQ;
                                        else
                                            if state /= sIDLE then
                                                if proc_req_ack /= 0 or proc_rdy_ack /=
0 then
                                                -- reset the timer if processor's acknowledgements
haven't come
                                                    timeout <= ( others => '0' ) after
cTCQ;
                                                    else
                                                        timeout <= timeout + 1 after cTCQ;
                                                    end if;
                                                end if;
                                            end if;
                                        end if;
                                    end if;
                                end process;

                                rd_req <= '1' when state = sREQUEST and timeout = x"00" else
'0'; -- read request to fifo

                                proc_req_out <= proc_in_work when state = sREQUEST and
timeout = x"00" else ( others => '0' ); -- request to processor

```

```

    proc_rdy_out <= proc_in_work when state = sTRANSMIT and
timeout = x"00" else ( others => '0' ); -- task write enable to
processor

```

```

    -- if we want to win one time step in sTRANSMIT, we should
comment line above and uncomment one below
    -- proc_rdy_out <= proc_in_work when state = sREQUEST and
proc_req_ack = proc_in_work else ( others => '0' );    -- task
write enable to processor

```

```

end Behavioral;

```

Файл processor.vhd

```

library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;

entity processor is
    port(
        clk          : in  std_logic;
        rst          : in  std_logic;

        -- scheduler interface
        task_in       : in  std_logic_vector(
15 downto 0 );
        proc_free_out : out std_logic;

        task_rdy_in   : in  std_logic;
        task_rdy_ack_out : out std_logic;
        task_req_in    : in  std_logic;
        task_req_ack_out : out std_logic );
end processor;

architecture Behavioral of processor is

    type fsm_proc is ( SIDLE, sREQUEST, sPROCESSING );
    -- processor's fsm
    signal sm_proc      : fsm_proc;

    constant cTCQ       : time      := 2 ns;
    constant cPERFORMANCE : integer := 5;
    -- less value is better performance

    signal task_received : std_logic_vector( 15 downto 0 );

    signal task_rdy      : std_logic;
    signal task_req       : std_logic;

```

```

    signal work_cntr          : std_logic_vector( cPERFORMANCE -
1 downto 0 );
    signal seed               : std_logic_vector( 31 downto
0 );
    signal xor_1              : std_logic_vector( 31 downto 0 );
    signal xor_2              : std_logic_vector( 31 downto 0 );
    signal xor_3              : std_logic_vector( 31 downto 0 );

begin

    process( clk )
        -- input latch
    begin
        if rising_edge( clk ) then
            task_rdy <= task_rdy_in after cTCQ;
            task_req <= task_req_in after cTCQ;
        end if;
    end process;

    process( clk )
        -- quasi-handler for acquired task
    begin
        if rising_edge( clk ) then
            if rst = '1' then
                task_received <= ( others => '0' ) after
cTCQ;
            else
                if task_rdy = '1' then
                    task_received <= task_in after cTCQ;
                end if;
            end if;
        end if;
    end process;

    process( clk )
        -- finite state machine
    begin
        if rising_edge( clk ) then
            if rst = '1' then
                sm_proc <= sIDLE after cTCQ;
            else
                case sm_proc is

                    when sIDLE =>
                        if task_req = '1' then
                            sm_proc <= sREQUEST after
cTCQ;

                        end if;

                    when sREQUEST =>
                        sm_proc <= sPROCESSING after cTCQ;

```

```

        when sPROCESSING =>
            if work_cntr = 0 then
                -- if processor has handled all the data
                sm_proc <= sIDLE after cTCQ;
            end if;

        when others =>
            sm_proc <= sIDLE after cTCQ;

    end case;
end if;
end if;
end process;

proc_free_out <= '1' when sm_proc = sIDLE else '0';

process( clk )
    -- work imitation counter
begin
    if rising_edge( clk ) then
        if rst = '1' then
            work_cntr <= ( others => '0' ) after cTCQ;
        else
            if sm_proc = sREQUEST then
                work_cntr <= seed( cPERFORMANCE - 1
downto 0 ) after cTCQ;
            elsif sm_proc = sPROCESSING then
                if work_cntr /= 0 then
                    work_cntr <= work_cntr - 1 after
cTCQ;
                end if;
            end if;
        end if;
    end if;
end if;
end process;

task_req_ack_out <= '1' when sm_proc = sREQUEST else '0';
    -- acknowledge scheduler's request
task_rdy_ack_out <= task_rdy;
    -- signal to scheduler when processor has got the
data

----- pseudorandom number generator -----
---

--  uint32_t x = seed;
--  x ^= x << 13;
--  x ^= x >> 17;
--  x ^= x << 5;
--  seed = x;

```

```

--      return x;

xor_1 <= seed xor seed( 18 downto 0 ) & "00000000000000";
-- combination locomotions
xor_2 <= xor_1 xor "000000000000000000" & xor_1( 31 downto 17
);
xor_3 <= xor_2 xor xor_2( 26 downto 0 ) & "00000";

process( clk )
    -- seed trigger
begin
    if rising_edge( clk ) then
        if rst = '1' then
            seed <= x"9f2599a2" after cTCQ;
        else
            seed <= xor_3 after cTCQ;
        end if;
    end if;
end process;

end Behavioral;

```

Файл task_spawner_imitation

```

library IEEE;
    use IEEE.STD_LOGIC_1164.all;
    use IEEE.std_logic_unsigned.all;

entity task_spawner is
    Port(      clk                : in  std_logic;
            rst                  : in  std_logic;

            -- external interface
            new_task_out         : out  std_logic_vector( 15
downto 0 );
            enable_out           : out  std_logic );

end task_spawner;

architecture Behavioral of task_spawner is

    constant cTCQ              : time := 2 ns;
    constant cCOUNT           : integer := 2;

    signal task_gen             : std_logic_vector( 15 downto
0 );
    signal enable_gen          : std_logic_vector( 3 downto 0 );

begin

```

```

        process( clk )
        packets generator
        begin
            if rising_edge( clk ) then
                if rst = '1' then
                    task_gen <= ( others => '0' ) after
cTCQ;
                else
task_gen <= task_gen + 1 after cTCQ;
                end if;
            end if;
        end process;
        new_task_out <= task_gen;

        process( clk )
        begin
            if rising_edge( clk ) then
                if rst = '1' then
                    enable_gen <= "0010" after cTCQ;
                else
                    enable_gen <= ( enable_gen(1) xor enable_gen(0) )& enable_gen( 3
downto 1 ) after cTCQ; -- linear feedback shift register
                end if;
            end if;
        end process;
        enable_out <= enable_gen(0);
    end Behavioral;

```

7. ПРИЛОЖЕНИЕ В

ПРЕЗЕНТАЦИЯ



Тема работы: Аппаратный планировщик с глобальной очередью задач

Автор работы : студентка группы 15BB1 Алексеева А.А.

Научный руководитель : дтн, профессор Бикташев Р.А.

Актуальность работы

В настоящее время главные недостатки программного обеспечения, на котором базируются все традиционные операционные системы, могут быть устранены путем реализации функционала ядра аппаратными средствами

Программа разработана в виде набора модулей, которые могут использоваться в системах реального времени с применением аппаратной поддержки для выполнения функции синхронизации, связанной с планированием задач, что приведет к увеличению коэффициента использования процессоров, а также сокращению времени ответа, улучшению надежности и безопасности систем управления.

Изм.	Лист	№ докум.	Подпись	Дата

ПГУ 2.090301.01.001 ПЗ

Лист

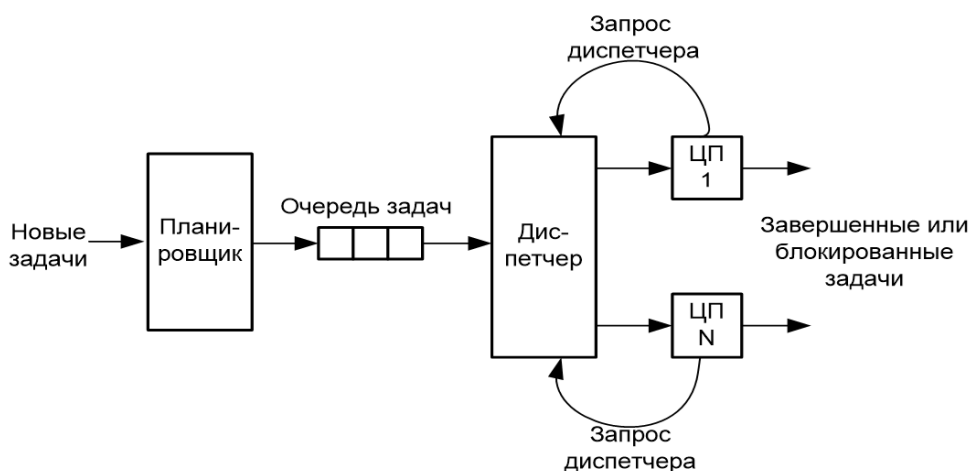
58

Цель работы: разработка аппаратного планировщика с глобальной очередью задач, обеспечивающего высокую реакцию на запрос

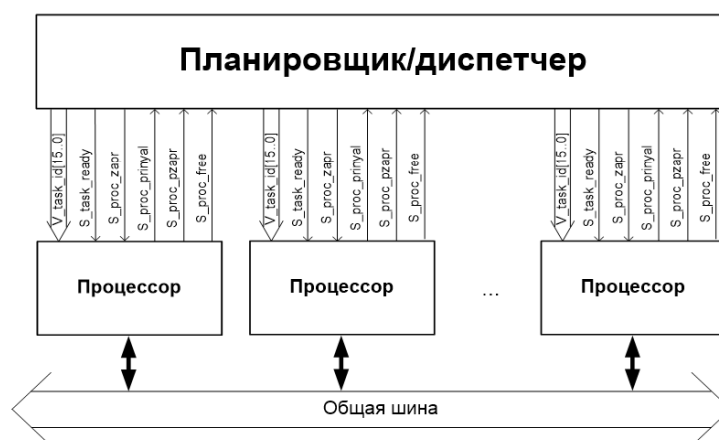
Основные задачи:

- генерация идентификаторов (ID) задач;
- имитация работы процессоров многопроцессорной системы;
- выполнение функции планировщика задач, формирующего и поддерживающего очередь задач;
- выбор задач из очереди;
- запуск задач на выполнение в любом свободном процессоре;
- удаление задач из очереди, получивших полное обслуживание.

Общая схема алгоритма диспетчеризации

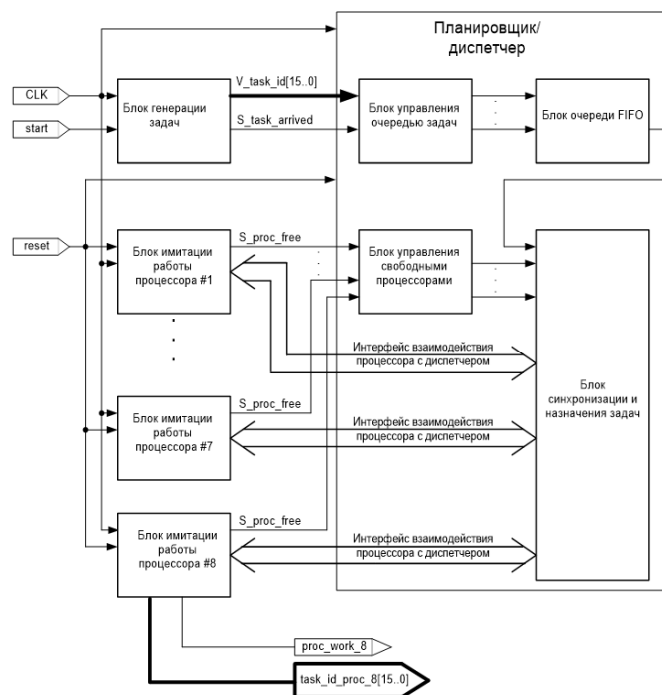


Общий вид многопроцессорной системы с аппаратной поддержкой планировщика задач



Интерфейс взаимодействия процессора с диспетчером



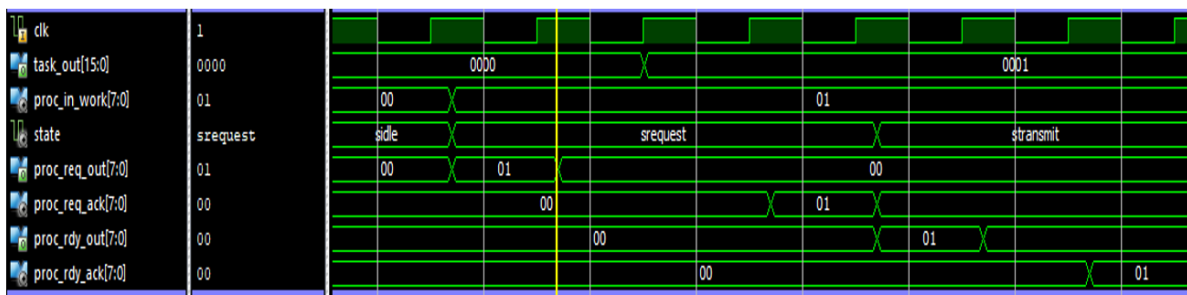


Моделирование планировщика на языке VHDL

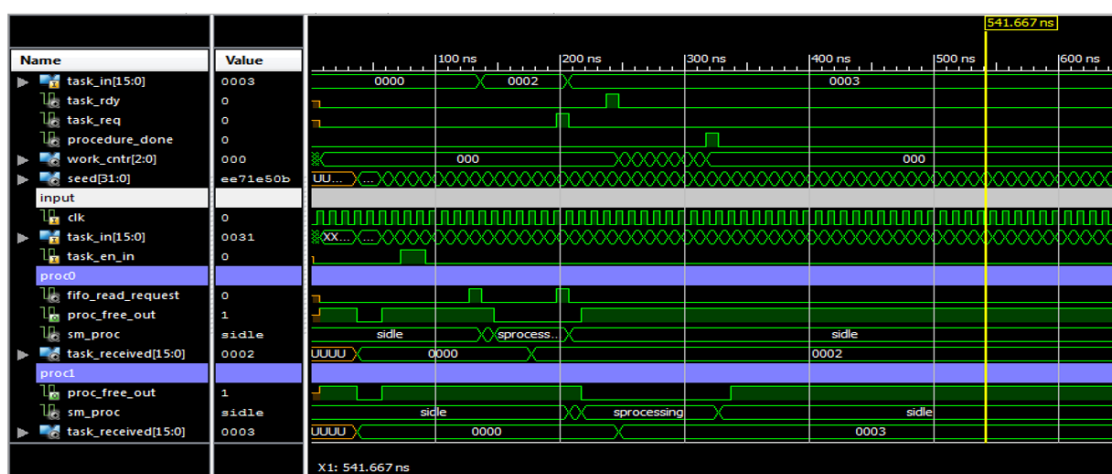
Модель устройства планировщика была реализована на языке *VHDL* в виде четырех программных модулей.

- Первый модуль реализует логические функции блока управление очередью задач.
- Второй модуль реализует блок FIFO.
- Третий модуль реализует блок управления свободными процессорами,
- Четвертый модуль реализует блок синхронизации

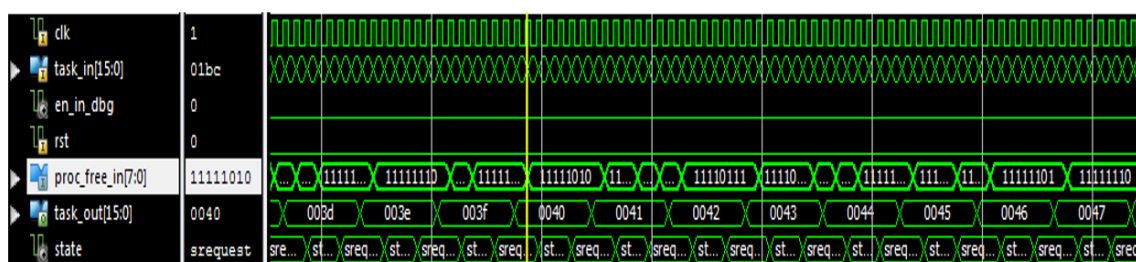
Работа синхронизатора



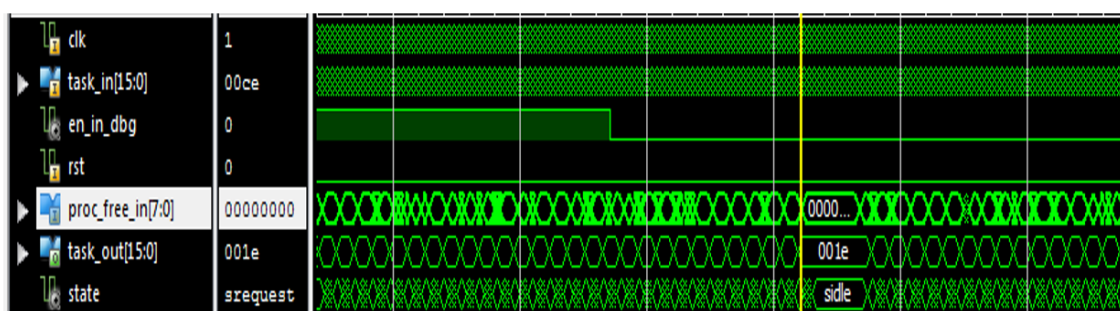
Производительность диспетчера



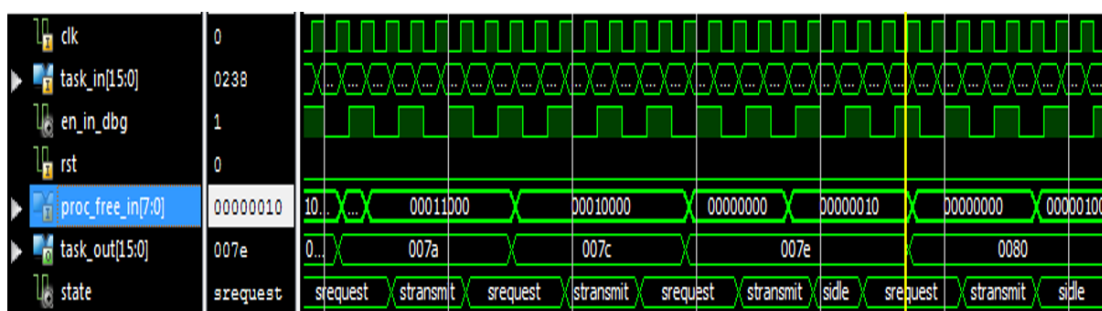
Нормальный режим работы диспетчера



Перегрузка диспетчера



Сильная перегрузка диспетчера



В результате выполнения работы были получены следующие результаты:

- Реализация данного аппаратного планировщика с глобальным разделением задач демонстрирует повышения производительности многопроцессорной системы.
- Полученное значение задержки назначения задачи при условии отсутствия очереди в процессорном узле не превышает одного такта, что выполняет условие повышения производительности аппаратной реализации алгоритма.
- Дальнейшая оптимизация может быть достигнута изменением способа формирования запроса, выработкой сигнала комбинационной схемой, а не по синхросигналу.



Спасибо за внимание!