**Федеральный исследовательский центр «Информатика и управление»
Российской Академии Наук (ФИЦ ИУ РАН)**

На правах рукописи

Хилько Дмитрий Владимирович

ИССЛЕДОВАНИЕ И РАЗРАБОТКА ПОТОКОВОЙ РЕКУРРЕНТНОЙ АРХИТЕКТУРЫ ДЛЯ ЭФФЕКТИВНОЙ РЕАЛИЗАЦИИ ПАРАЛЛЕЛИЗМА В ОБЛАСТИ ЦИФРОВОЙ ОБРАБОТКИ СИГНАЛОВ

2.3.2 «Вычислительные системы и их элементы»

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель
к.т.н., ведущий научный сотрудник
Степченков Юрий Афанасьевич

Москва – 2023

Оглавление

Введение

Актуальность исследования

Одна из ключевых характеристик любой вычислительной системы – это ее производительность. На сегодняшний день большинство вычислительных систем основаны на традиционной архитектуре, предложенной фон-Нейманом. Долгое время высокого показателя производительности фон-Неймановских систем удавалось достигать путем повышения тактовой частоты. Но уже на уровне частот 3-4 ГГц проблемы питания и охлаждения процессора становятся определяющими. Поэтому ведутся исследования в области разработки высокопроизводительных архитектур вычислительных систем. Одним из основных способов повышения производительности является реализация параллельных вычислений на различных уровнях: команд, данных и задач.

Современные фон-Неймановские процессоры реализуют широкий набор средств реализации параллелизма уровня команд, а также состоят из множества вычислительных ядер. Однако, в основе традиционной архитектуры лежит последовательный вычислительный процесс, что приводит к высокой аппаратной избыточности механизмов реализации параллелизма. Это привело к необходимости разработки новых нетрадиционных архитектур вычислительных систем. Наиболее перспективной нетрадиционной архитектурой является потоковая архитектура, в которой поток данных имеет приоритет над потоком команд и является инициатором вычислений. Вычислительные системы, функционирующие под управлением потока данных, теоретически могут обеспечить большую производительность параллельных вычислений по сравнению с фон-неймановскими. Связано это с отсутствием «узких мест», характерных для фон-неймановской архитектуры, а также исключением вероятности обработки неподготовленных данных.

Исследование и разработка систем потоковой модели вычислений и архитектуры является актуальной задачей начиная с 1980-х годов и по настоящее время. Тем не менее, исследователи столкнулись с целым рядом проблем, которые не позволили построить эффективные вычислительные системы на основе потоковой архитектуры. К данным проблемам относятся: реализация рекурсии, реализация циклов, обработка константных данных, высокая степень избыточности тегированных данных, сложность и дороговизна аппаратной реализации механизмов сравнения тегированных данных, неэффективность при вычислении последовательных фрагментов алгоритмов и др. Поэтому данный класс систем не получил широкого распространения.

В поисках путей усовершенствования потоковой модели вычислений коллективом Института кибернетики имени В. М. Глушкова НАН Украины (Палагин А.В., Яковлев Ю.С. Махиборода А.В, и др.) была предложена идея новой потоковой модели вычислений, которая

впоследствии была развита и доработана сотрудниками ИПИ РАН (в настоящее время ФИЦ ИУ РАН) – Степченковым Ю.А, Филином А.В., Петрухиным В.С. и др. Данная модель вычислений была названа рекуррентно-потоковой.

Другой группой российских исследователей потоковой модели вычислений был коллектив ИПИ РАН под руководством академика Бурцева В.С. (Стемпковский А.Л., Хайлов И.К, Твердохлебов М.В. и др.). После кончины академика Бурцева большая часть коллектива перешла в ИППМ РАН. К настоящему времени коллектив под руководством Стемпковского А.Л. получил несколько грантов для исследования этой проблематики и занимается разработкой параллельной потоковой вычислительной системы «Буран» (ППВС «Буран»). «Буран» является системой массового параллелизма и реализует мелкозернистый параллелизм.

Наиболее коммерчески успешной отечественной архитектурой, основанной на использовании принципов потоковой модели вычислений, является Мультиклеточная архитектура. Данный проект начал свое развитие в 2001 году и к 2013 году успел получить несколько дипломов и наград. Процессоры на основе данной архитектуры позиционируются как отказоустойчивые. Основным принципом функционирования является широковещательная рассылка исполняемой программы и данных каждому вычислительному ядру процессора. Таким способом достигается использование естественного параллелизма, внеочередного исполнения команд и реализация мелкозернистого параллелизма.

На базе рекуррентно-потоковой модели в ФИЦ ИУ РАН ведутся работы по созданию нетрадиционной рекуррентной архитектуры, предназначенной для реализации параллельных вычислений ограниченной размерности в области цифровой обработки сигналов (ЦОС). Новая архитектура названа многоядерной потоковой рекуррентной архитектурой (МПРА). В настоящей реализации это гибридная двухуровневая архитектура рекуррентного обработчика сигналов (ГАРОС) с традиционным процессором на верхнем (управляющем) уровне и рекуррентным операционным устройством (РОУ) на нижнем уровне. В состав РОУ входят четыре вычислительных ядра.

В ФИЦ ИУ РАН также разработана идеология и методология проектирования самосинхронных схем, правильное функционирование которых не зависит от задержек составляющих их элементов. Данные схемы и системы на их основе обладают рядом свойств, выделяющих их из общего ряда цифровых устройств. Они «естественно надежны», поскольку гарантируют сохранение работоспособности аппаратуры в широком диапазоне дестабилизирующих факторов. Самосинхронизация на логическом уровне (по готовности данных в потоковой модели) хорошо сочетается с самосинхронизацией на аппаратном уровне (по готовности результата). Поэтому реализация МПРА также ориентирована на самосинхронный базис.

В основе рекуррентно-потоковой модели лежат принципы самодостаточных данных и рекуррентности. Самодостаточность данных заключается в организации единого потока данных и инструкций. Каждый элемент единого потока (операнд) образует исполняемый пакет, который спускается на исполнение в свободное ядро РОУ. Наличие в пакете и инструкций и данных упраздняет необходимость адресации памяти программ и памяти данных в привычном понимании. Необходимость адресации остается на уровне выше для корректного формирования исполняемых пакетов и их рассылки по вычислительным ресурсам. Это позволяет исключить этапы выборки данных и инструкций и совместить этап дешифрации инструкции с этапом ее выполнения. В результате общее количество этапов обработки пакета снижается практически вдвое ценой относительно небольшого увеличения его размера.

Одной из ключевых проблем потоковой модели вычислений является высокая степень избыточности тегированных данных. В некоторых случаях объем теговой информации может в разы превышать объем данных, которым эта информации принадлежит. Для решения этой проблемы в рекуррентно-потоковой модели вычислений предложен принцип рекуррентности. Данный принцип заключается в вычислении каждой следующей инструкции в ходе развития вычислительного процесса как функции от текущей исполняемой инструкции. В процессе вычислений происходит порождение новых исполняемых пакетов (рекуррентная развертка), которые отправляются на дальнейшую обработку. Следовательно, исходный поток инструкций рекуррентно сворачивается (сжимается), что позволяет резко сократить накладные расходы, связанные с опережающим хранением трассы вычислительного процесса. Сжатие также позволяет упростить описание и хранение вычислительного контекста. В результате реализации принципа рекуррентности общая избыточность теговой информации значительно снижается.

Комбинирование принципов самодостаточности и рекуррентности приводит к тому, что исполняемой программы не существует в МПРА в привычном смысле. Есть только начальное сжатое состояние значений тегов операндов, которые динамически подвергаются рекуррентной развертке. Данное сжатое состояние формирует контекст вычисляемого алгоритма и называется капсулой. Представление решаемой большой задачи в виде совокупности капсул было названо капсульным стилем программирования. Таким образом, МПРА реализует мелкозернистый параллелизм на уровне операндов и крупнозернистый параллелизм на уровне капсул. Очевидно, что разработка программного обеспечения в данном стиле является нетривиальной задачей.

Область ЦОС была выбрана по двум причинам. Во-первых, задачи ЦОС необходимо решать во многих областях человеческой деятельности. Следовательно, разработка вычислительного устройства для задач ЦОС является актуальной. Во-вторых, принципы потоковых архитектур и требования со стороны алгоритмов ЦОС хорошо сочетаются друг с другом в приложениях, для которых характерна высокая степень внутреннего параллелизма.

Основной сдерживающей причиной широкого практического использования такой интеграции является, прежде всего, стоимостной фактор, который делает нецелесообразным прямолинейное использование в них технических решений из области потоковых систем массового параллелизма.

На момент начала выполнения диссертационного исследования МПРА существовала на уровне технической спецификации и небольшого программного прототипа. Результаты предварительной апробации показали, что архитектура имеет высокий потенциал эффективности, но ее существующая структура и набор функциональных возможностей не удовлетворяют требованиям производительности для задач ЦОС реального времени. Поэтому построение высокоэффективной потоковой рекуррентной архитектуры для решения задач ЦОС реального времени, а также разработка элементов методологии программирования и отладки ГАРОС (как самой архитектуры, так и специализированного программного обеспечения), являются необходимыми и актуальными задачами.

Объект исследования: рекуррентно-потоковая модель вычислений и архитектура на ее основе.

Предмет исследования: структурная организация и алгоритмы функционирования ключевых компонент новой архитектуры, а также методы и средства ее программирования и отладки.

Цель и задачи диссертационной работы

Целью диссертационной работы является разработка элементов потоковой рекуррентной архитектуры и элементов методологии программирования и отладки для создания прототипа устройства рекуррентного обработчика сигналов, который обладает требуемым уровнем производительности для решения задач ЦОС реального времени.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Разработать структурные элементы архитектуры, методы и алгоритмы их функционирования, которые позволят: эффективно реализовать поддержку параллелизма на различных уровнях; минимизировать избыточность тегированных данных; достичь требуемого уровня производительности для задач ЦОС реального времени.

2) Разработать теоретические основы программируемости ГАРОС, которые включают в себя: методики и алгоритмы реализации различных этапов разработки и отладки ПО; программную и аппаратную поведенческие модели архитектуры для проведения испытаний; набор средств аппаратно-программного моделирования и отладки архитектуры.

3) Осуществить испытания всех разработанных средств путем реализации демонстрационной задачи распознавания изолированных слов и комплекта типовых алгоритмов ЦОС для подтверждения эффективности полученных результатов работы путем сравнения с современным высокопроизводительным сигнальным процессором.

Научная новизна диссертационной работы состоит в следующем:

1) Впервые предложены методы и алгоритмы организации суперскалярных вычислений на уровне микроархитектуры вычислительных ядер ГАРОС, учитывающие специфику представления тегированных самодостаточных данных и процесса рекуррентной развертки. Разработанные решения позволяют использовать вычислительные ядра в двух, трех, а в некоторых случаях и в четырех задачном режиме.

2) Впервые предложены методы и алгоритмы реализации аппаратной поддержки алгоритма БПФ, которые учитывают специфику хранения и обработки тегированных самодостаточных данных и суперскалярность вычислительных ядер ГАРОС. Разработанные решения позволяют использовать вычислительные ядра в четырех задачном режиме на протяжении всего процесса вычисления БПФ, а также обеспечивают рациональное использование памяти самодостаточных данных путем максимального снижения их избыточности.

3) Впервые разработаны элементы методологии программирования и отладки ГАРОС, включающие в себя: методики и алгоритмы, комплект моделей и инструментов аппаратно-программного моделирования. Разработанный набор инструментов позволил успешно реализовать задачу распознавания изолированных слов и синтезировать ПЛИС прототип ГАРОС, который позволяет решать эту задачу в реальном времени.

Методы проведения исследования: Математическую основу исследования составляют системный анализ, теория алгоритмов, теория графов, теория языков программирования и методы организации архитектур вычислительных систем. Практические результаты диссертации были получены с привлечением таких экспериментальных методов исследования как: имитационное моделирование на программном и аппаратном уровнях; методов разработки и тестирования Test-Driven Development для программных средств и Assertion-Based Design для аппаратных средств; синтез аппаратного прототипа и натурный эксперимент.

Результаты диссертационной работы реализованы:

1) в виде комплекта моделей, элементов методологии программирования и отладки ГАРОС, а также комплекта специализированного ПО для проведения испытаний в ходе

выполнения НИР «Информационные, управляющие и телекоммуникационные системы 2017-2021» в рамках государственного задания № 0063-2019-0010 (11) по направлению «Концептуальные и методологические основы создания семейства потоковых самосинхронных процессоров и средств поддержки их проектирования»;

2) в виде ПЛИС прототипа в ходе выполнения гранта РНФ № 19-11-00334 «Инновационная архитектура самосинхронных цифровых сигнальных процессоров, управляемых потоком данных 2019-2021»;

3) в виде программного комплекса моделирования и отладки «ПК ПОТОК» в ходе выполнения научного проекта «Методы построения и моделирования сложных систем на основе интеллектуальных и суперкомпьютерных технологий, направленные на преодоление больших вызовов 2020-2023», финансируемого Минобрнауки по Соглашению No 075-2020-799 от 29 сентября 2020 г.

Результаты диссертационной работы были получены соискателем лично, за исключением аппаратной модели, для которой автор: разрабатывал алгоритмы; проектировал отдельные функциональные блоки; осуществлял тестирование.

Личный вклад автора

Разработанные элементы методологии программирования и отладки ГАРОС в составе методик, алгоритмов, технологии, архитектуры программного комплекса «ПК ПОТОК» и его основных компонент получены соискателем лично. Программная модель прототипа архитектуры, программная реализация «ПК ПОТОК» также разработаны соискателем лично, либо под непосредственным его руководством.

Автор принимал участие в разработке аппаратной модели под руководством Дьяченко Ю.Г. В частности: разрабатывал алгоритмы; проектировал отдельные функциональные блоки; осуществлял тестирование, верификацию и валидацию средствами «ПК ПОТОК».

Под руководством Морозова Н.В. автор также принимал участие в синтезе ПЛИС прототипа, его отладке средствами «ПК ПОТОК» и накоплении и обработке результатов натурных испытаний.

Теоретическая значимость: разработаны методы и алгоритмы организации вычислений в рамках МПРА, которые позволили создать эффективный прототип ГАРОС для решения задач ЦОС в реальном времени. Разработаны элементы методологии программирования и отладки ГАРОС, которые позволили организовать полноценный и эффективный процесс программирования и отладки полученного прототипа. Разработанный набор инструментов

может быть использован в качестве основы для дальнейшего развития архитектуры как для применения в области ЦОС, так и для других классов задач.

Практическая значимость: создан ПЛИС прототип ГАРОС, который способен эффективно решать задачу распознавания изолированных слов, а также ряда других типовых задач в области ЦОС (различные виды цифровых фильтров, частотный анализ при помощи алгоритма БПФ, алгоритм Витерби кодирования сигнала, поиск минимума/максимума и др.). Полученные результаты могут быть использованы в качестве основы для разработки других вычислительных устройств на отечественной элементной и архитектурной базе.

Достоверность результатов обеспечивается:

- корректностью применении выбранных моделей, методов и алгоритмов для разработки, программирования и отладки архитектур вычислительных систем и их прототипов;
- корректностью ограничений и допущений при проведении моделирования;
- достоверностью исходных данных демонстрационной задачи распознавания изолированных слов;
- адекватностью (бит-экзектностью) результатов программного моделирования, аппаратного моделирования и натурного эксперимента над ПЛИС прототипом и исходных данных демонстрационной задачи.

Апробация результатов работы выполнена в ряде конференций. Наиболее значимые из них:

1) 2016 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Санкт-Петербург, Россия, 02-03 февраля 2016 г. (индексируется в Scopus).

2) Проблемы разработки перспективных микро- и наноэлектронных систем – 2016 (МЭС-2016), Москва, Зеленоград, Россия, 03-07 октября 2016 г. (индексируется в РИНЦ и ВАК).

3) IEEE East-West Design & Test Symposium (EWDTS’2016), Ереван, Армения, 14-17 октября 2016 г. (индексируется в Scopus).

4) 2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Санкт-Петербург, Россия, 01-03 февраля 2017 г. (индексируется в Scopus).

5) 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Москва, Россия, 29 января – 01 февраля 2018 г. (индексируется в Scopus).

6) Проблемы разработки перспективных микро- и наноэлектронных систем – 2018 (МЭС-2018), Москва, Зеленоград, Россия, 01-05 октября 2018 г. (индексируется в РИНЦ и ВАК).

7) 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Москва, Россия, 28-31 января 2019 г. (индексируется в Scopus).

8) 2019 IEEE EAST-WEST DESIGN & TEST SYMPOSIUM (EWDTS'2019), Батуми, Грузия, 13-16 сентября 2019 г. (индексируется в Scopus).

9) 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Москва, Россия, 27-30 января 2020 г. (индексируется в Scopus).

10) 2020 International Conference Engineering Technologies and Computer Science (EnT 2020), Москва, Россия, 24-27 июня 2020 г. (индексируется в Scopus).

11) Проблемы разработки перспективных микро- и наноэлектронных систем — 2020 (МЭС-2020), Москва, Зеленоград, Россия, октябрь 2020 г. (индексируется в РИНЦ и ВАК).

12) 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), Санкт-Петербург, Россия, 26-29 января 2021 г. (индексируется в Scopus).

13) 2021 International Conference Engineering Technologies and Computer Science (EnT 2021), Москва, Россия, 18-19 августа 2021 г. (индексируется в Scopus).

14) IEEE East-West Design & Test Symposium (EWDTS'2021), Батуми, Грузия, 10-13 сентября 2021 г. (индексируется в Scopus).

15) Проблемы разработки перспективных микро- и наноэлектронных систем – 2021 (МЭС-2021), Москва, Зеленоград, Россия, октябрь 2021 г. (индексируется в РИНЦ и ВАК).

16) 2022 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), Санкт-Петербург, Россия, 25-28 января 2022 г. (индексируется в Scopus).

17) Проблемы разработки перспективных микро- и наноэлектронных систем – 2022 (МЭС-2022), Москва, Зеленоград, Россия, октябрь 2022 г. (индексируется в РИНЦ и ВАК).

18) 2023 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), Санкт-Петербург, Россия, 24-27 января 2023 г. (индексируется в Scopus).

Кроме того, соискателем получены 11 свидетельств о регистрации программ и 5 патентов Российской Федерации на изобретение.

Область исследования диссертации соответствует требованиям паспорта специальности ВАК Минобрнауки 2.3.2 «Вычислительные системы и их элементы» в части пунктов:

п. 5 – Разработка научных методов и алгоритмов организации арифметической, логической, символьной и специальной обработки данных, хранения и ввода-вывода информации;

п. 7 – Разработка научных методов и алгоритмов организации параллельной и распределенной обработки информации, многопроцессорных, многомашинных и специальных вычислительных систем.

Публикации. По теме диссертации автором опубликовано 50 печатных работ, в том числе: 14 публикаций в рецензированных научных изданиях, входящих в системы цитирования Web of Science и Scopus; 24 публикаций в рецензированных научных изданиях, входящих в систему цитирования РИНЦ и перечень ВАК Минобрнауки России («Перечень рецензируемых научных изданий, в которых должны быть опубликованы основные научные результаты диссертаций на соискание ученой степени кандидата наук, на соискание ученой степени доктора наук»); 11 свидетельств о регистрации программ; 3 патента Российской Федерации на изобретение.

Структура и объем работы. Диссертационная работа состоит из введения, четырех глав, заключения, списка литературы. Общий объем диссертации – 200 стр., в том числе 156 стр. основного текста, одно приложение, 35 иллюстраций в основном тексте, 16 таблиц в основном тексте. Список литературы состоит из 116 наименований.

Научные положения, выносимые на защиту:

1) методы и алгоритмы организации вычислительного процесса позволяющие достичь требуемого уровня производительности прототипа архитектуры для задач ЦОС реального времени;

2) структурные элементы архитектуры, а также методы и средства аппаратной поддержки алгоритма БПФ позволяющие минимизировать избыточность тегированных данных, которая является ключевой проблемой потоковых систем, и использовать дорогие элементы памяти рациональным образом;

3) элементы методологии программирования и отладки ГАРОС, позволяющие организовать полноценный и эффективный процесс разработки и отладки ПО для прототипа архитектуры, охватывающие большую часть этапов жизненного цикла ПО и позволяющие существенно сократить непроизводительные затраты разработчиков за счет частичной автоматизации процессов верификации и валидации.

Глава 1 Исследование потоковой рекуррентной архитектуры и ее прототипа ГАРОС

Современное многообразие и сложность решаемых вычислительных задач потребовала создания большого количества разнообразных моделей вычислений и архитектур вычислительных систем. Часть из них используют схожие базовые механизмы функционирования, в то время как другие – основаны на принципиально других решениях. Однако, большинство этих архитектур объединяет то, что на том или ином уровне они реализуют механизмы эффективной поддержки параллелизма. В главе рассматриваются основные способы организации параллельных вычислений. Рассматриваются ключевые принципы потоковой модели вычислений. Приводится анализ характерных проблем реализации потоковых архитектур и существующих подходов их решения. Представлена классификация существующих вычислительных систем или их прототипов на основе потоковой архитектуры.

Рассматриваются концептуальные основы рекуррентно-потоковой модели вычислений и ключевые особенности ее прототипа ГАРОС. Обозначается место архитектуры в классификации потоковых архитектур. Также в главе приводится сравнительный анализ ГАРОС и других отечественных архитектур, основанных на потоковой модели вычислений. Ставятся задачи диссертационного исследования.

1.1 Параллельные вычислительные системы

Любая вычислительная система реализует некоторый вычислительный процесс. Все процессы, выполняемые в вычислительных системах, могут быть распределены по двум основным категориям: последовательные и параллельные.

Последовательный вычислительный процесс характеризуется тем, что в каждый момент времени вычислительная система может исполнять только один из дочерних процессов. Такие вычислительные системы еще называются однозадачными или скалярными.

Параллельный вычислительный процесс характеризуется тем, что в каждый момент времени вычислительная система может исполнять один или более дочерних процесса одновременно. Такие вычислительные системы называются многозадачными (суперскалярными). Дочерние процессы, которые одновременно имеют активное состояние, называются параллельными процессами соответственно.

По мере развития теорий вычислений и проектирования вычислительных систем стало очевидно, что производительности машин, основанных на использовании последовательного вычислительного процесса, недостаточно для покрытия возрастающей сложности задач, решаемых с их помощью. В связи с этим, со второй половины XX века в мире ведутся работы и

исследования, направленные на разработку архитектур, реализующих параллельный вычислительный процесс.

Идея параллельных вычислений возникла и обсуждается в научном сообществе уже давно. В 1958 году Гилл (Gill) написал работу [1], посвященную проблемам параллельной обработки информации, которую можно считать одной из наиболее ранних в этой области. Следует отметить, что эксплуатация параллелизма – очевидное на первый взгляд решение по увеличению производительности, сопряжено с большим количеством проблем [2].

1.1.1 Классификация параллельных архитектур Кришнамарфи

Разнообразие моделей вычислений и основанных на них архитектур вычислительных систем привело к необходимости создания соответствующих классификаций. В книгах Воеводина В.В. и Михайлова Б.М. приводится подробное описание большинства классификаций [2, 3]. Наиболее полной с точки зрения классификации параллельных вычислительных систем является классификация Е. Кришнамарфи. Кришнамарфи предложил использовать четыре основные характеристики:

1. степень гранулярности;
2. способ реализации параллелизма;
3. топология и природа связи процессоров;
4. способ управления процессорами.

Построения классификации осуществляется в соответствии со следующим алгоритмом. Для каждой степени гранулярности рассматриваются все возможные способы реализации параллелизма. Для каждого полученного варианта рассматриваются все комбинации топологии связи и способов управления процессорами. В результате получается дерево, представленное на рисунке 1.1, в котором каждый ярус соответствует своей характеристике, каждый лист представляет отдельную группу компьютеров в данной классификации, а путь от вершины дерева однозначно определяет значения указанных выше характеристик. На рисунке также отмечено, к каким классам параллельных архитектур относится МПРА.

Несмотря на то, что классификация Е. Кришнамарфи построена лишь на четырех признаках, она позволяет выделить и описать такие "нетрадиционные" параллельные системы, как систолические массивы, потоковые машины и др. Но эта простота является и основной причиной ее недостатков. Некоторые архитектуры нельзя однозначно отнести к тому или иному классу, например, компьютеры с архитектурой гиперкуба и ассоциативные процессоры. Для более точного описания таких машин потребуется ввести еще целый ряд характеристик, таких, как размещение задач по процессорам, способ маршрутизации сообщений, возможность реконфигурации, аппаратная поддержка языков программирования и другие.

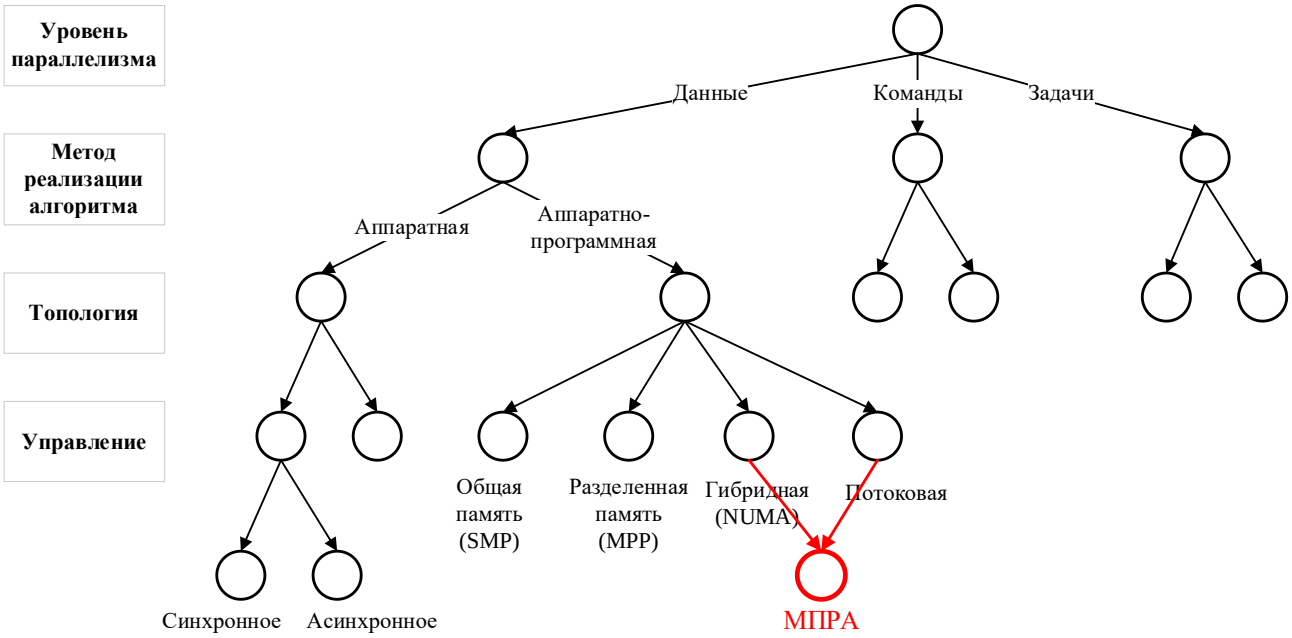


Рисунок 1.1 – Классификация Кришнамарфи

1.1.2 Способы организации параллельных вычислений

1.1.2.1 Параллельная обработка данных

Архитектуры, построенные согласно данному принципу, описывают процесс вычислений в виде взаимодействия совокупности конечного множества идентичных вычислительных устройств, каждое из которых решает некоторую часть поставленной задачи. При этом общая производительность системы при такой организации вычислений ограничивается сверху в соответствии с законом Амдала [4]. По способу организации коммутационной среды выделяют следующие типы моделей параллельных вычислений [5]:

1) Процесс/канал. В этой модели программы состоят из одного или более процессов, распределенных по процессорам. Процессы выполняются одновременно, их число может измениться в течение времени выполнения программы. Процессы обмениваются данными через каналы, которые представляют собой однонаправленные коммуникационные линии, соединяющие только два процесса. Каналы можно создавать и удалять.

2) Общей памяти. В этой модели все процессы совместно используют общее адресное пространство. Процессы асинхронно обращаются к общей памяти с запросами чтения/записи, что создает проблемы блокировки памяти. Для управления доступом к общей памяти используются стандартные механизмы синхронизации - семафоры и блокировки процессов. Архитектуру параллельных вычислительных систем с общей памятью называют также симметричной многопроцессорной архитектурой (symmetric multiprocessing – SMP).

3) Разделенной памяти. В этой модели каждый процессор имеет собственную оперативную память, которая хранит исполняемую программу и связанные с ней данные.

Взаимодействие между процессорами осуществляется путем обмена, используя подпрограммы приема/передачи данных некоторой коммуникационной системы. Архитектуру параллельных вычислительных систем с разделяемой памятью также называют массивно-параллельной (massive parallel processing – MPP).

4) Неоднородной памяти (non uniform memory access - NUMA). Данная модель является гибридной и объединят в себе принципы общей и разделенной памяти. Память физически распределена по различным частям системы, но логически она является общей. Система построена из однородных базовых модулей, состоящих из небольшого числа процессоров и блока памяти. Модули объединены с помощью высокоскоростного коммутатора. Поддерживается: единое адресное пространство; аппаратный доступ к удаленной памяти, т.е. к памяти других модулей. Доступ к локальной памяти осуществляется в несколько раз быстрее, чем к удаленной. Другими словами, NUMA является MPP архитектурой, где в качестве отдельных вычислительных элементов берутся SMP-узлы.

5) Параллелизм данных (векторная обработка). В этой модели единственная программа задает распределение данных между всеми процессорами компьютера и операции над ними. Распределяемыми данными обычно являются массивы (вектора). Как правило, языки программирования, поддерживающие данную модель, допускают операции над массивами, позволяют использовать в выражениях целые массивы и фрагменты массивов. Распараллеливание операций над массивами позволяет увеличить производительность программы. Компилятор отвечает за генерацию кода, осуществляющего распределение элементов массивов и вычислений между процессорами. Каждый процессор отвечает за то подмножество элементов массива, которое расположено в его локальной памяти.

1.1.2.2 Параллелизм на уровне инструкций

Основными методами реализации параллелизма уровня инструкций являются [6]:

1) Конвейерная обработка инструкций

Данный метод является наиболее распространенным и используется в подавляющем большинстве микроархитектур процессоров. Благодаря разбиению процесса выполнения инструкции на короткие по длительности стадии достигается высокий уровень тактовой частоты. В тоже время, реализация техники одновременного исполнения разных стадий разных инструкций в один и тот же такт позволяет значительно повысить пропускную способность процессора и его производительность в целом. Однако у данного подхода есть и свои проблемы.

Первая проблема заключается в возникновении конфликтов доступа к одним и тем же ресурсам процессора различными инструкциями, которые находятся на разных стадиях исполнения. Возникает она, когда имеет место зависимость по данным между инструкциями. Данная проблема может решаться с помощью пересылки (bypass – байпаса) промежуточного

результата зависимой инструкции до его записи в регистровый файл. Если же байпас невозможен, то необходимо приостанавливать конвейер. Приостановка конвейера, очевидно, приводит к задержкам и потери производительности.

Вторая проблема заключается в возникновении конфликтов управления – т.е. при необходимости выполнить условный переход. Данное событие является очень разрушительным для конвейерной обработки. Для его обработки не только производится приостановка, но и также сброс состояния конвейера, инициализация нового архитектурного состояния и заполнение конвейера. Зачастую, обработка перехода приводит к потерям сотен тактов на перенастройку.

2) Предсказание переходов

Данный метод используется для раннего вычисления условий переходов с помощью специальных аппаратных средств, которые пытаются «предсказать» на основе накопленной истории какую следующую инструкцию необходимо обработать. Если предсказание оказалось верным, то конвейер продолжает функционирование в текущем состоянии. В противном случае, осуществляется очистка конвейера от неверных инструкций. С целью минимизировать количество инструкций, которые необходимо очистить, логика вычисления условий переходов перемещается в начало конвейера.

3) Спекулятивные вычисления

Спекулятивные вычисления – это набор методов, направленных на улучшение механизма предсказания переходов, рассмотренного ранее. Суть их заключается в вычислении заранее всей ветки, либо её части, еще до того, как будет известно, выполнится она или нет. Данный подход позволяет существенно снизить задержки переключения контекстов при переходе. Несмотря на свою эффективность, опережающие вычисления создают уязвимость с точки зрения безопасности, т.к. в архитектурном состоянии одновременно присутствуют обе ветки вычислений. С этой уязвимостью столкнулись все крупные разработчики процессоров, позволяя злоумышленникам получить доступ к фрагментам регистров памяти с привилегированным доступом из процесса общего доступа [7].

4) Суперскалярная микроархитектура

Еще один широко распространенный метод повышения характеристики instruction per cycle (IPC) – количества выполняемых инструкций в такт. Он заключается в том, что в состав микроархитектуры процессора вводится несколько блоков АЛУ, несколько блоков сдвига и округления и т.п. Данные блоки располагаются таким образом, что могут работать параллельно. Следовательно, процессор может за один такт выполнять сразу несколько инструкций на всех блоках (называемых также каналами). Данную характеристику еще называют multi-issue (многозадачность).

Совмещение методов конвейерного исполнения команд (временного параллелизма) и суперскалярности (пространственного параллелизма) позволяет добиться впечатляющего уровня производительности. Однако существующие проблемы конвейерной обработки лишь усугубляются при использовании данного подхода. В частности, помимо конфликтов «запись после чтения» (характерных для всех конвейерных процессоров) при использовании метода внеочередного выполнения инструкций возникают также конфликты «чтение после записи» и «запись после записи». Данные конфликты в большинстве случаев привносятся программистом и возникают в случае зависимости по данным между инструкциями.

Разрешение конфликтов в суперскалярной микроархитектуре требует использования либо мощного компилятора, который может эффективно упорядочивать инструкции для обеспечения максимальной пропускной способности конвейера и характеристики IPC, либо существенных трудозатрат программиста для ручной оптимизации кода.

В реальных программах встречается много зависимостей по данным, поэтому суперскалярные процессоры с высоким параметром multi-issue редко могут использовать все свои функциональные блоки полностью. Более того, большое количество функциональных блоков и сложности с организацией байпаса требуют множества дополнительных логических элементов и потребляют значительное количество электроэнергии [6].

5) Внеочередное выполнение инструкций

Данный метод является дополнением к суперскалярной микроархитектуре и используется для повышения характеристики IPC. Достигается это путем введения в процессор специализированной логики, которая осуществляет опережающий просмотр большого количества инструкций для скорейшего обнаружения и запуска тех из них, которые являются не зависимыми по данным. Порядок запуска инструкций при этом не сохраняется, но процессор учитывает все зависимости, что позволяет правильно вычислить результат.

6) Переименование регистров

Ранее было отмечено, что при внеочередном выполнении инструкций суперскалярным процессором могут возникать конфликты «запись после чтения» и «запись после записи». Оба этих конфликта не являются неустранимыми и возникают как результат работы программиста или компилятора. Для устранения данных конфликтов используется метод переименования регистров. Суть его заключается в том, что в состав процессора помимо регистров архитектурного состояния вводятся также неархитектурные регистры, которые напрямую не доступны программисту. При обнаружении процессором одного из этих конфликтов происходит переименование (подмена ссылки) указанного в инструкции регистра на один из свободных неархитектурных регистров.

7) Very Long Instruction Word (VLIW)

Данный метод является разновидностью суперскалярной реализации, который основан на концепции «вычисления с явным параллелизмом машинных команд» (explicitly parallel instruction computing – EPIC) [8]. Загрузка вычислительных ресурсов возлагается управляется компилятором, что позволяет снизить аппаратные расходы. Получаемая таким образом программа реализует статическое распределение ресурсов. Данный подход также имеет свои недостатки. Во-первых, отсутствует обратная совместимость программ в случае обновления архитектуры. Во-вторых, построение эффективного VLIW-компилятора – это NP-полная задача, для решения которой на сегодняшний день существуют только эвристические методы.

8) SIMD инструкции

Данный способ обработки позволяет одной инструкции обрабатывать сразу серию данных. Этот метод эффективен, когда входные данные имеют небольшую размерность и могут быть упакованы в длинное 32 или 64 разрядное слово. Сама инструкция выполняется на одном из вычислительных блоков над этим словом (словами), интерпретируя их как несколько пар входных данных. Для этого требуется небольшая модификация вычислительных блоков. Например, такую технику используется в цифровых сигнальных процессорах (ЦСП) Texas Instruments для выполнения двух операций сложения 32-битных чисел на 64-битном АЛУ.

1.1.2.3 Параллелизм на уровне потоков

Основные методы реализации параллелизма уровня потоков [9]:

1) ВременнАя многопоточность

При таком методе реализации многопоточности процессор хранит состояния сразу нескольких потоков. Однако в каждый такт может выполнять только один из них. Существует два основных подвида временной многопоточности: крупнозернистая и тонкозернистая.

Крупнозернистая многопоточность характеризуется тем, что в конвейере процессора исполняется только один поток в течение достаточно продолжительного промежутка времени. Когда этому потоку необходимы данные или спустя заданное количество тактов, процессор сохраняет состояние потока и переключается на другой поток, данные для которого уже готовы.

При тонкозернистой реализации многопоточности процессор переключается между потоками каждый такт. Таким образом, гарантируется исполнение всех потоков ценой замедления исполнения каждого отдельного. Общая пропускная способность увеличивается. Данная реализация требует, чтобы потоки были слабо взаимосвязаны или не связаны вовсе.

2) Одновременная многопоточность

Данный метод организации многопоточности реализуется на конвейерных суперскалярных процессорах. Он позволяет решить проблему низкой IPC суперскалярных процессоров путем исполнения одновременно нескольких различных потоков, тем самым минимизируя эффект зависимостей инструкций по данным. Преимуществом данной реализации

многопоточности является существенно более высокая производительность, в то время как временная многопоточность имеет более низкое энергопотребление и, следовательно, тепловыделение. Известным примером реализации одновременной многопоточности является технология Hyper-threading, используемая в процессорах компании Intel.

1.1.2.4 Потоковая модель вычислений

Появление потоковых архитектур связано с развитием архитектур параллельных вычислительных систем. Поэтому, класс потоковых архитектур является подмножеством класса архитектур параллельных вычислительных систем. Модель потока данных [10-13] была предложена как альтернатива последовательной модели вычислений фон Неймана.

Фундаментальные принципы потока данных были разработаны Джеком Деннисом (Jack Dennis) в начале 1970-х годов. Модель потока данных отменяет два свойства модели фон Неймана – счетчик команд и глобальную обновляемую память, - которые стали узким местом в использовании параллелизма [14]. Правило вычислений, известное также как «правило срабатывания» модели потока данных, определяет условие исполнения команды. Основное правило срабатывания, общее для всех систем потока данных, заключается в следующем: команда считается готовой к исполнению, когда ей доступны все входные операнды, необходимые для ее исполнения. Говорят, что команда запущена на исполнение, если для нее выполняется это условие. Результатом запуска команды является потребление значений входных операндов и генерация значений выходных операндов.

1.2 Анализ потоковой модели вычислений и проблем ее реализации

1.2.1 Принципы потоковой модели вычислений

Большинство ранних проектов в мире по разработке прототипов потоковых архитектур были основаны на работе Dennis [15]. В рамках данной работы автор заложил фундаментальные принципы нового языка программирования, который предназначен для описания вычислительных процедур, основанных на концепции управления потоком данных. Согласно этой концепции, вычислительный процесс организован таким образом, что выборка и исполнение очередной инструкции осуществляется при условии готовности требуемых для нее данных. Данный подход позволяет отказаться от использования счетчика команд и глобальной обновляемой памяти. Каждая инструкция устанавливает однозначное соответствие каждому конкретному набору входных данных – набор выходных данных.

Потоковая модель вычислений – это абстрактная модель вычислений, которая реализует вычислительный процесс, запрограммированный с помощью языка потокового программирования. Данной модели характерны следующие фундаментальные принципы:

1) Вычислительный процесс представляется в виде направленного графа, в котором вершины обозначают инструкции, а дуги – перемещаемые между инструкциями данные;

2) Данные снабжаются специальными контейнерами – *токенами*, которые перемещаются по дугам графа;

3) Условием исполнения узла графа, называемым также «правилом срабатывания», является наличие токенов на **всех** входных дугах;

4) Результат выполнения узла графа – это чистая функция входных значений, т.к. гарантируется отсутствие неявных взаимодействий между узлами посредством побочных эффектов (например, через разделяемую память);

5) Следствием п.п. 4) является свойство потоковой программы, которое заключается в преобразовании каждого единственного набора входных данных в ровно один набор выходных данных. Этот принцип также называется *правилом однократного присваивания*. Потоковый граф, который обладает данным свойством называется *хорошо-структурированным (well-behaved)*;

6) Локализованность вычислений –инструкции в потоковом графе не имеют «длинных» зависимостей по данным. Другими словами, каждый токен является локальным и существует в рамках своего фрагмента потокового графа. Связь с другими фрагментами графа осуществляется только посредством входных и выходных дуг.

Рассмотренные принципы формируют два ключевых свойства потоковой модели вычислений: естественная поддержка параллелизма и детерминизм. Под естественной поддержкой параллелизма понимается возможность одновременного исполнения всех независящих между собой по данным узлов потокового графа (мелкозернистый параллелизм). Детерминизм потоковой модели заключается в том, что, независимо от порядка исполнения узлов графа, гарантируется получение корректного результата вычислений.

В современных фон-неймановских архитектурах потоковые принципы нашли отклик в виде принципа внеочередного исполнения команд, где окно исполнения линейно и последовательно продвигается в соответствии с принципами архитектуры, однако в рамках данного окна, инструкции могут быть исполнены в порядке готовности данных.

1.2.2 Структурные элементы потоковой модели вычислений

Аналогично другим абстрактным моделям вычислений – потоковая модель включает в себя набор инструментов, обеспечивающих ей эквивалентную вычислительную мощность. К данным инструментам относятся: множество допустимых инструкций (система команд); множество допустимых атомарных типов данных; способы построения сложных структур данных; средства поддержки условных переходов и циклических вычислений; средства поддержки определяемых программистом функций; способы организации памяти. В отличие от

классической модели вычислений и архитектуры, предложенных фон-Нейманом, реализация данных инструментов в потоковой модели имеет свои специфические особенности.

Система команд и организация ветвления

Основой потоковой модели вычислений, как уже было отмечено ранее, является направленный потоковый граф, удовлетворяющий критерию *хорошей структурированности*. Как показано в работах [11, 15] данному критерию удовлетворяют все ациклические потоковые графы для арифметических и логических операций. Поэтому, система команд потоковой модели включает в себя минимально необходимый набор арифметических и логических операций аналогичный другим абстрактным моделям. Операции условного перехода и организация циклических процедур нарушают условия критерия структурированности потокового графа. Для преодоления данного ограничения в состав системы команд необходимо было ввести специализированные операторы, названные авторами классических работ – *switch* и *merge*.

Оператор *switch* принимает на вход анализируемое значение и управляющий сигнал, который имеет логическое значение. В зависимости от значения управляющего сигнала *True* или *False*, оператор *switch* порождает ровно один токен, который он отправляет на один из выходов – *True* или *False* соответственно. Оператор *merge*, получает на входы логические значения True или False от каждой из веток вычислений и осуществляет выбор одного из них по значению входного управляющего сигнала, объединяя тем самым ветки вычислений.

Сами по себе данные операторы не являются *well-behaved*, однако, их применение позволяет строить потоковый графы, удовлетворяющий данному критерию. Существует еще одна особенность применения данных операторов при построении циклических программ. Если тела циклов достаточно большие или не зависят друг от друга по данным, то, в силу естественной поддержки параллелизма в потоковой модели, в процессе вычислений может быть создано большое количество графов тел цикла, каждое из которых будет иметь свой контекст номера итерации. Данное поведение также называют *динамическим разворачиванием циклов*.

Типы и структуры данных

Исходное описание потоковой модели вычислений включает в себя следующие атомарные типы данных: логический (boolean), целые числа (integer), действительные числа (real) и строки (strings). Построенная таким образом потоковая модель вычислений обладает свойством Тьюринг-эквивалентности, как показано в работе [16]. Однако, ее применение было бы в существенной степени ограниченным, в силу отсутствия поддержки структур данных.

В качестве решения данной проблемы авторами в [15] было предложено рекурсивное построение сложных структур данных на основе атомарных. Для этого структура данных должна быть представлена в виде конечного множества пар вида:

$$[\langle s_1 : v_1 \rangle, \dots, \langle s_k : v_k \rangle] \quad (1.1)$$

где s_i – это селекторы, которые могут иметь тип «целые» или «строка», а v_i – это данные, которые могут быть либо атомарного типа, либо структурой данных. Пустое множество также является структурой данный и обозначается специальным символом nil. Полученная структура данных несет в себе некоторую избыточность. Поэтому в работе [11] она была упрощена до множества из двух пар с селекторами *first* и *rest* соответственно.

Построение этой структуры данных осуществляется с помощью специальных операторов в системе команд: *cons* – объединяет пару данных и снабжает их селекторами *first* и *rest*; *append* – является расширением *cons* для построения массивов. Очевидно, что при рекурсивном построении полученные структуры данных могут быть очень большими, поэтому потоковая модель должна предусматривать возможность передачи по дугам токенов большого объема. На практике, как правило, токены несут в себе ссылки на реальные данные, что позволяет значительно уменьшить требования по памяти. Для доступа к элементам структуры данных в систему команд вводится оператор *select*, который осуществляет выборку данного с требуемым значением селектора.

Функции и процедуры

Возможность определения и вызова функций, определенных программистом, является необходимым элементом любого современного языка программирования. На уровне потокового графа функция – это инкапсуляция фрагмента этого графа в некоторую абстракцию. К данной абстракции применяются те же требования, что и для потокового графа. Во-первых, инкапсулированный фрагмент графа должен удовлетворять критерию *well-behaved*. Во-вторых – исходный потоковый граф после подстановки функции должен также удовлетворять критерию *well-behaved*.

Таким образом, нерекурсивные функции могут быть поддержаны на этапе компиляции. Для более обобщенной поддержки функций в потоковой модели необходимо ввести в систему команд оператор *apply*. Данный оператор получает на вход абстракцию функции, которая описывает некоторый потоковый граф, и набор входных токенов, после чего осуществляет вызов этой функции на заданном наборе токенов.

Реализация оператора *apply* сопряжена с рядом проблем. Например, в какой момент времени необходимо создавать потоковый граф, который соответствует вызываемой функции (когда пришли все входные токены или только один конкретный)? Кроме того, независимо от выбранной реализации, архитектура должна поддерживать динамическое расширение исполняемого потокового графа и методы маршрутизации токенов на дуги вновь созданного

графа. Также может возникнуть потребность повторного использования копии созданного графа, следовательно, необходимо поддерживать разделение контекстов для различных вызовов.

Основной цикл вычислений

Организация основного цикла исполнения инструкций в потоковой модели вычислений является прямым следствием принципов ее функционирования. Таким образом, можно выделить следующие этапы цикла:

1. Определение готовности узла графа к исполнению (по готовности токенов).
2. Определение инструкции, которую необходимо выполнить.
3. Вычисление результата.
4. Формирование токена результата.

По утверждению авторов в [11] данный цикл является базовым для любой потоковой машины. Однако, его реализация в конкретной архитектуре несет в себе большую гибкость.

В отличие от классической модели фон-Неймана, исполнение инструкций в потоковой модели зависит от прибытия операндов и соответствующих им токенов. Следовательно, задачи управления хранилищем токенов и планирования выполнения инструкций тесно связаны в любой потоковой архитектуре. Другим важным наблюдением является тот факт, что потоковые графы эксплуатируют два вида параллелизма в процессе исполнения инструкций: пространственный параллелизм (любые два узла могут быть выполнены одновременно при условии отсутствия зависимости по данным); временной параллелизм (заключается в конвейеризации независимых вычислительных потоков через один и тот же граф).

1.2.3 Характерные проблемы реализации потоковых архитектур

Проблема организации памяти токенов

В предыдущем разделе уже были рассмотрены некоторые проблемы реализации потоковой модели вычислений в конкретной архитектуре. Однако, наиболее существенной является проблема разработки механизма хранения и обработки токенов. Потоковая модель вычислений подразумевает, что дуги графа представляют собой FIFO неограниченного размера, что, очевидно, невозможно на практике. Поэтому были разработаны и исследованы два основных подхода к решению проблемы организации памяти токенов.

Первый подход называется *статический поток данных*. Основное ограничение в статической потоковой архитектуре заключается в том, что не более одного токена может быть размещено на дуге потокового графа в каждый момент времени. Для удовлетворения данного ограничения необходимо также модифицировать «правило срабатывания» таким образом, чтобы активация узла графа была возможна только в том случае, когда все выходные дуги узла свободны. Введение данных ограничений позволяет значительно снизить накладные расходы для памяти токенов и выделять ее в объеме, необходимом непосредственно для выполнения

конкретного потокового графа. Однако, для обеспечения корректной работы вычислительной машины на основе статической потоковой архитектуры токены на дугах должны быть физически упорядочены. Кроме того, столь жесткое ограничение приводит к низкой реентерабельности потокового графа и снижению производительности системы в целом [17].

Второй подход был назван *динамический поток данных* или *тегированные токены*. Суть данного подхода заключается в предоставлении возможности активации одного и того же узла потокового графа одновременно. Данный эффект достигается за счет динамического прикрепления к токенам дополнительной информации, называемой тегом. Тегированные токены несут в себе информацию: выполняемом в данный момент потоковом графе, адресе выполняемой инструкции, контексте выполняемой функции, контексте текущей итерации цикла. «Правило срабатывания» активируется при условии совпадения тегов у двух токенов.

Такая организация вычислительного процесса позволяет поддерживать только логический порядок хранения токенов на дугах потокового графа, независимо от физического времени их появления. Основная проблема реализации данного подхода заключается в необходимости создания очень большого по объему и скорости работы ассоциативного хранилища памяти, которое обеспечивает поиск совпавших пар тегов [17]. Авторы в работе [11] дают оценку физического объема хранилища порядка 100 тысяч токенов, а также необходимого для работы с ним виртуального пространства, адресуемого 40 или более битным адресом.

Проблема построения структур данных

В потоковой модели существует определенная проблема при работе со структурами данных. Так как в модели отсутствует глобальная разделяемая память, то все токены (согласно принципам функционального программирования) обладают свойством неизменяемости. С одной стороны, это позволяет избегать типовых проблем гонок и тупиков в параллельной вычислительной системе. С другой – приводит к необходимости создания новых токенов и соответствующих им структур данных при попытке изменения уже существующих токенов.

Таким образом, вычислительная система, построенная на основе потоковой модели, должна иметь значительный объем активной памяти токенов для обеспечения корректного функционирования. Кроме того, постоянная генерация новых токенов и структур данных сопряжены с существенными временными затратами и общим снижением производительности вычислительных систем потоковой архитектуры.

Проблема распределения вычислительной нагрузки

Аналогично традиционным архитектурам вычислительных систем в потоковых архитектурах существует проблема распределения фрагментов исполняемой программы между их вычислительными блоками. Очевидно, что оптимальное распределение задач между вычислительными блоками оказывает прямое влияние на общий уровень производительности

потоковой архитектуры в целом. Однако данная проблема не имеет тривиального решения, т.к. в работе [18] показано, что это NP-полная задача. Основными задачами, которые необходимо решать при распределении нагрузки являются: *разбиение* (графа на подграфы) и *назначение* (каждого подграфа для исполнения на конкретном вычислительном модуле).

Проблема конвейеризации последовательных вычислений

В работе [11] авторы отмечают, что важной характеристикой вычислительной системы на основе потоковой архитектуры является использование параллелизма для сокрытия задержки коммуникации между вычислительными блоками. Подобное также верно и для сокрытия задержек конвейера для выполнения инструкций, которые имеют зависимость по данным между собой. Однако данное утверждение корректно только при условии, что на каждом вычислительном блоке загружено достаточное количество активных для исполнения пакетов инструкций (для которых активировалось «правило срабатывания»).

Потоковая модель вычислений подразумевает эффективную реализацию мелкозернистого параллелизма и является функциональной. Это означает, что в рамках абстрактной модели вычислительные блоки не должны иметь состояния (т.е. регистров или памяти). Тогда в результате вычислений всегда будут формироваться токены, которые будут отправляться в коммутационную среду и, тем самым, проходить все стадии конвейера для участия в последующих вычислениях.

В случае, если выполняемая задача обладает низкой степенью параллелизма или вообще является последовательной, то даже на одном вычислительном блоке готовые для исполнения пакеты инструкций будут отсутствовать. Следовательно, время выполнения очередной инструкции будет определяться глубиной конвейера, что приведет к крайне низкой производительности при решении последовательных задач.

Проблема поддержки высокопроизводительной микроархитектуры

Для достижения наилучшей производительности современные процессоры реализуют сложную микроархитектуру. Высокопроизводительная микроархитектура обладает высокой пропускной способностью инструкций, низким временем простоя вычислительных блоков и высокой степенью наполнения конвейера. Достигается это за счет реализации рассмотренных в разделе 1.1.2 механизмов. Концепция внеочередного исполнения команд является частью потоковой модели вычислений и поэтому поддерживается интуитивным образом. Более того, именно исследования в области потоковых архитектур привели к ее внедрению в процессоры традиционной архитектуры. Однако реализация в потоковых архитектурах других способов извлечения параллелизма уровня инструкций сопряжена со множеством проблем.

Условные переходы в потоковой модели вычислений обрабатываются совершенно иначе в сравнении с традиционными архитектурами. Потоковый граф требует вычисления обеих

ветвей с последующим выбором полученного результата. Поэтому невозможно предсказать переход и заранее подготовить пакеты инструкций, которые могли бы быть выполнены из-за того, что их активация возможна только после формирования токена результата перехода.

Суперскалярные вычисления подразумевают выполнение на одном вычислительном блоке более одной инструкции на каждом шаге, которые могут использовать в качестве источников данных внутренние регистры. Также для выполнения некоторых стандартных инструкций требуется сохранять результат накопления в каком-либо регистре состояния. Как было отмечено ранее, вычислительные блоки «классической» потоковой архитектуры не должны иметь состояния, что противоречит предыдущему наблюдению. Кроме того, узлы потокового графа могут нести в себе только одну инструкцию.

Тэгированные данные несут в себе достаточно большой объем избыточной информации, необходимой для однозначной идентификации токена. Поэтому, инструкции потокового графа, которые содержат описание ожидаемых для обработки токенов также имеют большой объем. Увеличение размера пакета инструкций для организации суперскалярных вычислений может стать непреодолимым препятствием для физической реализации таких инструкций.

Проблема обработки константных данных

Согласно принципу *однократного присваивания* потоковой модели вычислений токены на входе узла потокового графа поглощаются (т.е. удаляются), а результат выполнения инструкции формирует новые токены. Таким образом, токены, которые несут в себе константные данные, должны каким-то образом сохраняться и циркулировать в потоковой архитектуре. В работе [11] приводится описание решения, в котором константные данные кэшируются в программной памяти и извлекаются из нее при выполнении инструкции. При такой организации константные данные уже перестают быть токенами и участвовать в активации firing-rule. Кроме того, данное решение усложняет кодировку инструкций.

1.2.4 Реализация структур данных в потоковых архитектурах

С целью обеспечения функциональности создания, хранения и обработки сложных структур данных (таких как массивы) при условии сохранения основных принципов потоковой модели вычислений был разработан ряд подходов решения рассматриваемой проблемы. К двум основным подходам представления массивов и аналогичных им структур данных относятся: *прямой* и *непрямой* доступ [19]. В схеме *прямого доступа* элементы массива интерпретируются как отдельные токены, что позволяет убрать необходимость реализации массива, как специальной структуры данных. В свою очередь, в схеме *непрямого доступа* массивы хранятся в специальной памяти, а доступ к ним осуществляется явным образом с помощью специальных операторов «чтение» и «запись».

1.2.4.1 Схема прямого доступа – переименование токенов

Организация схемы *прямого доступа* к элементам структуры данных типа массива осуществляется с помощью механизма *переименования токенов*. Суть данного механизма заключается в переименовании тегов, ассоциированных с каждым токеном в массиве, за счет чего достигается прямая пересылка данных массива его потребителю. Это позволяет преобразовать массив в множество индивидуальных токенов. Реализация такой схемы требует ввода в вычислительный процесс специальной функции переименования, которая обычно не известна заранее и трудно получаема в процессе компиляции [19]. Поэтому ее вычисление симулируется с помощью специальной техники.

Схема прямого доступа совместима с основными принципами потоковой модели вычислений. Следовательно, ее использование приводит к построению более однородной архитектуры и дает более высокую производительность, в силу того что нет нужды обрабатывать токены с данными с помощью специального контроллера структур данных. Тем не менее, для данной схемы характерны два узких места. Во-первых, поддержка концепции выполнения инструкций, которые не имеют побочных эффектов, приводит к тому, что структура данных целиком должна быть передана от одного узла графа к другому или даже размножена для множества узлов. Это приводит к значительному росту требований к объему хранилища токенов в блоке их сопоставления. Во-вторых, далеко не всегда можно трактовать нотацию массива как единой сущности, которую можно преобразовать в множество отдельных токенов.

Далее будут рассматриваться *схемы непрямого доступа*.

1.2.4.2 Куча

В статической потоковой машине, разработанной в MIT, массивы были представлены в виде направленного ациклического графа, который хранится в отдельной вспомогательной памяти [20]. Такой ациклический граф (также называемый *кучей*) всегда образует древовидную структуру с одним корневым элементом, которая обладает свойством, что каждый лист графа достижим из корня. Массивы в рамках данной схемы интерпретируются как указатель на вершину дерева. Это позволяет значительно сократить объем пересылаемых между узлами токенов (вместо всей структуры данных целиком только указатель на нее). Основная идея использования кучи заключается в исключении излишнего копирования путем разрешения совместного использования одних и тех же вложенных структур несколькими структурами данных, что является несомненным преимуществом использования *кучи*.

Тем не менее, представление массивов в виде деревьев не лишено недостатков. Наиболее очевидным недостатком является долгое время доступа к элементам дерева. Как известно, линейно организованные массивы имеют константное время доступа. Древовидная же структура имеет O(log(n)) время доступа к своим элементам. Другим недостатком является то,

что операции могут быть выполнены только над отдельными элементами массива, а не всей структурой целиком, из-за низкого уровня, на котором применяется модель секвенирования потока данных. Поэтому, выполнение двух независящих друг от друга операций над массивами все равно будет осуществлено последовательно. Структуры с таким поведением называют *ограниченными* или *прямолинейными*.

1.2.4.3 I-структуры

I-структуры впервые были предложены Арвином и Томасом, в качестве основного механизма построения структур данных для потоковой архитектуры с тегированными токенами [21, 22]. I-структуры – это асинхронные структуры данных, которые ведут себя похожим образом с массивами. Отличительной особенностью рассматриваемой структуры данных является наличие ограничений на создание и доступ к ним. А именно, I-структура **создается ровно один раз** и каждый ее элемент может быть **записан ровно один раз**. Асинхронная природа структуры позволяет исполняемой программе пытаться получить доступ на чтение элементов до того, как они будут записаны. Структуры с таким поведением называются *неограниченными* или *непрямолинейными*. На рисунке 1.2 (рисунок 19 из [17]) приводится пример взаимодействия с I-структурой.

Использование I-структур по сравнению с древовидными структурами обеспечивает более высокую скорость доступа и более низкие накладные расходы по памяти, но в ряде случаев может плохо сочетаться с решаемой задачей.

Достигается асинхронное поведение за счет реализации концепции «потребление до завершения формирования». Для этого в состав элементов структуры данных вводят специальные биты, называемые *битами готовности*. Эти биты задают три допустимых состояния элемента структуры:

- PRESENT – означает, что данное присутствует в ячейке памяти и может быть считано;
- ABSENT – означает, что записи в данную ячейку памяти не производилось с момента ее выделения и попыток чтения не было;
- WAITING - означает, что записи в данную ячейку памяти не производилось с момента ее выделения, но попытки чтения были. В этом состоянии запросы на чтение откладываются в специальный связный список из *deferred read* запросов.

Данный подход был разработан в процессе улучшения базовой Манчестерской машины и был назван Extended MANchester (EXMAN) computer [23]. Данная машина снижает избыточность памяти путем хранения в токенах указателя на начало массива и увеличивает скорость доступа за счет использования традиционных структур со случайным доступом. Исходный массив данных хранится последовательным образом в виде структуры со случайным константным временем доступа.

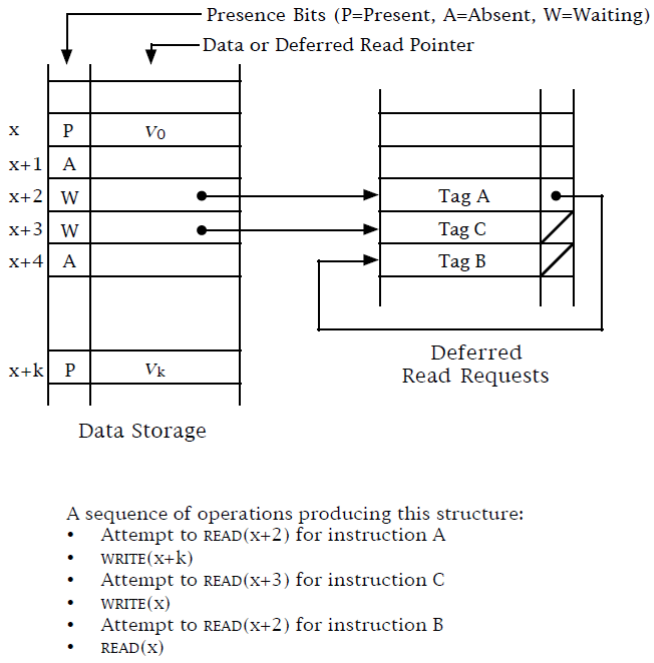


Рисунок 1.2 – Пример I-структуры (рис. 19 из [17])

1.2.4.4 Усовершенствованный Манчестеровский подход

Чтобы выполнить операцию APPEND, создается новый узел, который содержит новое значение тегированного токена и модифицируемый индекс. Таким образом, для корректного выполнения операцию APPEND необходимо снабдить тремя аргументами: ссылкой на массив, индексом обновляемого элемента и значением. Последовательность операций APPEND над массивом формируют структуру связного списка из новых элементов.

Операция SELECT снабжается двумя аргументами: ссылкой на структуру данных; индексом считываемого элемента. Выполнение операции SELECT осуществляется путем обхода сперва связного списка модифицированных элементов. В случае, если в ходе обхода в списке находится совпадающий индекс, возвращается соответствующее значение. Иначе, данное считывается из исходного массива по заданному индексу.

Таким образом, подход EXMAN избегает излишнего копирования за счет совместного использования динамических указателей и структур данных с константным случайным доступом. Это позволяет снизить накладные расходы по памяти. Однако, по мере роста числа измененных элементов, скорость выполнения операции SELECT будет снижаться, значительно уменьшая производительность.

1.2.4.5 Гибридная схема

Гибридная схема формирования структур данных была предложена Ли и Харсоном [24]. Основная идея данной схемы заключается в сопоставлении концептуальному массиву специальный шаблон, называемый *структурный шаблон*. Это позволят минимизировать

копирование за счет размещения в новом массиве только измененных элементов. Каждый массив представляется *гибридной структурой*, которая состоит из структурного шаблона и вектора элементов массива. Гибридная схема хорошо сочетается с I-структурами.

Существует проблема выполнения операции APPEND. Выполнение данной операции приводит к созданию абсолютно нового массива. Это является удовлетворительным поведением в случае, если: размер массива небольшой или выполняемая программа подразумевает обновление всех элементов массива. Далеко не всегда эти условия выполняются. Поэтому гибридная схема позволяет разбивать большие массивы на множество более мелких массивов, каждый из которых интерпретируется как гибридная схема. С помощью таблицы доступа все небольшие массивы связываются в единую структуру данных. Это обеспечивает константное время доступа ко всем элементам.

1.2.5 Балансировка вычислительной нагрузки

Существует две основные проблемы распределения вычислительной нагрузки, которые могут быть обозначены как *разбиение* и *назначение* [25]. Разбиение – это деление алгоритма на процедуры, модули и процессы. Назначение – это распределение, полученных в ходе разбиения элементов, по процессорам. Конечная цель заключается в разработке схемы эффективной балансировки нагрузки, которая будет избегать конкуренции между процессорами и уменьшать взаимодействия в коммуникационной среде. Однако обе указанные задачи находятся в прямом конфликте между собой, что делает решение данной проблемы нетривиальным [17]. Поэтому, создание единственной схемы балансировки нагрузки, которая смогла быть покрыть широкий спектр потоковых архитектур – чрезвычайно сложная задача. В работе [25] рассматриваются два основных подхода балансировки подзадач потокового графа: статический и динамический.

При статической балансировке нагрузки задачи распределяются на этапе компиляции за счет использования глобальной информации о структуре вычислительной системы и исполняемой программы. Эта процедура выполняется один раз. Главным недостатком статической балансировки является ее неэффективность в случае, если оценка характеристик времени исполнения неточна или некорректна.

С другой стороны, динамическая балансировка нагрузки основана на измерениях загрузки процессоров в реальном масштабе времени в зависимости от поведения программы. Назначение задач происходит для самых низко нагруженных процессоров. Основным недостатком динамической схемы является существенная избыточность, связанная с оценкой степени загруженности процессоров и распределением задач в реальном времени.

Ранее было отмечено, что задача эффективной балансировки нагрузки является NP-полной. Поэтому был разработан ряд эвристических алгоритмов, основанных на планировании списка критических путей [24]. Основной идеей данного алгоритма является разбиение

потокового графа на множество линейных подграфов, которые затем могут быть выполнены параллельно. Для этого каждый узел потокового графа взвешивается параметром максимального времени исполнения подграфа, который начинается в этом узле и заканчивается в выходном узле. Полученный вес и называется критическим путем. На основе полученных критических путей формируется упорядоченный список узлов, которые затем динамически распределяются по процессорам в порядке убывания их весов.

Основной нерешенной проблемой балансировки нагрузки является проблема организации и обработки динамического параллелизма. Например, концепция динамического разворачивания циклов, требует реализации механизмов поддержки динамического параллелизма. Однако, одиночный процессор не допускает возможности одновременного исполнения разных копий одного и того же узла. Поэтому, распределение вычислительной нагрузки без использования какого-либо механизма предиктивных вычислений приводит к невозможности использования параллелизма в полном объеме.

Другой важной проблемой при балансировке нагрузки является степень гранулярности подзадач. Теоретически потоковая модель вычислений поддерживает мелкозернистый параллелизм. Но на практике достижение максимальной степени использования мелкозернистого параллелизма не достижимо. Задержки межпроцессорных взаимодействий и затраты на планирование задач начинают превышать потенциальный выигрыш от параллельных вычислений. Поэтому, использование более крупнозернистого параллелизма имеет свои преимущества. Следовательно, вопрос следует сформулировать так: какова подходящая степень гранулярности? Ответ на него кроется в нахождении баланса между накладными расходами на организацию параллельных вычислений и, собственно, параллельным исполнением.

1.2.6 Комбинированные потоково-фон-неймановские архитектуры

Интерес к комбинированным (гибридным) потоково-фон-неймановским (ПФН) архитектурам возник практически одновременно с появлением потоковой модели вычислений. Гибридные модели разрабатывались в стремлении объединить положительные особенности фон-неймановских и потоковых моделей. По мнению авторов [26], кроме общеизвестных недостатков фон-неймановской организации вычислений, "одной из ее главных сил является универсальный характер ее программной организации. Она универсальна не только в смысле эквивалентности по Тьюрингу, но способна эффективно поддерживать спектр стилей программирования и эффективно моделировать многообразие структур алгоритмов".

1.2.6.1 Объединенная одноуровневая ПФН-архитектура

В [26] рассматривались три главных класса-конкурента организации программ: потока данных, многопоточное управление на базе фон-неймановских принципов и различные типы

редукций. Для получения верной организации программы общего назначения необходимо интегрировать концепции, лежащие в основе этих трех моделей. В работе [27] описываются результаты разработки модели организации программы для параллельной компьютерной архитектуры, управляемой данными, которая интегрирует концепции чистых вычислений потока данных с концепцией вычислений многопоточного управления. Также описывается аппаратное устройство, разработанное для поддержки этой организации программ.

В объединенной модели токен может рассматриваться как переносящий один из трех возможных типов операнда - значение, название или пустой указатель. *Токен данных* может сообщать частичный результат или как прямое значение, или как имя, используемое для обращения к ячейке памяти, хранящей частичный результат; *токен управления* можно считать токеном данных, несущим нулевое значение. Когда набор токенов заканчивается и активизирует команду, только данные от токенов данных проходят к команде как входы. Команды, активизированные исключительно набором токенов управления, не принимают никаких входных значений от токенов.

1.2.6.2 Гибридная ПФН-архитектура с эффективной обработкой последовательного кода

Одна из проблем потоковых архитектур - низкая производительность реализации последовательного кода. На коротких участках последовательного кода, имеющих все необходимые данные в локальной высокоскоростной памяти (регистрах и кэш), любые накладные расходы на синхронизацию расточительны. В связи с этим в большинстве гибридных моделей добиваются сокращения издержек, применяя некую форму кластеризации: некоторые последовательности узлов объединяются в потоки, которые выполняются последовательно и не несут расходов, связанных со сравнением токенов.

Некоторые из этих гибридных технологий [28] сохраняют понятие маркера и напоминают традиционные теговые машины, за исключением кластеризации узлов в потоки. Другие, так называемые "потоковые архитектуры без потока данных" [29], сохраняют выполнение, управляемое данными, но выбирают все данные из общедоступного ЗУ.

1.2.6.3 Гибридная P-RISC архитектура

Эта модель [30] была предложена для того, чтобы иметь возможность выполнять традиционные программы, используя преимущества параллельной обработки на базе потокового принципа [31]. Другая идея состояла в том, чтобы эффективно выполнять последовательные шаги вычислений в компьютере потока данных с использованием недорогого механизма управления, составляющего основу фон-неймановских машин. Для реализации этой цели нужно было решить три проблемы.

1) изменить реализацию RISC-процессора, чтобы сделать его пригодным для многопоточной обработки [32];

2) расширить систему команд существующего RISC-процессора, чтобы облегчить обработку множественных потоков; вводятся три дополнительные команды: FORK (ветвление), JOIN (объединение) и START (начало), позволяющие поддерживать мелкозернистый параллелизм потока данных;

3) дополнить RICS-процессор памятью I-структур.

Команды FORK и JOIN позволяют моделировать мелкозернистный асинхронный параллелизм выполнения потока данных [33]. Команда START - дальнейшее расширение, облегчающее связь между фреймами различных процессоров в машине. Детали выполнения этих трех команд и их реализации даны в [33].

Информации, приведенной в [30], недостаточно для полного понимания функционирования предложенной модели. Поэтому остается только поверить авторам, что подобная архитектура позволяет эксплуатировать мелкозернистый параллелизм при сохранении эффективности последовательных вычислений.

1.2.6.4 Комбинированные двухуровневые архитектуры

В описанных выше гибридных архитектурах рассматривались, по существу, универсальные одноуровневые модели, связывающие коммуникационной сетью идентичные гибридные (data-flow/control-flow) процессоры.

Другой подход − использование двухуровневых моделей с ведущим фон-неймановским процессором на верхнем уровне и многими потоковыми процессорами на нижнем уровне, связанных между собой коммуникационной сетью [34-38]. На ведущий процессор могут быть возложены функции:

- связи многопроцессорной потоковой системы с внешним миром;

- интерфейса между стандартным программным обеспечением и многопоточным вычислительным процессом, базирующимся на потоковой парадигме;

- вычислителя для последовательных частей программы;

- устройства обработки исключительных ситуаций в многопроцессорной сети;

- устройства компоновщика самодостаточных капсул в случае ГАРОС для их исполнения на нижнем (операционном) уровне архитектуры.

Нужно приложить серьезные усилия для того, чтобы ведущий процессор не стал узким местом всей гибридной ПФН-модели. Один из способов решения этой задачи - организация многопоточного фон-неймановского процессора, пропускная способность которого соответствовала бы требованиям многопроцессорной потоковой модели на нижнем уровне. Другое возможное (или дополняющее) решение - вовлечение в обработку операций

ввода/вывода (при их высокой интенсивности) всех (или поднабора) потоковых процессоров при минимальной координации со стороны ведущего фон-неймановского процессора [38].

1.2.7 Позиционирование МПРА в классификации потоковых архитектур

Прототип МПРА – ГАРОС реализуется в виде комбинированной двухуровневой ПФН-архитектуры. Кроме того, потоковые процессоры в его составе, реализуют принципы динамического распределения данных и инструкций. На рисунке 1.3 обозначается позиция ГАРОС в классификации потоковых архитектур.

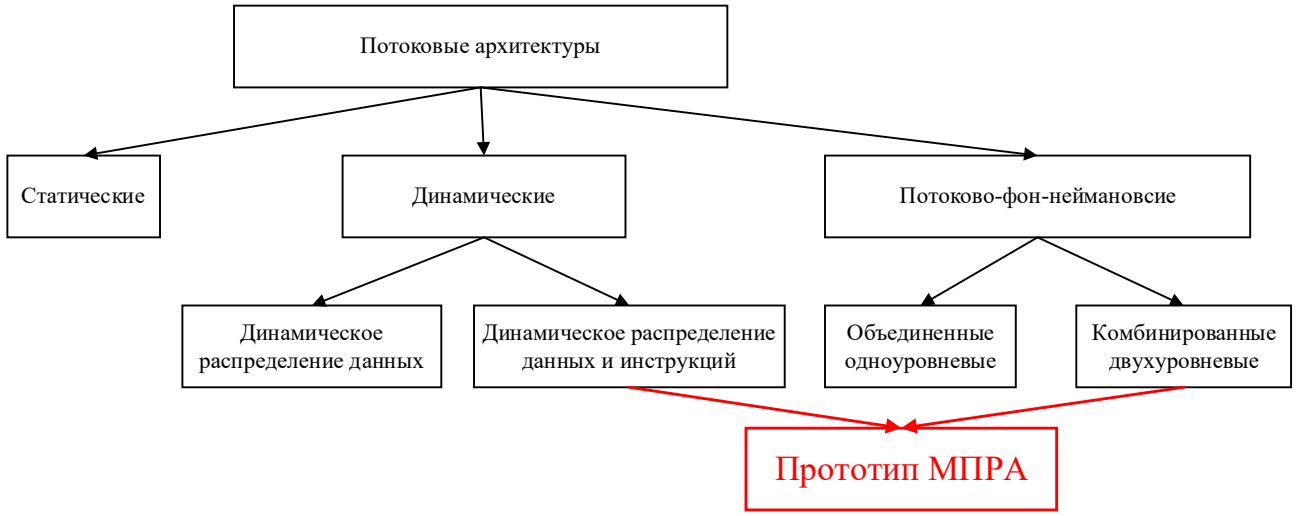


Рисунок 1.3 – Позиция ГАРОС в классификации потоковых архитектур

1.3 Концептуальные основы многоядерной потоковой рекуррентной архитектуры

В данном разделе рассматриваются ключевые принципы рекуррентно-потоковой модели вычислений и многоядерной потоковой рекуррентной архитектуры, которая является одной из возможных реализаций данной модели и предназначена для организации параллельных вычислений ограниченной размерности. В качестве предметной области для испытаний МПРА выбрана область ЦОС, а демонстрационной задачи - задача распознавания изолированных слов (РИС). Выбор данной предметной области и демонстрационной задачи был обоснован в работах [39, 40]. Исследование абстрактной МПРА осуществляется на примере ее прототипа – ГАРОС.

1.3.1 Рекуррентно-потоковая модель вычислений и архитектура на ее основе

Новая рекуррентно-потоковая модель вычислений объединяет в себе ключевые принципы потоковой модели вычислений и рекуррентно-динамической парадигмы вычислений. Рекуррентно-динамическая парадигма вычислений расширяет основной элемент потоковой модели вычислений (потоковый граф) с помощью графодинамического метода. В работе [41] вводятся основные термины и определения элементов данной парадигмы. Для сохранения терминологической чистоты далее приводятся некоторые из этих определений.

Графодинамика – совокупность методов описания и изучения динамических задач, в которых развитие событий связано с изменением представляющих их графов, то есть с процессами, где во времени меняется либо сама алгоритм решения, либо структура задачи.

Графодинамический метод (описания задач) – дискретный метод, сводящийся к рассмотрению совокупности объектов с позиций графодинамики, существенные свойства которых описываются связями между ними в особых точках графовой траектории. При этом рассматриваемые объекты изображаются в местах появления этих точек на графовой траектории «кружочками», называемыми *вершинами*, а связи между ними – линиями (произвольной конфигурации), называемыми *ребрами*.

Динамическая задача – задача, структура которой может меняться во время использования; при этом объектом исследования и реализации служит граф в целом, который представляет собой графовую траекторию.

Статический параллелизм – процесс преобразования исходного (последовательного) алгоритма в параллельный (путем выявления частей, способных выполняться одновременно), выявляемый всегда заблаговременно – до начала программирования и исполнения.

Динамический параллелизм – процесс преобразования исходного (последовательного) алгоритма в параллельный, при котором выявление частей, способных выполняться одновременно, осуществляется в ходе его исполнения. При этом стратегия дальнейшего развития процесса – перехода к следующему шагу – уточняется на каждом текущем шаге, в результате чего процесс развертывается рекуррентно. Этот вид параллелизма также может выявляться заблаговременно и отображаться в алгоритмах.

Рекуррентность (рекуреррентная вложенность) – фундаментальное структурное свойство активных (самодостаточных) объектов, выражающееся в том, что любой такой объект:

- состоит только из множества *подобных* ему объектов младшего ранга;
- всегда является составляющим элементом подобного объекта старшего ранга (подчиняется свойству *иерархической упорядоченности*);
- характеризуется *возвратностью*, т.е. повторяемостью всех свойств объектов (структурных и функциональных) на любых уровнях вложенности.

Самодостаточность – свойство активного объекта (технической или биологической системы), обеспечиваемое наличием в нем всего необходимого для реализации логичного и эффективного поведения в определенных (собственных) целях.

Элемент самодостаточных данных – это машинное слово, состоящее из двух частей (полей) – «*Функция*» и «*Аргумент*». Аргумент сопровождается кортежем подфункций (интегрально – *Функция*), обеспечивающих ему свойство самодостаточности. В

функциональной части слова рекуррентным способом закодирована процедура (последовательность шагов) обработки аргумента.

Основным методом рекуррентно-потоковой парадигмы является *графодинамический метод*, который сводится к построению *рекуррентно-динамических графов*. Суть метода заключается в применении некоторого функционального оператора **F** к текущему поколению графа **g(i)**, в результате которого формируется граф **g(i+1)**, и так далее. Цепочка таких преобразований формирует последовательность графов, которая называется *графовой траекторией*. Другими словами, графовая траектория – это исполняемая программа (подпрограмма), алгоритм которой может быть представлен в виде последовательности графов **{g(n)}** и рекуррентно сворачивается в начальный граф **g(0)**. Данный граф **g(0)** кодирует начальное условие *рекуррентной саморазвертки* и называется *рекуррентной цепочкой*.

Совокупность рекуррентных цепочек является исполняемой программой, называемой *капсулой*. Обобщая графодинамический метод до уровня задачи, получим представление решаемой задачи в виде совокупности капсул – *капсульный стиль программирования*. Каждая капсула является потоковой структурой данных, аналогичной I-структуре. Следовательно, капсула является предлагаемым решением проблемы *построения структур данных* в МПРА. Реализуется графодинамический метод за счет двух основных принципов: самодостаточность и рекуррентность.

Принцип самодостаточности заключается в логическом и физическом объединении элемента данных и соответствующей ему рекуррентной цепочки в элемент самодостаточных данных (ЭСД), называемый также операндом. Множество операндов образуют единый поток данных, в отличие от классической потоковой модели, которая подразумевает наличие отдельного потока данных и отдельного потока инструкций. Данный подход позволяет сократить до теоретически возможного минимума количество этапов обработки инструкций, за счет устранения необходимости постоянной выборки и дешифрации инструкций. Таким образом, принцип самодостаточности эффективно решает проблему *конвейеризации последовательных вычислений*.

Принцип рекуррентности заключается в построении и аппаратной реализации такого оператора **F**, который обеспечит сжатие (рекуррентную свертку) потокового графа алгоритма в граф **g(0)** и распаковку (рекуррентную саморазвертку) графовой траектории с помощью специального устройства Преобразователя тегов (ПТ). В зависимости от того, насколько «удачно» будет построен оператор **F**, объем исполняемой программы может быть сжат вплоть до предельного значения в один операнд. В МПРА реализован универсальный оператор, обеспечивающий развертку рекуррентной цепочки в графовую траекторию длины 3. Таким образом, принцип рекуррентности эффективно решает проблему *организации памяти токенов*

и частично проблему *распределения вычислительной нагрузки* за счет уменьшения общего числа инструкций. Формула (1.2) содержит описание универсального оператора **F**.

$$\boldsymbol{F} = \begin{cases} f(t,k) = \begin{cases} f(t-1,k) - 2^m, \text{ если } f(t,k) > 0 \\ \text{останавливается, если } f(t,k) \le 0 \end{cases} \\ g(t,k) = \begin{cases} g(t-1,k) + 1, \text{ если } f(t,k) > 0 \\ \text{останавливается, если } f(t,k) \le 0 \end{cases} \\ g(0,k) = 0,\ k = \overline{0,\ n-1} \\ f(0,0) = f_0 \\ f(0,k) = g(t_0, k-1),\ k = \overline{1,\ n-1} \end{cases} \quad (1.2)$$

Здесь**:**

- k – номер шага рекуррентной развертки;
- t – номер шага вычислений функций f и g;
- t_0 – номер шага, на котором остановились функции f и g;
- m – размер функционального поля в разрядах;
- $f(t,k)$ – функция преобразования значения функционального поля на шаге преобразований k;
- $g(t,k)$ – функция вычисления нового значения функционального поля на шаге преобразований k.

В дополнение к сжатию инструкций в МПРА реализуется механизм сжатия и упаковки нескольких элементов данных в ЭСД. Это позволяет использовать одну и ту же рекуррентную цепочку для некоторой последовательности обрабатываемых данных исходя из особенности выполняемого алгоритма. В области ЦОС число алгоритмов конечно, знание их особенностей позволяет применять различные методы сжатия данных, что невозможно для произвольного алгоритма. Комбинирование методов сжатия данных и сжатия инструкций в МПРА достигается еще большая эффективность решения проблемы *организации памяти токенов*.

В работе [42] проанализированы принципиальные отличия трех классов моделей вычислений и соответствующих им архитектур. К первому принципиальному классу относятся традиционные фон-Неймановские архитектуры. Для выполнения программы в традиционной архитектуре требуется некоторый объем памяти, которая функционально или физически разделена на две области: программ и данных. Инициатором вычислений является поток инструкций, считываемый из памяти программ. Тогда поток инструкций является первичным, а поток данных – вторичным. Программа-инициатор процесса хранится в памяти инструкций в статическом виде. Данную архитектуру можно классифицировать как Control-Flow/Static (CF/S).

Ко второму принципиальному классу относятся потоковые архитектуры. Для потоковых архитектур (DF – data-flow) сохраняется разделение ресурсов памяти на память токенов и память пакетов (инструкций). Инициатором вычислений является поток данных, источником

которого является память токенов. Токены содержат в себе информацию об исполнительном адресе инструкции (микрокоманды), которая должна быть считана из памяти программ. Полный объем привлекаемых инструкций также хранится в статическом виде. Данную архитектуру можно классифицировать как Data-Flow/Static (DF/S).

К третьему принципиальному классу относится МПРА. Характерной особенностью архитектур данного класса является управление вычислительным процессом единым потоком самодостаточных данных. Также МПРА относится к подклассу динамических потоковых архитектур с тегированными токенами. Теги содержат начальную сжатую информацию (начальный граф **g(0)**), обеспечивающую выполнение требуемой процедуры. Каждый следующий шаг процедуры рекуррентно самоопределяется путем преобразований над графом и «разворачивания» рекуррентной цепочки при помощи автономного устройства преобразования тегов (ПТ), которое применяет функциональные преобразования к сжатой рекуррентной цепочке и обеспечивает «саморазвертку» рекуррентного вычислительного процесса.

Устройство ПТ функционирует параллельно с вычислительным ядром и определяет инструкцию, которая должна быть выполнена на следующем шаге вычислительного процесса (модифицирует тегированные данные). В памяти нет исполняемой программы в традиционном смысле. Есть только поток ЭСД, которые динамически подвергаются рекуррентной «саморазвёртке» устройством ПТ. Поэтому, рассматриваемая архитектура относится к третьему принципиальному классу и может быть классифицирована как Data-Flow/Dynamic (DF/D).

Рисунок 1.4 (рисунок П1 из [42]) иллюстрирует сравнение представленных принципиальных классов. Рисунок 1.5 (рисунок П2 из [42]) иллюстрируют сравнение описанных принципиальных классов по организации конвейера обработки инструкции.

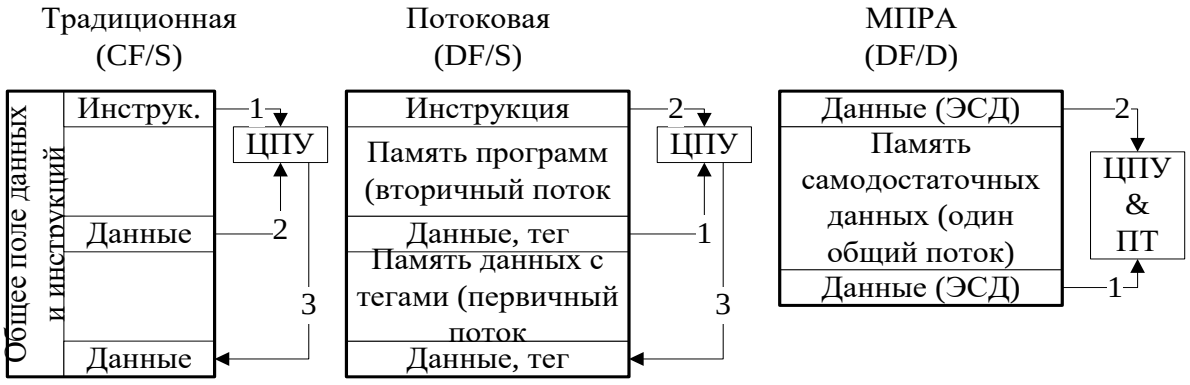


Рисунок 1.4 – Принципиальные отличия сравниваемых архитектур (рис. П1 из [42])

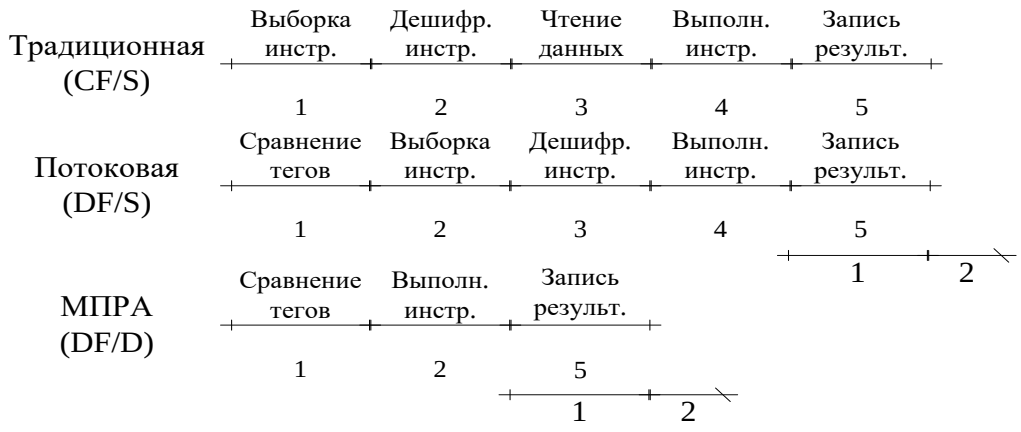


Рисунок 1.5 – Выполнение инструкций в сравниваемых архитектурах (рис. П2 из [42])

1.3.2 Высокоуровневый прототип МПРА

Анализ рекуррентно-потоковой модели вычислений позволил выделить следующие ключевые особенности, указанные авторами в работах [42, 43]:

1) Объединение двух, характерных для подавляющего числа моделей вычислений, потоков - данных и инструкций в один единый поток самодостаточных данных. Благодаря наличию всего одного потока, а также некоторым другим архитектурным решениям, в рамках МПРА обработка одной инструкции может быть осуществлена практически в два раза быстрее по сравнению с другими архитектурами.

2) Отказ от ассоциативной памяти, использование вместо память адресной проверки для определения совпадающих пар тегированных данных. Это решение дает существенный рост производительности и уменьшение энергозатрат, но накладывает некоторые ограничения на структуру входного потока данных.

3) Рекуррентность – свойство новой модели, которое заключается в динамическом развитии вычислительного процесса. В системе самодостаточных данных каждый следующий шаг обработки порождается в ходе развития процесса как функция предыдущего. Исходный поток инструкций рекуррентно сворачивается, тем самым позволяя резко сократить накладные расходы, связанные с опережающим хранением трассы вычислительного процесса.

В разделе 1.2.6 был рассмотрен подход разработки потоковых архитектур в виде комбинированных двухуровневых ПФН-архитектур. Были показаны потенциальные преимущества такой реализации архитектуры для решения задач ЦОС. Поэтому было принято решение разрабатывать и испытывать на ПЛИС прототип МПРА в виде ГАРОС. В работе [44] рассматриваются проблемы и особенности реализации данного прототипа. На рисунке 1.6 (рисунок 1 из [44]) приводится концептуальная структура ГАРОС.

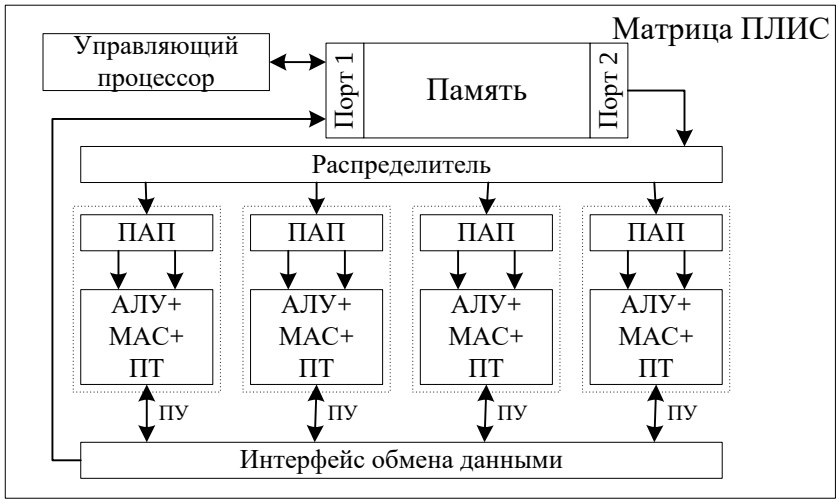


Рисунок 1.6 – Концептуальная структура ГАРОС (рис. 1 из [44])

На верхнем управляющем уровне (УУ) используется фон-Неймановский процессор общего назначения, который исполняет функции процессора ввода-вывода, а также управляет подготовкой потока ЭСД, который спускается на исполнение на нижний уровень. Данный процессор также может быть использован для реализации последовательных фрагментов исполняемого алгоритма, вычисление которых на потоковом процессоре неэффективно.

На нижнем операционном уровне прототипа используется РОУ. Именно РОУ реализует рекуррентно-потоковую модель вычислений. Оно принимает на вход посредством двух-портовой Буферной памяти (БП) сформированные на верхнем уровне ЭСД, а также размещает в этой памяти выходные данные. Операционный уровень состоит из селектирующей среды, названной «Распределитель», а также четырех идентичных вычислительных ядер, каждое из которых содержит модули Памяти адресной проверки (ПАП), блоки умножения с накоплением (Multiply Accumulate – MAC), регистровый файл, а также блоки ПТ.

В текущей версии прототипа ЭСД имеют размер равный 56 бит. Для упрощения реализации и испытаний прототипа вычислительные модули реализуют арифметику с фиксированной точкой, а разрядность обрабатываемых данных составляет 16 бит.

В рамках спецификации РОУ используются следующие термины и понятия.

Операнд – элемент самодостаточных данных. Он может включать в себя содержательную, функциональную и вспомогательную информацию, организованную в виде наборов полей.

Степень параллелизма (СП) - количество путей данных (секций) на операционном уровне, способных функционировать в параллель. Значение СП в ГАРОС принято равное 4.

Секция - один из параллельных путей данных, связанный с одной парой операндов.

Горсть - комплект операндов, перемещаемых совместно в пределах операционного уровня. В горсть может сгребаться некоторое количество смежных компонентов векторов или

произвольных независимых операндов. Количество операндов в горсти может быть в диапазоне от 0 (пустая горсть) до СП (полная горсть). Горсть может рассматриваться как логический интерфейс между ступенями и синхронизатор шагов.

В работе [42] подробно рассмотрены несколько подходов к реализации РОУ, а именно: слоевая модель, распределенная структура, смешанная структура и централизованная структура. Для реализации ГАРОС была выбрана централизованная структура РОУ, расширенная дополнительными структурными блоками и механизмами. Спецификация расширенной версии архитектуры РОУ может быть найдена в отчетах [45, 46].

В состав расширенной версии архитектуры РОУ входят следующие блоки:

• Распределитель – буфер FIFO, принимающий капсулы с процедурного уровня архитектуры. Получая капсулы, Распределитель осуществляет анализ типов операндов и формирует последовательности исходных (source) S-операндов с данными, которые отправляются на дальнейшую обработку. Кроме того, Распределитель извлекает вспомогательные и управляющие операнды и осуществляет настройку РОУ.

• Экспликатор – дополнительная ступень Распределителя. Обработка данных в РОУ организована в виде горстей, поэтому поток ЭСД из капсулы должен быть предварительно преобразован в горсть. Данный процесс называется экспликацией горстей. Экспликация является составной операцией и осуществляется в два этапа. Первый этап называется *репликацией* (размножением или векторизацией) операнда (на требуемое количество процессоров). Второй этап называется, собственно, *экспликацией* (формированием горсти).

• Ключ (представляет из себя блок мультиплексоров) на входах ПАП – выбирает очередную горсть для направления операндов в ПАП, где происходит образование пар.

• Итератор - специальный активный буфер заменяемых операндов, которые могут привлекать данные из регистров состояния Исполнителей и снабжать их требуемыми функциональными полями. Также данные операнды могут организовывать циклические процедуры и привлекать константы из памяти констант.

• Память адресной проверки – обеспечивает хранение операндов, а также формирование и экспозицию пар операндов. РОУ содержит комплект из СП запоминающих устройств (по одному на каждую секцию), способных запомнить сразу полную горсть.

• Жонглер – блок перестановки (жонглирования) операндов, который обеспечивает разрешение противоречий и корректную расстановку операндов на входы Исполнителя.

• Исполнитель – предназначен, как правило, для выполнения операций над парами экспонированных горстей. РОУ содержит комплект из СП Исполнителей, работающих в параллель и способных обрабатывать пару. В каждом Исполнителе подразумеваются Вычислитель и Преобразователь тегов. Каждая из секций Вычислителя содержит АЛУ,

Умножитель-аккумулятор, Сдвигатель и Аккумулятор. Преобразователь служит для преобразования функциональных полей выходных операндов, которые рассылаются по E-шине обратно в ПАП или на выход в процедурный уровень. Вспомогательная **A**-шина может быть использована различным образом в зависимости от выбранного подхода реализации прототипа. Наиболее очевидное использование A-шины заключается в передачи по ней содержимого Аккумулятора.

• Импликатор ответственен за формирование горстей выходных операндов. Он получает операнды, в том числе операнды-шаблоны, и выдает последовательность выходных операндов, используя Шаблоны для импликации.

• Тасовщик – блок тасования выходных операндов Исполнителей, который рассылает их по секциям назначения или в Импликатор.

• Память подкачки тегов и память переходов – источники состояния частичного и полного комплекта функциональных полей выходного **P**-операнда после срабатывания условия перехода.

• Регистр тега – регистр временного хранения значений функциональных полей содержательных операндов. Широкий класс алгоритмов в области обработки сигналов характеризуется регулярной структурой обрабатываемых данных: небольшие участки программного кода (2-5 вычислительных шагов) обрабатывают большой массив входных данных. В рамках рекуррентного подхода это выражается в идентичности функциональных полей массива входных данных. Регистр тега и специальная процедура его инициации и деактивации позволяют минимизировать величину капсулы для такого рода входных данных при пересылке ее на операционный уровень РОС. При этом один S-операнд в капсуле может содержать от одного до восьми значений данных (в зависимости от конфигурации).

• Глобальная память констант (ПК_Г) – постоянная память хранения констант, часто используемых в выбранном классе алгоритмов (поворотных коэффициентов, коэффициентов корреляции, моделей слов в системах распознавания и пр.). Используется для хранения глобальных констант, одновременно обрабатываемых во многих секциях РОУ.

• Секционная память констант (ПК_С) – постоянная память для хранения констант локального назначения в рамках определенной секции РОУ.

• Умножитель-аккумулятор – комплект из СП секционных MAC-блоков с накоплением результатов умножения в одном из двух регистров-аккумуляторов (B или C).

Устройство MAC выполняет следующие операции: 1) умножение; 2) умножение с накоплением; 3) арифметический, логический, циклический сдвиг; 4) округление результата. Архитектура РОУ предусматривает параллельное функционирование следующих вычислительных ресурсов: MAC-устройства, состоящего из 40-разрядного арифметического

устройства (AU-40), умножителя (M) и двух регистров-аккумуляторов (B и C); 16-разрядного АЛУ; сдвигателя и устройства округления (BS&R).

Наиболее значимым нововведением в расширенной версии архитектуры является регистр тега. Он используется в рамках нового механизма упаковки данных. Суть данного механизма заключается в том, что вводится специальный тип операндов – «упакованный операнд». Данный операнд вместо большинства функциональных полей несет в себе несколько элементов данных. В момент «распаковки» данного операнда формируется несколько ЭСД, функциональные поля которых считываются из регистра тега. Инициируется данный регистр с помощью специального операнда «Загрузчик тегов».

На рисунке 1.7 (рисунок 2.1 из [45]) представлена структурная схема расширенной версии прототипа РОУ.

1.3.3 Модель программирования МПРА

1.3.3.1 Типизация операндов

Операционный уровень архитектуры специфицирует четыре типа операндов Тип операнда указывается в обязательном для всех операндов ***t*~**поле.

Пустой операнд (***E:*** *тип*) не несет никакой информации, не может быть экспонирован и не обрабатывается, не имеет структуры.

Вспомогательные операнды (***A:*** *тип*) тоже не могут быть экспонированы и не обрабатываются, но несут вспомогательную конфигурационную информацию.

Управляющие операнды (***C:*** *тип*) организуют вычислительный процесс в качестве управляющих субъектов. Они могут экспонироваться, но не обрабатываются. У них есть две части: управляющая и функциональная.

Содержательные операнды (***D:*** *тип*) несут данные, подлежащие обработке. Они могут экспонироваться и обрабатываются как объекты процесса вычислений. У них есть две части: содержательная и функциональная.

Всего в рамках базовой спецификации РОУ [94] было определено 26 типов операндов, представленные в таблице 1.1 со своим описанием и сигнатурой. Символами '[' и ']' обозначаются функциональные части операндов, символом '%' – вспомогательные поля, символом '@' – управляющие, символами ′{′ и ′}′ – рекуррентные функциональные поля .

Формально единицы самодостаточных данных могут быть представлены в виде тройки:

$$S_i = < t, d_i, I_i > \quad (1.3)$$

где t – тип операнда, S_i – элемент потока с номером i, d_i – элемент данных, I_i – набор инструкций.

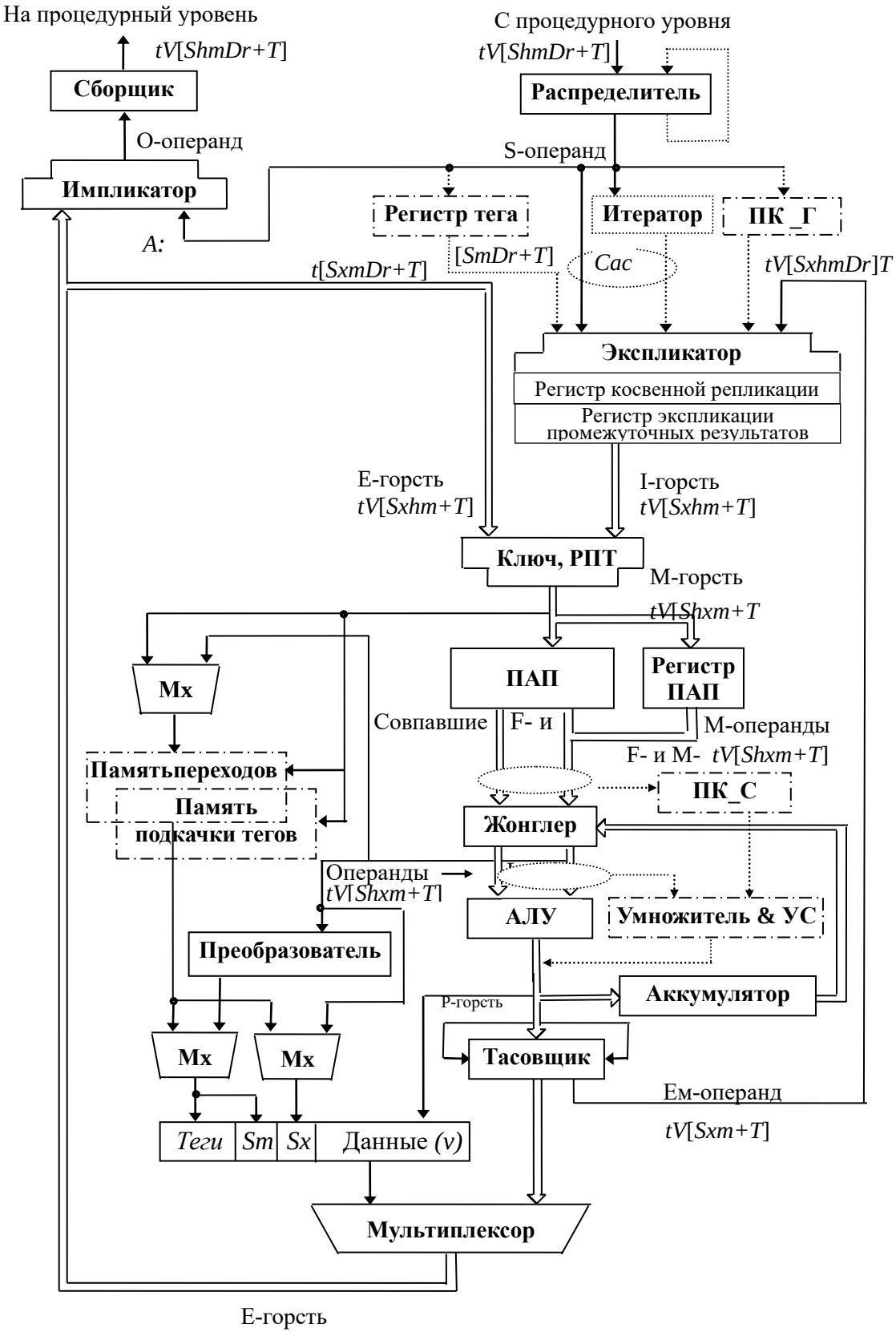


Рисунок 1.7 – Расширенный вариант операционного уровня (рис. 2.1 из [45])

Следует отметить, что в рамках принятых разработчиками соглашений по именованию и обозначению было установлено:

- все тегируемые данные называются функциональными полями, в том числе элемент данных - тоже функциональное поле;
- поле с данными называется содержательной частью;
- поле с набором инструкций, подвергающихся рекуррентным преобразованиям, называется функциональной частью;
- поле с набором управляющих инструкций называется управляющей частью;
- поле с набором вспомогательных инструкций называется вспомогательной частью;
- совокупность некоторого набора функциональных полей называется операндом;
- операнды могут быть разных типов и назначений в зависимости от комбинации из четырех частей, указанных выше;
- программа в ГАРОС является последовательностью операндов и называется капсулой.

Таблица 1.1 – Специфицированные операнды текущей версии прототипа

Тип	**Операнд**	**Сигнатура**
Ad	Терминатор НД	Без тела
Az	Терминатор капсулы	Без тела
Acg	Глобальный конфигуратор	%Cxpcrefn
Ai	Инициализатор	@I0nis@I1nis
At	Шаблон	%Trhmuctse[DrShmOu]
Am	Расширенная маска репликации	%Mvtom
Asi	Стартер входных операндов	%St[ShmOuctDrse]{SmOucDs}
Afi	Финишер входных операндов	Без тела
Aso	Стартер выходных операндов	%Stn [DrShmOuctDse]
Afo	Финишер выходных операндов	Без тела
Apdi_3x16	Упакованный операнд с 3 16-разрядными данными	V1V2V3[Sh1h2h3]
Apdi_6x8	Упакованный операнд с 6 8-разрядными данными	V1V2V3V4V5V6[Sh1h2h3h4h5h6]
Ccs	Конфигуратор секции	@Cjldbsmt1[ShmOuctDrse]
Ccl	Конфигуратор констант	@Caml[ShmOuctDrse]
Cad	Мусорщик для ПАП	@At[ShmOcutDrse]
Cacn	Заменяемый обычный	[ShmOuctDrse]{SmOucDs}
Caco	Заменяемый специальный	@Adiu[ShmOuctDrse]{SmOucDs}
Cah	Тормоз	[ShmOuctDrse]
Car	Погонщик	[ShmOuctDrse]
Cbb	Переход	@Bhmuctseb[DrShmOu]{SmOu}
Cbc	Контроллер циклов	@Bhmuctse[DrShmOu]{SmOu}
Cbf	Загрузчик регистра флагов	@Bf[ShmOuctDrse]
Cbi	Загрузчик счетчика циклов	@Bi[ShmOuctDrse]
Cbt	Загрузчик тегов	@Buchmse[OutShmDr]
Di	Число с фиксированной запятой	@sVi[ShmOuctDrse]{SmOucDs@s}
Di_38	Число с фиксированной запятой 38-разрядное	@sV38[ShmOuc]{@s}

Не все функциональные поля операндов подвергаются рекуррентным преобразованиям. В частности, не предусмотрены преобразования вспомогательной и управляющей информации по настройке отдельных регистров, флагов и режимов работы.

Приведенные понятия «частей» операнда требуют внесения уточнений в формальное определение единицы самодостаточных данных:

$$S_i = < t, [D_i], [I_i] > \quad (1.4)$$

$$I_i = < [F_i], [C_i], [A_i] > \quad (1.5)$$

В формулах (3.2 – 3.3) ″[ ]″ обозначает необязательный элемент; D_i – содержательная часть, содержит один элемент данных; F_i – функциональная (functional) часть, содержит непосредственную информацию для управления вычислительным процессом; C_i – управляющая (control) часть, содержит информацию по настройке аппаратных средств на соответствующие режимы работы; A_i – вспомогательная (additional) часть, содержит, в основном, информацию по управлению наборами входных и выходных данных.

Таким образом, представление программы в терминах модели языка есть $\{S_n\}$ - множество из некоторого положительного количества n операндов. В поисках подходящего названия для программ МПРА разработчики ориентировались на такие их свойства, как сокрытие данных и инструкций программы от пользователя, предоставление доступа только к внешнему интерфейсу вызова и чтения результирующих данных программы, защищенность данных программы. Все перечисленные свойства характерны для понятия *инкапсуляция* – одного из основных механизмов объектно-ориентированного подхода в программировании. Поэтому было решено программе, работающей в среде МПРА, дать название *капсула*, а соответствующий стиль программирования назвать *капсульным программированием* или *парадигмой капсульного программирования*.

1.3.3.2 Капсула как способ реализации сложной структуры данных и ее строение

В разделе 1.2.4 были рассмотрены существующие подходы построения структур данных. Потоковая рекуррентная архитектура реализует динамическую потоковую модель вычислений с тегированными токенами. Исследуемый прототип архитектуры ГАРОС является двухуровневой архитектурой. Основным интерфейсом взаимодействия между верхним управляющим уровнем и нижним операционным уровнем является капсула. С точки зрения организации процесса этого взаимодействия капсула является структурой данных, а РОУ осуществляет функциональное преобразование над входной капсулой, заполненной входными данными, в результате которого формируются выходные данные, размещаемые в этой же капсуле. Следовательно, капсулу можно интерпретировать как массив самодостаточных данных.

Согласно спецификации, существует два типа капсул: входные и выходные. Выходные капсулы формируются динамическим образом путем отбора промежуточных данных,

формируемых блоками Исполнитель. Отобранные данные снабжаются функциональными полями, значения которых извлекаются из заранее загруженных в Импликатор шаблонов. На выходе РОУ формируется выходная. Входные же капсулы, напротив, формируются статическим образом на этапе программирования и компиляции. Эти капсулы, по сути, являются шаблонами, которые хранятся в двух-портовой буферной памяти (см. рисунок 1.6) и наполняются данными со стороны УУ на этапе решения конкретной задачи в ГАРОС. Однако, сам механизм наполнения капсулы данными можно считать динамическим, т.к. только после своего заполнения капсула как структура данных обретает семантический смысл.

Программа УУ может одновременно заполнять разные шаблоны капсул, которые могут располагаться нелинейно. При этом запуск очередной капсулы на исполнение в РОУ не может быть ограничен условием полного заполнения капсулы, иначе накладные расходы ожидания готовности капсулы будут непомерно высоки. Следовательно, капсула является непрямолинейной структурой данных. Из всех рассмотренных в разделе 1.2.4 подходов к организации структур данных в потоковых архитектурах только I-структуры являются непрямолинейными. Следовательно, капсула является близким аналогом I-структуры.

Аналогично I-структуре, каждый ЭСД в капсуле снабжается битом готовности, который индицирует готовность к обработке данного операнда в РОУ. При этом данный операнд может ожидать загрузки данных или быть константой с битом готовности равным 1 по умолчанию. Разгрузка капсулы может быть начата в любой момент времени, но, в отличие от I-структуры, РОУ не формирует списка отложенных запросов чтения к капсуле. Вместо этого, операнды считываются последовательно в соответствии с запрограммированным алгоритмом устройства управления памятью. При обнаружении операнда с битом готовности равным 0 буферная память посылает сигнал РОУ, который замораживает конвейер до того момента, пока для текущего операнда не будет взведен бит готовности. Это позволяет экономить аппаратные средства ценой потенциального снижения производительности.

На рисунке 1.8 приведена структура капсулы в общем виде.

Конфигурационный раздел
Раздел входных данных
Раздел упакованных входных данных
Раздел входных данных
Раздел выходных данных

Рисунок 1.8 – Структура капсулы

Каждому полю тегированных операндов поставлено в соответствие некоторое множество значений, причем эти значения - двух видов: символьные (мнемонические) и числовые.

Капсула, у которой значения всех функциональных имеют символьный вид, называется символьной. Символьная капсула создается на более ранних этапах реализации алгоритма, т.к. является шаблоном для последующих числовых капсул.

Числовая капсула – это капсула, заполненная данными. В частности, функциональные поля, подвергающиеся рекуррентным преобразованиям, обязательно должны быть означены корректно, т.к. в случае неправильного задания их начальных числовых значений развертка вычислительного процесса будет ошибочной.

Функции рекуррентных преобразований заложены как в числовые, так и в символьные капсулы. Это позволяет, моделируя символьные капсулы, отлаживать алгоритмические ошибки, а моделируя числовые капсулы – обнаруживать потери точности вычислений.

1.3.3.3 Представление программ и проблемы программируемости

Капсула является последовательностью ЭСД и инкапсулирует некоторый алгоритм, представленный в виде совокупности рекуррентных цепочек. Следовательно, конкретная решаемая с помощью ГАРОС задача может быть представлена в виде множества капсул. Кроме того, не все алгоритмы решаемой задачи целесообразно инкапсулировать в капсулы. Часть из них должен решать процессор на управляющем уровне ГАРОС. Например, задачи ввода-вывода или короткие последовательные участки алгоритмов. Из всего сказанного следует, что исполняемая программа в ГАРОС – это набор:

1) Капсул, которые исполняются на РОУ и обеспечивают набольшую производительность вычисления конкретного алгоритма.

2) Последовательных участков программы, а также задач ввода-вывода, реализованных на языке программирования высокого уровня.

С точки зрения аппаратной реализации РОУ является сопроцессором для процессора УУ. Поэтому выбор основной парадигмы и инструмента программирования будет зависеть от процессора УУ. В исследуемом прототипе роль процессора УУ исполняет фон-Неймановский процессор общего назначения. Следовательно, его программирование осуществляется с использованием императивной парадигмы программирования [47] и одного из императивных языков программирования (например, C или C++). В тоже время, РОУ реализует потоковую модель вычислений, которая основана на парадигме функционального программирования [48]. Следовательно, разработка капсул должна включать в себя использование инструментов функционального программирования.

Рассматриваемая модель программирования ГАРОС характеризуется рядом проблем. В своих ранних работах [49-53] автор данной диссертационной работы рассматривает различные проблемы программируемости МПРА и ее прототипа. К ним относятся:

1) Как оптимально распределить алгоритмы решаемой задачи между УУ и РОУ?

Основная сложность заключается в оценке накладных расходов на взаимодействие УУ и РОУ. Без итеративных измерений производительности невозможно оценить заранее – имеет ли смысл инкапсулировать последовательные алгоритмы в капсулу или нет.

2) Как оптимально распределить фрагменты потокового графа конкретного алгоритма по Исполнителям РОУ с учетом рекуррентной свертки и развертки?

В данном случае недостаточно простого использования эвристических алгоритмов балансировки нагрузки, рассмотренных в разделе 1.2.5. Отдельные фрагменты потокового графа вычисляемого алгоритма сжимаются в рекуррентные цепочки. В результате сжатия и повторного запуска алгоритма балансировки может произойти переоценка и перераспределение сжатых и не сжатых узлов графа. Данное перераспределение может привести к снижению производительности, т.к. для случая активного межсекционного взаимодействия накладные расходы пересылки данных между узлами и рекуррентными цепочками могут оказаться выше реального прироста производительности. Поэтому, предпочтительной реализацией параллелизма в капсуле является крупнозернистый параллелизм.

3) Как выбрать и настроить требуемые функции рекуррентных преобразований?

Данная проблема тесно связана с проблемой 2). Удачный выбор реализуемой в ПТ функции преобразования может значительно упростить балансировку нагрузки и получить существенный прирост производительности. С другой стороны, поиск подходящей функции преобразования сама по себе сложная вычислительная задача, которая либо должна быть решена заранее на этапе компиляции, (тогда выборка функций преобразований будет сильно ограничена), либо – в процессе вычислений (что может привести к потере производительности). Настройка ПТ на сложную функцию преобразований затрачивает достаточно много циклов исполнения алгоритма. Следовательно, данную настройку необходимо выполнять как можно реже, что также снижает вариативность возможных функций преобразований.

4) Какие инструменты императивного и функционального программирования использовать для автоматизации разработки (компиляции) капсул?

В работах [49-53] рассматриваются возможности использования языка функционального программирования SISAL (Streams and Iterations in a Single Assignment Language) [54] для автоматизации построения исходного потокового графа и его балансировки на заданное количество процессоров. Также рассматривается ряд технологий императивного программирования (OpenMP, MPI и др.) и оценивается их применимость для описания программы, исполняемой на УУ. Тем не менее, подходящих инструментов для осуществления рекуррентной свертки просто не существует.

5) Как отлаживать и оптимизировать капсулы и программы УУ?

Решение данной проблемы требует разработки специализированного инструментария для компиляции, моделирования и измерения производительности полученной программной реализации задачи. Из пунктов. 1) и 2) следует, что разработка каждой капсулы и всей программы в целом является сложным итеративным процессом, не все этапы которого можно выполнить в автоматизированном режиме.

1.3.4 Анализ функциональных возможностей отдельных блоков прототипа МПРА

1.3.4.1 Распределитель

Ключевым компонентом ГАРОС является «Распределитель». Данный компонент отвечает за репликацию и рассылку операндов в секционные Исполнители. Основными функциями Распределителя, а также Экспликатора и Ключа (как его составных частей) являются:

1) Запрос и прием операндов от Буферной Памяти.

2) Анализ типов входных S-операндов.

3) Глобальная конфигурация РОУ.

4) Конфигурация и инициация блока Итератор.

5) Репликация (размножение) операндов в соответствии с их Маской Репликации.

6) Экспликация операндов – это преобразование потока операндов в форму горстей, которые затем отправляются на обработку в Исполнители.

7) Выбор или объединение горстей из различных источников на этапах работы Экспликатора и Ключа.

8) Управление Маской Репликации в режиме косвенной репликации.

Механизм репликации операндов

Маска репликации - битовая маска секций назначения при экспликации горстей. В общем случае она имеет СП разрядов, по разряду на секцию. Код репликации (4 разряда) не зависит от СП и количества задействованных секций. Спецификация РОУ определяет три типа маски репликации: прямая, косвенная и репликация Е-операндов.

Маска прямой репликации (МПР) инициируется ненулевым кодом репликации и прямо указывает секции назначения операндов; операнды с одиноковой маской репликации всегда направляются в одни и те же секции.

Инициация маски косвенной репликации (МКР) производится нулевым значением кода репликации (*[Dr=0000]*). После каждого шага косвенной репликации значение МКР меняется в зависимости от вида косвенной репликации; поэтому два последовательных операнда в капсуле, использующих косвенную репликацию, реплицируются в разные секции. Режим косвенной репликации задается вводом вспомогательного операнда **Am:**. Он обеспечивает настройку на любые требуемые сочетания секций, в которые производится репликация операндов. Операнд **Am:** (расширенная маска репликации) поступает на операционный уровень из капсулы и

содержит часть исходного значения МПР или МКР. В таблице 1.2 приводятся специфицированные режимы управления косвенной репликацией.

Таблица 1.2 – Управление косвенной репликацией

%Cc~подполе (УКР)		Режим репликации при нулевой маске	Действие над МКР после шага репликации
Символ	**Код**		
d	0	Репликации нет	МКР не меняется
g	1	По коду в МКР	Ротация влево (ст. в мл.)
a	2	По коду в МКР	Сдвиг влево (мл. в мл.)
h	3	По коду в МКР	Горстевой сдвиг влево
h_f	4	По коду в МКР	Горстевой сдвиг влево

Режим *g*, называемый *гирляндой*, осуществляет равномерное распределение операндов по заданным в МКР кодам. Это достигается путем использования текущего значения МКР для репликации очередного операнда с нулевой маской репликации. После чего значение МКР модифицируется в соответствии с алгоритмом, указанным в таблице 1.2. Вновь рассчитанное значение МКР используется для репликации следующего операнда и т.д.

Режим *a*, называемый *лавиной*, осуществляет прогрессирующую репликацию, постепенно заполняя все секции. МКР используется и модифицируется аналогично режиму *g*.

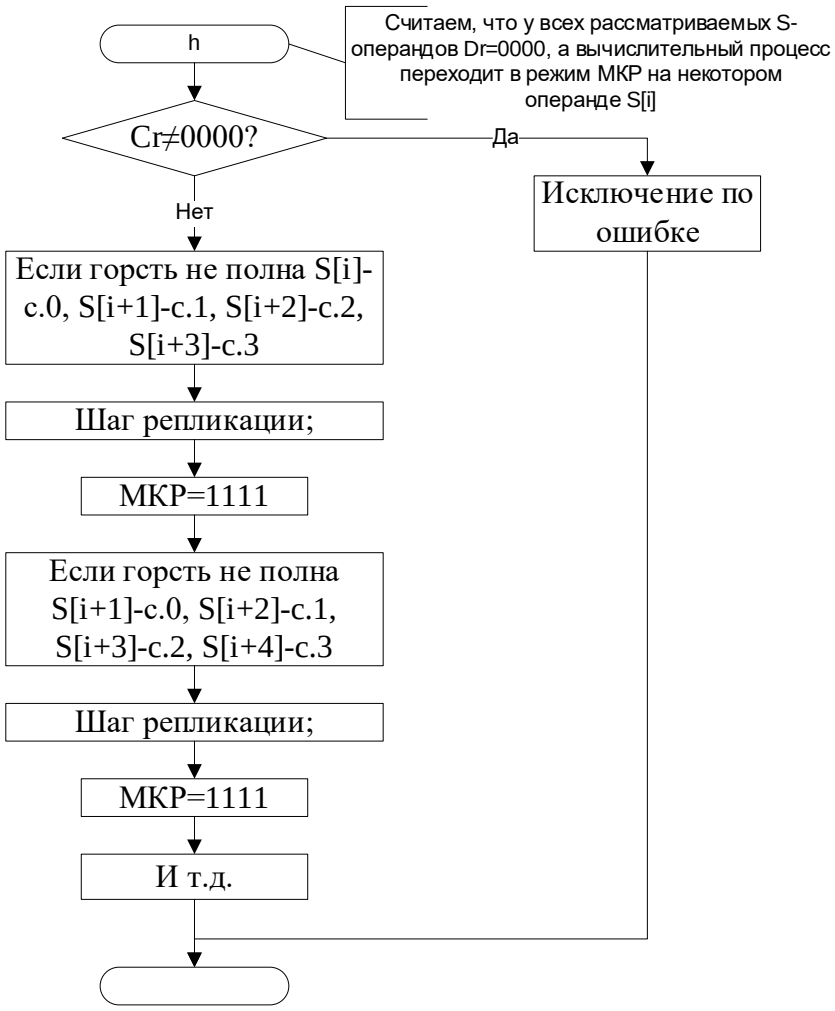
В режиме горстевого сдвига осуществляется рассылка нескольких последовательно расположенных в буфере Распределителя операндов (по количеству '1' в МКР). После чего значение МКР модифицируется, а из буфера Распределителя удаляется один операнд. Рисунок 1.9 иллюстрирует алгоритм работы режима *h*.

Режим косвенной репликации прерывается по факту прихода очередного операнда, значение поля [Dr] которого не равно нулю. Кроме того, данное событие также является условием завершения формирования горсти.

Таким образом, режимы косвенной репликации обеспечивают достаточно гибкий механизм рассылки операндов по параллельным секциям, который частично позволяет обойти ограничения механизма разгрузки капсулы из БП, рассмотренные в разделе 1.3.3.

В РОУ определено три вида памятей констант:

1) Память констант Глобальная. Как показано на рисунке 1.7 данная память располагается на том же уровне, что и Распределитель. Таким образом, формируемые на ее выходе операнды участвуют в полном цикле образования горстей, проверки на совпадение и экспозицию на Жонглере, прежде чем попасть в Исполнители. Преимуществом данной памяти, по сравнению с другими подходами является малый объем накладных расходов и простота реализации. Недостатком – константа занимает место в горсти, тем самым снижая пропускную способность входных данных.



Горстевой сдвиг, МКР постоянна и МКР=1111, осуществляется сдвиг последовательности операндов из капсулы

Рисунок 1.9 – Алгоритм репликации режима h

1.3.4.2 Памяти констант

2) Память констант Секционная. Как показано на рисунке 1.7 данная память располагается на том же уровне, что и Вычислитель. Таким образом, константы из данной памяти попадают напрямую в Вычислитель и могут быть использованы в качестве третьего входного данного. Преимуществом данного подхода является повышение пропускной способности данных, что потенциально позволяет задействовать больше вычислительных блоков внутри Вычислителя. Недостатками являются: высокая стоимость накладных расходов и избыточность, т.к. данная память дублируется во все секции.

3) Память констант Секционная Регистровая. Данная память располагается непосредственно в Вычислителе и позволяет хранить 4 константы, которые загружаются из капсулы с помощью специальных операндов Ccl:. Константы из данной памяти попадают

напрямую в Вычислитель и могут быть использованы в качестве третьего входного данного. Преимуществом данного подхода является повышение пропускной способности данных, что потенциально позволяет задействовать больше вычислительных блоков внутри Вычислителя. Недостатками являются: малый объем и медленная скорость загрузки.

Каждая константа имеет разрядность равную 16, а также, при необходимости, может быть дополнена комплектом функциональных полей. Использование констант позволяет сократить объем входной капсулы и потенциально повысить производительность Вычислителей за счет увеличения пропускной способности данных.

1.3.4.3 Исполнитель

Вычислитель

Ключевым блоком Вычислителя является устройство MAC. Оно предназначено для выполнения операций: умножение; умножение с накоплением; арифметический, логический, циклический сдвиг; округление результатов. В MAC входят следующие функциональные устройства: умножитель 16x16; сумматор/вычитатель; логика округления; 40-разрядный сдвигатель; регистры результатов [B] и [C]. Регистры [B], [C] и аккумулятор [A] являются 40-разрядными.

Устройство MAC работает параллельно АЛУ. На вход MAC поступают 16-разрядные содержательные части операндов. В соответствии с процедурой команды умножения с накоплением Умножитель вычисляет произведение двух 16-разрядных сомножителей, выравнивает 32-разрядное произведение по правой границе и расширяет знаком до 40-разрядного значения; это значение суммируется с 40-разрядным содержимым используемого регистра [C], и результат сохраняется в [C]. Команда умножения без накопления выполняется так же, но на правое плечо сумматора/вычитателя поступает нулевое значение.

Приведенная структура MAC-устройства положена в основу наиболее быстродействующих ЦСП, например, TMS320C54x (Texas Instruments) и dsPIC30F6014 (Microchip). MAC – дорогостоящий ресурс, эффективность использования которого зависит от сбалансированности производительности собственно вычислительных ресурсов и каналов подкачки обрабатываемых данных. Если пропускная способность информационных каналов недостаточна, то вычислительные ресурсы будут простаивать.

Поэтому блок Вычислитель на аппаратном уровне должен поддерживать суперскалярные вычисления. В расширенной спецификации архитектуры введена аппаратная поддержка выполнения основной операции алгоритма БПФ. Особенности реализации данной поддержки будут подробно рассмотрены в главе 2.

Структура устройства MAC изображена на рисунке 1.10.

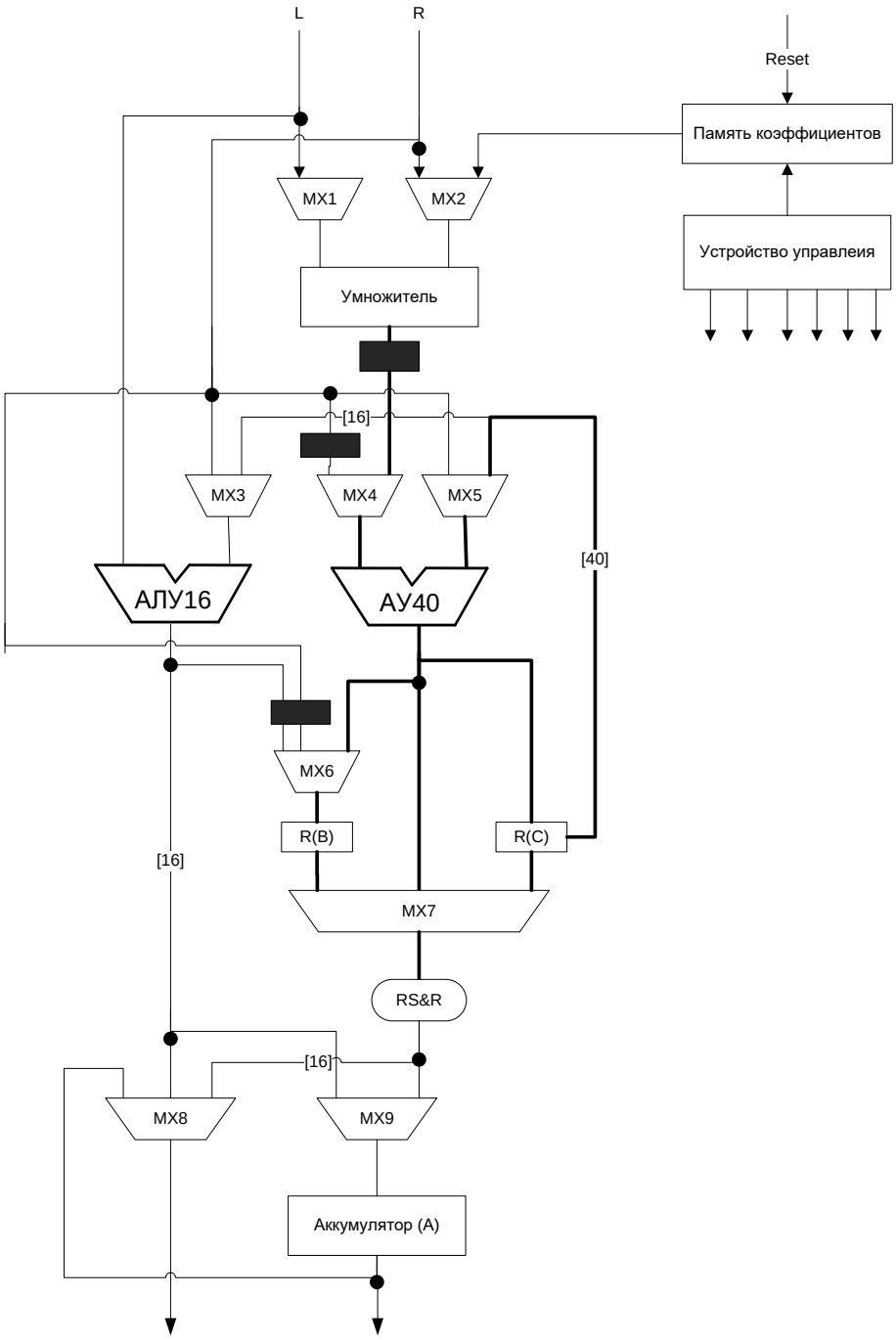


Рисунок 1.10 – Структура устройства MAC

С учетом доработки архитектуры Вычислителя в рамках расширенной спецификации также была доработана программная модель и система команд. Модель состоит из следующих регистров: системные регистры; регистры пользователя; вспомогательные регистры.

Системные регистры предназначены для настройки параметров и режимов работы РОС при выполнении капсул. Запись информации в системные регистры осуществляется непосредственно с помощью специальных операндов, помещаемых в исполняемые капсулы. К регистрам пользователя относятся аккумулятор [A] и два регистра [B], [C]. К вспомогательным регистрам относятся счетчик циклов и регистр флагов.

Аккумулятор [A] используется при выполнении команд. Аккумулятор предназначен для хранения результатов операций, выполняемых в АЛУ. Он позволяет уже на следующем такте вовлечь результат операции внутри каждой секции в вычислительный процесс. Пересылка содержимого аккумулятора осуществляется через **A**-шину. Однако возможность чтения содержимого аккумулятора в архитектуре не обеспечена. Поэтому предлагается использовать тот же механизм доступа, организованный для системных регистров. Доступ к отдельным частям 40-разрядных регистров [B] и [C] предусмотрен на уровне команд. Регистр флагов предназначен для установки и индикации режимов работы исполнительного устройства.

Для кодировки представленного множества команд оказалось достаточно [Oc] поля, имеющего размер равный 6 разрядам. Тем не менее, расширенная спецификация РОУ не определяет сценарии использования многозадачности. Поэтому на момент начала исследования ГАРОС Вычислители РОУ фактически были однозадачными.

В таблице 1.3 приводится система команд блока Вычислитель.

Преобразователь тегов

Функциональная часть секционного Исполнителя – Преобразователь тегов – предназначен для выполнения требуемой модификации функциональных полей операндов, поступивших на обработку в Исполнитель.

Существует множество вариантов реализации ПТ. Одним из главных критериев выбора конкретного варианта является область применения МПРА. С точки зрения такого критерия можно выделить четыре класса реализации МПРА: общего назначения; проблемно-ориентированная; специализированная; с автономным механизмом реализации рекуррентной развертки функциональных полей [55].

Универсальность ПТ может базироваться на использовании *временной* избыточности (ПТ_ВИ) при настройке на рекуррентную развертку конкретного алгоритма или *кодовой* избыточности функциональных полей (ПТ_КИ). Оба вида преобразователей предполагают как определенные временные потери при их настройке на реализацию конкретного вида алгоритма, так и кодовую избыточность. Временные издержки при использовании ПТ_ВИ существенно выше, чем при ПТ_КИ, за счет обязательного введения в структуру обрабатываемой капсулы специальных настроечных операндов. ПТ_КИ исключает необходимость в каких-либо настроечных операндах и использует специальный стиль программирования, базирующийся исключительно на кодовой избыточности.

Таблица 1.3 – Система команд Вычислителя

[Oc]~, @Bc~ и *%Tc~подполя*			**Операция в Вычислителе**
Символ прогр.	**Символ графа**	**Код**	
NOP	N	0x00	Резерв (отсутствие операции). Устанавливается автоматически после сброса Операционного уровня
ADD	+	0x01	Сложение
ADC	+c	0x02	Сложение с учетом переноса
SUB	-	0x03	Вычитание
SBC	-c	0x04	Вычитание с учетом займа
MULuub	*ub	0x05	Умножение (беззнаковое на беззнаковое) результат в регистр B
MULuuc	*uc	0x06	Умножение (беззнаковое на беззнаковое) результат в регистр C
MULssb	*sb	0x07	Умножение (знаковое на знаковое) результат в регистр B
MULssc	*sc	0x08	Умножение (знаковое на знаковое) результат в регистр C
MACab	*+b	0x09	Умножение с накоплением в регистре B
MACsв	*-b	0x0A	Умножение с вычитанием из регистра B
MACac	*+c	0x0B	Умножение с накоплением в регистре C
MACsc	*-c	0x0C	Умножение с вычитанием из регистра C
CLRc	Rc	0x0D	Очистка регистров C
CLRb	Rb	0x0E	Очистка регистров B
AND	&	0x0F	Конъюнкция
OR	1	0x10	Дизъюнкция
NOT	~	0x11	Инвертирование
SHR	l>	0x12	Логический сдвиг вправо
SHL	l<	0x13	Логический сдвиг влево
ASR	a>	0x14	Арифметический сдвиг вправо
ASL	a<	0x15	Арифметический сдвиг влево
Jccz	↑z	0x16	Переход если флаг Z=1
Jccc	↑c	0x17	Переход если флаг C=1
If	<>	0x18	Выполнение цикла
LI	Ld	0x19	Загрузка счетчика циклов
FR	f0	0x1A	Сброс флагов в "0"
FS	f1	0x1B	Установка флагов в "1"
MOVlb	←lb	0x1C	Пересылка младшей части регистра B
MOVdb	←db	0x1D	Пересылка средней части регистра B
MOVhb	←hb	0x1E	Пересылка старшей части регистра B
MOVlc	←lb	0x1F	Пересылка младшей части регистра C
MOVdc	←db	0x20	Пересылка средней части регистра C
MOVhc	←hb	0x21	Пересылка старшей части регистра C
LMb	→db	0x22	Загрузка средней части регистра B MAC
LMc	→db	0x23	Загрузка средней части регистра C MAC
RNDb	rb	0x24	Округление содержимого регистра B MAC
RNDc	rc	0x25	Округление содержимого регистра C MAC
BUTT	¥	0x26	Базовая операция БПФ
ED	€	0x27	Евклидово расстояние
LMS	ß	0x28	LMS-коэффициенты
LSP	µ	0x29	LSP-параметры
EBS		0x3F	Отсутствие специфицированного кода операции
		0x2A-0x2E	Резерв

В [55] рассмотрены основные особенности реализации различных исполнений ПТ, а также обоснован выбор реализации универсального ПТ с кодовой избыточностью. Универсальный ПТ порождает древовидную структуру графа, которая сходится к нулевому самовозвратному полюсу (нулевой вершине). Этот составной ПТ представляет собой совокупность двух конкатенируемых двухразрядных элементарных преобразователей, каждый из которых реализует логическую функцию сложения по модулю два со сдвигом вправо на 1 разряд. Рассмотренный ПТ характеризуется 200% избыточностью.

1.3.4.4 Организация конвейера РОУ

Двухуровневая организация архитектуры, представленная на рисунке 1.6, включает в себя основной интерфейсный компонент взаимодействия между уровнями – Буферную Память. Использование данной памяти позволяет организовать *асинхронный* механизм взаимодействия между УУ и РОУ. Это означает, что процессор УУ может иметь свой, сколь угодно длинный конвейер, и функционировать на своей частоте. На рисунке 1.5 показано, что минимально необходимое число шагов выполнения инструкции в МПРА (и, следовательно, в РОУ) равно 3. Поэтому РОУ был разбит на 3 стадии конвейера.

На первой стадии конвейера функционируют компоненты Распределитель, Экспликатор, Ключ и Импликатор. Распределитель и его ступени (Экспликатор и Ключ) осуществляют функции предварительного сравнения тегов и экспликации горстей по секциям и в Память ветвлений. Импликатор выполняет функции записи результатов.

На второй стадии конвейера функционируют Память Адресной Проверки и Жонглер. Память выполняет функции сравнения тегов и хранения неполных пар операндов. Жонглер обеспечивает разрешение противоречий функциональных полей совпавшей пары операндов и корректную расстановку входных данных для Вычислителя.

На третьей стадии конвейера функционируют Вычислитель, Памяти констант (кроме глобальной), Память ветвлений и Преобразователь тегов. Вычислитель и Память констант работают совместно для выполнения требуемой операции (и снабжения ее константой при необходимости). Память ветвлений снабжает Исполнительный блок необходимой теговой информацией в случае фиксации факта выполнения перехода. Преобразователь тегов выполняет функцию рекуррентных преобразований над входным набором функциональных полей. В случае отсутствия перехода выходной результат снабжается преобразованными полями, иначе – теговой информацией из памяти ветвлений.

1.4 Отечественные вычислительные системы на основе потоковой архитектуры

1.4.1 Многоклеточная архитектура «Мультиклет»

В 2001 году коллективом организации «Уральская архитектурная лаборатория» были начаты работы над прототипом мультиклеточного процессора, автором которого является Н.В. Стрельцов [56]. Этот процессор, получивший название «синпьютер», в 2003 году получил приз «Лучший продукт года» на форуме новых продуктов, проводимом в рамках ежегодной международной конференции IEEE по цифровой обработке сигналов. С 2003 года и по настоящее время проект завоевал еще несколько значимых наград, а также был запатентован (подана только одна заявка) и выпущен в серийное производство.

Несмотря на внушительный перечень наград за участие в различных выставках и конференциях, существует лишь несколько весомых публикаций, посвященных этой перспективной архитектуре, разработанной в Российской Федерации. Наиболее подробно основные аспекты архитектуры отражены в работе [57] и в архитектурной документации [58], опубликованной на официальном сайте компании производителя «Мультиклет». В работе [57] автор выделяет ряд особенностей и отличий мультиклеточной архитектуры от архитектур фон Неймана и потоковой, и приводит их краткое описание:

1) Использование «параграфа», как основной программной единицы. Под параграфом в данном контексте понимается “группа информационно-замкнутых команд (операций) с непосредственным указанием информационных связей между операциями”, также называемых линейными участками. Применение параграфов позволяет снять требование упорядоченного размещения команд в памяти (аналогично потоковым моделям вычислений).

2) Применение императивных языков программирования, использование последовательной выборки команд, тегирование информационных связей и применение расширенного правила срабатывания (аналогично гибридным потоковым архитектурам).

3) Система команд клетки является аппаратной реализацией входного языка программирования. Основным структурным элементом в рамках данной реализации является *триада*, описание которой приводится ниже.

4) Высокая степень защищенности неупорядоченного множества триад от вмешательства посредством стороннего ПО.

5) Триады обеспечивают эксплуатацию естественного параллелизма. Это достигается за счет реализованного механизма исполнения команд.

6) Использование *полносвязной* интеллектуальной коммутационной среды, работающей в режиме «широковещательной» рассылки. Данное решение, по мнению разработчиков, обеспечивает универсальность и эффективную масштабируемость архитектуры.

7) Высокая отказоустойчивость и надежность процессора за счет динамической реконфигурации в случае сбоя одной или нескольких клеток без необходимости перекомпиляции программы. Это достигается за счет широковещательной коммутационной среды и использования контекстов.

8) Асинхронная и децентрализованная организация мультиклеточного процессора на всех уровнях.

Под *триадой* в документации понимается тройка следующего вида [58]:

«<op><arg1><arg2>, где: op – поле кода операции; arg1 – поле первого операнда; arg2 – поле второго операнда.

В качестве операндов используются либо указатели на таблицу символов (для имен, определенных программистом, или констант), либо указатели на триады–источники используемых результатов.»

Мультиклеточный процессор (Мультиклет) состоит из множества N идентичных скалярных клеток, которые объединены коммутационной средой. Каждая клетка может выполнять только одну инструкцию в такт и, в свою очередь, состоит из:

- блок памяти программ (program memory – PM);
- устройство управления (control unit – CU);
- буферные устройства активных триад, ожидающих данных (BUF);
- коммутационное устройство (switching unit – SU).

Совокупность всех SU образует коммутационную среду (switching block – SB).

Программа в Мультиклете представляет собой неупорядоченную последовательность параграфов. Каждый параграф – это линейный участок инструкций (триад), информационная связь между которыми – прямая. Сами параграфы связаны между собой косвенно – через память. Несмотря на то, что разработчики утверждают о неупорядоченности последовательности параграфов, предлагаемый ими язык программирования Мультиклета поддерживает явные ссылки параграфов друг на друга. Следовательно, исполняемая программа обладает свойством *частичной упорядоченности* на уровне параграфов.

На рисунке 1.11 (рисунок 1 из [58]) приводится структура мультиклеточного процессора.

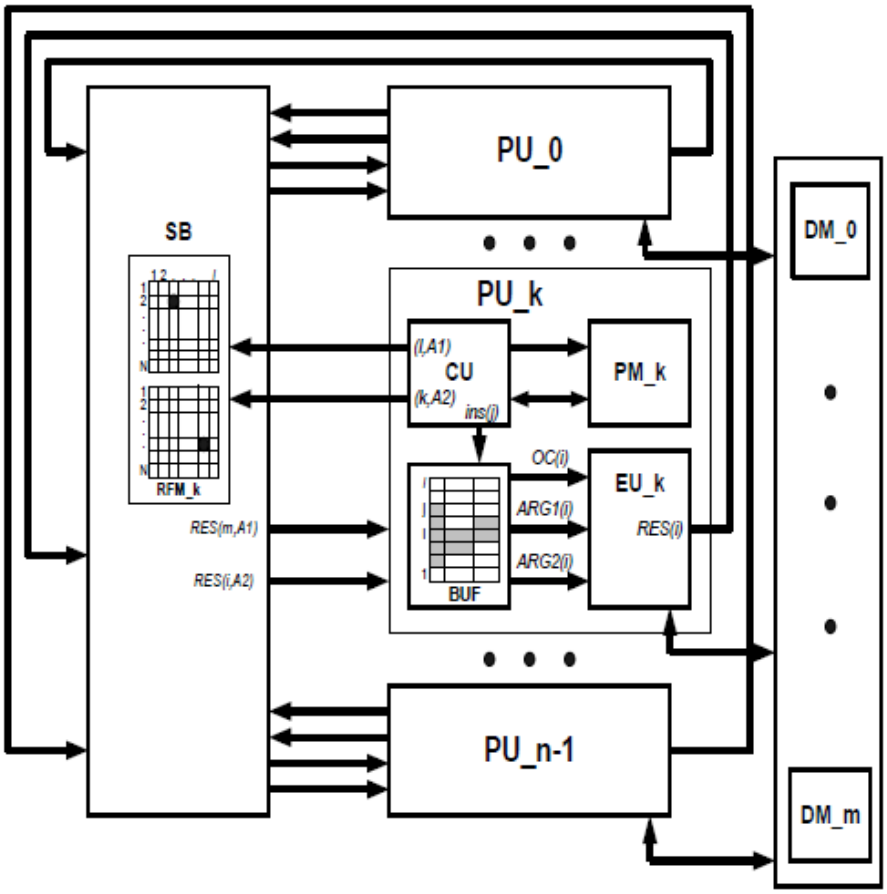


Рисунок 1.11 – Структура Мультиклет (рис. 1 из [58])

Исполнение программы в мультиклеточной архитектуре осуществляется в три этапа. На первом этапе происходит выборка очередного параграфа. На втором этапе инструкции из параграфа равномерно распределяются между всеми клетками таким образом, чтобы адрес каждой следующей инструкции в памяти программ конкретной клетки находился на расстоянии равному N (количеству клеток) относительно исходной последовательности в параграфе. Такое распределение позволяет с большой вероятностью обеспечить высокую степень загрузки всех клеток и *внеочередное исполнение* инструкций.

На третьем этапе осуществляется синхронная выборка инструкций всеми клетками. Выбранные инструкции (триады) получают приоритет, накапливаются в блоках памяти BUF и ожидают активации по мере их наполнения входными данными. После активации инструкция исполняется в соответствии со своим приоритетом. Результаты выполнения инструкций накапливаются в клетках и при возникновении запроса на их использование рассылаются широковещательным способом во все клетки. Промежуточное хранение этих данных в клетках осуществляется в специальной *буферной памяти*. В архитектуре предусмотрены механизмы предотвращения переполнений внутренних буферов клеток, что обеспечивает отсутствие тупиковых ситуаций и разрушения вычислительного процесса.

С точки зрения программиста состояние системы изменяется только по факту исполнения параграфа. Тогда, в процессе исполнения параграфа в архитектуре отсутствует понятие состояния. А клетки – это достаточно простые вычислительные модули, которые могут выполнять только одну инструкцию за раз. Тогда можно сделать вывод, что мультиклеточная архитектура реализует *мелкозернистый* параллелизм.

1.4.2 Параллельная потоковая вычислительная система «Буран»

В институте проблем проектирования в микроэлектронике РАН (ИППМ РАН) ведутся работы над высокопроизводительной архитектурой, основанной на потоковой модели вычислений. Данная архитектура получила название Параллельная потоковая вычислительная система «Буран» (ППВС «Буран»). Авторы ППВС видят в ней «огромный потенциал для создания новых суперкомпьютерых вычислительных систем петафлопсного и сверхпетафлопсного уровня производительности» [59, 60].

Разработчики в работе [59] приводят следующие наиболее значимые особенности ППВС:

- Динамически формируемый контекст – механизм динамической «настройки» токена, который позволяет программе исполняемого узла в процессе исполнения задавать все необходимые атрибуты для результирующего токена.
- Маскирование – дополнительный инструмент управления токенами, позволяющий исключать образование «лишних» пар.
- Кратность – счетчик количества раз использования одного и того же токена, что позволяет снизить нагрузку на коммуникационную среду (не нужно размножать токены).
- Исполняемая программа размножается на все вычислительные модули.
- Аппаратная реализация ассоциативной памяти.

Также в работе [60] авторы вводят понятия парадигм программирования «сбора» и «раздачи». На рисунке 1.12 (рисунок 1 из [60]) приводится принципиальное сравнение данных парадигм. Реализация парадигмы «сбора» требует дополнительных временных задержек, необходимых для сбора данных, требуемых для исполнения очередной инструкции. Поэтому в ППВС «Буран» реализуется парадигма «раздачи», согласно которой сгенерированные в результате вычислений токены сразу рассылаются в места своего потребления.

По утверждению авторов, использование парадигмы «раздачи» наилучшим образом сочетается с использованием ассоциативной памяти. В процессе написания параллельной потоковой программы в парадигме «раздачи» программист может задавать функции распределения вычислительной нагрузки, эффективность которых прямым образом влияет на степень нагрузки на ассоциативную память и, следовательно, на требования к ее объему (как к одному из самых дорогостоящих элементов ППВС) [61].

Рисунок 1.12 – Сравнение парадигм «сбора» (а) и раздачи (б) (рис. 1 из [60])

Следовательно, представленное авторами решение позволяет оптимизировать распределение вычислительной нагрузки чисто программными средствами на этапе программирования и компиляции. Для этих целей авторы ППВС создали специальный язык параллельного потокового программирования DFL, который обеспечивает нативную поддержку парадигмы «раздачи».

Базовым элементом системы является ядро, которое состоит из:

- модуль ассоциативной памяти;
- исполнительное устройство;
- блок регулирования параллелизма;
- внутренний коммутатор, который связывает ядро с другими ядрами и обеспечивает обратную связь;
- блок хеширования для локализации вычислений по группам ядер;
- блок хеширования подмножества.

В работе [59] представлены пути масштабирования ППВС. Это достигается, в частности, за счет использования *полносвязной* коммутационной среды как между отдельными ядрами в группах ядер (небольшого размера, например по 4 ядра), так и между группами ядер. Такая архитектура памяти теоретически позволяет неограниченно наращивать количество ядер в ППВС «Буран». Опираясь на описание архитектуры данной системы, представленное в работах [59-61], можно сделать вывод, что ППВС «Буран» предназначена для реализации

мелкозернистого параллелизма в суперкомпьютерных системах. Но на сегодняшний день эта вычислительная система существует только на модельном уровне. В своих работах авторы представили результаты только модельных испытаний, которые демонстрируют многообещающие результаты.

Подразумевается, что вычислительные ядра данной системы имеют простую микроархитектуру. Следовательно, они могут не поддерживать рассмотренные в разделе 1.1.2.2 виды параллелизма уровня инструкций. Более того, в работе [60] авторы критикуют использование данных механизмов, провозглашая их вынужденными решениями, обеспечивающими простоту реализации парадигмы «сбора». Однако это утверждение является спорным. Механизмы эксплуатации параллелизма уровня инструкций позволяют существенно повысить пропускную способность и производительность отдельно взятого ядра ценой усложнения микроархитектуры. В случае если потоковую архитектуру, функционирующую в парадигме «раздачи», снабдить соответствующими средствами балансировки вычислительной нагрузки, которые учитывают особенности микроархитектуры ядер, то эти механизмы будут реализуемы и в ней.

В целом ППВС «Буран» является перспективной отечественной разработкой, предназначенной для построения суперкомпьютеров. Особого внимания заслуживает специализированный язык потокового программирования DFL. Но, несмотря на использование динамической схемы балансировки нагрузки, из представленных разработчиками результатов не очевидно, будут ли функции балансировки нагрузки работать эффективнее, чем представленные в [24, 25] эвристические алгоритмы. Кроме того, эффективная реализация мелкозернистого параллелизма на практике не представляется возможной. Организация памяти сопоставления токенов в виде полносвязной ассоциативной памяти также не лишена недостатков (дороговизна аппаратных затрат и низкая энергоэффективность).

1.5 Сравнительный анализ ГАРОС, Мультиклет и ППВС Буран

В таблице 1.4 представлены результаты сравнительного анализа ГАРОС, ППВС «Буран» и Мультиклет. Из таблицы видно, что ППВС «Буран» и ГАРОС предназначены для решения абсолютно разных классов задач и предлагают для этого существенно отличающиеся функциональные возможности. В тоже время, Мультиклет и ГАРОС имеют много совпадающих параметров. В частности, обе архитектуры основаны на принципах ПФН-архитектуры, имеют одинаковое количество ядер и схожую модель программирования (параграфы – это аналоги капсул, а триады – аналог ЭСД). Результаты этого анализа позволяют сделать вывод, что Мультиклет является наиболее перспективной архитектурой для сравнения с ГАРОС. Однако, автору не удалось найти в открытом доступе данные испытаний Мультиклета на наборе ключевых алгоритмов ЦОС, кроме алгоритма БПФ.

Таблица 1.4 – Сравнительный анализ архитектур

Критерий	Мультиклет	ППВС Буран	ГАРОС
Система команд	Аналог RISC	Предположительно аналог RISC	Аналог RISC
Класс решаемых задач	Общего назначения, ЦОС	Массового параллелизма	ЦОС
Количество ядер	4	Зависит от конфигурации	4
Межъядерный обмен	Полносвязный	Полносвязный	Централизованный
Тип информационной связи	Прямая	Косвенная	Прямая
Кол-во инструкций, выполняемых за 1 такт на 1 ядре	1	1	Прототип: 1 Теоретически: до 4-х
Кол-во блоков памяти на ядро	Три: - память программ - буфер активных триад - буферная память промежуточных данных	Один: - модуль ассоциативной памяти	Два: - память адресной проверки - память ветвлений
Средства программирования	Библиотека расширения языка Си	Язык DFL	- Язык Си для УУ - Ассемблер для разработки капсул
Представление программы	Двухуровневая: параграфы и триады	Одноуровневая: на языке DFL	Трехуровневая: управляющий код, капсулы, ЭСД
Размещение команд в программе	Частично упорядоченное	Неупорядоченное	Частично упорядоченное
Балансировка нагрузки	На уровне параграфов: статическая На уровне триад: динамическая	Динамическая	На уровне капсул: динамическая На уровне ЭСД: статическая
Степень реализации параллелизма	Мелкозернистый	Мелкозернистый	Мелкозернистый / Крупнозернистый (предпочтительно)
Реконфигурация при сбое	Обеспечивается	Не обеспечивается	Не обеспечивается
Состояние разработки	Серийная модель	Программная модель	Аппаратная модель
Скорость вычисления 256-точечного комплексного БПФ (такты)	1192	Н/Д	1078

1.6 Выводы к главе 1. Постановка задачи исследования

В ходе исследования МПРА и ее прототипа ГАРОС были установлены их потенциальные возможности для решения характерных проблем реализации потоковых архитектур.

Проблема *организации памяти токенов* решается путем использования механизма прямо адресуемой памяти для сравнения тегированных токенов.

Проблема *построения структур данных* частично решается путем представления капсул в виде I-структур. Данные структуры данных хранятся в специальной Буферной Памяти, которая является интерфейсом взаимодействия между двумя уровнями ГАРОС. Построение и загрузка капсулы входными данными осуществляется управляющим уровнем архитектуры. А наполнение ее выходными результатами – операционным уровнем архитектуры.

Проблема *распределения вычислительной нагрузки* в рамках капсулы решается статическим образом. Однако делается вывод о недостаточности реализации и использования только эвристических алгоритмов.

Проблема *конвейеризации последовательных вычислений* решается с точки зрения двух подходов. Во-первых, последовательные фрагменты решаемой задачи могут быть вычислены с помощью процессора управляющего уровня архитектуры. А во-вторых, длина конвейера операционного уровня архитектуры сокращена до минимально возможной и равна 3. Это позволяет иметь минимальные задержки пересылки промежуточных данных и, соответственно, - время простоя вычислительных блоков.

Проблема *поддержки высокопроизводительной микроархитектуры* решена только частично. С одной стороны, компоновка функциональных блоков Вычислителей теоретически позволяет выполнять до 4 инструкций за такт. С другой стороны, отсутствуют какие-либо механизмы настройки Вычислителей на подобные режимы функционирования. Исключением является только специализированная операция «бабочка» алгоритма БПФ.

Проблема *обработки константных данных* достаточно полно решена за счет реализации нескольких видов Памятей констант. Однако практика разработки капсул автором диссертации показала, что этого оказалось недостаточно.

Также в главе приводятся результаты сравнительного анализа отечественных вычислительных систем потоковой архитектуры. Результаты данного анализа показывают конкурентоспособность ГАРОС по сравнению с Мультиклетом и обосновывают актуальность дальнейшей разработки прототипа МПРА и его испытаний путем синтеза ПЛИС прототипа.

Таким образом, для достижения поставленной во введении цели диссертационного исследования необходимо решить следующие **задачи**:

1) Разработать структурные элементы архитектуры, методы и алгоритмы их функционирования, которые позволят: эффективно реализовать поддержку параллелизма на различных уровнях; минимизировать избыточность тегированных данных; достичь требуемого уровня производительности для задач ЦОС реального времени.

Для этого необходимо:

- исследовать существующую версию архитектуры и ее функциональных возможностей, а также выполнить анализ проблем реализации задач ЦОС средствами существующего прототипа ГАРОС;
- разработать спецификацию ГАРОС путем доработки или ввода новых структурных блоков и механизмов их функционирования для повышения степени эффективности решения типовых проблем потоковых архитектур, а также для решения проблем, которые будут обнаружены в результате выполнения анализа;

- разработать методы организации суперскалярных вычислений на уровне микроархитектуры вычислительных ядер ГАРОС и алгоритмы их функционирования в суперскалярном режиме, чтобы повысить эффективность использования параллелизма уровня инструкций;
- разработать методы и средства аппаратной поддержки реализации алгоритма БПФ по основанию два с прореживанием по времени, который является ключевым алгоритмом ЦОС;

2) Разработать теоретические основы программируемости ГАРОС, которые включают в себя: методики и алгоритмы реализации различных этапов разработки и отладки ПО; программную и аппаратную поведенческие модели архитектуры для проведения испытаний; набор средств аппаратно-программного моделирования и отладки архитектуры.

3) Осуществить испытания всех разработанных средств путем реализации демонстрационной задачи распознавания изолированных слов и комплекта типовых алгоритмов ЦОС для подтверждения эффективности полученных результатов работы путем сравнения с современным высокопроизводительным сигнальным процессором.

Глава 2 Разработка прототипа МПРА для задач цифровой обработки сигналов

В данной главе приводятся основные научные и практические результаты диссертационной работы. Осуществляется анализ существующих проблем реализации алгоритмов ЦОС средствами прототипа архитектуры. На основе результатов данного анализа приводятся методы, алгоритмы и механизмы функционирования усовершенствованных элементов ГАРОС. Были усовершенствованы (или разработаны): микроархитектура Вычислителей и их система команд; расширенные алгоритмы и механизмы косвенной репликации в Распределителе; методы и алгоритмы обеспечения многократного исполнения капсул; методы и алгоритмы обработки потока выходных данных. Наиболее значимым результатом является существенная переработка средств аппаратной поддержки вычисления Быстрого Преобразования Фурье, которая затронула все ключевые компоненты архитектуры. Усовершенствованные средства поддержки БПФ имеют на ~20% меньшие аппаратные издержки и вычисляют типовой 256-точечный БПФ по основанию 2 на ~12% быстрее.

2.1 Анализ проблем реализации задач ЦОС в существующей версии прототипа МПРА

ГАРОС разрабатывается с целью подтверждения гипотезы, что МПРА, будет обладать высоким уровнем производительности. В работах [39, 40] показано, что область ЦОС наилучшим образом подходит для тестирования и испытания МПРА. Анализ приведенных требований и области существующих массовых приложений позволил выделить в качестве представительских задач для тестирования предлагаемой архитектуры задачи обработки и передачи как голосовой, так и цифровой информации. В качестве тестовой выбрана задача РИС, т.к. уже существовало ее эталонное решение на традиционной архитектуре.

Макетная реализация задачи РИС обладает следующими характеристиками:

1) Эталонная целочисленная реализация на языке C++.

2) Реализация на ассемблере цифрового сигнального процессора dsPIC30F, которая обладает бит-экзектности относительно C++ реализации.

3) Уровень распознавания 96% на библиотеке слов без шумов и 92% с максимальным уровнем шума -15 dB.

4) Параметры скоростей вычисления (в циклах) основных алгоритмов РИС на одноядерном dsPIC30F.

Опыт разработки автором настоящей работы капсул для решения задачи РИС показал, что функциональных возможностей существующей версии прототипа МПРА недостаточно для эффективной реализации многих алгоритмов ЦОС. В главе 4 работы будут представлены

результаты декомпозиции задачи РИС на набор капсул. Несмотря на то, что каждый из выбранных алгоритмов удалось реализовать в виде капсул, среднее значение коэффициента ускорения вычислений относительно одноядерного dsPIC30F составило примерно 2-2.5. Фактически наиболее быстро вычисленным алгоритмом оказался БПФ за счет активного использования суперскалярности микроархитектуры вычислительных ядер. Для 4-ядерного вычислительного устройства данный результат не мог считаться удовлетворительным.

Поэтому был проведен анализ узких мест ГАРОС, которые не позволили добиться приемлемого уровня производительности и/или накладных расходов реализации. На основе результатов данного анализа были сформулированы следующие проблемы:

1) Низкая мощность регистрового файла Вычислителей.

Ранее на рисунке 1.10 приводилась структурная схема устройства MAC – основного модуля вычислительного ядра РОУ. Модуль включает в себя всего 3 регистра, доступных программисту для хранения промежуточных значений. Это приводит к необходимости организовывать плотный трафик промежуточных E-операндов. Межсекционный обмен организован в РОУ централизованным образом посредством Распределителя, следовательно, каждый промежуточный операнд проходит все этапы конвейера. Это приводит к снижению производительности из-за задержек конвейера.

2) Необходимость дублирования некоторых входных данных

Данная проблема является следствием предыдущей. Данные, которые могли бы храниться в регистровом файле и быть повторно использованы, приходится хранить в ПАП. Емкость ПАП значительно ограничена (всего 16 56-разрядных операндов). Поэтому иногда при программировании капсулы приходится дублировать одни и те же данные, чтобы обеспечить эффективную балансировку нагрузки. Необходимость дублирования приводит к усложнению процесса упаковки входных данных и усложнению алгоритма УУ для загрузки капсулы.

3) Недостаточная полнота системы команд

Практика разработки капсул показала, что ряд команд из таблицы 1.3 не использовались. В тоже время для работы с регистрами [A], [B], [C] (а в случае увеличения мощности регистрового файла тем более) команд оказалось недостаточно.

4) Фактическая однозадачность Вычислителей

Большинство современных высокопроизводительных ЦСП процессоров (например, TMS C55x серии) активно используют суперскалярность микроархитектуры их вычислительных ядер. Это позволяет использовать параллелизм уровня инструкций и исполнять 2 и более инструкций за 1 такт. В микроархитектуре MAC-блока есть все необходимое для поддержки многозадачности, но в текущей версии прототипа она реализована только в рамках команды BUTT («бабочка» алгоритма БПФ). Реализация суперскалярных вычислений также требует

данных, которые обрабатываются одновременно исполняемыми инструкциями. Поэтому данная проблема косвенно связана с проблемой 1), т.к. наличие большего количества регистров позволит обеспечить данными большее количество инструкций.

5) Работа с константными данными

Несмотря на то, что в РОУ уже реализовано несколько видов памятей констант, разработка капсул задачи РИС показала, что существует необходимость использования очень большого количества констант, которые описывают библиотеку моделей распознаваемых слов. При этом нет никакой возможности хранить подобный объем данных средствами памятей констант РОУ. Это требует ввода дополнительных средств подкачки констант, сохраняющих идентичное с существующими видами памяти констант поведение.

6) Реентерабельность капсул

Многие задачи ЦОС, являются задачами реального времени. Одним из типовых алгоритмов является фильтр с конечной импульсной характеристикой. Существует две его разновидности: блочный и реального времени. Блочный фильтр за один запуск должен отфильтровать весь блок входных отсчетов и на выходе сформировать блок выходных отсчетов того же размера. Фильтр реального времени за один фрейм обрабатывает ровно 1 отсчет.

В случае реализации блочного фильтра входной массив отсчетов должен быть многократно обработан. Аналогичное поведение должно быть реализовано и для капсулы. В текущей версии прототипа это может быть реализована только с помощью программы УУ, которая будет для каждой итерации формировать новую капсулу и запускать ее на исполнение. Такое поведение приводит к существенным потерям производительности.

7) Неполнота механизмов централизованного управления потоками данных

Данная проблема является следствием проблем 2) и, косвенно 6). Существующих механизмов косвенной репликации операндов в Распределителе оказалось недостаточно для организации эффективной балансировки нагрузки между секциями. Из-за чего приходилось дублировать некоторые входные данные. Кроме того, между упакованными данными и механизмами косвенной репликации оказалась плохая синергия. В ряде случае приходилось отказываться от упаковки данных только для того, чтобы эффективно сбалансировать нагрузку (в том числе дублируя данные). Поэтому необходимо разработать более функциональные режимы косвенной репликации, которые будут хорошо сочетаться как с упакованными данными, так и с многократным исполнением капсул.

8) Обработка выходных данных

Первоначальная спецификация механизма обработки выходных данных подразумевала формирование выходной капсулы из выходных наборов данных. Однако с введением Буферной Памяти этот механизм стал избыточен. Кроме того, в процессе разработки капсул задачи РИС

возникла необходимость загрузки промежуточных данных вычислений в капсулу для дальнейшего использования в многократном режиме. Поэтому потребовалась переработка метода обработки выходных данных.

9) Аппаратная поддержка алгоритма БПФ

Существующая реализация аппаратной поддержки алгоритма БПФ в ГАРОС оказалась слишком громоздкой по аппаратным затратам, а также накладным расходам организации перехода от одной стадии алгоритма БПФ к другой. Размещение промежуточных данных на место входных плохо сочетается с упаковкой данных, которую нельзя не использовать ввиду значительного объема окна отсчетов (256 и более отсчетов). Поэтому требуется существенная переработка механизмов аппаратной поддержки БПФ.

2.2 Развитие микроархитектуры Вычислителей

2.2.1 Структура усовершенствованного MAC-блока

Для решения проблем 1) и 2) раздела 2.1 необходимо ввести в состав Вычислителя новые регистры, а также усовершенствовать существующие. Этими регистрами стали:

1) Два регистра на входе Вычислителя

Каждый Вычислитель принимает на вход от Жонглера два входных данных. Поэтому естественным способом увеличения регистрового файла стало добавление двух 40-разрядных регистров на входе Вычислителя, названных [RL] и [RR]. Регистры [RL] и [RR] соответствуют L- и R- входам Вычислителя и могут быть использованы для хранения данных, которые приходят на эти входы. Запись в данные регистры осуществляется несколькими способами:

- Напрямую с помощью соответствующего кода операции.
- С помощью механизма записи одного из данных пары, который активируется в случае, если одно из входных данных представлено заменяемым Cacn-операндом. Например, на L-вход пришло данное с именем *a*, на R-вход пришел Cacn-операнд, который несет в себе кодировку конфигурации записи в [RL]. Тогда *a* будет записано в середину регистра [RL], если оно 16-разрядное, или целиком, если оно 38-разрядное.
- Оба регистра могут быть записаны одновременно таким образом, что в [RL] попадает данное с L-входа, а в [RR] – данное с R-входа. Это происходит в случае получения Вычислителем настройки на специальный режим выполнения операции в ***[Ot]***-подполе.
- В регистры могут быть записаны результаты выполнения операций сдвига, округления, пересылки и взаимного обмена над данными других регистров.

Данные регистры также могут быть использованы для «прокачки» данных путем копирования их содержимого на последнем подшаге текущего такта работы Вычислителя в регистры [B] или [C]. Программируемость рассмотренных функциональных возможностей

обеспечивается с помощью модифицированных форматов операндов и функциональных полей. Для повышения гибкости работы с регистрами Вычислителя (т.к. их стало больше) ограничение на прием только одного Cacn-операнда было снято.

Множество доступных инструментов записи в регистры [RL] и [RR] может привести к ситуации ошибочного программирования, в которой на одном такте будет осуществлено более одной попытки записи. Для избегания данной ситуации разработан алгоритм, который определяет приоритетность исполнения соответствующей операции записи и учитывает возможность получения двух Cacn-операндов на входе Вычислителя.

2) Аккумулятор [A] увеличен до 40-разрядов

Данное изменение обеспечивает единообразие взаимодействия с регистрами посредством Cacn-операндов и соответствующих кодов операций.

3) Специальный регистр [S]

В ходе реализации алгоритмов задачи РИС зачастую было необходимо выполнять операции нормализации вида: $result = result >> (5 - Order)$. Другими словами, возникла необходимость осуществить сдвиг значения регистра на величину, вычисляемую программным образом. Для существующей версии Вычислителя и его системы команд такой возможности не предусмотрено. В тоже время dsPIC30F позволяет осуществлять сдвиги не только на #lit (от *litera* – значение константы), но и на значение, хранящееся в пользовательских регистрах [wX].

Для решения данной проблемы в состав Вычислителя введен 5-разрядный регистр [S] (shift), значение которого обновляется одновременно с записью значения в регистр [A]. Разряды [4..0] регистра [S] принимают значения разрядов [20..16] регистра [A] соответственно. Данные регистра [S] интерпретируется как знаковое. Отрицательное значение означает сдвиг влево, положительное значение – сдвиг вправо. С помощью данного регистра может быть запрограммирован сдвиг в диапазоне от 15 разрядов влево, до 15 разрядов вправо. Величина сдвига вычисляется программным образом в соответствии с алгоритмом. Инициация данного режима сдвига осуществляется при выполнении EXOP операций RND и SFTR в том случае, если разряды [7..3] имеют значение [10000]. EXOP операция интерпретируется как Extended Operation (расширенная операция), а ее семантика будет рассмотрена в разделе 2.2.2. Для символьного обозначения в капсуле значению разрядов [7..3] = [10000] соответствует символ “S”. Тогда, например, операция сдвига значения регистра [A] на величину, хранящуюся в регистре [S], и помещением результата в регистр [A] в капсуле будет выглядеть так «s_aSa».

4) Скрытые регистры [W2], [W3]

Данные 40-разрядные регистры были введены для организации эффективного вычисления алгоритма БПФ. Их назначение и методы доступа будут рассмотрены в разделе 4.7.

На рисунке 2.1 представлена структура доработанного Вычислителя.

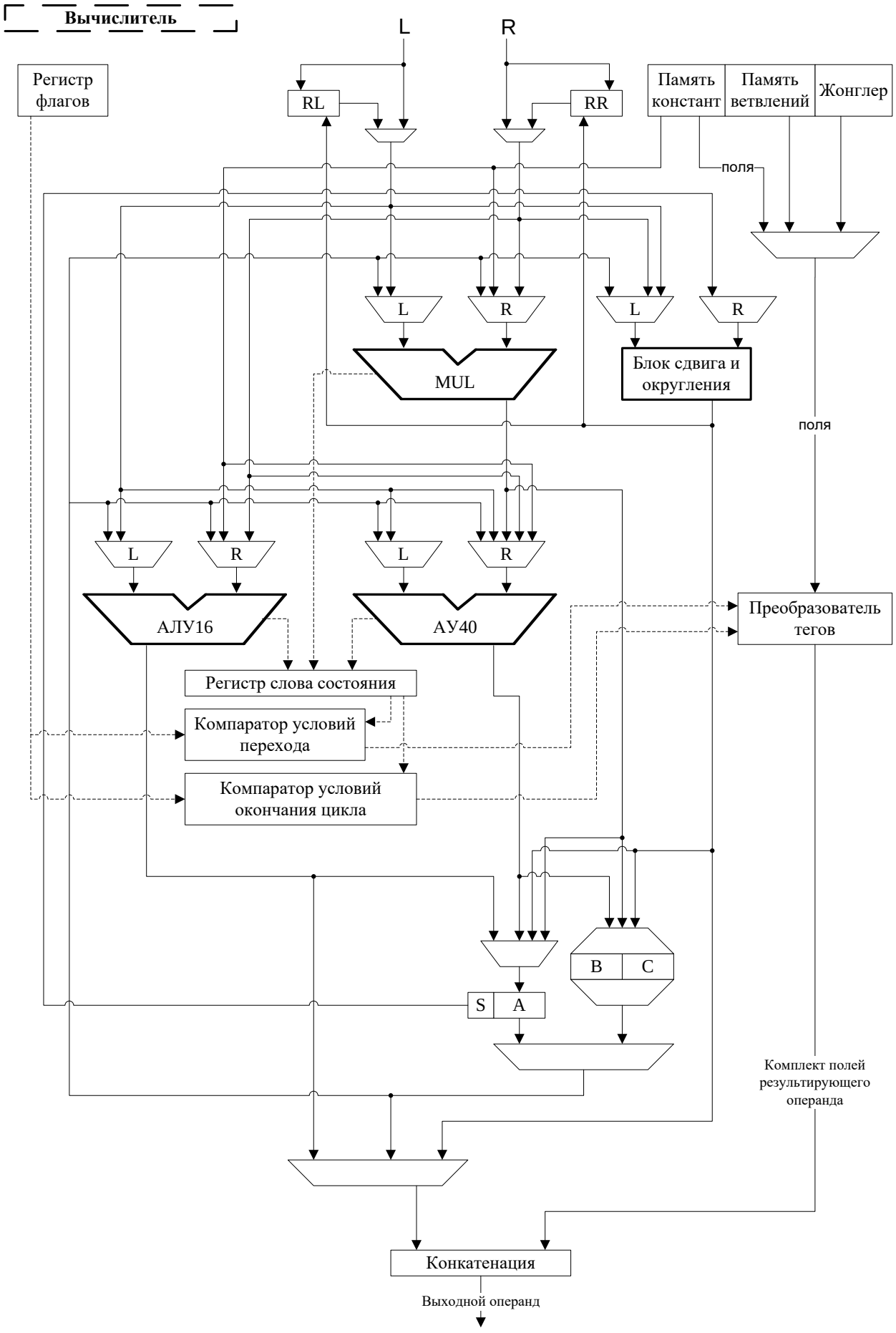


Рисунок 2.1 – Структура усовершенствованной версии Вычислителя

2.2.2 Развитие системы команд

Появление новых структурных блоков Вычислителя потребовало переработки и расширения существующей системы команд. Она была логически разделена на 2 части: операции, требующие наличие данных в содержательной части операнда, и – не требующие, соответственно. С точки зрения реализации – второе множество команд может быть описано в содержательной части. Это подразумевает двойную дешифрацию (исходного кода операции и дешифрацию содержательной части). Содержательная часть представляет собой 16-ти разрядное слово. Для 38-разрядных содержательных операндов применять двойные команды нецелесообразно, т.к. они имеют ограниченный набор функциональных полей. Новая составная операция получила название Extended Operation (EXOP).

Увеличение количества регистров Вычислителя потребовало ввода новых кодов операций для управления их содержимым. Данные операции могут быть закодированы в содержательной части, т.к. не требуют входных данных. Это позволило разработать следующий механизм кодировки EXOP-операций, представленный в таблице 2.1. Таким образом, используя содержательную часть можно закодировать еще 32 базовых операции, а общая мощность системы команд составляет 64 кода операции.

Таблица 2.1 – Структура кодировки содержательной части для EXOP-операции

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
								знак							
Код операции					Регистр источник			Длина сдвига					Регистр приемник		

Основными результатами переработки системы команд РОУ стали:

1) Неиспользуемые коды операций были удалены для освобождения кодировок, к ним относятся: SHR, SHL, LMS, LSP, SUB, MACab, MACsb.

2) Большинство кодов операций для манипуляции регистрами [A], [B], [C] были удалены и заменены на EXOP-кодировки, к ним относятся: CLR, MOV, MOVl, MOVd, MOVh, RND.

3) Добавлены новые операции в основные коды, а именно: SUBi, SUBp, EDAC, LMl, LMr, EXOP, SQ, SQac, SQsc, BSR.

4) Добавлены новые операции в расширенные коды, а именно: SFTR, EXCR, MPR, MACR, MSCR, ADDR, SUBR.

В таблице А.1 Приложения А приводится переработанная система команд РОУ.

2.2.3 Методы поддержки многозадачности

Из рисунка 2.1 видно, что микроархитектура Вычислителя содержит 4 блока, которые могут функционировать одновременно: АЛУ, АУ40, Умножитель и Сдвигатель-Округлитель. Таким образом, Вычислитель теоретически может выполнять до четырех инструкций

одновременно и, следовательно, является четырехзадачным (в англоязычной терминологии – 4-issue). Однако входной интерфейс Вычислителя позволяет получить на входе только два входных данных и 2 инструкции. Данное ограничение не может быть преодолено в рамках данного исследования. Поэтому Вычислитель следует рассматривать как двухзадачный (2-issue).

Основной метод организации суперскалярных вычислений заключается в передаче на L- и R- входы Вычислителя двух различных кодов операций, которые должны быть совместимы между собой. Под совместимостью понимается отсутствие конфликтов доступа к регистрам и исполнительным блокам (например, операции MUL и MAC несовместимы между собой, т.к. требуют для исполнения умножитель).

Вторым методом организации суперскалярных вычислений является поддержка двух схем вычислительного процесса. Первая схема (или первый тип суперскалярных вычислений) является типовой и заключается в *одновременном* исполнении двух различных инструкций. Результат одной из них помещается в один из регистров [A], [B] или [C], а результат второй – отправляется на E-шину в качестве выходного результата. Вторая схема (или второй тип суперскалярных вычислений) заключается в *последовательном* исполнении двух инструкций. При этом результат первой инструкции используется в качестве входного данного для второй инструкции. Схема суперскалярных вычислений кодируется в ***%Cs-поле*** конфигуратора секции.

Источниками данных для суперскалярных вычислений являются:

- L- и R- входы Вычислителя;
- константа из ПК_С, ПК_СП или ПК_СР;
- регистры [RL], [RR], [A], [B], [C];
- результат исполнения первой инструкции при втором типе суперскалярных вычислений.

Для формирования совместимых пар кодов операций система команд была разбита на несколько подмножеств, названных OpSet’ами. Объявлены следующие множества: OpSet0 (операции умножения); OpSet1 (операции умножения с накоплением), OpSet2 (арифметические операции, сдвиги и пересылки); OpSet3 (операции, не вошедшие в OpSet0 и OpSet1); OpSetAr (арифметические операции и сдвиги); OpSetMOV (пересылки).

Коды операций из OpSet3 являются несовместимыми для суперскалярных вычислений. Если один из полученных Вычислителем кодов операций входит в OpSet3, то блок Конфигуратор настраивает Вычислитель на скалярные вычисления по приоритету инструкций. При настройке РОУ на суперскалярный режим вычислений Жонглер определяет только набор функциональных полей и пару кодов операций, а распределение данных по шинам Вычислителя зависит от сочетания кодов операций. Для выполнения операций сложения вычитания на

АЛУ16 с привлечением значений из [A], [B], [C] извлекаются [31..16] разряды соответствующего 40-разрядного регистра («средняя» часть).

На рисунке 2.2 приводится листинг описания данных множеств в формате XML.

```
<OpSets>
	<OpSet name="0">
		<value name="MULuua" code="00110" symbol="*ua" EXOP="false"/>
		<value name="MULssa" code="00111" symbol="*sa" EXOP="false"/>
		<value name="MULuub" code="01000" symbol="*ub" EXOP="false"/>
		<value name="MULuuc" code="01001" symbol="*uc" EXOP="false"/>
		<value name="MULssb" code="01010" symbol="*sb" EXOP="false"/>
		<value name="MULssc" code="01011" symbol="*sc" EXOP="false"/>
		<value name="SQ" code="11001" symbol="*2" EXOP="false"/>
		<value name="MPR_" code="01101" symbol="mr_" EXOP="true"/>
	</OpSet>
	<OpSet name="1">
		<value name="MACac" code="01100" symbol="*+c" EXOP="false"/>
		<value name="MACsc" code="01101" symbol="*-c" EXOP="false"/>
		<value name="SQac" code="11010" symbol="*2+c" EXOP="false"/>
		<value name="SQsc" code="11011" symbol="*2-c" EXOP="false"/>
		<value name="MACR_" code="01110" symbol="m+r_" EXOP="true"/>
		<value name="MSCR_" code="01111" symbol="m-r_" EXOP="true"/>
	</OpSet>
	<OpSet name="2">
		<value name="ADD" code="00001" symbol="+" EXOP="false"/>
		<value name="SUBi" code="00011" symbol="-" EXOP="false"/>
		<value name="SUBp" code="00100" symbol="-p" EXOP="false"/>
		<value name="ASR" code="10011" symbol="a&amp;gt;" EXOP="false"/>
		<value name="ASL" code="10100" symbol="a&amp;lt;" EXOP="false"/>
		<value name="BSR" code="11100" symbol="*br" EXOP="false"/>
		<value name="MOVl_" code="00110" symbol="&amp;lt;l_" EXOP="true"/>
		<value name="MOVd_" code="00111" symbol="&amp;lt;d_" EXOP="true"/>
		<value name="MOVh_" code="01000" symbol="&amp;lt;h_" EXOP="true"/>
		<value name="RND_" code="01001" symbol="r_" EXOP="true"/>
		<value name="MOV_" code="01010" symbol="&amp;lt;_" EXOP="true"/>
		<value name="SFTR_" code="01011" symbol="s_" EXOP="true"/>
	</OpSet>
	<OpSet name="3">
		<value name="NOT" code="10000" symbol="~" EXOP="false"/>
		<value name="EDAC" code="10001" symbol="edac" EXOP="false"/>
		<value name="ASR" code="10011" symbol="a&amp;gt;" EXOP="false"/>
		<value name="ASL" code="10100" symbol="a&amp;lt;" EXOP="false"/>
		<value name="LMb" code="10110" symbol="&amp;gt;db" EXOP="false"/>
		<value name="LMc" code="10111" symbol="&amp;gt;dc" EXOP="false"/>
		<value name="LMl" code="11000" symbol="&amp;gt;dl" EXOP="false"/>
		<value name="LMl" code="11001" symbol="&amp;gt;dr" EXOP="false"/>
		<value name="EXOP" code="11010" symbol="e" EXOP="false"/>
		<value name="SQ" code="11001" symbol="*2" EXOP="false"/>
	</OpSet>
	<OpSet name="Ar">
		<value name="ADD" code="00001" symbol="+" EXOP="false"/>
		<value name="SUBi" code="00011" symbol="-" EXOP="false"/>
		<value name="SUBp" code="00100" symbol="-p" EXOP="false"/>
		<value name="RND_" code="01001" symbol="r_" EXOP="true"/>
		<value name="SFTR_" code="01011" symbol="s_" EXOP="true"/>
	</OpSet>
	<OpSet name="MOV">
		<value name="MOVl_" code="00110" symbol="&amp;lt;l_" EXOP="true"/>
		<value name="MOVd_" code="00111" symbol="&amp;lt;d_" EXOP="true"/>
		<value name="MOVh_" code="01000" symbol="&amp;lt;h_" EXOP="true"/>
		<value name="MOV_" code="01010" symbol="&amp;lt;_" EXOP="true"/>
	</OpSet>
</OpSets>
```

Рисунок 2.2 – Описание множеств кодов операций

Суперскалярный режим первого типа кодируется значением %Cs=1. В суперскалярном режиме первого типа операции выполняются параллельно на независимых вычислительных блоках, поэтому значения из регистров [A], [B], [C] используются до их возможной модификации («старые» значения). Также при Cs=1 режим Совместимыми операциями для

организации суперскалярных вычислений первого типа являются сочетания из: OpSet0 и OpSet2; OpSet1 и OpSet2. Для всех остальных сочетаний фиксируется скалярный режим.

Суперскалярный режим второго типа кодируется значением %Cs=2. В суперскалярном режиме второго типа операции выполняются последовательно, поэтому значения из регистров [A], [B], [C] используются после их возможной модификации («новые» значения). Совместимыми операциями для организации суперскалярных вычислений второго типа являются сочетания из: OpSet0 и OpSetMOV; OpSet1 и OpSetMOV; OpSetAr и OpSetMOV; OpSet0 и OpSetAr. Для всех остальных сочетаний фиксируется скалярный режим.

2.2.4 Алгоритмы функционирования усовершенствованного Вычислителя

После усовершенствования компонента алгоритм обобщенной работы Вычислителя состоит из 3 последовательных этапов:

1) Дешифрация комплекта функциональных полей и инструкций на L- и R- входах Вычислителя. Выполняет блок Дешифратор, который формирует на выходных шинах комплект распознанных параметров конфигурации. К ним относятся:

- код режима суперскалярных вычислений Cs-поля (Cs);
- код операции на L-входе / R-входе (LOpcode / ROpcode);
- признак того, что LOpcode / ROpcode это EXOP операция (LIsExop / RIsExop);
- признак того, что LOpcode / ROpcode входит в множество кодов операций OpSet0 (LInOpSet0 / RInOpSet0);
- признак того, что LOpcode / ROpcode входит в множество кодов операций OpSet1 (LInOpSet1 / RInOpset1);
- признак того, что LOpcode / ROpcode входит в множество кодов операций OpSet2 (LInOpSet2 / RInOpset2);
- признак того, что LOpcode / ROpcode входит в множество кодов операций OpSetAr (LInOpSetAr / RInOpsetAr);
- признак того, что LOpcode / ROpcode входит в множество кодов операций OpSetMOV (LInOpSetMOV / RInOpsetMOV);
- входное данное на L-входе / R-входе (LData / RData);
- признак того, что LData / RData не 16-разрядное (L38 / R38);
- имя регистра-источника данного на L-входе / R-входе (LSourceReg / RSourceReg), имеет значение NON если на L-входе / R-входе не пришел Cacn-операнд;
- признак того, что нужно считать константу из ПК_С (UseCM_S);
- признак того, что нужно считать константу из ПК_СР (UseCM_SR);
- признак того, что нужно считать константу из ПК_СП;

• код настройки Ar-поля, определяющего режим чтения или записи регистров [RL] и [RR] (Ar);

• код настройки Aab-поля, определяющего режим «прокачки» содержимого регистров [RL] и [RR] или режим обработки результата вычислений (Aab);

• код настройки Ot-поля, определяющего требуется ли: осуществлять запись в регистры [RL] и [RR] входных данных, привлекать для выполнения константу из ПК_С / ПК_СР / ПК_СП, инициировать проверку флагов перехода, инициировать декрементацию счетчика циклов и проверку окончания цикла (Ot);

• код типа перехода (BranchType);

• код условия перехода Bb-поля (Bb);

• код адреса чтения функциональных полей из памяти ветвлений при срабатывании условия перехода Bm-поля (Bm);

• код загрузки регистра флагов перехода Bf-поля (Bf);

• код источника значения счетчика циклов Bs-поля (Bs);

• код значения счетчика циклов Bi-поля (Bi);

• признак того, что результат должен быть передан по E-шине в Распределитель (shDs);

• признак того, что результат должен быть передан на Em-шину (shDe);

• признак того, что результат должен быть помещен в аккумулятор [A] или обработан иным образом (shAab);

• код типа, который должен быть присвоен результирующему операнду (Ti).

2) Конфигурация. Выполняет блок Конфигуратор, который переключает все необходимые мультиплексоры и выставляет на шины требуемые данные и параметры. В зависимости от полученных настроек обеспечивает скалярный или суперскалярный режим исполнения инструкций. Конфигурация состоит из нескольких этапов, к которым относятся:

• Настройка в соответствии с секционным конфигуратором Ccs-операндом, если он пришел в паре операндов. Алгоритм тривиален, настраиваются параметры длины сдвига для операции BSR и режим автоматического округления результатов умножения.

• Настройка записи в регистры [RL] и [RR] в соответствии со значениями параметров Ot, Ar, LSourceReg, RSourceReg. Алгоритм представлен в Приложении А на рисунке А.1.

• Настройка источников данных L- и R-входов в соответствии со значениями параметров Ar, LSourceReg, RSourceReg. Алгоритм представлен в Приложении А на рисунке А.2. После настройки значений порту LSourcePort устанавливается в соответствие значение RealLInput, а порту RSourcePort – RealRInput соответственно.

• Настройка памяти констант в соответствии со значением параметра Ot. Алгоритм тривиален, в зависимости от значения Ot считывается константа из соответствующей памяти констант и выставляется в качестве источника данных.

• Настройка режима суперскалярных вычислений в соответствии с настройками Ccs, а также значениями параметров LOpcode, ROpcode, Ot. Алгоритм настройки однозадачных вычислений представлен в Приложении А на рисунке А.3. Алгоритм настройки суперскалярного режима типа 1 представлен в Приложении А на рисунке А.4. Алгоритм настройки суперскалярного режима типа 2 представлен в Приложении А на рисунке А.5.

• Настройка режима перехвата и обработки переходов в соответствии со значениями параметров Ot, RData, R38, Bs, Bi, BranchType, Bb, Bf. Алгоритм представлен в Приложении А на рисунке А.6.

• Настройка обработки результатов вычислений в соответствии со значением параметра Aab. Алгоритм представлен в Приложении А на рисунке А.7.

• Выполнение инструкции(-ий) и формирование результата на E-шине. Инструкции исполняются вычислительными блоками (АЛУ, Умножитель, АУ40, BS&R), преобразование полей и формирование результата осуществляется Преобразователем тегов.

При записи данных в регистры учитывается их размер. Если они 38-разрядные, то данные записываются целиком (Full). Если же они 16-разрядные, то данные выравниваются в 40-разрядную сетку и записываются в разряды [31..16] (Middle). Также сохраняется признак знака. При чтении данных предоставляется доступ ко всем частям регистров: старшей (High), средней (Middle), младшей (Low) и целиком (Full).

2.3 Память констант подгружаемая

Ранее было указано, что существующих видов памяти констант оказалось недостаточно для работы с большими библиотеками констант. Поэтому был разработан еще один вид памяти констант, который был назван память констант секционная подгружаемая (ПК_СП).

ПК_СП предназначена для хранения данных двух типов:

• данные заранее известны, имеют достаточно большой объем и хранятся во внешней памяти (вне РОУ и БП). Примером таких данных для РИС являются тренированные модели слов, с которыми поочередно и будет производиться сравнение произнесенного слова диктором. Для РИС этот объем не превышает 100 слов, а для расширенного словаря – 5-10 тысяч слов;

• данные ограниченного объема, заранее неизвестны, но их объем существенно больше объема ПК_СР.

Программист-разработчик капсул принимает решение о целесообразности помещения входных данных не в капсулу, а именно в ПК_СП, если последние могут быть использованы в

суперскалярных вычислениях, предоставляя третий источник данных для обработки секционным Вычислителем на текущем вычислительном шаге. Именно поэтому помещение таких данных в капсулу было бы менее эффективным решением – удлинением числа вычислительных шагов по их обработке.

ПК_СП организована в виде двух отдельных однопортовых блоков небольшой размерности на 16 операндов – ПК_СП_1 и ПК_СП_2. При запуске ГАРОС оба блока имеют состояние «пустой». Программа УУ инициирует заполнение ПК_СП по каналу прямого доступа начиная с блока ПК_СП_1. Одновременно в контроллер секционной ПК_СП передается адрес последнего операнда, который должен быть считан. После окончания заполнения ПК_СП_1, сразу же автоматически начинается заполнение ПК_СП_2, а блок ПК_СП_1 переводится в режим чтения по инициативе секционного РОУ.

Таким образом, происходит параллельная работа двух однопортовых блоков ПК_СП: заполнение (запись) ПК_СП_2 а режиме прямого доступа и чтение из ПК_СП_1. После окончания заполнения ПК_СП_2 котроллер прямого доступа готов перейти к заполнению блока ПК_СП_1, но только после того как в последнем завершится чтение последней ячейки. Теперь блоки ПК_СП_1 и ПК_СП_2 меняются ролями: в ПК_СП_1 происходит запись, а из ПК_СП_2 - чтение. Так происходит до тех пор, пока не произойдет запись в требуемый блок, по требуемому адресу (адрес последнего операнда). Т.е. при данной организации ПК_СП управляющий уровень должен настроить контроллер ПК_СП на последний используемый операнд: номер блока (ПК_СП_1 или ПК_СП_2), число итераций использования данного блока и последнюю используемую ячейку памяти в этом блоке.

Инициация чтения из ПК_СП производится как результат формирования Жонглером подполя Ot на R-операнде Вычислителя, если оно равно 1 (loadable). Инициация ПК_СП, один из способов организации суперскалярных вычислений (в Вычислитель могут поступать 3 операнда). При этом Вычислитель выполняет обычную операцию в соответствии со значением подполя [Oc] и одновременно на Вычислитель поступает содержимое ПК_СП, что может инициировать в последнем суперскалярные вычисления. Опустошение ПК_СП_1 и ПК_СП_2 происходит идентичным образом с записью. По мере считывания констант сбрасываются их биты готовности. Как только все биты готовности текущего блока будут сброшены, происходит переключение на следующий блок. И так пока все константы не будут считаны.

Использование двухблочной однопортовой ПК_СП позволяет одновременно считывать со стороны РОУ и загружать со стороны УУ новые константы из больших библиотек. Это позволило эффективно реализовать алгоритмы: вычисления Евклидового расстояния между текущим вектором произнесения и всеми векторами кодовой книги; алгоритма Витерби поиска подходящей модели слова из библиотеки.

2.4 Многократное исполнение капсул

Для решения проблемы реентерабельности капсул потребовалась существенная доработка контроллера БП и некоторых механизмов Распределителя:

1) В контроллер БП введены два индексных регистра IR0 и IR1, предназначенные для адресации операндов. Это позволило на логическом уровне разделить всю БП на два банка, адресуемых регистрами IR0 и IR1 соответственно.

2) На входе распределителя введены два буфера F0 и F1, предназначенные для хранения операндов, считанных из БП на текущем такте вычислений.

3) Введены три режима работы контроллера БП и обработки операндов в Распределителе.

- По умолчанию – существующий режим работы, который модифицирует только регистр IR0 на заданное значение приращения адреса. Считанные операнды помещаются в буферы F0 и F1 последовательно, образуя кольцо (при заполнении F0 следующий операнд помещается в F1 и так далее). Распределитель не делает разницы между операндами, считанными из F0 и F1.
- Селективный – режим, который четко определяет очередность попадания операндов в полугорсти. В разделе 1.3.4.1 было показано, что Распределитель управляет потоком операндов путем репликации горстей. Горсть состоит из двух полугорстей. В селективном режиме операнды, считанные по адресу IR0, помещаются в буфер F0, а по IR1 – в буфер F1. Распределитель, в свою очередь, при формировании первой полугорсти считывает данные из F0, а при формировании второй – из F1. Также Распределитель сохраняет признак принадлежности первой или второй полугорсти для хранящихся в своем буфере операндов.
- FFT – специализированный режим работы контроллера БП, предназначенный для реализации аппаратной поддержки алгоритма БПФ.

4) Многократное исполнение капсулы реализуется средствами самой капсулы за счет введения специальных операндов Acm: (конфигуратор многократного исполнения капсулы) и Abm: (контроллер итерации капсулы). Операнд Acm: устанавливает режим функционирования БП, загружает счетчик цикла, начальные значения IR0 и IR1, модули и алгоритмы приращения IR0 и IR1. Операнд Abm: фиксирует факт окончания очередной итерации, настраивает дополнительное смещение от головы капсулы.

5) Для организации обработки новых операндов, настройки и управления многократным исполнением в состав контроллера БП введено вспомогательное устройство управления (ВУУ). При считывании Acm: ВУУ переключает БВ в требуемый режим и загружает настроенные параметры. При считывании Abm: ВУУ декрементирует счетчик итераций и устанавливает значения IR0 и IR1 на основе настроечных параметров Acm:. По достижении счетчиком

итераций нулевого значения ВУУ переключает БВ в режим «по умолчанию» и устанавливает значения IR0 = IR1 = адрес (Abm:) + 1.

В таблице 2.2 приводится структура операндов Acm: и Abm: и описание их полей.

Таблица 2.2 – Структура Acm: и Abm: операндов

Поле	**Описание**	**Разряды**	**Размер (разрядов)**
Структура Acm: операнда			
%Ci	Количество итераций части капсулы (iterations)	7..0	8
%C0s	Начальное значение IR0	16..8	9
%C1s	Начальное значение IR1	25..17	9
%C0d	Приращение (delta) IR0 за итерацию	31..26	6
%C1d	Приращение (delta) IR1 за итерацию	37..32	6
%Cdm	Режим работы БП и Распределителя	39..38	2
%C0m	Модуль модификации значения адреса регистра IR0	43..40	4
%C1m	Модуль модификации значения адреса регистра IR1	47..44	4
Структура Abm: операнда			
%Csh		7..0	8

Модификация IR0 и IR1 при переходе на следующую итерацию осуществляется в соответствии со следующими формулами:

$$IR_0 = Csh + C0s + C0d * N_{\text{итер}} \quad (2.1)$$

$$IR_1 = Csh + C1s + C1d * N_{\text{итер}} \quad (2.2)$$

Модификация IR0 и IR1 в процессе итерации осуществляется в соответствии со следующими формулами:

$$IR_0 = IR_0 + C0m \quad (2.3)$$

$$IR_1 = IR_1 + C1m \quad (2.4)$$

Механизм многократного исполнения использовался для реализации большинства капсул задачи РИС, которые приводятся в главе 4.

2.5 Механизм косвенной репликации

Первый метод повышения гибкости механизма косвенной репликации заключается в разделении единого режима косвенной репликации горсти для разных полугорстей. Данное разделение позволяет использовать разные МКР для формирования разных полугорстей, а также разные алгоритмы их модификации. Это позволяет существенно повысить гибкость репликации операндов по секциям. Однако не решает проблему более равномерного распределения последовательно поступающих операндов по разным секциям в силу особенностей режимов репликации, представленных в таблице 1.2. Заключается она в том, что только режимы *h* и *h_f* обеспечивают рассылку более одного операнда за шаг.

Второй метод повышения гибкости механизма косвенной репликации заключается в введении для режимов g и a из таблицы 1.2 их горстевых аналогов, которые позволяют

рассылать более одного операнда за шаг репликации. В комбинации с первым методом достигается максимально возможная гибкость распределения операндов по секциям. В таблице 2.3 приводится описание режимов косвенной репликации. На рисунке А.8 в Приложении А приводится алгоритм режима горстевой гирлянды, а на рисунке А.9 в Приложении А – алгоритм режима горстевой лавины.

Таблица 2.3 – Управление косвенной репликацией

%Mf/Mn~подполя **(УКР)**		**Режим репликации при нулевой маске**	**Действие над МКР после шага репликации**
Символ	**Код**		
d	0000	Режим пропуска	МКР не меняется
g	0001	Полугорстевая гирлядна	Ротация влево (ст. в мл.)
a	0010	Полугорстевая лавина	Сдвиг влево (мл. в мл.)
h	0011	Горстевой сдвиг	МКР не изменяется (константа)
s	0100	Полугорстевая рассылка одиночного	МКР не изменяется (константа)
fa	0101	Горстевой режим	Сдвиг влево (мл. в мл.)
hg	0110	Горстевая гирлянда	Ротация влево (ст. в мл.)
ha	0111	Горстевая лавина	Сдвиг влево (мл. в мл.)

Режим fa является результатом переработки режима h_f и необходим для обеспечения совместимости после разделения на две полугорсти.

Третий метод повышения гибкости механизма косвенной репликации заключается в введении режима совместного использования прямой и косвенной репликации. Это позволяет совмещать упакованные и полноценные операнды без необходимости прерывать формирование горсти. Следовательно, повышается пропускная способность операндов распределителем и снижается степень дублирования операндов.

2.6 Обработка выходных данных

Для решения проблемы 8) из раздела 2.1 этого потребовалось:

1) Вместо формирования выходной капсулы из двух наборов выходных данных в капсулу добавляются выходные разделы данных, которые оформляются специальными синтаксическими конструкциями. Выходные разделы могут иметь два типа: истинный или промежуточный. Истинный выходной раздел накапливает реальные выходные данные, которые будут переданы УУ. Промежуточный набор используется для конфигурации алгоритма записи промежуточных E-операндов обратно в капсулу для последующего использования в многократном исполнении капсулы.

2) Полностью переработать формат Ai: операнда и семантику его полей:

In – Имя (для символьной капсулы) / адрес (для числовой капсулы) выходного набора данных (НД). Размер поля изменен, т.к. максимальный объем БП составляет 4096

операндов и она будет содержать порядка 8-12 капсул (в данном случае 10 разрядов предоставляют некоторый запас по максимальной длине капсулы).

Ii – Режим инициализации буфера выходного НД (поле остается).

Is – размер буфера выходного НД. Размер поля изменен, с учетом потребностей алгоритма БПФ – по 256 действительных и мнимых частей (то есть 512 в сумме, 10 разрядов предоставляют некоторый запас).

It – тип выходного НД:

It=o (0) по умолчанию, *output* – выходной НД.

It=t (1) *temporary* – промежуточный НД (для перезаписи в капсулу).

Im – режим функционирования импликатора при обработке промежуточного НД (только для *It=t*):

Im=d (0) по умолчанию, *delta* – режим приращения адреса с постоянным шагом.

Im=p (1) *preset* – предустановленный режим, в котором адреса записи промежуточных операндов в капсулу настроены заранее.

Ia – *address* – начальный адрес операнда для перезаписи в капсуле (только для *It=t, Im=d*). Размер поля в 10 разрядов выбран с учетом размера поля *In*.

Id – *delta* – модуль приращения адреса записи промежуточных данных в капсулу (только для *It=t, Im=d*). Размер поля в 10 разрядов выбран с учетом размера поля *In*. При необходимости его можно уменьшить (требуются дополнительные данные экспериментов).

If – *fields* – режим использования функциональных полей выходных данных (только для *It=t*).

If=n (0) по умолчанию, *no* – используются функциональные поля из шаблона капсулы

If=y (1) *yes* – используются функциональные поля из шаблона выходных данных (*At*-операнда)

3) Добавлена поддержка новых типов выходных разделов в At: операнд.

4) Определены структуры истинного и промежуточного выходных разделов, представленные в таблице 2.4.

В случае, если Импликатор настроен на режим *%Im=d*, то содержимое НД temporary игнорируется и перезапись в капсулу осуществляется в режиме приращения адреса.

В случае, если Импликатор настроен на режим *%Im=p,* то перезапись в капсулу осуществляется по предустановленным настройкам и адресам. В качестве адреса назначения используется значение поля *Adr*. Счетчик *Counter* используется в случае, если необходимо перезаписать данное в шаблоне капсулы, которое размещается сразу в нескольких операндах.

Таблица 2.4– Структуры выходных разделов

Истинный выходной раздел
%In=Имя_набора %It=o
V=…
V_38=…
V=…
…
Ad:;
Промежуточный выходной раздел
%In=Имя_набора %It=t %Im=d/p %Ia=начальный_адрес %Id=приращение %If=n/y
V=… Adr=… Counter=число
V_38=… Adr=… Counter=число
V=… Adr=… Counter=число
…
Ad:;

Каждый вычислительный шаг Импликатор осуществляет запись одного промежуточного операнда из своего буфера, счетчик которого не равен 0. После очередной записи в БП со стороны Имликатора осуществляется декрементация счетчика. В случае, если счетчик принимает нулевое значение, операнд считается обработанным, и Импликатор переходит к обработке следующего промежуточного операнда из своего буфера. Если такой операнд отсутствует, то Имликатор ожидает его прибытия.

Разрядность операндов в БП равна 64 битам, поэтому каждый операнд (кроме Apdi4) имеет поле Z=00000000. Это поле может быть эффективно использовано для хранения адреса для перезаписи «следующего» операнда, поступающего из Импликатора. Другими словами, при помощи операции чтение-модификация-запись, существует возможность считать значение поля Z текущего перезаписываемого операнда и, если его значение отлично от 0 и не выходит за границы шаблона текущей капсулы, то использовать его в качестве адреса для размещения следующего промежуточного операнда в буфере Импликатора.

Если подобный механизм используется в режиме *Im=d*, то приращение адреса не выполняется, вместо этого осуществляется прямая подстановка считанного адреса.

Если подобный механизм используется в режиме *Im=p*, то прямая подстановка осуществляется только для тех промежуточных данных, чей исходный *Counter*>1. Другими словами, если процесс многократной перезаписи одного и того же данного запущен, то должен быть доведен до конца; в случае же перехода к следующему промежуточному данному буфера Импликатора, подставляется определенный в его поле *Adr* адрес.

Дополнительно Apdi_x4 операнд можно использовать в промежуточном наборе данных:

• в режиме *Im=p* поле V0 специфицирует конкретный **адрес** записи Apdi_x4 операнда **целиком**;

• в режиме *Im=d* для записи упакованного Apdi_x4 операнда **целиком** по адресу, определяемому текущим вычисленным значением (в соответствии с механизмом Im=d).

5) Допускается размещение более одного выходного набора данных одинакового типа в шаблоне капсулы. Заполнение шаблонов одного и того же типа осуществляется последовательно в порядке их определения в капсуле и в соответствии с настройкой поля *%At*. В момент окончания заполнения одного шаблона осуществляется переключение Импликатора на заполнение следующего шаблона и так далее.

2.7 Аппаратная поддержка алгоритма БПФ

В области ЦОС существует две наиболее распространенные процедуры: цифровая фильтрация и дискретное преобразование Фурье (ДПФ). На сегодняшний день ДПФ используется практически во всех инженерных областях, а также для анализа и обработки сигналов в физике и технике [62].

Несмотря на всю мощь, предоставляемую Фурье анализом, сама математическая процедура ДПФ неэффективна. Это связано с необходимостью вычисления очень большого количества комплексных умножений, что делает прямое вычисление ДПФ нецелесообразным. В 1965 году Кули и Тюки в работе [63] предложили быстрый алгоритм вычисления ДПФ, который сегодня известен как БПФ. Применение БПФ сделало возможным проведение Фурье анализа с помощью цифровых сигнальных процессоров (ЦСП).

Дальнейшее развитие алгоритма БПФ привело к появлению целого семейства алгоритмов: Radix-2, Radix-2^2, Radix-4 и др. Среди указанных алгоритмов БПФ, наиболее производительными являются Radix-4 и Radix-2^2. Однако их реализация вносит существенное ограничение на поддерживаемое число отсчетов. Для покрытия отсчетов равных степени двойки в любом случае потребуется реализация Radix-2 алгоритма. Вариации алгоритма с прореживанием по времени и по частоте имеют одинаковую вычислительную сложности. Для реализации в ГАРОС был выбран алгоритм Radix-2 с прореживанием по времени, т.к. данная вариация считается эталонной в области ЦОС. Несмотря на значительно более высокую скорость вычисления алгоритм БПФ все еще требует значительных временных затрат, особенно при увеличении размерности алгоритма (более 512 отсчетов). Поэтому современный высокопроизводительный ЦСП должен предоставлять набор инструментов для эффективного вычисления БПФ для окон отсчетов различных размеров.

Существует два основных метода повышения эффективности и производительности вычисления БПФ: оптимизация программного обеспечения; введение в состав ЦСП модулей

аппаратной поддержки одного или нескольких алгоритмов из семейства БПФ. Комбинируя оба метода, можно добиться требуемого уровня баланса между производительностью, аппаратными затратами и энергоэффективностью.

Первый метод направлен на модификацию алгоритма БПФ и повышение эффективности использования параллелизма уровня инструкций для уже имеющихся аппаратных средств конкретного ЦСП. D работе [64] авторы предлагают эвристический алгоритм планирования задач, позволяющий вычислять БПФ на ЦСП TMSC66x до 50% быстрее, чем предлагаемая разработчиками библиотечная функция без использования аппаратных ускорителей БПФ.

Второй метод заключается в частичной или полной аппаратной реализации типовых вычислительных операций, называемых «бабочка», семейства алгоритмов БПФ. Данный метод позволяет ценой аппаратной избыточности очень быстро (теоретически вплоть до 1 цикла) вычислять «бабочки». На практике используют конвейеризацию для повышения производительности и снижения аппаратных затрат. В документе [65] приводится описание аппаратного блока Radix-2^2, позволяющего вычислять за 1 прогон сразу две стадии БПФ.

В состав ГАРОС на предшествующих этапах разработки были введены механизмы аппаратной поддержки алгоритма БПФ, кратко описанные в работах автора [66-69]. В процессе синтеза ПЛИС прототипа архитектуры оказалось, что накладные аппаратные расходы слишком велики. Поэтому необходимо было провести исследование существующих решений для эффективной реализации БПФ и разработать специализированное решение для ГАРОС. Основные результаты усовершенствования средств аппаратной поддержки БПФ и их испытаний на комплекте бенчмарков BDTI опубликованы автором в работах [70-72].

В книге [73] рассматривается продукция компании BDTI, как одного из лидеров бенчмаркинга ЦСП. В 1994 году BDTI представила свои тесты ЦСП, которые получили название «BDTI Benchmarks». BDTI Benchmarks включает 12 ядер алгоритмов, которые представляют основные операции ЦОС, используемые в общих приложениях ЦОС. Алгоритм БПФ также входит в качестве одного из ядер, которое измеряет скорость вычисления 256-Point-In-Place FFT (256-точечное БПФ с записью выходных данных на место входных). Данный алгоритм используется для анализа производительности разработанного решения.

2.7.1 Описание алгоритмов ДПФ и БПФ

Алгоритм БПФ – вычислительно эффективный алгоритм вычисления ДПФ, с количеством отсчетов равным натуральным числам во второй степени. ДПФ $X(k)$, где $k = 0, \dots, N-1$. ДПФ имеет вид, представленный в формуле (2.5).

$$X(k) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)\left[\cos\left(\frac{2\pi nk}{N}\right) - \mathrm{i}\sin\left(\frac{2\pi nk}{N}\right)\right], k = 0, \dots, N-1 \qquad (2.5)$$

Поворотные коэффициенты $W_N^{nk} = \left[\cos\left(\frac{2\pi nk}{N}\right) - \mathrm{i}\sin\left(\frac{2\pi nk}{N}\right)\right]$ размещены на единичной окружности в комплексной плоскости. Они симметричны и периодичны, что позволяет существенно сократить количество умножений, требуемых для вычисления ДПФ.

Алгоритм вычисления БПФ по основанию 2 (Radix-2), разделяет вычисление ДПФ на серию 2-точечных ДПФ. Каждое такое преобразование называется операцией «Бабочка». Для работы такого алгоритма, требуется, чтобы число отсчетов N являлось натуральной степенью двойки $N = 2^s, s \in \mathbb{N}$. Тогда для вычисления БПФ потребуется провести s каскадов. Результаты вычисления каждого каскада могут быть сохранены в тех же самых ячейках памяти, которые изначально хранили входные отсчеты.

БПФ с прореживанием по времени (Decimation-in-time, DIT) вычисляет каскады от самого мелкого до самого крупного, используя типовую операцию «бабочка», приведенную на рисунке 2.3 (рисунок 5.13 из [74]). Для корректной работы алгоритма, необходимо загружать входные отсчеты в бит-реверсированном порядке.

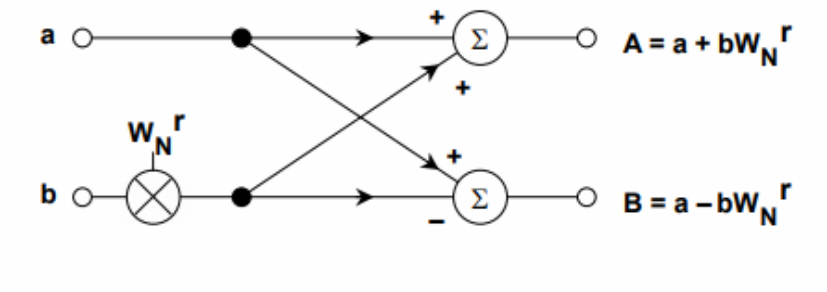


Рисунок 2.3 – Операция Radix-2 DIT «бабочка» (рис. 5.13 из [74])

Пример БПФ на 8 отсчетов с прореживанием по времени приведен на рисунке 2.4 (рисунок 5.15 из [74]).

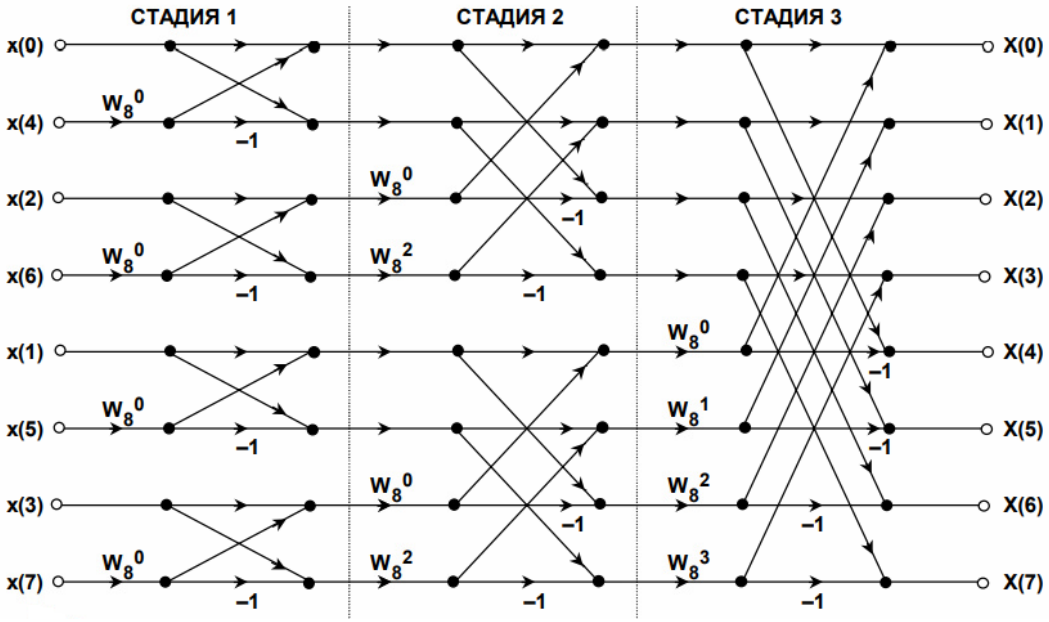


Рисунок 2.4 – Три стадии 8-точечного Radix-2 DIT (рис. 5.15 из [74])

2.7.2 Анализ существующих реализаций БПФ

2.7.2.1 Инструменты аппаратной поддержки

Базовая операция БПФ «бабочка» объединяет в себе значительный набор арифметических операций и операций доступа к памяти. Поэтому прямолинейная реализация данного алгоритма с использованием одних только стандартных вычислительных средств архитектуры общего назначения и средств компиляции дают неудовлетворительные результаты производительности для задач реального времени. Одним из путей повышения производительности является ввод в состав архитектуры специализированных аппаратных блоков поддержки операции «бабочка».

Аппаратные блоки БПФ

Одна из ведущих компаний в области микроэлектроники Xilinx предоставляет для синтеза в ПЛИС готовые IP-блоки БПФ-процессора, реализующие Radix-структуру аппаратным образом [75]. Представленные БПФ-процессоры обеспечивают компромисс между аппаратными затратами и скоростью вычисления преобразования. Различают четыре вида IP-блоков:

1) Pipelined, Streaming I/O

Конвейеризует несколько специальных аппаратных блоков вычисления Radix-2 «бабочек», позволяя организовать непрерывный вычислительный процесс. Каждый Radix-2 блок имеет свой банк памяти для хранения как входных, так и промежуточных данных, и одновременно вычисляет очередную «бабочку» и пересылает результаты в «соседний» блок. Такая организация обеспечивает наибольшую пропускную способность и производительность за счет максимального наполнения конвейера ценой наибольших аппаратных затрат.

2) Radix-4, Burst I/O

Данное решение включает в себя один аппаратный блок вычисления Radix-2^2 «бабочки». Для корректной работы требуется загружать и выгружать данные отдельно от непосредственных вычислений. Запуск вычислений производится после завершения загрузки всего фрейма обработки. А выгрузка осуществляется после завершения преобразования. Загрузку и выгрузку можно перекрыть, если выходные данные считываются в бит-реверсивном порядке. Такая организация ввода-вывода называется Burst. Для данного решения характерно значительно меньший объем аппаратных затрат за счет снижения скорости вычислений.

3) Radix-2, Burst I/O

Данное решение включает в себя один аппаратный блок вычисления Radix-2 «бабочки». Процесс загрузки и выгрузки данных идентичен блоку Radix-4, Burst I/O. Следует также отметить, что блок содержит два сумматора, работающих параллельно. Это позволяет вычислять преобразованные значения действительных и мнимых частей одновременно. Данное

решение требует еще меньше аппаратных затрат чем Radix-4, но и имеет меньшую производительность, т.к. Radix-2 более медленный алгоритм.

4) Radix-2 Lite, Burst I/O

Данный IP-блок по основным принципам работы аналогичен Radix-2, Burst I/O. Отличается же он тем, что содержит один сумматор, используемый для вычисления как действительных, так и мнимых частей. Это позволяет задействовать еще меньше аппаратуры, но и преобразование вычисляется еще медленнее.

Таким образом, конвейеризованное решение дает наилучшую производительность, но при максимальных аппаратных затратах. В качестве примера похожего аппаратного решения можно привести процессоры TMSC55x компании Texas Instruments. Согласно представленной документации [65] они содержат блок аппаратного ускорения HWAFFT, реализует Radix-2^2 структуру «бабочки» и подходит под классификацию Radix-4, Burst I/O.

Другим интересным аппаратным решением является высокоточный Radix-2 БПФ процессор, представленный в работе [76]. Данный процессор позволяет вычислять БПФ размерности до 1024 отсчетов в формате плавающей точки. Поэтому он предоставляет очень высокую точность. Однако операции с плавающей точкой выполняются значительно медленнее. Для того чтобы обеспечить требуемый уровень производительности авторы использовали IP-блок Radix-2, Burst I/O, расширенный дополнительными функциональными блоками. Также разработчики объединили концепции Pipilened, Streaming I/O и параллельное вычисление действительных и мнимых частей. На рисунке 2.5 (рисунок 11 из [76]) представлена архитектура данного БПФ-процессора.

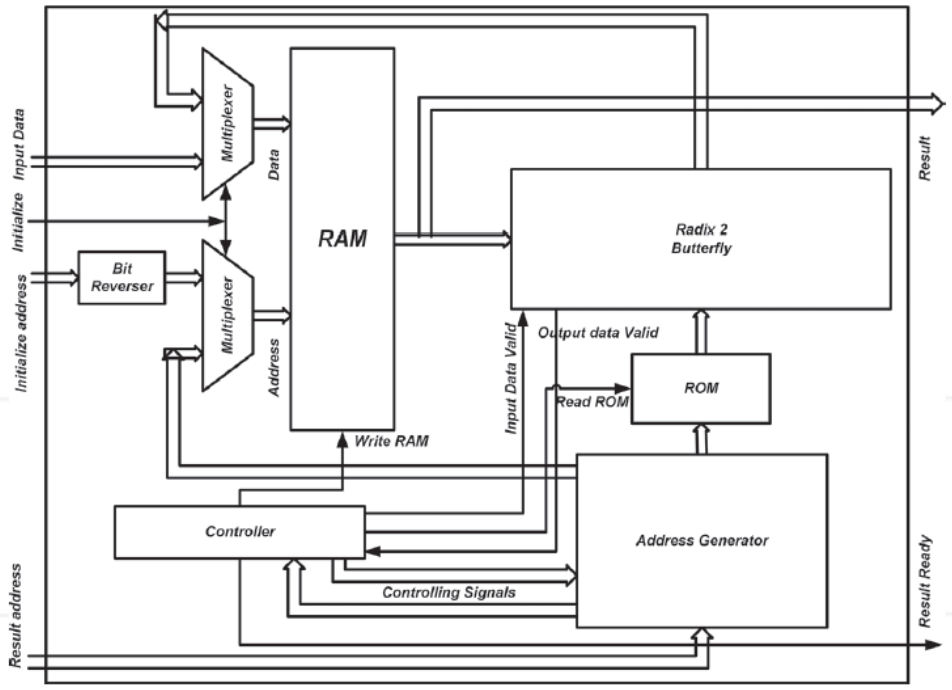


Рисунок 2.5 – Высокоточный БПФ-процессор (рис. 11 из [76])

2.7.2.2 Конвейеризация БПФ процессоров

Аппаратные БПФ-блоки являются достаточно громоздкими. Следовательно, попытка размещения всех требуемых элементов в рамках одного вычислительного цикла приведет к слишком низкой тактовой частоте и, таким образом, к потере производительности. Поэтому в современных БПФ-блоках используется конвейеризация вычислений для достижения максимальной производительности. Например, в TMSC55x блок HWAFFT вычисляет Radix-2^2 «бабочку» за 9 циклов, а Radix-2 «бабочку» за 5 циклов. Высокоточный БПФ-процессор в [76] вычисляет «бабочку» за 11 циклов, однако в режиме заполненного конвейера эффективно вычисляет «бабочку» за 1 цикл.

2.7.2.3 Программная оптимизация

Вторым способом повышения скорости вычисления БПФ является оптимизация алгоритма БПФ с целью повышения использования параллелизма уровня инструкций (Instruction Level Parallelism – ILP) для конкретной выбранной архитектуры. В работе [64] авторы приводят исследование эффективности эвристического алгоритма планирования нагрузки для ЦСП TMSC66x. Предлагаемая реализация не использует ресурсы аппаратного ускорителя БПФ, входящего в состав C66x. Компания Texas Instruments сопровождает C66x библиотекой алгоритмов ЦОС, оптимизированной для выполнения на указанном процессоре. Для исследования была выбрана библиотечная реализация Radix-2^2 алгоритма.

Авторы произвели анализ эффективности использования ILP в библиотечной функции и пришли к выводу, что для вычисления одной «бабочки» требуется 11 циклов. При этом, 100% времени выполняются операции чтения/записи, 59% времени – сложения/вычитания и 28% времени – умножения. Метрика ILP для C66x имеет значение равное 8 (до 8 инструкций может быть выполнено одновременно). Тогда C66x мог бы выполнить за 11 циклов 88 инструкций. Реально же библиотечная реализация исполняет за 11 циклов 54 инструкции. Коэффициент эффективности использования ILP для библиотечной реализации составил 61%.

Низкое значение коэффициента обусловлено значительным простаиванием ресурсов блоков умножения. Поэтому авторы предложили заменить чтение поворотных коэффициентов на их вычисление. Из формулы (4.1) можно вывести, что поворотные коэффициенты для «бабочек» с номерами n и $n-1$ связаны между собой линейными соотношениями и могут быть вычислены за одну операцию умножения. Это позволяет сбалансировать нагрузку на аппаратные блоки и добиться скорости вычисления «бабочки» за 8 циклов.

При этом 100% времени выполняются операции чтения/записи, 75% времени – умножения, 100% времени – сложения и вычитания. Таким образом, из 64 возможных инструкций такая реализация выполняет 60, а коэффициент эффективности использования ILP составляет 94%. Получение такого значительного требует дополнительного упорядочивания

последовательности вычисления «бабочек» (чтобы правильно вычислять поворотные коэффициенты). Решение этой задачи возлагается на предлагаемый эвристический алгоритм, который выполняется на этапе компиляции программы для C66x.

В таблице 2 работы [64] приводятся экспериментальные результаты применения предложенного метода, согласно которым ускорение составляет от 37% до 56% (в зависимости от размерности преобразования). Такое значительное ускорение **без использования** дополнительных аппаратных ресурсов, говорит об эффективности данного подхода.

2.7.2.4 Существующие средства поддержки БПФ в ГАРОС

Текущая версия прототипа ГАРОС содержит средства аппаратной поддержки вычисления БПФ, которые обеспечиваются вычисление 256-точечного Radix-2 DIT алгоритма в формате фиксированной точки с записью результатов на место входных данных (in-place реализация). Данные средства и соответствующие им механизмы архитектуры характеризуются следующими особенностями:

- используется механизм упаковки входных данных по 4 16-битных элемента, действительные части упакованы в одном разделе капсулы, мнимые – в другом;
- раздельная адресация действительных и мнимых упакованных данных контроллером компонента Буферная память ГАРОС;
- упакованные данные хранятся в бит-реверсивном порядке;
- контроллер БП реализует специальный алгоритм вычисления адресов упакованных данных, которые должны быть считаны/записаны на текущей стадии операции «бабочка»;
- поворотные коэффициенты хранятся в блоке ПК_С в количестве 128 действительных и 128 мнимых 16-битных констант;
- архитектура содержит 4 работающих параллельно секции, т.е. имеет 4 вычислительных блока и 4 копии ПК_С;
- контроллер ПК_С реализует специальный алгоритм вычисления адресов поворотных коэффициентов, которые должны быть считаны на текущей стадии операции «бабочка»;
- компонент Распределитель ГАРОС поддерживает специальный режим распаковки входных данных, который необходим для корректного формирования потока данных, рассылаемого по 4 секциям;
- решение использует архитектуру типа Radix-2 Lite, Burst I/O (только сам Radix-2 блок не реализуется аппаратным способом);
- в систему команд введена специальная конвейеризованная инструкция «Butterfly», которая обеспечивает четырех-стадийное вычисление Radix-2 DIT «бабочки».

Ключевым элементом аппаратной поддержки БПФ является инструкция «Butterfly». Эта инструкция на аппаратном уровне интерпретируется вычислительными блоками, как готовый четырех-стадийный сценарий функционирования. Для каждой стадии выполнения инструкции определена своя схема вычислений. Данный подход аналогичен эвристическому алгоритму, с той лишь разницей, что схема вычислений определена заранее, а не планируется компилятором или планировщиком задач.

На рисунке 2.6 представлена схема четырех-стадийной инструкции «Butterfly». Символом «*» отмечены отсчеты и константы, участвующие в вычислении *i-ой* «бабочки», а символом «**» - *(i+1)-ой* «бабочки». Индекс *j* соответствует входу *a* на рисунке 2,6, а индекс *k* – входу *b*. В режиме наполненного конвейера «бабочка» вычисляется за 4 цикла.

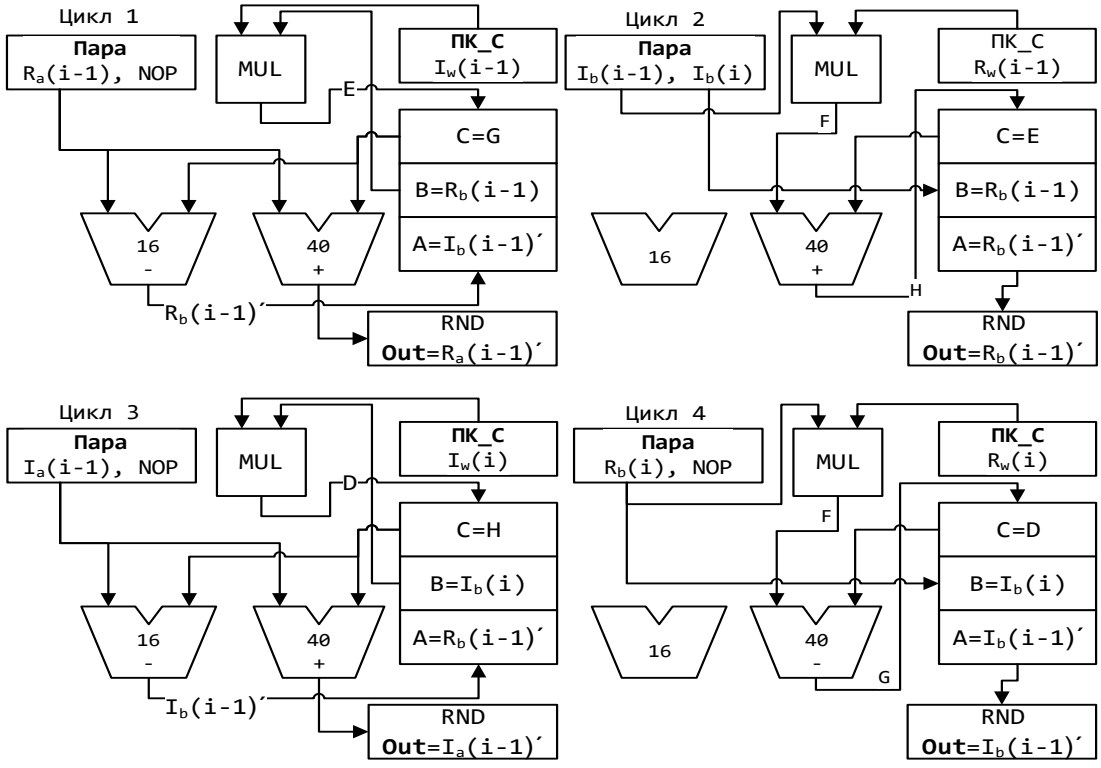


Рисунок 2.6 – Существующая схема инструкции «Butterfly»

Метрика ILP для вычислительных блоков ГАРОС имеет значение 4 (1 умножитель, 1 блок сдвига и округления, 1 АЛУ 16-битное, 1 АУ 40-битное). За 4 цикла вычислительный блок может максимально выполнить 16 инструкций. Данная инструкция обеспечивает выполнение 14 инструкций. Коэффициент эффективности использования ILP составляет 87,5%.

Проблемы существующих средств аппаратной поддержки БПФ в ГАРОС:

1) Избыточные накладные расходы хранения поворотных коэффициентов, связанные с особенностями разделенной по секциям ПК_С.

Фактически, вместо хранения 256 действительных и мнимых констант (для 256-точечного комплексного БПФ) ГАРОС хранит их в n-раз больше (где n – число параллельных

секций, в данном случае n=4). Для большего количества отсчетов нужно хранить больше констант, что приводит к еще большей избыточности.

2) Хранение отсчетов в капсуле в упакованных операндах.

Упаковка данных снижает избыточность тегированных данных, что является несомненным преимуществом. Но на последних двух стадиях БПФ контроллеру БП приходится считывать и записывать части упакованных данных (вместо целого операнда) по разным адресам. Это приводит к усложнению схемы памяти и алгоритма вычисления адресов, росту накладных расходов и потенциальному снижению частоты ее работы.

3) Нецикличный паттерн считывания действительных и мнимых частей отсчетов.

Из рисунка 2.6 видно, что входные отсчеты считываются в следующей последовательности (символы «*» и индексы заменены так, чтобы соответствовать рисунку):

Цикл 1: Ra(i)

Цикл 2: Ib(i), a Ib(i+1) уже должен быть в памяти операндов заранее!

Цикл 3: Ia(i)

Цикл 4: Rb(i+1)

Как видно, действительные и мнимые части считываются непоследовательно и даже из разных «бабочек» в ходе одного прогона инструкции. Это дополнительно осложняет алгоритм вычисления адресов в контроллере БП.

4) Цикл №2 инструкции «Butterfly» требует специального режима распаковки и рассылки упакованных данных.

В ГАРОС за распаковку и рассылку упакованных данных отвечает компонент Распределитель. Особенность цикла №2 заключается в том, что на вход вычислительного блока должны прийти две мнимые части *i* и *(i+1)-ой* «бабочек». Это поведение выбивается из основной схемы рассылки данных, что потребовало ввода специального режима, который используется в ГАРОС только в рамках одного алгоритма.

5) Для изменения размера окна отсчетов БПФ необходимо значительно перерабатывать капсулу.

Анализ проблем текущей версии средств аппаратной поддержки БПФ в целом свидетельствует о большой степени избыточности аппаратных издержек. Кроме того, масштабирование алгоритма или масштабирование архитектуры приведет к кратному росту издержек из-за особенности хранения поворотных коэффициентов. Исходя из результатов анализа можно выделить два основных направления модернизации архитектуры: модификация конвейеризированной инструкции «бабочка» и модификация памяти. При этом модификация памяти должна включать в себя как изменение принципа хранения поворотных коэффициентов и отсчетов, так и контроллера.

2.7.3 Модифицированные средства аппаратной поддержки БПФ

2.7.3.1 Усовершенствование конвейеризованной инструкции

В предыдущем подразделе был сделан вывод о необходимости переработки четырех-стадийной инструкции «Butterfly». При этом технически ее характеристики не должны ухудшиться с точки зрения времени исполнения. Кроме того, переработанная схема инструкции должна также решать проблему 3), обозначенную в разделе 2.7.2.4. На рисунке 2.7 представлены результаты переработки инструкции «бабочка».

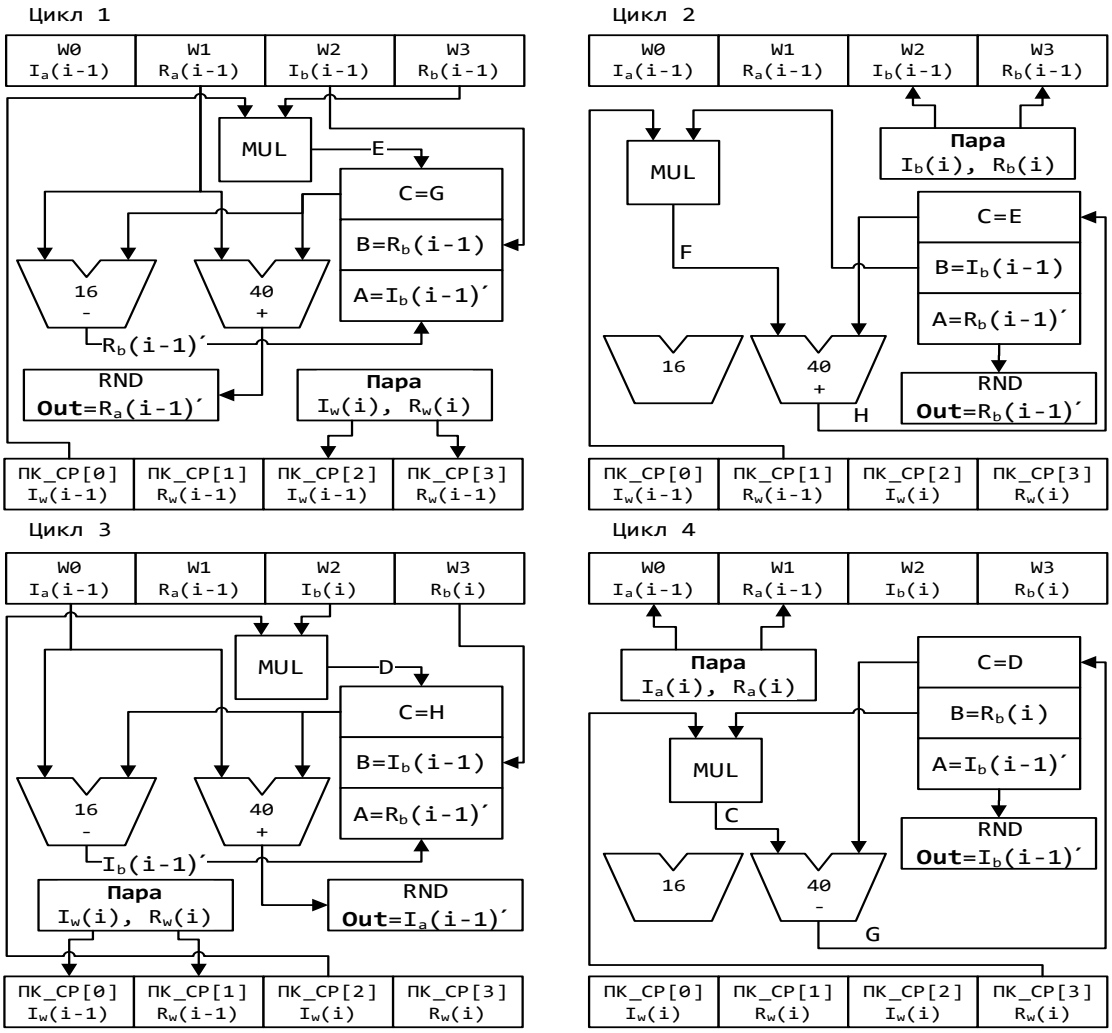


Рисунок 2.7 – Усовершенствованная схема инструкции «Butterfly»

В новой версии схемы инструкции поочередно считываются константы и данные. Причем для цикла 1 и 3 считываются одни и те же константы. А данные считываются только для i-ой бабочки (в отличие от предыдущей версии). Таким образом, проблема 3) и, следовательно, 4) решаются при внедрении данного решения.

С точки зрения использования ILP предлагаемое решение также обеспечивает выполнение 14 инструкций из 16 за 4 цикла и имеет коэффициент эффективности использования ILP равный 87,5%.

2.7.3.2 Доработка вычислительных блоков

Для обеспечения возможности исполнения новой схемы инструкции «Butterfly» в состав вычислительных блоков ГАРОС необходимо ввести дополнительные входные пользовательские регистры [W2] и [W3]. Таким образом, регистровый файл будет расширен до 4 регистров. Для корректного выполнения БПФ исходной инициализации регистрового файла **не требуется**. Также в состав вычислительного блока входит ПК_СР имеющая объем в 4 16-битных константы. В новой реализации необходимо обеспечить возможность записи сразу двух констант (ранее можно было записать только одну).

2.7.3.3 Модификация Распределителя

Так как проблемы 3) и 4) решаются за счет изменения схемы инструкции, то из Распределителя удалены специализированные режимы распаковки и рассылки операндов. Кроме того, особенности функционирования модифицированного контроллера БП требуют, чтобы Распределитель на завершающей стадии вычисления алгоритма БПФ не замораживал конвейер, а генерировал пустые горсти операндов (что равносильно NOP операциям).

2.7.3.4 Модификация Буферной памяти

Самым существенным изменениям подверглась БП и ее контроллер. В состав БП необходимо ввести новый автономный банк памяти и блок управления этой памятью, который включает в себя два работающих параллельно подблока: генератор адресов и контроллер чтения-записи. Данное решение было принято по аналогии с архитектурой БПФ процессора на рисунке 2.5 (дополнительный блок Address Generator и дополнительный Controller).

1) Блок памяти состоит из блоков: IBlock, RBlock, - хранят мнимые и действительные части отсчетов преобразуемого сигнала соответственно; IwBlock, RwBlock – хранят мнимые и действительные части поворотных коэффициентов W соответственно. Входные последовательности отсчетов должны быть записаны в бит-реверсированном порядке.

Отсчеты имеют формат Q15, поэтому хранятся в 16-битных словах. Объем IBlock равен объему RBlock и составляет 1024 16-битных слова. Данные в IBlock и RBlock записываются со стороны УУ. Поэтому в VHDL-реализации реализован интерфейс доступа для записи и чтения данных блоков. Режим записи отсчетов: как пакетный, так и одиночный.

Поворотные коэффициенты имеют формат Q15, поэтому хранятся в 16-битных словах. Объем IwBlock равен объему RwBlock и составляет 512 16-битных слова. Эти коэффициенты являются константами, рассчитываются заранее и записываются на этапе прошивки ПЛИС.

2) Контроллер памяти для режима FFT.

Данный контроллер обеспечивает параллельное функционирование блоков вычисления адресов и управления чтения и записи. Также он принимает данные от компонента Импликатор

ГАРОС и передает их для записи в блок памяти. Инициализация блока осуществляется в момент считывания основным контроллером БП Acm: операнда, который имеет %Cdm=fft (то есть перенастраивает режим памяти). После чего требуется подтверждение от операционного уровня архитектуры о том, что Acm: операнд получен и все нужные горсти сформированы, которое и инициирует выдачу операндов. По факту получения данного подтверждения начинается основной цикл функционирования.

При этом размер Radix2 DIT БПФ определяется по значению поля количества итераций %Ci. Число стадий БПФ напрямую связано с количеством отсчетов по формуле «количество стадий» = $Log_2(N)$. Тогда, например, для %Ci=8 N=256. Предлагаемая реализация БПФ контроллера в модели поддерживает настраиваемое количество секций исполнения алгоритма. Однако пока алгоритм рассчитан на 4 секции.

3) Блок вычисления адресов отсчетов и констант для «бабочек».

Данный блок должен за 4 цикла вычислить адреса действительных и мнимых частей отсчетов, а также требуемых констант для текущих «бабочек». Особенностью алгоритма является то, что IBlock и RBlock адресуются одним и тем же адресом. Таким образом, необходимо вычислить 4 адреса констант и 4 адреса «бабочек».

Особенностью схемы вычисления инструкции «Butterfly» является наличие 4 дополнительных циклов для запуска процесса. Поэтому данный блок должен работать «на шаг вперед» относительно блока чтения-записи, чтобы второй был всегда обеспечен новыми адресами. Также адреса и параметры самых первых «бабочек» будут всегда идентичными вне зависимости от размера БПФ. Поэтому первая группа адресов вычисляется в процессе инициализации моделей и аппаратуры.

Наконец, для данного блока требуется особое поведение в завершающей стадии алгоритма, когда адреса для всех «бабочек» уже рассчитаны, но из-за отставания блока чтения-записи на 1 «бабочку» и конвейера блок должен продолжать работу вплоть до записи последнего выходного данного на последней стадии алгоритма.

4) Блок управления чтением-записью

Данный блок обеспечивает считывание по присланным адресам констант и отсчетов, требуемых для выполнения каждой конкретной стадии инструкции «бабочка». Кроме того, данный блок также осуществляет запись выходных операндов, полученных от Импликатора. Однако, в этом процессе есть одна особенность. Запись выходных данных происходит по адресам «предыдущей бабочки». Это означает, что необходимо хранить эти адреса.

Ранее было отмечено, что выходные данные от Импликатора приходят с задержкой. То есть данный блок должен рассчитывать реальные адреса записи выходных отсчетов наперед и хранить их в FIFO, чтобы сохранить корректность in-place реализации. Это также влияет на

условие окончания работы данного блока. Здесь необходимо учитывать два момента: наличие 4 дополнительных шагов для запуска всего процесса; а также необходимость дождаться записи всех выходных отсчетов (т.е. дождаться опустошения FIFO).

Из рисунка 2.7 видно, что последовательность поступления выходных отсчетов для одной секции, следующая: Re, Re, Im, Im. Следовательно, каждые 8 записей необходимо переключать блоки RBlock и IBlock соответственно. Результатом работы блока управления чтением-записью на каждом шаге является два упакованных Apdi_x4: операнда, содержащие либо константы для 4-х секций, либо отсчеты для 4-х секций. Таким образом, константы поступают в РОУ стандартным механизмом формирования горстей. В стадии завершения всего алгоритма и ожидания последних выходных отсчетов блок не формирует новых операндов.

На рисунке 2.8 представлена разработанная архитектура средств поддержки БПФ.

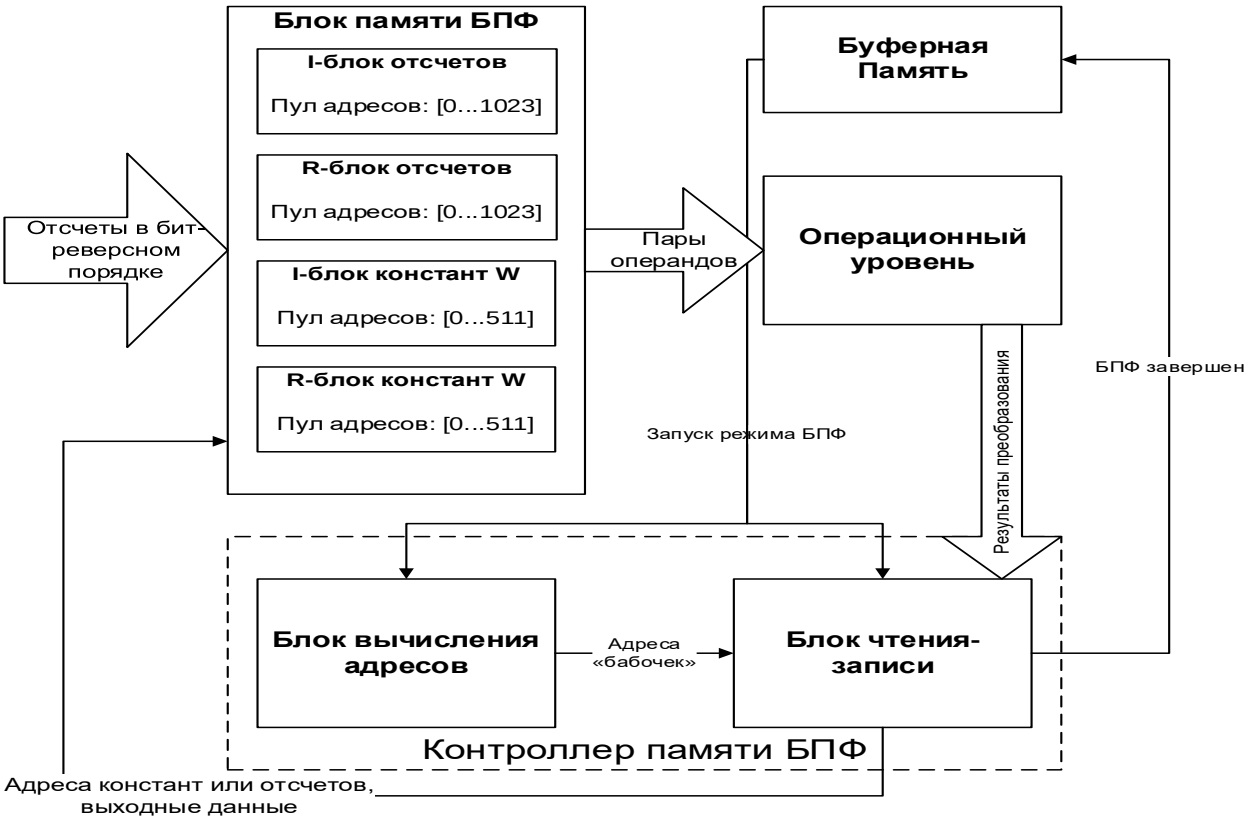


Рисунок 2.8 – Архитектура средств поддержки БПФ

2.7.3.5 Результаты VHDL-реализации

В результате синтеза на ПЛИС нового проекта ГАРОС с предложенными архитектурными решениями были получены следующие результаты:

- Существенно снизилось требуемое число ALM.
- Незначительно уменьшилось количество ALUTs.
- Очень существенно сократился объем используемой регистровой памяти за счет сокращения объема секционной памяти констант

• Взамен, выросли затраты блоков памяти, необходимые для реализации новой буферной памяти, в состав которой также входит блок констант FFT.

Объем блоков памяти существенно менее критичен для дальнейшей реализации архитектуры на кристалле в сравнении с регистровой памятью, поскольку для них могут использоваться внешние микросхемы памяти.

Данные результаты представлены в таблице 2.5.

Таблица 2.5 – Результаты синтеза

Имя узла в иерархии компиляции		**ALMs**	**ALUTs**	**Registers (bits)**	**Memory (bits)**	**M10Ks Blocks**	**DSP Blocks**	**Описание**
ros:rou	Initial	72765,2	84524	43996	147148	52	4	РОУ целиком
	Updated	61917,2	79973	28298	229068	62	4	
buffer_storage:c_BM		18308,9	24454	10989	141900	32	0	Буферная память
		15426,9	21568	11131	223820	42	0	
rou:c_rou		54396	60061	32887	5248	20	4	ROU без учета БП
		46431,2	58394	17042	5248	20	4	
compdev:c_compdev_0		4423,5	5671	576	0	0	1	Одно ядро Вычислител
		5203,4	6957	614	0	0	1	
distributor:c_distributor		16808,3	21669	4481	0	0	0	Распределитель
		16431,4	21483	4481	0	0	0	
Относительно предыдущей версии		-15%	-5%	-36%	+56%	+19%	0%	Увеличение/снижение издержек

2.7.3.6 Результаты испытания программной и аппаратной моделей

В процессе реальных испытаний тестовые запуски реальной капсулы осуществлялись на 4 секциях ГАРОС как на предыдущей версии средств аппаратной поддержки, так и на модифицированной. В таблице 2.6 приводятся сводные сравнительные результаты по накладным расходам и скорости вычисления для двух версий. Данные приведены для N=256 точек.

Таблица 2.6– Сравнительные результаты испытаний

Параметр	**Старая версия**	**Новая версия**
Объем памяти для хранения констант (16-битных слов)	256*4 = 1024	1024
Поддерживаемые размеры окон БПФ (точек)	256	от 8 до 1024
Размер капсулы (операндов)	132 для отсчетов + 23	23
Объем памяти для отсчетов (16-битных слов)	528	512
Benchmark (циклов)	$4 * N/2 * \log_2 N / 4 = 1024$	$4 * N/2 * \log_2 N / 4 = 1024$
Overhead (циклов)	34	16
Итого (циклов)	1078	1040

Полученные результаты испытаний позволили сделать вывод об успешной оптимизации и модификации средств аппаратной поддержки БПФ в ГАРОС. Из таблиц 2.5 и 2.6 видно, что новая версия средств не только обладает большей гибкостью и производительность., но и

позволила сократить накладные расходы капсулы. Кроме того, полученная реализация средств поддержки позволила решить все перечисленные в разделе 4.7.2.4 проблемы.

Проблема 1) была решена путем размещения поворотных коэффициентов в отдельном блоке памяти, причем его чтение осуществляется по стандартному в ГАРОС механизму за счет формирования на выходе пары упакованных операндов. Более того, в том же самом объеме памяти было размещено гораздо больше различных значений поворотных коэффициентов, за счет чего повысилась гибкость и масштабируемость алгоритма.

Проблема 2) была решена также путем размещения отсчетов в отдельном блоке памяти и формировании пары упакованных операндов на выходе при его чтении. Более того, это автоматически решило проблему 5), т.к. теперь модификация капсулы – это всего лишь переконфигурация размера вычисляемого БПФ.

Проблемы 3) и 4) были решены за счет существенной переработки инструкции «Butterfly», как было отмечено в разделе 4.7.3.1.

В различных областях ЦОС требуется вычислять не только БПФ с прореживанием по времени, но и по частоте, а также обратный БПФ. Кроме того, алгоритм по основанию 2 является менее быстрым, чем по основанию 4. В связи с чем, необходимо обеспечить возможности эффективной реализации в будущем и других алгоритмов из семейства БПФ.

Другим направлением работ в данном направлении должна стать апробация IP-блоков БПФ для эффективной аппаратной реализации «бабочек» и еще большего повышения производительности ГАРОС на этом классе задач.

2.8 Выводы к главе 2

В рамках работ по разработке и развитию прототипа ГАРОС был выявлен ряд проблем его существующей версии, рассмотренных в разделе 2.1. Для решения этих проблем автором диссертации был разработан набор методов, алгоритмов и механизмов.

Решение проблемы 1) было достигнуто за счет введения новых 40-разрядных регистров в состав Вычислителя и инструментов доступа к ним. В тоже время проблемы 1), 2) и 7) тесно связаны между собой. Поэтому решение проблем 2) и 7) путем развития режимов косвенной репликации усилило эффект от увеличения мощности регистрового файла Вычислителя.

Проблема 3) тесно связана с проблемами 1) и 4). Результаты решения данной проблемы представлены в таблице А.1 в виде обновленной системы команд, которая позволила не только эффективно управлять новым регистровым файлом, но и обеспечила широкие возможности организации суперскалярных вычислений.

Проблемы 6) и 8) также тесно связаны между собой. Возможности многократного исполнения одной и той же капсулы были бы существенно ниже без адекватного механизма переиспользования в рамках итераций промежуточных выходных данных. Решение проблем 6)

и 8) позволило снизить до необходимого минимума взаимодействия между УУ и РОУ, т.к. именно эта дисциплина является наиболее медленной по времени исполнения (в тактах). Предложенный механизм обработки выходных данных в некотором смысле нарушает один из принципов построения структур данных в потоковой модели вычислений, рассмотренный в разделе 2.1.3, а именно – неизменяемость исходной структуры. Поэтому в последующих исследованиях данный механизм будет доработан с целью устранить этот недостаток.

Одним из наиболее значимых научно-практических результатов данной работы является решение проблемы 4). Автором предложен набор методов организации и алгоритмов конфигурирования суперскалярных схем вычислений. Решение проблемы 5) путем ввода дополнительного типа памяти констант также позволило повысить гибкость конфигурации схем суперскалярных вычислений. Эксплуатация параллелизма уровня инструкций является одним из наиболее эффективных подходов повышения производительности вычислительных ядер в целом. Новая версия вычислительного ядра является 4-х задачной в режиме БПФ и 2-х задачной во всех остальных случаях.

Следующим наиболее значимым научно-практическим результатом является разработка новой версии средств аппаратной поддержки БПФ. Предложенные решения не только позволили снизить аппаратные затраты синтеза прототипа, но и повысили скорость вычисления point-in-place БПФ по основанию 2 с прореживанием по времени. В таблице 1.4 приводится сравнение ГАРОС и Мультиклета по скорости вычисления 256-точеченого преобразования. Мультиклету требуется 1192 такта, старой версии ГАРОС – 1078, а новой – 1040. В тоже время остается потенциал развития функциональных возможностей разработанных средств с целью поддержки наиболее быстрых алгоритмов Radix-4 и Radix-2^2.

Тем не менее, не решенными (в полном объеме) остались задачи реализации механизмов эксплуатации параллелизма, рассмотренные в разделе 1.3.1: внеочередного исполнения команд (только частично) и переименования регистров. Решение этих задач позволит упростить процесс программирования и повысить производительность прототипа.

Глава 3 Разработка отдельных элементов методологии программирования и отладки ГАРОС

Данная глава посвящена основным результатам отдельных элементов методологии программирования и отладки ГАРОС. Организация итеративного процесса разработки прототипа ГАРОС невозможна без использования специализированных средств моделирования и отладки как элементов архитектуры, так и капсул. Приводится доработанная модель языка капсульного программирования и обобщенная методика капсульного программирования. Рассматриваются основные программно-аппаратные средства разработки и отладки как прототипа архитектуры, так и программного обеспечения.

3.1 Развитие модели программирования ГАРОС

Ранее в разделе 1.3.3 приводилось описание модели программирования МПРА, а также краткое описание формата операндов (в таблице 1.1). В процессе развития архитектуры и ее функциональных возможностей потребовалась существенная переработка формата операндов. Связано это с несколькими обстоятельствами:

1) Шина взаимодействия процессора УУ NIOS II и РОУ имеет разрядность равную 32. Поэтому все операнды были выровнены из 56-разрядной в 64-разрядную сетку.

2) Увеличение длины операндов до 64-разрядов сделало возможным упаковать 4 16-разрядных операнда.

3) При реализации упаковки 4-х операндов все разряды операнда оказываются заняты данными. В результате становится невозможно однозначно идентифицировать тип операнда. Поэтому помимо бита-готовности был введен бит-признак упаковки, значение '1' которого означает, что операнд несет в себе 4 содержательных части.

4) Режима многократного исполнения капсул потребовал изменения механизма сброса битов готовности. Контроллер БП устанавливает биты готовности по факту записи операндов и сбрасывает – по факту чтения. При многократном исполнении большинство операндов не требует сброса битов готовности. Поэтому в дополнение к битам готовности и упаковки был введен бит многократного использования. Кроме того, в разделе 4.4 приводится описание двух новых типов операндов, которые необходимы для программирования этого режима.

5) Доработка механизмов обработки выходных данных потребовало серьезной переработки Ai: операнда. Также был добавлен At_38: операнд шаблон обработки 38-разрядных выходных данных и Do: операнд, предназначенный для описания выходных разделов капсулы.

6) Появление большого разнообразия памятей констант потребовало переработки Ccl: операнда – конфигуратора памяти констант. Добавлены операнды Ccl_38f: и Ccl_38d: для

конфигурации 38-разрядных констант. Добавлен операнд Asc: для настройки режима упаковки констант, подлежащих для загрузки в ПК_СР.

7) Удалены операнды Afi:, Apdi_6x8, т.к. необходимость в них отпала.

8) Форматы более половины полей были переработаны, изменена их разрядность и семантика в связи с развитием функциональных возможностей архитектуры.

В результате было специфицировано 35 операндов, общее описании которых представлено в таблице А.2 Приложения А.

3.2 Элементы методологии программирования ГАРОС

Методология – это учение о методах, методиках, способах и средствах познания.

Методологию можно рассматривать в двух срезах: как теоретическую (формируется разделом философского знания гносеология), так и практическую (ориентированную на решение практических проблем и целенаправленное преобразование мира). Теоретическая методология описывает модели идеального знания. Практическая – это программа (алгоритм), набор приёмов и способов того, как достичь желаемой практической цели и не погрешить против истины, или того, что считается истинным знанием. Качество (успешность, эффективность) метода проверяется практикой, решением научно-практических задач – то есть поиском принципов достижения цели, реализуемых в комплексе реальных дел и обстоятельств.

В методологии можно выделить следующую структуру:

- основания методологии: философия, логика, системология, психология, информатика, системный анализ, науковедение, этика, эстетика;
- характеристики деятельности: особенности, принципы, условия, нормы деятельности;
- логическая структура деятельности: субъект, объект, предмет, формы, средства, методы, результат деятельности, решение задач;
- временная структура деятельности: фазы, стадии, этапы;
- технология выполнения работ и решения задач: средства, методы, способы, приемы.

Методология также делится на содержательную и формальную. Содержательная методология включает изучение законов, теорий, структуры научного знания, критериев научности и системы используемых методов исследования. Формальная методология связана с анализом методов исследования с точки зрения логической структуры и формализованных подходов к построению теоретического знания, его истинности и аргументированности.

Методология программирования – это система принципов, совокупность идей, понятий, методов, способов и средств, определяющих стиль написания, отладки и сопровождения программ, организации теоретической, практической и производственной деятельности, а также учение о системах знаний, понятий и предметной и производственной деятельности.

Для решения проблем программируемости, рассмотренных в разделе 1.3, были разработаны отдельные элементы *рекуррентно-потоковой методологии программирования.*

Основаниями данной методологии являются *информатика* и *системный анализ*. Информатика включает в себя такой аспект, как разработка языков программирования. Системный анализ используется для определения параметров отдельных компонент архитектуры, на основе которых разрабатывается специальное программное обеспечение, предназначенное для моделирования, проведения экспериментов и отладки.

Принцип рекуррентно-потоковой методологии программирования заключается в представлении исходной задачи в виде совокупности подпрограмм специального типа, называемых *капсулами*. Каждая капсула реализует один или несколько алгоритмов решаемой задачи и представляет собой набор самодостаточных данных. Данные такого типа содержат также и всю необходимую управляющую информацию для их обработки. Такое представление программ называется *капсульным стилем программирования.*

Особенности капсульного программирования подробно рассмотрены в разделе 1.3 настоящей работы. Главной особенностью является представление программы в виде самопорождающейся последовательности динамических графов. Этот процесс порождения называется рекуррентной разверткой. Другая важная особенность состоит в том, что программы, написанные в капсульном стиле, также являются и потоковыми.

Условием начала вычислительного процесса, помимо управляющего сигнала «пуск», является событие появления входных данных в символьных капсулах. Для капсул переопределено «правило срабатывания» – условия запуска потоковой программы.

Субъектом деятельности является *программист* – разработчик ПО для ГАРОС.

Объектом деятельности является *капсула*.

Предметом деятельности является процесс разработки капсул.

В*ременная структура деятельности* реализуется в виде обобщенной методики программирования в среде ГАРОС.

Результаты разработки элементов методологии были опубликованы автором в отчете о НИР, шифр «КАПСУЛА2» [77], и в статье [78].

Обобщенная методика программирования ГАРОС

Этап I

Провести анализ задачи и определить, имеется ли необходимость производить сложные и ресурсоемкие вычисления, и какую долю (приблизительно) в решении задачи они составляют.

Этап II

Если имеют место сложные вычислительные задачи, и их доля достаточно велика, то декомпозировать исходную задачу на ряд подзадач по принципу:

Класс 1) - множество подзадач управления вычислительным процессом;

Класс 2) - множество последовательных подзадач или подзадач низкой вычислительной сложности;

Класс 3) - множество распараллеливаемых подзадач ил подзадач высокой вычислительной сложности.

Классы задач 1) и 2) нужно решать при помощи верхнего УУ архитектуры, класс 3) – при помощи нижнего уровня РОУ.

В качестве основных критериев оценки сложности задачи (подзадачи) приняты:

1) большой объем входного потока данных (например, цифровые фильтры, свертки);

2) скрытый или явный параллелизм вычислительного графа задачи степени 2 или более;

3) малый объем последовательных вычислений;

4) большой объем потока промежуточных данных;

5) большой объем выходного потока данных;

6) высокий порядок сложности (не ниже n*log(n)).

Этап III

Осуществить реализацию всех подзадач (как для УУ, так и для РОУ). Результатом этого этапа будет управляющая программа, исполняемая средствами УУ, и набор специализированных программ, называемых капсулами, решающих подзадачи класса 3. Управляющая программа осуществляет подготовку данных для капсул, обработку результатов их вычисления, а также решает подзадачи классов 1 и 2.

Этап IV

Произвести комплексную отладку набора капсул и управляющей программы. Если уровень производительности полученной программы оказался ниже требуемого, вернутся к Этапу II, либо установить невозможность дальнейшей оптимизации капсул.

Таким образом, данная обобщенная методология носит итеративный характер. Ключевыми этапами данной методологии являются этапы II и III: первый определяет результирующую декомпозицию задачи, а второй содержит в себе квинтэссенцию парадигмы капсульного программирования и называется методикой капсульного программирования.

Реализация и внедрение предлагаемой обобщенной методики подразумевает также и разработку специализированных программных средств, которые обеспечат частичную или полную автоматизацию всех этапов. В рамках текущей работы автором были разработаны различные программно-аппаратные средства, которые позволили частично автоматизировать итеративный процесс разработки как прототипа архитектуры, так и программного обеспечения. Данные средства были объединены в программный комплекс ПК ПОТОК.

Подэтап III-1. Определение функциональной нагрузки на капсулу

1. Назначить капсуле функциональность *задача.*

2. Анализировать математическую постановку и модель (если имеется) задачи, решаемой при помощи МПРА, и определить *глобальную степень параллельности* (ГСП) алгоритма. Наиболее подходящие «кандидаты» для выявления глобального параллелизма -циклы верхнего уровня, тела которых либо не зависят по данным, либо сцепливаются по данным (выходные данные одного тела являются входными для второго).

3. Блоки алгоритма, которые могут быть выполнены параллельно, обозначить *тело.*

4. Провести анализ математической сложности тел (воспользоваться теорией сложности алгоритмов).

5. Провести анализ тел на потенциальный параллелизм и определить их СП. Для этого применить существующие методы распараллеливания алгоритмов (например, каскадные схемы с различными способами редукции, эвристические алгоритмы).

6. Если ГСП >> 4, или сложность тел велика, или СП тел велика *(понятие 'велика' необходимо уточнять),* то декомпозировать капсулу на последовательность капсул, каждой из которых назначить функциональность *тела*. Иначе подэтап III-1 завершен.

7. Перейти к анализу капсул в последовательности:

   7.1. обозначить ГСП = СП текущей анализируемой капсулы;

   7.2. обозначить *алгоритм* – задача, решаемая текущей капсулой;

   7.3. вернуться к шагу 3.

8. Корректировать результаты подэтапа III-1 в соответствии с результатами подэтапов III-2 и III-3, т.е. вернуться к шагу 1).

Результат применения подэтапа III-1 методики:

– упорядоченная последовательность капсул, решающая поставленную задачу;

– каждая капсула решает часть общей задачи;

– СП каждой капсулы сравнима с 4.

Подэтап III-2. Разработка капсул

1. Инициализация всех капсул статусом *не завершена*

2. Выбрать очередную незавершенную капсулу; если таковых нет, перейти к подэтапу III-3.

3. Построить потоковый вычислительный граф алгоритма в соответствии с его математическим описанием (или моделью, если имеется).

4. Провести анализ графа с целью выделения повторяющихся и потенциально параллельно исполняемых блоков. Этот анализ может быть осуществлен вручную или с

применением программных средств для автоматического распараллеливания алгоритмов (например – компилятор языка программирования Sisal).

5. Применить существующие методологии распараллеливания алгоритмов (каскадные схемы, методы распараллеливания последовательных алгоритмов, эвристические алгоритмы из раздела 2.1.5 и др.) и преобразовать граф таким образом, чтобы его СП <= 4.

6. Осуществить поиск циклически повторяющегося фрагмента графа или цепочку вершин графа, которая может быть свернута рекуррентным образом, и выполнить рекуррентную свертку этого фрагмента / цепочки в вершины(-у) графа. Полученный граф называется *динамическим*.

7. Преобразовать динамический граф в соответствии со спецификацией функциональных возможностей РОУ. Результатом этого преобразования должна быть развернутая (детализированная) граф-капсула.

8. Если полученный граф неэффективен (имеет низкий коэффициент ускорения относительно последовательной реализации) с точки зрения временных затрат на конфигурацию, пересылку промежуточных данных и т.п., вернуться к шагу 6. В случае, когда найти приемлемое эффективное решение не удается, перейти к подэтапу III-1 шаг 8 и декомпозировать капсулу.

9. Преобразовать граф-капсулу в символьную капсулу.

10. Установить статус капсулы в *отлаживается* и перейти к шагу 2.

Результат подэтапа III-2 методики – одна или более капсул, готовых к тестированию.

Подэтап III-3 Отладка капсул

1. Отладить очередную капсулу со статусом *отлаживается;* если таковых нет, разработка завершена.

2. Анализировать результаты исполнения капсулы на соответствие требованиям точности и времени исполнения.

3. Если требования не удовлетворены, то статус капсулы – *не завершена;* вернуться к подэтапу III-2, иначе перейти к шагу 4.

4. Установить статус капсулы в *готова* и вернуться к шагу 1).

Итеративное применение методики капсульного программирования позволяет получить множество оптимизированных и отлаженных капсул, каждая из которых инкапсулирует отдельный алгоритм решаемой задачи.

Последний элемент в структуре методологии - *технология выполнения работ*.

В процессе выбора модели жизненного цикла разработки ПО для ГАРОС ставились требования его гибкости и расширяемости. Вопросы, связанные с анализом пожеланий заказчика, разработкой требований и т.п., были не столь критичными ввиду того, что в рамках

данного проекта авторы являются и заказчиками. В результате выбор пал на спиральную модель разработки - специально проработанный вариант итерационной модели. В качестве базовой методологии разработки ПО выбрана модель экстремального программирования.

Реализация предложенной методики разработки ПО в среде ГАРОС представлена далее в виде обобщенной технологии программирования.

Обобщенная технология программирования ГАРОС

1. Выполнить этап I обобщенной методики программирования ГАРОС. По его результатам сделать вывод о целесообразности решения выбранной задачи средствами ГАРОС. В случае положительного решения перейти к пункту 2).

2. Выполнить этап II. Сформировать множества подзадач классов 1, 2 и 3.

3. Выполнить этап III обобщенной методики программирования ГАРОС.

3.1. Подзадачи классов 1 и 2 решить с применением средств программирования NIOS II (язык С). Результатом этапа станет управляющая программа, которая будет исполняться на УУ ГАРОС.

3.2. Выполнить этап I методики капсульного программирования. По его результатам сформировать необходимое множество капсул для разработки, решающих все требуемые подзадачи класса 3.

Дальнейшие пункты технологии применить для разработки *каждой* капсулы из сформированного множества и учитывать, что многие описываемые программные средства только предстоит разработать в будущем.

3.3. Воспользоваться средствами ПК ПОТОК.

3.4. Описать выбранную задачу в терминах специализированного языка программирования высокого уровня, входящего в состав ПК ПОТОК.

3.5. Построить потоковый граф алгоритма, реализуемого капсулой с применением средств графического программирования ПК ПОТОК.

3.6. Преобразовать полученный потоковый граф с учетом выбранной степени параллельности, с применением средств автоматизации выявления параллелизма ПК ПОТОК.

3.7. Преобразовать полученный граф в динамический с применением средств построения рекуррентных сверток ПК ПОТОК.

3.8. Скомпилировать динамический граф в граф-капсулу с помощью вспомогательных компиляторов ПК ПОТОК.

3.9. Скомпилировать (рекуррентно свернуть) граф-капсулу в символьную и затем числовую капсулу с помощью специализированного компилятора.

3.10. Выполнить этап III методики капсульного программирования с применением средств отладки ПК ПОТОК.

4. Выполнить этап IV обобщенной методики программирования ГАРОС на специализированном тестовом стенде – имитационной модели ГАРОС в составе ПК ПОТОК.

Таким образом, в настоящем разделе описаны все необходимые компоненты структуры методологии в соответствии с определением. В Приложении А приводятся алгоритмы реализации различных временных этапов обобщенной методики программирования ГАРОС.

3.3 Инструменты моделирования ГАРОС

3.3.1 Программная имитационная модель

3.3.1.1 Структурные элементы модели

Разработанная программная имитационная модель полностью соответствует спецификации ГАРОС, что подтверждается средствами автоматизированной верификации и валидации, описание которых приводится в разделе 3.4.5. Основные результаты разработки и отладки программной имитационной модели и отдельных механизмов архитектуры опубликованы автором в статьях [79-92]. Основными компонентами модели являются:

1) Имитатор УУ –управляющая программа распознавателя изолированных слов, предназначенная для имитации запросов на вычисление капсул;

2) Буферная память – компонент, логически разделенный на два; адресуется двумя индексными регистрами IR0 и IR1; является интерфейсным между УУ и РОУ;

3) Интерфейсные регистры: IR0, IR1, F0, F1, предназначенные для моделирования взаимодействия между УУ, БП и РОУ, а также для поддержки библиотеки интерфейса;

4) Вспомогательное устройство управления, организующее многократное исполнение капсулы и ее фрагментов;

5) Распределитель – основной компонент, отвечающий за обработку потока операндов и их рассылки по секциям назначения;

6) Импликатор – компонент обработки выходных результатов, отвечающий за своевременный отбор выходных данных потока промежуточных операндов и запись их в БП;

7) Память констант (секционная, секционная-регистровая и секционная подгружаемая) – компонент, предназначенный для хранения постоянных данных, таких как параметры моделей распознаваемых слов, поворотные коэффициенты БПФ и др.;

8) Память совпадений (ПС) – переименованная память адресной проверки;

9) Память ветвлений, хранящая условия переходов и функциональные поля операнда-результата перехода; структурно аналогична памяти совпадений;

10) Жонглер (Ж), предназначенный для анализа пар операндов и принятия решения о характере протекания вычислений в вычислителе;

11) Вычислитель – совокупность вычислительных блоков (умножитель, АЛУ16, АУ40 и преобразователь тегов ПТ), поддерживающая разные режимы суперскалярных вычислений;

12) Шины E- и Em- - магистрали пересылки промежуточных данных;

13) Конструктор капсул – средство разработки капсул.

Структура имитационной модели имеет вид, представленный на рисунке 3.1.

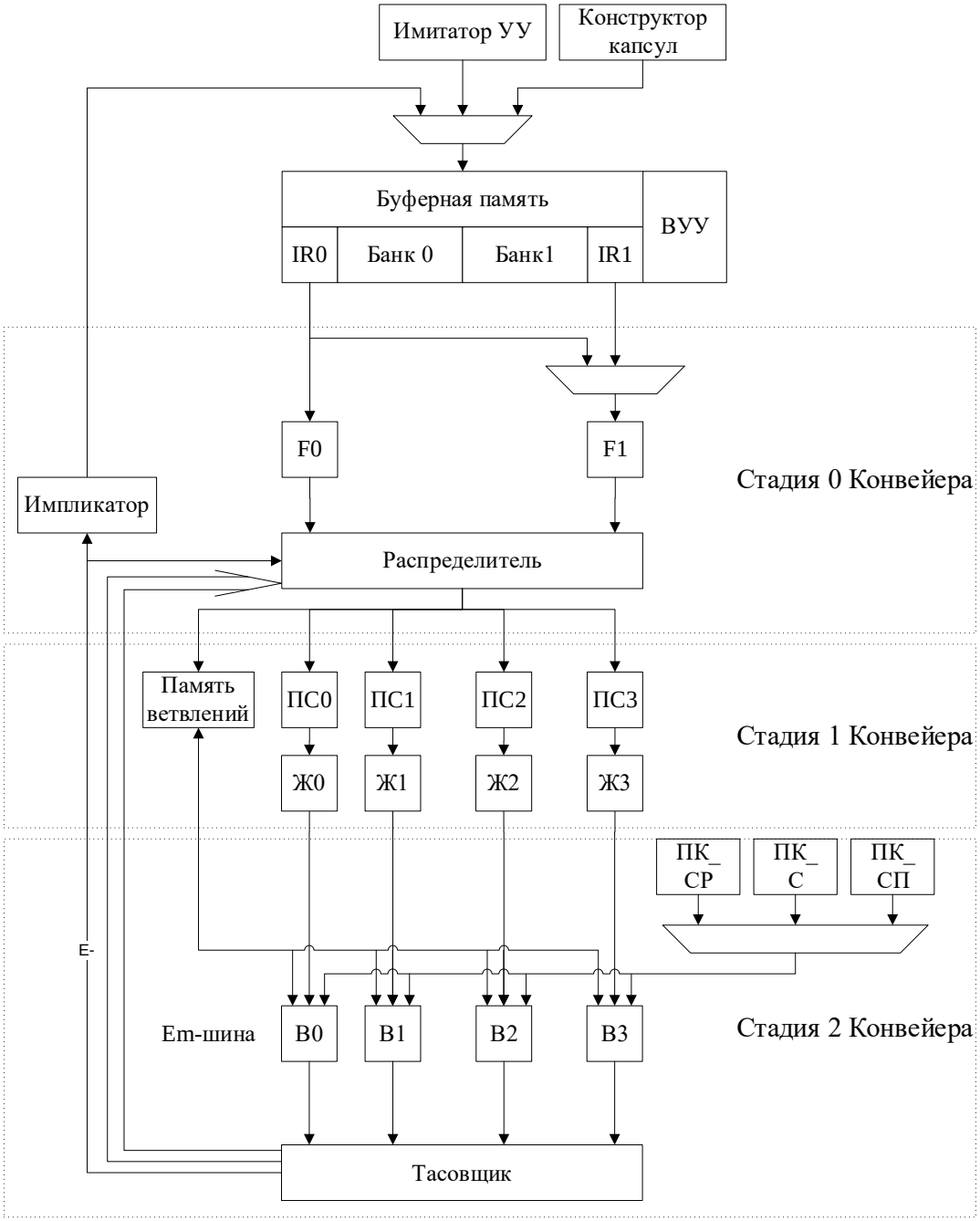


Рисунок 3.1 – Структура имитационной модели ГАРОС

Компоненты РОУ распределены по трем стадиям конвейера. В тоже время, УУ и БП функционируют параллельно РОУ и, поэтому, образуют две дополнительные стадии

макроконвейера ГАРОС. На стадии 0 конвейера функционируют компоненты Распределитель и Импликатор. На стадии 1 – Память совпадений, Память ветвлений и Жонглер. На стадии 2 – Вычислитель, Память констант и Тасовщик.

Компонент Распределитель – централизованный ресурс управления потоком входных, выходных и промежуточных данных. Он отвечает за распределение ЭСД по вычислительным ядрам, а также за обмен с БП. Состоит из двух основных элементов – Экспликатора и Ключа. Экспликатор предназначен для формирования горстей из трех источников данных. Ключ осуществляет выборку данных, рассылаемых по вычислительным ядрам, в зависимости от занятости ресурсов памяти. На рисунке А.10 Приложения А приводится структура компонента Распределитель. А на рисунке А.11 – структура составных блоков Распределителя.

Функция ПС заключается в фиксации факта совпадения и формировании пары операндов, которая будет передана для дальнейшей обработки в Вычислитель. Основная задача Жонглера - формирование комплекта функциональных полей, который будет подвергаться рекуррентным преобразованиям устройством ПТ в составе Вычислителя. Жонглер определяет также тип операции, которую будет выполнять Вычислитель. На рисунке А.12 Приложения А приводится структура совмещенных секционных ПС и Жонглера. Структура Вычислителя рассматривалась ранее в разделе 2.2.1 на рисунке 2.1.

3.3.1.2 Интерфейс взаимодействия УУ, БП и РОУ

Для обеспечения четкой регламентации процесса взаимодействия различных уровней ГАРОС автор совместно с, ныне покойным, Шнейдером А.Ю. разработал программный интерфейс взаимодействия. Описание интерфейса опубликовано в научном отчете «КАПСУЛА2» [77]. В рамках разработанного интерфейса вводятся следующие понятия:

1) Операндный сегмент (16 р.) – единица физического взаимодействия с РОУ;

2) Операнд (56 р.) – единица самодостаточных данных (ЭСД);

3) Операндное слово (64 р.) – наименование физического представления операндов; операндное слово = 4 * операндный сегмент;

4) Абсолютный адрес операндного слова - № операндного слова в БП от ее начала;

5) Капсула – именованная непрерывная последовательность операндных слов;

6) Относительный адрес (позиция)– адрес операнда (операндного слова), формируемый из имени (номера) капсулы и номера операнда в ней; допустимо взаимнооднозначное преобразование абсолютного и относительного адресов;

7) Карта памяти – основной элемент управления памятью, содержит информацию о размещении капсул и свободном пространстве;

8) Карта размещения данных (карта данных) – вторичная карта, содержит информацию о позициях содержательных операндных слов внутри капсулы;

9) Операнд шаблон – содержательный операнд, данные которого либо не инициализированы, либо подлежат изменению;

10) Бит операнда шаблона – физический разряд, указывающий на то, что операнд является шаблоном;

11) В дальнейшем термины «операнд» и «операндное слово» считаются синонимами, если специально не оговорено обратное; при этом предполагается, что операнд располагается в битах (55:0) операндного слова.

Используется статическая модель памяти – карта формируется до начала вычислений. При необходимости производится очистка всей памяти, и формирование карты начинается заново. Карта памяти и карты данных загружаются в оперативную память программой УУ.

В рамках интерфейса были введены следующие структуры данных:

- Операнд – семибайтная (56-разрядная) структура данных, несущая в себе ЭСД.
- Операндное слово – восьмибайтная (64-разрядная) структура данных, 7 байтов которой составляет операнд, а один байт - свободный.
- Капсула – в данном контексте массив операндных слов фиксированной длины, заданной программистом.
- БП и адрес в БП, как структуры данных. БП – массив операндных слов фиксированной длины, заданной реализацией РОУ и предназначенный для хранения всех исполняемых капсул. Адрес - номер (индекс) соответствующего элемента в массиве.
- Элемент данных и входной поток данных. Элемент данных – знаковое число с фиксированной точкой. Входной поток данных – массив элементов данных, где все элементы имеют одинаковый размер. Элементы данных могут преобразовываться в содержательную часть операндов, и, наоборот, содержательная часть может преобразовываться в элементы данных.
- Карта памяти – структура данных, отображающая размещение капсул в БП.
- Карта данных – структура данных, которая создается для каждой капсулы и определяет размещение элементов данных в содержательной части операндов.

Здесь и далее – разделы операнда, несущие данные или функциональные поля, выровненные по разрядам [55..48], [47..32], [31..16], [15..0], называются *слотами;* если слот содержит данные, то он называется *слотом элемента данных или слотом данных*.

В рамках интерфейса выделено четыре типа операндов: несодержательный [стандартный] (не несет в себе никаких данных), содержательный (несет в себе 1 слот данных), упакованный3 (несет в себе три слота данных), упакованный4 (несет в себе четыре слота данных). Адрес в БП содержит нули в старших 16-ти битах (используются как резерв), поэтому есть возможность использовать его для задания абсолютного адреса слота.

Операдное слово – 64 бита, которые могут рассматриваться как: монолитное 64-битное слово; структура, в которой младшие 56 битов представляют содержание операнда; четыре 16-битовых сегмента (слота), которые пронумерованы от 0 до 3 слева направо.

На рисунке 3.2 приводится побитовое представление основных структур данных.

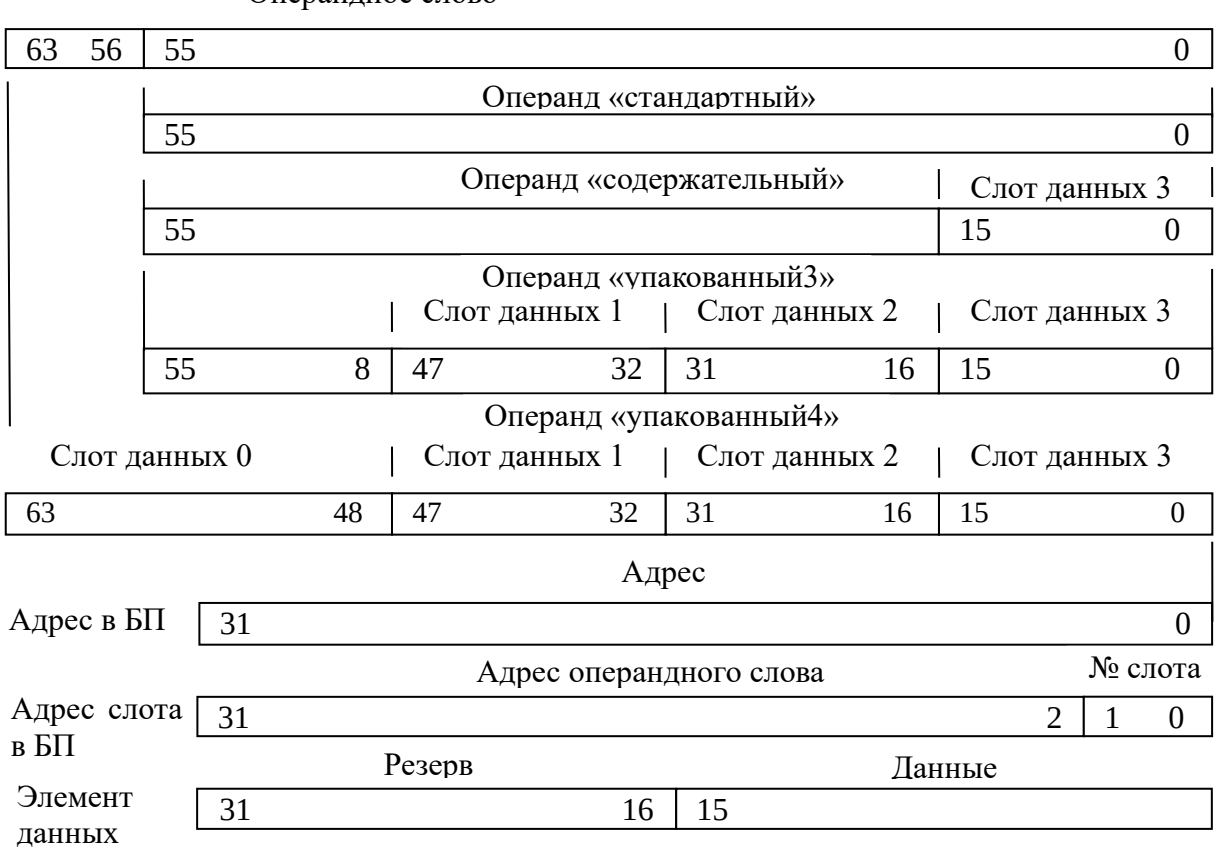


Рисунок 3.2 – Представление основных структур данных интерфейса

Буферная память предназначена для хранения капсул и наборов данных и интерпретируется как массив операндных слов фиксированного размера. Для БП применяются две схемы логической адресации: адресация операндных слов и адресация операндных сегментов, которые показаны на рисунке 3.3.

Рисунок 3.3 – Схемы адресации БП

Карта памяти представлена в виде структуры, представленной в таблице 3.1:

Таблица 3.1 – Структура карты памяти

№ капсулы (диапазона)	Адрес начала диапазона	Адрес конца диапазона	Признак занятости	Количество операндных слов	Ссылка на карту данных
0	0	15	1	16	адрес
1	16	48	1	32	адрес
2	49	101	1	52	адрес
3	102	8191	0	8089	null

Карта данных является структурой, содержащей два элемента: массив адресов слотов для размещения входных данных и массив адресов слотов для размещения выходных данных.

С учетом особенностей введенных структур данных разработаны специальные механизмы записи и чтения БП, представленные на рисунках 3.4 и 3.5.

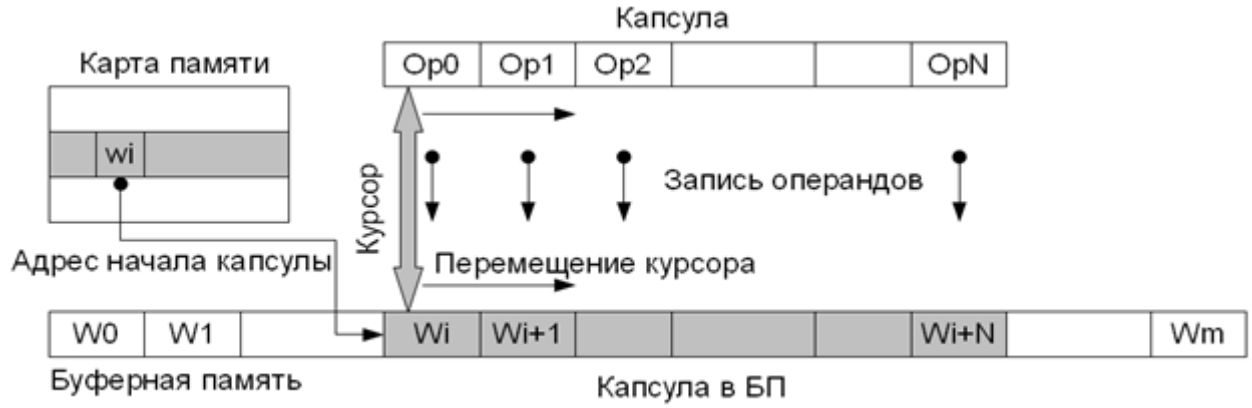


Рисунок 3.4 – Механизм записи капсул/операндов в буферную память

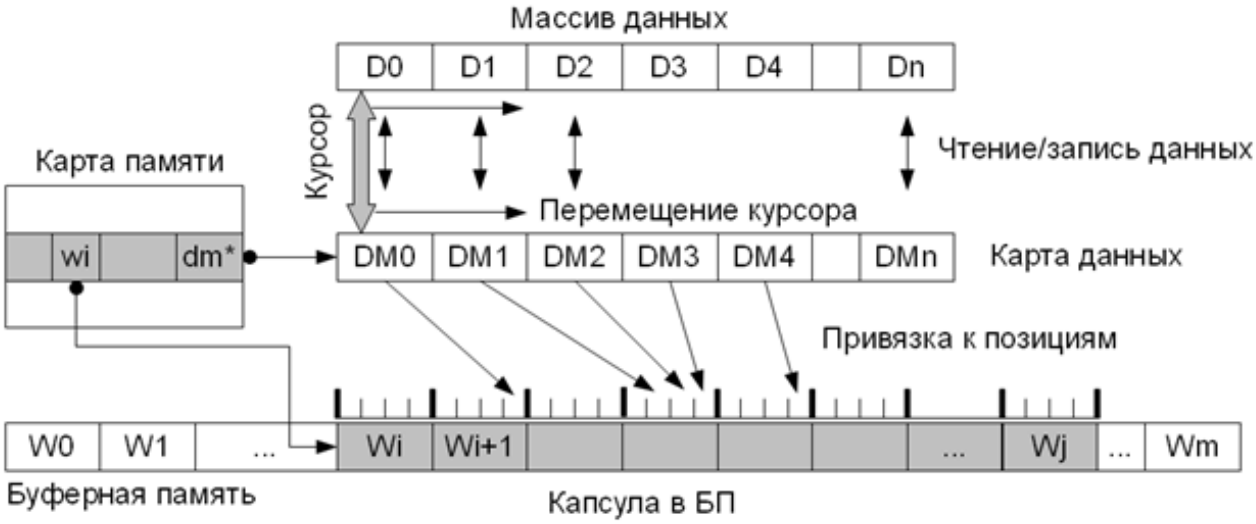


Рисунок 3.5 – Механизм чтения и записи элементов данных в буферную память

В Приложении А на рисунке А.13 представлен общий алгоритм функционирования программной имитационной модели.

3.3.2 Аппаратная VHDL-модель

Основные усилия автора диссертации в ходе исследований были направлены на развитие механизмов архитектуры и средств имитационного моделирования для отладки предложенных решений. Тем не менее, в соавторстве с Дьяченко Ю. Г. автор принимал активное участие в разработке отдельных компонент аппаратной VHDL-модели. На рисунке 3.6 представлена структура VHDL-модели ГАРОС.

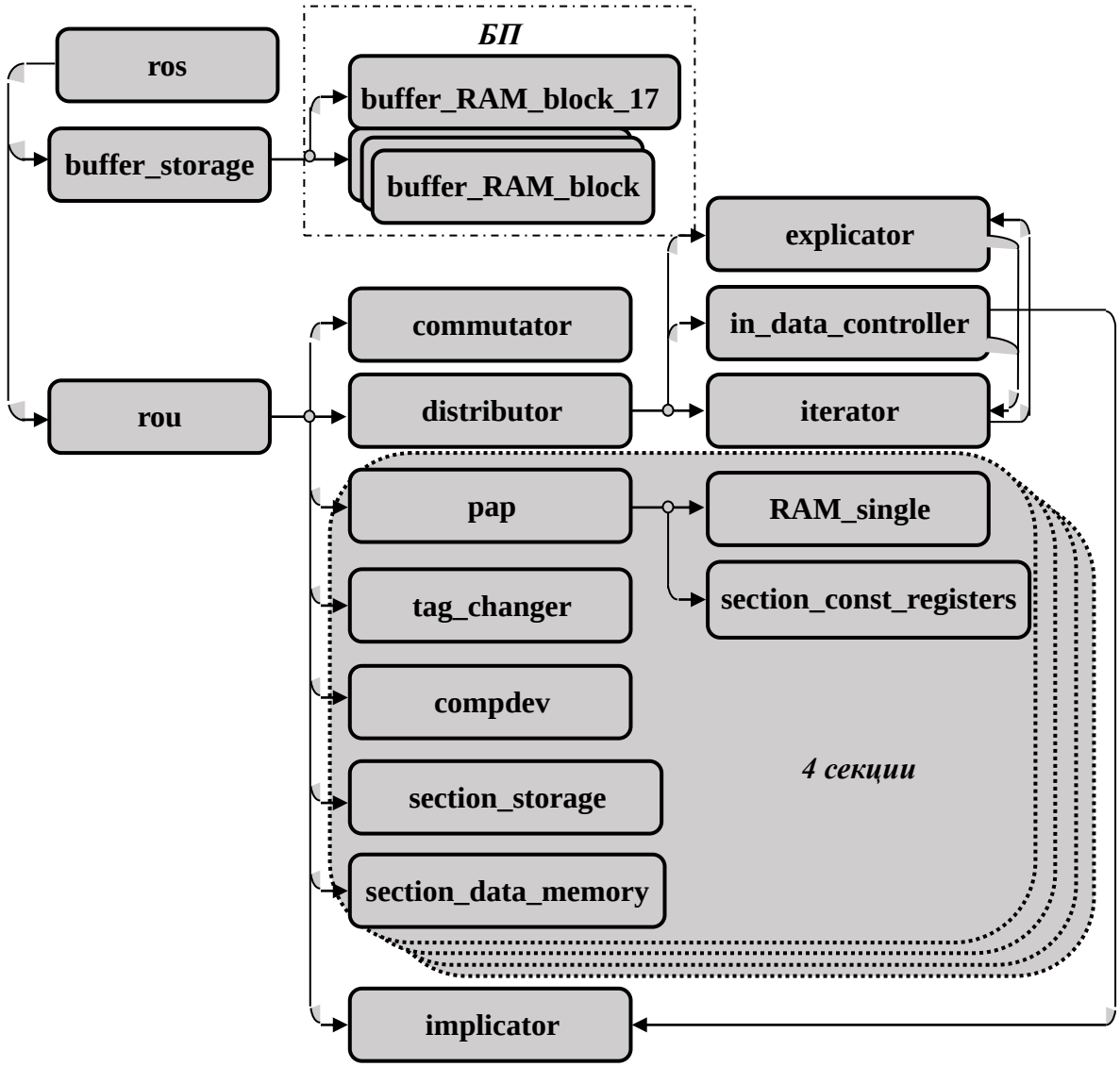


Рисунок 3.6 – Структура VHDL-модели ГАРОС

Модуль **ros** объединяет модули **buffer_storage** и **rou**. Он содержит арбитр обращений к БП со стороны УУ (чтение и запись) и РОУ (запись) и мультиплексор, формирующий флаг и данные для записи в БП со стороны УУ или РОУ.

Модуль **buffer_storage** описан в файле **RA_Buffer_Memory.vhd**. Вызывается из модуля **ros**. Модуль **buffer_storage** объединяет 4 банка БП (3 модуля **buffer_RAM_block** и 1 модуль **buffer_RAM_block_17**) и логику контроллера БП, управляющего записью и чтением в/из БП. Банки БП являются двухпортовыми оперативно запоминающими устройствами (ОЗУ).

Модуль **rou** объединяет модули **commutator, distributor**, четыре **constant_storage,** четыре **pap**, четыре **compdev**, четыре **section_data_memory,** четыре **tag_changer** и **implicator**. Совокупность модулей **constant_storage, pap, section_data_memory, tag_changer** и **compdev** образует вычислительную секцию.

Модуль **buffer_RAM_block** описан в файле **RA_Buffer_RAM_block.vhd**. Вызывается из модуля **buffer_storage**. Модуль **buffer_RAM_block** представляет собой двухпортовое ОЗУ с организацией 4096x16.

Модуль **buffer_RAM_block_17** описан в файле **RA_Buffer_RAM_block_17.vhd**. Вызывается из модуля **buffer_storage**. Модуль **buffer_RAM_block_17** представляет собой двухпортовое ОЗУ с организацией 4096x17. Старший бит хранит признак упакованности операнда.

Модуль **commutator** (Коммутатор данных, или Тасовщик) описан в файле **RA_Commutator.vhd**. Вызывается из модуля **rou**.

Модуль **compdev** (Вычислитель) описан в файле **RA_Compdev.vhd**. Вызывается из модуля **rou**.

Модуль **distributor** (Распределитель) описан в файле **RA_Distributor.vhd**. Вызывается из модуля **rou**.

Модуль **pap** (ПС) описан в файле **RA_Pap.vhd**. Вызывается из модуля **rou**.

Модуль **tag_changer** (Преобразователь Тегов, ПТ) описан в файле **RA_Tag_changer.vhd**. Вызывается из модуля **rou**. Реализует универсальный ПТ.

Модуль **explicator** (Экспликатор) описан в файле **RA_Explicator.vhd**. Вызывается из модуля **distributor**. Блок представляет собой совмещенную реализацию Экпликатора и Ключа.

Модуль **implicator** (Импликатор + Сборщик) описан в файле **RA_Implicator.vhd**. Вызывается из модуля **rou**.

Модуль **in_data_controller** (Контроллер входных данных) описан в файле **RA_In_data_controller.vhd**. Вызывается из модуля **distributor**. Данный модуль не присутствует в имитационной модели и реализуется с целью преодолеть ограничения аппаратуры (необходимо избежать образование колец, которое приводит к резкому падению тактовой частоты). Используется для распаковки операндов.

Модуль **iterator** (Итератор) описан в файле **RA_Iterator.vhd**. Вызывается из модуля **distributor**.

Модуль **RAM_Single** описан в файле **RA_RAM_Single.vhd**. Вызывается из модуля **pap**. Читает и записывает операнды только при наличии сигналов разрешения чтения и записи.

Модуль **section_const_registers** (реализует ПК_СР) описан в файле ***RA_Section_const_registers.vhd***. Вызывается из модуля **pap**. Константа постоянно присутствует

на выходе ПК_СР. Вычислитель формирует сигнал подтверждения ее использования, который инициирует чтение следующей константы из ПК_СР.

Модуль **section_constant_memory** (реализует ПК_С) описан в файле ***RA_Section_constant_memory.vhd***. Вызывается из модуля **rou**. Константа постоянно присутствует на выходе ПК_С. Вычислитель формирует сигнал подтверждения ее использования, который инициирует чтение следующей константы из ПК_С.

Модуль **section_data_memory** (реализует ПК_СП) описан в файле ***RA_Section*_data_*memory.vhd***. Вызывается из модуля **rou**.

Проект ГАРОС реализован в виде кольцевого конвейера с входным-выходным интерфейсом по шине Avalon, обеспечивающим запись в РОУ капсул, интерактивное взаимодействие между РОУ и управляющим процессором (УП) при обработке капсул и чтение результатов из РОУ в УП.

3.4 Средства аппаратно-программного моделирования и отладки

3.4.1 Программный комплекс моделирования потоковой многоядерной вычислительной системы

В ходе разработки прототипа ГАРОС и совершенствования его архитектуры инструментальная среда разработки также претерпела существенные изменения. Было разработано несколько версий данных программных средств, каждая из которых прошла процедуру государственной регистрации. Первоначальная версия средств разработки получила название GAROS IDE [93] и являлась по сути прототипом, который позволил разработать базовый комплект капсул и выявить проблемы существовавшей версии ГАРОС.

Получив опыт использования GAROS IDE и оценив недостающие функциональные возможности, была осуществлена серьезная переработка данного программного средства. В ходе выполнения гранта РНФ № 19-11-00334 «Инновационная архитектура самосинхронных цифровых сигнальных процессоров, управляемых потоком данных 2019-2021» была разработана новая версия инструментальной среды, получившая название HARSP IDE [94], а позднее и вторая версия HARSP IDE [95].

Последняя версия инструментальных средств была разработана в 2022 году в ходе выполнения научного проекта «Методы построения и моделирования сложных систем на основе интеллектуальных и суперкомпьютерных технологий, направленные на преодоление больших вызовов», финансируемого Минобрнауки по Соглашению No 075-2020-799 от 29 сентября 2020 г. Данная версия получила название «Программный комплекс моделирования потоковой рекуррентной многоядерной вычислительной системы (ПК ПОТОК)» [96]. На рисунке 3.7 представлена структура ПК ПОТОК.

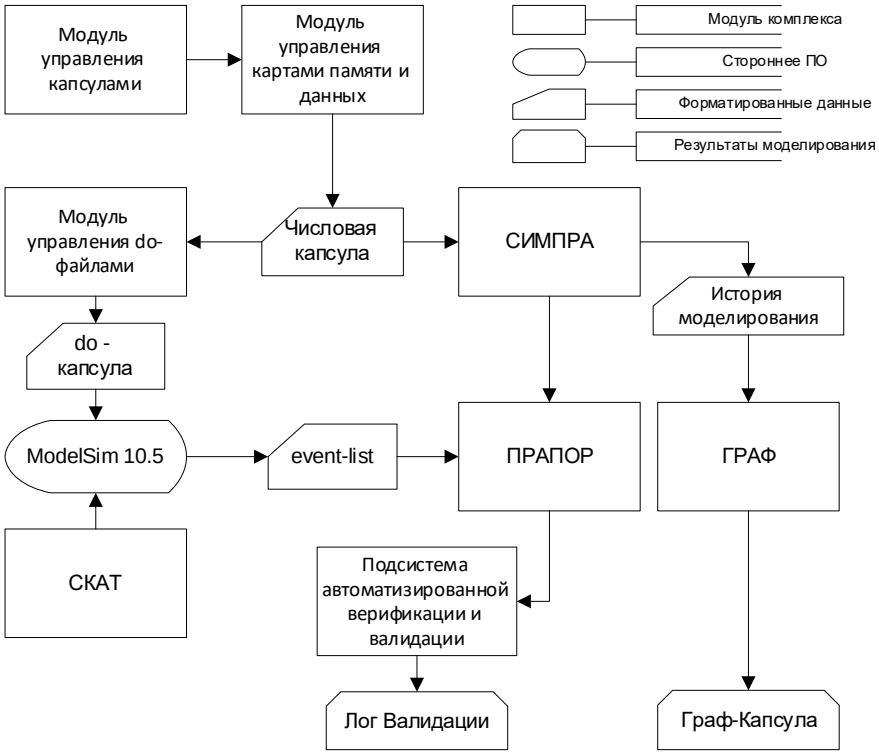


Рисунок 3.7 – Структура ПК ПОТОК

Модуль управления картами памяти и данных реализует:

- библиотеку интерфейса взаимодействия УУ и РОУ;
- создание и редактирование карты памяти для решаемой задачи;
- создание и редактирование карты данных для выбранной символьной капсулы;
- формирование числовой капсулы на основе шаблона символьной капсулы и входного комплекта данных;
- преобразование карты данных в массив констант для программы УУ;
- интерфейс с подсистемой автоматизированной верификации и валидации, который позволяет последней в автоматическом режиме создавать тестовые числовые капсулы.

Фактически модуль управления картами памяти и данных является основным связующим звеном между имитатором УУ и моделями РОУ. Модуль позволяет как в ручном, так и в автоматизированном режиме создавать числовые капсулы, которые используются для отладки моделей и символьных капсул.

Модуль управления капсулами реализует:

- текстовый редактор символьных капсул для подсистемы СИМПРА;
- визуальный конструктор капсул для СКАТ;
- средства синтаксического анализа капсул;
- преобразование символьной или числовой капсулы в формат, совместимый с подсистемой СКАТ;

• преобразование символьной или числовой капсулы в последовательность констант для программы УУ.

Модуль управления капсулами – это основной инструмент программирования капсул, которые позволяет удобным образом конвертировать их для различных подсистем моделирования и внешних средств программы УУ.

Модуль управления do-файлами реализует:

• преобразование символьных и числовых капсул в формат do-файла;

• преобразование СКАТ-капсул в формат do-файлов;

• формирование do-файлов для специальных режимов моделирования, имитирующих фрагменты программы УУ;

• интерфейс с подсистемой автоматизированной верификации и валидации, который позволяет последней в автоматическом режиме создавать тестовые do-файлы.

Рассматриваемые do-файлы являются основным инструментом определения сценария моделирования средствами СКАТ. Поэтому данный модуль обеспечивает интеграцию подсистемы СКАТ.

Модуль обработки результатов моделирования ПРАПОР [97] реализует:

• извлечение результатов моделирования капсул средствами подсистемы СИМПРА;

• извлечение результатов моделирования капсул средствами подсистемы СКАТ;

• сравнение полученных данных моделирования с эталонными, полученными от программы УУ или имитатора УУ;

• интерфейс с подсистемой автоматизированной верификации и валидации, который позволяет последней в автоматическом режиме осуществлять верификацию корректного исполнения тестовой капсулы.

Данный модуль является одним из ключевых средств автоматизации процесса отладки как компонент архитектуры, так и капсул. Автоматизация процесса извлечения и сравнения результатов существенно ускоряет разработку и отладку шаблонов капсул.

3.4.2 Подсистема имитационного моделирования СИМПРА

Для целей отладки архитектуры и капсул была разработана подсистема имитационного моделирования СИМПРА [98, 99]. В основе подсистемы имитационного моделирования СИМПРА лежит программная имитационная модель, описание которой представлено в разделе 3.3.1. СИМПРА обладает следующими функциональными возможностями:

• Разработка числовых капсул.

• Моделирование числовой капсулы средствами имитационной модели.

• Поддержка особых режимов моделирования, имитирующих различные сценарии взаимодействия с имитатором УУ.

• Отладка капсул.

• Обработка результатов моделирования средствами программы ПРАПОР.

Составными частями СИМПРА являются следующие компоненты:

• Текстовый редактор символьных и числовых капсул.

• Средства синтаксического анализа капсул (парсер).

• Средства управления симуляцией имитационной модели (симулятор).

• Визуализатор результатов моделирования.

• Программа ПРАПОР.

Текстовый редактор капсул – основной компонент разработки капсул. Реализован с использованием свободно распространяемой библиотеки текстового редактора Scintilla. Предоставляет средства редактирования операндов, а также подстановки шаблонов операндов различных типов, для ускорения процесса разработки капсул.

Парсер капсул – основной компонент синтаксического анализа капсул. Предназначен для анализа синтаксической корректности капсул с привлечением файла-спецификации формата операндов, формирования объектного представления операндов для обработки внутри имитационной модели. Также в рамках инструментальной среды используется в модуле управления капсулами для формирования капсул различных форматов.

Симулятор – основной компонент выполнения моделирования капсул. Используется для инициализации имитационной модели, загрузки капсулы, выполнения симуляции в различных режимах моделирования, пошаговой отладки процесса моделирования. Для реализации и отладки задачи РИС были введены следующие режимы симуляции:

• Стандартный режим симуляции числовой капсулы. В данном режиме осуществляется загрузка капсулы в модуль БП модели и ее пошаговое моделирование.

• Режим симуляции алгоритма RASTA-фильтрации. В данном режиме симулируется капсула, которая совмещает выполнение двух алгоритмов: вычисления RASTA-фильтра и вычисления экспоненты. В режиме многократного исполнения осуществляется подгрузка результатов фильтрации в упакованный операнд для вычисления экспонент этих значений.

• Режим симуляции алгоритма рекурсии Дурбина-Скурра. В данном режиме осуществляется перехват промежуточных результатов вычислений и симуляция операции деления на УУ.

• Режим симуляции алгоритма Евклидового расстояния. В данном режиме капсула исполняется в многократном режиме, а имитатор УУ осуществляет подгрузку констант библиотеки квантованных векторов в ПК_СП.

• Режим симуляции алгоритма Витерби. В данном режиме капсула исполняется в многократном режиме, а УУ осуществляет подгрузку констант библиотеки слов в ПК_СП.

• Режим симуляции алгоритма БПФ. В данном режиме УУ осуществляет загрузку отсчетов в специальную память отсчетов. Далее осуществляется стандартная симуляция под управлением специализированного контроллера памяти для БПФ.

Визуализатор результатов моделирования – используется для отображения состояния модели на каждом шаге моделирования. Хранит историю каждого шага моделирования. Предоставляет информацию в текстовом виде.

Впоследствии СИМПРА была полностью интегрирована в HARSP IDE в качестве одной из ключевых подсистем и дополнена средствами поддержки интерфейса взаимодействия с подсистемой автоматизированной верификации и валидации, который позволяет последней в автоматическом режиме осуществлять верификацию корректного исполнения тестовой капсулы.

3.4.3 Подсистема аппаратного моделирования СКАТ

Для разработки и отладки капсул, предназначенных для исполнения на аппаратной VHDL модели, разработана система капсульного программирования и отладки (СКАТ) [100, 101].

Составными частями СКАТ являются: визуальный конструктор капсул; компонент подготовки данных; симулятор; отладчик; визуализатор результатов.

Визуальный конструктор капсул – позволяет при помощи удобного графического интерфейса формировать капсулу из отдельных операндов. Над операндами можно выполнять следующие действия: добавление, редактирование, удаление, перемещение по капсуле.

Компонент подготовки данных – набор программных средств, которые обеспечивают создание данных, необходимых для выполнения моделирования РОУ.

Симулятор – компонент, выполняющий процесс моделирования поведенческой модели РОУ, описанной на языке VHDL. Этот компонент является сторонним продуктом под названием ModelSim корпорации MentorGraphics.

Отладчик – набор программных средств СКАТ, предоставляющих пользователю инструменты для проведения процесса отладки:

• проведение пошаговой отладки:

• просмотр хода вычислительного процесса по шагам;

• просмотр состояния внутренней памяти на любом шаге вычислительного процесса;

• возможность редактирования определенного набора элементов РОУ на любом шаге вычислительного процесса;

• просмотр временных диаграмм работы РОУ;

• анализ исключительных ситуаций, произошедших в ходе работы РОУ.

Визуализатор результатов – компонент, который можно логически выделить в СКАТ, но формально он состоит из самостоятельных элементов, задача которых - показать пользователю результаты выполнения алгоритма в РОУ: временные диаграммы, текстовое представление выходных данных, граф выполнения алгоритма.

3.4.4 Подсистема автоматизированного построения граф-капсул ГРАФ

Программные средства отладки капсул в силу их комплексности разрабатывались долго и постепенно. В условиях отсутствия отладочных средств моделирования и исполнения капсул распределение ресурсов РОУ – это комплексная и трудоемкая задача. Поэтому в целях упрощения ручного распределения ресурсов была разработана специальная графовая нотация представления вычислительного процесса, названная граф-капсулой.

Нотация граф-капсулы позволяет наглядно и компактно распределять ресурсы РОС, тем самым обеспечивая разработчиков капсул инструментарием отладки. Именно поэтому этап разработки граф-капсулы был включен в методику капсульного программирования рекуррентно-потоковой методологии, как необходимый (неизбежный) промежуточный этап разработки капсулы. На рисунке 3.8 в качестве примера приводится фрагмент граф-капсулы реализации алгоритма вычисления Евклидова расстояния.

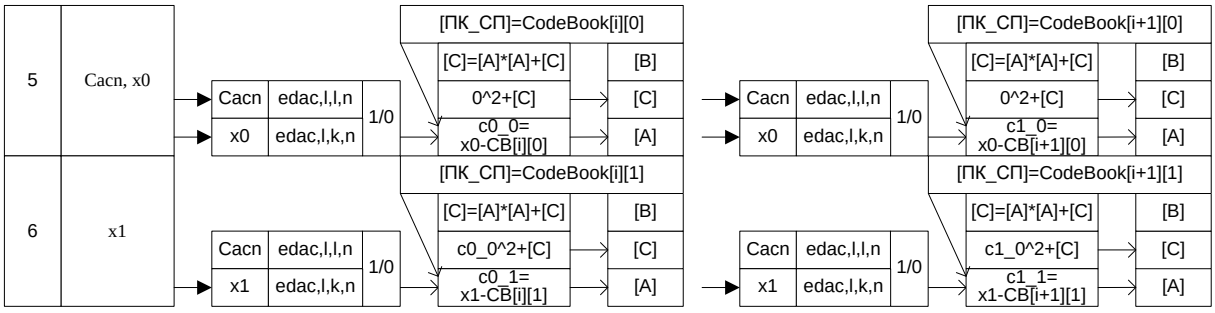


Рисунок 3.8 – Фрагмент граф-капсулы Евклидова расстояния (ручное построение)

Основное назначение граф-капсулы: компактное и наглядное представление прохождения капсулы в символьном виде, максимально приближенное к виду потокового ориентированного графа, построение которого осуществляется на предшествующих этапах разработки. Конвейерная организация РОУ привнесла дополнительные проблемы эффективного распределения его ресурсов. Чтобы преодолеть данные проблемы, при ручном проектировании граф-капсулы учитывалась последовательность функционирования стадий конвейера. При этом все стадии конвейера визуально объединялись в единую графовую структуру в рамках одного шага. На рисунке 3.9 приводится подробное схематическое распределение элементов граф-капсулы для одного шага и одной секции по ступеням конвейера и компонентам.

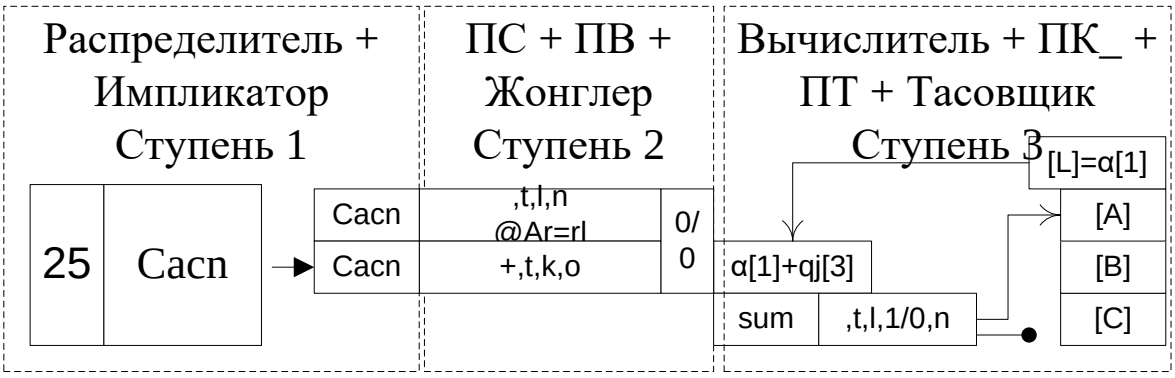


Рисунок 3.9 – Представление ресурсов РОУ на граф-капсуле

Компонеты ГАРОС обладают рядом особенностей, которые необходимо учитывать при построении граф-капсулы. В частности, компонент Вычислитель обладает суперскалярной микроархитектурой и позволяет выполнять от 1 до 4 инструкции за один цикл вычислений. В рамках граф-капсулы требуется отображать каждую выполняемую инструкцию и ее параметры (источник данных и т.п.). Другими словами, ручная версия граф-капсулы предоставляет не полное отображение ресурсов РОУ. Это связано с высокой трудоемкостью ее построения, а также с чисто физическими ограничениями свободного места на каждой странице графа.

По результатам анализа программных средств построения графов было принято решение об использовании средств GraphViz [102] для построения автоматической граф-капсулы (АГК). Библиотека GraphViz использует для описания графов специализированный встроенный язык DOT. Удобство DOT заключается в наличии большого количества параметров настройки визуального представления вершин и дуг, а также поддержка HTML-разметки для описания вершин со сложной структурой. Кроме того, данный язык использует иерархическую организацию и группировку вершин.

В силу того, что GraphViz не позволяет строить интерактивные графы, было принято решение о создании готовых шаблонов основных элементов АГК на языке DOT. Формирование графа при таком подходе заключается в наполнении готовых текстовых шаблонов данными, получаемыми в виде результатов моделирования каждой конкретной капсулы. Опыт разработки ПО в рамках капсульного стиля программирования позволил создать и эффективно использовать шаблоны фрагментов граф-капсул, которые описывают типовые схемы вычислений. Разработанные шаблоны были объединены в универсальный шаблон.

Построенный универсальный шаблон был разбит на несколько основных вершин и дуг. Для каждой из этих вершин было разработано описание на языке DOT с использованием HTML-разметки, что позволило дать уникальные идентификаторы каждому из портов, входящих в их состав. Аналогично, для каждой из дуг было разработано типовое описание на языке DOT. С учетом возможностей иерархической организации и компоновки графа в GraphViz, построение граф-капсулы было сведено к программной генерации множества дуг и

вершин, с последующей их компоновкой в кластеры и наполнением именованных портов данными из истории моделирования.

Результирующая компоновка одного цикла одной секции АГК состоит из четырех основных узлов: ПС + Жонглер, ПК + регистры вычислителя, ПТ + Тасовщик, Вычислитель. Компоненты, общие для всех секций ГАРОС были сгруппированы в два узла: Распределитель, Итератор + Импликатор. Полученная компоновка данных узлов представлена на рисунке 3.10.

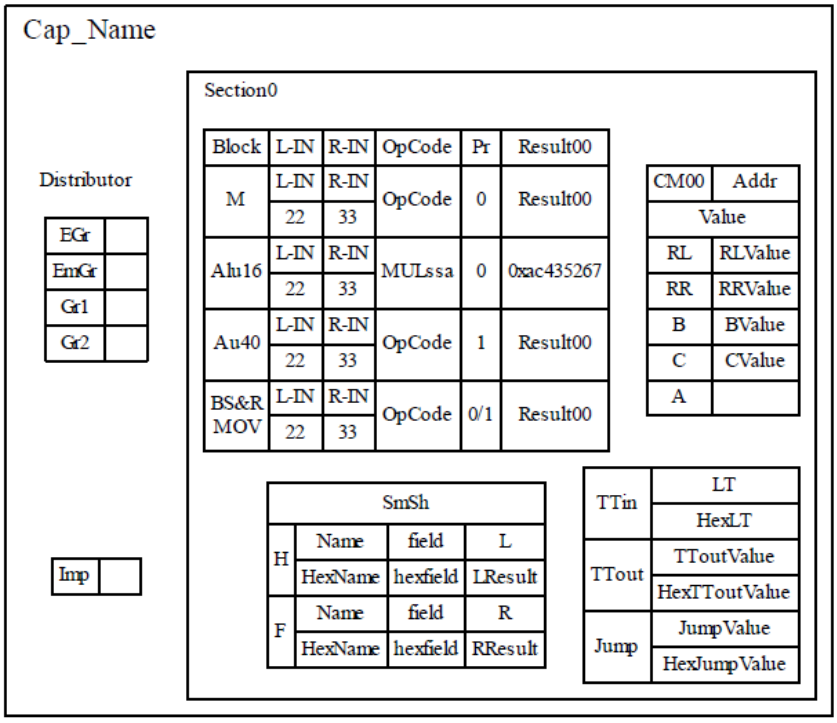


Рисунок 3.10 – Общая схема компоновки (для одной секции)

Набор утилит, входящих в состав GraphViz, содержит ряд ограничений по размеру входного графа, для которого может быть осуществлено визуальное построение. Поэтому для обхода данных ограничений общий граф разбивается на «страницы», каждая из которых содержит описание не более чем четырех вычислительных шагов. Для полученного множества графов осуществляется построение с использованием векторной графики и дальнейшее сохранение в формате многостраничного pdf-документа. Принятое решение также обеспечило хорошее качество печати граф-капсул, как элементов общей документации.

Результаты разработки подсистемы автоматического построения граф-капсул опубликованы автором в статье [103], а также получено свидетельство о государственной регистрации программы ГРАФ [104]. На рисунке 3.11 приводится обобщенная схема функционирования средства автоматизированного построения граф-капсул.

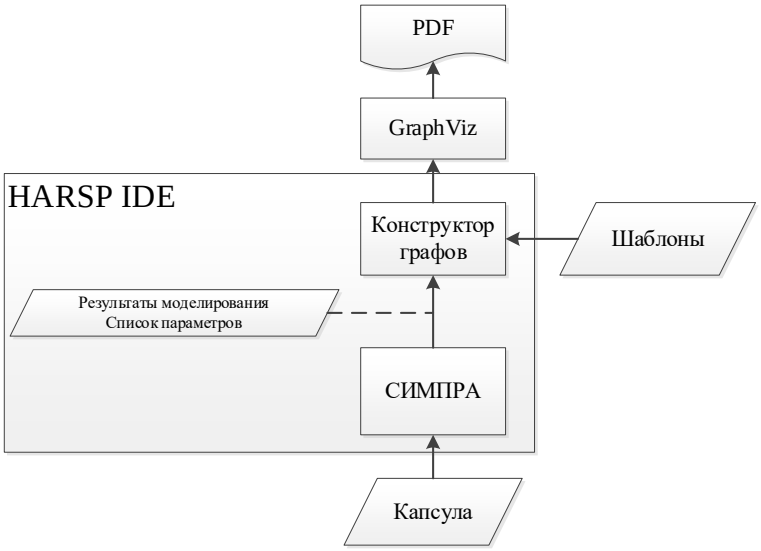


Рисунок 3.11 – Схема функционирования средств построения АГК

3.4.5 Подсистема автоматизированной верификации и валидации ГАРОС

Для обеспечения качества разработки инженерных систем применяют подходы верификации и валидации разрабатываемых компонент. Верификация – это комплекс процессов, позволяющих убедиться в соответствии разрабатываемого элемента системы требованиям спецификации. Применяя верификацию на каждом этапе разработки, мы можем убедиться в соответствии созданной системы ее спецификации. Валидация – процесс подтверждения соответствия системы требованиям к конечному продукту. Результатом валидации является объективное свидетельство корректности работы системы в целом [105].

Внедрение верификационных процессов на наиболее ранних этапах проектирования позволяет существенно повысить эффективность и качество разрабатываемых систем [106]. В качестве тестовой задачи для верификации и валидации ПЛИС прототипа МПРА была выбрана задача РИС, которая имеет эталонное решение. Необходимым условием получения высокого уровня распознавания средствами ГАРОС является обеспечение бит-экзектности эталонной программы и её капсульной реализации.

Для решения задач верификации и валидации ГАРОС используются имитационная модель на языке C# и аппаратная модель на языке VHDL. Программная модель необходима для уточнения спецификации и отладки программного обеспечения, а аппаратная – для синтеза ПЛИС прототипа. Чтобы установить, соответствуют ли разработанные модели спецификации архитектуры, был создан специальный тестовый фреймворк, основанный на Assertion-Based Design (ABD) [107] и методологии тестирования, предложенной в ИСП РАН [108]. В результате на основе данных подходов был разработан подход к разработке и тестированию ГАРОС, представленный на рисунке 3.12.

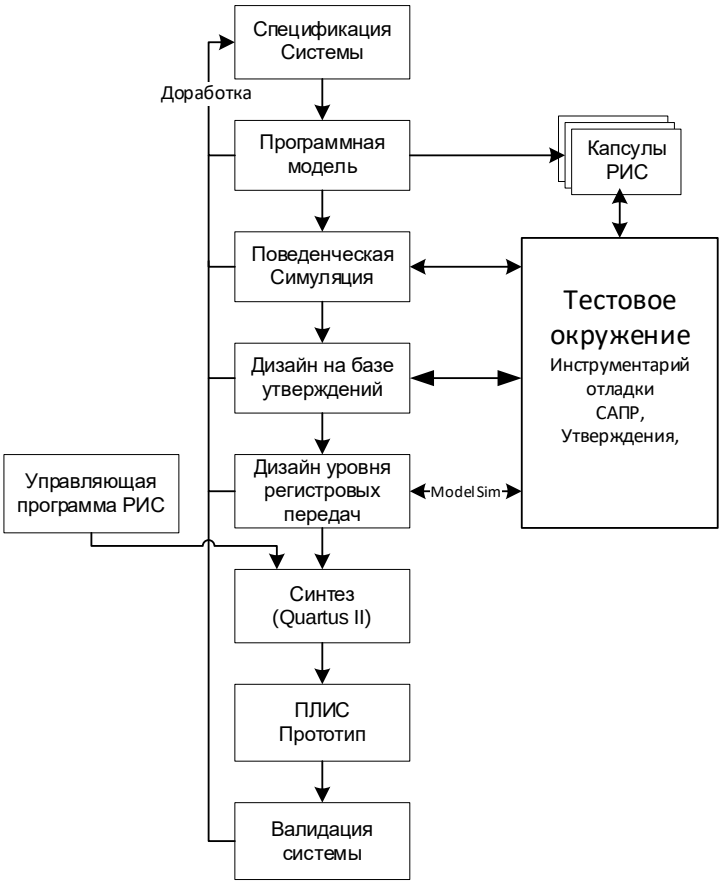


Рисунок 3.12 – Схема процесса разработки и тестирования МПРА

В качестве исходных требований валидации используется набор алгоритмов задачи РИС. При этом программная реализация данной задачи на языке C++ является источником эталонных значений результатов для сопоставления с результатами работы моделей и ПЛИС прототипа. Программа РИС используется в качестве программы управляющего уровня архитектуры. Ряд вычислительных алгоритмов реализуются в виде капсул для исполнения на РОУ.

Валидация программной модели в данном контексте является валидацией реализации вычислительных алгоритмов в виде капсул. Результатом валидации является подтверждение того, что ПО, созданное с учетом механизмов спецификации, вычисляет корректный результат. Для выполнения этой валидации требуется провести верификацию каждой капсулы на корректность вычислений и их соответствие заданной точности.

Валидация аппаратной модели осуществляется на том же множестве капсул, что и для программной модели. Результатом валидации является подтверждение того, что РОУ как устройство позволяет решать задачу РИС, и модель может быть использована для синтеза ПЛИС прототипа. Проведение валидации на проверенных корректных алгоритмах позволяет изолировать ошибки реализации VHDL-модели от ошибок реализации алгоритма, и повысить долю обнаруживаемых и устраняемых ошибок.

После извлечения данных, требуемых для верификации капсул, было сформировано более 6000 комплектов данных для каждой капсулы. Для обработки данных была разработана подсистема автоматизированной валидации, как программной, так и аппаратной моделей. Система состоит из:

1) модифицированной программы РИС;

2) облегченной программной модели;

3) аппаратной VHDL-модели;

4) тестирующего ПО.

Модифицированная программа РИС позволяет извлекать промежуточные данные вычислений в качестве входных и эталонных выходных наборов для каждой капсулы, исполнять сконфигурированное подмножество капсул на ПЛИС прототипе, и, в полной конфигурации – валидировать его.

Облегченная программная модель – модель, с переработанным интерфейсом запуска, позволяющим исполнять и контролировать модель сторонней программой, а также сниженными затратами памяти, за счет отключения отладочной информации.

VHDL-модель используется без модификации, ее симуляция проводится в ModelSim 10.5, а подготовка сценариев запуска и извлечение данных обеспечивается тестирующим ПО.

Тестирующее ПО – обеспечивает единообразную автоматизацию валидации различных моделей. Валидация обеспечивается сравнением эталонных данных с содержимым выходного раздела компонента РОУ "Буферная Память". Элементами тестирующего ПО являются:

– *инициализатор,* подготавливающий символьную версию капсулы, и ее карту данных к моделированию на выбранной модели;

– *загрузчик данных*, получающий все комплекты входных и эталонных данных, сгенерированных программой РИС для подготовленной капсулы;

– *Model Runner*, который на основании капсулы и входных данных генерирует сценарий моделирования для используемой модели и исполняет этот сценарий на модели;

– *валидатор,* извлекает из модели результаты вычислений, сопоставляет их с эталонами и формирует отчет валидации.

Основным элементом является Model Runner. Этот компонент является группой различных Runner'ов, с общим интерфейсом, обеспечивающих различные стратегии моделирования как для особенностей отдельных капсул, так и для особенностей запуска разных моделей. Две основных категории Runner’ов разделяются по типу модели, с которой они работают (программная или VHDL).

Результаты разработки подсистемы автоматизации верификации и валидации опубликованы автором в статье [109]. На рисунке 3.13 приводится схема функционирования средства тестирования.

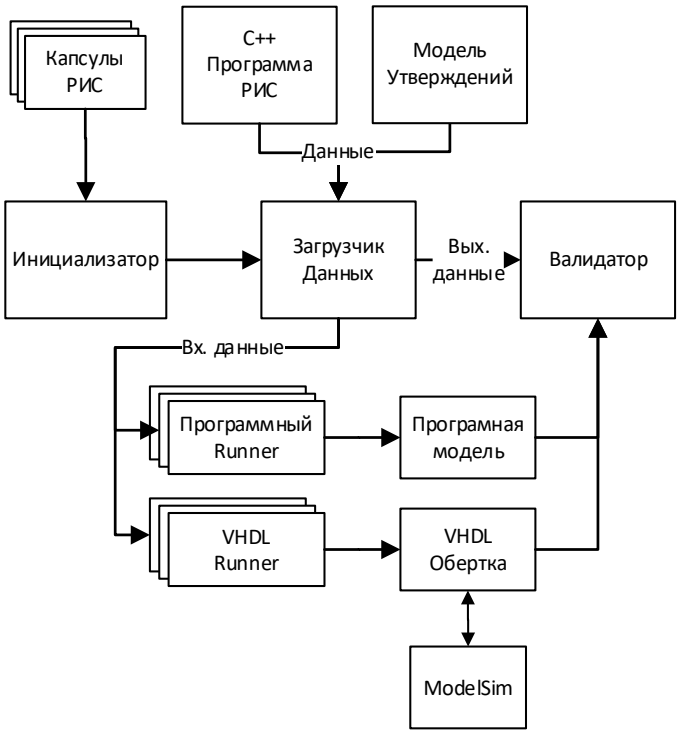


Рисунок 3.13 – Схема подсистемы верификации и валидации

3.5 Выводы к главе 3

В ходе выполнения диссертационной работы непосредственно автором или под его руководством был создан мощный методический и программно-аппаратный базис разработки и отладки ГАРОС, а также программного обеспечения – капсул. Основными результатами, рассмотренными в данной главе, являются:

1) Отдельные элементы рекуррентно-потоковой методологии программирования ГАРОС. На базе этой специализированной методологии удалось формализовать процесс разработки ПО, а также поставить задачу дальнейшего развития средств программирования.

2) В процессе реализации имитационной модели ГАРОС автором разработаны ключевые алгоритмы функционирования основных ее компонент, отражающие требования спецификации. Данные алгоритмы позволили другим разработчикам, принимающим участие в этом проекте, вести успешную реализацию аппаратной VHDL-модели ГАРОС.

3) Программный комплекс ПК ПОТОК, интегрирующий мощный инструментарий моделирования, отладки, верификации и валидации ГАРОС. Данный комплекс позволил завершить разработку ГАРОС, установить идентичность программной и аппаратной моделей.

Глава 4 Результаты программных и аппаратных испытаний ГАРОС

В данной главе приводятся результаты испытаний всех разработанных методов, алгоритмов и механизмов ГАРОС на комплекте задач ЦОС. Рассматриваются результаты реализации демонстрационной задачи распознавания изолированных слов. Испытания проведены как на модельном уровне, так и на уровне синтезированного ПЛИС прототипа. Модельные испытания показали высокие показатели производительности усовершенствованной версии прототипа ГАРОС, продемонстрировав коэффициент ускорения ~x4.5 по сравнению с эталонной одноядерной реализацией. Результаты аппаратных испытаний на уровне ПЛИС прототипа оказались ниже. Были установлены причины, реализован асинхронный алгоритм взаимодействия между уровнями ГАРОС и проведены измерения. Ускорение асинхронной реализации составило ~x3.6, что оказалось достаточно близким к модельным результатам.

Также в главе рассматриваются модельные оценки производительности ГАРОС на комплекте бенчмарков BDTIMark2000, используемых ведущими компаниями производителями цифровых сигнальных процессоров. По результатам сделан вывод о высоком потенциале конкурентоспособности ГАРОС в случае ее полноценного цифрового сигнального процессора.

4.1 Постановка демонстрационной задачи РИС

Задача РИС подробно рассмотрена в отчете [110] Распознавание слов осуществляется в три основных этапа. Первый этап – анализ речевого сигнала и выделения характеристических векторов – реализуется совокупностью следующих алгоритмов:

– автоматический контроль коэффициента усиления;

– префильтрация входного речевого сигнала фильтром Баттерворда четвертого порядка;

– вычисление барковского спектра 17-ю полосовыми фильтрами четвертого порядка;

– оценка уровня фонового шума по каждой из 17 барковских полос;

– шумоподавление с помощью преобразования барковского спектра (логарифмирование, взвешивание, RASTA-фильтрация, экспонирование) по каждой из 17 полос;

– авторегрессионное моделирование;

– дельта-расширение характеристического вектора;

– определение окончания реального произнесения слова.

Второй этап – квантование характеристических векторов реализуется поиска в кодовой книге вектора, евклидово расстояние от которого до квантуемого минимально.

Третий этап – определение степени соответствия последовательности квантованных векторов (наблюдений) моделям слов из библиотеки скрытых марковских моделей – реализуется с помощью алгоритма Витерби.

На рисунке 4.1 представлена блок-схема распознавателя слов.

Рисунок 4.1 – Бок-схема распознавателя слов

4.2 Результаты применения методики программирования для РИС

Применительно к задаче РИС Этап I обобщенной методики уже был выполнен на этапе постановки задачи. На этапе II был осуществлен глубокий анализ документации РИС, эталонной целочисленной Си-программы реализации, а также ассемблерной dsPIC-реализации.

В результате была получена детальная декомпозиция алгоритмов РИС. В соответствии с методикой, было осуществлено распределение полученных задач по уровням архитектуры. На рисунке 4.2 приводится детализированный алгоритм функционирования распознавателя слов.

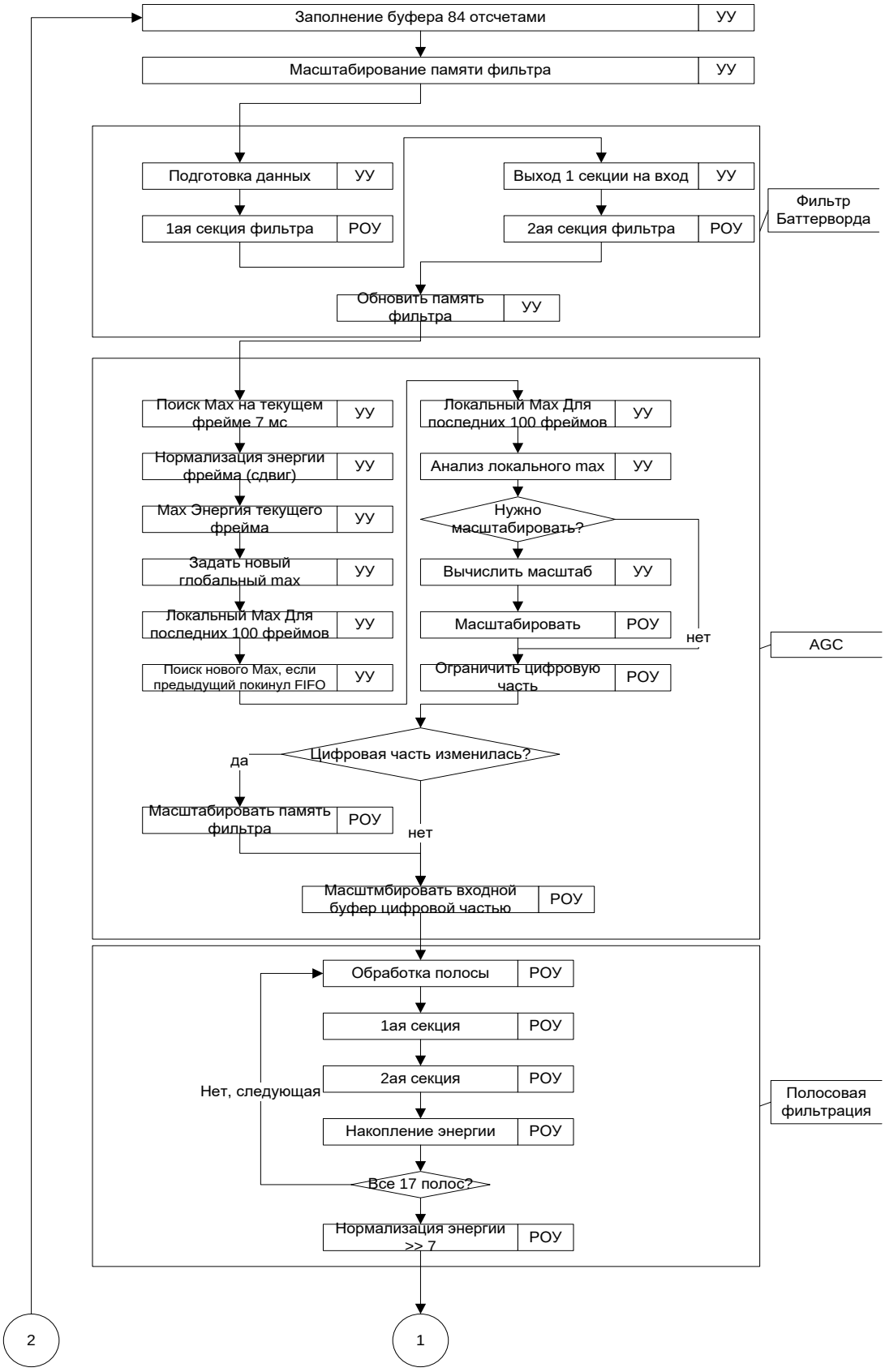


Рисунок 4.2 – Детализированный алгоритм функционирования распознавателя слов

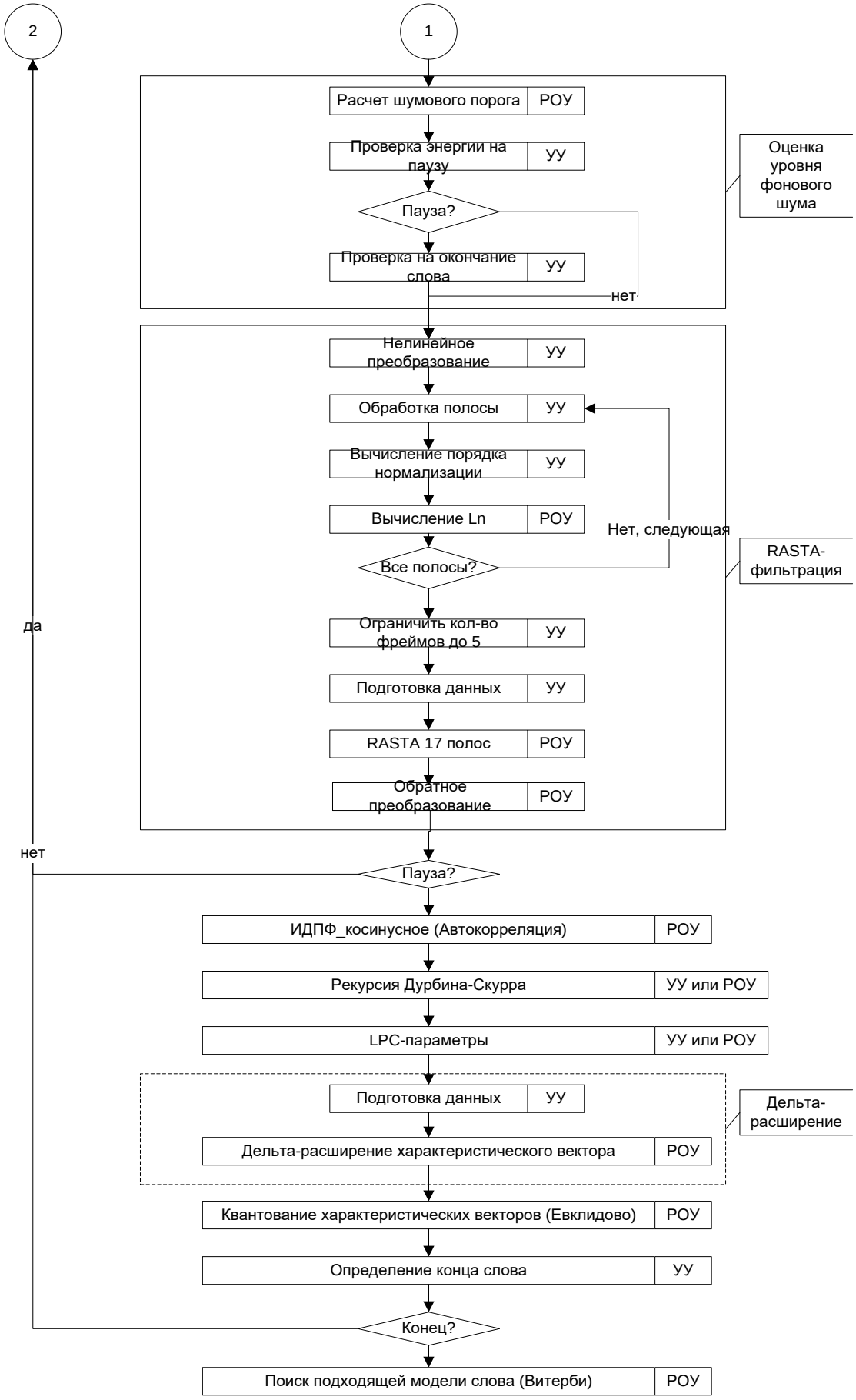


Рисунок 4.2 (продолжение) – Детализированный алгоритм функционирования распознавателя слов

Для демонстрации этапа III разработанной методологии был выбран один из алгоритмов – полосовая фильтрация. Рассмотрим применение методики капсульного программирования.

Подэтап III-1 Методики капсульного программирования

1) В качестве *задачи* принять реализацию полосового фильтра. Предположительно требуется 1 капсула, которой назначается функциональность «полосовой фильтр».

2) Привести математическую постановку.

Речевой сигнал $x_{pf}(n)$, состоящий из 84 отсчетов, фильтруется на банке из 17 полосовых фильтров. В качестве полосовых фильтров используются двухсекционные биквадратные фильтры. Передаточная характеристика фильтра описывается формулой:

$$H(z) = \frac{a}{1+2b_{11}z^{-1}+b_{12}z^{-2}} * \frac{a}{1+2b_{21}z^{-1}+b_{22}z^{-2}} \quad (4.1)$$

где a, b_{11}, b_{12}, b_{21}, b_{22} – коэффициенты фильтра, значения которых - константы, хранящиеся в отдельной таблице. Таким образом, необходимо рассчитать 17 полосовых фильтров. Отсюда следует, что *ГСП=17*.

Раскрыв z-преобразование в формуле (4.1), получим два последовательных фильтра:

$$\begin{cases} H_1(z) \leftrightarrow y_i = a * x_i - 2 * b_{11} * y_{i-1} - b_{12} * y_{i-2} \\ H_2(z) \leftrightarrow y_i = a * x_i - 2 * b_{21} * y'_{i-1} - b_{22} * y'_{i-2} \end{cases} \quad (4.2)$$

Уравнения (4.2) описывают рекурсивные фильтры второго порядка. Выходные данные первого фильтра являются входными данными для второго фильтра, откуда следует, что выделить большее количество глобальных степеней параллельности не получится. Кроме того, следует отметить, что фильтруется несмещенный входной вектор.

3) В качестве *тела* обозначить один фильтр, заданный формулой (6.1).

4) Воспользоваться теорией сложности для оценки вычислительной сложности фильтра:

Для данного алгоритма *n=84*. Секции фильтра рассчитываются последовательно, значит, их сложности складываются. Скорости выполнения операций сложения и умножения одинаковые, поэтому время вычисления одной секции фильтра оценивается как 6*n; тогда максимальное время расчета всего фильтра составляет 2*6*n., что эквивалентно O(n). Относительно низкая вычислительная сложность позволяет сделать предположение, что выбранное «тело» обладает невысокой степенью параллельности.

5) Фильтр обрабатывает несмещенный вектор. Это означает, что результат вычислений первой секции фильтра может быть сразу использован для вычисления второй секции. Таким образом, минимальная локальная СП=2. Рассмотрим одну из секций фильтра более подробно. Нетрудно заметить, что есть три независимых операции умножения, следовательно, СП секции равна 3. Таким образом, для фильтра, заданного формулой (4.1), СП=6.

6) Имеем: ГСП=17, СП=6. ГСП на порядок больше 4, следовательно, задачу необходимо декомпозировать на 17 капсул, каждая из которых рассчитывает отдельный полосовой фильтр.

7) Завершить итерацию Этапа I. Назначить *задачу* – полосовой фильтр. ГСП=6.

8) Обе секции фильтра можно рассчитывать параллельно, поэтому нужно выделить два *тела*, каждое из которых соответствует секции фильтра.

9) Сложность каждого тела O(n).

10) СП=3

11) Имеем: ГСП=6, СП=3. Декомпозиция завершена, результат - последовательность из 17 капсул, каждая капсула рассчитывает обе секции фильтра.

Для расчета каждой секции фильтра требуется два вычислительных устройства. Таким образом, необходимо реализовать алгоритм с СП=3, используя два вычислителя.

Этап II Методики капсульного программирования

1) Для краткости опустим подробное описание п.п. 1-4 этого этапа и приведем преобразованный потоковый граф с СП=4, изображенный на рисунке А.14 Приложения А.

2) Динамический граф может быть получен путем сворачивания подграфов на рисунке А.14, выделенных пунктиром, которые циклически повторяются, в вершины графа.

3) Преобразования графа в граф-капсулу представлено на рисунке А.15 Приложения А.

4) Полученная граф-капсула сворачивается в символьную, объем которой составил 87 операндов (без учета выходного раздела, т.к. на момент создания именно этой версии капсулы механизмы обработки выходных данных еще не были усовершенствованы).

5) Полученная символьная капсула моделируется средствами программы СИМПРА. Следующим шагом отладки является формирование числовой do-капсулы средствами программы СКАТ и моделирование ее при помощи аппаратной VHDL-модели.

После введение в архитектуру и реализации в модели усовершенствованных механизмов, описанных в главе 2, появилась возможность оптимизировать потоковый граф вычисления полосового фильтра. В результате, используя новые ресурсы Вычислителей, удалось реализовать вычисление одной секции фильтра на одной секции РОУ за 5 шагов. Таким образом, фильтр целиком вычисляется на двух секциях РОУ. Это позволило одновременно вычислять сразу 2 полосовых фильтра.

Следует также отметить, что старая версия фильтра требовала 439 циклов вычислений, а новая – 437. В результате, реализация усовершенствованных механизмов архитектуры привела к **более чем двукратному росту** скорости вычисления алгоритма полосовой фильтрации по сравнению с предыдущей версией прототипа ГАРОС. Для большинства оставшихся алгоритмов были получены сопоставимые результаты ускорения вычислений.

На рисунке А.16 Приложения А приводится фрагмент оптимизированного потокового графа. А на рисунке А.17 – фрагмент автоматической граф-капсулы.

4.3 Результаты программно-аппаратных испытания РИС на моделях

В результате применения разработанных элементов рекуррентно-потоковой методологии реализовано большинство алгоритмов задачи распознавания слов. Для оценки эффективности и корректности работы ГАРОС используется проект распознавания слов, реализованный ранее в рамках сотрудничества с компанией Microchip. Программа РИС, реализованная для микроконтроллера dsPIC30F является эталонной для сравнения с ГАРОС версией.

Приведенные модельные оценки коэффициента ускорения рассчитаны для одноядерного микроконтроллера dsPIC30F с учетом следующих ограничений:

- Вычислители ГАРОС функционируют в суперкалярном режиме, а dsPIC30F функционирует в суперскалярном режиме только при исполнении ограниченного множества команд, для которых определена подобная семантика.
- На модельном уровне время заполнения капсул данными не могло быть адекватно оценено. Поэтому в процессе измерений не учитывались временные затраты на подготовку данных как для dsPIC30F, так и для ГАРОС.

Результаты сравнения реализации алгоритмов распознавателя в среде четырехядерного ГАРОС и одноядерного dsPIC30F представлены в таблице 4.1. В разделе 2.1 было отмечено, что для предыдущей версии ГАРОС среднее значение коэффициента ускорения реализации капсул составило 2-2.5. Из таблицы 4.1 видно, что среднее значения коэффициента ускорения для новой версии прототипа составляет ~4.5. Это свидетельствует о высокой степени эффективности предложенных и реализованных механизмов. Результаты различных периодов испытаний опубликованы автором в работах [66-69, 83-85].

Таблица 4.1 – Результаты реализации алгоритмов распознавания

Название алгоритма	**Объем капсулы (в операндах)**	**Использо-ванная ПК**	**Число циклов dsPIC30F**	**Число цик-лов РОУ**	**Коэф. ускорения**
БПФ2_256	155	256 констант	~19000	1040	~18,2
Баттеруорт (две секции)	89	ПК_СР	1360	437	3,11
Полосовой фильтр (две полосы)	93	ПК_СР	1428*2	437	6,5
Натуральный логарифм (NaturLog1_x4)	58	-	196	119	1,65
RASTA фильтр + Экспоненцирование	105	ПК_С (6 кон-т)	~1000	164	6,1 (для 5 процессоров)
Натуральный логарифм (NaturLog0_x1)	25	-	36	15	2,4
Косинусное ИДПФ	26	ПК_С (38 кон-т)	1100 (без) / 484 (с ПК)	51	21,6 (без)/9,5(с ПК)
Рекурсия Дурбина-Скурра	297	-	~800	124	~6,5
PLP параметры	63	-	144	30	4,8
Delta-расширение	48	-	91	27	3,4
Евклидово расстояние	89 (edac)	ПК_СП (16 кон-т)	5632	1285 (edac)	4,4
Витерби (для текущего N=61)	124	ПК_СП (11 кон-т)	5408*4	5648	3,83

4.4 Результаты испытаний РИС на ПЛИС прототипе архитектуры

На основе верифицированной VHDL-модели была синтезирована первая версия макетного образца ГАРОС, реализованная в ПЛИС технологии с использованием Cyclone V GT Development Kit. В таблице 4.2 приводятся результаты синтеза микросхемы ПЛИС, полученные с помощью утилиты синтеза, которая входит в Development Kit.

Таблица 4.2 – Результаты синтеза ПЛИС прототипа

Synthesis summary	**Fitter Summary**		
	Total summary	***ROU summary***	***NIOS+BM summary***
Fitter Status	Successful - Tue Sep 04 16:28:53 2018		
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Standard Edition		
Revision Name	NROS		
Top-level Entity Name	NROS		
Family	Cyclone V		
Device	5CGTFD9E5F35C7		
Timing Models	Final		
Logic utilization (in ALMs)	55,642 / 113,560 (49 %)	50979	4663
Total registers	51215	43385	7830
Total pins	53 / 616 (9 %)	0	53
Total virtual pins	0	0	0
Total block memory bits	8,808,780 / 12,492,800 (71 %)	147148	8661632
Total RAM Blocks	1,127 / 1,220 (92 %)	52	1075
Total DSP Blocks	8 / 342 (2 %)	4	4

Основной целью создания данной версии макетного образца являлась проверка целостности и универсальности архитектуры в области ЦОС, а также – в верификации поведенческой модели прототипа на реальной аппаратуре. Для достижения поставленной цели, задачи оптимизации основных аппаратных блоков макетного образца были отодвинуты на второй план. Также в качестве УУ был выбран фон-неймановский процессор общего назначения NIOSII, что позволило упростить как отладку макетного образца, так и реализацию демонстрационной задачи. Как видно из таблицы 4.2, синтезированный макетный образец использует примерно половину всех блоков ПЛИС, при этом УУ занимает около 9-10% от общего размера проекта. В будущем процессор общего назначения будет заменен на упрощенный процессор ввода/вывода, а реализация РОУ существенно оптимизирована, что позволит значительно снизить аппаратные расходы.

Ограничения реализации макетного образца позволило достичь уровня тактовой частоты NIOSII в 100 МГц, а РОУ – 12,5 МГц. Таким образом, продолжительность 1 цикла работы РОУ составляет 8 циклов работы NIOSII. Данное соотношение было использовано для синхронизации УУ и РОУ, а также при оценке уровня производительности макетного образца.

Полученные в ходе разработки и испытаний капсулы были использованы в ходе испытания ПЛИС прототипа на полном наборе распознавания 100 произнесений. Результаты последовательного распознавания 100 слов библиотеки представлены в таблице 4.3.

Таблица 4.3 – Результаты апробации ПЛИС прототипа

Название капсулы	**Циклов NIOSII на обработку 100 слов**			**Коэффициент ускорения**	
	"Чистый" NIOSII	***Синхронный* ГАРОС***	***Асинхронный* ГАРОС***	***Синхронный ГАРОС***	***Асинхронный ГАРОС***
Баттеруорт (две секции)	2378	2809	1457	0.85	1.63
Полосовой фильтр (две полосы)	42298	11520	11520	3.67	3.67
Натуральный логарифм (NaturLog1_x4)	1601	2303	2102	0.70	0.76
RASTA фильтр + Экспоненцирование	1743	3349	716	0.52	2.43
Косинусное ИДПФ	570	274	115	2.08	4.96
Натуральный логарифм (NaturLog0_x1)	47	161	58	0.29	0.81
PLP параметры	151	269	79	0.56	1.91
VectQuant_ED	20841	7018	3247	2.97	6.42
Delta-расширение	165	513	100	0.32	1.65
Евклидово расстояние	13335	10815	4576	1.23	2.91
Другое	1592	1670	-	0.95	-
Итого	85799	40701	23970	2.11	3.58

* Синхронный ГАРОС – реализует синхронную дисциплину взаимодействия управляющего и операционного уровней архитектуры. Асинхронный ГАРОС – реализуется асинхронную дисциплину взаимодействия, соответственно

Первоначально для успешного запуска ПЛИС прототипа были максимально упрощены алгоритмы загрузки и чтения данных со стороны УУ. Была реализована синхронная дисциплина взаимодействия УУ и РОУ. Данное решение привело к тому, что общий коэффициент ускорения составил всего ~2.11. Такой результат подтвердил предположение, что взаимодействие УУ и РОУ станет узким местом архитектуры. Поэтому была реализована асинхронная дисциплина взаимодействия. В результате был получен общий коэффициент ускорения x3.6, что достаточно близко к рассчитанному в разделе 3.3 коэффициенту x4.5. Таким образом, полученные результаты испытаний могут считаться успешными, т.к. они подтвердили потенциал архитектуры и, в тоже время, позволили обнаружить проблемы в реализации взаимодействия между УУ и РОУ, а также в ряде капсул.

Одним из путей снижения издержек взаимодействия может стать такая модификация капсул, которая снизит трафик между УУ и РОУ. Одним из возможных способов этого добиться может стать укрупнение капсул, что позволит РОУ управлять промежуточными данными без участия УУ. Другим возможным способом является развитие механизмов передачи управления от капсулы к капсуле внутри самого РОУ.

Результаты испытаний ПЛИС прототипа опубликованы автором в работе [111]. А уточненные результаты реализации и верификации ПЛИС прототипа в статьях [112-115].

4.5 Оценка производительности ГАРОС на комплекте бенчмарков BDTIMark2000

В книге [73] рассматривается программный продукт компании Berkeley Design Technology, Inc. (BDTI), как одного из лидеров бенчмаркинга ЦСП. Цитата: «BDTI introduced its core suite of DSP benchmarks, formally called the “BDTI Benchmarks,” in 1994. The BDTI Benchmarks consist of a suite of 12 algorithm kernels that represent key DSP operations used in common DSP applications. BDTI revised, expanded, and published information about the 12 DSP algorithm kernels in its BDTI Benchmark in 1999». В таблице 4.4 (таблица 10.7 из [73]) приводится таблица алгоритмических ядер, измеряемых в BDTI Benchmarks.

Таблица 4.4 – BDTI Benchmark Kernels

Function	Function Description	Example Applications
Real Block FIR	Finite impulse response filter that operates on a block of real (not complex) data.	Speech processing (e.g., G.728 speech coding).
Complex Block FIR	FIR filter that operates on a block of complex data.	Modem channel equalization.
Real Single-Sample FIR	FIR filter that operates on a single sample of real data.	Speech processing, general filtering.
LMS Adaptive FIR	Least-mean-square adaptive filter; operates on a single sample of real data.	Channel equalization, servo control, linear predictive coding.
IIR	Infinite impulse response filter that operates on a single sample of real data.	Audio processing, general filtering.
Vector Dot Product	Sum of the pointwise multiplication of two vectors.	Convolution, correlation, matrix multiplication, multi-dimensional signal processing.
Vector Add	Pointwise addition of two vectors, producing a third vector.	Graphics, combining audio signals or images.
Vector Maximum	Find the value and location of the maximum value in a vector.	Error control coding, algorithms using block floating-point.
Viterbi Decoder	Decode a block of bits that has been convolutionally encoded.	Error control coding.
Control	A sequence of control operations (test, branch, push, pop, and bit manipulation).	Virtually all DSP applications include some control code.
256-Point In-Place FFT	Fast Fourier Transform converts a time-domain signal to the frequency domain.	Radar, sonar, MPEG audio compression, spectral analysis.
Bit Unpack	Unpacks variable-length data from a bit stream.	Audio decompression, protocol handling.

Алгоритмическими ядрами называются функции, которые представляют собой основные строительные блоки большинства задач обработки сигналов. Эти ядра являются наиболее ресурсоемкими частями задач ЦОС. Подразумевается, что эти ядра высоко оптимизированы и разрабатываются вручную для каждого конкретного ЦСП. Поэтому было принято решение разработать и оценить производительность РОУ на собственном набора алгоритмических ядер, основанном на аналогичных алгоритмах («BDTI Benchmarks» является платным программным

продуктом). В дальнейшем планируется использовать данные разработки для создания собственной низкоуровневой библиотеки алгоритмов ЦОС.

В качестве эталона для реализации и сравнения была использована публичная документация ЦСП TMS320C55x (семейства C55x) компании Texas Instruments (TI). Причинами данного выбора являются:

- Продукты компании TI занимают лидирующие позиции на рынке ЦСП;
- Библиотека TMS320C55x DSP Library хорошо документирована и содержит описание, реализацию и оценки производительности по количеству циклов различных алгоритмов ЦОС;
- Указанная библиотека в полном объеме реализует BDTI Benchmarks.

В результате анализа TMS320C55x DSP Library были выбраны необходимые функции, реализующие алгоритмические ядра BDTI Benchmarks. Оказалось, что указанная библиотека содержит несколько версий реализации этих ядер, ввиду специфики аппаратной реализации линейки C55x. Кроме того, в C55x DSP Library четко прописано следующее: «Assumes all data is in on-chip dual-access RAM (provided linker command file reflects those conditions)». Это означает, что производительность функций библиотек вычисляется в предположении, что все данные уже присутствуют на чипе. Поэтому, были реализованы соответствующие версии для ГАРОС, оцениваемые с аналогичными предположениями. В Приложении А приводится подробное описание каждого алгоритмического ядра. В таблице 4.5 представлены предварительные оценки реализации всех рассмотренных алгоритмов.

Таблица 4.5 – Предварительные оценки производительности ГАРОС

Алгоритм	**Производительность в циклах**		
	C55x	***ГАРОС***	
Block FIR 1 MAC	C[a]: nx * (2 + nh) / O[b]: 25	C: nx * (1 + nh) / O: 12	
Block FIR 2 MAC	C: nx/2 * (6 + nh) / O: 25	C: nx/2 * (1 + nh) / O: 12	
Single Sample FIR 1 MAC	C: (1 + nh) / O: 44	C: (1 + nh) / O:15	
Single Sample FIR 2 MAC	C: (1 + nh/2) / O: 24	***Real-time***	***Quasi real-time***
		C: (3 + nh/2) / O: 15	C: (1 + nh/2) / O: 15
Complex Block FIR	C: nx * [8 + 2*(nh-2)] / O: 51	C: nx * (1 + 2*nh) / O: 14	
LMS Adaptive FIR	C: nx * (5 + 2*nh) / O: 26	C: nx * (4 + 3*nh) / O: 12	
IIR Double Precision Form II	C: nx * (7 + 31*nbiq[c]) / O: 77	C: nx * (4 + 21 * nbiq) / O: 12	
IIR 4 Coefficient Form II	C: nx * (2 + 3*nbiq) / O: 44	C: nx * (2 + 5*nbiq) / O: 12	
IIR 5 Coefficient Form II	C: nx * (5 + 5*nbiq) / O: 60	C: nx * (2 + 6*nbiq) / O: 12	
IIR 5 Coefficient Form I	C: nx * (5 + 8*nbiq) / O: 68	C: nx * (1 + 7*nbiq) / O: 12	
Vector Dot Product	C: nx + 1 / O: 44	C: nx + 1 / O: 15	
Vector Add	C: nx * 3 / O: 23	C: nx * 2 / O: 14	
Vector Maximum	C: nx * 3 / O: 8	C: nx * 5 / O: 12	
Viterbi Decoder[d]	C: Frame * 34 / O: 121	C: Frame * 109 / O: n/a	
256-Point In-Place FFT	C: 5 * N/2 * $\log_2 N$	C: 4 * N/2 * $\log_2 N$	

Результаты бенчмаркинга HARSP позволяют сделать вывод, что архитектура является жизнеспособной и обладает хорошим потенциалом производительности в классе задач ЦОС. В частности, большинство kernel benchmarks показали идентичный уровень производительности, как и продукт компании TI, лидирующей на рынке ЦСП.

Тем не менее, бенчмаркинг выявил ряд архитектурных проблем, из-за которых определенные задачи решаются неэффективно. Поэтому были более глубоко и подробно изучены архитектуры современных ЦСП и методы реализации типов задач из этого класса. В результате был разработан набор предложений по развитию ГАРОС, внедрение которых позволит достигнуть желаемого уровня производительности.

Дальнейшее развитие ГАРОС автор видит в разработке на основе алгоритмических ядер и DSP Library C55x собственной высоко оптимизированной библиотеки ЦОС. В состав этой библиотеки должны в максимально полном объеме войти реализации всех наиболее часто используемых алгоритмов в ЦОС, в том числе и **различные версии реализации БПФ**, представленные в библиотеке C55x (от 8 до 1024 отсчетов, БПФ двойной точности и др.).

Эту библиотеку следует также снабдить средствами высокоуровневого описания на языке C. Успешная разработка данной библиотеки обеспечит конкурентоспособность ГАРОС на рынке отечественных ЦСП.

Результаты оценки производительности опубликованы автором в работе [116].

4.6 Выводы к главе 4

Программно-аппаратные испытания и испытания ПЛИС прототипа продемонстрировали несколько важных результатов:

1) Прототип ГАРОС обладает высокой степенью завершенности и способен решать задачи ЦОС.

2) Прототип ГАРОС обладает высоким теоретическим уровнем производительности на классе задач ЦОС и, в частности, на задаче РИС.

3) ПЛИС прототип и программа управляющего уровня имеют ряд проблем реализации, которые не позволили в полной мере раскрыть потенциал синтезированного образца.

4) В случае устранения проблем опытный, а в последствии и серийный, образец ГАРОС может быть конкурентно способным на рынке ЦСП решений.

5) ГАРОС следует снабдить объемной библиотекой оптимизированных капсул, реализующих основные алгоритмы ЦОС.

Заключение

В соответствии с целью диссертационного исследования были решены актуальные научно-практические задачи:

1) разработаны структурные блоки, методы и алгоритмы организации вычислительного процесса в многоядерной потоковой рекуррентной архитектуре и ее прототипе ГАРОС;

2) разработаны элементы методологии программирования и отладки ГАРОС.

Решение данных задач позволило разработать прототип ГАРОС, который на уровне модельных оценок сопоставим по производительности с ЦСП компании Texas Instruments серии TMS C55x. Так на основе полученных результатов был синтезирован ПЛИС прототип ГАРОС, который успешно решает демонстрационную задачу распознавания изолированных слов.

В соответствии с задачами диссертационной работы были получены следующие основные результаты:

1) Выполнен анализ существующих методов организации параллельных вычислений, основной акцент был сделан на методах реализации параллелизма уровня инструкций. Данный вид параллелизма реализуется большинством современных высокопроизводительных ЦСП.

2) Выполнен анализ характерных проблем потоковой модели вычислений и проведено исследование рекуррентно-потоковой модели вычислений и архитектуры на ее основе на предмет решения данных проблем. Установлено, что большинство типовых проблем потоковых моделей вычислений в ГАРОС было успешно решено.

3) Выполнен поиск характерных представителей отечественных вычислительных систем потоковой архитектуры. Осуществлен их сравнительный анализ с МПРА. Сделан вывод, что процессор Мультиклет является ближайшим аналогом ГАРОС, но предназначен для решения других классов задач. Поэтому для оценки производительности ГАРОС сделан выбор в пользу представителей современных высокопроизводительных ЦСП.

4) Выполнен анализ функциональных возможностей существующей версии прототипа архитектуры на предмет эффективности реализации задач ЦОС. Было обнаружено, что прототип не предоставляет ряда возможностей, характерных современным ЦСП. Поэтому были сформулированы проблемы реализации задач ЦОС средствами ГАРОС.

5) Разработаны структурные блоки и механизмы их функционирования, которые позволили решить проблемы, выявленные на этапах анализа 2) и 4). К этим блокам и механизмам относятся: память констант подгружаемая; механизм многократного исполнения специализированных программ – капсул; механизмы косвенной репликации; механизмы обработки выходных данных.

6) Разработаны две схемы организации суперскалярных вычислений в вычислительных ядрах ГАРОС. В рамках первой схемы функциональные блоки ядер работают в параллельном

режиме и не зависят друг от друга по данным, обеспечивая двух, трех или четырех задачный режим. Во второй схеме функциональные блоки работают в последовательном режиме на разных подтактах синхросигнала и зависят друг от друга по данным, также обеспечивая двух, трех или четырех задачный режим.

7) Разработаны методы и средства аппаратной поддержки алгоритма БПФ по основанию 2 с числом входных отсчетов от 8 до 1024, а также алгоритмы их функционирования. Были введены специальные блоки памяти для хранения поворотных коэффициентов и входных отсчетов сигнала, которые позволяют хранить промежуточные данные вычислений (in-place реализация) и формировать на выходе самодостаточные данные. Также была введена специальная многоцикловая инструкция, реализация которой обеспечивает четырех задачный режим функционирования вычислительных ядер на всем протяжении вычисления БПФ. Степень использования параллелизма уровня инструкций составила 87,5%.

8) Разработаны ключевые элементы методологии программирования и отладки капсул для ГАРОС. Представленные методология, методики, алгоритмы и технология обеспечивают четко выстроенный итеративный процесс разработки капсул, который позволяет снизить требования к уровню знаний программиста об устройстве архитектуры. Сформирован перечень инструментальных средств, которые необходимо разработать, чтобы обеспечить наиболее полную поддержку представленных методов.

9) Разработаны программная и аппаратная поведенческие модели ГАРОС и набор средств аппаратно-программного моделирования и отладки. Данные средства позволили в автоматизированном режиме осуществить верификацию и валидацию моделей. Результаты верификации и валидации подтвердили бит-экзектность моделей. На основе аппаратной модели был синтезирован ПЛИС прототип ГАРОС.

10) Реализована демонстрационная задача распознавания изолированных слов в виде набора капсул и программы управляющего уровня на языке Си. Модели ГАРОС и ее ПЛИС прототип прошли успешную апробацию на данной задаче, подтвердив корректность своего функционирования. Средства аппаратно-программного моделирования были использованы для автоматического прогона полного комплекта тестовых воздействий (в виде 100 произношений слов в различных уровнях шума) на программной и аппаратной моделях, а также и ПЛИС прототипе. Результаты испытаний подтвердили идентичность всех моделей и прототипа.

11) Была осуществлена оценка производительности в циклах ПЛИС прототипа относительно микроконтроллера dsPIC30F компании MicroChip, которая показала высокий потенциал производительности ГАРОС, но и выявила ряд проблем полученной реализации.

12) Реализован комплект типовых алгоритмов ЦСП, которые используются в одном из наиболее распространенных бенчмарков ЦСП – BDTI Benchmark Kernels. Полученные

модельные оценки производительности были использованы для проведения сравнительного анализа ГАРОС и современного ЦСП серии TMS C55x. Результаты свидетельствуют о сопоставимом уровне производительности указанных вычислительных систем. Это означает, что ГАРОС имеет дальнейший потенциал развития и применения для решения задач ЦОС.

Реализованные в МПРА и ее прототипе ГАРОС принципы самодостаточности и рекуррентности являются уникальными. Автору не известны другие вычислительные системы, которые бы реализовывали нечто подобное. Кроме того, результаты диссертационной работы показали, что применение данных принципов позволило разработать действительно эффективную потоковую параллельную вычислительную систему.

К сожалению, автору не удалось приблизиться к самосинхронному исполнению ГАРОС на аппаратном уровне в силу необходимости использования для этого зарубежного синхронного ПЛИС, который не поддерживает асинхронность в полной мере. Тем не менее, удалось реализовать асинхронность на логическом (программном) уровне. Кроме того, в рамках выполнения исследования соискатель внес определенный вклад в решение проблем самосинхронного исполнения ГАРОС став соавтором 5 патентов РФ на изобретение в области самосинхронного исполнения аппаратуры.

Дальнейшее развитие МПРА и ее прототипа ГАРОС видится в рамках направлений:

1) Разработка семейства специализированных устройств преобразования тегов, что в перспективе позволит удлинить цепочки рекуррентных преобразований и добиться большей степени сжатия тегированных данных.

2) Размещение устройств преобразования тегов на более ранних стадиях конвейера ГАРОС. Данное решение позволить повысить степень наполнения конвейера ГАРОС инструкциями и повысить его производительность на последовательных фрагментах алгоритма.

3) Разработка механизма объединения капсул. В текущей версии «общение» между капсулами осуществляется при помощи управляющего уровня, что приводит к дополнительным временным издержкам и снижению производительности. Устранение посредника позволить существенно сократить накладные расхода. Кроме того, такая концепция хорошо сочетается с концепцией крупнозернистого параллелизма уровня капсул.

4) Разработка дополнительных элементов методологии программирования и отладки, таких как грамматика капсульного языка программирования, грамматика языка описания капсул высокого уровня, методы трансляции языка высокого уровня в капсулы. В совокупности данные средства позволят разработать средства компиляции и приблизят процесс программирования ГАРОС к более понятному функциональному стилю программирования.

5) Самосинхронное исполнение прототипа ГАРОС на аппаратном уровне, что должно существенно повысить уровень параллелизма и надежности.

Список сокращений и условных обозначений

ЦОС	–	Цифровая обработка сигналов
ЦСП	–	Цифровой сигнальный процессор
БПФ	–	Быстрое преобразование Фурье
МПРА	–	Многоядерная потоковая рекуррентная архитектура
ГАРОС	–	Гибридная архитектура рекуррентного обработчика сигналов
РОУ	–	Рекуррентное операционное устройство
УУ	–	Управляющий уровень
БП	–	Буферная память
ПТ	–	Преобразователь тегов
ЭСД	–	Элемент самодостаточных данных
РИС	–	Распознавание изолированных слов
ПО	–	Программное обеспечение
ПЛИС	–	Программируемая логическая интегральная схема
АЛУ	–	Арифметико-логическое устройство
АУ	–	Арифметическое устройство
IPC	–	Instruction per cycle – Инструкций в цикл
FIFO	–	First In First Out
ПФН	–	Потоково-фон-Неймановский
ПАП	–	Память адресной проверки
MAC	–	Multiply Accumulate
СП	–	Степень параллелизма
ГСП	–	Глобальная степень параллелизма
ПК_Г	–	Память констант Глобальная
ПК_С	–	Память констант Секционная
ПК_СР	–	Память констант Секционная Регистровая
ПК_СП	–	Память констант Секционная Подгружаемая
МКР	–	Маска косвенной репликации
ППВС	–	Параллельная потоковая вычислительная система
НД	–	Набор данных
ILP	–	Instruction Level Parallelism
VHDL	–	Very High-Speed Integrated Circuit Hardware Description Language
ПК	–	Программный комплекс
АГК	–	Автоматическая граф-капсула

Список литературы

1. Gill S. Parallel programming. — The Computer J. V.l, No 1, 1958, p.2-10.
2. Воеводин В.В. Математические проблемы параллельных вычислений. [Электронный ресурс] https://parallel.ru/sites/default/files/info/voevodin.doc (дата обращения 24.07.2023).
3. Михайлов Б.М., Халабия Р.Ф. Классификация и организация вычислительных систем. Учебное пособие. - М.: МГУПИ. 2010. - 144 с.
4. Воеводин В.В., Воеводин Вл. В Параллельные вычисления. – СПБ.: БХВ-Петербург, 2002. – С. 78-93.
5. Калайда В.Т. Теория вычислительных процессов. Методическое пособие для студен-тов. – 2007. – С. 106-109.
6. Дэвид М. Харрис, Сара Л. Харрис Цифровая схемотехника и архитектура компьютера. / пер. с англ. Imagination Technologies. – М.: ДМК Пресс, 2018. – 792 с.: цв. ил.
7. Kocher, Paul; Genkin, Daniel; Gruss, Daniel; Haas, Werner; Hamburg, Mike; Lipp, Moritz; Mangard, Stefan; Prescher, Thomas; Schwarz, Michael; Yarom, Yuval "Spectre Attacks: Exploiting Speculative Execution" (2018). [Электронный ресурс] https://spectreattack.com/spectre.pdf (дата обращения 10.01.2022)
8. Schlansker and Rau EPIC: An Architecture for Instruction-Level Parallel Processors. HP Laboratories Palo Alto, HPL-1999-111 (February 2000). [Электронный ресурс] https://www.hpl.hp.com/techreports/1999/HPL-1999-111.pdf (дата обращения: 06.01.2022).
9. Jack L. Lo, Joel S. Emer, Henry M. Levy, Rebecca L. Stamm, Dean M. Tullsen, and S. J. Eggers. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. ACM Trans. Comput. Syst. 15, 3 (Aug. 1997), 322–354.
10. T. Agerwala and Arvind, Data flow systems, IEEE Computer, 15 (Feb. 1982), pp. 10_13.
11. Arvind and D.E. Culler, Data flow architectures, Ann. Review in Comput. Sci., 1 (1986), pp. 225-253.
12. Jack B. Dennis, The variaties of data flow computers, in Proc. 1st Intl. Conf. Distr. Comput. Syst., Oct. 1979, pp. 430-439.
13. J.-L. Gaudiot and L. Bic, Advanced topics in dataflow computing, Prentice Hall, 1991.
14. Arvind and R.A. Iannucci, A critique of multiprocessing von Neumann style, in Proc. 10th ISCA, June 1983, pp. 426-436.
15. Dennis J.B. First version of a Data Flow Procedure Language. Proceedings of the Colloque sur la Programmation, Vol. 19, Lecture Notes in Computer Science, Springer-Verlag, 1974, pp. 362-376.
16. Jaffe J.M. The Equivalence of R. E. Programs and Data Flow Schemes. TM-121, Laboratory for Computer Science, MIT, Cambridge, MA, January, 1979.

17. Ben Lee, A.R. Hurson, Issues in Dataflow Computing, Editor(s): Marshall C. Yovits, Advances in Computers, Elsevier, Volume 37, 1993, Pages 285-333.
18. Polychronopoulos, C. D. and Banerjee, U., “Processor Allocation for Horizontal and Vertical Parallelism and Related Speedup Bounds,” *IEEE Transactions on Computers*, Vol. C-36, No. 4, April 1987, pp. 410-420.
19. Gaudiot, J.L. and Wei, Y. H., “Token Relabeling in a Tagged Token Data-Flow Architecture,” *IEEE Transactions on Computers*, Vol. 38. No. 9, Sept. 1989, pp. 1225-1239.
20. Ackerman, W. B., “A Structure Processing Facility for Dataflow Computers,” *Proc. Of the International Conference on Parallel Processing*, Aug. 1978, pp. 166-172.
21. Arvind and Thomas, “I-Structers: An Efficient Datatype for Functional Languages,” *MIT Laboratory for Computer Science Technical Report TM-178*, Cambridge, Mass., Sept. 1980.
22. Arvind, Nikhil, R. S., and Pingali, K. K., “I-structures: Data Structures for Parallel Computing,” *Proceedings of the Workshop on Graph Reduction*, Los Alamos, NM, 1986.
23. Patnaik, L. M., Govindarajan, R., and Ramados, N. S., “Design and Performance Evaluation of EXMAN: An Extended MANchester Dataflow Computer,” *IEEE Transactions on Computers*, Vol. C-35, No. 3, March 1986, pp. 229-243.
24. B. Lee, A. R. Hurson and B. Shirazi, "A hybrid scheme for processing data structures in a dataflow environment," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, no. 1, pp. 83-96, Jan. 1992.
25. Hwang, K. and Briggs, F. A., Computer Architecture and Parallel Processing. New York: McGraw-Hill, 1981.
26. P.C.Treleaven, R.P.Hopkins, P.W.Rautenbach. Combining Data Flow and Control Flow Computing. The Computer Journal, v. 25, N 2, 1982, p. 207-217.
27. R.P.Hopkins et al. A Computer supporting Data Flow, Control Flow and Updateable Memory. Technical Report 144, Computing Laboratory, The University of Newcastle upon Tyne (September 1979).
28. L.Bic, M.D.Nagel, and J.M.A.Roy. On Array Partitioning in PODS. Advanced Topics in Dataflow Computing, ed. L.Bic and J.L.Gaudiot, Prentice Hall, 1991.
29. G.R.Gao, R.Tio, and H.J.Hum. Design of an Efficient Dataflow Architecture Without Dataflow. Proc. of the International Conf. on Fifth Generation Computers, Tokyo, Japan, December 1988, p. 861- 868.
30. Arvind. The evolution of dataflow architecture from static dataflow to P-RISC. Proc. of Workshop on Massive Parallelism: Hardware, Programming and Appication, Amalfi, Italy, October 1989, Academic Press, 1990.

31. R.S.Nikhil and Arvind. Can dataflow subsume von Neumann computing? 16th Annual International Symposium on Computer Architecture, Jerusalem, May 24--June 1 1989, p. 262-272.
32. Anant Agarwal, Ben-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. Proc. 17th Annual Intl. Symp. on Computer Architecture, Seattle, Washington, U.S.A., May 28-31, p. 104-114.
33. Rishiyur S. Nikhil and Arvind. Can Dataflow Subsume von Neumann Computing? 16th International Symposium on Computer Architecture, May 1989.
34. P.Aarnio, I.Hartimo, K.Kronlof, O.Simula, and J.Skytta. On the use of the data flow princi-ple in digital signal processing. Proc. ECCTD'8], The Hague, The Netherlands, Aug. 25-28, 1981, p. 453-458.
35. K.Kronlof, J.Skytta, 0.Simula, and I.Hartimo. Simulation of a digital signal processing architecture based on the data flow principle. Proc. ISCAS'82, Rome, Italy, May 10-12, 1982, p. 1053-1056.
36. K.Kronlof, I.Hartimo, and O.Simula. On the VLSI implementation of a data flow signal processor. Proc. ICCC'82, New York, NY, Sept. 28-Oct.l, 1982, p. 594-597.
37. K.Kronlof, 0.Simula and J.Skytta. DFSP: A Data flow Signal Processor. IEEE Translations on Computers, v. C-35, N 1, January, p. 23-33.
38. P.Evripidou and J-L.Gaudiot. Input/Output Operations for Hybrid Data-flow/Control-Flow Systems. IEEE, 1991, p. 318-323.
39. Рождественский Ю.В., Дьяченко Ю.Г. Оценка фундаментальной параллельности в системах обработки голосовых сигналов // Системы и средства информатики, вып. 12. М.: ИПИ РАН, 2002. - С. 250–254.
40. Н.В. Морозов, Ю.А. Степченков. Средства моделирования и анализа параллельных процессов в рекуррентном операционном устройстве // Системы и средства информатики. Вып. 12. – М.: «Наука», 2002. – С. 255 – 266.
41. Филин А.В. Динамический подход к выбору архитектуры вычислительных устройств обработки сигналов // Системы и средства информатики: Вып. 11 – М.:Наука, 2001. – С. 247-261.
42. Степченков Ю.А., Петрухин В.С., Филин А.В. Рекуррентное операционное устройство для процессоров обработки сигналов // Системы и средства информатики. Вып. 11. М.: Наука, 2001. – С. 283-315.
43. Степченков Ю.А., Петрухин В.С. Перспективы развития цифровых, сигнальных процессоров и возможная реализация рекуррентного обработчика сигналов / Специальный выпуск «Методы и средства разработки информационно-вычислительных систем и сетей». – М.: ИПИ РАН, 2004. – С. 92-140.

44. Степченков Ю.А., Петрухин В.С. Особенности гибридного варианта реализации на ПЛИС рекуррентного обработчика сигналов // Системы и средства информатики: Доп. Вып. – М.: ИПИ РАН, 2008. – С. 118-129.
45. Исследование новой вычислительной парадигмы и разработка на её основе логического проекта динамического многопоточного процессора обработки сигналов. Отчет о НИР (заключительный), книга 1, шифр "Сигнал", № г.р. 01.2.00 104927 – М.: ИПИ РАН, 2003 – 126 с.
46. Исследования в области проектирования рекуррентных обработчиков сигналов с программируемой архитектурой. Отчет о НИР, шифр “ГАРОС”, № г.р. 01.2.007 03035 – М.: ИПИ РАН, 2009, Книга 1 — С.14-73.
47. Роберт В. Себеста Основные концепции языков программирования — 5-е изд. — М.: «Вильямс», 2001. — С. 672.
48. А. Филд, П. Харрисон Функциональное программирование: Пер. с англ. — М.: Мир, 1993. — 637 с.
49. Степченков Ю.А., Петрухин В.С., Хилько Д.В. Выбор языковых средств представления параллельных алгоритмов для рекуррентного обработчика сигналов. Системы и средства информатики. 2008. Доп. Вып. С. 149-158.
50. Зеленов Р.А., Степченков Ю.А., Волчек В.Н., Хилько Д.В., Шнейдер А.Ю., Прокофьев А.А. Система капсульного программирования и отладки. Системы и средства информатики. 2010. Т. 20. № 1. С. 24-30.
51. Хилько Д.В., Степченков Ю.А. Вопросы программируемости многоядерной вычислительной архитектуры с единым потоком для эффективной реализации рекуррентных вычислений. Сборник статей региональной научно-практической конференции «Многоядерные процессоры и параллельное программирование». 2011. С. 98-104.
52. Хилько Д.В. Постановка задачи программирования многоядерной потоковой рекуррентной архитектуры. Вторая школа молодых ученых ИПИ РАН. Сборник докладов. - М: ИПИ РАН, 2011. С. 72-80.
53. Хилько Д.В. Средства программирования нетрадиционной многоядерной архитектуры и перспективы их развития. Сборник статей II региональной научно-практической конференции «Многоядерные процессоры и параллельное программирование». 2012. С.62-70.
54. Gaudiot J.-L., BohmW., Najjar W., DeBoni T., Feo J., Miller P. The Sisal Model of Functional Programming and its Implementation // Proceedings of the Second Aizu International

Symposiumon Parallel Algorithms/Architectures Synthesis (pAs ’97), Aizu-Wakamatsu, Japan, March 17-21, 1997.

55. Исследование новой вычислительной парадигмы и разработка на её основе логического проекта динамического многопоточного процессора обработки сигналов. Шифр: "Сигнал". № г. р. 01.2.00.104927. Отчет о НИР (промежуточный за этап 2), книга 1. – М.: ИПИ РАН, 2002, 156 с.
56. Мультиклет. История компании. [Электронный ресурс] http://multiclet.com/index.php/ru/company/history (дата обращения 24.07.2023).
57. В. Овчинников. Мультиклеточные процессоры – новое поколение вычислительных устройств // Компонены и технологии, №6, 2011. СпБ: Файнстрит. С. 70 – 73.
58. ОАО Мультиклет. Мультиклеточная архитектура: особенности, реализация и перспективы развития. Екатиренбург 2014. [Электронный ресурс] http://multiclet.com/docs/PO/Мультиклеточная%20архитектура%20особенности,%20реализация%20и%20перспективы%20развития.pdf (дата обращения 24.07.2023).
59. Стемпковский А.Л., Левченко Н.Н., Окунев А.С., Цветков В.В. Параллельная потоковая вычислительная система – дальнейшее развитие архитектуры и структурной организации вычислительной системы с автоматическим распределением ресурсов // журнал «ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ», №10, 2008, С.2 – 7.
60. Климов А.В., Левченко Н.Н., Окунев А.С., Стемпковский А.Л. Вопросы применения и реализации потоковой модели вычислений // Проблемы разработки перспективных микро- и наноэлектронных систем - 2016. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2016. Часть II. С. 100-106.
61. Змеев Д.Н., Климов А.В., Левченко Н.Н. Средства распределения вычислений в ППВС «Буран» и варианты реализации блока выработки хэш-функций // Проблемы разработки перспективных микро- и наноэлектронных систем - 2016. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2016. Часть II. С. 107-113.
62. Ричард Лайонс. Цифровая обработка сигналов: Второе издание. Пер. с англ. – М.: ООО «Бином-Пресс», 2006 г. – 656 с.: ил.
63. Cooley, J. and Tukey, J. «An Algorithm for the Machine Calculation of Complex Fourier Series», Math. Comput., Vol. 19, No. 90, Apr. 1965, pp. 297-301.
64. Bahtat, M., Belkouch, S., Elleaume, P. et al. Instruction scheduling heuristic for an efficient FFT in VLIW processors with balanced resource usage. EURASIP J. Adv. Signal Process. 2016, 38 (2016). https://doi.org/10.1186/s13634-016-0336-0
65. Texas Instruments. Mark McKeown. FFT Implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs. Application report SPRABB6BB–June 2010–

Revised January 2013. [Электронный ресурс] https://www.ti.com/lit/an/sprabb6b/sprabb6b.pdf (дата обращения 28.07.2022).

66. Степченков Ю.А., Дьяченко Ю.Г., Хилько Д.В., Петрухин В.С. Рекуррентная потоковая архитектура: особенности и проблемы реализации // Проблемы разработки перспективных микро- и наноэлектронных систем — 2016. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2016. Часть 2. С. 120-127.
67. Д.В. Хилько, Ю.А. Степченков, Д.И. Шикунов, Ю.И. Шикунов. Рекуррентная потоковая архитектура: технические аспекты реализации и результаты моделирования // Проблемы разработки перспективных микро- и наноэлектронных систем – 2016. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2016. Часть II. С. 128-135.
68. Yu. A. Stepchenkov, Yu. G. Diachenko, D. V. Khilko, V.S. Petrukhin. Recurrent data-flow architecture: features and realization problems // Problems of Advanced Micro- and Nanoelectronic Systems Development, 2017, Part II, Moscow, IPPM RAS, P. 52-58.
69. D.V. Khilko, Yu. A. Stepchenkov, D. I. Shikunov, Yu. I. Shikunov. Recurrent data-flow architecture: technical aspects of implementation and modeling results // Problems of Advanced Micro- and Nanoelectronic Systems Development, 2017, Part II, Moscow, IPPM RAS, P. 59-64.
70. Yury A. Stepchenkov, Dmitry V. Khilko, Yury I. Shikunov, Georgy A. Orlov. DSP Filter Kernels Preliminary Benchmarking for Recurrent Data-flow Architecture // 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) St. Petersburg, Moscow, Russia, January 26-29, 2021. — IEEE, P. 2040-2044.
71. Хилько Д.В., Степченков Ю.А., Шикунов Ю.И., Дьяченко Ю.Г., Орлов Г.А. Оптимизация аппаратной поддержки быстрого преобразования Фурье в рекуррентном сигнальном процессоре // Системы и средства информатики, 2021. Т. 31. № 4. С. 71-83.
72. Yury Stepchenkov, Dmitry Khilko, Yury Shikunov, Georgy Orlov. Optimizing Data-flow Processor Architecture for Efficient Implementation of DSP Algorithms // 2022 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) St. Petersburg, Moscow, Russia, January 25-28, 2022. — IEEE, 5 P.
73. Lavagno L., Martin G., Markov I.L., Scheffer L.K. Electronic Design Automation for IC System Design, Verification, and Testing. CRC Press; 2nd edition, 2016 - 664 p.
74. Mixed-signal and DSP Design Techniques, ed. by W. Kester, Analog Devices Inc., 2003, P.410.
75. Xilinx Logic Core. Fast Fourier Transform. LogiCORE IP Product Guide. Vivado Design Suite. PG109 August 6, 2021. Version 9.1. Xilinx. [Электронный ресурс]

https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_1/pg109-xfft.pdf (дата обращения 28.07.2022).

76. Rozita Teymourzadeh. High-Resolution Single-Chip Radix II FFT Processor for High-Tech Application. Fourier Transforms -High-tech Application and Current Trends, 2017, 10.5772/66745. hal-01802070.
77. Отработка системы программирования многоядерных потоковых рекуррентных компьютерных систем предметной области: Отчет о НИР (заключительный), Шифр «КАПСУЛА2», № г.р. 01201368527. М.: ИПИ РАН, 2013, 34 С.
78. Д. В. Хилько, Ю. А. Степченков. Теоретические аспекты разработки методологии программирования рекуррентной архитектуры // Системы и средства информатики, – М.: ТОРУС ПРЕСС, Т. 23, № 2, 2013 – С. 133-153.
79. Д. В. Хилько, Ю. А. Степченков. Модель потоковой архитектуры на примере распознавателя слов устройства // Системы и средства информатики, – М.: ТОРУС ПРЕСС, Т. 22, № 2, 2012 – С. 48-57.
80. Хилько Д.В., Шикунов Ю.И. Разработка инструментальной среды проектирования программного обеспечения для рекуррентно-потоковой модели вычислений // Четвертая школа молодых ученых ИПИ РАН. Сборник докладов – М.: ИПИ РАН, 2013 – С. 65-78.
81. Хилько Д.В., Степченков Ю.А., Шикунов Ю.И. Средства имитационного моделирования многоядерной потоковой рекуррентной архитектуры // Сборник статей II всероссийской научно-практической конференции “Многоядерные процессоры, параллельное программирование, ПЛИС, системы обработки сигналов” Барнаул, 28 февраля 2014 г. С. 58−69.
82. Хилько Д.В., Шикунов Ю.И., Степченков Ю.А. Особенности программной реализации имитационной модели потоковой рекуррентной архитектуры // Труды Второй молодежной научной конференции «Задачи современной информатики» – М.: ФИЦ ИУ РАН, 2015. – С. 220-227.
83. Д. В. Хилько, Ю. А. Степченков, Ю. Г. Дьяченко, Ю. И. Шикунов, Н. В. Морозов. Аппаратно-программное моделирование и тестирование рекуррентного операционного устройства // Системы и средства информатики, – М.: ТОРУС ПРЕСС, Т. 25, № 4, 2015 – С. 78-90.
84. Yuri Shikunov, Dmitry Khilko, Yuri Stepchenkov. Hardware and Software Modelling and Testing of Non-Conventional Data-Flow Architecture // 2016 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) St. Petersburg, Russia, February 02-03, 2016. — IEEE, P 360-364.

85. Yuri Stepchenkov, Dmitry Khilko, Yuri Diachenko, Yury Shikunov and Dmitry Shikunov. Testing of Software and hardware testing of dataflow recurrent digital signal processor // Proceedings of IEEE East-West Design & Test Symposium (EWDTS'2016), Yerevan, October, 14 — 17, 2016. P. 168-171.
86. Yury Shikunov, Yury Stepchenkov, Dmitry Khilko, Dmitry Shikunov. Data redundancy problems in data-flow computing and solutions implemented on the recurrent architecture // 2017 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) St. Petersburg, Russia, 1-3 Feb., 2017. — IEEE, P. 335 — 338.
87. Yu. Shikunov, Yu. Stepchenkov, D. Khilko. Recurrent mechanism developments in the data-flow computer architecture //2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) Moscow, Russia, 29 Jan.-1 Feb., 2018. — IEEE, P. 1413 – 1418.
88. Д.В. Хилько, Ю.А. Степченков, Ю.И. Шикунов, Г.А. Орлов. Развитие средств капсульного программирования потоковой рекуррентной архитектуры // Проблемы разработки перспективных микро- и наноэлектронных систем – 2018. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2018. Часть III. С. 2–9.
89. Dmitry Khilko, Yuri Stepchenkov, Yury Shikunov and George Orlov. Modeling and debugging tools development for recurrent architecture // 2019 IEEE EAST-WEST DESIGN & TEST SYMPOSIUM Batumi, Georgia, September 13 — 16, 2019.
90. D.V. Khilko, Yu. A. Stepchenkov, Yu.I. Shikunov, G.A. Orlov. Development of Capsule Programming Means for Recurrent Data-flow Architecture // Problems of Advanced Micro- and Nanoelektronic Systems Development – 2019, Issue II, Moscow, IPPM RAS, P. 40-45.
91. Yury A. Stepchenkov, Dmitry V. Khilko, Yury I. Shikunov, Georgii A. Orlov. Iterator component development for data redundancy solution in data-flow architecture // 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)Moscow, Russia, January 27-30, 2020. — IEEE, P. 1869-1872.
92. Степченков Ю.А., Хилько Д.В., Шикунов Ю.И., Орлов Г.А. Специализированные преобразователи тегов для рекуррентного обработчика сигналов // Проблемы разработки перспективных микро- и наноэлектронных систем — 2020. Сборник трудов / под общ. ред. академика РАН А.Л. Стемпковского. М.: ИППМ РАН, 2020. Выпуск 2. С. 73-80.
93. Государственная регистрация программы для ЭВМ № 2015614004 от 01.04.2015 (опубликовано 20.04.2015). Инструментальная среда проектирования ПО для гибридной архитектуры рекуррентного обработчика сигналов (GAROS IDE). Хилько Дмитрий Владимирович, Степченков Юрий Афанасьевич, Шикунов Юрий Игоревич; заявитель

Федеральное государственное бюджетное учреждение науки Институт проблем информатики Российской академии наук (ИПИ РАН). - № заявки 2015610945, дата поступления заявки 17.02.2015.

94. Государственная регистрация программы для ЭВМ № 2019665933 от 03.12.2019 Бюл. №12. Инструментальная среда разработки HARSP IDE. Хилько Дмитрий Владимирович, Шикунов Юрий Игоревич, Орлов Георгий Александрович, Степченков Юрий Афанасьевич; заявитель Федеральное государственное учреждение «Федеральный исследовательский центр «Информатика и управление» Российской академии наук» (ФИЦ ИУ РАН). № заявки 2019664931, дата поступления заявки 21.11.2019.
95. Государственная регистрация программы для ЭВМ № 2021668788 от 19.11.2021 Бюл. №11. Инструментальная среда разработки HARSP IDE. Версия 2. Хилько Дмитрий Владимирович, Шикунов Юрий Игоревич, Орлов Георгий Александрович, Степченков Юрий Афанасьевич; заявитель Федеральное государственное учреждение «Федеральный исследовательский центр «Информатика и управление» Российской академии наук» (ФИЦ ИУ РАН). № заявки 2021668056, дата поступления заявки 12.11.2021.
96. Государственная регистрация программы для ЭВМ № 2022667594 от 22.09.2022 Бюл. No 10. Программный комплекс моделирования потоковой рекуррентной многоядерной вычислительной системы (ПК ПОТОК). Хилько Д.В., Шикунов Ю.И., Орлов Г. А., Степченков Ю.А.; заявитель и правообладатель ФИЦ ИУ РАН. № заявки 2022667012, дата поступления заявки 16.09.2022.
97. Свидетельство № 2013610200 Российская Федерация. Программа обработки результатов моделирования потоковой рекуррентной архитектуры (ПРАПОР); свидетельство об официальной регистрации программы для ЭВМ / Хилько Д.В., Степченков Ю.А., Шнейдер А.Ю.; заявитель и правообладатель ИПИ РАН. - №; дата регистрации 09.01.2013 г.
98. Свидетельство № 2013610199 Российская Федерация. Средства имитационного моделирования потоковой рекуррентной архитектуры (СИМПРА); свидетельство об официальной регистрации программы для ЭВМ / Хилько Д.В., Степченков Ю.А.; заявитель и правообладатель ИПИ РАН. - №; дата регистрации 09.01.2013 г.
99. Государственная регистрация программы для ЭВМ № 2014610123 от 09.01.2014 (опубликовано 20.02.2014). Средства имитационного моделирования потоковой рекуррентной архитектуры (СИМПРА). Версия 2. Хилько Дмитрий Владимирович, Степченков Юрий Афанасьевич, Шикунов Юрий Игоревич, Дьяченко Юрий Георгиевич; заявитель Федеральное государственное бюджетное учреждение науки

Институт проблем информатики Российской академии наук (ИПИ РАН).- № заявки 2013619997, дата поступления заявки 01.11.2013.

100. Свидетельство № 2010610715 Российская Федерация. Система капсульного программирования и отладки (СКАТ); свидетельство об официальной регистрации программы для ЭВМ / Зеленов Р.А., Степченков Ю.А., Волчек В.Н., Петрухин В.С., Прокофьев А.А., Хилько Д.В.; заявитель и правообладатель ИПИ РАН.; дата регистрации 20.01.2010.

101. Свидетельство № 2013610198 Российская Федерация. Система капсульного программирования и отладки (СКАТ). Версия 2; свидетельство об официальной регистрации программы для ЭВМ / Зеленов Р.А., Степченков Ю.А., Волчек В.Н., Петрухин В.С., Хилько Д.В.; заявитель и правообладатель ИПИ РАН. - №; дата регистрации 09.01.2013 г.

102. Graphviz - Graph Visualization Software. URL: http://www.graphviz.org/ (дата обращения 28.07.2022).

103. Yu. Shikunov, Yu. Stepchenkov, D. Khilko, G. Orlov. Graph-capsule construction toolset for data-flow computer architecture // 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) Moscow, Russia, 29 Jan.-1 Feb., 2018. — IEEE, P. 1419 — 1423.

104. Государственная регистрация программы для ЭВМ № 2018611551 от 02.02.2018 Бюл. №12. Программа автоматизированного построения граф-капсул (ГРАФ). Хилько Дмитрий Владимирович, Степченков Юрий Афанасьевич, Шикунов Юрий Игоревич, Орлов Георгий Александрович; заявитель Федеральное государственное учреждение «Федеральный исследовательский центр «Информатика и управление» Российской академии наук» (ФИЦ ИУ РАН). № заявки 2017662648, дата поступления заявки 06.12.2017.

105. Engel A. Verification, Validation, and Testing of Engineered Systems. Wiley, London, 2010.

106. Михайлов М., Грушвицкий Р., Проектирование в условиях временных ограничений: верификация проектов на основе ПЛИС. Часть 1 // Компоненты и Технологии, № 3, 2008. С. 96-102.

107. Foster H., Krolnic A., Lacey D. Assertion-Based Design. Kluwer Academic Publishers, 2003.

108. В.В. Кулямин. V.V. Методы верификации программного обеспечения. Институт системного программирования, Москва, 2008. 111 С.

109. Yury Stepchenkov, Dmitry Khilko, Yury Shikunov and Georgy Orlov. Design validation of recurrent signal processor FPGA prototype // Proceedings of IEEE East-West Design & Test Symposium (EWDTS'2021), Batumi, Georgia, September, 10 — 13, 2021, P. 157-161.

110. Построение компьютерных систем новых поколений на основе нетрадиционных подходов и архитектурных решений. Отчет о НИР (заключительный), шифр ГАРОС2/этап 2010, № г.р. 01200903872 – М.: ИПИ РАН, 2010. - 185 с.

111. Yury Stepchenkov, Nikolai Morozov, Dmitry Khilko, Yury Shikunov, Georgy Orlov. Hybrid multi-core recurrent architecture approbation on FPGA // 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) Moscow, Russia, January 28-31, 2019. — IEEE, P. 1705 — 1708.

112. Степченков Ю. А., Морозов Н. В., Дьяченко Ю. Г., Хилько Д. В., Степченков Д. Ю. Развитие гибридной многоядерной рекуррентной архитектуры на ПЛИС // Системы и средства информатики, 2020. Т. 30. № 4. С. 95-101.

113. Степченков Ю.А., Морозов Н.В., Дьяченко Ю.Г., Хилько Д.В. Аппаратная реализация рекуррентного обработчика сигналов // Системы и средства информатики, 2021. Т. 31. № 3. С. 113-122.

114. Дьяченко Ю.Г., Степченков Ю.А., Морозов Н.В., Хилько Д.В., Степченков Д.Ю., Шикунов Ю.И. Аппаратная верификация рекуррентного обработчика сигналов на ПЛИС // Проблемы разработки перспективных микро- и наноэлектронных систем (МЭС). 2021. Выпуск 2. С. 77-82.

115. Ю.А. Степченков, Н.В. Морозов, Ю.Г. Дьяченко, Д. В. Хилько, Д.Ю. Степченков, Ю. И. Шикунов. Аппаратная реализация алгоритмов цифровой обработки сигналов в рекуррентном потоковом процессоре на ПЛИС // М.: Известия вузов. Электроника. 2022. Том 27. № 3. С. 356-366. DOI: 10.24151/1561-5405-2022-27-3-356-366.

116. Yury A. Stepchenkov, Dmitry V. Khilko, Yury I. Shikunov, Georgy A. Orlov. DSP Filter Kernels Preliminary Benchmarking for Recurrent Data-flow Architecture // 2021 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus) St. Petersburg, Moscow, Russia, January 26-29, 2021. — IEEE, P. 2040-2044.

Приложение А
(справочное)

Таблица А.1 – Переработанная система команд

Символ	Код	Символ графа	Операция в Вычислителе
EBS	0x00		Отсутствие специфицированного кода операции
ADD	0x01	+	Сложение
ADC	0x02	+c	Сложение с учетом переноса
SUBi	0x03	-	Вычитание
SUBp	0x04	-p	Вычитание (уменьшаемое)
SBC	0x05	-c	Вычитание с учетом займа
MULuua	0x06	*ua	Умножение (беззнаковое на беззнаковое) результат в регистр A
MULssa	0x07	*sa	Умножение (знаковое на знаковое) результат в регистр A
MULuub	0x08	*ub	Умножение (беззнаковое на беззнаковое) результат в регистр B
MULuuc	0x09	*uc	Умножение (беззнаковое на беззнаковое) результат в регистр C
MULssb	0x0A	*sb	Умножение (знаковое на знаковое) результат в регистр B
MULssc	0x0B	*sc	Умножение (знаковое на знаковое) результат в регистр C
MACac	0x0C	*+c	Умножение с накоплением в регистре C
MACsc	0x0D	*-c	Умножение с вычитанием из регистра C
AND	0x0E	&	Конъюнкция
OR	0x0F	1	Дизъюнкция
NOT	0x10	~	Инвертирование
EDAC	0x11	edac	Примитив операции вычисления Евклидова расстояния Семантика - суперскалярная операция типа 1: 1) [C]=[A]*[A]+[C] 2) [A]= L-операнд - R-операнд
RESERVED	0x12	reserved	Резерв
ASR	0x13	a>	Арифметический сдвиг вправо на один разряд
ASL	0x14	a<	Арифметический сдвиг влево на один разряд
BUTT	0x15	><	Базовая операция БПФ
LMb	0x16	>db	Загрузка средней части регистра B MAC
LMc	0x17	>dc	Загрузка средней части регистра [C] MAC. Если операнд с данным кодом операции 38-разрядный, то регистр [C] загружается целиком
LMl	0x18	>dl	Загрузка средней части регистра [RL] MAC. Если операнд с данным кодом операции 38-разрядный, то регистр [RL] загружается целиком
LMr	0x19	>dr	Загрузка средней части регистра [RR] MAC. Если операнд с данным кодом операции 38-разрядный, то регистр [RR] загружается целиком
EXOP	0x1A	e	Сложная команда, в содержательной части которой указан код выполняемой операции
SQ	0x1B	*2	Возведение в квадрат
SQac	0x1C	*2+c	Возведение в квадрат с накоплением в регистре C
SQsc	0x1D	*2-c	Возведение в квадрат с вычитанием из регистра C
BSR	0x1E	*br	Округление/сдвиг, суперскалярная операция, определяемая конфигуратором секции
NOP	0x1F	n	Отсутствие операции. Устанавливается автоматически после сброса РОУ
			Значения EXOP-кодов операций
Jccz	0x00	^z	Переход если флаг Z=1
Jccc	0x01	^c	Переход если флаг C=1
If	0x02	<>	Выполнение цикла
FR	0x03	f0	Сброс флагов в "0"
FS	0x04	fs	Установка флагов в "1"
CLR_	0x05	c_	Очистка регистра (регистр задается в [10,8] разрядах)
MOVl_	0x06	<l_	Пересылка младшей части регистра (регистр задается в [10,8] разрядах)

Символ	Код	Символ графа	Операция в Вычислителе
MOVd_	0x07	<d_	Пересылка средней части регистра (регистр задается в [10,8] разрядах)
MOVh_	0x08	<h_	Пересылка старшей части регистра (регистр задается в [10,8] разрядах)
RND_	0x09	r_	Округление содержимого регистра и запись в целевой регистр (регистр округления задается в [10,8] разрядах, сдвиг задается в [7..3] разрядах, целевой регистр задается в [2..0] разрядах*)
MOV_	0x0A	<_	Считывание 38-разрядного значения регистра и запись в целевой регистр (регистр чтения задается в [10,8] разрядах, целевой регистр задается в [2..0] разрядах*)
SFTR_	0x0B	s_	Сдвиг регистра и запись в целевой регистр (регистр сдвига задается в [10,8] разрядах, длина и направление сдвига в [7..3] разрядах, целевой регистр задается в [2..0] разрядах*)
EXCR_	0x0C	ex_	Обмен значений между регистрами (регистры задаются в разрядах [10..8] и [2..0] соответственно)
MPR_	0x0D	mr_	Знаковое умножение регистров Разряд [10] – указатель части регистра источника левого операнда{0(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [9..7] регистр источник левого операнда Разряд [6] – указатель части регистра источника правого операнда(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [5..3] регистр источник правого операнда Разряды [2..0] регистр размещения результата
MACR_	0x0E	m+r_	Знаковое умножение с накоплением регистров Разряд [10] – указатель части регистра источника левого операнда(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [9..7] регистр источник левого операнда Разряд [6] – указатель части регистра источника правого операнда(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [5..3] регистр источник правого операнда Разряды [2..0] регистр накопления результата
MSCR_	0x0F	m-r_	Знаковое умножение с вычитанием регистров Разряд [10] – указатель части регистра источника левого операнда(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [9..7] регистр источник левого операнда Разряд [6] – указатель части регистра источника правого операнда(d) = читать среднюю часть; 1(l) = читать младшую часть} Разряды [5..3] регистр источник правого операнда Разряды [2..0] регистр вычитания результата
ADDR_	0x10	ar_	Сложение регистров Разряды [9..7] регистр источник левого операнда Разряды [5..3] регистр источник правого операнда Разряды [2..0] регистр размещения результата
SUBR_	0x11	sr_	Вычитание регистров Разряды [9..7] регистр источник левого операнда Разряды [5..3] регистр источник правого операнда Разряды [2..0] регистр размещения результата

Значения [10..8] разрядов содержательной части		
000	NON	Регистр не задан (по умолчанию для команд, не требующих работы с регистрами [A], [B], [C], [RL], [RR])
001	a	Регистр [A]
010	b	Регистр [B]
011	c	Регистр [C]
100	l	Регистр [RL]
101	r	Регистр [RR]

Алгоритмы функционирования Вычислителя

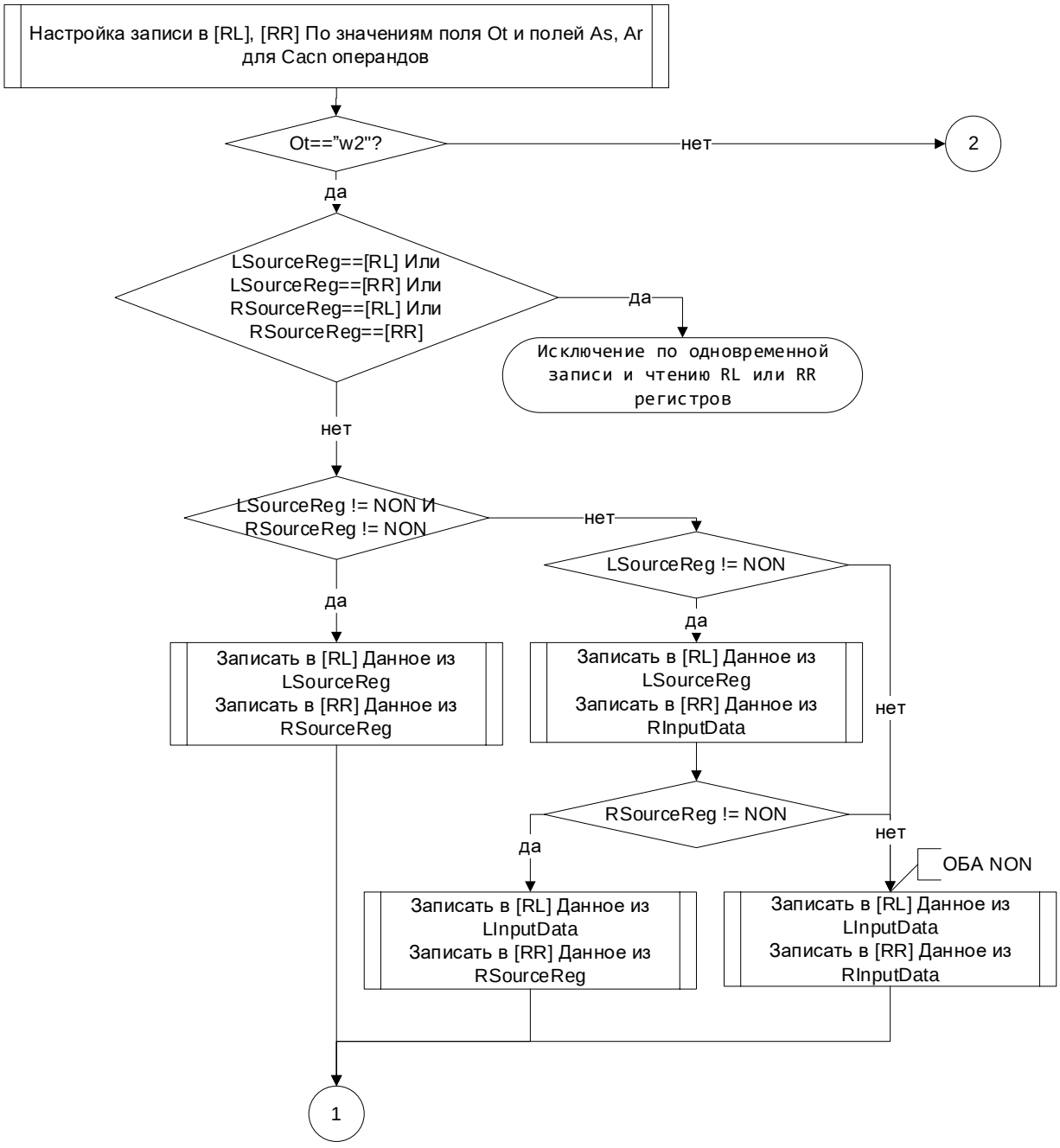


Рисунок А.1 – Алгоритм настройки записи в регистры [RL] и [RR]

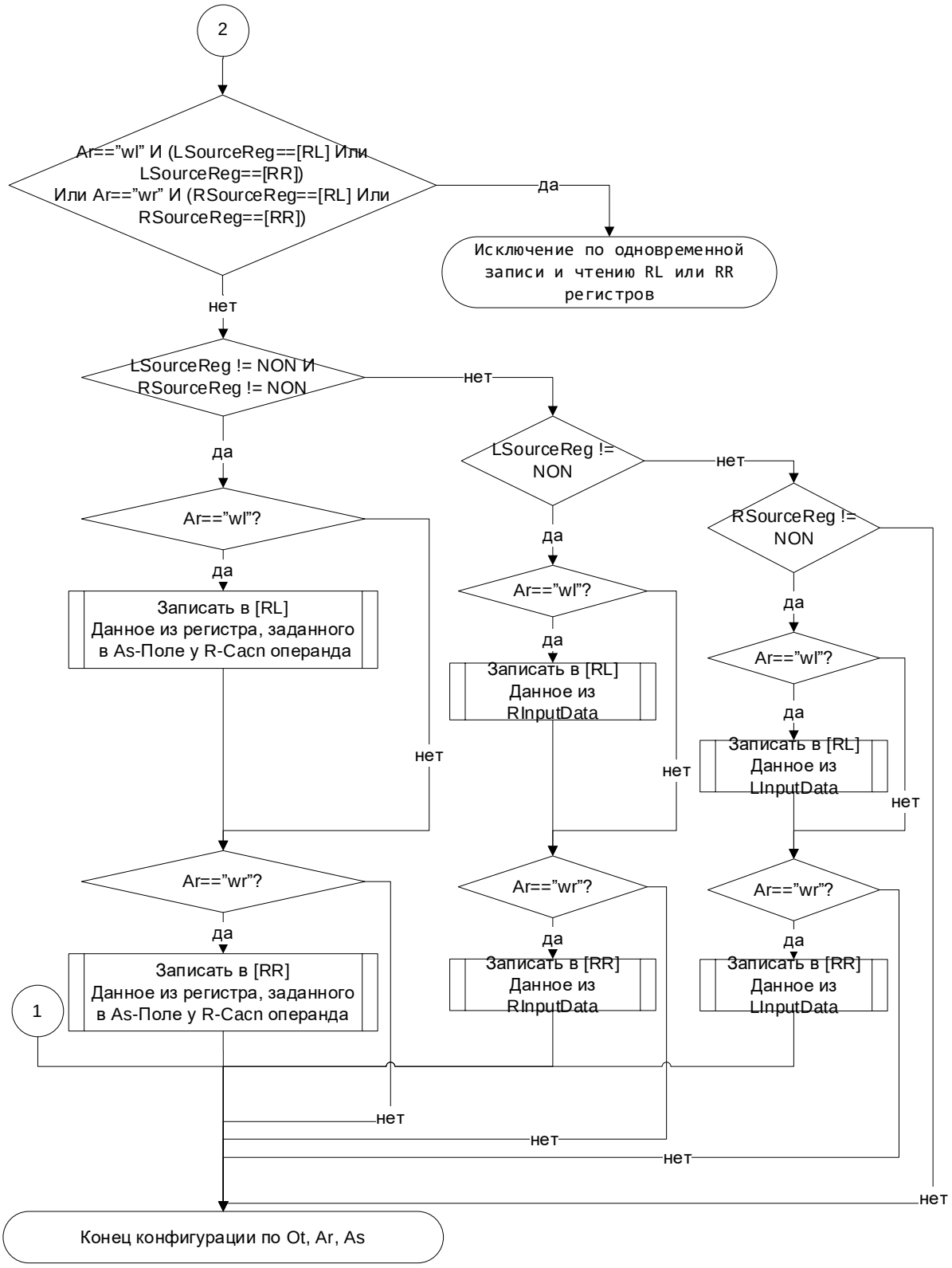


Рисунок А.1 (продолжение) – Алгоритм настройки записи в регистры [RL] и [RR]

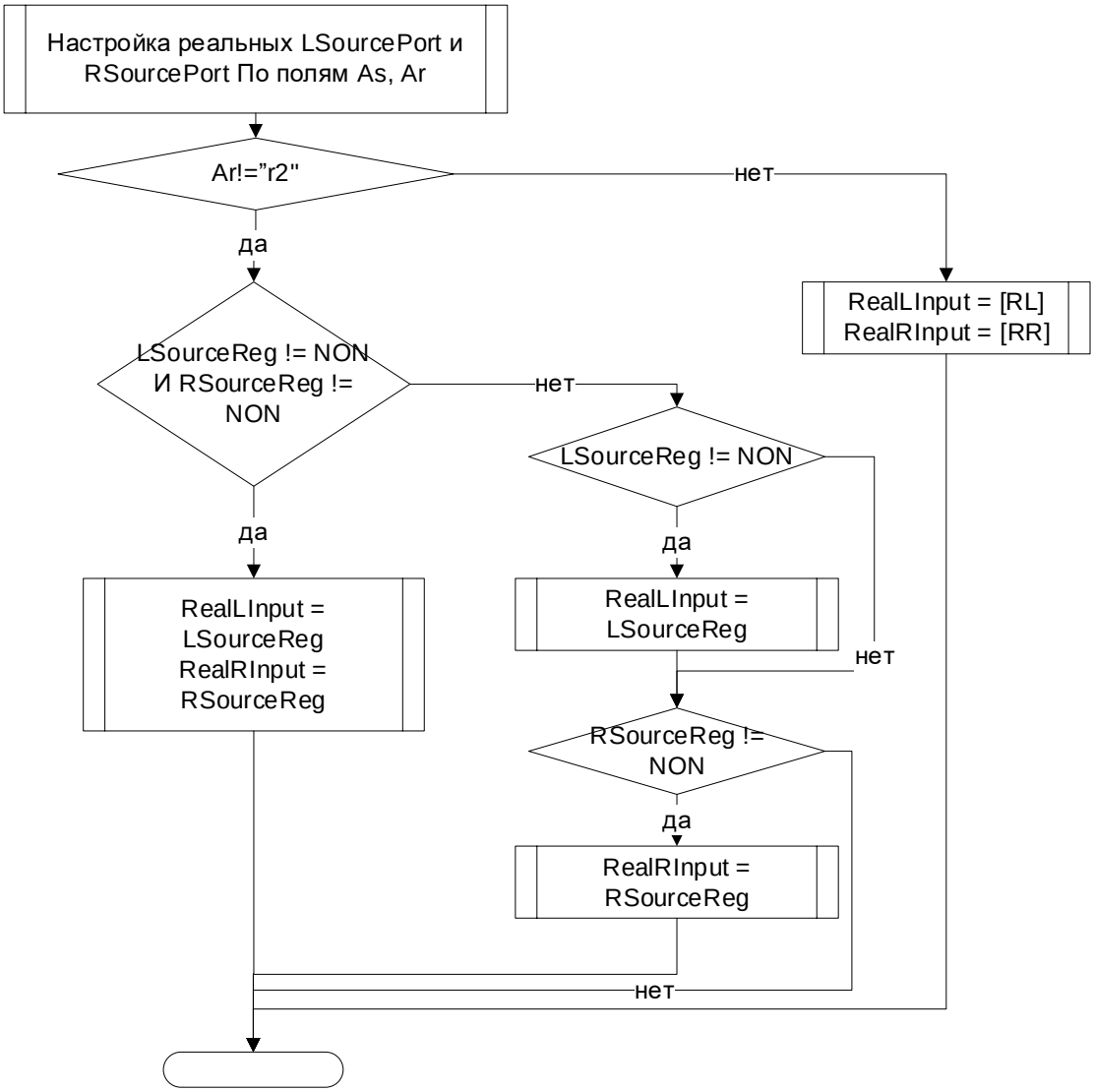


Рисунок А.2 – Алгоритм настройки источников данных для L- и R- входов

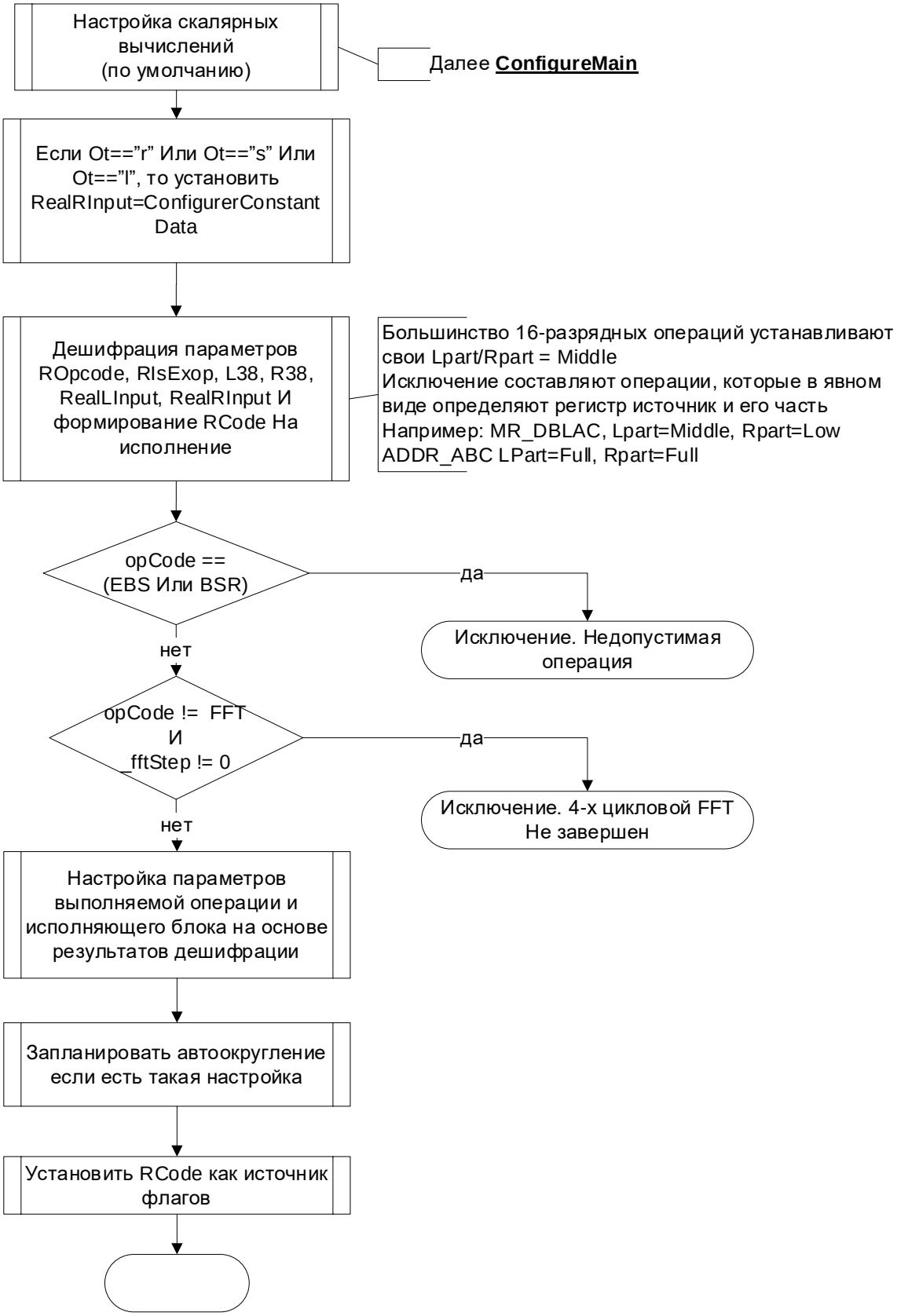


Рисунок А.3 – Алгоритм настройки однозадачных вычислений (по умолчанию)

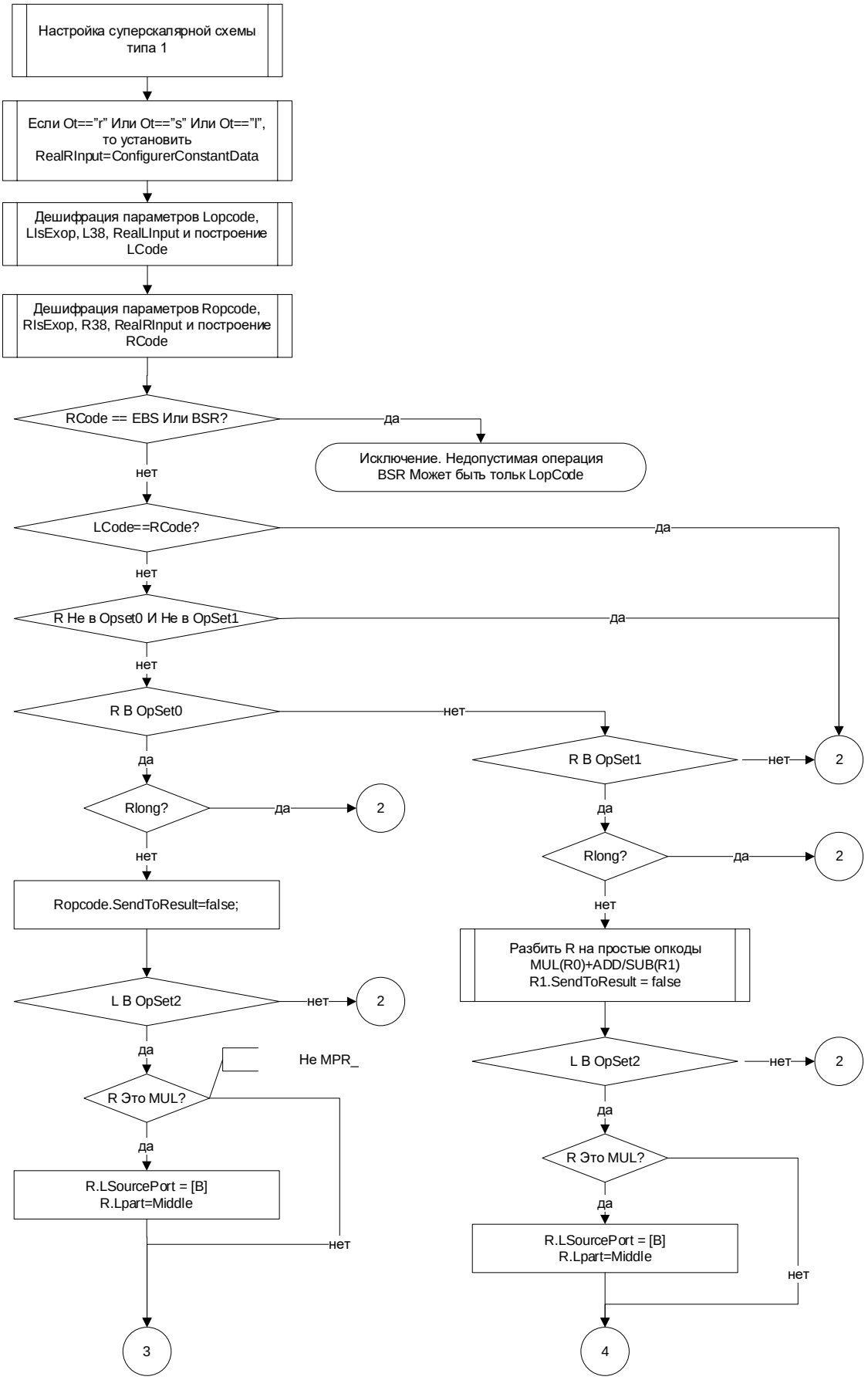


Рисунок А.4 – Алгоритм настройки суперскалярной схемы типа 1

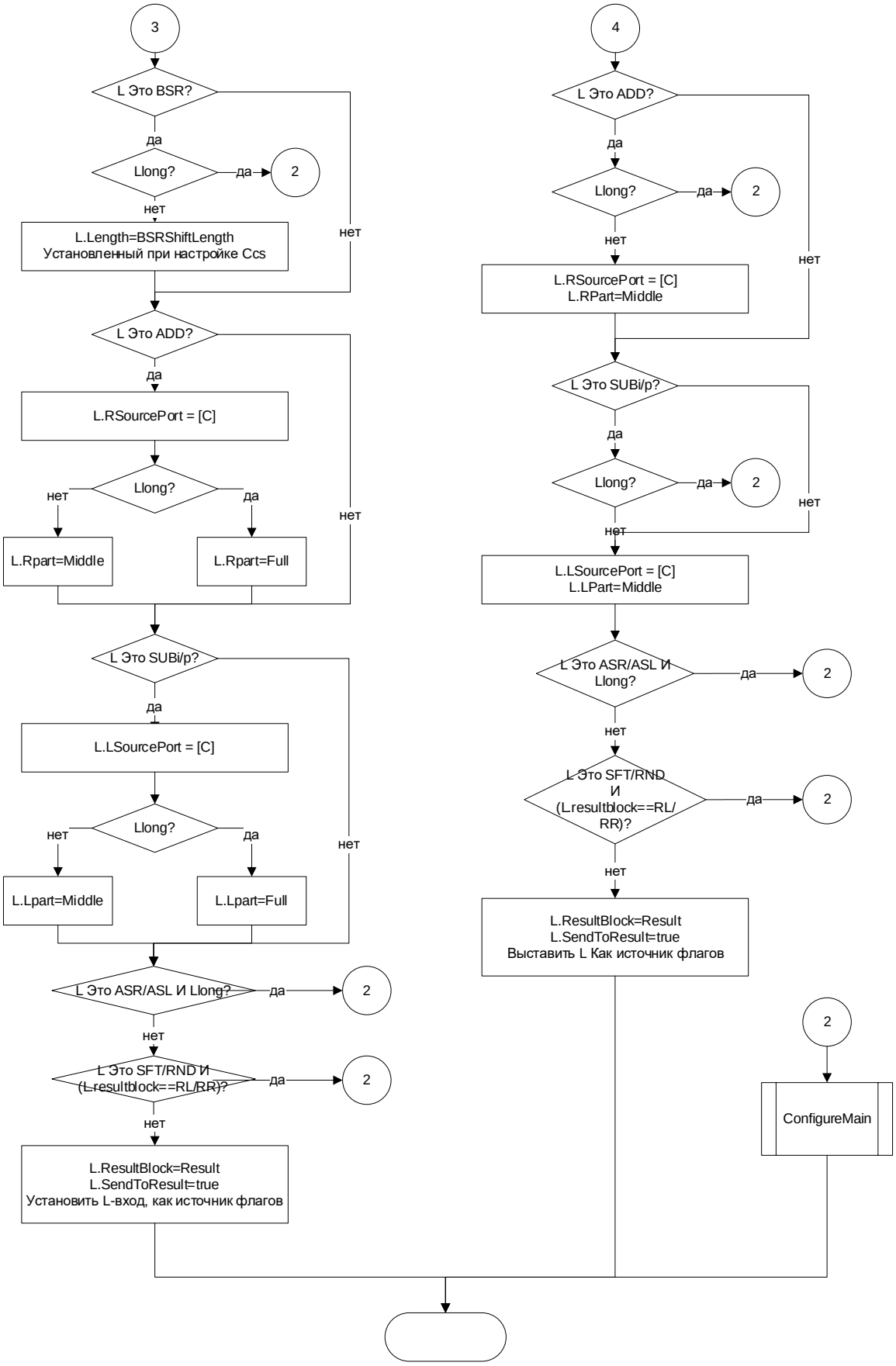


Рисунок А.4 (продолжение) – Алгоритм настройки суперскалярной схемы типа 1

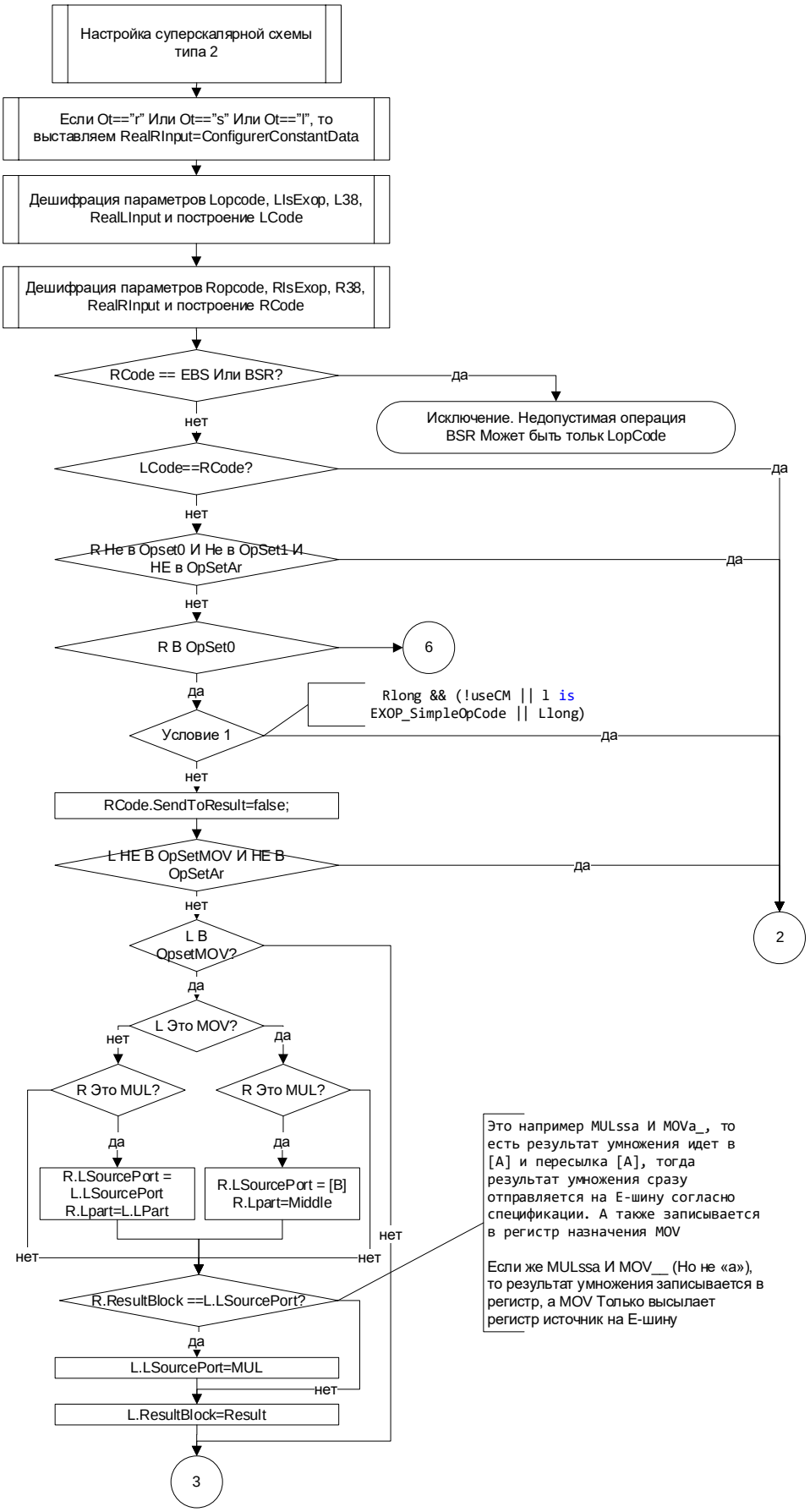


Рисунок А.5 – Алгоритм настройки суперскалярной схемы типа 2

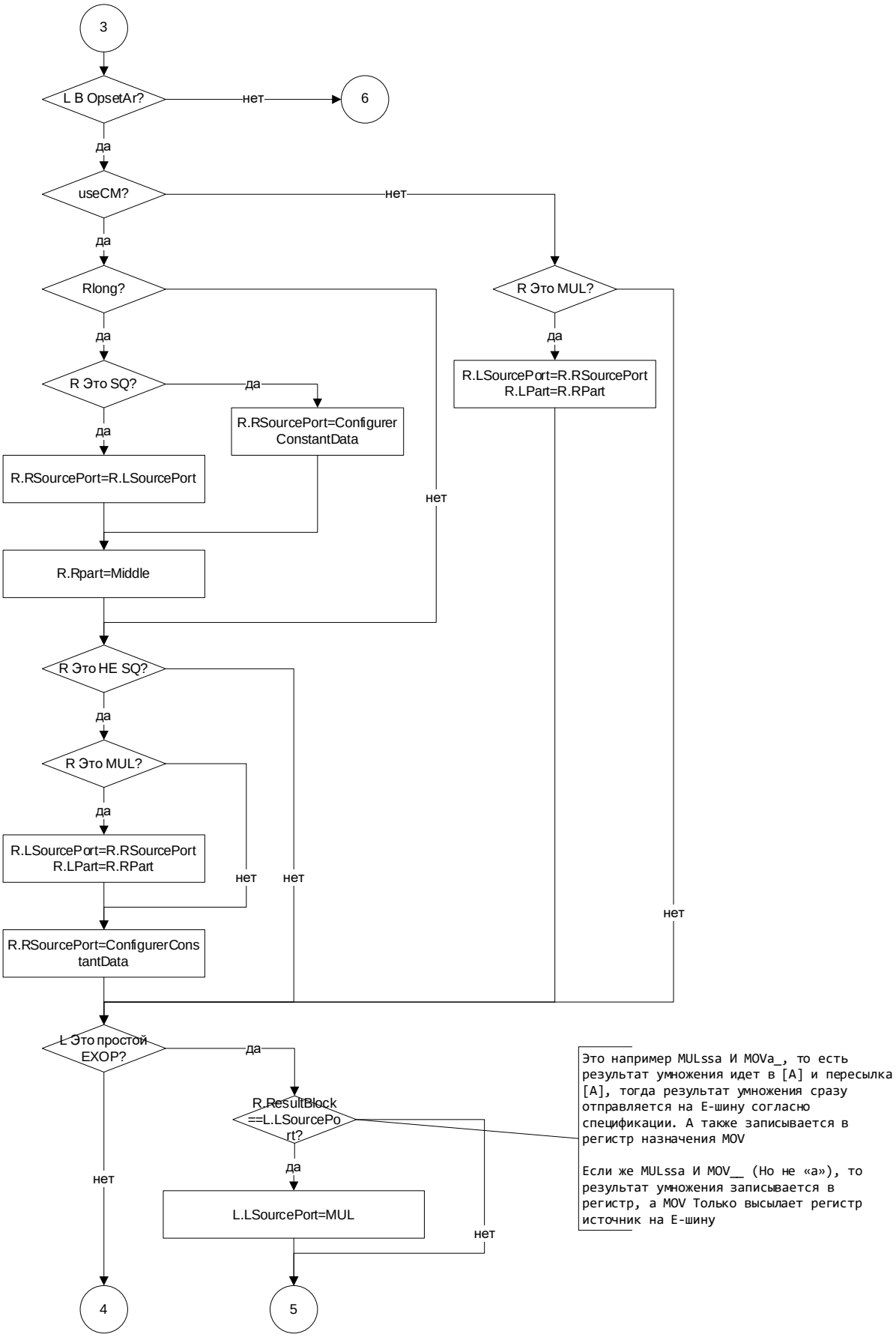


Рисунок А.5 (продолжение) – Алгоритм настройки суперскалярной схемы типа 2

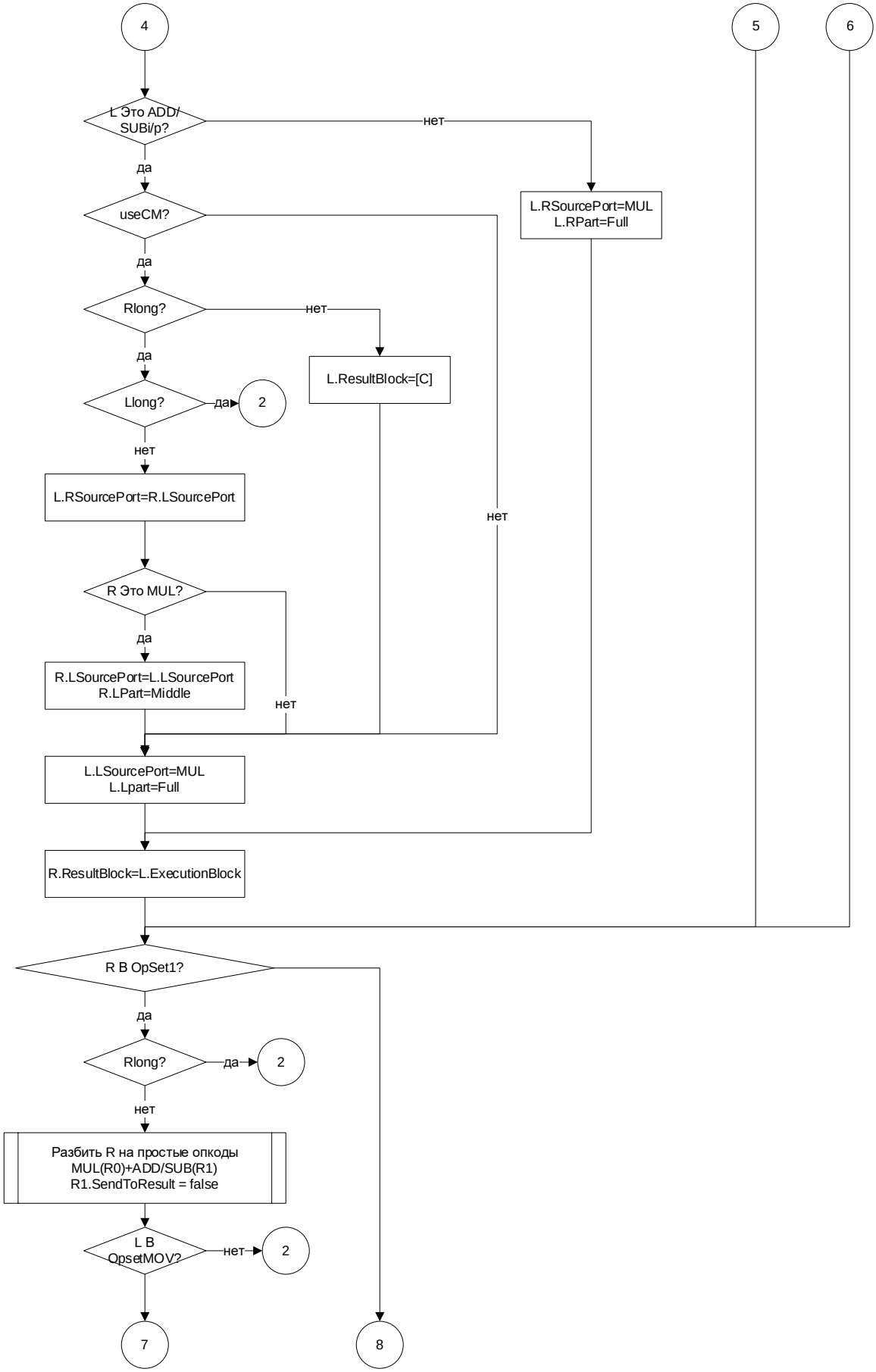


Рисунок А.5 (продолжение) – Алгоритм настройки суперскалярной схемы типа 2

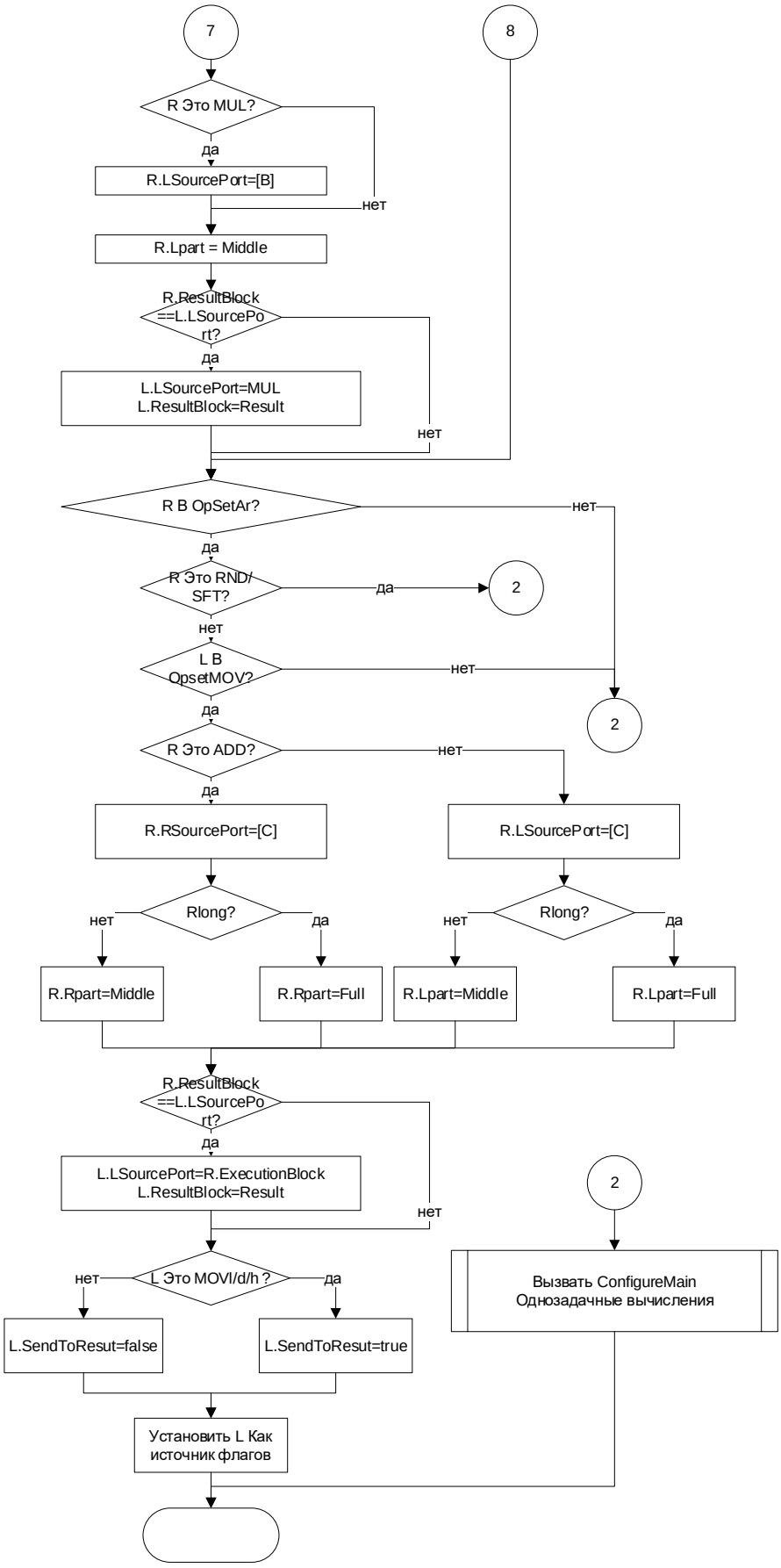


Рисунок А.5 (продолжение) – Алгоритм настройки суперскалярной схемы типа 2

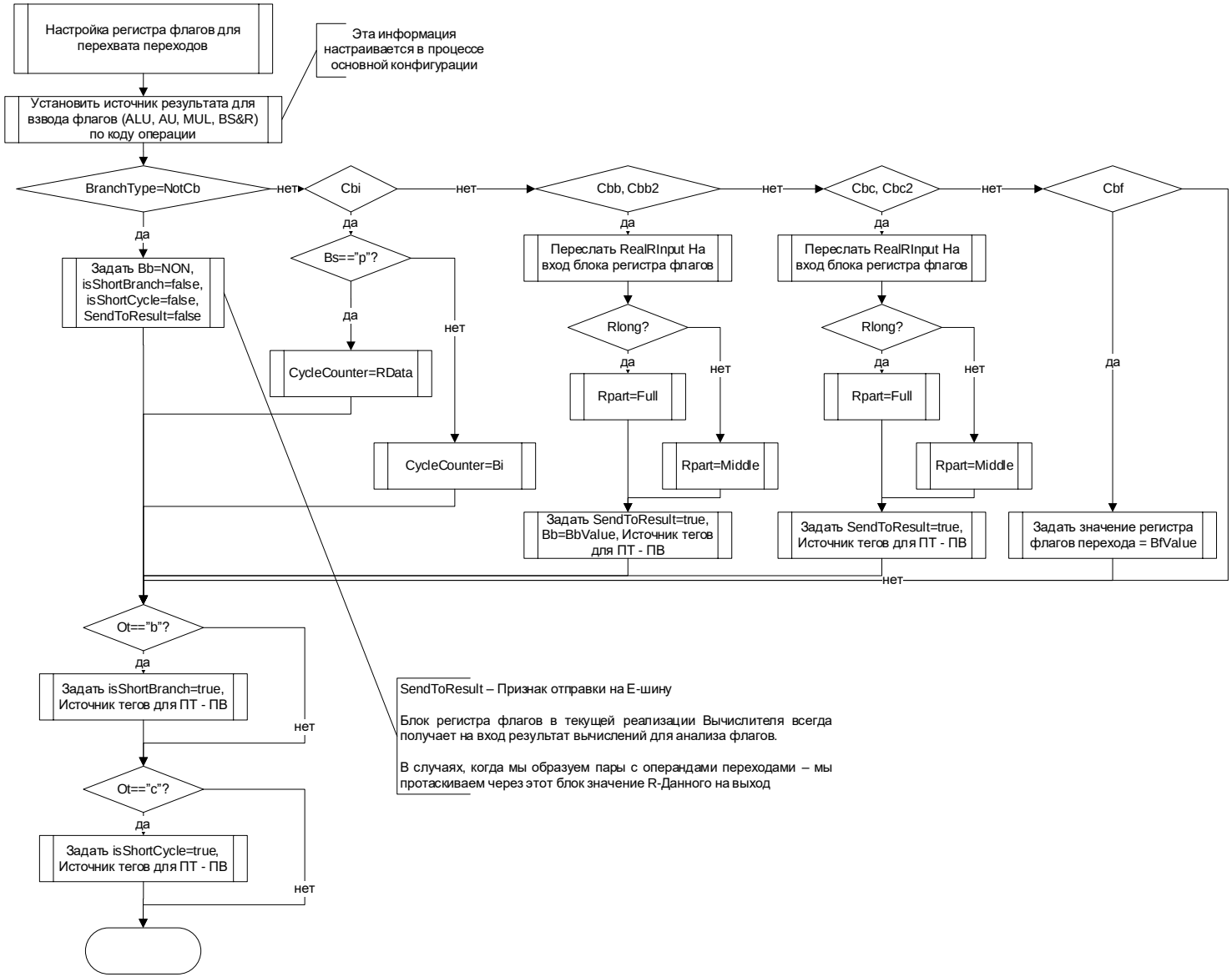


Рисунок А.6 – Алгоритм настройки режима перехвата и обработки переходов

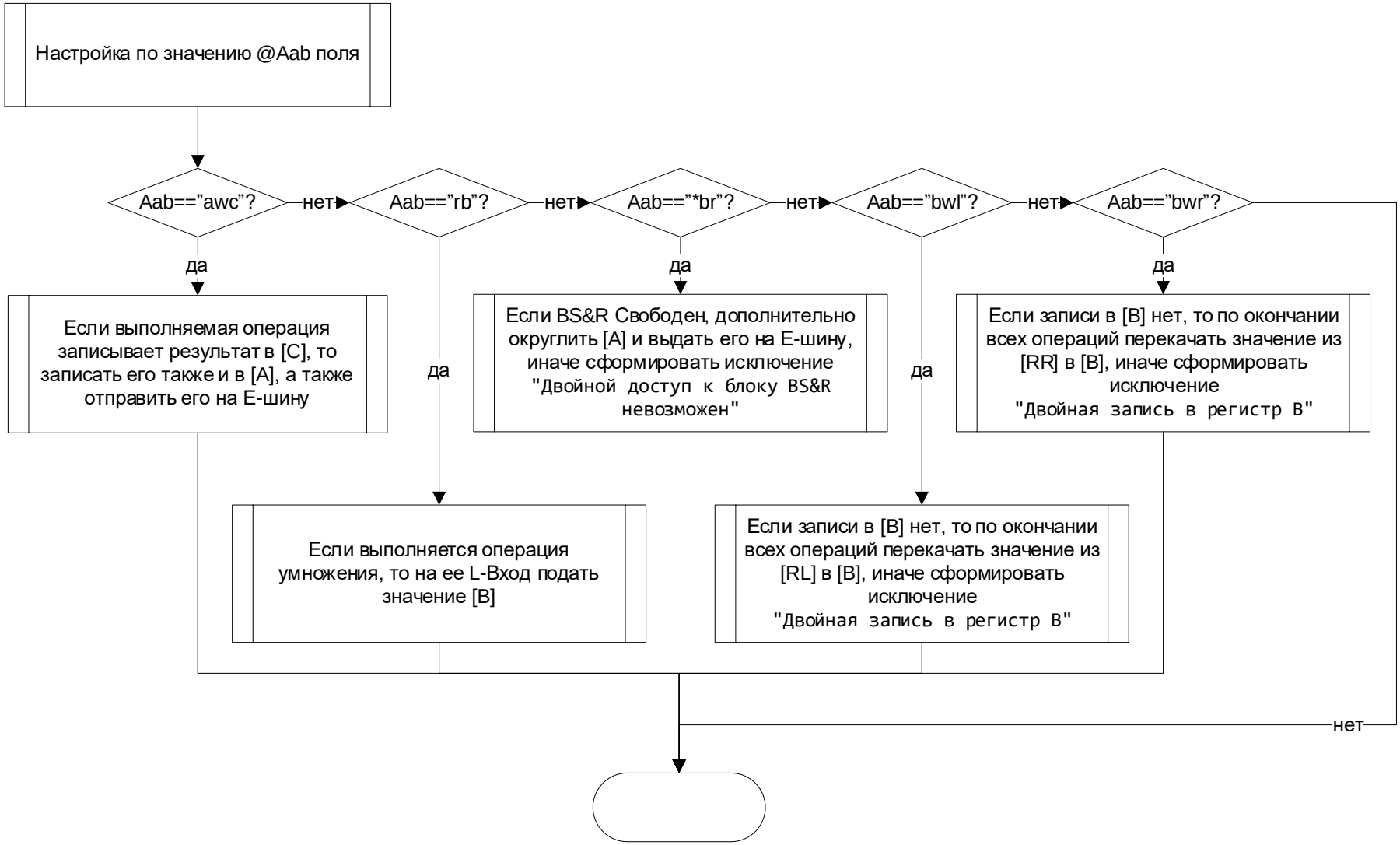


Рисунок А.7 – Алгоритм пост обработки результатов вычислений

Алгоритмы косвенной репликации

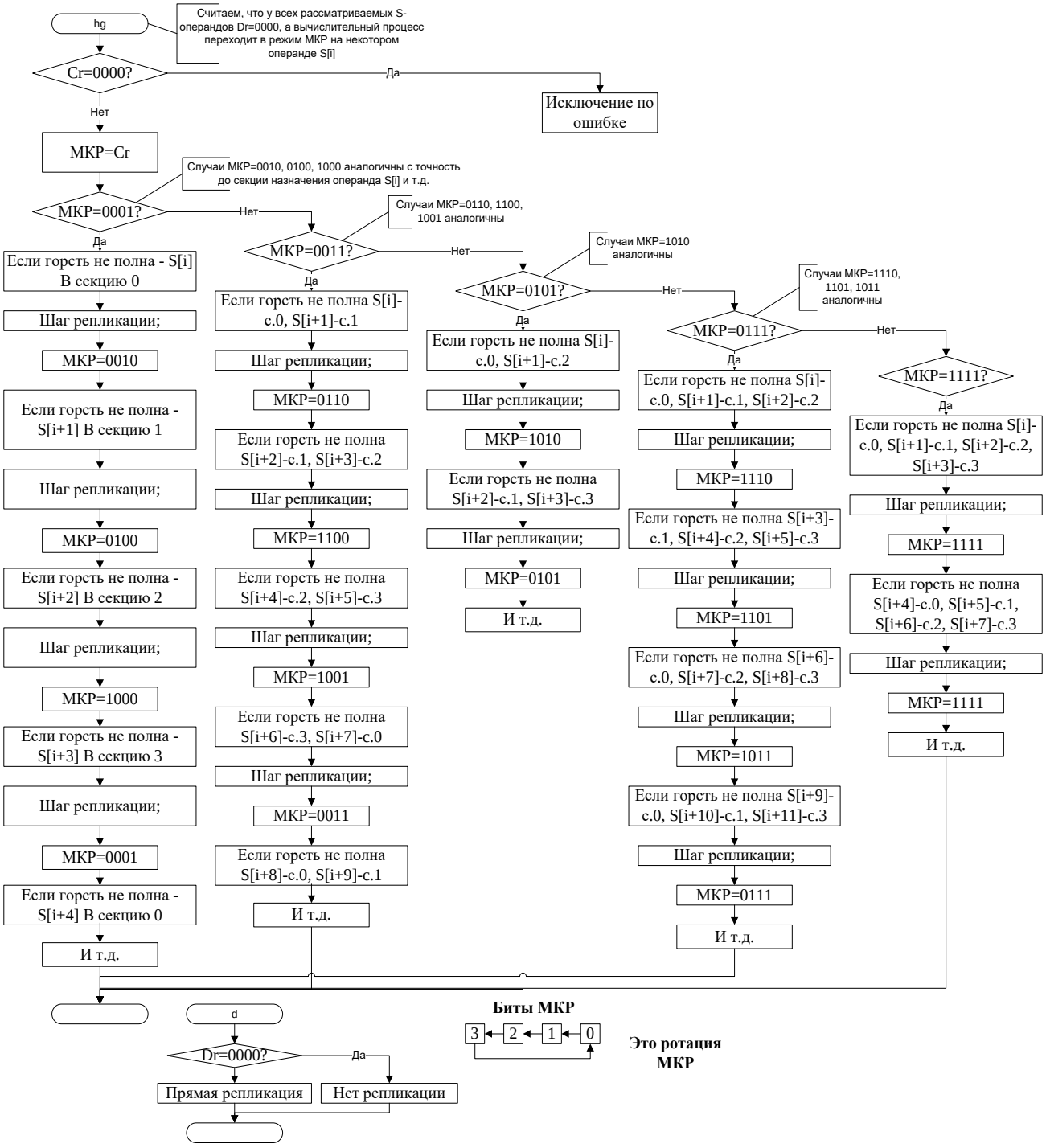


Рисунок А.8 – Алгоритм режима репликации hg

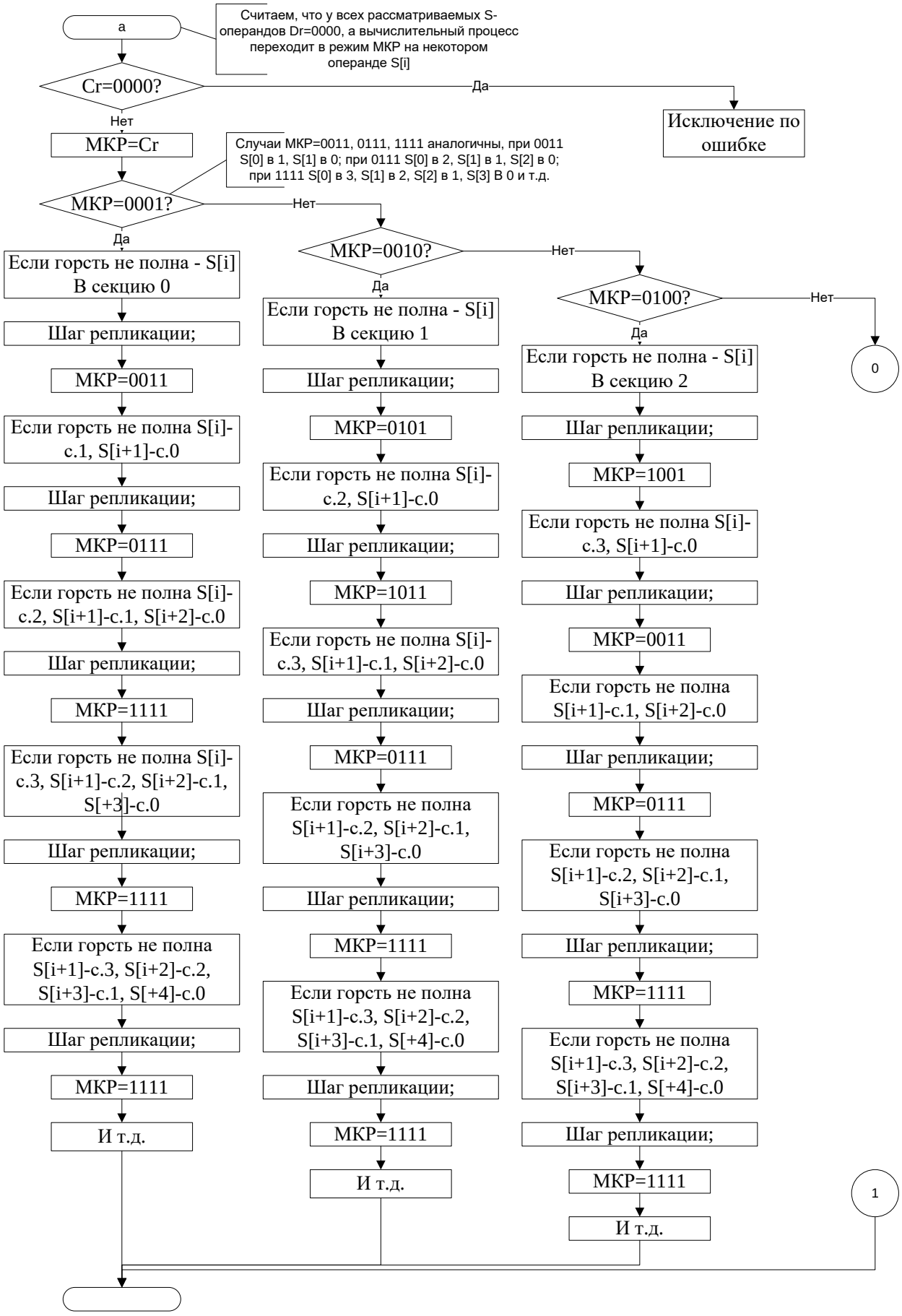


Рисунок А.9 – Алгоритм режима репликации ha

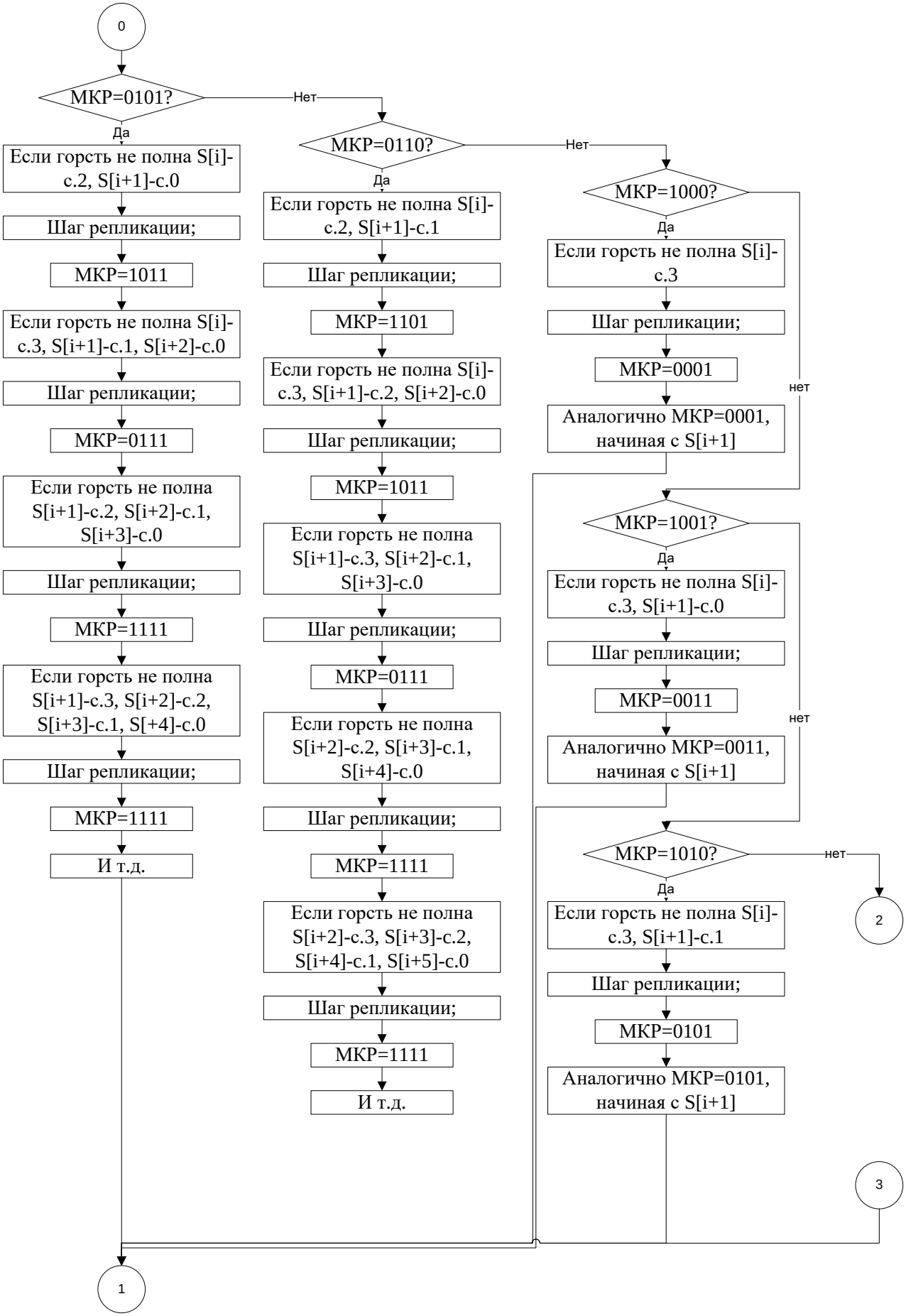


Рисунок А.9 (продолжение) – Алгоритм режима репликации ha

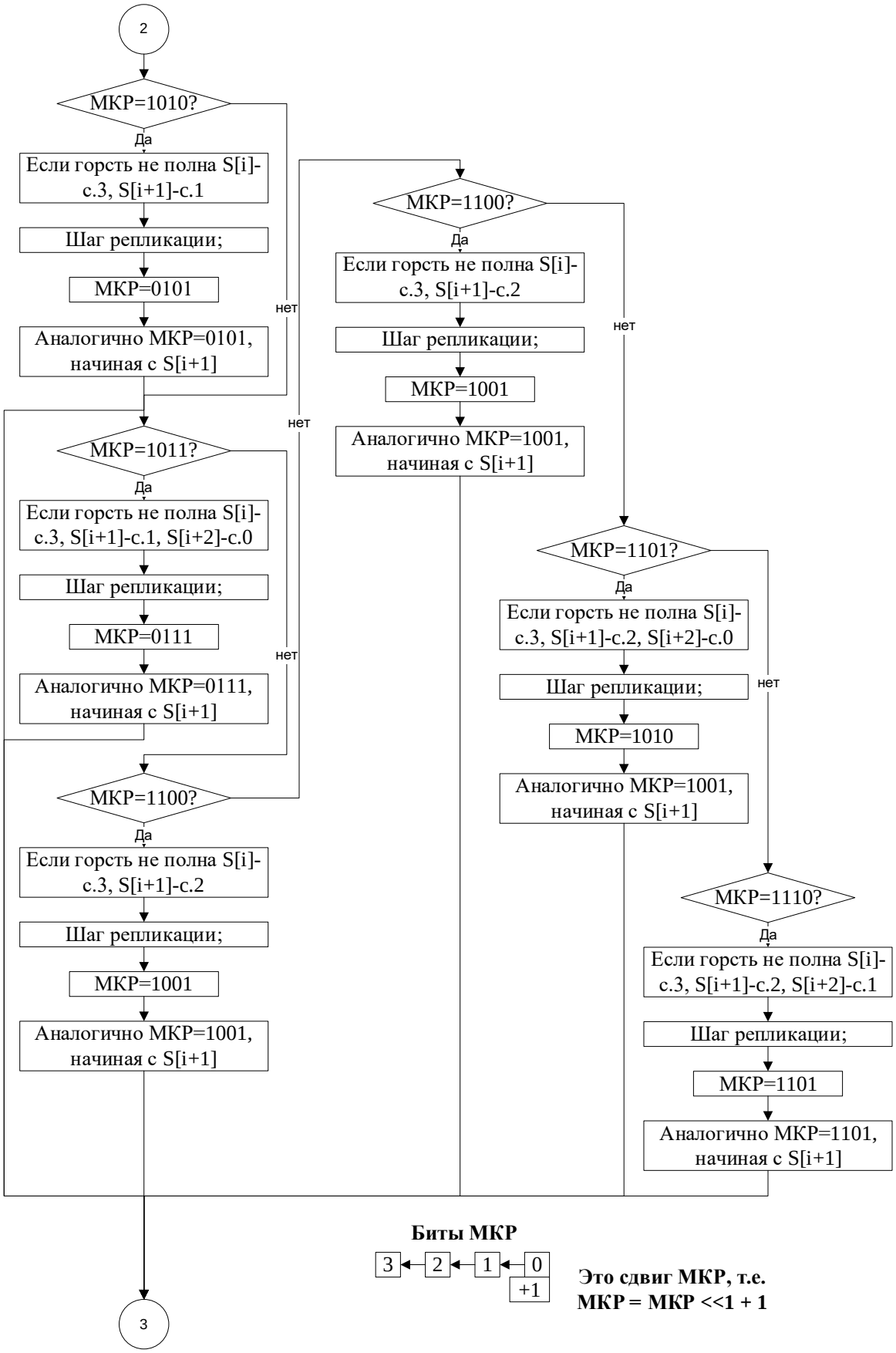


Рисунок А.9 (продолжение) – Алгоритм режима репликации ha

Таблица А.2– Обновленные форматы операндов

Тип	Операнд	Сигнатура
Ad	Терминатор НД	Без тела
Az	Терминатор капсулы	@t
Acg	Глобальный конфигуратор	%Cxpse_ca_cm_ct
Am	Расширенная маска репликации	%M_rf_rn_efn_rs_mowc
Acm	Конфигуратор многократного исполнения капсулы	%C_di_br_1m_0m_dm_1d_0d_1s_0s_i
Abm	Контроллер итерации капсулы	%C_di_dm_sh
Ai	Инициализатор выходного раздела	@Inistmadf
At_16	Шаблон для Di: операнда	%DrShmAt[ShmOuctDrse]{SmOucDs@s}
Asc	Стартер упакованных констант	@DrShmOuctDse@{SmOuOcDs@s}%Snl@s
At_38	Шаблон для Di_38: операнда	%DrShmAt[DrShmOucDs]
Asi	Стартер входных операндов	%Sa_nl_tShmOuctDes[ShmOuctDrse]{SmOucDs@s}
Afo	Финишер выходных операндов	Без тела
Aso	Стартер выходных операндов	@Intmadf
Apdi_3x16	Упакованный операнд с 3 16-разрядными данными	V0V1V2[Sh0h1h2]
Apdi_x4	Упакованный операнд с 4 16-разрядными данными	V0V1V2V3
Ccl_16	Конфигуратор констант 16-разрядных	[Dr]@ShmOuctDse{SmOucDs@s}V
Ccl_38f	Конфигуратор констант 38 (источник функциональной части)	[Dr]@ShmOucDs
Ccl_38d	Конфигуратор констант 38 (источник содержательной части)	[Dr]V38
Ccs	Конфигуратор секции	@C_ca_cm_cs_ct_ta_tm_ts_tt @Cjldbfmrsct
Cad	Мусорщик для ПАП	@Atc[ShmOcutDrse]@s
Cacn	Действующий обычный	@Asr_ab_[ShmOuctDrse]{SmOucDs@s}@s
Caco	Действующий специальный	@Adiu[ShmOuctDrse]{SmOucDs}
Cah	Тормоз	[ShmOuctDrse]@s
Car	Погонщик	[ShmOuctDrse]@s
Cbt	Загрузчик тегов	[DrShm]@ShmOuctDse{SmOucDs@s}@s@Ti
Cbt_38	Загрузчик тегов 38-разрядного операнда	[DrShm]@ShmOcDs@s@Ti
Cbb	Переход	[DrShm]@ShmOuctDse{SmOucDs@s}@s@Bmb
Cbc	Контроллер циклов	[DrShm]@ShmOuctDse{SmOucDs@s}@s@Bm
Cbf	Загрузчик регистра флагов	@Bf[ShmOuctDrse]{SmOucDs@s}@s
Cbi	Загрузчик счетчика циклов	@Bis[ShmOuctDrse]{SmOucDs@s}@s
Cbb2	Переход 2	@Bmb[ShmOuctDrse]{SmOucDs@s}@s
Cbc2	Контроллер циклов 2	@Bm[ShmOuctDrse]{SmOucDs@s}@s
Di	Число с фиксированной запятой	@sV[ShmOuctDrse]{SmOucDs@s}
Di_38	Число с фиксированной запятой 38-разрядное	@sV38[ShmOuc]{@s}
Do	Число с фиксированной запятой (выходное данное)	@sV38@Address

Распределение задач по уровням ГАРОС

Для распределения задач по уровням ГАРОС предлагается использовать методы функциональной композиции и декомпозиции. Суть этих методов заключается в: порождении новых более сложных функций на основе существующих (композиция) и в представлении решаемой задачи в виде ниспадающей иерархии функции, т.е. разбиении сложных функций на несколько простых с последующим разбиением вновь образованных функций до требуемой степени детализации (декомпозиция).

Автор предлагает пользоваться двумя указанными методами функционального программирования для реализации первых двух этапов обобщенной методики программирования ГАРОС.

Реализация этапа I

Первый этап методологии - анализ поставленной задачи. В разделе 1.3.2 было показано, что РОУ является специализированным сопроцессором, предназначенным для решения задач ЦОС, что значительно сужает множество реализуемых алгоритмов.

Для успешной реализации первого этапа методологии необходимо:

1. провести системный анализ, декомпозировать исходную задачу на множество подзадач верхнего уровня и обобщенных алгоритмов их решения;
2. воспользоваться методологией модульного программирования и декомпозировать будущую программу на множество модулей, каждому из которых поставить в соответствие одну подзадачу;
3. построить последовательный алгоритм решения задачи, в узлах которого в виде подпрограмм разместить алгоритмы, выделенные в п. 1;
4. применить методы технологии OpenMP и разделить алгоритм из п. 3 на последовательные и параллельные участки;
5. в соответствии со спиральной моделью жизненного цикла ПО повторять пункты 1-4 до требуемого уровня декомпозиции исходной задачи;
6. в случае, если доля подзадач, требующих непосредственных вычислений, составляет более 50% (по мнению автора - достаточная), принять решение о реализации исходной задачи в среде ГАРОС.

Реализация этапа II

В результате реализации первого этапа методологии будет получено множество подзадач, а также общий алгоритм решения задачи, разбитый на последовательные и параллельные участки. На втором этапе нужно распределить все подзадачи между УУ и РОУ. Для этого:

1. каждую подзадачу переименовать в задачу и сформировать, тем самым, множество задач;

2. выбрать очередную задачу и декомпозировать ее на арифметическую подзадачу (непосредственно вычисления) и одну или несколько подзадач управления данными (это могут быть: загрузка данных, масштабирование данных, чтение выходных данных и т.п.);

3. в соответствии с критериями, указанными для второго этапа методологии, отнести вычислительную подзадачу либо к классу 2), либо к классу 3);

4. оставшиеся подзадачи отнести к классу 1);

5. п.п. 2-4 повторить для всех задач из п. 1.

Таким образом, реализация Этапа I и Этапа II дает ответ на вопрос 1) «Как оптимально распределить алгоритмы решаемой задачи между УУ и РОУ» из раздела 1.3.3.

Методика программирования задач управляющего уровня

В качестве управляющего уровня ГАРОС используется процессор общего назначения NIOS II, синтезируемый на ПЛИС компании Intel. Этот процессор из высокоуровневых языков программирования поддерживает только язык Си. Поэтому в процессе реализации задач из классов 1) и 2), сформированных на предыдущем этапе методологии, могут быть использованы только существующие методологии императивного программирования.

На этапе I уже применялась модульная методология, поэтому ее же необходимо применять при реализации задач на УУ. Кроме того, для обеспечения хорошей топологии программ УУ необходимо использовать структурную методологию. Применение обеих этих методологий - распространенная практика разработки системного программного обеспечения.

Реализация этапа капсульного программирования

Этап III обобщенной методики, названный этапом капсульного программирования - основной. Согласно методике капсульного программирования, описанной в разделе 3.2, необходимо пройти ряд этапов в разработке капсулы. Далее приводится подробное рассмотрение каждого этапа этой методики с точки зрения используемых методов и средств программирования для его реализации.

Подэтап III-1. Определение функциональной нагрузки на капсулу

1. На этом шаге осуществляется выборка очередной задачи из класса 3), полученного на втором этапе рекуррентно-потоковой методологии. Проектируется новая капсула, причем ей назначается функциональность выбранной задачи.

2. Для определения ГСП необходимо построить ярусно-параллельную форму (ЯПФ) графа [30] вычисления выбранной задачи. Эта форма в большинстве случаев имеет треугольный нисходящий вид, причем ярус p_{max} с наибольшим количеством узлов и определяет ГСП. В данном случае ГСП будет равняться количеству узлов на ярусе p_{max}. Построить ЯПФ графа можно при помощи компилятора языка программирования SISAL или других средств параллельного программирования.

3. Полученный граф ЯПФ может быть двух видов. Первый вид, характерный для алгоритмов с параллельными циклами, содержит большое количество достаточно длинных (в несколько ярусов) практически идентичных параллельных подграфов. При этом *телом* будет повторяющийся подграф. Второй вид графа характерен для итерационных и рекурсивных алгоритмов. В графе такого вида, скорее всего, не будет либо будет очень мало (например, два или три) повторяющихся подграфов. При этом целесообразно выделить не менее двух тел, которые образованы параллельными, но разными по структуре подграфами.

4. На данном шаге необходимо оценить математическую сложность выделенных на шаге 3 тел. Этот параметр впоследствии будет использован для принятия решения о дальнейшей декомпозиции разрабатываемой капсулы.

5. Для каждого выделенного тела необходимо осуществить оптимизацию соответствующих им подграфов с целью определения СП. Другими словами, можно (но не обязательно) привести каждый подграф к ЯПФ и тем самым определить их СП.

6. На данном шаге принимается решение о дальнейшей необходимости декомпозиции текущей капсулы на несколько капсул в силу ограничений ГАРОС (ГСП и СП не могут значительно превышать значения 4).

7. В случае, если капсула была декомпозирована на последовательность капсул, для каждой из них повторяются шаги 3-6 до тех пор, пока каждая капсула в итоговой последовательности не будет удовлетворять ограничениям ГАРОС.

Результат подэтапа I - последовательность капсул, обладающих высоким потенциалом оптимальной реализации в среде ГАРОС.

Подэтап III-2. Разработка капсул

1. Выбирается очередная капсула из последовательности для реализации.

2. Строится потоковый граф вычисления задачи, которую реализует капсула.

3. Данный шаг во многом аналогичен шагу 2 подэтапа III-1. Строится ЯПФ потокового графа, и определяется максимально возможная степень параллельности. В случае, если СП > 4, делается вывод о необходимости дальнейшего преобразования графа.

4. В случае необходимости ЯПФ преобразовывается в новый потоковый граф таким образом, чтобы он содержал не более четырех параллельных цепочек.

5. Применить «Алгоритм рекуррентной свертки», описание которого приводится далее в приложении. В результате применения этого алгоритма получается новый подграф, в котором часть дуг и узлов свернуты в единственный узел, который получает метку с функцией рекуррентной развертки. Граф, полученный в результате рекуррентной развертки, может быть восстановлен в прежнем виде.

6. Ключевой шаг в рамках всей методики, т.к. сопряжен с наибольшими трудностями ввиду отсутствия специализированных средств разработки и компиляции ПО для ГАРОС. Реализация этого шага требует от разработчика детального освоения свойств и особенностей ГАРОС, а также предлагаемой нотации граф-капсул, описывающей развернутое пошаговое исполнение капсулы в среде ГАРОС. Описание граф-капсулы приводится в разделе 3.4.4.

7. Оптимизация полученной граф-капсулы с учетом свободных ресурсов ГАРОС, а также вообще возможности организации указанного вычислительного процесса с учетом ограничений архитектуры. На этом шаге разработчик может также прийти к выводу, что выбранным способом реализовать капсулу невозможно. Тогда капсула декомпозируется, и процесс начинается заново.

8. Свертка граф-капсулы в символьную капсулу для дальнейшего использования в качестве шаблона I-структуры в Буферной памяти.

Применение данной методики при разработке каждой отдельной капсулы позволяет получить ответ на вопрос 2) «Как оптимально распределить фрагменты потокового графа конкретного алгоритма по параллельно функционирующим Исполнителям РОУ с учетом рекуррентной свертки и развертки» из раздела 1.3.3 (путем реализации шагов 5-7).

Обобщенная методика отладки капсул

В составе ПК ПОТОК были разработаны два программных средства отладки как ПО, так архитектуры. Первое из них – программа СИМПРА, описание которой приводится в разделе 3.4.2, а второе – программа СКАТ, описание которой будет приведено в разделе 3.4.3.

На основе применения этих средств была разработана обобщенная методика отладки капсул, реализующая подэтап III-3) методики капсульного программирования.

1. На основе математической постановки задачи, решаемой отлаживаемой капсулой, разработать программу «Эталон» на языке высокого уровня, учитывающую необходимые требования точности.

2. Сформировать представительную выборку комплектов входных данных.

3. На основе выборки данных с помощью программы «Эталон» сформировать комплекты выходных данных и принять их в качестве эталонных для сравнения.

4. Моделировать отлаживаемую капсулу для всех комплектов выборки входных данных с помощью имитационной модели ГАРОС. Если хотя бы один комплект выходных данных не совпадает с эталонным, отправить капсулу на доработку, иначе перейти к шагу 5.

5. Моделировать капсулу средствами программы СКАТ и VHDL-модели ГАРОС. Если хотя бы один комплект выходных данных не совпадает с эталонным, зафиксировать факт наличия ошибок в аппаратной модели и отправить ее на доработку.

Данная обобщенная методика должна быть применена ко всем капсулам. Реализуется данная методика с помощью подсистемы автоматизированной верификации и валидации ГАРОС программного комплекса ПК ПОТОК, описание которой приводится в разделе 3.4.5.

После завершения отладки капсул можно перейти к этапу IV обобщенной методики программирования ГАРОС и осуществить комплексную отладку программ управляющего и операционного уровней.

Алгоритм рекуррентной свертки

Один из основных этапов проектирования капсулы - этап преобразования потокового графа в динамический, а затем – в граф-капсулу. Такое преобразование называется *рекуррентной сверткой* и выполняется в соответствии с алгоритмом, разработанным автором и названным «Алгоритм рекуррентной свертки».

Этот алгоритм реализует шаг 5 подэтапа III-2 рекуррентно-потоковой методологии и включает в себя следующие этапы:

1. Построить потоковый граф в соответствии с шагом 4) подэтапа III-2 рекуррентно-потоковой методологии.
2. Выполнить анализ потокового графа и определить все цепочки пересылки промежуточных данных (именно эти данные подвергаются рекуррентным преобразованиям в рамках рекуррентной модели).
3. Для каждой уникальной цепочки:
   - 3.1. построить частично-рекурсивную функцию рекуррентных преобразований, количество шагов рекуррентных преобразований должно совпадать с длиной цепочки;
   - 3.2. выполнить свертку всех дуг цепочки в графе в вершину начала этой цепочки;
   - 3.3. запомнить параметры (функцию) развертки для данной вершины.
   - 3.4. удалить свернутые дуги и разметить входные данные, которые появляются впервые.

Данный алгоритм позволяет получить частичный ответ на вопрос 3) «Как выбрать и настроить требуемые функции рекуррентных преобразований» из раздела 1.3.3. Частичный он потому, что помимо построения функции преобразований, необходимо также обеспечить возможность переконфигурации устройства ПТ. Текущая же версия прототипа ГАРОС реализует только универсальный ПТ, а другого решения в рамках данной работы не было предложено. Результат этого алгоритма - динамический граф, содержащий в качестве дуг только входные данные, что позволяет легко преобразовать его в шаблон капсулы.

Структурные элементы имитационной модели

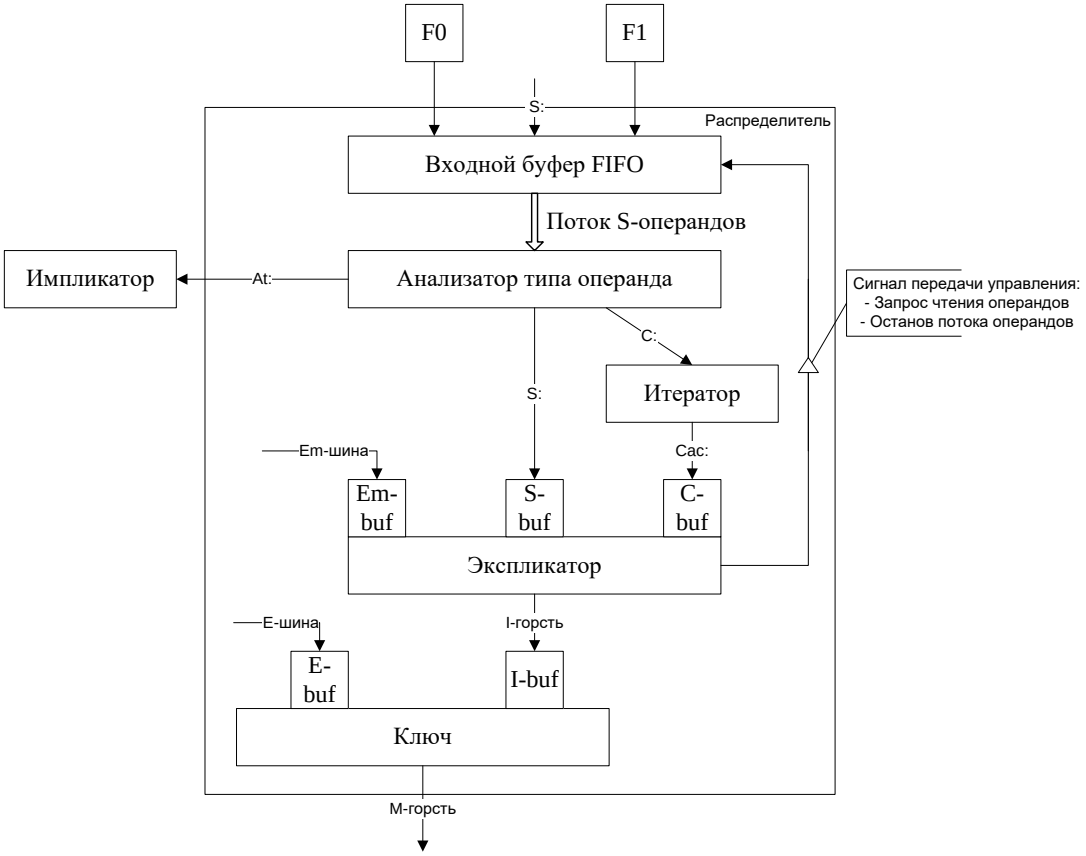
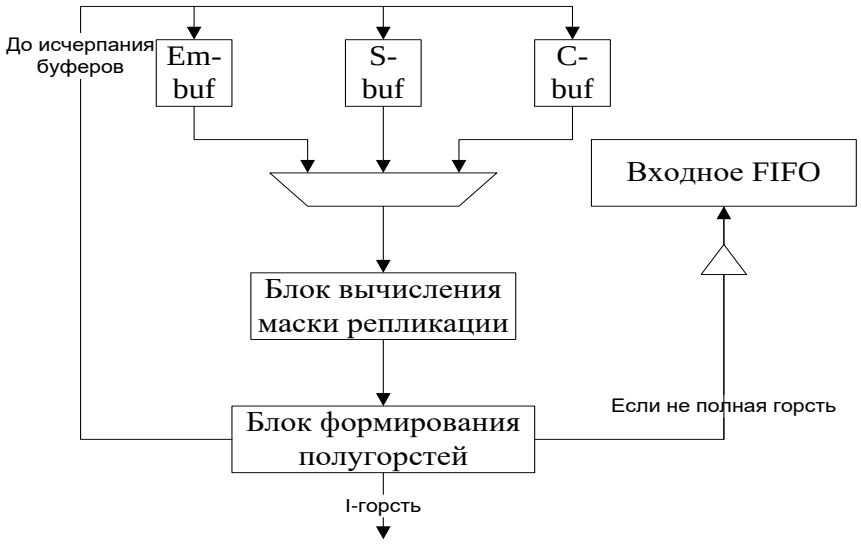


Рисунок А.10 – Структура компонента Распределитель

Экспликатор



Ключ

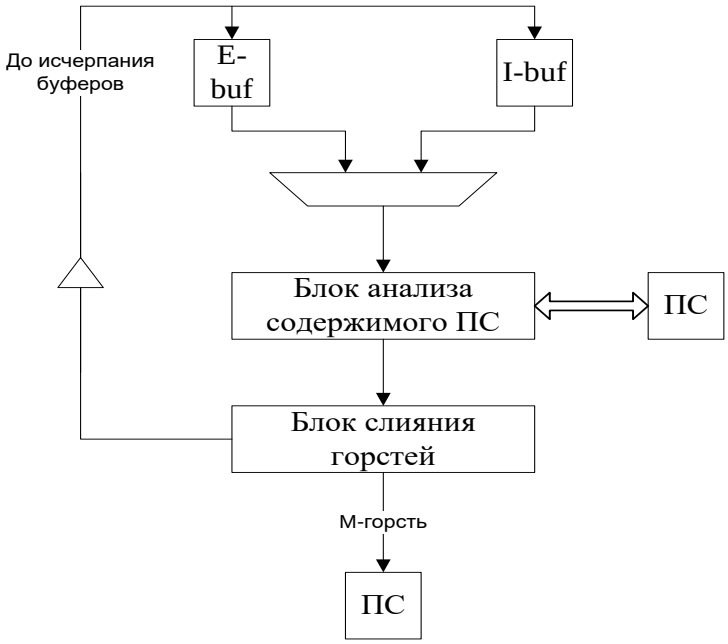


Рисунок А.11 – Структура блоков Экспликатор и Ключ

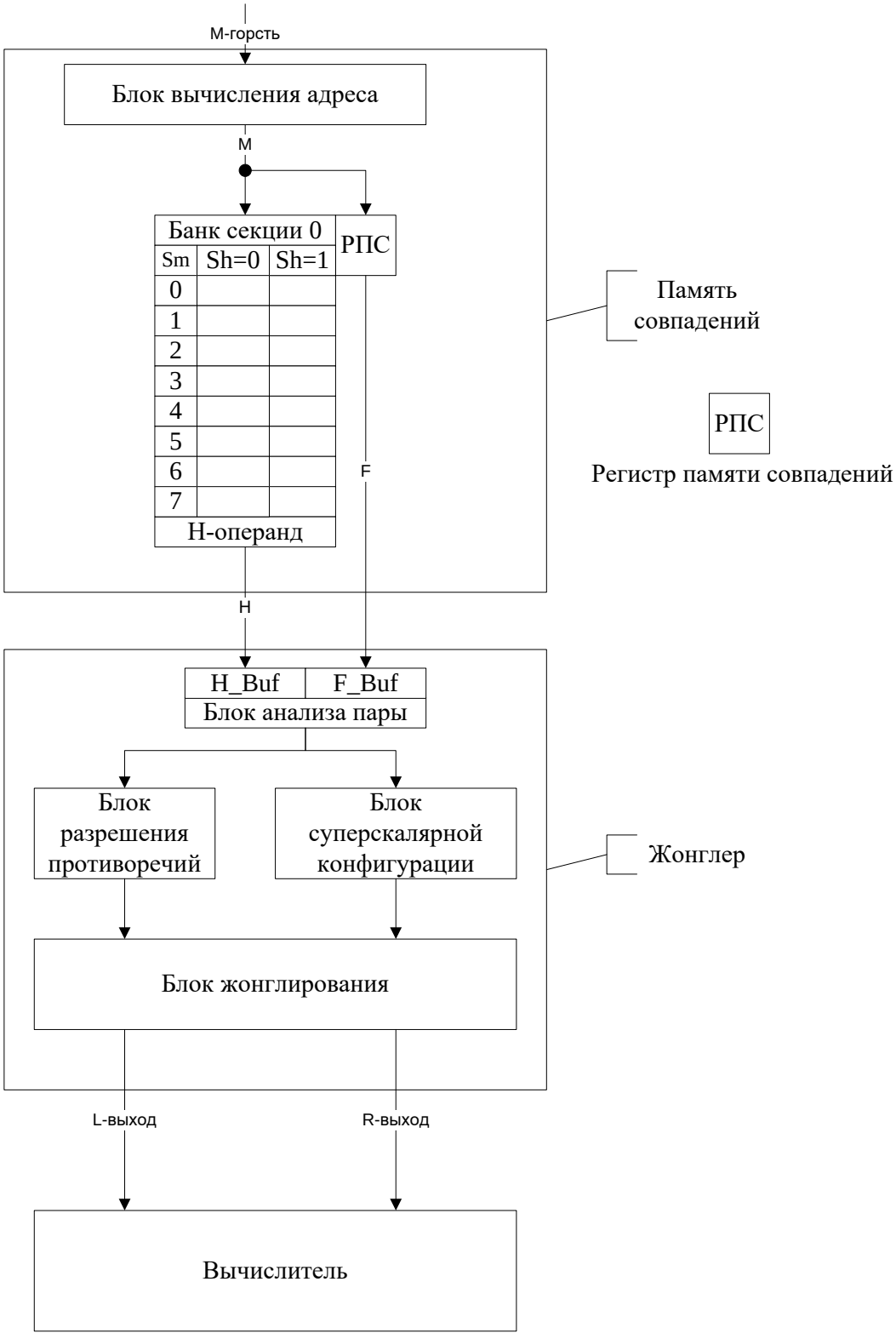


Рисунок А.12 – Структура компонентов ПС и Жонглер

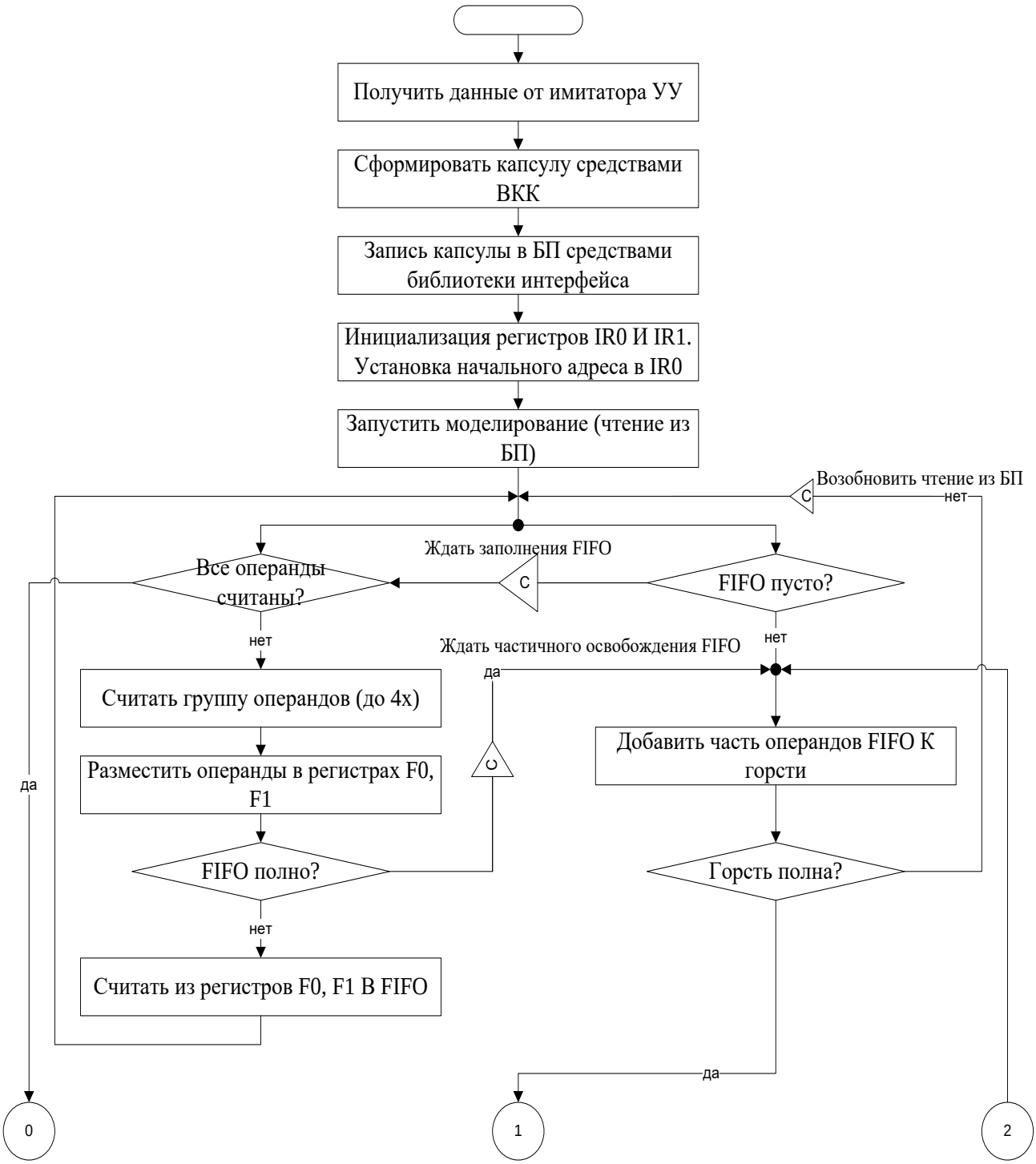


Рисунок А.13 – Общий алгоритм функционирования программной модели ГАРОС

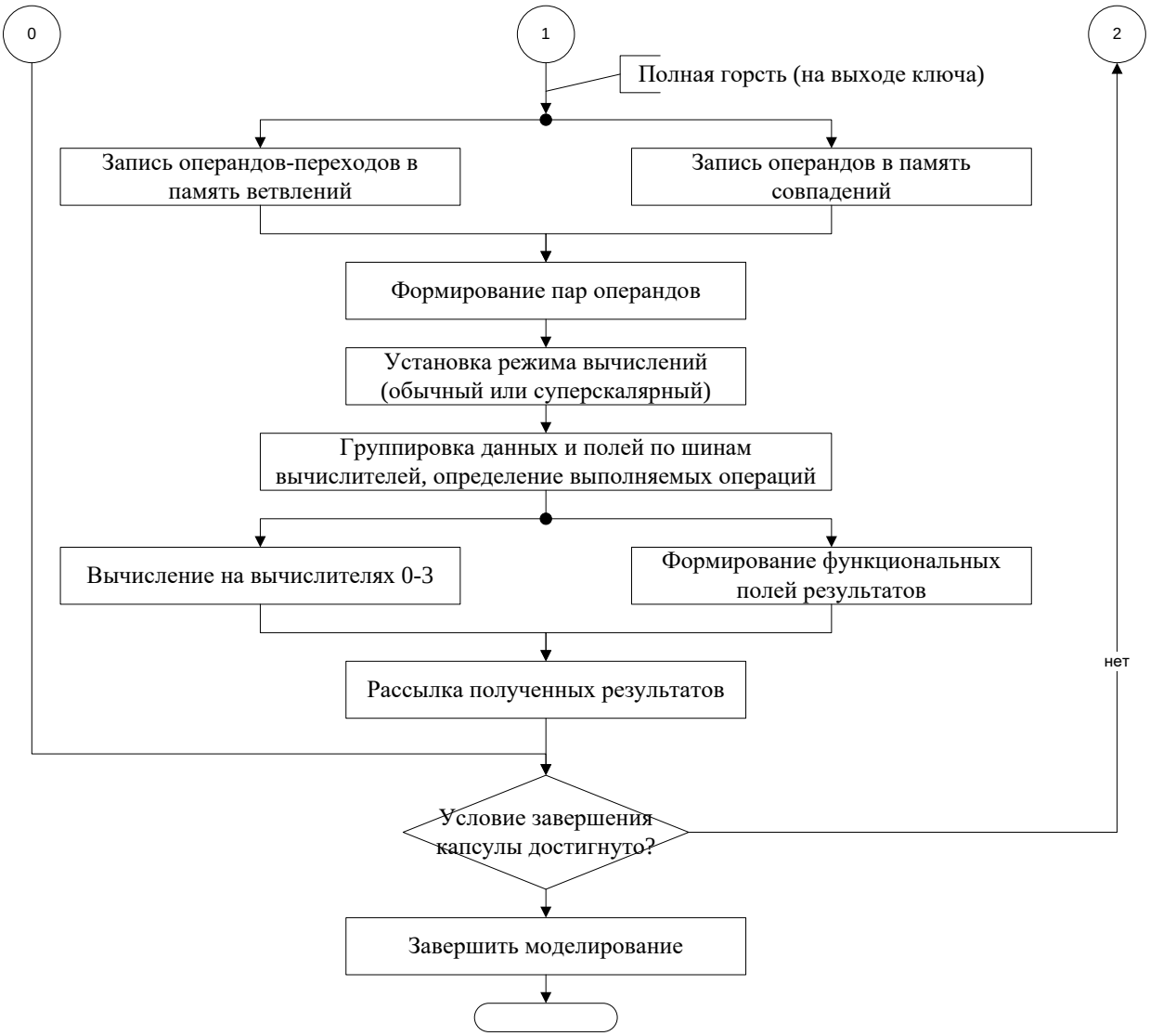


Рисунок А.13 (продолжение) – Общий алгоритм функционирования программной модели ГАРОС

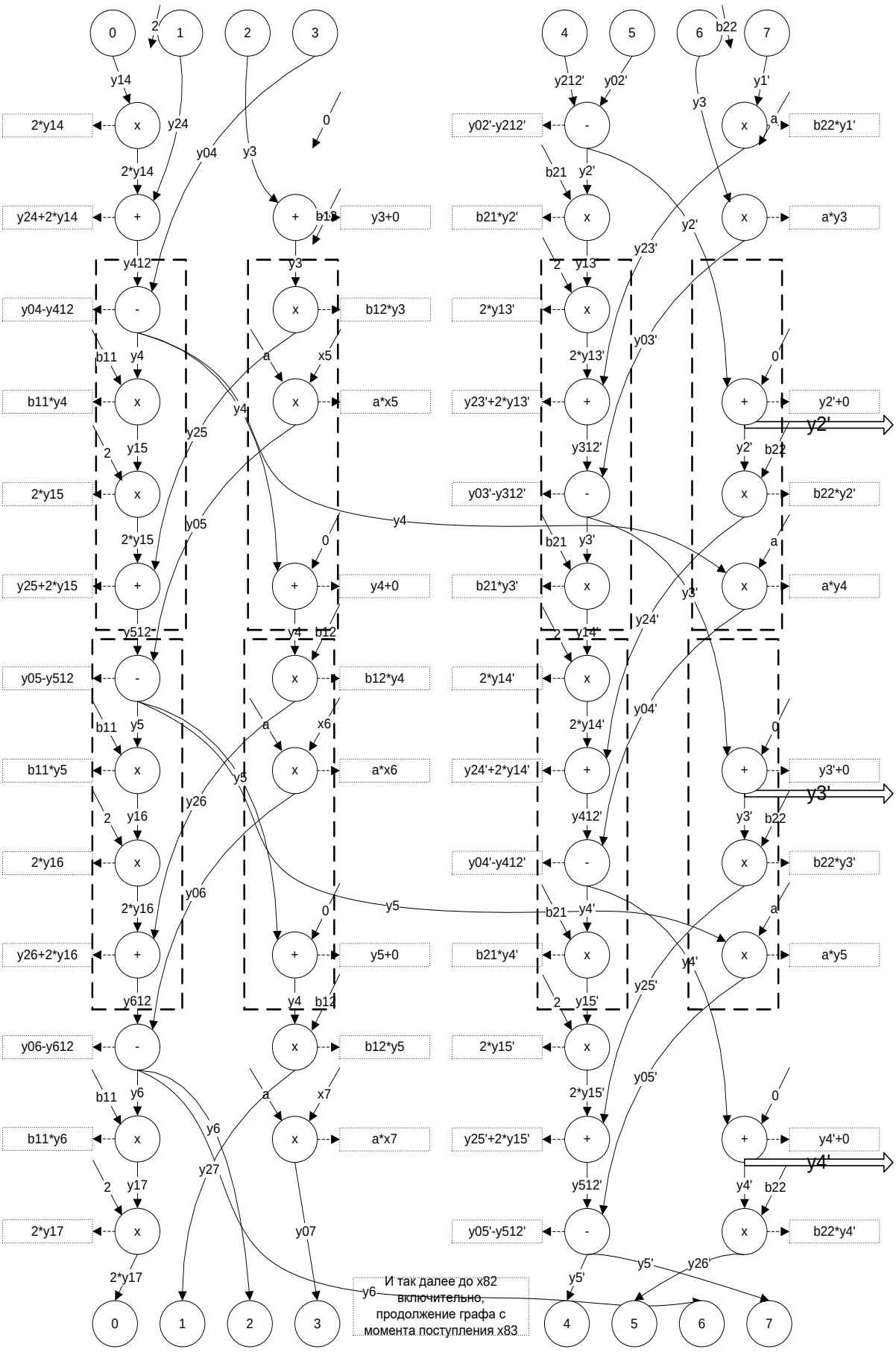


Рисунок А.14 – Фрагмент преобразованного потокового графа полосового фильтра

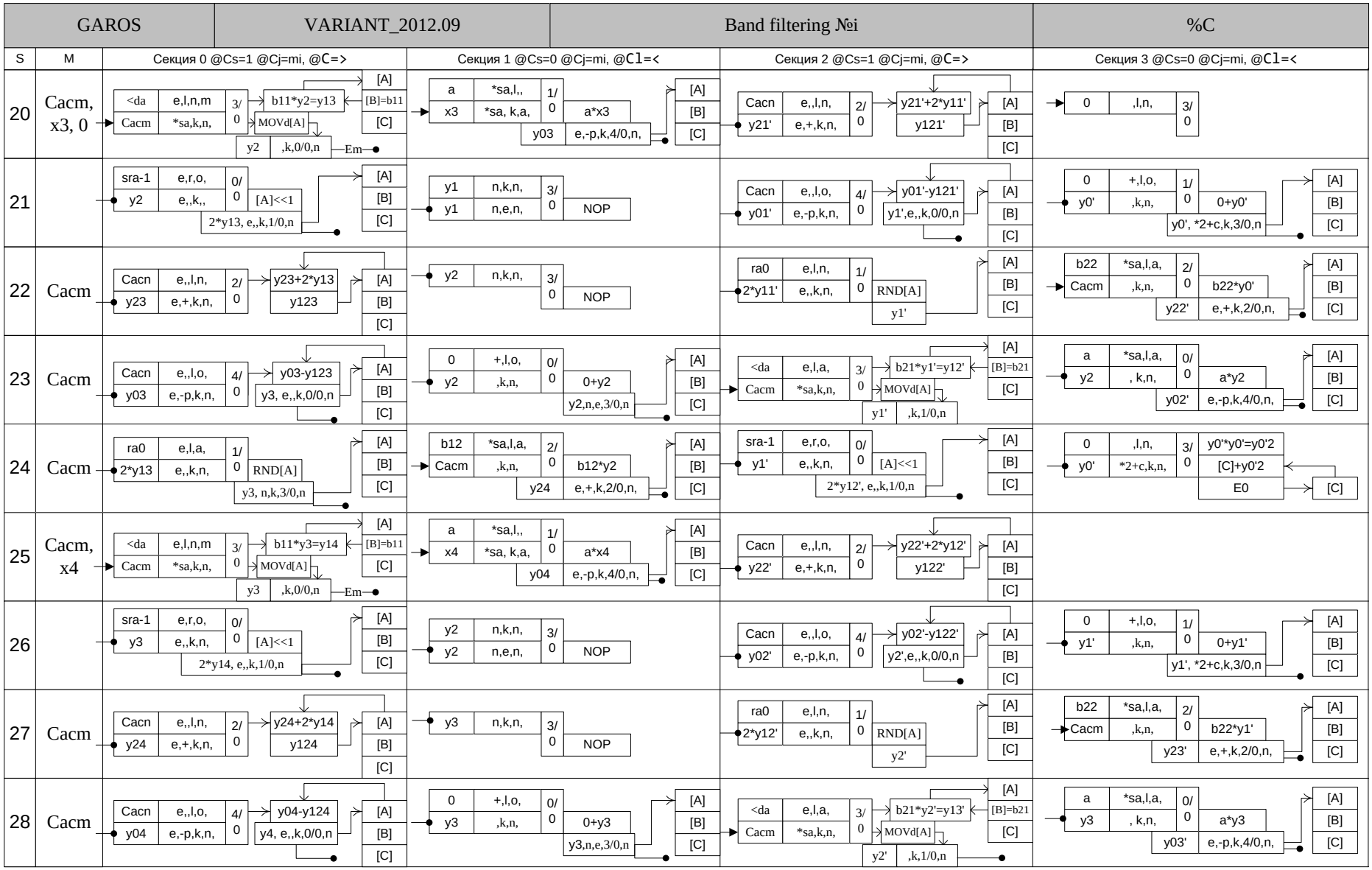


Рисунок А.15 – Фрагмент ручной граф-капсулы полосового фильтра

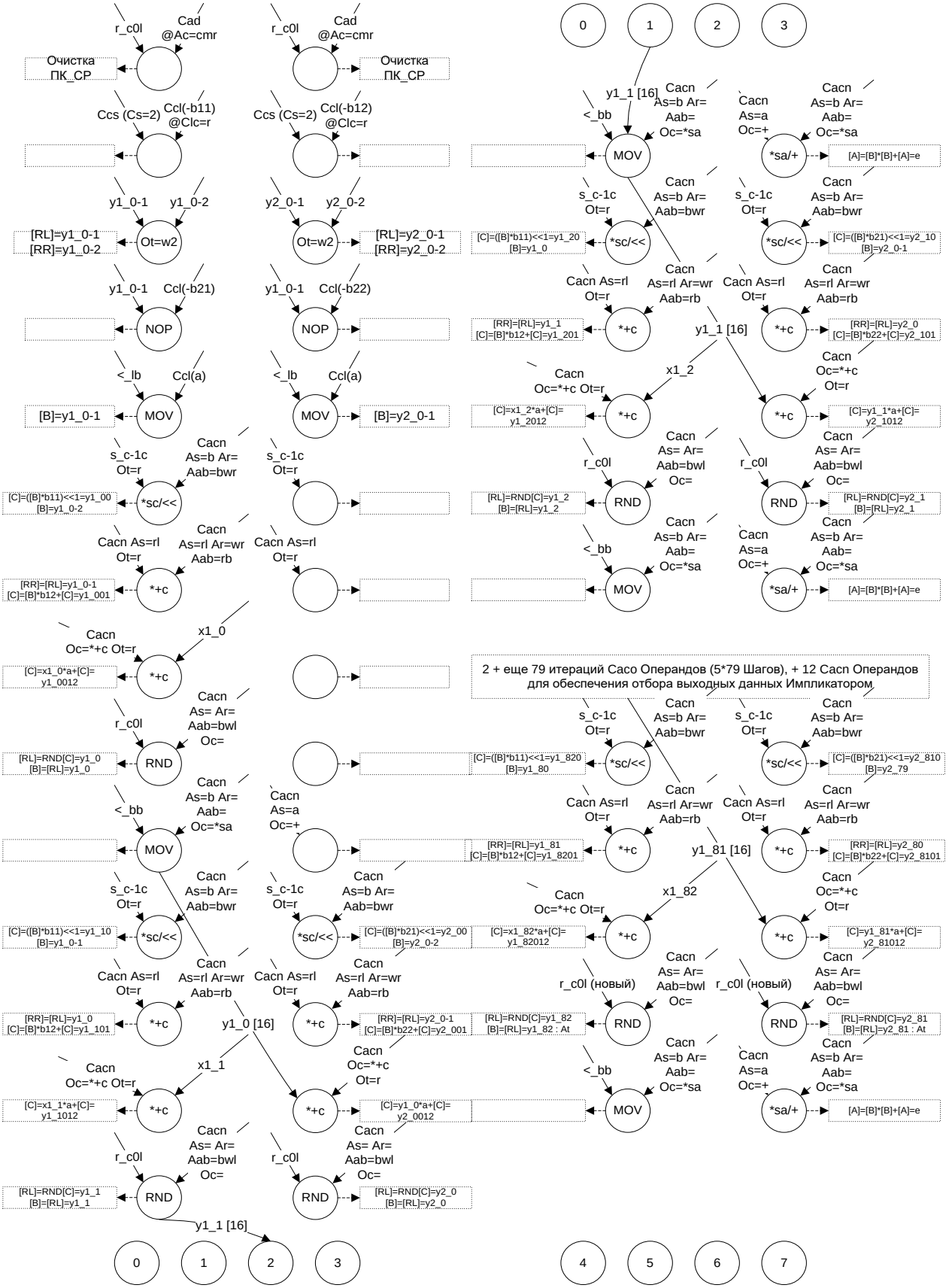


Рисунок А.16 – Фрагмент оптимизированного потокового графа полосового фильтра

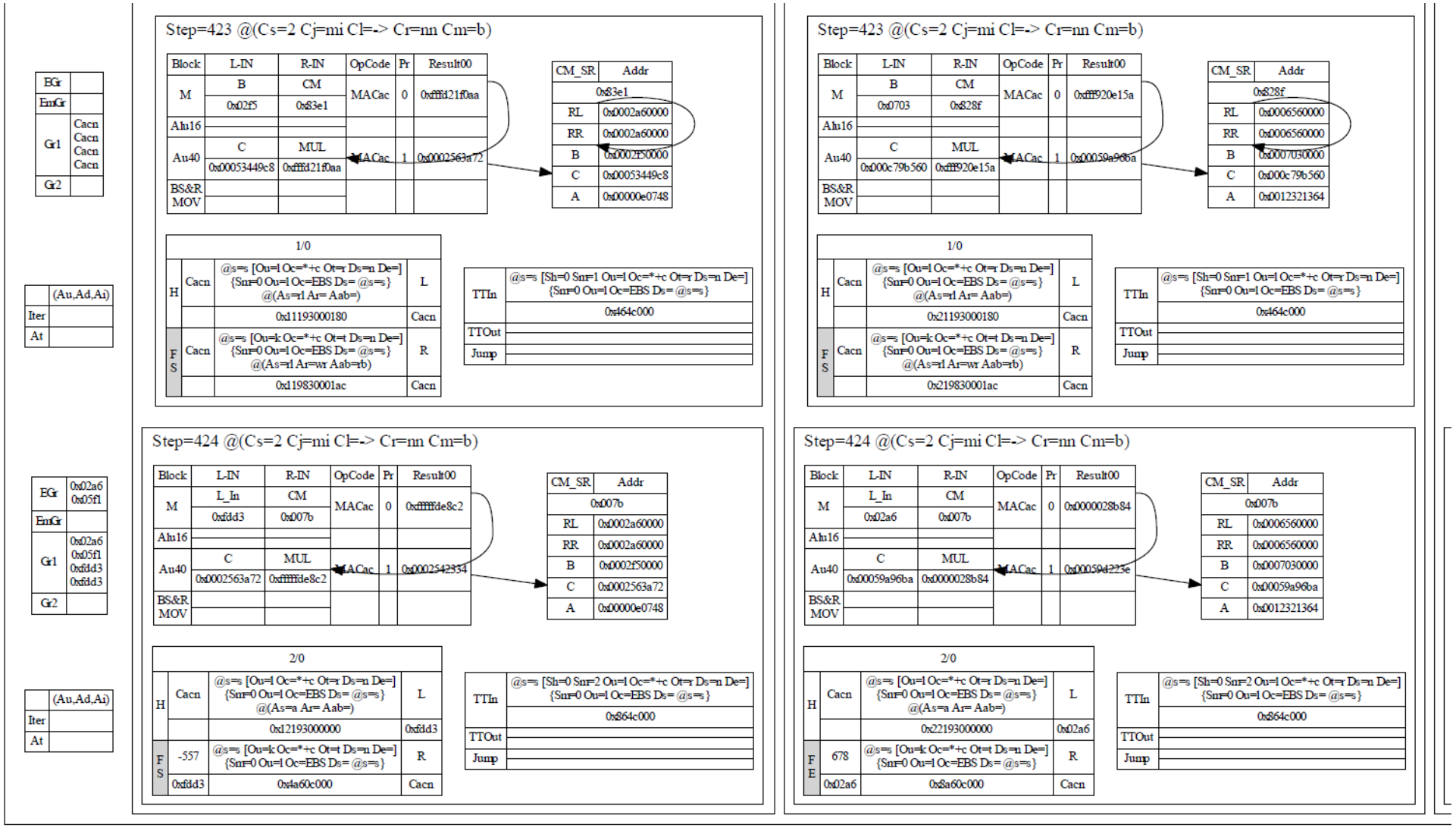


Рисунок А.17 – Фрагмент автоматической граф-капсулы оптимизированного полосового фильтра

Реализация алгоритмических ядер BDTI Kernels

Real Block FIR

Данное алгоритмическое ядро реализует блочный фильтр с конечной импульсной характеристикой (КИХ-фильтр), передаточная функция которого в общем виде задается формулой (1).

$$r[j] = \sum_{k=0}^{nh-1} h[k]x[j-k], 0 \le j \le nx \quad (1)$$

Здесь:

x – блок входных отсчетов

h – блок коэффициентов фильтра

nh – количество коэффициентов фильтра

nx – количество отсчетов

Особенностью блочного фильтра является то, что он осуществляет обработку сразу целого блока данных и поэтому не может быть использован в реальном времени (т.к. имеет непостоянную скорость получения фильтрованных отсчетов).

В библиотеке C55x DSP определено по крайней мере две версии реализации данного фильтра. Первая версия имеет следующую сигнатуру:

ushort oflag = fir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh), DATA имеет формат Q15

Данная функция реализует фильтр с использованием одного из двух имеющихся MAC-блоков в C55x. Согласно оценке, представленной в библиотеке, скорость вычисления данного фильтра равна:

Cycles Core: nx * (2 + nh)

Overhead: 25

Здесь *overhead* – это количество циклов, которое необходимо выполнить независимо от параметров фильтра (сюда входит подготовка данных, инициализация регистров и т.п.).

Реализация данного фильтра в ГАРОС имеет следующие особенности:

- Используется селективный режим работы, когда разные полугорсти формируются из операндов, считанных из разных блоков капсулы;
- Блок коэффициентов фильтра размещается в капсуле и адресуется регистром IR0;
- Блок входных отсчетов плюс память фильтра размещается в капсуле и адресуется регистром IR1;
- Используется одна секция, что соответствует одному MAC-блоку C55x;
- Используется прямая репликация для рассылки коэффициентов;
- Используется прямая репликация для рассылки отсчетов;
- Количество итераций капсулы равно nx;

• Дополнительной рутины на управляющем уровне не требуется.

Полученная оценка производительности составляет:

Cycles Core: nx * (1 + nh)

Overhead: 12

Небольшое ускорение относительно C55x достигается, скорее всего, за счет потоковой модели вычислений.

Вторая версия данного фильтра в DSP Library использует два MAC-блока и имеет следующую сигнатуру:

ushort oflag = fir2 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh) , DATA имеет формат Q15

Данная версия рассчитывает два отсчета каждые nh циклов. Согласно оценке, представленной в библиотеке, скорость вычисления данного фильтра равна:

Cycles Core: nx * (3 + nh/2) = nx/2 * (6 + nh)

Overhead: 25

В ГАРОС данная версия реализуется аналогичным образом, за исключением следующих особенностей:

• Используется две секции, что соответствует двум MAC-блокам C55x;

• Используется косвенная репликация для рассылки входных отсчетов по двум секциям.

Аналогично C55x версии эта версия капсулы рассчитывает два отсчета каждые nh циклов. Поэтому полученная оценка производительности составляет:

Cycles Core: nx/2 * (1 + nh)

Overhead: 12

Real Single-Sample FIR

Основным отличием данного фильтра от блочного является постоянная скорость получения выходных отсчетов. На каждый один входной отсчет приходится один выходной. Таким образом, данный фильтр может быть использован для обработки отсчетов, поступающих в реальном масштабе времени.

Функция фильтрации задается также формулой (1) с тем лишь отличием, что nx=1. В библиотеке C55x DSP также определено две версии реализации данного фильтра. Первая версия имеет следующую сигнатуру:

ushort oflag = convol (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)

Данная функция реализует фильтр с использованием одного из двух имеющихся MAC-блоков в C55x. Согласно оценке, представленной в библиотеке, скорость вычисления данного фильтра равна:

Cycles Core: nx * (1 + nh) = (1 + nh)

Overhead: 44

Реализация данного фильтра в ГАРОС имеет следующие особенности:

- Используется селективный режим работы, когда разные полугорсти формируются из операндов, считанных из разных блоков капсулы;
- Блок коэффициентов фильтра размещается в капсуле и адресуется регистром IR0;
- Память фильтра плюс входной отсчет размещается в капсуле и адресуется регистром IR1, при этом он имеет длину равную 2*nh;
- Используется одна секция, что соответствует одному MAC-блоку C55x;
- Используется прямая репликация для рассылки коэффициентов;
- Используется прямая репликация для рассылки отсчетов;
- Количество итераций капсулы равно 1;
- Требуется рутина управляющего уровня, которая записывает по одному входному отсчету в блок памяти фильтра и отсчетов в капсуле;
- Требуется дополнительная рутина управляющего уровня, которая накапливает память фильтра длины nh и записывает ее в верхнюю половину блока отсчетов капсулы (длины nh*2) и перезапускает капсулу;
- Две дополнительные рутины обеспечивают потенциально бесконечную фильтрацию в реальном времени;
- Фильтр формирует один выходной отсчет на каждый входной отсчет.

Полученная оценка производительности составляет:

Cycles Core: nx * (1 + nh) = (1+ nh)

Overhead: 15

Вторая версия данного фильтра в DSP Library использует два MAC-блока и имеет следующую сигнатуру:

ushort oflag = convol2 (DATA *x, DATA *h, DATA *r, ushort nr, ushort nh)

Данная версия рассчитывает один отсчет каждые nh/2 циклов. Согласно оценке, представленной в библиотеке, скорость вычисления данного фильтра равна:

Cycles Core: nx * (1 + nh/2) = 1 + nh/2

Overhead: 24

В ГАРОС данная версия реализуется аналогичным образом, за исключением следующих особенностей:

- Используется две секции, что соответствует двум MAC-блокам C55x;
- Используется косвенная репликация для рассылки входных отсчетов по двум секциям;
- Используется косвенная репликация для рассылки коэффициентов фильтра;

• Фильтр формирует один выходной отсчет на каждый входной отсчет.

При такой реализации полученная оценка производительности составляет:

Cycles Core: nx/2 * (6 + nh) = (3+ nh)

Overhead: 15

Небольшая потеря происходит при пересылке данных между секциями.

Другая возможная реализация характеризуется следующими особенностями:

• Используется две секции, что соответствует двум MAC-блокам C55x;

• Используется косвенная репликация для рассылки входных отсчетов по двум секциям;

• Фильтр формирует один два выходных отсчета на каждые два входных отсчета, таким образом данная реализация является псевдо-реального времени.

При такой реализации полученная оценка производительности составляет:

Cycles Core: nx * (1 + nh/2) = 1 + nh/2

Overhead:

Complex Block FIR

Рассматриваемое алгоритмическое ядро является блочной версией FIR-фильтра (1), в котором и входные отсчеты, и коэффициенты являются комплексными числами. Для вычисления данного фильтра необходимо произвести преобразования уравнения (1). Полученные уравнения (2), (3) имеют следующий вид:

$$r_r(j) = \sum_{k=0}^{nh-1}(h_r(k) * x_r(j-k) - h_i(k) * x_i(j-k)) \quad (2)$$

$$r_i(j) = \sum_{k=0}^{nh-1}(h_i(k) * x_r(j-k) + h_r(k) * x_i(j-k)) \quad (3)$$

Комплексная версия данного фильтра в DSP Library C55x имеет следующую сигнатуру:

ushort oflag = cfir (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nx, ushort nh)

Наличие в C55x двух MAC-блоков позволяет одновременно вычислять произведения с накоплением (вычитанием) действительной части входного отсчета на обе части коэффициента и мнимой части входного отсчета на обе части коэффициента. Таким образом, вычисление обеих компонент суммы осуществляется за 2 цикла. Что и видно по оценке производительности в DSP Library C55x:

Cycles Core: nx * [8 + 2*(nh-2)]

Overhead: 51

Реализация данного фильтра в ГАРОС имеет следующие особенности:

• Используется режим формирования горстей по умолчанию;

• Блок коэффициентов фильтра размещается в памяти констант подгружаемой;

• Блок входных отсчетов плюс память фильтра размещается в капсуле и адресуется регистром IR0;

• Используется две секции, что соответствует двум MAC-блокам C55x;

• Используется косвенная репликация для рассылки отсчетов;

• Количество итераций капсулы равно nx;

• Дополнительной рутины на управляющем уровне не требуется.

Полученная оценка производительности составляет:

Cycles Core: nx * (1 + 2*nh)

Overhead:14

LMS Adaptive FIR

Адаптивный фильтр используется в случае фильтрации зашумленного сигнала. При этом его коэффициенты пересчитываются от шага к шагу, что позволяет более эффективно удалять шумы и приближать выходной сигнал к ожидаемому. В библиотеке DSP Library C55x реализован адаптивный алгоритм наименьших квадратов с задержкой. Данный фильтр включает в себя этап FIR, определяемый формулой (1), а адаптация описывается формулами (4) и (5):

$$e[i] = des[i] - r[i] \quad (4)$$

$$h_k[i+1] = h_k[i] + 2 * \mu * e[i] * x[i-k] \quad (5)$$

Реализация данного фильтра в DSP Library использует два MAC-блока и имеет следующую сигнатуру:

ushort oflag = dlms (DATA *x, DATA *h, DATA *r, DATA *des, DATA *dbuffer, DATA step, ushort nh, ushort nx)

Этот алгоритм вычисляется за 3 стадии: сперва, вычисляется обычный FIR фильтр, затем вычисляется ошибка, а затем пересчитываются коэффициенты. В DSP Library используется алгоритм с задержкой: то есть используется ошибка предыдущего шага и отсчет предыдущего шага при расчете коэффициентов для наиболее эффективного использования специальной LMS инструкции в C55x.

Согласно оценке, представленной в библиотеке, скорость вычисления данного фильтра равна:

Cycles Core: nx * (1 + nh) = nx * (5 + 2*nh)

Overhead: 26

Реализация данного фильтра в ГАРОС имеет следующие особенности:

• Используется селективный режим работы, когда разные полугорсти формируются из операндов, считанных из разных блоков капсулы;

• Блок коэффициентов фильтра размещается в капсуле и адресуется регистром IR0;

• Память фильтра плюс входной отсчет размещается в капсуле и адресуется регистром IR1, при этом он имеет длину равную 2*nh;

• Используется одна секция, что соответствует одному MAC-блоку C55x;

- Используется косвенная репликация для рассылки коэффициентов;
- Используется косвенная репликация для рассылки отсчетов;
- Количество итераций капсулы равно nx;
- Коэффициенты перезаписываются в капсулу на свои же места (in-place);
- Дополнительной рутины на управляющем уровне не требуется;
- Реализуется фильтр без задержки.

Полученная оценка производительности составляет:

Cycles Core: nx * (4 + 3*nh)
Overhead: 12

Real Single-Sample IIR

Высококачественные частотные FIR фильтры, как правило, требуют большой ширины окна (количества коэффициентов). Альтернативным решением является использование каскадных фильтров с бесконечной импульсной характеристикой (IIR), которые являются рекурсивными и позволяют значительно сократить количество коэффициентов.

В рамках DSP Library C55x реализованы две основные формы записи одной секции каскадных IIR фильтров, называемой также biquad: Direct Form I and Direct Form II. Формула (6) определяет IIR Form I, а формулы (7) и (8) – IIR Form II соответственно.

$$y[n] = b_0 * x[n] + b_1 * x[n-1] + b_2 * x[n-2] - a_1 * y[n-1] - a_2 * y[n-2] \quad (6)$$

$$d[n] = x[n] - a_1 * d[n-1] - a_2 * d[n-2] \quad (7)$$

$$y[n] = b_0 * d[n] + b_1 * d[n-1] + b_2 * d[n-2] \quad (8)$$

Здесь a1, a2, b0, b1, b2 – коэффициенты биквада. Количество каскадов (биквадов) задается параметром *nbiq*.

В DSP Library C55x определено четыре версий данного ядра, использующих 2 MAC блока. Первая версия реализует фильтр двойной точности в Form II (т.е. все отсчеты и коэффициенты это 32-разрядные числа). Сигнатура функции, реализующей фильтр двойной точности, имеет следующий вид:

ushort oflag = iir32 (DATA *x, LDATA *h, DATA *r, LDATA *dbuffer, ushort nbiq, ushort nr)

Фильтрация двойной точности требует вычисления сложной операции умножения 32p x 32p на 16-разрядном умножителе. Поэтому оценка производительности в DSP Library равна:

Cycles Core: nx * (7 + 31*nbiq)
Overhead: 77

Реализация данного фильтра в ГАРОС имеет следующие особенности:

- Используется режим формирования горстей по умолчанию;

- Блок коэффициентов для всех каскадов фильтра и входной отсчет размещается в капсуле и адресуется регистром IR0;
- Используется две секции, что соответствует двум MAC-блокам C55x;
- Используется прямая репликация;
- Используется Итератор для манипуляции регистрами «Вычислителя» и памятью фильтра;
- Память фильтра с каждого биквада перезаписывается обратно в капсулу;
- Количество итераций капсулы равно nx;
- Требуется рутина управляющего уровня, которая записывает по одному входному отсчету в капсулу;
- Дополнительная рутина обеспечивает потенциально бесконечную фильтрацию в реальном времени;
- Фильтр формирует один выходной отсчет на каждый входной отсчет.

Полученная оценка производительности составляет:

Cycles Core: nx * (4 + 21*nbiq)
Overhead: 12

Реализация ГАРОС содержит элемент оптимизации, который был внесен по аналогии с ассемблерной реализацией в DSP Library. Он заключается в том, что умножение low(32)*low(32) не вычисляется ввиду того, что последующий его сдвиг на 32 разряда вправо делает его значение равным нулю. По всей видимости, в документации библиотеки представлена устаревшая оценка, т.к. в ассемблерной реализации приведена оценка 24*nbiq. Тем не менее, в ГАРОС была получена оценка 21*nbiq.

Вторая версия реализует фильтр с 4 коэффициентами биквада в Form II. Чтобы получить данный фильтр, нужно принять $b_0 = 1$. Сигнатура функции, реализующей данный фильтр, имеет вид:

ushort oflag = iircas4 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)

За счет эффективного использования двух MAC блоков в C55x фильтр в данной форме фактически сводится к вычислению трех операций умножения с накоплением. Поэтому оценка производительности в DSP Library равна:

Cycles Core: nx * (2 + 3*nbiq)
Overhead: 44

Реализация данного фильтра в ГАРОС аналогична реализации iir32, с точностью до размеров коэффициентов и отсчетов.

Полученная оценка производительности составляет:

Cycles Core: nx * (2 + 5*nbiq)

Overhead: 12

Необходимость долго межсекционной пересылки приводит к временным затратам в два цикла, что и отражено в оценке 5*nbiq относительно C55x.

Третья версия реализует фильтр с 5 коэффициентами биквада в Form II (формулы (7) и (8)). Сигнатура функции, реализующей данный фильтр, имеет вид:

ushort oflag = iircas5 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)

Оценка производительности в DSP Library равна:

Cycles Core: nx * (5 + 5*nbiq)

Overhead: 60

Реализация данного фильтра в ГАРОС аналогична реализации iircas4, с точностью до одной дополнительной операции умножения с накоплением.

Полученная оценка производительности составляет:

Cycles Core: nx * (2 + 6*nbiq)

Overhead: 12

Необходимость долго межсекционной пересылки приводит к временным затратам в два цикла, что и отражено в оценке 6*nbiq.

Четвертая версия реализует фильтр с 5 коэффициентами биквада в Form I (формула (6)). Сигнатура функции, реализующей данный фильтр, имеет вид:

ushort oflag = iircas51 (DATA *x, DATA *h, DATA *r, DATA *dbuffer, ushort nbiq, ushort nx)

Особенность данной записи заключается в том, что все умножения содержат разные сомножители и поэтому эффективное использование 2-х MAC блоков в C55x невозможно. Поэтому оценка производительности в DSP Library равна:

Cycles Core: nx * (5 + 8*nbiq)

Overhead: 44

В ГАРОС данная версия реализуется аналогично iircas5 образом, за исключением следующих особенностей:

- Используется одна секции, что соответствует одному MAC-блоку C55x;
- Используется эффективный механизм перекачки памятей биквадов, отработанный на алгоритме фильтра Баттерворта в демонстрационной задаче распознавания слов.

Полученная оценка производительности составляет:

Cycles Core: nx * (1 + 7*nbiq)

Overhead: 12

Vector Dot Product

Скалярное произведение векторов (свертка векторов) – это скалярная величина, которая вычисляется по формуле (9).

$$D = a * b = \sum_{k=0}^{nx-1}(a_k * b_k) \quad (9)$$

Здесь nx – размер векторов (количество координат), очевидно, что вектора должны иметь одинаковый размер.

В DSP Library для вычисления свертки используется функция convol, уже описанная ранее. Поэтому оценка производительности взята аналогичная:

Cycles Core: (1 + nx)

Overhead: 44

Реализация в ГАРОС аналогична и имеет такую же оценку:

Cycles Core: = (1+ nx)

Overhead: 15

Vector Add

Сумма векторов – это векторная величина, которая вычисляется по формуле (10):

$$c[i] = a[i] + b[i] \quad (10)$$

Здесь *a[i], b[i], c[i]* – это координаты векторов длины nx.

В DSP Library помимо вычисления координат вектора суммы используется 1 АЛУ блок и также осуществляется контроль за переполнением. Поэтому оценка производительности равна:

Cycles Core: 3 * nx

Overhead: 23

Реализация данного фильтра в ГАРОС имеет следующие особенности:

- Используется режим формирования горстей по умолчанию;
- Координаты векторов размещаются в капсуле и адресуется регистром IR0;
- Используется одна секция, что соответствует одному АЛУ блоку C55x;
- Используется Итератор для реализации проверки переполнения и условных переходов;
- Количество итераций капсулы равно 1;
- Дополнительной рутины на управляющем уровне не требуется.

Полученная оценка производительности составляет:

Cycles Core: 2 * nx

Overhead: 14

Vector Maximum

Данное алгоритмическое ядро реализует функцию поиска максимума и его индекса во входном векторе. Используется 1 АЛУ блок. Оценка производительности, представленная в библиотеке C55x равна:

Cycles Core: 3 * nx

Overhead: 8

Реализация данного фильтра в ГАРОС имеет следующие особенности:

- Используется режим формирования горстей по умолчанию;
- Координаты вектора размещаются в капсуле и адресуется регистром IR0;
- Используется одна секция, что соответствует одному АЛУ блоку C55x;
- Используется Итератор для подсчета индекса максимума;
- Количество итераций капсулы равно 1;
- Дополнительной рутины на управляющем уровне не требуется.

Полученная оценка производительности составляет:

Cycles Core: 5 * nx

Overhead: 12

Микропроцессор C55x имеет более высокую производительность ввиду наличия специальной инструкции, которая позволяет за 1 цикл осуществить сравнение и обновить хранилище экстремума, а также получить значения разности и бита знака.

Viterbi Decoder

При передаче данных с помощью беспроводных средств и технологий возникает необходимость корректировки возникающих ошибок, чтобы избежать искажения сигналов. Для этого используются так называемые коды коррекции ошибок. Одним из наиболее распространенных таких кодов является конволюционный код. А метод кодирования – конволюционное кодирование. Алгоритм декодирования Витерби используется на стороне приемнике для декодирования конволюционно закодированного сигнала.

Существуют различные варианты конволюционных кодеров. В зависимости от выбранного кодера изменяется и реализация алгоритма Витерби. В рекомендациях компании TI приводятся особенности реализации GSM стандарта кодера-декодера, которые позволяют существенно оптимизировать вычисления алгоритма Витерби. Данным стандартом является GSM Half Rate Convolutional Encoder с параметрами R=1/n =1/2, K=5, Frame=189.

Сам алгоритм исполняется за две основных стадии: расчет метрики и обратная трассировка. В силу архитектурных особенностей ГАРОС мы не можем реализовать обратную трассировку. Потому далее приводятся оценки для этапа вычисления метрики.

Такие параметры кодера позволяют воспользоваться свойствами симметрии алгоритма и представить основную операцию Витерби как «бабочку» (по аналогии с БПФ). Поэтому быстродействие алгоритма зависит от скорости вычисления «бабочки». Для указанного стандарта, согласно информации, предоставленной компанией TI, «бабочка» вычисляется за 5 циклов. Тогда вычисление метрики может быть оценено по формуле (11):

$$Metric\ update, \frac{cycles}{frame} = (2^{K-2} * 5 + 2^{K-5} + 1 + n * 2^{n-1}) * Frame \qquad (11)$$

С учетом небольших оптимизаций финальная оценка для вычисления метрики, представленная компанией TI, составляет:

Cycles Core: 34 * Frame

Overhead: 121

Реализация данного алгоритма в ГАРОС сопряжено с множеством трудностей. В первую очередь данный алгоритм работает напрямую с битами, что значительно усложняет подачу данных на обработку. Однако это проблема является решаемой, хоть и не самым эффективным образом. Второй проблемой является насыщенность алгоритма условными переходами и поисками максимума/минимума, что как видно из предыдущего ядра ГАРОС выполняет не особо эффективно. Тем не менее, удалось разработать первую версию решения с использованием 1 секции РОУ.

Полученная оценка равна:

Cycles Core: 109 * Frame

Overhead: N/A

Такая значительная разница в скорости вычислений связана с:

• наличием в C55x специальной инструкции (см. Vector Maximum), которая позволяет за 1 цикл искать максимум, сохранять его и бит знака (нужный для построения трассы в специальном регистре);

• возможностью C55x использовать 40-битные регистры и АЛУ в режиме обработки сразу двух 16-разрядных данных, что фактически означает эмуляцию двух 16-разрядных АЛУ.

256-Point In-Place FFT

Быстрое преобразование Фурье является одним из основных и наиболее часто используемых алгоритмов ЦОС. В связи с этим существует множество различных методов его реализации, обеспечивающих высокую скорость вычислений.

Рассматривается реализация без аппаратного ускорителя для C55x. Используется масштабирование данных для избегания переполнения. Согласно DSP Lib C55x данная версия обрабатывает одну операцию «бабочка» за 5 циклов.

Тогда время выполнения всего алгоритма БПФ составляет: $5 * \frac{\mathrm{N}}{2} * \log_2 \mathrm{N}$

Для эффективной реализации в ГАРОС алгоритма БПФ разработана мощная аппаратная поддержка, рассмотренная в разделе 2.7.

Общее время выполнения всего алгоритма БПФ в циклах составляет: $4 * \frac{N}{2} * \log_2 N$