

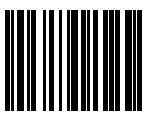
Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА  
GRADUATION THESIS

Механизмы автоматизированного синтеза микроархитектуры крупногранулярных  
реконфигурируемых вычислителей

**Обучающийся / Student** Закусило Виталий Андреевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники  
**Группа/Group** P42171  
**Направление подготовки/ Subject area** 09.04.04 Программная инженерия  
**Образовательная программа / Educational program** Технологии промышленного программирования 2020  
**Язык реализации ОП / Language of the educational program** Русский  
**Статус ОП / Status of educational program**  
**Квалификация/ Degree level** Магистр  
**Руководитель ВКР/ Thesis supervisor** Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

Обучающийся/Student

Документ подписан	
Закусило Виталий Андреевич	
01.06.2022	

(эл. подпись/ signature)

Закусило  
Виталий  
Андреевич

(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ подписан	
Пенской Александр Владимирович	
31.05.2022	

(эл. подпись/ signature)

Пенской  
Александр  
Владимирович

(Фамилия И.О./ name  
and surname)

Министерство науки и высшего образования Российской Федерации  
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО  
ITMO University

АННОТАЦИЯ  
ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
SUMMARY OF A GRADUATION THESIS

**Обучающийся / Student** Закусило Виталий Андреевич  
**Факультет/институт/кластер/ Faculty/Institute/Cluster** факультет программной инженерии и компьютерной техники  
**Группа/Group** P42171  
**Направление подготовки/ Subject area** 09.04.04 Программная инженерия  
**Образовательная программа / Educational program** Технологии промышленного программирования 2020  
**Язык реализации ОП / Language of the educational program** Русский  
**Статус ОП / Status of educational program**  
**Квалификация/ Degree level** Магистр  
**Тема ВКР/ Thesis topic** Механизмы автоматизированного синтеза микроархитектуры крупногранулярных реконфигурируемых вычислителей  
**Руководитель ВКР/ Thesis supervisor** Пенской Александр Владимирович, кандидат технических наук, Университет ИТМО, факультет программной инженерии и компьютерной техники, доцент (квалификационная категория "ординарный доцент")

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ  
DESCRIPTION OF THE GRADUATION THESIS

**Цель исследования / Research goal**

Автоматизация генерации микроархитектуры целевой системы в рамках проекта NITTA

**Задачи, решаемые в ВКР / Research tasks**

- 1) Анализ процесса синтеза специализированных процессоров NITTA и формирование стратегий принятия решений о размещении вычислительных блоков.
- 2) Разработка критериев и методов (алгоритмов) оценки решений о размещении вычислительных блоков.
- 3) Интеграция разработанных критериев и методов в проект NITTA. Тестирование.

**Краткая характеристика полученных результатов / Short summary of results/findings**

В работе были проанализированы особенности процесса синтеза процессоров проекта NITTA. В соответствии с этим были рассмотрены альтернативные пути реализации автоматической генерации микроархитектуры и выбран один из них. Выбранное решение было реализовано и протестировано.

Обучающийся/Student

Документ подписан	
Закусило Виталий Андреевич	

Закусило  
Виталий

01.06.2022

(эл. подпись/ signature)

Андреевич

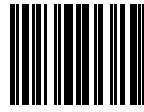
(Фамилия И.О./ name  
and surname)

Руководитель ВКР/  
Thesis supervisor

Документ  
подписан

Пенской  
Александр  
Владимирович

31.05.2022



(эл. подпись/ signature)

Пенской  
Александр  
Владимирович

(Фамилия И.О./ name  
and surname)

## СОДЕРЖАНИЕ

СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ.....	8
ВВЕДЕНИЕ.....	9
1 Обзор специализированных процессоров NITTA.....	12
1.1 Архитектура процессора NITTA .....	12
1.2 Процесс синтеза .....	13
1.3 Выводы.....	22
2 Критерии и методы изменения микроархитектуры.....	24
2.1 Изменение микроархитектуры в процессе синтеза .....	24
2.2 Метрики для оценки шага изменения микроархитектуры .....	28
2.3 Оценка шага по изменению микроархитектуры.....	34
2.4 Оценка результата синтеза по итогам обхода.....	36
2.5 Выводы.....	37
3 Реализация автоматического синтеза микроархитектуры .....	39
3.1 Реализация шагов по изменению микроархитектуры.....	39
3.2 Разбиение исходного алгоритма на этапы выполнения.....	39
3.3 Расчет метрик и оценка изменения микроархитектуры .....	40
3.4 Изменение метода оценки результатов синтеза .....	42
3.5 Конфигурирование набора вычислительных устройств.....	43
3.6 Изменение пользовательского интерфейса .....	44
3.7 Тестирование разработанного решения .....	45
3.8 Выводы.....	52
ЗАКЛЮЧЕНИЕ .....	54
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....	56
ПРИЛОЖЕНИЕ А – Исходный код buildProcessWaves.....	58

ПРИЛОЖЕНИЕ Б – Исходный код модульных тестов .....	59
ПРИЛОЖЕНИЕ В – Исходный код программ на языке Lua.....	61

## СПИСОК СОКРАЩЕНИЙ И УСЛОВНЫХ ОБОЗНАЧЕНИЙ

ВУ – вычислительное устройство

ПЛИС – программируемая логическая интегральная схема

САПР – система автоматизированного проектирования

ЦПОС – цифровой процессор обработки сигналов

ASIC – специализированная интегральная схема (Application-Specific Integrated Circuit)

CGRA – крупногранулярный реконфигурируемый вычислитель (Coarse-Grained Reconfigurable Array)

DFG – граф потока данных (Data Flow Graph)

FRAM – сегнетоэлектрическая оперативная память (Ferroelectric RAM)

GPP – процессор общего назначения (General-Purpose Processor)

GPU – графический процессор (Graphics Processing Units)

HLS – высокоуровневый синтез (High-Level Synthesis)

HPC – Haskell Program Coverage

NISC – No Instruction Set Computing

TTA – Transport Triggered Architecture

## ВВЕДЕНИЕ

В настоящее время эмпирические законы Мура и масштабирования Деннарда перестают работать, что означает невозможность решения проблемы производительности за счет уменьшения транзисторов и наращивания тактовой частоты [1,2].

Одновременно с этим, при разработке встраиваемых и киберфизических систем, существуют требования гибкости и энергоэффективности. Использование процессоров общего назначения (GPP, general-purpose processor) зачастую не может удовлетворить требования к производительности. Использование графических процессоров (GPU) и цифровых сигнальных процессоров (ЦПОС) в данной области не удовлетворяет требованиям энергоэффективности. Обе этих проблемы решает использование специализированных интегральных схем (ASIC), однако их использование затруднено часто меняющимися требованиями бизнеса и, как следствие, коротким жизненным циклом [3].

Наиболее частым решением описанных выше проблем является использование программируемых логических интегральных схем (ПЛИС) и крупногранулярных реконфигурируемых вычислителей (CGRA, coarse-grained reconfigurable array). Они обладают намного более высокими показателями производительности и энергоэффективности по сравнению с GPP, GPU и ЦПОС, а возможность изменения конфигурации и перепрограммирование обеспечивает гибкость, большую чем у ASIC [4].

Разработка на ПЛИС предполагает проектирование системы на уровне регистровых передач (RTL) или реализацию прикладного алгоритма на языке высокого уровня с последующим применением систем высокоуровневого синтеза (HLS) для автоматического отображения в RTL. При этом HLS призван решать такие проблемы проектирования на уровне RTL как:

- требование глубокого понимания принципов цифровой схемотехники;
- принятие ключевых микроархитектурных решений на ранних стадиях проекта;
- длительный процесс синтеза и медленная симуляций.

Однако существующие решения обладают недостатками, препятствующими их эффективному использованию, среди которых [5]:

- высокая сложность, непрозрачность и нестабильность существующих решений, что выражается в непредсказуемости любых изменений в высокоуровневом представлении прикладного алгоритма;
- неотделимость большинства промышленных инструментов высокоуровневого синтеза от более крупных систем автоматизированного проектирования (САПР), в состав которых они входят.

Данная работа выполнена в рамках проекта НИТТА, который посвящен разработке САПР для программирования ПЛИС, генерации и программирования CGRA процессоров. НИТТА может применяться для следующих целей [6]:

- Разработка встроенных и киберфизических систем.
- Тестирование аппаратного и программного обеспечения и быстрое прототипирование.
- Разработка аппаратных ускорителей и сопроцессоров [7].

Особенностью проекта является прозрачность процесса синтеза. Пользователь может остановить синтез в произвольной точке, рассмотреть уже принятые и требующие дальнейшего рассмотрения варианты развития синтеза, а также самостоятельно выбрать нужный ему вариант, тем самым направив процесс синтеза в необходимую сторону.

Однако, на данный момент, для работы НИТТА необходимо в качестве исходных данных предоставить микроархитектуру целевой системы. Это



связанно с тем, что в проекте процесс синтеза не поддерживает автоматическую генерацию микроархитектуры и этот этап разработки по-прежнему является ответственностью пользователя.

Соответственно, целью данной работы является автоматизация генерации микроархитектуры целевой системы в процессе синтеза.

Задачи работы:

- 1) Анализ процесса синтеза специализированных процессоров NITTA и формирование стратегий принятия решений о размещении дополнительных вычислительных устройств.
- 2) Разработка критериев и методов (алгоритмов) оценки решений по изменению микроархитектуры.
- 3) Интеграция изменения микроархитектуры в процесс синтеза и тестирование разработанного решения.

# 1 Обзор специализированных процессоров NITTA

## 1.1 Архитектура процессора NITTA

В проекте используется гибридная реконфигурируемая архитектура NITTA, сочетающая подходы NISC (No instruction set computing) и TTA (Transport triggered architecture) и включающая в себя [8–10]:

- устройство управления;
- вычислительные устройства;
- сигнальную шину;
- порты ввода/вывода;
- шину данных.

Рисунок 1 демонстрирует пример архитектуры специализированных процессоров NITTA.

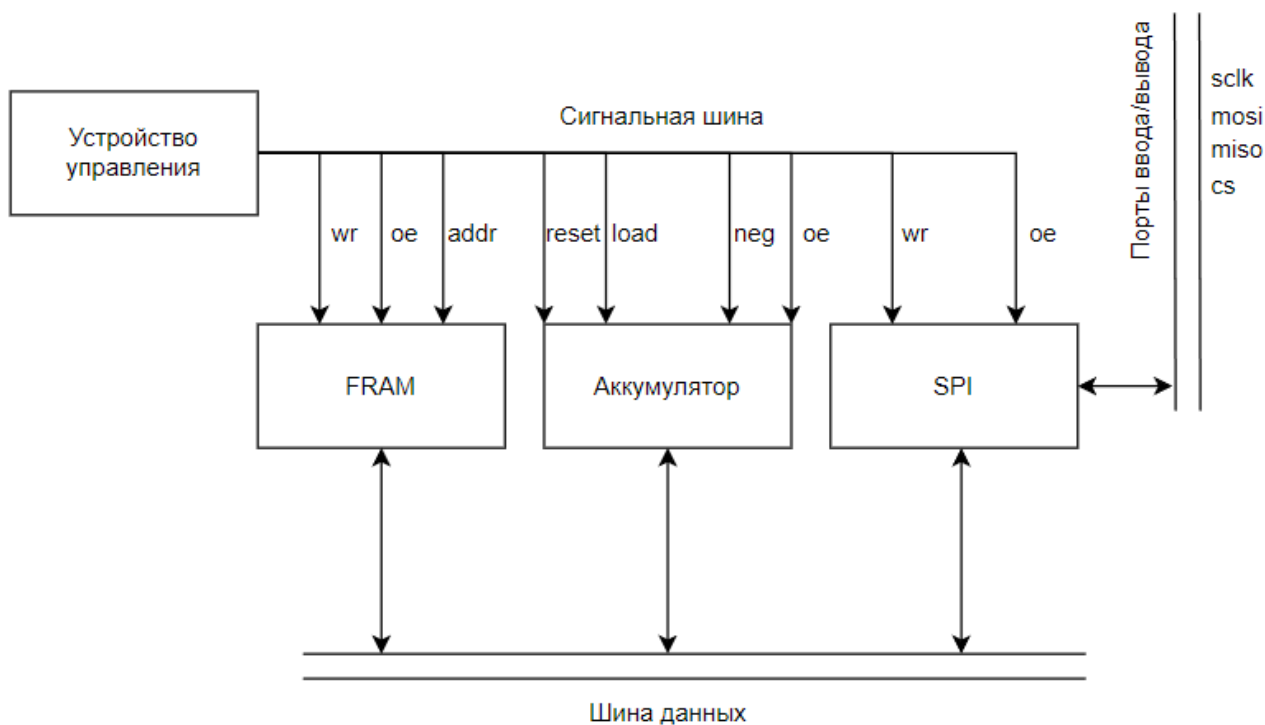


Рисунок 1 – Архитектура процессоров NITTA

Стоит отметить, что физически шина данных на схеме не присутствует. Она реализуется посредством объединения выходов вычислительных устройств с помощью логического ИЛИ и подачи полученного сигнала на входы вычислительных устройств.

Подход NISC позволяет хранить сигналы управления в прямом виде, поэтому в архитектуре не присутствует дешифратора команд. В то же время, использование ТТА подхода позволяет организовывать пересылку значений напрямую между вычислительными устройствами, без сохранения промежуточного результата.

В рамках данной работы под архитектурой мы будем подразумевать элементы целевой системы, определяющие способ взаимодействия между вычислительными устройствами (устройство управление, шина данных, сигнальная шина). Под микроархитектурой подразумевается точный набор вычислительных устройств, входящих в состав целевой системы.

## 1.2 Процесс синтеза

Работу САПР NITTA можно разделить на следующие этапы (Рисунок 2).

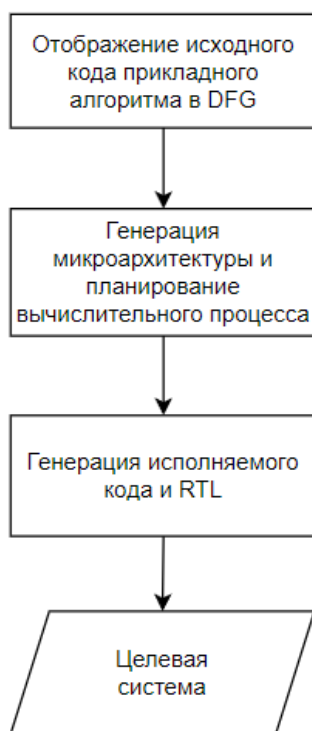


Рисунок 2 – Этапы работы САПР NITTA

Соответственно, под процессом синтеза, в рамках данной работы, будут подразумеваться этапы генерации микроархитектуры и планирования вычислительного процесса. На данный момент автоматическая генерация микроархитектуры в проекте NITTA не реализована. Пользователю необходимо задавать микроархитектуру вручную в файлах конфигурации.

### 1.2.1 Промежуточное представление алгоритма

Одним из входных параметров для NITTA является код программы на одном из поддерживаемых языков (Lua или XMILE). Для того, чтобы процесс синтеза не зависел от языка, код переводится в промежуточное представление, которое представляет из себя связный ориентированный граф (DFG, data flow graph). В качестве вершин графа выступают функции, ребра графа — это переменные.

Рассмотрим пример функции на языке Lua, которую демонстрирует Листинг 1.

Листинг 1 – Пример функции счетчика

```
function counter(x1)
  send(x1)
  x2 = x1 + 1
  counter(x2)
end
counter(1)
```

Данная функция будет преобразована в DFG, который демонстрирует Рисунок 3.

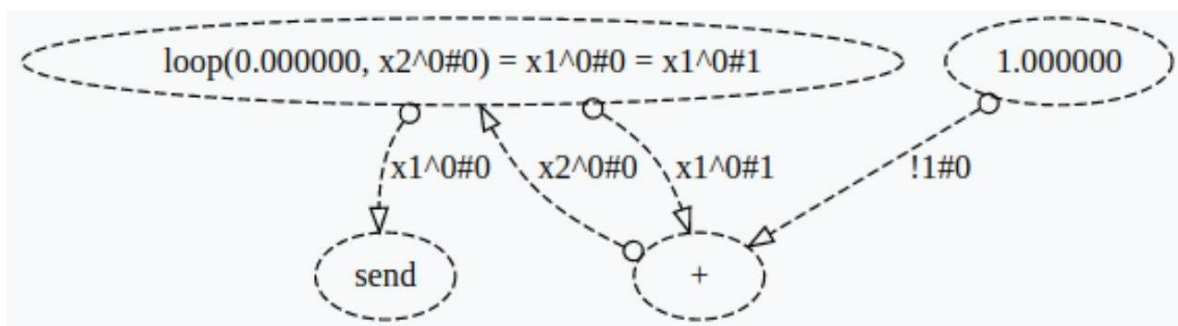


Рисунок 3 – DFG функции счетчика

Данный пример демонстрирует 2 особенности:

- 1) Поскольку вершинами графа могут быть только функции, то константы преобразовываются в функции, у которых нет входных параметров, а выходным параметром является константа.
- 2) При рекурсивном вызове функции (написание и вызов других функций на данный момент не поддерживается) передача значений между переменными осуществляется с помощью специальной функции loop.

В процессе синтеза DFG может измениться вследствие оптимизации или перестройки вычислительного процесса исходного алгоритма.

Например, шаги свертки констант убирают из DFG функции сложения, а когда невозможно организовать пересылку значений напрямую между вычислительными устройствами, то в DFG добавляется сохранение значений в буфер.

### 1.2.2 Компоненты процесса синтеза

Процесс синтеза происходит путем взаимодействия метода синтеза с моделью целевой системы (Рисунок 4).

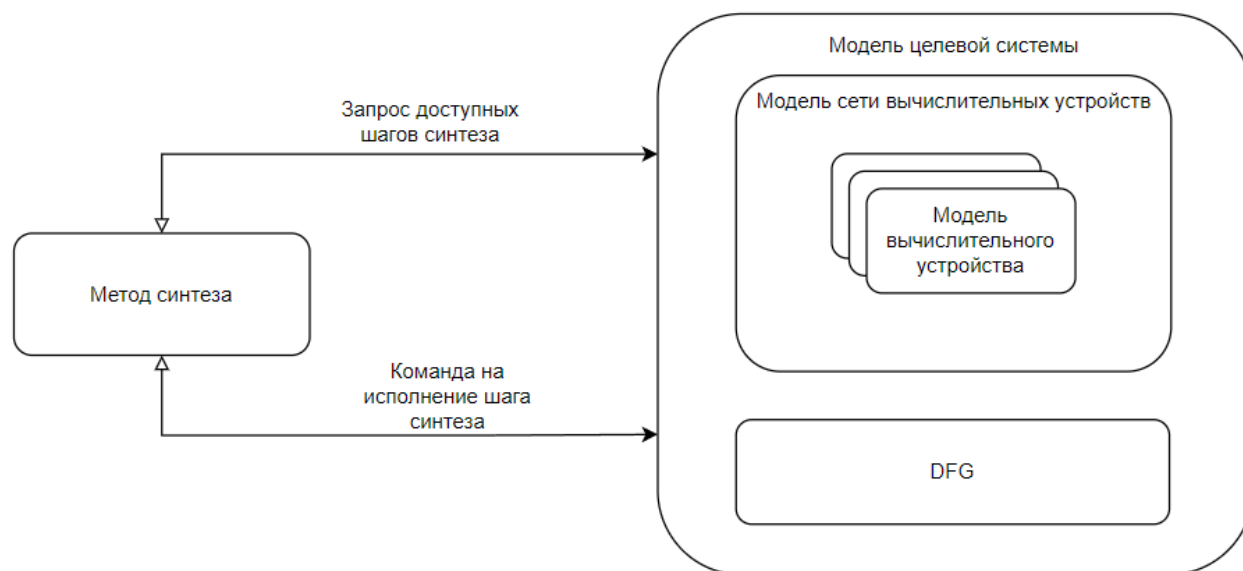


Рисунок 4 – Взаимодействие метода синтеза с моделью целевой системы

В типовом случае взаимодействие между методом синтеза и моделью целевой системы происходит до тех пор, пока список доступных шагов синтеза не станет пустым, либо не будет достигнут предел числа выполненных

шагов (предел необходим для избежания бесконечного синтеза). Метод синтеза последовательно выполняет следующие действия:

- запрашивает список доступных шагов синтеза;
- рассчитывает метрики, релевантные для каждого конкретного шага;
- рассчитывает численную оценку каждому шагу на основе метрик;
- выбирает дальнейшие шаги для исполнения в соответствии с методом синтеза.

Описанный процесс можно представить в виде дерева, вершины которого представляют из себя состояния модели системы, а ребра отображают исполнения шагов синтеза (Рисунок 5).

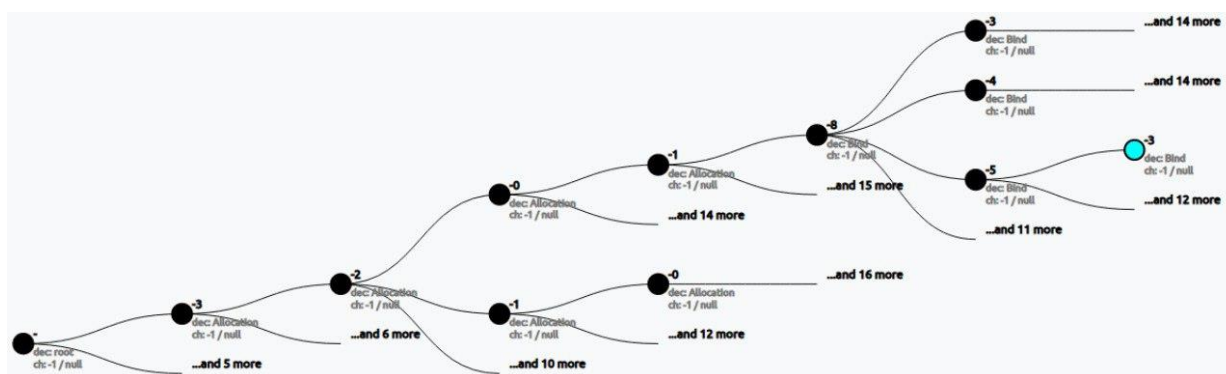


Рисунок 5 – Дерево решений процесса синтеза

Исходя из описания типового метода синтеза можно сформулировать следующие задачи:

- Разработка критериев доступности шагов изменения микроархитектуры.
- Разработка набора метрик для оценки шага изменения микроархитектуры.
- Разработка методов численной оценки шага изменения микроархитектуры.

Также стоит учитывать, что в общем случае метод синтеза может объединять в себе несколько других методов синтеза, с последующим выбором лучшего результата. На данный момент метод синтеза, в качестве конечного результата, выбирает вариант, у которого длительность цикла вычислений наименьшая. Если таких систем несколько, то выбирается первая

из них. В соответствии с этим возникает необходимость разработки методов учета микроархитектуры при выборе конечного результата.

Помимо метода синтеза, процесс синтеза также включает в себя модель целевой системы. Описание компонентов входящих в состав модели целевой системы и их назначение приведено ниже в обратном порядке иерархии.

Data flow graph:

- содержит информацию о промежуточном представлении алгоритма;
- предоставляет информацию о том, как может быть оптимизирован вычислительный процесс на уровне алгоритма и исполняет соответствующие шаги синтеза.

Модель вычислительного устройства:

- предоставляет информацию о том, какие функции может выполнить вычислительное устройство, а какие нет;
- предоставляет информацию о том, какие переменные необходимо предоставить (и в какой момент времени это можно сделать) для выполнения функции;
- предоставляет информацию о том, какие переменные вычислены и могут быть переданы другим вычислительным устройствам;
- предоставляет информацию о том, может ли вычислительный процесс быть оптимизирован и исполнять соответствующие шаги синтеза.

Модель сети вычислительных устройств:

- агрегирует информацию входящих в состав моделей вычислительных устройств и транслирует команды на выполнения шагов синтеза;
- предоставляет информацию о том, какие данные могут быть переданы между вычислительными устройствами и планирует их передачу;
- предоставляет информацию о том, как вычислительный процесс может быть оптимизирован на уровне сети и исполняет соответствующие шаги синтеза;
- предоставляет информацию о функциях, не запланированных для выполнения вычислительными устройствами, и исполняет шаги привязки функций к вычислительным устройствам.

Модель целевой системы:

- агрегирует информацию входящих в состав моделей сети вычислительных устройств (на данный момент поддерживается использование только одной сети);
- хранит и изменяет DFG в соответствии с применяемыми шагами синтеза;
- осуществляет привязку функций к моделям сетей вычислительных устройств.

В соответствии с описанием компонентов модели целевой системы в рамках данной работы возникает вопрос определения компонента, на уровне которого должна осуществляться генерация и исполнение шагов изменения микроархитектуры.



### 1.2.3 Типовой процесс синтеза

Процесс синтеза можно разделить на несколько этапов (Рисунок 6).

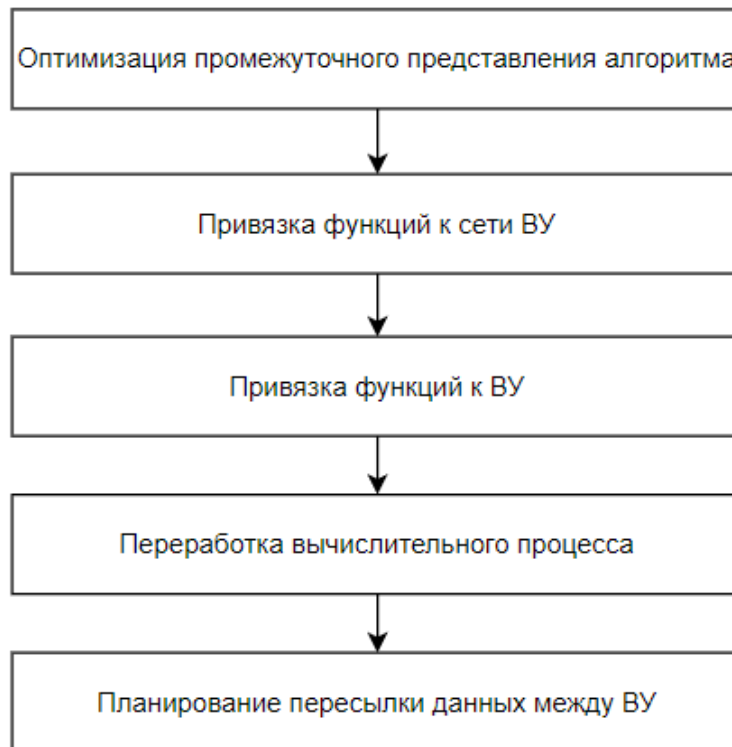


Рисунок 6 – Этапы процесса синтеза

Однако, такое разделение не является строгим и служит только для наглядной демонстрации зависимости отдельных шагов процесса синтеза друг от друга. В действительности, шаги процесса синтеза, относящиеся к разным этапам, могут выполняться параллельно. Рассмотрим процесс синтеза на примере простой функции  $h$  (Листинг 2).

Листинг 2 – Функция  $h$  на языке Lua

```
function h(a)
    local b = 1 + 1
    local c = a * b
    h(c)
end
h(1)
```

Промежуточное представление этой функции демонстрирует Рисунок 7.

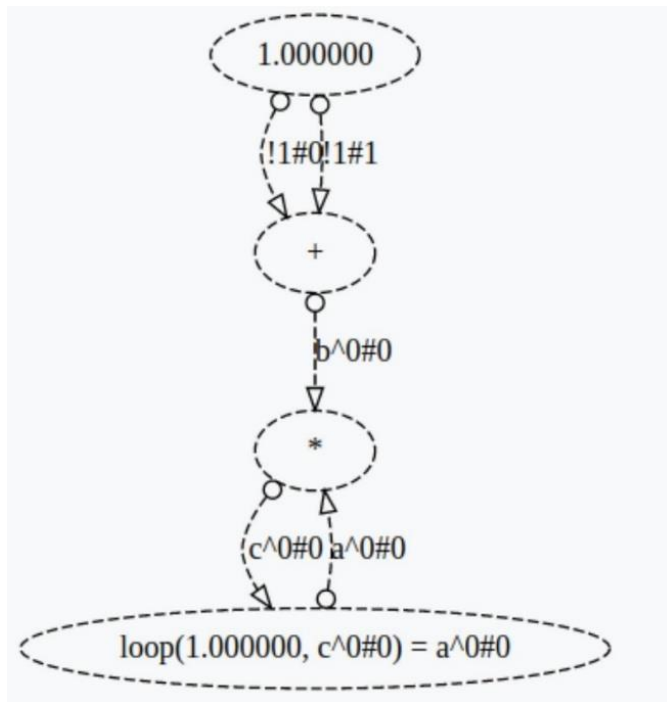


Рисунок 7 – DFG функции h

Также, в качестве входного параметра, будет использоваться микроархитектура, содержащая умножитель и FRAM (сегнетоэлектрическая оперативная память, Ferroelectric RAM).

На первом этапе синтеза будет произведена оптимизация промежуточного представления исходного алгоритма. В случае с функцией h метод синтеза произведет свёртку констант, в результате чего из DFG будет удалена функция сложения (Рисунок 8). Оптимизация DFG идет перед привязкой функций поскольку после того, как функция привязана к ВУ (вычислительное устройство) она не может быть удалена из DFG.

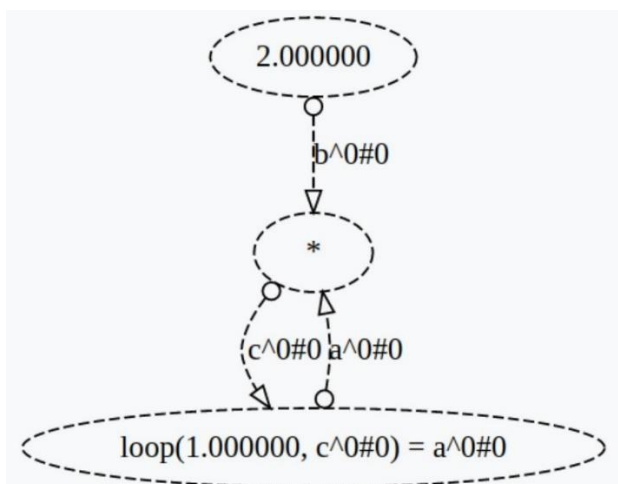


Рисунок 8 – DFG функции h после свёртки констант

Следующим этапом синтеза является привязка методов к вычислительным устройствам. После привязки функций к конкретным вычислительным устройствам становится возможным удаление циклов и взаимоблокировки. В случае с функцией  $h$  после привязки функции `loop` к `fram`, она будет разделена на функции `LoopBegin` и `LoopEnd` (Рисунок 9).

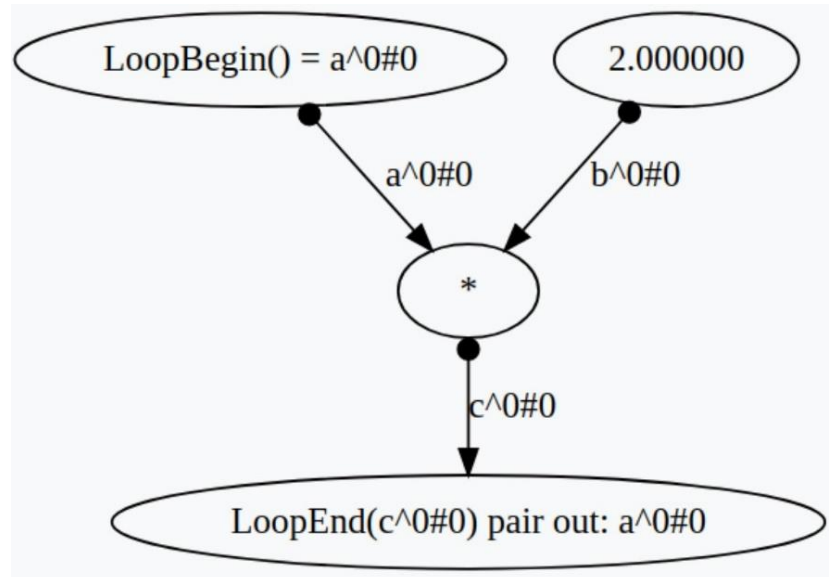


Рисунок 9 - DFG функции  $h$  после удаления циклов

После того как удалены все циклы и взаимоблокировки, становится возможным планирование пересылки данных между вычислительными устройствами.

Таблица 1 демонстрирует итоговую последовательность шагов процесса синтеза функции  $h$ . В ней видно, что переработка вычислительного процесса начинается перед тем, как все функции привязаны к вычислительным устройствам.

Таблица 1 – Шаги процесса синтеза функции h

Номер	Тип	Описание
1	Свертка констант	Сложение двух единиц преобразуется в константу 2.
2	Привязка функции	Функция loop привязывается к FRAM
3	Удаление цикла	функция loop разбивается на LoopBegin и LoopEnd
4	Привязка функции	Умножение привязывается к умножителю
5	Привязка функции	Функция const привязывается к FRAM
6	Планирование пересылки данных	Пересылка $a^{0\#0}$ из FRAM в умножитель планирует на первый такт
7	Планирование пересылки данных	Пересылка $b^{0\#0}$ из FRAM в умножитель планирует на второй такт
8	Планирование пересылки данных	Пересылка $c^{0\#0}$ из умножителя в FRAM планируется на пятый такт

### 1.3 Выводы

В рамках данной главы были рассмотрены особенности архитектуры специализированных процессоров NITTA. Также были рассмотрены особенности процесса синтеза и в соответствии с целью работы, сформулированы следующие задачи, требующие решения:

- Определение зависимостей шага изменения микроархитектуры от существующих шагов процесса синтеза.
- Разработка критериев предоставления шага по изменению микроархитектуры методу синтеза.
- Разработка метрик, необходимых для оценки шага изменения микроархитектуры.
- Разработка методов оценки шага изменения микроархитектуры.
- Определение компонента, (модель целевой системы, модель сети, модель вычислительного устройства) на уровне которого должна осуществляться генерация и исполнение шагов изменения микроархитектуры.

## **2 Критерии и методы изменения микроархитектуры**

Под изменением микроархитектуры в данной работе подразумевается добавление моделей вычислительных устройств и моделей сетей в модель целевой системы. Удаление моделей не рассматривается, поскольку удаление возможно только для устройств, которые не участвуют в вычислениях заданного алгоритма, а наличие таких устройств указывает либо на ошибку в процессе синтеза, либо на особенность конфигурации микроархитектуры.

### **2.1 Изменение микроархитектуры в процессе синтеза**

Сразу стоит отметить, что шаги изменения микроархитектуры не могут генерироваться и исполняться на уровне моделей вычислительных устройств, так как результатом выполнения шага является добавление новой модели сети или вычислительного устройства в модель целевой системы. Остается два возможных варианта:

- генерация и исполнение любых шагов выполняется на уровне модели целевой системы;
- генерация и исполнение шагов по добавлению моделей сетей выполняется на уровне модели целевой системы, а шаги по добавлению вычислительных устройств генерируются и выполняются на уровне модели сети.

Второй вариант предпочтителен, поскольку не нарушает принцип полиморфизма [11]. То есть, если в будущем появится необходимость в реализации нового типа сети вычислительных устройств, то реализация изменения микроархитектуры внутри этой сети не затронет логику изменения микроархитектуры на уровне модели целевой системы.

#### **2.1.1 Критерии предоставления шага по изменению микроархитектуры**

Одним из условий завершения процесса синтеза является то, что вершина дерева, на которой остановился метод синтеза, является конечной (Рисунок 10). Другими словами, список возможных последующих шагов синтеза является пустым.

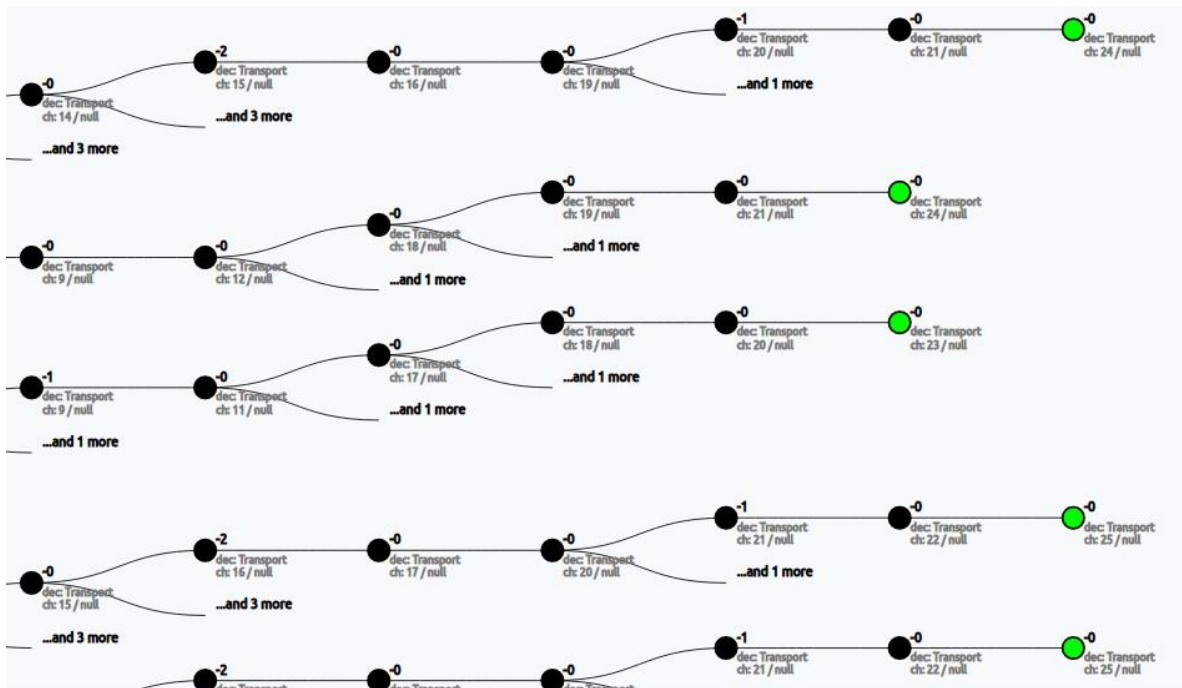


Рисунок 10 – Конечные вершины дерева синтеза

Для решения этой проблемы были рассмотрены следующие варианты:

- 1) делать шаги изменения микроархитектуры всегда доступными и не учитывать их при определении того, завершен ли синтез;
- 2) делать шаги изменения микроархитектуры доступными только при наличии функций, не привязанных к конечным вычислительным устройствам.

Минус первого варианта в том, что постоянное наличие шагов по изменению микроархитектуры требует их оценки, что может существенно замедлить синтез. Также не исключены ошибки синтеза, приводящие к выделению вычислительных устройств, когда это уже не является необходимым.

Для избежания повторной оценки одних и тех же шагов изменения микроархитектуры может быть использован кэш. Благодаря этому после выполнения шагов синтеза, которые не влияют на оценку изменения микроархитектуры, (например, шаги планирования пересылки данных между вычислительными устройствами) не рассчитывать оценку повторно.

Минус второго варианта заключается в том, что мы строго определяем последовательность шагов синтеза привязки функций к модели сети

вычислительных устройств и добавлением моделей вычислительных устройств. Сначала требуется привязать к модели сети функции, а затем выделить вычислительные устройства (Рисунок 11).

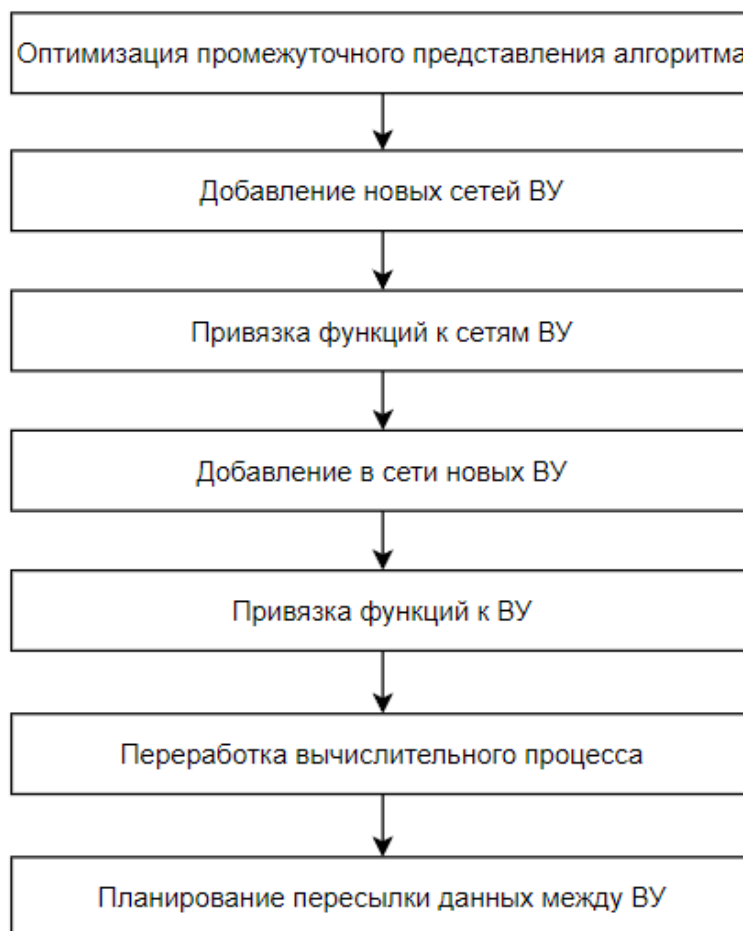


Рисунок 11 – Шаги изменения микроархитектуры в процессе синтеза

При такой последовательности шагов, во время привязки функции к сети вычислительных устройств, необходимо учитывать количество входящих в состав функций, вычислительных устройств и их прототипов. Это нужно для избежания ситуаций, когда после привязки функции оказывается, что прототипов вычислительных устройств доступных для выделения меньше, чем нужно для выполнения привязанных к модели сети функций.

Стоит отметить, что при обратном порядке выполнения шагов также возможны ошибки. В микроархитектуру может быть добавлено вычислительное устройство, которое в последствии не будет использоваться. В отличии от предыдущего примера, в этом ошибка синтеза не является явной. То есть процесс синтеза может быть успешно завершён без каких-либо



предупреждений о наличии бездействующих вычислительных устройств. Для выявления подобных случаев необходима разработка дополнительных средств проверки результатов процесса синтеза.

Поскольку, на данный момент, проект NITTA не поддерживает работу с несколькими сетями вычислительных устройств, а привязка функций к сети вычислительных устройств жестко закодирована и не выделена в отдельный шаг процесса синтеза, было принято решение делать шаги по изменению микроархитектуры доступными только при наличии функций, не привязанных к конечным вычислительным устройствам.

### 2.1.2 Подбор прототипов вычислительных устройств для выполнения функции

Прототипами вычислительных устройств будем называть устройства, которые не содержатся в микроархитектуре, но могут быть добавлены в нее в процессе синтеза.

В соответствии с решением, принятым в предыдущем разделе, для генерации шагов по изменению микроархитектуры, для каждой непривязанной функции требуется определить прототипы каких вычислительных устройств способны выполнить каждую из привязанных функций. Самым простым решением этой проблемы является полный перебор всего набора прототипов для каждой из непривязанных функций.

Сложность вычислений на каждом шаге синтеза такого подхода будет равна

$$O(n * m), \quad (1)$$

где  $n$  – количество функций во входном алгоритме,  $m$  – количество прототипов вычислительных устройств.

Соответственно, для входных алгоритмов, содержащих большое число операций, (сотни и миллионы тысяч) скорость синтеза будет существенно снижена. Для ускорения генерации шагов изменения микроархитектуры при

определении того, может ли прототип выполнить функцию, можно учитывать только её тип. Следовательно, можно для каждого типа функции хранить количество функций непривязанных к моделям вычислительных устройств и перебирать только те типы, для которых количество непривязанных функций больше нуля.

Также можно заметить, что на каждом шаге синтеза, типы функций не меняются, меняется только их набор, следовательно, для каждого типа функций можно сохранить набор подходящих прототипов (или уже готовые шаги по изменению микроархитектуры) и использовать его на каждом шаге синтеза вместо того, чтобы перебирать весь набор прототипов.

## **2.2 Метрики для оценки шага изменения микроархитектуры**

На момент оценки решения об изменении микроархитектуры доступны следующие данные:

- граф потока данных;
- текущая микроархитектура;
- набор функций, ожидающих привязки к моделям вычислительных устройств;
- набор прототипов вычислительных устройств, доступных для выделения.

В дополнение к приведенному списку, глядя на модель вычислительного устройства, также можно получить информацию о поддерживаемом уровне параллелизма: параллельное выполнение, последовательное выполнение, конвейерная обработка. То есть определить способно ли вычислительное устройство выполнять функции

- параллельно – вычисление нескольких функций может быть начато в один момент времени, и они могут вычисляться одновременно;
- в режиме конвейера – вычисление следующей функции может быть начато до того, как закончилось вычисление текущей функции;
- последовательно – вычисление следующей функции может быть начато только после окончания вычислений текущей функции.

В случае со специализированными моделями вычислительных устройств, такими как сумматор или делитель, уровень параллелизма зависит от реализации и может быть жестко задан в момент разработки. Однако ситуация меняется, когда речь заходит о моделях вычислительных устройств общего назначения. Проблемой таких моделей является то, что на момент разработки неизвестно как будет использоваться вычислительное устройство.

Единственным примером модели вычислительного устройства общего назначения в проекте NITTA может служить BusNetwork – модель устройства, представляющая сеть вычислительных устройств. На момент разработки не известно какие вычислительные устройства будут входить в состав BusNetwork и, соответственно, невозможно заранее предсказать как будет использована данная модель и какой у неё уровень параллелизма.

Для установки уровня параллелизма для BusNetwork были рассмотрены следующие альтернативы:

- устанавливать минимальный уровень параллелизма из моделей вычислительных устройств, входящих в состав;
- сигнализировать об ошибке при попытке узнать уровень параллелизма для BusNetwork.

В связи с тем, что на данный момент в проекте NITTA не поддерживается использование нескольких сетей и, как следствие, выделение нового вычислительного устройства типа BusNetwork невозможно, было принято решение сигнализировать об ошибке.

Всего в качестве метрик для оценки шага изменения микроархитектуры было выделено пять показателей (Таблица 2).

Таблица 2 – Метрики для оценки шага изменения микроархитектуры

Номер	Имя	Описание
1	mParallelism	уровень параллелизма вычислительного устройства
2	mRelatedRemains	количество непривязанных функций, которые могут быть выполнены вычислительным устройством
3	mMinPusForRemains	минимальное количество вычислительных устройств, которые могут выполнить не привязанные функции
4	mMaxParallels	максимальный уровень параллелизма алгоритма
5	mAvgParallels	средний уровень параллелизма алгоритма

Первые два показателя, по сути, являются исходными данными. Тем не менее, уровень параллелизма является важным, поскольку способен предотвратить добавление в микроархитектуру ненужных вычислительных устройств, а количество непривязанных функций позволяет избежать добавление вычислительных устройств на поздних этапах синтеза.

Третий показатель вычисляется путем подсчёта количества вычислительных устройств, способных выполнить функцию для каждой из непривязанных функций и взятием минимального количества. Этот показатель позволяет выявить ситуации, в которых исходный алгоритм содержит функции, для которых в текущей микроархитектуре нет вычислительных устройств способных их выполнить.

Четвертый и пятый показатели рассчитываются исходя из анализа графа потока данных. Рассмотрим пример простой функции  $f$ , написанной на языке Lua (Листинг 3).

### Листинг 3 – Функция f на языке Lua

```
function f(a1, a2, a3, a4)
  a5 = a1 + a2
  a6 = a3 * a4
  a7 = a5 * a5
  f(a4, a5, a6, a7)
end
f(1, 1, 2, 3)
```

Данная функция будет преобразована в следующий граф потока данных (Рисунок 12).

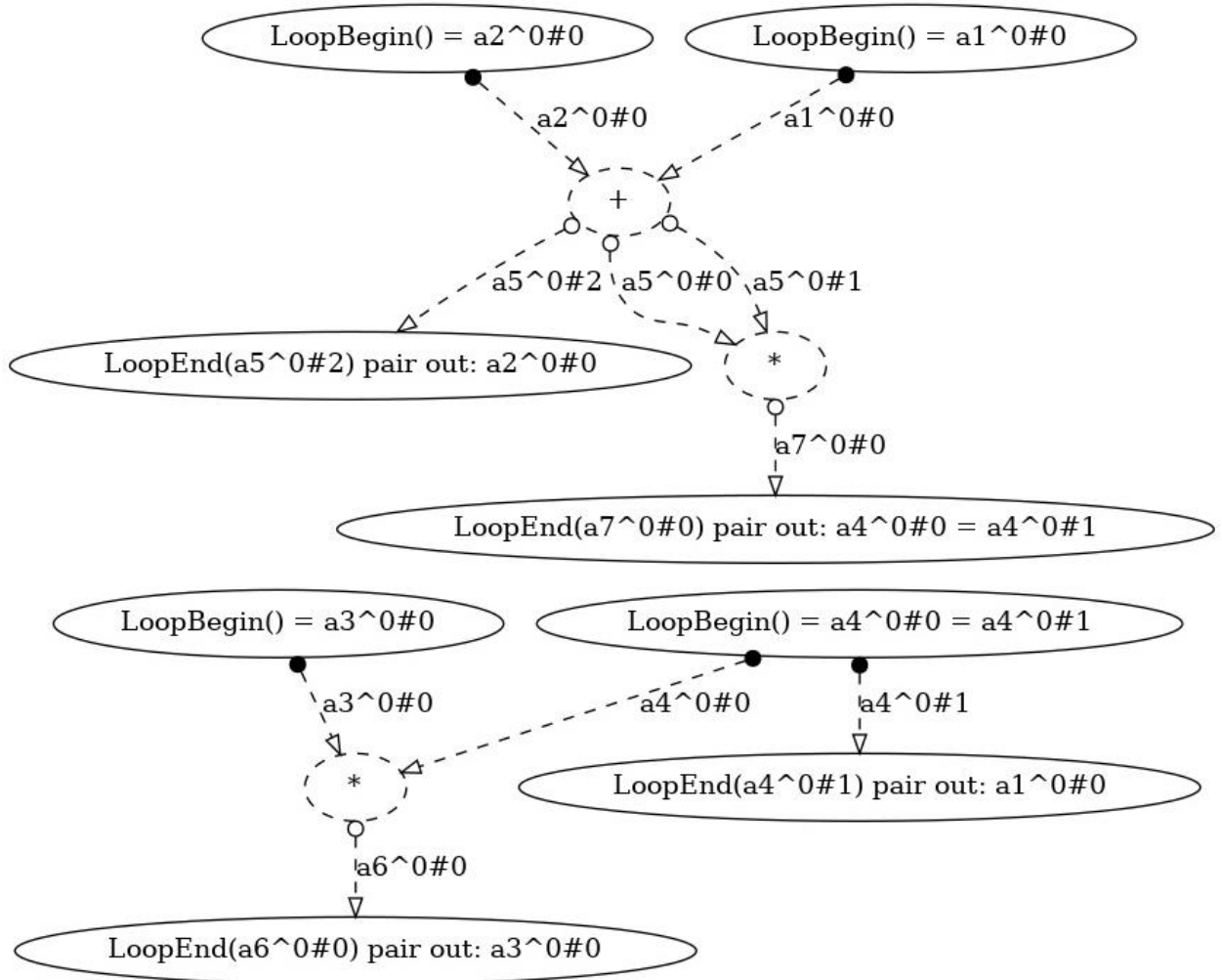


Рисунок 12 – Граф потока данных функции f

Исходя из входных и выходных параметров функций и опираясь на условия Бернштейна [12], выполнение функций можно разделить на этапы выполнения таким образом, что в рамках одного этапа функции могут быть выполнены параллельно.

Условия Бернштейна гласят, что две операции могут быть выполнены параллельно если

$$M(u) \cap M(v) = M(u) \cap R(v) = R(u) \cap M(v) = \emptyset, \quad (2)$$

где  $M(u)$  – множество ячеек памяти, которые редактирует операция  $u$ ,  $M(v)$  – множество ячеек памяти, которые редактирует операция  $v$ ,  $R(u)$  – множество ячеек памяти, которые читает операция  $u$ ,  $R(v)$  – множество ячеек памяти, которые читает операция  $v$ .

В соответствии с этим мы получаем четыре последовательных этапа (Рисунок 13).

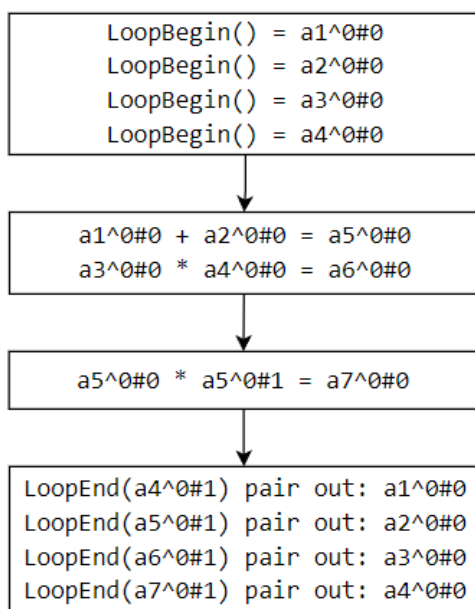


Рисунок 13 – Потенциальные места параллельного вычисления

Таким образом, для сумматора и умножителя максимальная степень параллелизма алгоритма равна 1, а для вычислительного устройства FRAM – 4. Средняя степень параллелизма алгоритма для вычислительных устройств, соответственно, равняется 0.25, 0.25 и 2.

Такой метод расчета уровня параллелизма не является точным, поскольку функции выполняются разное количество времени. Если учесть, что функция сложения выполняется намного быстрее функций умножения, то

используя два умножителя, вычислительный процесс можно было бы организовать следующим образом (Рисунок 14).

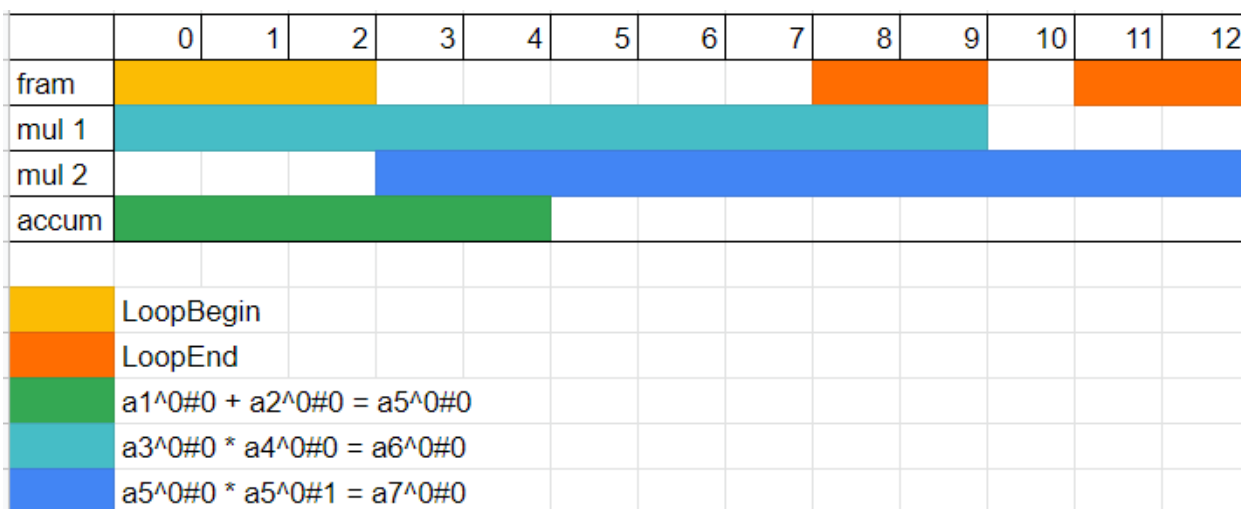


Рисунок 14 – Вычисление функции f с использованием двух умножителей

Однако данный подход не учитывает время выполнения функций, поэтому вычисления будут организованы следующим образом (Рисунок 15).

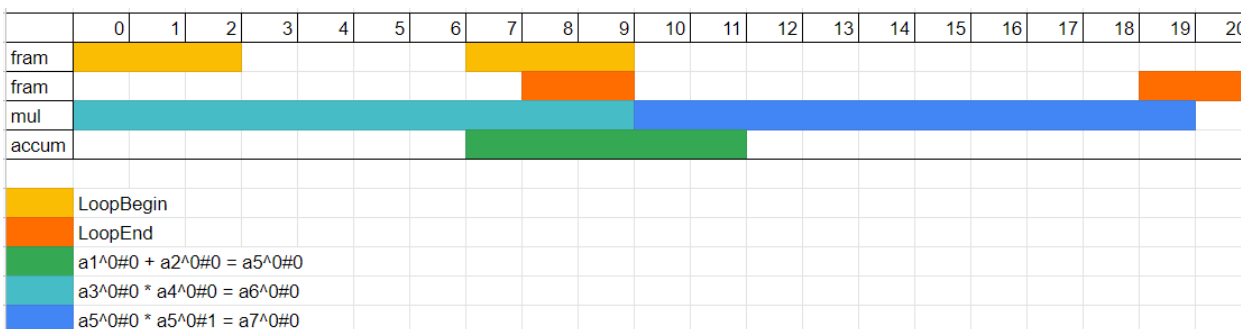


Рисунок 15 – Вычисление функции f с использованием одного умножителя

Два FRAM на рисунке изображены потому, что FRAM может выполнять функции параллельно и обозначение этого на одной линии было бы запутанным. В действительности подразумевается наличие в микроархитектуре только одного FRAM.

Для того, чтобы можно было определить, что выполнение двух функций с разных этапов выполнения может пересекаться во времени, необходимо обладать информацией о длительности выполнения функций вычислительными устройствами. Эту информацию можно получить в результате привязки функций к вычислительным устройствам.

Следовательно, информацию о времени выполнения функций можно собрать непосредственно перед началом процесса синтеза.

Другим вариантом является сбор информации вспомогательной программой. Такое решение ускорит работу NITTA, однако потребует поддержки актуальности информации о времени выполнения функций.

В рамках данной работы учет времени выполнения функций не был реализован, поскольку целью работы является внедрение, а не оптимизация шага изменения микроархитектуры в процессе синтеза.

Также не были реализованы метрики, специфичные для конкретных вычислительных устройств. Это означает, что если в процессе синтеза для выделения будут доступны два вычислительных устройства с разным набором характеристик, но одного типа, то выбор одного из них будет произведен случайным образом.

Тем не менее, разработанные метрики можно использовать для вычислительных устройств:

- для которых разработка специфичных метрик экономически не целесообразна (затраты на разработку несоизмеримы с получаемой выгодой) или не имеет смысла;
- для которых разработка еще не закончена (в качестве временного решения).

### **2.3 Оценка шага по изменению микроархитектуры**

При расчете оценки шага изменения архитектуры следует учитывать наличие приоритетных шагов. Такие шаги должны быть исполнены в первую очередь, поэтому они имеют константную оценку 5000 или больше. Такое значение выбрано поскольку оценка других шагов не может превышать это значение. Примерами приоритетных шагов служат:



- оптимизация вычислений аккумулятора;
- разрешение взаимоблокировок (Deadlock);
- свёртка констант (Constant folding).

Шаги изменения микроархитектуры, преследующие цель обеспечения возможности выполнения заданного алгоритма, относятся к приоритетным, поскольку их выполнение обязательно для успешного окончания процесса синтеза, а их невыполнение уменьшает количество вариантов действий. Следовательно, при минимальном количестве вычислительных устройств, которые могут выполнить непривязанные функции равным нулю, мы можем оценивать шаг константным значением 5000.

Стоит отметить, что оценка не может быть больше, чем у шага свёртки констант, поскольку это может привести к выделению лишних вычислительных устройств (если все сложения в исходном коде применяются только к константам, то наличие аккумулятора избыточно).

Шаги изменения микроархитектуры, нацеленные на ускорение вычислительного процесса, следует выполнять после применения оптимизаций, следовательно они не являются приоритетными. Из этого следует, что граф потока данных практически не будет меняться и, как следствие, не будут меняться метрики, основывающиеся на нем (оценки степени параллелизма алгоритма). Также не будет меняться тип параллелизма для вычислительных устройств. Из чего следует, что все решения об изменении микроархитектуры могут быть приняты перед привязкой функций к вычислительным устройствам.

Следовательно, оценку таких шагов можно свести к проверке прохода пороговых значений для метрик. Например, если средний уровень параллелизма для функции умножения больше 2, то мы оцениваем этот шаг добавления умножителя в 4900 – число, меньшее 5000, чтобы шаг применялся после оптимизаций и большее, чем оценка шага привязки функций. Если пороговое значение меньше, то мы оцениваем этот шаг в -1, чтобы

вычислительное устройство не могло быть выделено после начала привязки функций.

Итоговый алгоритм функции оценки шага выделения вычислительного устройства получен опытным путем (Рисунок 16).

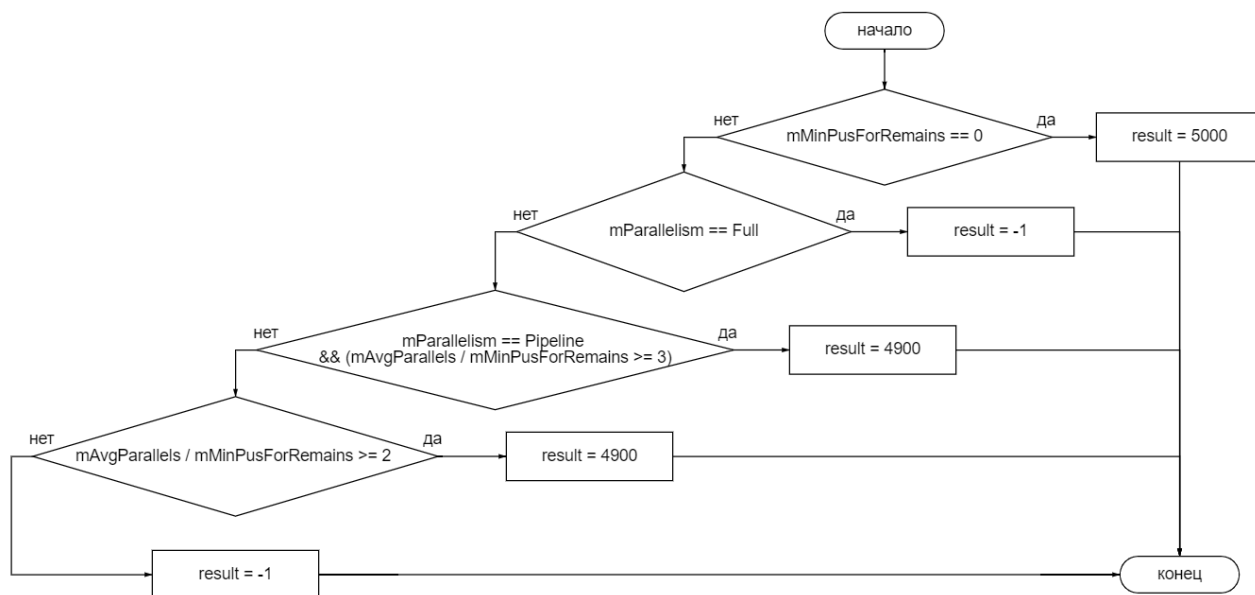


Рисунок 16 – Алгоритм оценки шага изменения микроархитектуры

## 2.4 Оценка результата синтеза по итогам обхода

При оценке конечных вершин дерева синтеза, метод синтеза учитывает только длительность цикла вычислений. Если вершин с одинаковой длительностью вычислений несколько, то будет выбрана первая из них. Рассмотрим следующий пример функции  $g$  (Листинг 4).

Листинг 4 – Функция  $g$  на языке Lua

```

function g(a)
  b = 1 + 1
  c = a * b
  g(c)
end
g(1)
  
```

Данная функция содержит сложение, однако после сверстки констант, сложений не останется, а значит для выполнения этой функции хватит FRAM и умножителя.

Однако в начале процесса синтеза сложение присутствует, а значит методу синтеза будет предложен шаг по добавлению аккумулятора в микроархитектуру. Поскольку метод синтеза обходит дерево, применяя не

только шаги с наилучшей оценкой, то в результате мы получим две конечные вершины, единственным отличием которых будет наличие в микроархитектуре аккумулятора, который никак не используется в вычислениях.

Таким образом, после добавления возможности автоматического синтеза микроархитектуры, итоговая оценка конечных вершин методом синтеза оказалась несостоятельной.

Для устранения этой проблемы в модель целевой системы и модель сети вычислительных устройств, была добавлена информация о количестве вычислительных устройств, входящих в состав. А метод синтеза был исправлен на выбор вершин с минимальной длительность цикла вычислений и последующим выбором вершины с минимальным количеством вычислительных устройств.

Такое решение было принято, как самое простое. Недостатками решения является то, что:

- количество вычислительных устройств может давать неправильное представление об итоговом размере целевой системы;
- решение не учитывает показатели энергоэффективности целевой системы;
- решение не учитывает приоритеты разработки (размер схемы может быть в приоритете перед производительностью).

## **2.5 Выводы**

В рамках данной главы были определены оценки и метрики, подлежащие реализации, а также рассмотрены проблемы, наиболее сильно влияющие на реализацию автоматического синтеза микроархитектуры. В соответствии с этим был составлен следующий план работ:

- реализация генерации шагов изменения микроархитектуры в рамках модели сети вычислительных устройств;
- реализация алгоритма разбиения исходного алгоритма на этапы выполнения;
- реализация расчета метрик и оценки шагов изменения микроархитектуры;
- изменение метода оценки конечных вершин методом синтеза;
- реализация загрузки набора прототипов вычислительных устройств из файлов конфигурации;
- реализация отображения шагов по изменению микроархитектуры в пользовательском интерфейсе;
- реализация модульных тестов;
- проведение интеграционного тестирования на примерах, присутствующих в проекте.

## 3 Реализация автоматического синтеза микроархитектуры

### 3.1 Реализация шагов по изменению микроархитектуры

Для генерации и исполнения шагов по изменению микроархитектуры была создана структура Allocation (Листинг 5), которая одновременно представляет и шаг, и решение по изменению микроархитектуры. Также был создан класс типов AllocationProblem (Листинг 6), который предоставляет методы для получения и исполнения возможных шагов по изменению микроархитектуры.

Листинг 5 – Структура данных Allocation

```
data Allocation tag = Allocation
  { -- |Tag of the BusNetwork where PU will be allocated
    bnTag :: tag
  , -- |Tag of the prototype that will be used for allocation
    puTag :: tag
  }
deriving (Generic, Eq)
```

Листинг 6 – Класс типов AllocationProblem

```
class AllocationProblem u tag | u -> tag where
  allocationOptions :: u -> [Allocation tag]
  allocationDecision :: u -> Allocation tag -> u
```

### 3.2 Разбиение исходного алгоритма на этапы выполнения

Для представления этапа выполнения исходного алгоритма была разработана структура ProcessWave (Листинг 7), содержащая информацию о функциях, которые выполняются на данном этапе, а также множество выходных переменных.

Листинг 7 – Структура данных ProcessWave

```
data ProcessWave v x = ProcessWave
  { -- |Functions that can be executed at this wave
    pwFs :: [F v x]
  , -- |Set of output variables related to the functions from this step
    pwOut :: S.Set v
  }
deriving (Show, Generic)
```

В качестве исходных данных алгоритм разбиения процесса вычислений на этапы (Рисунок 17) принимает набор переменных, готовых к использованию и список функций промежуточного представления исходного алгоритма.

Алгоритм разбиения упрощает то, что промежуточное представление исходного алгоритма обозначает повторное использование и изменение значения переменной разными переменными, то есть среди функций возможны пересечения только между входом и выходом. Соответственно, пересечения между входом и входом или выходом и выходом невозможны. Реализацию алгоритма разбиения процесса на этапы содержит ПРИЛОЖЕНИЕ А.

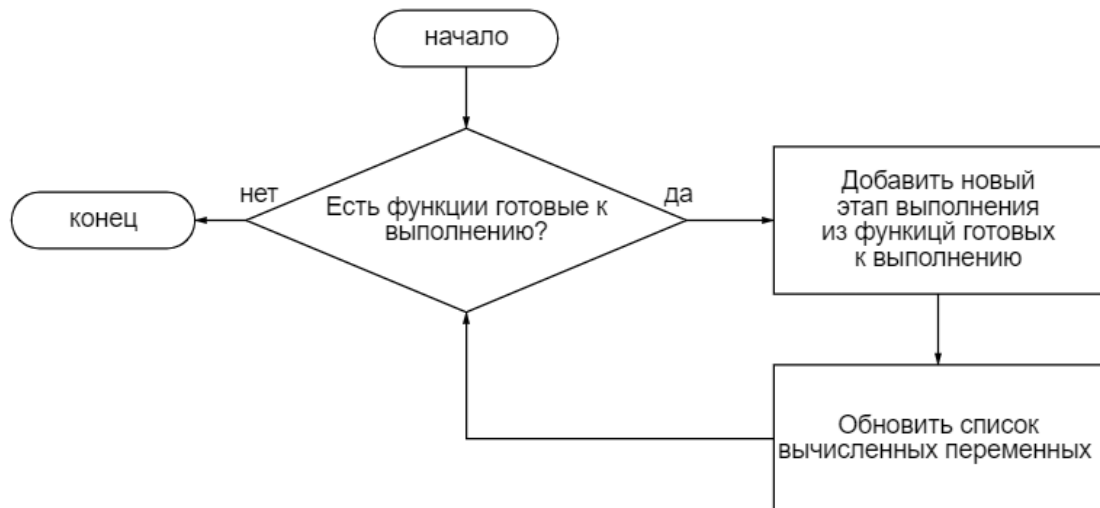


Рисунок 17 – Алгоритм разбиения процесса на этапы вычислений

### 3.3 Расчет метрик и оценка изменения микроархитектуры

Для хранения метрик шага изменения микроархитектуры была реализована структура AllocationMetrics (Листинг 8).

Листинг 8 – Структура данных AllocationMetrics

```

data AllocationMetrics = AllocationMetrics
  { -- |PU prototype parallelism type
    mParallelism :: ParallelismType
  , -- |The number of remaining functions that can be bound to pu
    mRelatedRemains :: Float
  , -- |The minimum number of PUs for each of the remaining functions that
    can process it
    mMinPusForRemains :: Float
  , -- |The maximum number of functions that could be processed in parallel
    mMaxParallels :: Float
  , -- |The number of functions that can be processed in parallel on average
    mAvgParallels :: Float
  }
  deriving (Generic)
  
```

Для определения типа поддерживаемого параллелизма вычислительного устройства в класс типов ProcessorUnit была добавлена функция

parallelismType, возвращающая тип данных ParallelismType для указанной модели вычислительного устройства (Листинг 9).

### Листинг 9 – Класс типов ProcessorUnit и структура ParallelismType

```
class (VarValTime v x t) => ProcessorUnit u v x t | u -> v x t where
  -- If the processor unit can execute a function, then it will return the PU
  -- model with already bound function (only registration, actual scheduling
  -- will be happening later). If not, it will return @Left@ value with a
  -- specific reason (e.g., not support or all internal resources is over).
  tryBind :: F v x -> u -> Either String u

  -- Get a computational process description. If the processor unit embedded
  -- another PUs (like "NITTA.Model.Networks.Bus"), the description should
  -- contain process steps for all PUs.
  --
  -- 'ProcessStepID' may change from one call to another.
  process :: u -> Process t (StepInfo v x t)

  -- |Indicates what type of parallelism is supported by 'ProcessorUnit'
  parallelismType :: u -> ParallelismType
  parallelismType _ = None

  -- |Provide the processor unit size. Now it's just the number of
  subprocessors
  puSize :: u -> Float
  puSize _ = 1

-- |Processor unit parallelism type
data ParallelismType
= -- |All operations can be performed in parallel mode
  Full
| -- |All operations can be performed in pipeline mode
  Pipeline
| -- |Other processor units
  None
deriving (Show, Generic, Eq)
```

Для разных моделей вычислительных устройств функция была реализована следующим образом:

- Divider – возвращается значение Pipeline;
- FRAM – возвращается значение Full;
- BusNetwork – возвращается сообщение об ошибке;
- для всех остальных вычислительных устройств возвращается значение None.

Расчет остальных метрик осуществлялся на основе информации, предоставляемой моделью сети вычислительных устройств и этапами выполнения исходного алгоритма (массивом структур ProcessWave).

Итоговый метод расчета метрик шага изменения микроархитектуры содержит Листинг 10.

### Листинг 10 – Метод расчета метрик шага изменения микроархитектуры

```
parameters
  SynthesisState
  { sTarget = TargetSystem{mUnit}
  , processWaves
  , numberOfProcessWaves
  }
  Allocation{puTag} _ =
  let pus = M.elems $ bnPus mUnit
      tmp = bnPUPrototypes mUnit M.! puTag
      mParallelism PUPrototype{pProto} = parallelismType pProto
      canProcessTmp PUPrototype{pProto} f = allowToProcess f pProto
      canProcessPU PU{unit} f = allowToProcess f unit
      relatedRemains = filter (canProcessTmp tmp) $ bnRemains mUnit
      fCountByWaves = map (\ProcessWave{pwFs} -> length $ filter
(canProcessTmp tmp) pwFs) processWaves
      in AllocationMetrics
        { mParallelism = mParallelism tmp
        , mRelatedRemains = fromIntegral $ length relatedRemains
        , mMinPusForRemains = fromIntegral $ foldr (min . (\f -> length $
filter (`canProcessPU` f) pus)) (maxBound :: Int) relatedRemains
        , mMaxParallels = fromIntegral $ maximum fCountByWaves
        , mAvgParallels = (fromIntegral (sum fCountByWaves) :: Float) /
(fromIntegral numberOfProcessWaves :: Float)
        }
```

### 3.4 Изменение метода оценки результатов синтеза

Чтобы при оценке конечных вершин метод синтеза учитывал количество использованных вычислительных устройств в класс типов ProcessorUnit был добавлен метод puSize. Реализация метода:

- по умолчанию возвращает 1;
- для модели сети вычислительных устройств возвращается сумма размеров устройств, входящий в состав сети;
- для модели целевой системы возвращается размер модели сети вычислительных устройств (потому что множественные сети на данный момент не поддерживаются).

В метод поиска лучшей конечной вершины было добавлено использование метода puSize, чтобы учитывать количество вычислительных устройств входящих в модель целевой системы (Листинг 11).



## Листинг 11 – Метод поиска лучшей конечной вершины

```
bestLeaf :: (VarValTime v x t, UnitTag tag)
  => DefTree tag v x t
  -> [DefTree tag v x t]
  -> DefTree tag v x t
bestLeaf tree leafs =
  let successLeafs = filter (\node -> isComplete node && isLeaf node) leafs
      target = sTarget . sState
  in case successLeafs of
    _ : _ -> minimumOn (\l -> (processDuration $ target l, puSize $
target l)) successLeafs
    [] -> headDef tree leafs
```

### 3.5 Конфигурирование набора вычислительных устройств

В качестве формата конфигурационного файла для задания микроархитектуры в проекте используется формат TOML [13]. Для возможности задания прототипов в конфигурацию вычислительных устройств было добавлено булево поле `proto`. Соответственно, при истинном значении этого параметра модель вычислительного устройства помещается в набор прототипов.

Также при загрузке конфигурации прототипов учитывается в какой сети вычислительных устройств задан прототип. Таким образом, пользователю предоставляется возможность ограничить использование прототипа выбранной вычислительной сетью.

Для возможности ограничения числа использований прототипа вычислительного устройства была добавлена поддержка шаблонов имен вычислительных устройств:

- если имя вычислительного устройства содержит подстроку “{x}”, то при добавлении в микроархитектуру подстрока будет заменена на индекс, а прототип не будет удален из набора доступных прототипов;
- иначе после добавления в микроархитектуру прототип удаляется из набора.

Листинг 12 демонстрирует пример конфигурации.

## Листинг 12 – Пример конфигурации микроархитектуры

```
type = "fx32.32"  
ioSync = "Sync"  
  
[[networks]]  
name = "net1"  
  
[[networks.pus]]  
type = "Fram"  
name = "fram{x}"  
size = 32  
proto = true  
  
[[networks.pus]]  
type = "Accum"  
name = "accum{x}"  
isInt = true  
proto = true
```

### 3.6 Изменение пользовательского интерфейса

Пользовательский интерфейс проекта NITTA написан с использованием библиотеки React и представлен в виде веб-интерфейса.

Для отображения шагов по изменению микроархитектуры в раздел Subforest пользовательского интерфейса была добавлена таблица, которая содержит набор шагов, предоставляющих варианты изменения микроархитектуры, а также описание шага, его оценку, метрики (Рисунок 18).



The screenshot shows the NITTA web interface. On the left, there is a circuit diagram with nodes and connections. On the right, there are several tables and sections:

- Binding**: A table with columns: s..., Z(d), description, crit, lock, wave, outputs, alt, rest, newDF, newBlnd, [inputs].
- Refactor**: NOTHING
- Allocation**: A table with columns: s..., Z(d), description, parallelism, related remains, max parallels, avg parallels, min PUs for remains.
- Dataflow**: NOTHING
- Other**: NOTHING

The Allocation table contains the following data:

s...	Z(d)	description	parallelism	related remains	max parallels	avg parallels	min PUs for remains
0>	5000	net1 ← accum{x}	None	1	1	0.5	0
1>	-1	net1 ← fram{x}	Full	2	2	1	1

Рисунок 18 – Пользовательский интерфейс NITTA

Описание шага изменения микроархитектуры генерируется по шаблону:  
“имя сети ← имя прототипа”.

### 3.7 Тестирование разработанного решения

По итогам разработки была реализована группа модульных тестов (ПРИЛОЖЕНИЕ Б). Таблица 3 демонстрирует результаты выполнения тестов.

Таблица 3 – Результаты проведения модульного тестирования

Номер	Название теста	Результат
1	target system: manual synthesis, allocation works correctly	Пройден
2	target system: autosynthesis, allocate required PUs	Пройден
3	target system: autosynthesis, allocation comes after constant folding	Пройден
4	target system: autosynthesis, allocation comes after accum optimization	Не пройден

Тест 4 не был пройден из-за того, что системе не удалось завершить синтез. Данный тест был проведен повторно с жестко заданной микроархитектурой, результат проведения оказался таким же, из чего был сделан вывод, что проблема проистекает не из автоматического синтеза микроархитектуры.

Для того, чтобы выяснить почему синтез не может быть завершен, функция написанная на Lua была упрощена до следующего вида (Листинг 13).

Листинг 13 – Функция sum на языке Lua

```
function sum(x1, x2)
  x1 = x1 + x2
  x2 = x2 + 3
  sum(x1, x2)
end
sum(1, 2, 3)
```

После этого пример был запущен в пользовательском интерфейсе, чтобы можно было определить в каком состоянии находится граф потока данных после остановки процесса синтеза (Рисунок 19).

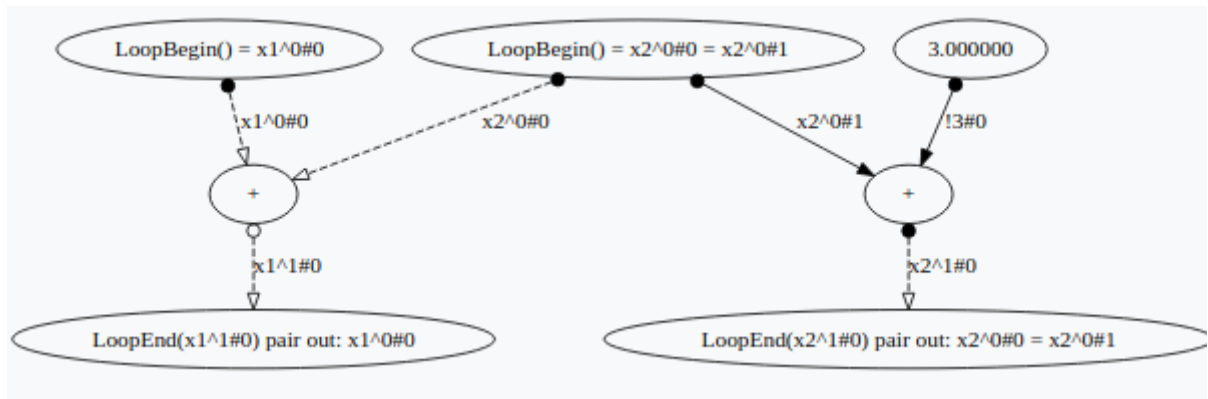


Рисунок 19 – DFG в момент остановки метода синтеза

Рисунок 20 показывает, что все функции уже привязаны к вычислительным устройствам, а также запланированы пересылки переменных  $x2^0\#1$  и  $!3\#0$ . Для успешного завершения процесса синтеза остается только запланировать пересылку оставшихся переменных:  $x1^0\#0$ ;  $x1^1\#0$ ;  $x2^0\#0$ ;  $x2^1\#0$ .

Однако, пересылка  $x1^0\#0$ ,  $x1^1\#0$ ,  $x2^0\#0$  невозможна, пока  $x2^1\#0$  не будет переслано из аккумулятора в FRAM. В то же время пересылка  $x2^1\#0$  невозможна потому, что она осуществляется в ячейку памяти, в которой хранится  $x2^0\#0$  и чье значение еще необходимо в дальнейших вычислениях. Таким образом, происходит взаимоблокировка двух вычислительных устройств, которая обычно решается с помощью отдельного шага синтеза – ResolveDeadlock, сохраняющего одно из значений в буфер. Соответственно, проблема заключается в том, что взаимоблокировка не была распознана и методу синтеза не было предложено соответствующего варианта действий. На странице проекта на сайте GitHub было оставлено сообщение о данной проблеме [14].

Также стоит обратить внимание, что в тесте под номером 2 SPI добавляется в модель сети вычислительных устройств не как прототип, а как готовое вычислительное устройство. Ограничение возможности добавить в модель сети прототип SPI вызвано тем, что на данный момент невозможно использовать более одного SPI из-за ошибки в реализации вычислительного устройства. Сообщение о данной проблеме также было оставлено на странице проекта на сайте GitHub [15].

Также было проведено ручное тестирование работы NITTA на примерах программ, присутствующих в проекте. Исходный код программ и конфигурация микроархитектуры целевой системы содержится в ПРИЛОЖЕНИИ В. Результаты тестирования демонстрирует Таблица 4.

Таблица 4 – Результаты ручного тестирования автоматического синтеза

Номер	Имя программы	Результат синтеза	Устройства, добавленные в микроархитектуры, тип/кол-во
1	constantFolding.lua	успешный	аккумулятор/1 FRAM/1
2	counter.lua	успешный	аккумулятор/1 FRAM/1
3	sum.lua	успешный	аккумулятор/1 FRAM/1
4	double_receive.lua	успешный	аккумулятор/1
5	fibonacci.lua	успешный	FRAM/2 аккумулятор/1
6	shift.lua	успешный	FRAM/1 сдвиговый регистр/1
7	spi1.lua	успешный	FRAM/1
8	spi2.lua	успешный	аккумулятор/1
9	spi3.lua	успешный	аккумулятор/1
10	teacup.lua	успешный	аккумулятор/1 FRAM/1 умножитель/1 делитель/1
11	pid.lua	неудачный	-

Из результатов видно, что синтез программы pid.lua окончился неудачей. Это связано с ошибкой логики планирования пересылки данных между вычислительными устройствами. Эта ошибка происходит на тридцать

шестом шаге синтеза. Дальнейшие шаги синтеза, доступные в этот момент, демонстрирует Таблица 5.

Таблица 5 – Доступные шаги синтеза в момент ошибки

Оценка	Источник	Описание
1982	net1_fram1	from: net1_fram1 1) $P^0$ net1_accum1 @ 13 ... 13
1982	net1_fram4	from: net1_fram4 1) $I^0$ net1_accum1 @ 13 ... 13
1980	net1_mul2	from: net1_mul2 1) $D^0$ net1_accum1 @ 15 ... 15

На графе потока данных (Рисунок 21) в этот момент видно, что к аккумулятору привязано две функции. Первая принимает на вход  $I^0$  и  $_0$  и записывает результат в  $I^1$ , вторая принимает на вход  $I^1$ ,  $D^0$  и  $P^0$  и записывает результат в  $PID^0$ . То есть между функциями есть зависимость, и первая должна быть выполнена раньше второй.

Ошибка заключается в том, что процесс синтеза планирует пересылку переменной  $P^0$  в аккумулятор, когда  $I^1$  еще не вычислена. Таким образом, аккумулятор становится заблокированным, поскольку переменную  $I^1$  тоже должен вычислить он.

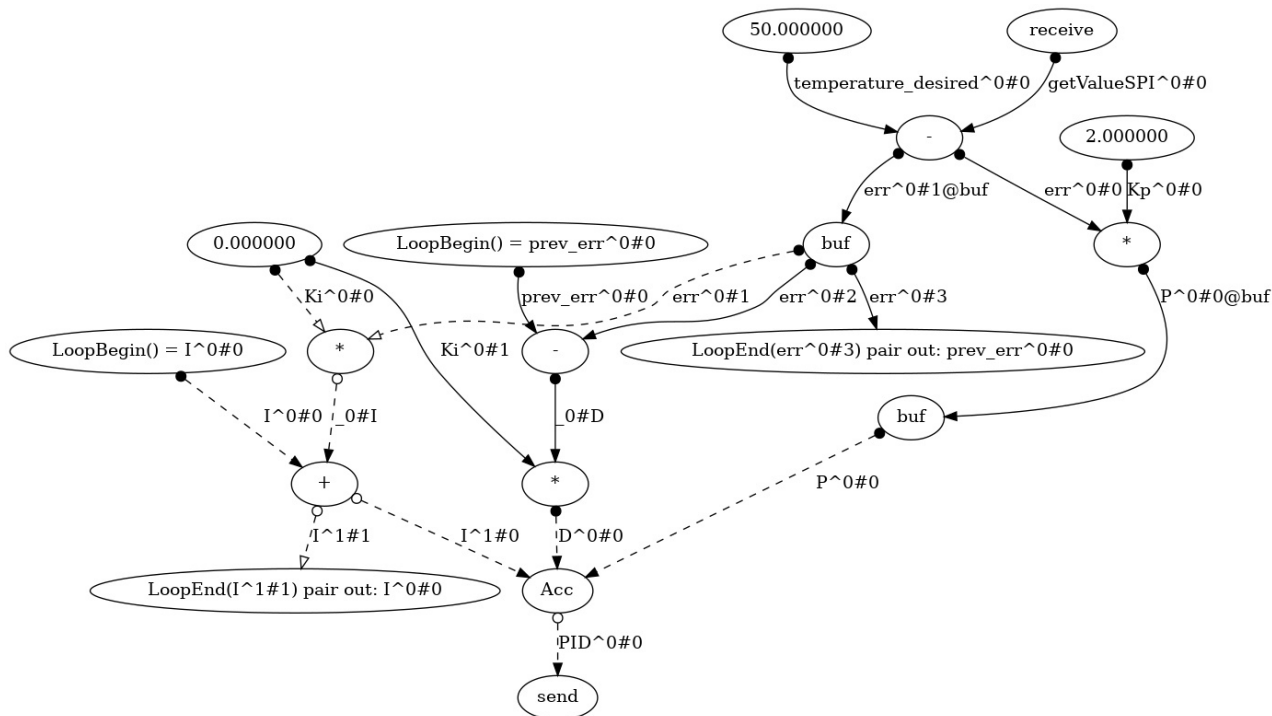


Рисунок 21 – DFG в момент ошибки

Стоит отметить, что если процесс синтеза выбрал в качестве следующего шага пересылку переменной  $I^0$ , то синтез завершился успехом.

Таким образом, недостатком метода синтеза является то, что при планировании пересылки данных между вычислительными устройствами не учитываются зависимости исполняемых функций, что приводит к взаимоблокировке. Поскольку данная проблема возникает после привязки функций к вычислительным блокам, она не может быть решена добавлением новых вычислительных устройств в микроархитектуру.

Соответственно, для решения данной проблемы было решено добавить в метрики шагов планирования пересылки данных параметр отображающий этап вычислений прикладного алгоритма.

Таким параметром стал `rMinWave`, отображающий минимальный номер этапа вычислений прикладного алгоритма, к которому относятся функции принимающие пересылаемые данные в качестве входных параметров. Для расчета этапов вычисления прикладного алгоритма была повторно использована структура `ProcessWave`. Листинг 14 демонстрирует итоговые

методы расчета и оценки шагов планирования пересылки данных между вычислительными устройствами.

#### Листинг 14 – Расчет метрик и оценка планирования пересылки данных

```
parameters
  SynthesisState
  { transferableVars
  , sTarget
  , processWaves
  }
  DataflowSt{dfSource, dfTargets} _ =
  let TimeConstraint{tcAvailable, tcDuration} = epAt $ snd dfSource
      vs = unionsMap (variables . snd) dfTargets
      lvs = length vs
      wave = length . takeWhile (\ProcessWave{pwFs} -> lvs == length (vs
`S.difference` unionsMap inputs pwFs)) $ processWaves
  in DataflowMetrics
  { pWaitTime = fromIntegral (inf tcAvailable)
  , pRestrictedTime = fromEnum (sup tcDuration) /= maxBound
  , pNotTransferableInputs =
      let fs = functions $ mUnit sTarget
          affectedFunctions = filter (\f -> not $ null (inputs f
`S.intersection` vs)) fs
          notTransferableVars = map (\f -> inputs f S.\
transferableVars) affectedFunctions
          in map (fromIntegral . length) notTransferableVars
      , pMinWave = fromIntegral wave :: Float
  }

estimate SynthesisState{numberOfDataflowOptions} _o _d
  DataflowMetrics
  { pWaitTime
  , pNotTransferableInputs
  , pRestrictedTime
  , pMinWave
  } =
  2000
  + (numberOfDataflowOptions >= threshold) <?> 1000
  + pRestrictedTime <?> 200
  - sum pNotTransferableInputs * 5
  - pWaitTime
  - pMinWave
```

В результате повторного тестирования разработанного решения на примерах программ, присутствующих в проекте НИТТА, все результаты оказались положительными. Отличий в наборе вычислительных устройств, добавленных в микроархитектуру, не было. При работе с pid.lua в микроархитектуру были добавлены следующие вычислительные устройства:



- два FRAM;
- один аккумулятор;
- один умножитель.

### 3.7.1 Генерация отчета о тестовом покрытии

Для создания отчета о тестовом покрытии использовалась утилита НРС (Haskell program coverage) [16]. Она анализирует тестовое покрытие на трех уровнях:

- объявления – анализируется обращение к объявлениям верхнего уровня;
- альтернативные пути исполнения – анализируются операторы ветвления;
- выражения – анализируются все выражения;

Также если отчет представлен в формате HTML, то в нем будет содержаться исходный код, в котором будут выделены (Рисунок 22):

- желтым цветом – выражения, которые не были покрыты тестами;
- зеленым цветом – условные выражения, всегда возвращающие истину;
- красным цветом – условные выражения, всегда возвращающие лож.

```
execBuilder :: Ord v => Builder v x -> Int -> Builder v x
execBuilder builder@Builder{pwRemains} prev
  | S.null pwRemains = builder
  | S.size pwRemains == prev = error "Process waves construction stuck in a loop"
  | otherwise = execBuilder (foldl applyRemaining builder pwRemains) $ S.size pwRemains
```

Рисунок 22 – Пример отчета НРС

Ниже представлено тестовое покрытие модулей, содержащих основную логику разработанного решения (Таблица 6).

Таблица 6 – Отчет о тестовом покрытии

Модуль	Объявления верхнего уровня		Альтернативные пути исполнения		Выражения	
	%	покрыто/ всего	%	покрыто/ всего	%	покрыто/ всего
NITTA.Intermediate.Analysis	64	9/14	81	9/11	94	135/143
NITTA.Model.Networks.Bus	69	48/69	72	36/50	90	2110/2329
NITTA.Model.Problems.Allocation	50	2/4	-	0/0	81	9/11
NITTA.Model.TargetSystem	87	21/24	-	0/0	77	74/96
NITTA.Synthesis.Explore	81	9/11	25	3/12	73	205/278
NITTA.Synthesis.Method	91	11/12	70	12/17	82	182/221
NITTA.Synthesis.Steps.Allocation	37	3/8	60	3/5	78	79/101

Генерация и публикация отчета о тестовом покрытии была автоматизирована с помощью GitHub Actions [17]. Публикация отчета осуществляется с помощью GitHub Pages [18]. Таким образом, полный отчет о тестовом покрытии проекта доступен на странице проекта на сайте GitHub [19].

### 3.8 Выводы

В соответствии с решениями, принятыми во второй главе, в процесс синтеза проекта NITTA был внедрен шаг автоматического изменения микроархитектуры. Также для обеспечения возможности управления этим процессом были внесены изменения:

- в пользовательский интерфейс – для отображения истории изменения и дальнейших вариантов синтеза;
- в структуру конфигурации микроархитектуры – для обеспечения возможности настройки ограничений процесса изменений микроархитектуры.

Разработанное решение было протестировано, в результате чего в существующей логике процесса синтеза был выявлен ряд проблем. Для

каждой проблемы были оставлены сообщения на странице проекта NITTA на сайте GitHub. Часть обнаруженных проблем была решена, после чего результаты процесса синтеза на примерах программ присутствующих стали успешными. Также, с помощью GitHub Actions и GitHub Pages, была автоматизирована генерация и публикация отчета о тестовом покрытии.

## ЗАКЛЮЧЕНИЕ

В результате работы, был проведен анализ процесса синтеза проекта NITTA и выделены ключевые вопросы внедрения генерации микроархитектуры в процесс синтеза.

Для каждого вопроса были рассмотрены альтернативы, их плюсы и минусы. В соответствии с ними, для каждого ключевого вопроса было выбрано решение, после чего был составлен план интеграции возможности изменения микроархитектуры в процессе синтеза.

В соответствии с составленным планом интеграции, в проекте NITTA была реализована возможность изменения микроархитектуры в процессе синтеза. Также была реализована возможность настройки процесса изменения микроархитектуры с помощью файлов конфигурации и отображение соответствующих шагов синтеза в пользовательском интерфейсе.

Тестирование выявило ряд недостатков процесса синтеза, не связанных с логикой изменения микроархитектуры. При этом часть из этих недостатков была устранена. Успешные результаты итогового тестирования показывают фактическую применимость разработанного решения. Однако стоит отметить, что область применения разработанного решения ограничена:

- автоматическое добавление в микроархитектуру вычислительных устройств SPI не поддерживается из-за ошибок в реализации;
- оценка шага изменения микроархитектуры целевой системы не учитывает особенности вычислительных устройств (например, для FRAM не учитывается размер);
- у пользователя отсутствует возможность приоритизации характеристик целевой системы;
- проект не поддерживает работу с несколькими сетями вычислительных устройств, поэтому изменение микроархитектуры целевой системы на уровне сетей вычислительных устройств не было реализовано.

Таким образом, дальнейшими направлениями развития проекта НИТТА в области процесса синтеза являются:

- реализация автоматического добавления в микроархитектуру вычислительных устройств и портов ввода/вывода;
- устранение существующих ошибок синтеза;
- реализация использования нескольких сетей ВУ;
- реализация шагов синтеза для привязки функций к сетям ВУ;
- реализация учета специфических параметров вычислительных устройств при добавлении их в микроархитектуру.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Dennard R.H. et al. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions // Proceedings of the IEEE. 1999. Vol. 87, № 4. P. 668–678.
2. Denning P.J., Lewis T.G. Exponential laws of computing growth // Commun ACM. 2016. Vol. 60, № 1. P. 54–65.
3. Liu L. et al. A Survey of Coarse-Grained Reconfigurable Architecture and Design // ACM Computing Surveys. 2020. Vol. 52, № 6. P. 1–39.
4. Nowatzki T. et al. Pushing the limits of accelerator efficiency while retaining programmability // 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2016. P. 27–39.
5. Penskoi A.V. et al. High-level synthesis system based on hybrid reconfigurable microarchitecture // Scientific and Technical Journal of Information Technologies, Mechanics and Optics. 2019. Vol. 19, № 2. P. 306–313.
6. Penskoi A. Synthesis Method for CGRA Processors based on Imitation Model // 2021 10th Mediterranean Conference on Embedded Computing (MECO). IEEE, 2021. P. 1–4.
7. Aleksandr Penskoi, Ivan Perl. Hardware Accelerator for System Dynamic Modeling. Bergen: The 2020 Conference of the System Dynamics Society, 2020.
8. Jörg Henkel, Sri Parameswaran. Designing Embedded Processors / ed. Henkel J., Parameswaran S. Dordrecht: Springer Netherlands, 2007.
9. Corporaal H., Arnold M. Using Transport Triggered Architectures for Embedded Processor Design // Integrated Computer-Aided Engineering. 1998. Vol. 5, № 1. P. 19–38.
10. Aleksandr Penskoi. NITTA internal [Electronic resource]. 2021. URL: <https://github.com/ryukzak/nitta/wiki/NITTA-internal> (accessed: 13.04.2022).
11. Cardelli L., Wegner P. On understanding types, data abstraction, and polymorphism // ACM Computing Surveys. 1985. Vol. 17, № 4. P. 471–523.

12. Kielmann T. et al. Bernstein's Conditions // Encyclopedia of Parallel Computing. Boston, MA: Springer US, 2011. P. 130–134.
13. Tom Preston-Werner, Pradyun Gedam. Tom's Obvious Minimal Language [Electronic resource]. URL: <https://toml.io/en/> (accessed: 02.05.2022).
14. Vitaliy Zakusilo. Completion of synthesis fails [Electronic resource]. URL: <https://github.com/ryukzak/nitta/issues/217> (accessed: 03.05.2022).
15. Vitaliy Zakusilo. Multiple SPIs lead to errors [Electronic resource]. URL: <https://github.com/ryukzak/nitta/issues/194> (accessed: 03.05.2022).
16. Haskell program coverage [Electronic resource]. URL: [https://wiki.haskell.org/Haskell\\_program\\_coverage](https://wiki.haskell.org/Haskell_program_coverage) (accessed: 21.05.2022).
17. GitHub Actions [Electronic resource]. URL: <https://github.com/features/actions> (accessed: 21.05.2022).
18. GitHub Pages [Electronic resource]. URL: <https://pages.github.com/> (accessed: 21.05.2022).
19. NITTA Test Coverage [Electronic resource]. URL: [https://ryukzak.github.io/nitta/hpc/hpc\\_index.html](https://ryukzak.github.io/nitta/hpc/hpc_index.html) (accessed: 21.05.2022).

## ПРИЛОЖЕНИЕ А – Исходный код buildProcessWaves

```
buildProcessWaves :: (Var v, Val x) => [v] -> [F v x] -> [ProcessWave v x]
buildProcessWaves vars fs =
  let pwIn = S.fromList vars
      (loops, other) = L.partition isLoop fs
      beginning =
        [ ProcessWave
          { pwFs = loops
            , pwOut = unionsMap outputs loops
          }
        | not (null loops)
        ]
      builder =
        Builder
          { pwRemains = S.fromList other
            , pwGraph = beginning
            , pwIn
            , pwReadyIn = pwIn `S.union` unionsMap pwOut beginning
          }
  in pwGraph $ execBuilder builder 0

execBuilder :: Ord v => Builder v x -> Int -> Builder v x
execBuilder builder@Builder{pwRemains} prev
  | S.null pwRemains = builder
  | S.size pwRemains == prev = error "Process waves construction stuck in a
loop"
  | otherwise = execBuilder (foldl applyRemaining builder pwRemains) $
S.size pwRemains

applyRemaining :: Ord v => Builder v x -> F v x -> Builder v x
applyRemaining builder@Builder{pwRemains, pwGraph, pwIn, pwReadyIn} func =
  let fIn = inputs func
      fOut = outputs func
      pendingIn = S.difference fIn pwReadyIn
  in if not $ null pendingIn
      then builder
      else
        builder
          { pwReadyIn = S.union fOut pwReadyIn
            , pwGraph = insertF func (S.difference fIn pwIn) fOut
          }
  pwGraph
  , pwRemains = S.delete func pwRemains
  }

insertF :: Ord v => F v x -> S.Set v -> S.Set v -> [ProcessWave v x] ->
[ProcessWave v x]
insertF f fIn fOut []
  | null fIn = [ProcessWave{pwFs = [f], pwOut = fOut}]
  | otherwise = error "Cannot calculate process wave for the function"
insertF f fIn fOut (ps@ProcessWave{pwFs, pwOut} : pss)
  | null fIn = ps{pwFs = f : pwFs, pwOut = S.union fOut pwOut} : pss
  | otherwise = ps : insertF f (S.difference fIn pwOut) fOut pss
```



## ПРИЛОЖЕНИЕ Б – Исходный код модульных тестов

```
testGroup
  "Allocation synthesis step"
  [ unitTestCase "target system: manual synthesis, allocation works
correctly" def $ do
    setNetwork $
      Bus.defineNetwork "net1" ASync $ do
        Bus.addPrototype "fram{x}" FramIO
        Bus.addPrototype "accum" AccumIO
    setBusType pInt
    assignLua
      [__i|
        function sum(a)
          local d = a + 1
          sum(d)
        end
        sum(0)
      ]
    doAllocation "net1" "accum"
    doAllocation "net1" "fram{x}"
    assertAllocation 1 =<< mkAllocation "net1" "fram{x}"
    assertAllocation 1 =<< mkAllocation "net1" "accum"
    assertAllocationOptions =<< mkAllocationOptions "net1" ["fram{x}"]
    assertPU "net1_accum" (Proxy :: Proxy (Accum T.Text Int Int))
    assertPU "net1_fram1" (Proxy :: Proxy (Fram T.Text Int Int))
    synthesizeAndCoSim
  , unitTestCase "target system: autosynthesis, allocate required PUs" def
$ do
    setNetwork $
      Bus.defineNetwork "net1" ASync $ do
        Bus.addCustomPrototype "fram{x}" (framWithSize 32) FramIO
        Bus.addPrototype "accum{x}" AccumIO
        Bus.addPrototype "mul{x}" MultiplierIO
        Bus.add "spi" $ -- use addPrototype when
https://github.com/ryukzak/nitta/issues/194 will be fixed
        SPISlave
          { slave_mosi = InputPortTag "mosi"
            , slave_miso = OutputPortTag "miso"
            , slave_sclk = InputPortTag "sclk"
            , slave_cs = InputPortTag "cs"
          }
    setBusType pInt
    assignLua
      [__i|
        function counter(x1)
          send(x1)
          x2 = x1 + 1
          counter(x2)
        end
        counter(0)
      ]
    synthesizeAndCoSim
    assertAllocation 1 =<< mkAllocation "net1" "fram{x}"
    assertAllocation 1 =<< mkAllocation "net1" "accum{x}"
    assertAllocation 0 =<< mkAllocation "net1" "mul{x}"
  , unitTestCase "target system: autosynthesis, allocation comes after
constant folding" def $ do
    setNetwork $
      Bus.defineNetwork "net1" ASync $ do
        Bus.addCustomPrototype "fram{x}" (framWithSize 32) FramIO
        Bus.addPrototype "accum{x}" AccumIO
        Bus.addPrototype "mul{x}" MultiplierIO
    setBusType pInt
```

```

assignLua
  [__i|
    function mul3(x1)
      x1 = (1 + 1 + 1) * x1
      mul3(x1)
    end
    mul3(1)
  ]
synthesizeAndCoSim
assertAllocation 1 =<< mkAllocation "net1" "fram{x}"
assertAllocation 1 =<< mkAllocation "net1" "mul{x}"
assertAllocation 0 =<< mkAllocation "net1" "accum{x}"
, unitTestCase "target system: autosynthesis, allocation comes after
accum optimization" def $ do
  setNetwork $
    Bus.defineNetwork "net1" ASync $ do
      Bus.addCustomPrototype "fram{x}" (framWithSize 32) FramIO
      Bus.addPrototype "accum{x}" AccumIO
  setBusType pInt
  assignLua
    [__i|
      function sum(x1, x2, x3, x4, x5)
        x1 = x1 + x2 + x3 + x4
        x2 = x2 + 3
        x3 = x3 + 3
        x4 = x4 + 3
        x5 = x5 + 3
        sum(x1, x2, x3, x4, x5)
      end
      sum(1,2,3,4,5)
    ]
  synthesizeAndCoSim
  assertAllocation 1 =<< mkAllocation "net1" "fram{x}"
  assertAllocation 1 =<< mkAllocation "net1" "accum{x}"
  assertAllocation 0 =<< mkAllocation "net1" "mul{x}"
]

```

## ПРИЛОЖЕНИЕ В – Исходный код программ на языке Lua

### Конфигурация микроархитектуры:

```
type = "fx32.32"  
ioSync = "Sync"  
  
[[networks]]  
name = "net1"  
  
[[networks.pus]]  
type = "Fram"  
name = "fram{x}"  
size = 32  
proto = true  
  
[[networks.pus]]  
type = "Shift"  
name = "shift{x}"  
sRight = true  
proto=true  
  
[[networks.pus]]  
type = "Multiplier"  
name = "mul{x}"  
mock = true  
proto=true  
  
[[networks.pus]]  
type = "Accum"  
name = "accum{x}"  
isInt = true  
proto=true  
  
[[networks.pus]]  
type = "Divider{x}"  
name = "div"  
mock = true  
pipeline = 4  
proto=true  
  
[[networks.pus]]  
type = "SPI"  
name = "spi"  
mosi = "mosi"  
miso = "miso"  
sclk = "sclk"  
cs = "cs"  
isSlave = true  
bufferSize = 6  
bounceFilter = 0
```

### Исходный код программы constantFolding.lua:

```
function constantFolding(i)  
    local v = 1 + 2 + 3  
    local res = i + v + 3  
    constantFolding(res)  
end  
constantFolding(0)
```

### Исходный код программы counter.lua:

```

function counter(x1)
    send(x1)
    x2 = x1 + 1
    debug.trace(x1, x2)
    counter(x2)
end
counter(0)

```

#### Исходный код программы sum.lua:

```

function sum(a, b, c)
    local d = a + b + c
    sum(d, d, d)
end
sum(0,0,0)

```

#### Исходный код программы double\_receive.lua:

```

function sum()
    local a = receive()
    local x = a + a
    send(x)
    sum()
end
sum()

```

#### Исходный код программы fibonacci.lua:

```

function fib(a, b)
    b, a = a + b, b
    fib(a, b)
end
fib(0, 1)

```

#### Исходный код программы shift.lua:

```

function shift(a)
    local out = a << 9
    shift(out)
end
shift(1)

```

#### Исходный код программы spi1.lua:

```

function foo()
    local a = receive()
    local x = buffer(a)
    send(x)
    foo()
end
foo()

```

#### Исходный код программы spi2.lua:

```

function foo()
    local a = receive()
    local b = receive()
    local c = a + b
    send(c)
    foo()
end
foo()

```

#### Исходный код программы spi3.lua:

```

function foo()
    local a = receive()
    local b = receive()
    local c = a + a + b
    send(c)
    foo()
end
foo()

```

### Исходный код программы teacup.lua:

```

function teacup(time, temp_cup)
    local temp_ch = 10
    local temp_room = 70
    local time_step = 0.125

    send(time)
    send(temp_cup)

    time = time + time_step
    local acc = temp_room - temp_cup
    local temp_loss, _ = acc / temp_ch

    local delta = temp_loss * time_step
    temp_cup = temp_cup + delta

    teacup(time, temp_cup)
end
teacup(0, 180)

```

### Исходный код программы pid.lua:

```

function pid(I, prev_err)
    local Kp = 2
    local Ki = 0
    local temperature_desired = 50
    local getValueSPI = receive()
    err = temperature_desired - getValueSPI
    P = Kp * err
    I = I + Ki * err
    D = Ki * (err - prev_err) -- Kd * (err - prev_err)
    local PID = P + I + D
    send(PID)
    pid(I, err)
end
pid(0, 0)

```