Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“Национальный исследовательский университет ИТМО”

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

АВТОМАТИЗАЦИЯ ПОСТРОЕНИЯ МОДЕЛЕЙ ВЫЧИСЛИТЕЛЬНЫХ БЛОКОВ ДЛЯ СИСТЕМЫ ПРОЕКТИРОВАНИЯ СПЕЦИАЛИИРОВАННЫХ ВЫЧИСЛИТЕЛЕЙ

Автор____Ницер Ксения Александровна___ _____
(Фамилия, Имя, Отчество) (Подпись)

Направление подготовки (специальность)____09.03.01_____
(код, наименование)
«Информатика и вычислительная техника»_____

Квалификация _____бакалавр_____
(бакалавр, магистр, инженер)*

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

Санкт-Петербург, 2020 г.

Обучающийся___Ницер Ксения Александровна_____
(ФИО полностью)
Группа____P3401_____Факультет/институт/кластер___ПИиКТ_____

Направленность (профиль), специализация __09.03.01 «Информатика и вычислительная техника»_____

Консультант (ы):
а) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

б) _____ _____
(Фамилия, И., О., ученое звание, степень) (Подпись)

ВКР принята “____”_____20 ____г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты “____”_____20 ____г.

Секретарь ГЭК _____ _____
(ФИО) (подпись)

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

“НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ”

УТВЕРЖДАЮ

Руководитель ОП

Алиев Т. И. _____

(Фамилия, И.О.) (подпись)

«____» «_____» 20____ г.

З А Д А Н И Е

НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ

Студенту *Ницер Ксении Александровне* **Группа** *P3401* **Факультет** *ПИиКТ* **Руководитель ВКР** *Пенской Александр Владимирович, к.т.н., Университет ИТМО, факультет ПИиКТ, доцент*

(ФИО, ученое звание, степень, место работы, должность)

1 Наименование темы: *Автоматизация построения моделей вычислительных блоков для системы проектирования специализированных вычислителей*

Направление подготовки (специальность) *09.03.01 - «Информатика и* вычислительная техника*»*

Направленность (профиль) *«Вычислительные машины, комплексы, системы и сети»*

Квалификация *Бакалавр*

2 Срок сдачи студентом законченной работы « 25 » « мая » 2020 г.

3 Техническое задание и исходные данные к работе *В рамках выпускной работы необходимо решить следующие задачи:*

- модернизация и автоматизация маршрута проектирования вычислительных блоков.

- разработка формата спецификации последовательных вычислительных блоков для семейства специализированных процессоров, достаточного для планирования вычислительного процесса в рамках САПР NITTA;

- разработка модуля планирования вычислительного процесса для САПР на основе формата

спецификации последовательных вычислительных блоков;

- разработка алгоритма генерации спецификации последовательного вычислительного блока на основании списка реализуемых им функций и его аппаратной реализации на языке Verilog;

4 Содержание выпускной квалификационной работы (перечень подлежащих разработке вопросов)

1. Введение

2. Системы автоматизированного проектирования специализированных вычислителей

2.1 Обзор платформы NITTA, ее процесса синтеза и маршрута проектирования

2.2 Проблемы стандартного маршрута проектирования вычислительных блоков

3. Автоматизированная система проектирования моделей вычислительного блока

3.1 Автоматизированный маршрут проектирования вычислительных блоков

3.2 Формат спецификации вычислительного блока

3.3 Модуль планирования вычислительного процесса

3.4 Алгоритм генерации спецификации вычислительного блока

4. Интеграция автоматизированного маршрута проектирования вычислительных боков

5. Заключение

5 Перечень графического материала (с указанием обязательного материала)

1. Презентация в формате pdf, содержащая следующие слайды:

- Введение

- Постановка задач

- Старый маршрут проектирования вычислительного блока

- Новый маршрут проектирования вычислительного блока

6 Исходные материалы и пособия

1. Пенской А. Разработка и исследование архитектурных стилей проектирования уровней организации встроенных систем: … канд. техн. наук : 05.13.12 / Пенской А.В. – Санкт-Петербург, 2016. – 169 с.

2. Липовача, М. Изучай Haskell во имя добра! / М. Липовача; пер. с анг. Д. Леушин, А. Синицын, Я. Арсанукаев, ред. В.Н. Брагилевский, Р.В. Душкин. – Москва: ДМК-ПРЕСС, 2017. – 490 с.

7 Дата выдачи задания «25 » «октября » 2019 г.

Руководитель ВКР_____
(подпись)

Задание принял к исполнению_____ « 25 » «октября » 2019 г.
(подпись)

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

АННОТАЦИЯ

ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

Обучающийся Ницер Ксения Александровна
(ФИО)

Наименование темы ВКР: Автоматизация построения моделей вычислительных блоков для системы проектирования специализированных вычислителей

Наименование организации, где выполнена ВКР Университет ИТМО

ХАРАКТЕРИСТИКА ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЫ

1 Цель исследования автоматизация построения моделей вычислительных блоков для системы проектирования специализированных вычислителей с целью сокращения трудозатрат на разработку вычислительных блоков и требований к квалификации разработчика моделей

2 Задачи, решаемые в ВКР модернизация и автоматизация маршрута проектирования вычислительных блоков; разработка формата спецификации последовательных вычислительных блоков для семейства специализированных процессоров; разработка конфигурируемого модуля планирования вычислительного последовательного процесса вычислительного блока на основе формата спецификации; разработка программного модуля, позволяющего сгенерировать спецификацию последовательного вычислительного блока на основе списка реализуемых им функций и его аппаратной реализации.

3 Число источников, использованных при составлении обзора 8

4 Полное число источников, использованных в работе 10

5 В том числе источников по годам

Отечественных			**Иностранных**		
Последние 5 лет	От 5 до 10 лет	Более 10 лет	Последние 5 лет	От 5 до 10 лет	Более 10 лет
4	1	0	0	2	3

6 Использование информационных ресурсов Internet Да, 2
(Да, нет, число ссылок в списке литературы)

7 Использование современных пакетов компьютерных программ и технологий (Указать, какие именно, и в каком разделе работы)

Пакеты компьютерных программ и технологий	**Раздел работы**
Visual Studio Code	2,3
Stack	1,2,3
Glasgow Haskell Compiler	1,2,3
Git	3

8 Краткая характеристика полученных результатов

1. Предложен новый маршрут проектирования вычислительных блоков
2. Разработан формат спецификации последовательных вычислительных блоков
3. Разработан алгоритм генерации спецификации последовательного вычислительного блока
4. Разработан конфигурируемый модуль планирования вычислительного последовательного процесса вычислительных блоков

9 Полученные гранты, при выполнении работы Нет
(Название гранта)

10 Наличие публикаций и выступлений на конференциях по теме выпускной работы Нет
(Да, нет)

а) 1 _____
(Библиографическое описание публикаций)
2_____
3_____

б) 1_____
(Библиографическое описание выступлений на конференциях)
2_____
3_____

Обучающийся Ницер Ксения Александровна _____
(ФИО) (подпись)

Руководитель ВКР Пенской Александр Владимирович _____
(ФИО) (подпись)

“_____”_____20___г.

ОГЛАВЛЕНИЕ

АББРИВИАТУРЫ

HS – Hardware Systems

HDL – Hardware Description Language

NISC – No Instruction Set Computing

TTA – Transport Triggered Architecture

ВБ – Вычислительный блок

ВПл – Вычислительная платформа

ВсС – Встроенные системы

КФС – Киберфизические системы

САПР – Система автоматизированного проектирования

ФБ – функциональный блок

ВВЕДЕНИЕ

В последнее время наблюдается тенденция внедрения компьютеров во все аспекты жизни человека. При этом требования к устройствам со временем растут, они должны реализовывать все больше возможностей, что делает их создание все сложнее. Это привело к созданию концепции киберфизических систем (КФС), которая обозначает интеграцию вычислений с физическими процессами. В КФС встроенные компьютеры и сети контролируют физические процессы, которые влияют на вычисления и наоборот [1].

В КФС время, необходимое для выполнения задачи, имеет решающее значение для правильного функционирования системы. Поэтому сегодня в области разработки встроенных и киберфизических систем актуально применение специализированных вычислителей, так как они позволят значительно повысить эффективность процесса моделирования с точки зрения временных и энергетических затрат, а также реализовать его в реальном времени [2] Разработкой специализированных вычислителей занимается вычислительная платформа (ВПл) NITTA.

ВПл NITTA находится на этапе разработки, поиска более простых решений некоторых задач для более понятного использования этого продукта программистами и прикладными специалистами. ВПл NITTA представляет собой систему высокоуровнего синтеза на основе гибридной микроархитектуры NISC/TTA, с помощью которой вычислительный процесс строится на основе транзакций между вычислительными блоками [3]. Вычислительный блок (ВБ) содержит всю обработку и хранение данных, взаимодействие с периферийным оборудованием и задачи организации вычислительного процесса. В данной работе рассматриваются модели ВБ, которые представляют совокупность численных параметров или характеристик исследуемой системы, которые могут пересчитываться с течением времени [2].

Создание нового ВБ привносит неудобства в разработку, так как требует самостоятельного описания аппаратной реализации и модели для САПР. Это требует много времени и делает взаимодействие пользователя с платформой более сложным.

Поэтому было решено автоматизировать создание моделей ВБ для избавления программистов и будущих пользователей от этих задач. Это позволит упростить интерфейс взаимодействия конечного продукта с пользователем.

Цель работы: автоматизация построения моделей вычислительных блоков для системы проектирования специализированных вычислителей с целью сокращения трудозатрат на разработку вычислительных блоков и требований к квалификации разработчика моделей.

Задачи:

1) Разработка формата спецификации последовательных вычислительных блоков для семейства специализированных процессоров, достаточного для планирования вычислительного процесса в рамках САПР NITTA.
2) Разработка конфигурируемого модуля планирования вычислительного последовательного процесса вычислительных блоков для САПР на основе формата спецификации последовательных вычислительных блоков.
3) Разработка алгоритма генерации спецификации последовательного вычислительного блока на основании списка реализуемых им функций и его аппаратной реализации на языке Verilog.
4) Модернизация и автоматизация маршрута проектирования вычислительных блоков.

ГЛАВА 1. СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ СПЕЦИАЛИЗИРОВАННЫХ ВЫЧИСЛИТЕЛЕЙ

1.1 ОБЗОР ПЛАТФОРМЫ NITTA

Разрабатываемая вычислительная платформа NITTA является специализированным инструментом, цель которой состоит в предложении нового решения по проектированию на ПЛИС. Она обладает такими особенностями, как простая модель вычислительного процесса, а именно управляемость процессом синтеза и его прозрачность на разных этапах. Это дает возможность не только автоматизировать процесс трансляции прикладного алгоритма на высокоуровневом языке, но и также иметь возможность принять участие в процессе синтеза пользователю.

Такие возможности предоставляет используемая система высокоуровневого синтеза на основе гибридной микроархитектуры NISC/TTA. В данном случае NISC позволяет генерировать эффективный высокопараллельный программный код для вычислителя, сочетающего в себе разнотипные ВБ. TTA архитектура позволяет обеспечить высокий уровень параллелизма аппаратной составляющей, определить пути для масштабирования вычислителя и сформировать простую и наглядную модель описания вычислительного процесса и аппаратной составляющей [3].

Данная платформа направленна на автоматизацию разработки реконфигурируемых высокопроизводительных вычислителей реального времени, а также синтез целевой вычислительной системы, циклически исполняющей прикладной алгоритм с высоким уровнем параллелизма в режиме жесткого реального времени. Может применяться для:

- Разработки киберфизических систем, использующих адаптивные робастные алгоритмы управления, элементы искусственного интеллекта.
- Программно-аппаратного тестирования и быстрого прототипирования (HIL и PIL).

– Разработки программируемых ускорителей и сопроцессоров.
– Разработки проблемно-ориентированных спецпроцессоров (ASIC и IP-cores).

В основе вычислительного процесса данной системы лежит взаимодействие между вычислительными блоками без использования системы команд, минуя промежуточный регистры. В этих блоках сосредоточены обработка и хранение данных, а программируются они с помощью аппаратной реализации на языке Verilog и модели для ПО на языке Haskell. За счет принципа совместного проектирования, заключающегося в параллельной разработке программного и аппаратного обеспечения, получается добиться высокого уровня интеграции и, как следствие, высокой эффективности.

Для синтеза целевого вычислителя необходимо корректное и достаточное представление доступных ВБ в рамках САПР, что осуществляется при помощи соответствующих моделей. Именно модель ВБ задает список поддерживаемых функций, организует вычислительный процесс и генерирует управляющее программное обеспечение. При этом реализация ВБ выполненная на HDL должна быть согласована с моделью в составе САПР [4]. Моделирование является итеративным процессом, где на каждом шаге рассчитываются новые значения для модели [2].

1.2 ПРОЦЕСС СИНТЕЗА

Процесс синтеза рассматривается как последовательное принятие решений, где для каждого решения можно посмотреть его причины, отменить или изменить его, ознакомиться с альтернативными вариантами. Синтез может быть представлен в виде графа (дерева), где каждый «узел» описывает целевую систему и каждое «ребро» решений. Тогда процесс синтеза можно представить как процесс нахождение лучшего листа дерева, а

лучший метод синтеза - метод, который напрямую проходит по дереву к лучшему листу без неправильных шагов.

В процессе синтеза решаются следующие задачи:

- Оптимизация прикладного алгоритма с учетом конфигурации процессора и уровня загрузки отдельных ресурсов.
- Формирование конфигурации процессора (коммуникация и управляющая инфраструктура, состав и схема подключения ВБ).
- Планирование потока управления и потока данных.
- Планирование вычислительного процесса с точки зрения передачи данных.
- Управление внутренними ресурсами ВБ.

Процесс синтеза при очередном принятии решения получает список всех доступных для планировщика ВБ на текущем такте. У каждого доступного ВБ опрашиваются опции, которые он может реализовать на данном этапе. После этого, в зависимости от типа синтеза, выбирается одна из опций одной модели. Процесс синтеза уведомляет выбранную модель о принятом решении, и та, в свою очередь, планирует выбранный шаг. После этого все действия повторяются для нового состояния, пока синтез не дойдет до успешного или неуспешного завершения.

Процесс планирования работы отдельного ВБ происходит с помощью трех операций, выполняемых в цикле: `tryBind`, `endpointOptions` и `endpointDecision`.

Операция `tryBind` осуществляет привязку функций к вычислительным блокам. Она позволяет проверить, может ли функция вычисляться этой моделью, а если может – сохранить ее в модель для дальнейшей работы. Отказ от привязки может быть связан либо с тем типом функций, который не поддерживается, либо с пустыми внутренними ресурсами ВБ.

Операция `endpointOptions` предлагает планировщику все возможные варианты действия с данным ВБ на определенном шаге планирования. Она возвращает массив доступных переменных для выгрузки или загрузки и их интервалов, в течение которых может реализовываться взаимодействие с данной переменной. Опции указывают не на конкретный момент для работы, а на доступный интервал, который описывает, в какое время можно выполнять загрузку и выгрузку, и сколько времени процесс может продолжаться. В стандартной реализации возвращается интервал от доступного такта до бесконечности.

Операция `endpointDecision` осуществляет принятое решение о переходе вычислительного процесса в новое состояние с помощью полученной команды, после чего состояние модели ВБ обновляется.

Результатом планирования является описание одного вычислительного цикла, который впоследствии можно преобразовать в микрокод.

Поток данных определяет вычисления для одного вычислительного цикла. Но обычно требуется передавать данные между циклами и контроллер должен повторять алгоритм бесконечное количество раз. Для этого и предназначена функция `Loop` (рис. 1). В первом цикле функция `Loop` создает начальное значение, после этого в каждом цикле `Loop` выдает переменное значение из предыдущего цикла и потребляет новое значение в конце цикла.

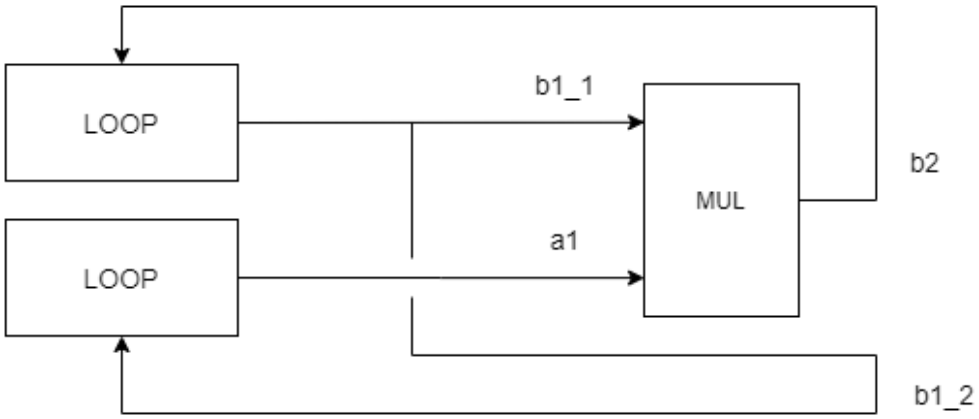


Рисунок 1 - График потока данных

Рассмотрим пример процесса синтеза двойного цикла умножения с использованием метода принятия решения, который выбирает первый доступный узел, удовлетворяющий следующим условиям:

- Не будет возможности получить deadlock
- Количество привязанных альтернатив равно единице

Первым делом происходит привязка значения к ячейке памяти №1 (рис. 2а). После этого планировщик обнаруживает, что ячейка памяти №1 имеет два цикла. CAD не может синтезировать целевую систему с циклом, так как такой алгоритм является слишком сложным. Для решения этой проблемы используется специальный модуль, определяющий тип настроек для исправления ситуации. В данном случае происходит разбиение циклов первой ячейки (рис. 2б). Тоже самое планировщик проделывает с ячейкой памяти №0, у которой имеется один цикл (рис. 2в, 2г).

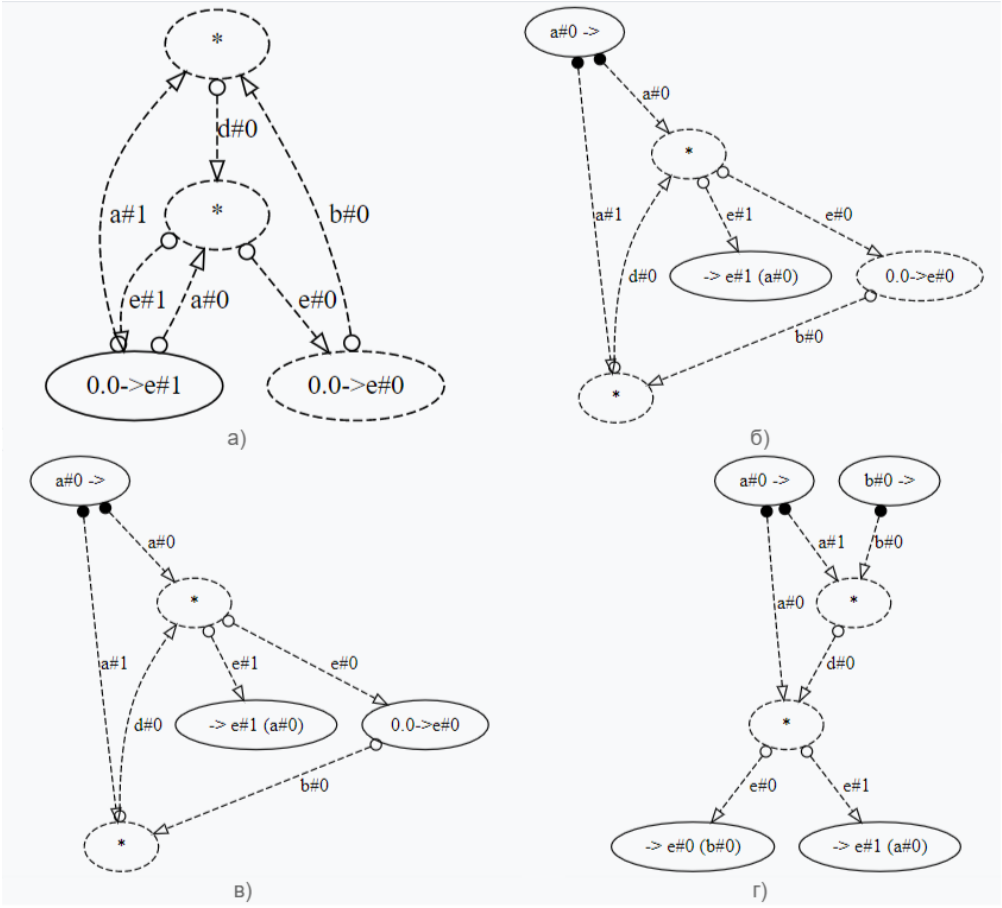


Рисунок 2 – а) привязка значения к ячейке №1; б) разбиение цикла ячейки №1; в) привязка значения к ячейке №0; г) разбиение цикла ячейки № 0

Дальше планировщик привязывает функцию к умножителю (рис. 3а). Привязка проходит успешно, поэтому следующим шагом планировщик загружает сначала значение `a#1` (рис. 3б), а затем `b#0` в умножитель (рис. 3в). После этого происходит первая операция умножения переданных значений, после которой результат готов к выгрузке.

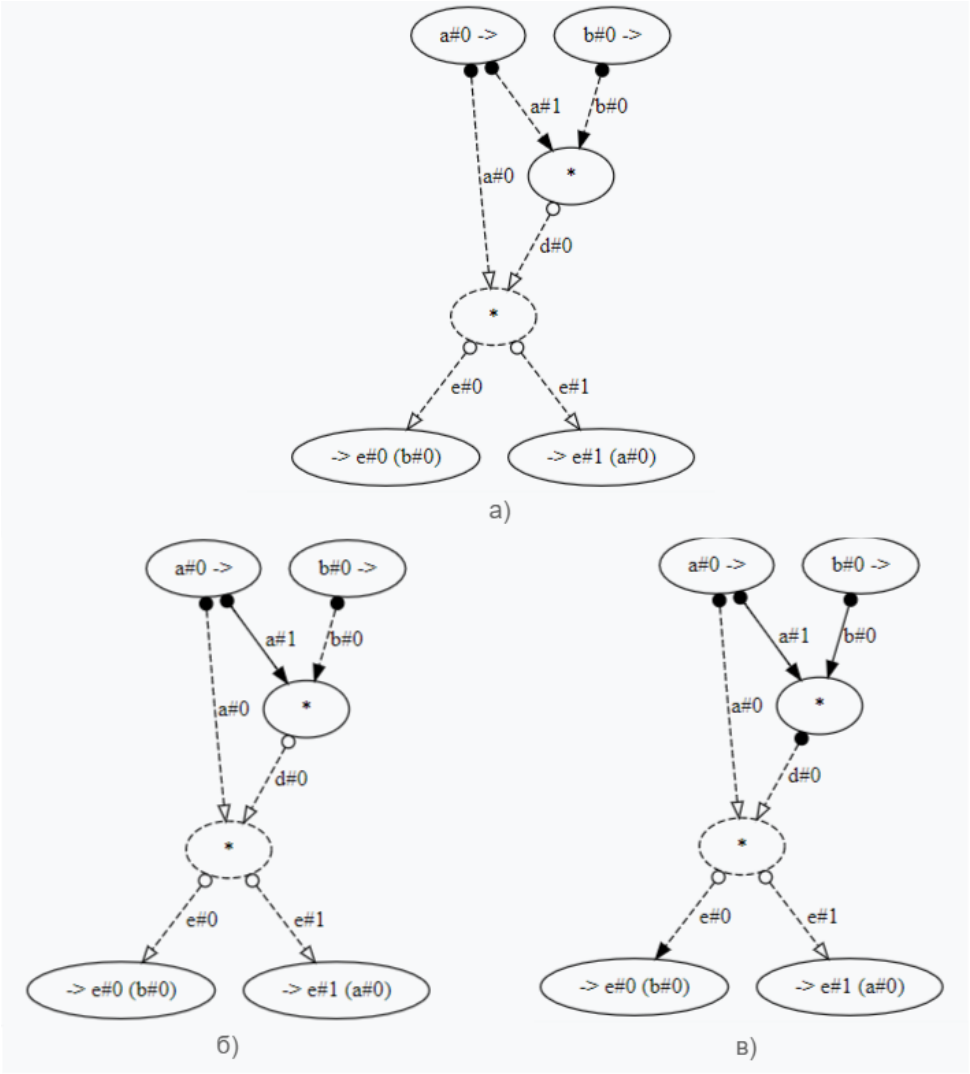


Рисунок 3 – а) привязка функции умножения к модели;
б) загрузка значения `a#1`; в) загрузка значения `b#0`

Для следующей операции умножения в качестве первой переменной используется то же значение `a#0`, а в качестве второй переменной результат первой операции умножения. Для начала происходит привязка новой функции умножения к умножителю (рис. 4а). Попытка выгрузки результата первого умножения приводит к тому, что появляется deadlock, с которым CAD не может синтезировать целевую систему. Дело в том, что первую и вторую функции умножения обрабатывают на одном и том же блоке процесса. В этом случае происходит тупик, который можно исправить, вставив буферный регистр между функциями (рис. 4б). Прошлый результат

умножения сохраняется в созданный буфер (рис. 4в), после чего оба значения готовы к выгрузке (рис 4г).

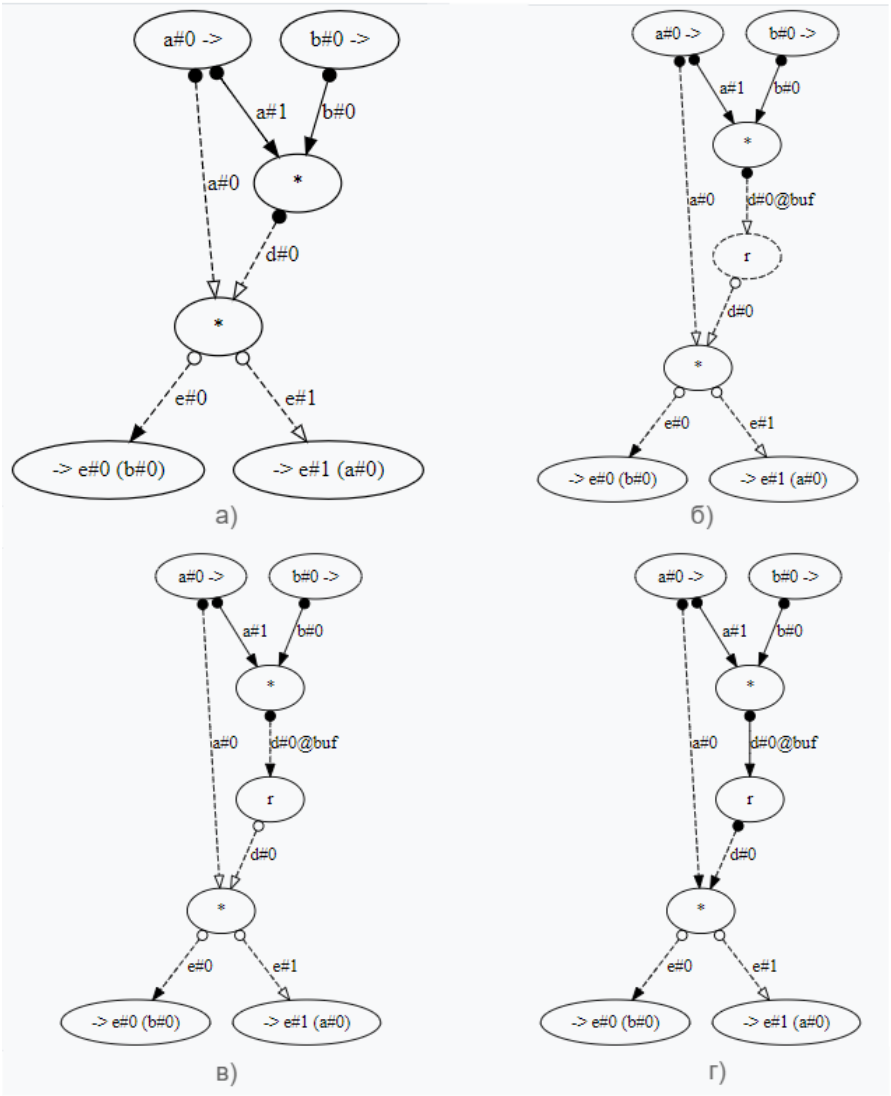


Рисунок 4 – а) привязка функции умножения к модели; б) вставка буферного регистра; в) сохранение результата умножения в буферный регистр; г) оба значение готовы к выгрузке

Планировщик загружает значение `a#0` (рис. 5а) и `d#0`, взятого из буферного регистра (рис. 5б), в умножитель, где происходит последняя операция умножения. После этого можно выгружать конечный результат, а синтез считается успешно завершенным (рис. 5в).

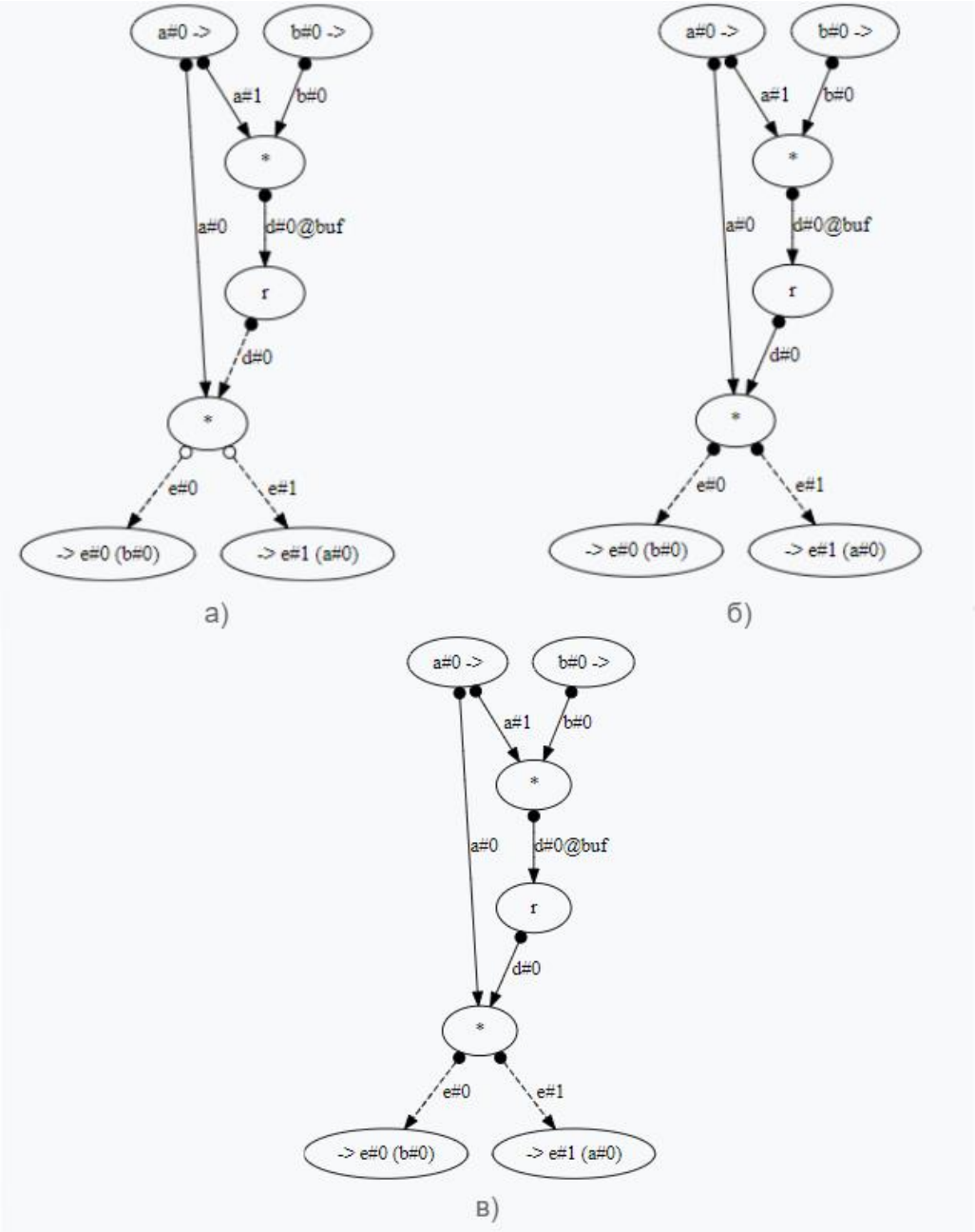


Рисунок 5 – а) загрузка значения `a#0`;

б) загрузка значения `d#1`; в) выгрузка конечного результата

1.3 ПРИМЕР РАБОТЫ САПР С МОДЕЛЬЮ ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Разработка ВБ, исполняющего роль умножителя, начинается с разработки аппаратной реализации и модели в САПР для синтеза (рис. 6). При разработке модели указываются структура умножителя, содержащая всю необходимую информацию, управляющие сигналы, операции планирования

и многие другие вещи, необходимые для работы модели. Оба блока реализуются вручную и параллельно. Затем они направляются на прохождение верификации, и при успешном результате блок считается завершенным и готовым к работе.

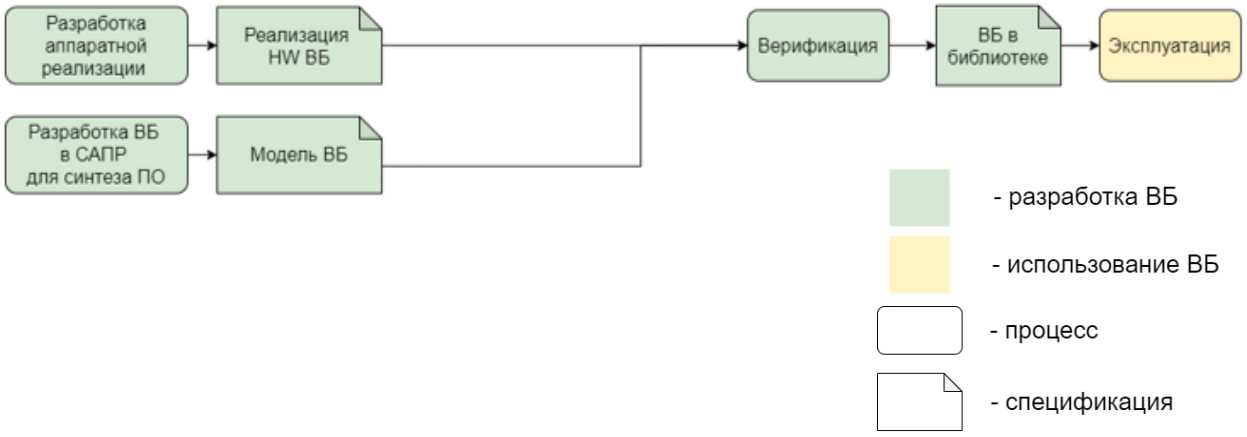


Рисунок 6 – Маршрут проектирования ВБ в стандартной версии

Рассмотрим пример планирования вычислительного процесса для ВБ, исполняющего роль умножителя. При запуске платформа инициализирует ВБ. Некоторым ВБ назначаются начальные параметры, другие же используют инициализацию по умолчанию. В данном случае будет использоваться ручная инициализация с параметром `mock`, описывающий, будет ли использоваться сгенерированный файл для аппаратного блока или ссылка на IP ядра. Используем сгенерированный файл, указывав для параметра значение `True`. Дальнейшие листинги будут приведены с использованием синтаксиса языка Haskell [5].

```
> let st0 = multiplier True
> st0
Multiplier {remain=[], targets=[], sources=[], doneAt=Nothing,
      currentWork=Nothing, currentWorkEndpoints=[],
      process_=Process{steps=[], relations=[], nextTick=0,
      nextUid=0}, tick=0, isMocked=True}
```

В процессе синтеза алгоритма для умножителя находится соответствующая функция, имеющая вид:

```
> f
c = d = a * b
```

Планировщик пытается привязать данную функцию к умножителю. Эта операция может быть выполнена в любое время работы с моделью, в том числе, когда процесс расчета полностью запланирован. Однако, если работа полностью спланирована, то ее необходимо выполнить, и любая ее часть не может быть потеряна внутри модели. Поэтому если функция соответствует данной модели, то она успешно привязывается, сохраняясь в поле `remain`.

```
> let Right st1 = tryBind f st0
> st1
Multiplier {remain=[c=d=a*b], targets=[], sources=[],
doneAt=Nothing, currentWork=Nothing,
currentWorkEndpoints=[], process_=Process{steps=[],
relations=[], nextTick=0, nextUid=0}, tick=0,
isMocked=True}
```

При дальнейшем обращении планировщика к умножителю происходит опрос доступных опций в текущем состоянии. Это происходит через операцию `endpointOptions`**.** В данном случае у нас есть два варианта планирования вычислительного процесса, которые соответствуют разным последовательностям загрузки аргументов. Они похожи с точки зрения времени выполнения: загрузка может начаться с первого такта или после произвольной задержки. Для загрузки одного аргумента нужен только один такт, но он может продолжаться произвольное время.

```
> endpointOptions st1
[?Target "a"@(1..∞ /P 1..∞),?Target "b"@(1..∞ /P 1..∞)]
```

Получая такие данные, планировщик решает, какое действие в рамках выбранного метода синтеза является лучшим выбором. Допустим, он выбрал первый вариант и после уведомил об этом умножителя с помощью операции

endpointDecision. Умножитель, получив решение планировщика, осуществляет планирование вычислительного процесса.

```
> let st2 = endpointDecision st1 $ EndpointSt (Target "a")(0...2)
> st2
Multiplier {remain=[], targets=["b"], sources=["c","d"],
           doneAt=Nothing, currentWork=Just (1,c=d=a*b),
           currentWorkEndpoints=[0], process_=Process{
           steps=[Step{sKey=1, sTime=0...2, sDesc=Load A},
           Step{sKey=0, sTime=0...2,sDesc=Target "a"}],
           relations=[Vertical 0 1], nextTick=2,nextUid=2},
           tick=2, isMocked=True}
```

Дальше планировщик опять опрашивает умножителя на наличие доступных опций. Умножитель предлагает оставшийся загружаемый аргумент, после чего снова происходит отправка решения и осуществление вычислительного процесса. Опция выгрузки не может быть предложена до загрузки всех необходимых значений, так как такая аппаратная и алгоритмическая реализация невозможна.

```
> endpointOptions st2
?Target "b"@(3..∞ /P 1..∞)
> let st3 = endpointDecision st2 $ EndpointSt (Target "b")(3...3)
> st3
Multiplier {remain=[], targets=[], sources=["c","d"],
           doneAt=Just 6, currentWork=Just(1,c=d=a*b),
           currentWorkEndpoints=[2,0], process_=Process{
           steps=[Step {sKey=3, sTime=3...3, sDesc=Load B},
           Step {sKey=2, sTime=3...3, sDesc=Target "b"},
           Step {sKey=1, sTime=0...2, sDesc=Load A},
           Step {sKey=0, sTime=0...2, sDesc=Target "a"}],
           relations=[Vertical 2 3,Vertical 0 1], nextTick=3,
           nextUid=4}, tick=3, isMocked=True}
```

Следующей опцией после загрузки всех аргументов является выгрузка переменных. Эти переменные могут быть выгружены одновременно или последовательно. Рассмотрим вариант с последовательной выгрузкой.

```
> endpointOptions st3
?Source (fromList ["c", "d"])@(6..∞ /P 1..∞)
```

```
> let st4 = endpointDecision st3 $ EndpointSt (Source $ fromList
["c"])(6...6)
> st4
Multiplier {remain=[], targets=[], sources=["d"], doneAt=Just 6,
      currentWork=Just(1,c=d=a*b),currentWorkEndpoints=[4,2,0],
      process_=Process{steps=[Step{sKey=5,sTime=6...6, sDesc=Out},
      Step{sKey=4, sTime=6...6, sDesc=Source "c"},
      Step{sKey=3, sTime=3...3, sDesc=Load B},
      Step{sKey=2, sTime=3...3, sDesc=Target "b"},
      Step{sKey=1, sTime=0...2, sDesc=Load A},
      Step{sKey=0, sTime=0...2, sDesc=Target "a"}],
      relations=[Vertical 4 5,Vertical 2 3,Vertical 0 1],
      nextTick=6, nextUid=6}, tick=6, isMocked=True}
> endpointOptions st4
?Source (fromList ["d"])@(7..∞ /P 1..∞)
> let st5 = endpointDecision st4 $ EndpointSt(Source $ fromList
["d"])(7...7)
> st5
Multiplier{remain=[],targets=[], sources=[], doneAt=Nothing,
      currentWork=Nothing, currentWorkEndpoints=[],
      process_=Process{steps=[Step{sKey=8, sTime=1...7,
      sDesc="c"="d"="a"*"b"},Step{sKey=7, sTime=7...7,
      sDesc=Out},Step{sKey=6, sTime=7...7, sDesc=Source "d"},
      Step{sKey=5, sTime=6...6, sDesc=Out},
      Step{sKey=4, sTime=6...6, sDesc=Source "c"},
      Step{sKey=3, sTime=3...3, sDesc=Load B},
      Step{sKey=2, sTime=3...3, sDesc=Target "b"},
      Step{sKey=1, sTime=0...2, sDesc=Load A},
      Step{sKey=0, sTime=0...2, sDesc=Target "a"}],
      relations=[Vertical 8 6,Vertical 8 4,Vertical 8 2,
      Vertical 8 0,Vertical 6 7,Vertical 4 5,
      Vertical 2 3,Vertical 0 1], nextTick=7, nextUid=9},
      tick=7, isMocked=True}
> endpointOptions st5
[]
```

Когда все варианты планирования вычислительных процессов исчерпаны и все связанные функции запланированы генерируется микрокод, который организовывает описанный вычислительный процесс умножителя.

1.4 ПРОБЛЕМЫ МАРШРУТА ПРОЕКТИРОВАНИЯ ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Проблема такого маршрута проектирования (рис. 6) заключается в том, что при создании новой функциональности нужно создавать новый ВБ, для которого описываются новые файлы аппаратной реализации и реализации модели. Это требует написание большого количества кода. Также разработка усложняется тем, что модели для синтеза разрабатываются на языке Haskell, который считается сложным для изучения, а от пользователя требуется глубокое понимание устройства САПР. Это все замедляет процесс разработки, делая его долгим и сложным.

Также процесс планирования не гонится за поиском последовательности загрузки/выгрузки данных, которая бы обеспечила оптимальную работу вычислителя. Каждый выбор доступных опций модели ВБ на каждом шаге возвращает действие с интервалом от одного до бесконечности. Это объясняется тем, что с используемой архитектурой мы не можем быть уверены, сколько тактов займет то или иное действие, и поэтому вынуждены ставить неограниченные интервалы.

1.5 ВЫВОДЫ

В данной главе была рассмотрена платформа NITTA, ее стандартный маршрут проектирования ВБ и процесс синтеза. Был рассмотрен пример работы САПР с моделью ВБ и выделены проблемы стандартного маршрута проектирования ВБ.

Основываясь на результатах, было решено автоматизировать разработку ВБ с поиском оптимальных последовательностей опций для

процесса планирования. Так как автоматизация проектирования аппаратной части нереализуема в силу своей специфики, все силы были направлены на автоматизацию построения моделей вычислительных блоков. Для этого были определены следующие этапы:

1) Модернизация и автоматизация маршрута проектирования вычислительных блоков.
2) Разработка формата спецификации последовательных вычислительных блоков для семейства специализированных процессоров, достаточного для планирования вычислительного процесса в рамках САПР NITTA.
3) Разработка конфигурируемого модуля планирования вычислительного последовательного процесса вычислительных блоков для САПР на основе формата спецификации последовательных вычислительных блоков.
4) Разработка алгоритма генерации спецификации последовательного вычислительного блока на основании списка реализуемых им функций и его аппаратной реализации на языке Verilog.

ГЛАВА 2 АВТОМАТИЗИРОВАННАЯ СИСТЕМА ПРОЕКТИРОВАНИЯ МОДЕЛЕЙ ВЫЧИСЛИТЕЛЬНЫХ БЛОКОВ

2.1 АВТОМАТИЗИРОВАННЫЙ МАРШРУТ ПРОЕКТИРОВАНИЯ ПОЛЬЗОВАТЕЛЬСКОГО ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Для автоматизации маршрута проектирования последовательного ВБ было решено заменить этап разработки модели для ПО на программный модуль, который позволил бы с указанием определённых настроек сам построить модель ВБ с необходимой нам функциональностью (рис. 7). Стандартный вид модели ВБ был приведен к универсальному, куда поступают необходимые настройки, за счет которых строится конечная модель. Также в универсальную модель передаются функции, которые должен уметь исполнять конечный ВБ и которые используются для генерации спецификации модели. Спецификация описывает реализуемые функции конечного ВБ и используется для построения оптимального маршрута планирования каждой указанной функции данной модели.

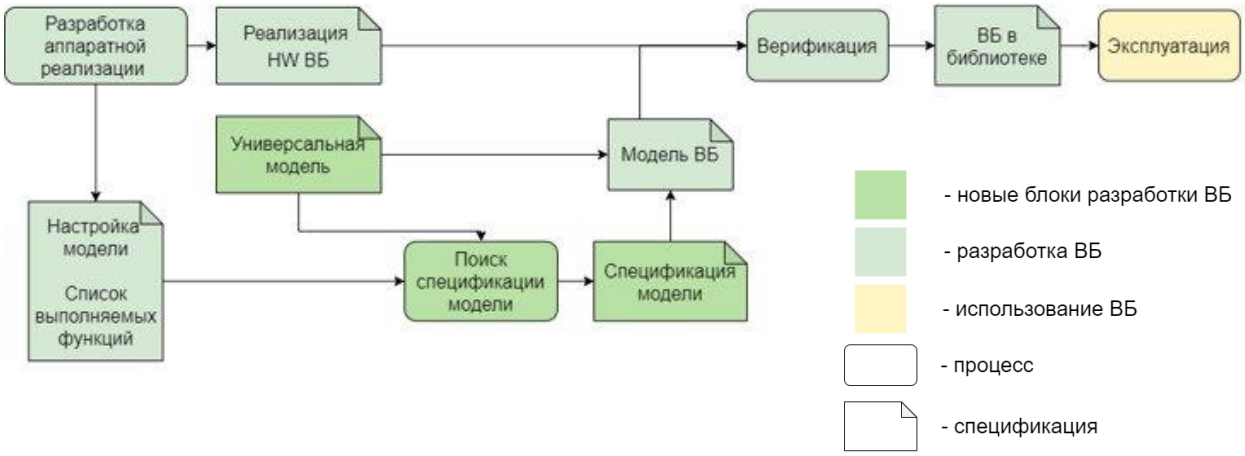


Рисунок 7 – Автоматизированный маршрут проектирования ВБ

Такой маршрут проектирования ВБ оставляет за пользователем разработку аппаратной реализации и требует указание настроек модели и списка реализуемых ВБ функций. Остальные процессы автоматизированы и

происходят без вмешательства пользователя. Это позволяет значительно сократить количество нового кода, что экономит много времени. Также от пользователя не требуется глубокого понимания устройства САПР, так как настройки модели реализуются в виде структуры, а создание новых функций не требует много знаний. Плюсом к этому становится получение последовательности оптимальных интервалов для загрузки и выгрузки, которая упрощает работу для планировщика и делает работу конечного вычислителя быстрее.

2.2 СПЕЦИФИКАЦИЯ ПОСЛЕДОВАТЕЛЬНЫХ ВЫЧИСЛИТЕЛЬНЫХ БЛОКОВ

Ключевым элементом автоматизированного маршрута проектирования ВБ является спецификация. Спецификация содержит набор реализуемых шаблонов функций и правила их обработки, включающие интервалы для каждого шаблона функций без привязки к конкретным значениям. Спецификация используется для определения возможности привязки функции к модели, а при помощи ее шаблонов происходит поиск оптимальной последовательности загрузки/выгрузки опций и их интервалов для процесса планирования.

При инициализации модели ВБ пользователь передает в конструктор функции, которые должен реализовывать ВБ. Универсальная модель в свою очередь сопоставляет переданные функции со списком доступных шаблонов (рис. 8). Для каждого найденного шаблона обработчик функции сохраняет соответствующие правила в спецификацию.

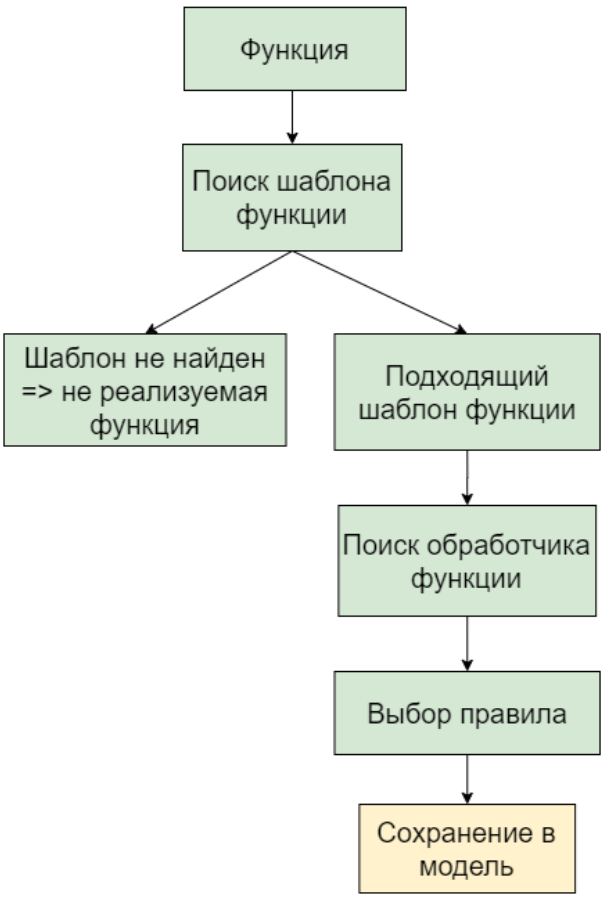
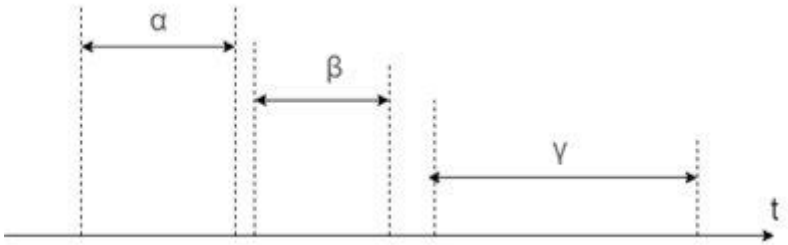


Рисунок 8 – Алгоритм работы конфигурируемой модели со спецификацией

Правила для функции представляют собой последовательность выгружаемы и загружаемых опций, не имеющих привязку к конкретному значению. Последовательность располагается алгебраически правильно и имеет для каждого значения интервал выполнения каждой опции и интервал времени, когда эта опция может быть реализована. Интервалы являются абстрактным и могут зависеть от последовательности загрузки/выгрузки опций. Так для функции умножения имеются две переменные загрузки и одна выгрузки (рис. 9). Переменные могут загружаться поочередно, что влечет за собой ограничения для второй переменной, которая может начать загружаться только после того, как загрузится первая и пройдет один такт. А выгружать результат можно только после того, как загрузят все переменные и пройдут три такта, в течение которых произойдет умножение переданных значений.



Время реализации опции

α: 0 ... ∞

β: $\alpha + 1$ ... ∞

γ: $\beta + 3$ ... ∞

Время выполнения опции

α: 1 ... 1

β: 1 ... 1

γ: 1 ... ∞

Рисунок 9 – Правила обработки функции умножения

Спецификация будет использоваться для:

– Проверки, что модель поддерживает функцию, которую пытается привязать планировщик во время процесса синтеза.

– Получение оптимальной последовательности опций и интервалов при помощи правил, используемой для планирования процесса синтеза.

Ниже приведен код со структурой данных спецификации и ее составляющей на языке Haskell:

```
data Specification v x t = Specification
     { rules :: [ Rule v x t ]
     }

data Rule v x t = Rule
     { funType  :: TypeRep
     , actions  :: [ Action v t ]
     } deriving ( Show )

data Action av t = Action
     { role                    :: EndpointRole av
     , begin                   :: Interval t
     , dur                     :: Interval t
     } deriving (Show)
```

2.3 ГЕНЕРАЦИЯ СПЕЦИФИКАЦИИ ПОСЛЕДОВАТЕЛЬНОГО ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Модуль генерации спецификации ответственен за поиск оптимальной последовательности опций и интервалов на основе абстрактного шаблона переданной функции. В конечном результате он возвращает оптимальную последовательность по времени выполнения, которая сохраняется в модель и будет использоваться при процессе планирования. Это дает такие преимущества, как:

- Автоматическое составление спецификации, согласование аппаратной реализации и модели.
- Использование оптимальной последовательности загрузки/выгрузки данных и интервалов по времени работы.
- Сокращение времени работы конечных вычислителей.

Для генерации спецификации необходимо передать в модуль привязываемую функцию и иметь доступ к правилам обработки этой функции и к аппаратной реализации (рис. 10). На основе переданных данных модуль генерирует возможную рабочую последовательность, которую затем отправляет в блок тестирования. Эти действия повторяются до тех пор, пока не будет найдена оптимальная последовательность по времени работы алгоритма, после чего она сохраняется в модель и используется при процессе планирования.

Блок генерации последовательности сопоставляет переданную функцию с шаблонами, реализуемыми данным ВБ. Если такая функция присутствует, то она сопоставляется с правилами обработки данной функции. Это позволяет определить правила составления последовательности опции, корректной для данной функции и ее аппаратной реализации, и их ограничения для выполнения и времени работы. Для разных функций последовательности опций могут отличаться в силу специфичности функций или микросхемы. Так, стоит учитывать, что при

многооперационной функции планировщик должен снять первый полученный результат до того, как загрузить следующую переменную в ВБ для повторного выполнения операции, иначе результат пропадет и работа алгоритма будет неверной.

Рисунок 10 – Алгоритм генерации спецификации ВБ

При поиске интервалов для каждой возможной последовательности опций накладываются ограничения по правилам спецификации, после чего идет их улучшение при помощи сокращения длительности работы. Этот шаг можно представить в виде алгоритма, состоящего из двух частей. В первой части (рис. 4) ищется средний интервал времени выполнения опции для всех элементов последовательности, достаточный для корректной работы системы. Вторая часть алгоритма (рис. 5) получает результат работы первой части и поочередно улучшает интервал для каждого элемента. На каждом

этапе поиска рабочей оптимальной последовательности происходит тестирование, которое говорит о том, рабочая ли данная последовательность.

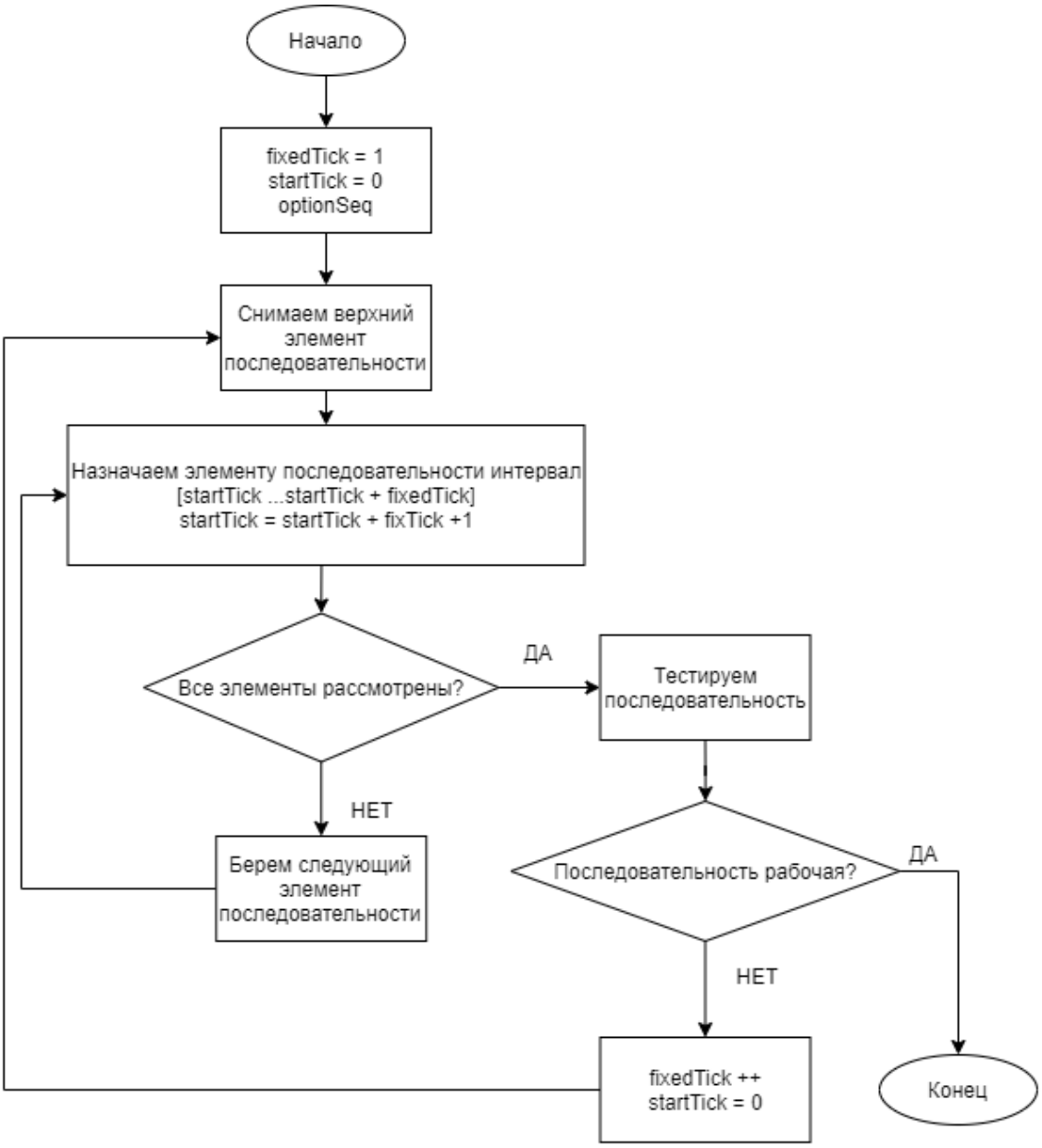


Рисунок 11 - Первая часть алгоритм поиска оптимальных интервалов

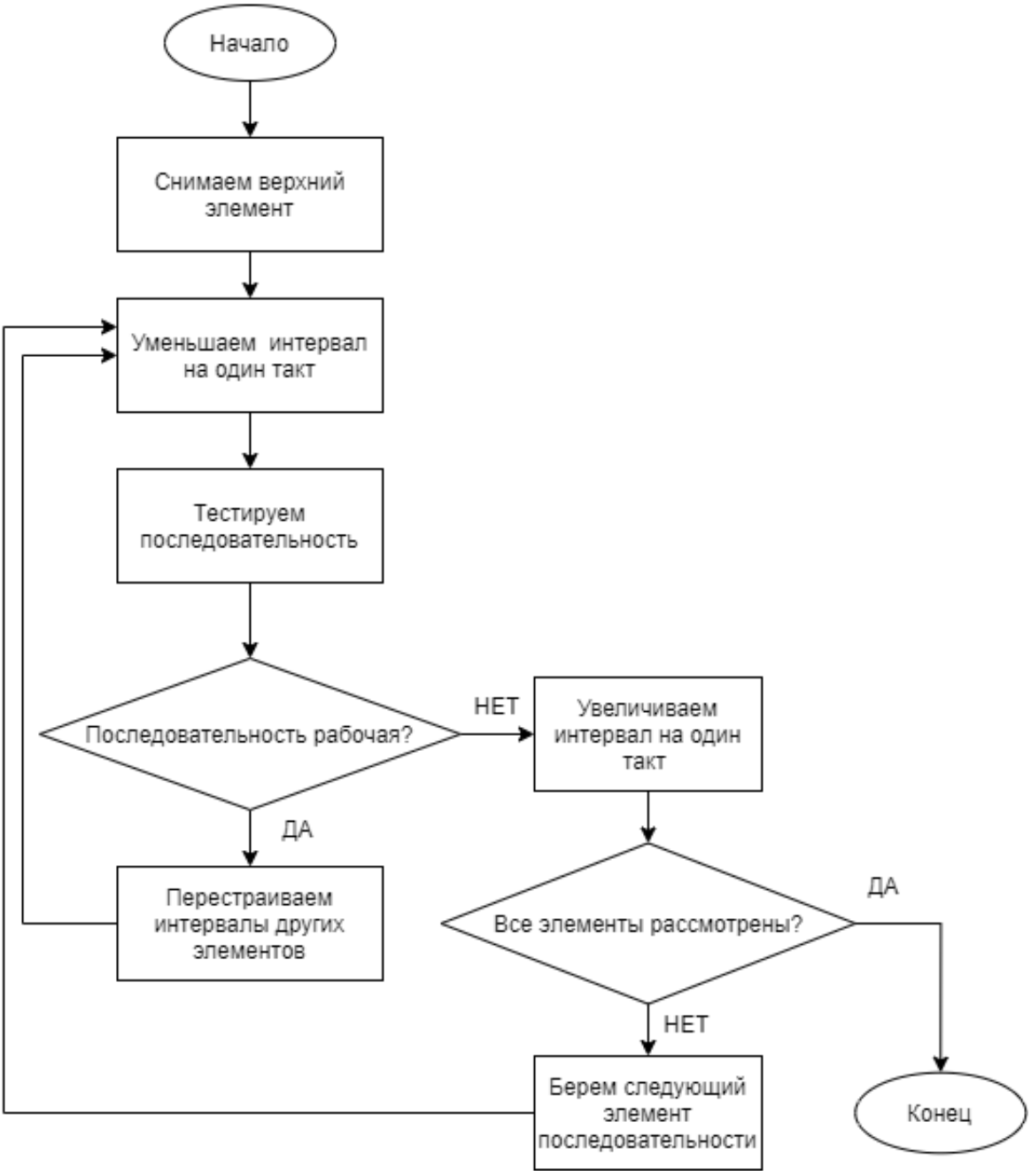


Рисунок 12 - Вторая часть алгоритм поиска оптимальных интервалов

Блок тестирования проводит верификацию сгенерированной модели и аппаратной реализации с помощью библиотеки QuickCheck, предоставляющей инструменты для работы со свойствами и генераторами случайных входных данных [6]. Благодаря ей можно описать генератор тестового набора для тестирования САПР, например: функциональных блоков, переменных и констант, решений, принимаемых системой синтеза. Процесс тестирования включает следующие шаги:

1) Выполняется генерация функциональных блоков (ФБ), определяющих алгоритм, в соответствии с возможностями ВБ.
2) В соответствии со сгенерированным алгоритмом происходит автоматический процесс синтеза, где решения принимаются случайным образом, включая: привязку ФБ к ВБ и планирование вычислительного процесса.
3) На основе полученного описания генерируется RTL реализация целевого аппаратного вычислителя и его машинный код.
4) Осуществляется проверка свойств: – проверка на соответствие работы ВБ и его аппаратной реализации средствами симуляции; – формальная проверка полноты выполнения работы ВБ.

Данная процедура повторяется много раз. Это позволяет экспериментально проверить корректность работы системы синтеза с заданной спецификацией на большом наборе тестовых [4].

На последнем шаге из всех полученных рабочих последовательностей опций и их интервалов выбирается та, что работает быстрее. Эти данные передаются в модель, которая их сохраняет и использует при дальнейшей работе.

В итоге мы можем получить рабочие последовательности опций с оптимальными интервалами для каждой функции. Эта информация сохраняется в спецификацию и будет использоваться во время процесса планирования. Генерация такой последовательности позволяет упростить работу планировщику, так как она позволяет сократить во время синтеза дерево возможностей (ребер) для планировщика, что позволит дереву не разрастаться, а решения принимать быстрее.

2.4 КОНФИГУРИРУЕМАЯ МОДЕЛЬ ПОСЛЕДОВАТЕЛЬНОГО ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Благодаря новому маршруту проектирования ВБ все необходимые модели являются автоматически сгенерированными. Это дает такие преимущества, как:

- Автоматически сгенерированная спецификация.
- Минимум кода при создании вычислительного блока.
- Экономия времени.
- Улучшение процесса планирования.
- Интеграция с существующими средствами верификации планирование.

Процесс планирования вычислительного процесса для САПР строится на сгенерированной модели, сохраненной в нее спецификации и ее зависимых частей. Спецификация генерирует оптимальную последовательность, которая преимущественно используется в операциях `endpointOptions` и `endpointDecision`.

Операция `endpointOptions`, отвечающая за помощь планировщику в принятии решения, будет поочередно предлагать опции из последовательности для рассчитываемой в данный момент функции. Теперь интервалы указывают на конкретное время работы, а количество предлагаемых опций равно единице.

Операция `endpointDecision` осуществляет принятое решение о переходе вычислительного процесса в новое состояние с помощью полученной команды. Для данной операции используются новые поля модели, `options` и `intervals`, которые содержат информацию об обрабатываемой функции. После перехода в новое состояние выполняемая на данном шаге опция и ее интервалы убираются из полей `options` и `intervals`, что говорит о выполнении данного действия.

Данные, переданные в настройках модели, завершают ее построение. Они определяют взаимодействие модели с аппаратной составляющей и ее особенности. В настройках модели передаются:

- Названия ВБ.
- Установка значений управляющих сигналов.
- Установка значений сигналов микрокода.
- Использование сгенерированного файла для аппаратного блока или ссылка на IP ядра.
- Ссылка на файл аппаратной реализации.

2.5 ИНТЕРФЕЙСЫ МОДЕЛИ ВЫЧИСЛИТЕЛЬНОГО БЛОКА

Целью модели ВБ является научить САПР с ней работать. Для САПР необходимо знать:

- Какие функции модель ВБ может реализовывать.
- Как управлять моделью ВБ для оценки конкретной функции.
- Как перевести инструкции в микрокод.
- Какие параметры процесса вычисления модели ВБ доступны (передача или извлечение переменной из модели ВБ)
- Планирование вычислительного процесса.

Эти задачи определяют интерфейсы ВБ и реализуемые ими классы.

Класс типа `ProcessorUnit` осуществляет привязку функций к вычислительным блокам. Он позволяет проверить, может ли функция вычисляться этой моделью, а если может - выполнить эту функцию. Содержит метод `tryBind`, осуществляющий привязку к модели ВБ, метод `process` для получения описания процесса вычисления и метод `setTime`, используемый для установки времени модели ВБ.

За процесс планирования отвечают методы `endpointOptions` и `endpointDecision` класса `EndpointProblem`. ВБ спрашивают об

опциях, которые он может реализовать через операцию `endpointOptions`, результатом которой является один из следующих списков:

- Список вариантов загрузки в модель ВБ, которые необходимы для работающей функции.
- Список вариантов выгрузки из модели ВБ.
- Список переменных загрузки в модель ВБ, загрузка любой из которых приведет к фактическому началу работы с данной моделью.

Планирование процесса или принятие решения о переходе ВБ в новое состояние осуществляется с помощью операции `endpointDecision`. Выделяются следующие состояния:

- Модель ожидает загрузку переменной.
- Модель ждет, что будут выгружаться переменные из нее.
- В данный момент не выполняется ни одна функция. Нужно выбрать функцию из списка назначенных функций, выполнить ее для работы и только затем принять решение и спланировать фрагмент вычислительного процесса.

Тип класса `Controllable` отвечает за планирование вычислительного процесса на аппаратном уровне. Определяет два уровня представления поведения: уровень инструкции и уровень микрокода. Инструкция по управлению ВБ описывает поведение модуля в каждом цикле модели ВБ. Так, например, множитель может загружать только аргументы и выгружать результат умножения. Если инструкция не определена для некоторых циклов - она должна интерпретироваться как `NOP`. Микрокод описывает управляющие сигналы на каждом цикле модели ВБ. Также класс содержит такие вспомогательные методы как `mapMicrocodeToPorts` – сопоставление микрокода с сигнальными портами устройства, `portsToSignals` – получение списка сигналов от `Ports` ВБ и `signalsToPorts` – получение порта из списка сигналов.

Состояние по умолчанию для микрокода определяется классом `Default` (соответствует неявной функции `NOP`). Это состояние означает, что модель ВБ находится в состоянии бездействия. Состояние по умолчанию используется для остановки, паузы или ожидания модели ВБ.

Интерфейс класса `UnambiguouslyDecode` используется для декодирования инструкции в микрокод, что требует их однозначного сопоставления независимо от состояния и настроек модели.

Класс `Simulatable` осуществляет генерацию стандартных значений, с которыми сравнивается фактический результат модели ВБ в симуляторе. Этот класс играет главную роль в тестировании.

Методы класса `TargetSystemComponent` используются для генерации процессоров и тестов, которые использует модель ВБ. Эти методы вызываются при создании проекта или при генерации тестов. Метод `moduleName` отвечает за именование модуля аппаратной реализации, метод `hardware` описывает генератор аппаратной реализации, `hardwareInstance` отвечает за генерацию фрагмента исходного кода для создания экземпляра модели ВБ в процессоре, а метод `software` является генератором программного обеспечения процессоров.

Класс `Testable` отвечает за создание автоматических тестов, изолированных для модели ВБ. Это позволяет создавать тесты для вычислительной единицы в соответствии с ее моделью и запланированным вычислительным процессом. На основе описанной модели ВБ генерируется входные сигналы и данные, а также проверяется последовательность выходных сигналов и данных. Выходные данные сравниваются с результатами функционального моделирования и, если они совпадают, тестирование считается успешным.

Также используются такие классы, как `IOTestBench` – получение ширины шины данных от уровня типа, `Connected` и `IOConnected` – описание сигналов процессора, `WithFunctions` – сервисный класс,

используемый для извлечения всех функций, связанных с моделью ВБ и `Locks` – отслеживающий внутренние зависимости данных, сгенерированных моделью ВБ.

2.6 ВЫВОДЫ

В данном разделе был предложен автоматизированный маршрут проектирования ВБ, где подробнее были рассмотрены формат спецификации ВБ, модуль генерации спецификации и конечная конфигурируемая модель ВБ.

Новый автоматизированный маршрут проектирования ВБ оставил за пользователем разработку аппаратной реализации и требовал указание настроек модели и списка реализуемых ВБ функций. Остальные процессы автоматизированы и происходят без вмешательства пользователя.

Спецификация определяет функциональность ВБ и содержит абстрактный шаблон последовательности без привязки к конкретным значениям, который используется в процессе планирования. Модуль генерации спецификации позволил автоматически составлять оптимальную последовательность опций и интервалов на основе абстрактного шаблона и переданной функции, а также согласовывать аппаратную реализацию и модель. В результате этого конечная модель является автоматически сгенерированной, а значит снимает с пользователя необходимость ее реализации, что экономится время разработки, а полученная последовательность оптимальных интервалов для загрузки и выгрузки упростила процесс планирования.

ГЛАВА 3 ИНТЕГРАЦИЯ АВТОМАТИЗИРОВАННОГО МАРШРУТА ПРОЕКТИРОВАНИЯ ВЫЧИСЛИТЕЛЬНЫХ БЛОКОВ

3.1 ПРОЕКТИРОВАНИЕ МОДЕЛИ УМНОЖИТЕЛЯ

Рассмотрим пример проектирования ВБ, исполняющего роль умножителя. Для начала обозначим настройки модели умножителя.

- Название: “pu_multiplier”
- Использование сгенерированного файла для аппаратного блока (значение True для соответствующего поля).
- Ссылка на файл аппаратной реализации: "multiplier/mult_mock.v".
- Установка значений управляющих сигналов: `wr=1` (для получения данные из bus); `wrSel=2` (для определения аргументов в bus); `oe=3` (для отправки результат в bus).
- Установка значений сигналов микрокода:

```
Microcode
{ wrSignal=False
, selSignal=False
, oeSignal=False
}
```

Следующим шагом указываются реализуемые нашим ВБ функции. В данном случае ВБ будет реализовывать только функцию умножения. Тип функции умножения передается в модель, где модель сопоставляет тип функции со списком шаблонов, находит шаблон умножения и сохраняет правила для его обработки в спецификацию. Для умножения правила обработки выглядят следующим образом:

```
Rule
{ funType=typeRep (Proxy :: Proxy (Mul v x t))
, actions=
    [ Action{role=Target α, begin=0...maxBound, dur=1...1}
    , Action{role=Target β, begin=1...maxBound, dur=1...1}
    , Action{role=Source γ, begin=2...maxBound, dur=1...maxBound}
    ]
```

```
}
```

Здесь значение поля `role` описывает тип опции, который может быть загрузкой (`Target`) или выгрузкой (`Source`). Поле `begin` описывает доступное время выполнения опции, а поле `dur` – время на его выполнение. Значение `maxBound` равняется максимальному значению типа, в данном случае бесконечности.

Полученная модель считается готовой для синтеза, поэтому она проходит верификацию с аппаратной реализацией и сохраняется в качестве нового компонента.

3.2 ГЕНЕРАЦИЯ СПЕЦИФИКАЦИИ МОДЕЛИ УМНОЖИТЕЛЯ

Генерация спецификации происходит после привязки новой функции к модели и начала ее реализации. В модуль генерации спецификации передается исполняемая функция, после чего модуль находит соответствующее правило для обработки этой функции. Модуль генерации понимает, что имеет дело с умножением и устанавливается ее вид:

```
Multiply (I a) (I b) (O c)
```

Используя правила обработки и установленный вид функции генерируются все возможные валидные последовательности загружаемых/выгружаемых данных. Шаблон функции умножения содержит `(I a)` и `(I b)` – первый и второй множитель, и `(O c)` – произведение, которое имеет два экземпляра "`c`" и "`d`". Правила математики позволяют сделать для умножителя следующие валидные перестановки:

1) `(I a) (I b) (O c) (O d)`
2) `(I a) (I b) (O d) (O c)`
3) `(I b) (I a) (O c) (O d)`
4) `(I b) (I a) (O d) (O c)`

Для каждой возможной последовательности модуль начинает просчитывать интервалы, опираясь на правила обработки функции:

```
α: begin=0...maxBound, dur=1...1
```

```
β: begin=1...maxBound, dur=1...1
γ: begin=2...maxBound, dur=1...maxBound
```

Полученная последовательность тестируется, после чего модуль решает, является ли последовательность рабочей и оптимальной или нужно их уменьшать или увеличивать. Из всех оптимальных последовательностей для всех перестановок выбирается та, что работает быстрее всех. Именно эта передается обратно в модель как самая оптимальная. Для примера она может выглядеть следующим образом:

```
bestOptions=[Target“a”, Target“b”, Source["c"], Source["d"]]
bestIntervals=[(1..1, 1..1),(2..2, 1..1),(6..6, 1..1)
,(7..7, 1..1)]
```

3.3 ПРИМЕР ПЛАНИРОВАНИЯ ВЫЧИСЛИТЕЛЬНОГО ПРОЦЕССА ДЛЯ КОНФИГУРИРУЕМОЙ МОДЕЛИ УМНОЖИТЕЛЯ

Рассмотрим пример планирования вычислительного процесса для конфигурируемого ВБ, исполняющего роль умножителя. После инициализации ВБ, начально состояние модели выглядит следующим образом:

```
> st0
SmartPU {remain=[], options=[], intervals=[], doneAt=Nothing,
     currentWork=Nothing, currentWorkEndpoints=[],
     process_=Process{steps=[], relations=[], nextTick=0,
     nextUid=0}, tick=0, isMocked=True, specification,
     dependentPartsModel}
```

В процессе синтеза алгоритма для умножителя находится соответствующая функция, имеющая вид:

```
> f
c = d = a * b
```

Данная функция реализуема нашим ВБ, поэтому она успешно привязывается к модели.

```
> let Right st1 = tryBind f st0
> st1
SmartPU {remain=[c=d=a*b], options=[], intervals=[],
```

```
    doneAt=Nothing, currentWork=Nothing,
    currentWorkEndpoints=[], process_=Process{steps=[],
    relations=[], nextTick=0, nextUid=0}, tick=0,
    isMocked=True, specification, dependentPartsModel}
```

После привязки функции планировщик начинает опрашивать умножитель о доступных опциях в текущем состоянии через операцию endpointOptions. Умножитель видит, что у него не просчитаны последовательности опций, а в поле `remain` имеется функция, поэтому он отправляет данную функцию в модуль генерации спецификации, получает оптимальную последовательность и сохраняет ее в поля `options` и `intervals`. После этого умножитель предлагает один вариант планирования вычислительного процесса, который соответствует лучшей последовательности загрузки аргументов.

```
> endpointOptions st1
[?Target "a"@(1..1 /P 1..1)]
> st1
SmartPU {remain=[], options=[Target“a”, Target“b”, Source["c"],
    Source ["d"]], intervals=[(1..1, 1..1), (3..3, 1..1),
    (6..6, 1..1), (7..7, 1..1)], doneAt=Nothing,
    currentWork=Just (1,c=d=a*b), currentWorkEndpoints=[],
    process_=Process{steps=[], relations=[], nextTick=0,
    nextUid=0}, tick=0, isMocked=True, specification,
    dependentPartsModel}
```

Если планировщик решает воспользоваться предложением умножителя, то он уведомляет умножитель об этом варианте через операцию `endpointDecision`. Умножитель, получив решение планировщика, осуществляет планирование вычислительного процесса.

```
> let st2 = endpointDecision st1 $ EndpointSt (Target "a")(1...1)
> st2
SmartPU{remain=[], options=[Target“b”, Source["c"], Source["d"]],
    intervals=[(3..3, 1..1), (6..6, 1..1), (7..7, 1..1)],
    doneAt=Nothing, currentWork=Just (1,c=d=a*b),
    currentWorkEndpoints=[0], process_=Process{steps=[Step{
    sKey=1, sTime=1...1, sDesc=Load A}, Step{sKey=0,
```

```
      sTime=1...1, sDesc=Target "a"}], relations=[Vertical 0 1],
      nextTick=1, nextUid=1}, tick=1, isMocked=True,
      specification, dependentPartsModel}
```

Планировщик снова опрашивает умножителя на наличие доступных опций. Умножитель предлагает следующий элемент последовательности, после чего происходит отправка решения и осуществление вычислительного процесса.

```
> endpointOptions st2
?Target "b"@(3..3 /P 1..1)
> let st3 = endpointDecision st2 $ EndpointSt (Target "b")(3...3)
> st3
SmartPU {remain=[], options=[ Source ["c"], Source ["d"]],
      intervals=[(6..6, 1..1), (7..7, 1..1)], doneAt=Just 6,
      currentWork=Just(1,c=d=a*b),currentWorkEndpoints=[2,0],
      process_=Process{steps=[Step {sKey=3, sTime=3...3,
      sDesc=Load B}, Step {sKey=2, sTime=3...3, sDesc=Target "b"},
      Step {sKey=1, sTime=1...1, sDesc=Load A},
      Step {sKey=0, sTime=1...1, sDesc=Target "a"}],
      relations=[Vertical 2 3, Vertical 0 1], nextTick=3,
      nextUid=4}, tick=3, isMocked=True, specification,
      dependentPartsModel}
```

Следующей опцией после загрузки всех аргументов является выгрузка переменных. Переменные могут быть выгружены одновременно или последовательно. Рассмотрим вариант с последовательной выгрузкой.

```
> endpointOptions st3
?Source (fromList ["c"])@(6..6 /P 1..1)
> let st4 = endpointDecision st3 $ EndpointSt (Source $ fromList
["c"])(6...6)
> st4
SmartPU {remain=[], options=[Source ["d"]], intervals=[(7..7,
      1..1)], doneAt=Just 6, currentWork=Just(1,c=d=a*b),
      currentWorkEndpoints=[4,2,0], process_=Process{
      steps=[Step{sKey=5,sTime=6...6, sDesc=Out}, Step{sKey=4,
      sTime=6...6, sDesc=Source "c"}, Step{sKey=3, sTime=3...3,
      sDesc=Load B}, Step{sKey=2, sTime=3...3, sDesc=Target "b"},
      Step{sKey=1, sTime=1...1, sDesc=Load A}, Step{sKey=0,
```

```
      sTime=1...1, sDesc=Target "a"}], relations=[Vertical 4 5,
      Vertical 2 3,Vertical 0 1], nextTick=6, nextUid=6}, tick=6,
      isMocked=True, specification, dependentPartsModel}
> endpointOptions st4
?Source (fromList ["d"])@(7..7 /P 1..1)
> let st5 = endpointDecision st4 $ EndpointSt (Source $ fromList
["d"])(7...7)
> st5
SmartPU {remain=[], options=[], intervals=[], doneAt=Nothing,
      currentWork=Nothing, currentWorkEndpoints=[],
      process_=Process{steps=[Step{sKey=8, sTime=1...7,
      sDesc="c"="d"="a"*"b"},Step{sKey=7, sTime=7...7,
      sDesc=Out},Step{sKey=6, sTime=7...7, sDesc=Source "d"},
      Step{sKey=5, sTime=6...6, sDesc=Out}, Step{sKey=4,
      sTime=6...6, sDesc=Source "c"}, Step{sKey=3, sTime=3...3,
      sDesc=Load B}, Step{sKey=2, sTime=3...3, sDesc=Target "b"},
      Step{sKey=1, sTime=1...1, sDesc=Load A}, Step{sKey=0,
      sTime=1...1, sDesc=Target "a"}], relations=[Vertical 8 6,
      Vertical 8 4,Vertical 8 2, Vertical 8 0,Vertical 6 7,
      Vertical 4 5, Vertical 2 3, Vertical 0 1], nextTick=7,
      nextUid=9}, tick=7, isMocked=True, specification,
      dependentPartsModel}
> endpointOptions st5
[]
```

Когда все варианты планирования вычислительных процессов исчерпаны и все связанные функции запланированы генерируется микрокод, который организовывает описанный вычислительный процесс на умножителе.

3.5 ВЫВОДЫ

В данной главе была спроектирована модель умножителя на основе автоматизированного маршрута проектирования, рассмотрен процесс генерации спецификации и проведен процесс планирования для спроектированной модели.

Использование абстрактного описания функции без оперирования конкретными значениями позволило задавать функциональность пользователем через ВБ до начала процесса синтеза.

Оптимальная последовательность опций и их интервалов генерировалась при первом опросе модели о доступных опциях после привязки функции. При генерации спецификации все последовательности прогонялись через блок тестирования, что уже позволило гарантировать работоспособность конфигурации модели. Лучшая последовательность сохранялась в модель и использовалась при планировании.

При синтезе системы использование сохраненной последовательности моделью обеспечило прямой путь планирования. Операция endpointOptions указывала конкретные рабочие интервалы времени реализации и времени работы, что упростило работу планировщика.

ЗАКЛЮЧЕНИЕ

В ходе данной работы был модернизирован и автоматизирован маршрут проектирования вычислительного блока. Это позволило освободить пользователя от создания моделей ВБ в САПР, оставив за ним разработку аппаратной реализации и указание настроек модели и списка реализуемых ВБ функций. Это позволило сделать создание новых компонентов проще и быстрее, а также снизить требования к уровню квалификации пользователей.

Был разработан формат спецификации последовательных вычислительных блоков для семейства специализированных процессоров NITTA, который стал одним из элементов нового маршрут проектирования. Спецификация определяет функциональность ВБ и содержит абстрактный шаблон последовательности, который не оперирует конкретными значениями и используется в процессе планирования. Это позволяет задавать функциональность пользователем через ВБ до начала процесса синтеза.

Также был разработан алгоритм генерации спецификации последовательного вычислительного блока на основании списка реализуемых им функций и его аппаратной составляющей. На основе этого алгоритма модуль генерации спецификации может автоматически составлять оптимальную последовательность опций и их интервалов на основе абстрактного шаблона и переданной функции, а также согласовывать аппаратную реализацию и модель. Это позволяет гарантировать работоспособность модели и делает работу конечного вычислителя эффективнее по времени.

Разработка конфигурируемого модуля планирования вычислительного последовательного процесса ВБ реализовала применение оптимальной последовательности во время синтеза, что упростило процесс планирования и визуализацию результата, так как дерево синтеза стало иметь меньше решений («ребер»), а значит и меньше возможных состояний целевой системы («узлов»).

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. E. Lee and S. A. Seshia, Introduction to Embedded Systems A Cyber-Physical Systems Approach, vol. 1. 2015. – 519 с.
2. Пенской А.В. Проектирование вычислительной платформы для моделирования динамических систем, 2018.
3. Пенской А.В., Платунов А.Е., Яналов Р.И., Система высокоуровневого синтеза на основе гибридной NISC/TTA микроархитектуры, 2018.
4. Анощенков Д.И. Верификация систем высокоуровнего синтеза аппаратных вычислителей, 2018.
5. L. Miran, Learn You a Haskell for Great Good! 2012. – 490 с.
6. What is QuickCheck [Электронный ресурс]. – Режим доступа: http://www.cse.chalmers.se/~rjmh/QuickCheck/manual_body.html /, своб.
7. Bollaert T. High-Level Synthesis: From Algorithm to Digital Circuit // High-Level Synthesis: From Algorithm to Digital Circuit / ed. Coussy P., Morawiec A. Springer, 2008. 29-52 p.
8. Corporaal, H. Computation in the Context of Transport Triggered Architectures / H. Corporaal, J. Janssen, M. Arnold. // Int. J. Parallel Program. 28. - 2000. - pp. 401- 427.
9. Haskell [Электронный ресурс]. – http://hackage.haskell.org/, своб.
10. Teich, J. Hardware/Software Codesign: The Past, the Present, and Predicting the Future / J. Teich. - Proc. IEEE - 2012. - vol. 100. - pp. 1411-1430
11. Быковский С.В., Горбачев Я.Г., Ключев А.О., Пенской А.В., Платунов А.Е. Сопряженное проектирование встраиваемых систем (Hardware/Software Co-Design). Часть 1: Учебное пособие - Санкт-Петербург: Университет ИТМО, 2016. - 108 с. 3.
12. Платунов А.Е. Теоретические и методологические основы высокоуровневого проектирования встраиваемых вычислительных систем : Автореф. дис. ... д-ра техн. наук : 05.13.12 / Платунов А.Е. - Санкт-Петербург, 2010. – 477 с.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД МОДУЛЯ ГЕНЕРАЦИИ СПЕЦИФИКАЦИИ

```
module NITTA.Model.ProcessorUnits.ModelSpecification
    ( Specification(..)
    , VarTmp(..)
    , specificationMaker
    , specificationTraining
    ) where

import           Data.List                          ( (\\) )
import           Data.Set                           (elems, fromLi
st, size)
import           Data.Typeable
import qualified NITTA.Intermediate.Functions     as F
import           NITTA.Intermediate.Types
import           NITTA.Model.Problems.Endpoint
import           Numeric.Interval                   (sup, (...))
import           NITTA.Test.SmartUnit               (smartTest)

---------------------- Specification ----------------------------

data Specification v x t = Specification
    { rules :: [ Rule v x t ]
    , dependencyVar            :: [ (EndpointRole v, EndpointRol
e v) ]
    } deriving ( Show )

data Rule v x t = Rule
    { funType  :: TypeRep
    , actions  :: [ Action v t ]
    } deriving ( Show )

-- av -- abstract variable
data Action av t = VarTmp
    { role                     :: EndpointRole av
    , begin                    :: Interval t
    , dur                      :: Interval t
    } deriving (Show)

mulSpec = Specification
[ Rule
 { funType=typeRep (Proxy :: Proxy (Mul v x t))
  , actions=
   [ Action{role=Source α, begin=0...maxBound, dur=1...1}
   , Action{role=Source β, begin=1...maxBound, dur=1...1}
   , Action{role=Source ɣ, begin=2...maxBound, dur=1...maxBound}
   ]
  }
```

```
]


checkSpec F{fun} = splitTyConApp $ typeOf fun

specificationMaker f@F{fun}
    | ((tyConName funType) == (tyConName tmpMulType)) = specific
ation (size $ inputs fun) (size $ outputs fun)
    | otherwise = Left $ "Error. No such function template"
        where
            funType = fst $ checkSpec f
            tmpMulType = fst $ checkSpec $ (F.multiply "x" "x" [
"x", "x"] :: F String Int)
            specification targetVarCount sourceVarCount = do
                let targetVars = variableMaker targetVarCount []
 True
                let sourceVars = variableMaker sourceVarCount []
 False
                Right Specification{
                      tmpFunction = funType
                    , tmpVar = targetVars ++ sourceVars
                    , dependencyVar = []
                }

variableMaker 0 arr _ = arr
variableMaker count arr isTarget = do
    let newVar = if isTarget then targetVarTmp else sourceVarTmp
    variableMaker  (count - 1) (arr ++ [newVar]) isTarget

targetVarTmp = VarTmp{option = Target Proxy, shift = 1, endIntre
val = maxBound }
sourceVarTmp = VarTmp{option = Source $ fromList [Proxy], shift
= 3, endIntreval = maxBound}

-------------------Specification generation--------------------
-- for get all working variables with every varTemp + choose
-- best variant
specificationTraining pu function specification
    | varTemplateList <- getEpRole function
    , result <- map (autoRun pu) varTemplateList
    , (bestoptions, bestintervals) <-
 foldl (\tmp@(_, ints) new@(_, nints) -
> if (null ints || (sup $ last ints) > (sup $ last nints)) then
new else tmp ) ([], []) result
     = (bestoptions, bestintervals)

autoRun pu varTemp = countBase pu 2 varTemp
    let vergeInterval = 2
    countBase pu const varTemp
    let countRes = count pu varTemp const
```

```
    case (countRes) of
        (Right result) -> return result
        (Left  result) -> return []

countBase pu vergeInterval varTmp = do
    let intTmp = map (\_ -> 1 ... vergeInterval) varTmp
    let tmpFunInfo = FunInfo function varTmp intTmp
    let tmpSpecification = Specification [tmpFunInfo]
    let pu' = pu{ modelSpecification=tmpSpecification }
    case (smartTest (pu' :: AutoMultiplier String Int Int)) of
         ( _ ) -> (varTmp, intTmp)
         ( _ ) -> countBase pu function const+1 varTmp

getEpRole f
        | Just (F.Division (I a) (I b) (O c) (O d)) <-
 castF f = map ([Target a, Target b] ++) $ makeEpRoleSeq [Source
 c, Source d]
        | Just (F.Multiply (I a) (I b) (O c))       <-
 castF f = concatMap (\x -
> map ( ++ x ) $ makeEpRoleSeq [Target a, Target b]) $ makeEpRol
eSeq [Source c]
        | otherwise = error "getEpRole error"

makeEpRoleSeq [Source a]
    | size a > 1 = makeEpRoleSeq $ map (\x -
> Source $ fromList [x] ) (elems a)
makeEpRoleSeq [a] = [[a]]
makeEpRoleSeq epRoleSeq = concatMap (\x -
> map ( x : ) (makeEpRoleSeq $ epRoleSeq \\ [x])) epRoleSeq
```

ПРИЛОЖЕНИЕ Б. ИСХОДНЫЙ КОД КОНФИГУРИРУЕМОЙ МОДЕЛИ

```
module NITTA.Model.ProcessorUnits.SmartModel
    ( smartModel
    , SmartModel
    , Ports(..), IOPorts(..)
    , testBenchImplementation
    ) where

import           Control.Monad         (when)
import           Data.Bits             (finiteBitSize)
import           Data.Default
import           Data.List             (find, partition, (\\))
import           Data.Set              (elems, fromList, member)
import qualified NITTA.Intermediate.Functions     as F
import           NITTA.Intermediate.Types
import           NITTA.Model.Problems.Endpoint
import           NITTA.Model.Problems.Refactor
import           NITTA.Model.ProcessorUnits.Time
import           NITTA.Model.Types
import           NITTA.Project.Implementation
import           NITTA.Project.Parts.TestBench
import           NITTA.Project.Types
import           NITTA.Utils
import           NITTA.Utils.ProcessDescription
import           Numeric.Interval      (sup, (...), Interval)
import           Text.InterpolatedString.Perl6    (qc)
import           NITTA.Model.ProcessorUnits.ModelSpecification


---------------------------Model maker----------------------------

data DependentPartsModel = DependentPartsModel
 {
   name                :: String
 , verFile             :: String
 } deriving( Show )

smartModel mock puName verFile
 | dependentPartsModel <- DependentPartsModel puName verFile
 = SmartModel{ remain=[], doneAt=Nothing, currentWork=Nothing
     , currentWorkEndpoints=[], process_=def, tick=def
     , isMocked=mock, options=[], intervals=[]
     , dependentPartsModel=dependentPartsModel
     , modelSpecification = Nothing }


-----------------------------Model------------------------------
```

```
-- options - sequence of loaded and unloaded variables
-- intervals -list of intervals for a sequence of loaded and
-- unloaded variables
data SmartModel v x t = SmartModel
 {
   remain               :: [ F v x ]
  , doneAt               :: Maybe t
  , currentWork          :: Maybe ( t, F v x )
  , currentWorkEndpoints :: [ ProcessUid ]
  , process_             :: Process v x t
  , tick                 :: t
  , isMocked             :: Bool
  , options              :: [ EndpointRole v ]
  , intervals            :: [ Interval t ]
  , dependentPartsModel  :: DependentPartsModel
  , modelSpecification   :: Maybe (Specification v x t)
 } deriving(Show)


instance ( VarValTime v x t
         ) => ProcessorUnit (SmartModel  v x t) v x t where
 tryBind f pu@SmartModel  { remain }
   | Just F.Multiply{} <- castF f = Right pu{remain=f : remain}
   | otherwise=Left $ "The function is unsupported by
          AutoMultiplier: " ++ show f
 process = process_
 setTime t pu@SmartModel  {} = pu{ tick=t }

execution pu@SmartModel { remain, options=[], tick } f
   | Right specification <- specificationMaker f
   , (options, intervals) <-
          specificationTraining pu f specification
   = pu
     { currentWork=Just (tick + 1, f)
     , remain=remain \\ [ f ]
     , options = options
     , intervals = intervals
     , modelSpecification = Just specification
     }
execution _ _=error "SmartModel: internal execution error."


instance RefactorProblem (SmartModel v x t) v x

instance ( Time t ) => Default (SmartModel v x t) where
 def = SmartModel
  { remain=[]
  , doneAt=Nothing
  , currentWork=Nothing
```

```
  , currentWorkEndpoints=[]
  , process_=def
  , tick=def
  , isMocked=True
  , options=[]
  , intervals=[]
  , dependentPartsModel=DependentPartsModel "smartMul" "mock.v"
  , modelSpecification=Nothing
  }

instance ( Var v ) => Locks (SmartModel  v x t) v where
 locks SmartModel  { remain, options } =
   [ Lock{ lockBy, locked }
   | locked <- concatMap (elems . variables) remain
   , lockBy <- map (\x -> oneOf $ variables x) options
   ]


checkTarget (Target _) = True
checkTarget _ = False

checkSource (Source _) = True
checkSource _  = False

instance ( VarValTime v x t
        ) => EndpointProblem (SmartModel v x t) v t where
 endpointOptions SmartModel{options=vs@((Target _):_)
     ,intervals=(ins:_),tick}
  =[EndpointSt(head vs)$TimeConstrain(tick+1...tick+sup ins)ins]
 endpointOptions AutoMultiplier{ options=(v:_)
     , intervals=(ins:_), doneAt=Just at, tick }
  =[EndpointSt v$TimeConstrain(max at tick+1...max(at+sup ins)
          (tick+sup ins))ins]
 endpointOptions pu@SmartModel{remain}
          =concatMap(endpointOptions . execution pu) remain

 endpointDecision pu@SmartModel{ remain, options=[] } d
   | let v = oneOf $ variables d
   , Just f <- find (\f -> v `member` variables f) remain
     = endpointDecision (execution pu f) d

 endpointDecision pu@SmartModel{ options=vts@((Target _):_)
     , intervals, currentWorkEndpoints }
     d@EndpointSt{ epRole=Target v, epAt }
  | ([_], xts) <- partition (== (Target v)) vts
  , let sel = if checkSource $ head xts then B else A
  , let (newEndpoints, process_') = runSchedule pu $ do
        updateTick (sup epAt)
        scheduleEndpoint d $ scheduleInstruction epAt $ Load sel
  = pu
```

```
    { process_=process_'
     , currentWorkEndpoints=newEndpoints ++ currentWorkEndpoints
     , doneAt=if checkSource $ head xts
           then Just $ sup epAt + 3 else Nothing
     , tick=sup epAt
     , options=xts
     , intervals=tail intervals
    }

 endpointDecision pu@SmartModel{ options, intervals, doneAt
     , currentWork=Just (a, f), currentWorkEndpoints }
     d@EndpointSt{ epRole=Source x, epAt }
  | not $ null options
  , let vs' = (elems $ variables $ head options) \\ elems x
  , vs' /= (elems $ variables $ head options)
  , let (newEndpoints, process_') = runSchedule pu $ do
   endpoints <- scheduleEndpoint d $ scheduleInstruction epAt Out
           when (null vs') $ do
                high <- scheduleFunction (a ... sup epAt) f
                let low = endpoints ++ currentWorkEndpoints
                establishVerticalRelations high low
           updateTick (sup epAt)
           return endpoints
  = pu
  { process_=process_'
  , doneAt=if null vs' then Nothing else doneAt
  , currentWork=if null vs' then Nothing else Just (a, f)
  , currentWorkEndpoints=if null vs'
         then [] else newEndpoints ++ currentWorkEndpoints
  , tick=sup epAt
  , options=if null vs' then tail options
         else (Source $ fromList vs') : (tail options)
  , intervals=tail intervals
  }

 endpointDecision pu d = error $
     "AutoMultiplier decision error\npu: " ++ show pu
     ++ ";\nd:" ++ show d ++ "\n"


data ArgumentSelector = A | B
 deriving ( Show, Eq )

instance Controllable (SmartModel v x t) where
 data Instruction (SmartModel v x t)
    = Load ArgumentSelector
    | Out
 deriving (Show)

 data Microcode (SmartModel v x t)
```

```
   = Microcode
     {
       wrSignal :: Bool
     , selSignal :: Bool
     , oeSignal :: Bool
     }
    deriving ( Show, Eq, Ord )

 mapMicrocodeToPorts Microcode{..} SmartModelPorts{..}
  =
   [ (wr, Bool wrSignal)
   , (wrSel, Bool selSignal)
   , (oe, Bool oeSignal)
   ]

 portsToSignals SmartModelPorts{ wr, wrSel, oe }=[wr, wrSel, oe]

 signalsToPorts (wr:wrSel:oe:_) _ = SmartModelPorts wr wrSel oe
 signalsToPorts _               _ = error
     "pattern match error in signalsToPorts SmartModelPorts"

instance Default (Microcode (SmartModel v x t)) where
 def = Microcode
        { wrSignal=False
        , selSignal=False
        , oeSignal=False
        }

instance UnambiguouslyDecode (SmartModel v x t) where
 decodeInstruction (Load A)=def{wrSignal=True, selSignal=False}
 decodeInstruction (Load B)=def{wrSignal=True, selSignal=True}
 decodeInstruction Out     =def{oeSignal=True}

instance Connected (SmartModel v x t) where
 data Ports (SmartModel v x t) = SmartModelPorts
    { wr    :: SignalTag -- ^get data from the bus (data_in)
    , wrSel :: SignalTag
    , oe    :: SignalTag -- ^send result to the bus
    } deriving ( Show )

instance IOConnected (SmartModel v x t) where
  data IOPorts (SmartModel v x t) = SmartModelIO
        deriving ( Show )

instance ( VarValTime v x t, Integral x
         ) => Simulatable (SmartModel v x t) v x where
 simulateOn cntx _ f
  | Just f'@F.Multiply{} <- castF f = simulate cntx f'
  | otherwise=error$"Can't simulate on AutoMultiplier: "++show f
```

```
instance ( VarValTime v x t
         ) => TargetSystemComponent (SmartModel v x t) where
 moduleName _title _pu = "pu_multiplier"
 software _ _ = Empty
 hardware tag pu@SmartModel{ dependentPartsModel
     =DependentPartsModel{verFile} }
  = Aggregate Nothing
    [ FromLibrary verFile
    , FromLibrary $ "multiplier/" ++ moduleName tag pu ++ ".v"
    ]

 hardwareInstance tag _pu TargetEnvironment{
     unitEnv=ProcessUnitEnv{..}, signalClk, signalRst }
     SmartModelPorts{..} SmartModelIO
   = codeBlock [qc|
       pu_multiplier #
           ( .DATA_WIDTH( { finiteBitSize (def :: x) } )
           , .ATTR_WIDTH( { parameterAttrWidth } )
           , .SCALING_FACTOR_POWER({fractionalBitSize (def::x)})
           , .INVALID( 0 )
           ) { tag }
           ( .clk( {signalClk} )
           , .rst( {signalRst} )
           , .signal_wr( { signal wr } )
           , .signal_sel( { signal wrSel } )
           , .data_in( { dataIn } )
           , .attr_in( { attrIn } )
           , .signal_oe( { signal oe } )
           , .data_out( { dataOut } )
           , .attr_out( { attrOut } )
           );
      |]
 hardwareInstance _title _pu TargetEnvironment{
     unitEnv=NetworkEnv{} } _ports _io
  = error "Should be defined in network."

instance IOTestBench (SmartModel v x t) v x

instance (Ord t)=>WithFunctions (SmartModel v x t) (F v x) where
 functions SmartModel{ process_, remain, currentWork }
   = functions process_
     ++ remain
     ++ case currentWork of
        Just (_, f) -> [f]
        Nothing     -> []

instance ( VarValTime v x t, Integral x
         ) => Testable (SmartModel v x t) v x where
 testBenchImplementation prj@Project{ pName, pUnit }
   = Immediate (moduleName pName pUnit ++ "_tb.v")
```

```
$ snippetTestBench prj SnippetTestBenchConf
  { tbcSignals=["oe", "wr", "wrSel"]
  , tbcPorts=AutoMultiplierPorts
       { oe=SignalTag 0
       , wr=SignalTag 1
       , wrSel=SignalTag 2
       }
  , tbcIOPorts=AutoMultiplierIO
  , tbcSignalConnect= \case
       (SignalTag 0) -> "oe"
       (SignalTag 1) -> "wr"
       (SignalTag 2) -> "wrSel"
        _ -> error "testBenchImplementation wrong signal"
  , tbcCtrl= \Microcode{ oeSignal, wrSignal, selSignal} ->
       [qc|oe <= {bool2verilog oeSignal}; wr <=
         {bool2verilog wrSignal}; wrSel <=
         {bool2verilog selSignal};|]
  , tbDataBusWidth=finiteBitSize (def :: x)
  }
```