|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

***К КУРСОВОЙ РАБОТЕ***

***НА ТЕМУ:***

***Система визуализации использования ресурсов процессора входной программой. Применение системы для виртуализации троичного процессора.***

Студент ИУ7-21М **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Ермаков А. Ю.**

(Подпись, дата)

Руководитель курсовой работы **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Щетинин Г. А.**

(Подпись, дата)

Консультант **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Строганов Ю. В.**

(Подпись, дата)

*2020 г.*

Содержание

[Глоссарий 2](#_Toc38345631)

[Введение 3](#_Toc38345632)

[1 Аналитический раздел 7](#_Toc38345633)

[1.1 Обзор предметной области 7](#_Toc38345634)

[1.1.1 Intel 4004 Emu 8](#_Toc38345635)

[1.1.2 E4004 (Intel 4004 System Emulator) 9](#_Toc38345636)

[1.1.3 Proteus ISIS 10](#_Toc38345637)

[1.1.4 Эмулятор EMU 11](#_Toc38345638)

[1.1.5 Виртуализация 13](#_Toc38345639)

[1.2 Устройство современных процессоров 15](#_Toc38345640)

[1.2.1 Использование конвейера 16](#_Toc38345641)

[1.2.2 Суперскалярность 18](#_Toc38345642)

[1.2.3 Многоядерность 21](#_Toc38345643)

[1.2.4 Архитектура как набор команд 22](#_Toc38345644)

[1.2.5 Микрокод 24](#_Toc38345645)

[1.3 Компиляторы 25](#_Toc38345646)

[1.4 Троичная система счисления 26](#_Toc38345647)

[1.4.1 Трёхзначная логика Лукасевича 27](#_Toc38345648)

[1.4.2 Трёхзначные логики Клини и Приста 29](#_Toc38345649)

[1.5 Функционал приложения и сфера применения 31](#_Toc38345650)

[1.6 Выводы 33](#_Toc38345651)

[Список использованных источников 34](#_Toc38345652)

# Глоссарий

Система – совокупность элементов и их связей, образующая единое целое и имеющая набор свойств, отсутствующий у элементов по отдельности.

Визуализация – представление информации в виде, пригодном для зрительного анализа.

Платформа – совокупность аппаратного и управляющего им программного обеспечений.

Архитектура процессора – набор свойств процессора, определяемый устройством процессора, составными компонентами процессора и связями между ними.

Архитектура фон Неймана – устройство процессора, при котором данные и команды располагаются в единой (общей) памяти.

Гарвардская архитектура – устройство процессора, при котором данные и команды хранятся в раздельной памяти.

CISC (RISC, VLIW, MISC) архитектура – устройство процессора, при котором он поддерживает соответствующий набор команд (CISC, RISC, VLIW, MISC).

Система счисления – метод записи чисел, позволяющий однозначно представить число с помощью последовательности символов.

Позиционная система счисления – вид системы счислений, в которой позиция(разряд) символа определяет его значение.

Основание системы счисления – количественная характеристика системы счисления, определяющая количество различных символов, используемых для записи числа в этой системе счисления.

Троичная система счисления – позиционная система счисления с основанием 3.

# Введение

Количество устройств, оснащённых процессорами и микроконтроллерами, в современном мире неуклонно растёт: «умные» телевизоры, холодильники и часы уже давно ни для кого не в новинку, а смартфоны, персональные компьютеры и ноутбуки и вовсе давно стали обыденностью [1]. Программируемые станки используются для автоматизации производства, терминалы - для ускорения и упрощения выбора и покупок товаров, выполнения банковских операций. Столь широкое распространение вычислительных устройств приводит к большому спросу на квалифицированных разработчиков, способных писать программное обеспечение для различных платформ [2]. Встаёт вопрос обучения и переобучения кадров: несмотря на то, что в последние десятилетия всё большую популярность обретают интерпретируемые языки программирования и языки программирования высокого уровня, которые не имеют жёсткой привязки к платформе (для запуска программы, написанной на интерпретируемом языке программирования на различных платформах достаточно иметь интерпретатор, разработанный для этой платформы – исходный текст программы при этом не претерпит никаких изменений, для языков программирования высокого уровня в случае, если не использовались платформо-зависимые библиотеки, для запуска программы на другой платформе достаточно скомпилировать программу под необходимую платформу), машинно-зависимые языки программирования по-прежнему широко распространены [3]. На сегодняшний день распространены 8-битные микроконтроллеры AVR, 32-битные микроконтроллеры STM, 32-битные микроконтроллеры ESP, 8-битные, 16-битные и 32-битные микроконтроллеры PIC [4]. Эти микроконтроллеры имеют различный набор доступных программисту регистров, команд, памяти. Несмотря на то, что существуют C/C++ компиляторы для этих микроконтроллеров, периодически возникают задачи, для которых необходимо писать код на низкоуровневых языках программирования. Связано это, в первую очередь, с быстродействием. Очевидно, что для различных микроконтроллеров исходный код будет отличаться.

При подготовке разработчиков программного обеспечения им часто читаются курсы машинно-зависимых языков программирования, архитектуры электронных вычислительных машин и операционных систем [5]. Таким образом, обучающимся закладывают понимание того, каким образом функционируют вычислительные устройства, какие операции выполняются при исполнении программного кода, каким образом программист может сэкономить время выполнения программы или объём используемой памяти при написании программного продукта. Обозначенные выше дисциплины тесно связаны между собой, и вместе с тем достаточно сложны в изучении и понимании.

Визуализация процессов, протекающих в процессоре (микроконтроллере) во время выполнения кода программы может существенно упростить обучение и понимание происходящего. Отслеживание на экране текущих значений, хранящихся в различных регистрах процессора, а также их изменения, вызванные выполнением очередной команды, позволяет более тонко и полно понять влияние очередной команды на состояние процессора, определить побочные эффекты, наиболее используемые и неиспользуемые компоненты процессора. Предоставление отчёта, содержащего статистику использования компонентов за всё время выполнения программы, позволяет определить, какие компоненты процессора используются часто или не используются за всё время выполнения программы, а не только очередной команды. Подобная статистика может помочь программисту использовать доступные ресурсы равномерно, снизить нагрузку на определённые компоненты.

Целью данной работы является разработка программного обеспечения, пошагово (покомандно) исполняющего поданную на вход программу для выбранного процессора, отображающего текущее состояние процессора, а также собирающего статистику об использовании ресурсов процессора программой и предоставляющего по завершении выполнения отчёт, содержащий эту статистику. Для достижения поставленной цели необходимо решить следующие задачи:

* провести обзор предметной области: рассмотреть существующие аналоги, определить их достоинства и недостатки;
* рассмотреть структуру процессоров;
* определить сферу применения разработанного программного обеспечения;
* рассмотреть существующие подходы к моделированию процесса выполнения программного кода на процессоре;
* определить, какую статистику необходимо собирать и включать в отчёт;
* разработать представление процессора как совокупности составляющих его компонентов и их значений;
* разработать представление команд процессора;
* определить способ влияния команд на значения компонентов процессора;
* определить способ сбора статистики;
* разработать интерфейс для отображения значений элементов и отображения отчёта с собранной статистикой;
* реализовать программное обеспечение;
* реализовать тесты, проверяющие работоспособность программного обеспечения;
* визуализировать процесс выполнения программы, транслированной различными компиляторами;

В качестве примере было решено использовать троичный процессор, поэтому дополнительно необходимо проанализировать существующие троичные логики, определить наиболее подходящую для построения на её основе процессора, произвести обзор существующих решений и определить архитектуру эмулируемого процессора.

# 1 Аналитический раздел

В данном разделе проведён обзор предметной области: рассмотрены существующие аналоги, определены их достоинства и недостатки, изучены существующие процессоры и принципы их разработки. Рассмотрены существующие троичные логики, и на их основе составлено описание эмулируемого процессора. Опираясь на полученные результаты, определены основные функции разрабатываемого программного продукта, сфера применения разработанного продукта. Определена степень детализации симуляции использования ресурсов процессора входной программой. Сформулирован список параметров, для которых в процессе симуляции необходимо собирать статистику. Основываясь на проделанных исследованиях выдвинуть требования к программному продукту.

## Обзор предметной области

Всё множество существующих эмуляторов было сокращено до четырёх, основываясь на следующих критериях:

* свободное распространение;
* функционал эмуляторов должен отличаться;
* рассматриваются эмуляторы с различным интерфейсом;
* эмуляторы различных процессоров, чтобы оценить различия решаемых задач;
* два эмулятора одного процессора, чтобы оценить различия реализации одной задачи;
* различные варианты распространения (онлайн-сервис, приложение, требующее установки, приложение, не требующее установки).

Рассматриваются следующие эмуляторы процессоров:

* Intel 4004 Emu;
* E4004 (Intel 4004 System Emulator);
* Proteus ISIS;
* Эмулятор EMU.

Отдельно от эмуляторов также рассмотрена технология виртуализации.

### 1.1.1 Intel 4004 Emu

Эмулятор процессора Intel 4004, разработанный на языке python. Распространяется в виде скрипта, который можно запустить после скачивания с репозитория разработчика на GitHub [6], также доступна online-версия [7]. Существует документация, оформленная в виде wiki на том же репозитории [8]. Графический интерфейс пользователя отсутствует, для исполнения программы на языке ассемблера имя файла с программой передаётся в качестве аргумента командой строки при запуске скрипта. Для отслеживания значений, хранящихся в регистрах, необходимо в исходный код программы на ассемблере добавить вызов функции печатающей на экран значения регистров. Пример исполнения программы, печатающей на экран строку, и программы, подсчитывающей сумму первых N членов арифметической прогрессии с разностью прогрессии равной 1, представлен на рисунке 1.

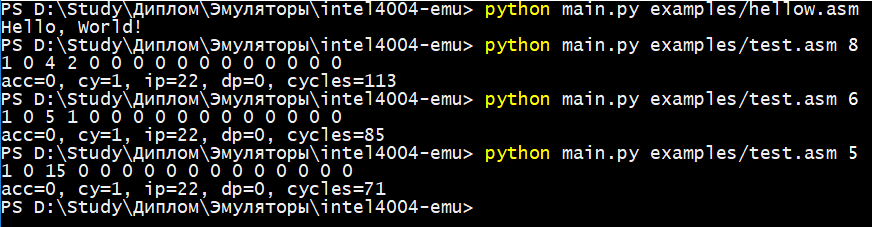


Рисунок 1. Исполнение программ на эмуляторе Intel 4004 Emu

Online-версия эмулятора имеет графический интерфейс пользователя, который содержит поле ввода исходного кода программы, а также поле, отображающее значения всех регистров после выполнения программы. Дополнительно имеется поле ввода, позволяющее задавать исходные значения регистров (по умолчанию все регистры занулены). Имеется возможность пользовательского ввода данных, если программа считывает данные с клавиатуры. Интерфейс online-версии эмулятора представлен на рисунке 2.

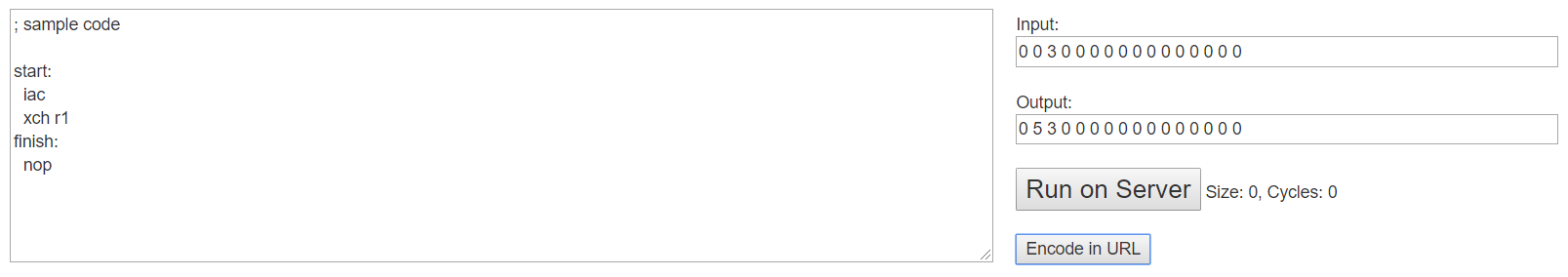


Рисунок 2. Интерфейс online-версии эмулятора Intel 4004 Emu

### 1.1.2 E4004 (Intel 4004 System Emulator)

E4004 также является эмулятором процессора Intel 4004. Не требует установки, так как запускается из браузера (существует только online-версия) [9]. Эмулирует не только работу центрального процессора, но и всей экосистемы (ОЗУ и ПЗУ). На вход эмулятору подаётся машинный код, который необходимо загрузить в ПЗУ. Также можно загрузить в ПЗУ код на языке ассемблера. Существует три режима эмуляции: STEP (пошаговый), ANIMATE (то же, что и пошаговый, но переключение шагов происходит автоматически) и RUN (исполнение программы). Есть возможность остановки эмуляции и сброса к исходному состоянию. Пользователь может отслеживать значения, хранящиеся в ОЗУ (4 чипа по 4 банки, состоящие из 4 регистров размером в 4 бита), состояние всех регистров (16 регистров общего назначения, аккумулятор), стек, флаги carry и test. В дополнение к эмулятору предоставляется компилятор и дизассемблер машинного кода. Интерфейс эмулятора представлен на рисунке 3.

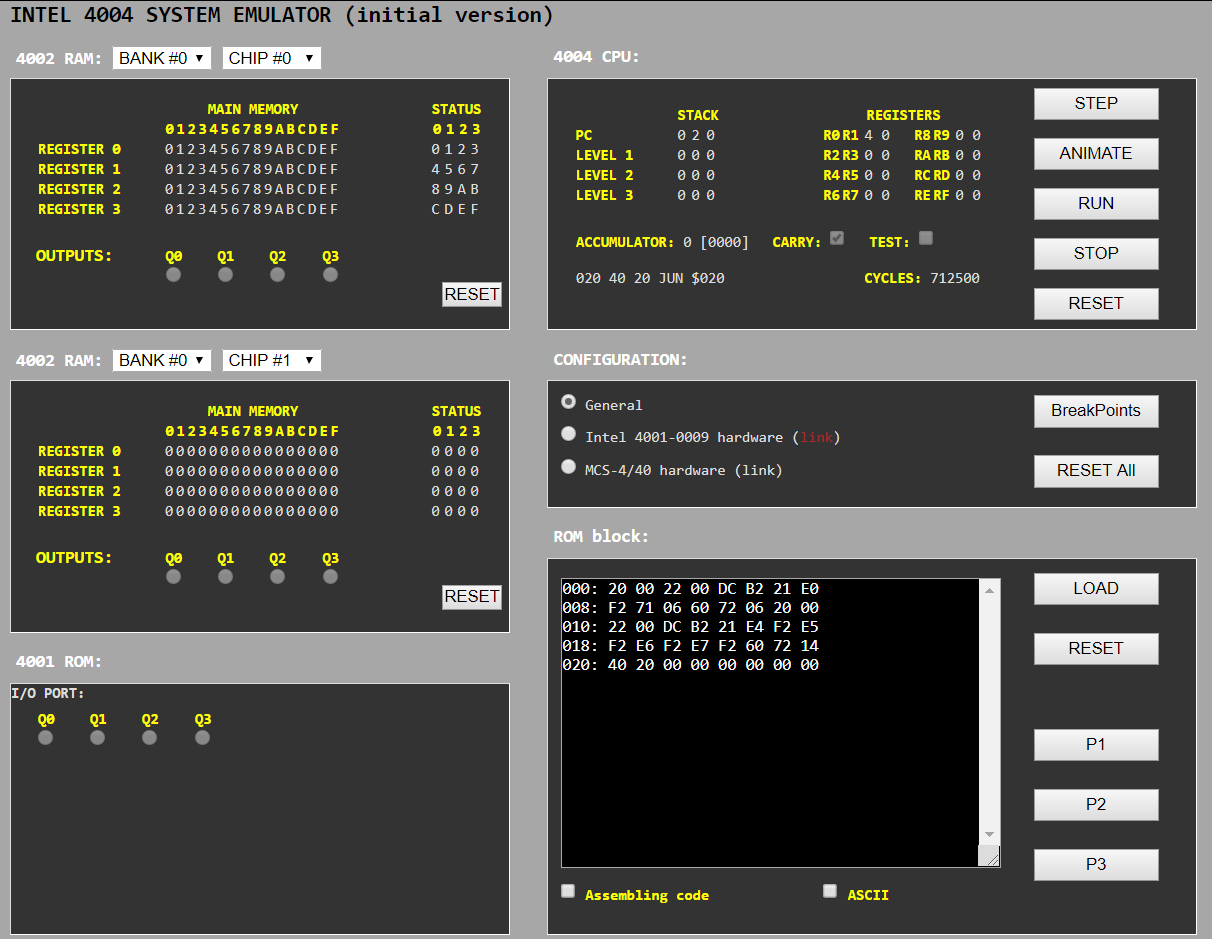


Рисунок 3. Интерфейс эмулятора E4004 (Intel 4004 System Emulator)

### 1.1.3 Proteus ISIS

Proteus ISIS является приложением для разработки и моделирования электронных схем [10]. Пользователю доступны различные активные и пассивные элементы, начиная с резисторов, конденсаторов и индуктивностей и заканчивая микросхемами и микроконтроллерами. Имеется набор средств измерения, включающий в себя амперметр, вольтметр, частотомер, осциллограф и прочее. Если собранная цепь включает в себя микроконтроллер, пользователь может загрузить для него готовый файл прошивки (.hex файл), или скомпилировать программу, написанную на языке ассемблера или C/C++. Таким образом, у пользователя есть выбор микроконтроллера, на котором он может протестировать программу. Возможность симуляции пошагово не предусмотрена. Также пользователь не может отслеживать состояние различных регистров микроконтроллера, что делает отладку программного кода затруднительной, а процесс изучения нового аппаратного обеспечения тяжким. Proteus ISIS рассчитан в первую очередь на людей, которые разрабатывают электронные устройства, и призван упростить процесс их разработки и прототипирования. Пример симуляции схемы с микроконтроллером представлен на рисунке 4.

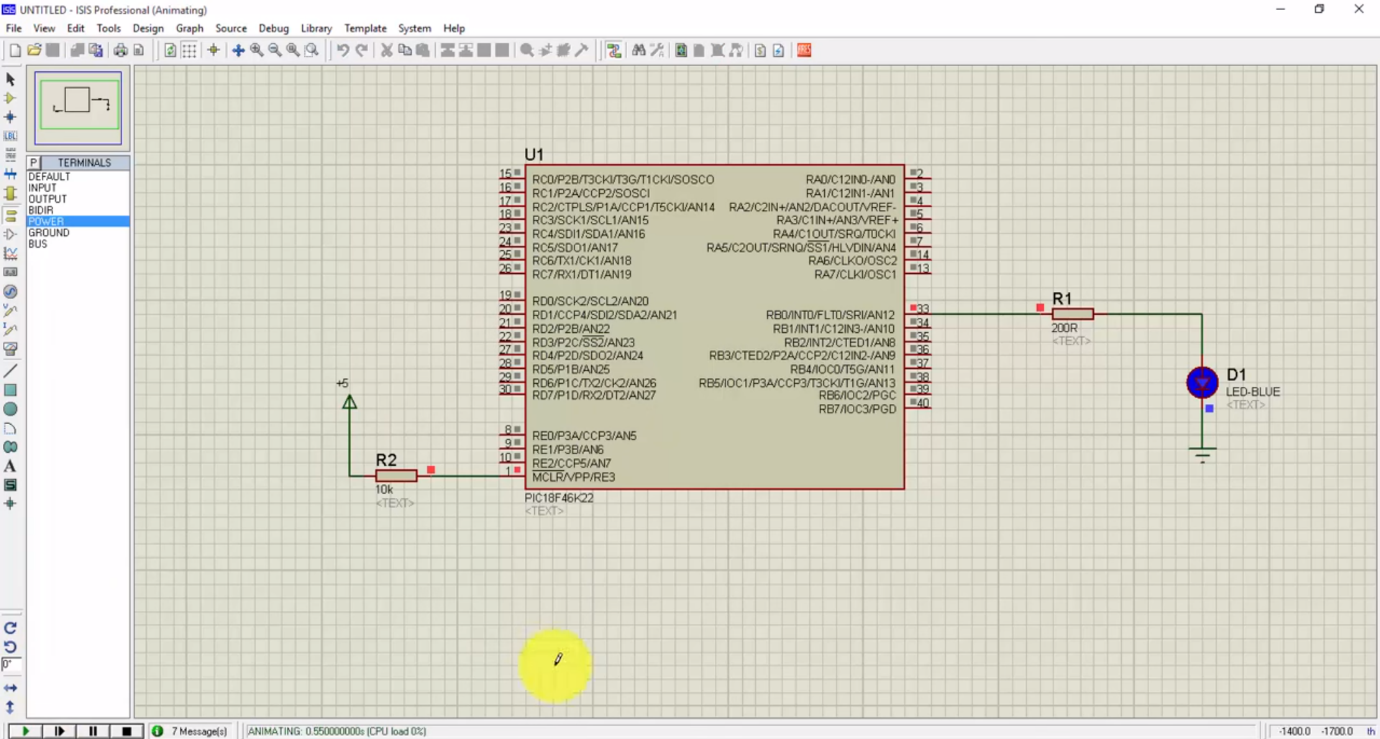


Рисунок 4. Симуляция схемы с микроконтроллером и программой, переключающей состояния светодиода

### 1.1.4 Эмулятор EMU

Эмулятор EMU [11] представляет собой виртуальную машину, позволяющую пользователю исследовать поведение различных компонентов ЭВМ. В состав эмулятора входят следующие программные модели:

* процессор x86;
* чипсет Intel i815EP;
* контроллер клавиатуры PS/2;
* контроллер накопителя на гибких магнитных дисках;
* контроллер последовательного порта;
* контроллер параллельного порта;
* монитор состояния процессора и системной платы;
* видеоадаптер VGA;
* монитор.

Эмулятор поддерживает пошаговый и непрерывный режим выполнения программ, остановку программы в заданных точках (break point), просмотр значений регистра центрального процессора, регистров контроллеров, содержимого оперативной памяти и стека. Пример работы эмулятора представлен на рисунке 5.

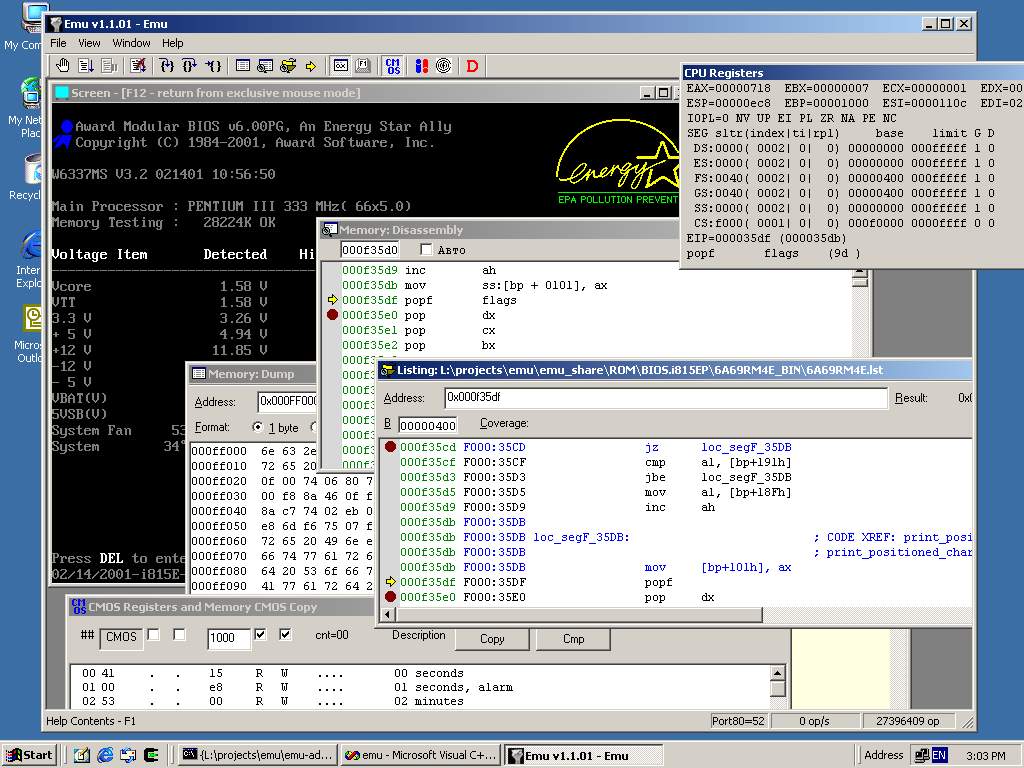


Рисунок 5. Пример работы эмулятора EMU

Результаты сравнения рассмотренных выше эмуляторов представлены в таблице 1.

Таблица 1 – Сравнение возможностей рассмотренных эмуляторов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Эмулятор | Наличие пошагового режима | Возможность выбора процессора | Возможность добавления процессора | Отображение значений регистров |
| Intel 4004 EMU | - | - | - | + |
| E4004 (Intel 4004 System Emulator) | + | - | - | + |
| Proteus ISIS | - | + | + | - |
| Эмулятор EMU | + | - | - | + |

### 1.1.5 Виртуализация

Под виртуализацией понимают предоставление вычислительных ресурсов, абстрагированное от их аппаратной реализации, и изоляцию процессов, выполняемых на одном физическом ресурсе. [12] Иными словами, виртуализацию можно понимать как программную прослойку между аппаратной реализацией и вычислительными процессами, предоставляющую единый интерфейс доступа к аппаратному обеспечению вне зависимости от его реализации, а также решающую проблему изоляции вычислительных процессов друг от друга. Яркими примерами виртуализации являются: виртуальная память (виртуализация оперативной памяти, изолирующая адресные пространства процессов), виртуальная файловая система (виртуализация файловой системы, предоставляющий процессами единообразный доступ к конкретным реализациям файловой системы), виртуальные частные сети (виртуализация сетевых соединений поверх других, уже существующих соединений).

Виртуализация, как правило, находит своё применение в следующих областях:

* виртуальные машины;
* виртуализация ресурсов;
* виртуализация приложений.

Виртуальная машина представляет собой окружение, которое исполняет роль аппаратного обеспечения для исполняемой на ней операционной системы. Операционная система, исполняемая внутри виртуальной машины, не имеет представления о том, что её аппаратным обеспечением на самом деле является его программная реализация. Подобные виртуальные машины позволяют запускать одновременной на одной реальной машине несколько операционных систем, в том числе разработанных для различных устройств и архитектур процессоров. Исполняемые операционные системы не могут непосредственно влиять друг на друга и на «домашнюю» операционную систему. Рассмотренные выше эмуляторы относятся к данной области применения. Виртуальные машины также часто применяют для эмуляции игровых консолей, а также тестирования программного обеспечения и обучения, затрагивающего компоненты операционной системы и потенциально опасного для неё. Также существует отдельный вид виртуальных машин, реализующий паравиртуализацию – такие виртуальные машины не представляются аппаратным обеспечением «гостевой» операционной системы, а вместо этого предоставляют специальный API для взаимодействия с аппаратным обеспечением.

Под виртуализацией ресурсов понимают технологию замены одного реального ресурса множеством виртуальных: например, виртуализация одного физического жёсткого диска множеством виртуальных. При этом зачастую виртуализированный ресурс доступен ограниченному ряду пользователей. В частности, жёсткий диск сервера может быть представлен несколькими виртуальными, причём количество виртуальных дисков совпадает с количеством пользователей, и каждому пользователю доступен только один виртуальный диск. Возможна и обратная ситуация, при которой множество физических ресурсов представляется одним виртуальным. Например, множество жёстких дисков может быть представлено одним виртуальным. Подобным подход используется в популярной технологии RAID (Redundant Array of Independent Disks – избыточный массив независимых дисков).

Виртуализация приложения – процесс, при котором приложение, требующее установки преобразуется в приложение, установки не требующее. Для виртуализации приложения специальная программа-виртуализатор определяет, какие компоненты операционный системы необходимы для запуска виртуализируемого приложения и эмулирует их. Таким образом, для запускаемого приложения создаётся среда, которая изолирует это приложение от окружающей операционной системы. Виртуализация приложений позволяет избежать эмуляции всей системы, что повышает быстродействие по сравнению с запуском приложения на виртуальной машине.

## 1.2 Устройство современных процессоров

Современные процессоры представляют собой устройства высокой сложности и состоят из множества элементов. Структура элементов, их связи и количество, функционал процессоров – все эти параметры зависят от конкретной модели. Вместе с тем, существуют определённые подходы и решения, применяемые во многих современных процессорах.

Рассмотрение процессоров стоит начать с рассмотрения архитектуры фон Неймана [13]. Под архитектурой фон Неймана понимают принцип организации хранения команд и данных, при котором данные и команды хранятся в общей памяти. При таком подходе отличить данные от команд не представляется возможным, что открывает перед разработчиками большие возможности: имеется возможность написания программ, которые в ходе исполнения могут менять свой собственный код. Более распространён на практике и другой подход: генерация кода программы другой программой. Такую роль, как правило, играют компиляторы, преобразующие программы с языков программирования в машинные инструкции. Однако, из-за того, что память и данные используют общую память, появляется проблема, известная как узкое место архитектуры фон Неймана – программа и данные не могут быть доступны в одно время, пропускная способность канала «процессор-память», ровно как и скорость памяти, оказывают существенное влияние на быстродействие системы. Одним из способов решения данной проблемы является добавление дополнительной памяти с высокой скоростью доступа, но низким объёмом. Такая память, как правило, располагается на одном кристалле с процессором и называется кэш-памятью. Как правило, существует несколько уровней кэш-памяти с различной скоростью доступа и объёмом. Основная идея кэш-памяти заключается в том, что данные, считанные из оперативной памяти, помещаются в кэш-память и, при следующей попытке считать эти же данные из оперативной памяти, считываются из кэша.

Другим способом борьбы с проблемой фон Неймана является использование гарвардской архитектуры [13]. При реализации такой архитектуры данные и команды хранятся на отдельных физических устройствах. Таким образом, возможно одновременное считывание данных и команд, что позволяет повысить быстродействие и избавиться от кэш-памяти. При использовании гарвардской архитектуры, тем не менее, значительно увеличивается количество выводов, так как теперь имеются отдельные шины данных и адреса для команд и данных. Модифицированная гарвардская архитектура позволяет использовать для внешних устройств одну шину данных и адреса, а внутри процессора – отдельные шины данных, команд и две шины адреса. В современных микроконтроллерах общая шина команд и данных используется даже на одном кристалле.

В целях повышения быстродействия процессоров прибегают к различным способам повышения параллелизма. В пределах процессора существует три способа увеличения производительности за счёт параллельного исполнения команд:

* использование конвейера;
* суперскалярность;
* многоядерность.

### 1.2.1 Использование конвейера

При использовании конвейера каждая команда разбивается на последовательность действий, отдельные блоки процессора выполняют эти действия, что приводит к тому, что одновременно могут выполняться различные команды. Таким образом, выполнение очередной инструкции может происходить не после выполнения предыдущей, а через несколько стадий после неё. Скорость конвейера определяется скоростью самого медленного действия. Подобный процесс обработки команд гораздо быстрее, чем последовательное выполнение команд полностью. Процесс обработки последовательности команд на пятиуровневом конвейере представлен на рисунке 6.

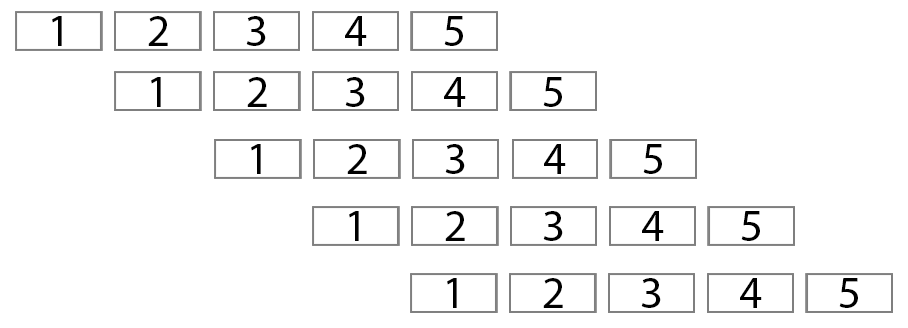


Рисунок 6. Обработка последовательности команд на пятиуровневом конвейере.

Вдоль горизонтальной оси откладывается время, вдоль вертикальной – независимые команды. Цифрами обозначены уровни конвейера.

Конвейер с большим количеством стадий (20 и более) называют супер-конвейером (гипер-конвейером). Если в конвейере в единицу времени не все стадии конвейера нагружены, такой конвейер называют недонагруженным.

Ситуация, при которой невозможно выполнить очередную команду в очередной такт, называется конфликтом конвейера. Для разрешения конфликтов необходимо, чтобы часть команд была задержана, а другая продолжила выполнение. Выделяют три типа конфликтов:

* структурные конфликты;
* конфликты по данным;
* конфликты по управлению.

Структурный конфликт возникает в том случае, если процессор физически не может выполнить текущий набор команд. Например, если одна из команд обращается к памяти, а другая команда в этот момент из памяти считывается, при этом процессор не может одновременно выполнить доступ к памяти и для получения данных, и для получения команды. Подобная ситуация возникает, как правило, если процессор не полностью поддерживает конвейер.

Конфликт по данным возникает в том случае, если последовательность команд, находящихся в конвейере, имеет зависимость по данным такую, что от порядка выполнения команд зависит результат.

Конфликт по управлению возникает в тех случаях, когда в последовательности команд встречаются команды, изменяющие счётчик команд произвольным способом: условные и безусловные переходы, циклы, вызовы подпрограмм. Для избегания конфликта по управлению применяют предсказатель переходов, точность которого в современных процессорах достигает 0.9.

### 1.2.2 Суперскалярность

Технология, обеспечивающая параллелизм на уровне инструкций, за счёт включения в состав процессора дублирующих функциональных узлов. Распределением инструкций между одинаковыми функциональными блоками занимается динамическое вычислительное ядро или статический компилятор. Производительность повышается за счёт одновременного выполнения нескольких инструкций на одинаковых функциональных блоках процессора.

В состав суперскалярых процессоров входит специальный блок, называемый диспетчером инструкций (Instruction Dispatcher). Этот узел отвечает за чтение инструкций из памяти, определение, какие инструкции могут быть выполнены одновременно, распределение инструкций между функциональными блоками процессора. Для повышения производительности необходимо, чтобы диспетчер инструкций одновременно задействовал максимально возможное количество функциональных блоков. Если диспетчер инструкций не справляется со своими функциями, производительность суперскалярного процессора не превышает производительности скалярного процессора. На рисунке 7 изображён процесс обработки последовательности команд на суперскалярном конвейерном процессоре.

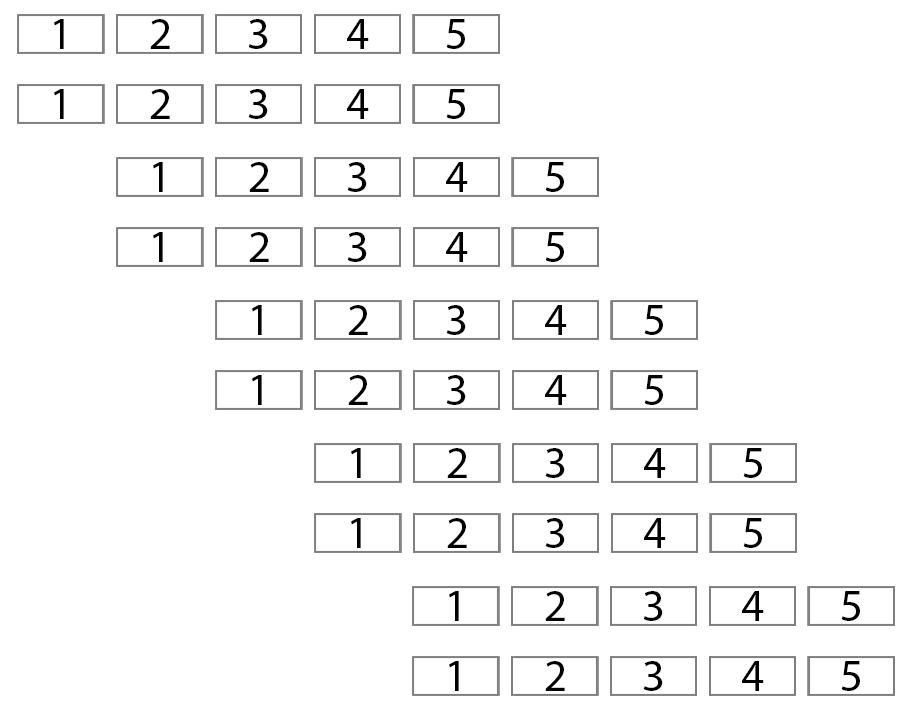


Рисунок 7. Процесс обработки команд на суперскалярном пятиуровневом конвейерном процессоре

Вдоль горизонтальной оси откладывается время, вдоль вертикальной – независимые команды. Цифрами обозначены уровни конвейера.

Для суперскалярных процессоров характерны следующие методы увеличения производительности:

* внеочередное исполнение;
* переименование регистров.

Суть внеочередного исполнения заключается в том, что, в отличии от исполнения по порядку, инструкции выполняются в порядке готовности, а не в порядке следования. При выполнении инструкций по порядку процессор выполняет следующие действия:

* считывание очередной инструкции;
* ожидание загрузки операндов;
* передача инструкции на выполнение ответственному за неё модулю;
* выполнение инструкции;
* запись результата выполнения инструкции.

Внеочередное исполнение инструкций выполняется в соответствии со следующими шагами:

* считывание инструкции;
* помещение инструкции в очередь ожидания;
* ожидание доступности любых операндов из очереди ожидания;
* передача на выполнение инструкции, операнды которой доступны, соответствующему модулю;
* выполнение инструкции;
* запись результата в очередь результатов;
* извлечение из очереди тех результатов, перед которыми в очереди инструкций не осталось невыполненных инструкций;
* удаление соответствующей инструкции из очереди инструкций;
* запись результата.

За счёт применения внеочередного исполнения инструкций удаётся снизить простой, вызванный ожиданием получения операндов из оперативной памяти.

Переименование регистров – метод ослабления взаимосвязи команд при их внеочередном исполнении. Также используется с конвейерными процессорами. Суть метода заключается в том, что несколько команд (находящихся на конвейере или в очереди инструкций) могут производить запись в один и тот же регистр. При этом если команды не производят считывания данных из этого регистра, зависимости по данным нет. Однако конечный результат зависит от того, какая команда последней записала данные в регистр. Для того чтобы данные в регистр были записаны в соответствии с порядком, определяемым последовательностью команд, архитектурные регистры дублируют набором физических. Таким образом, одному архитектурному регистру может соответствовать множество физических. Процессор при этом отслеживает, состояние какого физического процессора соответствует состоянию архитектурного.

Стоит понимать, что существует ряд ограничений, снижающих эффективность суперскалярности. Главным образом это:

* степень внутреннего параллелизма инструкций;
* сложность поиска зависимостей между инструкциями;
* сложность схемы переименования регистров;
* обработка инструкций ветвления.

Степень внутреннего параллелизма инструкция зависит от того, насколько инструкции зависят от других инструкций. Это определяется использованием общих ресурсов или использованием результата выполнения других инструкций. Невозможно одновременно выполнить инструкции A и B, если выполнение инструкции B зависит от результата выполнения инструкции A.

Увеличение количества одновременно выполняемых инструкций упирается в сложность поиска зависимостей между ними и время, которое на это уходит. Нецелесообразно дальнейшее увеличение одновременно выполняемых инструкций, если при этом на определение зависимостей будет уходить более одного такта.

### 1.2.3 Многоядерность

Многоядерность – метод увеличения производительности процессора, путём размещения нескольких вычислительных ядер на одном кристалле. Помимо многоядерных процессоров существуют также многочиповые процессоры – в них несколько ядер размещены не на одном кристалле, а на нескольких, но все кристаллы помещены в общий корпус. Современные многоядерные процессоры имеют раздельные кэши первого и, иногда, второго уровня, а также общий кэш третьего и, иногда, второго уровня. Если кэши второго уровня реализованы для каждого ядра отдельно, как правило, существует механизм их синхронизации. Современные многоядерные процессоры, как правило, имеют заниженную частоту, т.к. с повышением частоты энергопотребление растёт согласно кубической зависимости. При этом увеличение количества ядер позволяет добиться сопоставимую производительность при меньшем энергопотреблении. Современные процессоры, применяемые в мобильных устройствах, оснащены технологиями автоматической подстройки тактовой частоты к нагрузке на процессоре, что позволяет в случае необходимости увеличить производительность, а вместе с ней и энергопотребление. Применение суперскалярных ядер позволяет использовать «логические» ядра.

Отдельные ядра процессора могут иметь одинаковую архитектуру и использоваться для решения одинаковых задач – такие процессоры называют процессорами с гомогенной архитектурой – а также могут иметь различную архитектуру и использоваться для решения различных задач – в таком случае процессоры называют процессорами с гетерогенной архитектурой.

Стоит понимать, что производительность многоядерных процессоров напрямую зависит от того, насколько оптимизирована программа для запуска в многопоточном режиме. В случае, если программа не оптимизирована и выполняется в одном потоке, будет задействовано лишь одно ядро процессора, что может не только не дать прироста в производительности, но и привести к её снижению в сравнении с одноядерным процессором с повышенной тактовой частотой.

### 1.2.4 Архитектура как набор команд

Помимо различий в аппаратной реализации, современные процессору предоставляют программистам различный набор доступных регистров и команд. Выделяют несколько концепций построения процессоров, наиболее популярные из которых:

* CISC (Complex Instruction Set Computing)
* RISC (Reduced Instruction Set Computing)
* MISC (Multipurpose Instruction Set Computing)
* VLIW (Very Long Instruction Word)

Архитектура CISC характеризуется большим числом различных команд, различной длинной команд, применением большого числа различных режимов адресации и сложной кодировкой инструкций. К достоинствам CISC архитектуры можно отнести компактность исходного кода программы, что уменьшает объём программ и количество обращений к памяти. К сожалению, декодирование команд занимает достаточно большое время, распараллеливание CISC-команд затруднено, а стоимость аппаратной реализации достаточно высока.

Архитектура RISC характеризуется упрощённым форматом команд. Длина команд фиксирована, количество режимов адресации ограничено: обращение к памяти происходит посредством специальных команд загрузки и записи данных, все остальные команды работают с регистрами. Дешифрация команд упрощена по сравнению с архитектурой CISC. Инструкции RISC архитектуры проще поддаются параллелизму. В связи с тем, что архитектура RISC проще архитектуры CISC, стоимость аппаратной реализации ниже. Энергопотребление RISC-процессоров также низкое. Командный набор архитектур RISC и CISC несовместим. На протяжении долгого времени домашние компьютеры имели в своём составе процессоры с CISC архитектурой, т.к. именно под него было разработано большинство существующего программного обеспечения. Процессоры с RISC архитектурой применялись, в основном, для мобильных устройств.

Архитектура MISC является объединением архитектуры RISC и CISC с целью получения достоинств обеих архитектур. Ядром MISC архитектуры является RISC-процессор, к которому подключается специальный модуль, отвечающий за трансляцию поступающих CISC-команд в соответствующий набор RISC-команд. Подобный подход позволяет объединить простоту и скорость RISC архитектуры с компактностью кода и совместимостью с CISC-архитектурой, что позволяет использовать MISC процессоры в современных домашних компьютерах.

VLIW-архитектура подразумевает наличие нескольких дублирующих друг-друга функциональных блоков. При этом решение о одновременном выполнении команд принимает не специальный блок процессора, а компилятор. Команды VLIW архитектуры могут достигать размера в 128-256, т.к. в них кодируются инструкции, которые должны быть вычислены одновременно, а также модули, на которых эти инструкции должны быть выполнены. Также как и процессоры с RISC-архитектурой, процессоры с VLIW-архитектурой сравнительно просты в исполнении, а потому дёшевы. Энергопотребление VLIW-процессоров также низко.

### 1.2.5 Микрокод

Микрокод – программа, реализующая машинную инструкцию. Процессоры, использующие микрокод, преобразуют очередную машинную инструкцию в набор команд микрокода, который затем выполняется процессором. Часто микрокод хранится не в самом процессоре, а в отдельной микросхеме памяти. Такая память может быть доступна как только для чтения, так и для чтения-записи, что позволяет изменять набор доступных машинных инструкций с целью расширения последнего или модификации и оптимизации существующих команд. Микрокод является самым низкоуровневым набором команд и используется для управления аппаратным обеспечением. В частности при помощи микрокода может происходить коммутация различных блоков процессора. Микрокод является альтернативой аппаратной реализации машинных инструкций и упрощает процесс разработки, отладки и тестирования процессоров. Иногда под микрокодом понимают RISC-команды, полученные из CISC-команд в MISC-архитектуре. Несмотря на то, что первые RISC-процессоры не использовали микрокод, в современных экземплярах он всё же применяется.

## 1.3 Компиляторы

Компилятор – набор программ и утилит, преобразующий код на языке программирования в набор машинных команд. Процесс преобразования кода на ассемблере в машинный код выглядит следующим образом:

* препроцессинг исходного кода программы (удаление комментариев, подстановка макросов, подключение внешних модулей);
* замена кода на языке ассемблера на машинный код и сохранение полученных объектных файлов;
* связь (линковка) объектных файлов;
* преобразование связанных объектных файлов в исполняемый файл – добавление заголовков и другой мета-информации в зависимости от операционной системы.

В случае с языками программирования высокого уровня препроцессинг производится над исходным программным кодом, после чего происходит процесс трансляции – преобразование кода на языке программирования высокого уровня в код на языке ассемблера. Дальнейшие шаги остаются без изменений. Стоит понимать, что компиляторы должны на выходе выдавать код в наборе команд, поддерживаемых процессором, на котором этот код будет исполняться. Также компиляторы должны генерировать исполняемый файл в соответствии с требованиями, предъявляемыми той операционной системой, на которой этот файл будет запускаться. В противном случае операционная система не сможет распознать файл как исполняемый даже в том случае, если набор машинных команд будет поддерживаться процессором. С учётом вышесказанного очевидна необходимость правильного выбора компилятора для конечной платформы.

Современные компиляторы способны выдавать высоко оптимизированный, эффективный код. В состав компиляторов входят утилиты для анализа кода, способные найти и удалить неиспользуемые фрагменты кода, выполнять арифметические операции во время компиляции и многое другое. Компиляторы для архитектуры VLIW способны определить независимые наборы команд и объединить их в одновременно выполняемые инструкции.

В языке программирования Java существует концепция виртуальной Java-машины – технологии, позволяющей писать платформо-независимые программы. На каждого устройство, на котором необходимо запускать java-приложения, устанавливается комплект запуска Java. В состав этого комплекта входит виртуальная машина. Любая программа, написанная на языке Java, компилируется не в машинный, платформозависимый код, а в специальный байт-код. Этот код подаётся Java-машине, которая в режиме исполнения кода преобразует поступающие команды на байт-коде в соответствующие команды, совместимые с процессором, на котором запущена Java-машина. Таким образом, Java-машина занимается виртуализацией аппаратных вычислительных ресурсов.

Также существует компилятор LLVM – Low Level Virtual Machine – набор утилит, реализующий виртуальную машину, совместимую с набором RISC-команд. В состав этого компилятора входят утилиты для анализа кода, его преобразования и оптимизации. Может применяться для компиляции кода в машинный, для интерпретации кода и для его JIT-компиляции (Just In Time – компиляция времени выполнения). Также, как и в случае языка Java, LLVM компилирует программы не в машинный код, а в байт-код. Байт-код, в зависимости от необходимости, либо преобразуется в машинный код, либо используется виртуальной машиной. В отличии от виртуальной машины Java, LLVM поддерживает множество языков, включая интерпретируемые языки программирования.

## 1.4 Троичная система счисления

Современная цифровая электроника и информатика базируются на двух «столпах»: двоичной системе счисления и булевой алгебре. Существуют, однако, и более экзотические варианты цифровых устройств, в основе которых лежат троичная система счисления и трёхзначная логика. Рассмотрим различные их варианты прежде чем перейдём к обзору существующих троичных устройств.

### 1.4.1 Трёхзначная логика Лукасевича

Трёхзначная логика, созданная Я. Лукасевичем в 1920 г., отличается от привычной нам математической логики, т.к. в ней высказывания могут быть не только истинными и ложными, но и неопределёнными. Множество значений трёхзначной логики Лукасевича определяется множеством , иногда также используют множество . Второе, однако, может вводить в заблуждение, т.к. в таком случае за значение «Истина» принимают элемент 2, а элемент 1 используют для значения неопределённости. В дальнейшем будем использовать первое множество. Результаты операций импликации и отрицания для элементов множества сохраняют те же значения, что и в стандартной математической логике, а в остальных случаях доопределяются следующим образом:

Аналогично стандартной математической логике, операции дизъюнкции, конъюнкции и эквивалентности выводятся из операций импликации и отрицания следующим образом:

Таким образом, подставив (1), (2), (3) в (4), (5), (6) получаем таблицы истинности для основных логических операций - отрицания, импликации, дизъюнкции, конъюнкции и эквивалентности – отражённые в таблицах 2-6.

Таблица 2 – таблица истинности отрицания в трёхзначной логике Лукасевича

|  |  |
| --- | --- |
| a |  |
| 0 | 1 |
|  |  |
| 1 | 0 |

Таблица 3 – таблица истинности импликации в трёхзначной логике Лукасевича

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 |  | 1 |
| 0 | 1 | 1 | 1 |
|  |  | 1 | 1 |
| 1 | 0 |  | 1 |

Таблица 4 – таблица истинности дизъюнкции в трёхзначной логике Лукасевича

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 |  | 1 |
| 0 | 0 |  | 1 |
|  |  |  | 1 |
| 1 | 1 |  | 1 |

Таблица 5 – таблица истинности конъюнкции в трёхзначной логике Лукасевича

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 |  | 1 |
| 0 | 0 | 0 | 0 |
|  | 0 |  |  |
| 1 | 0 |  | 1 |

Таблица 6 – таблица истинности эквивалентности в трёхзначной логике Лукасевича

|  |  |  |  |
| --- | --- | --- | --- |
|  | 0 |  | 1 |
| 0 | 1 |  | 0 |
|  |  | 1 |  |
| 1 | 0 |  | 1 |

Из изложенного выше очевидно, что значения функций дизъюнкции, конъюнкции и эквивалентности выводятся из значений функций отрицания и импликации. Последние, в свою очередь, выведены без какого-либо обоснования, т.е. заменой значений (таблиц истинности) функций отрицания и импликации можно получить другую трёхзначную логику.

### 1.4.2 Счётная трёхзначная логика

В стандартной математической логике определена операция сложения по модулю два, также часто называемая исключающим или ⊕. Особенностью этой операции является тот факт, что результат a ⊕ b = c совпадает с результатом выражения a + b в одноразрядной двоичной арифметике. В свою очередь операция конъюнкции a ∧ b совпадает с результатом выражения a × b в одноразрядной двоичной арифметике. Таким образом, при помощи указанных логических операций возможно построение простейшего вычислительного блока, позволяющего производить операции умножения и сложения над одноразрядными двоичными числами. В стандартной логике также справедливы следующие выражения:

Если в некоторой логике заданы операции исключающего или и конъюнкции, то из выражений (7) и (8) можно вывести операции дизъюнкции и отрицания. Сохраним определение этих операций для некоторой трёхзначной логики, построенной на множестве . Заданное множество часто обозначают как . Рассмотрим некоторое число X, записанное в такой троичной системе счисления, называемой симметричной или уравновешенной:

При такой записи число k определяет количество значащих символов (k + 1) при записи числа X в троичной позиционной системе счисления, т.к. |ak| = 1.Очевидно, что при k = 0 знак числа X определяется знаком числа a0. Рассмотрим, определяется ли знак числа X числом ak для k > 0. Для этого сравним слагаемые и :

Так как 3k > 0, разделим на него левую и правую часть выражения (10):

Для того, чтобы ak определял знак числа X необходимо, чтобы для него было верно следующее неравенство:

Значение выражения слева = 1, значение выражения справа достигает своего максимума в том случае, когда ai = 1, i = 0…k-1:

Нетрудно заметить, что выражение (13) – сумма первых k-1 элементов убывающей геометрической прогрессии с основанием b1 = 3-1 и знаменателем q = 3-1. Таким образом, сумма всех элементов убывающей геометрической прогрессии Sn:

Т.к. S1 = 3-1, Sn = 2-1, 1 ≤ k < ∞,

Таким образом, правая часть выражения (12) всегда меньше левой. Аналогичным способом доказывается достаточность. Из чего следует, что знак цифры старшего разряда определяет знак всего числа.

Рассмотрим сложение и умножение двух одноразрядных чисел в симметричной троичной системе счисления. Результат сложения и умножения чисел в троичной системе счисления должен совпадать с результатом в десятичной системе счисления. В таблице 7 приведены результаты сложения и умножения двух одноразрядных чисел в симметричной троичной системе счисления: в первом столбце записан первый операнд, во втором – второй, в третьем – результат сложения в симметричной троичной системе счисления, в четвёртом – в десятичной системе счисления, в пятом – результат умножения в симметричной троичной системе счисления, в шестом – в десятичной системе счисления.

Таблица 7 – результаты сложения и умножения двух одноразрядных чисел в симметричной троичной системе счисления

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| a | b | +3 | +10 | ×3 | ×10 |
| -1 | -1 | -11 | -2 | 1 | 1 |
| -1 | 0 | -1 | -1 | 0 | 0 |
| -1 | 1 | 0 | 0 | -1 | -1 |
| 0 | -1 | -1 | -1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 |
| 1 | -1 | 0 | 0 | -1 | -1 |
| 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1-1 | 2 | 1 | 1 |

По аналогии с двоичной логикой, введём операцию сложения по модулю три ⊕, так, чтобы она соответствовала одноразрядному сложению в троичной симметричной системе счисления, и операцию конъюнкции так, чтобы она соответствовала одноразрядному умножению в троичной симметричной логике. Операции дизъюнкции и отрицания определим, опираясь на формулы (7) и (8). В таблице 8 представлены таблицы истинности основных бинарных операции получившейся троичной логики: конъюнкции, дизъюнкции, сложение по модулю три, таблица истинности унарной операции отрицания представлена в таблице 9.

Таблица 8 – таблица истинности основных бинарных операций в данной троичной логике

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *a* | *b* |  |  |  |
| -1 | -1 | 1 | 1 | -1 |
| -1 | 0 | -1 | 0 | -1 |
| -1 | 1 | 0 | -1 | -1 |
| 0 | -1 | -1 | 0 | -1 |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | -1 | 0 | -1 | -1 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | -1 | 1 | 0 |

Таблица 9 – таблица истинности унарной операции отрицания в данной троичной логике

|  |  |
| --- | --- |
| *x* |  |
| -1 | 0 |
| 0 | 1 |
| 1 | -1 |

Также введём операцию импликации, сохранив имеющуюся в двоичной логике связь между этой операцией, а также уже определёнными в троичной логике операциями дизъюнкции и отрицания:

Таблица истинности получившейся импликации представлена в таблице 9.

Таблица 9 – таблица истинности импликации в получившейся троичной логике

|  |  |  |  |
| --- | --- | --- | --- |
| → | -1 | 0 | 1 |
| -1 | -1 | 0 | 1 |
| 0 | -1 | 1 | 0 |
| 1 | -1 | -1 | -1 |

## 1.5 Функционал приложения и сфера применения

Предполагается, что приложение должно облегчить освоение машино-зависимых языков программирования, помочь начинающим в понимании влияния команд на состояние процессора и результата выполнения очередной команды от текущего состояния процессора. Также приложение должно помочь тем разработчикам, у которых есть опыт низкоуровневой разработки для других процессоров/микроконтроллеров, в аспектах, связанных со спецификой нового для них процессора/микроконтроллера. Визуализация ресурсов процессора должна облегчить понимание процесса исполнения кода на выбранном устройстве. Полученная в результате исполнения кода статистика призвана помочь программисту в задаче равномерного распределения ресурсов процессора, нахождения узких мест с точки зрения используемых ресурсов.

Основываясь на вышеизложенных результатах исследований аналогов и сфере применения, можно сформулировать основные требования к программному продукту:

* возможность пошагового исполнения кода;
* возможность исполнения кода с автоматическим переключением между шагами с заданной задержкой;
* исполнение кода без задержек;
* отображение значений компонентов процессора на текущем шаге;
* считывание данных о процессоре из внешнего источника;
* сбор и отображение статистики использования ресурсов по окончании выполнения программы.

Целесообразно собирать статистику по следующим показателям:

* вызываемые команды с количеством их вызова;
* компоненты процессора, к которым было обращение, с указанием количества обращений;
* наиболее используемые команды и компоненты процессора;
* часто встречающиеся последовательности команд.

Последнее представляет интерес, т.к. наиболее часто используемые последовательности в целях повышения производительности разработчики процессоров заменяют на одну команду. За счёт этого экономится время на декодирование команды, обращения к памяти. Также, если в коде программы последовательность команд встречается часто, программисту стоит убедиться в том, что у данного процессора уже не реализована команда, заменяющая подобную последовательность. В определённых случаях может оказаться целесообразной замена такой последовательности команд на подпрограмму.

Считывание данных о процессоре из внешнего источника нужно для того, чтобы пользователь имел возможность выбора или модификации процессора: в противном случае процессор будет являться частью программы и его изменения становятся невозможными без изменения исходного кода программы. Такой подход усложняет разработку программного продукта, но, вместе с тем, делает его более гибким.

Режим пошагового исполнения кода позволяет пользователю оценить изменения, вносимые каждой очередной командой. Режим пошагового исполнения с заданной задержкой удобен в том случае, если пользователь примерно представляет, какие изменения производит каждая команда, и ему необходимо оценить количественные параметры изменения (например, значения регистров). Наконец, режим выполнения кода без остановок позволяет быстро прогнать программу и получить статистическую сводку. Пользователь может переключаться из одного режима в другой во время выполнения программы: например, начать выполнение программы в пошаговой режиме, затем перейти в режим с задержкой, а после – без задержки исполнения.

## 1.6 Выводы

В ходе данного раздела была проанализирована предметная область: изучены существующие аналоги, современные процессор, их устройство и принципы построения. Был рассмотрен процесс преобразования исходного кода в машинный, а также существующие решения виртуализации. В конце раздела, основываясь на проведённые исследования, определена сфера применения программного продукта, выдвинуты основные требования.

# 2 Конструкторский раздел

В данном разделе рассмотрены вопросы конструирования программного продукта: рассмотрены потоки данных между компонентами программы, разработано представление процессора и его элементов, определено представление команд процессора и способ описания влияния команд на значения элементов процессора. Определён способ сбора статистики, а также разработан пользовательский интерфейс.

## 2.1 Разработка системы

Опираясь на требования, полученные в результате рассмотрения предметной области и сформулированные в предыдущем разделе, построена IDEF0-диаграмма, представленная на рисунке 8.

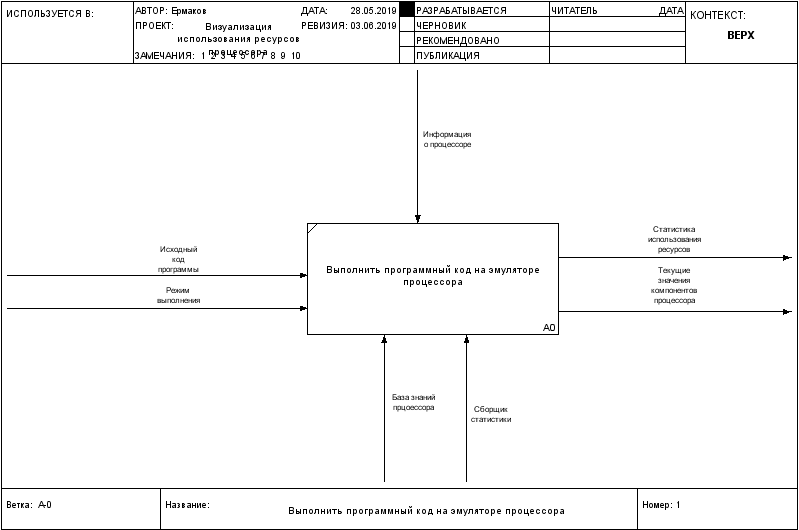


Рисунок 8. IDEF0-диаграмма процесса выполнения кода

Возможные сценарии взаимодействия пользователя с системой представлены в виде диаграммы прецедентов на рисунке 9.

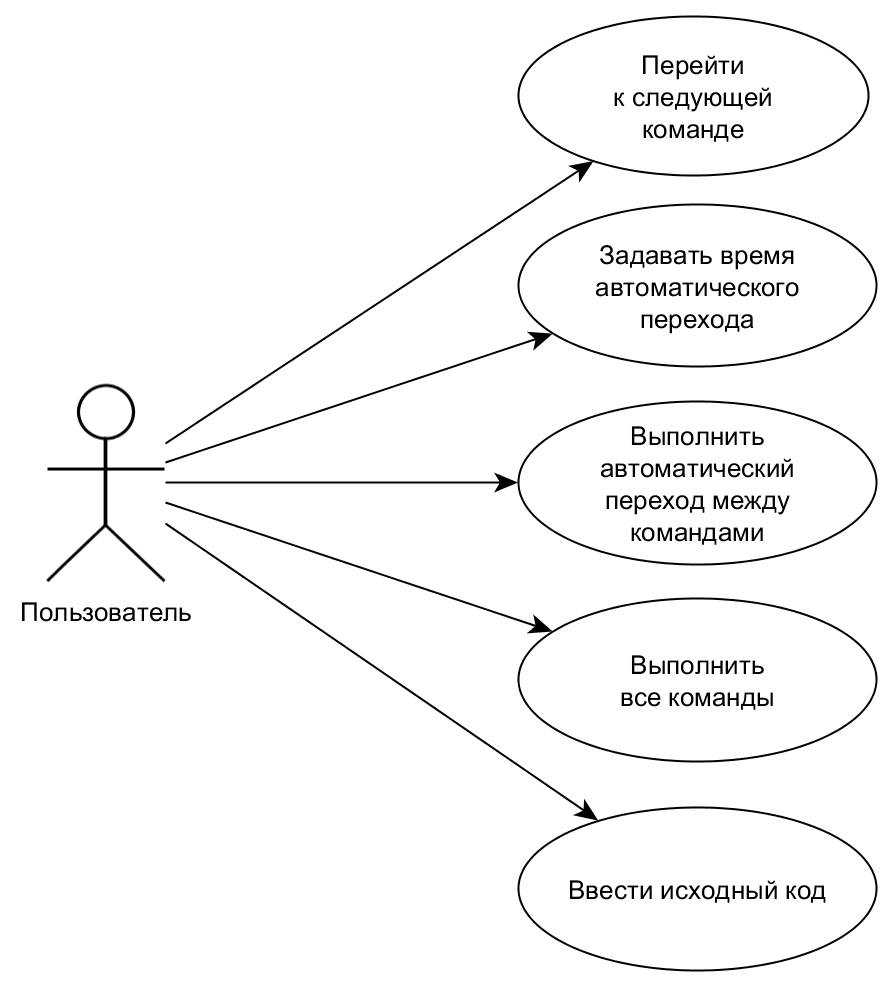


Рисунок 9. Диаграмме прецедентов взаимодействия пользователя и системы

Одним из ключевых требований, выдвинутых к системе, является считывание данных о процессоре из внешнего источника. Источник, из которого считываются данные, должен быть легко заменяем, при этом между источником данных и программой должно быть налажено взаимодействие. В качестве источника данных предлагается использование базы данных, данные в которой описывают процессор, а именно: его функциональные блоки, состояние которых необходимо оценивать во время симуляции (от точности описания блоков зависит информация, доступная во время симуляции: если процессор описан как набор регистров и команд, тогда во время симуляции можно получить значения, хранящиеся в регистрах; если при этом описаны АЛУ, а в командах указано обращение к этим АЛУ, то в процессе симуляции можно отследить обращения к этим блокам), а также набор команд, поддерживаемых данным процессором. Процессор оперирует набором команд, длина которых зависит от выбранной архитектуры и модели процессора. Для пользователя команды в таком виде не являются очевидными, в отличие от мнемонических команд, используемых в языке ассемблера. Так как системе безразлично, в каком виде представлена команда, предлагается в дальнейшем хранить мнемонические команды, вместо их машинных кодов.

В целях упрощения разработки, тестирования и отладки конечного продукта предлагается разбить его на три основных модуля:

* модуль данных;
* модуль визуализации;
* модуль синхронизации.

Рассмотрим каждый из этих модулей в отдельности.

### 2.1.1 Модуль данных

Модуль данных – единственная часть системы, напрямую взаимодействующая с источником данных о процессоре. Получая на вход очередную команду, модуль данных обращается к источнику, пытаясь определить наличие данной команды в списке поддерживаемых процессором. В случае если команда поддерживается процессором, модуль данных должен произвести над процессором действия, описанные в источнике данных. Предполагается, что результат выполнения команды может зависеть от того, какие команды были выполнены ранее. Иными словами, некоторые элементы процессора могут иметь связанные с ним значения. Очередная команда может изменять эти значения, в том числе опираясь на их текущую величину. Таким образом, множеством всех элементов и связанных с ними значениями в текущий момент времени можно описать состояние процессора. Очередная команда, изменяя значения элементов, переводит процессор из одного состояния в другое. Модуль данных хранит текущее состояние процессора и, при получении очередной команды при помощи источника данных определяет, в какое состояние переходит система. Переход системы из одного состояния детерминирован, т.е. зависит только от состояния системы в текущий момент времени и пришедшей команды.

Обозначим текущее состояние *Si*через кортеж значений элементов *vj*, где *vj* – значение j-го элемента:

где *Si* – i-е состояние системы;

– множество всех возможных значений j-го элемента;

*–* множество всех возможных состояний, определяемое декартовым произведением множеств возможных значений всех элементов процессора.

Команды, поддерживаемые процессором, образуют множество C. Таким образом, можно утверждать, что *C* – алфавит, а - буква алфавита. Процессор имеет некоторое начальное состояние *q0*, множество конечных состояний *F*, а также функцию переходов *δ*, описываемую источником данных о процессоре как изменение значений элементов при выполнении команды. Следовательно, имеем детерминированный конечный автомат вида:

где *C* – входной алфавит;

– множество внутренних состояний;

*q0* ∈– начальное состояние;

– множество конечных состояний;

-функция переходов;

В результате, модуль данных является программной реализацией конечного автомат: входной алфавит определяется множеством поддерживаемых процессором команд, множество внутренних состояний определяется элементами процессора и их возможными значениями, начальное состояние определяется пользователем, который задаёт начальные значения элементов процессора, функция перехода задаётся изменением значений элементов процессора при выполнении команды. Перечисленные выше параметры считываются из источника данных, кроме начального состояния, которое передаётся модулю при первом обращении вместе с командой. Текст входной программы является словом в языке, построенном на заданном алфавите *C*.

ER-диаграмма моделей данных представлена на рисунке 10.



Рисунок 10. ER-диаграмма моделей данных в нотации Чена

Стоит понимать, что атрибут «Новое значение» модели «Действие» должен описывать, каким образом действие изменяет значение некоторого элемента. Для этих целей можно реализовать набор простых команд (скопировать, сложить и т.д.), образующих базис, на основе которого можно описать любое действие. Совокупность таких базисных команд и элементов процессора, на которые это действие влияет, образуют описание команды процессора.

Для сбора статистики удобно после перехода в новое состояние возвращать список изменённых элементов процессора с указанием их новых значений. Таким образом, на вход модуль данных получает:

* очередную команду;
* исходное состояние системы (при первом обращении).

По окончании обработки команды модуль возвращает:

* флаг успеха (переданная команда поддерживается и валидна);
* список изменённых элементов и их новых значений;

Алгоритм работы модуля данных приведён на рисунке 11.

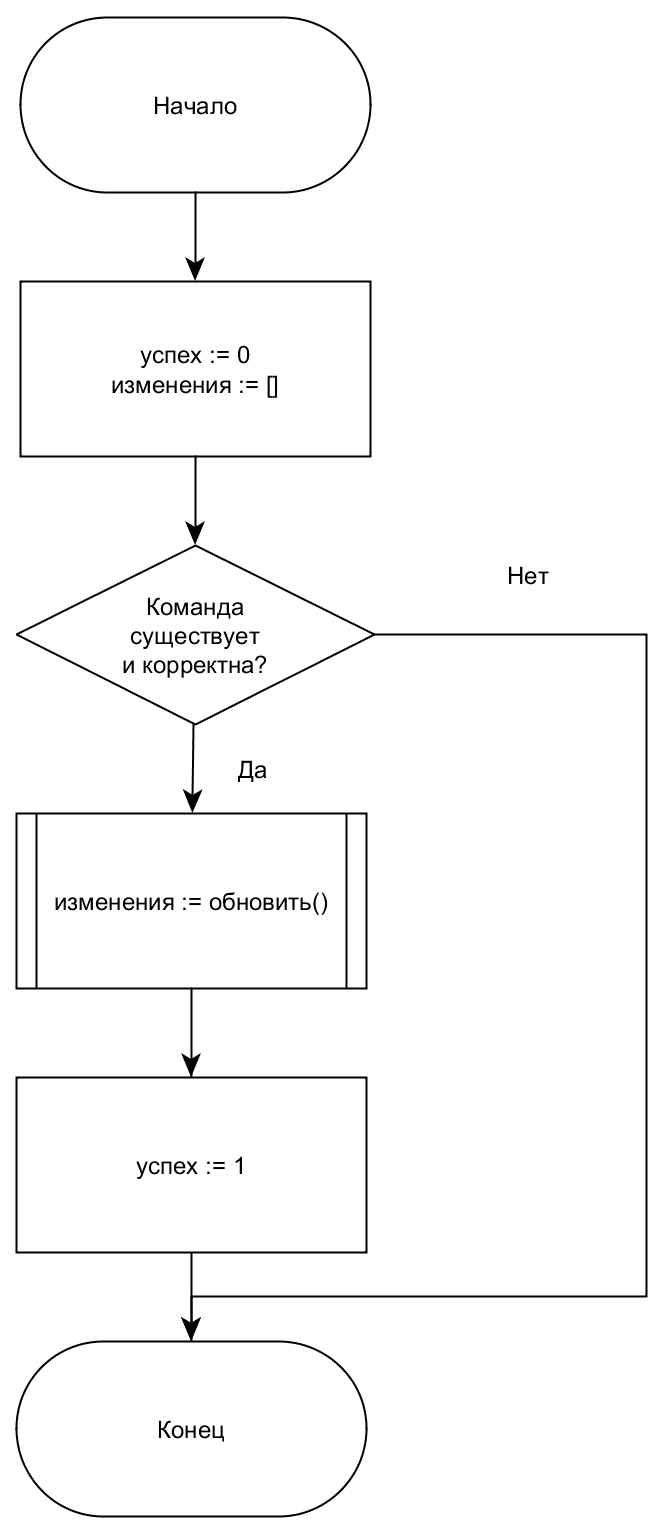


Рисунок 11. Алгоритм работы модуля данных

Алгоритм работы функции обновления состояния изображён на рисунке 12.

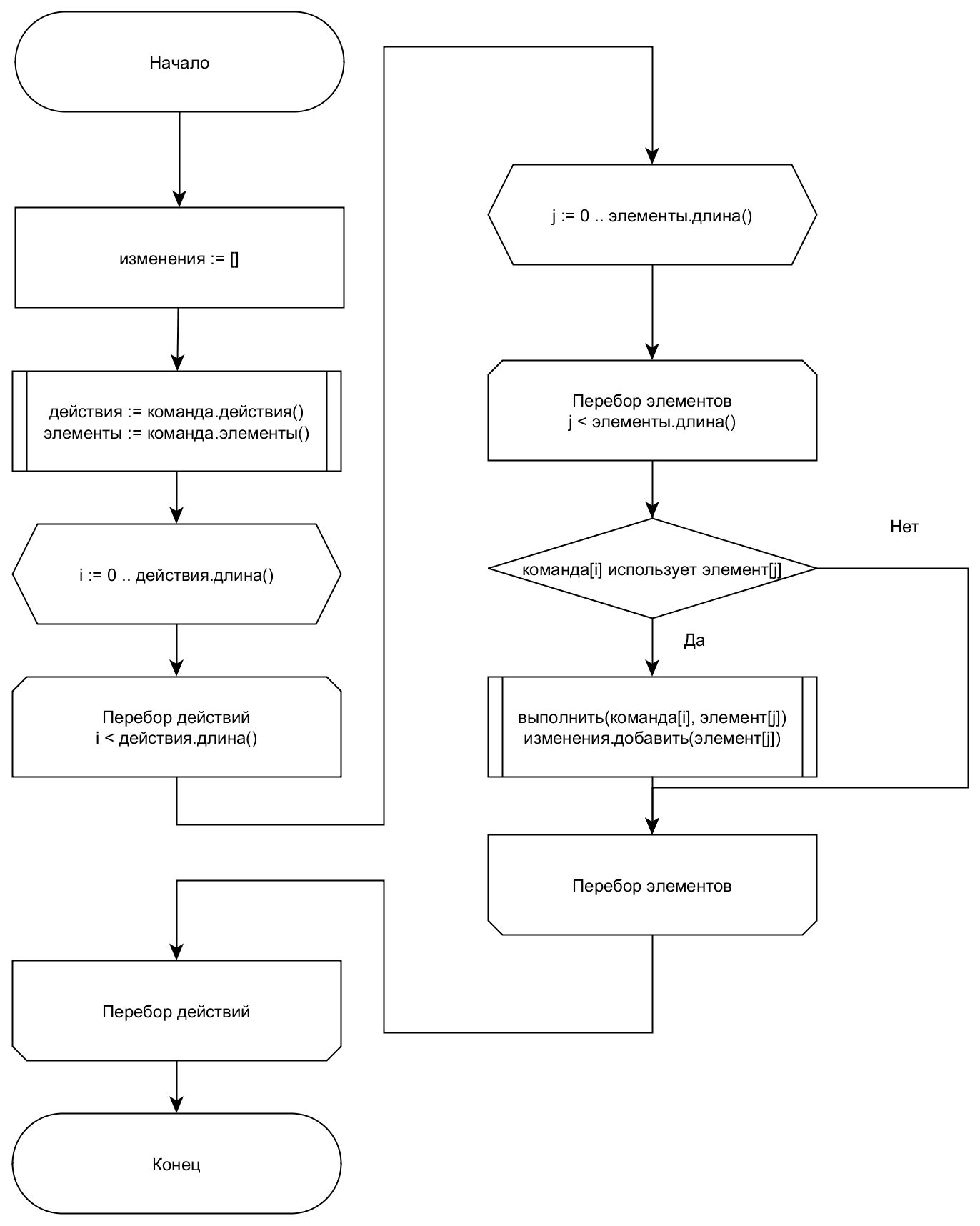


Рисунок 12. Схема алгоритма функции обновления состояния

### 2.1.2 Модуль визуализации

Модуль визуализации предназначен для взаимодействия пользователя с программным продуктом. Модуль отображает текущее состояние процессора, а также предоставляет пользователю возможность повлиять на ход симуляции: задать начальные значения, выбрать режим симуляции. По завершении симуляции модуль отображает полученную в ходе выполнения статистику. Также пользователь задаёт набор команд (входную программу)

При получении данных об изменении модуль обновляет отображаемые значения компонентов. При получении отчёта о статистики использования, модуль отображает предоставленный отчёт. Также модуль может отправлять команды для перехода к следующей команде, перехода к следующей команде с заданной задержкой, а также команду для полного выполнения программы. Набросок интерфейса основного окна показан на рисунке 13.

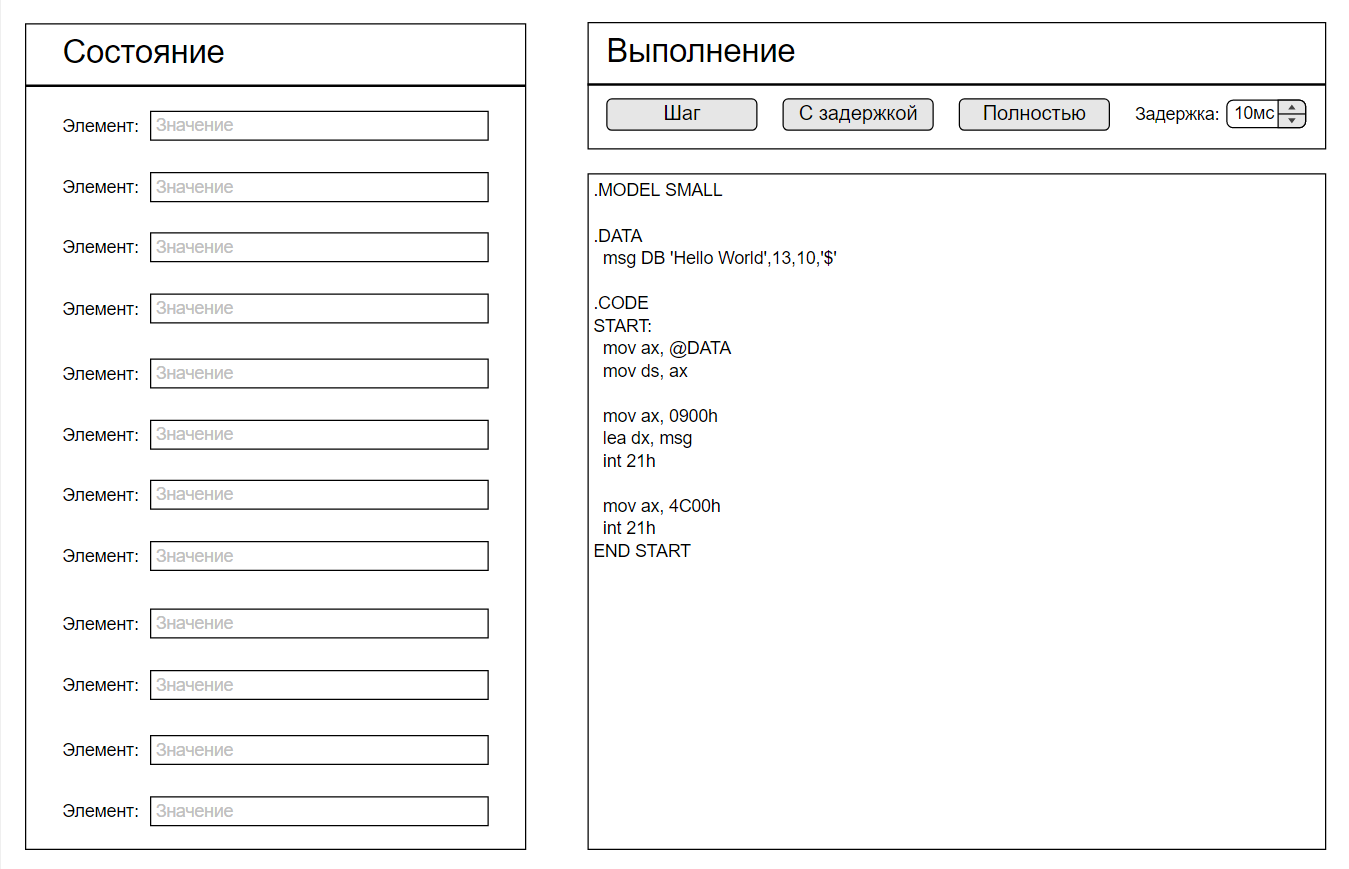


Рисунок 13. Набросок интерфейса основного окна программы

Набросок интерфейса окна отчёта приведён на рисунке 14.

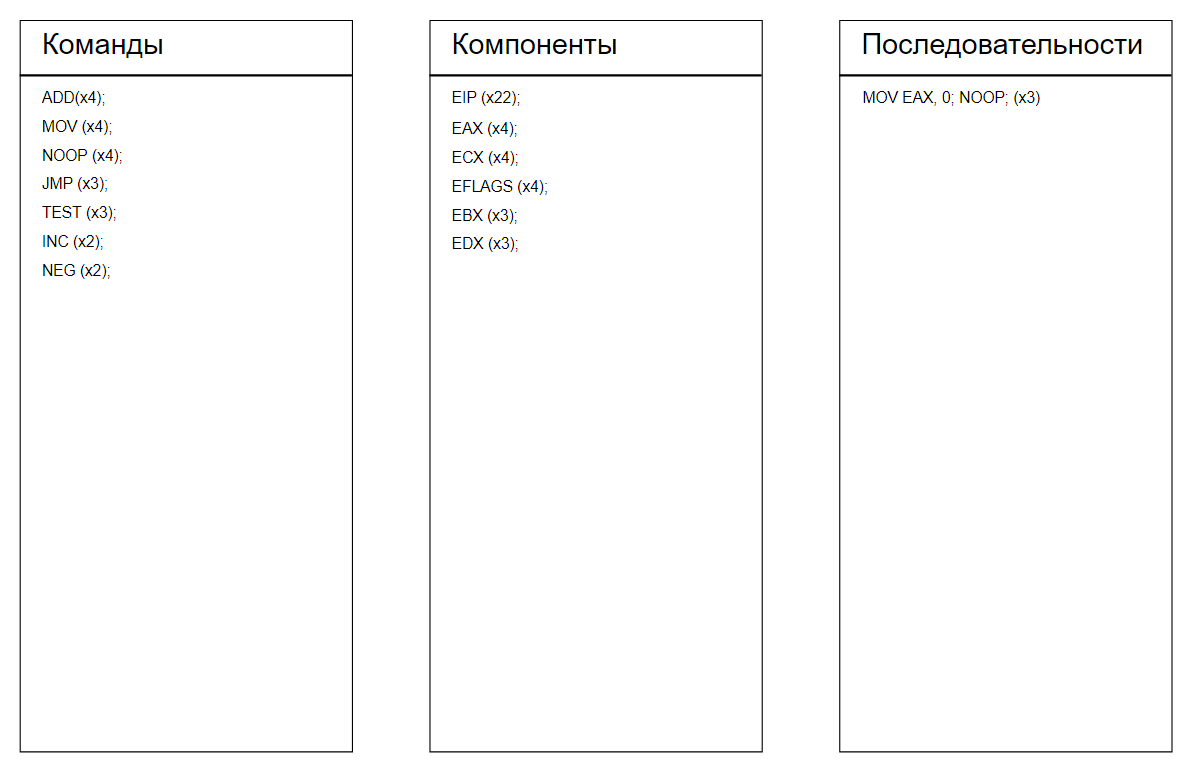


Рисунок 14. Набросок интерфейса окна отчёта

### 2.1.3 Модуль синхронизации

Подобно тому, как тактовый генератор синхронизирует работу всех элементов процессора, модуль синхронизации отвечает за слаженную работу всех модулей системы. Считывая очередную команду к выполнению, модуль синхронизации делает запрос к модулю данных для изменения его состояния. В случае если ответ модуля данных содержит флаг успеха, модуль синхронизации обновляет статистические данные: обновляет счётчики использования элементов процессора (информация об использовании возвращается из модуля данных), обновляет счётчик использованных команд, добавляет текущую команду в очередь выполненных команд. Данные об изменении состояния передаются модулю визуализации. В случае если очередь команд пуста, модуль генерирует отчёт о выполнении, в который включает собранную статистику использования, а также анализирует очередь выполненных команд на наличие часто встречающихся последовательностей команд. Сформированный отчёт также отправляется модулю визуализации. Модуль синхронизации может работать в одном из трёх режимов, определяемых пользователем: режиме пошагового выполнения, режиме автоматического пошагового выполнения с задержкой, а также режиме полного выполнения программы. Схема алгоритма работы модуля синхронизации представлена на рисунке 15.

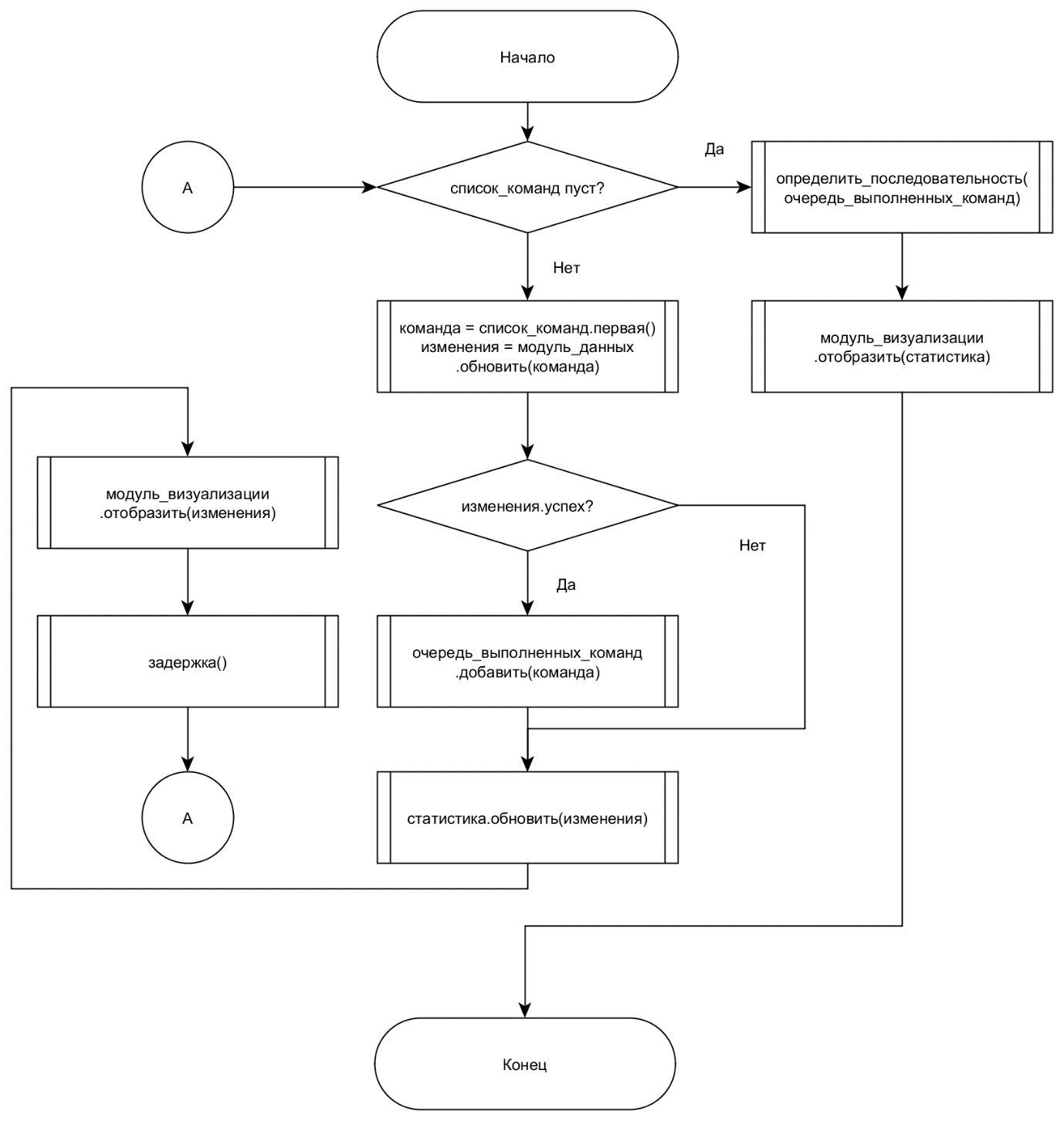


Рисунок 15. Схема алгоритма работы модуля синхронизации

Диаграмма претендентов взаимодействия модулей представлена на рисунке 16.

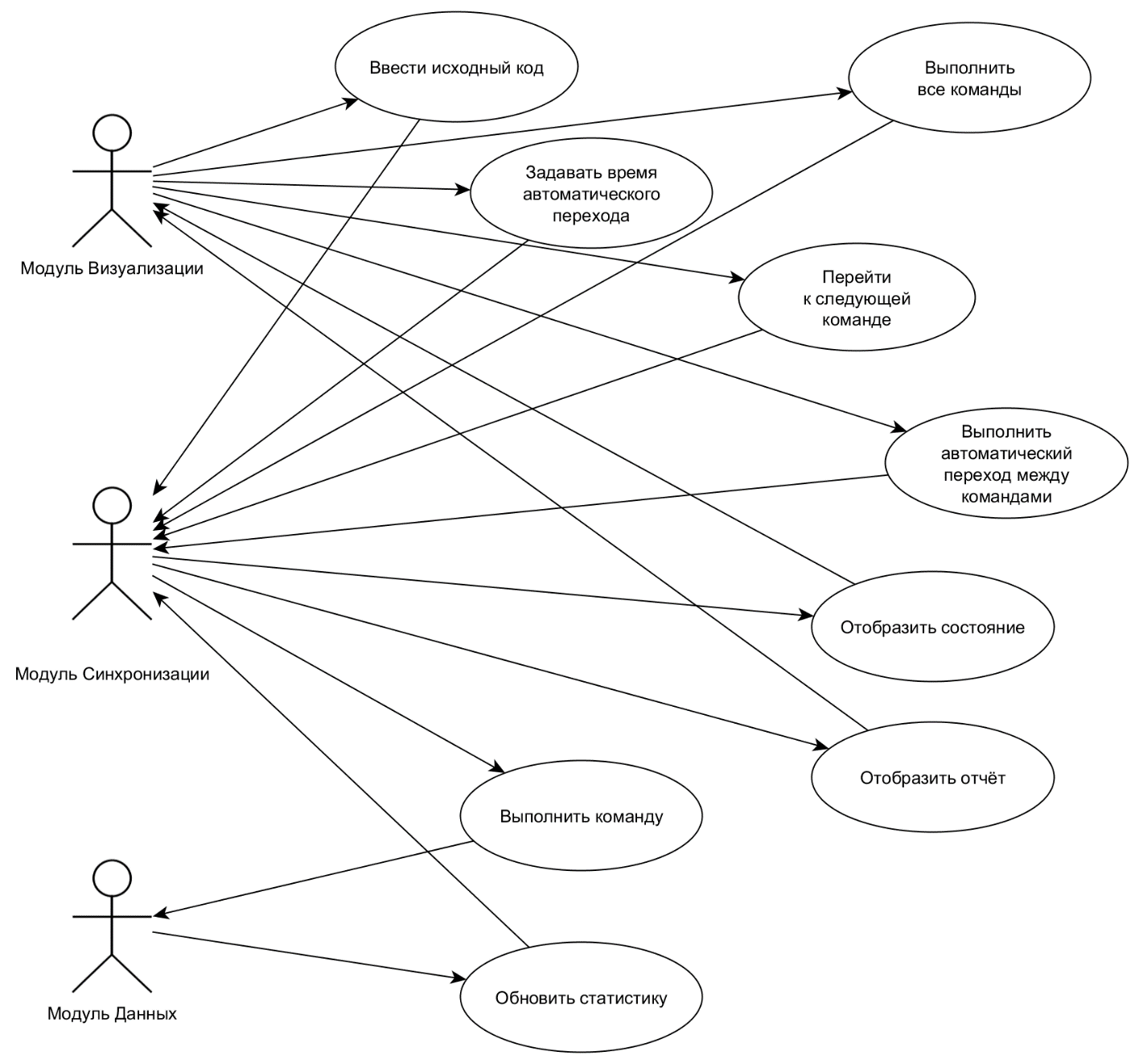


Рисунок 16. Диаграмма претендентов взаимодействия модулей

## 2.2 Рассмотрение существующих алгоритмов

Ниже рассмотрен ряд алгоритмов и механизмов, из которых выбраны наиболее подходящие для реализации в данной системе. Рассмотрены алгоритмы и механизмы:

* протяжки модельного времени;
* нахождения максимальной повторяющейся подпоследовательности последовательности;
* разработка интерфейса пользователя;
* передачи данных между модулями.

На основе проведённого анализа определены наиболее подходящие для решения поставленной задачи алгоритмы и механизмы разработки.

### 2.2.1 Протяжка модельного времени

В основном реализуются два принципа протяжки модельного времени:

* принцип Δt;
* событийный принцип;
* комбинированный метод.

Принцип Δt строится на анализе всех блоков системы в момент времени t + Δt в зависимости от их состояния в момент t. На основе анализа принимается решение об имитации событий в системе. Система обладает рядом очевидных недостатков:

* значительные затраты машинного времени на анализ состояния системы;
* при уменьшении Δt возрастают затраты на анализ;
* при увеличении Δt возрастает вероятностью пропуска событий.

В основе событийного принципа лежит тот факт, что состояние системы изменяется в дискретные моменты времени, соответствующие появлению событий в системе. Иными словами, переход системы из одного состояния в другое происходит при появлении в системе событий. Поэтому удобно анализировать состояние блоков системы лишь в моменты появления событий. Для определения времени поступления события создаётся список будущих событий, в который заносятся события системы и сортируются по времени появления.

Комбинированный метод заключается в объединении двух вышеперечисленных методов протяжки времени. В реальных системах часто события распределены группами, причём группы событий характеризуются появлением некоторого значимого события. Суть метода заключается в том, что используется событийная протяжка времени для событий, образующих группы событий, и использование метода, стремящегося к методу протяжки времени Δt для событий внутри групп. Комбинированный метод адаптируется к плотности событий, становясь более похожим на метод событийной протяжки модельного времени или на Δt метод. Это достигается путём использования иерархической структуры циклических списков будущих событий. На самом верхнем уровне находится список, содержащий события, которые влекут за собой появление группы событий. Каждая ячейка такого списка содержит ссылку на список более низкого уровня, каждый из которых содержит набор событий, входящий в группу. При этом происходит масштабирование модельного времени. Каждый элемент этого более низкого списка также может содержать ссылки на другие списки и так далее. В результате, задача сводится к определению ближайшего события самого верхнего уровня, после него необходимо определить ближайшее событие следующего уровня и так далее. При каждом перемещении вниз по иерархии списков модельное время масштабируется.

Для реализации данной системы предлагается использование метода Δt, так как события системы совпадают с приходом очередного тактового импульса и характеризуют собой смену текущего состояния системы. Шаг по времени, в данном случае, определяет частоту импульсов, никакие события в системе не могут быть пропущены, так же как и простоев в системе, вызванных анализом состояния системы и отсутствием в системе изменений, быть не может.

### 2.2.2 Нахождение максимальной повторяющейся подпоследовательности последовательности

Пусть дана некоторая последовательность {an}. Назовём последовательность {bk} подпоследовательностью {an}, если k ≤ n и ∃ индекс i < n – k : ai = b0 … ai+k-1 = bk-1,. Если для заданных {an} и {bk} индекс i может принимать разные значение, так что {bk} является подпоследовательностью {an}, то {bk} будем называть повторяющейся подпоследовательностью последовательности {an}. Длиной подпоследовательности {an} будем называть целое неотрицательное число |an|, равное количеству элементов последовательности {an}. Максимальной повторяющейся подпоследовательностью последовательности {an} будем называть такую подпоследовательность, модуль которой больше или равен модулю любой другой подпоследовательности {an}. Рассмотрим алгоритмы нахождения максимальной повторяющейся подстроки:

* наивный алгоритм;
* построение суффиксных деревьев.

Наивный алгоритм заключается в построении матрицы совпадений M, и отыскании по ней совпадений. Матрица M заполняется следующим образом:

Очевидно, что полученная матрица симметрична относительно главной диагонали, а сама диагональ заполнена единицами. Все подпоследовательности образуют диагонали из единиц. После построения матрицы остаётся лишь выбрать наиболее длинные серии из 1 по диагонали и соответствующие им подпоследовательности. Определение максимальной подпоследовательности последовательности имеет сложность O(n2).

### 2.2.3 Разработка интерфейса пользователя

При разработке пользовательского интерфейса необходимо учитывать не только внешнее расположение компонентов, очевидность пользовательского интерфейса и опыт пользователя (UI & UX), но также и процесс взаимодействия интерфейса с бизнес-логикой приложения и передачу данных между компонентами интерфейса, интерфейсом и блоками, реализующими бизнес-логику. Потребность в создании простых, гибких, тестируемых компонентах интерфейса привела к появлению ряда архитектурных паттернов:

* Model-View-Controller [14];
* Model-View-Presenter;
* Model-View-ViewModel;
* Flux [15].

Вкратце рассмотрим каждый из них. Введём определения понятий для первых трёх паттернов. Модель (model) – абстракция, предоставляющая непосредственный доступ к данным, а также предоставляющая возможность подписки на событие изменения этих данных. Модель может выступать в роли обёртки над данными, хранящимися в файле, в базе данных, удалённо в сети или в оперативной памяти. В некоторых случаях модель может не хранить данные, а генерировать их. Представление (view) – компонент интерфейса, отображающий переданные данные и запрашивающий изменение данных.

Суть паттерна MVP заключается в том, что между представлением и моделью вводится сущность презентер (presenter), которая отвечает за взаимодействие модели и представления. Презентер подписывается на события представления и модели, а также содержит ссылку на них. Модель и представление не знают ни о существовании друг друга, ни о существовании презентера. Любая передача данных управляется презентером, где и реализуется бизнес-логика приложения. Одна представление связывается с одним презентером. Взаимодействие модели, представления и презентера представлено на рисунке 17.

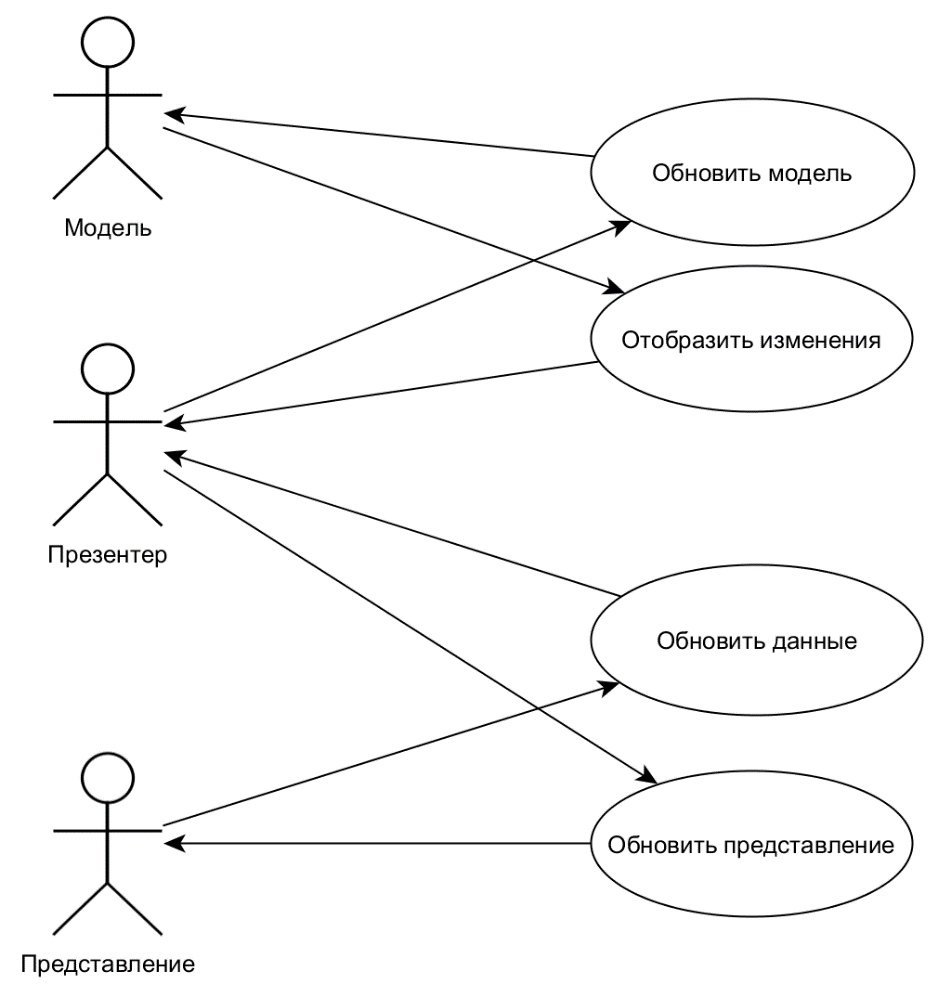


Рисунок 17. Диаграмма прецедентов взаимодействия модели, презентера и представления

Паттерн MVVM применяется при двухстороннем связывании данных и заключается в том, что вводится новая сущность ViewModel (модель представления). В случае если представление использует двухстороннее связывание данных, сущность модель представления подписывается на это связывание и является для представления моделью. При этом модель представления подписывается на события модели и является для неё представлением. Взаимодействие модели, модели представления и представления показано на рисунке 18.

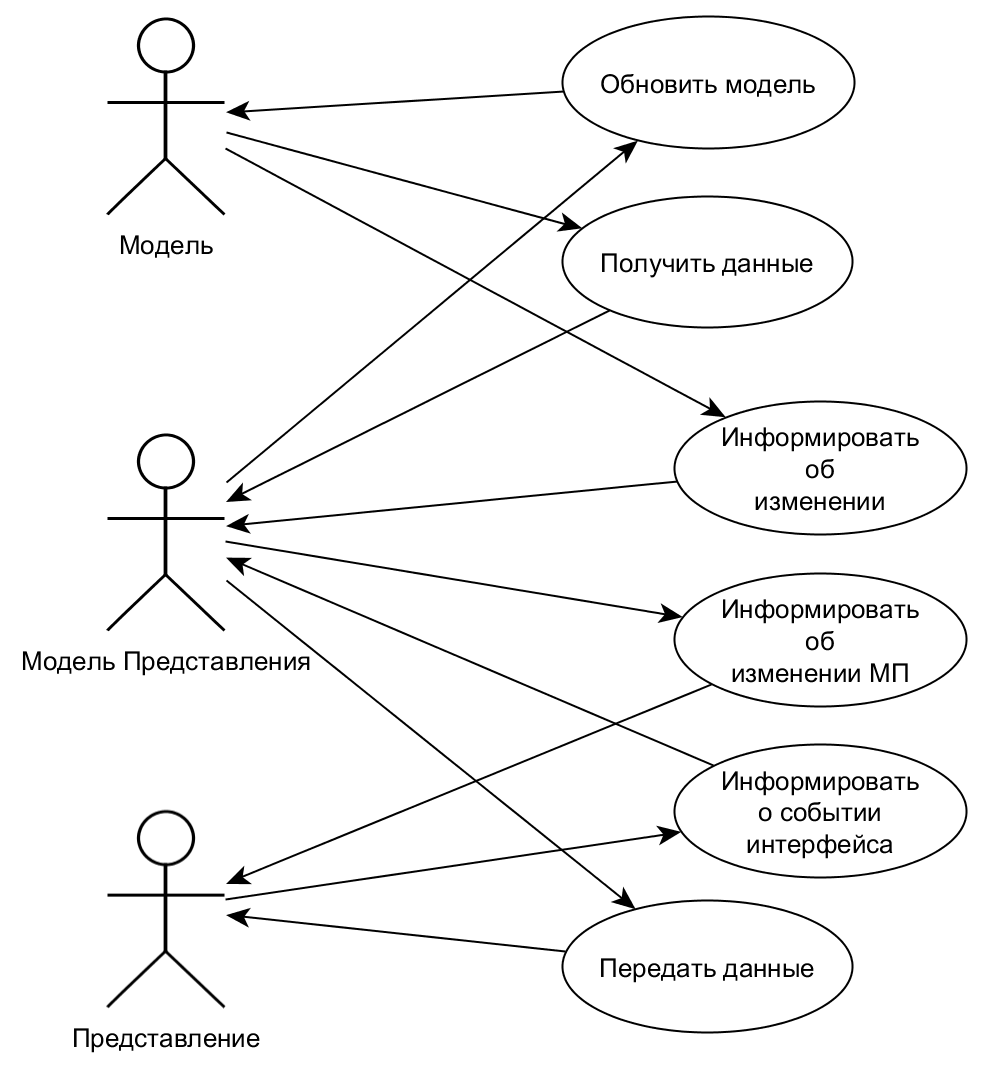


Рисунок 18. Диаграмма прецедентов взаимодействия модели, представления и модели представления

Паттерн MVC предполагает внедрение сущности контроллера (controller), отвечающее за одностороннее взаимодействие модели и представления. Суть паттерна заключается в том, что представление и контроллера зависят от модели, но модель независима от этих двух сущностей. Модель, как и ранее, предоставляет возможность подписки на события изменения данных и интерфейс изменения данных. Представление подписывается на изменения модели и использует методы контроллера. Контроллер, в свою очередь, содержит ссылку на модель и может изменять данные модели. Бизнес-логика приложения реализуется в контроллере. Взаимодействие модели, представления и контроллера представлено на рисунке 19.

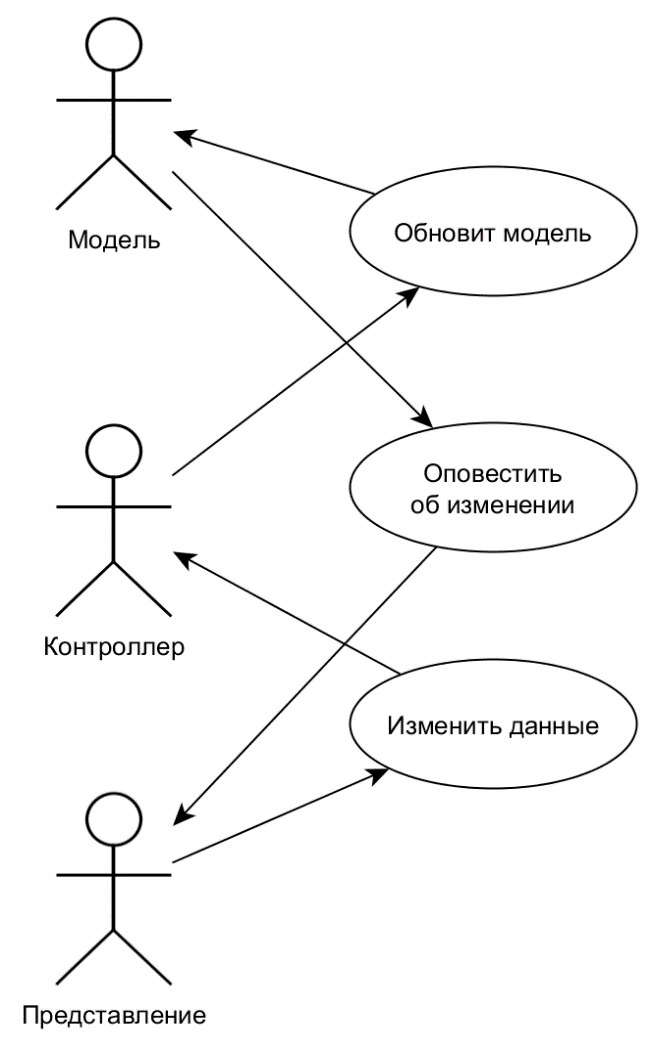


Рисунок 19. Диаграмма прецедентов взаимодействия модели, контроллера и представления

Архитектурный паттерн flux оперирует иным набором терминов, определение которых приведено ниже.

Хранилище (storage) или состояние (state) – глобальная неизменяемая древовидная структура данных, хранящая все данные, относящиеся к бизнес-логике приложения. Реализовано хранилище по принципу паттерна одиночка. При получении действия определяет, каким образом изменить данные хранилища. После изменения состояния генерирует соответствующее событие.

Действие (action) – событие, посылаемое диспетчеру.

Фабрика действий (action factory) – сущность, реализующая паттерн фабрика и отвечающая за создание действий в системе.

Диспетчер – сущность, которая определяет, какие хранилища реагирует на полученное действие.

«Умный» компонент – компонент, подписанный на изменения состояния и создающий новые действия.

«Глупый» компонент – компонент, отвечающий за отображение данных и регистрацию пользовательского ввода. Как правило, «умный» компонент соединён с одним или несколькими «глупыми» компонентами. Разделение на «глупые» и «умные» компоненты позволяет отделить логику приложения от отображения, что позволяет переиспользовать «глупые» компоненты. Взаимодействие компонентов представлено на рисунке 20.

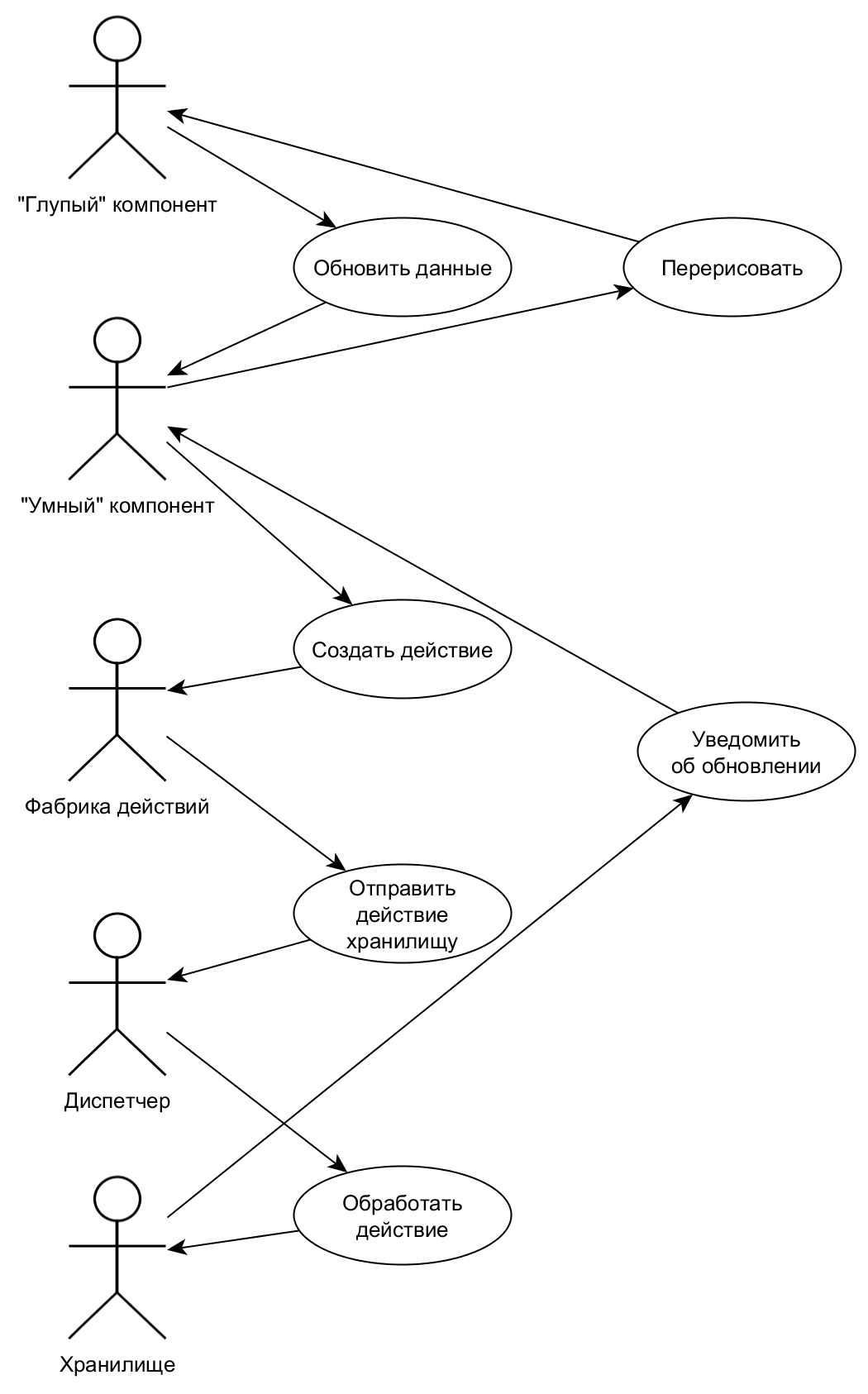


Рисунок 20. Диаграмма прецедентов взаимодействия элементов паттерна Flux

Для данной системы предлагается использовать архитектурный паттерн Flux, так как он позволяет уменьшить зависимость интерфейса от данных, позволяет простое добавление новых данных и обслуживающей их бизнес-логики, а также заточен для клиент-серверных приложений, что по сути схоже с получившейся модульной системой.

### 2.2.4 Передача данных между модулями

Более подробно механизмы передачи данных между модулями был рассмотрен в статье [16]. Здесь же приведён результат сравнения следующих механизмов передачи данных между модулями: при помощи аргументов командной строки, программных каналов, сигналов, файлов и сокетов. Проведено сравнение механизмов по ключевым критериям. Результат сравнительного анализа приведён в таблице 2.

Таблица 2 – сравнение механизмов передачи данных между процессами

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Механизм | Передача  при запуске | Передача  в процессе выполнения | Передача  на локальной машине | Передача  по сети |
| Аргументы командной строки | + | - | + | + |
| Каналы | + | + | + | - |
| Сигналы | + | + | + | - |
| Файлы | + | + | + | + |
| Сокеты | + | + | + | + |

Предлагается использовать сокеты для передачи данных, так как они позволяют передачу данных в любой момент времени на локальной машине и по сети, являются двухсторонним каналом связи.

## 2.3 Метод виртуализации

Описанный выше метод описания процессора позволяет не только использовать процессор для эмуляции и визуализации состояния, но также для виртуализации. Имея описания двух процессоров и набора команд, поддерживаемых первым процессором, можно попытаться определить, какие команды второго процессора позволяют получить тот же результат. Подобного результата можно добиться, выполнив на процессоре А поддерживаемый им набор команд, а затем перебрать команды процессора Б, сравнивая состояния обоих процессоров. Если в результате перебора состояния процессоров совпадают, значит, последовательность команд, преобразовавшая исходное состояние процессора Б в текущее, и является подбираемой. Дополнительно уточнить, какие компоненты процессоров необходимо сравнить, можно в таблице соответствий. Для более оптимизированного процесса перебора предлагается использовать метод ветвей и границ. Суть метода заключается в разбиении исходного множества X на подмножества меньшего размера. Допускается дальнейшее разбиение подмножеств, что увеличивает высоту дерева. Получившееся дерево называют деревом поиска, узлами которого являются получившиеся в результате разбиения подмножества. Для каждого подмножества определяется верхняя и нижняя граница, приемлемая для поставленной задачи. Затем происходит обход дерева в глубину, при котором сравнивается нижняя граница текущего узла с верхними границами предков. В случае, если нижняя граница узла больше, чем верхняя граница любого предка, текущий узел может быть исключен из рассмотрения. Таким образом, удаётся сократить множество перебираемых значений. IDEF0-диаграмма метода виртуализации представлена на рисунке 21.

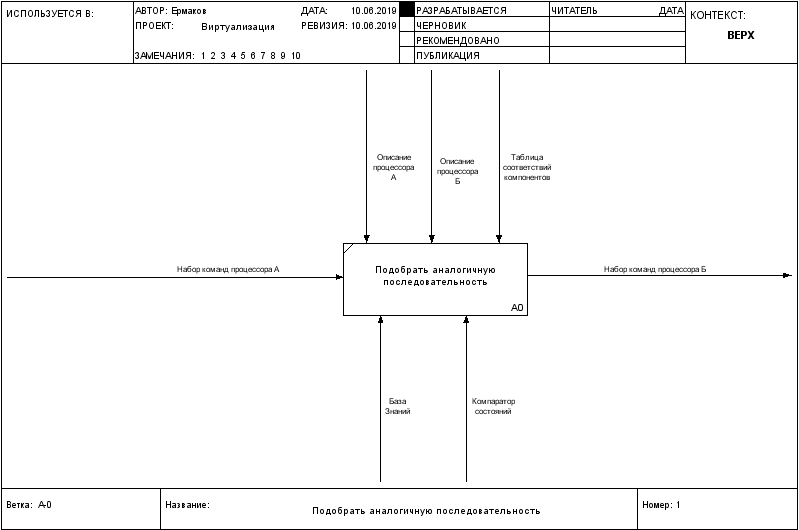


Рисунок 21. IDEF0-диаграмма процесса виртуализации

## 2.4 Выводы

В ходе данного раздела была описана разрабатываемая система, произведено разбиение на функциональные модули, описаны механизмы передачи данных между модулями. Представлен способ описания процессора. Разработан интерфейс пользователя. Рассмотрены существующие алгоритмы и выбраны наиболее подходящие для разработки данной системы. Рассмотрена возможность использования данного метода описания процессора для решения задачи виртуализации.

# 3 Технологический раздел

В данном разделе рассмотрены языки программирования, технологии и библиотеки, использованные для реализации описанного приложения. Приведено обоснование выбора, а также рассмотрены альтернативы. Наглядно продемонстрированы основные моменты реализации, включая фрагменты кода с пояснением и скриншоты интерфейса получившегося программного продукта.

## 3.1 Средства программной реализации

### 3.1.1 Модуль данных

Модуль данных имеет ключевое значение для всей системы, потому выбор технологий для его реализации имеет весомое значение. Для хранения данных процессора предполагалось использование базы данных. Основываясь на результатах, полученных в ходе аналитического и конструкторского разделов, становится очевидно, что структура данных представления процессора не является чёткой и может изменяться в зависимости от конкретной модели процессора. Использование реляционной базы данных в таком случае затруднительно, т.к. для описания таблиц реляционной базы данных необходимо иметь структуру представляемых данных. Применение документоориентированных или графовых баз данных позволяет хранить данные различной структуры, не задавая эту структуру на этапе конструирования базы данных. Помимо базы данных в таком случае необходимо написать менеджер, представляющий собой обёртку над базой данных и реализующий логику работы процессора: хранение текущего состояния, обработка поступившей команды, смена состояния, передача данных модулю синхронизации. Применение языка логического программирования Prolog [17] позволяет единообразно описать данные и действия с ним. База знаний Prolog позволяет хранить описание процессора в виде отношений, а также в этой базе можно хранить текущее состояние процессора. Используя приёмы логического программирования можно описать любые команды процессора, что избавляет от необходимости реализовывать собственный интерпретатор, применяемый при переходе из одного состояния процессора в другое при выполнении команды. Использование пролога освобождает от необходимости реализации метода ветвей и границ, т.к. он используется при подборе термов в процессе алгоритма резолюции. В качестве диалекта Prolog выбран SWI-Prolog, т.к. он поддерживает полиморфизм, обладает большим набором разработанных программных модулей, подробную документацию, 64-битный многопоточный интерпретатор, работающий на операционных системах Windows и Linux. Для передачи данных между модулями используется библиотека websocket [18], реализующая передачу данных через TCP и UDP сокеты. В упомянутой выше статье предлагалось передавать данные при помощи текстового формата JSON [19] (JavaScript Object Notation). Кодирование программных объектов строкой приводит к увеличению объёма хранимой информации. Для того, чтобы этого избежать, предлагается использовать двоичный формат BSON (Binary JSON). BSON является надмножеством JSON, позволяет пользователю хранить дополнительно двоичные данные, а также при передаче данных объект сериализуется в двоичный массив, а не текстовую строку, что позволяет сократить размеры и, следовательно, скорость передачи данных.

Базу знаний о процессоре можно условно разделить на две составляющие: служебную и описательную. Служебная часть состоит из термов, отвечающих за взаимодействие модуля данных и модуля синхронизации. Это термы передачи и приёма данных, обработчики команд модуля синхронизации, терм выполнения очередной команды. Предполагается, что эти термы не изменяются при описании процессора.

Описательная часть представляет собой термы, описывающие знания о процессоре: команды, компоненты, значения компонентов и их взаимосвязи. В качестве примера рассмотрим описание регистров. Пусть у 16-битного процессора есть набор 16-битных регистров, причём регистры общего назначения (рассмотрим для примера ax, bx), состоят из двух восьми битных половин, к которым также имеется доступ (ah, al, bh, bl). В листинге 1 приведён вариант описания таких компонентов.

|  |
| --- |
| /\* Описание объектов \*/  object(ah, register).  object(al, register).  object(ax, register).  object(bh, register).  object(bl, register).  object(bx, register).  /\* Описание связей между регистрами \*/  consists(ax, ah, al).  consists(ah, nil, nil).  consists(al, nil, nil).  consists(bx, bh, bl).  consists(bh, nil, nil).  consists(bl, nil, nil).  /\* Разрядность регистров \*/  capacity(nil, 0).  capacity(ah, 8).  capacity(al, 8).  capacity(bh, 8).  capacity(bl, 8).  capacity(R, C):- consists(R, XH, XL),  capacity(XH, XHC), capacity(XL, XLC), !,  C is XHC + XLC. |

Листинг 1. Описание регистров процессора

В первом блоке предикат object задаёт отношение описываемых объектов и терма register. В дальнейшем, это позволяет получить значения только тех объектов, которые является регистрами. Предикат consists задаёт отношение включения, указывая, из каких регистров состоят данные. Наконец, предикат capacity задаёт объём регистров. Отдельно стоит рассмотреть правило capacity(R, C). В отличии от перечисленных выше фактов, правило capacity(R, C) не хранит значение объёма регистра, а проверяет, является ли он составным и, если является, определяет объёмы регистров, составляющих данный. Исходя из объёмов регистров-компонентов, определяется объём составного регистра.

Для описания команд используется предикат command(), определяющий отношение команды и её аргументов. Так как команды должны выполнить некоторые действия, предикат command() состоит из набора правил, а не фактов. Пример описания команд представлен в листинге 2. Описаны команды mov(регистр-регистр), mov(регистр-число), inc, add(регистр-регистр), add(регистр-число).

|  |
| --- |
| command(mov, To, From):- number(From),setValue(To, From), !, incStat('mov(регистр-число)', command), incStat(To, register).  command(mov, To, From):- value(From, V), setValue(To, V), !, incStat('mov(регистр-регистр)', command), incStat(To, register), incStat(From, register).  command(mov, From):- command(mov, From, ax).  command(inc, Reg):- value(Reg, V), setValue(Reg, V + 1), !, incStat('inc', command), incStat(Reg, register).  command(add, To, From):- number(From), value(To, V), setValue(To, V + From), !, incStat('add(регистр-число)', command), incStat(To, register).  command(add, To, From):- value(From, V1), value(To, V2), setValue(To, V1 + V2), !, incStat('add(регистр-регистр)', command), incStat(To, register), incStat(From, register). |

Листинг 2. Пример описания команд процессора

Предикат value(From, V) получает значение, сопоставленное с атомом, связанным с From, и записывает его в атом V. Предикат setValue(Comp, Value) задаёт компоненту Comp значение Value. Так как значения могут изменяться в процессе выполнения команды, было необходимо изменять значение терма value(From, V). Для этого применялась процедура retract(term), удаляющая заданный терм, и процедура asserta(term), добавляющая указанный терм. Для использования указанных термов необходимо объявить терм value как динамический.

Предикат incStat(Object, Type) предназначен для учёта статистики использования. Использование предиката приводит к увеличению счётчика использования объекта Object типа Type. Это позволяет указать, какие компоненты процессора использует вызов команды и учесть это при сборе статистики.

### 3.1.2 Модуль синхронизации

Модуль синхронизации взаимодействует с модулями данных и визуализации, а также содержит в себе сборщик статистики. Модуль синхронизации отвечает за функционирование всей системы в целом. К языку программирования, используемого для реализации модуля синхронизации, выдвигалось требование простоты и распространённости: в случае, если приложение необходимо будет поддерживать или модифицировать другому человеку, усилия, затраченные на разработку, должны быть минимальными. Исходя из этого требования, пришлось отказаться от языка программирования C++. Популярные интерпретируемые языки программирования было решено не использовать, так как они медленнее, чем компилируемые, а также требуют дополнительного программного обеспечения (интерпретатора). В настоящее время высокой популярностью пользуются языки C# и Java [20], однако было принято решение отказаться от них в пользу языка GoLang. Причиной этого является растущая популярность языка, подробная документация, обширное сообщество разработчиков, простота при разработке многопоточных приложений. По заверению компании Google, занимавшейся разработкой языка GoLang [21], язык разрабатывался для решения реальных проблем, возникающих при разработке программного обеспечения в Google. Основные требования, предъявляемые к языку Go:

* ортогональность;
* простая грамматика – минимум ключевых слов, простой и читаемый код;
* простая работа с типами;
* отсутствие неявных преобразований;
* сборщик мусора;
* простые встроенные механизмы распараллеливания кода;
* разделение интерфейсов и реализации;
* эффективная система пакетов, обеспечивающая быструю сборку проекта.

В целом, синтаксис языка Go схож с синтаксисом языка Си, с дополнением скриптовых языков, таких как TypeScript [22].

Для реализации данного модуля также используются пакеты, реализующие работу с сокетами и преобразующие данные в формат BSON.

Одним из главных достоинств языка Golang является простота написания многопоточных приложений. Для запуска функции в отдельном потоке достаточно перед вызовом функции указать ключевое слово go. Это позволило с лёгкостью запустить сервер, обслуживающий веб-сокеты в параллельном потоке, а основной цикл оставить в главное потоке. Листинг 3 демонстрирует запуск двух параллельных потоков, выполняющих функции server.Setup() (настройка и запуск сервера) и server.mainLoop() (основной цикл приложения).

|  |
| --- |
| func main() {  go server.Setup()  server.MainLoop()  } |

Листинг 3. Запуск сервера и основного цикла в параллельных потоках

Помимо простоты запуска параллельных потоков, go также предоставляет удобные механизмы блокировки и передачи данных между потоками. Такими механизмами являются программные каналы. Программный канал – это особого вида переменная, предоставляющая операции блокирующей записи и чтения. Программный канал может быть рассчитан на 1 элемент, или на буфер элементов. Принцип действия каналов следующий: в некотором месте программы поток пытается считать данные из канала. Если в канал записаны данные, то поток считывает их и продолжает выполнение. Если канал пуст, поток блокируется до тех пор, пока другой поток не запишет данные в канал. Это позволяет не только передавать данные между потоками и организовывать монопольный доступ, но и синхронизировать потоки. Пример использования каналов представлен в листинге 4.

|  |
| --- |
| func MainLoop() {  initLoop()  for {  fmt.Println("Ожидание ввода")  go delay()  <- control  opt.pointer++  if opt.pointer >= len(opt.code) {  fmt.Println("Код выполнен")  opt.pointer = -1  go gatherStatistics()  continue  }  line := opt.code[opt.pointer]  sendCommand(command{  Cmd: "runCode",  Data: line,  }, )  opt.evaluated = append(opt.evaluated, line)  fmt.Println("Исполнение кода: ", line)  }  } |

Листинг 4. Пример использования канала для блокирования потока

В данном примере канал control блокирует исполнение кода в потоке до тех пор, пока в канал не будут записаны некоторые данные. Сами данные в этом случае не представляют ценности и игнорируются. Оператор <- chan считывает данные из канала, оператор chan <- записывает данные в канал. Запись данных в канал происходит в отдельной функции step(), вызываемой при нажатии на кнопку «Шаг» в интерфейсе, или при истечении таймера в случае выполнения с задержкой. Пример записи в канал представлен в листинге 5.

|  |
| --- |
| func step() {  control <- true  } |

Листинг 5. Запись данных в канал

Важно, что чтение данных из канала должно предшествовать записи. В противном случае, поток блокируется навсегда, т.к. запись в канал возможно только в том случае, если из него в данный момент ожидается чтение.

Помимо простых операций считывания и записи в канал язык Golang реализует оператор коммутации, позволяющий отслеживать состояния нескольких каналов и считывать из них данные по мере появления данных в канале. Пример использования коммутатора представлен в листинге 6.

|  |
| --- |
| for {  select {  case cmd := <- cmdChannel:  writeCommand(&cmd, c)  stat := state{}  readJSON(&stat, c)  for \_, elem := range stat.Objects {  log.Println(elem)  }  if cmd.Cmd == "getStats" {  statChannel <- stat  log.Println("Update Stats sent")  } else {  clientSocket.Emit("updateState", stat)  log.Println("Update State sent")  }  case f := <- finishChannel:  return  }  } |

Листинг 6. Пример коммутации нескольких каналов

В данном случае отслеживается состояние каналов cmdChannel и finishChannel и, в случае появления данных в любом из них, происходит выполнение соответствующей ветви кода.

### 3.1.3 Модуль визуализации

Модуль виртуализации представляет собой клиентское приложение, используемое для общения пользователя с системой. Поэтому модуль визуализации выполнен в качестве Web-интерфейса. В качестве frontend-сервера используется nginx. В обязанности nginx входит раздача статических файлов. Клиентское приложение реализовано на языке TypeScript – типизированном языке пользовательских сценариев, транслируемом в JavaScript. Язык TypeScript разработан компанией Microsoft и направлен на минимизацию ошибок времени исполнения, связанных с структурами и типами данных. Язык TypeScript, в отличии от языка JavaScript, обладает строгой типизацией, что позволяет отлавливать ошибки на этапе разработки.

Для разработки стилей используется расширение CSS SCSS [23]. Данное расширение позволяет писать стили с более наглядным синтаксисом, поддерживает константы и функции, позволяет генерировать стили по заданным шаблонам. SCSS-стили транслируются в CSS-стили, которые используются в проекте. Также для упрощения разработки используется css-фреймворк bootstrap [24], упрощающий вёрстку за счёт готового набора стилей.

На сегодняшний день значительной популярностью обладает фреймворк React, используемый для вёрстки и управления логикой приложения. React [25] поддерживает TypeScript, существуют специальные библиотеки для настройки маршрутизации. В свободном доступе находится библиотека Redux, реализующая паттерн Flux для React. Тем не менее, для разработки данного модуля был выбран набирающий популярность фреймворк Vue [26]. Также как и React, Vue поддерживает TypeScript, имеет библиотеки маршрутизации, реализует паттерн Flux в библиотеке Vuex [27]. Выбор в пользу Vue сделан по той причине, что Vue поддерживает двухстороннюю связку данных, что позволяет существенно упростить разработку интерфейса, сократить объём кода, связанного с получением значения компонентов интерфейса.

Для обмена данными при помощи сокетов использована библиотека socket.io [28], для работы с BSON – bson.

Для сборки приложения, трансляции TypeScript и SCSS, минимизации приложения и разбиения его на части используется утилита webpack [29]. Эта утилита позволяет собрать множество исходных файлов в один, выполняя при этом различные преобразования над исходными файлами.

Скриншот интерфейса программы в процессе выполнения программного кода представлен на рисунке 22.

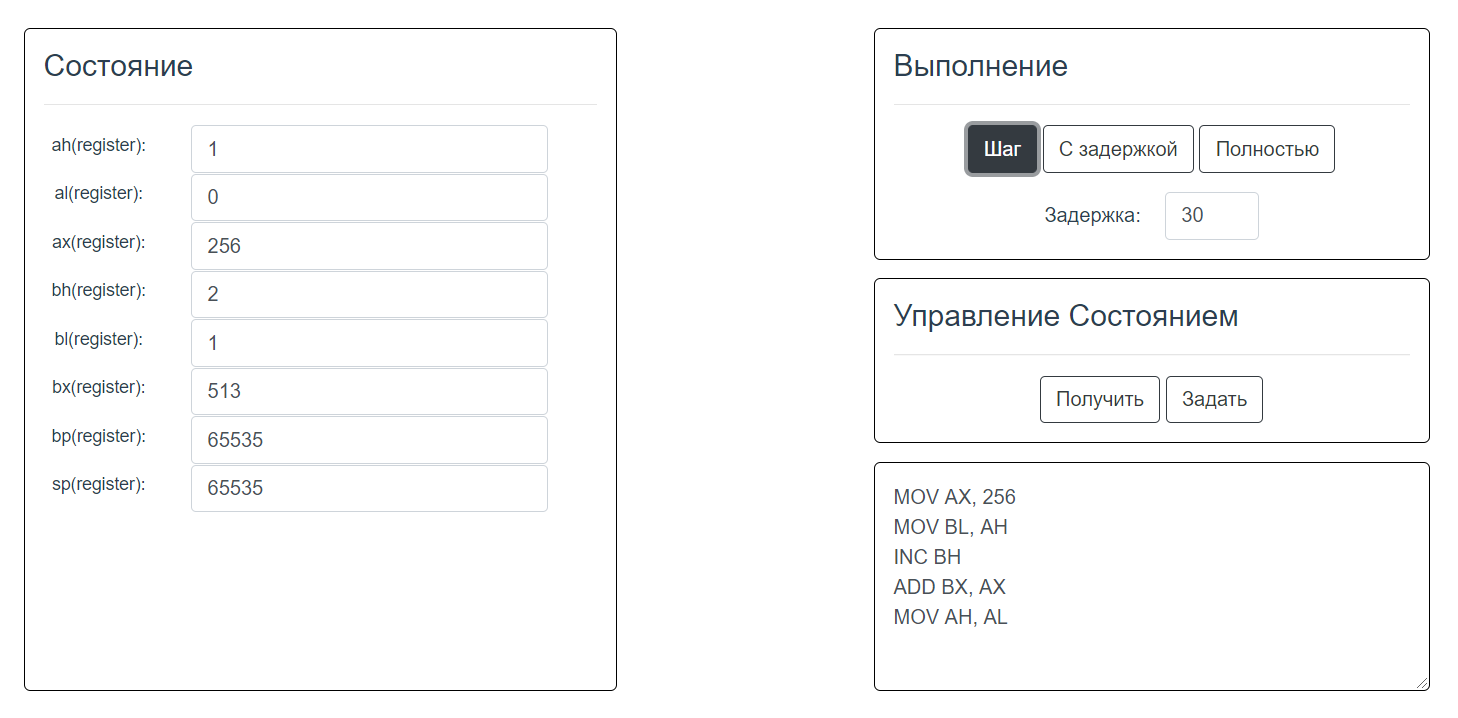


Рисунок 22. Интерфейс программы в процессе выполнения кода

Статистика, собираемая в процессе выполнения, отображается в всплывающем модальном окне по окончании выполнения кода. Рисунок 23 содержит возможный вариант отчёта по статистике.

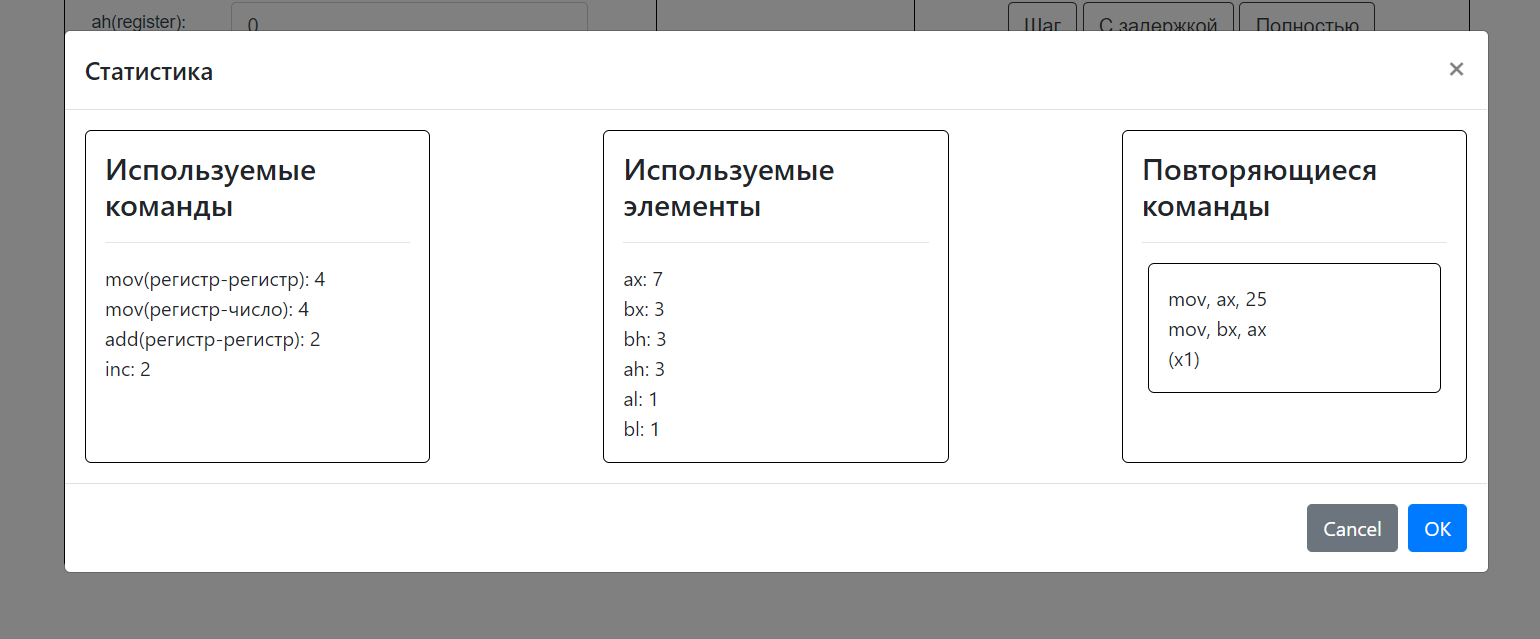


Рисунок 23. Отчёт, содержащий статистику выполнения программного кода

## 3.2 Выводы

В данном разделе рассмотрены основные технологии, языки программирования и библиотеки, использованные для разработки программного продукта. Дано описание использованных технологий, сравнение и обоснование выбора. Приведены примеры программного кода, демонстрирующие выбор технологий. Предоставлены скриншоты работающего программного продукта.

# Заключение

В ходе выполнения курсовой работы был спроектирован, реализован и опробован программный продукт, позволяющий визуализировать использование ресурсов процессора входной программой. Разработанный метод описания процессора также позволяет использовать его для решения задачи виртуализации. В ходе реализации программного продукта были получены следующие результаты:

* разработан метод описания процессора;
* реализован метод выполнения программного кода на описанном процессоре;
* реализован метод виртуализации процессора указанным процессором;
* реализован программный продукт, позволяющий визуально отслеживать изменение состояния процессора при выполнении программного кода.

Разработанное программное обеспечение соответствует требованиям технического задания.

К достоинствам разработанного программного продукта можно отнести:

* возможность модификации и замены эмулируемого процессора;
* возможность сбора статистики использования, позволяющая отлаживать написанный программный код для данного процессора;
* возможность виртуализации процессора, подбора необходимой последовательности команд на другой архитектуре;
* модульность, позволяющая использовать отдельные модули системы или заменять существующие.

К недостаткам можно отнести:

* сложность описания процессора (умение работать на языке Prolog);

В качестве дальнейшего развития предлагается реализация графического языка, транслируемого в пролог и упрощающего процесс описания процессора, реализация временных задержек при выполнении команд.

# Список использованных источниковx

1. IoT. Основные тенденции развития потребительских IoT-устройств [Электронный ресурс] // iot.ru: [сайт]. [2016]. URL: https://iot.ru/gadzhety/osnovnye-tendentsii-razvitiya-potrebitelskikh-iot-ustroystv (дата обращения: **0**1.**06**.2019).

2. Самые востребованные профессии в мире в 2019 году [Электронный ресурс] // EduNews: [сайт]. [2019]. URL: https://edunews.ru/professii/rating/vostrebovannie-v-mire.html (дата обращения: 01.06.2019).

3. Мировой рейтинг языков программирования на февряль 2019 [Электронный ресурс] // PR-CY: [сайт]. [2019]. URL: https://pr-cy.ru/news/p/7329-mirovoy-reyting-yazykov-programmirovaniya-na-fevral-2019 (дата обращения: 01.06.2019).

4. Микроконтроллеры: краткий обзор [Электронный ресурс] // MyRobot: [сайт]. URL: https://myrobot.ru/stepbystep/mc\_meet.php (дата обращения: 01.06.2019).

5. Классификация знаний в области программирования [Электронный ресурс] // Habr: [сайт]. [2015]. URL: https://habr.com/ru/post/249983/ (дата обращения: 01.06.2019).

6. Intel 4004 Emu [Электронный ресурс] // GitHub: [сайт]. [2015]. URL: https://github.com/CodeAbbey/intel4004-emu (дата обращения: 01.06.2019).

7. Intel 4004 emulator [Электронный ресурс] // codeabbey: [сайт]. [2015]. URL: http://www.codeabbey.com/index/wiki/intel-4004-emulator (дата обращения: 01.06.2019).

8. Intel 4004 emu wiki [Электронный ресурс] // GitHub: [сайт]. [2015]. URL: https://github.com/CodeAbbey/intel4004-emu/wiki (дата обращения: 01.06.2019).

9. Intel 4004 system emulator [Электронный ресурс] // e4004: [сайт]. URL: http://e4004.szyc.org/emu/ (дата обращения: 01.06.2019).

10. LabCenter home page [Электронный ресурс] // https://www.labcenter.com/: [сайт]. [2019]. URL: https://www.labcenter.com/ (дата обращения: 01.06.2019).

11. Эмулятор ЭМУ [Электронный ресурс] // strelets: [сайт]. [2007]. URL: http://www.strelets.ru/emu.php (дата обращения: 01.06.2019).

12. Golden B. Virtualization For Dummies. 3rd ed. Indianapolis: Wiley Publishing Inc, 2011. 3 pp.

13. Э.С. Т. Архитектура Компьютера. 4th ed. Москва: С^ППТЕР, 2003. 83-84 pp.