Министерство образования Республики Беларусь

Учреждение образования   
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: архитектура вычислительных систем

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовой работе  
на тему

**РЕШЕНИЕ МАТЕМАТИЧЕСКОЙ ЗАДАЧИ ПРИ ПОМОЩИ СО-ПРОЦЕССОРА**

Студент: гр. 053505 Слуцкий Н. С.

Руководитель: Калиновская А.А.

Минск 2022

Содержание

[Введение 3](#_Toc121252133)

[1 Архитектура вычислительной системы 4](#_Toc121252134)

[1.1 Понятие архитектура и со-процессор 4](#_Toc121252135)

[1.2 Информация о возможных для выбора архитектурах 4](#_Toc121252136)

[1.2.1 RISC-V 4](#_Toc121252137)

[1.2.2 IA-32 8](#_Toc121252138)

[1.2.3 IA-64 8](#_Toc121252139)

[1.2.4 AMD64 9](#_Toc121252140)

[1.3 Обоснование выбора 10](#_Toc121252141)

[1.4 Анализ выбранной системы 10](#_Toc121252142)

[1.5 Floating Point Unit 11](#_Toc121252143)

[1.5.1 Команды передачи данных 13](#_Toc121252144)

[1.5.2 Арифметические команды 13](#_Toc121252145)

[1.5.3 Команды трансцендентных функций 14](#_Toc121252146)

[1.5.4 Команды управления сопроцессором 14](#_Toc121252147)

[2 Платформа программного обеспечения 15](#_Toc121252148)

[2.1 Flat Assembler 15](#_Toc121252149)

[2.2 Microsoft Windows 17](#_Toc121252150)

[2.3 Kernel 32 18](#_Toc121252151)

[3 Теоретическое обоснование разработки программного продукта 19](#_Toc121252152)

[3.1 Используемые технологии программирования 20](#_Toc121252153)

[3.2 Некоторые дополнительные принципы, на которых основана разработка 21](#_Toc121252154)

[3.2.1 DRY – Don't Repeat Yourself or DIE – Duplication Is Evil 21](#_Toc121252155)

[3.2.2 KISS – keep it short simple / keep it simple, stupid 22](#_Toc121252156)

[3.2.3 YAGNI – You ain't gonna need it 22](#_Toc121252157)

[3.2.4 Комментарии 22](#_Toc121252158)

[3.2.5 Именование сущностей 22](#_Toc121252159)

[4 Проектирование функциональных возможностей программы 23](#_Toc121252160)

[5 Архитектура разрабатываемой программы 24](#_Toc121252161)

[Заключение 25](#_Toc121252162)

[Список используемой литературы 27](#_Toc121252163)

[ПРИЛОЖЕНИЯ 28](#_Toc121252164)

# Введение

Целью выполнения данной курсовой работы ставится попытка решения некоторой математической задачи с использованием со-процессора и его инструкций. В качестве задачи было выбрано решение дифференциального уравнения первого порядка с заданным начальным условием. Иначе говоря, целью ставится разработать решение задачи Коши, как можно ближе приблизившись к уровню инструкций процессора.

Одними из дополнительных составляющих данной цели являются:

– создание программного продукта на языке ассемблера с применением концепций из мира более высокоуровневых языков программирования;

– при необходимости симуляция парадигм функционального программирования;

– применение некоторых шаблонов проектирования на низком уровне, если в этом возникнет необходимость;

– создание читаемой и легко поддерживаемой с точки зрения кода программы.

Для достижения поставленной цели определены следующие подзадачи (этапы) разработки:

– выбор конкретной архитектуры;

– выбор со-процессора;

– выбор типа однородного дифференциального уравнения;

– выбор конкретного языка ассемблера;

– реализация решения на выбранной архитектуре с ориентированием на описанные выше цели;

– эмуляция работы реализованного алгоритма.

Создаётся интерес, насколько быстро с точки зрения времени (тактов) выполнения задачи будут решаться и можно ли вообще на выбранной архитектуре относительно адекватно решить поставленную задачу.

В данном курсовом проекте будет выбрана по определению решаемая задача, которая и с точки зрения методов численного анализа представляет интерес, и с точки зрения интересующей дисциплины архитектуры вычислительных систем.

# 1 Архитектура вычислительной системы

## 1.1 Понятие архитектура и со-процессор

Прежде чем рассмотреть основные виды архитектур процессоров, необходимо понять, что это такое. Под архитектурой процессора обычно понимают две разные сущности.

С программной точки зрения архитектура процессора – это совместимость с определённым набором команд (Intel x86), их структуры (система адресации, набор регистров) и способы исполнения. То есть набор поддерживаемых инструкций, конфигурации регистров и некоторые другие аспекты.

С аппаратной точки зрения архитектура процессора – это некоторый набор свойств и качеств, присущий целому семейству процессоров (Skylake – процессоры Intel Core 5 и 6 поколений). Сюда, вероятно, можно вынести и конкретную аппаратную реализацию того или иного процессора (однотактность/многотактность, кеш, возможности арифметическо-логического устройства и другие параметры, которые существуют именно на уровне реализации).

Архитектура вычислительных систем – это совокупность характеристик и параметров, определяющих функционально-логическую и структурную организацию системы. Понятие архитектуры охватывает общие принципы построения и функционирования, наиболее существенные для пользователей, которых больше интересуют возможности систем, а не детали их технического исполнения.

## 1.2 Информация о возможных для выбора архитектурах

### 1.2.1 RISC-V

В качестве первой архитектуры рассматиривается архитектура RISC. RISC-V – открытая и свободная система команд и процессорная архитектура на основе концепции RISC для микропроцессоров и микроконтроллеров.

Спецификации доступны для свободного и бесплатного использования, включая коммерческие реализации непосредственно в кремнии или конфигурировании ПЛИС. Имеет встроенные возможности для расширения списка команд и подходит для широкого круга применений.

Создана в 2010 году исследователями из отделения информатики Калифорнийского университета в Беркли при непосредственном участии Дэвида Паттерсона.

Для развития и продвижения RISC-V в 2015 году создан международный фонд RISC-V и ассоциация со штаб-квартирой в Цюрихе. С 2018 года RISC-V Foundation работает в тесном партнёрстве с The Linux Foundation. В руководство и технические комитеты входят две русские компании разработчики процессорных ядер -- Syntacore и CloudBEAR.

В феврале 2022 года компания Intel объявила об инвестировании в развитие RISC-V одного миллиарда долларов и вошла в состав руководства RISC-V.

В архитектуре RISC-V имеется обязательное для реализации небольшое подмножество команд (набор инструкций I – Integer) и несколько стандартных опциональных расширений.

В базовый набор входят инструкции условной и безусловной передачи управления/ветвления, минимальный набор арифметических/битовых операций на регистрах, операций с памятью (load/store), а также небольшое число служебных инструкций.

Операции ветвления не используют каких-либо общих флагов, как результатов ранее выполненных операций сравнения, а непосредственно сравнивают свои регистровые операнды. Базис операций сравнения минимален, а для поддержки комплементарных операций операнды просто меняются местами.

Базовое подмножество команд использует следующий набор регистров: специальный регистр x0 (zero), 31 целочисленный регистр общего назначения (x1 – x31), регистр счётчика команд (PC, используется только косвенно), а также множество CSR (Control and Status Registers, может быть адресовано до 4096 CSR).

Для встраиваемых применений может использоваться вариант архитектуры RV32E (Embedded) с сокращённым набором регистров общего назначения (первые 16). Уменьшение количества регистров позволяет не только экономить аппаратные ресурсы, но и сократить затраты памяти и времени на сохранение/восстановление регистров при переключениях контекста.

При одинаковой кодировке инструкций в RISC-V предусмотрены реализации архитектур с 32, 64 и 128-битными регистрами общего назначения и операциями (RV32I, RV64I и RV128I соответственно).

Разрядность регистровых операций всегда соответствует размеру регистра, а одни и те же значения в регистрах могут трактоваться целыми числами как со знаком, так и без знака.

Нет операций над частями регистров, нет каких-либо выделенных «регистровых пар». Операции не сохраняют где-либо биты переноса или переполнения, что приближено к модели операций в языке программирования Си. Также аппаратно не генерируются исключения по переполнению и даже по делению на 0. Все необходимые проверки операндов и результатов операций должны производиться программно.

Целочисленная арифметика расширенной точности (большей, чем разрядность регистра) должна явно использовать операции вычисления старших битов результата. Например, для получения старших битов произведения регистра на регистр имеются специальные инструкции.

Размер операнда может отличаться от размера регистра только в операциях с памятью. Транзакции к памяти осуществляются блоками, размер в байтах которых должен быть целой неотрицательной степенью 2, от одного байта до размера регистра включительно. Операнд в памяти должен иметь «естественное выравнивание» (адрес кратен размеру операнда).

Архитектура использует только модель little-endian – первый байт операнда в памяти соответствует младшим битам значений регистрового операнда.

Для пары инструкций сохранения/загрузки регистра операнд в памяти определяется размером регистра выбранной архитектуры, а не кодировкой инструкции (код инструкции один и тот же для RV32I, RV64I и RV128I, но размер операндов 4, 8 и 16 байт соответственно), что соответствует размеру указателя, типам языка программирования C size\_t или разности указателей.

Для всех допустимых размеров операндов в памяти, меньших, чем размер регистра, имеются отдельные инструкции загрузки/сохранения младших битов регистра, в том числе для загрузки из памяти в регистр есть парные варианты инструкций, которые позволяют трактовать загружаемое значение как со знаком (старшим знаковым битом значения из памяти заполняются старшие биты регистра) или без знака (старшие биты регистра устанавливаются в 0).

Инструкции базового набора имеют длину 32 бита с выравниванием на границу 32-битного слова, но в общем формате предусмотрены инструкции различной длины (стандартно – от 16 до 192 бит с шагом в 16 бит) с выравниванием на границу 16-битного слова. Полная длина инструкции декодируется унифицированным способом из её первого 16-битного слова.

Для наиболее часто используемых инструкций стандартизовано применение их аналогов в более компактной 16-битной кодировке (C – Compressed extension).

Операции умножения, деления и вычисления остатка не входят в минимальный набор инструкций, а выделены в отдельное расширение (M – Multiply extension). Имеется ряд доводов в пользу разделения и данного набора на два отдельных (умножение и деление).

Стандартизован отдельный набор атомарных операций (A – Atomic extension). Поскольку кодировка базового набора инструкций не зависит от разрядности архитектуры, то один и тот же код потенциально может запускаться на различных RISC-V архитектурах, определять разрядность и другие параметры текущей архитектуры, наличие расширений системы инструкций, а потом автоконфигурироваться для целевой среды выполнения.

RISC-V имеет 32 (или 16 для встраиваемых применений) целочисленных регистра. При реализации вещественных групп команд есть дополнительно 32 вещественных регистра.

Рассматривается вариант включения в стандарт дополнительного набора из 32 векторных регистров с вариативной длиной обрабатываемых значений, длина которых указывается в CSR vlenb.

Для операций над числами в бинарных форматах плавающей запятой используется набор дополнительных 32 регистров FPU (Floating Point Unit), которые совместно используются расширениями базового набора инструкций для трёх вариантов точности: одинарной – 32 бита (F extension), двойной – 64 бита (D – Double precision extension), а также четверной – 128 бит (Q – Quadruple precision extension).

Следующая для рассмотрения архитектура – Intel. Intel Architecture – система архитектур процессора, разрабатываемых компанией Intel. Данные архитектуры были совместимы только со своим набором инструкций и одна из них использовалась в процессорах других компаний. В настоящее время подразделяется на две архитектуры: IA-32 и IA-64.

### 1.2.2 IA-32

IA-32 (Intel Architecture, 32-bit) – микропроцессорная архитектура, третье поколение архитектуры x86, ознаменовавшееся переходом на 32-разрядные вычисления. Первый представитель архитектуры – микропроцессор Intel 80386, выпущенный 17 октября 1985 года. Также архитектуру часто называют i386 (по имени первого выпущенного на ней процессора) и x86-32 (по применяемому набору команд). Эти метонимы получили широкое распространение, в том числе в справочной литературе и документации.

Архитектура IA-32, созданная корпорацией Intel в 1985 году, на двадцать лет стала доминирующей среди микропроцессоров для персональных компьютеров. В дальнейшем была вытеснена 64-разрядной архитектурой x86-64.

Процессоры с архитектурой IA-32 также производились AMD, Cyrix, Via, Transmeta, SiS, UMC и многими другими. После 2010 года процессоры архитектуры IA-32 всё ещё разрабатываются и производятся, например Intel Atom, AMD Geode и VIA C7, которые позиционируются как процессоры для мобильных и встраиваемых систем.

### 1.2.3 IA-64

IA-64 (Intel Architecture-64) – 64-битная аппаратная платформа: микропроцессорная архитектура и соответствующая архитектура набора команд, разработанная совместно компаниями Intel и Hewlett Packard. Реализована в микропроцессорах Itanium и Itanium 2.

Основана на VLIW или, в терминах Intel, EPIC (сокр. от англ. Explicity Parallel Instruction Computing, вычисления с явной параллельностью инструкций). Несовместима с архитектурой x86. Изначально предлагалась и в качестве платформы для домашних компьютеров, но после выпуска фирмой AMD 64-битной архитектуры AMD64, сохранившей совместимость с x86, актуальность использования платформы IA-64 где-либо, кроме серверов, пропала, несмотря на то, что в конце 2001 года для IA-64 была выпущена специальная версия Windows XP 64-bit for IA-64. Также на архитектуру IA-64 портирована операционная система OpenVMS, принадлежащая HP.

### 1.2.4 AMD64

x86-64 (также AMD64/Intel64/EM64T) – 64-битная версия (изначально – расширение) архитектуры x86, разработанная компанией AMD и представленная в 2000 году[1], позволяющая выполнять программы в 64-разрядном режиме.

Это расширение архитектуры x86, а ныне – версия архитектуры x86, почти полностью обратно совместимая с 32-разрядной версией архитектуры x86, известной ныне как IA-32.

Корпорации Microsoft и Oracle используют для обозначения этой версии архитектуры x86 термин «x64», однако каталог с файлами для архитектуры в 64-разрядных Microsoft Windows и называется «amd64» («i386» для соответственно архитектуры x86). Подобное наблюдается и в репозиториях большинства Linux-дистрибутивов.

Разработанный компанией AMD набор инструкций x86-64 (позднее переименованный в AMD64) – расширение архитектуры Intel IA-32 (x86-32). Основной отличительной особенностью AMD64 является поддержка 64-битных регистров общего назначения, 64-битных арифметических и логических операций над целыми числами и 64-битных виртуальных адресов. Для адресации новых регистров для команд введены так называемые «префиксы расширения регистра», для которых был выбран диапазон кодов 40h-4Fh, использующихся для команд INC <регистр> и DEC <регистр> в 32-битных режимах. Команды INC и DEC в 64-битном режиме должны кодироваться в более общей, двухбайтовой форме.

Архитектура x86-64 имеет:

– 16 целочисленных 64-битных регистров общего назначения (RAX, RBX, RCX, RDX, RBP, RSI, RDI, RSP, R8 – R15);

– 8 80-битных регистров с плавающей точкой (ST0 – ST7);

– 8 64-битных регистров Multimedia Extensions (MM0 – MM7, имеют общее пространство с регистрами ST0 – ST7);

– 16 128-битных регистров SSE (XMM0 – XMM15);

– 64-битный указатель RIP и 64-битный регистр флагов RFLAGS.

## 1.3 Обоснование выбора

Описанные выше архитектуры используются повсеместно. Есть относительно большое комьюнити разработчиков, документация и для старта разработки это хорошие варианты. Но необходимо остановиться на одной архитектуре. Ввиду факта, что ранее изучалась архитектура IA-32, выбор в общем случае сделан в пользу этой архитектуры. Для разработки будет использоваться язык ассемблера Flat Assembler и библиотеки функций операционной системы Windows. Ввиду относительно большого количества обсуждений и документации про FASM, это можно считать хорошим выбором.

## 1.4 Анализ выбранной системы

В процессорах IA32, если говорить о самых первых моделях, были только инструкции общего назначения. Поддержка вещественных чисел отсутствовала. Одним из вариантов решения этой проблемы была установка дополнительного так называемого со-процессора к основному процессору. Например, со-процессор 8087, который можно было установить к процессору 8086. В Turbo Pascal возможность эмуляции вещественного типа данных Real появилась достаточно рано. Тип без проблем эмулировался компилятором. Была возможность работать, как будто имелся со-процессор, однако на самом деле он отсутствовал. Разница наблюдалась лишь в скорости работы.

Начиная с 486 процессора появилась встроенная поддержка вещественных чисел. Можно сказать, со-процессор был объединён с основным процессором. Поэтому в современных процессорах IA-32, начиная с 80486, кроме 486SX, поддержка команд со-процессора 8087 является встроенной.

Вышеописанный со-процессор предлагает:

– 8 регистров данных для хранения вещественных чисел;

– управляющий регистр;

– регистр статуса.

Сами регистры представляют собой 80-битные ячейки данных. Поддерживаемыми типами могут быть:

– целочисленные: 16, 32, 64 бита;

– вещественные: 32, 64, 80 бит.

## 1.5 Floating Point Unit

FPU (Floating Point Unit) используется для ускорения и упрощения вычислений с плавающей точкой.

Сопроцессор (другое название FPU) ориентирован на математические вычисления - в нем отсутсвуют операции с битами, зато расширен набор математических функций: тригонометрические, логарифм и т.д.

Система команд сопроцессора включает в себя около 80 машинных команд, включающих в себя:

– команды передачи данных;

– команды сравнения данных;

– арифметические команды;

– команды трансцендентных функций;

– команды управления сопроцессором.

Мнемоническое обозначение команд сопроцессора характеризует особенности их работы и в связи с этим может представлять определенный интерес. Поэтому коротко рассмотрим основные моменты образования названий команд:

– все мнемонические обозначения начинаются с символа f (float);

– вторая буква мнемонического обозначения определяет тип операнда в памяти, с которым работает команда: – i – целое двоичное число; – b – целое десятичное число; – отсутствие буквы – вещественное число;

– последняя буква мнемонического обозначения команды р означает, что последним действием команды обязательно является извлечение операнда из стека;

– последняя или предпоследняя буква r (reversed) означает реверсивное следование операндов при выполнении команд вычитания и деления, так как для них важен порядок следования операндов.

Система команд сопроцессора отличается большой гибкостью в выборе вариантов задания команд, реализующих определенную операцию, и их операндов. Минимальная длина команды сопроцессора – 2 байта. Все команды сопроцессора оперируют регистрами стека сопроцессора. Если операнд в команде не указывается, то по умолчанию используется вершина стека сопроцессора (логический регистр st(0)). Если команда выполняет действие с двумя операндами по умолчанию, то эти операнды – регистры st(0) и st(1).

До некоторых моделей 80486 FPU исполнялся в виде отдельной микросхемы, устанавливаемой опционально. В некоторых 486 и во всех Pentium (и выше), FPU интегрирован в процессор, поэтому можно считать, что он присутствует в каждом более-менее современном компьютере. Начиная с Pentium Pro FPU имеет дополнительные команды, упрощающие сравнение чисел. В любом случае, если FPU недоступен, или неподдерживает какие-либо ф-ции, то возможна их программная эмуляция.

Для работы с FPU существует отдельная группа команд, название которых начинается с символа "f".

Сопроцессор работает паралельно CPU, и, поэтому, ранее необходимо было перед многими командами FPU вставлять команду fwait, чтобы приостановить выполнение до окончания вычислений.

В последующих сопроцессорах эта команда встроена в инструкции FPU, поэтому ее использовать не нужно. Но остались также не ожидающие команды, имеющие приставку "n". Пример fsave/fnsave - одно и тоже действие, но первая команда вызовет исключение, если оно произошло до этого, но еще не вызывлось. На самом деле эти две команды - одни и теже, но перед fsave компилятор добавляет команду fwait.

Операндом команд является, как правило, один из регистров st(), или ячейка памяти. Другой операнд обычно подразумевается st(0). Комнады с операндом "память" могут иметь приставку "i", что означает операнд - обычное число, без приставки - в FP формате.

Большинство команд не работает с регистрами общего назначения. Для работы с числом из регистра общего назначения можно, например, использовать стэк.

Несмотря на то, что многие команды работают только с st(0), можно легко использовать число из другого FP регистра, поменяв значения регистров командой fxch.

Некоторые команды имеют суффикс "r", что означает назначение операции - второй операнд.

Также, если в команде присутствует суффикс "p", то происходит выталкивание числа из стэка сопроцессора.

Процедуры, работающие с FPU обычно придерживаются следующего соглашения:

На входе и выходе стэк сопроцессора пуст. Иногда для возврата FP числа используется st(0).

Можно использовать все регистры FPU, за исключением CW.

Команд FPU не много, и их легко отличить по первой букве "f".

Можно долго рассуждать, почему регистры FPU недоступны напрямую, а только в виде стека. Достаточно верным окажется предположение об экономии средств. Действительно, создание восьми новых регистров повлекло бы за собой увеличение затрат на формирование команд. Стек FPU несколько отличается от классического привычного стека.

### 1.5.1 Команды передачи данных

Группа команд передачи данных предназначена для организации обмена между регистрами стека, вершиной стека сопроцессора и ячейками оперативной памяти. Команды этой группы имеют такое же значение для процесса программирования сопроцессора, как и команда mov основного процессора. С помощью этих команд осуществляются все перемещения значений операндов в сопроцессор и из него. По этой причине для каждого из трех типов данных, с которыми может работать сопроцессор, существует своя подгруппа команд передачи данных. Собственно на этом уровне все его умения по работе с различными форматами данных и заканчиваются. Главной функцией всех команд загрузки данных в сопроцессор является преобразование их к единому представлению в виде вещественного числа расширенного формата. Это же касается и обратной операции – сохранения в памяти данных из сопроцессора.

### 1.5.2 Арифметические команды

Команды сопроцессора, входящие в данную группу, реализуют четыре основные арифметические операции – сложение, вычитание, умножение и деление. Имеется также несколько дополнительных команд, предназначенных для повышения эффективности использования основных арифметических команд. С точки зрения типов операндов, арифметические команды сопроцессора можно разделить на команды, работающие с вещественными и целыми числами.

### 1.5.3 Команды трансцендентных функций

Сопроцессор имеет ряд команд, предназначенных для вычисления значений тригонометрических функций, а также значений логарифмических и показательных функций. Значения аргументов в командах, вычисляющих результат тригонометрических функций, должны задаваться в радианах. Данная группа команд не имеет операндов. Результат сохраняется в регистре ST(0). Сбрасывает в 0 признак C1 при пустом стеке, устанавливают в 1 при округлении.

### 1.5.4 Команды управления сопроцессором

Данная группа команд предназначена для общего управления работой сопроцессора. Команды этой группы имеют особенность – перед началом своего выполнения они не проверяют наличие незамаскированных исключений. Однако такая проверка может понадобиться, в частности для того, чтобы при параллельной работе основного процессора и сопроцессора предотвратить разрушение информации, необходимой для корректной обработки исключений, возникших в сопроцессоре. Поэтому некоторые команды управления имеют аналоги, выполняющие те же действия плюс одну дополнительную функцию – проверку наличия исключения в сопроцессоре.

# 2 Платформа программного обеспечения

В используемом рабочем компьютере используется архитектура на основе процессора AMD Ryzen 7 4800H (7 nm). В качестве ОС выступает OS Windows 10. В рамках данного окружения, зная факт, что в процессоре присутствует математический со-процессор (который и интересует автора в данном курсовом проекте), можно исполнять разными способами на этом со-процессоре (возможно, прибегая к эмуляторам и трансляторам) код, написанный на языке низкого уровня (TASM, FASM, NASM, RISC-V, ASM, Си). Разработка может производиться в специальных программах по типу Flat Assembler for Windows, TASM IDE или Microsoft Visual Studio Code. Эти программы при связи с компилятором и линкером способны обеспечить правильное исполнение инструкций, которые автор затребует в рамках написания программы. Учитывая, что программный продукт разрабатывается для работы в среде операционной системы Windows, для более комфортной и корректной разработки также потребуется использование некоторых системных функций, предоставляемых операционной системой. Это такие функции, как, например printf. В рамках данного курсового проекта некоторые подобные функции будут реализованы с нуля, но при этом для корректной работы именно на ОС Windows будет необходимо прибегать к использованию конкретно системных функций.

## 2.1 Flat Assembler

Наряду с описанным компилятором TASM сосуществуют и другие, менее именитые разработки.

Почему хорошей альтернативой является FASM? Для этого выбора есть несколько причин. Во-первых, он являлся одним из наиболее динамично развивающихся компиляторов. Его автор Tomasz Grysztar регулярно выкладывает новые версии на свою страничку http://fasm.sourceforge.net/, откуда их может получить любой желающий. Архив версии 1.46 от 9 апреля 2003 года занимает всего 240 Кб, если предполагается работать в режиме командной строки DOS, и 550 Кб – если разработку планируется вести в среде Windows. И это при том, что в дистрибутив для Windows входит подробная документация в формате PDF, которая содержит описание как самого компилятора, так и машинных инструкций процессоров Intel включая набор команд MMX, SSE, SSE2 и AMD 3DNow! Все перечисленные команды могут быть использованы в программах на FASM.

Стоит отметить, что работать компилятор FASM будет только на компьютерах, оснащенных процессором не хуже Intel 80386, однако сегодня это вряд ли можно отнести к недостаткам. Тем более, что он позволяет генерировать код как для самых современных процессоров, так и для стареньких Intel 8086.

На этом этапе нужно обратить внимание на еще одну особенность рассматриваемого продукта. Дело в том, что FASM является компилятором и компоновщиком одновременно. Разработчик, использующий его, не нуждается ни в каких дополнительных утилитах. На входе FASM получает текст программы на языке ассемблера, а на выход выдается машинная программа в формате COM или EXE для DOS, DLL или PE (Portable Executable) для Windows, уже готовая к выполнению.

Такой механизм работы FASM вызывает неоднозначную оценку. С одной стороны, это упрощает процесс получения исполняемого файла, с другой – делает невозможным использование традиционных OBJ- и LIB-модулей. Приходится накапливать подпрограммы в текстовых файлах и подключать к основному модулю с помощью директивы INCLUDE. Такая технология ведет к неизбежному замедлению процесса компиляции, однако справедливости ради нужно отметить, что на современной технике это замедление не является критическим. Естественно, эффективность генерируемого машинного кода при этом нисколько не страдает.

Fasm стремится использовать минимально возможный набор директив препроцессора, т.е. в предустановленном наборе директив не допускается внедрение новых директив, функциональность которых может быть достигнута имеющимся набором директив. Исключение исторически сложившиеся взаимозаменяемые директивы.

Fasm – многопроходный ассемблер с оптимистическим предсказанием, т.е. на первом же проходе ассемблер делает предположение, что все инструкции принимают свою минимально возможную по размеру форму. Многопроходность также позволяет неограниченно использовать выражения до их объявления.

Fasm не включает в выходной файл объявления не используемых процедур (реализовано посредством макрокоманд).

Содержимое выходного файла зависит только от содержания исходного кода и не зависит от окружения операционной системы или от параметров переданных в командной строке. Для тех кому данный принцип был неудобен для win32 была разработана обертка FA, позволяющая подключить к файлу другой файл не непосредственно в коде, а через командную строку.

Исходный код для fasm может собираться сразу в исполняемый файл, минуя стадии создания промежуточных объектных файлов и их компоновки.

## 2.2 Microsoft Windows

Windows – группа семейств коммерческих проприетарных операционных систем корпорации Microsoft, ориентированных на управление с помощью графического интерфейса. MS-DOS – является прародителем Windows. Каждое семейство обслуживает определённый сектор компьютерной индустрии. Активные семейства Microsoft Windows включают Windows NT и Windows IoT; они могут включать подсемейства (например, Windows Server или Windows Embedded Compact) (Windows CE). Неактивные семейства Microsoft Windows включают Windows 9x, Windows Mobile и Windows Phone. Изначально Windows была всего лишь графической программой-надстройкой для распространённой в 1980-х и 1990-х годах операционной системы MS-DOS. Согласно данным ресурса Net Applications, по состоянию на август 2014 года под управлением операционных систем семейства Windows работает около 88 % персональных компьютеров. Windows работает на PC-совместимых архитектурах с процессорами x86, x86-64, а также на архитектуре ARM. Существовали также версии для DEC Alpha, MIPS, IA-64, PowerPC и SPARC. Последней на данный момент операционной системой Microsoft является Windows 11, представленная 24 июня 2021 года.

Первая система данного семейства – Windows 95 – была выпущена в 1995 году. Её отличительными особенностями являлись новый пользовательский интерфейс, поддержка длинных имён файлов, автоматическое определение и конфигурация периферийных устройств Plug and Play (с англ. – «Подключи и играй»), способность исполнять 32-битные приложения и наличие поддержки TCP/IP прямо в системе. Windows 95 использовала вытесняющую многозадачность и выполняла каждое 32-битное приложение в своём адресном пространстве. К данному семейству относятся также Windows 98 и Windows Me.

Операционные системы этого семейства не являлись такими безопасными многопользовательскими системами, как Windows NT, поскольку из соображений совместимости вся подсистема пользовательского интерфейса и графики оставалась 16-битной и мало отличалась от той, что была в Windows 3.x. Так как этот код не был потокобезопасным, все вызовы в подсистему оборачивались в мьютекс по имени Win16Lock, который ещё и находился всегда в захваченном состоянии во время исполнения 16-битного приложения. Таким образом, «зависание» 16-битного приложения немедленно блокировало всю ОС. Но уже в 1999 году вышло второе исправленное издание.

Программный интерфейс был подмножеством Win32 API, поддерживаемым Windows NT, но имел поддержку Юникода в очень ограниченном объёме. Также в нём не было должного обеспечения безопасности (списков доступа к объектам и понятия «администратор»).

В составе Windows 95 присутствовала MS-DOS 7.0, однако её роль сводилась к обеспечению загрузки и исполнения 16-битных DOS-приложений. Исследователи заметили, что ядро Windows 95 – VMM – обращается к DOS под собой, но таких обращений довольно мало, главнейшая функция ядра DOS – файловая система FAT – не использовалась. В целом же интерфейс между VMM и нижележащей DOS никогда не публиковался, и DOS была замечена Эндрю Шульманом (книга «Недокументированный Windows 95») в наличии недокументированных вызовов только для поддержки VMM.

## 2.3 Kernel 32

Kernel32.dll – динамически подключаемая библиотека, являющаяся ядром всех версий ОС Microsoft Windows. Она предоставляет приложениям многие базовые API Win32, такие как управление памятью, операции ввода-вывода, создание процессов и потоков и функции синхронизации.

# 3 Теоретическое обоснование разработки программного продукта

Всегда представлялось интересным и важным для сообщества научиться автоматизировать как можно больше решаемых задач. Не исключением является желание научить машины решать сначала простые, а потом более сложные математические задачи. Это представляет большое значение особенно для других сфер, которые при этом плотно связаны с математикой (астрономическая сфера, физическая сфера и так далее).

В рамках изучаемой студентами второго и третьего курсов специальности “Информатика и технологии программирования” дисциплины “Методы численного анализа” изучаются приёмы решения математических задач: дифференциальных уравнений, систем линейных алгебраических уравнений, аппроксимации, интерполяции и других. Для практики в рамках данной дисциплины требуется реализовывать алгоритмы решения задач с использованием языков программирования: Python или C++. Это языки относительно высокого уровня. Напрямую с поддерживающимися инструкциями процессором взаимодействовать в них, конечно, можно, но в любом случае не так явно, как через язык Assembler-а, который напрямую общается с регистрами и т.д. (В случае с C++ речь идёт о классической консольной/оконной C++ программе, собираемой с помощью CMake и компилируемой с помощью компиляторов семейства GNU, без подключения дополнительных, вероятно, системных библиотек и без ассемблерных вставок). Как было описано во введении, в качестве задачи было выбрано решение дифференциального уравнения первого порядка с начальным условием – задачи Коши. В алгоритме решения будет использоваться неявный метод Адамса высокого порядка.

Метод Адамса – конечноразностный многошаговый метод численного интегрирования обыкновенных дифференциальных уравнений первого порядка. В отличие от конкурирующего метода Рунге-Кутты использует для вычисления очередного значения искомого решения не одно, а несколько значений, которые уже вычислены в предыдущих точках.

Метод назван по имени предложившего его в 1855 году английского астронома Джона К. Адамса.

Пусть дана система дифференциальных уравнений первого порядка

,

для которой необходимо найти решение на сетке с постоянным шагом. В настоящей курсовой работе будет применяться явный метод Адамса, который также называется экстраполяционным или методом Адамса-Башфорта.

Методы Адамса k-го порядка требуют предварительного вычисления решения в k начальных точках. Для вычисления начальных значений обычно используют одношаговые методы, например, 4-стадийный метод Рунге – Кутты 4-го порядка точности.

Структура погрешности метода Адамса такова, что погрешность остаётся ограниченной или растёт очень медленно в случае асимптотически устойчивых решений уравнения. Это позволяет использовать этот метод для отыскания устойчивых периодических решений, в частности, для расчёта движения небесных тел.

## 3.1 Используемые технологии программирования

Для разработки программного продукта для решения выбранной задачи используется язык ассемблера, компилятор TASM **FASM** и эмуляция архитектуры Intel IA32 (x86).

В ходе решения математической задачи будут использоваться доступные регистры или их части, выделяемая область стека, возможности создания переменных и хранения там данных. Будут использоваться такие возможности языка, как ветвления (в том числе и безусловные), процедуры.

При написании кода программы целью является и в том числе следующее:

– создать максимально читаемую и в будущем легко поддерживаемую программу;

– создать (в рамках конкретно этой задачи, но для начала для выделенной размерности) легко расширяемую программу, чтобы при незначительных изменениях подзадач выбранной математической задачи требовались незначительные изменения в коде функций;

– декомпозировать все относительно атомарные (в рамках шагов решения математической задачи) действия на отдельные функции, придерживаясь принципа DRY и KISS;

– внедрить понятие “codestyle” из языков и программ высокого уровня в язык ассемблера и написать лаконичный для последующей работы или просмотра код;

– протестировать решение на нескольких возможных наборах данных.

## 3.2 Некоторые дополнительные принципы, на которых основана разработка

В соответствии с поставленной целью, функциональность программы будет максимально декомпозирована на независимые и неделимые в рамках математической подзадачи “методы”, чтобы чётко следовать принципам функционального программирования, принципам единой ответственности, модульности и расширяемости. Поэтому после точки входа будут реализованы процедуры, соответствующие нуждам решения поставленной задачи. Каждая процедура может сопровождаться понятным пояснением насчёт:

– требуемое размещение входных параметров;

– размещение выходных параметров;

– затрагиваемые регистры при отработке процедуры;

– будет ли задействован стек при работе процедуры;

– затрагиваемые переменные для работы процедуры;

– другие пояснения по необходимости.

Ставится целью написать в том числе как можно более “чистый” код.

Легкоподдерживаемый читаемый код – то, к чему стремится любой опытный разработчик. Это код, который легко читать через 2 месяца, полгода, год и больше после его написания, причём не только автору, но и любому другому программисту. А так как в большинстве случаев код разрабатывается в командах – ваши сокомандники должны иметь возможность легко разбираться в вашем кусочке приложения, не прилагая усилий, чтобы расшифровать написанную логику.

### 3.2.1 DRY – Don't Repeat Yourself or DIE – Duplication Is Evil

Принцип призывает не повторяться при написании кода. При несоблюдении этого принципа программист будет вынужден вносить изменения в несколько повторяющихся фрагментов кода, вместо одного. Также дублирующийся код приводит к разрастанию программы, а значит, усложняет ее понимание, читабельность.

### 3.2.2 KISS – keep it short simple / keep it simple, stupid

Чем проще код, тем легче в нём разобраться. Под простотой подразумевается отказ от использования хитроумных приемов и ненужного усложнения.

### 3.2.3 YAGNI – You ain't gonna need it

Всё, что не предусмотрено заданием проекта, не должно быть в нём.

### 3.2.4 Комментарии

Необходимо пояснять, комментировать код, где это возможно.

Комментарии могут использоваться для пояснения следующих моментов:

– задача кода;

– предпочтительность выбранного решения.

В то же время не стоит задача покрыть комментариями весь код. Использование значимых названий переменных и функций, разбитие кода на логические фрагменты с помощью функции и другие практики помогают сделать код максимально читаемым и понятным не прибегая к комментариям (самодокументирующийся код).

### 3.2.5 Именование сущностей

Необходимо придерживаться единого стиля именования файлов в проекте. В рамках данного курсового проекта будет использоваться именование переменных и процедур буквами нижнего регистра с разделением в виде символа нижнего подчеркивания.

# 4 Проектирование функциональных возможностей программы

Создание программного продукта будет начато с базовой настройки проектного файла, прогнозирования необходимой модели программы, размера стека, набора переменных для хранения некоторых данных, возможно, набора необходимых для работы констант, переменных, списка используемых библиотек с учётом того, что программный продукт разрабатывается для работы в среде операционной системы Windows.

Будут написаны необходимые функции для работы с числами, строками, такие, как:

– конвертация;

– вывод чисел;

– поиск.

Когда выше речь шла о константах, подразумевались такие полезные для работы программы значения, как:

– знак “минус” для вывода вместе с отрицательным положительным числом;

– шаблонные строки форматирования для передачи в параметр функции printf из пространства системных функций операционной системы Microsoft Windows;

– константы, которые непосредственно относятся к выбранному решаемому дифференциальному уравнению (коэффициенты);

Что касается переменных, которые будут переиспользоваться в течение жизненного цикла работы алгоритма программы, можно выделить некоторые примеры таких переменных:

– переменная для записи значения цифры числа во время вывода этого числа на экран (макрос printf забирает значения из переданных переменных, а не регистров);

– различные флаги;

– размерность сетки, которая будет определяться из констант, определённых в условии задачи;

Также будут реализованы функциональные возможности ввода чисел, ввода строк, возможно, функции по работе с многочленами и так далее.

Далее может понадобиться реализовать инструменты для работы с математическими примитивами и не только. В теории это могут быть функции:

– для подсчёта разных норм математических функций заданного формата (возможно, сеточных);

– вычитания двух функций;

– вычисление нормы элементов линейного пространства (в простейшем случае – модуля);

– дифференцирования многочлена;

– нахождения корней квадратного трёхчлена;

– вывода сеточной функции в приемлемую для анализа табличку;

– другие.

# 5 Архитектура разрабатываемой программы

Исходный ресурс программного продукта представляет собой ASM файл, в котором написан программный код на низкоуровневом языке ассемблера.

Программа разбита на секцию со стеком, данными, кодом.

По аналогии с классическими консольными программами на таких языках программирования, как C/C++, Java, C#, программа здесь будет иметь точку входа, своеобразную функцию main.

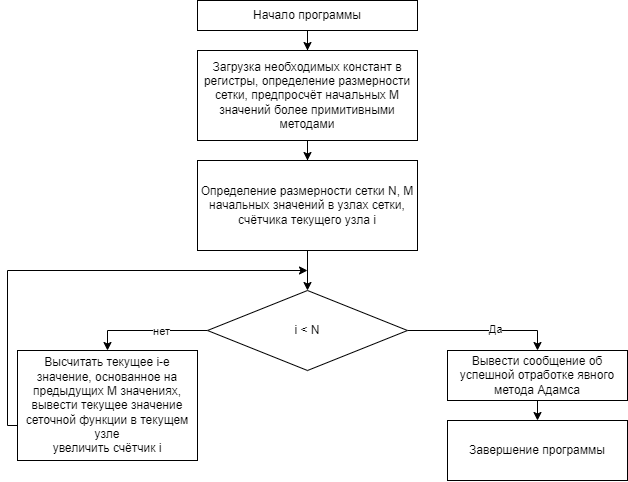
****

Рисунок 1 Общая блок-схема алгоритма программы

# Заключение

В ходе выполнения данной курсовой работы был разработан программный продукт, решающий поставленную задачу в виде решения дифференциального уравнения первого порядка с начальным условием, называемой задачей Коши.

С подключением инструкций и регистров отдельного математического со-процессора 8087 стало возможным реализовать алгоритм решения задачи, использующий вещественные числа и операции над ними, которые предоставляет модуль Floating Point Unit (FPU) из вышеупомянутого со-процессора.

С помощью библиотечных функций операционной системы windows программа может запускаться в консольном режиме и вести себя в общем случае аналогично консольной программе, написанной на более высокоуровневом языке программирования, таком, как, например, C, C++, C#, Java. В скомпилированном виде программа представляет собой исполняемый файл для операционной системы Windows, который может быть самостоятельно запущен.

Цели, поставленные во введении к данной курсовой работе, можно считать достигнутыми.

# Список используемой литературы

* Официальная документация Microsoft   
  (режим доступа:   
  URL: <https://learn.microsoft.com/ru-ru/cpp/intrinsics/?view=msvc-160>)
* Справочник инструкций для x86, AMD 64:  
  (режим доступа:  
  URL: <https://www.felixcloutier.com/x86/>)
* Основное руководство по Flat Assembler:  
  (режим доступа: URL: <http://flatassembler.narod.ru/fasm.htm>)
* Кодирование математических выражений на FPU  
  (режим доступа: URL: <https://allasm.ru/proc_02.php#5>)
* Стандарт предприятия БГУИР 2017  
  (режим доступа: URL: <https://www.bsuir.by/m/12_100229_1_122976.pdf>)
* Список инструкций для Floating Point Unit  
  (режим доступа:   
  URL: <https://linasm.sourceforge.net/docs/instructions/fpu.php>)
* Использование системных функций Windows для вывода чисел из FPU модуля  
  (режим доступа:   
  URL: <https://programmersforum.ru/showthread.php?t=126830>)

# ПРИЛОЖЕНИЯ

FORMAT PE Console

ENTRY main

INCLUDE 'win32a.inc'

include 'C:\Users\User\Downloads\fasmw17330\INCLUDE\API\kernel32.inc'

include 'C:\Users\User\Downloads\fasmw17330\INCLUDE\API\user32.inc'

SECTION '.data' data readable writeable

minus\_symbol db '-', 0 ; 0 => string end

digit\_temp dd ?

X dq 1.57

fX db 'sin',0

ffmt db '%s = %.10f',0

SECTION '.code' code readable writeable executable

main:

;stdcall print\_signed\_integer, -27766889

finit ; initialize FPU module

fld [X]

fsin

fstp [X]

mov eax, dword[X]

cinvoke printf,ffmt,fX,dword[X],dword[X+4]; ????????

;invoke ftoa [string], [nums\_count]

;fstp test\_double

;invoke printf, double\_format\_string test\_double

finish\_program:

invoke getch

invoke ExitProcess, 0

proc print\_signed\_integer, number

mov eax, [number]

action\_for\_negative\_numbers:

cmp eax, 0

jge output\_preparation ; if (ax > 0) goto count\_digits

neg eax ; ax = (-1) \* ax

mov [temp], eax ; printf spoils registers => need to remember EAX

invoke printf, minus\_symbol

mov eax, [temp] ; restore EAX

pop ecx

output\_preparation:

xor ecx, ecx ; <=> mov cx, 0

xor edx, edx ; <=> mov dx, 0

xor ebx, ebx

mov ebx, 10 ; for dividing on bxs

count\_digits:

div ebx ; div <=> (dx+ax) / 10 => num / 10 -> ax, num % 10 -> dx

push edx ; remember received digit

xor edx, edx ; made dx = 0

inc ecx ; increased counter of digits

test eax, eax ; ax === 0 ?

jnz count\_digits ; if yes, will repeat and get another digit

print\_digits:

pop edx ; take digit

add edx, 30h ; += 30h to convert it in ASCII view for printing

mov [digit\_temp], edx

push ecx ; printf spoils registers => so need to save it

invoke printf, digit\_temp

pop edx ; printf also pushes addresses of used parameters => so need to clear pushed parameters

pop ecx ; restore ECX, because I need it for loop

loop print\_digits

ret

endp

SECTION '.idata' data import readable

library kernel, 'kernel32.dll',msvcrt, 'msvcrt.dll'

import kernel, ExitProcess, 'ExitProcess'