Министерство образования Республики Беларусь
Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина «Архитектура вычислительных систем»

«К ЗАЩИТЕ ДОПУСТИТЬ»
Руководитель курсового проекта
магистр техн.наук, ассистент
_____А.А. Калиновская
___.____.2022

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему:
«РЕШЕНИЕ МАТЕМАТИЧЕСКОЙ ЗАДАЧИ ПРИ ПОМОЩИ СОПРОЦЕССОРА»

БГУИР КП 1-40 04 01 020 ПЗ

Выполнил студент группы 053505
Слуцкий Никита Сергеевич

_____
(подпись студента)
Курсовой проект представлен на проверку ___.____.2022

_____
(подпись студента)

Минск 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью выполнения данной курсовой работы ставится попытка решения некоторой математической задачи с использованием сопроцессора 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 сопроцессора;

– 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 сопроцессор

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 [1] (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, кэш, 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 рассматривается 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).

1.2.2 IA-32

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.

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 (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, позволяющая выполнять программы в 64-разрядном режиме [2].

Это расширение архитектуры x86, а в данный момент – 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 сопроцессора к основному процессору. Например, сопроцессор 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 сопроцессор, однако на самом деле он отсутствовал. Разница наблюдалась лишь в скорости работы.

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

Вышеописанный сопроцессор 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 [3] (Floating Point Unit) используется для ускорения и упрощения вычислений с плавающей точкой.

Сопроцессор ориентирован на математические вычисления – в нем отсутствуют операции с битами, но расширен набор математических функций: тригонометрические, логарифм и другие.

Система команд [4] сопроцессора включает в себя около 80 машинных команд, включающих в себя:

– команды передачи данных;

– команды сравнения данных [5];

– 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 p 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 [6]. 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 не поддерживает какие-либо функции, то возможна их программная эмуляция.

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

Сопроцессор работает параллельно 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».

Операндом команд является, как правило, один из регистров st(), или ячейка памяти. Другой операнд обычно подразумевается st(0). Команды 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 стек.

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

Если в команде присутствует суффикс «p», то происходит выталкивание числа из стека сопроцессора.

Процедуры, работающие с FPU обычно придерживаются следующего соглашения:

На входе и выходе стек 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.

1.6 Стандарт IEE754

Как известно, существует несколько возможных способов представления в машинной памяти множества вещественных чисел. Более того, вплоть до середины 80-х годов существовало не менее нескольких десятков коммерчески значимых архитектур, каждая из которых имела свою реализацию этих способов. И хотя зачастую эти реализации отличались друг от друга лишь параметрами представления чисел, этот факт превращал процесс разработки стандартных программ для вычислений в сущий кошмар для программистов. Такая ситуация сподвигла разработчиков аппаратного обеспечения на выработку единого стандарта на представление вещественных чисел, окончательно принятого в 1985 году и сейчас известного как стандарт IEEE-754 [7].

Стандарт был разработан со следующими целями:

– упростить перенос существующего программного обеспечения на новые платформы, соответствующие данному стандарту;

– предоставить новые возможности, полезные и безопасные, для программистов, которые, не будучи экспертами в численном анализе, могут писать чрезвычайно запутанные программы;

– предоставить возможность экспертам писать и распространять устойчивые и эффективные вычислительные программы, которые были бы хорошо переносимыми между машинами, соответствующими данному стандарту;

– предоставить прямую поддержку диагностики времени исполнения, обработки исключительных ситуаций и интервальной арифметики;

– предоставить возможности для разработки стандартных элементарных функций, высокоточной арифметики и символьных вычислений.

Для этого были определены:

– базовые и расширенные форматы чисел с плавающей точкой;

– операции сложения, вычитания, умножения, деления, квадратного корня, остатка и сравнения;

– преобразования между целочисленными и плавающими форматами;

– преобразования между различными плавающими форматами;

– преобразования между базовыми форматами чисел с плавающей точкой и десятичными строками;

– исключительные ситуации, возникающие при вычислениях с плавающей точкой, и их обработка, включая вычисления с не числами;

За рамками стандарта остались форматы десятичных строк и чисел, интерпретация знака и мантиссы не чисел, и преобразования между двоичным и десятичным представлением чисел в расширенных форматах.

Ниже описаны основные определения стандарта IEE754.

Смещенный порядок (biased exponent) – сумма порядка и константы (смещения), выбранной так, что сумма неотрицательна.

Двоичное число с плавающей точкой (binary floating point number) – битовая строка, характеризующаяся тремя составляющими: знаком (sign), знаковым порядком (signed exponent) и мантиссой (significand). Ее численное значение (если оно существует) равно произведению знака, мантиссы и двойки, возведенной в степень, равную порядку.

Денормализованное число (denormalized number) – ненулевое число с плавающей точкой, порядок которого имеет некоторое зарезервированное значение (обычно это минимальное представимое в данном формате число), и чей ведущий (явный или неявный бит) нулевой.

Место назначения (destination) – то, куда записывается результат бинарной или унарной операции. Может быть явно назначено пользователем или неявно – системой (к примеру, промежуточные результаты вычисления выражений). Некоторые языки не позволяют пользователям распоряжаться промежуточными результатами. Тем не менее, стандарт описывает результат операции в терминах целевого формата (destination’s format) и значений операндов.

Порядок (exponent) – одна из трех составляющих двоичного числа с плавающей точкой, обычно означающая степень двойки при вычислении

значения числа. Иногда порядок называется знаковым (signed) или несмещенным (unbiased).

Дробная часть (fraction) – часть мантиссы, лежащая справа от двоичной точки.

Не-число (NaN) – символическое значение, кодируемое в плавающем формате. Существует два типа не-чисел.

Сигнализирующие не-числа (signaling NaN) генерируют возникновение исключительной ситуации INVALID в случае, когда они встречаются в качестве операндов. «Тихие» не-числа (quiet NaN) могут без всяких исключений существовать на протяжении почти всех вычислительных операций.

Результат (result) – битовая строка (обычно представляющая некоторое число), возвращаемая в место назначения.

Мантисса (significand) – компонента двоичного числа с плавающей точкой, состоящая из явного или неявного ведущего бита слева от подразумеваемой двоичной точки и дробной части справа.

Флаг статуса (status flag) – переменная, которая может принимать два значения. Флаг может быть установленным (set), или неустановленным (clear). Пользователь может очистить флаг, копировать его, или восстановить предыдущее состояние. Будучи установленным, флаг может содержать дополнительную информацию, зависящую от платформы, и, возможно, недоступную некоторым пользователям. Операции, определенные стандартом, могут иметь побочным эффектом выставление следующих флагов: неточный результат (inexact result), исчезновение порядка (underflow), перполнение (overflow), деление на ноль (divide by zero) и неверная операция (invalid operation).

Пользователь (user) – человек, аппаратура или программа, чьи действия не определяются стандартом, имеющий доступ к программному окружению и контролирующий выполнение операций, определяемых стандартом.

Стандарт IEEE-754 на вычисления с плавающей точкой определяет четыре плавающих формата, которые делятся на базовые и расширенные. Уровень соответствия может различаться в зависимости от того, какие форматы реализованы.

1.7 Математический сопроцессор Intel 8087

Intel 8087 – первый математический сопроцессор для линейки процессоров 8086, реализующий архитектуру набора команд x87 и выпущенный в 1980 году компанией Intel.

Сопроцессор 8087 был предназначен для увеличения быстродействия при вычислениях с плавающей точкой за счёт ускорения таких операций как сложение, вычитание, деление и извлечение квадратного корня. Он также мог вычислять трансцендентные функции, например экспоненциальную функцию, логарифмы и тригонометрические функции. Прирост производительности от установки сопроцессора составлял от 20 % до 500 %, в зависимости от специфики задач. Выгода от установки 8087 проявлялась только при выполнении математических операций.

С выпуском фирмой IBM компьютера IBM PC, имевшего сокет для установки сопроцессора, продажи 8087 значительно повысились. Появление сопроцессора привело к созданию стандарта IEEE 754-1985 для арифметики с плавающей точкой. Поздние процессоры Intel, начиная с 80486, имеют встроенный арифметический сопроцессор (за исключением 486SX – для них выпускался сопроцессор 487SX, который можно было не устанавливать).

Ранее компанией Intel выпускались микросхемы 8231 «Арифметического процессора» и 8232 «Процессора операций с плавающей точкой». Они были разработаны для использования с процессором 8080 или его аналогами и использовали 8-битную шину данных. Основной процессор взаимодействовал с ними через инструкции ввода-вывода, либо через контроллер DMA.

Первые шаги в разработке 8087 предпринял Билл Полман – менеджер проекта, контролировавший разработку микропроцессора 8086 в Intel. Он обеспечил поддержку математического сопроцессора, который ещё только предстояло разработать, со стороны 8086.

В 1977 году Полман получил «зелёный свет» на разработку математического сопроцессора 8087. Архитектором был назначен Брюс Ревенел, в качестве помощника архитектора и математика проекта был нанят Джон Палмер. Вместе они разработали новаторскую архитектуру, предусматривавшую использование для промежуточных вычислений 80-битного вещественного числа с 64-битной мантиссой и 16-битной экспонентой, стековую организацию сопроцессора с восемью 80-битными регистрами и набор инструкций, обеспечивающий вычисление большого числа математических функций. 80-

битный формат решал ряд известных трудностей организации вычислений и создания программного обеспечения для числовой обработки: было значительно снижено влияние ошибок округления при работе с 64-битными вещественными операндами, а также обеспечена точность вычислений для 18-значных двоично-десятичных и целых 64-битных чисел. Палмер отмечал, что большое влияние на проект оказали публикации Уильяма Кэхэна по вычислениям с плавающей точкой.

Руководство Intel в Санта-Кларе прохладно отнеслось к проекту 8087 из-за его высоких требований. В конце концов, разработка была передана в израильское отделение компании, а руководителем, ответственным за изготовление микросхемы был назначен Рафи Неф. Палмеру, Ревенелу и Нефу был выдан патент на архитектуру сопроцессора. Роберту Келеру и Джону Бейлису был выдан патент на способ передачи сопроцессору инструкций с определенной битовой комбинацией.

Сопроцессорор 8087 был выпущен в 1980 году и содержал 45000 транзисторов. Он был изготовлен по техпроцессу 3 мкм. Производство Intel 8087 осуществлялось в Малайзии.

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 сопроцессор (который и интересует автора в данном курсовом проекте), можно исполнять разными способами на этом сопроцессоре (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 [8], предоставляемых операционной системой. Это такие функции, как, например printf. В рамках данного курсового проекта некоторые подобные функции будут использованы для корректной работы именно на операционной системе Windows.

2.1 Flat Assembler

Flat assembler – это свободно распространяемый многопроходной ассемблер [9].

Для выбора FASM есть несколько причин. Во-первых, он являлся одним из наиболее динамично развивающихся компиляторов. Его автор Tomasz Grysztar регулярно выкладывает новые версии на свою страницу, 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, 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 Windows API

Windows API [10] – общее наименование набора базовых функций интерфейсов программирования приложений операционных систем семейств Microsoft Windows корпорации Майкрософт. Предоставляет прямой способ взаимодействия приложений пользователя с операционной системой Windows. Для создания программ, использующих Windows API, корпорация Майкрософт выпускает комплект разработчика программного обеспечения, который называется Platform SDK и содержит документацию, набор библиотек, утилит и других инструментальных средств для разработки.

Windows API спроектирован для использования в языке C++ или C для написания прикладных программ, предназначенных для работы под управлением операционной системы MS Windows. Работа через Windows API – это наиболее близкий к операционной системе способ взаимодействия с ней из прикладных программ. Более низкий уровень доступа, необходимый только для драйверов устройств, в текущих версиях Windows предоставляется через Windows Driver Model.

Windows API представляет собой множество функций [11], структур данных и числовых констант, следующих соглашениям языка Си. В то же время конвенция вызова функций отличается от cdecl, принятой для языка C: Windows API использует stdcall (winapi). Все языки программирования, способные вызывать такие функции и оперировать такими типами данных в программах, исполняемых в среде Windows, могут пользоваться этим API. В частности, это языки C++, C#, Pascal, Visual Basic и многие другие.

Для облегчения программирования под Windows в компании Microsoft и сторонними разработчиками было предпринято множество попыток создать библиотеки и среды программирования, частично или полностью скрывающие от программиста особенности Windows API и предоставляющие ту или иную часть его возможностей в более удобном виде. В частности, сама Microsoft в разное время предлагала библиотеки Active Template Library (ATL)/Windows Template Library (WTL), Microsoft Foundation Classes (MFC), .Net/WinForms/WPF, TXLib. Компания Borland (ныне Embarcadero, её преемник в части средств разработки) предлагала OWL и VCL. Есть кроссплатформенные библиотеки, такие как Qt, Tk и многие другие. Весомая часть этих библиотек сконцентрирована на облегчении программирования графического интерфейса пользователя.

Для облегчения переноса на другие платформы программ, написанных с опорой на Windows API, сделана библиотека Wine.

Существуют следующие версии Windows API:

– Win16 – первая версия WinAPI для 16-разрядных версий Windows. Изначально назывался Windows API, позднее был ретроспективно переименован в Win16 для отличия от Win32. Описан в стандарте ECMA-234;

– Win32 – 32-разрядный API для современных версий Windows. Самая популярная ныне версия. Базовые функции реализованы в динамически подключаемых библиотеках kernel32.dll и advapi32.dll; базовые модули графического интерфейса пользователя – в user32.dll и gdi32.dll. Win32 появился вместе с Windows NT и затем был перенесён в несколько ограниченном виде в системы серии Windows 9x. В современных версиях Windows, происходящих от Windows NT, работу Win32 GUI обеспечивают два модуля: csrss.exe (процесс исполнения клиент-сервер), работающий в пользовательском режиме, и win32k.sys в режиме ядра. Работу же системы обеспечивает ядро – ntoskrnl.exe;

– Win32s – подмножество Win32, устанавливаемое на семейство 16-разрядных систем Windows 3.x и реализующее ограниченный набор функций Win32 для этих систем;

– Win64 – 64-разрядная версия Win32, содержащая дополнительные функции Windows на платформах x86-64 и IA-64;

2.4 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 Ассемблера, 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 первого порядка.

Метод Адамса [12] – 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, представленная соотношением 1.

$$y' = f(x, y), y(x_0) = y_0 \quad (1)$$

Для системы необходимо найти решение на сетке с постоянным шагом. В настоящей курсовой работе применяется явный метод Адамса [13], 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 [14] 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 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» [15] 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 – участники команды должны иметь возможность легко разбираться в кусочке приложения, не прилагая усилий, чтобы расшифровать написанную логику.

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

Принцип [16] 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

Принцип KISS [17] подразумевает следующее. 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. Это так называемый стиль «Snake case» [18].

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

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

Реализованы необходимые функции для работы с примитивами:

– вывод целого числа на экран;

– вывод дробного числа на экран с определённым количеством знаков после запятой;

– вывод дробного числа на экран без форматирования;

– вывод строки на экран;

– отдельная функция вывода символа переноса строки для лучшей читаемости кода;

Также написаны функции уже для более конкретных нужд разрабатываемого продукта:

– копирование восьмибайтовых переменных друг в друга;

– вывод массива со значениями аргументов сеточной функции;

– вывод на экран массива со значениями сеточной функции;

– вывод на экран форматированной таблицы готовой сеточной функции

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 [19] для передачи в параметр функции printf из пространства системных функций операционной системы Microsoft Windows;

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

– тексты сообщений, которые могут выводиться на экран во время работы алгоритма и при выводе результатов.

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

– временные переменные, используемые во время вычисления значений производной функции в точке;

– флаг режима вывода отладочной информации;

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

– непосредственно массив со значениями сеточной функции.

Программа представляет из себя консольное окно с отображающимся текстом внутри. Программа считывает значения констант и решает поставленную разработчиком задачу. После компиляции и запуска можно ознакомиться с выведенной информацией – результатом работы алгоритма: шагом сетки, значениями производных и значениями получившейся функции в узлах сетки. Нажав клавишу Enter, можно закрыть окно консоли. Поменяв константы, данные уравнения, откомпилировав и открыв заново, можно получить решение для новых данных.

Для попытки решения выбрано дифференциальное уравнение первого порядка, представленное соотношением 2:

$$y' = f(x, y) = \frac{a \times (1 - y^2)}{(1 + m) \times x^2 + y^2 + 1}, y(0) = 0 \quad (2)$$

Учитывая, что для метода Адамса первого порядка необходима лишь одна известная точка, все начальные данные уже есть. По рекуррентным формулам метода Адамса можно найти все y_i, зная y_{i-1}. Формула задаётся соотношением 3:

$$y_{i+1} = y_i + h \times f(x_{i-1}, y_{i-1}) \quad (3)$$

За один проход можно получить значения функции в узлах. Точность такого решения составляет O(h). То есть она линейно зависит от шага разбиения: чем меньше шаг, тем выше точность.

В приложении Б можно ознакомиться с блок-схемой общего принципа работы алгоритма метода Адамса первого порядка.

Перед началом нахождения сеточной функции необходимо, исходя из количества узлов, определить шаг разбиения или, наоборот, исходя из шага разбиения, определить количество узлов будущей сетки. Предпочтительным является первый вариант.

Эти данные, необходимые для работы алгоритма, будут высчитаны и помещены в переменные, подготовленные заранее для хранения этих свойств.

Перед началом отработки алгоритма необходимо, прибегая к более примитивным методам, которые, возможно, обеспечат не самую лучшую точность просчитать значения сеточной функции в первых M узлах сетки. Учитывая, что был выбран метод Адамса первого порядка (метод Эйлера), начальное значение известно из условия и представляет собой начальное условие задачи.

Таким образом, функционально необходимо спроектировать сущности в коде, которые позволят вычислять значения сначала производной, а затем и функции в конкретном узле.

Очевидно, что основой всему является являться цикл, который заполнит значения массива для узлов сеточной функции. На каждом этапе извлекается предыдущее значение из массива с аргументами и массива со значениями для сеточной функции. Эти два значения, возможно, просто по индексу, передаются в функцию, которая вычислит значение производной. По возвращении в цикл это значение будет использовано для вычисления текущего значения в настоящем узле сеточной функции.

5 АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ

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

Программа разбита на сегменты с данными, кодом и подключаемыми и импортируемыми модулями.

По аналогии с классическими консольными программами на таких языках программирования, как C/C++, Java, C#, программа здесь будет иметь точку входа [20], своеобразную функцию main.

Так основной вычислительный модуль – это математический сопроцессор Intel 8087, который предоставляет возможности Floating Point Unit, необходимо «дождаться» обнаружения сопроцессора соответствующей командой. После этого можно инициализировать модуль для работы с числами с плавающей точкой и приступать непосредственно к вычислениям, используя инструкции FPU.

FPU располагает стековой структурой регистров, что необходимо учитывать. В процессе разработки продукта наблюдалось следующее явление. После 7 операций, затрагивающих регистры FPU стека, следующие операции начинали возвращать некорректные значения. Можно сделать предположение, что каждые 8 операций необходимо «перезагружать» FPU модуль. Вероятно, это число связано с количеством регистров.

Ввиду вышенаписанного при использовании FPU модуля в циклах по работе с вычислениями сначала аргументов сетки, а затем и её значений, на каждой итерации вызывалась команда finit [21], которая перезагружала модуль.

Таким образом удалось обойти проблему перестающего работать FPU модуля. Вероятно, такое решение идёт в ущерб производительности, однако в рамках данной задачи видимых ухудшений не наблюдалось.

Все дробные константы хранятся в восьмибайтном виде в сегменте данных. Инструкциями FPU можно загрузить эти данные в стек и поместить их на верхушку стека. Также можно загрузить и целые числа, которые будут расширены до дробного формата. Имеющийся набор инструкций FPU позволяет производить следующие действия над операндами:

– складывать;

– вычитать;

– перемножать;

– делить;

– вычислять тригонометрические функции;

– производить другие более специфические операции.

Для решения предложенного дифференциального уравнения первого порядка набора этих функций достаточно. Точности хранения данных в регистрах и переменных, а также точности проведения вычислений над этими данными встроенными инструкциями сопроцессора также достаточно в рамках данного курсового проекта.

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 сопроцессора 8087 стало возможным реализовать алгоритм решения задачи, использующий вещественные числа и операции над ними, которые предоставляет модуль Floating Point Unit (FPU) из вышеупомянутого сопроцессора.

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. С помощью аналогичной программы, написанной на языке программирования Python на одинаковых тестовых данных была проверена корректность работы алгоритма. Используемые шестидесятичетырёхбитные ячейки памяти позволяют получить достойную точность в рамках учебного проекта.

Цели, поставленные во введении к курсовому проекту, можно считать достигнутыми.

СПИСОК ИСПОЛЬЗУЕМОЙ ЛИТЕРАТУРЫ

[1] Справочник инструкций для x86, AMD 64 [Электронный ресурс]. – Режим доступа: https://www.felixcloutier.com/x86/. – Дата доступа: 06.12.2022.

[2] Архитектура AMD 64 [Электронный ресурс]. – Режим доступа: https://pvs-studio.com/ru/blog/posts/a0029/. – Дата доступа: 01.12.2022.

[3] Кодирование математических выражений на FPU [Электронный ресурс]. – Режим доступа: https://allasm.ru/proc_02.php#5. – Дата доступа: 06.10.2022.

[4] Список инструкций для Floating Point Unit [Электронный ресурс]. – Режим доступа: https://linasm.sourceforge.net/docs/instructions/fpu.php. – Дата доступа: 05.11.2022.

[5] Команды сравнения FPU [Электронный ресурс]. – Режим доступа: https://osinavi.ru/asm/FPUexpansion/5.html. – Дата доступа: 27.11.2022.

[6] Стек регистров сопроцессора [Электронный ресурс]. – Режим доступа: https://redirect.cs.umbc.edu/courses/undergraduate/313/fall04/burt_katz/lectures/Lect12/stack.html. – Дата доступа: 07.10.2022

[7] Стандарт IEE754 [Электронный ресурс]. – Режим доступа: https://www.h-schmidt.net/FloatConverter/IEEE754.html. – Дата доступа: 20.10.2022.

[8] Использование системных функций Windows для вывода чисел из FPU модуля [Электронный ресурс]. – Режим доступа: https://programmersforum.ru/showthread.php?t=126830. – Дата доступа: 06.12.2022.

[9] Основное руководство по Flat Assembler [Электронный ресурс]. – Режим доступа: http://flatassembler.narod.ru/fasm.htm. – Дата доступа: 30.09.2022.

[10] Официальная документация Microsoft [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/cpp/intrinsics/?view=msvc-160. – Дата доступа: 02.11.2022.

[11] Введение в Windows API [Электронный ресурс]. – Режим доступа: https://users.physics.ox.ac.uk/~Steane/cpp_help/winapi_intro.htm. – Дата доступа: 13.10.2022.

[12] Метод Адамса для решения дифференциальных уравнений первого порядка с начальным условием [Электронный ресурс]. – Режим доступа: https://ru.wikipedia.org/wiki/Метод_Адамса. – Дата доступа: 23.11.2022.

[13] Свойства явного и неявного метода Адамса [Электронный ресурс]. – Режим доступа: https://intellect.icu/metod-adamsa-8575. – Дата доступа: 22.11.2022.

[14] Структура погрешности метода Адамса [Электронный ресурс]. – Режим доступа: https://matica.org.ua/metodichki-i-knigi-po-matematike/chislennye-metody-iu-ia-katcman/6-6-interpoliatcionnye-metody-adamsa. – Дата доступа: 08.12.2022.

[15] Понятие форматированного кода [Электронный ресурс]. – Режим доступа: https://habr.com/ru/company/manychat/blog/468953/. – Дата доступа: 30.09.2022.

[16] Принцип проектирования кода программы DRY [Электронный ресурс]. – Режим доступа: https://www.digitalocean.com/community/tutorials/what-is-dry-development. – Дата доступа: 02.10.2022.

[17] Принцип проектирования кода KISS [Электронный ресурс]. – Режим доступа: https://www.interaction-design.org/literature/article/kiss-keep-it-simple-stupid-a-design-principle. – Дата доступа: 01.10.2022.

[18] Змеиный стиль написания кода [Электронный ресурс]. – Режим доступа: https://www.codingem.com/what-is-snake-case/. – Дата доступа: 02.10.2022.

[19] Форматные строки для функции вывода [Электронный ресурс]. – Режим доступа: https://www.cprogramming.com/tutorial/printf-format-strings.html. – Дата доступа: 01.10.2022.

[20] Точка входа ассемблерной программы [Электронный ресурс]. – Режим доступа: https://board.flatassembler.net/topic.php?t=4438. – Дата доступа: 01.11.2022.

[21] Команда инициализации FPU [Электронный ресурс]. – Режим доступа: https://c9x.me/x86/html/file_module_x86_id_97.html. – Дата доступа: 01.12.2022.

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

```
FORMAT PE Console
ENTRY main

INCLUDE 'win32a.inc'
; link to these files on other machine
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
        ; constants for output
        format_string_double db  '%.5f', 0
        format_string_double_unformatted db '%g', 0
        format_string_integer db '%d', 0
        format_string_text db '%s', 0
        whitespace_text db '  ', 0
        format_string_endline db 13, 10, 0
        minus_symbol db '-', 0 ; 0 => string end
        interval_divider_text db '...', 0

        digit_temp dd ?

        ; input constants for equation
        x0 dq 0.0
        y0 dq 0.0

        m dq 1.0
        a dq 0.7

        interval_left_border dq 0.0
        interval_right_border dq 1.0

        ; constants and variables for solving
        number_of_split_segments dd 27 ; works for 2..27
        step dq ?
        x_values dq 24 dup(?)
        y_values dq 21 dup(0.0)
        x_prev dq ?
```

```
        ;some comments output strings
        found_step_message db 'Grid splitting step size: ', 0
        number_of_nodes_message db 'Number of nodes: ', 0
        current_index_message db 'Index = ', 0
        x_values_message db 'Array of X: ', 0
        output_table_message db 'Table of values of the grid function on
the interval', 0
        table_header_text db ' x          y', 0
        exit_program_message db 'Press Enter to close console window or
just close it yourself   ', 0

        ; variables, needed for storing temporary (intermediate) data
        temp_integer dd ?
        temp_double dq ?
        temp_x dq ? ; used in y' = f(x, y)  to take x[i - 1]
        temp_y dq ? ; used in y' = f(x, y) to take x[i]
        temp_numerator dq ? ; for a * (1 - y^2)
        temp_denominator dq ? ; for (1 + m)*x^2 + y^2 + 1
        temp_derivative_output dq ? ; for total value of y' = f(x, y)
        y_previous dq ? ; to put here y[i-1]
        current_y_total dq ?

        counter dd 0

        SHOW_DEBUG_INFO_FLAG dd 0 ; 1 <=> show intermediate results in
computing f, y | 0 <=> do not show

SECTION '.code' code readable executable
        main:
                initial_data_ouput:
                        stdcall print_string, number_of_nodes_message
                        stdcall print_integer, number_of_split_segments
                        stdcall print_endline

                prepare_fpu:
                        fwait ; wait for mathematics co-processor
                        finit ; initialize FPU module

                define_step_size:
```

```
                fld [interval_right_border]
                fld [interval_left_border]
                fsubp
                fild [number_of_split_segments]
                fdivp
                fst [step]

                stdcall print_string, found_step_message
                stdcall print_double_formatted, step
                stdcall print_endline

        initialize_x_values:
                stdcall copy_qword_variable, x0, x_values

                mov ecx, [number_of_split_segments]
                inc ecx ; N ranges <=> N + 1 points

                mov eax, x_values
                add eax, 8
                initialize_current_x:
                        finit
                        ; load x[i - 1] --> temp_double
                        mov edx, eax
                        sub edx, 8
                        stdcall copy_qword_variable, edx,
temp_double
                        fld [temp_double]
                        fld [step]
                        faddp
                        fst [temp_double]

                        stdcall copy_qword_variable, temp_double,
eax

                        add eax, 8  ;  <=> ++index

                        loop  initialize_current_x


        compute_y_values:
```

```
                        mov ecx, [number_of_split_segments]
                        ;inc ecx

                        mov edx, 8

                        compute_current_y:
                                finit
                                mov [temp_integer], edx

                                stdcall compute_derivative_in_point
                                mov eax, y_values
                                add eax, edx
                                sub eax, 8 ; to get [i - 1]
                                stdcall copy_qword_variable, eax,
y_previous

                                fld [step]
                                fld [temp_derivative_output]
                                fmulp
                                fld [y_previous]
                                faddp
                                fst [current_y_total]
                                add eax, 8
                                stdcall copy_qword_variable,
current_y_total, eax
                                add edx, 8

                                loop compute_current_y

                stdcall print_algorithm_full_response


        finish_program:
                stdcall print_string, exit_program_message
                invoke getch
                invoke ExitProcess, 0

proc print_algorithm_full_response
     stdcall print_endline
     stdcall print_string, output_table_message
```

```
     stdcall print_string, whitespace_text
     stdcall print_double_unformatted, interval_left_border
     stdcall print_string, interval_divider_text
     stdcall print_double_unformatted, interval_right_border
     stdcall print_endline
     stdcall print_endline
     stdcall print_result_function_table
     stdcall print_endline
     stdcall print_endline

     ret
endp

proc compute_derivative_in_point
     pusha

     mov ecx, [temp_integer] ; shift / index I of CURRENT x[i] or y[i]
     sub ecx, 8 ; we need x[i - 1] and y[i - 1]

     ; we need x[i - 1] and y[i - 1], so:
     mov eax, x_values
     add eax, ecx ; so now EAX points to x_values[i - 1]

     mov ebx, y_values
     add ebx, ecx ; so now EBX points to y_values[i - 1]

     stdcall copy_qword_variable, eax, temp_x
     stdcall copy_qword_variable, ebx, temp_y

     ; so now we have x and y for computing f(x, y) = y'

     fld1 ; load +1.0

     ; y^2
     fld [temp_y]
     fld [temp_y]
     fmulp

     ; 1 - y^2
     fsubp
```

```
; a * (1 - y^2)
fld [a]
fmulp
fst [temp_numerator]

; 1 + m
fld1
fld [m]
faddp
; x^2
fld [temp_x]
fld [temp_x]
fmulp
; (1+m)*x^2
fmulp
;y^2
fld [temp_y]
fld[temp_y]
fmulp
; (1+m)*x^2 + y^2
faddp
; 1
fld1
; (1+m)*x^2 + y^2 + 1
faddp
fst [temp_denominator]

; final f(x, y)
fld [temp_numerator]
fld [temp_denominator]
fdivp
fst [temp_derivative_output]

; should print debug info ?
mov eax, [SHOW_DEBUG_INFO_FLAG]
cmp eax, 0
je  finish_computing_derrivative

stdcall print_double_formatted, temp_x
```

```
     stdcall print_string, whitespace_text
     stdcall print_double_formatted, temp_y
     stdcall print_string, whitespace_text
     stdcall print_double_formatted, temp_numerator
     stdcall print_string, whitespace_text
     stdcall print_double_formatted, temp_denominator
     stdcall print_string, whitespace_text
     stdcall print_double_formatted, temp_derivative_output
     stdcall print_endline


     finish_computing_derrivative:
        popa

        ret
endp

; prints a floating number, stored in dq variable
; number of signs after dot is in format string in constants
proc print_double_formatted number
     pusha
     mov eax, [number]
     cinvoke printf, format_string_double, DWORD[eax], DWORD[eax + 4]
     popa

     ret
endp

proc print_double_unformatted number
     pusha
     mov eax, [number]
     cinvoke printf, format_string_double_unformatted, DWORD[eax],
DWORD[eax + 4]
     popa

     ret
endp

proc print_integer number ; prints an integer number, stored in dd
variable
```

```
     pusha
     mov eax, [number]
     cinvoke printf, format_string_integer, [eax]
     popa

     ret
endp

proc print_endline ; prints a new line character
     pusha
     cinvoke printf, format_string_endline
     popa
     ret
endp

proc print_string string ; prints passed string
     pusha
     cinvoke printf, format_string_text, [string]
     popa
     ret
endp

proc copy_qword_variable from, to
     pusha

     mov ecx, [from]
     mov edx, [to]

     mov eax, DWORD[ecx]
     mov DWORD[edx], eax
     mov eax, DWORD[ecx + 4]
     mov DWORD[edx + 4], eax

     popa

     ret
endp

proc print_x_values_array
     pusha
```

```
     mov ecx, [number_of_split_segments]
     mov eax, x_values

     print_current:
        stdcall copy_qword_variable, eax, temp_double
        stdcall print_double_formatted, temp_double
        stdcall print_string, whitespace_text
        add eax, 8
        loop print_current

     popa

     ret
endp

proc print_y_values_array
     pusha

     mov ecx, [number_of_split_segments]
     mov eax, y_values

     print_current:
        stdcall copy_qword_variable, eax, temp_double
        stdcall print_double_formatted, temp_double
        stdcall print_string, whitespace_text
        add eax, 8
        loop print_current

     popa

     ret
endp

proc print_result_function_table
     pusha

     stdcall print_string, table_header_text
     stdcall print_endline
```

```
    mov ecx, [number_of_split_segments]
    inc ecx
    mov eax, x_values
    mov ebx, y_values

    print_current:
       ; print x[i]
       stdcall copy_qword_variable, eax, temp_double
       stdcall print_double_formatted, temp_double

       stdcall print_string, whitespace_text
       ; print y[i]
       stdcall copy_qword_variable, ebx, temp_double
       stdcall print_double_formatted, temp_double

       stdcall print_endline

       add eax, 8
       add ebx, 8
       loop print_current

    popa

    ret
endp

SECTION '.idata' data import readable writable
        library kernel, 'kernel32.dll', msvcrt, 'msvcrt.dll'
        import kernel, ExitProcess, 'ExitProcess'
        import msvcrt, printf, 'printf', getch, '_getch'
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Функциональная схема

ПРИЛОЖЕНИЕ В

(обязательное)

Блок схема алгоритма

ПРИЛОЖЕНИЕ Г

(обязательное)

Графический интерфейс

ПРИЛОЖЕНИЕ Д

(обязательное)

Ведомость документов