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**

1. [Введение](file:///C:\Users\User\Desktop\Курсовая_Хасанов_053506%20(1).docx#_heading=h.gjdgxs) 5
2. Архитектура вычислительной системы 4
   1. Структуры, история возможных архитектур 4
   2. Структуры, история возможных архитектур 4
      1. RISC-V 4
      2. Intel Architecture 8
      3. AMD64 10
   3. акпак
3. Платформа программного обеспечения 12
   1. Microsoft Visual Studio Code 10
   2. Turbo Assembler x86 11
   3. MS DOS 12
4. Теоретическое обоснование разработки программного продукта 15
5. Проектирование функциональных возможностей программы 18
   1. ...
   2. ... 20
   3. ... 20
   4. ... 21
   5. ... 22
6. Заключение 24
7. [Список использованных источников](file:///C:\Users\User\Desktop\Курсовая_Хасанов_053506%20(1).docx#_heading=h.1ksv4uv) 25

**ВВЕДЕНИЕ**

Целью выполнения данной курсовой работы ставится попытка решения выбранной математической задачи с использованием со-процессора и его инструкций. Математический пример должен быть подобран таким образом, чтобы архитектура оставалась основной темой, но при этом чтобы пример не являлся слишком примитивным с точки зрения методов численного анализа.

Одними из составляющих данной цели являются:

* Создание программного продукт 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 математической задачи,
* выбор конкретного языка ассемблера,
* реализация решения на выбранной архитектуре с ориентированием на описанные выше цели,
* эмуляция работы реализованного алгоритма.

Создаётся интерес по поводу решения следующих вопросов. Какие математические задачи позволяет решать та или иная архитектура, 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.

**АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ**

**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.

**ИНФОРМАЦИЯ О ВОЗМОЖНЫХ ДЛЯ ВЫБОРА АРХИТЕКТУР**

**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.

**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[4][5].

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.

**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.

**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.

**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. В условиях фактической отдалённости разработчиков прикладного ПО от непосредственно процессора или со-процессоров, взаимодействия с ними с помощью исполняемых инструкций можно сделать выбор с учётом этого фактора. В выбранной среде разработки Microsoft Visual Studio Code с помощью плагина TASM/MASM и виртуального эмулятора MS DOS – DOSBox можно создать среду для эмуляции выполнения инструкций для Instruction Architecture Set 32 (IA32). Для этого можно задействовать компилятор и линкер Turbo (TASM и TLINK) и язык Turbo Assembler /**FLAT ASSEMBLER**. Также наличие у автора хорошей литературы по данной технологии в том числе способствует выбору именно этой архитектуры, этого языка ассемблера.

**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 TASM IDE или Microsoft Visual Studio Code. Эти обе программы способны обеспечить правильное исполнение инструкций, которые автор затребует в рамках написания программы.

Рассмотрим более подробно выбранную среду Visual Studio Code. Причиной выбора служит возможность исполнять в рамках DOSBox, который можно установить расширением в данный редактор, инструкции IA32 на языке ассемблера Turbo Assembler.

**Microsoft Visual Studio Code**

Microsoft Visual Studio Code (VS Code) — текстовый редактор, разработанный Microsoft для Windows, Linux и macOS. Позиционируется как «лёгкий» редактор кода для кроссплатформенной разработки веб- и облачных приложений. Включает в себя отладчик, инструменты для работы с Git[9], подсветку синтаксиса, IntelliSense и средства для рефакторинга. Имеет широкие возможности для кастомизации: пользовательские темы, сочетания клавиш и файлы конфигурации. Распространяется бесплатно, разрабатывается как программное обеспечение с открытым исходным кодом, но готовые сборки распространяются под проприетарной лицензией.

**Turbo Assembler x86**

Turbo Assembler (TASM) — программный пакет компании Borland, предназначенный для разработки программ на языке ассемблера для архитектуры x86. Кроме того, TASM может работать совместно с трансляторами с языков высокого уровня фирмы Borland, такими как Turbo C и Turbo Pascal. Как и прочие программные пакеты серии Turbo, TASM больше не поддерживается.

Но в целом помимо TASM существуют и другие языки ассемблера. Это FASM, NASM, MASM.

В качестве основной операционной системы используется Microsoft Windows, но особый интерес представляет именно ОС, которая “умеет помогать программисту и процессору плотно общаться” – MS DOS.

**MS DOS**

DOS (англ. Disk Operating System — «дисковая операционная система», ДОС) — семейство операционных систем для PC-совместимых компьютеров, происходящих от MS DOS и PC DOS и совместимых с ними[1][2]. До появления Windows 95 системы класса DOS были наиболее популярными операционными системами для PC, несмотря на их ограничения. После широкого распространения Windows и других многозадачных ОС, DOS сохранили ограниченное распространение как системы для низкопроизводительных машин, а также как предустановленные системы для ноутбуков, рассчитанные на дальнейшую замену современной операционной системой. Для запуска программ, рассчитанных на работу с DOS под другими операционными системами существуют эмуляторы, такие как DOSBox.

Существовали другие операционные системы с аббревиатурой DOS в названии, например, Apple DOS, TR-DOS, но они несовместимы с MS-DOS и между собой, потому не относятся к данному понятию.

Аббревиатура DOS впервые использовалась в названии операционной системы DOS/360 (англ.), точнее это было семейство операционных систем, включавших BOS/360 (базовая ОС), TOS/360 (ОС для магнитных лент) и т. д. Позже аббревиатура DOS стала использоваться в названии множества операционных систем, все они были несовместимы между собой и могли иметь различные реализации взаимодействия с пользователем — например, ProDOS для компьютеров Apple IIGS имела встроенную графическую оболочку.

В 1979 году компания SCP стала выпускать комплекты для сборки компьютеров на основе процессора Intel 8086. Не дождавшись, когда Digital Research, владевшая правами на CP/M, портирует эту систему на новый процессор, программисты компании написали собственную ОС. Она получила неофициальное название QDOS (англ. Quick and Dirty OS — быстрая и «сырая» ОС), и в дальнейшем стала продаваться как 86-DOS.

При разработке IBM PC было необходимо быстро разработать для него операционную систему. Изначально предполагалось заказать порт PC/M у Digital Research, но компаниям не удалось договориться об условиях лицензирования. Задачу перепоручили компании Microsoft, где было решено купить права на 86-DOS и адаптировать её для новой машины. Однако программисты Microsoft не справились с задачей в срок, из-за чего вмешались уже программисты компании IBM. Совместными усилиями компаний была рождена операционная система, которая первоначально получила название PC-DOS, однако сторонним клиентам она продавалась под названием MS DOS. В течение года Microsoft заключила множество договоров об OEM-поставке MS DOS на компьютеры 70 различных компаний. В основном это были OEM-версии DOS, адаптированные под конкретные машины, но с начала 90-х Microsoft начала продавать «коробочные» версии MS-DOS 5.0.

Когда в Digital Research осознали, что PC DOS во многом похожа на CP/M, они стали угрожать судебными исками. В IBM решили пойти на компромисс: покупателям разрешено будет выбирать между PC DOS и CP/M-86 (англ.), однако из-за более высокой цены (на $200 больше, чем у PC DOS) покупатели предпочитали разработку IBM/Microsoft.

Чтобы вернуть себе позиции на рынке, Digital Research выпустила несколько операционных систем, совместимых с MS DOS и CP/M-86: Concurrent DOS, FlexOS, DOS Plus и Multiuser DOS, позже — DR-DOS, совместимую только с MS-DOS. В конце концов компанию Digital Research приобрела Novell и её продукты стали выпускаться от её имени.

В самой Microsoft MS-DOS не считали перспективным продуктом, планируя в будущем заменить DOS на более перспективную ОС Xenix. В компании IBM долго не хотели ничем заменять DOS, но после того, как AT&T начала продавать свою версию Unix, было решено начать разработку OS/2. Тем не менее, IBM и Microsoft не смогли договориться о том, будет ли наследником DOS Windows или OS/2, в результате чего пути компаний разошлись, MS-DOS и PC DOS в дальнейшем развивались независимо[11]. Последней версией MS-DOS стала 6.22, после чего она перестала продаваться как отдельный продукт, войдя в состав Windows 95 как её подсистема, в то время как компания IBM позже выпустила PC DOS 2000 (он же PC DOS 7.1), а в качестве OEM развивала её до версии 7.10.

После того, как Microsoft объявила о прекращении разработки MS-DOS, программисты Джим Холл, Пат Виллани и Тим Норман начали разработку аналога, который бы публиковался под свободной лицензией. Этот проект позже получил название FreeDOS.

Несмотря на выход Windows 95 и прекращение поддержки MS DOS компанией Microsoft, программы под DOS продолжали выходить ещё какое-то время. В частности, это были компьютерные игры, которые под DOS могли более полно использовать ресурсы оборудования, а также прикладные программы для слабых машин.

DOS — семейство операционных систем с монолитным ядром, предназначенных для работы с PC-совместимыми компьютерами[2]. В системе отсутствует слой аппаратных абстракций, поддержка большинства устройств, за исключением самых базовых функций, таких как работа с клавиатурой, текстовым дисплеем и файловой системой, возложена на пользовательское ПО.

DOS работают в реальном режиме адресации x86, что позволяет использовать для нужд операционной системы не более 1 МБ адресного пространства, с учётом специальных областей памяти становится доступно не более 640КБ ОЗУ. Доступ к остальной оперативной памяти осуществляется с помощью специальных драйверов, наподобие HIMEM.SYS. Как правило используются только 16-битные команды процессора для сохранения совместимости с 8088.

DOS не предоставляют стандартных средств для осуществления многозадачности, однако существует особый режим, называемый резидентным, когда после завершения программы вся она или её часть остаётся в памяти и может быть вызвана из другого приложения с помощью специального прерывания. Кроме того, в DOS отсутствовала защита памяти и контроль доступа к файлам, что привело к широкому распространению компьютерных вирусов.

Основной способ взаимодействия с пользователем — через ввод команд. Возможно использование программ-оболочек, таких, как Norton Commander, но они не являются частью системы.

**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 высокого уровня. Напрямую с со-процессором взаимодействовать в них, конечно, можно, но в любом случае не так гибко, 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).

Для разработки программного продукта для решения выбранной задачи используется язык ассемблера, компилятор TASM 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

**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.

Будут написаны функции для работы с числами, строками:

* Поиск
* Сравнения
* Максимальный/минимальный элемент
* Другие

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

Далее необходимо реализовать инструменты для работы с математическими примитивами и не только. 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

**АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ**

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

Программа будет 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. 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 будет 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

При именовании функций (процедур), переменных предположительно будет использоваться единый стиль именования сущностей — так называемый snake\_case. 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.

1. DRY — Don't Repeat Yourself or DIE — Duplication Is Evil

Дублирование – это зло!

Принцип призывает Вас не повторяться при написании кода. Всё, что Вы пишете в проекте, должно быть определено только один раз. 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.

2. KISS — keep it short simple / keep it simple, stupid

Порой наиболее правильное решение – это наиболее простая реализация задачи, в которой нет ничего лишнего.

Чем проще код, тем легче в нём разобраться, как вам, так и другим людям, занимающимся его поддержкой. Под простотой подразумевается отказ от использования хитроумных приемов и ненужного усложнения.

3. YAGNI — You ain't gonna need it

Вам это не понадобится!

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

Проще говоря, вам не стоит писать функциональность наперед, если прямо сейчас в ней нет необходимости. Желание написать код на будущее может иметь много неприятных последствий:

Вы тратите драгоценное время, которое могло бы быть использовано на добавление, тестирование и улучшение действительно необходимой функциональности.

Новые функции должны быть протестированы, задокументированы.

Ваши ненужные новые функции могут впоследствии помешать добавить новые, но уже нужные.

Если написанный код в итоге понадобится, он может оказаться неправильным.

Лишний код может в итоге никогда и не понадобиться.

Лишний код усложняет вашу программу (см. KISS).

Добавление новой ненужной функциональности может привести к желанию добавить ещё больше новой лишней функциональности, приводя к эффекту «снежного кома».

4. Комментарии

Поясняйте код при необходимости, где это возможно.

Используйте комментарии для пояснения кода:

* Что он покрывает?
* Какова его задача?
* Почему такое решение используется или предпочтительно?

В то же время не стоит пытаться 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).

Тем не менее, бывают случаи, когда поясняющие комментарии нужны, вне зависимости от того, как выглядит ваш код.

Обычно это происходит, когда вам нужно добавить контекст к неинтуитивному решению.

6. Именование файлов

Придерживайтесь единого стиля именования файлов в проекте. Рекомендуем использовать нижний регистр написания с разделением слов дефисом.

Также удобно разделять файлы на (если иное не указано в тексте задания). Покомпонентная структура папок тоже приемлема (имеется в виду хранение в одной папке всех файлов, относящихся к определённой компоненте).

**СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ**

* 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/>)