**Курсовая работа**

**по дисциплине: «»**

**«»**

2019 г

Оглавление

[Введение 3](#_Toc11857957)

[1 Архитектура электронно-вычислительных машин 5](#_Toc11857958)

[1.1 Зарождение и развитие ЭВМ. 5](#_Toc11857959)

[1.2 Архитектура ЭВМ 7](#_Toc11857960)

[2 Архитектура приложения 9](#_Toc11857961)

[2.1 Ввод данных с клавиатуры. 10](#_Toc11857962)

[2.2 Ввод данных из файла. 10](#_Toc11857963)

[3 Описание реализованных классов 12](#_Toc11857964)

[4 Работа приложения 15](#_Toc11857965)

[Заключение 17](#_Toc11857966)

[Список литературы 18](#_Toc11857967)

[ПРИЛОЖЕНИЕ А. Исходный код программы 19](#_Toc11857968)

[ПРИЛОЖЕНИЕ B. Содержимое командного файла 29](#_Toc11857969)

# Введение

Компьютер – одно из самых значимых изобретений ХХ века. История вычислений уходит глубокими корнями вглубь. Вся хозяйственная деятельность человека так или иначе связана с расчетами – натуральный или товарно-денежный обмен, учёт запасов, делёж добычи и т.д. Потребность в поиске решений всё более и более сложных задач и, как следствие, все более сложных и длительных вычислений, поставила человека перед необходимостью находить способы, изобретать приспособления, которые могли бы ему в этом помочь. В разных странах исторически сложились собственные денежные единицы, меры веса, длины, объёмов и расстояний. Это естественным образом привело к созданию изобретений, помогающих счёту.

К таким изобретениям можно отнести и «абак», специальную доску для вычислений, и «суммирующие машины» Паскаля, Лейбница и других, менее известных учёных, и бухгалтерские «счёты» с логарифмической линейкой. Однако все эти приспособления были механическими, и их несовершенство не позволяло удовлетворять всем требованиям научно-технического прогресса, набравшего полный ход к началу XX века, когда назрела насущная потребность в машинах, способных производить сложные многократно повторяющиеся вычисления. Расчёты требовались и в фундаментальных науках, таких как математика и физика, и в прикладных инженерных дисциплинах. Идея создания электронно-вычислительной машины (ЭВМ) буквально висела в воздухе в те годы.

Кроме того, научно технический прогресс не только выдвигал требования к ускорению вычислений, но и способствовал появлению математического и инженерного аппарата для воплощения их в жизнь. Предпосылки появления ЭВМ сложились на самом деле задолго до создания первой вычислительной машины.

Так, в 1800 году американским изобретателем Т. Эдисоном было открыто явление термоэлектронной эмиссии, что, в свою очередь, стало толчком к созданию в 1904 году диода – прибора с односторонней проводимостью электрического тока. Это открытие сделал физик Дж. Флеминг. Позже был создан триод – еще один вакуумный прибор.

Еще одним открытием, опережающим своё время и послужившим толчком к изобретению ЭВМ, стали правила логики, описанный Дж. Булем в 1884г. Впоследствии, они были названы его именем – «булева алгебра». Суть его правил состояла в том, что алгебраические элементы способны принимать только два значения – ложь (false) или истина (true). Конструирование логических схем основывалось на этой логике.

И, наконец, в 1918 году эта концепция была воплощена «в железе» русским ученым М.А. Бонч-Бруевичем, создавшим электронного реле, которое могло находиться в двух состояниях, условно обозначаемых в качестве нуля и единицы. На базе этого реле был изобретен триггер.

И вот наконец, первая ЭВМ на электронных лампах была построена в 1945г, в США. Основные идеи, по которым долгие годы – вплоть до настоящего времени – развивалась вычислительная техника и о которых пойдёт речь в этой работе, были разработаны американским математиком Джоном фон Нейманом.

Данная работа предназначена для демонстрации работы простейшей ЭВМ, построенной на базе архитектуры фон Неймана. Целью работы является получение практических навыков по проектированию и реализации программной системы на языке высокого уровня C#, а также закрепление изученного материала по теме архитектуры ЭВМ.

Пользовательский интерфейс программы-эмулятора выполнен в виде текстовой консоли, с возможностью чтения исходных данных из файла.

# Архитектура электронно-вычислительных машин

## Зарождение и развитие ЭВМ.

Электронная вычислительная машина – это комплекс технических и программных средств, предназначенных для автоматизации, подготовки и решения задач пользователей. Под пользователем понимают человека, в интересах которого проводится обработка данных на ЭВМ.

Первая большая ЭВМ ЭНИАК была создана в 1945 г. Она имела массу более 50 т, быстродействие несколько сотен операций в секунду, оперативную память емкостью 20 чисел и занимала огромный зал площадью около 100м2.

С тех пор внешний облик и элементная баз многократно менялись, сохраняя при этом схожую архитектуру. Можно выделить несколько поколений в процессе эволюции электронно-вычислительных машин в XX веке – начале XX века.

К ЭВМ первого поколения относятся машины 50-х годов XX века. Их элементной базой были электровакуумные лампы. ЭВМ были весьма громоздкими сооружениями, содержавшими в себе тысячи ламп, занимавшими иногда сотни квадратных метров территории, потреблявшими электроэнергию в сотни киловатт. Эти машины отличались крайне низкой надёжность и были очень требовательны к условиям содержания – например, в помещениях с ЭВМ постоянно поддерживалось повышенное атмосферное давление, во избежание попадания пыли и насекомых.

ЭВМ второго поколения характерны для 60-х годов XX века. Элементной базой к тому времени становятся транзисторы – полупроводниковые приборы, постепенно вытеснявшие громоздкие и ненадёжные электровакуумные лампы. Полупроводниковая элементная база с тех пор легла в основу создания ЭВМ вплоть до настоящего времени.

Машины стали более производительными, компактными, надежными и менее энергоемкими. Большое развитие получили устройства внешней памяти на магнитных носителях: магнитные барабаны, накопители на магнитных лентах.

В этот период стали развиваться языки программирования высокого уровня: ФОРТРАН, АЛГОЛ, КОБОЛ. Составление программы перестало зависеть от конкретной модели машины, сделалось проще, понятнее, доступнее.

Третье поколение ЭВМ принято относить к концу 60-х – началу 70-х годов XX века. Эти машины создавалось на новой полупроводниковой платформе – интегральных схемах, специальных пластинах, на которых стало возможным размещать и полупроводниковые транзисторы, и соединения между ними. В ходе дальнейшей эволюции, вплоть до настоящего времени, количество транзисторов, которое удавалось разместить на единицу площади интегральной схемы, увеличивалось приблизительно вдвое каждый год.

ЭВМ, которые принято относить к четвертому поколению, производились с 1974 по 1982 гг. К этому периоду появились большие интегральные схемы (БИС), где на одном кристалле в несколько слоёв могло размещаться несколько десятков тысяч электрических элементов

Пятое поколение (80-е – 90-е) годы XX века – это вычислительные машины на сверхсложных микропроцессорах с параллельно-векторной структурой, одновременно выполняющих десятки последовательных команд программы;

В этот период также широкое распространение получают компьютерные сети, появляется Интернет.

Многопроцессорные, многоядерные вычислительные машины настоящего времени относят к шестому поколению. Программы и программируемые интегральные схемы всё чаще строятся по принципу нейронных сетей, ЭВМ всё лучше обрабатывает аудиовизуальную информацию, для взаимодействия с человеком разрабатываются нейроинтерфейсы.

## Архитектура ЭВМ

Архитектура электронно-вычислительной машины – это многоуровневая иерархия программно-аппаратных средств, из которых строится ЭВМ. Каждый из уровней допускает много вариантов построения и применения. Конкретная реализация уровней определяет особенности структурного построения ЭВМ.

Основы учения об архитектуре вычислительных машин были заложены Джон фон Нейманом. Совокупность этих принципов породила классическую (фон-неймановскую) архитектуру ЭВМ.

Принципы фон Неймана:

1. Принцип однородности памяти.

Команды и данные вводятся и хранятся в двоичном представлении в одной и той же памяти и внешне неразличимы. Отличить их можно только по способу использования; то есть одно и то же значение в ячейке памяти может использоваться и как данные, и как команда, и как адрес в зависимости лишь от способа обращения к нему. Это позволяет производить над командами те же операции, что и над числами.

1. Принцип адресности.

Структурно основная память состоит из пронумерованных ячеек, причём процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются на единицы информации, называемые словами, и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек – адреса.

1. Принцип программного управления.

Все вычисления, предусмотренные алгоритмом решения задачи, должны быть представлены в виде программы, состоящей из последовательности управляющих слов – команд. Каждая команда предписывает некоторую операцию из набора операций, реализуемых вычислительной машиной. Команды программы хранятся в последовательных ячейках памяти вычислительной машины и выполняются в естественной последовательности, то есть в порядке их положения в программе. При необходимости, с помощью специальных команд, эта последовательность может быть изменена. Решение об изменении порядка выполнения команд программы принимается либо на основании анализа результатов предшествующих вычислений, либо безусловно.

Фон Нейман не только выдвинул основополагающие принципы логического устройства ЭВМ, но и предложил ее структуру:

1. Арифметико-логическое устройство – выполняет логические и арифметические операции.
2. Управляющее устройство – управляет ходом вычислений и работой остальных устройств компьютера.
3. Устройства ввода и вывода, отвечающие за поступление исходных данных в ЭВМ (с перфокарты, магнитной ленты, клавиатуры, из сети или цифрового порта и т.д.) и вывод результатов вычислений во внешний мир (на монитор, принтер или плоттер, в сеть или цифровой порт и т.д.).

Запоминающие устройства обеспечивают хранение исходных и промежуточных данных,

В построенной по схеме фон Неймана ЭВМ происходит последовательное считывание команд из памяти и их выполнение.

# Архитектура приложения

Демонстрационная программа, представленная в данной работе, эмулирует работу электронно-вычислительной машины с архитектурой фон Неймана. Модель ЭВМ в программе включает в себя типовые модули: оперативную память с произвольным доступом, процессор (АЛУ и УУ) и устройство вывода на экран, как показано на рисунке 1.

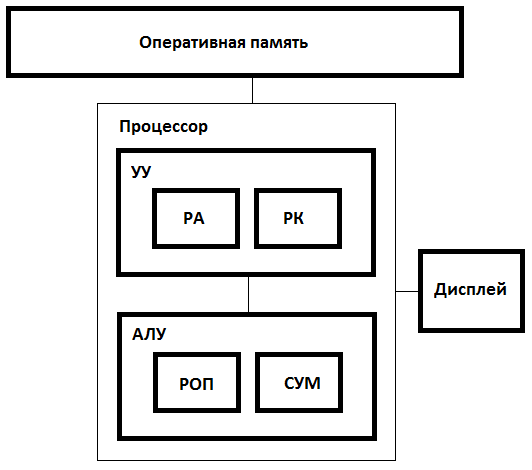


Рисунок 1 – Архитектура эмулируемой ЭВМ

Исходными данными в программе эмулятора является последовательность команд, которые можно поделить на три категории – манипуляция с «оперативной памятью» (запись и чтение данных), команда вывода на экран текущего значения сумматора и команды арифметических операций (сложение и вычитание).

Список поддерживаемых команд:

**output** – команда печати на экране текущего значения сумматора АЛУ;

**end** – команда завершения работы текущей программы ЭВМ;

**add** – команда сложения сумматора с регистром операнда;

**sub** – команда вычитания операнда из текущего значения сумматора;

**ld**, – команда загрузки в сумматор текущего значения регистра операнда;

**mv** – команда записи значения в оперативную память.

Ввод данных осуществляется следующим образом: в оперативную память и регистр АЛУ заносятся данные с помощью команд **mv** и **ld**, затем указывается последовательность требуемых арифметических операций. Завершается программа командой **end**.

Формат любой команды следующий: в начале идёт код команды, занимающий 1 байт, затем следует необязательное 4-байтное слово, представляющее собой адрес операнда в оперативной памяти. Наличие операнда зависит от типа команды. Команды **output** и **end** состоят только из 1-байтного кода, все остальные команды предполагают обязательное наличие операнда. Программа поддерживает операции только с целочисленными типами данных размером 1, 2 и 4 байта.

## Ввод данных с клавиатуры.

По умолчанию, если запустить программу без параметров, она будет принимать произвольный список команд в качестве пользовательского ввода с клавиатуры. Каждая введённая строка интерпретируется как очередная команда. Для того, чтобы ввести команду в программу, необходимо набрать её на клавиатуре и нажать Enter. После этого она будет обработана, а результат, при наличии такового, будет выведен на экран.

Завершить сеанс работы можно, либо введя команду END, либо нажав стандартное сочетание клавиш Ctrl + C.

## Ввод данных из файла.

Исходными данными в этом режиме является текстовый файл, содержащий предварительно заданные команды и, возможно, комментарии к ним. Для активации этого режима, программу необходимо запустить из командной строки с единственным аргументом – именем файла. После запуска и инициализации переменных, программа считывает содержимое файла построчно, интерпретируя каждую непустую строку, не являющуюся комментарием, как отдельную команду – так же, как если бы она была введена с клавиатуры.

# Описание реализованных классов

Исходный код приложения состоит из нескольких классов – стандартного класса **Program**, класса **Computer**, управляющего эмуляцией работы электронно-вычислительной машины, и семейства классов, представляющих собой абстракцию типовых компонентов ЭВМ. Листинг программы с комментариями приведён в Приложении А.

Класс **Program** – главный класс программы, «скелет» которого генерируется при создании пустого проекта. Этот класс содержит точку входа приложения – статическую функцию **Main**, с которой всегда начинается выполнение программы. Исходный код этого класс содержится одноимённом файле Program.cs.

Функция **Main** занимается анализом аргументов командной строки и, в зависимости от его результатов, организует получение исходных данных либо из файла – если первым аргументом передано его имя, либо непосредственно от пользователя, с клавиатуры – если программа запущена без аргументов.

Полученные так или иначе исходные данные интерпретируются в качестве последовательности строк, каждая из которых представляет собой команду ЭВМ, и передаются для обработки методу класса **addCommand** класса **Computer**. Эта функция предварительно анализирует тип команды, классифицирует её, подготавливает аргументы, если таковые имеются, и записывает в буфер «оперативной памяти».

Класс **Computer** управляет набором классов, эмулирующих работу типовых компонентов ЭВМ фон Неймана – «Оперативная память», «Арифметико-логическое устройство», «Устройство управления» и «Дисплей».

Класс **RAM** (Random Access Memory – память с произвольным доступом) является абстракцией оперативной памяти ЭВМ посредством динамического массива, имеющего тип **char[]**. Максимальный размер «оперативной памяти» в демонстрационной программе выбран равным 65536 байт. Класс предоставляет функции произвольного и последовательного доступа к «памяти».

Функции чтения данных:

* public char getByte(int addr) – считывает один байт по указанному адресу;
* public char getNextByte() – считывает один байт из текущей позиции чтения и смещает эту позицию на байт;
* public short getWord(int addr) – считывает одно 2-байтное слово по указанному адресу;
* public short getNextWord() – считывает одно 2-байтное слово из текущей позиции чтения и смещает эту позицию на 2 байта;
* public int getDword(int addr) – считывает одно 4-байтное слово по указанному адресу;
* public int getNextDword() – считывает одно 4-байтное слово из текущей позиции чтения и смещает эту позицию на 4 байта.

Функции записи данных:

* public void setByte(int addr, char value) – записывает один байт по указанному адресу;
* public void setNextByte(char value) – записывает один байт из текущей позиции записи и смещает эту позицию на байт;
* public void setWord(int addr, short value) – записывает одно 2-байтное слово по указанному адресу;
* public void setNextWord(short value) – записывает одно 2-байтное слово из текущей позиции записи и смещает эту позицию на 2 байта;
* public void setDword(int addr, int value) – записывает одно 4-байтное слово по указанному адресу;
* public void setNextDword(int value) – записывает одно 4-байтное слово из текущей позиции записи и смещает эту позицию на 4 байта.

Кроме того, публичные свойства WriteAddr и ReadAddr позволяют установить и получить текущие позиции записи и чтения, соответственно.

Класс **ALU** (Arithmetic and Logic Unit) – абстракция арифметико-логического устройства в процессоре ЭВМ. Содержит два целочисленных регистра, представленных открытыми свойствами Reg и Sum – сумматор и регистр операнда, а также функцию execute, выполняющую текущую операцию.

Класс **CU** (Control Unit) – абстракция управляющего устройства в процессоре ЭВМ. Содержит два целочисленных регистра – адреса и команд, представленных открытыми свойствами RA и RC.

Класс **Display**, как и следует из его название, служит для вывода информации на экран компьютера и представляет собой обёртку над методом WriteLine стандартного класса System.Console.

# Работа приложения

Скомпилированная программа представляет собой исполняемый файл proc.exe и запускается на выполнение из командной строки, из проводника Windows – непосредственно через этот файл или через пиктограмму ярлыка, который на него ссылается, или непосредственно из интегрированной среды разработки Visual Studio.

В ходе тестирования приложения рассмотрено большое количество различных наборов входных данных, включая содержащие преднамеренную ошибку.

Для удобства тестирования создан командный файл, содержащий список команд и поясняющие комментарии. В ходе тестирования также убедимся, что программа работает с исходными данными с клавиатуры точно так же, как и при получении их из файла. Содержимое тестового командного файла – test.op – приведено в Приложении Б. Результат выполнения командного файла эмулятором приведён на рисунке 2.

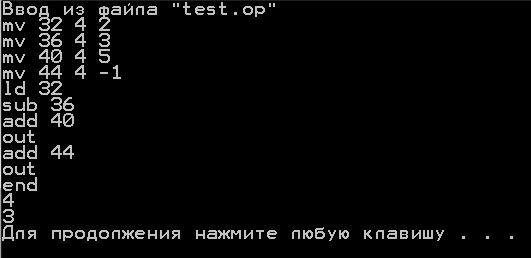


Рисунок 2 – Тестирование загрузки заданных команд из файла

Как видно из рисунка, в случае загрузки команд из файла, ЭВМ для удобства пользователя дублирует на экране все прочитанные команды, как если бы они были введены с клавиатуры.

Второй тест проверяет ввода команд с клавиатуры, результаты приведены на рисунке 3.

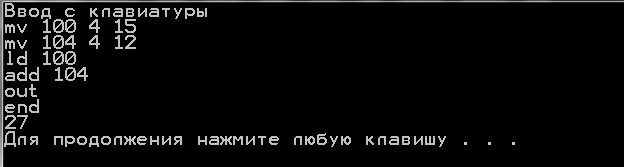


Рисунок 3. Тестирование ввода произвольных команд с клавиатуры

Первая команда записывает значение 15 (4-байтное слово) в оперативную память по адресу 100. Вторая команда записывает значение 12 (4-байтное слово) в оперативную память по адресу 104. Далее, команда **ld** указывает необходимость загрузки в сумматор значения, содержащегося в ОП по адресу 100. Команда **add** определяет сложение текущего значения сумматора с значением ячейки ОП с адресом 104. Завершают нашу «микропрограмму» команды вывода итогового значения сумматора на экран и выхода. Команда end, помимо того что записывается в память ЭВМ в конце списка операций, завершает также ввод данных, после которого вычислительная машина запустит программу на выполнение. Таким образом, после ввода заключительной команды и нажатия Enter мы увидим вывод на экран результатов работы ЭВМ (если, как в данном случае, программа содержит команду end) и сама программа-эмулятор завершить работу.

Как видно из рисунка 3, в результате выполнения микропрограммы произведена операция сложения целых чисел 15 и 12, результат равен 27.

# Заключение

В результате работы была спроектирована и разработана программа, эмулирующая работу простейшей электронно-вычислительной машины, построенной на базе архитектуры фон Неймана, позволяющая обрабатывать загружать команды и данные в «память» машины и запускать выполнение загруженной «программы».

В ходе работы над проектом получены практические навыки разработки программ на языке высокого уровня C#. Изучены особенности создания приложений с помощью этого языка программирования и среды разработки Microsoft Visual Studio. Кроме того, усвоены теоретические основы построения электронно-вычислительных машин и, в частности, получившая широчайшее распространение архитектура, предложенная Джоном фон Нейманом.

В целом, на базе практического применения, закреплены базовые знания, полученные ранее, при изучении теории.

# Список литературы

1. Атовмян И.О. Архитектура вычислительных систем. М.: МИФИ, 2002.

2. Гуров В.В., Чуканов В.О. Электронная книга. Архитектура и организация ЭВМ. М.: ИНТУИТ. Национальный открытый университет, 2005.

3. Гордеев А.В. Операционные системы: Учебник для вузов. 2-е изд. СПб.: Питер, 2007.

4. Жмакин А.П. Архитектура ЭВМ. 2-е изд. СПб.: БХВ-Петербург, 2010.

5. Бородакий, Ю. В. Эволюция информационных систем / Ю.В. Бородакий, Ю.Г. Лободинский. – Москва:СИНТЕГ, 2011. – 368 c.

6. Вагнер, Билл С# Эффективное программирование / Билл Вагнер. – М.: ЛОРИ, 2013. - 320 c.

7. Абрамян, Михаил Visual C# на примерах / Михаил Абрамян. – М.: БХВ-Петербург, 2015. – 572 c.

8. Культин, Н. С# в задачах и примерах / Н. Культин. – М.: БХВ-Петербург, 2016. – 952 c.

9. Ишкова, Э. А. Самоучитель С#. Начала программирования / Э.А. Ишкова. – М.: Наука и техника, 2017. – 496 c

10. ГОСТ 2.105-95 Общие требования к текстовым документам.

# ПРИЛОЖЕНИЕ А. Исходный код программы

**Program.cs**

using System;

using System.IO;

namespace proc

{

class Program

{

// Ссылка на класс вычислительной машины

static Computer computer = new Computer();

// Основная функция (точка входа) приложения

static void Main(string[] args)

{

// Загрузка программы

try

{

// будем записывать команды в начало памяти

computer.prepareLoad();

// Проверяем наличие аргументов командной строки

if (args.Length > 0)

{

// Первый аргумент считаем именем файла,

// считываем команды из него

string fileName = args[0];

Console.WriteLine("Ввод из файла \"" + fileName + "\"");

// Открываем файл на чтение

StreamReader reader = new StreamReader(fileName);

// Читаем строки до конца файла

while (true)

{

string cmd = reader.ReadLine();

if (cmd != null)

{

// Каждую прочитанную строку печатаем

// и отправляем в калькулятор

if (computer.addCommand(cmd, true))

{

break;

}

}

else

{

break;

}

}

// Закрываем файл

reader.Close();

}

else

{

// Программа запущена без параментров,

// считываем команды из консоли

Console.WriteLine("Ввод с клавиатуры");

while (true)

{

// Каждую прочитанную строку передаём

// калькулятору без вывода на экран

if (computer.addCommand(Console.ReadLine(), false))

{

break;

}

}

}

}

catch (System.Exception exception)

{

Console.WriteLine("Ошибка: " + exception.Message);

}

computer.runProgram();

}

}

}

**Computer.cs**

using System;

namespace proc

{

public enum Command

{

Output, // вывод

Stop, // останов

Add, // сложение

Sub, // вычитание

Ld, // загрузка

Mv, // выгрузка в ОП

}

class Computer

{

// Арифметико-логическое устройство

ALU alu = new ALU();

// Устройство управления

CU cu = new CU();

// Оперативная память

RAM ram = new RAM();

// Устройство вывода

Display display = new Display();

// Выполнение программы

public void runProgram()

{

// Загружаем в УУ адрес текущей команды

cu.RA = 0;

// Загружаем в УУ код текущей команды

cu.RС = (Command)ram.getByte(cu.RA);

while(cu.RС != Command.Stop)

{

if(hasOperand(cu.RС))

{

// Извлекаем адрес операнда текущей команды

int addr = ram.getWord(cu.RA + 1);

// Читаем из памяти значение операнда и загружаем его в АЛУ

alu.Reg = ram.getDword(addr);

}

// Выполняем текущую команду

if (cu.RС == Command.Output)

{

display.print(alu.Sum.ToString());

}

else

{

alu.execute(cu.RС);

}

// Смещаемся на адрес следующей команды

cu.RA += commandLength(cu.RС);

// Загружаем код следующей команды в АЛУ

cu.RС = (Command)ram.getByte(cu.RA);

}

}

// Признак наличия операнда у команды

bool hasOperand(Command cmd)

{

if (cmd == Command.Ld || cmd == Command.Add || cmd == Command.Sub)

return true;

return false;

}

// Полный размер команды

int commandLength(Command cmd)

{

if (cmd == Command.Output || cmd == Command.Stop)

return 1;

return 5;

}

// Подготовка к рагрузке в память

public void prepareLoad()

{

ram.WriteAddr = 0;

}

// Функция ввода отдельной команды

// Возвращаемое значение:

// false - продолжаем ввод команд

// true - завершаем работу

public bool addCommand(string cmd, bool trace)

{

// Комментарии и пустые строки пропускаем

if (cmd.Length == 0 || cmd[0] == '#')

{

return false;

}

// Выводим исходную команду на консоль

// (при вводе с клавиатуры не имеет смысла,

// т.к. пользователь уже её сам набрал в консоли)

if (trace)

{

Console.WriteLine(cmd);

}

// Переводим все символы в верхний регистр

// (убираем зависимость от регистра)

cmd = cmd.Trim().ToUpper();

// Разбиваем команду на имя команды

// и аргументы (если есть)

string[] tokens = cmd.Split(new char[] { ' ' });

// Первый элемент обязан содержать имя команды

if (tokens.Length == 0)

{

return false;

}

if (tokens[0] == "MV")

{

/\* Задать значение параметра

ожидаем: MV <адрес ОП> <размер> <значение>

примеры

MV 1024 1 16

MV 1025 2 821

MV 1027 4 223501

\*/

if (tokens.Length == 4)

{

int addr = int.Parse(tokens[1]);

int size = int.Parse(tokens[2]);

int value = int.Parse(tokens[3]);

switch (size)

{

case 1:

ram.setByte(addr, (char)value);

break;

case 2:

ram.setWord(addr, (short)value);

break;

case 4:

ram.setDword(addr, value);

break;

}

}

}

else if (tokens[0] == "LD")

{

// Загрузить слово из памяти в сумматор АЛУ

// ожидаем: LD <адрес ОП>

if (tokens.Length == 2)

{

int addr = int.Parse(tokens[1]);

ram.setNextByte((char)Command.Ld);

ram.setNextDword(addr);

}

}

else if (tokens[0] == "ADD")

{

// Загрузить слово из памяти в сумматор АЛУ

// ожидаем: ADD <адрес ОП>

if (tokens.Length == 2)

{

int addr = int.Parse(tokens[1]);

ram.setNextByte((char)Command.Add);

ram.setNextDword(addr);

}

}

else if (tokens[0] == "SUB")

{

// Загрузить слово из памяти в сумматор АЛУ

// ожидаем: ADD <адрес ОП>

if (tokens.Length == 2)

{

int addr = int.Parse(tokens[1]);

ram.setNextByte((char)Command.Sub);

ram.setNextDword(addr);

}

}

else if (cmd == "OUT")

{

// Отправляем значение сумматора в устройство вывода

ram.setNextByte((char)Command.Output);

}

else if (tokens[0] == "DUMP")

{

// Образ памяти для отладки

int size = 16;

if (tokens.Length > 1)

size = int.Parse(tokens[1]);

ram.dump(size);

}

else if (cmd == "END")

{

// Записываем команду: останов

ram.setNextByte((char)Command.Stop);

// Возвращаем признак завершения работы

return true;

}

else

{

// Сообщаем об ошибке, но не прерываем работу

Console.WriteLine("Неизвестная команда: \"" + cmd + "\"");

}

// Возвращаем признак продолжения работы

return false;

}

}

}

**ALU.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace proc

{

// Арифметико-логическое устройство (ALU - Arihmetic-Logic Unit)

class ALU

{

// Регистр текущего операнда

public int Reg

{

get

{

return \_reg;

}

set

{

\_reg = value;

}

}

// Регистр сумматора

public int Sum

{

get

{

return \_sum;

}

set

{

\_sum = value;

}

}

// Выполнить текущую операцию

public void execute(Command op)

{

switch (op)

{

case Command.Add:

\_sum += \_reg;

break;

case Command.Sub:

\_sum -= \_reg;

break;

case Command.Ld:

\_sum = \_reg;

break;

}

}

// Закрытые члены класса

int \_reg = 0;

int \_sum = 0;

}

}

**CU.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace proc

{

// Управляющее устройство (CU - Control Unit)

class CU

{

// Регистр адреса

public int RA { get; set; } = 0;

// Регистр команд

public Command RС { get; set; } = 0;

}

}

**RAM.cs**

using System;

using System.Collections.Generic;

using System.Text;

namespace proc

{

// Оперативная память (Random Access Memory)

class RAM

{

// Максимальный размер памяти

const int size = 65536;

// Массив байт (буфер памяти)

char[] mem = new char[size];

// Текущий адрес записи

public int WriteAddr { get; set; } = 0;

// Текущий адрес чтения

public int ReadAddr { get; set; } = 0;

// Операции чтения данных

public char getByte(int addr)

{

return mem[addr];

}

public char getNextByte()

{

return mem[ReadAddr++];

}

public short getWord(int addr)

{

short result = (short)mem[addr];

result |= (short)(mem[addr + 1] << 8);

return result;

}

public short getNextWord()

{

short result = (short)mem[ReadAddr++];

result |= (short)(mem[ReadAddr++] << 8);

return result;

}

public int getDword(int addr)

{

int result = (int)mem[addr];

result |= (mem[addr + 1] << 8);

result |= (mem[addr + 2] << 16);

result |= (mem[addr + 3] << 24);

return result;

}

public int getNextDword()

{

int result = (int)mem[ReadAddr++];

result |= (mem[ReadAddr++] << 8);

result |= (mem[ReadAddr++] << 16);

result |= (mem[ReadAddr++] << 24);

return result;

}

// Операции записи данных

public void setByte(int addr, char value)

{

mem[addr] = value;

}

public void setNextByte(char value)

{

mem[WriteAddr++] = value;

}

public void setWord(int addr, short value)

{

mem[addr + 0] = (char)(value >> 0);

mem[addr + 1] = (char)(value >> 8);

}

public void setNextWord(short value)

{

mem[WriteAddr++] = (char)(value >> 0);

mem[WriteAddr++] = (char)(value >> 8);

}

public void setDword(int addr, int value)

{

mem[addr + 0] = (char)(value >> 0);

mem[addr + 1] = (char)(value >> 8);

mem[addr + 2] = (char)(value >> 16);

mem[addr + 3] = (char)(value >> 24);

}

public void setNextDword(int value)

{

mem[WriteAddr++] = (char)(value >> 0);

mem[WriteAddr++] = (char)(value >> 8);

mem[WriteAddr++] = (char)(value >> 16);

mem[WriteAddr++] = (char)(value >> 24);

}

// Образ памяти для отладки

public void dump(int size)

{

Console.WriteLine("------------ MEMORY DUMP ------------");

for (int i = 0; i < size; i++)

{

int cell = (int)mem[i];

Console.WriteLine(i.ToString("X4") + " " + cell.ToString("X"));

}

Console.WriteLine("-------------------------------------");

}

}

}

**Display.cs**

using System;

namespace proc

{

class Display

{

public void print(string result)

{

Console.WriteLine(result);

}

}

}

# ПРИЛОЖЕНИЕ B. Содержимое командного файла

**test.op**

#simple arithmetic operations

mv 32 4 2

mv 36 4 3

mv 40 4 5

mv 44 4 -1

#for debug only

#dump 50

ld 32

sub 36

add 40

# 2 - 3 + 5 = 4

out

add 44

# 2 - 3 + 5 + (-1) = 3

out

end