Оглавление

[1 Цель работы 3](#_Toc513404310)

[2 Постановка задачи 4](#_Toc513404311)

[3 Краткие теоретические сведения 5](#_Toc513404312)

[3.1 Понятие сопроцессора 5](#_Toc513404313)

[3.2 Сопроцессорные конфигурации 6](#_Toc513404314)

[3.3 Программная модель сопроцессора 6](#_Toc513404315)

[3.4 Квадратное уравнение 7](#_Toc513404316)

[4 Описание работы программы 10](#_Toc513404317)

[5 Демонстрация работы программы 12](#_Toc513404318)

[6 Вывод 16](#_Toc513404319)

[7 Приложение. Код программы 17](#_Toc513404320)

# Цель работы

Создать программу, эмулирующую работу процессора с вещественными числами, а также способную с помощью математического сопроцессора решать задачу решения квадратного уравнения. Изучить принципы решения математических задач с помощью сопроцессора.

# Постановка задачи

1. Написать программу, находящую решение квадратного уравнения с помощью сопроцессора.

# Краткие теоретические сведения

## Понятие сопроцессора

Сопроцессор — специализированный процессор, расширяющий возможности центрального процессора компьютерной системы[1], но оформленный как отдельный функциональный модуль. Физически сопроцессор может быть отдельной микросхемой или может быть встроен в центральный процессор (как это делается в случае математического сопроцессора в процессорах для ПК начиная с Intel 486DX).

Различают следующие виды сопроцессоров:

* математические сопроцессоры общего назначения, обычно ускоряющие вычисления с плавающей запятой,
* сопроцессоры ввода-вывода (например — Intel 8089), разгружающие центральный процессор от контроля за операциями ввода-вывода или расширяющие стандартное адресное пространство процессора,
* сопроцессоры для выполнения каких-либо узкоспециализированных вычислений.

Сопроцессоры могут входить в набор логики, разработанный одной конкретной фирмой (например, Intel выпускала для процессоров 8086 и 8088 сопроцессоры 8087 и 8089, Motorola — сопроцессор Motorola 68881) или выпускаться сторонним производителем (например, Weitek (англ.) 1064 для Motorola m68k и 1067 для Intel 80286).

Сопроцессор в программировании

Сопроцессор расширяет систему инструкций центрального процессора, поэтому для его использования программа (компилируемая без интерпретации и вызова внешних библиотек) должна содержать эти инструкции. Настройки современных компиляторов для языков высокого уровня под процессоры семейства x86 зачастую позволяют выбирать: использовать математический сопроцессор или нет, что особенно важно при создании кода, который будет исполняться внутри обработчика аппаратного прерывания.

Также бывают периферийные процессоры, предназначенные для управления периферийными устройствами и разгрузки центрального процессора, а именно:

* звуковые процессоры
* графические процессоры

## Сопроцессорные конфигурации

Использование сопроцессора  позволяет значительно ускорить работу программ, выполняющих расчеты с высокой точностью, тригонометрические вычисления и обработку информации, которая должна быть представлена в виде действительных чисел. Сопроцессор подключается к системной шине параллельно с центральным процессором (CPU) и может работать только совместно с ним. Все команды попадают в оба процессора, а выполняет каждый свои. Сопроцессор не имеет своей программы и не может осуществлять выборку команд и данных. Это делает центральный процессор. Сопроцессор перехватывает с шины данные и после этого реализует конкретные действия по выполнению команды. Два процессора работают параллельно, что повышает эффективность системы.  Но возникают ситуации, когда их работа требует синхронизации (из-за разницы во времени выполнения команд).

**Синхронизация по командам**. Когда  центральный процессор выбирает для выполнения команду FPU, последний может быть занят выполнением предыдущей команды. Поэтому перед каждой командой сопроцессора в программе должна стоять специальная команда (wait), которая только проверяет текущее состояние FPU и, если он занят, переводит центральный процессор в состояние ожидания. Соответствующую команду в программу может вводить либо ассемблер, либо компилятор языка без указаний программиста.

**Синхронизация по данным**. Если выполняемая в  FPU команда записывает операнд в память перед последующей командой СРU, которая обращается к этой ячейке памяти, требуется команда проверки состояния FPU. Если данные еще не были записаны, СPU должен переходить в состояние ожидания. Автоматически учесть такие ситуации довольно сложно, поэтому вводить команды, которые проверяют состояние сопроцессора и при необходимости заставляют центральный процессор ожидать, должен программист.

## Программная модель сопроцессора

В программную модель  любого процессора включаются только те регистры, которые доступны программисту на уровне машинных команд. Основу программной модели FPU образует регистровый стек из восьми 80-битных регистров R0-R7. В них хранятся числа в вещественном формате. В любой момент времени 3-битное поле ST  в слове состояния определяет регистр, являющийся текущей вершиной стека и обозначаемый ST(0). При занесении в стек (push) осуществляется декремент поля  ST и загружаются данные в новую вершину стека. При извлечении из стека (pop) в получатель, которым чаще всего является память, передается содержимое вершины стека, а затем инкрементируется поле  ST .

В организации регистрового стека FPU есть отличия от классического стека.

1.  Стек имеет кольцевую структуру. Контроль за использованием стека должен осуществлять программист. Максимальное число занесений без промежуточных извлечений равно 8.

2.    В командах FPU допускается явное или неявное обращение к регистрам с модификацией или без поля  ST. Например, команда fsqrt  замещает число из вершины стека значением корня из него. В бинарных операциях допускается явное указание регистров. Адресация осуществляется относительно текущей вершины стека и обозначение ST (i)  0<i<7 , считая от вершины.

   3. Не все стековые команды автоматически модифицируют указатель вершины стека.

С каждым регистром стека ассоциируется 2-битный тег (признак), совокупность которых образует слово тегов. Тег регистра R0 находиться в младших битах, R7 – в старших. Тег фиксирует наличие в регистре действительного числа (код 00), истинного нуля (код 01), ненормализованного или бесконечности (код 10) и отсутствие данных (код 11).  Наличие тегов позволяет  FPU быстрее обнаруживать особые случаи (попытка загрузить в непустой регистр, попытка извлечь из пустого) и обрабатывать данные.

Остальными регистрами FPU являются регистр управления, регистр состояния, два регистра состояния команды и два регистра указателя данных. Длина их всех 16 бит.

## Квадратное уравнение

*Квадратное уравнение — это уравнение вида ax2 + bx + c = 0, где коэффициенты a, b и c — произвольные числа (.*

Прежде, чем изучать конкретные методы решения, заметим, что все квадратные уравнения можно условно разделить на четыре класса:

1. Не имеют корней;
2. Имеют ровно один корень;
3. Имеют два различных корня.
4. Имеют бесконечное множество корней.

В этом состоит важное отличие квадратных уравнений от линейных, где корень всегда существует и единственен. Как определить, сколько корней имеет уравнение? Для этого существует замечательная вещь — **дискриминант**.

**Дискриминант**

Пусть дано квадратное уравнение *ax*2 + *bx +c* = 0. Тогда дискриминант вычисляется по формуле .

**Корни квадратного уравнения**

Теперь перейдем, собственно, к решению. Если дискриминант *D* > 0, корни можно найти по формулам:

Формула корней квадратного уравнения

Основная формула корней квадратного уравнения

Когда *D* = 0, можно использовать любую из этих формул — получится одно и то же число, которое и будет ответом. Наконец, если *D* < 0, квадратное уравнение не имеет решений в действительных числах. В случае, когда , уравнение имеет бесконечно много решений.

**Неполные квадратные уравнения**

Бывает, что квадратное уравнение несколько отличается от того, что дано в определении. Например,

1. *x*2 + 9*x* = 0;
2. *x*2 − 16 = 0.

Несложно заметить, что в этих уравнениях отсутствует одно из слагаемых. Такие квадратные уравнения решаются даже легче, чем стандартные: в них даже не потребуется считать дискриминант. Итак, введем новое понятие:

Уравнение *ax*2 + *bx +c* = 0 называется неполным квадратным уравнением, если b = 0 или c = 0, т.е. коэффициент при переменной x или свободный элемент равен нулю.

Разумеется, возможен совсем тяжелый случай, когда оба этих коэффициента равны нулю: *b* = *c* = 0. В этом случае уравнение принимает вид a*x*2 = 0. Очевидно, такое уравнение имеет единственный корень: *x* = 0.

Рассмотрим остальные случаи. Пусть *b* = 0, тогда получим неполное квадратное уравнение вида *ax*2 + *c* = 0. Немного преобразуем его:

Решение неполного квадратного уравнения

Решение неполного квадратного уравнения

Поскольку арифметический квадратный корень существует только из неотрицательного числа, последнее равенство имеет смысл исключительно при (−*c*/*a*) ≥ 0. Вывод:

1. Если в неполном квадратном уравнении вида *ax*2 + *c* = 0 выполнено неравенство (−*c*/*a*) ≥ 0, корней будет два. Формула дана выше;
2. Если же (−*c*/*a*) < 0, корней нет.

Теперь разберемся с уравнениями вида *ax*2 + *bx* = 0, в которых свободный элемент равен нулю. Тут все просто: корней всегда будет два. Достаточно разложить многочлен на множители:

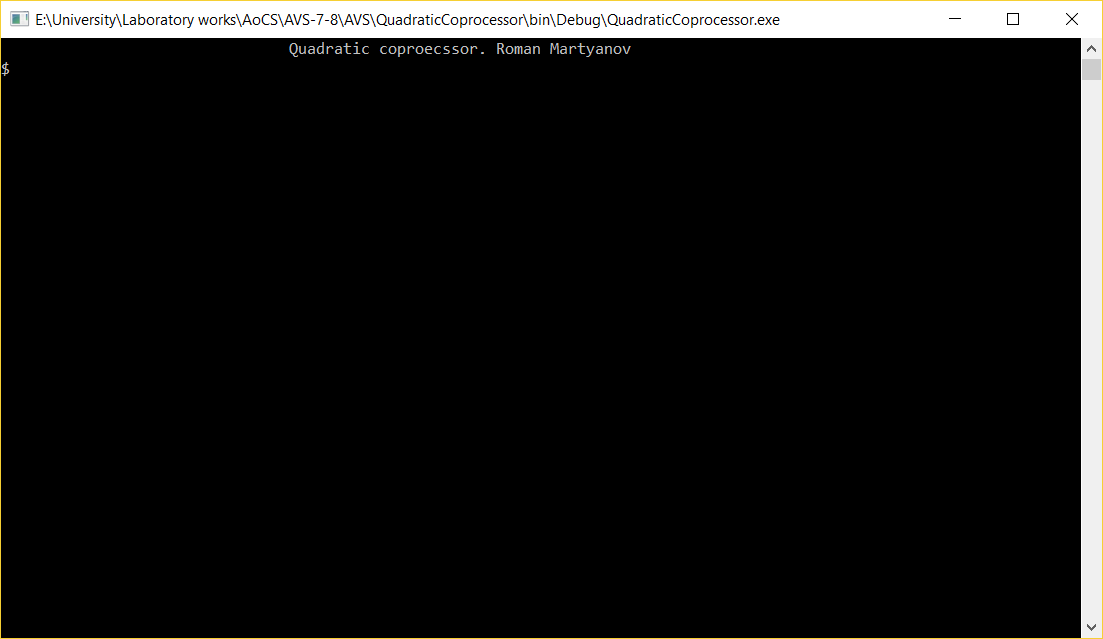
Разложение уравнения на множители

Вынесение общего множителя за скобку

Произведение равно нулю, когда хотя бы один из множителей равен нулю.

# Описание работы программы

Для написания эмулятора был выбран язык C#. Рассмотрим основные принципы работы программы. При запуске программы открывается консоль эмулятора.



В каждой строке после символа приветствия ($) необходимо вводить команды, которые интерпретируются и выполняются процессором.

Основные сущности:

1. Класс Program – в методе Main запускается работа эмулятора и происходит обработка ошибок.
2. Класс Interpreter – обрабатывает введенные команды, проверяет правильность ввода, разбирает текстовые команды на коды этих команд и аргументы, а затем передает преобразованные команды процессору.
3. Класс Processor – содержит в себе регистры, флаги, а также некоторый набор методов, главный метод – Execute, который исполняет поступившую на вход команду. Команды, которые относятся к процессору, он выполняет сам (например, reboot), а операции, связанные с арифметическими действиями, передает на исполнение ALU.
4. Класс Instruction – абстрактное представление инструкции процессора. Содержит в себе код и некоторое число аргументов (0 – 2).
5. Класс AssemblyTable – хранит в себе все коды инструкций и адреса регистров.
6. Класс ALU – выполняет арифметические операции с целым числами по алгоритмам, описанным выше.
7. Класс Register – содержит битовое представление числа с плавающей точкой (знак, мантисса, степень). Над объектами этого типа производится арифметические операции.

Список возможных команд:

1. stop – завершить работу эмулятора
2. reboot – сброс всех флагов и регистров.
3. state – вывод текущего состояния процессора.
4. mov – занести в регистр значение.
5. add – сложить первый аргумент со вторым и занести в первый.
6. sub – отнять от первого аргумента второй и занести в первый.
7. mul – умножить содержимое ax на аргумент.
8. div – разделить содержимое ax на аргумент.
9. solve – решить квадратное уравнение с коэффициентами a, b, c, где a – значение регистра ax, b – значение регистра bx, c – значение регистра cx.

Алгоритм решения квадратного уравнения:

1. Если все коэффициенты равны 0, то имеем бесконечно много решений.
2. Если a = 0, b = 0, c != 0, то решений нет.
3. Находим дискриминант, кладем результат в cx.
4. Если дискриминант отрицательный, то корней нет, выходим.
5. Если дискриминант равен 0, то решение одно, вычисляем его, результат заносим в ax.
6. Если дискриминант больше 0, то находим корни и заносим 1 корень в ax, 2 – в bx.

# Демонстрация работы программы

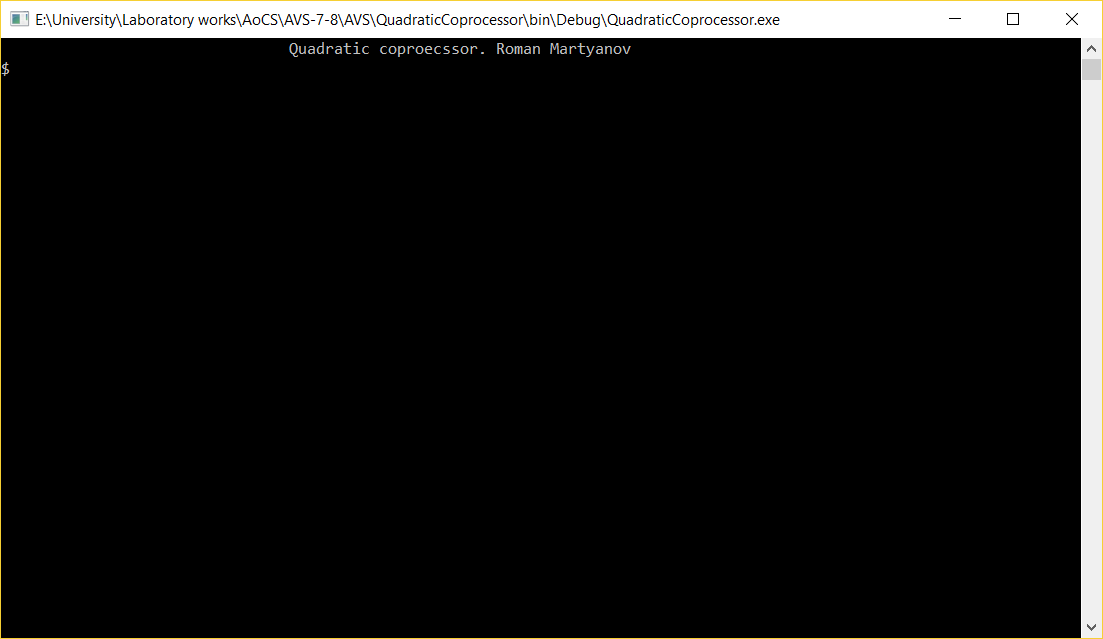


Рисунок 1. Запуск программы

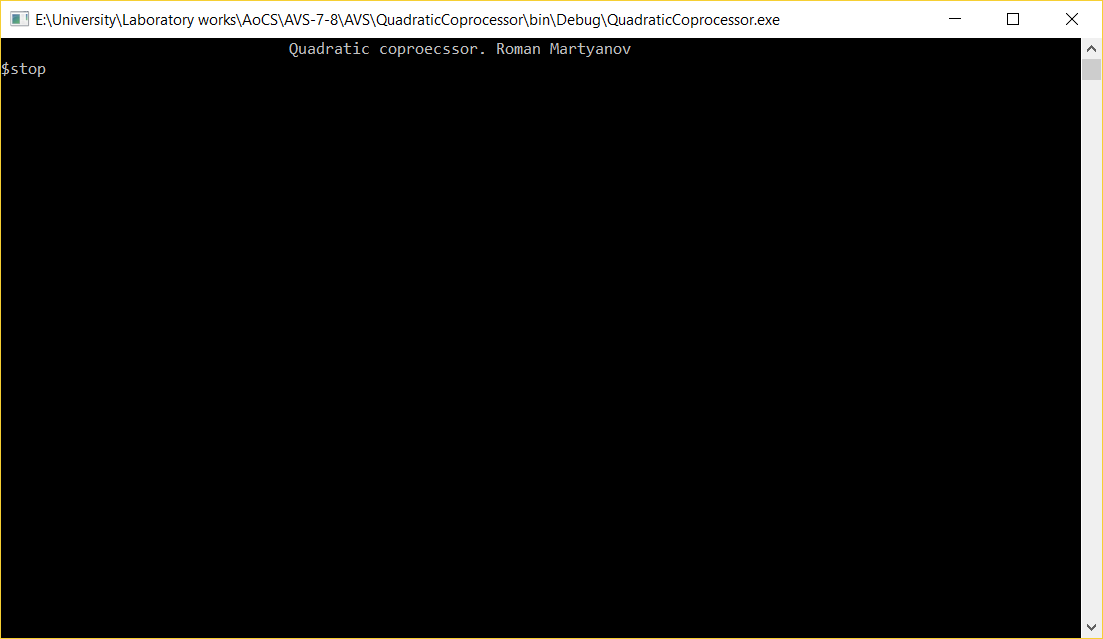


Рисунок 2. Выход из программы

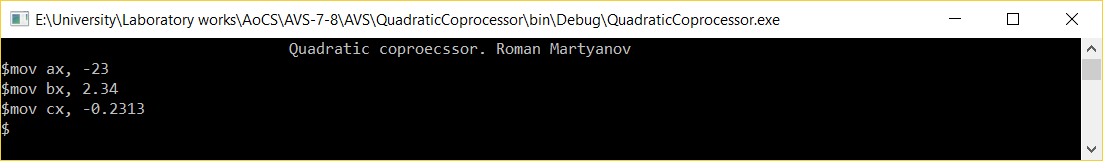


Рисунок 3. Заполнение регистров

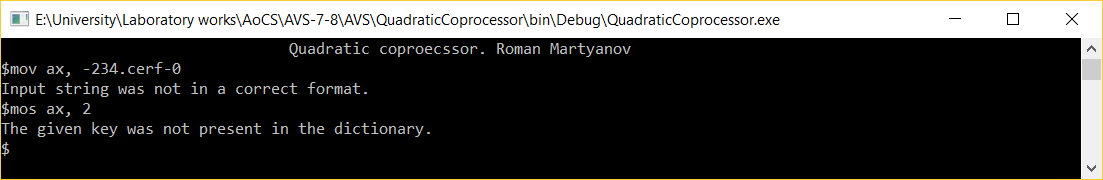


Рисунок 4. Проверка вводимых данных

Решение квадратного уравнения.

1. . В данном случае в регистре ax находится значение 2.466, в bx = 16, в cx = -1.3401.

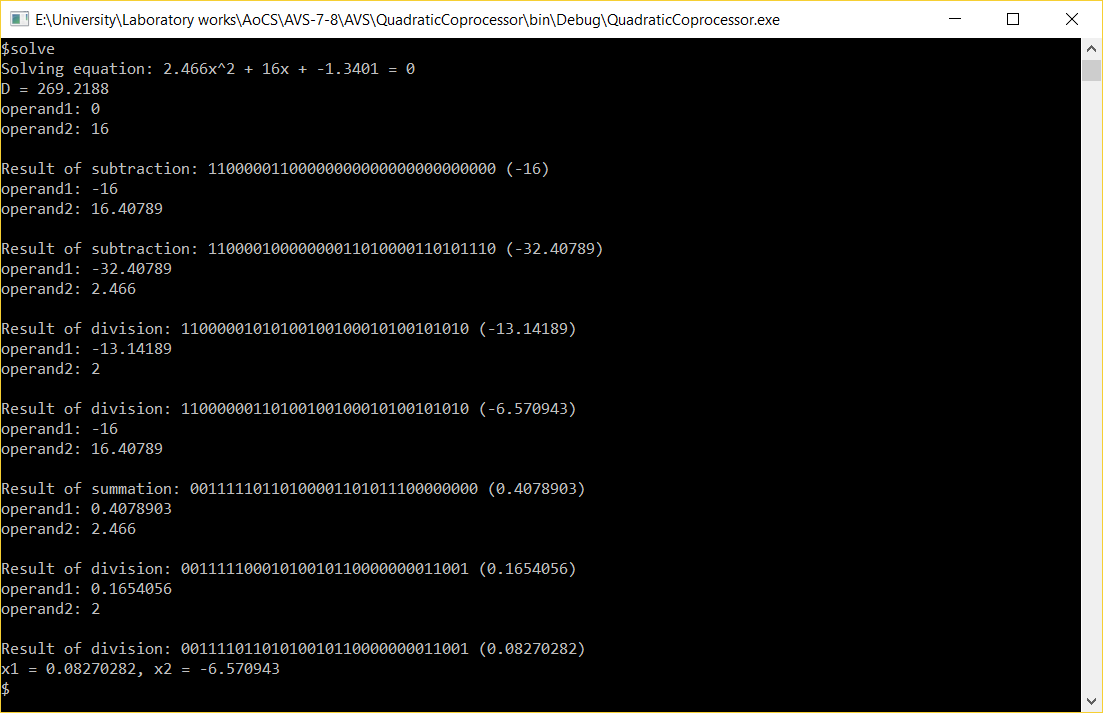


Рисунок 5. Решение квадратного уравнения (a≠0,b≠0,c≠0)

Проверим решение с помощью системы maple:



Рисунок 6. Проверка решения задачи по поиску корней уравнения

1. . В данном случае в регистре ax находится значение 0, в bx = 14, в cx = -1.02.

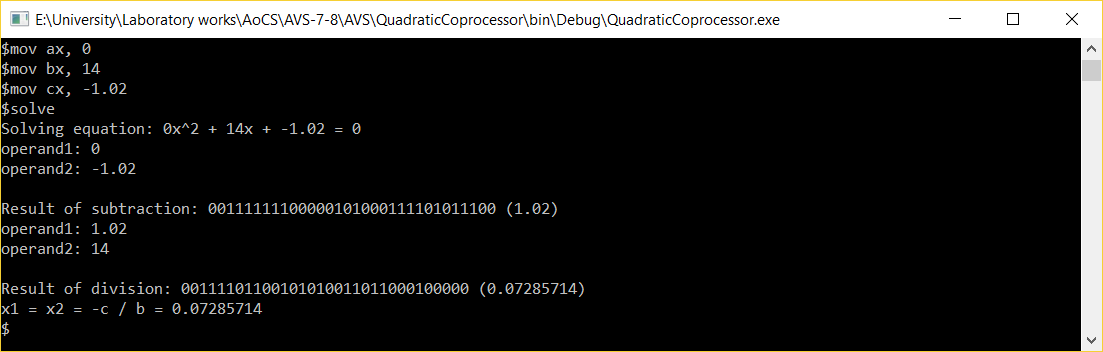


Рисунок 7. Решение квадратного уравнения (a=0,b≠0,c≠0)

1. . В данном случае в регистре ax находится значение 4.04, в bx = 0, в cx = -4.04.

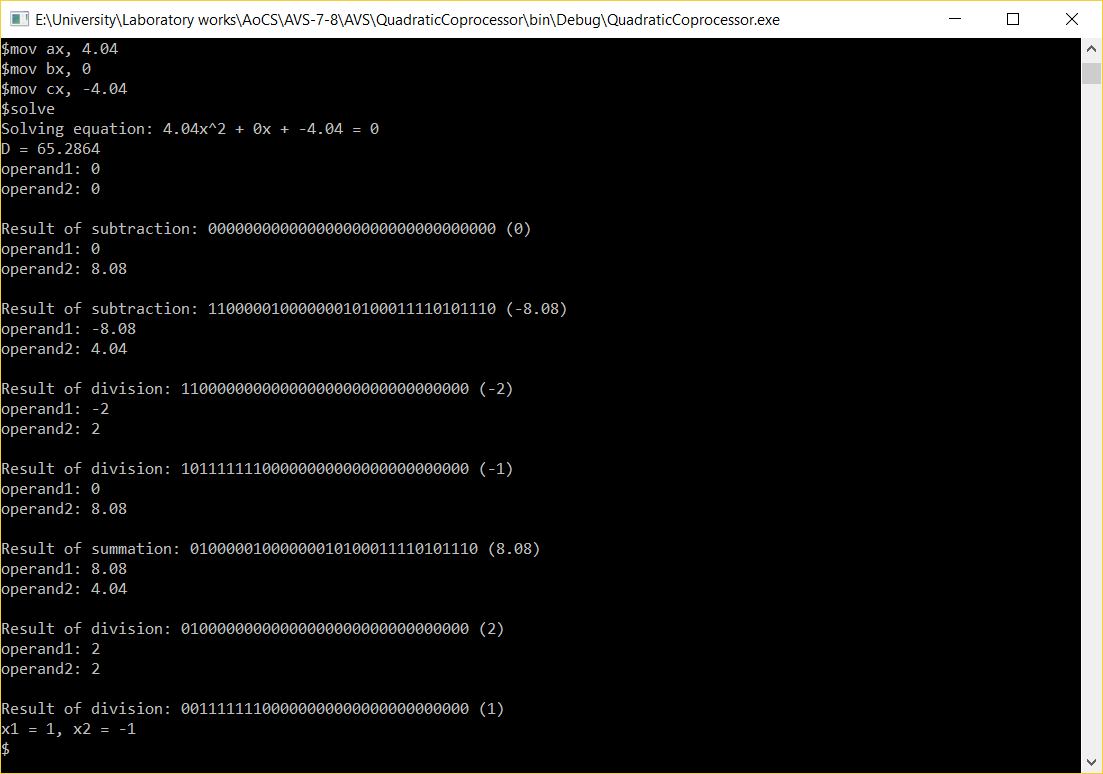


Рисунок 8. Решение квадратного уравнения (a≠0,b=0,c≠0)

1. a = 0, b = 0, c = 0. В данном случае в регистрах ax, bx и cx находятся 0.

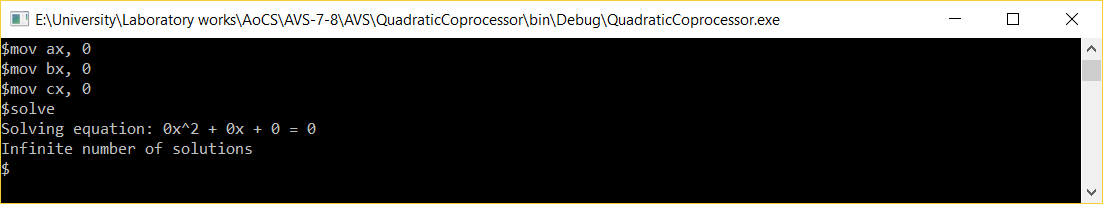


Рисунок 9. Решение квадратного уравнения (a=0,b=0,c=0)

1. Дискриминант меньше 0. В данном случае в регистре ax находится значение 23, в bx = 2.03, в cx = 12.34.

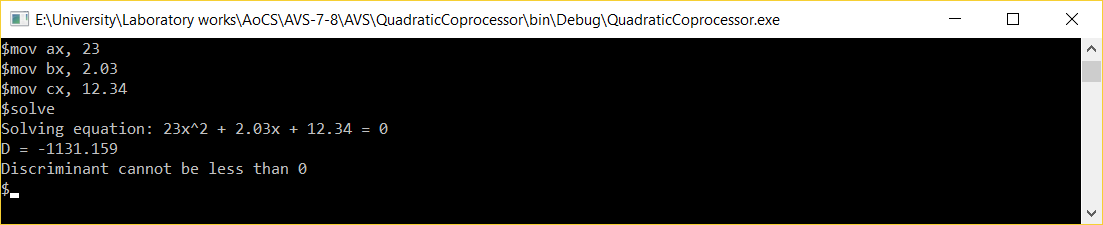


Рисунок 10. Решение квадратного уравнения (дискриминант меньше 0)

# Вывод

В результате выполнения лабораторной работе №7 была создана программа, эмулирующая работу процессора с вещественными числами, а также способную с помощью математического сопроцессора решать задачу решения квадратного уравнения.

# Приложение. Код программы

**Приложение 1. Код программы**

Program:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class Program

{

static void Main(string[] args)

{

string command;

Console.WriteLine("\t\t\t\tQuadratic coproecssor. Roman Martyanov"); while (true)

{

Console.Write("$");

command = Console.ReadLine();

if (command == "0") break;

try

{

Instruction instruction = Interpreter.Parse(command);

Processor.Execute(instruction);

}

catch (Exception e)

{

Console.WriteLine(e.Message);

//break;

}

if (Processor.State == ProcessorState.Stopped)

{

break;

}

}

}

}

}

Interpreter:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

static class Interpreter

{

private static AssemblyTable assemblyTable = new AssemblyTable();

public static Instruction Parse(string command)

{

command = Prepare(command);

string[] args = System.Text.RegularExpressions.Regex.Split(command, @"\s{1,}"); ;

if (args.Length > 3)

{

throw new Exception("Wrong command");

}

Instruction instruction = new Instruction()

{

Opcode = assemblyTable.GetInstructionCode(args[0]),

Name = args[0],

};

if (args.Length == 1)

{

instruction.InstructionType = InstructionType.Special;

return instruction;

}

instruction.Arg1 = args[1];

if (args.Length == 2)

{

instruction.InstructionType = InstructionType.Single;

return instruction;

}

instruction.Arg2 = args[2];

return instruction;

}

private static string Prepare(string command)

{

command = command.Trim().ToLower().Replace(',', ' ');

return command;

}

public static string Normalize(string bin, int len)

{

if (bin.Length > len)

{

bin = bin.Substring(bin.Length - len, len);

}

int d = len - bin.Length;

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

{

bin = "0" + bin;

}

return bin;

}

}

}

Processor:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace QuadraticCoprocessor

{

public static class Processor

{

private static Dictionary<string, Register> registers;

private static Dictionary<string, bool> flags;

private static ALU alu;

private static ProcessorState state = ProcessorState.NotStarted;

public static Dictionary<string, Register> Registers { get => registers; }

public static Dictionary<string, bool> Flags { get => flags; }

public static ProcessorState State => state;

public static void Start()

{

registers = new Dictionary<string, Register>();

flags = new Dictionary<string, bool>();

alu = new ALU();

registers["ax"] = new Register(0);

registers["bx"] = new Register(0);

registers["cx"] = new Register(0);

registers["dx"] = new Register(0);

registers["si"] = new Register(0);

registers["bp"] = new Register(0);

registers["ip"] = new Register(0);

flags["cf"] = false;

flags["pf"] = false;

flags["af"] = false;

flags["zf"] = false;

flags["sf"] = false;

flags["tf"] = false;

flags["if"] = false;

flags["df"] = false;

flags["of"] = false;

state = ProcessorState.Sleeping;

}

public static void Execute(Instruction instruction)

{

if (state == ProcessorState.NotStarted)

{

Start();

}

byte code = instruction.Opcode;

switch (code)

{

case 0x00:

Exit();

break;

case 0x01:

Reboot();

break;

case 0x02:

PrintState();

break;

case 0x03:

Mov(instruction.Arg1, instruction.Arg2);

break;

case 0x04:

Add(instruction.Arg1, instruction.Arg2);

break;

case 0x05:

Mul(instruction.Arg1);

break;

case 0x06:

Sub(instruction.Arg1, instruction.Arg2);

break;

case 0x07:

Div(instruction.Arg1);

break;

case 0x08:

Solve();

break;

default:

break;

}

}

public static void Mov(string dest, string source)

{

if (!IsRegister(dest))

{

Exit();

throw new Exception("Wrong operand in mov command");

}

Register destReg = registers[dest];

//add hex numbers

if (IsRegister(source))

{

Register sourceReg = registers[source];

destReg.Val = sourceReg.Val;

destReg.Sign = sourceReg.Sign;

destReg.Exponenta = sourceReg.Exponenta;

destReg.Mantissa = sourceReg.Mantissa;

registers[dest] = destReg;

}

else

{

destReg.Init(float.Parse(source));

registers[dest] = destReg;

}

}

public static void Add(string dest, string source)

{

string op1 = "", op2 = "";

if (!IsRegister(dest))

{

throw new Exception("Bad register name");

}

op1 = registers[dest].ToString();

if (IsRegister(source))

{

op2 = registers[source].ToString();

Console.WriteLine("operand1: " + registers[dest].Val);

registers[dest] = new Register(registers[dest].Val + registers[source].Val);

Console.WriteLine("operand2: " + registers[source].Val);

}

else

{

op2 = new Register(float.Parse(source)).ToString();

Console.WriteLine("operand1: " + registers[dest].Val);

registers[dest] = new Register(registers[dest].Val + float.Parse(source));

Console.WriteLine("operand2: " + float.Parse(source));

//check number

}

alu.Add(op1, op2, ref flags);

Console.WriteLine("\nResult of summation: " + registers[dest] + " (" + registers[dest].Val + ")");

}

public static void Sub(string dest, string source)

{

string op1 = "", op2 = "";

if (!IsRegister(dest))

{

throw new Exception("Bad register name");

}

op1 = registers[dest].ToString();

if (IsRegister(source))

{

op2 = registers[source].ToString();

Console.WriteLine("operand1: " + registers[dest].Val);

registers[dest] = new Register(registers[dest].Val - registers[source].Val);

Console.WriteLine("operand2: " + registers[source].Val);

}

else

{

op2 = new Register(float.Parse(source)).ToString();

Console.WriteLine("operand1: " + registers[dest].Val);

registers[dest] = new Register(registers[dest].Val - float.Parse(source));

Console.WriteLine("operand2: " + float.Parse(source));

}

alu.Sub(op1, op2, ref flags);

Console.WriteLine("\nResult of subtraction: " + registers[dest] + " (" + registers[dest].Val + ")");

}

public static void Mul(string source)

{

string op1 = "", op2 = "";

op1 = registers["ax"].ToString();

if (IsRegister(source))

{

op2 = registers[source].ToString();

Console.WriteLine("operand1: " + registers["ax"].Val);

float res = registers["ax"].Val \* registers[source].Val;

if (float.IsInfinity(res)) throw new Exception("Multiplication overflow");

registers["ax"] = new Register(res);

Console.WriteLine("operand2: " + registers[source].Val);

}

else

{

op2 = new Register(float.Parse(source)).ToString();

Console.WriteLine("operand1: " + registers["ax"].Val);

float res = registers["ax"].Val \* float.Parse(source);

registers["ax"] = new Register(res);

Console.WriteLine("operand2: " + float.Parse(source));

}

alu.Mul(op1, op2, ref flags);

Console.WriteLine("\nResult of multiplication: " + registers["ax"] + " (" + registers["ax"].Val + ")");

}

public static void Div(string source)

{

string op1 = "", op2 = "";

op1 = registers["ax"].ToString();

if (IsRegister(source))

{

op2 = registers[source].ToString();

Console.WriteLine("operand1: " + registers["ax"].Val);

if (registers[source].Val == 0) throw new Exception("Cannot divide by zero");

registers["ax"] = new Register(registers["ax"].Val / registers[source].Val);

Console.WriteLine("operand2: " + registers[source].Val);

}

else

{

op2 = new Register(float.Parse(source)).ToString();

Console.WriteLine("operand1: " + registers["ax"].Val);

if (float.Parse(source) == 0) throw new Exception("Cannot divide by zero");

registers["ax"] = new Register(registers["ax"].Val / float.Parse(source));

Console.WriteLine("operand2: " + float.Parse(source));

}

alu.Div(op1, op2, ref flags);

Console.WriteLine("\nResult of division: " + registers["ax"] + " (" + registers["ax"].Val + ")");

}

public static void Solve()

{

alu.print = false;

string a = registers["ax"].ToString();

string b = registers["bx"].ToString();

string c = registers["cx"].ToString();

float a\_val = registers["ax"].Val, b\_val = registers["bx"].Val, c\_val = registers["cx"].Val, x1, x2;

Console.WriteLine("Solving equation: " + a\_val.ToString() + "x^2 + " + b\_val.ToString() + "x + " + c\_val.ToString() + " = 0");

if (a\_val == 0)

{

if (b\_val == 0)

{

if (c\_val == 0) throw new Exception("Infinite number of solutions");

else throw new Exception("No solutions");

}

Mov("ax", "0");

Sub("ax", "cx");

Div("bx");

Console.WriteLine("x1 = x2 = -c / b = " + registers["ax"].Val.ToString());

}

else

{

float D = b\_val \* b\_val - 4 \* a\_val \* c\_val;

Console.WriteLine("D = " + D.ToString());

if (D < 0) throw new Exception("Discriminant cannot be less than 0");

D = (float)Math.Sqrt(D);

Mov("ax", "0");

Sub("ax", "bx");

Mov("cx", "ax");

Sub("ax", D.ToString());

Div(a\_val.ToString());

Div("2");

Mov("bx", "ax");

Mov("ax", "cx");

Add("ax", D.ToString());

Div(a\_val.ToString());

Div("2");

Console.WriteLine("x1 = " + registers["ax"].Val.ToString() + ", x2 = " + registers["bx"].Val.ToString());

}

alu.print = true;

}

public static bool IsRegister(string val)

{

foreach (var r in registers)

{

if (r.Key == val)

{

return true;

}

}

return false;

}

public static void PrintState()

{

Console.WriteLine(new string('=', 20));

Console.WriteLine("Registers: ");

foreach (var reg in registers)

{

if (reg.Value.Val != 0)

{

Console.WriteLine(new string('-', 20));

Console.WriteLine(reg.Key + " = " + reg.Value.Val);

Console.WriteLine("Sign: " + reg.Value.Sign + " (" +

(reg.Value.Sign == '1' ? "negative" : "positive") + ")");

Console.WriteLine("Exponenta: " + reg.Value.Exponenta + " (" +

reg.Value.GetExp().ToString() + ")");

Console.WriteLine("Mantissa: " + reg.Value.Mantissa);

}

else

{

Console.WriteLine(new string('-', 20));

Console.WriteLine(reg.Key + " = 0.0");

}

}

Console.WriteLine();

Console.WriteLine("Flags: ");

foreach (var flag in flags)

{

Console.Write(flag.Key + "=" + flag.Value + "; ");

}

Console.WriteLine(new string('=', 20));

}

public static void Reboot()

{

state = ProcessorState.Running;

registers["ax"] = new Register(0);

registers["bx"] = new Register(0);

registers["cx"] = new Register(0);

registers["dx"] = new Register(0);

registers["si"] = new Register(0);

registers["bp"] = new Register(0);

registers["ip"] = new Register(0);

flags["cf"] = false;

flags["pf"] = false;

flags["af"] = false;

flags["zf"] = false;

flags["sf"] = false;

flags["tf"] = false;

flags["if"] = false;

flags["df"] = false;

flags["of"] = false;

}

public static void Exit()

{

state = ProcessorState.Stopped;

registers["ax"] = new Register(0);

registers["bx"] = new Register(0);

registers["cx"] = new Register(0);

registers["dx"] = new Register(0);

registers["si"] = new Register(0);

registers["bp"] = new Register(0);

registers["ip"] = new Register(0);

flags["cf"] = false;

flags["pf"] = false;

flags["af"] = false;

flags["zf"] = false;

flags["sf"] = false;

flags["tf"] = false;

flags["if"] = false;

flags["df"] = false;

flags["of"] = false;

}

}

}

ALU:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class ALU

{

private string running = "";

private bool overflow = false;

public string AddInt(string op1, string op2)

{

if (running == "")

{

Console.WriteLine(op1);

Console.WriteLine("+");

Console.WriteLine(op2);

Console.WriteLine(new string('-', 32));

}

int len = op1.Length, rem = 0, cur = 0;

if (op2[0] == '0') Swap(ref op1, ref op2);

string res = "";

for (int i = len - 1; i >= 0; i--)

{

cur = Convert.ToInt32(op1[i]) + Convert.ToInt32(op2[i]) - 96 + rem;

rem = cur / 2;

res += cur % 2;

if (cur % 2 == 1 && running == "")

{

Console.WriteLine(Interpreter.Normalize(res, 23) + "\t|\tStep " + (op2.Length - i).ToString() +

", Cur res=" + Interpreter.Normalize(new string(res.ToCharArray().Reverse().ToArray()), 23).TrimStart('0'));

}

}

if (rem > 0) overflow = true;

res = new string(res.ToCharArray().Reverse().ToArray());

if (running == "")

{

Console.WriteLine(new string('-', 23));

Console.WriteLine("Answer: " + Interpreter.Normalize(res, 23));

}

return res;

}

public string SubInt(string op1, string op2)

{

return AddInt(op1, Neg(op2));

}

public string Add(string op1, string op2, ref Dictionary<string, bool> flags)

{

char sgn = '0';

string res = "";

int exp1 = Register.GetExp(op1), exp2 = Register.GetExp(op2);

if (exp1 < exp2 || (exp1 == exp2 &&

Convert.ToInt32(op1.Substring(9, 23), 2) < Convert.ToInt32(op2.Substring(9, 23), 2)))

{

Swap(ref op1, ref op2);

int t = exp2;

exp2 = exp1;

exp1 = t;

}

if (exp1 == exp2) overflow = true;

if (op1[0] == '1') sgn = '1';

string mant1 = op1.Substring(9, 23), mant2 = op2.Substring(9, 23);

int diff = exp1 - exp2;

if (diff > 0)

{

mant2 = (new string('0', diff - 1) + "1" + mant2).Substring(0, 23);

}

string exp = op1.Substring(1, 8);

bool wasSub = false;

Console.WriteLine("\nSumming mantisses: ");

if (op1[0] == op2[0])

{

res = AddInt(mant1, mant2);

}

else

{

res = SubInt("0" + mant1, "0" + mant2);

if (res[0] == '1')

{

res = res.Substring(1, 23);

for (int i = 0; i < res.Length; i++)

{

if (res[i] == '1')

{

Console.WriteLine("\nNormalizing exponent: ");

exp = SubInt(exp, Interpreter.Normalize(Convert.ToString(i + 1, 2), 8));

res = res.Substring(i + 1, res.Length - i - 1) + new string('0', i + 1);

break;

}

}

}

else res = res.Substring(1, 23);

wasSub = true;

}

if (overflow)

{

if (op1[0] == op2[0])

{

Console.WriteLine("Normalizing exponent: ");

exp = AddInt(exp, Interpreter.Normalize("1", 8));

}

if (!wasSub)

{

res = ("0" + res.Substring(0, 22));

}

}

overflow = false;

res = sgn + exp + res;

Console.WriteLine(res);

return res;

}

public string Sub(string op1, string op2, ref Dictionary<string, bool> flags)

{

char sgn = '0';

if (op2[0] == '0') sgn = '1';

return Add(op1, sgn + op2.Substring(1, op2.Length - 1), ref flags);

}

public string Mul(string op1, string op2, ref Dictionary<string, bool> flags)

{

char sgn = '0';

string res = "";

int exp1 = Register.GetExp(op1), exp2 = Register.GetExp(op2);

if (op1[0] != op2[0]) sgn = '1';

string mant1 = op1.Substring(9, 23), mant2 = op2.Substring(9, 23);

string exp = op1.Substring(1, 8);

res = MulInt('1' + mant1, '1' + mant2, ref flags);

int mod = 1;

for (int i = 0; i < res.Length; i++)

{

if (res[i] == '1')

{

res = res.Substring(i + 1, Math.Min(23, res.Length - i - 1));

break;

}

mod--;

}

exp = Interpreter.Normalize(Convert.ToString(exp1 + exp2 + mod + 127, 2), 8);

res = sgn + exp + res;

Console.WriteLine("Resulting mantissa: " + res);

Console.WriteLine("Resulting exponenta: " + exp);

return res;

}

public string MulInt(string op1, string op2, ref Dictionary<string, bool> flags)

{

running = "mul";

Console.WriteLine(op1);

Console.WriteLine('\*');

Console.WriteLine(op2);

Console.WriteLine(new string('-', 23));

string res = Interpreter.Normalize("", 100);

for (int i = op2.Length - 1; i >= 0; i--)

{

if (op2[i] == '1')

{

res = AddInt(res, Interpreter.Normalize(op1 + new string('0', op2.Length - i - 1), 100));

Console.WriteLine(Interpreter.Normalize(op1 + new string('0', op2.Length - i - 1), 23)

+ "\t|\tStep " + (op2.Length - i).ToString() +

", Cur res=" + Interpreter.Normalize(res, 23).TrimStart('0'));

}

}

Console.WriteLine(new string('-', 23));

//Console.WriteLine("Answer: " + Interpreter.Normalize(res, 23).TrimStart('0'));

running = "";

return res.Substring(res.Length - 48, 48);

}

public string Div(string op1, string op2, ref Dictionary<string, bool> flags)

{

char sgn = '0';

string res = "";

int exp1 = Register.GetExp(op1), exp2 = Register.GetExp(op2);

if (op1[0] != op2[0]) sgn = '1';

string mant1 = op1.Substring(9, 23), mant2 = op2.Substring(9, 23);

string exp = op1.Substring(1, 8);

res = DivInt("1" + mant1 + new string('0', 23), "1" + mant2);

int mod = 1;

for (int i = 0; i < res.Length; i++)

{

if (res[i] == '1')

{

res = res.Substring(i + 1, Math.Min(23, res.Length - i - 1));

break;

}

mod--;

}

exp = Interpreter.Normalize(Convert.ToString(exp1 - exp2 + 127, 2), 8);

Console.WriteLine("Resulting mantissa: " + res);

Console.WriteLine("Resulting exponenta: " + exp);

res = sgn + exp + res;

return res;

}

public string DivInt(string op1, string op2)

{

running = "div";

Console.WriteLine(op1 + "\t|\t" + op2);

string tmp = "", ans = "", tmpTrimmed = "";

for (int i = 0; i < op1.Length; i++)

{

tmp += op1[i];

if (Convert.ToInt64(tmp, 2) < Convert.ToInt64(op2, 2))

{

ans += '0';

}

else

{

tmpTrimmed = tmp.TrimStart('0');

int real\_len = tmpTrimmed.Length;

Console.Write(new string(' ', i - tmpTrimmed.Length + 1) + op2.TrimStart('0') + '\n');

tmp = SubInt(tmp, Interpreter.Normalize(op2, tmp.Length));

Console.WriteLine(new string(' ', i - tmpTrimmed.Length + 1) + new string('-', tmpTrimmed.Length));

Console.Write(new string(' ', i - tmpTrimmed.Length + 1) + tmp.Substring(tmp.Length - real\_len, real\_len) + '\n');

ans += '1';

if (tmp.TrimStart('0') == "") tmp = "";

}

}

tmpTrimmed = tmp.TrimStart('0');

if (tmpTrimmed == "") tmpTrimmed = "0";

Console.Write("Remainder: " + tmpTrimmed + '\n');

Console.WriteLine("Answer: " + ans.TrimStart('0'));

running = "";

return ans;

}

public string Inc(string s)

{

if (running == "") running = "Inc";

string res = "";

res = AddInt(s, Interpreter.Normalize("1", s.Length));

if (running == "Inc") running = "";

return res;

}

private int Abs(string val)

{

int x = 0;

if (IsNeg(val))

{

x = 65536 - Convert.ToInt32(val, 2) - 1;

}

else x = Convert.ToInt32(val, 2);

return x;

}

private bool IsNeg(string binNum)

{

return binNum[0] == '1';

}

private void Swap(ref string str1, ref string str2)

{

var temp = str2;

str2 = str1;

str1 = temp;

}

public string Neg(string op1)

{

string res = "";

foreach (char c in op1)

{

if (c == '0') res += '1';

else res += '0';

}

return Inc(res);

}

}

}

Instruction:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class Instruction

{

public string Name { get; set; }

public byte Opcode { get; set; }

public string Arg1 { get; set; }

public string Arg2 { get; set; }

public InstructionType InstructionType { get; set; }

public Instruction()

{

InstructionType = InstructionType.Double;

}

public Instruction(string name)

{

Name = name;

InstructionType = InstructionType.Double;

}

public void TransformName()

{

switch (Name)

{

case "exit":

Opcode = 0x00;

break;

case "reboot":

Opcode = 0x01;

break;

case "state":

Opcode = 0x02;

break;

case "mov":

Opcode = 0x03;

break;

case "add":

Opcode = 0x04;

break;

default:

Opcode = 0xFF;

break;

}

}

}

}

AssembleTable:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class AssemblyTable

{

private Dictionary<string, byte> registers;

private Dictionary<string, byte> flags;

private Dictionary<string, byte> instructions;

public AssemblyTable()

{

registers = new Dictionary<string, byte>();

flags = new Dictionary<string, byte>();

instructions = new Dictionary<string, byte>();

Init();

}

private void Init()

{

registers["ax"] = 0x00;

registers["bx"] = 0x01;

registers["cx"] = 0x02;

registers["dx"] = 0x03;

registers["si"] = 0x04;

registers["bp"] = 0x05;

registers["ip"] = 0x06;

flags["cf"] = 0x00;

flags["pf"] = 0x02;

flags["af"] = 0x04;

flags["zf"] = 0x06;

flags["sf"] = 0x07;

flags["tf"] = 0x08;

flags["if"] = 0x09;

flags["df"] = 0x0A;

flags["of"] = 0x0B;

instructions["stop"] = 0x00;

instructions["reboot"] = 0x01;

instructions["state"] = 0x02;

instructions["mov"] = 0x03;

instructions["add"] = 0x04;

instructions["mul"] = 0x05;

instructions["sub"] = 0x06;

instructions["div"] = 0x07;

}

public string GetInstruction(byte code)

{

foreach (var instruction in instructions)

{

if (instruction.Value == code)

{

return instruction.Key;

}

}

return "";

}

public byte GetInstructionCode(string instruction)

{

//!!

return instructions[instruction];

}

}

}

Register

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class Register

{

public float Val { get; set; }

public byte Id { get; set; }

public char Sign { get; set; }

public string Mantissa { get; set; }

public string Exponenta { get; set; }

public Register(float val)

{

Init(val);

Id = 0x00;

}

private void Parse(string s)

{

Sign = s[0];

Exponenta = s.Substring(1, 8);

Mantissa = s.Substring(9, 23);

}

public int GetExp() =>

(Convert.ToInt16(Exponenta, 2) == 0 ? 127 : Convert.ToInt16(Exponenta, 2)) - 127;

public static int GetExp(string s) =>

(Convert.ToInt16(s.Substring(1, 8), 2) == 0 ? 127 : Convert.ToInt16(s.Substring(1, 8), 2)) - 127;

public override string ToString()

{

return Sign + Exponenta + Mantissa;

}

public string Print()

{

return Sign + "|" + Exponenta + "|" + Mantissa;

}

public void Init(float val)

{

Val = val;

Sign = val >= 0 ? '0' : '1';

string s = Sign + Interpreter.Normalize(Interpreter.FloatToBinary(Math.Abs(val)), 31);

Parse(s);

}

public static implicit operator Register(string s)

{

Register r = new Register(0);

r.Sign = s[0];

r.Exponenta = s.Substring(1, 8);

r.Mantissa = s.Substring(9, 23);

return r;

}

}

}