Оглавление

[Оглавление 2](#_Toc511219193)

[1 Цель работы 3](#_Toc511219194)

[2 Постановка задачи 4](#_Toc511219195)

[3 Краткие теоретические сведения 5](#_Toc511219196)

[3.1 Прямой код 5](#_Toc511219197)

[3.2 Обратный код 6](#_Toc511219198)

[1.1 Дополнительный код 7](#_Toc511219199)

[4 Арифметические команды Assembler 9](#_Toc511219200)

[4.1 ADD 9](#_Toc511219201)

[4.2 SUB 10](#_Toc511219202)

[4.3 MUL 10](#_Toc511219203)

[4.4 Div 11](#_Toc511219204)

[5 Демонстрация работы программы 13](#_Toc511219205)

[6 Вывод 16](#_Toc511219206)

[Приложение. Исходный код программы 17](#_Toc511219207)

# Цель работы

Разработать АЛУ, позволяющий производить арифметические операции над целыми числами.

# Постановка задачи

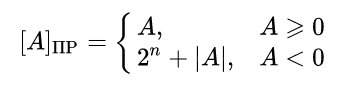
Написать эмулятор АЛУ, реализующего операции сложения, вычитания, умножения и деления с фиксированной точкой над двумя введенными числами. Также должна быть возможность пошагового выполнения алгоритмов.

# Краткие теоретические сведения

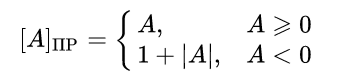
## Прямой код

При записи числа в прямом коде старший разряд (старший бит) объявляется знаковым разрядом (знаковым битом). Если знаковый бит равен 0, число [положительное](https://ru.wikipedia.org/w/index.php?title=Положительное_число&action=edit&redlink=1), иначе — [отрицательное](https://ru.wikipedia.org/wiki/Отрицательное_число). В остальных разрядах (которые называются цифровыми разрядами) записывается двоичное представление модуля числа.

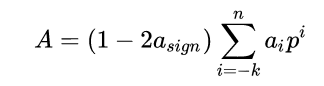
Функция кодирования двоичных чисел (в том числе целых чисел и смешанных дробей) в прямом коде имеет вид:

,

где n — номер знакового разряда (знакового бита). В частности, при кодировании правильных двоичных дробей (то есть чисел, удовлетворяющих неравенству -1 < A < 1), n=0 и функция кодирования принимает вид:

.

Величина числа A в прямом коде определяется по следующей формуле:

,

где:

i — номер разряда числа; отрицательное число — номер разряда справа от запятой; положительное число — номер разряда слева от запятой;

k — количество разрядов справа от запятой (кол-во разрядов дробной части числа);

n — количество разрядов слева от запятой (кол-во разрядов целой части числа);

ai — цифра в i-ом разряде;

p — основание системы счисления; равно 2 для двоичных чисел, 10 — для десятичных, 16 — для шестнадцатеричных и т. п.;

asign — значение знакового разряда (знакового бита);

A — число, имеющее k разрядов справа от запятой (дробная часть) и n разрядов слева (целая часть); учитываются только цифровые разряды.

Как видно из последней формулы, знаковый разряд в прямом коде не имеет разрядного веса. При выполнении арифметических операций это приводит к необходимости отдельной обработки знакового разряда в прямом коде.

|  |  |  |  |
| --- | --- | --- | --- |
| Десятичное число | Двоичное число | Код прямой двоичный 8-и разрядный | Примечание |
| 0 | 0 | 0000 0000 | Положительный ноль |
| -0 | -0 | 1000 0000 | Отрицательный ноль |
| 5 | 101 | 0000 0101 |  |
| 10 | 1010 | 0000 1010 |  |
| -5 | -101 | 1000 0101 |  |

Таблица 1. Представление чисел в прямом коде.

## Обратный код

Обратный код— метод вычислительной математики, позволяющий вычесть одно число из другого, используя только операцию сложения над натуральными числами. Ранее метод использовался в механических калькуляторах (арифмометрах). В настоящее время используется в основном в современных компьютерах.

Обратный n-разрядный двоичный код положительного целого числа состоит из одноразрядного кода знака (двоичной цифры 0), за которым следует (n−1)-разрядное двоичное представление модуля числа (обратный код положительного числа совпадает с прямым кодом).

*Пример.* Двоичное представление числа 5 есть 101. 10-разрядный двоичный код числа +5 записывается как 00 0000 0101.

Обратный n-разрядный двоичный код отрицательного целого числа состоит из одноразрядного кода знака (двоичной цифры 1), за которым следует (n−1)-разрядное двоичное число, представляющее собой инвертированное (n−1)-разрядное представление модуля числа. Следует отметить, что для изменения знака числа достаточно проинвертировать все его разряды, не обращая внимания, знаковый ли это разряд или информационный.

*Пример.* Двоичное представление числа 5 есть 101, его 10-разрядное двоичное представление — 00 0000 0101. Обратный 10-разрядный двоичный код числа -5 есть 11 1111 1010.

Для преобразования отрицательного числа в положительное тоже применяется операция инвертирования. Этим обратные коды удобны в применении. В качестве недостатка следует отметить, что в обратных двоичных кодах имеются два кода числа 0: «положительный нуль» 00 0000 0000 и “отрицательный нуль” 11 1111 1111 (приведены 10-разрядные обратные коды). Это приводит к некоторому усложнению операции суммирования. Поэтому в дальнейшем перешли к дополнительным кодам записи знаковых целых чисел.

n-разрядный обратный код позволяет представить числа от −2n−1+1 до +2n−1−1.

## Дополнительный код

Дополнительный код — наиболее распространённый способ представления [отрицательных целых чисел](https://ru.wikipedia.org/wiki/Отрицательное_число) в [компьютерах](https://ru.wikipedia.org/wiki/Компьютер). Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру [ЭВМ](https://ru.wikipedia.org/wiki/Компьютер). В англоязычной литературе обратный код называют первым дополнением, а дополнительный код называют вторым дополнением.

Дополнительный код для отрицательного числа можно получить инвертированием его двоичного модуля (первое дополнение) и прибавлением к инверсии единицы (второе дополнение), либо вычитанием числа из нуля.

Дополнительный код (второе дополнение) двоичного числа получается добавлением 1 к младшему значащему разряду его [первого дополнения](https://en.wikipedia.org/wiki/Signed_number_representation#_blank).

Второе дополнение двоичного числа определяется как величина, полученная вычитанием числа из наибольшей степени двух (из 2N для N-битного второго дополнения).

**Представление отрицательного числа в дополнительном коде**

При записи числа в дополнительном коде старший разряд является знаковым. Если его значение равно 0, то в остальных разрядах записано положительное [двоичное число](https://ru.wikipedia.org/wiki/Двоичная_система_счисления), совпадающее с [прямым кодом](https://ru.wikipedia.org/wiki/Прямой_код_(представление_числа)).

Двоичное 8-разрядное число со знаком в дополнительном коде может представлять любое целое в диапазоне от −128 до +127. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах, равно -1 = 127{\displaystyle 2^{7}-1=127}

Примеры:

|  |  |  |  |
| --- | --- | --- | --- |
| Десятичное представление | Двоичное представление (8 бит) | | |
| [прямой](https://ru.wikipedia.org/wiki/Прямой_код) | [обратный](https://ru.wikipedia.org/wiki/Обратный_код) | дополнительный |
| 127 | 0111 1111 | 0111 1111 | 0111 1111 |
| 1 | 0000 0001 | 0000 0001 | 0000 0001 |
| 0 | 0000 0000 | 0000 0000 | 0000 0000 |
| -0 | 1000 0000 | 1111 1111 | --- |
| -1 | 1000 0001 | 1111 1110 | 1111 1111 |
| -2 | 1000 0010 | 1111 1101 | 1111 1110 |
| -3 | 1000 0011 | 1111 1100 | 1111 1101 |
| -4 | 1000 0100 | 1111 1011 | 1111 1100 |

Таблица 2. Двоичное представление чисел

# Арифметические команды Assembler

## ADD

Синтаксис команды *ADD*:

Для сложения двух чисел предназначена команда ADD. Она работает как с числами со знаком, так и с числами без знака (это особенность дополнительного кода).

Операнды должны иметь одинаковый размер (нельзя складывать 16- и 8-битное значение). Результат помещается на место первого операнда. В общем, эти правила справедливы для большинства команд.

После выполнения команды изменяются флаги, по которым можно определить характеристики результата:

Флаг CF устанавливается, если при сложении произошёл перенос из старшего разряда. Для беззнаковых чисел это будет означать, что произошло переполнение и результат получился некорректным.

Флаг OF обозначает переполнение для чисел со знаком.

Флаг SF равен знаковому биту результата (естественно, для чисел со знаком, а для беззнаковых он равен старшему биту и особо смысла не имеет).

Флаг ZF устанавливается, если результат равен 0.

Флаг PF — признак чётности, равен 1, если результат содержит нечётное число единиц.

*Примеры:*

add ax, 10 ;AX = AX + 10

add dx, cx ;DX = DX + CX

Суммой может быть один из следующих:

* Область памяти (MEM)
* Регистр общего назначения (REG)

То есть эта команда не работает с сегментными регистрами. Комбинации сумма-число могут быть следующими:

REG, MEM

MEM, REG

REG, REG

MEM, IMM

REG, IMM

## SUB

Вычитание выполняется с помощью команды SUB. Результат также помещается на место первого операнда и опять же выставляются флаги. Единственная разница в том, что происходит вычитание, а не сложение.

На самом деле вычитание в процессоре реализовано с помощью сложения. Процессор меняет знак второго операнда на противоположный, а затем складывает два числа. Если вам необходимо в программе поменять знак числа на противоположный, можно использовать команду NEG. У этой команды всего один операнд.

Примеры:

sub si, dx ;SI = SI - DX

neg ax ;AX = -AX

Эта команда, также как и команда сложения, не работает с сегментными регистрами. Комбинации разность-число могут быть следующими:

REG, MEM

MEM, REG

REG, REG

MEM, IMM

REG, IMM

## MUL

Для умножения чисел без знака предназначена команда MUL. У этой команды только один операнд — второй множитель, который должен находиться в регистре или в памяти. Местоположение первого множителя и результата задаётся неявно и зависит от размера операнда:

Отличие умножения от сложения и вычитания в том, что разрядность результата получается в 2 раза больше, чем разрядность сомножителей. Также и в десятичной системе — например, умножая двухзначное число на двухзначное, мы можем получить в результате максимум четырёхзначное.

Запись «DX:AX» означает, что старшее слово результата будет находиться в DX, а младшее — в AX. Примеры:

mul bl ;AX = AL \* BL

mul ax ;DX:AX = AX \* AX

|  |  |  |
| --- | --- | --- |
| **Размер операнда** | **Множитель** | **Результат** |
| Байт | AL | AX |
| Слово | AX | DX:AX |

Если старшая часть результата равна нулю, то флаги CF и ОF будут иметь нулевое значение. В этом случае старшую часть результата можно отбросить. Это свойство можно использовать в программе, если результат должен быть такого же размера, как множители.

Эта команда не работает с сегментными регистрами, а также не работает непосредственно с числами. То есть вот так

Алгоритм работы команды MUL:

- Если число - это байт, то AX = AL \* число

- Если число - это слово, то (DX AX) = AX \* число

## Div

Для деления чисел без знака предназначена команда DIV

Формат команды:

DIV делитель

Местоположение делимого в командах DIV не указывается, оно жестко определено и зависит от размерности (байт, слово, двойное слово) делителя. По этой причине делителем не может быть явно заданное (в команде) число, так как явно заданное число не имеет размерности.

Делимое должно быть помещено:

* в AX - если делитель имеет размерность байт, тогда после деления частное находим в регистре AL, остаток от деления - в регистре AH;
* в DX:AX - если делитель имеет размерность слово (2 байта), тогда после деления частное находим в регистре AX, остаток от деления - в регистре DX;

При делении на слово или двойное слово делимое должно быть "расписано" на два регистра. Запись DX:AX означает делимое в виде двойного слова, два старших байта которого помещены в DX, два младших байта — в AX.

В делении, если делимое не имеет разрядов в той части, которая должна быть помещена в DX, регистр DX просто обнуляют.

В случае превышения (*overflow*) результатом деления размера того регистра, куда должен быть помещен этот результат, дальнейшее выполнение программы прерывается операционной системой. При беззнаковом делении на байт максимально допустимое частное составляет 255, при делении на слово - 65535. Ввиду малого диапазона обрабатываемых чисел и самого результата, деление на байт или слово, как операция, поддерживаемая современными процессорами Intel, имеет значение, в основном, для обеспечения совместимости с предыдущими версиями процессоров и соответствующими (устаревшими) программами. Однако, чем меньше размерность делителя, тем быстрее выполняется команда, и деление на байт и слово могут и теперь (при подходящих условиях) сослужить пользу в алгоритмах с большими объемами вычислений и критичных ко времени исполнения. Так, для 386-процессоров выполнение деления на двойное слово требует 38 тактов процессора, на слово - 22 такта, на байт - 14 тактов.

# Демонстрация работы программы

Основные команды в данном эмуляторе:

mov – помещение в регистр значения

add – добавление значения к регистру

sub – вычитание из регистра значения

mul – умножение регистра на значение

div – деление регистра ax на значение

state – просмотр состояния регистров

exit – выход из программы

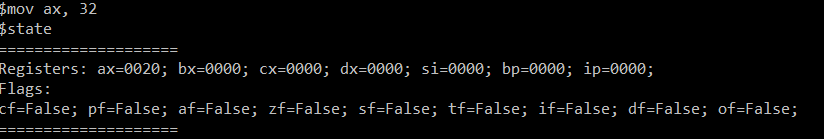


Рис 1. mov – помещение в регистр значения

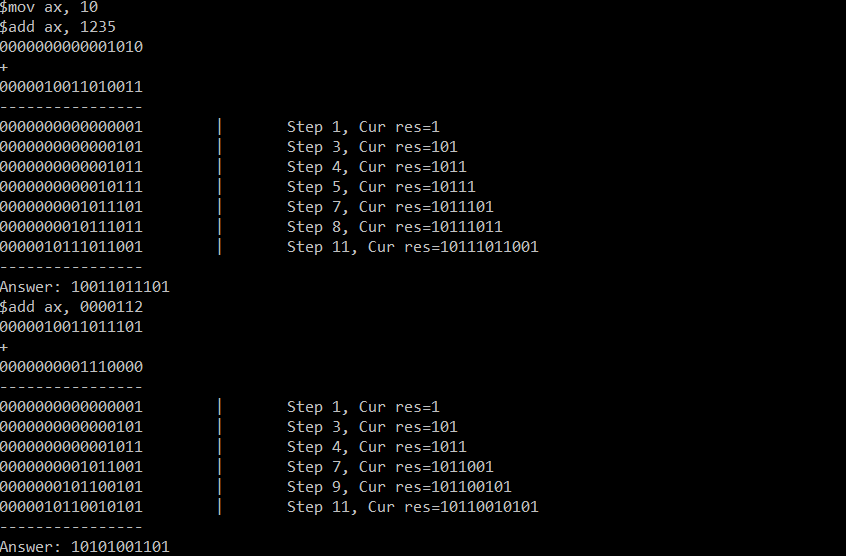


Рис 2. add – добавление значения к регистру

(также есть ввод с нулями вначале числа)

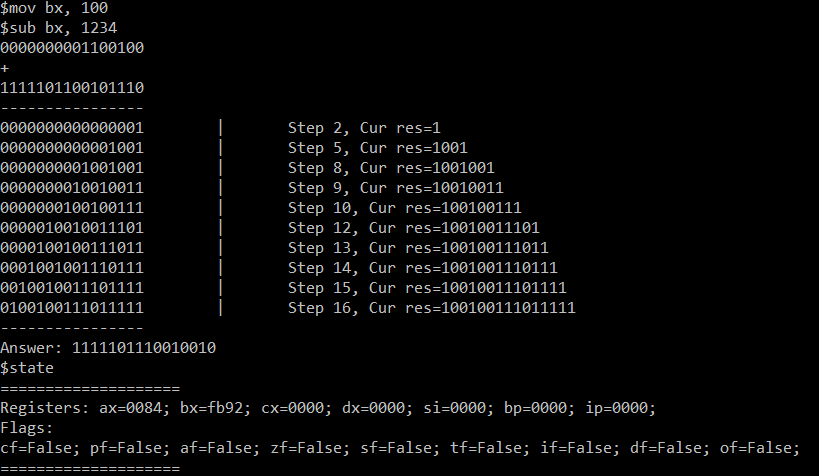


Рис 3. sub – вычитание из регистра значения

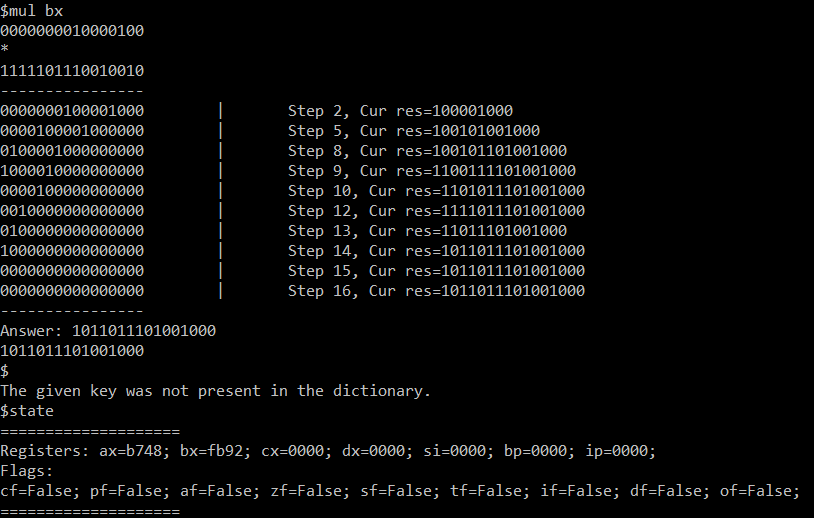


Рис 4. mul – умножение регистра на значение

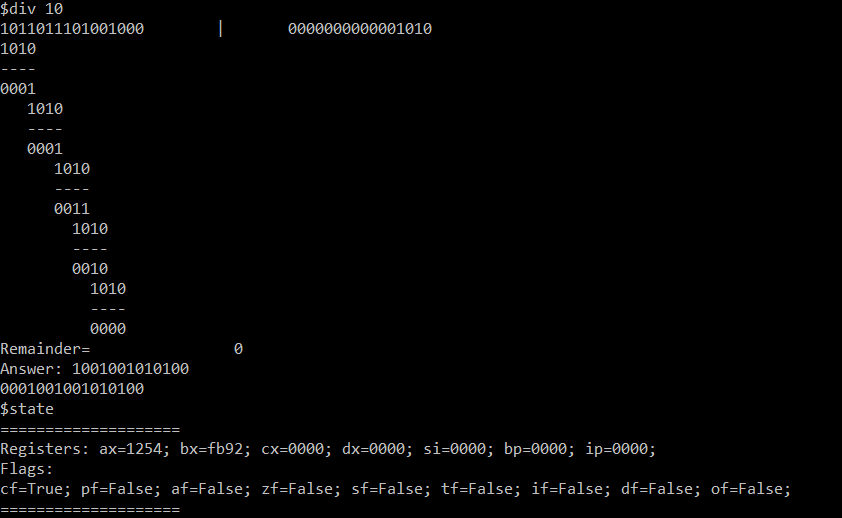


Рис 5. div – деление регистра ax на значение

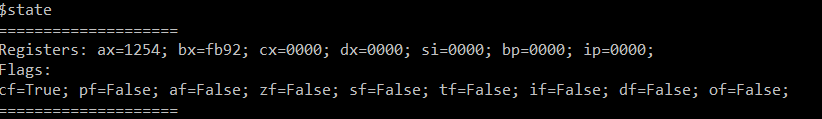


Рис 6. status – просмотр состояния регистров

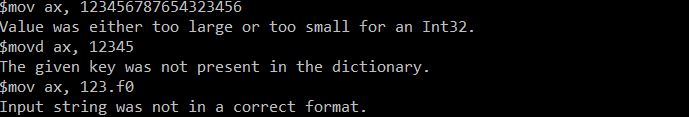


Рис 7. Исключительные ситуации (переполнения и неправильный ввод команд)

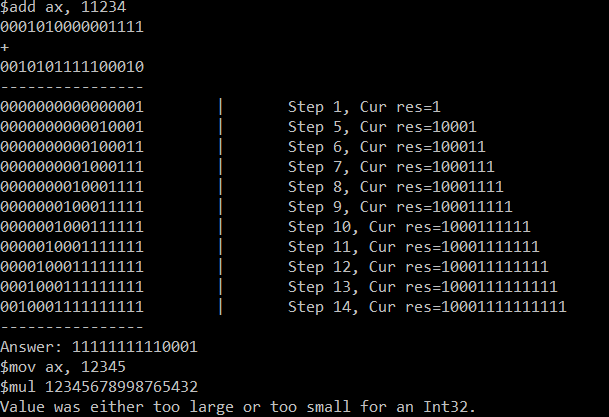


Рис 8. Обработка переполнения при умножении

# Вывод

**Плюсы прямого кода:**

1) Легко получить прямой код числа.

2) Из-за того, что 0 обозначает +, коды положительных чисел относительно беззнакового кодирования остаются неизменными.

3) Количество положительных чисел равно количеству отрицательных.

**Минусы прямого кода:**

1) Существует два нуля: +0(00000000) и -0(10000000), из-за чего усложняется арифметическое сравнение.

**Плюсы обратного кода:**

1. Простое получения кода отрицательных чисел.
2. Из-за того, что 0 обозначает +, коды положительных чисел относительно беззнакового кодирования остаются неизменными.
3. Количество положительных чисел равно количеству отрицательных.

**Минусы обратного кода:**

1) Выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора.

2) Существуют два нуля: +0 и -0.

**Плюсы дополнительного кода:**

1. Возможность заменить арифметическую операцию вычитания операцией сложения и сделать операции сложения одинаковыми для знаковых и беззнаковых типов данных, что существенно упрощает архитектуру процессора и увеличивает его быстродействие.
2. Нет проблемы двух нулей.

**Минусы дополнительного кода:**

1. Ряд положительных и отрицательных чисел несимметричен.
2. В отличие от сложения, числа в дополнительном коде нельзя сравнивать как беззнаковые или вычитать без расширения разрядности.

При сложении чисел просто суммируются их обратные коды (для положительных чисел они равны прямым), включая знаковый разряд.

Вместо операции вычитания производится сложение уменьшаемого с обратным или дополнительным кодом вычитаемого

Умножение производится как последовательность сложений и сдвигов.

Деление для компьютера является трудной операцией. Обычно оно реализуется путем многократного прибавления к делимому дополнительного кода делителя.

# Приложение. Исходный код программы

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\t\tALU Emulator, 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;

}

}

}

}

# }

# //----------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class ALU

{

private string running = "";

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

{

if (running == "")

{

Console.WriteLine(op1);

Console.WriteLine("+");

Console.WriteLine(op2);

Console.WriteLine(new string('-', 16));

}

int len = op1.Length, rem = 0;

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

string res = "";

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

{

int 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, 16) + "\t|\tStep " + (op2.Length - i).ToString() +

", Cur res=" + Interpreter.Normalize(res, 16).TrimStart('0'));

}

}

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

if (rem != 0)

{

flags["cf"] = true;

if (running == "mul")

{

flags["zf"] = true;

flags["of"] = true;

}

}

if (!IsNeg(op1) && !IsNeg(op2))

{

res = "0" + res.Substring(1, len - 1);

}

if (IsNeg(op1) && IsNeg(op2))

{

res = "1" + res.Substring(1, len - 1);

}

if (running == "")

{

Console.WriteLine(new string('-', 16));

Console.WriteLine("Answer: " + Interpreter.Normalize(res, 16).TrimStart('0'));

}

return res;

}

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

{

return Add(op1, Neg(op2, ref flags), ref flags);

}

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

{

running = "mul";

Console.WriteLine(op1);

Console.WriteLine('\*');

Console.WriteLine(op2);

Console.WriteLine(new string('-', 16));

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

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

{

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

{

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

Console.WriteLine(Interpreter.Normalize(op1 + new string('0', op2.Length - i - 1), 16)

+ "\t|\tStep " + (op2.Length - i).ToString() + ", Cur res=" + Interpreter.Normalize(res, 16).TrimStart('0'));

}

}

Console.WriteLine(new string('-', 16));

Console.WriteLine("Answer: " + Interpreter.Normalize(res, 16).TrimStart('0'));

running = "";

return res;

}

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

{

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.ToInt32(tmp, 2) < Convert.ToInt32(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 = Sub(Interpreter.Normalize(tmp, 16), op2, ref flags);

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= " + new string(' ', op1.Length - tmpTrimmed.Length) + tmpTrimmed + '\n');

Console.WriteLine("Answer: " + Interpreter.Normalize(ans, 16).TrimStart('0'));

running = "";

return Interpreter.Normalize(ans, 16);

}

public string Inc(string s, ref Dictionary<string, bool> flags)

{

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

string res = "";

res = Add(s, Interpreter.Normalize("1", 16), ref flags);

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, ref Dictionary<string, bool> flags)

{

string res = "";

foreach (char c in op1)

{

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

else res += '0';

}

return Inc(res, ref flags);

}

}

# }

//-----------------------------------------------------------------------------------------------

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];

}

}

}

//-------------------------------------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public class BinaryRegister

{

public byte Id { get; set; }

public string H { get; set; }

public string L { get; set; }

public BinaryRegister()

{

H = new string('0', 8);

L = new string('0', 8);

Id = 0x00;

}

public BinaryRegister(byte Id)

{

H = new string('0', 8);

L = new string('0', 8);

this.Id = Id;

}

public override string ToString()

{

return H + L;

}

public string Print(int system)

{

switch (system)

{

case 2:

return ToString();

case 8:

return Convert.ToInt16(ToString(), 2).ToString("O");

case 16:

string hex = Convert.ToString(Convert.ToInt16(ToString(), 2), 16);

int d = 4 - hex.Length;

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

{

hex = "0" + hex;

}

return hex;

case 10:

return Convert.ToString(Convert.ToInt16(ToString(), 2), 10);

default:

throw new Exception("Wrong system");

}

}

public static implicit operator BinaryRegister(string s)

{

BinaryRegister r = new BinaryRegister();

s = Interpreter.Normalize(s, 16);

r.H = s.Substring(0, 8);

r.L = s.Substring(8, 8);

return r;

}

}

}

//---------------------------------------------------------------------------------------------------

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;

}

}

}

}

//----------------------------------------------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public enum InstructionType

{

Single,

Double,

Special

}

}

//----------------------------------------------------------------------------

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;

}

}

}

//-----------------------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public static class Memory

{

private static byte[] mem = new byte[maxCapacity];

private const int maxCapacity = 65536;

public static void Reset()

{

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

{

mem[i] = 0;

}

}

public static byte Load(byte segment, byte address)

{

int addr = (int)segment \* 256 + address;

if (addr < 0 || addr >= maxCapacity)

{

throw new Exception("Can't acccess memmory");

}

return mem[addr];

}

public static void SetByte(byte segment, byte address, byte val)

{

int addr = (int)segment \* 256 + address;

if (addr < 0 || addr >= maxCapacity)

{

throw new Exception("Can't acccess memmory");

}

mem[addr] = val;

}

public static void Print()

{

for (int i = 0; i < 256; i++)

{

string s = "";

for (int j = 0; j < 256; j++)

{

s += mem[i \* 256 + j].ToString() + " ";

}

Console.WriteLine(s);

}

}

}

}

//------------------------------------------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public static class Processor

{

private static Dictionary<string, BinaryRegister> registers;

private static Dictionary<string, bool> flags;

//private static Dictionary<string, string> lRegisters;

private static Dictionary<string, string> smallRegisters;

private static ALU alu;

private static ProcessorState state = ProcessorState.NotStarted;

private const int maxBigNum = 32767;

private const int maxSmallNum = 127;

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

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

public static ProcessorState State => state;

public static void Start()

{

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

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

alu = new ALU();

//lRegisters = new Dictionary<string, string>();

smallRegisters = new Dictionary<string, string>();

registers["ax"] = new BinaryRegister(0x01);

registers["bx"] = new BinaryRegister(0x02);

registers["cx"] = new BinaryRegister(0x03);

registers["dx"] = new BinaryRegister(0x04);

registers["si"] = new BinaryRegister(0x05);

registers["bp"] = new BinaryRegister(0x06);

registers["ip"] = new BinaryRegister(0x07);

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;

smallRegisters["al"] = "ax";

smallRegisters["bl"] = "bx";

smallRegisters["cl"] = "cx";

smallRegisters["dl"] = "dx";

smallRegisters["ah"] = "ax";

smallRegisters["bh"] = "bx";

smallRegisters["ch"] = "cx";

smallRegisters["dh"] = "dx";

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;

default:

break;

}

}

public static void Mov(string dest, string source)

{

if (!IsBigRegister(dest))

{

if (IsBigRegister(source))

{

Exit();

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

}

if (IsSmallRegister(dest))

{

BinaryRegister destSmallBinReg = registers[smallRegisters[dest]];

if (IsSmallRegister(source))

{

if (dest[1] == 'l' && source[1] == 'l')

{

destSmallBinReg.L = registers[smallRegisters[source]].L;

}

else if (dest[1] == 'l' && source[1] == 'h')

{

destSmallBinReg.L = registers[smallRegisters[source]].H;

}

else if (dest[1] == 'h' && source[1] == 'l')

{

destSmallBinReg.H = registers[smallRegisters[source]].L;

}

else if (dest[1] == 'h' && source[1] == 'h')

{

destSmallBinReg.H = registers[smallRegisters[source]].H;

}

}

//if number -- add check for a number

else

{

if (Int32.Parse(source) > maxSmallNum)

{

throw new Exception("Constant too large");

}

if (dest[1] == 'l')

{

destSmallBinReg.L = Interpreter.Normalize(Convert.ToString(Int32.Parse(source), 2), 8);

}

else

{

destSmallBinReg.H = Interpreter.Normalize(Convert.ToString(Int32.Parse(source), 2), 8);

}

}

}

else

{

Exit();

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

}

return;

}

BinaryRegister destReg = registers[dest];

//add hex numbers

if (IsBigRegister(source))

{

BinaryRegister sourceReg = registers[source];

destReg.H = sourceReg.H;

destReg.L = sourceReg.L;

registers[dest] = destReg;

}

else

{

int val = Int32.Parse(source);

if (val > maxBigNum)

{

throw new Exception("Constant too large");

}

string binSource = Interpreter.Normalize(Convert.ToString(Int32.Parse(source), 2), 16);

destReg.H = binSource.Substring(0, 8);

destReg.L = binSource.Substring(8, 8);

registers[dest] = destReg;

}

}

public static void Add(string dest, string source)

{

string op1 = "", op2 = "";

if (IsBigRegister(dest))

{

op1 = registers[dest].ToString();

if (IsSmallRegister(source))

{

throw new Exception("Operand types do not match");

}

if (IsBigRegister(source))

{

op2 = registers[source].ToString();

}

else

{

op2 = Interpreter.Normalize(Convert.ToString(Convert.ToInt32(source), 2), 16);

//check number

}

registers[dest] = alu.Add(op1, op2, ref flags);

}

else if (IsSmallRegister(dest))

{

BinaryRegister smallReg = registers[smallRegisters[dest]];

string d = "l";

if (IsBigRegister(source))

{

throw new Exception("Operand types do not match");

}

if (dest[1] == 'l')

{

op1 = registers[smallRegisters[dest]].L;

}

else

{

op1 = registers[smallRegisters[dest]].H;

d = "h";

}

if (IsSmallRegister(source))

{

if (source[1] == 'l')

{

op2 = registers[smallRegisters[source]].L;

}

else

{

op2 = registers[smallRegisters[source]].H;

}

}

else

{

op2 = Convert.ToString(Convert.ToInt32(source), 2);

op2 = op2.Substring(op2.Length - 8, 8);

}

if (d == "l")

{

smallReg.L = alu.Add(op1, op2, ref flags);

}

else

{

smallReg.H = alu.Add(op1, op2, ref flags);

}

}

}

public static void Sub(string dest, string source)

{

string op1 = "", op2 = "";

if (IsBigRegister(dest))

{

op1 = registers[dest].ToString();

if (IsSmallRegister(source))

{

throw new Exception("Operand types do not match");

}

if (IsBigRegister(source))

{

op2 = registers[source].ToString();

}

else

{

op2 = Interpreter.Normalize(Convert.ToString(Convert.ToInt32(source), 2), 16);

//check number

}

registers[dest] = alu.Sub(op1, op2, ref flags);

}

else if (IsSmallRegister(dest))

{

BinaryRegister smallReg = registers[smallRegisters[dest]];

string d = "l";

if (IsBigRegister(source))

{

throw new Exception("Operand types do not match");

}

if (dest[1] == 'l')

{

op1 = registers[smallRegisters[dest]].L;

}

else

{

op1 = registers[smallRegisters[dest]].H;

d = "h";

}

if (IsSmallRegister(source))

{

if (source[1] == 'l')

{

op2 = registers[smallRegisters[source]].L;

}

else

{

op2 = registers[smallRegisters[source]].H;

}

}

else

{

op2 = Convert.ToString(Convert.ToInt32(source), 2);

op2 = op2.Substring(op2.Length - 8, 8);

}

if (d == "l")

{

smallReg.L = alu.Sub(op1, op2, ref flags);

}

else

{

smallReg.H = alu.Sub(op1, op2, ref flags);

}

}

}

public static void Mul(string source)

{

string op1 = "", op2 = "";

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

if (IsSmallRegister(source))

{

if (source[1] == 'l') op2 = Interpreter.Normalize(registers[smallRegisters[source]].L, 16);

else op2 = Interpreter.Normalize(registers[smallRegisters[source]].H, 16);

}

else if (IsBigRegister(source))

{

op2 = registers[source].ToString();

}

else

{

op2 = Interpreter.Normalize(Convert.ToString(Convert.ToInt32(source), 2), 16);

}

registers["ax"] = alu.Mul(op1, op2, ref flags);

Console.WriteLine(registers["ax"]);

}

public static void Div(string source)

{

string op1 = "", op2 = "";

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

if (IsSmallRegister(source))

{

if (source[1] == 'l') op2 = Interpreter.Normalize(registers[smallRegisters[source]].L, 16);

else op2 = Interpreter.Normalize(registers[smallRegisters[source]].H, 16);

}

else if (IsBigRegister(source))

{

op2 = registers[source].ToString();

}

else

{

op2 = Interpreter.Normalize(Convert.ToString(Convert.ToInt32(source), 2), 16);

}

registers["ax"] = alu.Div(op1, op2, ref flags);

Console.WriteLine(registers["ax"]);

}

public static bool IsBigRegister(string val)

{

foreach (var r in registers)

{

if (r.Key == val)

{

return true;

}

}

return false;

}

public static bool IsSmallRegister(string val)

{

return smallRegisters.Keys.Contains(val);

}

public static void PrintState()

{

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

Console.Write("Registers: ");

foreach (var reg in registers)

{

Console.Write(reg.Key + "=" + reg.Value.Print(16) + "; ");

}

Console.WriteLine();

Console.WriteLine("Flags: ");

foreach (var flag in flags)

{

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

}

Console.WriteLine("\n" + new string('=', 20));

}

private static bool CheckBig(string num)

{

return Convert.ToInt32(num) > 65535;

}

public static int MaxBig => maxBigNum;

private static bool CheckSmall(string num)

{

return Convert.ToInt32(num) > 255;

}

public static void Reboot()

{

state = ProcessorState.Running;

registers["ax"] = new BinaryRegister(0x01);

registers["bx"] = new BinaryRegister(0x02);

registers["cx"] = new BinaryRegister(0x03);

registers["dx"] = new BinaryRegister(0x04);

registers["si"] = new BinaryRegister(0x05);

registers["bp"] = new BinaryRegister(0x06);

registers["ip"] = new BinaryRegister(0x07);

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 BinaryRegister(0x01);

registers["bx"] = new BinaryRegister(0x02);

registers["cx"] = new BinaryRegister(0x03);

registers["dx"] = new BinaryRegister(0x04);

registers["si"] = new BinaryRegister(0x05);

registers["bp"] = new BinaryRegister(0x06);

registers["ip"] = new BinaryRegister(0x07);

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;

}

}

}

//---------------------------------------------------------------------------------------

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Emulator

{

public enum ProcessorState

{

NotStarted,

Running,

Sleeping,

Stopped

}

}