Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по лабораторной работе №4

Эмуляция АЛУ. Сложение и вычитание чисел с плавающей запятой

Студент: гр. 853503

Осипчик Тимофей Валентинович

Руководитель: старший преподаватель

Шиманский В.В.

Минск 2020

СОДЕРЖАНИЕ

1. Введение
2. Постановка задачи
3. Программная реализация
4. Выводы

Литература

Приложение 1 – Текст программы

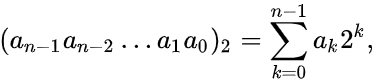
1.Введение

**1.1. Представление целых чисел.**

В двоичной системе счисления числа записываются с помощью двух [символов](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D0%BC%D0%B2%D0%BE%D0%BB) (0 и 1). Чтобы не путать, в какой системе счисления записано число, его снабжают указателем справа внизу. Например, число в десятичной системе 510, в двоичной 1012. Иногда двоичное число обозначают префиксом 0b или символом & (амперсанд), например, 0b101 или соответственно &101.

В двоичной системе счисления (как и в других системах счисления, кроме десятичной) знаки читаются по одному. Например, число 1012 произносится «один ноль один».

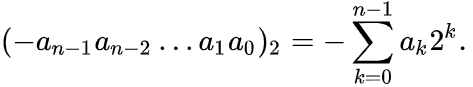
Натуральное число, записываемое в двоичной системе счисления как  {\displaystyle (a\_{n-1}a\_{n-2}\dots a\_{1}a\_{0})\_{2}}(*an – 1 an – 2 ... a1 a0*)2, имеет значение:

{\displaystyle (a\_{n-1}a\_{n-2}\dots a\_{1}a\_{0})\_{2}=\sum \_{k=0}^{n-1}a\_{k}2^{k},}

где:

* {\displaystyle n}*n* — количество [цифр](https://ru.wikipedia.org/wiki/%D0%A6%D0%B8%D1%84%D1%80%D1%8B) (знаков) в числе,
* {\displaystyle a\_{k}}*ak* — цифры из множества {0,1},
* {\displaystyle k}*k* — порядковый номер цифры.

Отрицательные двоичные числа обозначаются так же, как и десятичные: знаком «−» перед числом. А именно, отрицательное целое число, записываемое в двоичной системе счисления {\displaystyle (-a\_{n-1}a\_{n-2}\dots a\_{1}a\_{0})\_{2}}(-*an – 1 an – 2 ... a1 a0*)2, имеет величину:

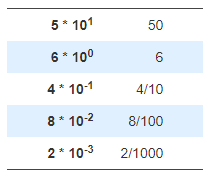


В вычислительной технике широко используется запись отрицательных двоичных чисел в дополнительном коде.

**1.2. Бинарные дроби.**

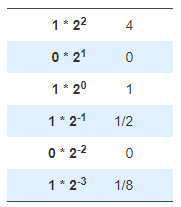
Двоичная система позиционной системы счисления. Это также базовая система счисления. Чтобы узнать больше об этом, прочитайте наше Введение в системы счисления. Когда мы перемещаем позицию (или цифру) влево, мощность, которую мы умножаем основание (2 в двоичном виде), увеличивается на 1. При перемещении вправо мы уменьшаемся на 1 (в отрицательные числа).

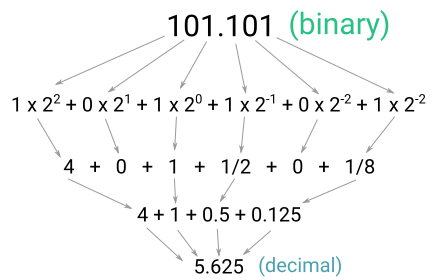
Таким образом, в десятичном числе число 56.482 фактически переводится как:



В двоичном коде это тот же процесс, но вместо этого мы используем степени 2.

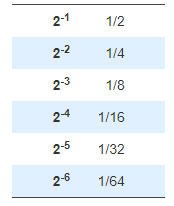
Таким образом, в двоичном виде число 101.101 переводится как:





В десятичном виде это довольно просто, так как мы перемещаем каждую позицию в дроби вправо, мы добавляем 0 к знаменателю. В двоичной системе мы удваиваем знаменатель.

По мере продвижения знаменатель удваивается, поэтому мы получаем следующие значения знаменателя:



и так далее.

Преобразование двоичной дроби в десятичную дробь - это просто вопрос добавления соответствующих значений для каждого бита, который равен 1.

**1.3. Плавающая запятая**

Ранее мы рассмотрели то, что называется двоичными дробями с фиксированной запятой. Это удобный способ представления чисел, но как только число, которое мы хотим представить, будет очень большим или очень маленьким, мы обнаружим, что нам нужно очень большое количество бит для их представления. Если мы хотим представить десятичное значение 128, нам нужно 8 двоичных цифр (10000000). Это более чем вдвое больше цифр для представления одного и того же значения. Это только ухудшается, поскольку мы получаем дальше от нуля.

Чтобы обойти это, мы используем метод представления чисел, называемый плавающей точкой. Плавающая точка очень похожа на научную нотацию как средство представления чисел. Однако мы теряем немного точности при работе с очень большими или очень маленькими значениями, что в целом приемлемо. Здесь я расскажу о стандарте IEEE для чисел с плавающей точкой (так как это в значительной степени де-факто стандарт, которым пользуются все).

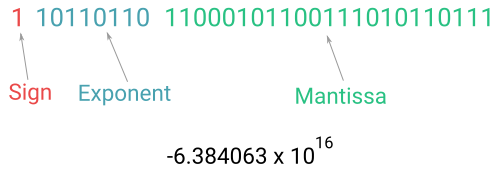
Здесь мы перемещаем не десятичную точку, а двоичную точку, и поскольку она перемещается, она называется плавающей. Ниже мы рассмотрим то, что называется стандартом IEEE 754 для представления чисел с плавающей запятой. Стандарт определяет количество битов, используемых для каждого раздела (экспонента, мантисса и знак) и порядок, в котором они представлены.

Ничто не мешает вам представлять число с плавающей запятой, используя вашу собственную систему, однако почти все используют IEEE 754. Это включает производителей оборудования (включая ЦП) и означает, что схемы для обработки чисел с плавающей запятой IEEE 754 существуют в этих устройствах. Используя стандарт для представления ваших чисел, ваш код может использовать это и работать намного быстрее. Это также означает, что улучшена совместимость, поскольку все представляют числа одинаково.

Стандарт определяет следующие форматы для чисел с плавающей запятой:

Одинарная точность, которая использует 32 бита и имеет следующую компоновку:

* 1 бит для знака числа. 0 означает положительный, а 1 означает отрицательный.
* 8 бит для показателя степени.
* 23 бита для мантиссы.



Двойная точность, которая использует 64 бита и имеет следующую компоновку.

* 1 бит для знака числа. 0 означает положительный, а 1 означает отрицательный.
* 11 бит для показателя степени.
* 52 бита для мантиссы.

например. 0 00011100010 0100001000000000000001110100000110000000000000000000

Двойная точность имеет больше битов, что позволяет представлять гораздо большие и меньшие числа. Поскольку мантисса также больше, степень точности также увеличивается (помните, что многие дроби не могут быть точно представлены в двоичном виде). Хотя числа с плавающей запятой двойной точности имеют эти преимущества, они также требуют большей вычислительной мощности. С увеличением вычислительной мощности ЦП и переходом на 64-разрядные вычисления многие языки программирования и программное обеспечение по умолчанию удваивают точность.

Ниже мы рассмотрим, как работают числа с плавающей запятой одинарной точности (просто потому, что это проще). Двойная точность работает точно так же, только с большим количеством бит.

**1.4 Знаковый бит.**

Это первый бит (самый левый бит) в числе с плавающей запятой, и это довольно просто. Как упоминалось выше, если ваше число положительное, установите этот бит равным 0. Если ваше число отрицательное, тогда сделайте его равным 1.

**1.5 Экспонента.**

Экспонента становится немного интереснее. Помните, что показатель степени может быть положительным (для представления больших чисел) или отрицательным (для представления небольших чисел, то есть дробей). Ваше первое впечатление может показаться, что дополнение двух было бы идеальным здесь, но стандарт имеет немного другой подход. Это сделано, поскольку это облегчает обработку и манипулирование числами с плавающей запятой.

С 8 битами и беззнаковым двоичным кодом мы можем представить числа от 0 до 255. Чтобы учесть отрицательные числа в плавающей запятой, мы берем наш показатель степени и добавляем 127 к нему. Диапазон показателей, которые мы можем представить, становится от 128 до -127. 128 не допускается, однако, и хранится в качестве особого случая для представления определенных специальных чисел, перечисленных ниже.

Например. скажем так:

* Мы хотим, чтобы наш показатель составлял 5,5 + 127 - это 132, поэтому наш показатель становится равным - 10000100
* Мы хотим, чтобы наш показатель был -7. -7 + 127 равно 120, поэтому наш показатель становится - 01111000

Здесь легко запутаться, поскольку знаковый бит для числа с плавающей запятой в целом имеет 0 для положительного и 1 для отрицательного, но это отражается для показателя степени из-за его использования механизма смещения. Это просто то, что вы должны иметь в виду при работе с числами с плавающей запятой.

**1.6 Мантисса.**

В научной нотации помните, что мы перемещаем точку так, что слева от нее остается только одна (не нулевая) цифра. Когда мы делаем это с двоичным кодом, эта цифра должна быть 1, так как другой альтернативы нет. Создатели стандарта с плавающей запятой использовали это в своих интересах, чтобы получить немного больше данных, представленных в числе.

После преобразования двоичного числа в научную запись, перед сохранением в мантиссе мы отбрасываем ведущий 1. Это позволяет нам хранить еще 1 бит данных в мантиссе.

Например, если бы наш номер для хранения был 111.00101101, то в научной нотации это было бы 1.1100101101 с показателем степени 2 (мы переместили двоичную точку на 2 места влево). Мы опускаем ведущий 1. и нужно только хранить 1100101101.

Если бы наш номер для хранения был 0,0001011011, то в научной нотации это было бы 1,011011 с показателем степени -4 (мы переместили двоичную точку на 4 позиции вправо). Мы отбрасываем ведущий 1. и нужно только хранить 011011.

2. Постановка задачи

2.1. Текст задания

Эмуляция АЛУ. Реализовать операцию сложения и вычитания чисел с плавающей запятой.

2.2. Примечание к заданию

Реализовать ввод двух чисел. Вычислить сумму в двоичной системе счисления в дополнительном коде и вывести результат на экран.

3. Программная реализация

3.1. С консоли вводятся два числа в десятичной системе счисления. Затем реализуется перевод их в двоичную систему счисления по стандарту IEEE754. Затем выравниваются экспоненты двух чисел и происходит сложение мантисс в дополнительном коде. Затем на экран выводятся результаты произведения в двоичной и десятичной системах счисления.

Для реализации алгоритма был выбран стандарт IEEE754 для двойной точности, поэтому размерность разрядной сетки 64 бита. Такая точность была выбрана для удобства сравнения результата алгоритма с результатом, полученным средствами выбранного языка программирования, т.к. такая точность дает минимальную погрешность при округлении результата.

3.2. Примеры

3.2.1. Tест для «a = 123456789,987654321, b = 987654321,1234567890»

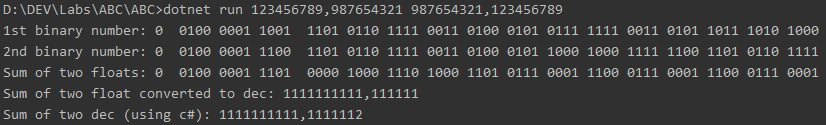


Рисунок 4. Скриншот для «a = 123456789,987654321, b = 987654321,1234567890»

Потеря значимости, из-за сдвига мантиссы при приведении меньшей экспоненты к большей, подробнее в выводе.

Ответ: Результат сложения 1111111111,1111111.

3.2.2. Тест для «a = -123456789,987654321, b = 987654321,123456789»

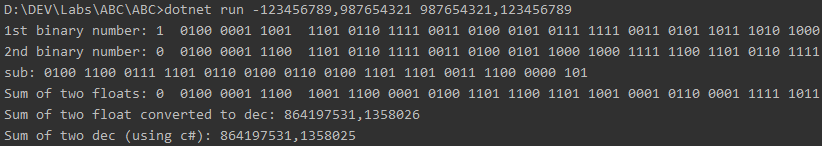


Рисунок 5. Скриншот для «a = -123456789,987654321, b = 987654321,123456789»

Потеря значимости, из-за сдвига мантиссы при приведении меньшей экспоненты к большей, подробнее в выводе.

Ответ: Результат вычитания 864197531,1358026.

3.2.4. Тест для «a = -123456789,987654321, b = -987654321,123456789»

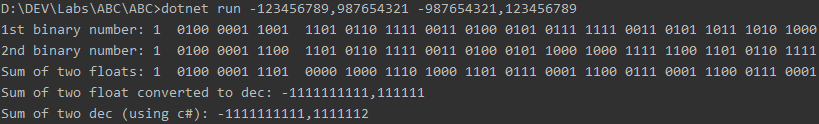


Рисунок 4. Скриншот для «a = -123456789,987654321, b = -987654321,123456789»

Потеря значимости, из-за сдвига мантиссы при приведении меньшей экспоненты к большей, подробнее в выводе.

Ответ: Результат сложения -1111111111,111111.

3.2.5. Тест для «a = -123456789,987654321, b = 0»

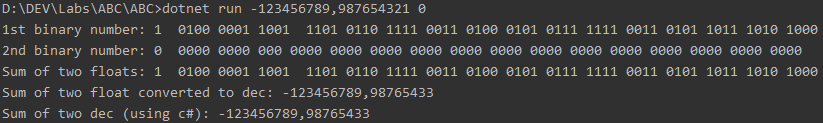


Рисунок 4. Скриншот для «a =-123456789,987654321, b = 0»

Ответ: Результат вычитания -123456789,987654321.

3.2.6. Тест для «a = 0,25, b = 0,25»

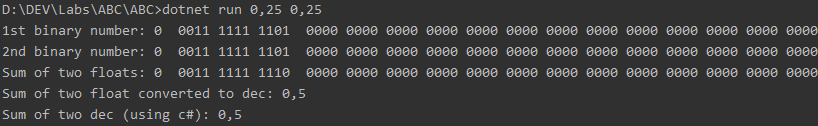


Рисунок 4. Скриншот для «a = 0,25, b = 0,25»

Ответ: Результат сложения 0,5.

3.2.7. Тест для «a = 0,25, b = -0,25»

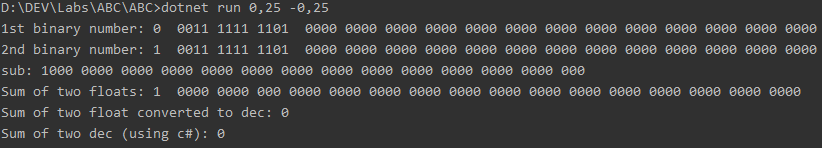


Рисунок 4. Скриншот для «a = 0,25, b = -0,25»

Ответ: Результат вычитания 0.

3.2.8. Тест для «a = 0, b = 0»

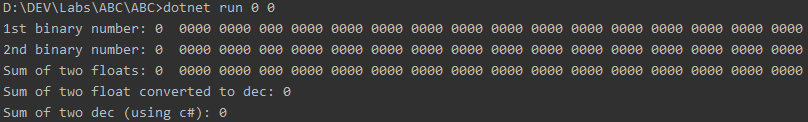


Рисунок 4. Скриншот для «a = 0, b = 0»

Ответ: Результат сложения 0.

3.2.9. Тест для «a = 1,7976931348623157E+308, b = -8,98846567431157E+307»

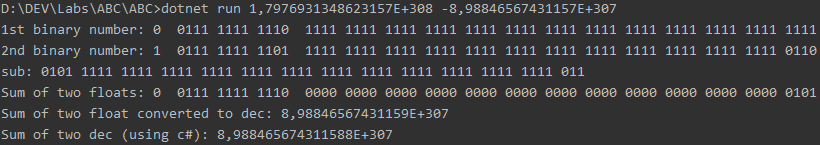


Рисунок 4. Скриншот для «a = 1,7976931348623157E+308, b = -8,98846567431157E+307»

Потеря значимости, из-за сдвига мантиссы при приведении меньшей экспоненты к большей, подробнее в выводе.

Ответ: Результат сложения 8,988465674311588E+307.

3.2.6. Резюме.

В своих примерах я старалась рассмотреть разные случаи. Были рассмотрено сложение двух положительных чисел, двух отрицательных чисел, отрицательного и положительного чисел, сделана проверка экстремальных значений.

4. Выводы

В ходе выполнения лабораторной работы я ознакомилась с понятиями прямого кода, дополнительного кода, сложения в прямом и дополнительном коде, изучила различные алгоритмы представления чисел с плавающей запятой в двоичный код. Также было продемонстрировано, что представление числа с двойной точностью дает незначительное округление результата, либо округление вовсе не происходит.

В примерах 3.2.1, 3.2.2, 3.2.9 произошла потеря точности это обусловлено тем что число округляется при сдвиге экспоненты, также сдвигается и значащая часть – мантисса, также при переводе в десятичный код происходит округление числа.

Так же рассмотрела работу арифметико-логического устройства (АЛУ). Для реализации использовал ОС windows 10, язык С# и интегрированную среду разработки Rider.

Литература

1. RyansTutorials, электронный ресурс, сайт. –URL: <https://ryanstutorials.net/binary-tutorial/binary-floating-point.php>

2. GeeksforGeeks, электронный ресурс, сайт. –URL: <https://www.geeksforgeeks.org/computer-arithmetic-set-2/>

3. Rider, ide, сайт. –URL: <https://www.jetbrains.com/rider/>

4. Учебное пособие / В. М. Яшин. - М.: ИНФРА-М, 2008. - 254 с.

5. Учебное пособие / Б. Н. Ковригин. – М.: МИФИ, 2007. – 40 c.

ПРИЛОЖЕНИЕ 1

Текст программы

Модуль 1. Program.cs

using System;  
using System.Collections.Generic;  
  
namespace ABC  
{  
 public static class ConsoleMain  
 {  
 private enum Operations  
 {  
 **Sum**, Sub, Mul, Div  
 }  
  
 private static void Main(string[] args)  
 {  
 if (!ReadArgs(args, out var numbers))  
 {  
 Console.WriteLine($"Incorrect input: {string.Join(", ", numbers)} it must be 2 floats");   
 return;  
 }  
   
 Console.WriteLine($"1st binary number: {new FloatBinary(numbers[0])}");  
 Console.WriteLine($"2nd binary number: {new FloatBinary(numbers[1])}");  
 try  
 {  
 var asd = Calculate(numbers, Operations.**Sum**);  
 Console.WriteLine($"Sum of two floats: {asd}");  
 Console.WriteLine($"Sum of two float converted to dec: {asd.ToDouble}");  
 Console.WriteLine($"Sum of two dec (using c#): {numbers[0] + numbers[1]}");  
 }  
 catch (Exception e)  
 {  
 Console.WriteLine(e.Message);  
 }  
 }  
   
 private static bool ReadArgs(IEnumerable<string> args, out double[] numbers)  
 {  
 var number = new List<double>();  
 foreach (var s in args) if (double.TryParse(s, out var num)) number.Add(num);  
 numbers = number.ToArray();  
   
 return numbers.Length == 2;  
 }  
   
 private static FloatBinary Calculate(IReadOnlyList<double> numbers, Operations operation)  
 {  
 return operation switch  
 {  
 Operations.**Sum** => new FloatBinary(numbers[0]) + new FloatBinary(numbers[1]),  
 \_ => throw new ArgumentException($"unknown operation: {operation}")  
 };  
 }  
 }

public class Binary  
{  
 private enum BinarySystem  
 {  
 **Base** = 2,  
 **MinLen** = 4,  
 **MaxLength** = 64  
 }  
   
 public IEnumerable<int> Value { get; }  
  
 public Binary(double num)   
 {  
 Value = num < 0 ? ToDirectCode(-num) : ToDirectCode(num);  
 if (num < 0) Value = ToComplementCode();  
 }  
  
 public Binary(IEnumerable<int> binary)  
 {  
 Value = binary;  
 }  
   
 private static IEnumerable<int> ToDirectCode(double num)  
 {  
 var binary = new List<int>();  
 while (num > 0)  
 {  
 num /= (int) BinarySystem.**Base**;  
 binary.Insert(0, Convert.ToInt32(num > Math.Truncate(num)));  
 num = Math.Truncate(num);  
 }  
  
 return NormalizeNumber(binary);  
 }  
  
 private static IEnumerable<int> NormalizeNumber(IEnumerable<int> binary)  
 {  
 var number = binary.ToList();  
  
 return number.Count >= (int) BinarySystem.**MaxLength** ? number   
 : number.ExpandBegin(Math.Abs(number.Count % (int) BinarySystem.**MinLen** - (int) BinarySystem.**MinLen**));  
 }  
   
 public Binary ToInvertCode()  
 {  
 var binary = new int[Value.Count()];  
 var count = 0;  
 foreach (var i in Value) binary[count++] = i == 1 ? 0 : 1;  
   
 return new Binary(binary);  
 }  
  
 public IEnumerable<int> ToComplementCode()  
 {  
 var code = this;  
 if (Value.First() != 1)  
 {  
 code = ToInvertCode() + new Binary(1);  
 code.Value.ToList()[0] = 1;  
 }  
   
 return code.Value;  
 }  
  
 public Binary ComplementToDirect => Value.First() != 1 ? this : (this + new Binary(-1)).ToInvertCode();  
   
 public static Binary operator +(Binary binary1, Binary binary2)  
 {  
 var (item1, item2) = NormalizeLists(binary1, binary2);  
 var res = CalculateSum(item1, item2, out \_).ToList();  
  
 return new Binary(res);  
 }  
  
 public static Binary Add(Binary binary1, Binary binary2)  
 {  
 var (item1, item2) = NormalizeLists(binary1, binary2);  
 var res = CalculateSum(item1, item2, out var addToNext).ToList();  
 if (addToNext != 0)  
 {  
   
 }  
   
 return new Binary(res);  
 }  
   
 public static Binary operator \*(Binary binary1, Binary binary2)  
 {  
 var (item1, item2) = NormalizeLists(binary1.ComplementToDirect, binary2.ComplementToDirect);  
 var res = new Binary(new List<int>().ExpandBegin(item1.Count));  
 var shift = 0;  
 item2.Reverse();  
   
 foreach (var i in item2)  
 {  
 if (i != 0)  
 {  
 var shifted = item1.ExpandEnd(shift);  
 res += new Binary(shifted);  
 }  
 shift++;  
 }  
  
 return NormalizeMul(binary1, binary2, new Binary(NormalizeNumber(res.Value)));  
 }  
  
 private static Binary NormalizeMul(Binary binary1, Binary binary2, Binary result)  
 {  
 if (binary1.Value.First() == binary2.Value.First()) result = result.ComplementToDirect;  
 else if (binary1.Value.First() != binary2.Value.First() && result.Value.First() != 1)  
 result = new Binary(result.ToComplementCode());  
  
 return result;  
 }  
  
 public static Binary operator /(Binary binary1, Binary binary2)  
 {  
 if (!SqueezeBin(binary2.Value).Any()) throw new DivideByZeroException("Attempted to divide by zero.");  
   
 var sign = binary1.Value.First() == binary2.Value.First();  
 var divider = binary2.Value.First() != 1 ? new Binary(binary2.ToComplementCode()) : binary2;  
 var res = GetIntPath(binary1.ComplementToDirect.Value, divider, out \_);  
 var asd = sign ? NormalizeNumber(SqueezeBin(res)) : new Binary(res).ToComplementCode();  
   
 return new Binary(asd);  
 }  
  
 private static IEnumerable<int> GetIntPath(IEnumerable<int> dividendNum, Binary divider, out List<int> dividend)  
 {  
 var res = new List<int>();  
 dividend = new List<int>();  
 foreach (var i in dividendNum)  
 {  
 dividend.Add(i);  
 var remains = new Binary(NormalizeNumber(dividend)) + divider;  
 if (remains.Value.First() != 1)  
 {  
 dividend = remains.Value.ToList();  
 res.Add(1);  
 }  
 else res.Add(0);  
 }  
  
 return res;  
 }  
  
 private static IEnumerable<int> SqueezeBin(IEnumerable<int> bin)  
 {  
 var list = bin.ToList();  
 var index = list.IndexOf(1);  
 list.RemoveRange(0, index > 0 ? index : list.Count);  
   
 return list;  
 }  
   
 private static List<int> NormalizeBinary(Binary binary, int lenSub)  
 {  
 var bin = binary.Value.ToList();  
   
 return bin.ExpandBegin(lenSub, bin[0] == 1 ? 1 : 0);  
 }  
  
 private static (List<int>, List<int>) NormalizeLists(Binary binary1, Binary binary2)  
 {  
 var bin1 = binary1.Value.ToList();  
 var bin2 = binary2.Value.ToList();  
 bin1 = bin1.Count < bin2.Count ? NormalizeBinary(binary1, bin2.Count - bin1.Count) : bin1;  
 bin2 = bin2.Count < bin1.Count ? NormalizeBinary(binary2, bin1.Count - bin2.Count) : bin2;  
  
 return (bin1, bin2);  
 }  
  
 private static IEnumerable<int> CalculateSum(IList<int> list1, IList<int> list2, out int addToNext)  
 {  
 addToNext = 0;  
 for (var i = list1.Count - 1; i >= 0; i--)  
 {  
 list1[i] = Add(list1[i] += list2[i] + addToNext, out var add);  
 addToNext = add;  
 }  
 if (list1.Count >= (int) BinarySystem.**MaxLength** && addToNext != 0) throw new OverflowException("Overflow");  
   
 return list1;  
 }  
   
 private static int Add(int binItem, out int add)  
 {  
 add = 0;  
 if (binItem >= (int) BinarySystem.**Base**)  
 {  
 binItem -= (int) BinarySystem.**Base**;  
 add = 1;  
 }  
  
 return binItem;  
 }  
  
 public double ToDouble()  
 {  
 var res = 0d;  
 var bin = (new Binary(Value)).ComplementToDirect.Value;  
 var enumerable = bin as int[] ?? bin.ToArray();  
   
 var pow = enumerable.Length - 1;  
 foreach (var i in enumerable)  
 {  
 res += i \* Math.Pow(2, pow);  
 pow--;  
 }  
  
 return Value.First() == 1 ? -res : res;  
 }  
   
 public override string ToString()  
 {  
 return Regex.Replace(string.Join("", Value), "**.{4}**", "$0 ");  
 }  
}

public class FloatBinary  
{  
 private enum BinaryConstants  
 {  
 **Base** = 2,  
 **Exponent** = 11,  
 **Mantissa** = 52,  
 **Exp** = 1023  
 }  
   
 private static readonly int[] ZeroExp = new int[(int) BinaryConstants.**Exponent**];  
 private static readonly int[] ZeroMantissa = new int[(int) BinaryConstants.**Mantissa**];  
  
 private bool \_isNegative;  
 private IEnumerable<int> \_exponent;  
 private IEnumerable<int> \_mantissa;  
  
 public FloatBinary(double number)  
 {  
 \_isNegative = number < 0;  
 if (Math.Abs(number) > 0)  
 {  
 var num = Math.Truncate(number);  
 var wholeNum = new Binary(Math.Abs(num)).Value;  
 var fraction = GetFraction(number - num);  
 CreateNum(wholeNum, fraction);  
 }  
 else  
 {  
 \_exponent = ZeroExp;  
 \_mantissa = ZeroMantissa;  
 }  
 }  
  
 private FloatBinary(bool isNegative, IEnumerable<int> exponent, IEnumerable<int> mantissa)  
 {  
 \_isNegative = isNegative;  
 \_exponent = exponent;  
 \_mantissa = mantissa;  
 }  
  
 private static IEnumerable<int> GetFraction(double number)  
 {  
 var num = Math.Abs(number);  
 var mantissa = new List<int>();  
 while (mantissa.Count != (int) BinaryConstants.**Exp**)  
 {  
 num \*= (int) BinaryConstants.**Base**;  
 mantissa.Add(num >= 1 ? 1 : 0);  
 num -= Math.Truncate(num);  
 }  
   
 return mantissa;  
 }  
  
 private void CreateNum(IEnumerable<int> wholeNum, IEnumerable<int> fraction)  
 {  
 var num = wholeNum.ToArray();  
 var bin = new List<int>(num);  
 bin.AddRange(fraction);  
 var spaces = bin.IndexOf(1) + 1;  
 \_exponent = (new Binary(num.Length - spaces) + new Binary((int) BinaryConstants.**Exp**)).Value;  
 \_mantissa = ExpandMantissa(bin.ToList().GetRange(spaces, bin.Count - spaces));  
 }  
  
 private static IEnumerable<int> ExpandMantissa(IEnumerable<int> mantissa)  
 {  
 var m = mantissa.ToList();  
 var count = m.Count;  
 if (count < (int) BinaryConstants.**Mantissa**) m.AddRange(new int[(int) BinaryConstants.**Mantissa** - m.Count]);   
  
 return m.GetRange(0, (int) BinaryConstants.**Mantissa**).ToArray();  
 }  
  
 public static FloatBinary operator +(FloatBinary fb1, FloatBinary fb2)  
 {  
 if (fb1.CheckZero()) return fb2;  
 if (fb2.CheckZero()) return fb1;  
 NormalizeNums(ref fb1, ref fb2, out var expDif);  
  
 return AddSignificands(fb1, fb2, expDif);  
 }  
  
 private static (List<int> m1, List<int> m2) Prepare(FloatBinary fb1, FloatBinary fb2, double expDif) =>  
 expDif.CompareTo(0) switch  
 {  
 1 => ExpandMantissas(fb1, fb2, new[] {1, 0}),  
 -1 => ExpandMantissas(fb1, fb2, new[] {0, 1}),  
 \_ => ExpandMantissas(fb1, fb2, new[] {1, 1})  
 };  
  
 private static (List<int> m1, List<int> m2) ExpandMantissas(FloatBinary fb1, FloatBinary fb2, int[] insert)  
 {  
 var m1 = fb1.\_mantissa.ToList();  
 var m2 = fb2.\_mantissa.ToList();  
 m1.InsertRange(0, new []{0, insert[0]});  
 m2.InsertRange(0, new []{0, insert[1]});  
  
 return (m1, m2);  
 }  
  
 private static FloatBinary AddSignificands(FloatBinary fb1, FloatBinary fb2, double expDif)  
 {  
 var exp = fb1.\_exponent;  
 var (m1, m2) = Prepare(fb1, fb2, expDif);  
 var sign = fb1.\_isNegative;  
 if (!fb1.\_isNegative && fb2.\_isNegative || fb1.\_isNegative && !fb2.\_isNegative)  
 {  
 var firstBigger = Math.Abs(fb1.ToDouble) > Math.Abs(fb2.ToDouble);  
 sign = firstBigger ? fb1.\_isNegative : fb2.\_isNegative;  
 GetComplementCode(ref firstBigger ? ref m1 : ref m2);  
 }  
   
 var res = (new Binary(m1) + new Binary(m2)).Value.ToList();  
  
 if(!fb1.\_isNegative && !fb2.\_isNegative || fb1.\_isNegative && fb2.\_isNegative) NormalizePositive(ref exp, ref res);  
 else NormalizeNegative(ref exp, ref res);  
   
 return new FloatBinary(sign, exp, res);  
 }  
  
 private static void GetComplementCode(ref List<int> m)  
 {  
 m = new Binary(m).ToComplementCode().ToList();  
 m.Insert(0, 0);  
 }  
  
 private static void NormalizePositive(ref IEnumerable<int> exp, ref List<int> res)  
 {  
 if (res[0] == 1)  
 {  
 exp = IncrementExponent(exp, 1);  
 res.RemoveAt(0);  
 res.RemoveAt(res.Count - 1);  
 }  
 else if (res[1] == 1) res.RemoveRange(0, 2);  
 }  
  
 private static void NormalizeNegative(ref IEnumerable<int> exp, ref List<int> sub)  
 {  
 Console.WriteLine($"sub: {new Binary(sub)}");  
 if (sub[0] == 0 && sub[1] == 1)  
 {  
 sub.RemoveAt(0);  
 sub = new Binary(sub).ComplementToDirect.Value.ToList();  
 }  
 sub.RemoveAt(0);  
 exp = IncrementExponent(exp, 1);  
 var index = sub.IndexOf(1);  
 if (index >= (int) BinaryConstants.**Mantissa** || index < 0)  
 {  
 exp = ZeroExp;  
 sub = ZeroMantissa.ToList();  
 }  
 else  
 {  
 exp = DecrementExponent(exp, index+1);  
 sub.RemoveRange(0, index + 1);  
 sub.AddRange(new int[(int) BinaryConstants.**Mantissa** - sub.Count]);  
 }  
 }  
   
 private static void NormalizeNums(ref FloatBinary fb1, ref FloatBinary fb2, out double expDiff)  
 {  
 expDiff = fb1.GetExponent - fb2.GetExponent;  
 if (expDiff > 0) fb2 = MakeExponentsEqual(fb2, (int) expDiff);  
 else if (expDiff < 0) fb1 = MakeExponentsEqual(fb1, (int) expDiff);  
 }  
  
 private static FloatBinary MakeExponentsEqual(FloatBinary floatBinary, int difference)  
 {  
 difference = Math.Abs(difference);  
 var exponent = floatBinary.\_exponent;  
 var mantissa = floatBinary.\_mantissa.ToList();  
 if (difference >= 0)  
 {  
 exponent = IncrementExponent(exponent, difference);  
 mantissa = ShiftMantissa(mantissa, difference);  
 mantissa.RemoveAt(0);  
 }  
   
 return new FloatBinary(floatBinary.\_isNegative, exponent, mantissa);  
 }  
  
 private static List<int> ShiftMantissa(List<int> mantissa, int count, bool addOne = true)  
 {  
 if (count < (int) BinaryConstants.**Mantissa**)  
 {  
 if (addOne) mantissa.Insert(0, 1);  
 mantissa.RemoveRange(mantissa.Count - 1 - count, count);  
 mantissa.InsertRange(0, new int[count]);  
 }  
 else mantissa = new List<int>(new int[(int) BinaryConstants.**Mantissa** + 1]);  
  
 return mantissa;  
 }  
  
 private static IEnumerable<int> IncrementExponent(IEnumerable<int> exponent, int count)  
 {  
 var exp = exponent.ToList();  
 exp = (new Binary(exp) + new Binary(count)).Value.ToList();  
 if (exp[0] != 0) throw new OverflowException("exponent overflow");  
   
 return exp;  
 }  
  
 private static IEnumerable<int> DecrementExponent(IEnumerable<int> exponent, int count)  
 {  
 var exp = exponent.ToList();  
 exp = (new Binary(exp) + new Binary(-count)).Value.ToList();  
  
 return exp;  
 }  
   
  
 public override string ToString()  
 {  
 var exp = Regex.Replace(string.Join("", \_exponent), "**.{4}**", "$0 ");  
 var mantissa = Regex.Replace(string.Join("", \_mantissa), "**.{4}**", "$0 ");  
  
 return $"{Convert.ToInt16(\_isNegative)} {exp} {mantissa}";  
 }  
  
 private double GetExponent => (new Binary(\_exponent) + new Binary(-(int) BinaryConstants.**Exp**)).ToDouble();  
   
 private bool CheckZero()  
 {  
  
 return \_exponent.Compare(ZeroExp) && \_mantissa.Compare(ZeroMantissa);  
 }  
   
 public double ToDouble => CheckZero() ? 0 : ConvertToDouble();  
  
 private double ConvertToDouble()  
 {  
 var exponent = GetExponent;  
 var res = 1.0;  
 var pow = -1;  
 foreach (var i in \_mantissa)  
 {  
 if (i == 1) res += Math.Pow((int) BinaryConstants.**Base**, pow);  
 pow--;  
 }  
 res \*= Math.Pow((int) BinaryConstants.**Base**, exponent);  
  
 return \_isNegative ? -res : res;  
 }  
  
 public static explicit operator Binary(FloatBinary b)  
 {  
 var bin = new List<int> {Convert.ToInt16(b.\_isNegative)};  
 bin.AddRange(b.\_exponent);  
 bin.AddRange(b.\_mantissa);  
   
 return new Binary(bin);  
 }  
}

public static class Extensions  
{  
 public static List<int> ExpandBegin(this IEnumerable<int> list, int discharge, int value = 0)  
 {  
 var expandedList = GenerateList(discharge, value).ToList();  
 expandedList.AddRange(list);  
 return expandedList;  
 }  
   
 public static IEnumerable<int> ExpandEnd(this IEnumerable<int> list, int discharge, int value = 0)  
 {  
 var expandedList = new List<int>(list);  
 expandedList.AddRange(GenerateList(discharge, value));  
 return expandedList;  
 }  
  
 public static bool Compare(this IEnumerable<int> thisEnumerable, IEnumerable<int> enumerable)  
 {  
 var ints = thisEnumerable.ToArray();  
 var second = enumerable.ToArray();  
   
 var result = ints.Union(second).Where(w => !(ints.Contains(w) && second.Contains(w)));  
 return !result.Any();  
 }  
  
 private static IEnumerable<int> GenerateList(int size, int value)  
 {  
 var arr = new int[size];  
 for (var i = 0; i < arr.Length; i++)  
 {  
 arr[i] = value;  
 }  
  
 return arr;  
 }  
}

}