Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ»

Кафедра информатики

Отчет по лабораторной работе №6

Эмуляция АЛУ. Операция деления чисел с плавающей точкой

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. Плавающая запятая**

Двоичные дроби 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).

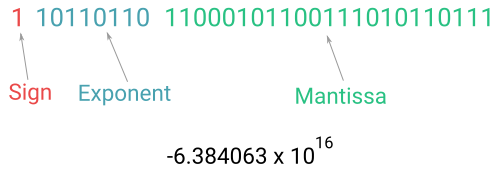
Здесь мы перемещаем не десятичную точку, а двоичную точку, и поскольку она перемещается, она называется плавающей. Ниже мы рассмотрим то, что называется стандартом IEEE 754 для представления чисел с плавающей запятой. Стандарт определяет количество битов, используемых для каждого раздела (экспонента,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

**1.2 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.

**1.3 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.

**1.4 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.

**1.5 Умножение чисел в форме с плавающей запятой**

Умножение чисел, представленных в форме с плавающей запятой, выполняется за четыре шага.

* Определяется знак произведения путем сложения по модулю два знаков сомножителей.
* Определяется порядок произведения путем алгебраического сложения (с учетом знаков) порядков сомножителей по правилам сложения чисел с фиксированной запятой.
* Определяется мантисса произведения путем умножения мантисс сомножителей по изложенным выше правилам для чисел с фиксированной запятой.
* Нормализуется результат умножения мантисс сомножителей, если произошла денормализация.

Пример.

Множимое  .

Множитель  .

Требуется найти  т.е. найти  и  .

Первый шаг. Определение знака произведения:

 .

Второй шаг. Определение порядка произведения путем сложения порядков сомножителей в модифицированном дополнительном коде:



Переполнение не учитывается



Третий шаг.

Определение модуля мантиссы произведения │mx│\*│my│=│mz│:

Произведение равно:

Ограничивая результат шестью разрядами, имеем ненормализованное значение модуля мантиссы произведения  .

Четвертый шаг. Нормализация результата умножения мантисс сомножителей путем сдвига модуля мантиссы произведения на один разряд влево и уменьшения порядка на единицу:

 .

Результат операции умножения  :

 .

 При умножении могут возникать случаи переполнение разрядной сетки периода (см.рис 8, рис. 9). А также отсечения 23 бит мантиссы, полученной при умножении двух исходных мантисс (см. рис. 5, рис. 6, рис 7).

**1.6 Методы ускоренного выполнения операции умножения Двоичных чисел**

По данным статистики, при выполнении инженерных и экономических расчетов из общего количества операций, выполняемых ЭВМ, на долю умножения приходится 30 % и более. В связи с тем, что на выполнение операции умножения тратится значительно больше времени, чем на логические операции и операцию сложения (вычитания), то в общем балансе времени доля, приходящаяся на умножение, оказывается еще более высокой. Поэтому ускорение выполнения операции умножения существенным образом сказывается на увеличении производительности ЭВМ.

На практике нашли применение различные методы ускорения выполнения операции умножения, которые можно разделить на логические, аппаратурные и комбинированные аппаратурно-логические методы.

Под логическими понимаются методы ускорения выполнения операции умножения, связанные в основном с некоторым усложнением схемы управления и не приводящие практически к изменению общей структуры арифметического устройства.

Под аппаратурными понимаются методы, связанные с существенным увеличением аппаратурного состава арифметического устройства и изменением его общей первоначальной структуры. Аппаратно-логические методы занимают промежуточное положение.

Логические методы ускоренного выполнения операции умножения делятся на две группы:

- методы, позволяющие уменьшить количество суммирований в процессе умножения;

- методы, позволяющие за один цикл умножения обработать несколько разрядов множителя путем некоторого усложнения схемы и введения в регистры дополнительных цепей сдвига.

Последние из указанных методов можно разделить на синхронные и асинхронные. Синхронные методы характеризуются тем, что в каждом цикле умножения обрабатывается фиксированное число (два и более) разрядов множителя, то есть длина цикла умножения постоянная.

При реализации асинхронных методов ускорения операции умножения количество разрядов множителя, обрабатываемых в каждом цикле умножения, зависит от комбинации разрядов множителя, анализируемых в данном цикле, то есть длина цикла умножения переменная.

Аппаратурные методы ускоренного выполнения операции умножения делятся на следующие группы:

- схемотехнические методы ускорения выполнения операций сложения и сдвига;

- метод, основанный на использовании при выполнении операции умножения прямого вычитания;

- метод, основанный на введении дополнительных цепей сдвига, позволяющих за один такт производить сдвиг информации в регистрах на несколько разрядов;

- схемотехнический метод совмещения по времени операций сложения и сдвига.

Ускорение выполнения операции сложения обычно достигается увеличением скорости работы логических элементов и триггеров, а также ускорением распространения переносов.

Для одновременной обработки нескольких разрядов множителя за один цикл умножения в регистры вводят дополнительные цепи, предназначенные для сдвига множителя и частичных произведений (или множимого) сразу на такое количество разрядов, сколько обрабатывается в данном цикле умножения.

Совмещение по времени операций сложения и сдвига может быть достигнуто соответствующим усложнением схемы сумматора. Введением дополнительных схем передачи результата суммирования в регистр суммы можно совмещать сложение со сдвигом одновременно на несколько разрядов.

Логические методы ускоренного выполнения операции умножения чаще всего применяются в арифметических устройствах параллельного действия. При этом наиболее широко используются метод пропуска такта суммирования, метод анализа сомножителей для определения сомножителя (множителя) с наименьшим числом единиц, метод расшифровки и одновременного умножения на два и более разрядов множителя и другие. Рассмотрим некоторые из этих методов.

2. Постановка задачи

2.1. Текст задания

Эмуляция АЛУ. Реализовать операцию деления целых чисел.

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. После двоичный код преобразуется в дополнительный. В дополнительном коде реализуется деление. Затем на экран выводятся результаты произведения в двоичной и десятичной системах счисления.

3.2. Примеры

3.2.1. Tест для «a = 0,25, b = 0,05»

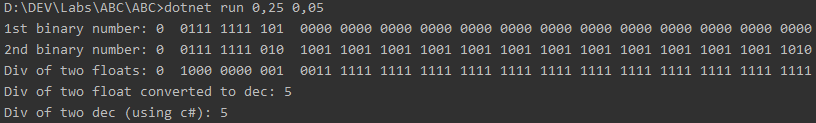


Рисунок 4. Скриншот для «a = 0,25, b = 0,05»

Ответ: Результат деления 5.

3.2.2. Тест для «a = 75,8, b = 0»

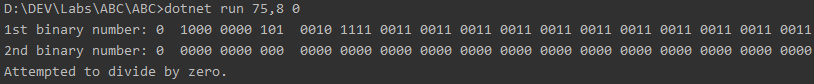


Рисунок 5. Скриншот для «a = 75,8, b = 0»

Ответ: Результат деления попытка делить на 0.

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

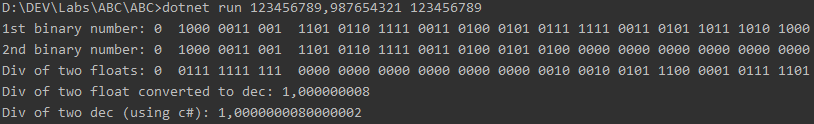


Рисунок 4. Скриншот для «a = 123456789,987654321, b = 123456789»

Потеря значимости при округлении, подробнее в выводе.

Ответ: Результат деления 1,000000008.

3.2.5. Тест для «a = 1,7976931348623157E+306, b = 1,7976931348623157E-306»

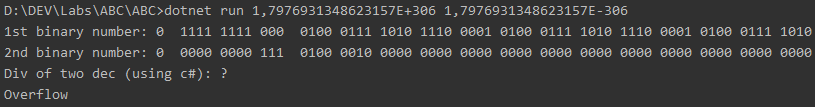


Рисунок 4. Скриншот для «a = 1,7976931348623157E+306, b = 1,7976931348623157E-306»

Ответ: Результат деления переполнение разрядной сетки.

3.2.6. Тест для «a = 5,125, b = -700,000008»

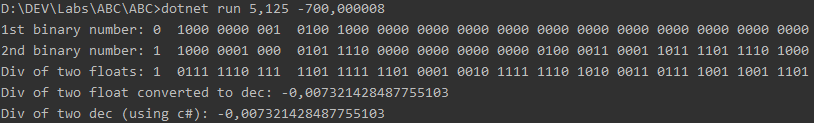


Рисунок 4. Скриншот для «a = 5,125, b = -700,000008»

Ответ: Результат деления -0,007321428487755103.

3.2.7. Тест для «a = -4433333,33333355555, b = -999999,99999999»

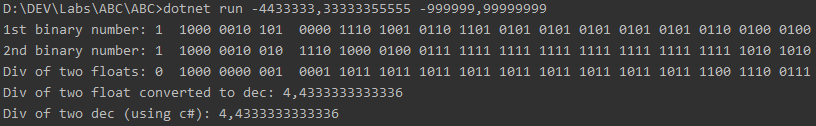


Рисунок 4. Скриншот для «a = -4433333,33333355555, b = -999999,99999999»

Ответ: Результат деления 4,4333333333336.

3.2.8. Тест для «a = 1,7976931348623157E+308, b = 1,7976931348623157E+30»

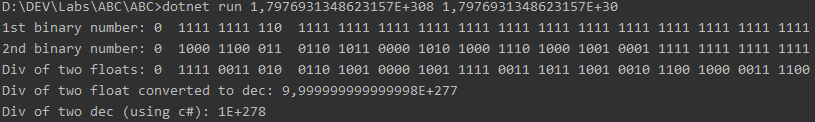


Рисунок 4. Скриншот для «a = 1,7976931348623157E+308, b = 1,7976931348623157E+30»

Потеря значимости при округлении, подробнее в выводе.

Ответ: Результат деления 9,999999999999998E+277.

3.2.9. Тест для «a = -1,79E+300, b = 0,25»

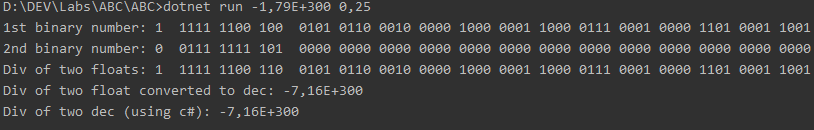


Рисунок 4. Скриншот для «a = -1,79E+300, b = 0,25»

Ответ: Результат деления -7,16E+300.

3.2.10. Тест для «a = -1,79, b = 1,79E-300»

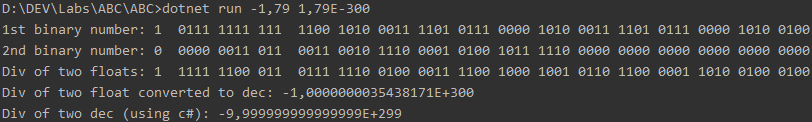


Рисунок 4. Скриншот для «a = -1,79, b = 1,79E-300»

Потеря значимости, подробнее в выводе.

Ответ: Результат деления -1,0000000035438171E+300.

3.2.11. Резюме.

В своих примерах я старался рассмотреть разные случаи. Было рассмотрено умножение 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, умножения в прямом коде, изучил различные алгоритмы представления чисел с плавающей запятой в двоичном коде. Также было продемонстрировано, что представление числа с двойной точностью дает незначительное округление результата, либо округление вовсе не происходит.

В примерах 3.2.4, 3.2.5, 3.2.10 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;  
using System.Linq;  
using System.Text.RegularExpressions;  
  
namespace ABC  
{  
 public static class ConsoleMain  
 {  
 private enum Operations  
 {  
 **Sum**, Sub, Mul, Div  
 }  
  
 private static void Main(string[] args)  
 {  
 *// var n1 = 2.555d;  
 // var n2 = 0.00500d;  
 // var asd = new FloatB(n1) / new FloatB(n2);  
 // Console.WriteLine(asd);  
 // Console.WriteLine(asd.ToDouble());  
 // Console.WriteLine("0 1000 0000 0100 0000 0000 0000 0000 000");  
 // Console.WriteLine(n1 / n2);  
 //  
   
 // var asd = new Binary(new []{0, 1,0,0,0, 1,0,0,0}) + new Binary(-1);  
 // Console.WriteLine(asd);  
 // Console.WriteLine(asd.ToDouble());* if (!ReadArgs(args, out var numbers))  
 {  
 Console.WriteLine($"Incorrect input: {string.Join(", ", numbers)} it must be 2 floats");   
 return;  
 }  
   
 Console.WriteLine($"1st binary number: {new FloatB(numbers[0])}");  
 Console.WriteLine($"2nd binary number: {new FloatB(numbers[1])}");  
 try  
 {  
 var asd = Calculate(numbers, Operations.**Sum**);  
 Console.WriteLine($"Div of two floats: {asd}");  
 Console.WriteLine($"Div of two float converted to dec: {asd.ToDouble()}");  
 Console.WriteLine($"Div 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 FloatB Calculate(IReadOnlyList<double> numbers, Operations operation)  
 {  
 return operation switch  
 {  
 Operations.**Sum** => new FloatB(numbers[0]) / new FloatB(numbers[1]),  
 \_ => throw new ArgumentException($"unknown operation: {operation}")  
 };  
 }  
 }

public class FloatB  
{  
 private enum BinaryConstants  
 {  
 **Base** = 2,  
 **Exponent** = 11,  
 **Mantissa** = 52,  
 **Exp** = 1023  
 }private readonly bool \_isNegative;  
 private IEnumerable<int> \_exponent;  
 private IEnumerable<int> \_mantissa;  
 private static IEnumerable<int> zeroExp = new int[(int) BinaryConstants.**Exponent**];  
 private static IEnumerable<int> zeroMan = new int[(int) BinaryConstants.**Mantissa**];  
   
   
 public FloatB(double num)  
 {  
 \_isNegative = num < 0;  
 if (num != 0)  
 {  
 var (exp, mantissa) = CreateNum(new Bin(Math.Truncate(Math.Abs(num))).ValueBin, GetFraction(num));  
 \_exponent = exp;  
 \_mantissa = mantissa;  
 }  
 else  
 {  
 \_exponent = zeroExp;  
 \_mantissa = zeroMan;  
 }  
 }  
  
 private FloatB(bool isNegative, IEnumerable<int> exponent, IEnumerable<int> mantissa)  
 {  
 \_isNegative = isNegative;  
 \_exponent = exponent;  
 \_mantissa = mantissa;  
 }  
  
 private static IEnumerable<int> GetFraction(double number)  
 {  
 number = Math.Abs(number);  
 number -= Math.Truncate(number);  
 var mantissa = new List<int>();  
 while (mantissa.Count != (int) BinaryConstants.**Exp**)  
 {  
 number \*= (int) BinaryConstants.**Base**;  
 mantissa.Add(number >= 1 ? 1 : 0);  
 number -= Math.Truncate(number);  
 }  
  
 return mantissa;  
 }  
   
 private static (IEnumerable<int> exp, IEnumerable<int> mantissa) 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;  
 var exponent = num.Length - spaces + (int) BinaryConstants.**Exp**;  
 var exp = NormalizeBin(new Bin(exponent).ValueBin, (int) BinaryConstants.**Exponent**, false);  
 var mantissa = NormalizeBin(bin.ToList().GetRange(spaces, bin.Count - spaces), (int) BinaryConstants.**Mantissa**);  
  
 return (exp, mantissa);  
 }  
  
 private static IEnumerable<int> NormalizeBin(IEnumerable<int> mantissa, int size, bool fromEnd = true)  
 {  
 var m = mantissa.ToList();  
 if (m.Count < size)  
 {  
 var add = new int[size - m.Count];  
 if (fromEnd) m.AddRange(add);   
 else m.InsertRange(0, add);  
 }  
  
 return m.GetRange(0, size);  
 }  
  
 public static FloatB operator /(FloatB fb1, FloatB fb2)  
 {  
 if (IsZero(fb2))  
 {  
 throw new DivideByZeroException("Attempted to divide by zero.");  
 }  
 if (IsZero(fb1))  
 {  
 return new FloatB(fb1.\_isNegative != fb2.\_isNegative, zeroExp, zeroMan);  
 }  
   
 var exp = CalculateExponent(fb1, fb2).ToList();  
 var mantissa = CalculateMantissa(fb1, fb2).ToList();  
 var (e, m) = NormalizeDiv(exp, mantissa);  
   
 return new FloatB(fb1.\_isNegative != fb2.\_isNegative, e, m);  
 }  
  
 private static IEnumerable<int> CalculateMantissa(FloatB fb1, FloatB fb2)  
 {  
 var m1 = fb1.\_mantissa.ToList();  
 var m2 = fb2.\_mantissa.ToList();  
 m1.InsertRange(0, new [] {0, 1});  
 m2.InsertRange(0, new [] {0, 1});  
 var mantissa = (new Binary(m1) / new Binary(m2)).Value.ToList();  
  
 if (mantissa[0] == 1 && mantissa[1] == 0)  
 {  
 mantissa.RemoveRange(0, 1);  
 mantissa.RemoveAt(mantissa.Count - 1);  
 }  
 else if (mantissa[0] == 0 && mantissa[1] == 1)  
 {  
 mantissa.RemoveRange(0, 2);  
 }  
   
 if (mantissa.Count < (int) BinaryConstants.**Mantissa**)  
 {  
 mantissa.AddRange(new int[(int) BinaryConstants.**Mantissa** - mantissa.Count]);  
 }  
  
 return mantissa;  
 }  
  
 private static IEnumerable<int> CalculateExponent(FloatB fb1, FloatB fb2)  
 {  
 var exp = fb1.GetExponent - fb2.GetExponent + (int) BinaryConstants.**Exp**;  
 var exponent = new Bin(exp).ValueBin.ToList();  
   
 if (exponent.Count > (int) BinaryConstants.**Exponent**)  
 {  
 throw new OverflowException("Overflow");  
 }  
  
 return exponent;  
 }  
  
 private static (IEnumerable<int> e, IEnumerable<int> m) NormalizeDiv(List<int> exp, List<int> mantissa)  
 {  
 if (exp.Count < (int) BinaryConstants.**Exponent**)  
 {  
 exp.InsertRange(0, new int[(int) BinaryConstants.**Exponent** - exp.Count]);  
 }  
  
 CheckUnderflow(exp);  
  
 var dif = (int) BinaryConstants.**Mantissa** - mantissa.Count;  
 if (dif < 0)  
 {  
 exp.Insert(0, 0);  
 exp = (new Binary(exp) + new Binary(dif)).Value.ToList();  
 exp.RemoveAt(0);  
 CheckUnderflow(exp);  
 }  
   
   
 return (exp, mantissa);  
 }  
  
 private static void CheckUnderflow(IEnumerable<int> exp)  
 {  
 var asd = exp.ToList();  
 if (asd.TakeWhile(i => i == 0).Count() == (int) BinaryConstants.**Exponent**)  
 {  
 throw new OverflowException("Underflow");  
 }  
 }  
   
 private double GetExponent => new Bin(\_exponent).ToDouble() - (int) BinaryConstants.**Exp**;  
  
 private static bool IsZero(FloatB fb)  
 {  
 var e = fb.\_exponent.TakeWhile(i => i == 0).Count();  
  
 return e == (int) BinaryConstants.**Exponent**;  
 }  
   
 public double ToDouble()  
 {  
 if (IsZero(this))  
 {  
 return 0;  
 }  
   
 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 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}";  
 }  
}

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 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 divider = binary2.Value.First() != 1 ? new Binary(binary2.ToComplementCode()) : binary2;  
 var res = GetIntPath(binary1.ComplementToDirect.Value, divider, out \_);  
  
 return new Binary(res);  
 }  
  
 private static IEnumerable<int> GetIntPath(IEnumerable<int> dividendNum, Binary divider, out List<int> dividend)  
 {  
 var res = new List<int>();  
 var num = dividendNum.ToList();  
 dividend = new List<int>(num.GetRange(0, divider.Value.Count() - 1));  
   
 var count = dividend.Count;  
 while (CompareExp(res, 54))  
 {  
 dividend.Add(count < num.Count ? num[count] : 0);  
 var remains = (new Binary(NormalizeNumber(dividend)) + divider).Value.ToList();  
 if (remains.First() != 1)  
 {  
 dividend = remains;  
 res.Add(1);  
 }  
 else res.Add(0);  
 count++;  
 }  
  
 return res;  
 }  
  
 private static bool CompareExp(List<int> bin, int exp)  
 {  
 var index = bin.IndexOf(1);  
   
 if (index >= 0)  
 {  
 var len = bin.GetRange(index, bin.Count - index).Count();  
 return len < exp;  
 }  
   
 return true;  
 }  
  
 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;  
 }  
   
 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 override string ToString()  
 {  
 return Regex.Replace(string.Join("", Value), "**.{4}**", "$0 ");  
 }  
}

public class Bin  
{  
 public IEnumerable<int> ValueBin { get; private set; }  
 public bool IsNegative { get; private set; }  
   
 public Bin(double num)  
 {  
 IsNegative = num < 0;  
 ValueBin = ToDirectCode(Math.Abs(num));  
 if (IsNegative)  
 {  
 ValueBin = ToComplementCode();  
 }  
 }  
  
 public Bin(IEnumerable<int> val)  
 {  
 ValueBin = val;  
 }  
  
 public Bin(IEnumerable<int> valueBin, bool isNegative = false)  
 {  
 ValueBin = valueBin;  
 IsNegative = isNegative;  
 }  
   
 private static IEnumerable<int> ToDirectCode(double num)  
 {  
 var binary = new List<int>();  
 while (num >= 1)  
 {  
 num /= 2;  
 binary.Insert(0, num > Math.Truncate(num) ? 1 : 0);  
 num = Math.Truncate(num);  
 }  
  
 if(binary.Count == 0) binary.Add(0);  
 return binary;  
 }  
  
 private static IEnumerable<int> ToInvertCode(IList<int> bin)  
 {  
 for (var i = 0; i < bin.Count; i++)  
 {  
 bin[i] = bin[i] == 1 ? 0 : 1;  
 }  
  
 return bin;  
 }  
  
 private IEnumerable<int> ToComplementCode()  
 {  
 var bin = ToInvertCode(ValueBin.ToList());  
 var res = new Bin(bin, IsNegative) + new Bin(1);  
   
 return res.ValueBin;  
 }  
  
 private static IEnumerable<int> CalculateSum(IList<int> list1, IList<int> list2)  
 {  
 var 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 (addToNext != 0)  
 {  
 list1.Insert(0, addToNext);  
 }  
   
 return list1;  
 }  
   
 private static int Add(int binItem, out int add)  
 {  
 add = 0;  
 if (binItem >= 2)  
 {  
 binItem -= 2;  
 add = 1;  
 }  
  
 return binItem;  
 }  
   
 public static Bin operator +(Bin bin1, Bin bin2)  
 {  
 var (item1, item2) = Prepare(bin1, bin2);  
 var add = CalculateSum(item1, item2);  
   
 return new Bin(add);  
 }  
  
 public static Bin operator \*(Bin bin1, Bin bin2)  
 {  
 var item1 = bin1.ValueBin.ToList();  
 var item2 = bin2.ValueBin.ToList();  
 var res = new Bin(0);  
 var shift = 0;  
 item2.Reverse();  
   
 var items = new List<Bin>();  
  
 foreach (var i in item2)  
 {  
 if (i != 0)  
 {  
 var shifted = item1.ExpandEnd(shift);  
 res += new Bin(shifted);  
 items.Add(new Bin(shifted));  
 }  
 shift++;  
 }  
   
 var asd = new Bin(0);  
 foreach (var bin in items)  
 {  
 asd += bin;  
 }  
   
   
 res.IsNegative = bin1.IsNegative != bin2.IsNegative;  
 return res;  
 }  
   
 private static (List<int> item1, List<int> item2) Prepare(Bin bin1, Bin bin2)  
 {  
 var binVal1 = bin1.ValueBin.ToList();  
 var binVal2 = bin2.ValueBin.ToList();  
  
 var dif = binVal1.Count - binVal2.Count;  
 for (var i = 0; i < dif; i++)  
 {  
 if (dif > 0) binVal2.Insert(0, bin2.IsNegative ? 1 : 0);  
 if (dif < 0) binVal1.Insert(0, bin1.IsNegative ? 1 : 0);  
 }  
  
 return (binVal1, binVal2);  
 }  
  
 public IEnumerable<int> ComplementToDirect => IsNegative switch  
 {  
 true => ToInvertCode((this + new Bin(-1)).ValueBin.ToList()),  
 \_ => ValueBin  
 };  
   
 public double ToDouble()  
 {  
 var res = 0d;  
 var bin = ComplementToDirect;  
 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 IsNegative ? -res : res;  
 }  
   
 public override string ToString()  
 {  
 return Regex.Replace(string.Join("", ValueBin), "**.{4}**", "$0 ");  
 }  
}

}