Lec 2

Кодирование чисел с плавающей точкой

1.2345 * 10^6 нормальная форма числа
1234500 не показательная форма (возникает иллюзия доп точности)

Как закодировать число `5.375` в формате с плавающей точкой:

- кодируем как двоичное: 101.011
- ставим точку после первой единицы и на сколько нужно отступить для исходника: $1.01011 * 2^2$

Мантиса: 1.01011
Экспонента: степень основания с.с.

> Оба числа являются знаковыми

как перевести десятичную в двоичную:
0.375

0.	**375**
1	75
1	5
0	0
0	0

0.1 -> 0x199A

Есть разное представление одного и того же числа на разном железе и будут проблемы кросс-платформы

IEEE-754

> так как почти всегда число с плавающей точкой начинается с 1.{}
> то эту часть можно не хранить (за исключением особых случаев)

Форма: s|exp|mant

- s - знак мантисы (бит под знак)
- exp - экспонента (форма со смещением с округлением сдвига вниз)
- mant - мантиса (скрытно хранится та самая 1.{})

имя	всего бит	exp	mant	пример
half-precision	16	5	10	5.375 -> 0|10001|0101100000 -> 0x4560
single-precision (float в c/cpp)	32	8	23	0.000110011 -> 1.10011..* *2^{-4} -> 0x3DCCCCCD

Самое маленькое положительное число в half-precision -> 1.0000000000 2^{-14}
a = 1.0000000001 2^{-14}
b = 1.0000000000 * 2^{-14}

```
if (a == b) false
a > b true
a-b == 0 true Так как 2^{-24} это уже ноль
```

если все биты экспоненты ноль то это денормализованное число вида

+- 0.mant * 2^{MIN_NORM}

a- b = 0.0000000001 * 2^{-14}

кодируем: 0 00000 0000000001

получаем два нуля. Все нули и знак 1/0

- Убрали багу с разностью соседних
- Получили дополнительные маленькие числа

денормализованные числа не поддерживаются т.е. железо считает это нулем.
Если результат операции денормализованное число то тоже считается нулем.
Просто считаем их все нулями.

Другой вариант, они могут полностью поддерживаться, но очень сильно замедляемся.

Если все биты экспоненты единицы, то
1. mant == 0 -> то это inf

Т.е. если мы получаем очень большое число, то это не модулярная арифметика, а бесконечность

```
1/0 -> inf в плавающей точке
-1/0 -> -inf
-1/-0 -> inf // наличие двух нулей оправдалось
```

2. mant != 0 -> то это зверек NaN (not a number)

- они выживают (NaN * 0 = Nan, sqrt(NaN) = NaN)

Это удобно для индикации ошибок. Деление 0/0 -> NaN, sqrt(-1) -> Nan, +inf - inf -> NaN.
Если появился NaN и ответ не зависит от него, то все хорошо. Иначе получим NaN

- NaN не имеет понятия знака
  Стандарт: NaN со знаком 1 можно выводить как -NaN

Если сделать операцию с двумя NaN, то выживет один из них (По стандарту не определено).

> Не особо логично, потому что одна из ошибок умирает

- Сравнение NaN`ов

Они никогда никому не равны
Проверки >, <, <=, >=, == -> ВСЕГДА false если хоть один аргумент NaN
Только если NaN != Nan -> true

```
float x;
if (x == x) // проверка на то, что x это не NaN (не глупая проверка)
```

Каст NaN в любой другой тип данных - UB (возможно)

NaN в зависимости от старшего битика мантисы:
Кто из них кто стандартом не было огласовано
1. qNaN если бит == 1 (тихий)
Не генерирует исключение
2. sNaN если бит == 0 (сигнальный)
Генерирует исключение

Такое распределение сделано чтобы легко успокоить любой NaN. Если бы наоборот, то есть NaN которого нельзя успокоит так как получим бесконечность

имя	всего бит	exp	mant	пример
half-precision (_Float16)	16	5	10	5.375 -> 0|10001|0101100000 -> 0x4560
single-precision (float в c/cpp)	32	8	23	0.000110011 -> 1.10011..* *2^{-4} -> 0x3DCCCCCD
double (double в c/cpp и long double под компилятором microsoft)	64	11		
guad (редко реализуют в железе) (long double значит он программный значит медленный)	128	15		
extended (long double) (околостандартный тип)	80	15	64 (включая ведущую единицу)	

half precision
Использование более коротких чисел уменьшает энергопотребление и увеличивает скорость но точность становится меньше

имя	всего бит	exp	mant	пример
bfloat / bf16 / E8M7 (single precision от которого отрезали хвост) Активно используется в нейронных сетях. Считает аппаратно .Можно на халяву конвертить в single precision.	16	8	7	
E5M2 / bf8 (half precision от которого тоже отрезали хвост)	8	5	2	
E4M3 / hf8 Есть E4M3FN (FN означает что нет бесконечностей и меньше NaN`ов)	8	4	3	
E4M3FNUZ (UZ - usigned zero, т.е один ноль с плюсом и один NaN в виде нуля с минусом)	8	4	3	
E2M1 (никаких NaN`ов и бесконечностей, зато два нуля:))	4	2	1	
MXE4M3 (числа хранятся блоками по 32 значений, где значений в формате E4ME)				
NVFP4 (блок из 16 E2M1) * E4M3FN				