Теоретические материалы по дисциплине «Научное программирование и машинное обучение»

П. В. Купцов

НИУ ВШЭ НН, 2026, kupav@mail.ru

3 февраля 2026 г.

Содержание

Тема 1. Архитектура ЭВМ с точки зрения программиста

- Центральный процессор и оперативной память
- Порты ввода-вывода

Тема 2. Машинное представление данных

- Стандартные числовые данные
- Беззнаковые целые числа и числа со знаком
- Кодирование беззнаковых целых чисел
- Кодирование целых со знаком
- Примеры работы с отрицательными числами
- Принципиальное отличие машинных вещественных чисел от «математических»
- Вещественные числа с фиксированной точкой
- Кодирование вещественных чисел с плавающей точкой
- Представление нуля. Отрицательный ноль
- Арифметика нуля со знаком
- Представление бесконечности
- Неопределённость NaN
- Денормализованные числа
- Машинное эпсилон
- Свойства множества машинных чисел с плавающей точкой
- Представимые и непредставимые числа

Тема 1. Архитектура ЭВМ с точки зрения программиста

Центральный процессор и оперативной память

Основной узел ЭВМ, выполняющий вычисления — центральный процессор (CPU).

Данные, которые использует центральны процессор в своей работе хранятся в оперативной памяти (RAM, ОЗУ).

• **Оперативная память** — это линейная последовательность ячеек памяти.

Каждая ячейка памяти имеет фиксированный размер, обычно это 1 байт.

Ячейка памяти в 1 байт хранит 8 бит информации. Это значит, что в ней может хранится целое число в диапазоне от 0 до 255.

Каждая ячейка памяти имеет порядковый номер, который называется адресом. По адресу можно обратиться к каждой ячейке — считать или записать данные.

• **Центральный процессор** автоматически под управлением программ считывает данные из ячеек оперативной памяти, некоторым образом их модифицирует и снова записывает их в память.

Кроме ячеек оперативной памяти процессор также работает собственные (встроенные) ячейки памяти, которые называются регистрами процессора.

• **Программа** — последовательность инструкций для центрального процессора (машинные инструкции), закодированных в цифровом виде.

Программа также как данные хранится в оперативной памяти.

Процессор имеет специализированный регистр IP (Instruction Pointer, указатель инструкции), в котором хранится адрес ячейки оперативной памяти с очередной инструкций программы.

• **Блок управления (БУ)** — отвечает за выполнение последовательности микрокоманд при обработке машинной инструкции.

• **Арифметико-логическое устройство (АЛУ)** выполняет арифметические операции, такие как сложение, вычитание, а также логические операции.

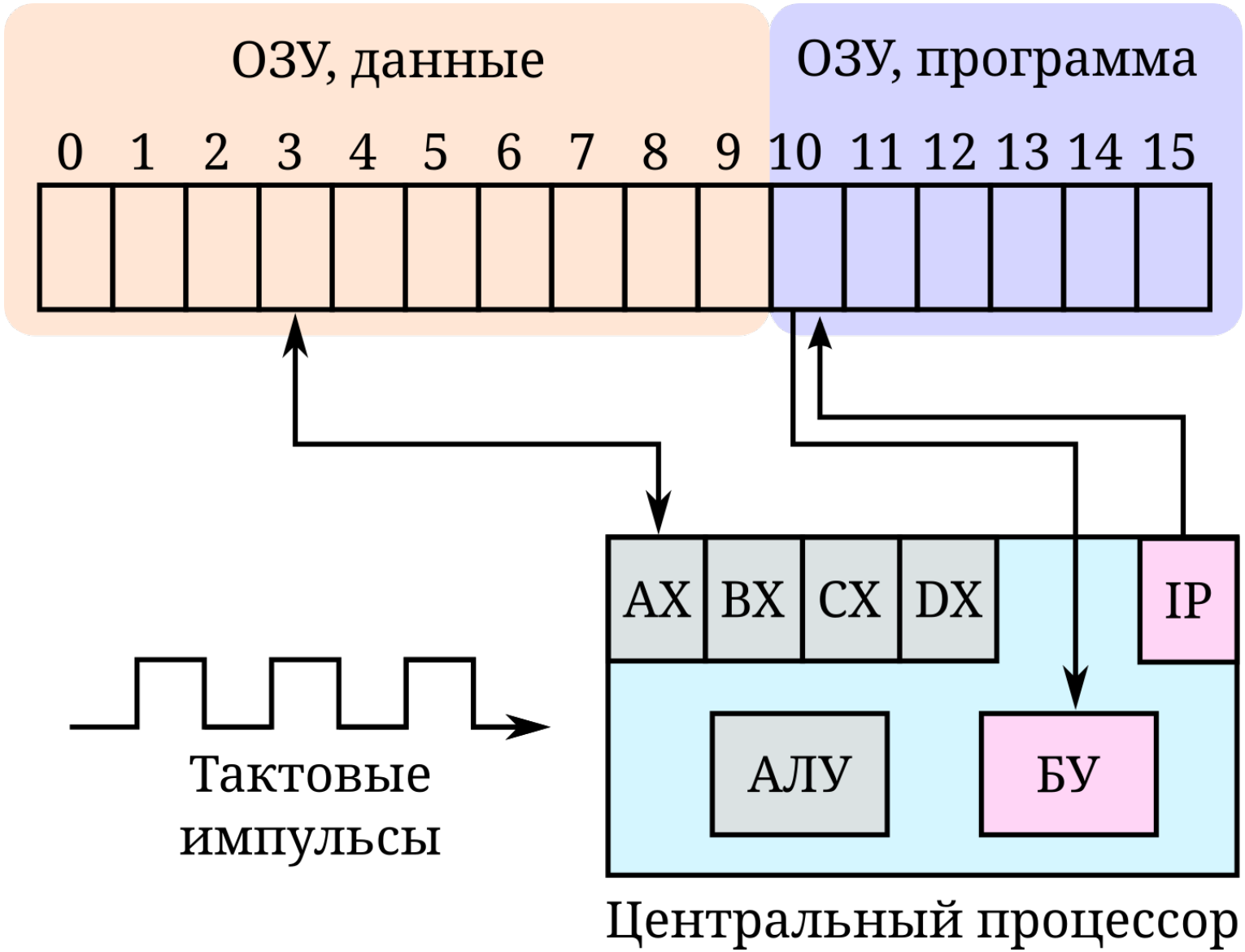
• **Тактовый генератор** — один из узлов ЭВМ который посылает импульсы на все другие узлы с заданной периодичностью.

• **Машинный такт** соответствует одному периоду импульсов тактового генератора и является основной единицей измерения времени выполнения команд процессором.

• **Машинный цикл** состоит из нескольких машинных тактов. Машинный цикл — это время, необходимое для выполнения одной команды.

При получении очередного тактового импульса процессор выполняет машинный цикл:

- Считывает очередную инструкцию из оперативной памяти; адрес ячейки памяти с инструкцией берётся из регистра IP и автоматически значение адреса в этом регистре увеличивается.
- БУ декодирует инструкцию и начинает выполнение действий:
  - чтение/запись содержимого ячейки памяти в/из регистра процессора;
  - выполнение АЛУ арифметических действий над данными в ячейках памяти и/или регистрах;
  - модификация регистра IP (условный или безусловный переход).


ОЗУ, данные
ОЗУ, программа
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
AX BX CX DX
IP
АЛУ
БУ
Тактовые импульсы
Центральный процессор

Порты ввода-вывода

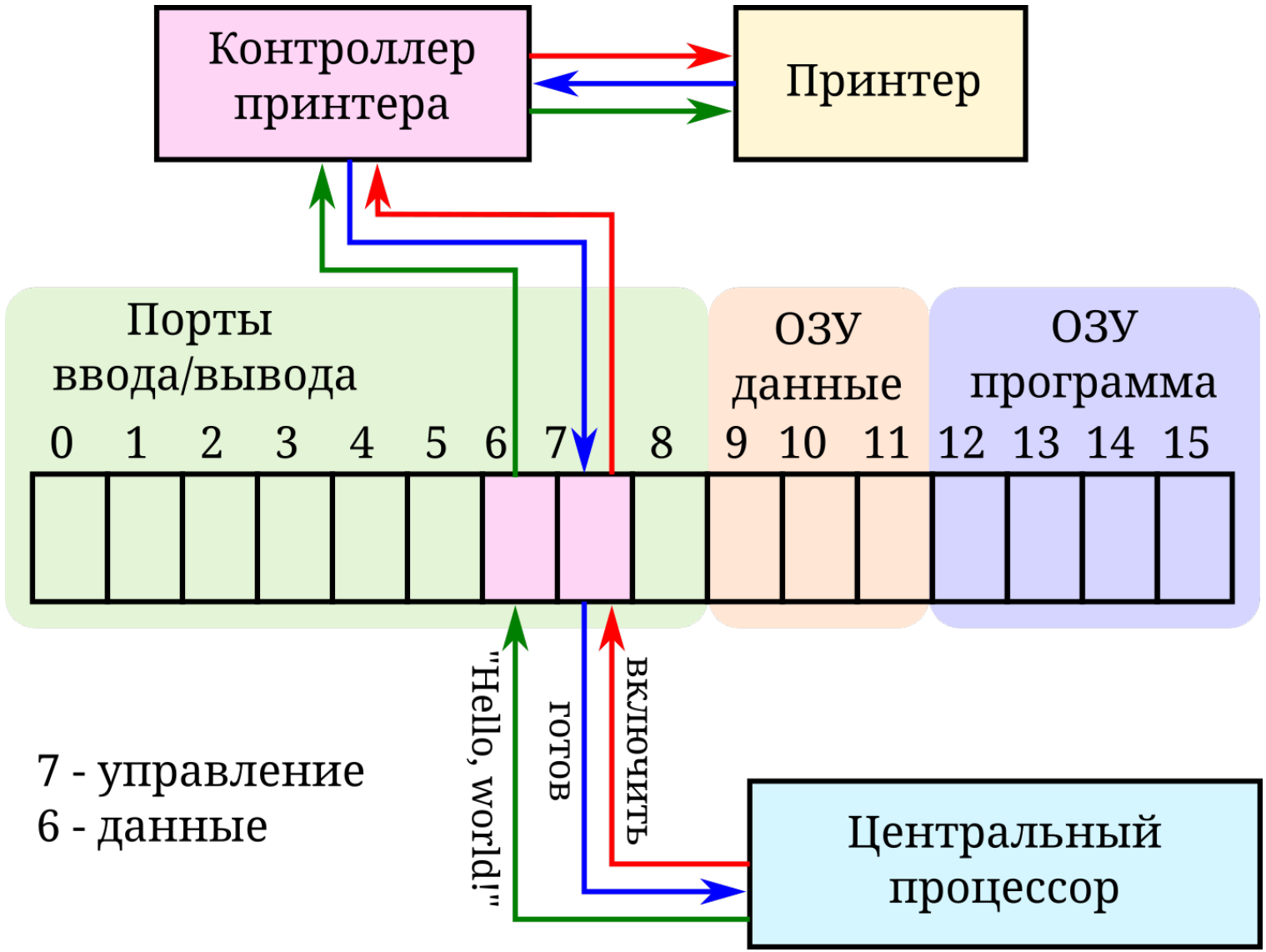
Взаимодействие с внешним миром центральный процессор осуществляет посредством обращений к ОЗУ.

• **Порты ввода вывода (I/O port)** — ячейки ОЗУ специального вида, который посредством контроллера ввода-вывода связаны с внешним устройством и предназначены для обеспечения взаимодействия процессора с этим устройством.

• **Контроллер ввода-вывода** — элемент ЭВМ, которое преобразует обращения процессора к портам ввода вывода в сигналы взаимодействия с внешним устройством.

Когда процессор пишет данные ячейку порта контроллер преобразует их сигналы, воспринимаемые устройством.

Если внешнему устройству необходимо передать данные или сообщение процессору, контроллер записывает их в ячейку порта.


Контроллер принтера
Принтер
Порты ввода/вывода
ОЗУ данные
ОЗУ программа
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
"Hello, world!"
готов
включить
7 - управление
6 - данные
Центральный процессор

Тема 2. Машинное представление данных

Стандартные числовые данные

• **Байт или полуслово** — последовательность из восьми бит. Байт является минимально адресуемым элементом памяти.

• **Слов** — последовательность из двух байтов. Адресом слова является адрес младшего байта и всегда является положительным (ноль рассматриваем как положительное число). Длина слова равна 16 битам.

• **Двойное слово** — последовательность из четырёх байтов. Адресом двойного слова является адрес младшего байта и всегда является положительным (ноль рассматриваем как положительное число). Длина двойного слова равна 32 битам.

К основным (стандартным) числовым данным, используемым в программировании, относятся:

- целые числа без знака
- целые числа со знаком
- числа с фиксированной точкой
- числа с плавающей точкой

Беззнаковые целые числа и числа со знаком

Беззнаковые целые — это числа, среди которых нет отрицательных.

Диапазон значений для беззнакового числа, занимающего n, бит равен 2^n

- Байт: $2^8 = 256$
- Слово: $2^{16} = 65536$
- Двойное слово: $2^{32} = 4294967296$

Целые числа со знаком включают как положительные так и отрицательные числа.

Максимальное по модулю значение, которое может принимать беззнаковое число в два раза больше, чем число со знаком.

Кодирование беззнаковых целых чисел

Рассмотрим ячейку памяти в половину байта, т. е. имеющую 4 бита.

В неё может быть записано 16 разных комбинаций нулей и единиц:

0000	0001	0010	0011
0100	0101	0110	0111
1000	1001	1010	1011
1100	1101	1110	1111

Пусть мы решили использовать эту ячейку как целое беззнаковое число. Как сопоставить комбинациям нулей и единиц числа?

Вообще говоря, можно произвольно сопоставить каждой комбинации какое-то число: 0000=13, 0001=7 и т. д.

Но при этом арифметика становится очень громоздкой — придётся хранить таблицы, в которых будут записаны результаты арифметических операций с такими числами.

Лучше, если будет некоторое правило перевода последовательностей нулей и единиц в числа. Тогда арифметические действия тоже можно будет производить по некотором правилам, без использования громоздких таблиц сложения.

Естественное правило — считать последовательности нулей и единиц, записанные в ячейку, двоичными числами.

При такой кодировке сразу понятно, как выполнять арифметические операции.

Таким образом, мы получаем возможность работать с 16 разными числами в диапазоне от 0 до 15.

$0000_2=0$	$0001_2=1$	$0010_2=2$	$0011_2=3$
$0100_2=4$	$0101_2=5$	$0110_2=6$	$0111_2=7$
$1000_2=8$	$1001_2=9$	$1010_2=10$	$1011_2=11$
$1100_2=12$	$1101_2=13$	$1110_2=14$	$1111_2=15$

Кодирование целых со знаком

Теперь предположим, что нам нужны как положительные так и отрицательные числа.

Разделим все имеющиеся комбинации нулей и единиц на две части. Одну из них будем считать содержащей положительные числа, вторую — отрицательные.

Наиболее простой вариант разделения — по символу в самой левой позиции. Комбинации в которых в самой левой позиции стоит 0 будем считать положительными числами, а если там 1, то это отрицательные числа.

Положительные числа

0000	0001	0010	0011
0100	0101	0110	0111

Отрицательные числа

1000	1001	1010	1011
1100	1101	1110	1111

Как теперь закодировать числа? Т. е. по какому правилу сопоставить разным комбинациям нулей и единиц разные числа?

Положительные кодируем также как и для беззнаковых — используя двоичную систему счисления.

Теперь нужно придумать такой способ кодирования, чтобы автоматически выполнялось равенство

$$x + (-x) = 0$$

Мы уже обсуждали, что преимущество кодирования с использованием двоичной системы счисления — простая и удобная арифметика.

Сложим какое-нибудь положительное число, например 0101_2=5, со всеми кандидатами в отрицательные числа чтобы найти такое, которое при сложении с 0101_2 даёт ноль в двоичной системе счисления:

0101	0101	0101	0101	0101	0101	0101	0101
+ 1000	+ 1001	+ 1010	+ 1011	+ 1100	+ 1101	+ 1110	+ 1111
1101	1110	1111	10000	10001	10010	10011	10100

Видим, что среди результатов нет нуля. Однако нужно принять во внимание, что у нас идёт речь о ячейке памяти размером в 4 бита. Следовательно, единица в 5-м разряде в числах 10000, 10001, 10010 будет отброшена — она просто не уместиться в разрядную сетку.

Примечание

Если в результате арифметической операции с машинными числами появляется единица в разряде, которого уже нет в ячейке (9-ый для байта, 17- для слова) это называется переполнение (overflow). Эта единица запоминается в специальной однобитовой ячейке процессора, которая называется флагом переполнения. Программист может проверить этот флаг после выполнения операции и как-то обработать эту ситуацию.

Значит нам нужно положить $1011_2{=}{-}5$, так как с учётом потери единицы в 5-м разряде

$$0101_2 + 1011_2 = 0000_2$$

Теперь нужно найти правило инвертирования чисел без механического перебора всех вариантов, т. е. для числа $0101_2{=}5$ нужно научиться находить двоичное представление числа -5.

Обратим внимание на столбец левее выбранного:

$$0101_2 + 1010_2 = 1111_2$$

Второе слагаемое получается из 0101 поразрядной инверсией битов (заменой на противоположные).

Видно, что нужная нам комбинация 1011 (нужная, потому что мы решили, что именно она будет принята за -5) получается из 1010 прибавлением единицы.

$$1010_2 + 0001_2 = 1011_2$$

Теперь мы готовы сформулировать правила.

• **Прямым кодом** называется двоичная запись неотрицательного целого числа.

• **Обратным кодом** называется запись, полученная поразрядной инверсией прямого кода.

• **Дополнительным кодом** называется запись, полученная прибавлением 1 (единицы) к обратному коду.

• **Целые отрицательные числа** в современной вычислительной технике кодируются с использованием дополнительного кода.

Таким образом, отрицательные машинные числа в двоичном представлении содержат единицу в старшем разряде и выражаются в дополнительном коде.

Примеры работы с отрицательными числами

Для нахождения отрицательного числа нужно инвертировать все биты, равного по модулю положительного числа, а затем прибавить 1.

Операция получения дополнительного кода — это инволюция, т. е. для получения положительного числа её нужно применить повторно.

Пример: $65 \to -65$, размер чисел — 1 байт.

Исходное двоичное число:	$01000001_2{=}65$
Инвертируем биты:	10111110
Прибавляем 1:	$10111111_2{=}{-}65$

В обратную сторону: $-65 \to 65$

Исходное двоичное число:	$10111111_2{=}{-}65$
Инвертируем биты:	01000000
Прибавляем 1:	$01000001_2{=}65$

Сумма 65 и -65 должна быть равна нулю:

$$\begin{array}{r} 01000001 \\ +\,10111111 \\ \hline 00000000 \end{array}$$

Единицу в 9-м, несуществующем разряде отбрасываем.

Принципиальное отличие машинных вещественных чисел от «математических»

Выше уже обсуждалось, что в отличие от «математических» чисел, машинные всегда имеют ограниченный диапазон значений.

Кроме этого, машинные числа всегда меняются с некоторым шагом.

У целых чисел шаг равен 1 и это соответствует свойствам «математических» целых чисел.

У «математических» вещественных чисел шага нет!

Примечание

Аксиома непрерывности: какими бы не были две вещественных числа a и b, $a < b$, всегда имеется бесконечно много чисел x, которые лежат между ними: $a < x < b$.

Машинные вещественные числа всегда представлены конечным числом бит. Следовательно кроме ограниченного диапазона они характеризуются машинным шагом.

Пример ниже показывает, что если прибавить к вещественному числу слишком маленькое число, то оно не изменится:

```
x1 = 1.0
dx = 1e-16
x2 = x1 + dx
print(f"x1={x1}, x2={x2}")
if x1 == x2:
    print("x1 равно x2")
```

```
Будет напечатано:
x1=1.0, x2=1.0
x1 равно x2
```

С точки зрения математики это ошибка — числа x_1 и x_2 должны быть разными!

• **Машинный шаг вещественного числа (дельта)** — минимальное число, которое нужно прибавить (или отнять) к машинному вещественному числу, чтобы получить отличающееся от него машинное число.

Вещественные числа с фиксированной точкой

• **Вещественные числа с фиксированной точкой** — представление вещественных чисел, при котором машинный шаг остаётся постоянным.

На рисунке машинный шаг равен $\Delta = 0.1$.

Пример — представление денежных сумм с точностью до копеек

Такие числа похожи на целые числа.

В современных вычислительных системах для них обычно не выделяют специальный тип данных.

Для вычислений используют либо целые числа (в примере на рисунке целые получатся домноженном всех чисел на 10) либо вещественные числа с плавающей точкой с округлением.

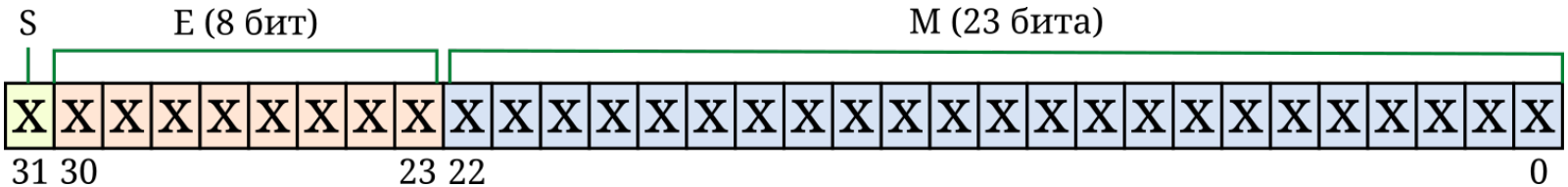
Кодирование вещественных чисел с плавающей точкой

Машинное вещественной число с плавающей точкой представляется в экспоненциальной форме по основанию 2 (вспомним, что выше мы обсуждали экспоненциальный формат по основанию 10):

$$x = (-1)^S \cdot M \cdot 2^E$$

Здесь S — бит кодирующий знак числа (sign), может быть либо 0 либо 1; M — мантисса (mantissa), E — порядок (exponent).

У чисел одинарной точности (float) под мантиссу M отводится 23 бита, под порядок E 8 бит, а знак всегда кодируется 1 битом.



Вспомним, что число в экспоненциальном формате можно записывать по-разному:

$$1.234 \cdot 10^{-3} = 12.34 \cdot 10^{-4} = 0.1234 \cdot 10^{-2}$$

Для чисел записанных по основанию 2 это выглядит так:

$$1.011_2 \cdot 2^{-3} = 10.11_2 \cdot 2^{-4} = 0.1011_2 \cdot 2^{-2}$$

Переведём это число в десятичную форму, просто чтобы лучше понять как оно устроено:

$$\begin{aligned} 1.011_2 \cdot 2^{-3} &= (1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 1 \cdot 2^{-3}) \cdot 2^{-3} = \\ &= (1 + 1/4 + 1/8)/8 = 0.171875 \end{aligned}$$

Так как есть несколько вариантов экспоненциальной записи, нужно соглашение о том, какой вариант выбирать.

Будем записывать числа таким образом, чтобы значение мантиссы было всегда больше или равно 1

$$1.011_2 \cdot 2^{-3} = \cancel{10.11_2 \cdot 2^{-4}} = \cancel{0.1011_2 \cdot 2^{-2}}$$

Так как двоичных символов всего два, 0 и 1, при такой записи в первом разряде мантиссы всегда будет стоять 1. И её можно не хранить, сэкономив один бит.

Таким образом для числа $1.011_2 \cdot 2^{-3}$ в поле мантиссы будут хранится биты $01100\ldots0$ (всего 23 бита) что соответствует мантиссе

$$M = 1.011$$

В отличие от целых чисел, для кодирования отрицательных чисел не используется дополнительный код.

Бит S явно указывает на знак числа: $S = 0$ число положительное, $S = 1$ — число отрицательное.

Порядок E может быть положительным и отрицательным. Для записи отрицательных порядков также не используется дополнительный код.

Пусть под порядок отводится 8 бит. Значит можно записать 256 разных значений. Половину из них естественно отвести под положительные числа, а вторую под отрицательные.

Вместо того, чтобы использовать дополнительный код перед записью в ячейку памяти значения порядка просто прибавим к нему 127. А перед использованием вычтем 127.

Минимально допустимое значение порядка будет равно -127. В этом случае биты порядка будут хранить все нули ($-127 + 127 = 0$): 00000000.

Максимально допустимое значение порядка равно 128. Такой порядок будет записан в память как число 255 ($128 + 127 = 255$, т. е. в ячейках единицы: 11111111.

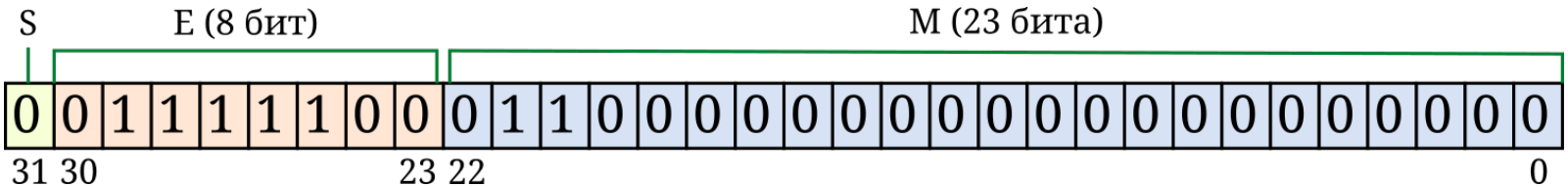
Получается 127 значений отводится отрицательных значений E, 128 значений для положительных E и одно для нулевого.

В общем случае, если для порядка отводится k бит, то прибавляется и вычитается $2^{k-1} - 1$. Проверим: при $k = 8$ $2^7 - 1 = 127$.

Пример: рассмотрим как будет записано число 0.171875 при использовании машинных чисел одинарной точности.

$$0.171875 = 1.011_2 \cdot 2^{-3}$$

$S = 0$ т. к. число положительное; $M = 1.011$, но запишется только 011; $E = -3$, но записано будет $-3 + 127 = 124 = 01111100_2$



Обратим внимание — нули в мантиссе в позициях с 19 по 0 соответствуют незначащему хвосту из нулей после последней цифры после запятой.

Числа у которых мантисса записана с подразумеваемой единицей в старшем разряде называются нормализованными.

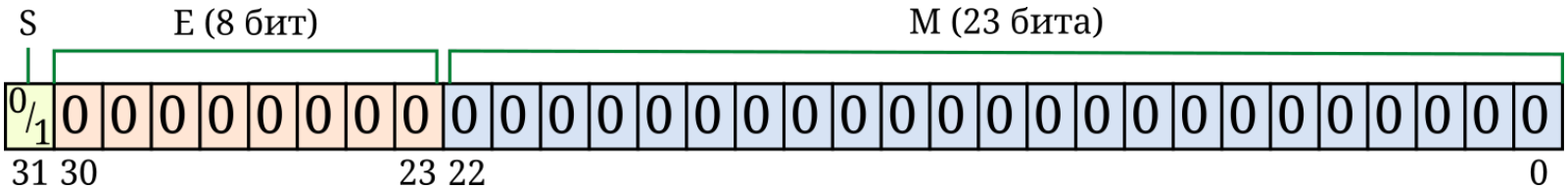
Несмотря на то, что физически под мантиссу отведено 23 байта, за счёт подразумеваемой единицы в старшем бите мы имеем дело с 24-битным кодированием значений мантиссы.

Представление нуля. Отрицательный ноль

Так как всегда предполагается наличие единицы правее старшего разряда мантиссы, описанным способом невозможно записать 0.

Решение проблемы — договорённость о том, что определённые комбинации нулей и единиц нужно интерпретировать специальным образом.

Число считается **нулём** если все его биты, кроме знакового, равны нулю.



В зависимости от значения бита знака ноль может быть как положительным так и отрицательным.

Арифметика нуля со знаком

Арифметика отрицательного нуля понятна интуитивно.

$$\frac{-0}{|x|} = -0 \text{ если } x \neq 0$$

$$(-0) \cdot (-0) = (+0)$$

$$|x| \cdot (-0) = (-0)$$

$$x + (\pm 0) = x$$

$$(-0) + (-0) = -0$$

$$(+0) + (+0) = +0$$

При сравнении отрицательный ноль равен положительному.

В стандарте сохранили знак нуля, чтобы выражения, которые в результате переполнения или потери значимости превращаются в бесконечность или в ноль, при умножении и делении все же могли представить максимально корректный результат.

Например, если бы у нуля не было знака, равенство

$$1/(1/x) = x$$

не выполнялось при $x = -\infty$. Тогда $1/-\infty$ было бы равно 0 а не -0 а выражение $1/(1/x)$ было бы равно ∞, а не $-\infty$ как должно быть.

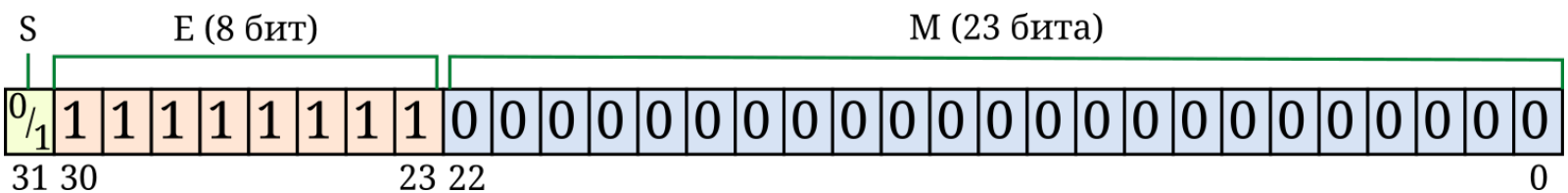
Представление бесконечности

При разработке стандарта для представления вещественных чисел потребовалось каким-то образом обрабатывать ошибочные ситуации вроде деления на ноль или переполнения разрядной сетки.

$$\frac{|x|}{0} = \infty, \; -\frac{|x|}{0} = -\infty$$

Для таких ситуаций были введена возможность представления бесконечностей $+\infty$ и $-\infty$.

Число с плавающей запятой считается равным **бесконечности**, если все двоичные разряды его порядка — единицы, а мантисса равна нулю. Знак бесконечности определяется знаковым битом числа.



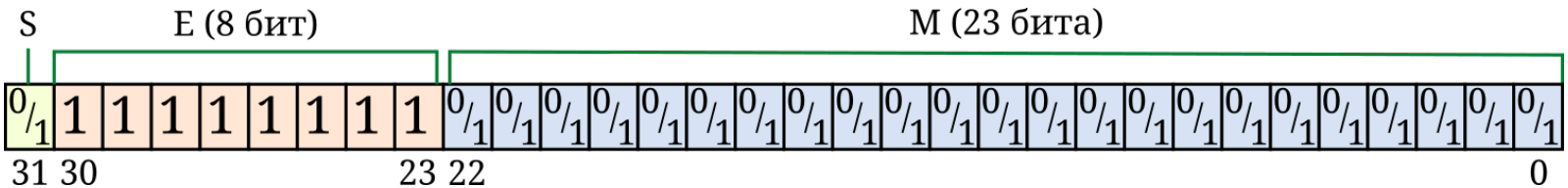
Неопределённость NaN

Существуют операции, результат которых нельзя выразить ни через конечные числа ни с использованием бесконечностей. Например

$$\infty + (-\infty) = ?,\ 0 \cdot \infty = ?,\ 0/0 = ?$$

Чтобы любая арифметическая операция возвращала осмысленный результат, ввели специальное значение NaN (Not a Number).

• **NaN** представляется как число, в котором все двоичные разряды порядка единицы, а мантисса не нулевая.



$$\infty + (-\infty) = \text{NaN},\ 0 \cdot \infty = \text{NaN},\ 0/0 = \text{NaN}$$

Любая операция с NaN возвращает NaN.

По определению NaN $\neq$ NaN, поэтому, для проверки значения переменной нужно просто сравнить ее с собой.

В некоторых языках программирования, например в C++, есть «тихий» (quiet) и «сигнальный» (signaling) NaN: первый, попав в любую операцию, возвращает NaN, второй — вызывает исключительную ситуацию и прерывание нормального выполнения программы.

В Python стандартными средствами (float) чаще всего работают именно с тихими NaN, в то время как сигнальные NaN требуют специфической настройки.

Обычно «тихий» или «сигнальный» определяется старшим битом мантиссы.

Денормализованные числа

Описанная схема кодирования вещественных чисел имеет особенность — между нулём и ближайшим к нему ненулевым числом находится достаточно большая область внутри которой нет чисел.

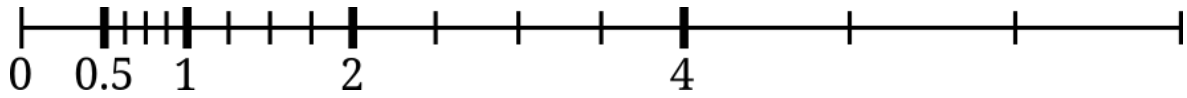
Поясним это на примере. Пусть мантисса и порядок кодируются двумя байтами. Будем рассматривать только положительные числа (для отрицательных будет все тоже самое).

В двух битах можно закодировать 4 значения порядка: -1, 0, 1 и 2.

Также 4 значения может принимать мантисса (не забудем что имеется еще один подразумеваемый единичный бит)

$1.00_2 \cdot 2^{-1} = 0.5$	$1.01_2 \cdot 2^{-1} = 0.625$	$1.10_2 \cdot 2^{-1} = 0.75$	$1.11_2 \cdot 2^{-1} = 0.875$
$1.00_2 \cdot 2^{0} = 1$	$1.01_2 \cdot 2^{0} = 1.25$	$1.10_2 \cdot 2^{0} = 1.5$	$1.11_2 \cdot 2^{0} = 1.75$
$1.00_2 \cdot 2^{1} = 2$	$1.01_2 \cdot 2^{1} = 2.5$	$1.10_2 \cdot 2^{1} = 3$	$1.11_2 \cdot 2^{1} = 3.5$
$1.00_2 \cdot 2^{2} = 4$	$1.01_2 \cdot 2^{2} = 5$	$1.10_2 \cdot 2^{2} = 6$	$1.11_2 \cdot 2^{2} = 7$

Вот так расположены эти числа на числовой оси



Расстояние между штрихами — машинный шаг (дельта). Видно что чем больше число по модулю, тем больше дельта, т. е. сетка становится все более и более разрежённой.

Но относительное значение шага остаётся постоянным (вычисляем это используя обычные, «математические» числа:

$$\frac{0.625 - 0.5}{0.5} = \frac{1.25 - 1}{1} = \frac{2.5 - 2}{2} = \frac{5 - 4}{4} = 0.25$$

Но шаг между нулём и первым ненулевым числом равен 0.5 — это в два раза больше.

Теперь используя машинную арифметику найдём разность $0.625 - 0.5$. По правилам математики должно быть 0.25, но у нас нет такого числа (см. таблицу имеющихся машинных чисел). Поэтому результат будет округлён до нуля.

Тоже самое получится если вычесть $1.5 - 1.25$: ожидаем 0.25, а получим 0.

Подытоживаем: вблизи нуля имеется «околонулевая яма» попадание в которую даёт значительно увеличение погрешности.

Очевидно, что когда мы используем не 2-битную кодировку вещественных чисел, а обычные числа одинарной или двойной точности, околонулевая яма значительно меньше.

Найдём минимальное ненулевое число одинарной точности.

Минимальное значение мантиссы — все нули. Тогда с учётом подразумеваемого бита это будет число 1. Порядок не может быть нулевым — число с нулевым порядком и нулями в мантиссе — это представление нуля. Поэтому минимальное значение порядка 1. С учётом сдвига на -127 получим порядок -126.

Поэтому для чисел одинарной точности минимальное ненулевое число равно 2^{-126}. А следующее число равно $(1 + 2^{-23}) \cdot 2^{-126}$ (вспомним что в мантиссе 23 бита).

Тогда шаг равен

$$\Delta = (1 + 2^{-23}) \cdot 2^{-126} - 2^{-126} = 2^{-149}$$

Получается, что шаг от 0 до первого ненулевого числа равен 2^{-126}, а шаг до следующего числа на 23 порядка меньше, 2^{-149}. Дельты между последующим числами будут также малы.

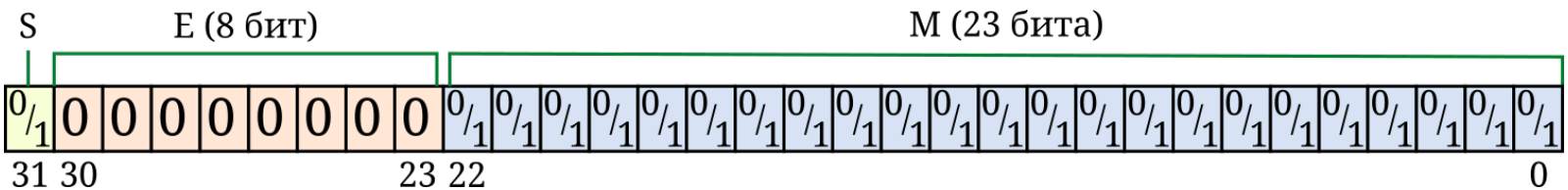
Точность представления вещественных числе вблизи нуля падает.

Следовательно, если в ходе вычислений значения попадают в околонулевую яму, точность тоже падает.

Эту проблему тщательно анализировали. Согласно статистике 70-х годов в среднем каждый компьютер в ходе вычислений попадает в околонулевую яму примерно раз в месяц.

Для борьбы с этим ввели денормализованные (subnormal) числа.

• **Денормализванное число** — вещественное число у которого все биты порядка равны нулю, а мантисса не равна нулю. Для таких чисел порядок принимается равным минимальному значению (2^{-127} для чисел одинарной точности), а подразумеваемый бит считается равным нулю.



Денормализованные числа заполняют околонулевую яму и поэтому проблема резкой потери точности решается.

Однако, из-за того, что такие числа нужно обрабатывать по-другому во всех арифметических операциях, трудно сделать работу в такой арифметике эффективной.

Работа с нормализованными вещественными числами реализуется на аппаратном уровне, а для денормализованных чисел это делается крайне редко (из-за сложности и, следовательно, дороговизны). Вместо этого используются программные реализации, работающие значительно медленнее.

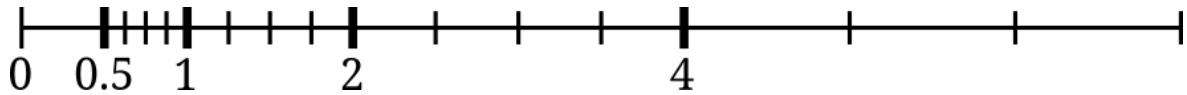
В современных процессорах обработка денормализованных чисел происходит в сотни раз медленнее, чем обработка нормализованных чисел.

Поскольку в стандартных форматах (одинарной и двойной точности) денормализованные числа получаются действительно очень маленькими и часто практически никак не влияют на результат некоторых вычислений (при этом заметно замедляя их скорость), то иногда они просто игнорируются.

Если результат операции — денормализованное число, то оно сразу обнуляется. Если на вход операции поступает денормализованное число оно принимается за ноль.

Машинное эпсилон

При обсуждении денормализованных чисел мы видели, что абсолютное значение шага между соседними числами растёт с ростом числа.



Относительное значение тоже меняется, но в целом остаётся вблизи некоторого значения.

$$\frac{1.25-1}{1}=0.25,\ \frac{1.5-1.25}{1.25}=0.2,\ \frac{1.75-1.5}{1.5}=0.1667,\ \frac{2-1.75}{1.75}=0.1429$$

Для оценки относительного шага между двумя соседними числами используется машинное эпсилон.

• **Машинное эпсилон** — наименьшее положительное число, такое что при машинном сложении

$$1 + \varepsilon \neq 1$$

Это относительная погрешность представления вещественных чисел.

Для вещественных чисел одинарной точности $\varepsilon = 2^{-23} \approx 1.19 \cdot 10^{-7}$.

Для вещественных чисел двойной точности $\varepsilon = 2^{-52} \approx 2.22 \cdot 10^{-16}$.

Свойства множества машинных чисел с плавающей точкой

- **Дискретность:** любые числа с плавающей точкой разделены на числовой оси конечным промежутком
- **Конечность:** имеются наименьшее и наибольшее числа с плавающей точкой
- **Неравномерность:** неравномерное распределение на числовой оси, расстояние между двумя соседними числами с плавающей точкой вблизи нуля меньше, чем вблизи наибольшего или наименьшего значений

Представимые и непредставимые числа

Вещественные числа, точно представимые в виде машинных чисел с плавающей точкой, называются представимыми.

Как уже говорилось выше машинные вещественные числа имеют вид

$$x = (-1)^S \cdot M \cdot 2^E$$

Мантисса M — это число в двоичной системе счисления. Поэтому:

$$x = (-1)^S \cdot (1 + b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \ldots + b_n 2^{-n}) \cdot 2^E$$

где n — число бит, выделенных под хранение мантиссы (для одинарной точности $n = 23$), а b_1, b_2 и т.д. — это двоичные цифры, т. е. либо 0, либо 1.

Вещественное число представимо, если его можно записать в указанной форме и при этом максимальны порядок i у слагаемого $b_i 2^{-i}$ не больше чем n.

Не каждое вещественное число представимо.

Чтобы проверить, представимо ли число нужно вспомнить, что все целые числа представимы (при условии, что используется достаточное количество байт).

Пусть X — вещественное (нецелое) числ, а x — его дробная часть. Например если $X = 13.231$ то $x = 0.231$.

Необходимое условие представимости (но не достаточное!): X может быть представимым, если для его дробной части x можно указать такое целое число N, что $x \cdot 2^N$ становится целым.

Например любое число с дробной частью $x = 0.65625$ может быть представимо, потому что $x \cdot 2^5 = 21$.

Представимость x означает, что его можно точно представить в двоичном виде. Покажем это:

$$0.65625 = (1 + 2^{-2} + 2^{-4}) \cdot 2^{-1} = 1.0101_2 \cdot 2^{-1}$$

Пусть необходимое условие представимости выполнено, т. е. для данного X нашлось N, что $x \cdot 2^N$ целое (вспомним, что x — это дробная часть X).

Это число всё ещё может оказаться непредставимым, так как оно может оказаться настолько большим, что количества бит выделенных для мантиссы окажется недостаточным, чтобы записать это число.

Поэтому дополнительно нужно проверить что число X по модулю меньше максимально допустимого для выбранного типа машинных вещественных чисел (одинарной точности, двойной и т. д.).

Правила машинной арифметики очень хорошо приспособлены для того, чтобы минимизировать погрешности, связанные с непредставимостью. Однако погрешности всё равно возникают и про это нужно всегда помнить.

Ниже приведён пример программы которая сравнивает умножение числа на 10 с десятикратным сложением числа самого с собой:

$$10x = x + x + x + x + x + x + x + x + x + x$$

Число 0.5 представимо. Результат точный:

```
x = 0.5  # Представимо
N = 10
s1 = x * N
s2 = 0.0
for _ in range(N):
    s2 += x
err = s2 - s1
print(f"s1={s1}\ns2={s2}\nerr={err}")
```

```
Будет напечатано:
s1=5.0
s2=5.0
err=0.0
```

Число 0.2 непредставимо. Обратите внимание на погрешность:

```
x = 0.2  # Непредставимо
N = 10
s1 = x * N
s2 = 0.0
for _ in range(N):
    s2 += x
err = s2 - s1
print(f"s1={s1}\ns2={s2}\nerr={err}")
```

```
Будет напечатано:
s1=2.0
s2=1.9999999999999998
err=-2.220446049250313e-16
```