Конспект 01 — Представление чисел в памяти

Содержание

”…конечно же, имеется в виду энтерпрайз $NVIDIA$’вское железо, а не то, что продаётся для нищих геймеров — ибо зачем продавать за два килобакса, когда можно продать за двадцать?”
— П. С. Скаков

Первые представления о числе

Адресация в памяти

Особо пытливому читателю известно, что современные электронно-вычислительные устройства хранят любую информацию в двоичной системе счисления, то есть в виде набора последовательно идущих *бит* (от англ. *«binary digit»* — двоичная цифра). Процессор, впрочем, работает с данными не целиком и не побитово — минимальный объем информации, который он может считать, равняется одному *байту*.

> ⓘ **Определение**
>
> **Байт** — наименьшая адресуемая ячейка памяти.

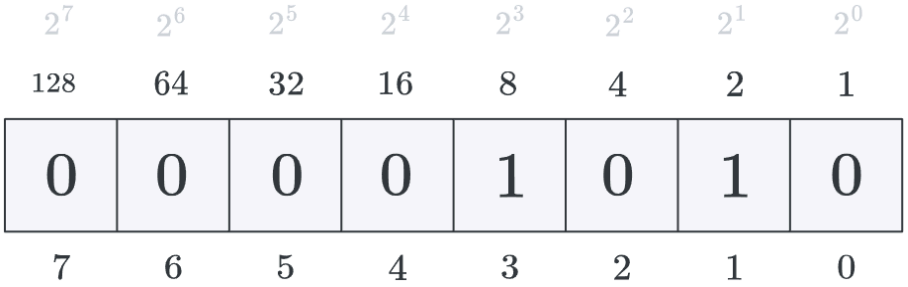
Размер байта может быть произвольным. Хотя практически вся современная вычислительная техника сошлась на значении в 8 бит, в прошлом мы могли иметь дело, например, с шестибитными или четырехбитными байтами — это, помимо прочего, можно считать одной из причин ряда странностей при работе с арифметическими вычислениями в языках C/C++.

В языке Си, кстати, понятия «байт» нет — вместо него используется тип `char`, который Стандарт определяет как, по сути, синоним байта. В частности, оператор `sizeof` возвращает размер именно в `char`'ах, не в байтах.
Для определения размера байта отведен специальный макрос `CHAR_BIT`, по умолчанию равный `8`, то есть формально C/C++ поддерживают не только восьмибитные системы, хотя на практике это, конечно, сомнительная авантюра.

Беззнаковые целые числа

Здесь и далее, когда будет идти речь о хранении чисел, будет подразумеваться ячейка памяти размером в 1 байт, если явно не обозначено иное.

Самое простое число, которое можно представить в памяти — беззнаковое целое. Если читателю достаточно повезло и он учился в школе, то он знает, что для этого достаточно просто перевести десятичное число в, собственно, двоичную систему:

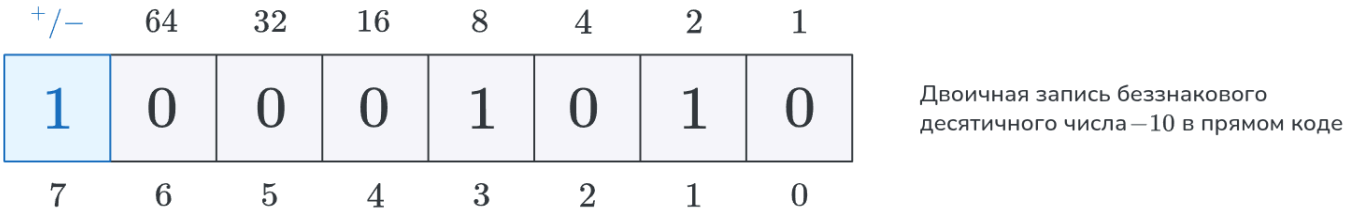


Биты в двоичной записи традиционно нумеруются с нуля, поскольку это позволяет удобно задавать веса разрядов. При этом однозначного порядка бит внутри одного байта не существует, поэтому вместо кажущейся порой естественной терминологии «первый» и «последний» бит следует оперировать понятиями «младший» и «старший» бит соответственно.

Диапазон беззнаковых чисел, которые может сохранить один байт, составляет `0..255`. Легко видеть, что размер этого диапазона — 256 уникальных значений — равен максимальному объёму информации, который вообще можно записать в 8 бит.

Знаковые числа

Ситуация становится интереснее, когда возникает желание обрабатывать отрицательные числа. Первое, что приходит на ум — использовать известную нам алгебраическую модель, в которой под знак числа отводится отдельный символ, после которого записывается значения модуля:

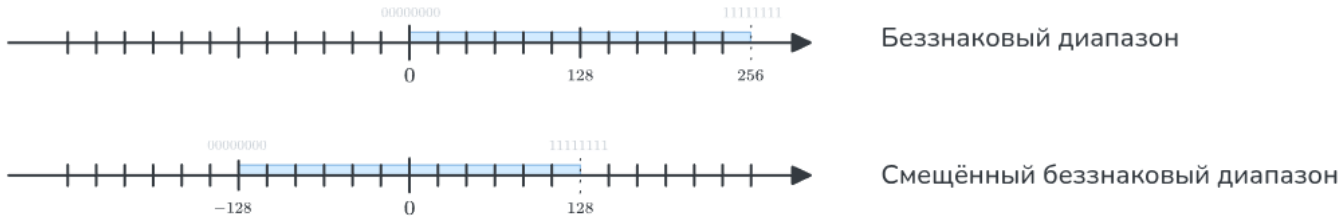


Под хранение знака чаще всего выносится старший бит, причём `-` обозначается единицей, а `+` — нулём, поскольку так запись соответствующих положительных чисел будет одинаковой и в знаковой, и в беззнаковой записи. Приведённый формат хранения знаковых чисел называется *прямым кодом*.

У него есть одна большая проблема. Диапазон его значений составляет `-127..127`, и размер этого диапазона внезапно равен 255, а не 256. Причина в том, что число 0 имеет два различных кода: `10000000` и `00000000`, которые между собой абсолютно равноправны. Даже если закрыть глаза на незначительную неэффективность утилизации памяти в такой модели, возникает множество вопросов, связанных с тем, как вообще

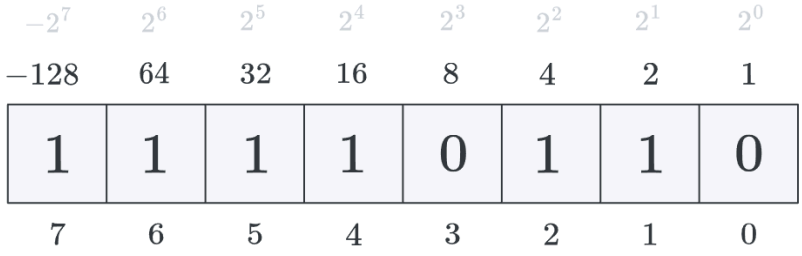
работать с этими двумя нулями. Короче говоря, приятным это не сделать, как ни старайся — читатель должен прекрасно понимать, почему.

Одним из решений проблемы двух нулей может стать использование *кода со сдвигом*. Мы просто «перенумеруем» все уникальные значения из беззнаковой системы так, чтобы они соответствовали нужному нам диапазону:



Можно использовать смещение как на 127, так и на 128 влево. Чтобы получить реальное значение числа в такой модели, необходимо каждый раз вычитать определенную нами величину сдвига, а чтобы закодировать число — прибавлять её. Так мы практически остаёмся в тривиальных рамках беззнаковых чисел, при этом ноль у нас всё ещё единственный — кажется, что схема рабочая, однако она печальным образом усложняет арифметику: обычные двоичные операции в ней не работают.

По этой причине на практике наиболее часто прибегают к концепции *дополнительного кода*. Практически вся идея состоит в том, чтобы установить вес старшего бита равным -128 вместо 128:



Двоичная запись знакового десятичного числа -10 в дополнительном коде

У этого подхода множество преимуществ.
Во-первых, представление положительных чисел в дополнительном коде совпадает с беззнаковым.
Во-вторых, мы лишены проблемы двух нулей.
В-третьих, по старшему биту числа легко определить, отрицательное ли оно — в этом смысле дополнительный код похож на прямой, хотя на этом все сходства заканчиваются.
В-четвёртых, простая арифметика для чисел в прямом коде практически не требует усложнения логики.

Кажется, что теперь-то жизнь точно прекрасна, однако с одной проблемой нам всё же приходится жить.
8 бит суммарно могут сохранить до 256 уникальных значений — то есть, в нашем случае, чисел. Исключим отсюда число 0, и получим суммарно 255 отрицательных и

положительных чисел, то есть 127.5 чисел по каждую сторону от нуля. Естественно, по половине числа нормальные люди не хранят. В этом, собственно, и проблема — как бы мы ни старались, диапазон дополнительного кода останется *несимметричным* — легко заметить, что он равен `-128..127`, то есть мы способны сохранить на одно отрицательное число больше.

Такое обстоятельство вызывает определенные трудности при, например, вычислении модуля, поскольку у числа -128 положительной пары просто нет. Хуже того, это является проблемой даже при работе со встроенной функцией `abs()`, которая зачем-то возвращает `int` вместо `unsigned int`, поэтому следующий код:

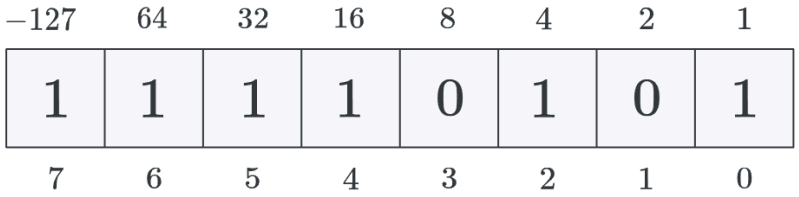
```
int x = std::numeric_limits<int>::min();
std::cout << abs(x);
```

выведет в консоль отрицательное число.

Вариации на тему

Дополнение до одного

Дополнение до одного использует ровно ту же идею, что и дополнение до двух, однако устанавливает значение старшего разряда равное -127 вместо -128:



Двоичная запись знакового десятичного числа −10 в дополнении до одного

Здесь, впрочем, всё ещё существует проблема двух нулей, да и с арифметикой всё достаточно тяжело — словом, никто этим не пользуется.

Форма с чередованием

Периодически возникает желание уметь «расширять» число нулями слева, не изменяя его значения. Это может быть полезно, например, когда мы работаем с числами переменного размера и хотим компактно их кодировать. В дополнительном коде так, конечно, сделать нельзя, поэтому возникает концепция *формы с чередованием*, которая

упорядочивает все числа по модулю и знаку и последовательно назначает им коды:

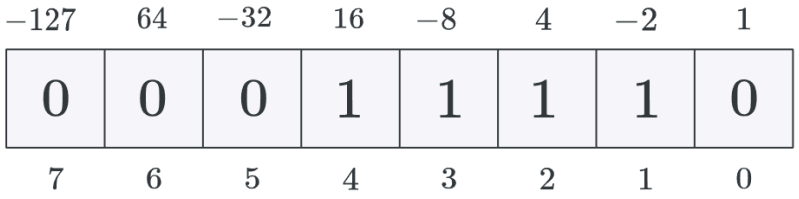
Код	Значение
00000000	0
00000001	−1
00000010	1
00000011	−2
00000100	2
...	...

Легко видеть, что в такой схеме младший бит отвечает за знак, а все оставшиеся — за «почти модуль». Это означает, что для положительных чисел это буквально модуль, а для отрицательных это значение на единицу меньше. Таким образом, можно сказать, что модуль числа равен сумме младшего бита и всех оставшихся.

Эта форма — вариация бита под знак, которая решает проблему двойного нуля и, собственно, расширения слева, однако она всё ещё непрактична для вычислений.

Отрицательное основание системы счисления

В качестве основания системы счисления можно выбрать число -2:



Двоичная запись знакового десятичного числа 10 в системе с основанием −2

Это математически любопытно, но снова не очень практично — во многом из-за сильной неравномерности диапазона, который в данном случае составит `-170..85`.

Троичная система счисления

В качестве фантазии можно попробовать хранить числа в троичной системе счисления, но несколько «сдвинутой»: вместо цифр 0, 1, 2 будем использовать -1, 0, 1, что иногда также записывают в виде z, 0, 1, заменяя отрицательные числа буквами *с конца* латинского алфавита, потому что числа кончаются как бы в другую сторону.

3^7	3^6	3^5	3^4	3^3	3^2	3^1	3^0
2187	729	243	81	27	9	3	1
0	0	0	0	0	1	z	1
7	6	5	4	3	2	1	0

Двоичная запись десятичного числа 7 в троичной системе счисления

Диапазон в такой системе полный и полностью симметричный. Инверсия знака тривиальна. Система масштабируется влево нулями. В целом, всё супер, но — увы — мы всё ещё работаем в двоичной. Терпите.

Целочисленная арифметика

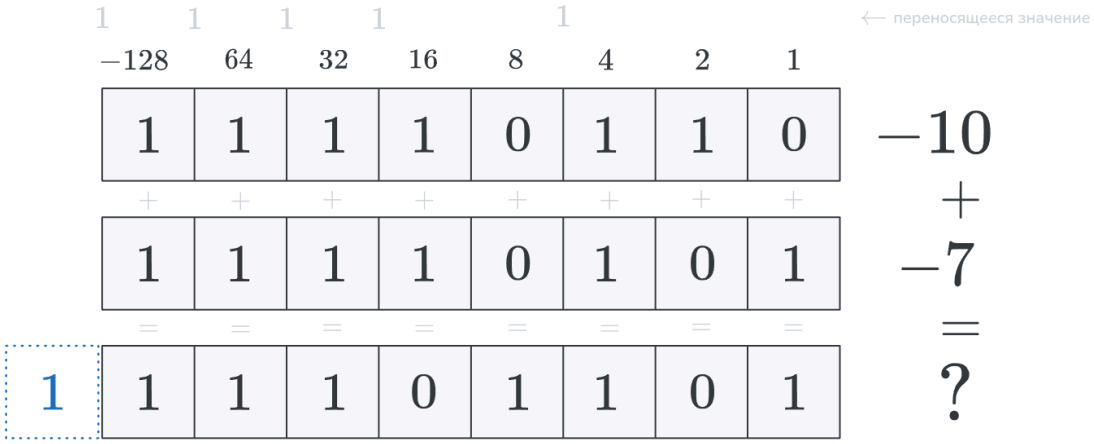
Проблема переполнения

Нам, конечно, хочется не просто хранить числа, но и производить с ними какие-то вычисления.

Самая простая арифметическая операция — это сложение. Прелесть дополнительного кода в том, что в нём оно производятся ровно так же, как и в беззнаковой форме:

			1	1				← переносящееся значение
-128	64	32	16	8	4	2	1	
1	1	1	1	0	1	1	0	-10
+	+	+	+	+	+	+	+	+
0	0	0	0	0	1	1	1	7
=	=	=	=	=	=	=	=	=
1	1	1	1	1	1	0	1	-3

Именно в этот момент мы впервые по-настоящему сталкиваемся с суровой реальностью, поскольку при попытке сложить два отрицательных числа (или, что аналогично, два достаточно больших беззнаковых числа) мы придем к тому, что у нас просто не хватит разрядов для того, чтобы сохранить результат:



Ситуация, в которой результат вычислений выходит за рамки диапазона допустимых значений, называется *переполнением*. Есть несколько подходов к решению этой проблемы:

> I. **Исключение**
>
> Если в результате операции возникло переполнение, будем выбрасывать исключение. Пусть сами разбираются, как с этим быть.

> II. **Арифметика с насыщением**
>
> Если в результате операции возникло переполнение, установим максимальное (минимальное) значение. Мы как бы определяем «потолок», в который упирается значение, если мы пытаемся увеличить (уменьшить) его сверх меры. Это имеет место, например, при обработке изображений — в частности, при изменении общей яркости.

> III. **Модулярная арифметика**
>
> Будем производить все вычисления в кольце вычетов по модулю так, чтобы при переполнении возвращаться в самое начало диапазона. Именно этот подход используется при работе с целочисленными вычислениями.

Теперь, когда мы знаем, как жить с переполнением, сложение угрозы не представляет. Так, в примере выше мы просто «отсечём» бит, выходящий за рамки диапазона, и вполне легитимно получим правильный результат, то есть -17.

Дополнение до двух

Поскольку инженеры, видимо, такие же ленивые, как и программисты, вычитание реализуется как сложение с противоположным числом, что, впрочем, достаточно разумно.

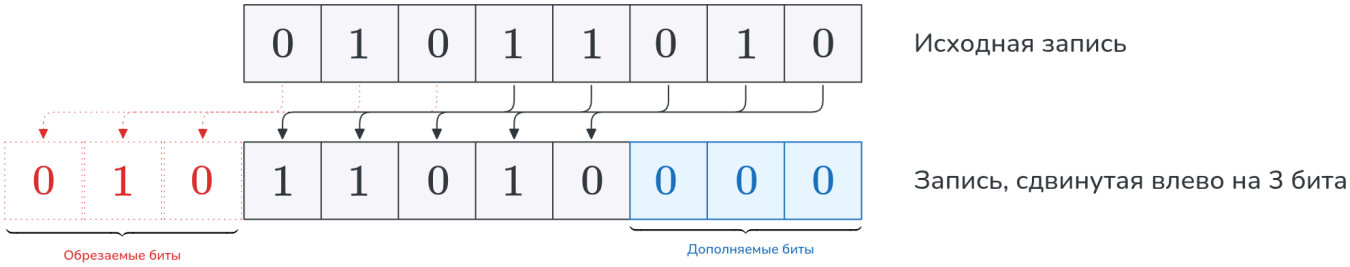
Если для того, чтобы инвертировать число в прямом коде, достаточно инвертировать знаковый бит, то в дополнительном коде всё состоит несколько хитрее, хотя всё ещё

достаточно просто. Алгоритм инверсии заключается в том, чтобы инвертировать все биты числа, после чего прибавить к результату единицу:

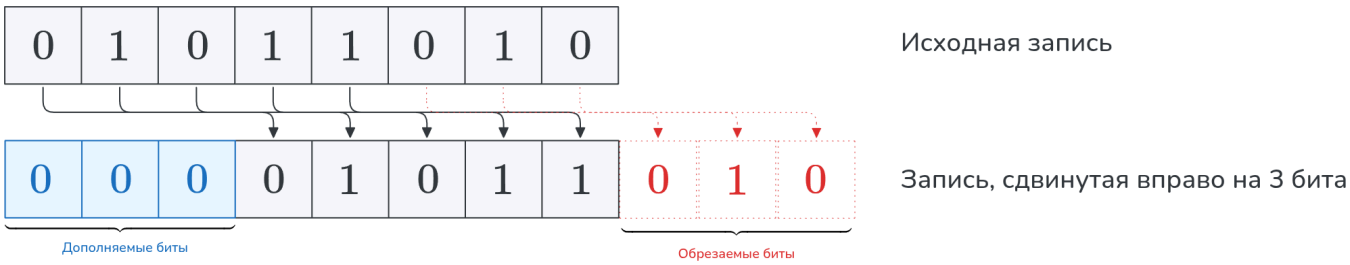
Эта операция называется *дополнением до двух* — поэтому, собственно, *дополнительный* код. Само это название мотивируется математической природой метода: инвертированное число — это в каком-то смысле «дополнение» до ближайшей степени двойки, превосходящей диапазон значений, поскольку сумма обычного и инвертированного чисел даёт именно этот результат, который в результате переполнения становится равен 0, как и ожидается.

Битовые сдвиги

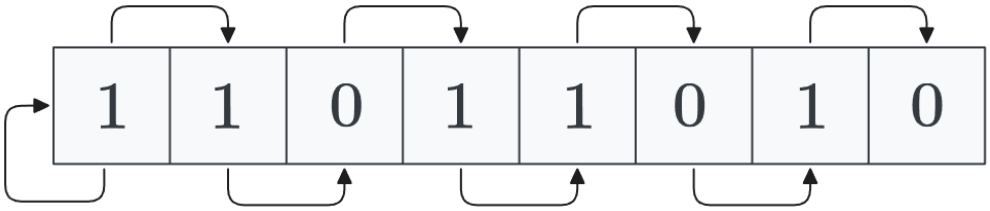
Двоичную запись числа можно сдвигать.
Битовый сдвиг влево практически представляет собой простое модулярное умножение на некоторую степень двойки — иными словами, все биты переходят на n позиций влево, излишки «обрезаются», а недостающие позиции заполняются нулями:



Сдвиг вправо — несколько более комплексная операция.
Для беззнаковых чисел он аналогичен сдвигу влево, но, как можно догадаться, наоборот: теперь это *деление* на степень двойки:



Такой сдвиг называется *логическим*. Для чисел со знаком, в свою очередь, используется *арифметичкский* свдиг вправо, при котором значение старшего разряда остаётся неизменным:



Умножение

Опять проблемы

Умножение — операция коварная.

Корень проблемы заключается в том, что, в общем случае, при умножении n-разрядного числа на m-разрядное мы получаем число разрядности $n + m$. Для нас это означает, что если в случае сложения максимальная величина ошибки составляла 1 бит, то теперь она практически ничем не ограничена.

Если оставлять результат в рамках разрядности операндов, то всё, в целом, ещё ничего. В такой ситуации мы можем позволить себе использовать обычное «школьное» умножение, основанное на на сдвигах, и получать вполне корректный результат, причем — и это очередное замечательное свойство дополнительного кода — вне зависимости от того, имеют ли операнды знак.

Стоит, впрочем, дополнительно пояснить, почему умножение в дополнительном коде действительно пока что не сломалось. Практически, при умножении чисел A и B мы вычисляем сумму

$$R = \sum_{i=0}^{n-1} A \times b_i \times 2^i,$$

где b_i — i-й разряд числа B. Если же мы установим старшему разряду значение -2^{n-1}, то разница между значениями разрядов составит $2^{n-1} - (-2^{n-1}) = 2^n$, и, поскольку мы проводим все вычисления по модулю 2^n, она просто нивелируется.

Такой подход не всегда устраивает нас. При «отсекающем» умножении двух 32-битных чисел мы в худшем случае потеряем ещё 32 бита информации, что бывает крайне нежелательно. По этой причине при умножении имеет место увеличение размерности, если это позволительно.
В случае умножения двух положительных чисел дополнительных затруднений не возникает: мы просто продолжаем сохранять результат в старшие разряды расширенного регистра. Настоящая проблема встаёт тогда, когда хотя бы один из операндов имеет отрицательный знак:

$$\begin{array}{rr} & 1100 \quad -4 \\ \times & 1001 \quad -7 \\ \hline & 1100 \\ + & 0000 \\ + & 0000 \\ + & 1100 \\ \hline & 01101100 \quad 108 \end{array}$$

Схема, осуществляющая обыкновенное беззнаковое умножение, очевидно, не знает о том, что какие-то из операндов могут быть отрицательными, и интерпретирует изображенный пример как 12×9, поэтому для неё получившийся результат абсолютно корректен.

Выходов из этой ситуации несколько.
Первый вариант — явно указывать схеме, что мы складываем *знаковые* числа, т.е преобразовывать промежуточные слагаемые в нужную форму. Так, если *множимое* имеет отрицательный знак, то нам необходимо его расширить, т.е. заполнить слева единицами, а не нулями, как это делается по умолчанию.
Если же *множитель* имеет отрицательный знак, то при умножении на знаковый бит мы должны вычесть сдвинутое множимое вместо того, чтобы сложить его — проще говоря, инвертировать знак промежуточной суммы. Обе эти коррекции в совокупности дают нам правильный результат:

$$\begin{array}{rr} & 1100 \quad -4 \\ \times & 1001 \quad -7 \\ \hline 11111100 & \text{Расширение знака} \\ 0000 & \\ 0000 & \\ 0100 & \text{Инверсия знака} \\ \hline 100011100 & 28 \end{array}$$

Другой, семантически более простой подход — перемножить модули чисел и отдельно, если необходимо, скорректировать знак. Он активно используется при работе с более сложными алгоритмами умножения, в которые слишком тяжело привнести нативную коррекцию знака.

В качестве ещё одной альтернативы можно корректировать результат уже после «неправильного» умножения, поскольку ошибка на самом деле чётко определена благодая математической строгости дополнительного кода.

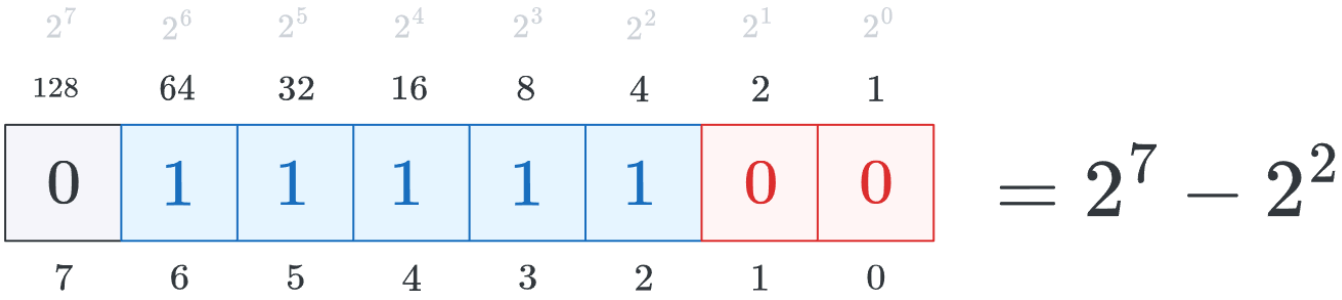
Алгоритм Бута

Примечание. *Не стоит реализовывать этот алгоритм в лабораторных или не дай бог использовать его для ручного умножения чисел. Он отсутствует в программе курса и приведён сугубо из соображений личного интереса автора.*

Все описанные методы неизбежно усложняют логику схем, требуя дополнительных сложных вычислений или даже затрат по памяти. Нам бы хотелось получить простой — как технически, так и математически — и универсальный алгоритм, который позволял бы одинаково эффективно производить умножение как беззнаковых, так и знаковых чисел.

Такой алгоритм был предложен в 1950 году Эндрю Дональдом Бутом. Он основывается на анализе последовательных пар битов множимого и использовании сложения, вычитания и битовых сдвигов для корректировки промежуточных результатов.

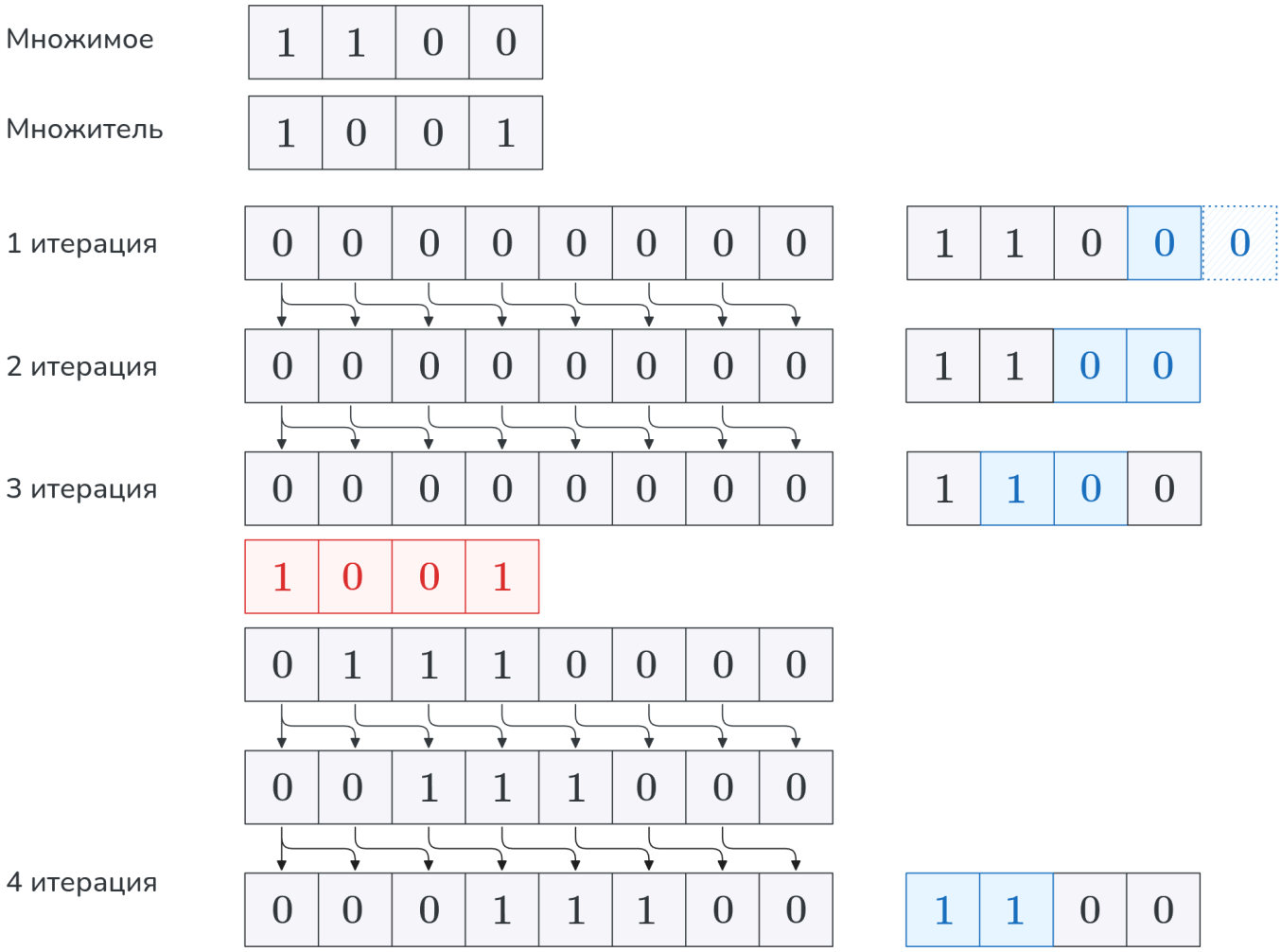
Алгоритм Бута пользуется тем тривиальным фактом, что любое двоичное число, представляющее собой непрерывную последовательность единиц, можно представить в виде единственной разности степеней двойки:



Когда алгоритм Бута встречает во множимом комбинацию битов `10`, он понимает, что это начало такой последовательности, и вычитает из промежуточного результата соответствующую степень. Комбинация `01` последовательность, наоборот, завершает, и в этом случае алгоритм должен выполнить сложение. Остальные комбинации, как легко видеть, никак на ответ не влияют. Важно, что при сложении и вычитании множителя он записывается начиная со старших разрядов.
После каждой итерации алгоритм Бута сдвигает промежуточный результат на один бит *вправо*, причём для этого используется именно арифметический сдвиг.

Суммируя, если мы хотим умножить два числа m и r, то мы должны:

1. последовательно анализировать пары битов m_n и m_{n-1}, начиная с младших, полагая при этом m_{-1} равным нулю:
   - если биты равны `10`, из промежуточного результата вычитается соответствующее смещённое значение множителя;
   - если биты равны `01`, к промежуточному результату прибавляется соответствующее смещённое значение множителя;
   - в противном случае никаких действий не предпринимается;
2. арифметически сдвигать промежуточный результат на один бит вправо.



Чтобы понять, почему этот алгоритм вообще работает, заметим сначала, что в результате всех сдвигов промежуточные суммы займут ровно те позиции, которые они заняли бы в стандартном квадратичном алгоритме, если поменять множимое и множитель местами. Теперь вспомним, что при умножении «в столбик» нам нужны были ровно две коррекции: *расширение* знака и его *инверсия*.

Ключевой момент в том, что обе эти коррекции алгоритм Бута проделывает неявно: расширение — засчёт использования именно *арифметического* сдвига, сохраняющего

знак числа, а инверсию — засчёт того, что в случае отрицательного знака последней операцией будет вычитание, ибо «правая граница» последовательности единиц, оканчивающейся в старшем разряде, не будет обработана.

Алгоритм Бута работает неадекватно, если один из операндов — наименьшее возможное значение из диапазона, потому что, как мы уже выяснили, оно как правило лишено положительной пары, которая понадобится при выполнении операции вычитания. Одним из возможных решений может послужить временное расширение битности.

Числа с фиксированной точкой

Концепция

Жизнь становится ещё менее радужной, когда нам приходится производить вычисления в вещественных числах.
Наиболее простой способ представить вещественное число — отвести фиксированное количество разрядов под целую и под дробную части, продолжив при этом веса разрядов дробной части в отрицательную сторону. Примером такого подхода служит форма 8.8 («восемь-восемь»: по восемь бит на каждую часть):

Это число воспринимается в памяти как единое целое: явно точки не существует нигде, кроме воображения программиста.

Здесь стоит отдельно отметить, как мы переводим дробные числа в двоичную систему счисления. Если в случае с целыми числами мы записываем остатки от деления на 2, то для дробных всё с точностью до наоборот: мы умножаем на 2 и записываем целые части:

Целая часть	Дробная часть
0^*	625
1	250
0	500
1	0

Перевод числа 0.625 в двоичную систему

* не входит в дробную часть — это часть исходного числа!

Особенно внимательный читатель помнит, что остатки при переводе целых чисел находятся в обратном порядке. В случае дробных чисел, как можно догадаться, биты находятся в прямом порядке. Об этом можно думать несколько иначе: биты находятся в порядке *от нуля* — это справедливо одновременно и для дробной, и для целой части.

Округление

Попробуем перевести в двоичную систему число $^1/_{10}$:

0	1
0	2
0	4
0	8
1	6
1	2
0	4
0	8
1	6
1	2
$\dots$	$\dots$

Проиграли! Потому что получили бесконечную периодическую дробь. Как её хранить — непонятно.
Эта проблема, конечно, возникает не только с бесконечными дробями — она встает всегда, когда дробная часть числа не умещается в 8 бит. Это, по сути, переполнение, но теперь нам не хватает младших разрядов, а не старших. Действия, которые мы можем предпринять для разрешения сложившейся ситуации, закреплены в понятии *округления*.

Выделяют следующие виды округления:

> I. **Вниз** (к $-\infty$)

> Число всегда округляется в меньшую сторону — иными словами, последний разряд всегда остаётся неизменным.

> II. **Вверх** (к $+\infty$)
>
> Число округляется в бо̀льшую сторону — иными словами, к последнему разряду всегда прибавляется единица.

> III. **К нулю**
>
> Положительные числа округляются вниз, отрицательные — вверх.

> IV. **От нуля**
>
> Положительные числа округляются вверх, отрицательные — вниз.

> V. **Математическое округление**
>
> Если значение излишка строго меньше половины последнего разряда, число округляется к нулю, иначе — от нуля.

Математическое округление — метод, к которому все привыкли со школы. Он кажется наиболее «правильным», но у него есть одна проблема: в десятичном случае к нулю округляются 4 цифры из 10, а от нуля — 5 цифр. Так происходит потому, что для цифры 0 никакое округление вообще не требуется. Так, если мы сгенерируем достаточно много равномерно распределенных случайных чисел от 0 до 1 и математически их округлим, то их среднее арифметическое внезапно будет заметно превосходить ожидаемое 0.5. По этой причине вводится

> VI. **Округление к ближайшему чётному**
>
> Если значение излишка строго меньше половины последнего разряда, число округляется к нулю. Если занчение излишка строго больше половины последнего разряда, число округляется от нуля. В противном случае число округляется к ближайшему чётному.
>
> Об этом способе также можно думать так: *округление к ближайшему, иначе — к чётному*.

Достаточно ясно, как это решает проблему неравномерности распределения округлённых значений: мы в одинаковом количестве случаев округляем к нулю и от нуля, а в единственном оставшемся — когда излишек в точности равен половине сохраняемого разряда — в среднем в половине случаев мы округлим к нулю, и так же в половине — от нуля, статистически сохранив равномерное распределение.

Арифметика с фиксированной точкой

Числа с фиксированной точкой во многом похожи на целые и, что особенно здорово, складываются и вычитаются они тоже как целые. С умножением, в свою очередь, снова возникают проблемы, причем даже несколько другого сорта.

Из-за того, что, как уже было отмечено, точка существует лишь в нашей больной фантазии, практически мы работаем с записью числа в виде $x \times 2^8$, и при умножении двух значений получаем результат в форме $x \times y \times 2^{16}$, что, конечно, не совсем нашему запросу удовлетворяет. Понятно, что результат необходимо дополнительно сдвинуть на 16 бит вправо.

Здесь мы трагически упираемся в тот факт, что в нашем положении умножение с «обрезанием» не прокатит, поскольку нам нужны исключительно старшие биты результата. Мы, к счастью, уже знаем, как с этим быть. Самого простого решения — повысить разрядность операндов — обычно бывает достаточно. Если этого сделать нельзя, приходится грустить и выдумать что-то более интеллектуальное.

Стоит также отметить, что по умолчанию при умножении как правило используется округление к нулю. Повлиять на это можно, анализируя остаток от деления.

Числа с плавающей точкой

Примечание. *Арифметика с плавающей точкой далее рассмотрена не будет, потому что эта тема достаточно ёмко покрыта в стандарте, а у автора нет желания заниматься тупым переводом.*

Общие представления

Работать с числами с фиксированной точкой достаточно удобно. Этот факт заставляет задуматься о том, что, наверное, с ними не так уж всё и радужно, поскольку не бывает такого, чтобы что-то полезное нормально работало. К сожалению, практика эти опасения подвтерждает.

Основной недостаток чисел с фиксированной точкой — ограниченный диапазон, которого далеко не всегда достаточно для вычислений, которые мы хотим проводить. Конечно, его можно расширить, увеличив размер числа, но, к сожалению, мы в этом плане неизбежно аппаратно ограничены. По этой причине нам хотелось бы придумать способ умещать в тот же объём памяти бо̀льший диапазон значений.

Здесь мы вспоминаем школьные уроки физики и *показательную форму* записи вещественного чила:

$$3.104 \times 10^7$$

По сравнению с обычной формой, она обладает несколькими основными преимуществами. Во-первых, она достаточно удобна при работе с очень большими и очень маленькими числами. Во-вторых, она позволяет записывать числа ровно с той точностью, которая нам известна, не добавляя незначащие или некорректные нули.

Ещё одно важное замечание заключается в характере погрешности, с которой работает фиксированная и плавающая точка.
Числа с фиксированной точкой закрепляют *абсолютную* погрешность, то есть вне зависимости от значения числа нам известно некоторое фиксированное количество его разрядов. При переходе к плавающей точке погрешность становится *относительной*, то есть нам гарантируется, что она не превышает какой-то доли исходного числа — иными словами, относительная погрешность тем выше, чем больше значение. Это достаточно логично и естественно, и, что главное, освобождает нас от необходимости соблюдать строгий формат, в который мы упираемся при работе с фиксированной точкой.

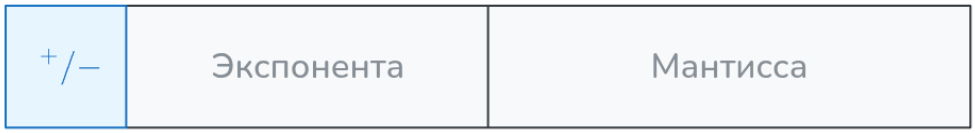
IEEE 754

Хотя, конечно, отсутствие строгого формата может стать ещё бо̀льшей проблемой, чем какая-то там неточность каких-то там вычислений. Число в форме с плавающей точкой практически представляет собой два числа: *мантиссу*, которая отвечает непосредственно за значение, и *экспоненту*, которая определяет сдвиг запятой. Легко видеть, что и мантисса, и экспонента должны иметь знак. В этот момент возникает непозволительное количество неоднозначностей: сколько битов выделять на каждую часть? как кодировать каждую часть? как их располагать?

И ситуация хуже, чем кажется. Предположим, что мы написали алгоритм, который использует в качестве константы число π, закодированное в форме с плавающей точкой, после чего перенесли его на другую машину. Если мы никак не регулируем формат таких чисел, то в лучшем случае наш алгоритм, скорее всего, просто выдаст неточный результат, а в худшем — вообще не сработает.

Умные люди из Института инженеров электротехники и электроники посмотрели на это досадное обстоятельство, а также на предыдущие попытки стандартизации чисел с плавающей точкой, после чего немного подумали и в 1985 году опубликовали IEEE 754 — стандарт, которого придерживается почти всё современное железо, когда дело доходит до вычислений в показательной форме.

Мантисса кодируется в форме *бит под знак*, причём знаковый бит хранится отдельно и занимает старший разряд числа, в то время как модуль мантиссы записывается в младшие разряды. Между знаком и модулем мантиссы хранится показатель экспоненты, который записывается в *форме со сдвигом*:



Общий вид числа с плавающей точкой

Важно, что сдвиг экспоненты при этом округляется *вниз*.

Заметим, что точку в числе можно произвольно сдвигать, если изменять при этом экспоненту соответствующим образом. Это позволяет нам записывать числа в *нормализованной форме*, устанавливая точку после первого значащего разряда. В частности, все числа (за исключением нуля, который будет рассмотрен отдельно) после нормализации будут начинаться на `1.`, поэтому эти разряды практически не несут информации и, как следствие, не сохраняются.

Числа с плавающей точкой по умолчанию округляются *к ближайшему чётному*.

На вопрос того, сколько всё-таки выделять бит под каждую часть, стандарт предлагает несколько вариантов в зависимости от общего объема занимаемой памяти:

	Общий размер	Экспонента	Мантисса
Half precision	16	5	10
Single precision	32	8	23
Double precision	64	11	52
Quad precision	128	15	112

Примечательно, что half precision долгое время использовался только для хранения, но не для вычислений. Однако современное железо всё активнее начинает вводить поддержку вычислений в этом формате, потому что «бесплатная» скорость, увы, закончилась, и самый простой вариант ускорения вычислений — снижение точности.

Отдельно стоит оценить пределы ёмкостей основных из этих типов:

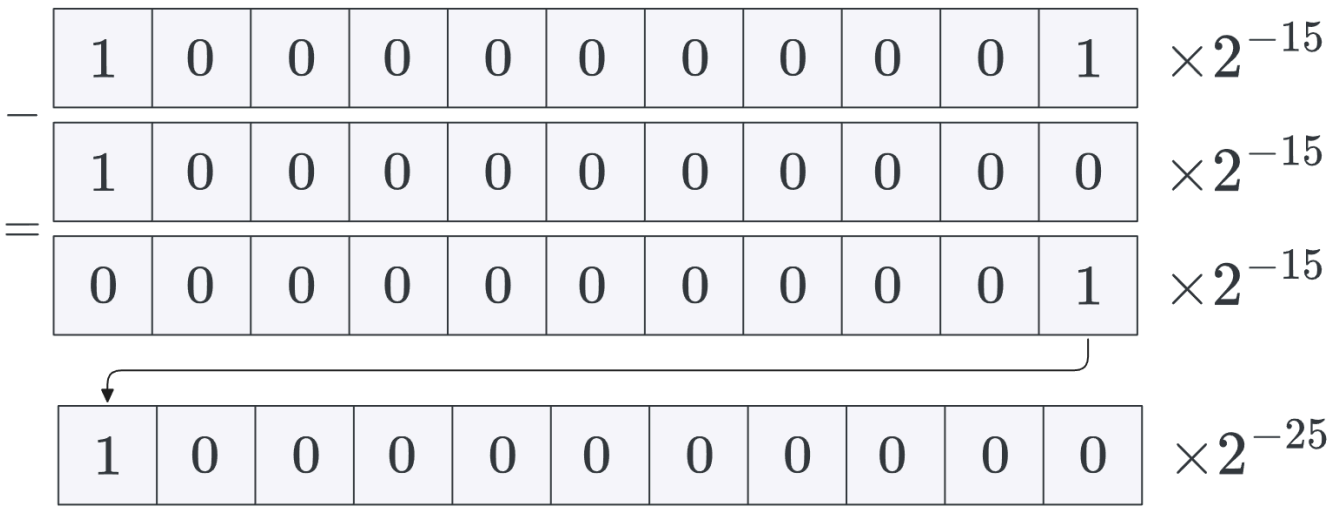
Half precision	Минимальное положительное — $2^{-15} \approx 10^{-5}$ Максимальное положительное — $\sim 2^{17} \approx 10^{6}$
Single precision	Минимальное положительное — $2^{-127} \approx 10^{-39}$ Максимальное положительное — $\sim 2^{129} \approx 10^{38}$
Double precision	Минимальное положительное — $2^{-1023} \approx 10^{-308}$ Максимальное положительное — $\sim 2^{1025} \approx 10^{308}$

Примечание. Эти расчёты используют некоторые крайние значения, которые мы в дальнейшем исключим из рассмотрения, из-за чего их следует воспринимать как сугубо теоретические и совершенно не как точные.

Денормализация

Рассмотрим минимальное число в форме half precision — $1.0000000000 \times 2^{-15}$ — и следующее за ним — $1.0000000001 \times 2^{-15}$ — и назовём их a и b. Мы можем однозначно сказать, что $a \neq b$ и, более того, что $a < b$.

Тем не менее, если мы попытаемся вычислить их разность, после нормализации она составит 2^{-25}:



Ясно, что наш формат не позволяет нам сохранять настолько точные числа, поэтому оно округлится до 0, приводя нас к весьма забавной ситуации: два различных числа при

вычитании дают ноль. Эта проблема принципиальна относительно самого построения чисел с плавающей точкой, то есть увеличение точности никак ситуацию не спасает, и такая патологическая пара будет всегда. Это поведение очень неприятно с алгоритмической точки зрения, и видеть его мы очень-очень не хотим.

Стандарт разрешает это недоразумение очень просто: отключает нормализацию.
Из диапазона «нормальных» значений экспоненты исключается самое меньшее и отводится ровно под эту задачу. Практически говоря, минимальное, то есть нулевое, значение экспоненты имеет следующий эффект:

1. экспонента принимает минимальное возможное «нормальное» значение (т.е. 2^{-14} в half precision);
2. реальное занчение мантиссы записывается, начиная с `0.`, т.е. число не нормализуется.

Введение денормализованных чисел позволяет нам расширить диапазон сохраняемых значений в сторону нуля, полностью решая исходную проблему. Более того, эта форма даёт нам единственный и самый что ни на есть естественный код для нуля — все нулевые биты. Денормализованная форма даже позволяет придать некоторый смысл минус нулю: его можно воспринимать как некоторый малый эпсилон, который настолько близок к нулю, что мы не знаем ни одного его бита, но нам точно известно, что он строго меньше нуля.

Стоит при этом отметить, что мы снова несколько жертвуем точностью, поскольку если при нормализации мы «бесплатно» получили один дополнительный бит в мантиссе от неявно сохранённой старшей единицы, то у денормализованных чисел старший разряд равен нулю, поэтому информацию несут лишь те биты, которые мы явно сохраняем.

Денормализованные числа не бесплатны и с точки зрения архитектуры, поэтому их повсеместная поддержка — вопрос неоднозначный. Из рубрики забавных фактов: они полностью отсутствовали в PlayStation 2, с чем связаны определенные трудности при попытке эмуляции этой консоли на современном железе.
В наше время они, впрочем, поддерживаются почти всеми нормальными процессорами, однако могут очень ощутимо (до 100 раз на процессорах Intel) замедлять вычисления.

Не числа

Нулевой код — не единственное специальное значение экспоненты: максимальная её величина, то есть код из всех единиц, отводится под целых два отдельных случая.

Максимальная экспонента и нулевая мантисса вместе кодируют *бесконечность*. Так, при переполнении в числах с плавающей точкой мы получаем не результат по модулю, а самую настоящую бесконечность, которая ведет себя совершенно корректно с математической точки зрения.

Запись числа $+\infty$ в форме half precision

Случай ненулевой мантиссы более примечателен. Число с таким кодом — это не число, то есть NaN (*Not a Number*):

Запись не числа в форме half precision

NaN возникает в результате математически некорректных операций: деления нуля на ноль, сложения плюс и минус бесконечности и прочих запросов, которые вычислительный модуль не в состоянии нормально обработать просто в силу природы чисел. Создатели стандарта решили не додумывать за природой, к какому странному результату приводят не менее странные действия, а вместо этого договорились просто сообщать программисту об ошибке. NaN — индикатор чего-то неправильного. Важно отметить, что при этом мы хотим именно получать сообщение об ошибке, а не аварийно завершать программу, поскольку результат вычислений, которые привели нас к NaN, может вообще никак не использоваться.

Главное свойство NaN как сущности — выживаемость. Ему глубоко всё равно на любое воздействие, то есть результатом *любой* арифметической операции с NaN должен быть NaN... должен, но не может же у нас всё быть совсем как у людей, так что NaN в нулевой степени и единица в степени NaN всё-таки могут давать единицу, если NaN тихий (*см. далее*).
И всё-таки в общем случае NaN сохраняется практически во всех вычислениях, и цель этого более чем ясна: если где-то возникла такая неприятная ошибка, то мы скорее хотим, чтобы она не затерялась в каких-то промежуточных вычислениях и точно дошла до нас.

Результат любой операции сравнения с NaN, кроме `!=`, равен `false`. В случае операции `!=` результат всегда равен `true`, откуда, в частности, следует, что NaN не равен NaN, то есть условие вида

```
if (x != x) {
    // ...
```

```
}
```

не лишено смысла, если `x` — число с плавающей точкой, поскольку оно позволяет проверить, является ли `x`, собственно, числом.

Стандарт также вводит дополнительную классификацию NaN, разделяя их на *тихие* и *сигнальные*. Отличительная особенность сигнальных NaN в том, что они провоцируют аппаратное исключение при попытке выполнить некорректную операцию. Вид NaN определяется старшим битом мантиссы, но, как ни досадно, первый стандарт IEEE 754 забыл уточнить, какой из них как задаётся. Потом, конечно, они всё же уточнили, что *рекомендуется* использовать для тихих NaN единицу, а для сигнальных — ноль, и, например, архитектура x86 использует ровно эту раскладку.

В результате операции может получиться только тихий NaN. В случае операции с двумя NaN стандарт не регулирует, какую из мантисс унаследует результат, но получившийся NaN всё ещё обязан быть тихим; это достигается засчёт того, что мы копируем не всю мантиссу, а ту ее часть, которая в стандарте названа *payload* — это, собственно, вся мантисса без старшего разряда.

Околостандартные типы

Помимо основных форматов, закреплённых в IEEE 754, существуют и другие широко распространенные конфигурации чисел с плавающей точкой. Так, среди них можно выделить:

> Extended precision
>
> | Общий размер | Экспонента | Мантисса |
> |---|---|---|
> | 80 | 15 | 64 |
>
> Этот тип «расширяет» диапазон double precision с минимальными затратами по памяти: размер экспоненты в нём совпадает с quad precision, но мантисса короче. Его важная особенность в том, что он явно сохраняет ведущую единицу в мантиссе.

> Brain float
>
> | Общий размер | Экспонента | Мантисса |
> |---|---|---|
> | 16 | 8 | 7 |
>
> О bfloat можно думать как о single precision, у которого порезали хвост. Поэтому его, в частности, легко в этот самый single precision конвертировать, просто дополнив младшие биты недостающими нолями.

> E5M2 (bf8)
>
> | Общий размер | Экспонента | Мантисса |
> |---|---|---|
> | 8 | 5 | 2 |

Практически, порезанный half precision. Это «довольно стандартный» тип, и всё работает в нём ровно так же, как и в half precision.

E4M3FN

Общий размер	Экспонента	Мантисса
8	4	3

FN в названии типа означает «finite» — *конечный*. Этот формат не резервирует код под хранение бесконечности, а также имеет всего два значения для кодирования NaN.

E4M3FNUZ

Общий размер	Экспонента	Мантисса
8	4	3

UZ в названии типа означает «unsigned zero». Под кодирование ноля выделен единственный код — все нули — а код минус нуля отдан под единственный NaN. Смещение экспоненты здесь округляется вверх.

E2M1

Общий размер	Экспонента	Мантисса
4	2	1

Это может показаться совсем смешным, но бывает и такое.
Числа в формате E2M1, впрочем, не совсем «самостоятельны» и обычно организованы в блоки по 32 значения, для которых дополнительно сохраняется общий множитель в формате E8M0; поэтому их вид ещё можно записать в форме MXE2M1.

NVFP4

Общий размер	Экспонента	Мантисса
4	2	1

Писк моды. Тот же E2M1, только блок хранит 16 чисел и единственный множитель в формате E4M3FN.

Практически все эти типы реально поддерживаются в современном железе, хотя здесь стоит сделать ремарку насчёт того, что значит «поддерживаются». Если для всех адекватных типов, в число которых входят четыре основных и extended precision, подразумевается полная поддержка, то для чисел-малышей поддерживается одна единственная операция — матричное умножение.

Плавающая точка в Си

Поддержка стандартных типов

В языках C/C++ описанные типы представлены стандартными типами данных. Легко догадаться, что `float` — это single precision, а `double` — это double precision. Если не вспоминать про существование `long double`, то всё хорошо.

Если вспомнить про существование `long double`, то всё не очень хорошо. Проблема в том, что он может быть представлен single, quad или extended precision, и заведомо достаточно трудно определить, какой из этих вариантов будет реализован на конкретной системе. Это поведение можно регулировать специальными ключами компилятора, но даже так мы не можем гарантировать, что на других системах нужные нам типы будут поддерживаться. Вывод из этого следует очень простой: не надо использовать `long double` без чёткого понимания, что и зачем мы делаем. Теперь мы про него забыли, и всё снова хорошо.

Что касается half precision, то в 23-м стандарте Си появился тип `_Float16`, который реализует этот формат. В C++ начиная, соответственно, с C++23, для этого используется тип `std::float16_t`.
При этом стоит помнить, что компиляторы — ребята довольно консервативные, и вполне вероятно, что без указания соответствующих ключей эти типы могут компилироваться в программной эмуляции и работать страшно медленно, поскольку, как уже было упомянуто, полноценная поддержка half precision появилась в железе только недавно.

Ловкость рук и никакого мошенничества

С программной точки зрения в плавающей точке также возникает бесчисленное множество спецэффектов. Так, например, мы ни в коем случае не можем гарантировать ассоциативность сложения в таком формате, то есть, более формально,

$$\exists a, b, c : (a + b) + c \neq a + (b + c)$$

Мы можем привести даже не один такой пример. Если, допустим, a и b — очень большие положительные числа, а c — очень большое отрицательное, то $(a + b) + c$ вернёт нам $+\infty$, а $a + (b + c)$ — какое-то адекватное число. Схожая ситуация может возникнуть, например, если числа b и c составляют $^1/_3$ младшего разряда числа a, и связана она в этом случае с работой механизма округления к ближайшему четному.

Другой презабавный артефакт — из $\neg(a > b)$ совершенно не следует $a \leq b$, потому что существуют не-числа, которые полностью ломают нам операции сравнения.

Всё ещё веселее, если считать в комплексных числах (которые, кстати, полностью поддерживаются в Си и не поддерживаются в C++ — там под комплексными числами понимаются некоторые шаблоны, которые, конечно, никакой нормальной конфигурации не подлежат, неизбежно компилируясь в то, во что компилируются): например, если мы хотим посчитать модуль комплексного числа 10^{20}, то получим, внезапно, не 10^{20}, а $+\infty$.

Такие досадные ограничения сильно связывают компилятору руки в вопросе оптимизаций, но, конечно, мы можем самостоятельно их ему развязать при помощи соответствующих флагов. Это мероприятие неприятно скажется на точности вычислений, однако, скорее всего, ощутимо поднимет производительность нашей программы.

Источники

1. Скаков П. С. — Лекции по аппаратному обеспечению вычислительных систем, 2 семестр, 2026
2. Э. Таненбаум — Архитектура компьютера (https://jasulib.org.kg/wp-content/uploads/2023/03/1-Tanenbaum_E_-_Arkhitektura_kompyutera_4-e_izdanie.pdf)
3. Дэвид М. Харрис, Сара Л. Харрис — Цифровая схемотехника и архитектура компьютера (https://is.ifmo.ru/books/2016/digital-design-and-computer-architecture-russian-translation_July16_2016.pdf)
4. Д. Паттерсон, Дж. Хеннеси — Архитектура компьютера и проектирование компьютерных систем (http://178.140.10.58:8083/read/831/pdf)
5. Andrew D. Booth — A Signed Binary Multiplication Technique (https://www.ece.ucdavis.edu/~bbaas/281/papers/Booth.1951.pdf)
6. Википедия — Алгоритм Бута (https://ru.wikipedia.org/wiki/%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%91%D1%83%D1%82%D0%B0)
7. Стандарт IEEE 754-2019 (https://www-users.cse.umn.edu/~vinals/tspot_files/phys4041/2020/IEEE%20Standard%20754-2019.pdf)