Конспект 01 — Представление чисел в памяти

Содержание

”В языке C++ можно очень коротко выразить очень большую глупость.”
— 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.

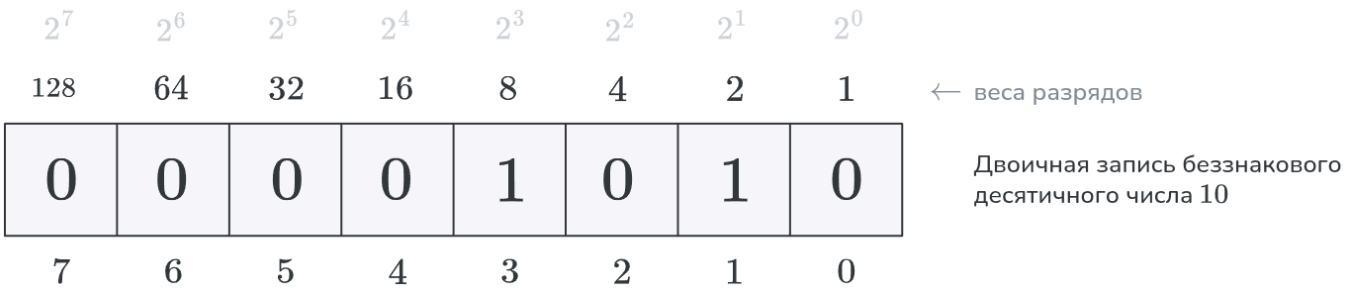
Размер байта может быть произвольным. Хотя практически вся современная вычислительная техника сошлась на значении в 8 бит, в прошлом мы могли иметь дело, например, с шестибитными или четырехбитными байтами — это, помимо прочего, можно считать одной из причин ряда странностей при работе с арифметическими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



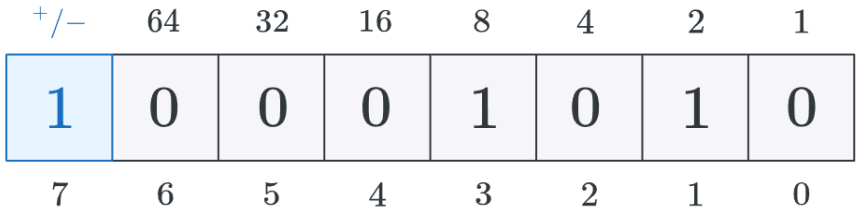
Биты в двоичной записи традиционно нумеруются с нуля, поскольку это позволяет удобно задавать веса разрядов. При этом однозначного порядка бит внутри одного байта не существует, поэтому вместо кажущейся порой естественной терминологии «первый» и

«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:

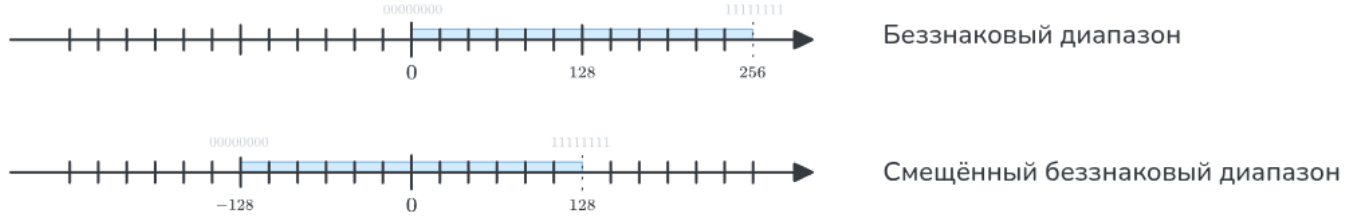


Двоичная запись беззнакового десятичного числа −10 в прямом коде

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*.

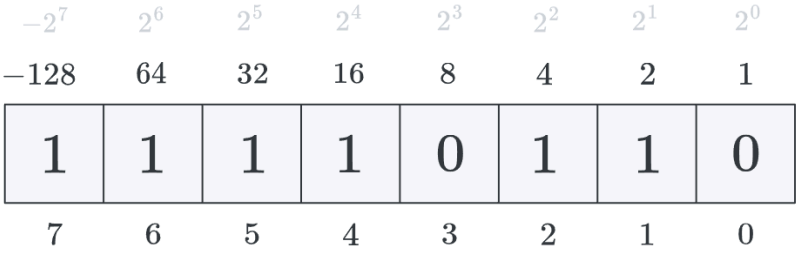
У него есть одна большая проблема. Диапазон его значений составляет `-127..127`, и размер этого диапазона внезапно равен 255, а не 256. Причина в том, что число 0 имеет два различных кода: `10000000` и `00000000`, которые между собой абсолютно равноправны. Даже если закрыть глаза на незначительную неэффективность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

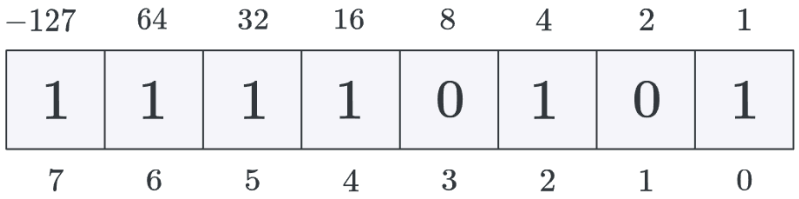
```
int x = std::numeric_limits<int>::min();
std::cout << abs(x);
```

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
...	...

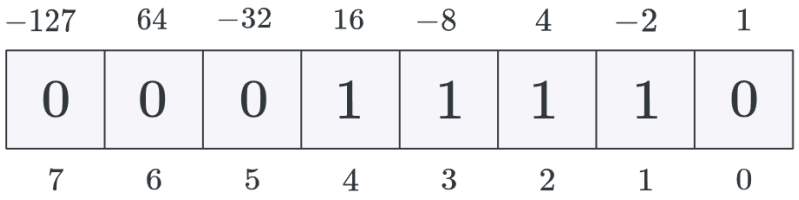
Легко видеть, что в такой схеме младший бит отвечает за знак, а все оставшиеся — за «почти модуль». Это означает, что для положительных чисел это буквально модуль, а для

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:

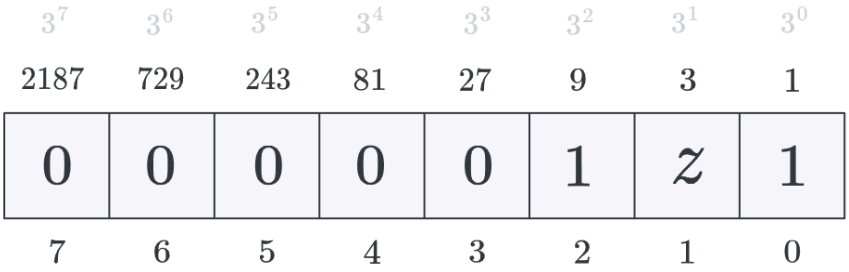


Двоичная запись знакового десятичного числа 10 в системе с основанием -2

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.



Двоичная запись десятичного числа 7 в троичной системе счисления

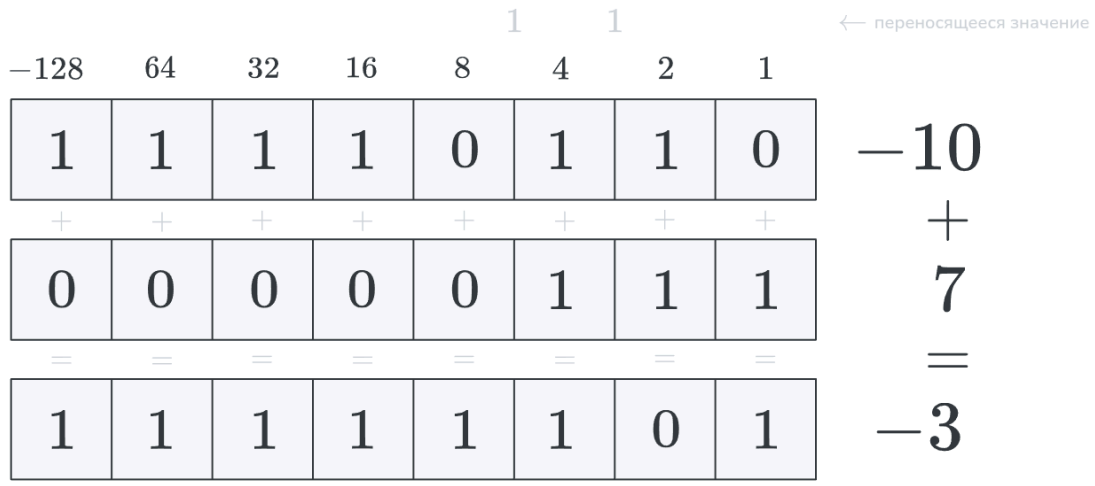
Диапазон в такой системе полный и полностью симметричный. Инверсия знака тривиальна. Система масштабируется влево нулями. В целом, всё супер, но — увы — мы всё ещё работаем в двоичной. Терпите.

Целочисленная арифметика

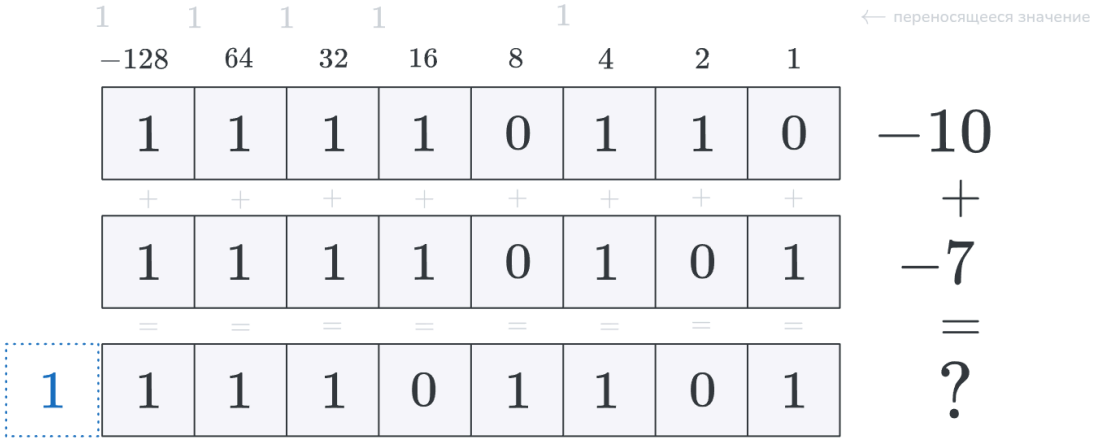
Проблема переполнения

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:



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

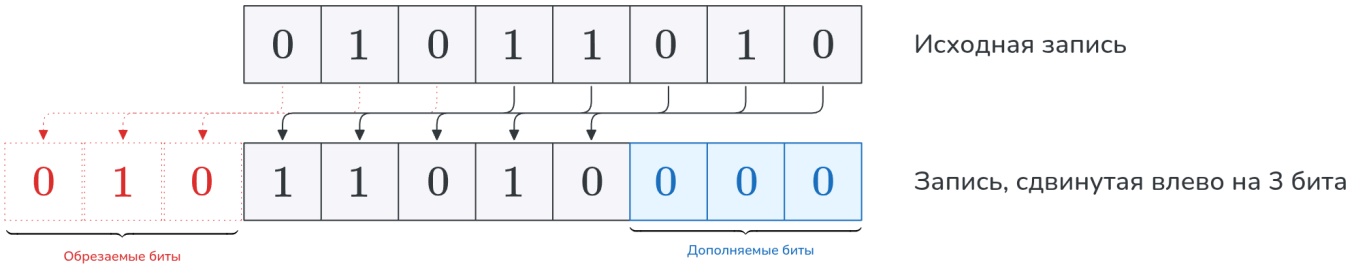
Поскольку инженеры, видимо, такие же ленивые, как и программисты, вычитание реализуется как сложение с противоположным числом, что, впрочем, достаточно разумно.

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:

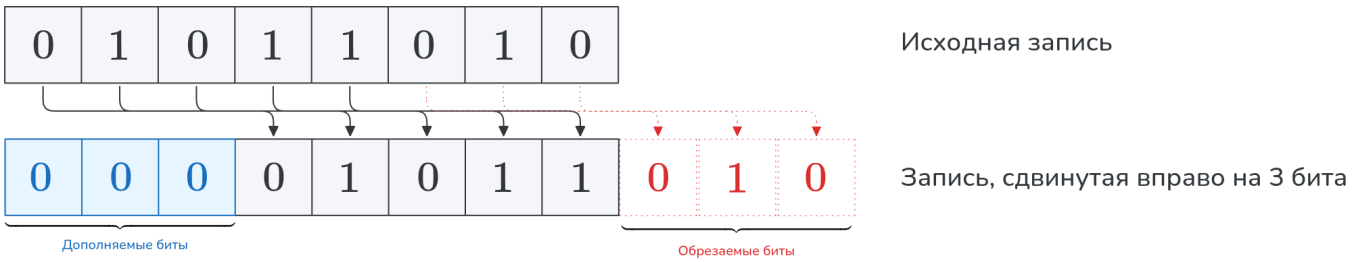
Эта операция называется *дополнением до двух* — поэтому, собственно, *дополнительный* код. Само это название мотивируется математической природой метода: инвертированное число — это в каком-то смысле «дополнение» до ближайшей степени двойки, превосходящей диапазон значений, поскольку сумма обычного и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

Битовые сдвиги

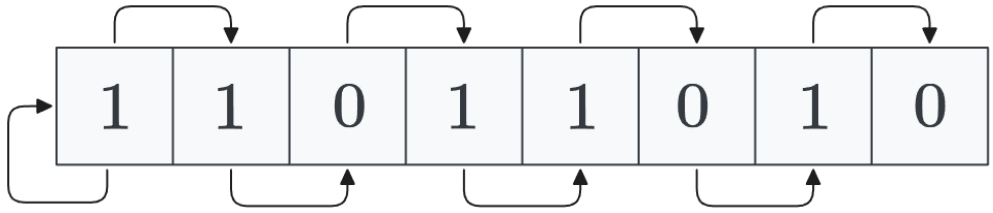
Двоичную запись числа можно сдвигать.
Битовый сдвиг влево практически представляет собой простое модулярное умножение на некоторую степень двойки — иными словами, все биты переходят на n позиций влево, излишки «обрезаются», а недостающие позиции заполняются нулями:



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:

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

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:

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.

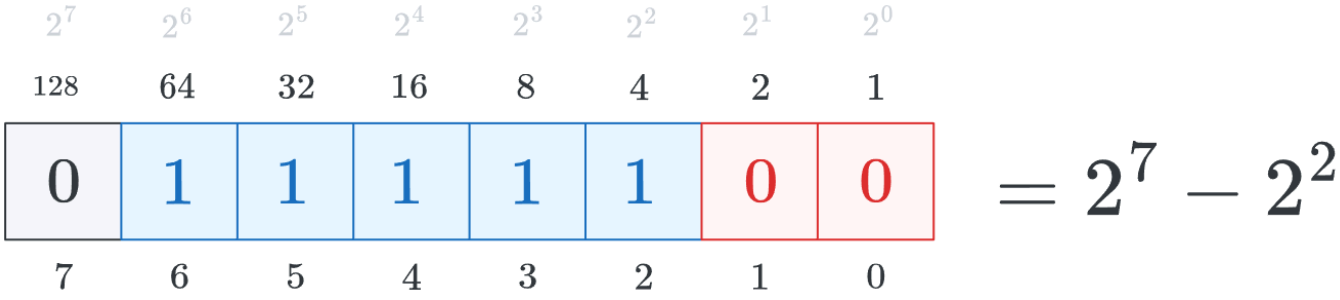
В качестве ещё одной альтернативы можно корректировать результат уже после «неправильного» умножения, поскольку ошибка на самом деле чётко определена благодая математической строгости дополнительного кода.

Алгоритм Бута

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.

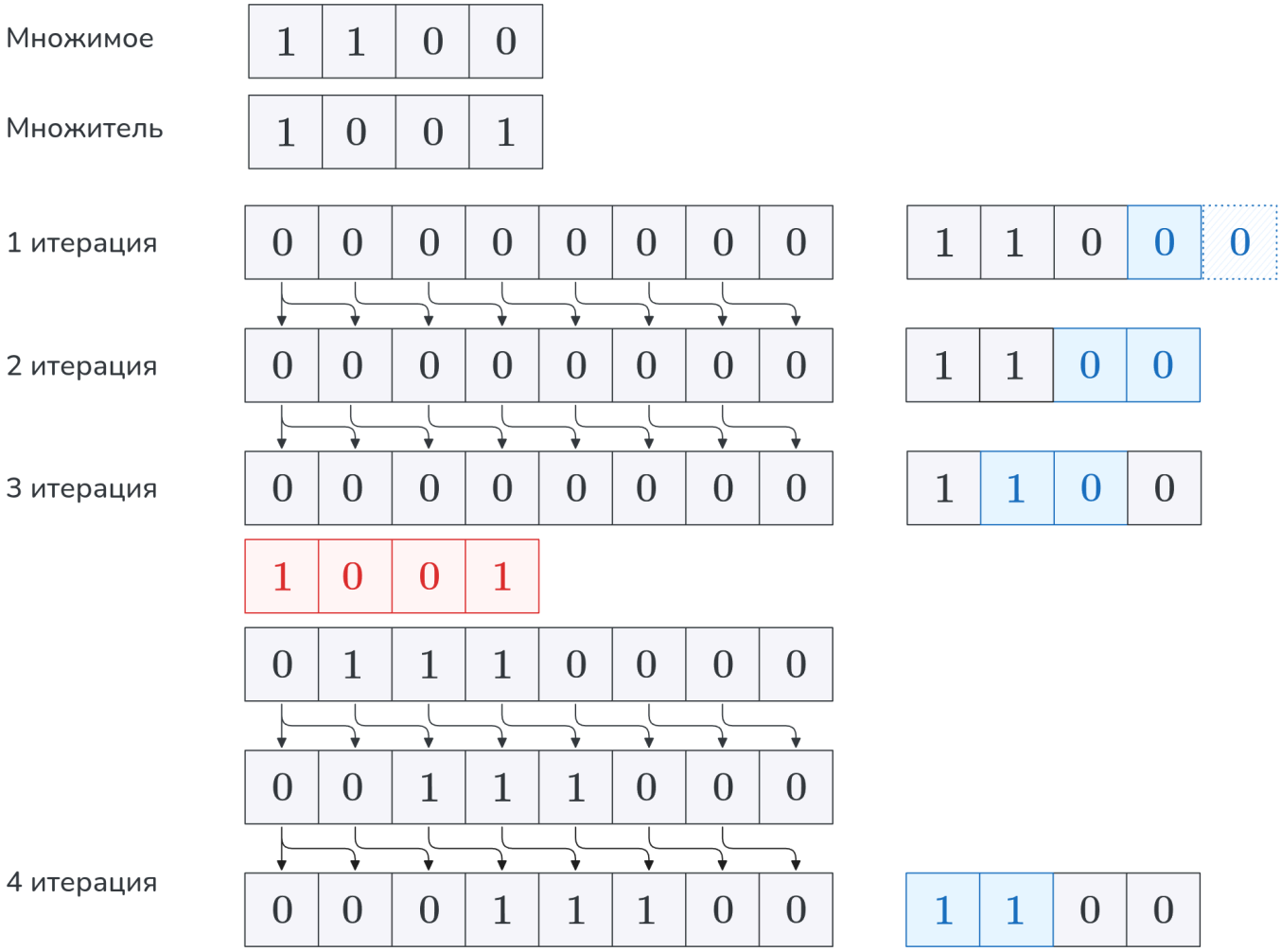
Алгоритм Бута пользуется тем тривиальным фактом, что любое двоичное число, представляющее собой непрерывную последовательность единиц, можно представить в виде единственной разности степеней двойки:



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.

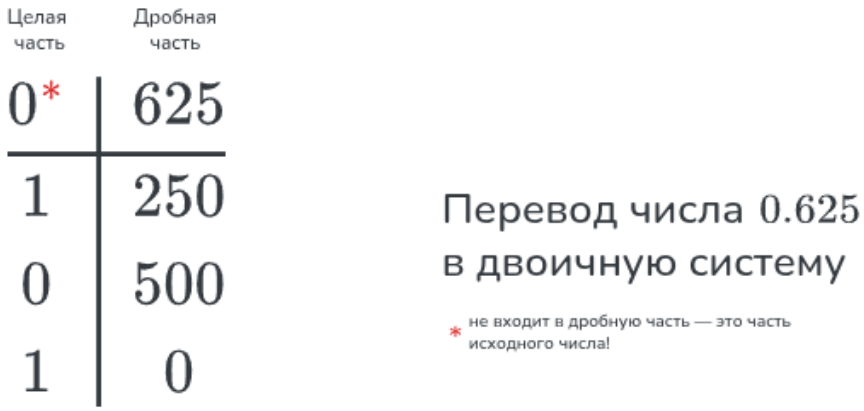
Числа с фиксированной точкой

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:



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
...	...

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 бо́льшую 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.

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)