Введение в машинную арифметику

Материал относится к курсу численного моделирования, посвящен представлению чисел в памяти компьютера и реализации арифметических операций с ними.

Представление 8-битных целых чисел в процессоре

Привычные нам числа в десятичной системе в процессоре представляются в двоичной системе. Рассмотрим на примере числа 100:

$$100_{10} = 1 \times 10^2 + 0 \times 10^1 + 0 \times 10^2 = 2^6 + 2^5 + 2^2 = 110010_2$$

В памяти компьютера 8-битное число представляется собственно 8-ю битами, то есть 1 байтом. Биты нумеруются справа налево - от 0 до 7. Представим биты в виде ячеек и получим следующую запись:

№ бита	7	6	5	4	3	2	1	0
Значение	0	1	1	0	0	1	0	0

Рассмотрим теперь число $200 = 2^7 + 2^6 + 2^3$. Заметим, что все степени двоек увеличились на 1, так как $200 = 100 \times 2$.

Запишем то, как выглядит 8-битное число в памяти компьютера:

№ бита	7	6	5	4	3	2	1	0
Значение	1	1	0	0	1	0	0	0

Отметим следующую особенность:

- Умножение на 2 смещает все биты влево
- Деление на 2 смещает все биты вправо

Потери и переполнения

Рассмотрим число $255 = 2^8 - 1 = \sum_{i=0}^{7} 2^i$. В Представление в памяти:

№ бита	7	6	5	4	3	2	1	0
Значение	1	1	1	1	1	1	1	1

Если к числу прибавить единицу, то в представлении памяти останутся только 0. Единица, которая могла бы быть в 8-м бите, сохраняется в специальном флаге процессора, однако в дальнейших вычислениях не используется.

№ бита		7	6	5	4	3	2	1	0
Значение	1	0	0	0	0	0	0	0	0

Получается, что в 8-битном представлении $255 + 1 = 0$

Сложение и вычитание целых чисел происходят по модулю 256

Примеры:

$100 + 200 = [2^8+]2^5 + 2^3 + 2^2 = 44$

Представление неотрицательных целых чисел в процессоре

Название типа	Размер	Значение
Byte (unsigned char, unsigned byte)	8 бит	$0..255$
Word (unsigned short)	16 бит	$0..65535$
Dword (unsigned, unsigned int, unsigned long)	32 бита	$0..4294967295$
Qword (unsigned long, unsigned long long)	64 бит	$0..2^{64}-1$

Представление, соответствующее архитектуре большинства нынешних процессоров – 64-битное.

Представление отрицательных чисел в процессоре

Число -100 в 8-битном представлении неотличимо от числа $256 - 100 = 156$.

$256 - 100 = 128 + 16 + 8 + 4 = 2^7 + 2^4 + 2^3 + 2^2 = [1|0|0|1|1|1|0|0]_2$

Рассмотрим более интересный вариант:

$-128 - 1 = 127[-256] = \sum_{i=0}^{6} 2^i = [0|1|1|1|1|1|1|1]_2$

Как видно, появляется необходимость различать, является число положительным или отрицательным.

Правило

Число является отрицательным, если в нем единица в самом старшем разряде. легко увидеть, что 127 – максимальное положительное число, которое можно так представить, а -128 это минимальное отрицательное.

> При попытке отнять от -128 единицу, мы получим 127, то есть перейдем от минимального числа к максимального («закольцуемся»)

$-100 - 100 = -200 = 56[-256] = 2^5 + 2^4 + 2^3 = [0|0|1|1|1|0|0|0]_2$

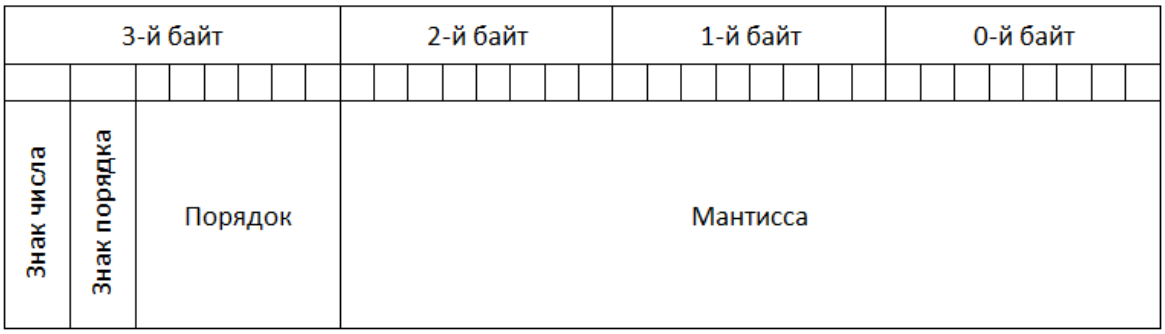
Название типа	Размер	Значение
Char (byte)	8 бит	$-128..127$
Short	16 бит	$-32768..32767$
Int (long)	32 бита	$-2147483648..2147483647$
Long (long long)	64 бит	$-2^{63}..2^{63}-1$

Представление чисел с плавающей точкой в процессоре (IEEE 754)

Числа с плавающей точкой тоже представляются в памяти компьютера как двоичные.

Например, это число имеет точное выражение в двоичной системе:

$$4.57815 = 4 + 1/2 + 1/16 + 1/64 = 2^2 + 2^{-1} + 2^{-4} + 2^{-6} = 2^2(1 + 2^{-3} + 2^{-6} + 2^{-8})$$



Экспонента – это показатель степени двойки в записи числа. В IEEE 754 записывается положительное число, равное сумме экспоненты и сдвига, где сдвиг равен 127.

Итак, запись числа 4.57815 : $[0|1000000|00100101000000000000000]$

Нормальное число = (1 - 2 x **Знак**) x 2 ^ (**Экспонента** - Сдвиг) x 1.**Мантисса**

Типы чисел с плавающей точкой

Название типа	**Размер, бит**	**Экспонента, бит**	**Мантисса, бит**	**Значения**
Float (single)	32	8	23	$-3.40282 \times 10^{38} .. 3.40282 \times 10^{38}$
Double (float)	64	11	52	$-1.79679 \times 10^{308} .. 1.79679 \times 10^{308}$
Extended (long double)	80	15	63	$-1.18973 \times 10^{4932} .. 1.18973 \times 10^{4932}$
Quadruple (float128)	128	15	112	$-1.18973 \times 10^{4932} .. 1.18973 \times 10^{4932}$

Чем больше длина мантиссы, тем точнее представление числа (больше двоичных знаков после точки).

Наиболее распространенными являются Single и Double. Тип Extended гораздо реже поддерживается в языках программирования и аппаратных режимах работы процессора. Хотя скорость работы с этим типом сильно не отличается от скорости работы с Float на большинстве операций.

Quadruple не поддерживается современными процессорами, практических нужд его использовать нет.

Проблемы с точностью

Возьмем число $1/3 = 2^{-2}(1 + 2^{-2} + 2^{-4} + 2^{-6}\dots)$. Как видно, точно записать его в таком виде не представляется возможным, учитывая, что мы имеем конечное число бит.

Даже число 0.1 не имеет точного выражения в виде числа с плавающей точкой.

Для таких чисел возможно только приближенное представление.

Специальные числа с плавающей точкой

Знак	Экспонента	Мантисса	Значение
0	00000000	$000\dots0$	$+0$
1	00000000	$000\dots0$	-0
0/1	00000000	$\neq 0$	Денормализованное число
0	00000000	$000..0$	$+\infty$
1	11111111	$000\dots0$	$-\infty$
	11111111	$\neq 0$	NaN (not a number)
	прочее		Нормальное число

- Как видно, нули имеют знак, так как первый бит может быть как 1, так и 0.
- NaN - не число, то есть что-то, что не может быть результатом математической операции.

Специальные числа: нули и бесконечности

Числа с плавающей точкой кардинально отличаются от целых чисел тем, что при их переполнении не происходит обнуления значащих битов и «закольцовывания» результатов. Если результат не может быть выражен как число с плавающей точкой, то в него записывается $+\infty$ или $-\infty$. Приведем несколько примеров:

$10^{20} \times 10^{20} = +\infty$

$\ln(\pm 0) = -\infty$

$\sqrt{+\infty} = +\infty$

$1/\pm 10^{-39} = \pm\infty \quad (\text{single})$

$1/\pm 10^{-309} = \pm\infty \quad (\text{double})$

$+\infty > 0 \qquad -\infty < 0$

$x - x = +0$

$x/\pm 0 = \pm\infty, x > 0 \qquad x/\pm 0 = \mp\infty, x < 0$

$+0 \times x = -0, x < 0, x \neq -\infty$

$+0 \times -0 = -0 \qquad -0 \times -0 = +0 \qquad +0 = -0$

$\pm\infty + x = \pm\infty, x \neq -\infty$

$\pm\infty \times x = \pm\infty, x > 0$

$\pm\infty \times x = \mp\infty, x < 0$

Специальные числа: NaN

$\ln(x) = \text{NaN}, x < 0$

$\sqrt{x} = \text{NaN}, x < 0$

$0/0 = \text{NaN}$

$\infty - \infty = \text{NaN}$

$0 \times \infty = \text{NaN}$

$\infty/\infty = \text{NaN}$

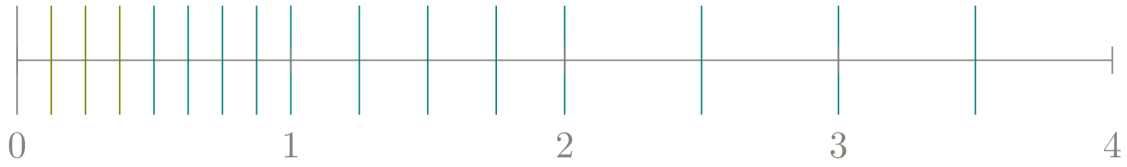
$0 \times \text{NaN} = \text{NaN}$

$\text{NaN}/\infty = \text{NaN}$

Денормализованные числа

Деномализованное число = 1 - (2 x **Знак**) * 2 ^ (-Сдвиг) * 0.**Мантисса**

Как видно, у них нет единицы в мантиссе, как было у нормального числа.



Если изобразить нормальные числа вот такими рисками, то при вычитании «соседних» чисел друг из друга, разность не будет выражаться в нормальном числе (точнее в нормальном числе получится 0).

Поэтому вводится специальный тип денормализованных чисел таким образом, чтобы однозначно определить:

$a = b \Leftrightarrow a - b = 0$

Ошибки округления чисел с плавающей точкой

Проблема не в точности самих чисел – она вполне достаточна для величин в физике.

Гарантируется корректное округление всех операций и библиотечных функций во всех значащих битах (в single и double, но не extended). Тем не менее (1):

$1 + 2^{-23} = 1.00000011920928955078125$

$1 + 2^{-24} \approx 1 \quad (\text{single})$

Из-за этих погрешностей нарушаются привычные законы арифметики:

- Ассоциативность: $(x + y) + z \neq x + (y + z)$
- Дистрибутивность: $(x + y) \times z \neq x \times z + y \times z$
- $x/y \neq x \times (1/y)$
- $(a + b) - (a + c) \neq b - c$. Если $a + b$ и $a + c$ большие, а их разность маленькая, то из-за ошибок округления равенство не будет выполнено. Пример (1) является частным случаем.
- $x + y = x, y \neq 0$ - и такое может происходить.

Контроль ошибки округления

Контроль возможен, потому что в процессоре гарантируется корректность значащих битов, последний бит округляется в правильную сторону.

Контроль ошибки суммы:

$c = a + b$

$e = (c - a) - b$

Таким образом «получаем обратно» младшие биты мантиссы, которые обусловлены разницей $c - a$. Если из этого вычесть b, то «вернем» биты, которые были в b и не попали в c. Таким образом получим точное значение ошибки округления. Иногда может быть, что $e = 0$, а если это не так, то ошибку можно учесть.

см. Kahan summation – алгоритм, предназначенный для суммирования n чисел. При суммировании массива можно учесть ошибки сложения предыдущих элементов.

Контроль ошибки произведения

см. fused multiply-add - специальная инструкция процессора, которая позволяет сразу выполнить сложение и умножение, гарантируя корректность всех значащих битов результата.

$a = a + b \times c$

Использование такой операции позволяет оценить ошибку произведения $b \times c$

Бывают и другие контроли ошибок.

Прочие виды чисел

Такие виды непосредственно в процессоре не реализованы, но эмулируются специализированными библиотеками, хоть и с потерей по времени

- Double-double: сумма большого и маленького числа
- Decimal: десятичный – актуален в финансовых расчетах, где важно округление десятичных разрядов, а не двоичных
- Bignum: «резиновые числа» – количество значащих битов не ограничено (хотя все же имеется некоторый барьер размера числа). Можно считать числами произвольной точности
- Rational: рациональные числа – полезны для алгебраических расчетов. Числитель и знаменатель представлены целыми числами
- Fixed-point: числа с фиксированной точкой, целые числа, которые трактуются так, что между какими-то двумя соседними битами была поставлена точка. Актуальны в приложениях, где порядки участвующих в расчетах числах известны и не очень различаются.