Арифметика в компьютерах

Егор Суворов

Курс «Парадигмы и языки программирования», подгруппа 3

Понедельник, 16 октября 2016 года

План занятия

1. Целые числа
   - Физическая часть
   - Типы данных
   - Беззнаковые числа
   - Знаковые числа
   - Порядок байт
2. Вещественные числа
   - Идеи
   - Детали
   - IEEE 754
   - Практические последствия
   - Прочее безумие
3. Дополнительные ссылки

Упрощённое представление о происходящем в «железе»:

1. Любой сигнал (в том числе бит) — это напряжение на проводе.
2. Два уровня напряжения распознавать проще, чем три.
3. Но три тоже было, не прижилось.
4. Вся логика построена на основе бинарных функций «И», «ИЛИ» и остальных (*гейты*)
5. Чем меньше гейтов — тем быстрее работает, тем меньше схема.
6. Числа надо складывать, вычитать, умножать, делить, сравнивать на равенство и меньше/больше.

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	0xC3

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	0xC3
1100 0011	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	0xC3
1100 0011	ret

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	0xC3
1100 0011	`ret`
0110 1000	

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

0001 0110	22
1000 0010	130
1000 0010	-126
0011 0000	48
0011 0000	'0'
1100 0011	0xC3
1100 0011	ret
0110 1000	22

Играем в игру

Чему соответствует бинарная запись в таблице ниже?
Используйте калькулятор или Python: `0b0100` и `int('1111', 2)`.

`0001 0110`	22
`1000 0010`	130
`1000 0010`	-126
`0011 0000`	48
`0011 0000`	`'0'`
`1100 0011`	`0xC3`
`1100 0011`	`ret`
`0110 1000`	22

Мораль: битовое представление ничего не говорит, если мы не договорились о том, как его интерпретировать («тип»).
Более того, представлений у одной и той же сущности может быть в некотором смысле много (`0xC3`, 195, `ret`).

Ликбез-1

- Основные типы чисел: целое, с фиксированной запятой, с плавающей запятой.
- Про строки и кодировки не говорим, там тоже довольно весело и интересно.
- Железо сейчас в основном поддерживает целые числа и с плавающей запятой.
- Железо умеет получать доступ к байту в памяти по его *адресу*.
- Считаем, что адрес — это некоторое целое неотрицательное число.

Ликбез-2

- Железо не может адресовать что-то внутри байта (биты).
- Но мы можем выполнять какие-то арифметические операции с байтами.
- Про порядок бит внутри байта говорить бессмысленно — мы никак его не проверим, у нас есть только арифметические операции.
- Будем рисовать *младшие/менее значимые* биты справа, как будто нормальные числа):

$$00010010_2 = 18_{10}$$

- Если какая-то конструкция занимает несколько байт подряд, то важно, в каком порядке они идут.
- Будем рисовать память слева направа: нулевой байт, первый...

Один байт

- Любое целое число можно представить в двоичной системе счисления:

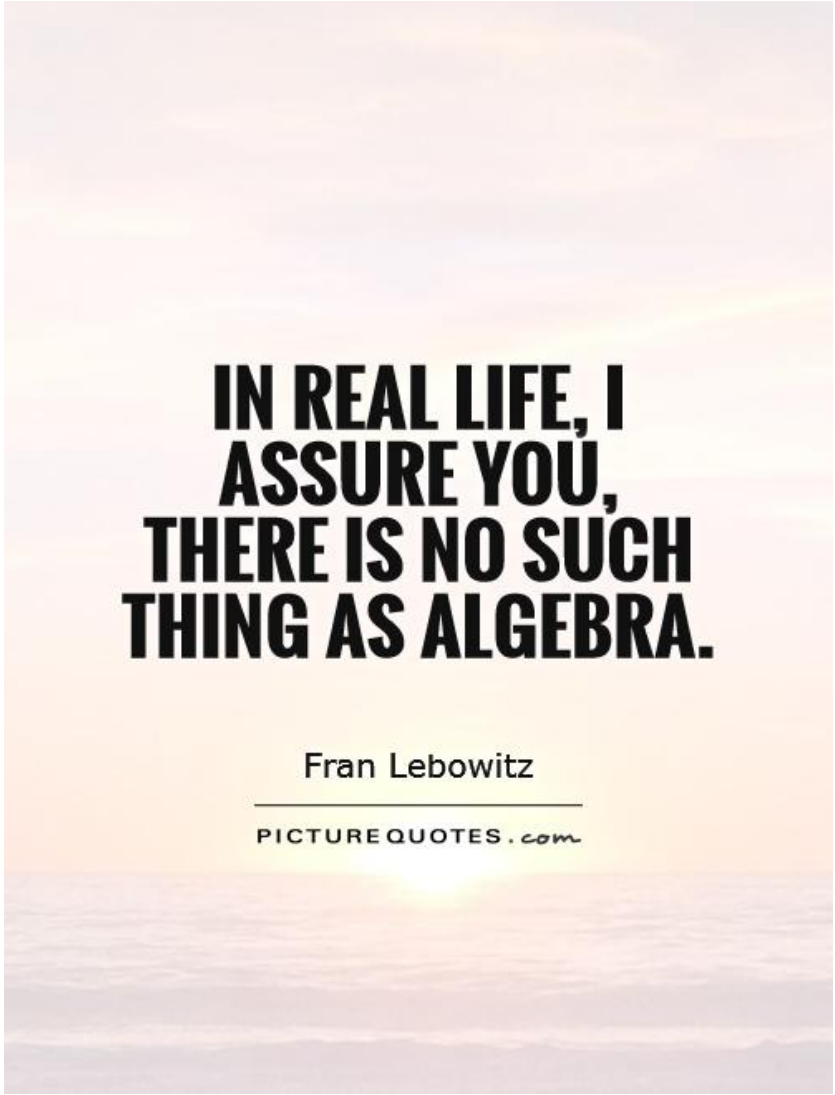
$$150 = 128 + 16 + 4 + 2 = 1001\ 0110_2$$

- Есть младшие (менее значимые) знаки/биты, есть старшие.
- На ближайших слайдах работаем внутри 1 байта (8 бит).
- Пока все промежуточные результаты вычислений от 0 до 255, нет никаких проблем — считаем и считаем.

Что делать, если произошло переполнение (overflow/underflow)?
Результат точно не сохраним.

1. Можно вызвать ошибку.
2. Можно откинуть младшие знаки.
3. Можно откинуть старшие знаки.

Если откинем младшие, то $255 + 1 - 1 \neq 255$, что неудобно, если мы хотим точные вычисления.


IN REAL LIFE, I ASSURE YOU, THERE IS NO SUCH THING AS ALGEBRA.
Fran Lebowitz
PICTUREQUOTES.com

А вот если считаем, что откидываем старшие, то получаем коммутативное кольцо с единицей $\mathbb{Z}/256\mathbb{Z}$.

- По сути — просто арифметика, где все числа берутся по модулю 256.
- Сложение, вычитание, умножение в таких объектах прекрасно определены и непротиворечивы.
- Можно делать что угодно, и мы всегда получим корректный результат по модулю 256.
- В железе реализовать просто — считаем только последние 8 бит результата.

Деление

С делением хуже (деление — обратное к умножению).
После взятия по модулю иногда можно однозначно восстановить ответ, а иногда нет (на алгебре расскажут, когда):

$$34/17 = 2$$
$$4386/17 = 258 = 2 \mod 256$$
$$48/4 = 12$$
$$\underbrace{(256 + 48)}_{304}/4 = 76$$

Поэтому деление всегда считает, что у нас числа помещаются в 8 бит, и делим мы с остатком.

- Можно сказать, что в первом бите храним знак, а в остальных — число, как раньше (*прямой код*).
- Тогда надо разбирать случаи в процессоре для всех арифметических и логических операций.
- Появляются $+0$ и -0, так что ещё и сравнение на равенство сильно менять.

А можно сказать, что, $-x$ по определению — это такое y, что $x + y = 0$. Тогда вспоминаем, что мы уже живём по модулю 256 (проверьте вычисления в Python самостоятельно!):

$$
\begin{aligned}
x &= x + 256 && \mod 256 \\
-56 &= -56 + 256 = 200 && \mod 256 \\
-56 + 56 &= -56 + 256 + 56 = 256 = 0 && \mod 256 \\
-17 \cdot 22 &= -374 = 138 && \mod 256 \\
-17 \cdot 22 &= (256 - 17) \cdot 22 = 239 \cdot 22 = 5258 = 138 && \mod 256
\end{aligned}
$$


North-South
Road
WESTFOLD
Emnet

Алгебра говорит, что $-x$ — это *обратный по сложению к* x. В кольцах он есть. Считаем, что обратный по сложению к x в кольце $\mathbb{Z}/256\mathbb{Z}$ и есть элемент $-x$.

Мы только поменяли, как мы интерпретируем числа, но не их битовую запись:

$$\mod 256$$

$$-56 + 100 = -56 + 256 + 100 = 200 + 100 =$$

$$= 1100\ 1000 + 0110\ 0100 = 1\ 0010\ 1100 = 0010\ 1100 = 44$$

Таким образом, сложение, вычитание, и даже умножение по-прежнему работают (спасибо алгебраистам, что доказали).

Упражнение: проверить, что:

$$a = b \mod 256 \to a \cdot x = b \cdot x \mod 256$$

С делением хуже:

$$-10/5 = (256 - 10)/5 = 246/5 = 49.5 = ???$$

Дополнительный код

1. Мы можем как угодно обозначить элементы кольца: какие-то назвать отрицательными числами, а какие-то — положительными.
2. Обычно разделяют отрезок ровно пополам: $[-128; 127]$.
3. Теперь по самому старшему биту можно определить знак: 1 — отрицательное, 0 — неотрицательное.

Такая конвенция называется *дополнительный код*: отрицательное и положительно число в сумме дают нули или дополняют до степени двойки 2^8.

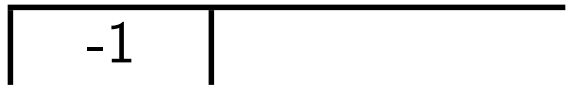
1. Надо разбирать случаи в сравнении чисел и в делении с остатком (поэтому они в ассемблере появляются знаковые/беззнаковые).
2. Есть операция смены знака: инвертировать все биты и добавить единицу (инвертация бит — это вычитание из $1111\ 1111_2 = 255_{10}$).

Упражнение

Как представлены следующие числа в дополнительном коде?

Упражнение

Как представлены следующие числа в дополнительном коде?



Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	0111 1111

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	0111 1111
128	

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	0111 1111
128	никак

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	0111 1111
128	никак

Что будет, если мы возьмём $-(-128)$?

Упражнение

Как представлены следующие числа в дополнительном коде?

-1	1111 1111
-128	1000 0000
127	0111 1111
128	никак

Что будет, если мы возьмём $-(-128)$?

$$-128 = 1000\ 0000$$

$$-(-128) = (\tilde{1}000\ 0000) + 1 = 0111\ 1111 + 1 = 1000\ 0000 = 128$$

Играем в игру

Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

Играем в игру

Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	

Играем в игру

Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	259

Играем в игру

Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	259
...	0000 0001	0000 0011	...	

Играем в игру

Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

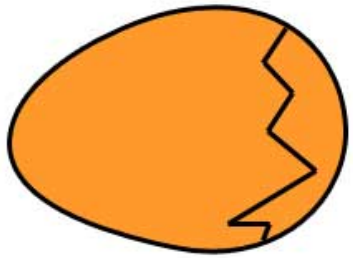
Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	259
...	0000 0001	0000 0011	...	769

Играем в игру

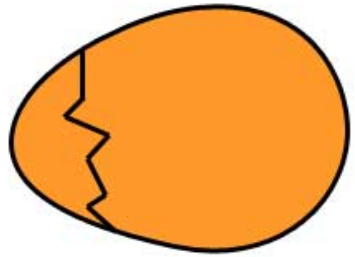
Напоминание: порядок бит в байте мы из программы никак не определим, на картинке рисуем слева старшие, справа младшие. Порядок байт в памяти — слева меньшие адреса, справа большие.

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	259
...	0000 0001	0000 0011	...	769

- В каком порядке идут байты в памяти? Они же тоже бывают старшие и младшие.
- Как договорились — так и идут. Договариваются по-разному на разных процессорах и в разных протоколах.
- Свойство «порядок байт» называется endianness.



BIG ENDIAN - The way people always broke their eggs in the Lilliput land



LITTLE ENDIAN - The way the king then ordered the people to break their eggs

Есть два клана: little-endian (остроконечники) и big-endian (тупоконечники). По-русски всегда используют английские термины.

Big-endian

Используется в низкоуровневых сетевых протоколах (TCP) и процессорах Atmel AVR (ATmega и прочие).
Младший байт имеет больший адрес. Читается просто:

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	$0000\ 00\mathbf{01}\ 0000\ 00\mathbf{11}_2 = 259_{10}$

Надо очень аккуратно помнить адрес и размер числа:

Адрес			Значение
	2		
...	0000 0001	...	$00000001_2 = 1_{10}$

Little-endian

Используется в x86: «младший байт имеет меньший адрес».
Читается хуже:

Адрес				Значение
	2	3		
...	0000 0001	0000 0011	...	$0000\ 00\mathbf{11}\ 0000\ 00\mathbf{01}_2 = 769_{10}$

Если только мы не Intel и не пишем к этому документацию:

Адрес				Значение
	3	**2**		
...	0000 0011	0000 0001	...	$0000\ 0011\ 0000\ 0001_2 = 769_{10}$

Они у себя всё пишут от старших к младшим: байты с меньшими адресами справа, младшие биты справа.

Особенности Little-endian-1

Можно почти безболезненно конвертировать между типами:

Адрес				Значение	
	2	3			
...	0000 0011	0000 0001	...	0000 0001 0000 0011_2	259_{10}
...	0000 0011	...	...	0000 0001_2	3_{10}
...	0000 0101	0000 0000	...	0000 0000 0000 0101_2	5_{10}
...	0000 0101	...	...	0000 0101_2	5_{10}
...	1111 0011	0000 0000	...	0000 0000 1111 0011_2	243_{10}
...	1111 0011	...	...	1111 0011_2	243_{10}
...	1111 0011	...	...	1111 0011_2	-13_{10}
...	1111 0011	1111 1111	...	1111 1111 1111 0011_2	-13_{10}

Особенности Little-endian-2

- Есть проблема со знаком, если мы переходим от меньшего типа к большему.
- Надо *расширять знак*, если у нас было знаковое число: заполнять старшие байты либо нулями, либо единицами (в зависимости от...).

Особенности Little-endian-2

- Есть проблема со знаком, если мы переходим от меньшего типа к большему.
- Надо *расширять знак*, если у нас было знаковое число: заполнять старшие байты либо нулями, либо единицами (в зависимости от старшего бита в числе).

Особенности Little-endian-2

- Есть проблема со знаком, если мы переходим от меньшего типа к большему.
- Надо *расширять знак*, если у нас было знаковое число: заполнять старшие байты либо нулями, либо единицами (в зависимости от старшего бита в числе).
- Если число было беззнаковое, то надо старший байт заполнить нулями.

Компиляторы низкоуровневых языков делают это автоматически, когда вы делаете какие-то присваивания. Разумеется, надо аккуратно следить за типами, иначе не сделают.
Замечание: иногда, несмотря на тип `int`, какие-то функции могут ожидать в нём на самом деле не число, а набор байт в определённом порядке. Яркий пример — номер порта в работе с сетью на C, функция `htons` — это оно.

Резюме

1. Целые числа хранятся в двоичной системе счисления.
2. Знаковость числа определяется лишь типом данных.
3. Самый распространённый способ кодирования отрицательных чисел — «дополнительный код».
4. В дополнительном коде старший бит отвечает за знак, а арифметика делатеся так же, как и в беззнаковых числах.
5. Операциям сравнения чисел на меньше/больше и делению важно знать, работаем ли мы в дополнительном коде или с беззнаковыми числами.
6. Порядок байт в числах может отличаться даже в разных местах внутри одного приложения. Читайте документацию, если работаете с чем-то на уровне байт!
7. Если работаете с little-endian и меняете количество байт в числе — позаботьтесь о знаке.
8. Если есть либо дополнительный код, либо переполнения, то важно количество бит/байт в числе.

Зачем нужны вещественные числа в компьютерах? Почему нельзя обойтись целыми?

Зачем нужны вещественные числа в компьютерах? Почему нельзя обойтись целыми?

- Работа с дробями и деление.
- Денежные единицы: копейки удобно считать сотыми частями рубля.
- Физические (и военные) вычисления: тригонометрия, расстояния, геокоординаты.

С фиксированной запятой

- Пример: рубли.
- После запятой всегда ровно два знака: 230.40.
- По сути те же целые числа, только надо помнить, где стоит запятая, и перемножаются чуть по-другому (со сдвигом запятой и, соответственно, обрезанием знаков).

Есть в некоторых базах данных, используются как раз для хранения количества денег.
Плюсы:

- Абсолютная точность, кроме, иногда, умножения и деления.
- Простые и предсказуемые операции.
- Простое описание допустимых значений.

Минусы

- Надо заранее знать, сколько знаков потребуется.
- Если может требоваться разное число знаков в разных местах — надо брать разные типы данных и конвертировать.
- Соответственно, нужен разный код, если есть вычисления и с «большими» числами, и с «маленькими».
- Не реализовать аппаратно, потому что неясно, сколько знаков отбрасывать при умножении; реализовывать много типов сложно.

С плавающей запятой

- Обобщение числа с фиксированной запятой.
- Храним отдельно число и отдельно — сколько у нас знаков идёт после запятой, а сколько — до.
- Умножение и деление остались примерно такими же по сложности, а вот сложение и вычитание усложнились (надо сравнивать порядок чисел).
- Теперь можно делать вычисления с числами любого порядка, и будем знать порядок ответа и первые сколько-то цифр.

Плюсы:

- Одинаково хорошо работаем и с маленькими, и с большими числами.
- Относительная погрешность вычислений сохраняется.

Первая попытка

Пусть храним числа так:

$$x = a \cdot 10^b$$

$$-32768 \leq a, b \leq 32767$$

Тут a — *мантисса*, b — *экспонента*.
Например:

$$12.3 = 123 \cdot 10^{-1}$$

$$231000 = 231 \cdot 10^3$$

$$12.3 \cdot 231000 = 123 \cdot 10^{-1} \cdot 231 \cdot 10^3 = (123 \cdot 231) \cdot 10^{-1+3} = 28413 \cdot 10^2$$

$$12.3 + 231000 = 123 \cdot 10^{-1} + 231 \cdot 10^3 = (123 + \underbrace{2310000}_{>>32767}) \cdot 10^{-1}$$

Трудности с десятичной системой

- При сложении у нас легко может произойти переполнение типа.
- Можно пытаться оставлять только самые значащие цифры.
- Но в общем случае надо домножать на большую степень десятки.
- Как это делать без очень больших чисел и без домножения на десятку каждый раз — неясно.
- Храним-то всё в двоичной системе, а там от умножения на десять меняется всё число.

Решение проблемы

Храним числа так:

$$x = a \cdot \mathbf{2}^b$$

$$-32768 \leq a, b \leq 32767$$

- Теперь стало легко и складывать, и перемножать, так как при домножении на двойку легко понять, сколько цифр не нужны.
- Можно реализовать в железе.
- Проблемы с тем, что нет чёткого соответствия между десятичными знаками после запятой и двоичными, нет точности:

$$0.75_{10} = 0.11_2$$

$$7 \cdot 10^{-1} + 5 \cdot 10^{-2} = 2^{-1} + 2^{-2}$$

$$0.1_{10} = 0.0001100110011001100110011001101\ldots_2$$

$$10^{-1} = 2^{-4} + 2^{-5} + 2^{-7} + 2^{-8} + \ldots$$

Упражнение

- Введите в Python:

```
x=0.1
print(x)
print(format(x, ".1000f"))
```

- Убедитесь, что вы не получили `0.1`
- Найдите, на какую степень двойки надо домножить x, чтобы он стал целым. По сути — грубое приближение числа знаков в мантиссе.

Упражнение

- Введите в Python:

```
x=0.1
print(x)
print(format(x, ".1000f"))
```

- Убедитесь, что вы не получили `0.1`
- Найдите, на какую степень двойки надо домножить x, чтобы он стал целым. По сути — грубое приближение числа знаков в мантиссе.

Ответ: на 2^{55}. Из этого можно сделать вывод, что в мантиссе порядка 55 знаков.

Что такое IEEE 754

- IEEE 754 — это стандарт хранения и обработки вещественных чисел с плавающей запятой, который используется почти везде.
- Но не вообще везде.
- Определяет несколько типов данных с разными размерами мантисс и экспонент: single precision, double precision, и ещё несколько.
- Достаточно разумно определяет операции для всех аргументов и решает некоторые проблемы.
- Из-за этого сложнее, чем идея с предыдущих слайдов.
- Есть $\pm\infty$, есть `NaN` (Not a Number, получается при делении нуля на ноль).
- В Python используется double-precision (тип `float`).
- В C++/Java есть как single-precision (`float`), так и double-precision (`double`).

Формат single-precision

Основная масса — *нормализованные числа*:

Биты		
31	30-23	22-0
знак (s)	экспонента (e)	мантисса (m)

1. Если $s = 0$, то число положительное, иначе отрицательное.
2. Предполагается, что экспонента подобрана так, чтобы перед запятой был ровно один знак — единица.
3. Мантисса хранит все знаки *строго после* этой единицы.
4. Если $e = 0$, то у нас число вида 1.1001010010 в двоичной записи (до запятой — ровно одна единица).
5. Итоговая формула (для *нормализованных* чисел):

$$x = (-1)^s \cdot 2^e \cdot (1 + m \cdot 2^{-23})$$

Хранение экспоненты

Экспонента хранится как беззнаковое число со сдвигом на 127:

Двоичное представление	e
0000 0000	-127
0000 0001	-126
⋮	⋮
0111 1110	-1
0111 1111	0
1000 0000	1
⋮	⋮
1111 1110	127
1111 1111	128

Обратите внимание, что отрезок чисел — от -127 до 128, а не от -128 до 127 (как в дополнительном коде).

Пример

$$x = -13.75_{10} = \frac{-220}{16} = -1101.1100_2$$

Подбираем экспоненту так, чтобы слева получилась ровно одна единица:

$$x = -1.\underbrace{1011100_2}_{m'=184_{10}} \cdot 2^3 = \quad e = 3$$

$$= -1.\underbrace{10111000000000000000000_2}_{m} \cdot 2^3$$

Так как в m предполагается 23 значащих знака (а у нас в m' только 7), надо дописать *справа* нулей. Итого:

Биты

31	30-23	22-0
1	$\underbrace{100\ 0001\ 0}_{e+127}$	$\underbrace{\mathbf{101\ 110}0\ 0000\ 0000\ 0000\ 0000}_{m}$

Проблемы с аксиомами

Какое наименьшее нормализованное число можно представить в таком формате? Очевидно, при минимальной экспоненте (-127) и мантиссе. Тогда самые маленькие числа таковы:

$$x = 2^{-127} \cdot 1$$
$$y = 2^{-127} \cdot (1 + 1 \cdot 2^{-23})$$
$$z = 2^{-127} \cdot (1 + 2 \cdot 2^{-23})$$

посчитаем что-нибудь:

$$y - x = 2^{-127} \cdot (1 + 1 \cdot 2^{-23} - 1) = 2^{-127} \cdot 1 \cdot 2^{-23} = 2^{-150}$$

Вопрос: как хранить 2^{-150}?

- Округлять к 2^{-127} странно: это далеко; тогда бы получили, что $x - y = x$, но $x \neq x + y$.
- Округлять к нулю тоже странно: $x \neq y$, но $x - y = 0$.

Денормализованные числа

- Вблизи нуля добавили *денормализованные числа*, чтобы повысить точность и избежать подобных проблем с аксиомами.
- Теперь стандарт гарантирует, что $x - y = 0 \iff x = y$.
- Денормализованное число — это число с минимально возможной экспонентой, у которого отсутствует единица перед запятой:

Нормализованные	$\mathbf{1}.\overbrace{0010011}^{\text{m}} \cdot 2^{e=4}$
Денормализованные	$\mathbf{0}.\underbrace{10010011}_{\text{m}} \cdot 2^{e=5}$

- Это просто 2^{23} чисел, равномерно распределённых от 0 до минимального нормализованного.
- Всё равно можно получить underflow:

```
x=2**(-1074)
print(x, x / 2 * 2)
```

Формат

Скажем, что если экспонента состоит из нулей, то у нас денормализованное число:

Биты		
31	30-23	22-0
знак (s)	нули	мантисса m

Тут мы уже считаем, что мантисса записана целиком, включая старшую единицу. Формула:

$$x = (-1)^s \cdot 2^{-126} \cdot m \cdot 2^{-23}$$

Тогда денормализованные числа лежат в диапазоне:

$$2^{-126} \cdot 2^{-23} \leq x \leq 2^{-126} \cdot (2^{23} - 1) \cdot 2^{-23}$$

А нормализованные — в таком:

$$2^{-126} \cdot 1 \leq x$$


EXCUSE ME WTF R U DOIN

Все особенности IEEE-754

Числа, доступные в IEEE 754 одинарной точности:

Биты			
31	30-23	22-0	Значение
0	111 1111 1	000 0000 0000 0000 0000 0000	$+\infty$
1	111 1111 1	000 0000 0000 0000 0000 0000	$-\infty$
?	111 1111 1	не нули	`NaN`
0	000 0000 0	000 0000 0000 0000 0000 0000	$+0$
1	000 0000 0	000 0000 0000 0000 0000 0000	-0
?	000 0000 0	не нули	д. число
?	что угодно	что угодно	н. число

Упражнение

1. Введите несколько простых десятичных дробей в Python и попробуйте методы `float.as_integer_ratio()` и `float.hex()`.
2. Составьте таблицу сравнения для 0, +0, -0 на Python: кто кому равен, кто кого меньше.
3. Добавьте в эту таблицу сравнения `NaN`.
4. На C++/Python составьте таблицу уможнения для ± 0, ± 1, $\pm \infty$ и `NaN`.
5. На C++ составьте таблицу деления для этих же чисел (на Python некоторые операции вызывают исключение

Хранятся только старшие знаки

```
for a in [2.0 ** x for x in [10, 20, 40, 60, 1000]]:
    print(a == a + 1, a == a * (1 + 2 ** (-40)), a + 1 - a)
```

В Python у нас тип двойной точности, поэтому до 2^{40} единица ещё будет играть роль, а вот после — нет.

Складывать/вычитать числа разных порядков — плохо. Сильно теряется точность.

```
print(1000 + 1e100 - 1e100)
print(1e100 + 1000 - 1000)
```

Одного порядка — нормально.

Ошибка может копиться

```
from random import shuffle
a=[123456 * 2 ** x for x in range(0, 200)]
print(format(sum(a), ".20e"))  # Точный результат
a = list(map(float, a))
for _ in range(10):
    shuffle(a)
    print(format(sum(a), ".20e"))
```

- Так как результаты округляются на каждом шаге, результат может зависеть от порядка вычислений, размеров чисел и фазы луны.
- Пример, где всё может быть плохо — метод Гаусса решения систем линейных уравнений (подробнее расскажут на алгоритмах).
- Уже для системы 15×15 можно подобрать матрицу, на которой точность теряется катастрофически быстро.

Оптимизатор

Оптимизатор

- Часто процессор поддерживает числа большего размера, чем double-precision.
- Тогда точность может в некоторых местах случайно возрастать, а оптимизатор об этом не догадывается.
- Из-за этого «побитово равные» числа могут оказаться различными: в одном месте сравнили куски памяти, а в других местах — лежащие в процессоре числа большей точности.
- Аналогично, можно случайно (не) увидеть какую-то маленькую погрешность.

Десятичные дроби неточны

```
print(0.1 + 0.2)  # Классика
print(format(0.1 + 0.2, ".100f"))
print(format((0.1 + 0.2) * 2 ** 52, ".100f"))
for n in [1000, 1024]:
    fail, total = 0, 0
    for a in range(n):
        for b in range(n):
            if (a + b) / n != a / n + b / n:
                fail += 1
            total += 1
    print(fail, total, fail / total)
```

Что делать?

- Не использовать вещественные числа.
- Особенно на контестах.
- Особенно у Серёжи.
- Если очень надо — готовиться к неточности и допускать погрешность при *любых* сравнениях (см. оптимизатор):

Было	Стало
`a == b`	`abs(a - b) < eps`
`a <= b`	`a <= b + eps`
`a < b`	`a < b - eps`

  Тут мы считаем, что числа равны, если отличаются не более, чем на `eps`.
- Следить за порядком операций и избегать сложения/вычитания чисел разного порядка.
- Иногда можно преобразовывать формулы: $x^2 - y^2 = (x - y)(x + y)$.

Замечания про сравнения

- Можно сравнивать числа ещё аккуратнее, если смотреть на относительную погрешность, а не абсолютную. Или если надо сравнивать в программе и большие числа, и маленькие.
- Подбор `eps` и правильного порядка операций — отдельное искусство.
- Обычно на контестах берут `eps` от 10^{-5} до 10^{-15}; мой любимый — 10^{-8}.
- При желании может строго теоретически обосновать правильный `eps`.

Замечания про сравнения

- Можно сравнивать числа ещё аккуратнее, если смотреть на относительную погрешность, а не абсолютную. Или если надо сравнивать в программе и большие числа, и маленькие.
- Подбор `eps` и правильного порядка операций — отдельное искусство.
- Обычно на контестах берут `eps` от 10^{-5} до 10^{-15}; мой любимый — 10^{-8}.
- При желании может строго теоретически обосновать правильный `eps`.
- Да и неправильный тоже :(

Странные вычисления

Осторожно с NaN: любое вычисление с ним немедленно породит NaN, а сравнения с ним всегда выдают false:

```
from math import sqrt
print(2 ** 1000 / 2 ** (-1000))
print(-2 ** 1000 / 2 ** (-1000))
print(sqrt(0), sqrt(2 ** (-1000)))
print(0 * float('inf'))
print(1 + float('nan'))
print(float('nan') == float('nan'))
```

В C++ его ещё можно получить, взяв корень из отрицательного числа или $\frac{0}{0}$ (Python может кинуть исключение).
Лайфхак: когда извлекаете корень, не извлеките его случайно из -eps. Для этого стоит писать sqrt(max(x, 0)), если x может быть близок к нулю.

Ручное округление

Самые популярные:

x	floor(x)	round(x)	ceil(x)	int(x)
-1.6	-2	-2	-1	-1
-1.5	-2	-2	-1	-1
-1.4	-2	-1	-1	-1
-0.6	-1	-1	0	0
-0.5	-1	0	0	0
-0.4	-1	0	0	0
0.4	0	0	1	0
0.5	0	0	1	0
0.6	0	1	1	0
1.4	1	1	2	1
1.5	1	2	2	1
1.6	1	2	2	1

Последние замечания

- Стандарт требует, что результат любой операции должен быть максимально точен, а округление должно быть до ближайшего числа.
- При этом округлять $\frac{1}{2}$ всегда вверх нехорошо — ошибка будет сильно накапливаться.
- Поэтому округляют до ближайшего чётного («банковское» округление).
- Округление не знает про погрешность вычислений:

```
from math import ceil
print(ceil(1))                    # 1
print(ceil(1.00000000000001))     # 2
```

  Поэтому на практике, если могло получиться число, близкое к целому, лучше использовать `round` или добавлять `eps`.
- Не используйте типы с плавающей точкой для хранения количества денег или чего-то подобного, что требует точности при рассчётах.

1. Презентация Ивана Казменко (кружок при СПбГУ).
2. Статья на Хабре.
3. Пересказ стандарта IEEE 754.
4. Онлайн-конвертер для single precision.
5. Статья на английском.
6. Документация Python на английском.