Аппаратное обеспечение вычислительных систем

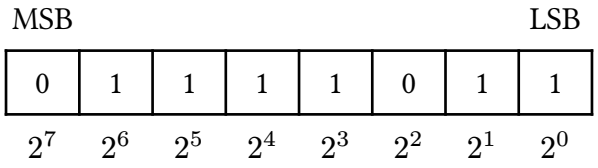
Содержание

itmo.arslee.me

Кодирование беззнаковых целых чисел

Компьютер оперирует *байтами* — минимальными адресуемыми ячейками памяти. Практически у всех современных компьютеров 1 байт равен 8 бит, хотя раньше существовали архитектуры, где 1 байт был равен, например, 7 бит.

Рассмотрим схему:



Каждая ячейка соответсвует биту. У каждого бита есть свой «вес», равный показателю степени двойки — основания системы счисления компьютера.

Поскольку машина оперирует байтами, то нельзя сказать что-то вроде «первый бит» или «последний бит». Для компьютера это не имеет значения. Так что принято говорить «младший бит» (LSB, Least Significant Bit) или «старший бит» (MSB, Most Significant Bit).

Декодируем число из схемы. Каждый бит ячейки умножается на его вес, а затем все складывается. В итоге имеем $2^6 + 2^5 + 2^4 + 2^3 + 2^1 + 2^0 = 123$.

Диапазон значений равен $[0;\ \ 2^n - 1]$. Например, для 8 бит это $[0; 255]$.

В языке Си гарантируется модулярная арифметика для беззнаковых чисел. Это означает, что выражение вида `uint8_t a = 254 + 7` будет равно `5`, то есть возьмется по модулю `256`.

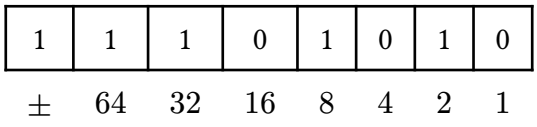
Кодирование знаковых целых чисел

Машина не знает про знак числа, находящегося в памяти. Представление знакового числа — забота самого программиста.

Прямой код

Самый очевидный способ закодировать число — выделить старший бит под знак. Причем 0 обозначает «+», а 1 обозначает «-», чтобы можно было легко преобразовать беззнаковые числа в знаковые.

На примере числа −106:



Имеем, что беззнаковое число 234 в прямом коде интерпретируется как −106.

Диапазон равен $\left[-2^{n-1} + 1;\ \ 2^{n-1} - 1\right]$. Например, для 8-битного числа это $[-127; 127]$

У прямого кода есть проблемы:
1. Существования числа -0.
2. Трудности с выполнением арифметических операций.

3. Нет зацикливания. Хотим, чтобы $127 + 1$ было равно -127, но вместо этого получим -0.

Обратный код

В зарубежных источниках именуется как «дополнение до 1» (1's complement).

Идея в том, чтобы старший бит интерпретировать как число, равное сумме всех младших весов, взятое со знаком «минус». То есть, для 8-битного числа старший бит будет интерпретироваться как -127.

На примере числа -34:

1	1	0	1	1	1	0	1
-127	64	32	16	8	4	2	1

Имеем, что беззнаковое число 221 в обратном коде интерпретируется как −34.

Данное кодирование называется «обратным кодом», поскольку для взятия обратного числа нужно просто инвертировать все биты.

Диапазон равен $\left[-2^{n-1}+1;\ 2^{n-1}-1\right]$. Например, для 8-битного числа это $[-127; 127]$.

Плюсы обратного кода:
- Симметричный диапазон.
- Имеется зацикливание! Для 8-битного числа, $127 + 1 = -127$.
- Сумма/разность двух чисел тривиальна и аналогична беззнаковым числам.

Проблемой остается число -0.

Дополнительный код

В зарубежных источниках именуется как «дополнение до 2» (2's complement).

Идея схожа с обратным кодом, но старший бит теперь равен сумме всех младших весов *плюс единица*, взятого со знаком «минус». Например, для 8-битного числа старший бит интерпретируется как -128.

На примере числа -107:

1	0	0	1	0	1	0	1
-128	64	32	16	8	4	2	1

Беззнаковое число 181 интерпретируется в дополнительном коде как -107.

Взятие отрицания работает в два шага:
1. Инверсия всех битов
2. Добавление единицы с отбрасыванием последнего переноса

Диапазон равен $\left[-2^{n-1};\ 2^{n-1}-1\right]$. Для

Дополнительный код имеет следующие плюсы:
- Имеется зацикливание.
- Числа -0 нет!

- Сложение и вычитание чисел тривиально.
  - Для умножения чисел в дополнительном коде существует алгоритм Бута.

Единственный минус — несимметричный диапазон. Если взять отрицание от 8-битного числа -128, то получится -128.

Практически все современные системы представляют знаковые числа в дополнительном коде.

Всякие экзотические коды

«Чередующийся» код
Идея в том, чтобы выделить *младший* бит под знак:

$$000_2 \mapsto 0 \qquad 001_2 \mapsto -1 \qquad 010_2 \mapsto 1 \qquad 011_2 \mapsto -2 \qquad \dots$$

Используется для кодирования чисел произвольной длины. Арифметика над данным кодом невозможна.

Основание −2
Вес каждого бита — это степень числа -2.

На примере числа -107:

1	0	0	1	0	1	0	1
$(-2)^7$	$(-2)^6$	$(-2)^5$	$(-2)^4$	$(-2)^3$	$(-2)^2$	$(-2)^1$	$(-2)^0$

Диапазон в случае 8 бит равен $[-170; 85]$. Число 0 кодируется однозначно!

Немного про Си, UB, оптимизации и всякое такое

UB, Undefined Behavior — «неопределенное поведение».

Когда делали стандарт Си, основной упор был на переносимость языка между различными архитектурами.

Стандарт ISO C не диктует формат хранения знаковых чисел. Более того, диапазоны знаковых типов симметричны, что позволяет использовать хоть прямой код. Например, для `int16_t` диапазон равен $[-32767; 32767]$. То есть число -32768 (в допкоде) — это UB!

Более того, в стандарте не написано, что знаковые числа должны зацикливаться. Так что `INT_MAX + 123` — тоже UB.

Но все современное железо использует и допкод, и модулярную арифметику. Почему бы просто не игнорировать UB? Проблема в том, что UB для компилятора означает что-то вроде «могу сделать что захочу», из-за чего может прийти к некорректным оптимизациям. Например, заменить цикл `for (int8_t i = 1; i > 0; i++)` на `while (1)`.

Но несколько лет назад комитет по стандартизации Си все же решили признать современные реалии, а потому закрепили в стандарте C23 и дополнительный код, и модулярную арифметику для знаковых чисел.

Числа с фиксированной точкой

Допустим, мы хотим записать 12.34 рубля, но в нашем распоряжении только целые числа. Не беда, будем просто хранить 1234 копейки.

Аналогичный пример из повседневной жизни — проценты. Например, число 45.67 можно записать в процентах как 4567. То есть, целое число — это множитель к дроби $\frac{1}{100}$.

Идея чисел с фиксированной точкой такая же: целое число интерпретируется как множитель к некоторой дроби.

Возьмем в качестве примера все те же 8 бит, но сделаем веса немного по-другому:

0	1	1	1	1	0	1	1
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}

Старшие 4 бита теперь означают целую часть, а младшие 4 бита — дробную. В итоге мы закодировали число 7.6875.

В общем случае, число с фиксированной точкой имеет вид $n' = n \cdot 2^{-b}$, где:
- n — целое число
- b — число бит под дробную часть

Действительно, для нашего примера:
- $n = 123$
- $b = 4$
- $n' = 123 \cdot 2^{-4} = 7.6875$.

Ничто не мешает выделить, например, 7 бит под целую часть и 1 бит под дробную, или наоборот. Более того, можно сделать число знаковым с помощью любого удобного кода (в т.ч. де-факто стандартный дополнительный код).

Для обозначения чисел с фиксированной точкой принято обозначение `QA.B`, где:
- `A` — количество бит под целую часть
- `B` — количество бит под дробную часть

Для примера выше, формат обозначается как `Q4.4`.

В данном документе мы отбросим букву `Q` и будем просто обозначать число как `A.B`.

Перевод десятичной дроби в двоичную в столбик

Алгоритм:
1. В правом столбце запишем дробную часть
2. Умножаем дробную часть на 2.
3. В левый столбец записываем целую часть, в правый столбец записываем дробную часть.
4. Повторяем процесс с шага 2 нужное количество раз
5. Выписываем биты из левого столбца сверху вниз

На примере 0.675:

	625

1	25
0	5
1	0

Итак, $0.675 = 0.101_2$.

Алгоритм был таков:
1. $0.625 \cdot 2 = 1.25$. Пишем 1 справа, 25 слева.
2. $0.25 \cdot 2 = 0.5$. Пишем 0 справа, 5 слева.
3. $0.5 \cdot 2 = 1$. Пишем 1 справа, 0 слева.
4. Выписываем сверху вниз правый столбец: 101. Это и есть двоичная запись десятичной дроби.

Периодическая двоичная дробь

Попробуем перевести число 0.1 в двоичное в столбик:

	1
0	2
0	4
0	8
1	6
1	2
0	4
0	8
1	6
1	2
...	...

В итоге получаем периодическую дробь: $0.1 = 0.0(0011)_2$.

Это говорит о том, что не всякое десятичное число можно закодировать как число с фиксированной точкой без потери точности.

Немного про побитовые сдвиги в Си

В Си существует две различные по поведению операции побитового сдвига: логический и арифметический.

Логический сдвиг

Определен для беззнаковых целых чисел.

- **Сдвиг вправо**. Переносим все биты на n разрядов вправо. При этом младшие n бит пропадают. Заметим, что это эквивалентно делению на 2^n с округлением к нулю. Пример:

```
(0b1101 >> 2) == 0b11
```

- **Сдвиг влево**. Переносим все биты на n разрядов влево. «Освободившиеся» младшие n бит заполняем нулями. Заметим, что это эквивалентно умножению на 2^n. Пример:

```
(0b1101 << 3) == 0b1101000
```

Арифметический сдвиг

Определен для знаковых целых чисел.

- **Сдвиг вправо**. Сдвигаем биты на `n` разрядов вправо, «освободившиеся» места заполняем старшим битом исходного числа. Это эквивалентно делению на 2^n, но *с округлением к* $-\infty$. Пример:

```
-3 << 2 == -1
```

- **Сдвиг влево**. Эквивалентно логическому сдвигу влево.

Арифметические операции

Пусть нам даны два числа с фиксированной точкой в одинаковом формате:

```
uint32_t a = ...;  // 16.16
uint32_t b = ...;  // 16.16
```

Сложение и вычитание

Просто берем и складываем/вычитаем два числа. Модулярная арифметика работает без проблем.

```
uint32_t sum = a + b;
uint32_t diff = a - b;
```

Умножение

Здесь сложнее. Вспомним аналогию с процентами. Умножим 40% на 50%. Очевидно, что 2000% тут не подойдет, а значит нам нужно «сдвинуть» число, поделив на 100 и получив в итоге 20%.

Распишем умножение алгебраически:

$$a' \cdot b' = (a \cdot 2^{-16}) \cdot (b \cdot 2^{-16}) = (a \cdot b) \cdot 2^{-32}$$

Как можно видеть, теперь у нас 32 бита под дробную часть! Надо привести обратно к 16, сдвинув число на 16 бит вправо. Еще надо учесть, что произведение может не поместиться в текущий тип, так что надо расширять размерность.

Итак, учитывая все вышеперечисленное, код выглядит так:

```
uint64_t mul = ((uint64_t)a * b) >> 16;
```

Стоит понимать, что, отбросив дробных 16 бит, мы потеряли точность, так что нужно производить округление.

Деление

Снова аналогия с процентами. Очевидно, что $40\% \div 50\% \neq 0.8\%$. Поэтому предварительно умножим делимое на 100, что и даст нам корректные 80%.

Посмотрим на алгебраическое представление:

$$\frac{a'}{b'} = \frac{a \cdot 2^{-16}}{b \cdot 2^{-16}} = \frac{a}{b}$$

Под дробную часть теперь 0 бит! Так быть не должно. Предварительно сдвинем a' влево на 16 бит:

$$\frac{a' \cdot 2^{-16}}{b'} = \frac{a \cdot 2^{-32}}{b \cdot 2^{-16}} = \frac{a}{b} \cdot 2^{-16}$$

Теперь то, что надо.

В коде деление выглядит так:

```
uint32_t div = ((uint64_t)a << 16) / b;
```

Деление происходит нацело, отчего теряется точность. Надо делать округление.

Округления

При умножении или делении мы вынуждены округлять наши результаты. Рассмотрим основные способы.

К нулю

По сути отбрасывание разрядов. На числовой прямой, округленное число становится ближе к нулю. Примеры:

$$2.6 \to 2 \qquad -3.4 \to -3$$

К $-\infty$

Если число положительное, отбрасываем разряды. Если отрицательное, то округляем к младшему числу. На числовой прямой, числа сдвигаются влево, к $-\infty$. Примеры:

$$3.8 \to 3 \qquad -3.1 \to -4$$

К $+\infty$

Если число отрицательное, отбрасываем разряды. Если положительньое, то округляем к старшему числу. На числовой прямой, числа сдвигаются вправо, к $+\infty$. Примеры:

$$3.1 \to 3 \qquad -2.9 \to -2$$

К ближайшему четному

Если число ближе к младшему целому, округляем к нему. Если число ближе к старшему целому, округляем к нему. Если число находится ровно «посередине», округляем к тому целому числу, что четное. Примеры:

$$1.2 \to 1 \qquad 2.6 \to 3$$
$$-5.3 \to -5 \qquad -7.7 \to -8$$

$$3.5 \to 4 \qquad 2.5 \to 2$$
$$-4.5 \to -4 \qquad -7.5 \to -8$$