Lec 1

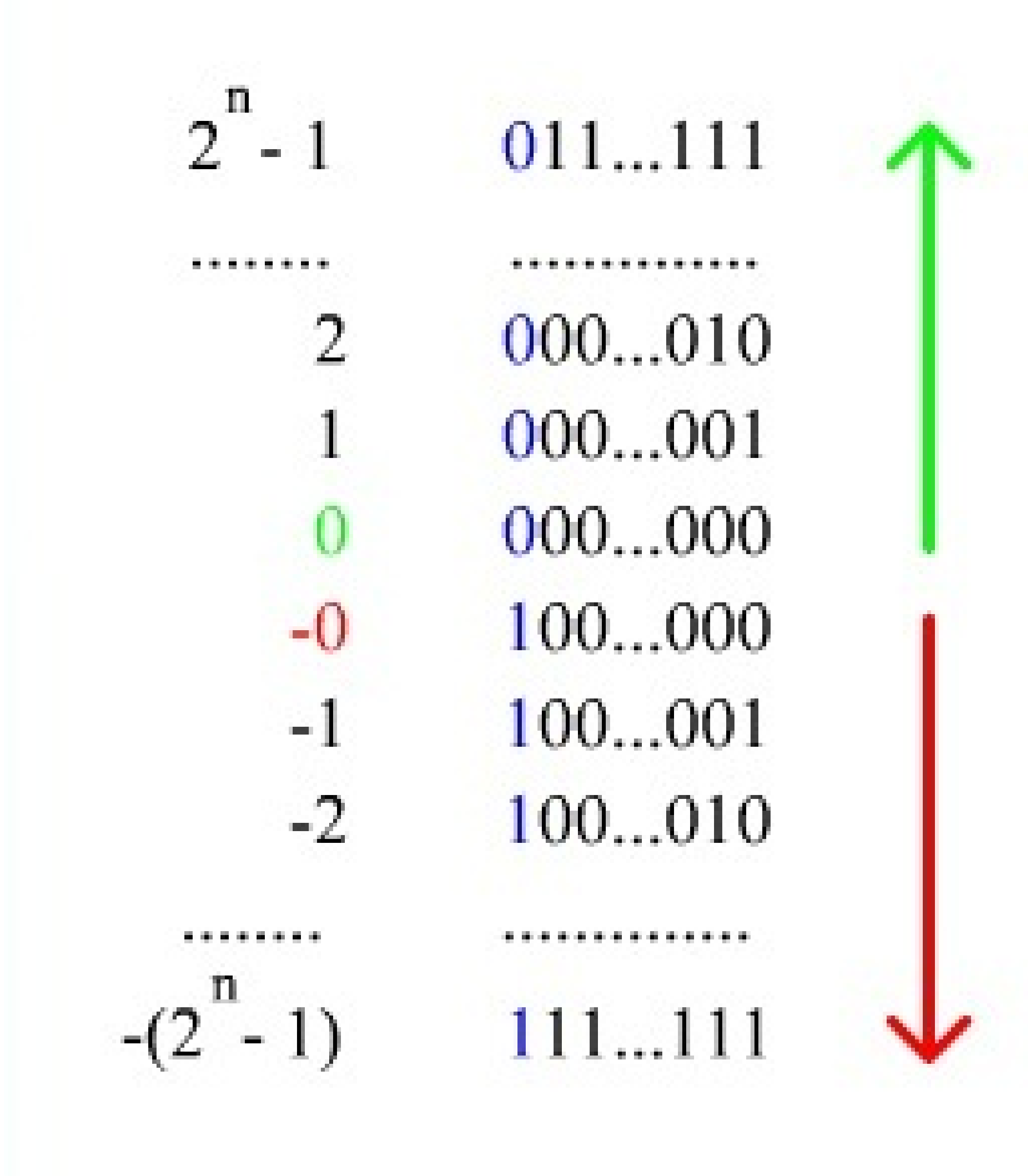
Кодирование целых чисел

байт - минимальная выделяемая область памяти (нигде не сказано, что ровно 8 бит)

Бит под знак (Прямой код)

Положительные числа кодируются с использованием разряда, умножения и последующим сложением (обычное двоичное беззнаковое число)

Для отрицательных чисел можно хранит старший бит под знак. (0 - плюс, 1 - минус; так как представление положительных чисел совпадает с без знаковой формой)



<u>Диапазон:</u> $[-2^{n-1} \dots 2^{n-1}]$
<u>Сравнение:</u> побитовое ИЛИ `+0` и `-0`
<u>Как представить отрицательное:</u> Сделать в двоичном виде положительное и заменить старший бит на `1`

🟩 Пример

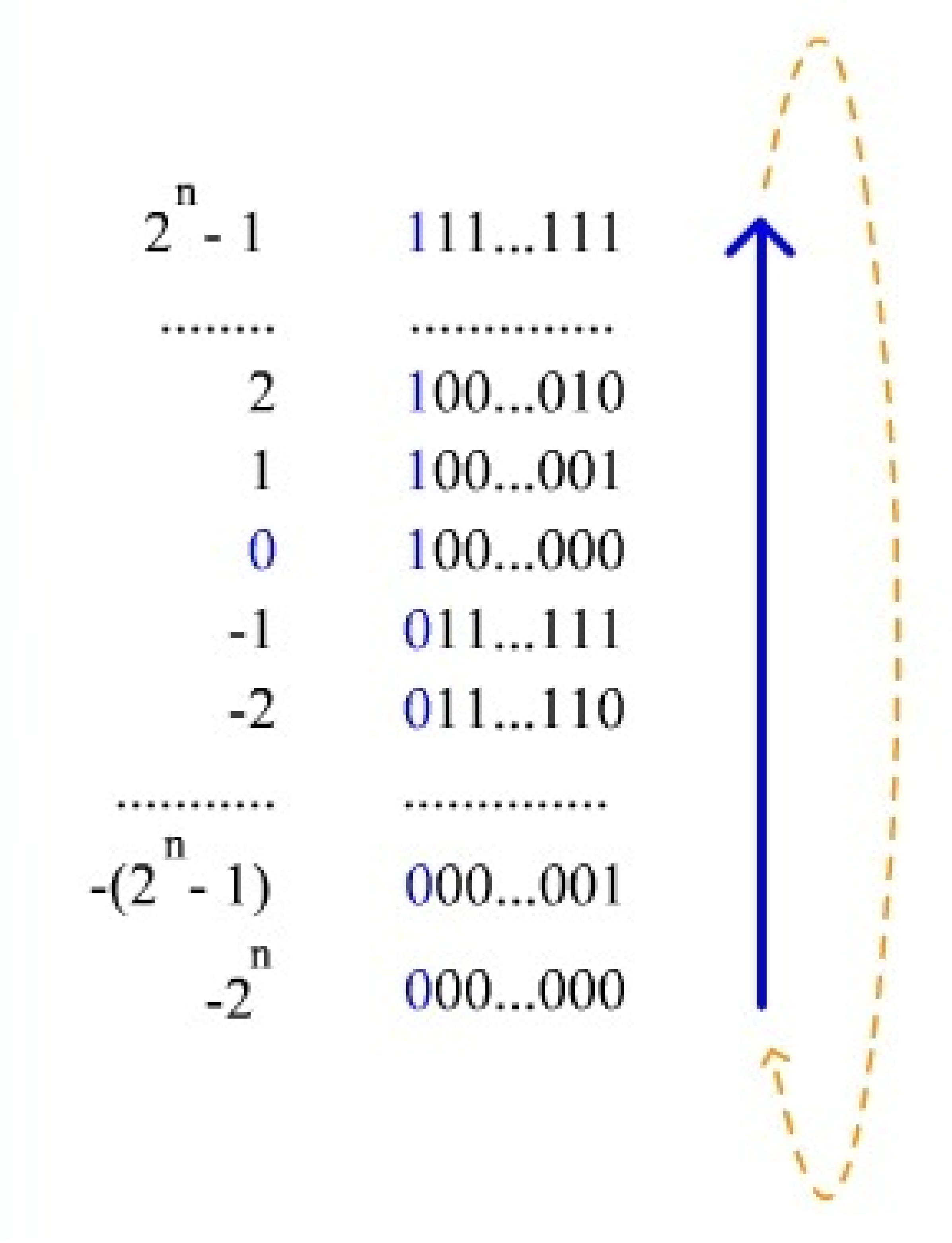
5: 0b00000101
-5: 0b10000101

🟧 Особенности:

➕ Кол-во положительных = Кол-ву отрицательных
➕ Два нуля (с минусом и плюсом)
➖ Усложненное арифметическое сравнение из-за двух нулей
➖ Сложное выполнение арифметических операций

Код со сдвигом

Еще вариант: взять беззнаковое представление и сдвинуть границу наполовину вправо



Диапазон: $[-2^{n-1} \dots 2^n - 1]$
Как представлять число: К искомому числу прибавить 2^{n-1} (n - кол-во бит) и перевести в двоичную

🟩 Пример

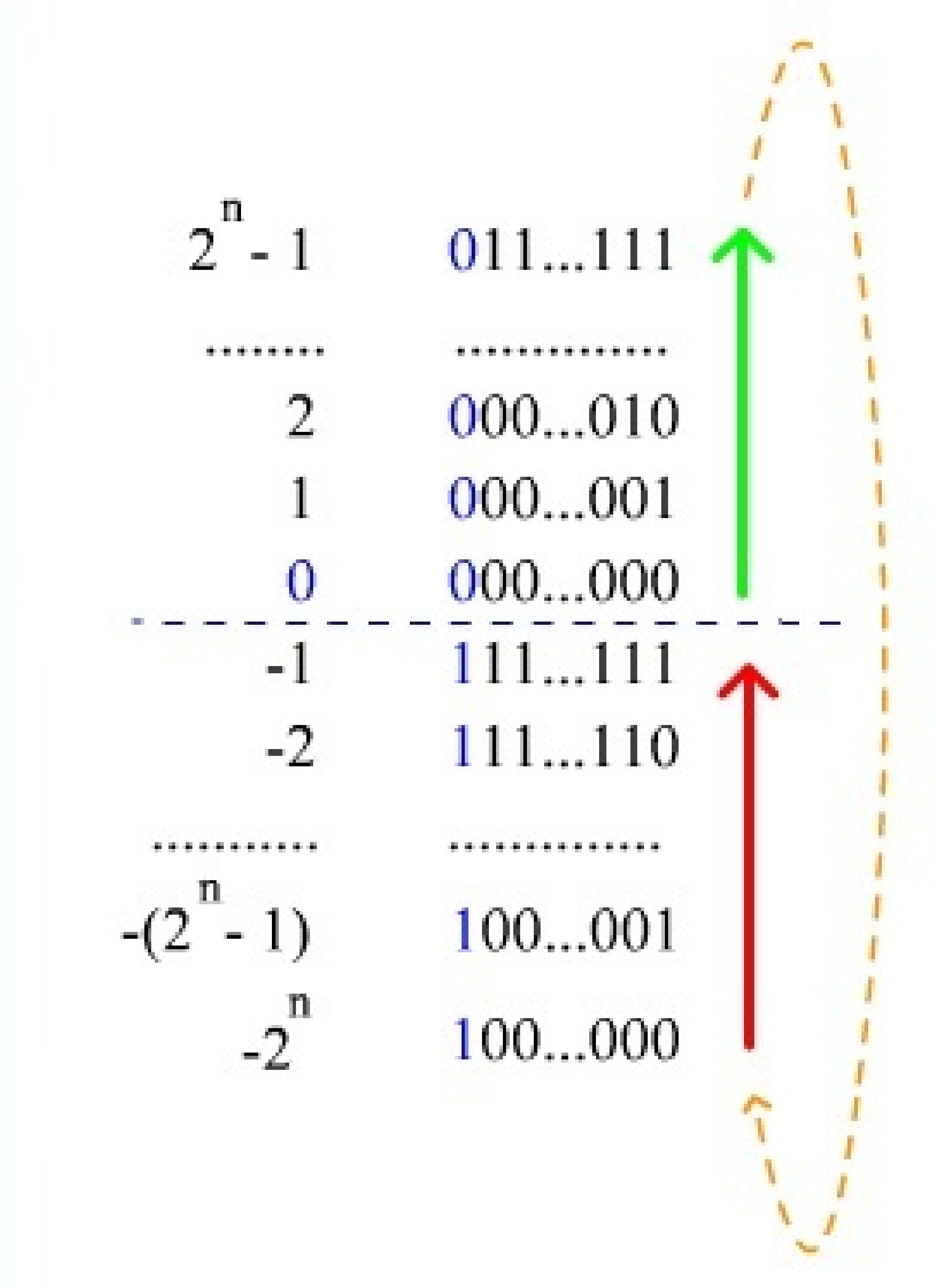
5: 0b10000101 (-128 + 5)
-5: 0b01111011 (123 так как первые 128 чисел отрицательные)

🟧 Особенности:

➕ Один ноль
➖ Кол-во положительных чисел не равно кол-ву отрицательных
➖ Нужно выбирать каких чисел будет на одно больше

Дополнение до двух

Можно сделать вес самого старшего разряда -2^n т.е. диапазон $[-2^n \dots 2^{n-1} - 1]$.



Пример

5: 0b00000101 *(5)*
-5: 0b11111011 *(-128 + 123)*
11: 0b00001011 *(11)*

Сложим -5 и 11:
-5: 0b11111011

11: 0b00001011
->6: 0b00000110

Еще пример с конкретными вычислениями

Вычисление чисел:

- если число неотрицательное, то в старший разряд записывается ноль, далее записывается само число;
- если число отрицательное, то все биты модуля числа инвертируются, то есть все единицы меняются на нули, а нули — на единицы, к инвертированному числу прибавляется единица, далее к результату дописывается знаковый разряд, равный единице.

🟧 Особенности:

➕ Доминирующая форма представления
➕ Легко определить знак числа (посмотреть старший)
➕ Проверка четности (посмотреть младший)
➖ Нечетное число чисел $\implies$ разное кол-во положительных и отрицательных чисел.
➖ Получаем результат по модулю от диапазона
➖ Умножение и деление работает по-другому

Проблема

```
int main() {
    int8_t x = INT8_MIN; // -128 <=> 0b10000000
    if (x < 0) {
        x = -x;
    }
} // x = -128
```

Что происходит:

1. Выполняется Integer promotion и int8_t кастится в int. Получаем -128 в 32-битном типе.

- из 0b10000000 получаем 0b11111111 11111111 11111111 10000000

2. Берется унарный минус от полученного числа (инвертирование + 1)

- 0b11111111 11111111 11111111 10000000 → 0b00000000 00000000 00000000 01111111 + 1 → 0b00000000 00000000 00000000 10000000

3. 0b00000000 00000000 00000000 10000000 = 128 в int
4. В int8_t это 0b10000000 = -128

> ⚠️ Если брать вместе int8_t → int, то при унарном минусе будет переполнение и это считается UB так как в int8_t оно кастится и в самом int переполнения нету, а потом усечение в 8 бит. А тут сразу работа с int, но зачастую компилятор сделает такую же логику

Неправильное решение

```
int example2() {
    int x = INT32_MIN;
    unsigned int y;
    if (x < 0) {
        y = -x; // вышли за диапазон
    } else {
        y = x;
    } // y = 2147483648
    return 0;

}
```

При увеличении значений все считается правильно, но решение все равно <u>не является корректным</u> с точки зрения ЭВМ.

1. Происходит вся так же логика с переполнением знакового типа, что UB
2. Но процессор просто тупо выполняет функцию унарного минуса и не делает никаких проверок
3. Все кладется в unsigned и магическим образом получается

Замечание

1. Для *беззнаковых* типов выполняется <u>модулярная арифметика</u> (по модулю 2^n)

2. Для знаковых: UB (так как железо вело себя по разному с разным представлением чисел) => решает компилятор, что делать, так как по стандарту UB - зачастую исправляется компилятором

Правильное решение

```
int example3() {
    int x = INT32_MIN;
    unsigned int y;
    if (x < 0) {
        y = -(unsigned int)x;
    } else {
        y = x;
    } // y = 2147483648
    return 0;
}
```

Почему работает:
1. Преобразование в uint определено стандартом (по модулю 2^{32}, т.е. берутся низшие 32 бита)
2. Унарный минус тоже в модулярной арифметике не UB

$\implies$ Безопасность ✔

> ⚠️ Некоторые компиляторы могут поставить *warning* так как `-uint` не имеет большого смысла. Но если добавить ключ компилятору на равенство варнингов и ошибок $\to$ ошибка компиляции

Деление

```
int a,b,c;
c = a / b

5/2 -> 2
-5/2 -> -2 // в C округление к нулю, для дробных к ближайшему четном
```

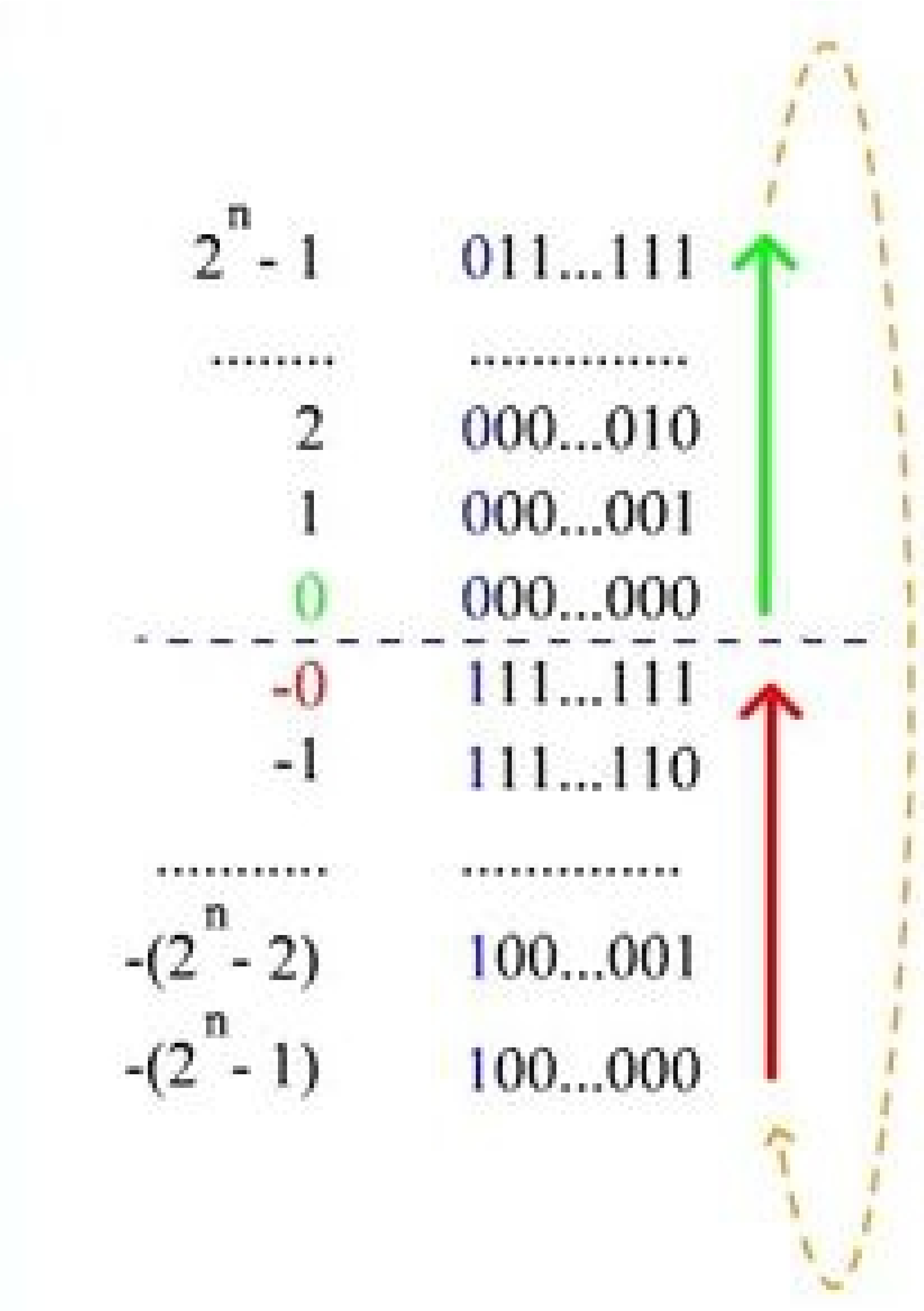
```
-5%2 -> -1 // зависит от определения деления (по определению остатка
-5%2 -> 1 // на Python так как округление к -inf
```

```
int a,b,c;

if (b != 0 && (a != INT_MIN || b != -1)) {
    c = a/b;
}
```

Дополнение до одного

Вес старшего разряда отрицательный с симметрией чисел



Получение кода числа:

- если число положительное, то в старший разряд (который является знаковым) записывается ноль, а далее записывается само число;
- если число отрицательное, то код получается инвертированием представления модуля числа (получается обратный код);
- если число является нулем, то его можно представить двумя способами: +0 (000…000) или −0 (111…111).

🟧 Особенности:

➕ Простое получение кода отрицательных чисел.
➕ Из-за того, что 0 обозначает +, коды положительных чисел относительно беззнакового кодирования остаются неизменными.
➕ Количество положительных чисел равно количеству отрицательных.
➖ Выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора.
➖ Два нуля

Форма с чередованием

код	значение	бинарка
0	0	0b000
1	-1	0b001
2	1	0b010
3	-2	0b011
4	2	0b100
5	-3	0b101

Для положительных - модуль
Для отрицательных - модуль - 1

> Теперь младший бит отвечает за знак

Можно увидеть с кодом переменной длины

- не для вычислений
- для компактного хранения и передачи

Можно выбрать основание системы `-2`

т.е. -128 64 -32 16 -8 4 -2 1

MIN = -170
MAX = 85

Значений: 256

- отрицательных больше в два раза чем положительных (если нечетное кол-во бит то наоборот)
- есть расширяемость влево битиками

Еще одна модель только не с четным основанием с.с.

Изначально: 012
Тут: -101 (иногда -1 пишут как `z`)

- Предлагается брать с конца алфавита для отрицательных чисел для непересечения (как берем сначала для положительных)

5: 001zz
-5: 00z11

- один ноль
- симметричный диапазон
- расширение влево ноликами работает

Кодирование дробных чисел

Идея: Можно поделить поровну n.m (n бит для целой, m бит для дробной части)

- Для представления со знаком берем идею целых чисел

> ❗ Очень необязательно делать разделение не поровну

Идея распределения: 128 64 32 16 8 4 2 1 | 1/2 1/4 1/8 ... 1/256

🟩 Пример

5.375: 0b00000101|01100000 (5 в целой и $\frac{375}{1000} = \frac{3}{8} = \frac{1}{4} + \frac{1}{8}$ в дробной)
5.375: 0x0560

Как получили 16-ку:

- переводим 4-битные блоки в hex:
  0000 $\to$ 0
  0101 $\to$ 5
  0110 $\to$ 6
  0000 $\to$ 0
- Составляем число 0x0560

Сложение дробных чисел

```
// используем форму 16.16
uint a,b,c;
a = 0x00056000 // 5.375
b = 0x00018000 // 1.5
c = a + b; // 6.875
```

Формат **Q16.16** (целых 16 бит, дробных 16 бит).
То есть любое число хранится как целое `raw`, где: $value = \frac{raw}{2^{16}}$

> $a = 0x00056000$
> $raw_a = 0x00056000 = 352256$
> $a = \frac{352256}{2^{16}} = \frac{352256}{65536} = 5.375$
> $b = 0x00018000$
> $raw_b = 0x00018000 = 98304$
> $b = \frac{98304}{65536} = 1.5$
> $raw_C = raw_A + raw_B = 352256 + 98304 = 450560$
> $c = \frac{450560}{65536} = 6.875c = 65536450560 = 6.875$
> $A = \frac{raw_A}{2^{16}}$
> $B = \frac{raw_B}{2^{16}}$
> $C = A + B = \frac{raw_A}{2^{16}} + \frac{raw_B}{2^{16}} = \frac{raw_A + raw_B}{2^{16}}$
> $rawC = rawA + rawB$

Т.е. для сложения дробных чисел достаточно сложить их представления `raw`. Никакой дополнительной корректировки не нужно, потому что у всех одно и то же «основание» 2^{16}.

Аналогия

Это абсолютно как со школьной арифметикой в десятичной системе:

- Если мы договорились, что пишем все числа в «сотых долях» (то есть умножаем всё на 100), то:
  - 5.37 → 537
  - 1.50 → 150
  - Складываем: 537 + 150 = 687 → это 6.87 после деления на 100.

С фиксированной точкой то же самое, только вместо «умножить на 100» — «умножить на 2^{16}».

Умножение дробных чисел

```
// A*2^16 * B*2^16 != C*2^16 => нужно поделить на 2^16 (поправка вто
c = (ulong long)a * b / 0x10000; // + защита от переполнения

int a,b,c;
c = (long long)a * b >> 16; // получили округление к -inf
```

> $raw_a = 0x00056000 = 352256$
> $raw_b = 0x00018000 = 98304$
> $A = \frac{raw_A}{2^{16}}$
> $B = \frac{raw_B}{2^{16}}$
> $A \cdot B = \frac{rawA \cdot rawB}{2^{32}}$
> $rawC = \frac{rawA \cdot rawB}{2^{16}}$
> $C = \frac{rawA \cdot rawB}{2^{32}}$

- нужно поделить правую часть (a*b) на 2^{16} для достижения равенства
- Используем `ulong long` так как из-за умножения 32x32 бит потребуется до 64бит
- если беззнаковое, то компилятор может заменить на сдвиг и посчитать быстрее

Округление к ближайшему, иначе к четному
3.5 -> 4
2.5 -> 2
-2.5 -> 2
2.8 -> 3

Источники:

- neerc: Представление целых чисел: прямой код, код со сдвигом, дополнительный код