Задача 2.

1.	A	B	D	$C_1 = A + B$, $C_2 = A - B$, $C_3 = -A + B$, C_4 = D + B.
	111	35	47,48	Вычислить C_1, C_3, C_4 в двоичном дополнительном коде, а C_2 в обратном (размер байт). Представить: C_1 в прямой коде; C_4 – в восьмеричном; C_3 – в 16-ричном; C_2 – в двоично-десятичном. Указать случаи переполнения

Преобразование в двоичный код

Первый этап – получение двоичного кода. Путь получения – любой метод. В данном примере двоичный код рассчитан через восьмеричный код.

```
_111|8      _13|8
  8 |13      8|1
 _31          5
  24
   7
```

$111_{10} = 157_8 = 01\ 101\ 111_2$

```
_ 35|8
  32|4
   3
```

$35_{10} = 43_8 = 00\ 100\ 011_2$

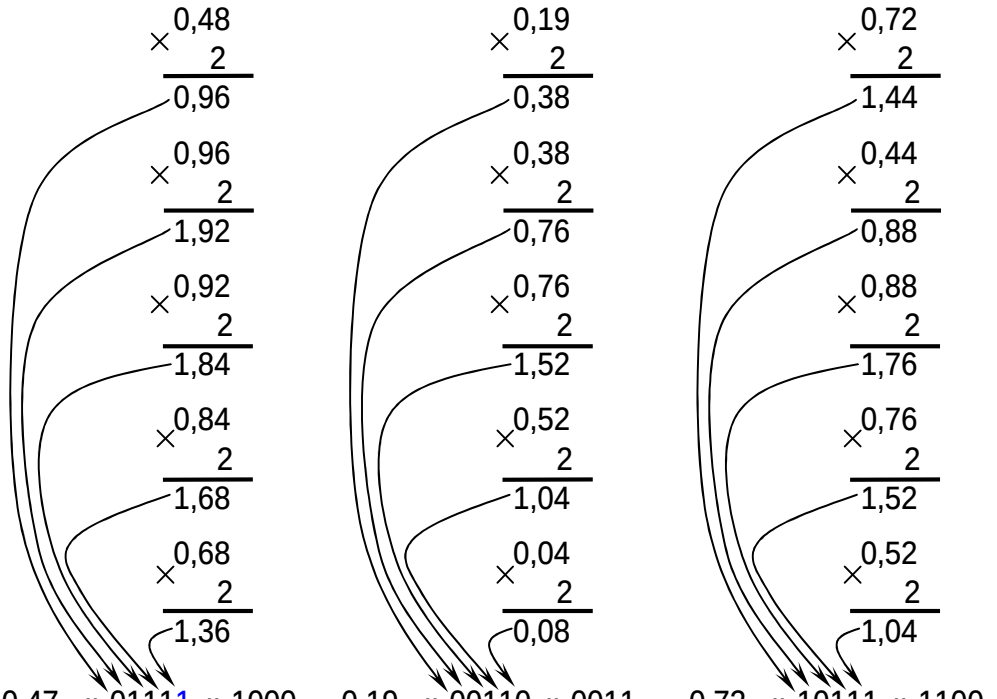
Третье число преобразуем по частям. Отдельно часть что до запятой и что после

```
_ 47|8
  40|5
   7
```

$47_{10} = 57_8 = 00\ 101\ 111_2$

Для десятичной достаточно четырех бит. И в целом число *D* будет занимать полтора байта.

Числа после запетой можно получать путем умножения на два или разложение с использованием ряда 2 с отрицательными степенями. Важным аспектом является округление, поскольку процесс преобразования часто не имеет окончания. Для получения полбайта надо будет получить 5 бит. Если последний пятый бит будет 1, то его добавляю к предыдущий, если ноль то предыдущие биты оставляют без изменения.



$0,47_{10} \approx 01111_2 \approx 1000_2$ $0,19_{10} \approx 00110_2 \approx 0011_2$ $0,72_{10} \approx 10111_2 \approx 1100_2$

После округления После округления После округления

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
0,5	0,25	0,125	0,0625	0,03125
_0,48	_0,23	_0,105	_0,0425	
0,25	0,125	0,0625	0,03125	
0,23	0,105	0,0425	0,01125	
0	1	1	1	1

Тогда $0,48_{10} \approx 01111_2 \approx 1000_2$

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
0,5	0,25	0,125	0,0625	0,03125
		_0,19	_0,065	
		0,125	0,0625	
		0,065	0,0025	
0	0	1	1	0

Тогда $0,19_{10} \approx 01100_2 \approx 0110_2$

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
0,5	0,25	0,125	0,0625	0,03125
_0,72		_0,22	_0,095	_0,0325
0,5		0,125	0,0625	0,03125
0,22		0,095	0,0325	0,00125
1	0	1	1	1

Тогда $0,72_{10} \approx 10111_2 \approx 1100_2$

С учетом преобразований число *D* в размере полтора байта будет иметь вид:
$D = 47,48 = 00101111\ 1000_2$.
В записи присутствую пробелы, но это для удобства. В реальный вычислительных системах с фиксированной точной никаких отделений нет. И правила суммирования применяются в общем виде.

Теория.

Размер сумматора в микропроцессоре всегда имеет ограничение на длину бит числа которые он может сложить. Исходя из этого, может наблюдаться, что получаемое значение будет выходить за отведенные рамки. Для выявления таких проблем существует механизм переполнения, который заключается в том, что перед непосредственно проведением операции суммирования бит знака удваивается. Если в результате суммирования эти два бита получаются одинаковые, значит суммирование прошло корректно, если нет, то образовалось переполнение.

Современные процессора редко содержат одновременно возможность суммирования и вычитания. Присутствует только одна команда. А противоположную операцию заменяю путем изменения знака числа.

$C_1 = A + B$

Значения чисел равны A = 01101111; B = 00100011_2. В модифицированном коде это будет A = 00.1101111; B = 00.0100 011. Точка несет вспомогательную роль для понимания. Поскольку числа положительные, то процесс сложения не отличается прямой, обратный или дополнительный это код.

В каждом разряде может быть 4 ситуации. Если в разряде один ноль, а другая единица, то результатом будет 1. Если оба нуля, то результат 0. Если две единицы, то результат в этом разряде 0, но одна единица переходит в следующий разряд. Если две единицы и еще третья с предыдущего разряда, то результат в этом разряде 1, одна единица переходит в следующий разряд. Переход единицы в следующий разряд обычно обозначается точкой.

```
    .  .  . . . .
+  00.1101111дк  (A)
   00.0100011дк  (B)
   01.0010010дк
```

Данный размер демонстрирует переполнение

$C_2 = A - B$

C_2 должно быть вычислено в обратном коде. Поэтому выражение преобразуем для исключения вычитания $C_2 = A + (-B)$ и преобразуем B в обратный код. Поскольку A положительное, преобразования не требуется. Если при выполнении суммирования в обратном коде получается 1 за рамками разрядной сетки, то ее следует прибавить к младшему биту.

–B = –(000100011) = $11.1011100_{ОК}$.

```
    . . .  . . . .
+  00.1101111ок  (A)
   11.1011100ок  (–B)
  100.1001011ок
+           1ок
  ------------
   00.1001100ок
```

Как видим, получилось положительное число. Переполнение отсутствует.

Для примера рассмотрим вариант сложения в обратном коде для $B - A$.

–A =– (00.1101111) = $11.0010000_{ОК}$.

```
+  00.0100011ок  (B)
   11.0010000ок  (–A)
   11.0110011ок
```

Получилось отрицательное число. Переполнение тоже отсутствует.

$C_3 = -A + B$

C_3 должно быть вычислено в дополнительном коде. Поэтому выражение преобразуем для исключения вычитания $C_2 = (-A) + B$ и преобразуем A в дополнительный код. Поскольку число B положительное, преобразования не требуется. Если при выполнении суммирования в дополнительном коде получается 1 за рамками разрядной сетки, то ее следует игнорировать.

$-A = -(00.1101111) = 11.0010001_{ДК}$.

```
          . .
  11.0010001 дк  (–A)
+ 00.0100011 дк  (B)
  ------------
  11.0110100 дк
```

Как видим, получилось отрицательное число. Переполнение отсутствует.

Для примера рассмотрим вариант сложения в обратном коде для $-B + A$.

$-B = -(000100011) = 11.1011101_{ДК}$.

```
    . . . . . . . . .
  11.1011101 дк  (–B)
+ 00.1101111 дк  (A)
  ------------
  100.1001100 дк
```

Получилось положительное число. Переполнение тоже отсутствует.

$C_4 = D + B$

Отличительной особенностью данного суммирования, что надо правильно расположить целую часть над целой.

```
      .  . . . .
  00.0100011000 дк  (B)
+ 00.0101111000 дк  (D)
  ------------
  00.1010010101000 дк
```

Формирование окончательного результата

Представить: C_1 в прямом коде; C_4 – в восьмеричном; C_3 – в 16-ричном; C_2 – в двоично-десятичном.

C_1 у нас получился с переполнение, поэтому результат на выходе может быть разный. Одни процессора «выбросят» флаг и потребую увеличения разрядной сетки. Тогда единицу в поле знака можно интерпретировать как весовой коэффициент 128. Старые процессора рассмотрят единицу как признак отрицательного числа.

$C_1 = 01.0010010_{ДК} \Rightarrow$ 00000000 10010010 = (128 + 16 + 2) = 146.

$C_1 = 01.0010010_{ДК} = 10010010_{ДК} = 10010001_{ОК} = 11101110_{ПК} = -(64 + 32 + 8 + 4 + 2) =$

$= -110$.

Получать десятичные числа не обязательно. Вычисления проведены для понимания проблемы переполнения.

$C_4 = 00.1010010101000 = 001\ 010\ 010\ 100\ 000 = 12240_8$

Для записи восьмеричного кода отдельно для целой части добавили количество нулей для кратности 3 битам. Последние три бита помечены синим цветом. А дальше три бита представлены с помощью десятичных чисел.

$C_3 = 11.0110100_{ДК} = 1011\ 0100_2 = B4_{16}$

Любой редактор шестнадцатеричного кода осуществляет представление не осуществляя преобразование в прямой код или еще куда.

Для представления числа в двоично-десятичном (BCD) необходимо получить десятичный код, а затем осуществить преобразование каждой десятичной цифры в двоичный код. двоично-десятичном код занимает обычно не менее двух байт.

$C_2 = 00.1001100_{ОК} = (64 + 8 + 4) = 76_{10} = 0000\ \ 0000\ 0111\ 0100_{2\text{-}10}$

0 0 7 6

В случае отрицательного числа под знак отводится все четыре бита

$-109_{10} = \ 1111\ 0001\ 0000\ 1001_{BCD}$

– 1 0 9