

Министерство образования Республики Беларусь
Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ
КАФЕДРА ИНФОРМАТИКИ

Отчёт по лабораторным работам №№1-3

По дисциплине «Архитектура вычислительных систем»
По теме «Арифметические операции с целыми числами»

Выполнил:
студент гр. 953501
Кременевский В.С.

Проверил:
Старший преподаватель
Шиманский В.В.

Минск 2021

Содержание

1. Цель работы	3
2. Постановка задачи.....	4
3. Теоретические сведения	5
3.1 Прямой код	5
3.2 Дополнительный код.....	7
3.3 Сложение и вычитание	8
3.4 Умножение и деление.....	9
4. Программная реализация	12
5. Выводы	21
Список литературы	22
Приложение 1. Текст программы.....	23

1. Цель работы

Рассмотреть представление чисел в прямом, обратном и дополнительном кодах. Изучить процессы выполнения арифметических операций над целыми числами с фиксированной точкой. Рассмотреть различные алгоритмы выполнения арифметических операций. Сравнить различные варианты алгоритмов и определить наиболее подходящие для реализации в АЛУ.

2. Постановка задачи

Задание к лабораторной работе 1

Написать программу эмулятора АЛУ, реализующего *Операции сложения и вычитания с фиксированной точкой* над двумя введенными числами, с возможностью пошагового выполнения алгоритмов.

Задание к лабораторной работе 2

Написать программу эмулятора АЛУ, реализующего *операцию умножения* над двумя введенными числами, с возможностью пошагового выполнения алгоритмов.

Задание к лабораторной работе 3

Написать программу эмулятора АЛУ, реализующего *операцию деления с фиксированной точкой* над двумя введенными числами, с возможностью пошагового выполнения алгоритмов

3. Теоретические сведения

Существует ряд способов хранения и представления целых чисел с фиксированной точкой в памяти компьютера. Среди них можно выделить: *прямой код* (ориентирован на представление положительных целых чисел в двоичном коде), *код со сдвигом*, *дополнительный код* (дополнение до единицы или до двух).

При выборе способа хранения целых чисел, следует ориентироваться на следующие условия:

- код не должен требовать усложнения архитектуры процессора для выполнения арифметических операций с отрицательными числами;
- не усложнял арифметические действия;
- хранил бы одинаковое количество положительных и отрицательных чисел;
- и др.

В рамках данной лабораторной работы рассматривается представление целых чисел в *прямом* и *дополнительном* кодах, а также создается программная реализация методов хранения информации и возможности произведения арифметических операций над ними (сложение, вычитание, умножение, деление).

3.1 Прямой код

Прямой код — способ представления двоичных чисел с фиксированной запятой в компьютерной арифметике. Главным образом используется для записи неотрицательных чисел.

Для представления целого положительного (неотрицательного) числа в компьютере используется следующее правило:

- число переводится в двоичную систему;
- результат дополняется нулями слева в пределах выбранного формата;
- последний разряд слева является знаковым, в положительном числе он равен 0.

При записи числа в прямом коде старший разряд (старший бит) объявляется знаковым (signed) разрядом (знаковым битом). Если знаковый бит равен 0, число положительное, иначе — отрицательное. В остальных разрядах записывается двоичное представление модуля числа.

Функция представления двоичных чисел (в том числе целых чисел и смешанных дробей) в прямом коде имеет вид:

$$[A]_{\text{пр}} = \begin{cases} A, & A \geq 0 \\ 2^n + |A|, & A < 0 \end{cases}$$

Величина числа A в прямом коде определяется по следующей формуле:

$$A = (1 - 2a_{\text{sign}}) \sum_{i=-k}^n a_i p^i$$

где:

i - номер разряда числа; отрицательное число — номер разряда справа от запятой; положительное число — номер разряда слева от запятой;

k - количество разрядов справа от запятой (кол-во разрядов дробной части числа);

n - количество разрядов слева от запятой (кол-во разрядов целой части числа);

p - основание системы счисления; равно 2 для двоичных чисел, 10 — для десятичных, 16 — для шестнадцатеричных и т. п.;

Например, число 6 в двоичном 8-разрядном представлении имеет вид:
0000 0110

Как и было сказано вначале — прямой код главным образом используется для записи и представления положительных (неотрицательных) чисел, потому что при использовании для чисел со знаком, у прямого кода есть два основных недостатка:

- В прямом коде есть два варианта записи числа 0 (например, 00000000 и 10000000 в восьмиразрядном представлении). Второе представление называется «отрицательный ноль»
- Использование прямого кода для представления отрицательных чисел в памяти компьютера предполагает или выполнение арифметических операций центральным процессором в прямом коде, или перевод чисел в другое представление (например, в дополнительный код) перед выполнением операций и перевод результатов обратно в прямой код (что неэффективно).



$2^n - 1$	011...111	
.....	
2	000...010	
1	000...001	
0	000...000	
-0	100...000	
-1	100...001	
-2	100...010	
.....	
$-(2^n - 1)$	111...111	

Рисунок 1. Нумерация двоичных чисел в двоичном представлении

3.2 Дополнительный код

Дополнительный код — наиболее распространённый способ представления отрицательных целых чисел в компьютерах. Он позволяет заменить операцию вычитания на операцию сложения и сделать операции сложения и вычитания одинаковыми для знаковых и беззнаковых чисел, чем упрощает архитектуру ЭВМ.

Дополнительный код для отрицательного числа можно получить инвертированием его двоичного модуля (первое дополнение) и прибавлением к инверсии единицы (второе дополнение), либо вычитанием числа из нуля.

Двоичное 8-разрядное число *со знаком* в дополнительном коде может представлять любое целое в диапазоне от -128 до $+127$. Если старший разряд равен нулю, то наибольшее целое число, которое может быть записано в оставшихся 7 разрядах, равно 127.

Достоинства представления чисел с помощью дополнительного кода:

- простое получение кода отрицательных чисел;
- из-за того, что 00 обозначает ++, коды положительных чисел относительно беззнакового кодирования остаются неизменными.
- количество положительных чисел равно количеству отрицательных.

Недостатки представления чисел с помощью кода с дополнением до единицы

- выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора;
- существуют два нуля: $+0+0$ и $-0-0$.

На рисунке 2 представлено изображение двоичных чисел в дополнительном коде:

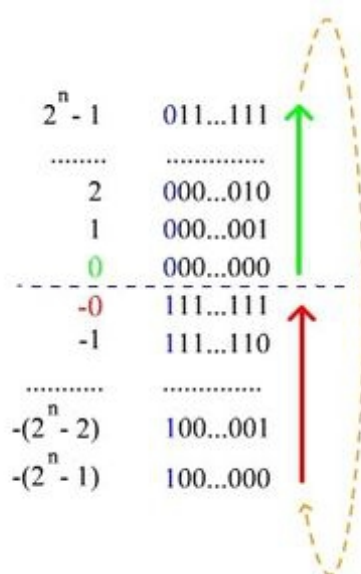


Рисунок 2. Нумерация чисел в дополнительном коде

Например, переведем число -13 в двоичный 8-разрядный код. Прямой код модуля -13 : 00001101. Инвертируем и получаем 11110010. Для получения из дополнительного кода самого числа достаточно инвертировать все разряды кода.

Так же важным с точки зрения работы с большими числами является понятие *расширения знака*:

Расширение знака — операция над двоичным числом, которая позволяет увеличить разрядность числа с сохранением знака и значения. Выполняется добавлением цифр со стороны старшего значащего разряда. Если число положительное (старший разряд равен 0), то добавляются нули, если отрицательное (старший разряд равен 1) — единицы.

Увеличение разрядности позволяет значительно увеличить диапазон значений числа. Например, в 16-битном представлении можно хранить 65 536 значений (формула была указана выше), в отличие от 256 в 8-битном представлении. Аналогичные преобразования для расширения разрядности используются и в *прямом коде*.

3.3 Сложение и вычитание

Рассмотрим операцию сложения на примере:

Выполнить сложение чисел $A = 53$ и $B = 14$ в двоичной системе счисления.

Переведем числа в двоичную систему счисления:

$$53_{10} = 110101_2,$$

$$14_{10} = 1110_2.$$

Запишем числа «А» и «В» столбиком, одно под другим, начиная с младших разрядов (нумерация разрядов начинается с нуля).

Сложим поразрядно числа «А» и «В» записывая результат в «С» начиная с младших разрядов. Весь процесс сложения наших чисел представлен в таблице 1.

Разряды	8	7	6	5	4	3	2	1
A	0	0	1	1	0	1	0	1
B	0	0	0	0	1	1	1	0
C	0	1	0	0	0	0	1	1

Таблица 1. Сложение чисел 53 и 14

Теперь опишем как происходит описанный выше процесс сложения и вычитания чисел в нашем эмуляторе АЛУ:

Изучим блоки эмулятора АЛУ. Главный блок представляет собой модуль сложения двух чисел. Процесс сложения происходит по правилам сложения беззнаковых чисел. Вычитание чисел достигается инвертированием второго числа в противоположный знак. Данный блок назовем *сумматором*.

Результат передается либо в один из регистров слагаемых (этот вариант показан на схеме), либо в третий регистр результата. Кроме кода результата сумматор формирует сигнал переполнения, который фиксируется в битовом флаге переполнения. Значение флага интерпретируется следующим образом: 0 — переполнение отсутствует, 1 – присутствует. При выполнении операции вычитания код вычитаемого, хранящийся перед началом операции в регистре В, передается на схему, выполняющую операцию отрицания, а уже с выхода этой схемы код поступает на вход сумматора.

3.4 Умножение и деление

Умножение и деление двоичных чисел практически не отличается от умножения и деления чисел, записанных в десятичной системе счисления. Единственным отличием является то, что при умножении в столбик не приходится находить произведение первого множителя на значения последовательных разрядов второго множителя, так как значение этих разрядов 1 или 0. А при делении в столбик не нужно подбирать неполное

делимое, так как учитывая специфику двоичных чисел, неполное делимое можно определить просто, посмотрев на делимое.

Введем правила умножение отдельных битов на таблице 2.

Первый	Второй	Результат
0	0	0
0	1	0
1	0	0
1	1	1

Таблица 2. Умножение отдельных битов

Пример умножения и деления чисел введем на рисунке 3. Как видно, правила визуально коррелируют с визуальным представлением сложением

$$\begin{array}{r}
 \times 1101111 \\
 \hline
 101101 \\
 1101111 \\
 1101111 \\
 1101111 \\
 1101111 \\
 \hline
 1001110000011
 \end{array}
 \qquad
 \begin{array}{r}
 - 111100 \mid 1010 \\
 \hline
 1010 \mid 110 \\
 - 1010 \\
 \hline
 1010 \\
 - 1010 \\
 \hline
 0
 \end{array}$$

Рисунок 3. Правила умножения и деления двоичного представления чисел

Реализация алгоритма умножения техническими или программными средствами позволяет несколько повысить его эффективность по сравнению с тем вариантом, который мы традиционно используем при вычислении в столбик вручную.

Конструкторы АЛУ предпочитают способ, который не требует выполнения дополнительного преобразования после завершения умножения. Одним из таких способов является *алгоритм Бута (Booth)*. Этот алгоритм также позволяет ускорить выполнение операции. Сомножители размещаются в регистрах Q (множитель) и M (множимое). Кроме них имеется одnorазрядный регистр Q-1 который связан с младшим разрядом (Q0) регистра Q. Произведение формируется в регистрах A и Q. В исходном состоянии в регистрах A и Q, записаны нули. Как и ранее, схема управления анализирует разряды множителя, но на сей раз анализируется пара соседних разрядов — основной и тот, который находится справа от него. Если оба разряда имеют одинаковые значения (11 или 00), все разряды регистров A, Q и Q-1 сдвигаются на 1 разряд вправо. Если соседние разряды имеют отличающиеся коды, то выполняется сложение или вычитание кода,

множимого из содержимого регистра А. Сложение, выполняется при комбинации кодов в соседних разрядах 01, а вычитание — при комбинации 10. Вслед за сложением или вычитанием выполняется сдвиг кодов в регистрах на один разряд вправо. Сдвиг выполняется таким образом, что старший разряд (крайний левый) регистра А (разряд A_{n-1}) сохраняется, хотя и переписывается в разряд A_{n-2} . Это необходимо для сохранения знака кода в регистрах А и Q. Такую операцию принято называть сдвигом с сохранением знака или арифметическим сдвигом

Для выполнения деления двоичных знаковых чисел также существует алгоритм, значительно отличающийся от ручного способа, но дающий преимущества в реализации в АЛУ. Ниже описан один из вариантов алгоритма деления чисел, представленных в дополнительном коде.

а. Загрузить делитель в регистр М, а делимое — в регистры А и Q. Делимое должно иметь формат $2n$ -разрядного дополнительного кода. Например, 4-разрядное число 0111 должно быть представлено в формате 00000111, а число 1001 должно быть преобразовано в 11111001.

б. Сдвинуть содержимое регистров А и Q на один разряд влево.

с. Если коды в регистрах М и А имеют одинаковые знаки, вычесть из содержимого А содержимое М и оставить результат в А. В противном случае добавить к содержимому А код из М.

д. Предыдущая операция считается успешной, если знак кода в регистре А не изменился в результате ее выполнения.

· Если операция была успешной или содержимое регистров А и Q равно нулю, то установить в младшем разряде частного (Q_0) код 1.

· Если операция не увенчалась успехом и содержимое одного из регистров А и Q (или обоих) отлично от нуля, то установить в младшем разряде частного (Q_0) код 0 и восстановить прежнее значение в регистре А.

е. Повторять операции, указанные в пунктах 2 и 4, столько раз, сколько разрядов в регистре Q.

ф. После завершения операции прочесть значение остатка в регистре А. Если знаки делимого и делителя одинаковы, значение частного извлечь из регистра Q; в противном случае значение частного равно содержимому регистра Q с обратным знаком (операция отрицания должна быть выполнена по правилам для дополнительного кода).

4. Программная реализация

В качестве языка для реализации эмулятора АЛУ был выбран язык C++.

Для работы с операциями сложения, вычитания, умножения и деления предусмотрено хранение двоичных чисел в 16-битном и вдвоенном 32-битном регистрах, что предусматривает диапазон в 65536 (2^{16}) и 4294967296 (2^{32}) значений соответственно. Предусмотрена возможность обработки ситуаций переполнения с выводом информации о проблеме.

На рисунках представлен результат работы программы. Выполняется сложение и вычитание двух чисел с фиксированной точкой. Демонстрируются операции умножения двух целых чисел и деления с восстановлением остатка. Показана обработка переполнений при операциях сложения, отрицания и деления.

Продемонстрируем результат программы при суммировании 2х чисел.

```
---Сложение---
Первое число: 1111 (00000100 01010111)
Второе число: 2222 (00001000 10101110)
|num_1| |num_2| |CF| |res|
  1      0      0      1
  1      1      0      0
  1      1      1      1
  0      1      1      0
  1      0      1      0
  0      1      1      0
  1      0      1      0
  0      1      1      0
  0      0      1      1
  0      0      0      0
  1      0      0      1
  0      1      0      1
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
1111 + 2222 = 3333
Результат: 3333 (00001101 00000101)
```

```
---Сложение---
Первое число: 128 (00000000 10000000)
Второе число: -16384 (11000000 00000000)
|num_1| |num_2| |CF| |res|
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  1      0      0      1
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      1      0      1
  0      1      0      1
128 + -16384 = -16256
Результат: -16256 (11000000 10000000)
```

```

---Сложение---
Первое число: -9999 (11011000 11110001)
Второе число: 10001 (00100111 00010001)
|num_1| |num_2| |CF| |res|
  1       1       0       0
  0       0       1       1
  0       0       0       0
  0       0       0       0
  1       1       0       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  0       1       1       0
  0       1       1       0
  0       1       1       0
  1       0       1       0
  1       0       1       0
  0       1       1       0
  1       0       1       0
  1       0       1       0
-9999 + 10001 = 2
Результат: 2 (00000000 00000010)

```

```

---Сложение---
Первое число: -16384 (11000000 00000000)
Второе число: -16384 (11000000 00000000)
|num_1| |num_2| |CF| |res|
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  0       0       0       0
  1       1       0       0
  1       1       1       1
-16384 + -16384 = -32768
Результат: -32768 (10000000 00000000)

```

Рис.1 Успешное суммирование двух чисел

```

---Сложение---
Первое число: -1 (11111111 11111111)
Второе число: -32768 (10000000 00000000)
|num_1| |num_2| |CF| |res|
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       0       0       1
  1       1       0       0
Переполнение
-1 + -32768 = 32767
Результат: 32767 (01111111 11111111)

```

```

---Сложение---
Первое число: 32767 (01111111 11111111)
Второе число: 1 (00000000 00000001)
|num_1| |num_2| |CF| |res|
  1       1       0       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  1       0       1       0
  0       0       1       1
Переполнение
32767 + 1 = -32768
Результат: -32768 (10000000 00000000)

```

Рис.2 Обработка переполнения при суммировании

Продемонстрируем результат программы при вычитании 2х чисел.

```

---Вычитание---
10000 - 1000
Первое число: 10000 (00100111 00010000)
Второе число: 1000 (00000011 11101000)
Инвертирование 2 числа:
~ 00000011 11101000 = 11111100 00011000 (-1000)
---Сложение---
Первое число: 10000 (00100111 00010000)
Второе число: -1000 (11111100 00011000)
|num_1| |num_2| |CF| |res|
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      1      0      1
  1      1      0      0
  0      0      1      1
  0      0      0      0
  0      0      0      0
  1      0      0      1
  1      0      0      1
  1      1      0      0
  0      1      1      0
  0      1      1      0
  1      1      1      1
  0      1      1      0
  0      1      1      0
10000 + -1000 = 9000
Результат: 9000 (00100011 00101000)

```

```

---Вычитание---
-128 - 128
Первое число: -128 (11111111 10000000)
Второе число: 128 (00000000 10000000)
Инвертирование 2 числа:
~ 00000000 10000000 = 11111111 10000000 (-128)
---Сложение---
Первое число: -128 (11111111 10000000)
Второе число: -128 (11111111 10000000)
|num_1| |num_2| |CF| |res|
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      0      0      0
  1      1      0      0
  1      1      1      1
  1      1      1      1
  1      1      1      1
  1      1      1      1
  1      1      1      1
  1      1      1      1
  1      1      1      1
-128 + -128 = -256
Результат: -256 (11111111 00000000)

```

```

---Вычитание---
1 - -10000
Первое число: 1 (00000000 00000001)
Второе число: -10000 (11011000 11110000)
Инвертирование 2 числа:
~ 11011000 11110000 = 00100111 00010000 (10000)
---Сложение---
Первое число: 1 (00000000 00000001)
Второе число: 10000 (00100111 00010000)
|num_1| |num_2| |CF| |res|
  1      0      0      1
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      1      0      1
  0      0      0      0
  0      0      0      0
  0      0      0      0
  0      1      0      1
  0      1      0      1
  0      1      0      1
  0      1      0      1
  0      0      0      0
  0      0      0      0
  0      1      0      1
  0      0      0      0
  0      0      0      0
1 + 10000 = 10001
Результат: 10001 (00100111 00010001)

```

```

---Вычитание---
-12345 - -9876
Первое число: -12345 (11001111 11000111)
Второе число: -9876 (11011001 01101100)
Инвертирование 2 числа:
~ 11011001 01101100 = 00100110 10010100 (9876)
---Сложение---
Первое число: -12345 (11001111 11000111)
Второе число: 9876 (00100110 10010100)
|num_1| |num_2| |CF| |res|
  1      0      0      1
  1      0      0      1
  1      1      0      0
  0      0      1      1
  0      1      0      1
  0      0      0      0
  1      0      0      1
  1      1      0      0
  1      0      1      0
  1      1      1      1
  1      1      1      1
  1      0      1      0
  1      0      1      1
  0      0      1      1
  0      1      0      1
  1      0      0      1
  1      0      0      1
-12345 + 9876 = -2469
Результат: -2469 (11110110 01011011)

```

Рис.3 Успешное вычитание двух чисел

```

---Вычитание---
0 - -32678
Первое число: 0 (00000000 00000000)
Второе число: -32678 (10000000 01011010)
Невозможно выполнить отрицание для -32678

```

Рис.4 Обработка невозможности выполнения вычитания

Продемонстрируем результат программы при умножении 2х чисел.

```

---Умножение---
1000 * 20 (Результат компьютера: 20000)
Первое число: 1000 (00000011 1101000) - reg_1
Второе число: 20 (00000000 00010100) - reg_2

      A      |      reg_2      | Q1 |
00000000 00000000 | 00000000 00010100 | 0 |
00000000 00000000 | 00000000 00010101 | 0 | >> shift
00000000 00000000 | 00000000 00001011 | 0 | >> shift
11111100 00010000 | 00000000 00000101 | 0 | 1-0 -> A = A-reg_1
11111110 00001000 | 00000000 00000010 | 1 | >> shift
00000001 11110100 | 00000000 00000010 | 1 | 0-1 -> A = A+reg_1
00000000 11110101 | 00000000 00000001 | 0 | >> shift
11111101 00010010 | 00000000 00000001 | 0 | 1-0 -> A = A-reg_1
11111110 10001001 | 00000000 00000000 | 1 | >> shift
00000010 01110001 | 00000000 00000000 | 1 | 0-1 -> A = A+reg_1
00000001 00111000 | 10000000 00000000 | 0 | >> shift
00000000 10011000 | 01000000 00000000 | 0 | >> shift
00000000 01001110 | 00100000 00000000 | 0 | >> shift
00000000 00100111 | 00010000 00000000 | 0 | >> shift
00000000 00010011 | 10001000 00000000 | 0 | >> shift
00000000 00001001 | 11000100 00000000 | 0 | >> shift
00000000 00000100 | 11100010 00000000 | 0 | >> shift
00000000 00000010 | 01110001 00000000 | 0 | >> shift
00000000 00000001 | 00111000 10000000 | 0 | >> shift
00000000 00000000 | 10011100 01000000 | 0 | >> shift
00000000 00000000 | 01001110 00100000 | 0 | >> shift

Регистр A      | Регистр reg_2
00000000 00000000 | 01001110 00100000
Результат умножения: 20000

---Умножение---
999 * -9999 (Результат компьютера: -9989001)
Первое число: 999 (00000011 11001111) - reg_1
Второе число: -9999 (11011000 11110001) - reg_2

      A      |      reg_2      | Q1 |
00000000 00000000 | 11011000 11110001 | 0 |
11111100 00011001 | 11011000 11110001 | 0 | 1-0 -> A = A-reg_1
11111110 00001100 | 11011000 01111000 | 1 | >> shift
00000001 11110011 | 11011000 01111000 | 1 | 0-1 -> A = A+reg_1
00000000 11111001 | 11101110 00111100 | 0 | >> shift
00000000 01111000 | 11111011 00011110 | 0 | >> shift
00000000 00111110 | 01111101 10001111 | 0 | >> shift
11111100 01010111 | 01111101 10001111 | 0 | 1-0 -> A = A-reg_1
11111110 00101011 | 10111110 11000111 | 1 | >> shift
11111111 00010101 | 11011111 01100011 | 1 | >> shift
11111111 10001010 | 11011111 10110001 | 1 | >> shift
11111111 11000101 | 01110111 11011000 | 1 | >> shift
00000011 10101100 | 01110111 11011000 | 1 | 0-1 -> A = A+reg_1
00000001 11010110 | 00111011 11101100 | 0 | >> shift
00000000 11101011 | 00011101 11110110 | 0 | >> shift
00000000 01110101 | 10001110 11111011 | 0 | >> shift
11111100 10001110 | 10001110 11111011 | 0 | 1-0 -> A = A-reg_1
11111110 01000111 | 01000111 01111101 | 1 | >> shift
11111111 00100011 | 10100011 10111110 | 1 | >> shift
00000011 00001010 | 10100011 10111110 | 1 | 0-1 -> A = A+reg_1
00000001 10000101 | 01010001 11011111 | 0 | >> shift
11111101 10011110 | 01010001 11011111 | 0 | 1-0 -> A = A-reg_1
11111110 11001111 | 00101000 11101111 | 1 | >> shift
11111111 01100111 | 10010100 01110111 | 1 | >> shift

Регистр A      | Регистр reg_2
11111111 01100111 | 10010100 01110111
Результат умножения: -9989001

---Умножение---
-98 * 50 (Результат компьютера: -4900)
Первое число: -98 (11111111 10011110) - reg_1
Второе число: 50 (00000000 00110010) - reg_2

      A      |      reg_2      | Q1 |
00000000 00000000 | 00000000 00110010 | 0 |
00000000 00000000 | 00000000 00011001 | 0 | >> shift
00000000 01100010 | 00000000 00011001 | 0 | 1-0 -> A = A-reg_1
00000000 00110001 | 00000000 00001100 | 1 | >> shift
11111111 11001111 | 00000000 00001100 | 1 | 0-1 -> A = A+reg_1
11111111 11100111 | 10000000 00000110 | 0 | >> shift
11111111 11110011 | 11000000 00000011 | 0 | >> shift
00000000 01010101 | 11000000 00000011 | 0 | 1-0 -> A = A-reg_1
00000000 00101010 | 11100000 00000001 | 1 | >> shift
00000000 00010101 | 01110000 00000000 | 1 | >> shift
11111111 10110011 | 01110000 00000000 | 1 | 0-1 -> A = A+reg_1
11111111 11011001 | 10111000 00000000 | 0 | >> shift
11111111 11101100 | 11011100 00000000 | 0 | >> shift
11111111 11101110 | 01101110 00000000 | 0 | >> shift
11111111 11110111 | 00110111 00000000 | 0 | >> shift
11111111 11111011 | 10011011 10000000 | 0 | >> shift
11111111 11111101 | 10011011 01110000 | 0 | >> shift
11111111 11111110 | 11001101 11000000 | 0 | >> shift
11111111 11111111 | 01100101 11100000 | 0 | >> shift
11111111 11111111 | 11011001 10111000 | 0 | >> shift
11111111 11111111 | 11101100 11011100 | 0 | >> shift

Регистр A      | Регистр reg_2
11111111 11111111 | 11101100 11011100
Результат умножения: -4900

---Умножение---
-12345 * -12345 (Результат компьютера: 152399025)
Первое число: -12345 (11001111 11000111) - reg_1
Второе число: -12345 (11001111 11000111) - reg_2

      A      |      reg_2      | Q1 |
00000000 00000000 | 11001111 11000111 | 0 |
00110000 00111001 | 11001111 11000111 | 0 | 1-0 -> A = A-reg_1
00011000 00011100 | 11100111 11100011 | 1 | >> shift
00001100 00001110 | 01110011 11110001 | 1 | >> shift
00000110 00000111 | 00111001 11111000 | 1 | >> shift
11010101 11001110 | 00111001 11111000 | 1 | 0-1 -> A = A+reg_1
11101010 11100111 | 00011100 11111100 | 0 | >> shift
11110101 01110011 | 10001110 01111110 | 0 | >> shift
11111010 10111001 | 11000111 00111111 | 0 | >> shift
00101010 11110010 | 11000111 00111111 | 0 | 1-0 -> A = A-reg_1
00010101 01111001 | 01100011 10011111 | 1 | >> shift
00001010 10111100 | 10110001 11001111 | 1 | >> shift
00000101 01011110 | 01011000 11100111 | 1 | >> shift
00000010 10101111 | 00101100 01110011 | 1 | >> shift
00000001 01010111 | 10010110 00111001 | 1 | >> shift
00000000 10101011 | 11001011 00011100 | 1 | >> shift
11100000 01110010 | 11001011 00011100 | 1 | 0-1 -> A = A+reg_1
11101000 00111001 | 01100101 10001110 | 0 | >> shift
11110100 00011100 | 10110010 11000111 | 0 | >> shift
00100100 01010101 | 10110010 11000111 | 0 | 1-0 -> A = A-reg_1
00010010 00101010 | 11011001 01100011 | 1 | >> shift
00001001 00010101 | 01101100 10110001 | 1 | >> shift

Регистр A      | Регистр reg_2
00001001 00010101 | 01101100 10110001
Результат умножения: 152399025

```

Рис.5 Успешное перемножение двух чисел

Продемонстрируем результат программы при умножении 2х чисел.

Пример 1.

```
---Деление---
10000 / 200 (Результат компьютера: 50 | Остаток: 0)
Первое число: 10000 (00100111 00010000) - reg_1
Второе число: 200 (00000000 11001000) - reg_2

      A          |      reg_1      |
00000000 00000000 | 00100111 00010000 |
00000000 00000000 | 01001110 00100000 | << shift | STEP 1
11111111 00111000 | 01001110 00100000 | Sub: A = A - reg_2
00000000 00000000 | 01001110 00100000 | Restore A
.....
00000000 00000000 | 10011100 01000000 | << shift | STEP 2
11111111 00111000 | 10011100 01000000 | Sub: A = A - reg_2
00000000 00000000 | 10011100 01000000 | Restore A
.....
00000000 00000001 | 00111000 10000000 | << shift | STEP 3
11111111 00111001 | 00111000 10000000 | Sub: A = A - reg_2
00000000 00000001 | 00111000 10000000 | Restore A
.....
00000000 00000010 | 01110001 00000000 | << shift | STEP 4
11111111 00111010 | 01110001 00000000 | Sub: A = A - reg_2
00000000 00000010 | 01110001 00000000 | Restore A
.....
00000000 00000100 | 11100010 00000000 | << shift | STEP 5
11111111 00111010 | 11100010 00000000 | Sub: A = A - reg_2
00000000 00000100 | 11100010 00000000 | Restore A
.....
00000000 00001001 | 11000100 00000000 | << shift | STEP 6
11111111 01000001 | 11000100 00000000 | Sub: A = A - reg_2
00000000 00001001 | 11000100 00000000 | Restore A
.....
00000000 00010011 | 10001000 00000000 | << shift | STEP 7
11111111 01001011 | 10001000 00000000 | Sub: A = A - reg_2
00000000 00010011 | 10001000 00000000 | Restore A
.....
00000000 00100111 | 00010000 00000000 | << shift | STEP 8
11111111 01011111 | 00010000 00000000 | Sub: A = A - reg_2
00000000 00100111 | 00010000 00000000 | Restore A
.....
00000000 01001110 | 00100000 00000000 | << shift | STEP 9
11111111 10000110 | 00100000 00000000 | Sub: A = A - reg_2
00000000 01001110 | 00100000 00000000 | Restore A
.....
00000000 10011100 | 01000000 00000000 | << shift | STEP 10
11111111 11010100 | 01000000 00000000 | Sub: A = A - reg_2
00000000 10011100 | 01000000 00000000 | Restore A
.....
00000001 00111000 | 10000000 00000000 | << shift | STEP 11
00000000 01110000 | 10000000 00000000 | Sub: A = A - reg_2
00000000 01110000 | 10000000 00000001 | reg_1[N] = 1
.....
00000000 11100001 | 00000000 00000010 | << shift | STEP 12
00000000 00011001 | 00000000 00000010 | Sub: A = A - reg_2
00000000 00011001 | 00000000 00000011 | reg_1[N] = 1
.....
00000000 00110010 | 00000000 00000110 | << shift | STEP 13
11111111 01101010 | 00000000 00000110 | Sub: A = A - reg_2
00000000 00110010 | 00000000 00000110 | Restore A
.....
00000000 01100100 | 00000000 00001100 | << shift | STEP 14
11111111 10011100 | 00000000 00001100 | Sub: A = A - reg_2
00000000 01100100 | 00000000 00001100 | Restore A
.....
```

```
.....
00000000 11001000 | 00000000 00011000 | << shift | STEP 15
00000000 00000000 | 00000000 00011000 | Sub: A = A - reg_2
00000000 00000000 | 00000000 00011001 | reg_1[N] = 1
.....
```

```
.....
00000000 00000000 | 00000000 00110010 | << shift | STEP 16
11111111 00111000 | 00000000 00110010 | Sub: A = A - reg_2
00000000 00000000 | 00000000 00110010 | Restore A
.....
```

```
Остаток(Регистр A) | Частное(Регистр reg_2)
00000000 00000000 | 00000000 00110010
```

```
Результат деления: 50 (00000000 00110010)
Остаток деления: 0 (00000000 00110010)
10000 = 50 * 200 + 0
```


Пример 2.

---Деление---

32767 / -32768 (Результат компьютера: 0 | Остаток: 32767)
Первое число: 32767 (01111111 11111111) - reg_1
Второе число: -32768 (10000000 00000000) - reg_2

A	reg_1	
00000000 00000000	01111111 11111111	
00000000 00000000	11111111 11111110	<< shift STEP 1
10000000 00000000	11111111 11111110	Add: A = A + reg_2
Переполнение		
00000000 00000000	11111111 11111110	Restore A
.....		
00000000 00000001	11111111 11111100	<< shift STEP 2
10000000 00000001	11111111 11111100	Add: A = A + reg_2
Переполнение		
00000000 00000001	11111111 11111100	Restore A
.....		
00000000 00000011	11111111 11111000	<< shift STEP 3
10000000 00000011	11111111 11111000	Add: A = A + reg_2
Переполнение		
00000000 00000011	11111111 11111000	Restore A
.....		
00000000 00000111	11111111 11110000	<< shift STEP 4
10000000 00000111	11111111 11110000	Add: A = A + reg_2
Переполнение		
00000000 00000111	11111111 11110000	Restore A
.....		
00000000 00001111	11111111 11100000	<< shift STEP 5
10000000 00001111	11111111 11100000	Add: A = A + reg_2
Переполнение		
00000000 00001111	11111111 11100000	Restore A
.....		
00000000 00011111	11111111 11000000	<< shift STEP 6
10000000 00011111	11111111 11000000	Add: A = A + reg_2
Переполнение		
00000000 00011111	11111111 11000000	Restore A

00000000 00111111	11111111 10000000	<< shift STEP 7
10000000 00111111	11111111 10000000	Add: A = A + reg_2
Переполнение		
00000000 00111111	11111111 10000000	Restore A
.....		
00000000 01111111	11111111 00000000	<< shift STEP 8
10000000 01111111	11111111 00000000	Add: A = A + reg_2
Переполнение		
00000000 01111111	11111111 00000000	Restore A
.....		
00000000 11111111	11111110 00000000	<< shift STEP 9
10000000 11111111	11111110 00000000	Add: A = A + reg_2
Переполнение		
00000000 11111111	11111110 00000000	Restore A
.....		
00000001 11111111	11111100 00000000	<< shift STEP 10
10000001 11111111	11111100 00000000	Add: A = A + reg_2
Переполнение		
00000001 11111111	11111100 00000000	Restore A
.....		
00000011 11111111	11111000 00000000	<< shift STEP 11
10000011 11111111	11111000 00000000	Add: A = A + reg_2
Переполнение		
00000011 11111111	11111000 00000000	Restore A
.....		
00000111 11111111	11110000 00000000	<< shift STEP 12
10000111 11111111	11110000 00000000	Add: A = A + reg_2
Переполнение		
00000111 11111111	11110000 00000000	Restore A
.....		
00001111 11111111	11100000 00000000	<< shift STEP 13
10001111 11111111	11100000 00000000	Add: A = A + reg_2
Переполнение		
00001111 11111111	11100000 00000000	Restore A

00011111 11111111	11000000 00000000	<< shift STEP 14
10011111 11111111	11000000 00000000	Add: A = A + reg_2
Переполнение		
00011111 11111111	11000000 00000000	Restore A
.....		
00111111 11111111	10000000 00000000	<< shift STEP 15
10111111 11111111	10000000 00000000	Add: A = A + reg_2
Переполнение		
00111111 11111111	10000000 00000000	Restore A
.....		
01111111 11111111	00000000 00000000	<< shift STEP 16
11111111 11111111	00000000 00000000	Add: A = A + reg_2
Переполнение		
01111111 11111111	00000000 00000000	Restore A
.....		

Так как знак Делимого(32767) не равен знаку Делителя (-32768)
Инвертируем значение полученного регистра reg_1
~ 00000000 00000000 = 00000000 00000000

Остаток(Регистр A) | Частное(Регистр reg_2)
01111111 11111111 | 00000000 00000000

Результат деления: 0 (00000000 00000000)
Остаток деления: 32767 (00000000 00000000)
32767 = 0 * -32768 + 32767

Пример 3.

```

---Деление---
-500 / 33 (Результат компьютера: -15 | Остаток: -5)
Первое число: -500 (11111110 00001100) - reg_1
Второе число: 33 (00000000 00100001) - reg_2

      A          |      reg_1      |
11111111 11111111 | 11111110 00001100 |
11111111 11111111 | 11111100 00011000 | << shift | STEP 1
00000000 00100000 | 11111100 00011000 | Add: A = A + reg_2
11111111 11111111 | 11111100 00011000 | Restore A
.....
11111111 11111111 | 11111000 00110000 | << shift | STEP 2
00000000 00100000 | 11111000 00110000 | Add: A = A + reg_2
11111111 11111111 | 11111000 00110000 | Restore A
.....
11111111 11111111 | 11110000 01100000 | << shift | STEP 3
00000000 00100000 | 11110000 01100000 | Add: A = A + reg_2
11111111 11111111 | 11110000 01100000 | Restore A
.....
11111111 11111111 | 11100000 11000000 | << shift | STEP 4
00000000 00100000 | 11100000 11000000 | Add: A = A + reg_2
11111111 11111111 | 11100000 11000000 | Restore A
.....
11111111 11111111 | 11000001 10000000 | << shift | STEP 5
00000000 00100000 | 11000001 10000000 | Add: A = A + reg_2
11111111 11111111 | 11000001 10000000 | Restore A
.....
11111111 11111111 | 10000011 00000000 | << shift | STEP 6
00000000 00100000 | 10000011 00000000 | Add: A = A + reg_2
11111111 11111111 | 10000011 00000000 | Restore A
.....
11111111 11111111 | 00000110 00000000 | << shift | STEP 7
00000000 00100000 | 00000110 00000000 | Add: A = A + reg_2
11111111 11111111 | 00000110 00000000 | Restore A
.....

```

```

.....
11111111 11111110 | 00001100 00000000 | << shift | STEP 8
00000000 00011111 | 00001100 00000000 | Add: A = A + reg_2
11111111 11111110 | 00001100 00000000 | Restore A
.....
11111111 11111100 | 00011000 00000000 | << shift | STEP 9
00000000 00011101 | 00011000 00000000 | Add: A = A + reg_2
11111111 11111100 | 00011000 00000000 | Restore A
.....
11111111 11111000 | 00110000 00000000 | << shift | STEP 10
00000000 00011001 | 00110000 00000000 | Add: A = A + reg_2
11111111 11111000 | 00110000 00000000 | Restore A
.....
11111111 11110000 | 01100000 00000000 | << shift | STEP 11
00000000 00010001 | 01100000 00000000 | Add: A = A + reg_2
11111111 11110000 | 01100000 00000000 | Restore A
.....
11111111 11100000 | 11000000 00000000 | << shift | STEP 12
00000000 00000001 | 11000000 00000000 | Add: A = A + reg_2
11111111 11100000 | 11000000 00000000 | Restore A
.....
11111111 11000001 | 10000000 00000000 | << shift | STEP 13
11111111 11000010 | 10000000 00000000 | Add: A = A + reg_2
11111111 11000010 | 10000000 00000001 | reg_1[N] = 1
.....
11111111 11000101 | 00000000 00000010 | << shift | STEP 14
11111111 11000110 | 00000000 00000010 | Add: A = A + reg_2
11111111 11000110 | 00000000 00000011 | reg_1[N] = 1
.....
11111111 11001100 | 00000000 00000110 | << shift | STEP 15
11111111 11001101 | 00000000 00000110 | Add: A = A + reg_2
11111111 11001101 | 00000000 00000111 | reg_1[N] = 1
.....
11111111 11011010 | 00000000 00001110 | << shift | STEP 16
11111111 11111011 | 00000000 00001110 | Add: A = A + reg_2
11111111 11111011 | 00000000 00001111 | reg_1[N] = 1
.....

```

```

.....
11111111 11011010 | 00000000 00001110 | << shift | STEP 16
11111111 11111011 | 00000000 00001110 | Add: A = A + reg_2
11111111 11111011 | 00000000 00001111 | reg_1[N] = 1
.....

```

Так как знак Делимого(-500) не равен знаку Делителя (33)

Инвертируем значение полученного регистра reg_1

~ 00000000 00001111 = 11111111 11110001

Осаток(Регистр A) | Частное(Регистр reg_2)

11111111 11111011 | 11111111 11110001

Результат деления: -15 (11111111 11110001)

Остаток деления: -5 (11111111 11110001)

-500 = -15 * 33 + -5

Пример 4.

```

---Деление---
-11111 / -2222 (Результат компьютера: 5 | Остаток: -1)
Первое число: -11111 (11010100 10011001) - reg_1
Второе число: -2222 (11110111 01010010) - reg_2

      A          |      reg_1      |
11111111 11111111 | 11010100 10011001 |
11111111 11111111 | 10101001 00110010 | << shift | STEP 1
00001000 10101101 | 10101001 00110010 | Sub: A = A - reg_2
11111111 11111111 | 10101001 00110010 | Restore A
.....
11111111 11111111 | 01010010 01100100 | << shift | STEP 2
00001000 10101101 | 01010010 01100100 | Sub: A = A - reg_2
11111111 11111111 | 01010010 01100100 | Restore A
.....
11111111 11111110 | 10100100 11001000 | << shift | STEP 3
00001000 10101100 | 10100100 11001000 | Sub: A = A - reg_2
11111111 11111110 | 10100100 11001000 | Restore A
.....
11111111 11111101 | 01001001 10010000 | << shift | STEP 4
00001000 10101011 | 01001001 10010000 | Sub: A = A - reg_2
11111111 11111101 | 01001001 10010000 | Restore A
.....
11111111 11111010 | 10010011 00100000 | << shift | STEP 5
00001000 10101000 | 10010011 00100000 | Sub: A = A - reg_2
11111111 11111010 | 10010011 00100000 | Restore A
.....
11111111 11110101 | 00100110 01000000 | << shift | STEP 6
00001000 10100011 | 00100110 01000000 | Sub: A = A - reg_2
11111111 11110101 | 00100110 01000000 | Restore A
.....
11111111 11101010 | 01001100 10000000 | << shift | STEP 7
00001000 10011000 | 01001100 10000000 | Sub: A = A - reg_2
11111111 11101010 | 01001100 10000000 | Restore A
.....
11111111 11010100 | 10010001 00000000 | << shift | STEP 8
00001000 10000010 | 10010001 00000000 | Sub: A = A - reg_2
11111111 11010100 | 10010001 00000000 | Restore A
.....
11111111 10101001 | 00110010 00000000 | << shift | STEP 9
00001000 01010111 | 00110010 00000000 | Sub: A = A - reg_2
11111111 10101001 | 00110010 00000000 | Restore A
.....
11111111 01010010 | 01100100 00000000 | << shift | STEP 10
00001000 00000000 | 01100100 00000000 | Sub: A = A - reg_2
11111111 01010010 | 01100100 00000000 | Restore A
.....
11111110 10100100 | 11001000 00000000 | << shift | STEP 11
00000111 01010010 | 11001000 00000000 | Sub: A = A - reg_2
11111110 10100100 | 11001000 00000000 | Restore A
.....
11111101 01001001 | 10010000 00000000 | << shift | STEP 12
00000101 11110111 | 10010000 00000000 | Sub: A = A - reg_2
11111101 01001001 | 10010000 00000000 | Restore A
.....
11111010 10010011 | 00100000 00000000 | << shift | STEP 13
00000011 01000001 | 00100000 00000000 | Sub: A = A - reg_2
11111010 10010011 | 00100000 00000000 | Restore A
.....
11110101 00100110 | 01000000 00000000 | << shift | STEP 14
11111101 11010100 | 01000000 00000000 | Sub: A = A - reg_2
11111101 11010100 | 01000000 00000001 | reg_1[N] = 1
.....
11111011 10101000 | 10000000 00000010 | << shift | STEP 15
00000100 01010110 | 10000000 00000010 | Sub: A = A - reg_2
11111011 10101000 | 10000000 00000010 | Restore A
.....
11110111 01010001 | 00000000 00000100 | << shift | STEP 16
11111111 11111111 | 00000000 00000100 | Sub: A = A - reg_2
11111111 11111111 | 00000000 00000101 | reg_1[N] = 1
.....

Остаток(Регистр A) | Частное(Регистр reg_2)
11111111 11111111 | 00000000 00000101

Результат деления: 5 (00000000 00000101)
Остаток деления: -1 (00000000 00000101)
-11111 = 5 * -2222 + -1

```

Рис.6 Успешное деление двух чисел

---Деление---

Первое число: 1 (00000000 00000001) - reg_1

Второе число: 0 (00000000 00000000) - reg_2

Ошибка при делении! Не предусмотрено деление на 0!

---Деление---

-32768 / -1 (Результат компьютера: 32768 | Остаток: 0)

Первое число: -32768 (10000000 00000000) - reg_1

Второе число: -1 (11111111 11111111) - reg_2

Переполнение при делении!

Рис.7 Обработка переполнений при делении

5. Выводы

Дополнительный код позволяет хранить знаковые целые числа с фиксированной точкой и проводить над ними арифметические операции.

Операция отрицания состоит из двух последовательных команд: инвертирования всех битов числа и добавления единицы. Из-за ограниченности диапазона значений в дополнительном коде, невозможно отрицать минимальное число (число формата $100\dots 00_2$), так как в результате выполнения команд получится то же самое число.

Операция сложения целых чисел в дополнительном коде представляет собой обычное суммирование «в столбик», начиная с младшего разряда, при этом перенос из старшего разряда игнорируется.

При выполнении сложения чисел с одинаковыми знаками результат может не поместиться в используемую разрядную сетку. Такое явление расценивается как переполнение, и для его обнаружения используется следующее правило: Если знаки слагаемых совпадают, то переполнение возникает, если знак суммы отличается от знака слагаемых.

Операция вычитания выполняется по следующему правилу: Для вычитания одного числа (вычитаемого) из другого (уменьшаемого) необходимо предварительно выполнить операцию отрицания над вычитаемым, а затем добавить результат к уменьшаемому по правилам сложения в дополнительном коде.

Одним из способов умножить два числа в дополнительном коде является алгоритм Бута, который позволяет обойтись меньшим количеством операций сложения и вычитания, чем простейший алгоритм, а также не требует выполнения дополнительного преобразования после завершения умножения.

Для выполнения деления двоичных знаковых чисел в дополнительном коде также существует алгоритм, значительно отличающийся от ручного способа, но дающий преимущества в реализации в АЛУ. Этот алгоритм также вычисляет остаток от деления.

При выполнении операции деления может возникнуть ситуация переполнения, если осуществляется попытка деления на ноль или в случае, если частное или остаток превышают размерность регистра.

В ходе выполнения лабораторной работы была написана программа, реализующая основные арифметические операции над числами в дополнительном коде. Также осуществлена проверка на возникновение исключительных ситуаций в процессе вычислений.

Список литературы

1. Волорова Н. А. Лабораторный практикум по курсу «Архитектура вычислительных систем» для студентов специальности «Информатика» / 93-444-487-2- Мн.: БГУИР, 2003. – 32с.:ил.

Приложение 1. Текст программы

Файл Register.h

```
#ifndef LAB123_REGISTER_H
#define LAB123_REGISTER_H

#include <iostream>
#include <vector>
#include <iomanip>

#define show_add false
#define show_sub false
#define show_mul false
#define show_div true

class Register{

    static const int N = 16;
    int number;
    std::vector<int> binary;

    struct Flags{
        int OF = 0;
    } flags;

    void printBits() const;

    static std::vector<int> makeBinary(int num);
    static void reverseArrBits(std::vector<int> &arr);
    static void twos_complement(std::vector<int> &bin);
    static int toInt(const Register& reg);

public:

    Register();
    explicit Register(int num);
    explicit Register(const std::vector<int> &bitsArr);
    Register(const Register& reg);

    void setBinary(const std::vector<int> &binArr);
    void setNumber(const int num);
    void setRegister(const Register& reg);

    static Register reverseBits(const Register &reg);
    void reverseBits();
    static int getBigNum(const Register& reg_1, const Register& reg_2);

    Register& operator=(const Register& reg);

private:
    int& operator[](const int index);
```

```

const int& operator[](const int index) const;

public:
    void setIndexedVal(int index, int value);
    Register& operator++();
    Register operator++(int);

    Register operator+(const Register &reg_2);
    Register operator+(int& num);
    friend Register operator+(int& num, Register& reg_2);

    Register operator-(const Register& reg_2);
    Register operator-(int& num);
    friend Register operator-(int& num, Register& reg_2);

    Register operator*(Register& reg_2);
    Register operator/(Register& reg_2);

    friend bool operator==(const Register& reg_1, const Register& reg_2);
    friend bool operator==(const Register& reg_1, const int& num);

    Register operator<<(const Register& reg2) const;
    Register operator<<(int num) const;
    Register operator>>(const Register& reg_2) const;
    Register operator>>(int num) const;

    friend std::ostream& operator<< (std::ostream &out, const Register& reg);
    friend std::istream& operator>> (std::istream &in, Register& reg);

    explicit operator int() const;
};

std::ostream& operator<< (std::ostream &out, const std::vector<int>& bitsArr);

#endif //LAB123_REGISTER_H

```

Файл Register.cpp

```

#include "Register.h"

void Register::setBinary(const std::vector<int> &binArr) {
    binary = binArr;
    number = (int)(*this);
}

void Register::setNumber(const int num){
    number = num;
    binary = makeBinary(abs(num));
    if (num < 0) {
        twos_complement(binary);
    }
}

```



```

    }
}

void Register::setRegister(const Register& reg) {
    number = reg.number;
    binary = reg.binary;
}

void Register::reverseBits(){
    reverseArrBits(binary);
    ++(*this);
}

Register Register::reverseBits(const Register& reg){
    Register reversed(reg.binary);
    reversed.reverseBits();
    return reversed;
}

void Register::twos_complement(std::vector<int>& bin){
    reverseArrBits(bin);
    int i = N-1;
    while(i >= 0){
        if (bin[i] == 1) {
            bin[i] = 0;
        }
        else {
            bin[i] = 1;
            break;
        }
        --i;
    }
}

void Register::printBits() const{
    for(int i = 0; i < N; ++i) {
        std::cout << binary[i];
    }
}

std::vector<int> Register::makeBinary(int num) {

    std::vector<int> binArr(N, 0);
    int i = 0;
    while(num > 0) {
        binArr[N-1-i] = num % 2;
        num /= 2;
        ++i;
    }
}

```

```

        return binArr;
    }

void Register::reverseArrBits(std::vector<int> &arr) {
    for(int i = 0; i < N; ++i) {
        arr[i] == 1 ? arr[i] = 0 : arr[i] = 1;
    }
}

int Register::toInt(const Register& reg){
    int intNum = 0;
    intNum -= reg.binary[0] * pow(2, N - 1);
    for(int i = N - 1; i > 0; --i){
        if (reg.binary[i] == 0){
            continue;
        }
        intNum += pow(2, N-i-1);
    }
    return intNum;
}

Register::Register() {
    binary = std::vector<int>(N, 0);
    number = 0;
}

Register::Register(int num) : Register() {
    setNumber(num);
}

Register::Register(const std::vector<int> &bitsArr) : Register() {
    setBinary(bitsArr);
}

Register::Register(const Register& reg){
    setNumber(reg.number);
}

Register& Register::operator=(const Register& reg){
    setRegister(reg);
    return *this;
}

int& Register::operator[](const int index){

    return this->binary[index];
}

```

```

const int& Register::operator[](const int index) const {
    return this->binary[index];
}

void Register::setIndexedVal(int index, int value){
    (*this)[index] = value;
    this->setBinary(this->binary);
}

Register& Register::operator++(){
    int i = N-1;
    while(i >= 0){
        if (binary[i] == 0){
            binary[i] = 1;
            break;
        }
        binary[i] = 0;
        --i;
    }
    return *this;
}

Register Register::operator++(int){
    Register temp(binary);
    ++(*this);
    return temp;
}

Register Register::operator+(const Register &reg_2){
    int r{};
    int r_prev{};
    Register result;
    if (show_add){
        std::cout << "---Сложение---\n";
        std::cout << "Первое число: " << int(*this) << " (" << this->binary << ")"<< '\n';
        std::cout << "Второе число: " << int(reg_2) << " (" << reg_2.binary << ")"<< '\n';

        std::cout << std::setw(5);
        std::cout << "|num_1|" << "|num_2|" << "|CF|" << "|res|\n";
    }
    for(int i = N-1; i >= 0; --i) {
        r_prev = r;
        if ((this->binary[i] + reg_2[i] + r) == 1) {
            r = 0;
            result[i] = 1;
        }
        else if ((this->binary[i] + reg_2[i] + r) == 2) {
            r = 1;
            result[i] = 0;
        }
    }
}

```

```

else if(this->binary[i] + reg_2[i] + r == 3) {
    r = 1;
    result[i] = 1;
}
else{
    r = 0;
    result[i] = this->binary[i] + reg_2[i];
}
if (show_add){
    std::cout << std::setw(4) << binary[i];
    std::cout << std::setw(8) << reg_2[i];
    std::cout << std::setw(6) << r_prev;
    std::cout << std::setw(6) << result[i] << '\n';
}
}
if ((this->binary[0] == reg_2[0]) && (this->binary[0] != result[0])){
    result.flags.OF = 1;
    std::cout << "Переполнение\n";
}

if(show_add){
    std::cout << (int)*this << " + " << (int)reg_2 << " = " << (int)result << '\n';
    std::cout << "Результат: " << (int)result << " (" << result.binary << ")\n";
}
result.setBinary(result.binary);
return result;
}

Register Register::operator+(int& num){
    Register reg_2(num);
    Register result = *this + reg_2;
    return result;
}

Register operator+(int& num, Register& reg_2){
    Register result = reg_2 + num;
    return result;
}

Register Register::operator-(const Register& reg_2){
    if (show_sub){
        std::cout << "---Вычитание---\n";
        std::cout << (int)*this << " - " << (int)reg_2 << '\n';
        std::cout << "Первое число: " << int(*this) << " (" << this->binary << ")<< '\n';
        std::cout << "Второе число: " << int(reg_2) << " (" << reg_2.binary << ")<< '\n';
    }

    Register reg_2_temp(reg_2);
    if (reg_2 == -32678){
        std::cout << "Невозможно выполнить отрицание для " << int(reg_2) << '\n';
        return Register(0);
    }
}

```

```

    reg_2_temp.reverseBits();
    if (show_sub){
        std::cout << "Инвертирование 2 числа:\n";
        std::cout << "~ " << reg_2.binary << " = " << reg_2_temp.binary << " (" << (int)reg_2_temp << ")\n";
    }
    Register result = *this + reg_2_temp;
    return result;
}

```

```

Register Register::operator-(int& num){
    Register reg_2(num);
    Register result = *this - reg_2;
    return result;
}

```

```

Register operator-(int& num, Register& reg_2){
    Register reg_1(num);
    Register result = reg_1 - reg_2;
    return result;
}

```

```

int Register::getBigNum(const Register& reg_1, const Register& reg_2){
    int num = 0;
    int j = 0;
    for(int i = 0; i < N; ++i){
        num += reg_2[N-i-1] * (int)pow(2, j);
        ++j;
    }
    for(int i = 0; i < N; ++i){
        num += reg_1[N-i-1] * (int)pow(2, j);
        ++j;
    }
    return num;
}

```

```

Register Register::operator*(Register& reg_2){
    if (show_mul){
        std::cout << "---Умножение---\n";
        std::cout << this->number << " * " << reg_2.number << " (Результат компьютера: " << (int)(*this) * (int)reg_2
        << ") "<<'\n';
        std::cout << "Первое число: " << int(*this) << " (" << this->binary << ") - reg_1\n";
        std::cout << "Второе число: " << int(reg_2) << " (" << reg_2.binary << ") - reg_2\n\n";
    }
}

```

```

Register A;
int Q1 = 0;
int A1 = 0;

```

```

if (show_mul) {
    std::cout << std::setw(10) << " A " << std::setw(9) << "|";
    std::cout << std::setw(14) << " reg_2 " << std::setw(6) << "| " << " Q1 | \n";
}

```

```

        std::cout << A.binary << " |";
        std::cout << std::setw(2) << reg_2.binary << " | ";
        std::cout << std::setw(2) << Q1 << " | \n";
    }

    for(int i = N-1; i >= 0; --i) {

        if (Q1 == 0 && reg_2[N-1] == 1){
            A = A - *this;
        }
        else if (Q1 == 1 && reg_2[N-1] == 0){
            A = A + *this;
        }

        if (show_mul && Q1 != reg_2[N-1]) {
            std::cout << A.binary << " |";
            std::cout << std::setw(2) << reg_2.binary << " | ";
            std::cout << std::setw(2) << Q1 << " | ";
            std::cout << reg_2.binary[N-1] << "-" << Q1 << "->";
            if (Q1 == 0)
                std::cout << " A = A-reg_1\n";
            else
                std::cout << " A = A+reg_1\n";

        }

        //    shifting
        A1 = A.binary[N-1];
        A = A >> 1;
        Q1 = reg_2.binary[N-1];
        reg_2 = reg_2 >> 1;
        reg_2.setIndexedVal(0, A1);
        //    reg_2[0] = A1;

        if (show_mul) {
            std::cout << A.binary << " |";
            std::cout << std::setw(2) << reg_2.binary << " | ";
            std::cout << std::setw(2) << Q1 << " | >> shift\n";
        }
    }

    if (show_mul){
        std::cout << '\n' << std::setw(20) << "Регистр A";
        std::cout << std::setw(7) << " | ";
        std::cout << std::setw(24) << "Регистр reg_2\n";
        std::cout << A.binary << " | " << reg_2.binary << '\n';
        std::cout << "Результат умножения: " << getBigNum(A, reg_2);
    }
    return A;
}

Register Register::operator/(Register& reg_2) {

    if (reg_2 == 0){
        std::cout << "Ошибка при делении! Не предусмотрено деление на 0!";
    }
}

```

```

    return Register(0);
}

if (*this == -32768 && reg_2 == -1) {
    std::cout << "Переполнение при делении!";
    return Register(0);
}

Register A;
if (this->binary[0] == 1){
    A.setNumber(-1);
}
Register M(reg_2);
Register reg_1(*this);

if (show_div) {
    std::cout << "---Деление---\n";
    std::cout << this->number << " / " << reg_2.number
        << " (Результат компьютера: " << (int) (*this) / (int) reg_2
        << " | Остаток: " << (int) (*this) % (int) reg_2 << ") " << "\n";
    std::cout << "Первое число: " << int(*this) << " (" << this->binary << ") - reg_1\n";
    std::cout << "Второе число: " << int(reg_2) << " (" << reg_2.binary << ") - reg_2\n\n";

    std::cout << std::setw(10) << " A " << std::setw(9) << "|";
    std::cout << std::setw(14) << " reg_1" << std::setw(7) << "| \n";
    std::cout << A.binary << " |";
    std::cout << std::setw(2) << reg_1.binary << " | \n";
}

int Qn{};
for (int i = N - 1; i >= 0; --i) {
    Qn = reg_1[0];
    A = A << 1;
    reg_1 = reg_1 << 1;
    A[N - 1] = Qn;

    if (show_div) {
        std::cout << A.binary << " |";
        std::cout << std::setw(2) << reg_1.binary << " | ";
        std::cout << "<< shift | STEP " << N-i << "\n";
    }

    int A_sign = A[0];

    if (M[0] == A[0]) {
        A = A - M;
        if (show_div) {
            std::cout << A.binary << " |";
            std::cout << std::setw(2) << reg_1.binary << " | ";
            std::cout << "Sub: A = A - reg_2\n";
        }
    } else {
        A = A + M;
        if (show_div) {

```

```

        std::cout << A.binary << " |";
        std::cout << std::setw(2) << reg_1.binary << " | ";
        std::cout << "Add: A = A + reg_2\n";
    }
}

if (A_sign == A[0] || (A == 0 && reg_1 == 0)) {
    reg_1.setIndexedVal(N - 1, 1);
    if (show_div) {
        std::cout << A.binary << " |";
        std::cout << std::setw(2) << reg_1.binary << " | ";
        std::cout << "reg_1[N] = 1\n";
    }
} else {
    if (M[0] == A_sign) {
        A = A + M;
    }
    else{
        A = A - M;
    }
    reg_1.setIndexedVal(N - 1, 0);
    if (show_div) {
        std::cout << A.binary << " |";
        std::cout << std::setw(2) << reg_1.binary << " | ";
        std::cout << "Restore A\n";
    }
}

if (show_div){
    for(int i = 0; i < 60; i++)
        std::cout << '.';
    std::cout << '\n';
}

if ((*this)[0] != M[0]) {
    Register saved_reg_1(reg_1);
    reg_1.reverseBits();

    if (show_div) {
        std::cout << "\nТак как знак Делимого(" << (int)*this << ") не равен знаку Делителя ("
        << (int)M << ") " << "\nИнвертируем значение полученного регистра reg_1\n";
        std::cout << "~ " << saved_reg_1.binary << " = " << reg_1.binary << '\n';
    }
}

if (show_div){
    std::cout << '\n' << std::setw(16) << "Остаток(Регистр A)";
    std::cout << std::setw(2) << " | ";
    std::cout << std::setw(24) << "Частное(Регистр reg_2)\n";
    std::cout << A.binary << " | " << reg_1.binary << '\n';
    std::cout << "\nРезультат деления: " << (int)reg_1 << " (" << reg_1.binary << ")";
    std::cout << "\nОстаток деления: " << (int)A << " (" << reg_1.binary << ")\n";
    std::cout << (int)(*this) << " = " << (int)reg_1 << " * " << (int)M << " + " << int(A) << "\n\n";
}

```



```

    return reg_1;
}

bool operator==(const Register& reg_1, const Register& reg_2){
    return reg_1.binary == reg_2.binary;
}

bool operator==(const Register& reg_1, const int& num){
    return (int)reg_1 == num;
}

Register Register::operator<<(const Register& reg2) const{
    Register shifted;
    shifted.setNumber((int)(*this) << reg2.number);
    return shifted;
}

Register Register::operator<<(int num) const{
    Register shifted;
    shifted.setNumber((int)(*this) << num);
    return shifted;
}

Register Register::operator>>(const Register& reg_2) const{
    Register shifted;
    shifted.setNumber((int)(*this) >> reg_2.number);
    return shifted;
}

Register Register::operator>>(int num) const{
    Register shifted;
    shifted.setNumber(this->number >> num);
    return shifted;
}

Register::operator int() const {
    int intNum = 0;
    intNum -= this->binary[0] * (int)pow(2, N - 1);
    for(int i = N - 1; i > 0; --i){
        if (this->binary[i] == 0){
            continue;
        }
        intNum += pow(2, N-i-1);
    }
    return intNum;
}

std::ostream& operator<< (std::ostream &out, const Register& reg) {
    for(int i = 0; i < Register::N; ++i){

```

```

        out << reg[i] << ' ';
        if (i == 7){
            out << '\n';
        }
    }
    out << '\n';
    return out;
}

std::ostream& operator<< (std::ostream &out, const std::vector<int>& bitsArr) {
    for(int i = 0; i < bitsArr.size(); ++i){
        out << bitsArr[i];
        if (i == 7){
            out << '\n';
        }
    }
    return out;
}

std::istream& operator>> (std::istream &in, Register& reg){
    int num;
    in >> num;
    Register newRegister(num);
    reg = newRegister;
    return in;
}

```

Файл main.cpp

```

#include <iostream>
#include <cmath>
#include <utility>
#include <vector>
#include "Register.h"

```

```

void handleInput();

```

```

int main() {

    Register a(-32768);
    Register b(-1);

    Register res;
    res = a / b;

    // handleInput();

    return 0;
}

```

```

void handleInput(){
    std::cout << "---Эмулятор АЛУ---\n";
    std::cout << "Выполнение заданной операции\n";
    std::cout << "Вводите исходные данные:\n";

    Register a;
    Register b;
    Register result;

    std::cout << "a = ";
    std::cin >> a;
    std::cout << "b = ";
    std::cin >> b;

    std::cout << "Операция ( + | - | * | / ) : ";
    char sign;
    std::cin >> sign;

    std::cout << "Выполнение: " << (int)a << " " << sign << " " << (int)b << "\n";
    if (sign == '+')
        result = a + b;
    else if (sign == '-')
        result = a - b;
    else if (sign == '*')
        result = a * b;
    else if (sign == '/')
        result = a / b;

    std::cout << "\nРезультат: " << (int)a << " " << sign << " " << (int)b
        << " = " << int(result) << "\n";

}

```