Лабораторная работа 5
Числа с плавающей точкой

Задание 1

Выведем на экран переменные типа **unsigned int** и **int** в том виде, в котором они хранятся в памяти компьютера.

```
1  #include <stdio.h>
2
3  int a = 101;
4  int k = 0;
5
6  int main()
7  {
8      printf("%d\n", a);
9
10     asm
11     (
12         "movl   a(%rip), %ebx\n"
13     );
14
15     for(int i = 0; i < 32; i++)
16     {
17         asm
18         (
19             "rol    $1, %ebx\n"
20             "setc   k(%rip)\n"
21         );
22
23         printf("%d", k);
24     }
25 }
```

Рисунок 1: ***ex1_1.c***

```
1  #include <stdio.h>
2
3  unsigned int a = 101;
4  int k = 0;
5
6  int main()
7  {
8      printf("%d\n", a);
9
10     asm
11     (
12         "movl   a(%rip), %ebx\n"
13     );
14
15     for(int i = 0; i < 32; i++)
16     {
17         asm
18         (
19             "rol    $1, %ebx\n"
20             "setc   k(%rip)\n"
21         );
22
23         printf("%d", k);
24     }
25 }
```

Рисунок 2: ***ex1_2.c***

```
101
00000000000000000000000001100101
```

Рисунок 3: Вывод программы ***ex1_1.c***

```
101
00000000000000000000000001100101
```

Рисунок 4: Вывод программы ***ex1_2.c***

Существуют команды, означающие циклический сдвиг:

/* ROtate Right */
ror ***<количество_сдвигов>, <назначение>***
/* ROtate Left */
rol ***<количество_сдвигов>, <назначение>***

Команда **rol** используется в строке 19 на Рис. 1 и 2. Разберёмся в том, что она делает на примере циклического сдвига влево на три бита: три старших («левых») бита «выдвигаются» из регистра влево и «вдвигаются» в него справа. При этом в флаг **CF** записывается самый последний «выдвинутый» бит (См. Рис. 5).

Поговорим о команде **setc**. Она устанавливает значение байта, если флаг CF равен единице. Таким образом, принцип работы программы можно сравнить с конвейером, на каждом шаге которого, мы сдвигаем число на один бит влево и выводим его значение на экран. Так как типы **int** и **unsigned int** имеют одинаковый размер — 4 байта. То для того,

чтобы получить его двоичное представление, нам нужно сделать 32 сдвига. Поэтому мы делаем 32 итерации в цикле в строке 15 на Рис. 1 и 2.

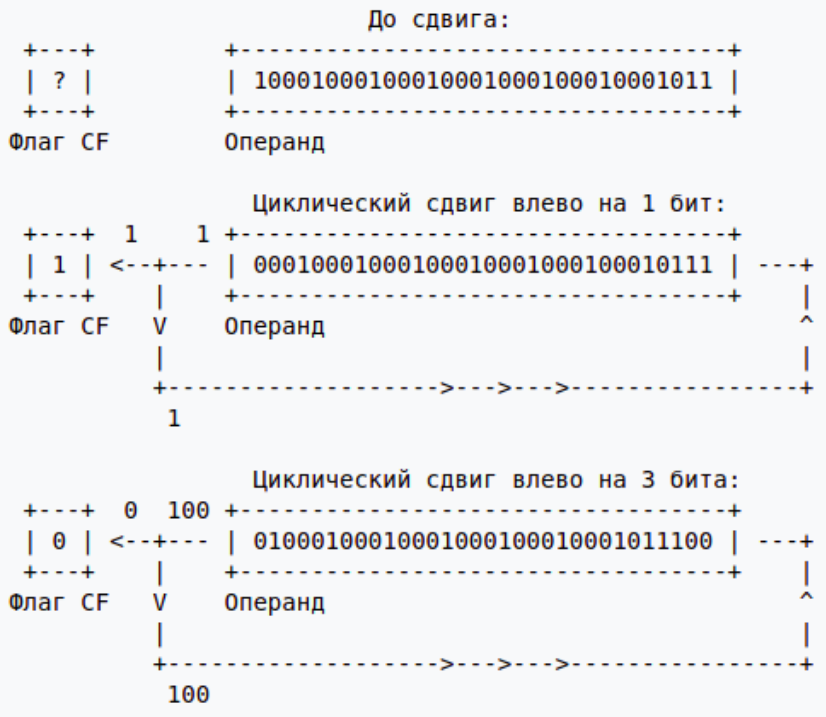


*Рисунок 5: Принцип работы команды **rol***

В обоих случаях в выводе мы получим одно и тоже число (См. Рис. 3 и 4). Убедимся в его правильности. Действительно $\mathbf{00000000000000000000000001100101_2 = 101_{10}}$.

Задание 2

Выведем на экран переменные типа **float** и **double** в том виде, в котором они хранятся в памяти компьютера (См. Рис. 7 - 17).

```c
#include <stdio.h>

int k = 0;
float a = 101.0;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 7: Программа **ex2_1.c***

```c
#include <stdio.h>

int k = 0;
double a = 101.0;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movq   a(%rip), %rbx\n"
    );

    for(int i = 0; i < 64; i++)
    {
        asm
        (
            "rol    $1, %rbx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 6: Программа **ex2_2.c***

```
101.000000
01000010110010100000000000000000
```

*Рисунок 8: Вывод программы **ex2_1.c***

```
101.000000
01000000010110010100000000000000000000000000000000000000000000
```

*Рисунок 9: Вывод программы **ex2_2.c***

```c
#include <stdio.h>

int k = 0;
float a = 0. / 0.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 10: Программа **ex2_3.c***

```c
#include <stdio.h>

int k = 0;
float a = 10. / 0.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 11: Программа **ex2_4.c***

```c
#include <stdio.h>

int k = 0;
float a = -10. / 0.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 12: Программа **ex2_5.c***

```
nan
01111111110000000000000000000000
```

*Рисунок 13: Вывод программы **ex2_3.c***

```
inf
01111111100000000000000000000000
```

*Рисунок 14: Вывод программы **ex2_4.c***

```
-inf
11111111100000000000000000000000
```

*Рисунок 15: Вывод программы **ex2_5.c***

С помощью программ **ex2_1.c** и **ex2_2.c** (См. Рис. 6, 7) мы получили представления числе типа **float double** в памяти компьютера. Как можно видеть из Рис. 8 и 9 числа данных типов имеют разную длинну. Это происходит из-за того, что они имеют разный объём: **float** — 4 байта (32 бита), **double** — 8 байт (64 бита). Поэтому в циклах для их вывода разное количество итераций. Получить **NaN** можно, например, поделив ноль на ноль, что мы и сделали в программе **ex2_3.c** (См. Рис. 10). Вывод данной программы на Рис. 13. Положительная и отрицательная бесконечности получаются делением положительного и отрицательно чисел на ноль соответственно (См. Рис. 11, 12). Результаты на Рис. 14 и 15.

```c
#include <stdio.h>

int k = 0;
float a = 0.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 16: Программа **ex2_6.c***

```c
#include <stdio.h>

int k = 0;
float a = -0.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 17: Программа **ex2_7.c***

```
0.000000
00000000000000000000000000000000
```

*Рисунок 18: Вывод программы **ex2_6.c***

```
-0.000000
10000000000000000000000000000000
```

*Рисунок 19: Вывод программы **ex2_7.c***

Теперь получим положительный и отрицательный нули в программах **ex2_6.c** и **ex2_7.c** (См. Рис. 16, 17). Выводы соответствующих программ на Рис. 18 и 19.

Разберёмся в том, как в компьютере представляются различные числа. Числа можно выражать в стандартной общепринятой экспоненциальной форме:

$$n = f \times 10^{e}$$

где ***f*** называется **мантиссой**, а ***e*** - **экспонентой**. Компьютерная версия такого представления называется представлением числа с **плавающей точкой**.

Область значений определяется по числу разрядов в экспоненте, а точность — по числу разрядов в мантиссе. Действительные числа отличаются от чисел с плавающей точкой. Первое отличие последних состоит в том, что их нельзя использовать для записи чисел из областей 1, 3, 5 и 7 (См. Рис. 20). Если в результате арифметической операции получится число из областей 1 или 7, то произойдет **ошибка переполнения**, и результат будет неверным. Причина — ограничение области значений чисел в данном представлении. Точно так же нельзя выразить результат из областей 3 или 5. Такая ситуация называется **ошибкой**

потери значимости. Эта ошибка менее серьезна, чем ошибка переполнения, поскольку часто нуль является вполне удовлетворительным приближением для чисел из областей 3 или 5.

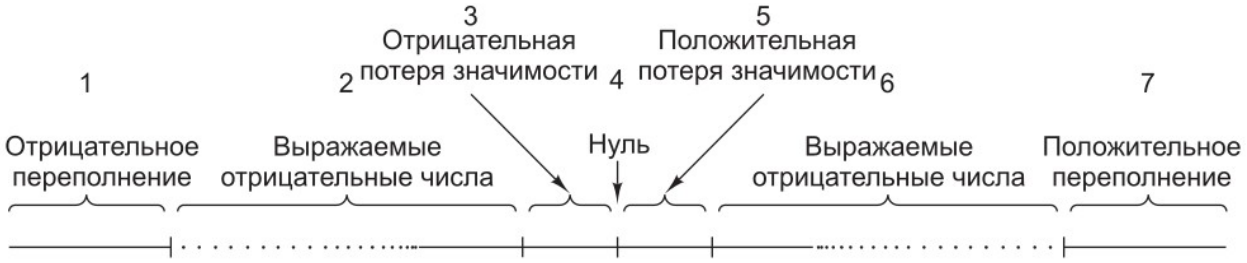


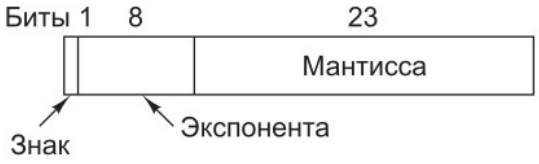
Рисунок 20: Области чисел с плавающей точкой на действительной оси

Второе важное отличие чисел с плавающей точкой от действительных чисел — их плотность. Между любыми двумя действительными числами x и y существует другое действительное число независимо от того, насколько близко к y расположено число x. Действительные числа формируют континуум. Числа с плавающей точкой континуума не формируют. Если полученное число нельзя выразить с помощью используемого представления, нужно брать ближайшее представимое число. Такой процесс называется **округлением**.

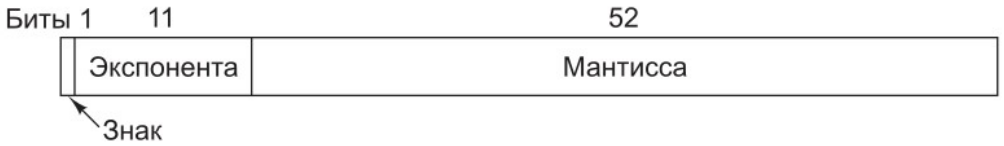
Промежутки между смежными числами, которые можно выразить в представлении с плавающей точкой, в областях 2 и 6 (См. Рис. 20) не постоянны. Однако если промежутки между числом и его соседом выразить как процентное отношение от этого числа, большой разницы в промежутках не будет. Другими словами, **относительная погрешность**, полученная при округлении, приблизительно равна и для малых, и для больших чисел.

До 80-х годов каждый производитель поддерживал собственный формат чисел с плавающей точкой. Все они отличались друг от друга. Чтобы изменить эту ситуацию, в конце 70-х годов институт **IEEE** учредил комиссию по стандартизации арифметики с плавающей точкой. В результате в 1985 году вышел стандарт **IEEE 754 [IEEE, 1985]**. В настоящее время большинство процессоров (в том числе **Intel**, **SPARC** и **JVM**) содержат команды с плавающей точкой, которые соответствуют этому стандарту.

Стандарт **IEEE 754** определяет три формата: с одинарной точностью (32 бита), с удвоенной точностью (64 бита) и с повышенной точностью (80 бит). Формат с повышенной точностью предназначен для уменьшения ошибки округления. Он применяется главным образом в арифметических устройствах с плавающей точкой, поэтому мы не будем о нем говорить. В форматах с одинарной и удвоенной точностью используются основание степени 2 для мантисс и смещенная экспонента (См. Рис. 21).



а



б

*Рисунок 21: Форматы стандарта **IEEE** чисел с плавающей точкой: а) одинарная точность, б) удвоенная точность*

Оба формата начинаются со знакового бита для всего числа: 0 указывает на положительное число, 1 — на отрицательное. Затем следует смещенная экспонента. Для формата одинарной точности смещение равно 127, а для формата удвоенной точности — 1023. Минимальная (0) и максимальная (255 и 2047) экспоненты не используются для нормализованных чисел. У них есть специальное предназначение, о котором мы поговорим позже. В конце идут мантиссы по 23 и 52 бита соответственно.

Нормализованная мантисса начинается с двоичной точки, за которой следует 1 бит, а затем — остаток мантиссы. Так как 1 бит перед мантиссой сохранять не нужно, а можно просто считать, что он там есть, то стандарт определяет мантиссу следующим образом. Она состоит из неявного бита, который всегда равен 1, неявной двоичной точке, за которыми идут 23 или 52 произвольных бита. Если все 23 или 52 бита мантиссы равны 0, то мантисса имеет значение 1,0. Если все биты мантиссы равны 1, то числовое значение мантиссы немного меньше, чем 2,0. Во избежание путаницы в английском языке для обозначения комбинации из неявного бита, неявной двоичной точки и 23 или 52 явных битов вместо термина мантисса (**mantissa**) используется термин значащая часть числа (**significand**). Значащая часть числа **s** всех нормализованных чисел лежит в диапазоне $\mathbf{1 \le s < 2}$. Характеристики форматов чисел с плавающей точкой даны на Рис. 22.

	Одинарная точность	**Удвоенная точность**
Количество битов в знаке	1	1
Количество битов в экспоненте	8	11
Количество битов в мантиссе	23	52
Общее число битов	32	64
Смещение экспоненты	Смещение 127	Смещение 1023
Область значений экспоненты	От −126 до +127	от −1022 до +1023
Самое маленькое нормализованное число	2^{-126}	2^{-1022}
Самое большое нормализованное число	Приблизительно 2^{128}	Приблизительно 2^{1024}
Диапазон десятичных дробей	Приблизительно от 10^{-38} до 10^{38}	Приблизительно от 10^{-308} до 10^{308}
Самое маленькое ненормализованное число	Приблизительно 10^{-45}	Приблизительно 10^{-324}

*Рисунок 22: Характеристики чисел с плавающей точкой стандарта **IEEE***

Теперь, используя все вышеперечисленные факты, мы можем ещё раз, уже более осмысленно, посмотреть на полученные нами числа (См. Рис. 8, 9, 13-15, 18, 19). Как мы можем заметить все они прекрасно соответствуют приведённой нами теории.

Задание 3

Напишем код на **C**, демонстрирующий эффект переполнения мантиссы. Это можно сделать всего одним присвоением (См. Рис. 24). Мы присвоим переменной **a** значение **2/3**. Так как оно представляет собой бесконечную периодическую дробь, то оно никак не может быть записано в мантиссу, имеющую конечные размеры. Поэтому компьютер просто округляет его, что мы и можем видеть на Рис. 23.

```
0.666667
00111111001010101010101010101011
```

*Рисунок 23: Вывод программы **ex3.c***

```c
#include <stdio.h>

int k = 0;
float a = 2. / 3.;

int main()
{
    printf("%f\n", a);

    asm
    (
        "movl   a(%rip), %ebx\n"
    );

    for(int i = 0; i < 32; i++)
    {
        asm
        (
            "rol    $1, %ebx\n"
            "setc   k(%rip)\n"
        );

        printf("%d", k);
    }
}
```

*Рисунок 25: Программа **ex3.c***

```c
#include <stdio.h>

float a = 2. / 3.;
float b = 1. / 17.;
float c = 16.;

int main()
{
    if((a + b) + c == a + (b + c))
    {
        printf("Ассоциативность есть");
    }
    else
    {
        printf("Ассоциативности нет");
    }
}
```

*Рисунок 24: Программа **ex4.c***

```
Ассоциативности нет
```

*Рисунок 26: Вывод программы **ex4.c***

Задание 4

Теперь напишем код на **C**, демонстрирующий неассоциативность арифметических операций (См. Рис. 24). Как можно видеть из вывода (См. Рис. 26) программы — ассоциативность нарушается.

Задание 5

Напишем код на **C**, использующий числа с плавающей точкой (См. Рис. 27, 28). Рассмотрим все простейшие арифметические операции для чисел типов **float** и **double**.

```c
#include <stdio.h>

float a = 2.;
float b = 1.;
float c = 0.;

int main()
{
    c = a + b;
    c = a - b;
    c = a * b;
    c = a / b;
}
```

*Рисунок 27: Программа **ex5_1.c***

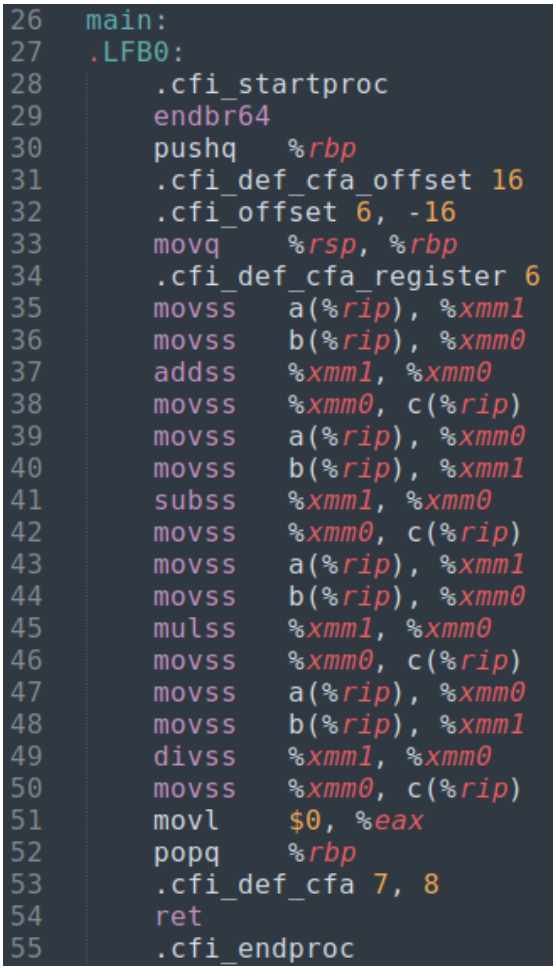
```c
#include <stdio.h>

double a = 2.;
double b = 1.;
double c = 0.;

int main()
{
    c = a + b;
    c = a - b;
    c = a * b;
    c = a / b;
}
```

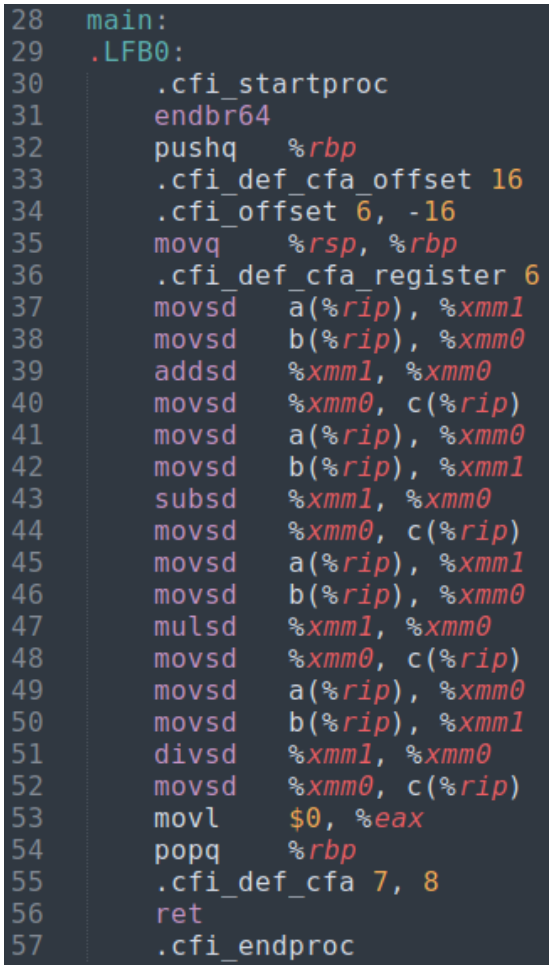
*Рисунок 28: Программа **ex5_2.c***

Посмотрим на ассемблерные листинги данных программ (См. Рис. 29, 30). Как мы можем видеть, функции main() в обоих случаях посимвольно идентичны. Отличается только объявление чисел этих типов в самом начале программ **ex5_1.s** и **ex5_2.s**. В силу вышесказанного, мы можем рассмотреть лишь один листинг.



```
26  main:
27  .LFB0:
28      .cfi_startproc
29      endbr64
30      pushq   %rbp
31      .cfi_def_cfa_offset 16
32      .cfi_offset 6, -16
33      movq    %rsp, %rbp
34      .cfi_def_cfa_register 6
35      movss   a(%rip), %xmm1
36      movss   b(%rip), %xmm0
37      addss   %xmm1, %xmm0
38      movss   %xmm0, c(%rip)
39      movss   a(%rip), %xmm0
40      movss   b(%rip), %xmm1
41      subss   %xmm1, %xmm0
42      movss   %xmm0, c(%rip)
43      movss   a(%rip), %xmm1
44      movss   b(%rip), %xmm0
45      mulss   %xmm1, %xmm0
46      movss   %xmm0, c(%rip)
47      movss   a(%rip), %xmm0
48      movss   b(%rip), %xmm1
49      divss   %xmm1, %xmm0
50      movss   %xmm0, c(%rip)
51      movl    $0, %eax
52      popq    %rbp
53      .cfi_def_cfa 7, 8
54      ret
55      .cfi_endproc
```

Рисунок 29: Фрагмент программы ***ex5_1.s***



```
28  main:
29  .LFB0:
30      .cfi_startproc
31      endbr64
32      pushq   %rbp
33      .cfi_def_cfa_offset 16
34      .cfi_offset 6, -16
35      movq    %rsp, %rbp
36      .cfi_def_cfa_register 6
37      movsd   a(%rip), %xmm1
38      movsd   b(%rip), %xmm0
39      addsd   %xmm1, %xmm0
40      movsd   %xmm0, c(%rip)
41      movsd   a(%rip), %xmm0
42      movsd   b(%rip), %xmm1
43      subsd   %xmm1, %xmm0
44      movsd   %xmm0, c(%rip)
45      movsd   a(%rip), %xmm1
46      movsd   b(%rip), %xmm0
47      mulsd   %xmm1, %xmm0
48      movsd   %xmm0, c(%rip)
49      movsd   a(%rip), %xmm0
50      movsd   b(%rip), %xmm1
51      divsd   %xmm1, %xmm0
52      movsd   %xmm0, c(%rip)
53      movl    $0, %eax
54      popq    %rbp
55      .cfi_def_cfa 7, 8
56      ret
57      .cfi_endproc
```

Рисунок 30: Фрагмент программы ***ex5_2.s***

Первое, что можно заметить при взгляде на листинг, так это появление регистров с непривычными для нас названиями - **%xmm0**, **%xmm1**. Также мы можем увидеть набор неизвестных нам команд в строчках 35 — 50 Рис. 29. Давайте разберёмся в чём же тут дело. Для этого немного углубимся в суть происходящего.

Сопроцессор — специализированный процессор, расширяющий возможности центрального процессора, но оформленный как отдельный функциональный модуль. Физически сопроцессор может быть отдельной микросхемой или может быть встроен в центральный процессор (как это делается в случае математического сопроцессора в процессорах для ПК начиная с **Intel 486DX**). Именно **математический сопроцессор**, основной задачей которого является ускорение вычислений с плавающей запятой, и является предметом нашего внимания.

Первоначально сопроцессор был с отдельным стеком. Для эффективной работы с плавающей точкой, расширение **MMX** добавило в архитектуру восемь 64-битных регистров общего пользования **MM0** — **MM7**. На самом деле, чистый **MMX** — это такая древность, которая уже не встречается. Поэтому вернёмся обратно в наше время.

SSE (англ. **Streaming SIMD Extensions**, потоковое **SIMD-расширение** процессора) — это **SIMD-** (англ. **Single Instruction**, **Multiple Data**, Одна инструкция — множество данных) набор инструкций, разработанный Intel и впервые представленный в процессорах серии Pentium III. Технология **SSE** позволяла преодолеть две основные проблемы **MMX**: при использовании **MMX** невозможно было одновременно использовать инструкции сопроцессора, так как его регистры были общими с регистрами **MMX**, и возможность **MMX** работать только с целыми числами.

В **SSE** добавлены шестнадцать (для x64) 128-битных регистров, которые называются **xmm0 - xmm15**. Каждый регистр может содержать четыре 32-битных значения с плавающей запятой одинарной точности.

Теперь приведём несколько команд, объяснив их смысл.

movss <приёмник>, <источник> – пересылка одного вещественного числа. Копирует младшие 32 бита из источника в приёмник. Если приёмник – регистр **SSE**, его старшие 96 бит обнуляются. Если приёмник – переменная в памяти, то старшие 96 бит не изменяются. Каждый из аргументов может быть либо регистром SSE, либо переменной в памяти (См. Рис. 29, стр. 35).

addss <приёмник>, <источник> – сложение одного вещественного числа. Выполняет сложение нулевых (занимающих биты 31—0) чисел с плавающей запятой в источнике и приёмнике. Результат записывается в биты 31—0 приёмника, биты 127—32 приёмника не изменяются (См. Рис. 29, стр. 37).

subss <приёмник>, <источник> – вычитание одного вещественного числа (аналогично) (См. Рис. 29, стр. 41).

mulss <приёмник>, <источник> – умножение одного вещественного числа (См. Рис. 29, стр. 45).

divss <приёмник>, <источник> – деление одного вещественного числа (См. Рис. 29, стр. 49).

В качестве дополнительных сведений опишем общий принцип работы процессора и сопроцессора. Оба устройства имеют свои раздельные системы команд и форматы обрабатываемых данных. Несмотря на то, что сопроцессор архитектурно представляет собой отдельное вычислительное устройство, он не может существовать отдельно от основного процессора. Но процессор и сопроцессор, являясь двумя самостоятельными вычислительными устройствами, могут работать параллельно. Но это распараллеливание распространяется только на выполнение команд. Оба процессора подключены к общей системной шине и имеют доступ к одной и той же информации. Инициирует процесс выборки очередной команды всегда основной процессор.

После выборки команда попадает одновременно в оба процессора. Любая команда сопроцессора имеет код операции, первые пять бит, которого имеют значение 11011. Когда код операции начинается этими битами, то основной процессор по дальнейшему содержимому кода операции выясняет, требует ли данная команда обращения к памяти. Если это так, то основной процессор формирует физический адрес операнда и обращается к памяти, после чего содержимое ячейки памяти выставляется на шину данных. Если обращение к памяти не требуется, то основной процессор заканчивает работу над данной командой (не делая попытки ее исполнения) и приступает к декодированию следующей команды из текущего входного командного потока.

Сопроцессор, определив по первым пяти битам, что очередная команда принадлежит его системе команд, начинает ее исполнение. Если команда требует операнды из памяти, то сопроцессор обращается к шине данных за чтением содержимого ячейки памяти, которое к этому моменту предоставлено основным процессором.

Задание 6

Напишем код на **C**, вычисляющий число пи по различным формулам. Так как в качестве итогового результата мы хотим получит графики зависимости посчитанного числа пи от количества итераций, то на каждой итерации мы будем записывать полученное значение в файл. Ввиду того, что полный код вышел довольно громоздким, то мы не будем его приводить здесь. Ознакомиться с ним можно, посмотрев файл **ex6.c**. Для большей наглядности сами алгоритмы и соответствующие им формулы всё же имеет смысл представить далее (См. Рис. 31 — 35 и Формулы 1 - 5).

$$\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\ldots=\frac{\pi}{4}$$

Формула 1: Ряд Лейбница

```
// 1) Ряд Лейбница

double PI_1 = 0.;
for(int i = 0; i < n; i++)
{
    PI_1 = PI_1 + 4 * (pow(-1, i) / (2 * i + 1));
}

printf("1) %g\n", PI_1);
```

Рисунок 31: Вычисление числа пи с использованием формулы Лейбница

$$\frac{\pi}{2}=\frac{2}{1}\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\frac{8}{7}\frac{8}{9}\ldots$$

Формула 2: Ряд Валлиса

```
// 2) Ряд Валлиса

double PI_2 = 2.0;
for(int i = 1; i < n; i++)
{
    PI_2 = PI_2 *(pow(2 * i, 2) / ((2 * i - 1) * (2 * i + 1)));
}

printf("2) %g\n", PI_2);
```

Рисунок 32: Вычисление числа пи с использованием формулы Валлиса

$$\frac{2}{\pi}=\frac{\sqrt{2}}{2}\frac{\sqrt{2+\sqrt{2}}}{2}\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}\ldots$$

Формула 3: Ряд Виета

```
// 3) Ряд Виета

double PI_3 = (sqrt(2.) / 2) * sqrt(2. + sqrt(2.)) / 2;
double k = sqrt(2. + sqrt(2. + sqrt(2.)));
for(int i = 0; i < n; i++)
{
    PI_3 = PI_3 * k / 2;
    k = sqrt(2. + k);
}

PI_3 = 2 / PI_3;

printf("3) %g\n", PI_3);
```

Рисунок 33: Вычисление числа пи с использованием формулы Виета

$$\pi=2\sqrt{3}\sum_{k=0}^{\infty}\left(\frac{(-1)^k}{3^k(2k+1)}\right)$$

Формула 4: Ряд Мадхавы

```
// 4) Ряд Мадхавы

double PI_4 = 0.;
for(int i = 0; i < n; i++)
{
    PI_4 = PI_4 + pow(-1, i) / (pow(3, i) * (2 * i +1));
}

PI_4 = 2 * sqrt(3) * PI_4;

printf("4) %g\n", PI_4);
```

Рисунок 34: Вычисление числа пи с использованием формулы Мадхавы

$$\pi=3+\frac{4}{2*3*4}-\frac{4}{4*5*6}+\frac{4}{6*7*8}\ldots$$

Формула 5: Ряд Нилаканта

```
// 5) Ряд Нилаканта

double PI_5 = 3.;
int num = 1;
for(int i = 2; i < n;)
{
    PI_5 = PI_5 + (4. * num) / (i * (i + 1) * (i + 2));
    num = num * (-1);
    i = i + 2;
}

printf("5) %g\n", PI_5);
```

Рисунок 35: Вычисление числа пи с использованием формулы Нилаканта

Теперь представим несколько графиков в наиболее интересных масштабах (См. Рис. 36 — 38). Графики были получены с помощью языка **python** с использованием библиотеки **matplotlib**. Весь код можно найти в файле **graphs_of_pi.py**. Все файлы, необходимые для построения зависимостей также приложены и имеют названия **file_1.txt — file_5.txt**.

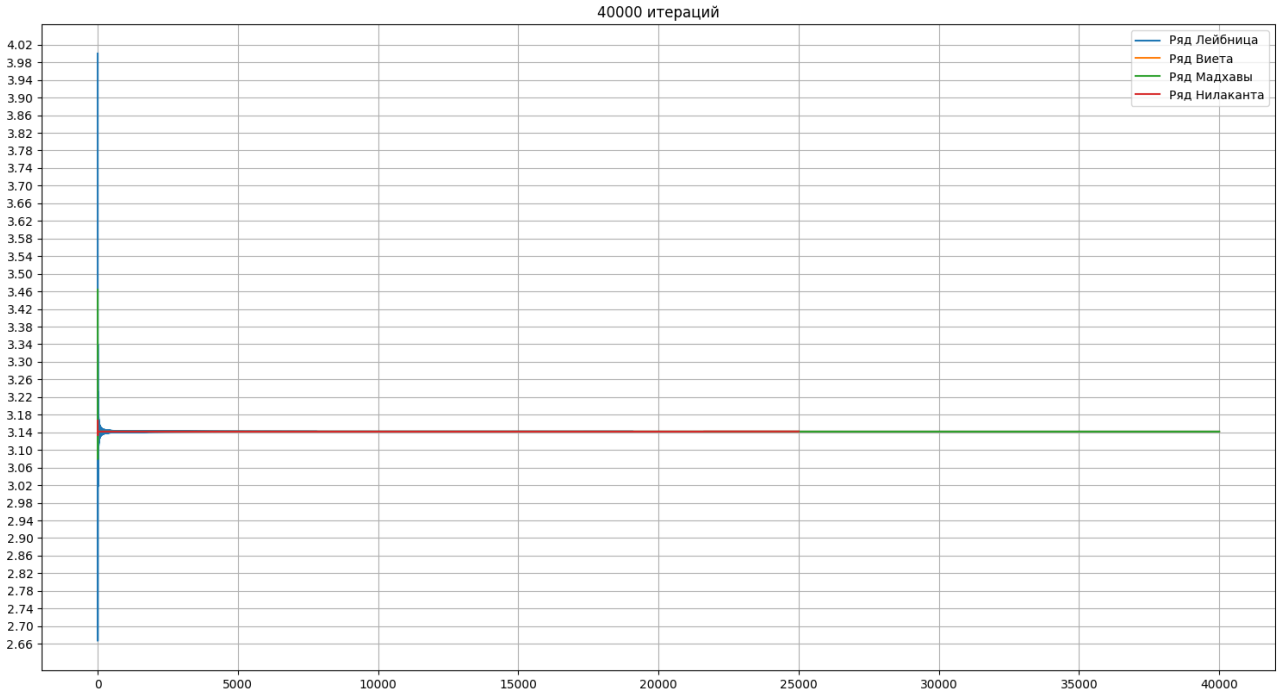


Рисунок 36: График зависимости значения вычисленного числа пи от количества итераций (для 40 000 итераций)

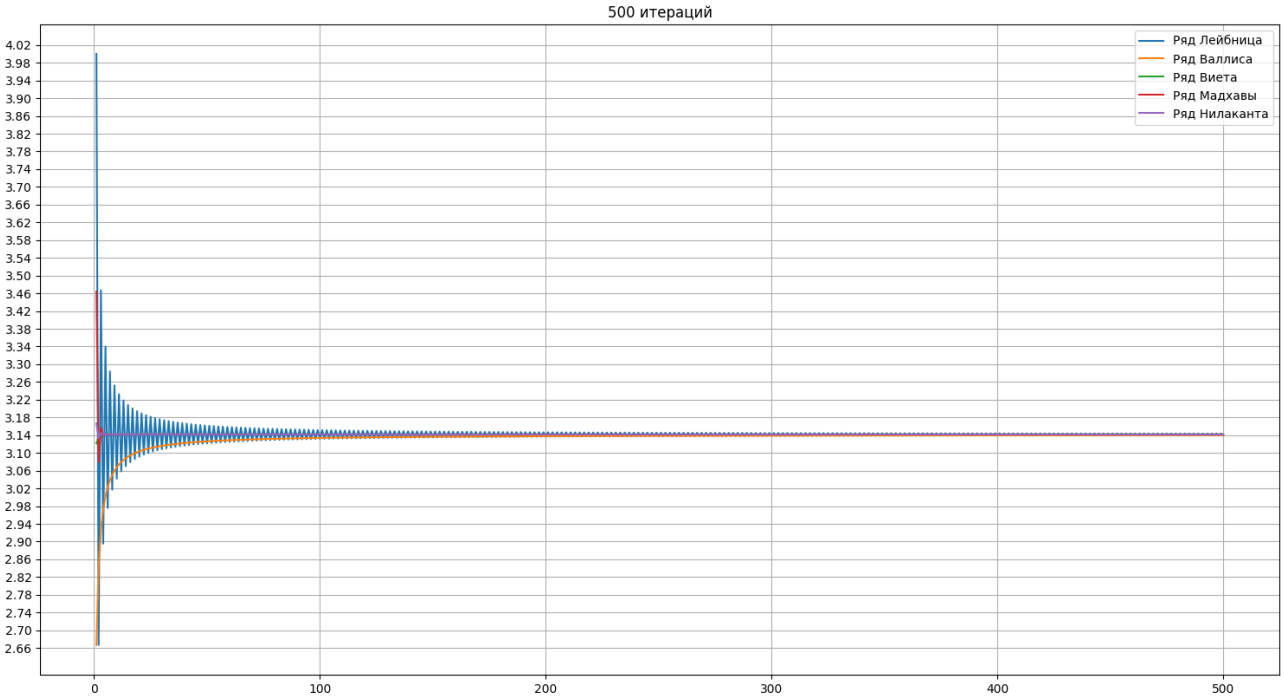


Рисунок 37: График зависимости значения вычисленного числа пи от количества итераций (для 500 итераций)

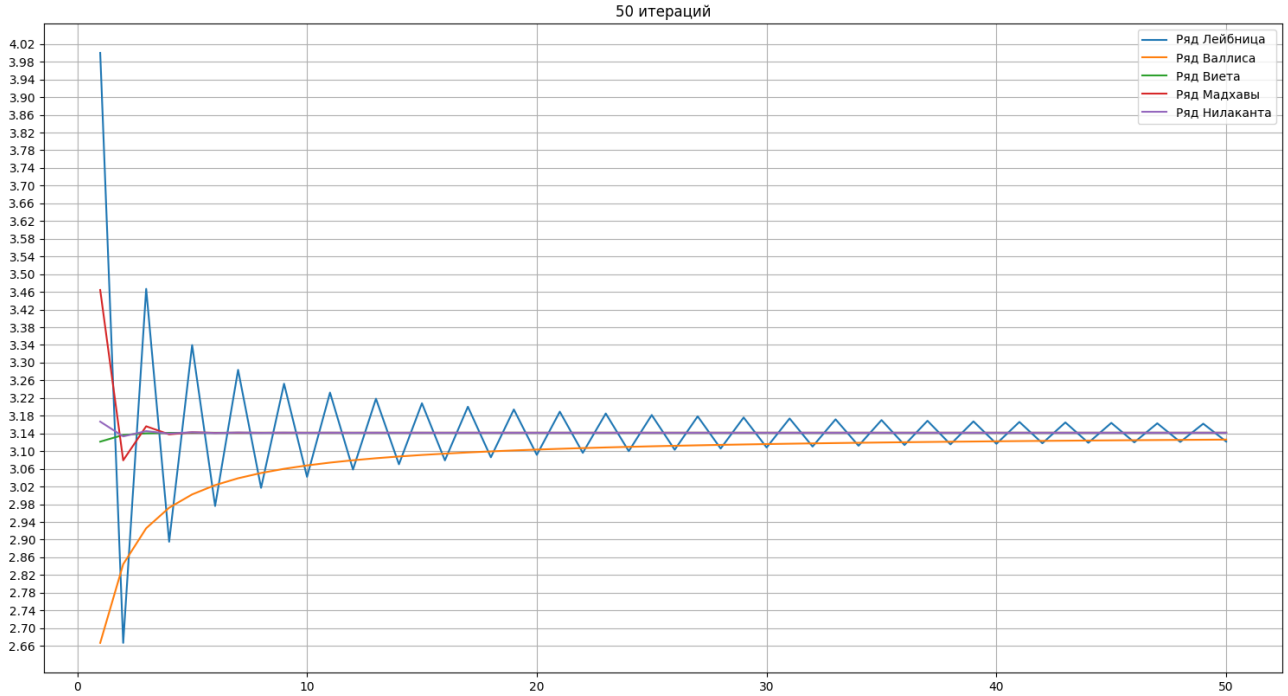


Рисунок 38: График зависимости значения вычисленного числа пи от количества итераций (для 50 итераций)

Проанализируем полученное. Рисунки 36 и 37 были приведены скорее для того чтобы дать читателю более общее представление о происходящем. Обратим внимание на Рисунок 38. Как можно видеть ряды Виета, Мадхавы сходятся к постоянному значению (в пределах точности наших вычислений) до десятой итерации. Это значение равно **3.14159**, что является верным результатом. Для того чтобы ряд Лейбница дошёл до этого значения необходимо более 50000 итераций.

Ряд Нилаканта показывает очень хорошие приближения, сравнимые с рядами Виета и Мадхавы на небольшом количестве итераций — порядка **1500**. После этого ошибка начинает медленно расти и на **100000** итераций она уже составляет уже около трёх десятитысячных.

С помощью ряда Валлиса также можно получить неплохое приближение числа пи. На **23000** итераций мы получаем число **3.14156**. Но где-то в этом же диапазоне итераций алгоритм начинает вести себя нестабильно и ряд расходится на знаковые бесконечности. Именно поэтому данный ряд не изображён на Рис. 36.

Задание 7

Традиционные проблемы, связанные с числами с плавающей точкой, - переполнение, потеря значимости (или антипереполнение) и неинициализированные числа. Поэтому помимо нормализованных чисел в стандарте предусмотрено еще 4 типа чисел (См. Рис. 39).

Проблема возникает в том случае, если абсолютное значение (модуль) результата меньше самого маленького нормализованного числа с плавающей точкой, которое можно представить в этой системе. Раньше аппаратное обеспечение действовало одним из двух способов: либо устанавливало результат на 0, либо вызывало ошибку потери значимости. Ни один из этих двух способов не является удовлетворительным, поэтому в стандарт IEEE введены **ненормализованные** (денормализованные или субнормальные) числа. Эти числа имеют экспоненту 0 и мантиссу, представленную следующими 23 или 52 битами. Неявный бит 1 слева от двоичной точки превращается в 0. Ненормализованные числа можно легко отличить от нормализованных, поскольку у последних не может быть нулевой экспоненты.

Самое маленькое нормализованное число с одинарной точностью содержит 1 в экспоненте и 0 в мантиссе и представляет 1.0×2^{-126} . Самое большое ненормализованное число содержит 0 в экспоненте и все единицы в мантиссе и представляет примерно

$0.9999999 \times 2^{-127}$, то есть почти то же самое число. Следует отметить, что это число содержит только 23 бита значимости, а все нормализованные числа — 24 бита.

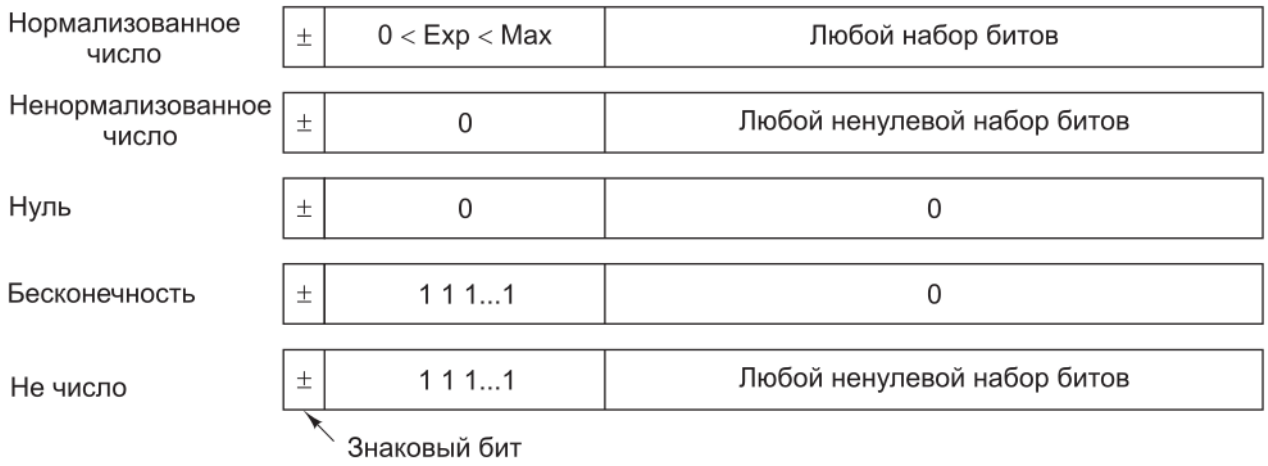


*Рисунок 39: Числовые типы стандарта **IEEE***

По мере уменьшения результата при дальнейших вычислениях экспонента по-прежнему остается равной 0, а первые несколько битов мантиссы превращаются в нули, что уменьшает и значение, и число значимых битов мантиссы. Самое маленькое ненулевое ненормализованное число содержит 1 в крайнем правом бите, а все остальные биты равны 0. Экспонента представляет 2^{-127} , а мантисса — 2^{-23} , поэтому значение равно 2^{-150} . Такая схема предусматривает постепенное исчезновение значимых разрядов, а не перескакивает на 0, когда результат не удается выразить в виде нормализованного числа.

В этой схеме присутствуют два нуля, положительный и отрицательный, определяемые по знаковому биту. Оба имеют экспоненту 0 и мантиссу 0. Здесь тоже бит слева от двоичной точки по умолчанию равен 0, а не 1.

Теперь напишем код на **C**, демонстрирующий эффект антипереполнения.

```c
#include <stdio.h>
#include <math.h>


float a = pow(2, -49);
float b = pow(2, -100);
float c = 0.;

double d = pow(2, -500);
double e = pow(2, -574);
double g = 0.;

int main()
{
    c = a * b;
    g = d * e;

    printf("%e\n", c);
    printf("%e\n", g);
}
```

*Рисунок 40: Программа **ex7.c** (нижняя граница значимости)*

```c
#include <stdio.h>
#include <math.h>


float a = pow(2, -50);
float b = pow(2, -100);
float c = 0.;

double d = pow(2, -500);
double e = pow(2, -575);
double g = 0.;

int main()
{
    c = a * b;
    g = d * e;

    printf("%e\n", c);
    printf("%e\n", g);
}
```

*Рисунок 41: Программа **ex7.c** (потеря значимости)*

```
1.401298e-45
4.940656e-324
```

Рисунок 43: Нижняя граница значимости

```
0.000000e+00
0.000000e+00
```

Рисунок 42: Потеря значимости

Для этого можно просто перемножить два очень маленьких числа, что мы и сделали в программе **ex7.c**. И как мы можем видеть из Рис. 42 и 43, нижнюю границу значимости для типа **float** можно приближённо оценить числами 2^{-150} или 10^{-45}. Для типа **double** - 2^{-1075} или 10^{-324}. Дополнительная теория и более подробные пояснения изложены в задании 2, а также минимальные ненормализованные числа для обоих типов приведены на Рис. 22.

Задание 8

Теперь проверим нижнюю границу нормализованных чисел (См. Рис. 44). Также как и ранее соответствующая теория обсуждалась в задании 2.

```c
#include <stdio.h>
#include <math.h>
#include <xmmintrin.h>

float a = pow(10, -7);
float b = pow(10, -30);
float c = 0.;

double d = pow(10, -300);
double e = pow(10, -7);
double g = 0.;

int main()
{
    // отключаем ненормализованные числа
    _mm_setcsr(_mm_getcsr() | 0x8040);

    c = a * b;
    g = d * e;

    printf("Нормализованный float:  %e\n", c);
    printf("Нормализованный double: %e\n", g);

    // включаем ненормализованные числа
    _mm_setcsr(_mm_getcsr() & ~0x8040);

    a = pow(10, -15);
    b = pow(10, -30);

    d = pow(10, -300);
    e = pow(10, -23);

    c = a * b;
    g = d * e;

    printf("Ненормализованный float:  %e\n", c);
    printf("Ненормализованный double: %e\n", g);
}
```

*Рисунок 44: Программа **ex8_1.c***

```
Нормализованный float:  1.000000e-37
Нормализованный double: 1.000000e-307
Ненормализованный float:  1.401298e-45
Ненормализованный double: 9.881313e-324
```

*Рисунок 45: Вывод программы **ex8_1.c***

Как мы можем видеть на Рис. 45, представленные значения соответствуют приведённым ранее (См. Рис. 22).

Поговорим о некоторых новых для нас понятиях. Компиляторы **C** включают ненормализованное равное нулю (**Denormals are zero - DAZ**) и флаги сброса в ноль (**Flush to zero - FTZ**) для **SSE** по умолчанию для уровней оптимизации выше -O0. Эффект **DAZ** заключается в том, чтобы обрабатывать ненормализованные входные аргументы для операций с плавающей запятой как ноль, а эффект **FTZ** - возвращать ноль вместо ненормализованного числа с плавающей запятой для операций, которые могли бы привести к ненормализованному числу с плавающей запятой, даже если входные аргументы сами не являются ненормализованными.

Команда «**_mm_setcsr(_mm_getcsr() | 0x8040)**» выключает ненормализованные числа, включая одновременно и **DAZ**, и **FTZ** (См. Рис 44 стр. 16). Команда «**_mm_setcsr(_mm_getcsr() & ~0x8040)**» совершает обратную операцию (См. Рис. 44 стр. 25).

Теперь мы попробуем сравнить производительность процессора при работе с нормализованными и ненормализованными числми. Для этого мы сделаем большое

количество арифметических операций с числами, находящимися на нижней границе значимости нормализованных и ненормализованных чисел. Приведем полный код программы **ex8_2.c** далее.

```c
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <xmmintrin.h>

double nenorm_num = pow(10, -300);
double norm_num = pow(10, -300);


void time_tester()
{
    nenorm_num = nenorm_num * pow(10, -22);
    printf("nenorm_num = %e\n", nenorm_num);

    // Начинаем измерение времени
    clock_t begin_1 = clock();

    for(int i = 0; i < 1000000000; i++)
    {
        if(i % 2 == 0)
        {
            nenorm_num = nenorm_num / 10;
        }
        else
        {
            nenorm_num = nenorm_num * 10;
        }
    }

    // Заканчиваем измерение времени
    clock_t end_1 = clock();
    double time_spent_1 = (double)(end_1 - begin_1) / CLOCKS_PER_SEC;
    printf("1) time_spent = %f\n", time_spent_1);

    // отключаем ненормализованные числа
    _mm_setcsr(_mm_getcsr() | 0x8040);

    norm_num = norm_num * pow(10, -6);
    printf("norm_num = %e\n", norm_num);

    // Начинаем измерение времени
    clock_t begin_2 = clock();

    for(int i = 0; i < 1000000000; i++)
    {
        if(i % 2 == 0)
        {
            norm_num = norm_num / 10;
        }
        else
        {
            norm_num = norm_num * 10;
        }
    }

    // Заканчиваем измерение времени
    clock_t end_2 = clock();
    double time_spent_2 = (double)(end_2 - begin_2) / CLOCKS_PER_SEC;
    printf("2) time_spent = %f\n", time_spent_2);
}

int main()
{
    time_tester();
}
```

*Рисунок 46: Программа **ex8_2.c***

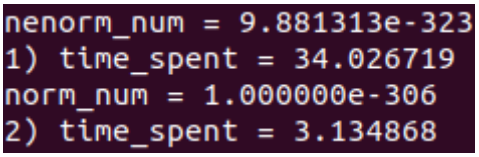


Рисунок 47: Вывод программы
ex8_2.c

Приведём несколько пояснений по коду **ex8_2.c**. Мы будем замерять время, работая с числами типа **double**. Сначала мы создаём число, находящееся на нижней границе значимости ненормализованных чисел (См. Рис. 46 стр. 12) и выводим его на экран (См. Рис. 47 стр. 1). После начинается основной блок с измерением времени работы. В переменную **begin_1** записывается время начала, в переменную **end_1** — время конца вычислений (См. Рис. 46 стр. 16 и 31). Между ними расположен основной цикл. Внутри него мы делим и умножаем на 10 наше число, всё время находясь на нижней границе значимости ненормализованных чисел. В цикле мы делаем один миллиард итераций, для того чтобы получить более менее усреднённое и осмысленное значение в итоге. Далее мы находим разность между **end_1** и **begin_1** и записываем значение в **time_spent_1**. Далее мы его выводим. Потом мы отключаем ненормализованные числа (См. Рис. 46 стр. 36) и повторяем то же самое с нормализованными числами. Поэтому вторая часть программы **ex8_2.c** не нуждается в столь же подробных пояснениях, так как она аналогична первой части.

Теперь посмотрим на полученные результаты (См. Рис. 47). Миллиард итераций с ненормализованными числами занял примерно 34 секунды, когда такое же количество итераций с нормализованными числами занимает примерно 3 секунды (На Рис. 47 приведены более менее усреднённые значения, полученные по нескольким запускам). Из этого мы можем сделать вывод о том, что производительность процессора при работе с ненормализованными числами примерно на порядок меньше, чем при работе с нормализованными числами.

Список используемой литературы

https://ru.wikibooks.org/wiki/%D0%90%D1%81%D1%81%D0%B5%D0%BC%D0%B1%D0%BB%D0%B5%D1%80_%D0%B2_Linux_%D0%B4%D0%BB%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%81%D1%82%D0%BE%D0%B2_C#%D0%90_%D1%81%D1%82%D0%BE%D0%B8%D1%82_%D0%BB%D0%B8?

https://cs.mipt.ru/wp/wp-content/uploads/2018/09/07-assembly.pdf

http://av-assembler.ru/source/flags.php

https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BD%D0%BE%D1%80%D0%BC%D0%B0%D0%BB%D0%B8%D0%B7%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B5_%D1%87%D0%B8%D1%81%D0%BB%D0%B0

https://star-wiki.ru/wiki/Denormal_number

https://coderoad.ru/5248915/%D0%92%D1%80%D0%B5%D0%BC%D1%8F-%D0%B2%D1%8B%D0%BF%D0%BE%D0%BB%D0%BD%D0%B5%D0%BD%D0%B8%D1%8F-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D1%8B-C

Таненбаум Э., Остин Т. Архитектура Компьютера. 6-е изд. - СПБ.: Питер, 2013. - 816 с.: ил.