ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
МОСКОВСКИЙ ФИЗИКО-ТЕХНИЧЕСКИЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ФАКУЛЬТЕТ АЭРОКОСМИЧЕСКИХ ТЕХНОЛОГИЙ

Лабораторная 5
Числа с плавающей точкой (C)

Рогозин Владимир
Группа Б03-106

Пункт 1: Хранение целых чисел в памяти

С помощью побитовых сдвигов через флаг CF выведем сначала представление в памяти целого и целого беззнакового чисел. Код и результат программы представлены ниже

```
#include <stdio.h>

char i[32] = {0};
char ui[32] = {0};

int main() {
        int a = -44;
        unsigned int b = 44;
        asm(    "movq   $0, %r8\n"
                "\tleaq i(%rip), %r9\n"
                "\tleaq ui(%rip), %r10\n"
                "\tmovl -8(%rbp), %ebx\n"
                "\tmovl -4(%rbp), %edx\n"
                "Cycle:\n"
                "\tcmp  $32, %r8\n"
                "\tjge  End\n"
                "\tleaq (%r9, %r8, 1), %r13\n"
                "\tleaq (%r10, %r8, 1), %r14\n"
                "\trol  $1, %ebx\n"
                "\trclb $1, (%r13)\n"
                "\trol  $1, %edx\n"
                "\trclb $1, (%r14)\n"
                "\taddq $1, %r8\n"
                "\tjmp  Cycle\n"
                "End:\n"
                                                        );
        for (int j = 0; j < 32; ++j)
                printf("%d", i[j]);
        printf("\n");
        for (int j = 0; j < 32; ++j)
                printf("%d", ui[j]);
        printf("\n");
        return 0;
}
```

Рис. 1: Программа для вывода побитового представления

```
clear@DESKTOP-FOMMSSB:~/assembler_3sem/Lab5$ ./Lab5.out
11111111111111111111111111010100
00000000000000000000000000101100
```

Рис. 2: Результат работы программы

Видно, что положительные целые и числа типа *unsigned int* хранятся в обычной двоичной записи, а отрицательные целые числа хранятся в виде дополнительного кода (инвертированная двоичная запись к которой прибавляется 1).

Пункт 2: Хранение чисел с плавающей точкой и специальных символов

Теперь попробуем таким способом вывести *float*, *double*, ± 0 и $\pm\infty$. Сперва числа с плавающей точкой, воспользуемся программой из прошлого пункта, только вместо целых чисел будем подавать нецелые (для компьютера это все также набор единиц и нулей, так что главное начать сдвигать с нужной ячейки и нужное количество раз, то есть программа выведет корректное представление нецелых чисел в памяти).

```
float a = -2.34;
double b = 2.34;
```

Рис. 3: *float* и *double* для вывода программе

```
11000000000101011100001010001111
0100000000000010101110000101000111101011100001010001111010111000
```

Рис. 4: *float* и *double* в памяти компьютера

Получили все так как и должно быть, первый бит отвечает за знак числа, затем 8 бит для *float* и 11 бит для *double* хранят (порядок $+127$ или $+1023$) для *float* и *double*, и мантисса 23 бита и 52 бита соответственно. Теперь то же самое только с бесконечностями.

```
float a = -2.34 / 0;
double b = 2.34 / 0;
```

Рис. 5: Создание $\pm\infty$ для вывода программе

```
11111111100000000000000000000000
0111111111110000000000000000000000000000000000000000000000000000
```

Рис. 6: $\pm\infty$ в памяти компьютера

Знак бесконечности определяется точно также как и у числа, то есть первым битом, мантисса нулевая, порядок максимально возможный, то есть все единички. То же самое сделаем для ± 0.

```
float a = -2.34 * 0;
double b = 2.34 * 0;
```

Рис. 7: Создание ± 0 для вывода программе

```
10000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Рис. 8: ± 0 в памяти компьютера

Первый бит все также указывает на знак нуля, мантисса состоит полностью из нулей, порядок тоже состоит только из нулей. Последним делом в этом пункте рассмотрим специальный символ NaN. Посмотрим как он хранится в памяти.

```c
float a = -0.f / 0;
double b = 0.f / 0;
```

Рис. 9: Создание NaN для вывода программе

```
11111111110000000000000000000000
1111111111111000000000000000000000000000000000000000000000000000
```

Рис. 10: NaN в памяти компьютера

Видим, что NaN только один (первый разряд всегда единица), порядок как и в случае бесконечностей максимальный, то есть забор из единичек, но мантисса NaN отличается тем, что она ненулевая. В данном примере в мантиссе только на первом месте стоит 1, но вообще в ней может быть что угодно, отличное от нуля.

Пункт 3: Переполнение мантиссы

При сложении чисел с различными порядками сначала нужно привести их у одному порядку. Для этого мантисса числа с меньшим порядком сдвигается на нужное количество знаков вправо. Отсюда получаем, что при сложении чисел сильно разного порядка возможно переполнение мантиссы – явление, при котором мантисса числа не влезает в отведённые ей 23 (52) бита.

```c
double a = 2e20;
double b = a + 2;
```

Рис. 11: Складываем числа сильно разного порядка

```
0100010000100101101011110001110101111000101101011000110001000000
0100010000100101101011110001110101111000101101011000110001000000
```

Рис. 12: Результат сложения; a сверху, b снизу

Видим, что b стало равно a, а не $a+2$. Мантисса вынуждена сдвинуться вправо слишком сильно, если различие больше чем на 52 двоичных порядка, то мантисса переполнится и станет нулевой, следовательно при сложении/вычитании один из операндов как бы становится нулём.

Пункт 4: Неассоциативность арифметических операций

Теперь попробуем написать код, который продемонстрирует неассоциативность арифметических операций с $float$'ами. Очевидно нужно брать большие и маленькие числа.

```c
#include <stdio.h>

int main() {
        float a = 1e30;
        float b = a;
        float c = 1.f;
        printf("%f %f\n", (a - b) + c, a + (-b + c));
        return 0;
}
```

Рис. 13: Код для демонстрации неассоциативности

```
clear@DESKTOP-FOMMSSB:~/assembler_3sem/Lab5$ gcc Test.c -o Test.out
clear@DESKTOP-FOMMSSB:~/assembler_3sem/Lab5$ ./Test.out
1.000000 0.000000
```

Рис. 14: Результат работы программы

Видим, что результаты действительно разные, хотя математика говорит, что должны быть одинаковыми.

Пункт 5: Числа с плавающей точкой в листинге

Посмотрим на то, как производятся арифметические операции с *float*'ами и *double*'ами. Ниже представлены две программы и их ассемблерные листинги. Можем видеть, что команды те же что и для целых чисел, но с другими суффиксами. К тому же сами числа хранятся в специальных 128-битных регистрах %xmm0, %xmm1 и т.д. Всего таких регистров 8 штук.

```
#include <stdio.h>

int main() {
        float a = 5.5;
        float b = 2.25;
        float c = a + b;
        float d = a - b;
        float e = a * b;
        float f = a / b;
        return 0;
}
```

(a) Арифм. операции *float*

```
#include <stdio.h>

int main() {
        double a = 5.5;
        double b = 2.25;
        double c = a + b;
        double d = a - b;
        double e = a * b;
        double f = a / b;
        return 0;
}
```

(b) Арифм. операции *double*

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movss   .LC0(%rip), %xmm0
        movss   %xmm0, -24(%rbp)
        movss   .LC1(%rip), %xmm0
        movss   %xmm0, -20(%rbp)
        movss   -24(%rbp), %xmm0
        addss   -20(%rbp), %xmm0
        movss   %xmm0, -16(%rbp)
        movss   -24(%rbp), %xmm0
        subss   -20(%rbp), %xmm0
        movss   %xmm0, -12(%rbp)
        movss   -24(%rbp), %xmm0
        mulss   -20(%rbp), %xmm0
        movss   %xmm0, -8(%rbp)
        movss   -24(%rbp), %xmm0
        divss   -20(%rbp), %xmm0
        movss   %xmm0, -4(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

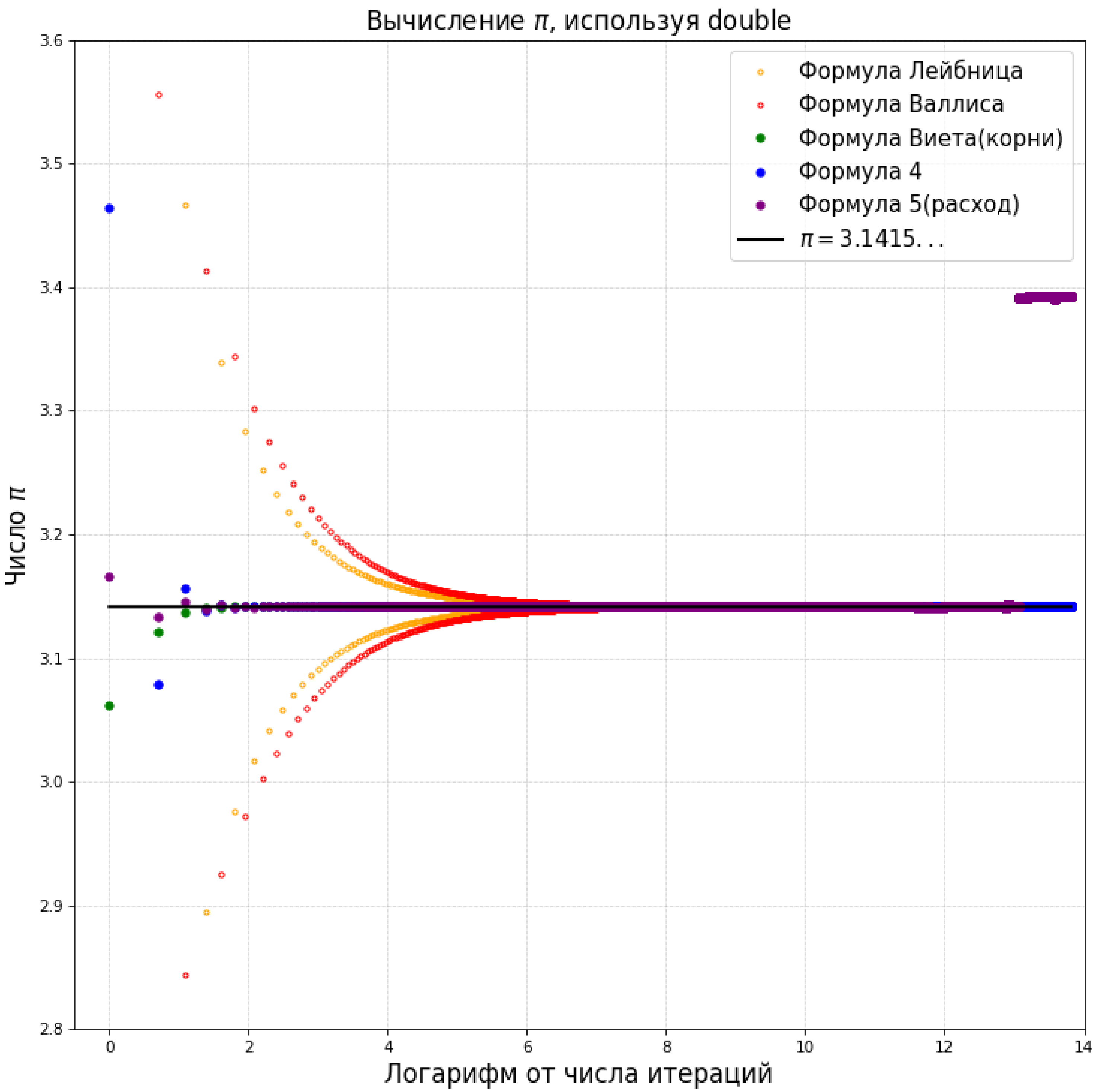
(a) Листинг *float*

```
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movsd   .LC0(%rip), %xmm0
        movsd   %xmm0, -48(%rbp)
        movsd   .LC1(%rip), %xmm0
        movsd   %xmm0, -40(%rbp)
        movsd   -48(%rbp), %xmm0
        addsd   -40(%rbp), %xmm0
        movsd   %xmm0, -32(%rbp)
        movsd   -48(%rbp), %xmm0
        subsd   -40(%rbp), %xmm0
        movsd   %xmm0, -24(%rbp)
        movsd   -48(%rbp), %xmm0
        mulsd   -40(%rbp), %xmm0
        movsd   %xmm0, -16(%rbp)
        movsd   -48(%rbp), %xmm0
        divsd   -40(%rbp), %xmm0
        movsd   %xmm0, -8(%rbp)
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

(b) Листинг *double*

Пункт 6: Вычисление числа π различными формулами

С помощью различных формул будем рассчитывать число π, запоминая значение на каждой итерации. По этим данным построим график числа π от логарифма итераций. Для $float$'ов график получается такой же, на таком масштабе различия не видны.



Из графика видно, что быстрее всего сходятся к π формулы 3, 4. Формула 5 начинает расходиться в районе $e^{11,5}$ итерации, сильно начинает отличаться (уже в первом знаке после запятой) примерно на e^{13} итерации.

Пункт 7: Антипереполнение и денормализованные числа

Теперь найдём минимальное по модулю значение для *float* и *double*, для этого вычтем минимальное нормализованное число из следующего за ним, получим минимальное по модулю число. У минимального нормализованного числа должна быть нулевая мантисса и порядок 1 (с учётом вычета 127 или 1023). У минимального денормализованного числа должен быть нулевым порядок (с учётом вычета 127 или 1023) и минимальная ненулевая мантисса. В результате программы верхнее число – минимальное нормализованное число, нижнее – минимальное денормализованное. Для *float* это 2^{-126} и 2^{-149} соответственно, для *double* – 2^{-1022} и 2^{-1074}. Антипереполнение получим в следующем пункте, когда научимся отключать денормализованные числа.

```
// Минимальная степень нормализованного числа 127 - 1 = 126, мантисса нулевая
float a = pow(2, -126);
// b теперь разность минимальных нормализованных чисел
float b = pow(2, -126) * (1 + pow(2, -23)) - pow(2, -126);
```

Рис. 17: Получаем минимальное значение *float*

```
00000000100000000000000000000000
00000000000000000000000000000001
```

Рис. 18: Минимальные значения *float* в памяти

```
// Минимальная степень нормализованного числа 1023 - 1 = 1022, мантисса нулевая
double a = pow(2, -1022);
// b теперь разность минимальных нормализованных чисел
double b = pow(2, -1022) * (1 + pow(2, -52)) - pow(2, -1022);
```

Рис. 19: Получаем минимальное значение *double*

```
0000000000010000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
```

Рис. 20: Минимальные значения *double* в памяти

Пункт 8: DAZ, FTZ и сравнение производительности

DAZ и FTZ – флаги которые влияют на появление/исчезновение денормализованных чисел. Включённый DAZ говорит обрабатывать денормализованные входные аргументы для операций, принимающих на вход числа с плавающей точкой, как 0, FTZ – возвращать денормализованное значение как 0 для операций, где оно может появиться. Посмотрим на различия при наличии/отсутствии флагов.

```
double c = pow(2, -1022);
double a = 1;
double b = 1;
// Сначала с выключенными флагами DAZ и FTZ
// а теперь денормализованное число
a = (1 + pow(2, -50)) * pow(2, -1022) - c;
// Включаем FTZ
_mm_setcsr(_mm_getcsr() | 0x8000);
// Теперь из-за флага FTZ результат операции становится 0
b = (1 + pow(2, -50)) * pow(2, -1022) - c;
```

Рис. 21: Программа с включённым FTZ

```
0000000000000000000000000000000000000000000000000000000000000100
0000000000000000000000000000000000000000000000000000000000000000
```

Рис. 22: Переменные a сверху и b снизу

Как и ожидалось, в результате операции вычитания получается денормализованное число, которое, если включён флаг FTZ, обнуляется. В ином случае оно остаётся ненулевым.

```
double c = pow(2, -1033);
double a = 1;
double b = 1;
// Сначала с выключенными флагами DAZ и FTZ
// а не равно нулю
a = pow(2, 1003) * c;
// Включаем DAZ
_mm_setcsr(_mm_getcsr() | 0x0040);
// Теперь из-за флага DAZ в с передаётся как 0
b = pow(2, 1003) * c;
```

Рис. 23: Программа с включённым DAZ

```
0011111100001000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

Рис. 24: Переменные a сверху и b снизу

Здесь, при взведённом флаге DAZ, в операцию умножения денормализованное число передаётся как 0, в результате умножения получается 0. В ином случае, как и должно быть в обычном режиме, получаем число, отличное от нуля.

Далее, выясним влияет ли отключение денормализованных чисел на скорость операций с ними. Проверим для сложения *addss* и умножения *divss*. Будем делать 20 итераций по 200 000 операций для каждого режима. Результаты представлены на картинках ниже.

```
denormals are zero (0): 154 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 153 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 153 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 154 us; denormals are not zero(9.9904e-38): 154 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 152 us
denormals are zero (0): 152 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 152 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 153 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 155 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 150 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 157 us
denormals are zero (0): 150 us; denormals are not zero(9.9904e-38): 151 us
denormals are zero (0): 151 us; denormals are not zero(9.9904e-38): 154 us
```

Рис. 25: Операция сложения *addss*

```
denormals are zero (0): 501 us; denormals are not zero(5e-40): 743 us
denormals are zero (0): 506 us; denormals are not zero(5e-40): 736 us
denormals are zero (0): 501 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 506 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 507 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 504 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 504 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 501 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 506 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 613 us; denormals are not zero(5e-40): 736 us
denormals are zero (0): 501 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 513 us; denormals are not zero(5e-40): 740 us
denormals are zero (0): 503 us; denormals are not zero(5e-40): 747 us
denormals are zero (0): 501 us; denormals are not zero(5e-40): 742 us
denormals are zero (0): 508 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 501 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 502 us; denormals are not zero(5e-40): 735 us
denormals are zero (0): 500 us; denormals are not zero(5e-40): 737 us
denormals are zero (0): 710 us; denormals are not zero(5e-40): 1078 us
denormals are zero (0): 503 us; denormals are not zero(5e-40): 740 us
```

Рис. 26: Операция деления *divss*

Как можем заметить, отключение денормализованных чисел ускоряет операцию деления, и никак не влияет на сложение чисел с плавающей точкой. Для операций вычитания и умножения ситуация аналогична сложению и делению соответственно.