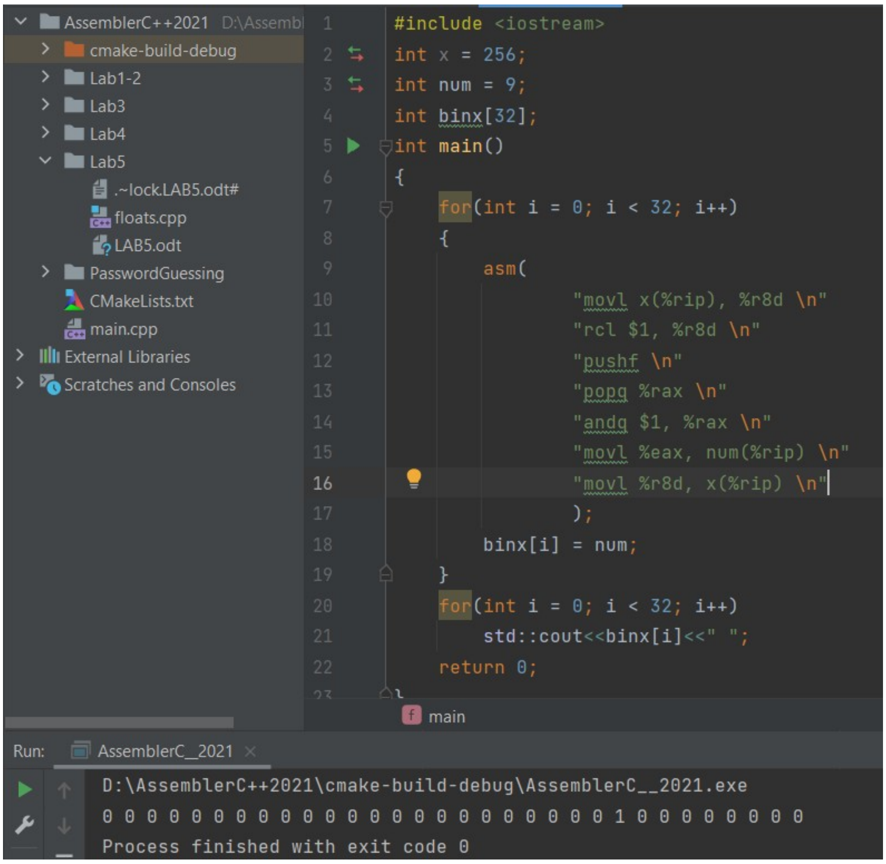
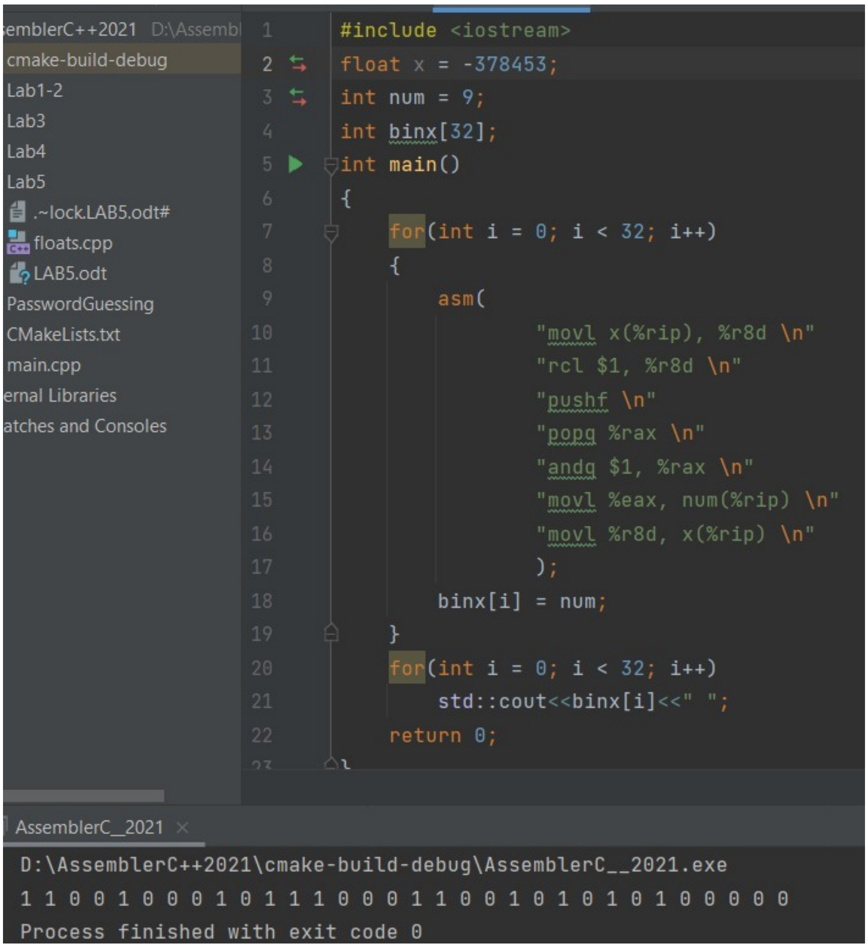
Хранение int > 0



Хранение int < 0



unsigned int выглядит также как int>0

Просто для примера:

float x = 1245.563345 вид: 0 1 0 0 0 1 0 0 1 0 0 1 1 0 1 1 1 0 1 1 0 0 1 0 0 0 0 0 0 1 1 1

±0 выглядит как 32 нуля независимо от знака(как и должно быть), если он int, а если float, то есть еще -0

-0 имеет вид: 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 0 0 0 0 0 0

+∞ имеет вид: 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

-∞ имеет вид 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

NaN имеет вид: 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Значения получены с помощью двоичного представления результатов следующих операций: 1.1/0.   -1.1/0.   1.1/0. - 1.1/0

Переполнение мантиссы:

float x = 0,2 вид: 0 0 1 1 1 1 1 0 0 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1 0 1

Неассоциативность арифметических операций (пример):

```
float a = pow( _X: 10,  _Y: 10);
x  = (a+1) - a;
y = (a - a) + 1;
std::cout<<x<<"\n"<<y<<"\n";
```

```
0.00000
1.00000
```

В первом случае 10^{10}+1 округлилось до 10^{10}, а во втором все понятно. Кстати без скобочек все работает также, потому что в любом случае они выполняются слева направо.

Арифметические операции с float и double.

```c
#include <stdio.h>
float x=1.2, y=3.4, z=5.6;
int main()
{
    x = y + z;
    y = x - y;
    z = x / y;
    z = x * y;
}
```

```asm
.cfi_startproc
endbr64
pushq   %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq    %rsp, %rbp
.cfi_def_cfa_register 6
movss   y(%rip), %xmm1
movss   z(%rip), %xmm0
addss   %xmm1, %xmm0
movss   %xmm0, x(%rip)
movss   x(%rip), %xmm0
movss   y(%rip), %xmm1
subss   %xmm1, %xmm0
movss   %xmm0, y(%rip)
movss   x(%rip), %xmm0
movss   y(%rip), %xmm1
divss   %xmm1, %xmm0
movss   %xmm0, z(%rip)
movss   x(%rip), %xmm1
movss   y(%rip), %xmm0
mulss   %xmm1, %xmm0
movss   %xmm0, z(%rip)
movl    $0, %eax
popq    %rbp
.cfi_def_cfa 7, 8
ret
```

Особенности:

Используется тип ss(Scalar-Single) для float sd(Scalar-Double) для double соответственно, отдельные регистры %xmm0...%xmm7

Объявление переменных различных типов

int

```asm
x:
        .long   1
        .globl  y
        .align 4
        .type   y, @object
        .size   y, 4
```

float

```asm
x:
        .long   1067030938
        .globl  y
        .align 4
        .type   y, @object
        .size   y, 4
```

double

```asm
x:
        .long   858993459
        .long   1072902963
        .globl  y
        .align 8
        .type   y, @object
        .size   y, 8
```

Графики числа π от числа итераций

Код

```
float pi1=0;
float pi2=1;
float pi3=0;
float pi4=3;
outfile << fixed;
outfile.precision( prec: 20);
for(int i=0; i<N; i++)
{
    if(i % 2 == 0) //ЧЕТНЫЕ
    {
        pi1 += 1/(2*float(i)+1);
        pi2 *= (float(i)+2)/(float(i)+1);
        pi4 += 4/(2*float(i)+2)/(2*float(i)+3)/(2*float(i)+4);
    }
    else //НЕЧЕНТНЫЕ
    {
        pi1 -= 1/(2*float(i)+1);
        pi2 *= (float(i)+1)/(float(i)+2);
        pi4 -= 4/(2*float(i)+2)/(2*float(i)+3)/(2*float(i)+4);
    }

    pi3 += pow( x: -1, i)/(pow( x: 3, i)*(2*float(i)+1));

    outfile<<i<<" "<<pi1*4<<" "<<pi2*2<<" "<<""<<pi3*2*sqrt( x: 3)<<" "<<pi4<<"\n";
```

Количество элементов N =1000000, используемые формулы указаны ниже. Ось X прологарифмированная.

$$\frac{1}{1}-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+\frac{1}{9}-\cdots=\frac{\pi}{4}$$ – pi1

$$\frac{\pi}{2}=\frac{2}{1}\cdot\frac{2}{3}\cdot\frac{4}{3}\cdot\frac{4}{5}\cdot\frac{6}{5}\cdot\frac{6}{7}\cdot\frac{8}{7}\cdot\frac{8}{9}\cdots$$ – pi2

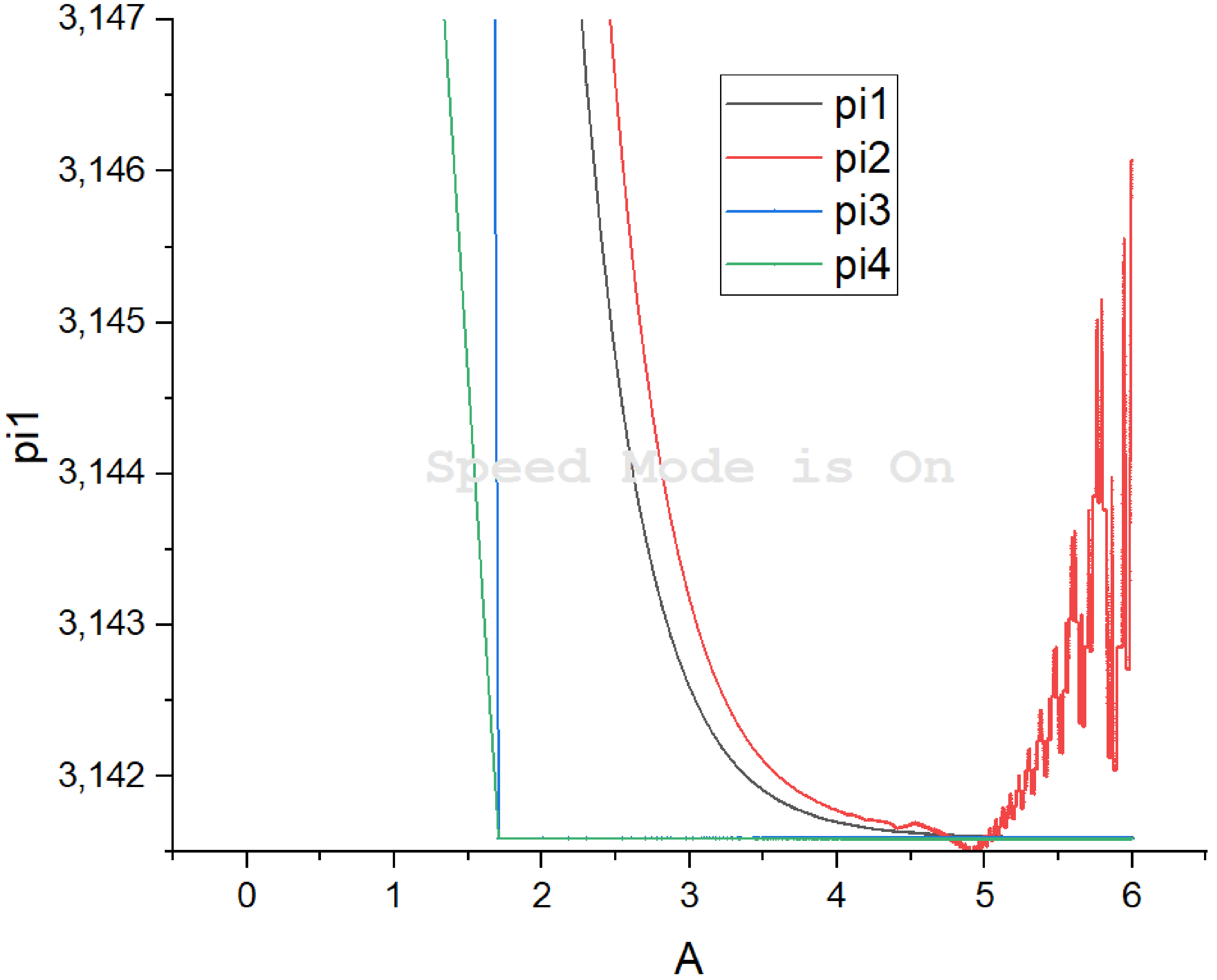
$$\frac{2}{\pi}=\frac{\sqrt{2}}{2}\cdot\frac{\sqrt{2+\sqrt{2}}}{2}\cdot\frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2}\cdots$$

– pi3

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

pi4

$$\pi = 3 + 4/(2*3*4) - 4/(4*5*6) + 4/(6*7*8) - 4/(8*9*10) + 4/(10*11*12) - (4/(12*13*14)$$



Ну по графику понятно, что самые точные это формулы pi3 и pi4. Pi2 вообще дает примерно похожие показания на очень узком интервале. Pi1 увеличивает точность с ростом итераций.

Среднее арифметическое с SSE и без

(Объяснение после картинок)

CMakeLists.txt | vafelniza.cpp

```cpp
        random_device rd;
        mt19937 mt(time(0)); //rd()
        uniform_real_distribution<float> dist(0.0, 1000.0);
        for(int i=0; i<n; ++i)
            arr[i] = dist(mt);

        //Засекаем время
        auto start = chrono::high_resolution_clock::now();
        //Сумма элементов
//        for(int i=0; i<n; ++i)
//            sum+=arr[i];
        for(int i=0; i< n-4; i+=4)
        {
            tmp[0] = arr[i];
            tmp[1] = arr[i+1];
            tmp[2] = arr[i+2];
            tmp[3] = arr[i+3];
            asm(
                "movups tmp, %xmm0 \n"
                "movups suma, %xmm1 \n"
                "addps %xmm0, %xmm1 \n"
                "movups %xmm1, suma \n"
                    );
        }
        sum = suma[0]+suma[1]+suma[2]+suma[3];
        sum = sum / float(n);
        auto end = chrono::high_resolution_clock::now();

        chrono::duration<double> diff = end - start;
```

*Размеры массива меняются от 16 до 16777216*2, умножаясь на 2.*

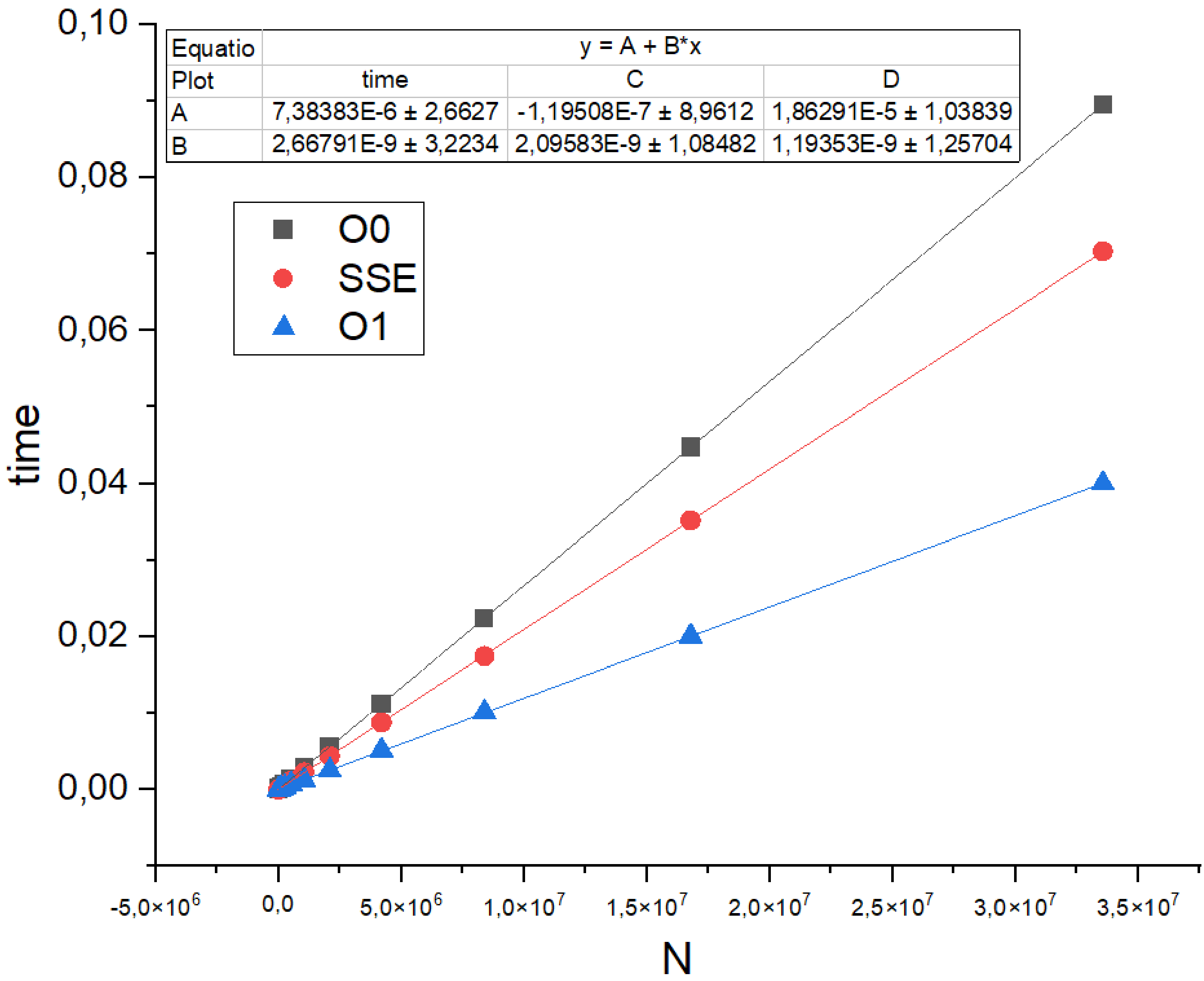
Массив заполняется случайными числами.

O0 алгоритм без оптимизации: проходим весь массив поэлементно и складываем все элементы по одному в переменную sum, потом делим на кол-во элементов.

O1 алгоритм: тоже самое с оптимизацией

SSE алгоритм: проходимся по массиву складываем каждые 4 числа в массив suma с помощью SSE, 4 значения в полученном массиве складываем и делим на кол-во элементов

На картинке ниже аппроксимировал и нашел коэффициенты угловой коэффициент SSE меньше в 1.3 раза, а не в 4. O1 быстрее O0 в 2.2 раза, скорее всего SSE замедляется из-за моего корявого алгоритма с кучей присваиваний. Был бы нормальный алгоритм, можно наверно и в 4 раза прирост получить.



Эффект антипереполнения

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{

    ofstream outfile;
    outfile.open( s: "D:/AssemblerC++2021/Lab5-6/data.txt");
    outfile << scientific;
    float a = 1;
    while(true)
    {
        a /= 10;
        outfile <<a<<"\n";
        if(a == 0)
            break;
    }
    outfile.close();
    return 0;
}
```

```
1.000000e-30
1.000000e-31
1.000000e-32
9.999999e-34
9.999999e-35
9.999999e-36
9.999999e-37
1.000000e-37
9.999999e-39
1.000000e-39
9.999946e-41
9.999666e-42
1.000527e-42
9.949219e-44
9.809089e-45
1.401298e-45
0.000000e+00
```

На картинке видно, что при делении $1.4*10^{-45}$ на 10 мы получаем 0, а не $1*10^{-46}$, это пример эффекта антипереполнения, когда результат операции с плавающей запятой становится настолько близким к нулю, что порядок числа выходит за пределы разрядной сетки и получаем машинный ноль в результате.

В пункте задан вопрос «Поддерживает ли ваш сопроцессор денормализованные числа?»

Ответ: да, т. к. у меня получилось число $\approx 1.4*10^{-45}$. И вот цитата из Википедии: «минимальное положительное денормализованное число одинарной точности — примерно $1.4 \cdot 10^{-45}$»

https://ru.wikipedia.org/wiki/Исчезновение_порядка

Получается самый маленький порядок, который он может выдать, равен 10^{-45} кстати, скриншот прекрасно демонстрирует дискретность значений, именно по этой причине у меня не строго 1.00000..., а округленное к ближайшему существующему значению. Для проверки еще делил на 2, а не на 10, получилась такая же степень.

Вот еще оставлю GitHub на всякий случай:

https://github.com/ArtemEvstafev/AssemblerC-2021.git