

Хранение `int > 0`

The screenshot shows the Visual Studio Code interface with the following components:

- File Explorer (Left):** Displays the project structure under "AssemblerC++2021". The expanded folder is "cmake-build-debug", which contains subfolders "Lab1-2", "Lab3", "Lab4", and "Lab5". Inside "Lab5", there are files ".~lock.LAB5.odt#", "floats.cpp", and "LAB5.odt". Other folders include "PasswordGuessing", and files include "CMakeLists.txt", "main.cpp", "External Libraries", and "Scratches and Consoles".
- Editor Window (Center):** Shows the source code for "main.cpp". The code includes a header guard, variable declarations, an array, and a main function that uses assembly instructions within a loop.

```
#include <iostream>

int x = 256;
int num = 9;
int binx[32];

int main()
{
    for(int i = 0; i < 32; i++)
    {
        asm(
            "movl x(%rip), %r8d \n"
            "rc1 $1, %r8d \n"
            "pushf \n"
            "popq %rax \n"
            "andq $1, %rax \n"
            "movl %eax, num(%rip) \n"
            "movl %r8d, x(%rip) \n"
        );
        binx[i] = num;
    }
    for(int i = 0; i < 32; i++)
        std::cout<<binx[i]<<" ";
    return 0;
}
```
- Run and Debug Console (Bottom):** Shows the command used to run the program: "D:\AssemblerC++2021\cmake-build-debug\AssemblerC_2021.exe". Below the command, it displays the output as a binary string: "0 1 0 0 0 0 0 0 0 0". At the bottom, it states "Process finished with exit code 0".

Хранение $\text{int} < 0$

```
AsmblC_2021  D:\AsmblC_2021\cmake-build-debug 1  #include <iostream>
cmake-build-debug 2  float x = -378453;
Lab1-2 3  int num = 9;
Lab3 4  int binx[32];
Lab4 5  int main()
Lab5 6  {
7  for(int i = 0; i < 32; i++)
8  {
9      asm(
10         "movl x(%rip), %r8d \n"
11         "rcl $1, %r8d \n"
12         "pushf \n"
13         "popq %rax \n"
14         "andq $1, %rax \n"
15         "movl %eax, num(%rip) \n"
16         "movl %r8d, x(%rip) \n"
17         );
18         binx[i] = num;
19     }
20     for(int i = 0; i < 32; i++)
21         std::cout<<binx[i]<<" ";
22     return 0;
23 }
```

AsmblC_2021 x

D:\AsmblC++2021\cmake-build-debug\AsmblC_2021.exe

1 1 0 0 1 0 0 0 1 0 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0

Process finished with exit code 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

$+\infty$ имеет вид: 0 1 1 1 1 1 1 1 1 0

$-\infty$ имеет вид 1 1 1 1 1 1 1 1 1 0

NaN имеет вид: 1 1 1 1 1 1 1 1 1 0

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

Переполнение мантииссы:

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.

```
#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;
}
```

```
.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

float

double

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

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

```
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 //НЕЧЕТНЫЕ
    {
        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);
    }

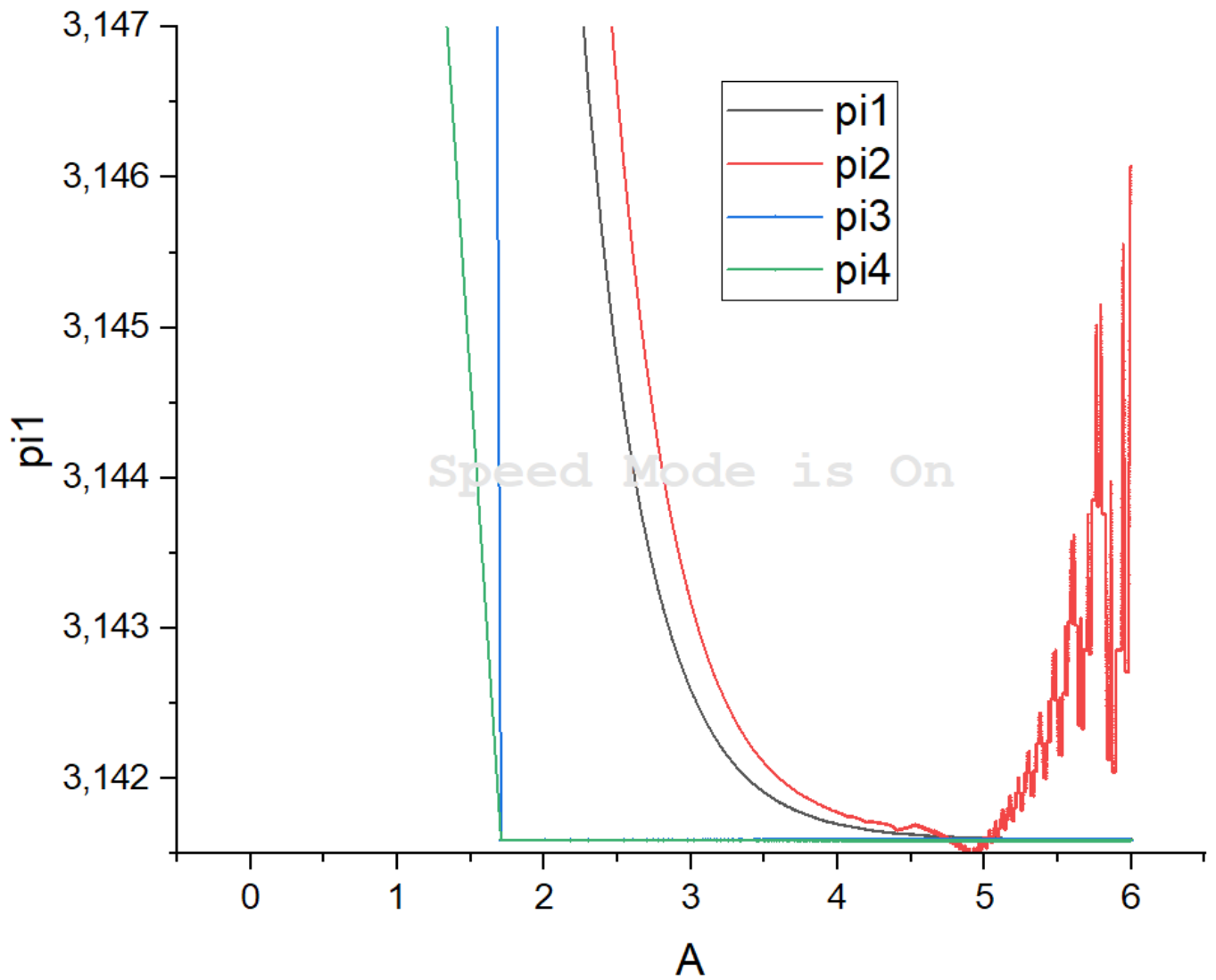
    pi3 += pow( x: -1, i)/(pow( x: 3, i)*(2*float(i)+1));

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

```

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

$$\begin{aligned}
 \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots &= \frac{\pi}{4} && \text{--- pi1} \\
 \frac{\pi}{2} &= \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \dots && \text{--- pi2} \\
 \frac{2}{\pi} &= \frac{\sqrt{2}}{2} \cdot \frac{\sqrt{2+\sqrt{2}}}{2} \cdot \frac{\sqrt{2+\sqrt{2+\sqrt{2}}}}{2} && \text{--- pi3, pi4} \\
 \pi &= 2\sqrt{3} \sum_{k=0}^{\infty} \frac{(-1)^k}{3^k (2k+1)} \\
 \pi &= 3 + 4/(2*3*4) - 4/(4*5*6) + 4/(6*7*8) - 4/(8*9*10) + 4/(10*11*12) - (4/(12*13*14)
 \end{aligned}$$



Ну по графику понятно, что самые точные это формулы π_3 и π_4 . π_2 вообще дает примерно похожие показания на очень узком интервале. π_1 увеличивает точность с ростом итераций.

Среднее арифметическое с SSE и без

(Объяснение после картинок)

```
CMakeLists.txt x vafelniza.cpp x
27 random_device rd;
28 mt19937 mt( sd: time( _Time: 0)); //rd()
29 uniform_real_distribution<float> dist( a: 0.0, b: 1000.0);
30 for(int i=0; i<n; ++i)
31     arr[i] = dist( &: mt);
32
33 //Засекаем время
34 auto start = chrono::high_resolution_clock::now();
35 //Сумма элементов
36 // for(int i=0; i<n; ++i)
37 //     sum+=arr[i];
38 for(int i=0; i< n-4; i+=4)
39 {
40     tmp[0] = arr[i];
41     tmp[1] = arr[i+1];
42     tmp[2] = arr[i+2];
43     tmp[3] = arr[i+3];
44     asm(
45         "movups tmp, %xmm0 \n"
46         "movups suma, %xmm1 \n"
47         "addps %xmm0, %xmm1 \n"
48         "movups %xmm1, suma \n"
49     );
50 }
51 sum = suma[0]+suma[1]+suma[2]+suma[3];
52 sum = sum / float(n);
53 auto end = chrono::high_resolution_clock::now();
54
55 chrono::duration<double> diff = end - start;
```

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

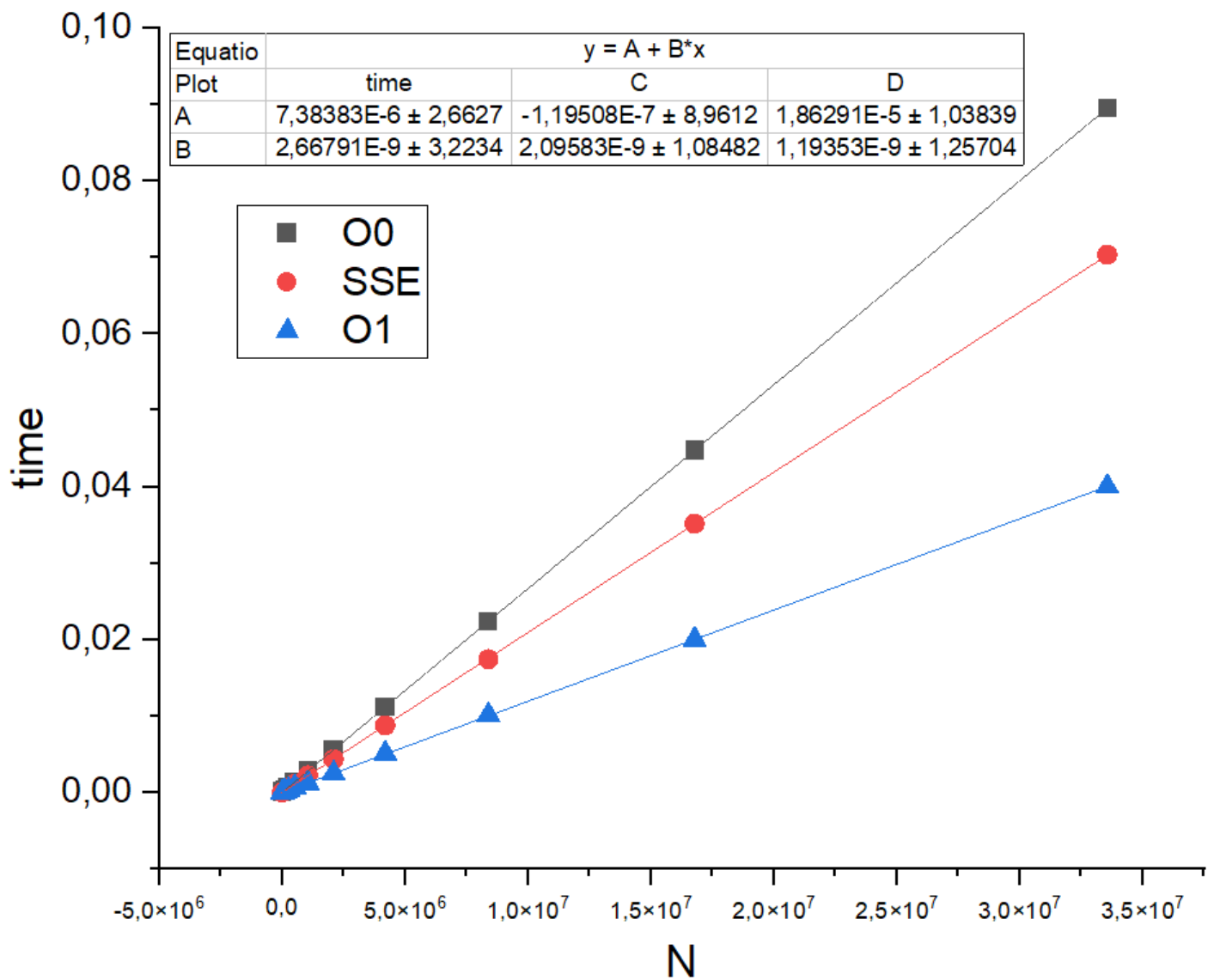
Массив заполняется случайными числами.

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

```

1  #include <iostream>
2  #include <fstream>
3  using namespace std;
4  int main()
5  {
6
7      ofstream outfile;
8      outfile.open("D:/AssemblerC++2021/Lab5-6/data.txt");
9      outfile << scientific;
10     float a = 1;
11     while(true)
12     {
13         a /= 10;
14         outfile << a << "\n";
15         if(a == 0)
16             break;
17     }
18     outfile.close();
19     return 0;
20 }

```

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

Вот самый маленький порядок 10^{-45} , который он может выдать, 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>