**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
 РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**НОВОСИБИРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Факультет информационных технологий**

**Кафедра параллельных вычислений**

**ОТЧЕТ**

**О ВЫПОЛНЕНИИ ПРАКТИЧЕСКОЙ РАБОТЫ**

ВВЕДЕНИЕ В АРХИТЕКТУРУ X86/X86-64

студентки 2 курса, группы 21205

**Евдокимовой Дарьи Евгеньевны**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

Кандидат технических наук, доцент

А.Ю.Власенко

Новосибирск 2022

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc116523379)

[ЗАДАНИЕ 3](#_Toc116523380)

[ОПИСАНИЕ РАБОТЫ 3](#_Toc116523381)

[Пошаговое описание выполненной работы 3](#_Toc116523382)

[ЗАКЛЮЧЕНИЕ 4](#_Toc116523383)

[ПРИЛОЖЕНИЕ 1. Листинг программы на языке Си 5](#_Toc116523384)

[ПРИЛОЖЕНИЕ 2. Листинг программы с оптимизацией -O0 6](#_Toc116523385)

[ПРИЛОЖЕНИЕ 3. Листинг программы с оптимизацией -O3 12](#_Toc116523386)

[ПРИЛОЖЕНИЕ 4. Таблица сравнения оптимизаций 17](#_Toc116523387)

# ЦЕЛЬ

1. Изучение программной архитектуры x86/x86-64.
2. Сгенерировать ассемблерные листинги для архитектуры x86/x86-64.
3. Проанализировать полученные листинги.

# ЗАДАНИЕ

Вариант задания: 4.

Изучить программную архитектуру x86/x86-64: набор регистров, основные арифметико-логические команды, способы адресации памяти, способы передачи управления, работу со стеком, вызов подпрограмм, передачу параметров в подпрограммы и возврат результатов, работу с арифметическим сопроцессором, работу с векторными расширениями.

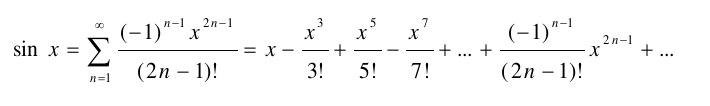
Сгенерировать листинги исходной программы с оптимизациями –O0 и –O3 и проанализировать полученные коды.

# ОПИСАНИЕ РАБОТЫ

В ходе задания использовался компьютер с архитектурой amd64, с операционной системой Ubuntu 20.04.5 LTS и процессором Intel® Core™ i3-6100U CPU @ 2.30GHz × 4.

###### Пошаговое описание выполненной работы

1. Был создан файл pract2.c
2. Была написана компьютерная программа, которая вычисляет sin(x) с помощью разложения в степенной ряд по первым N членам этого ряда (см. «Рис.1»). Код программы на языке Си представлен в Приложении 1.

  
Рис.1. Разложение синуса в ряд Тейлора

1. Были изучены основные принципы работы в языке ассемблер.
2. С помощью сайта GodBolt (URL: [https://godbolt.org](https://godbolt.org/)) были сгенерированы листинги исходной программы с оптимизациями -O0 (см. Приложение 2) и -O3 (см. Приложение 3).

**Сравнения работы оптимизаций**

См. Приложение 4.

# ЗАКЛЮЧЕНИЕ

Из приведенных описаний листингов с оптимизациями -O0 и -O3 можно сделать выводы об особенностях этих оптимизаций.

Про оптимизацию -O0 можем сказать, что каждому оператору из исходного кода на Си можно чётко поставить в соответствие набор команд из ассемеблерного листинга. Из недостатков оптимизации -O0 следует отметить, что компилятор делает много лишних действий, потому что компилятор рассматривает выражение из исходного кода независимо от сделанных им ранее действий.

Про оптимизацию -O3 можем сказать, что листинг программы с данной оптимизацией разбирать сложнее, потому что нельзя провести однозначного соответствия между ассемблерным кодом и кодом исходной программы.

# ПРИЛОЖЕНИЕ 1. Листинг программы на языке Си

1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <time.h>
4. #define PI 3.1415926535897
5. double CalcSin(double x, long long n){
6. double sinx = 0;
7. x = x \* PI / 180;
8. double sum = x;
9. for (long long i = 1; i <= 2 \* n - 1; i += 2){
10. sinx += sum;
11. sum = (sum \* x \* x \* (-1)) / ( (i + 1) \* (i + 2)) ;
12. }
13. return sinx;
14. }
15. int main(int argc, char \*\*argv){
16. struct timespec start, end;
17. clock\_gettime (CLOCK\_MONOTONIC\_RAW, &start);
18. if (argc == 1){
19. printf("Bad input. Enter x and n in command line");
20. return 0;
21. }
22. double x = atof(argv[1]);
23. long long n = atoll(argv[2]);
24. double sinx = CalcSin(x, n);
25. printf("%lf\n", sinx);
27. clock\_gettime(CLOCK\_MONOTONIC\_RAW, &end);
28. printf("Time taken: %lf sec.\n", end.tv\_sec-start.tv\_sec
29. + 0.000000001\*(end.tv\_nsec-start.tv\_nsec));
30. return 0;
31. }

# ПРИЛОЖЕНИЕ 2. Листинг программы с оптимизацией -O0

1. CalcSin:
2. pushq %rbp ##добавление адреса возврата в стек
3. movq %rsp, %rbp #сохранение адреса текущего кадра стека
4. movsd %xmm0, -40(%rbp) #в память по адресу rbp-40 кладем параметр x
5. movq %rdi, -48(%rbp) в память по адресу rbp-48 кладем параметр n
6. pxor %xmm0, %xmm0 #зануляем sinx
7. movsd %xmm0, -8(%rbp) #добавляем в стек по адресу rbp-8 значение sinx (которые равны нулю)
8. movsd -40(%rbp), %xmm1 #в регистр xmm1 кладем параметр x из rbp-40
9. movsd .LC1(%rip), %xmm0 #помещаем значения PI в регистр xmm0
10. mulsd %xmm1, %xmm0 #в x записываем результат умножения x на PI
11. movsd .LC2(%rip), %xmm1 #значения умножения (x \* PI) помещаем в регистр xmm1
12. divsd %xmm1, %xmm0 #в x записываем результат (x \* PI)/180
13. movsd %xmm0, -40(%rbp) #x помещаем в rbp-40
14. movsd -40(%rbp), %xmm0 #x, помещенный в rbp-40, помещаем в векторный регистр #xmm0
15. movsd %xmm0, -16(%rbp) #x помещаем в sum
16. movq $1, -24(%rbp) #добавление в стек счетчика i
17. jmp .L2 #прыгаем на метку .L2 (это цикл)
18. .L3:
19. movsd -8(%rbp), %xmm0 #помещаем текущее значение sinx в регистр xmm0
20. addsd -16(%rbp), %xmm0 #увеличиваем значение sinx на sum
21. movsd %xmm0, -8(%rbp) #кладем новое полученное значение sinx на стек по адресу rbp-8

#вычисление выражение со строки 13 (в исходном коде на Си, см. Приложение 1)

1. movsd -16(%rbp), %xmm0 #помещаем в регистр xmm0 sum
2. mulsd -40(%rbp), %xmm0 #умножаем sum на x
3. mulsd -40(%rbp), %xmm0 #умножаем sum на x
4. movq .LC3(%rip), %xmm1 #записываем число (-1) в вектор xmm1
5. xorpd %xmm1, %xmm0 #меняем знак на минус в значениях регистра xmm0
6. movq -24(%rbp), %rax #помещаем i в rax
7. leaq 1(%rax), %rdx #вычисляем выражение (i + 1)
8. movq -24(%rbp), %rax #помещаем в rax текущее состояние счетчика i
9. addq $2, %rax #сложение: (i + 2), результат помещаем в rax
10. imulq %rdx, %rax #знаковое умножение (i + 1) \* (i + 2)
11. pxor %xmm1, %xmm1 #зануление регистра xmm1
12. cvtsi2sdq %rax, %xmm1 #конвертируем значения int из регистра rdx в double и #помещаем результат в xmm1
13. divsd %xmm1, %xmm0
14. movsd %xmm0, -16(%rbp) #складываем полученный результат в стек по адресу rbp-16 (т.е. записываем результат в переменную sum)
15. addq $2, -24(%rbp) #увеличиваем счётчик на константу 2
16. .L2:
17. movq -48(%rbp), %rax #помещаем n в регистр rax
18. addq %rax, %rax #удваиваем значение n
19. cmpq %rax, -24(%rbp) #сравнение n (из rax)  и I (из rbp-24)
20. jl .L3 #если i < 2\*n-1, то прыгаем на метку .L3
21. movsd -8(%rbp), %xmm0 #если нет, то помещаем в регистр xmm0 полученное в цикле значение sinx
22. movq %xmm0, %rax #помещаем значение xmm0 в rax
23. movq %rax, %xmm0 #помещаем значение sinx из rax в xmm0
24. popq %rbp #очищаем rbp
25. ret #возврат из подпрограммы
26. .LC4: #строковая константа, которую используем в методе printf
27. .string "Bad input. Enter x and n in command line"
28. .LC5: #строковая константа, которую используем в методе printf
29. .string "%lf\n"
30. .LC7: #строковая константа, которую используем в методе printf
31. .string "Time taken: %lf sec.\n"
32. main:
33. pushq %rbp #сохраняем адрес базового указателя в стеке для функции main для того, чтобы вернуться назад
34. movq %rsp, %rbp #перемещаем указатель на вершину стека

#копируем значение указателя стека в указатель базы

1. subq $80, %rsp #выделяем место для локальных переменных
2. movl %edi, -68(%rbp) #записываем первый аргумент функции (т.е. argc)
3. movq %rsi, -80(%rbp) #записываем второй аргумент функции (т.е. argv)
4. leaq -48(%rbp), %rax
5. movq %rax, %rsi #clock\_gettime помещаем в rax
6. movl $4, %edi #помещаем CLOCK\_MONOTONIC\_RAW (как первый аргумент функции clock\_gettime
7. call clock\_gettime #вызов функции clock\_gettime
8. cmpl $1, -68(%rbp) #сравниваем колиечество арггументов (argc) с единицей
9. jne .L6 #если argc != 1 прыгаем на метку .L6
10. movl $.LC4, %edi #еcли argc = 1, то строковую константу помещаем в адрес edi
11. movl $0, %eax
12. call printf #вызов функции printf
13. movl $0, %eax #зануление регистра eax
14. jmp .L8 #безусловный переход на метку .L8
15. .L6: #оказались на этой метке, потому что argc != 1
16. movq -80(%rbp), %rax #копируем указатель на argv[0] в rax
17. addq $8, %rax #делаем сдвиг на 8 байт, тк хотим получить указатель на argv[1]
18. movq (%rax), %rax #получение значения аргумента argv[1]
19. movq %rax, %rdi #копирование значения argv[1] из регистра rax  в регистр rdi
20. call atof #вызов функции atof
21. movq %xmm0, %rax #результат функции atof помещаем в x (лежит в rax) из регистра xmm0 (для вещественного типа данных)
22. movq %rax, -8(%rbp) #сохранили значение x из rax по адресу rbp-8

1. movq -80(%rbp), %rax #копируем указатель на argv[0] в rax
2. addq $16, %rax #делаем сдвиг на 16 байт, тк хотим получить указатель на argv[2]
3. movq (%rax), %rax #получение значения аргумента argv[2]
4. movq %rax, %rdi #копирование значения argv[2] из регистра rax  в регистр rdi
5. call atoll #вызов функции atoll
6. movq %rax, -16(%rbp) #теперь значение n лежит в памяти по адресу rbp-16

#для вызова функции CalcSin подготваливаем регистры общего назначения (тк они видны везде)

1. movq -16(%rbp), %rdx #поместили n в rdx
2. movq -8(%rbp), %rax #поместили x в rax
3. movq %rdx, %rdi #поместили n из rdx в rdi
4. movq %rax, %xmm0 #поместили x в xmm0
5. call CalcSin #вызов функции CalcSin
6. movq %xmm0, %rax #сохраняем результат работы (т.е. sinx) функции CalcSin в регистр #rax
7. movq %rax, -24(%rbp) #добавляем полученный результат в локальный стек по адресу #rbp - 24
8. movq -24(%rbp), %rax #полученный результат перемещаем из rbp-24 в rax
9. movq %rax, %xmm0 #помещаем значение sinx  в xmm0
10. movl $.LC5, %edi берем указатель на строку с меткой .LC5 и помещаем его в регистр #edi
11. movl $1, %eax #помещаем номер файлового дескриптора в первый аргумент функции #printf
12. call printf #вызов функции printf
13. leaq -64(%rbp), %rax #подготовка к вызову фкнции clock\_gettime
14. movq %rax, %rsi
15. movl $4, %edi #помещаем CLOCK\_MONOTONIC\_RAW (как первый аргумент функции clock\_gettime
16. call clock\_gettime #вызов функции clock\_gettime

#начинается работа с функцией clock\_gettime()

1. movq -64(%rbp), %rdx #помещаем end.tv в rdx
2. movq -48(%rbp), %rax #помещаем start.tv в rax
3. subq %rax, %rdx #находим разность (end.tv – start.tv)
4. pxor %xmm1, %xmm1 #зануляем регистр xmm1
5. cvtsi2sdq %rdx, %xmm1 #конвертируем значения int из регистра rdx в double и #помещаем результат в xmm1 (т.е. нашли резузльтат в double (end.tv – start.tv)

#вывод времени (результат работы функции clock\_gettime())

1. movq -56(%rbp), %rdx #помещаем end.tv в rdx
2. movq -40(%rbp), %rax #помещаем start.tv в rax
3. subq %rax, %rdx #находим разность (end.tv – start.tv)
4. pxor %xmm2, %xmm2 #зануляем регистр xmm2
5. cvtsi2sdq %rdx, %xmm2 #конвертируем значения int из регистра rdx в double и #помещаем результат в xmm2 (т.е. нашли резузльтат в double (end.tv – start.tv)
6. movsd .LC6(%rip), %xmm0 #берем указатель на строку с меткой .LC6 и #помещаем его в xmm0
7. mulsd %xmm2, %xmm0 #умножаем 0.000000001 на (end.tv – start.tv) и #помещаем результат в xmm0
8. addsd %xmm0, %xmm1 #полученный результат из стр 104 складываем с #результатом со строки 111 и помещаем его в регистр xmm1
9. movq %xmm1, %rax #полученное значение времени добавляем в регистр rax
10. movq %rax, %xmm0 #полученное значение времени перемещаем в регистр #xmm0
11. movl $.LC7, %edi #берем указатель на строку с меткой .LC6 и помещаем его в #регистр edi
12. movl $1, %eax #помещаем номер файлового дескриптора в первый аргумент #функции printf
13. call printf #вызов функции printf
14. movl $0, %eax #зануляем регистр eax
15. .L8:
16. leave #сбросить кадр стека
17. ret #выход из подпрограммы
18. .LC1: #значения числа Пи
19. .long 1413753926
20. .long 1074340347
21. .LC2: #значения умножения (x \* PI)
22. .long 0
23. .long 1080459264
24. .LC3: #значения (-1)
25. .long 0
26. .long -2147483648
27. .long 0
28. .long 0
29. .LC6: #значения вещественного числа 0.000000001
30. .long -400107883
31. .long 1041313291

# ПРИЛОЖЕНИЕ 3. Листинг программы с оптимизацией -O3

1. CalcSin:
2. mulsd .LC1(%rip), %xmm0 #умножение PI на x
3. addq %rdi, %rdi #удвоение значения n
4. divsd .LC2(%rip), %xmm0 #выражение (PI \* x) делим на 180
5. cmpq $1, %rdi #сравнение константной единицы с 2\*n-1
6. jle .L4 #если 1 <= 2\*n-1, то прыгаем на метку .L4
7. movq .LC3(%rip), %xmm4 #если 1 > 2\*n-1, то помещаем значения из метки .LC3(%rip) в регистр xmm4
8. movapd %xmm0, %xmm1 #в sum кладем x
9. movl $1, %eax #помещаем единицу в регистр edx
10. pxor %xmm2, %xmm2 #зануление sinx
11. .L3:
12. addsd %xmm1, %xmm2 #sinx лежит в xmm2, sum лежит в xmm1. sinx = sinx + sum
13. mulsd %xmm0, %xmm1 #x лежит в xmm0. умножаем sum на x и записываем результат в sum (в регистре xmm1)
14. leaq 1(%rax), %rdx #rdx = 1 + rax, вычисляем выражение (i + 1), т.е. i лежит в rax
15. addq $2, %rax #сложение: (i + 2)
16. imulq %rax, %rdx #перемножаем: (i + 1) на (i + 2) и помещаем результат в rdx
17. pxor %xmm3, %xmm3 #зануляем регистр xmm3
18. cvtsi2sdq %rdx, %xmm3 #чтобы работать с дробными числами, конвертируем значения int из регистра rdx в double и помещаем результат в xmm3
19. mulsd %xmm0, %xmm1 #sum = sum \* x
20. xorpd %xmm4, %xmm1 #зануляем векторные значения из регистра xmm4 и помещаем результат в sum
21. divsd %xmm3, %xmm1 #делим выражение из регистра xmm1 на выражение xmm3. Помещаем результат в xmm1
22. cmpq %rdi, %rax #проверка на увеличение счётчика i
23. jl .L3 # если i < 2\*n-1, то прыгаем на метку .L3 (то есть здесь цикл)
24. movapd %xmm2, %xmm0 #если нет, то перемещаем значения из xmm2 в xmm0
25. ret #выход из подпрограммы
26. .L4:
27. pxor %xmm2, %xmm2 #зануляем значение sinx
28. movapd %xmm2, %xmm0 #загрузить значение sinx из xmm2 в xmm0
29. ret #выход из подпрограммы
30. .LC4: #строковая константа, которую используем в методе printf
31. .string "Bad input. Enter x and n in command line"
32. .LC5: #строковая константа, которую используем в методе printf
33. .string "%lf\n"
34. .LC7: #строковая константа, которую используем в методе printf
35. .string "Time taken: %lf sec.\n"
36. main:
37. pushq %rbp #сохраняем адрес базового указателя в стеке для функции main для того, чтобы вернуться назад (адрес возврата)
38. movl %edi, %ebp #перемещаем указатель на вершину стека
39. movl $4, %edi #помещаем CLOCK\_MONOTONIC\_RAW (как первый аргумент функции clock\_gettime
40. pushq %rbx
41. movq %rsi, %rbx #перемещаем второй аргумент функции в rbx
42. subq $56, %rsp #вычитаем из rsp значение 56, т.е. смещаем вершину стека на 56 байт вперед, резервируя место под локальные переменные
43. leaq 16(%rsp), %rsi #загружаем второй аргумент функции по адресу rsp-16
44. call clock\_gettime #вызов функции clock\_gettime
45. cmpl $1, %ebp #сравниваем единицу с количеством аргументов, т.е. argc лежит в ebp
46. je .L15 #если argc = 1, то прыгаем на метку .L15
47. movq 8(%rbx), %rdi #если не равно, то копируем значение из памяти по адресу rsi+8 и помещаем в регистр rdi
48. xorl %esi, %esi #заунление регистра esi
49. call strtod #вызов функции конвертирования string в double
50. movq 16(%rbx), %rdi #копируем результат atof(argv[1]) в регистр rdi
51. movl $10, %edx
52. xorl %esi, %esi #зануляем значения для второго аргумента функции
53. movsd %xmm0, 8(%rsp) #
54. call strtoll #вызов функции конвертации string в long int
55. movsd 8(%rsp), %xmm0 #копирование x из rsp+8 в xmm0
56. mulsd .LC1(%rip), %xmm0 #умножение константы из метки .LC1( PI) на x. Результат помещаем в xmm0
57. addq %rax, %rax #к значению регистра rax добавляется значение регистра rax (то есть хранится значение 2\*n-1 из цикла
58. divsd .LC2(%rip), %xmm0 #xmm0 = xmm0 / 180
59. cmpq $1, %rax #сравнение константной единицы с 2\*n-1
60. jle .L12 #если 1 <= 2\*n-1, то прыгаем на метку .L12
61. movq .LC3(%rip), %xmm4 #если 1 > 2\*n-1, то помещаем значения из метки .LC3(%rip) в регистр xmm4
62. movapd %xmm0, %xmm1 #в sum кладем x
63. movl $1, %edx
64. pxor %xmm2, %xmm2 #зануляем значения регистра xmm2
65. .L11:
66. addsd %xmm1, %xmm2 # sinx лежит в xmm2, sum лежит в xmm1. sinx = sinx + sum
67. mulsd %xmm0, %xmm1 #x лежит в xmm0. умножаем sum на x и записываем результат в sum (в регистре xmm1)
68. leaq 1(%rdx), %rcx #rdx = 1 + rax, вычисляем выражение (i + 1), т.е. i лежит в rax
69. addq $2, %rdx #сложение: (i + 2)
70. imulq %rdx, %rcx #перемножаем: (i + 1) на (i + 2) и помещаем результат в rdx
71. pxor %xmm3, %xmm3 #зануляем регистр xmm3
72. cvtsi2sdq %rcx, %xmm3 конвертируем значения int из регистра rdx в double и помещаем результат в xmm3
73. mulsd %xmm0, %xmm1 #sum = sum \* x
74. xorpd %xmm4, %xmm1 #зануляем векторные значения из регистра xmm4 и помещаем результат в sum
75. divsd %xmm3, %xmm1 #делим выражение из регистра xmm1 на выражение xmm3. Помещаем результат в xmm1
76. cmpq %rax, %rdx #сравнение значения i и 2\*n-1
77. jl .L11 #если i < 2\*n-1, то прыгаем на метку .L3 (то есть здесь цикл)
78. .L10:
79. movapd %xmm2, %xmm0 #кладем значение метода CalcSin в sinx
80. movl $.LC5, %edi #перемещаем строковую константу в edi
81. movl $1, %eax помещаем номер файлового дескриптора в первый аргумент функции printf
82. call printf #вызов функции printf
83. movl $4, %edi #помещаем CLOCK\_MONOTONIC\_RAW (как первый аргумент функции clock\_gettime
84. leaq 32(%rsp), %rsi #загрузить адрес rsp+32 в регистр rsi
85. call clock\_gettime #вызов функции clock\_gettime

#вычисления для замера времени работы программы

1. movq 40(%rsp), %rax
2. pxor %xmm0, %xmm0 #работаем с вещественным числом 0.000000001
3. subq 24(%rsp), %rax #вычисляем (end.tv\_nsec-start.tv\_nsec)
4. cvtsi2sdq %rax, %xmm0 #конвертируем (end.tv\_nsec-start.tv\_nsec) (тип int) из регистра rdx в double и помещаем результат в xmm0
5. pxor %xmm1, %xmm1
6. movq 32(%rsp), %rax
7. subq 16(%rsp), %rax #вычисляем (end.tv\_nsec-start.tv\_nsec)
8. mulsd .LC6(%rip), %xmm0 #0.000000001\*(end.tv\_nsec-start.tv\_nsec)
9. cvtsi2sdq %rax, %xmm1
10. movl $.LC7, %edi #перемещаем строковую константу в edi
11. movl $1, %eax помещаем номер файлового дескриптора в первый аргумент функции printf
12. addsd %xmm1, %xmm0 (end.tv\_sec-start.tv\_sec) \* 0.000000001\*(end.tv\_nsec-start.tv\_nsec)
13. call printf #вызов функции printf
14. .L9:
15. addq $56, %rsp
16. xorl %eax, %eax #зануляем eax (где будет храниться результат main)
17. popq %rbx #очищаем стек
18. popq %rbp #
19. ret #выход из подпрограммы
20. .L15: #здесь оказались если argc = 1
21. movl $.LC4, %edi #строковую константу помещаем в адрес edi
22. xorl %eax, %eax #зануляем регистр eax
23. call printf #вызов функции printf
24. jmp .L9 #прыгаем на метку .L9
25. .L12:
26. pxor %xmm2, %xmm2 #зануление регистра xmm2
27. jmp .L10 #прыгаем на метку .L10
28. .LC1: #значения числа Пи
29. .long 1413753926
30. .long 1074340347
31. .LC2: #значения умножения (x \* PI)
32. .long 0
33. .long 1080459264
34. .LC3: #значения (-1)
35. .long 0
36. .long -2147483648
37. .long 0
38. .long 0
39. .LC6: #значения вещественного числа 0.000000001
40. .long -400107883
41. .long 1041313291

# ПРИЛОЖЕНИЕ 4. Таблица сравнения оптимизаций

*Таблица 1*

**Сравнения оптимизаций –O0 и –O3**

|  |  |
| --- | --- |
| Оптимизация –O0 | Оптимизация –O3 |
| * 1. Заменили atof на strtod, atoll на strtoll | |
| call atof #вызов функции atof  call atoll #вызов функции atoll | call strtod #вызов функции конвертирования string в double  call strtoll #вызов функции конвертирования string в long int |
| * 1. В –O3 используются битовые операции при занулении регистра eax, в –O0 используется присваивание нуля регистру eax | |
| movl $0, %eax #зануляем регистр eax | xorl %eax, %eax #зануляем регистр eax |
| * 1. Значения помещаются в регистры xmm частями в –O0 | |
| mulsd -40(%rbp), %xmm0 #умножаем sum  на x  mulsd -40(%rbp), %xmm0 #умножаем sum на x | mulsd   %xmm0, %xmm1 |
| * 1. Обращаемся к 64-битному регистру, затем работаем с 32-битным в –O3, в –O1 – с 64-битным | |
| pushq   %rbp  movq    %rsp, %rbp | pushq %rbp  movl %edi, %ebp |
| * 1. В –O0 вызов функции clock\_gettime идёт непосредственно за добавлением в неё аргумента, в отличие от –O3 | |
| movl $4, %edi  call clock\_gettime | movl $4, %edi  ….  call clock\_gettime |
| * 1. На уровне –O0 компилятор рассматривает каждое выражение из исходного кода независимо, поэтому выполняет неразумные вещи. | |
| movq    %rax, -24(%rbp)  movq    -24(%rbp), %rax |  |

*Продолжение Таблицы 1*

|  |  |
| --- | --- |
| * 1. В –O3 происходит инлайнинг функции | |
| call CalcSin #вызов функции CalcSin | Нет вызова функции CalcSin |
| * 1. Работа с памятью | |
| Больше вычислений на стеке, следовательно, больше задействуется оперативная память, что негативно влияет на время исполнения программы. | Больше вычислений на регистрах, значит, меньше задействована оперативная память, что значительно уменьшает время работы программы.  С другой стороны, на уровне –O3 компилятору можно использовать столько памяти, сколько необходимо для максимальной оптимизации. |
| 9. Последовательность команд | |
| Компилятор выполняет прямую компиляцию, не изменяя порядка следования инструкций, и не предпринимая никаких других попыток оптимизации. | Команды переупорядочиваются с учетом информационных зависимостей таким образом, чтобы более равномерно и полно загружать вычислительные устройства процессора. |