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 ЛАБОРАТОРНОЙ РАБОТЫ

«ВВЕДЕНИЕ В АРХИТЕКТУРУ x86/x86-64»

студента Бородина Артёма Максимовича 2 курса, 19205 группы

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:
к.т.н, доцент
А.Ю. Власенко

Новосибирск 2020

СОДЕРЖАНИЕ

ЦЕЛЬ

1. Знакомство с программной архитектурой x86/x86-64.

2. Анализ ассемблерного листинга программы для архитектуры x86/x86-64.

ЗАДАНИЕ

1 вариант:

Алгоритм вычисления числа π с помощью разложения в ряд (ряд Грегори-Лейбница) по формуле Лейбница N первых членов ряда.

1. Изучить программную архитектуру x86/x86-64

2. Для программы на языке Си (из лабораторной работы 1) сгенерировать ассемблерные листинги для архитектуры x86 и архитектуры x86-64, используя различные уровни комплексной оптимизации.

3. Проанализировать полученные листинги.

4. Составить отчет по лабораторной работе.

ОПИСАНИЕ РАБОТЫ

1. Изучены основы архитектуры x86, группы регистров, доступные программисту, сопроцессор и векторные расширения.

2. Были сделаны 3 листинга: код программы, вычисляющей число π, код программы на ассемблере, скомпилированной с ключом -O0 и код программы на ассемблере, скомпилированной с ключом -O2.

3. Для просмотра кода программы на ассемблере был использован ресурс godbolt.org.

4. Компиляция обоих листингов на ассемблере проводилась с использованием ключа -m32, позволяющего компилировать код для 32битной архитектуры. Для просмотра кода с синтаксисом AT&T был использован соответствующий параметр. (листинги можно посмотреть по ссылке https://godbolt.org/z/8qss71)

5. При сравнении двух листингов на ассемблере становится заметно, что программа, скомпилированная с ключом -O2, требует гораздо меньше памяти для переменных и делает расчеты более эффективно (например, использование команды lea для расчета простых математических выражений), по сравнению с программой, скомпилированной с ключом -O0. Так же компилятор с ключом -O2 добавил проверку на 0 в функции `LeibnizFormula(int)`, что позволило избежать заход в цикл для расчета числа π, т.к. после инициализации переменных цикл пришлось бы сразу завершать.

ЗАКЛЮЧЕНИЕ

После изучения архитектуры x86, были рассмотрены программы на ассемблере, скомпилированные с ключами -O0 и -O2, и проведено последующее сравнение для поиска различий разных уровней оптимизации.

Приложение 1. 1ый листинг (код программы)

```
#include <cmath>
#include <cstdlib>

long double LeibnizFormula(int n) {
    long double pi = 0;
    for (int i = 0; i < n; ++i) {
        long double appendix = 1 / (long double) (2 * i + 1);
        pi += appendix*pow(-1, i);
    }
    pi *= 4;
    return pi;
}


int main() {
    int N = 1000000;
    long double pi = LeibnizFormula(N);
    return 0;
}
```

Приложение 2. 2ой листинг (ключ -O0)

```
LeibnizFormula(int):            // Начало функции LeibnizFormula
        pushl   %ebp            // Сохранить указатель кадра вызвавшей программы
        movl    %esp, %ebp      // Создать новый стек (база нового стека =  вершина прошлого стека)
        subl    $56, %esp       // Зарезервировать место под локальные переменные
        fldz                    // Загрузить число 0.0
        fstpt   -20(%ebp)       // Записать и вытолкнуть вещественное число pi = 0.0
        movl    $0, -24(%ebp)   // Записать по адресу -24(%ebp) значение $0(int i = 0)
.L3:
        movl    -24(%ebp), %eax // Записать в %eax значение из памяти (переменная i)
        cmpl    8(%ebp), %eax   // Сравнить значение из памяти (n) и %eax (i < n)
        jge     .L2             // Переход, если больше или равно
        movl    -24(%ebp), %eax // Записать в %eax значение из памяти (переменная i)
        addl    %eax, %eax      // Сложить с регистром %eax с %eax и записать в %eax (2 * i)
        addl    $1, %eax        // Добавить единицу туда же (2 * i + 1)
        movl    %eax, -44(%ebp) // Записать в память значение 2 * i + 1
        fildl   -44(%ebp)       // Загрузить целое число 2 * i + 1 в стек
        fld1                    // Загрузить константу +1.0 в стек
        fdivp   %st, %st(1)     // Деление вещественного числа с выталкиванием(1 / (2 * i + 1))
        fstpt   -36(%ebp)       // Записать и вытолкнуть вещественное число (сосчитали long double appendix)
        fildl   -24(%ebp)       // Загрузить целое число (i)
        leal    -8(%esp), %esp  // Загрузить адрес %esp-8 в регистр %esp
        fstpl   (%esp)          // Записать и вытолкнуть вещественное число
        fld1                    // Загрузить число +1.0
        fchs                    // Смена знака числа (-1.0)
        leal    -8(%esp), %esp  // Загрузить адрес %esp-8 в регистр %esp
        fstpl   (%esp)          // Записать и вытолкнуть вещественное число
        call    pow             // Вызов подпрограммы pow
        addl    $16, %esp       // Увеличить регистр %esp на 16
        fldt    -36(%ebp)       // Загрузить вещественное число (appendix)
        fmulp   %st, %st(1)     // Умножение вещественного числа с выталкиванием (appedix * pow(-1, i))
        fldt    -20(%ebp)       // Загрузить вещественное число appedix * pow(-1, i)
        faddp   %st, %st(1)     // Сложение с вещественным числом с выталкиванием(pi += appedix * pow(-1, i))
        fstpt   -20(%ebp)       // Записать значение pi в память
        addl    $1, -24(%ebp)   // Увеличить i на 1
        jmp     .L3             // Начать заново
.L2:
        fldt    -20(%ebp)       // Загрузить вещественное число pi
        fldt    .LC3            // Загрузить вещественные числа .LC3
        fmulp   %st, %st(1)     // Умножить pi и 4
        fstpt   -20(%ebp)       // Записать значение pi * 4 в память
        fldt    -20(%ebp)       // Загрузить вещественное число pi * 4
        leave                   // Выход из процедуры
        ret                     // Возврат из подпрограммы
main:
        leal    4(%esp), %ecx   // Поместить в %ecx адрес предыдущего элемента в стеке
        andl    $-16, %esp      // Выравнить указатель по границе 16 байт
        pushl   -4(%ecx)        // Добавить в стек указатель на вершину предыдущего стека (%ecx - 4 = %esp)
        pushl   %ebp            // Добавить в стек указатель на базу предыдущего стека
        movl    %esp, %ebp      // Создать новый стек (база нового стека = вершина прошлого стека)
        pushl   %ecx            // Добавить в стек %ecx
        subl    $20, %esp       // Зарезервировать место под локальные переменные
        movl    $1000000, -12(%ebp)     // Сохранить 1000000 в %ebp - 12 (N = 1000000)
        subl    $12, %esp       // Вычесть из %esp 12
        pushl   -12(%ebp)       // Добавить в стек %ebp - 12
        call    LeibnizFormula(int)     // Вызов подпрограммы LeibnizFormula
        addl    $16, %esp       // Добавить к %esp 16
        fstpt   -24(%ebp)       // Загрузить вещественное число из %ebp - 24 (pi)
        movl    $0, %eax        // %eax = 0 (return 0)
        movl    -4(%ebp), %ecx  // %ecx = %ebp - 4
        leave                   // Выход из процедуры
        leal    -4(%ecx), %esp  // %esp = %ecx - 4 = %ebp - 8
        ret                     // Возврат из подпрограммы
.LC3:                           // Константы, высчитанные компилятором
        .long   0
        .long   -2147483648
        .long   16385
```

Приложение 3. Зий листинг (ключ -O2)

```
LeibnizFormula(int):            // Начало функции LeibnizFormula
        pushl   %esi            // Положить в стек %esi (int n)
        pushl   %ebx            // Сохранить значение %ebx
        subl    $20, %esp       // Зарезервировать место под локальные переменные
        movl    32(%esp), %esi  // %esi = %esp + 32
        testl   %esi, %esi      // Проверка на ноль: если ноль, то флаг ZF = 0 (n&n)
        jle     .L14            // Переход, если ZF = 0 (n = 0 - значит считать ничего не нужно)
        xorl    %ebx, %ebx      // Обнулить %ebx (int i = 0)
        fldz                    // Загрузить число 0.0 (pi = 0)
.L13:
        movl    %ebx, 12(%esp)  // %esp + 12 = %ebx (int i )
        subl    $8, %esp        // Зарезервировать место под локальные переменные
        fstpt   8(%esp)         // Записать и вытолкнуть вещественное число
        fildl   20(%esp)        // Загрузить целое число
        fstpl   (%esp)          // Записать и вытолкнуть вещественное число
        pushl   $-1074790400    // Добавить в стек -1
        pushl   $0              // Добавить в стек 0
        call    pow             // Вызов подпрограммы pow
        leal    1(%ebx,%ebx), %eax      // %eax = 2 * %ebx + 1 (2 * i + 1)
        addl    $1, %ebx        // %ebx += 1
        movl    %eax, 28(%esp)  // %esp + 28 = %eax
        fildl   28(%esp)        // Загрузить целое число
        fdivrs  .LC2            // Обратное деление вещественного числа (appendix = 1 / (2 * i + 1))
        fmulp   %st, %st(1)     // Умножение вещественного числа с выталкиванием (appendix * pow(-1, i))
        fldt    16(%esp)        // Загрузить вещественное число appendix * pow(-1, i)
        addl    $16, %esp       // %esp += 16
        faddp   %st, %st(1)     // Сложение с вещественным числом с выталкиванием (pi += appendix * pow(-1, i))
        cmpl    %ebx, %esi      // Сравнение(i и n). Если равно, то флаг ZF = 1
        jne     .L13            // Переход, если не равно (ZF = 0) (i < n)
        fmuls   .LC3            // Умножить на 4 (pi *= 4)
        addl    $20, %esp       // Вернуть указатель на предыдущую вершину
        popl    %ebx            // Получить сохраненное значение %ebx
        popl    %esi            // Получить сохраненный индекс источника
        ret                     // Возврат из подпрограммы
.L14:
        addl    $20, %esp       // Вернуть указатель на предыдущую вершину
        fldz                    // Загрузить число 0.0 (pi = 0)
        popl    %ebx            // Получить сохраненное значение %ebx
        popl    %esi            // Получить сохраненный индекс источника
        ret                     // Возврат из подпрограммы
main:
        leal    4(%esp), %ecx   // Поместить в %ecx адрес предыдущего элемента в стеке
        andl    $-16, %esp      // Выравнить указатель по границе 16 байт
        pushl   -4(%ecx)        // Добавить в стек указатель на вершину предыдущего стека (%ecx - 4 = %esp)
        pushl   %ebp            // Добавить в стек указатель на базу предыдущего стека
        movl    %esp, %ebp      // Создать новый стек (база нового стека = вершина прошлого стека)
        pushl   %ecx            // Добавить в стек %ecx
        subl    $16, %esp       // Зарезервировать место под локальные переменные
        pushl   $1000000        // Добавить в стек 1000000 (N = 1000000)
        call    LeibnizFormula(int)     // Вызов функции LeibnizFormula
        fstp    %st(0)          // Записать и вытолкнуть вещественное число (pi)
        movl    -4(%ebp), %ecx  // %ecx = %ebp - 4
        addl    $16, %esp       // Сместить вершину стека на 16 (вернуть к предыдущему занчению)
        xorl    %eax, %eax      // Обнулить %eax
        leave                   // Выход из процедуры
        leal    -4(%ecx), %esp
        ret                     // Возврат из подпрограммы
.LC2:                           // Константы, высчитанные компилятором
        .long   1065353216      // 1
.LC3:
        .long   1082130432      // 4
```