###### 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

«Лабораторная работа 4»

Студента 1 курса, 19210 группы

**Пирожков Андрей Константинович**

Направление 09.03.01 – «Информатика и вычислительная техника»

Преподаватель:

(ученая степень, звание)

Д.С.Иванишкин

Новосибирск 2020

**СОДЕРЖАНИЕ**

[ЦЕЛЬ 3](#_Toc56618212)

[ЗАДАНИЕ 3](#_Toc56618213)

[ОПИСАНИЕ РАБОТЫ 4](#_Toc56618214)

[ЗАКЛЮЧЕНИЕ 6](#_Toc56618215)

[Приложение 1 *Листинг файла Lab4.c* 7](#_Toc56618216)

[Приложение 2 *Листинг файла Lab4(-O0).s* 8](#_Toc56618217)

[Приложение 3 *Листинг файла Lab4(-O3).s* 11](#_Toc56618218)

# ЦЕЛЬ

* Знакомство с программной архитектурой ARM
* Анализ ассемблерного листинга программы для архитектуры ARM.

# ЗАДАНИЕ

1. Изучить основы программной архитектуры ARM:
2. Для программы на языке Си (из лабораторной работы 1) сгенерировать ассемблерные листинги для архитектуры ARM, используя различные уровни комплексной оптимизации.
3. Проанализировать полученные листинги и сделать следующее.
   * Сопоставьте команды языка Си с машинными командами.
   * Определить размещение переменных языка Си в программах на ассемблере (в каких регистрах, в каких ячейках памяти).
   * Описать и объяснить оптимизационные преобразования, выполненные компилятором.
   * Продемонстрировать использование ключевых особенностей архитектур ARM на конкретных участках ассемблерного кода.
4. Составить отчет по лабораторной работе. Отчет должен содержать следующее.
   * Титульный лист.
   * Цель лабораторной работы.
   * Полный компилируемый листинг реализованной программы и команды для ее компиляции.
   * Листинг на ассемблере с описаниями назначения команд с точки зрения реализации алгоритма выбранного варианта.
   * Вывод по результатам лабораторной работы.

**Вариант задания: 7**

Алгоритм сортировки методом пузырька. Дан массив случайных чисел длины N. На первой итерации попарно упорядочиваются все соседние элементы; на второй – все элементы, кроме последнего элемента; на третьей – все элементы, кроме последнего элемента и предпоследнего элемента и т.п.

# **ОПИСАНИЕ РАБОТЫ**

За основу этой лабораторной работы была взята 1-ая лабораторная работа, написанная на языке Си (стандарта 1999г). Был написан алгоритм пузырьковой сортировки массива из N=120000 элементов. Входные данные этого массива генерируются рандомом (srand(1)).

Анализировал ассемблерный код двух видов оптимизации:

|  |  |  |  |
| --- | --- | --- | --- |
| Оптимизация | Команда для компиляции | Название файлов | Листинг |
| -O0 | arm-linux-gnueabihf-gcc -S -O0 -o Lab4(-O0).s Lab4.c | Lab4(-O0).s | [Приложение 2](#_Приложение_2) |
| -O3 | arm-linux-gnueabihf-gcc -S -O3 -o Lab4(-O3).s Lab4.c | Lab4(-O3).s | [Приложение 3](#_Приложение_3) |

Я подробно разобрал ассемблерский код ([Приложение 2](#_Приложение_2)). Поэтому не считаю нужным здесь его пересказывать. Могу отметить основные отличия между оптимизациями. Самое главное отличие – в логике сортировки пузырьком.

Сразу бросается в глаза очень странные преобразования и вычисления регистров для неоптимизированного случая. Постоянно делается ненужная операция в виде прибавления 800 к регистрам адресов. А потом отнимается уже нужное количество. Странно что это надо делать в две операции, когда можно сделать одной операцией. В оптимизированном варианте таких преобразований практически нет.

В обычном варианте инкрементация происходит в одной из ячеек памяти. При этом каждый раз загружая, редактируя и выгружая её обратно. И чтобы потом обратиться к ячейки используется регистр и его смещение. Тогда как в оптимизированном варианте используется адресная инкрементация. Т.е. просто адрес каждый раз увеличивается на 4, а потом по нему обращаемся.

Сортировка пузырьком занимает 88 строчек (против 13 строчек). Код ассемблера как будто постоянно забывает, что содержится в регистрах и после каждого обращения в память начинает заново вспоминать и загружать данные из стека в регистры.

Также в оптимизированном варианте есть применение условного оператора к двум следующим строчкам. Это как раз обмен местами двух ячеек массива. В обычном случае применяется переход по метке, чтобы не менять ничего местами.

Ну и ещё из интересного. Строки для вывода в консоль в обычном варианте находятся сверху, а в оптимизированном – снизу.

Вроде это всё по основным отличиям.ЗАКЛЮЧЕНИЕ

В ходе этой лабораторной работы я лабораторной работы я познакомился с программной архитектурой ARM и проанализировал ассемблерный листинг своей программы. Смог сопоставить команды языка си с машинным кодом. Определил размещение переменных языка Си в программах, указал основные оптимизации, и показал, чем отличаются эти оптимизации. Также я изучил набор регистров, а также их функции и назначение. Рассмотрел арифметико-логические команды, которые использовались в моей программе. Просмотрел способы адресации памяти и передачи управления. Разобрался в работе со стеком. Изучил как осуществляется вызов подпрограмм и их передачу параметров, а также способы возврата результатов. Немножко изучил работу сопроцессора и работу с векторными расширениями.

# Приложение 1

*Листинг файла Lab4.c*

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#define N 120000

int main()

{

//Объявленные переменные

int massive[N]; //массив который будем сортировать

int start; //фиксирование времени начала сортировки

int finish; //фиксирование времени после сортировки

//Сохдаём массив с рандомными значениями в ячейках

srand(1);

for (int i = 0; i < N; i++)

{

massive[i] = rand();

}

//Сортируем пузырьком

start = clock();

for (int i = 0; i < N; i++)

{

for (int j = i; j < N; j++)

{

if (massive[i] < massive[j])

{

int temp = massive[i];

massive[i] = massive[j];

massive[j] = temp;

}

}

}

finish = clock();

//Выводим время

printf("Time = %d\n", (finish - start));

printf("Frist element in massive: %d", massive[0]);

return 0;

}

# Приложение 2

*Листинг файла Lab4(-O0).s*

.arch armv7-a

.eabi\_attribute 28, 1

.fpu vfpv3-d16

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 6

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "Lab4.c"

.section .rodata

.align 2

.LC0:

.ascii "Time = %d\012\000"

.align 2

.LC1:

.ascii "Frist element in massive: %d\000"

.text

.align 2

.global main

.syntax unified

.thumb

.thumb\_func

.type main, %function

main:

@ args = 0, pretend = 0, frame = 480032

@ frame\_needed = 1, uses\_anonymous\_args = 0

push {r4, r7, lr} @сохраняем в стеке 3 регистра (lr ~ r14)

sub sp, sp, #479232 @sp=sp-479232

sub sp, sp, #804  @sp=sp-479232-804=sp-480036 возможно выделение памяти для программы

add r7, sp, #0 @r7=sp+0=sp (r7 - это верхняя граница)

movw r3, #:lower16:\_\_stack\_chk\_guard @какая-то защита стэка (возможно от переполнения) (записывает в последние 16 приемника)

movt r3, #:upper16:\_\_stack\_chk\_guard @какая-то защита стэка (возможно от переполнения) (записывает последние 16 источника в первые 16 приемника)

ldr r3, [r3] @получаем значение из адреса r3 (получили непонять что(тупо какое-то рандомное значение))

add r2, r7, #479232 @r2=r7+479232

add r2, r2, #796 @r2=r2+796=r7+479232+796=r7+480028 (r2 - нижняя граница)

str r3, [r2] @сохраняем значение r3(нечто странное) по адресу r2 (скорее всего фиксировать будет переполнение)

movs r0, #1 @r0=1 (Если указан S, флаги условий обновляются по результатам операции)

bl srand @вызов функции srand (Инструкции B и BL выполняют переход на указанный адрес. BL, кроме того, сохраняет адрес возврата (адрес команды, следующей за BL) в регистре LR (R14))

add r3, r7, #800 @r3=r7+800

sub r3, r3, #796 @r3=r3-796=r7+4 [это адрес i]

movs r2, #0 @r2=0 (i=0)

str r2, [r3] @записали нолик почти в самую верхнюю ячейку [r3]

b .L2 @примерно тут начинается цикл рандомного заполнения ячеек

.L3:

bl rand @вызываем функцию рандома

mov r1, r0 @r1=r0=1 (возможно получили рандомное значение по умолчанию)

add r3, r7, #800 @r3=r7+800 (вниз)

sub r3, r3, #772 @r3=r3+772=r7+28 (вверх) (это начало: massive[0])

add r2, r7, #800 @r2=r7+800 (вниз)

sub r2, r2, #796 @r2=r2+796=r7+4 (вверх)

ldr r2, [r2] @r2=i (забрала значение почти из самого вверха r7+4)

str r1, [r3, r2, lsl #2] @сохраняем r1 (massive[i]) по адресу r3+r2\*4 т.е. сохранили в r3+i\*4(LSL предоставляет значение регистра, умноженное на степень двойки, вставляя нули в освобожденные битовые позиции)

add r3, r7, #800 @r3=r7+800 (вниз)

sub r3, r3, #796 @r3=r3+796=r7+4 (вверх)

add r2, r7, #800 @r2=r7+800 (вниз)

sub r2, r2, #796 @r2=r2+796=r7+4 (вверх)

ldr r2, [r2] @r2=i

adds r2, r2, #1 @r2++ (i++)

str r2, [r3] @сохраняем r2(i) по адресу r7+4

.L2:

add r3, r7, #800 @r3=r7+800

sub r3, r3, #796 @r3=r3-796=r7+4 (r3=i)

ldr r2, [r3] @r2=i

movw r3, #54463 @записали в последние 16 битиков 54463 (1101010010111111)

movt r3, 1 @(где #?) записали в первые 16 битиков единичку (0000000000000001)

cmp r2, r3 @сравниваем r2(i) и r3 (итого r3=00000000000000011101010010111111=119999)

ble .L3 @<=

add r4, r7, #800 @r4=r7+800

sub r4, r4, #784 @r4=r7+16 (start)

bl clock @вызываем clock для стартового времени

str r0, [r4] @записываем стартовое время в r7+16

add r3, r7, #800 @r3=r7+800 (тут анчинается сортировка пузырьком)

sub r3, r3, #792 @r3=r7+8 (адрес i)

movs r2, #0 @r2=0 (i=0)

str r2, [r3] @сейвим r2(i) в r7+8

b .L4 @летим на проверку условия первого цикла

.L8: @тутачки вложеный цикл сортирвки

add r3, r7, #800 @r3=r7+800

sub r3, r3, #788 @r3=r7+12 (возможный адрес j)

add r2, r7, #800 @r2=r7+800

sub r2, r2, #792 @r2=r7+8 (это адрес i)

ldr r2, [r2] @r2=i

str r2, [r3] @закидываем по адресу r7+12 значение r2 (j=i)

b .L5 @летим на проверку условия второго цикла

.L7: @тут начинается swap

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это начало: massive[0])

add r2, r7, #800 @r2=r7+800

sub r2, r2, #792 @r2=r7+8 (это адрес i)

ldr r2, [r2] @r2=i

ldr r1, [r3, r2, lsl #2] @r1=massive[i] (по адресу r3+i\*4)

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это начало: massive[0])

add r2, r7, #800 @r2=r7+800

sub r2, r2, #788 @r2=r7+12 (это адрес j)

ldr r2, [r2] @r2=j

ldr r3, [r3, r2, lsl #2] @r3=massive[j]=[r3+i\*4]

cmp r1, r3 @сравниваем massive[i] и massive[j]

bge .L6 @=> (!<)

add r3, r7, #800 @r3=r7+800

sub r3, r3, #780 @r3=r7+20 (возможно это temp)

add r2, r7, #800 @r2=r7+800

sub r2, r2, #772 @r2=r7+28 (это начало (точка отсчета по координате i): massive[0])

add r1, r7, #800 @r1=r7+800

sub r1, r1, #792 @r1=r7+8 (это адрес i)

ldr r1, [r1] @r1=i

ldr r2, [r2, r1, lsl #2] @r2=massive[i] (по адресу r2+i\*4)

str r2, [r3] @temp=massive[i] (r7+20:=r2)

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это начало (точка отсчета по координате i): massive[0])

add r2, r7, #800 @r2=r7+800

sub r2, r2, #788 @r2=r7+12 (это адрес j)

ldr r2, [r2] @r2=j

ldr r1, [r3, r2, lsl #2] @r1=massive[j] (по адресу r2+j\*4)

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это начало (точка отсчета по координате i): massive[0])

add r2, r7, #800 @r2=r7+800

sub r2, r2, #792 @r2=r7+8 (это адрес i)

ldr r2, [r2] @r2=i

str r1, [r3, r2, lsl #2] @massive[i]=massive[j] (сохраним r1=massive[j] по адресу r3+i\*4=massive[i])

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это начало (точка отсчета по координате i): massive[0])

add r2, r7, #800 @r2=r7+800

sub r2, r2, #788 @r2=r7+12 (это адрес j)

ldr r1, [r2] @r1=massive[j]

add r2, r7, #800 @r2=r7+800

sub r2, r2, #780 @r2=r7+20 (это temp)

ldr r2, [r2] @r2=temp

str r2, [r3, r1, lsl #2] @massive[j]=temp (temp загружает в r3+j\*4=massive[j])

.L6:

add r3, r7, #800 @r3=r7+800

sub r3, r3, #788 @r3=r7+12 (это j)

add r2, r7, #800 @r2=r7+800

sub r2, r2, #788 @r2=r7+12 (это i)

ldr r2, [r2] @r2=j

adds r2, r2, #1 @r2++ (j++)

str r2, [r3] @закидываем по адресу r7+12 значение j

.L5:

add r3, r7, #800 @r3=r7+800

sub r3, r3, #788 @r3=r7+12 (адрес j)

ldr r2, [r3] @r2=j

movw r3, #54463 @записали в последние 16 битиков 54463 (1101010010111111)

movt r3, 1 @(где #?) записали в первые 16 битиков единичку (0000000000000001)

cmp r2, r3 @сравниваем r2(j) и r3 (итого r3=00000000000000011101010010111111=119999)

ble .L7 @<= (меняем местами)

add r3, r7, #800 @r3=r7+800

sub r3, r3, #792 @r3=r7+8 (адрес i)

add r2, r7, #800 @r2=r7+800

sub r2, r2, #792 @r2=r7+8 (адрес i)

ldr r2, [r2] @r2=i

adds r2, r2, #1 @i++

str r2, [r3] @закидываем по адресу r7+8 значение i

.L4: @это первый цикл сортировки

add r3, r7, #800 @r3=r7+800

sub r3, r3, #792 @r3=r7+8 (адрес i)

ldr r2, [r3] @r2=i

movw r3, #54463 @записали в последние 16 битиков 54463 (1101010010111111)

movt r3, 1 @(где #?) записали в первые 16 битиков единичку (0000000000000001)

cmp r2, r3 @сравниваем r2(i) и r3 (итого r3=00000000000000011101010010111111=119999)

ble .L8 @<=

add r4, r7, #800 @r4=r7+800

sub r4, r4, #776 @r4=r7+24 (finish)

bl clock @выщываем функции clock

str r0, [r4] @загружаем полученное время по адресу r7+24

add r2, r7, #800 @r2=r7+800

sub r2, r2, #776 @r2=r7+24

add r3, r7, #800 @r3=r7+800

sub r3, r3, #784 @r3=r7+16

ldr r2, [r2] @r2=finish

ldr r3, [r3] @r3=start

subs r3, r2, r3 @r3=r2-r3=finish-start

mov r1, r3 @r1=r3=количесву времени

movw r0, #:lower16:.LC0 @в r0 записываются первые 16 битиков строки

movt r0, #:upper16:.LC0 @в r0 записываются вторые 16 битиков строки

bl printf @вызываем функцию printf

add r3, r7, #800 @r3=r7+800

sub r3, r3, #772 @r3=r7+28 (это massive[0])

ldr r3, [r3] @r3=massive[0]

mov r1, r3 @r1=r3

movw r0, #:lower16:.LC1 @в r0 записываются первые 16 битиков строки

movt r0, #:upper16:.LC1 @в r0 записываются вторые 16 битиков строки

bl printf @вызываем функцию printf

movs r3, #0 @r3=0

mov r0, r3 @r0=r3=0

movw r3, #:lower16:\_\_stack\_chk\_guard @какая-то защита стэка (возможно от переполнения) (записывает в последние 16 приемника)

movt r3, #:upper16:\_\_stack\_chk\_guard @какая-то защита стэка (возможно от переполнения) (записывает последние 16 источника в первые 16 приемника)

add r2, r7, #479232 @r2=r7+479232

add r2, r2, #796 @r2=r7+480028

ldr r2, [r2] @r2=нечто странному что в начале было записано из \_\_stack\_chk\_guard

ldr r3, [r3] @r3=нечто странному что сейчас было записано из \_\_stack\_chk\_guard

cmp r2, r3 @сравниваем и видимо выявляем утечку памяти или переполнение стека

beq .L10 @если они равны то скипем

bl \_\_stack\_chk\_fail @иначе у нас ошибка стека

.L10:

add r7, r7, #479232 @r7=r7+479232

add r7, r7, #804 @r7=r7+480036

mov sp, r7 @sp=r7 зачитска памяти

@ sp needed

pop {r4, r7, pc} @выпихиваем 3 регистра из стека

.size main, .-main

.ident "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"

.section .note.GNU-stack,"",%progbits

# Приложение 3

*Листинг файла Lab4(-O3).s*

.arch armv7-a

.eabi\_attribute 28, 1

.fpu vfpv3-d16

.eabi\_attribute 20, 1

.eabi\_attribute 21, 1

.eabi\_attribute 23, 3

.eabi\_attribute 24, 1

.eabi\_attribute 25, 1

.eabi\_attribute 26, 2

.eabi\_attribute 30, 2

.eabi\_attribute 34, 1

.eabi\_attribute 18, 4

.file "Lab4.c"

.section .text.startup,"ax",%progbits

.align 2

.global main

.syntax unified

.thumb

.thumb\_func

.type main, %function

main:

@ args = 0, pretend = 0, frame = 480008

@ frame\_needed = 0, uses\_anonymous\_args = 0

push {r4, r5, r6, r7, lr}

movw r6, #:lower16:\_\_stack\_chk\_guard

sub sp, sp, #479232

movt r6, #:upper16:\_\_stack\_chk\_guard

sub sp, sp, #780

movs r0, #1 @для srand(1)

ldr r3, [r6]

add r2, sp, #479232

add r2, r2, #772

mov r4, sp @r4=sp

str r3, [r2]

bl srand

add r3, sp, #4 @r3=sp+4

add r5, r3, #479232

add r5, r5, #764 @r5=sp+4+479996 (sp+119999+1)

.L2:

bl rand

str r0, [r4, #4]! @неявная адресная инкрементация записываем r0 по адресу r4+4

cmp r5, r4

bne .L2 @пока не равны

bl clock

add r5, sp, #479232

add r4, sp, #4 @r4=sp+4

add r5, r5, #772 @r5=sp+480004

mov r7, r0 @r7=start

.L3:

mov r3, r4 @r3=r4=sp+4 (j=i)

.L5:

ldr r0, [r3] @r0=massive[j]

adds r3, r3, #4 @j++

ldr r1, [r4] @r1=massive[i]

cmp r1, r0 @сравниваем massive[i] massive[j]

itt lt @сравнение < (signed) (применяется на следующие 2 инструкции)

strlt r0, [r4]

strlt r1, [r3, #-4]

cmp r3, r5 @условия внутреннего

bne .L5

adds r4, r4, #4 @i++

cmp r4, r3 @условия внешнего цикла

bne .L3

bl clock

movw r1, #:lower16:.LC0

movt r1, #:upper16:.LC0

subs r2, r0, r7 @finish - start

movs r0, #1

bl \_\_printf\_chk

ldr r2, [sp, #4]

movw r1, #:lower16:.LC1

movs r0, #1

movt r1, #:upper16:.LC1

bl \_\_printf\_chk

add r3, sp, #479232

add r3, r3, #772

movs r0, #0

ldr r2, [r3]

ldr r3, [r6]

cmp r2, r3

bne .L12

add sp, sp, #479232

add sp, sp, #780

@ sp needed

pop {r4, r5, r6, r7, pc}

.L12:

bl \_\_stack\_chk\_fail

.size main, .-main

.section .rodata.str1.4,"aMS",%progbits,1

.align 2

.LC0:

.ascii "Time = %d\012\000"

.space 1

.LC1:

.ascii "Frist element in massive: %d\000"

.ident "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"

.section .note.GNU-stack,"",%progbits