МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4
по курсу «ЭВМ и периферийные устройства»

ВВЕДЕНИЕ В АРХИТЕКТУРУ ARM

Выполнил: студент 2-го курса гр. 17208
Гафиятуллин А.Р.

Новосибирск, 2018

1. ЦЕЛИ РАБОТЫ:

1. Знакомство с программной архитектурой ARM;
2. Анализ ассемблерного листинга программы для архитектуры ARM;

2. ХОД РАБОТЫ:

Для достижения поставленных целей был выбран 7 вариант задания: 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.

1. Написана программа на языке C, которая реализует алгоритм сортировки методом пузырька;

   Исходный код программы:

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *left, int *right)
{
	int tmp = *left;
	*left = *right;
	*right = tmp;
}
void bubble_sort(int *begin, int *end)
{
	for(int *out_iter = end - 1; out_iter != begin; out_iter--)
		for(int *in_iter = begin; in_iter != out_iter; in_iter++)
			if(*in_iter > *(in_iter + 1))
				swap(in_iter, in_iter + 1);
}
int main()
{
```

```
    int size = 0;
    scanf("%d", &size);
    int *array = (int*)malloc(size * sizeof(int));
    for(int i = 0; i < size; i++)
        scanf("%d", array + i);
    bubble_sort(array, array + size);
    for(long long i = 0; i < size; i++)
        printf("%d", array[i]);
    return 0;
}
```

Для компияции под архитектуру ARM был использован кросс-компилятор arm-linux-gnueabihf-gcc.

Команда компиляции (без оптимизаций): arm-linux-gnueabihf-gcc main.c -o main

При компиляции с оптимизациями добавляется ключ -O{0, 1, 2, 3, s, fast, g), например arm-linux-gnueabihf-gcc -O1 main.c -o main

Компиляция и тестирование программы проходили на Linux-машине с Elementary OS 64 bit: Linux kernel 4.15.0-36-generic, Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz и установленной вирутальной машиной Qemu для эмулирования архитектуры ARM(со статической линковкой библиотек).

2. Для вышеприведенной программы были сгенерированны ассемблерные листинги, с использованием различных уровней комплексной оптимизации.

   Команда генерирования ассемблерных листингов: arm-linux-gnueabihf-gcc -S main.c

   При компиляции с оптимизациями добавляется ключ -O{0, 1, 2, 3, s, fast, g), например arm-linux-gnueabihf-gcc -S -O1 main.c

3. Листинг с описаниями назначения команд с точки зрения реализации сортировки пузырьком:

-O0:

```
.arch armv7-a              //архитектура armv7-a
.eabi_attribute 28, 1      //выбор различных атрибутов
.fpu vfpv3-d16             //выбор математического сопроцессора
.eabi_attribute 20, 1      //выбор различных атрибутов
.eabi_attribute 21, 1      //выбор различных атрибутов
.eabi_attribute 23, 3      //выбор различных атрибутов
.eabi_attribute 24, 1      //выбор различных атрибутов
.eabi_attribute 25, 1      //выбор различных атрибутов
.eabi_attribute 26, 2      //выбор различных атрибутов
.eabi_attribute 30, 6      //выбор различных атрибутов
.eabi_attribute 34, 1      //выбор различных атрибутов
.eabi_attribute 18, 4      //выбор различных атрибутов
.file   "main.c"           //имя компилируемого файла
.text
.align 2          //установка выравнивания - выравнивание на 4 байта
.global     swap//swap - глобальный идентификатор, виден линковщику
.syntax unified//выбор единого синтаксиса для ARM и THUMB
.thumb            //выбор генерации системы команд THUMB
.thumb_func    //следующая функция - функция системы команд THUMB
.type  swap, %function       //swap - это функция
swap:
@ args = 0, pretend = 0, frame = 16
@ frame_needed = 1, uses_anonymous_args = 0
@ link register save eliminated.
push   {r7}//сохранение на стеке значений регистров r7(общий регистр)
                                        //void swap(int *left, int *right){
sub    sp, sp, #20    //выделение 20 байт на стеке
```

```
add    r7, sp, #0      //r7 = sp
str    r0, [r7, #4]    //сохранение знаечния left на стеке
str    r1, [r7]        //сохранение знаечния right на стеке
ldr    r3, [r7, #4]    //r3 = left
ldr    r3, [r3]        //r3 = *r3, т.е r3 = *left
str    r3, [r7, #12]   //сохраняем значение, лежащее по адресу left на стеке
                                                    //int tmp = *left;
ldr    r3, [r7]        //r3 = right
ldr    r2, [r3]        //r2 = *right
ldr    r3, [r7, #4]    //r3 = left
str    r2, [r3]        //*r3 = r2, т.е *left = *right          //*left = *right;
ldr    r3, [r7]        //r3 = right
ldr    r2, [r7, #12]   //r2 = temp
str    r2, [r3]        //*r3 = temp                            //*right = tmp;
nop                    //ничего не делать
adds   r7, r7, #20     //возвращаем 20 байт на стеке
mov    sp, r7          //sp = r7
@ sp needed
ldr    r7, [sp], #4    //r7 = sp, которое было до входа в функцию, снимаем со стека это значение
bx     lr              //смена системы команд от THUMB к ARM при возврате к вызывающей функции (если потребуется)    //}
.size  swap, .-swap//директива компилятора для подсчета размера функции
.align 2               //установка выравнивания - выравнивание на 4 байта
.global    bubble_sort //bubble_sort - глобальный идентификатор, виден линковщику
.syntax unified        //выбор единого синтаксиса для ARM и THUMB
.thumb                 //выбор генерации системы команд THUMB
.thumb_func    //следующая функция - функция системы команд THUMB
```

```
.type  bubble_sort, %function    //bubble_sort - это функция
bubble_sort:
@ args = 0, pretend = 0, frame = 16
@ frame_needed = 1, uses_anonymous_args = 0
push  {r7, lr} //сохранение на стеке значений регистров r7(общий регистр), lr(регистр связи)          //void bubble_sort(int *begin, int *end){
sub    sp, sp, #16   //выделение 16 байт на стеке
add    r7, sp, #0    //r7 = sp
str    r0, [r7, #4]  //записываем указатель array на стек
str    r1, [r7]      //записываем указатель (array + size * 4) на стек
ldr    r3, [r7]      //r3 = *r7, т.е r3 = array + size * 4
subs   r3, r3, #4//r3 = r3 - 4, т.е r3 = array + size * 4 – 4//int *out_iter = end - 1;
str    r3, [r7, #8]  //загрузим значение итератора out_iter на стек
b  .L3               //переход к проверке условия внешнего цикла
.L7:
ldr    r3, [r7, #4]  //r3 = begin
str    r3, [r7, #12] //записываем значение итератора in_iter на стек
                                                    //int *in_iter = begin;
b  .L4               //переходим к проверке условия внутреннего цикла
.L6:
ldr    r3, [r7, #12] //r3 = in_iter
ldr    r2, [r3]      //r2 = *in_iter
ldr    r3, [r7, #12] //r3 = in_iter
adds   r3, r3, #4    //r3 = in_iter + 4
ldr    r3, [r3]      //r3 = *r3, т.е r3 = *(in_iter + 4)
cmp    r2, r3    //сравниваем значения в r2 и r3//if(*in_iter > *(in_iter + 1))
ble    .L5           //если r2 <= r3, то переходим на следующую итерацию
ldr    r3, [r7, #12] //r3 = in_iter
adds   r3, r3, #4    //r3 = in_iter + 1
```

```
mov   r1, r3     //r1 = r3, r1 = in_iter + 1, r1 - второй аргумент функции swap
ldr   r0, [r7, #12] //r0 = in_iter, r0 - первый аргумент функции swap
bl    swap       //вызов функции swap с сохранением адреса возврата в lr
                                                    //swap(in_iter, in_iter + 1);
.L5:
ldr   r3, [r7, #12]       //r3 = in_iter
adds  r3, r3, #4          //r3 = in_iter + 4
str   r3, [r7, #12] //обновим значение итератра in_iter на стеке//in_iter++
.L4:
ldr   r2, [r7, #12]       //r2 = in_iter
ldr   r3, [r7, #8]        //r3 = out_iter
cmp   r2, r3              //сравниваем итераторы          //in_iter != out_iter;
bne   .L6                 //если не равны, то переходим в тело цикла
ldr   r3, [r7, #8]        //r3 = out_iter
subs  r3, r3, #4          //r3 = out_iter - 4
str   r3, [r7, #8]  //обновим значение итератра out_iter на стеке//out_iter--
.L3:
ldr   r2, [r7, #8]  //r2 = out_iter
ldr   r3, [r7, #4]  //r3 = array, т.е r3 = begin
cmp   r2, r3        //сравниваем итреторы           //out_iter != begin;
bne   .L7           //если не равны, то переходим к внутреннему циклу
nop                 //иначе, ничего не делать
adds  r7, r7, #16   //возвращем 16 байт на стеке
mov   sp, r7        //sp = r7
@ sp needed
pop   {r7, pc} //вернем данные со стека, которые сохранили при входе в
функцию, обратно в регистры                                          //}
.size  bubble_sort, .-bubble_sort    //директива компилятора для подсчета
размера функции
```

```
.section .rodata //секция глобальных и статических неизменяемых данных
.align 2 //установка выравнивания - выравнивание на 4 байта
.LC0:
.ascii "%d\000" //форматная строка
.text //секция скомпилированного машинного кода
.align 2 //установка выравнивания - выравнивание на 4 байта
.global main //main - глобальный идентификатор, виден линковщику
.syntax unified //выбор единого синтаксиса для ARM и THUMB
.thumb //выбор генерации системы команд THUMB
.thumb_func //следующая функция - функция системы команд THUMB
.type main, %function //main - это функция
main:
@ args = 0, pretend = 0, frame = 32 //int main(){
@ frame_needed = 1, uses_anonymous_args = 0
push {r4, r7, lr} //сохранение на стеке значений регистров r4, r7(общие регистры), lr(регистр связи)
sub sp, sp, #36 //резервирование 36 байт на стеке
add r7, sp, #0 //r7 = sp(указатель на голову стека)
movwr3, #:lower16:__stack_chk_guard //во вторые 16 бит r3 записывается вторая половина __stack_chk_guard (защита стека)
movt r3, #:upper16:__stack_chk_guard //в первые 16 бит r3 записывается первая половина __stack_chk_guard (защита стека)
ldr r3, [r3] //r3 = *r3
str r3, [r7, #28] //значение в __stack_chk_guard сохраняется на стеке
movs r3, #0 //r3 = 0
str r3, [r7, #4] //сохранение значения из r3 на стеке //int size = 0;
adds r3, r7, #4 //получение в r3 адреса переменной size на стеке
mov r1, r3//r1 = r3, т.е r1 = &size, r1 - это второй аргумент функции scanf
```

movwr0, #:lower16:.LC0//во вторые 16 бит r0 записывается вторая половина LCO ("%d")

movt r0, #:upper16:.LC0//в первые 16 бит r0 записывается первая половина LCO ("%d"), r0 - первый аргумент функции scanf

bl __isoc99_scanf //вызов функции scanf с сохранением адреса возврата в lr //scanf("%d", &size);

ldr r3, [r7, #4] //r3 = size

lsls r3, r3, #2 //r3 = r3 * 4, т.е r3 = (size * sizeof(int))

mov r0, r3 //r0 = r3, r0 - аргумент функции malloc

bl malloc //вызов функции malloc с сохранением адреса возврата //int *array = (int*)malloc(size * sizeof(int));

mov r3, r0 //r3 = r0, в r0 вернулся указатель на аллоцированную память (array)

str r3, [r7, #12] //сохранение значения r3 (array) на стеке

movs r3, #0 //r3 = 0

str r3, [r7, #8] //сохранение значения из r3 (i) на стеке //int i = 0;

b .L9 //переход к проверке условия входа в цикл

.L10:

ldr r3, [r7, #8] //r3 = i

lsls r3, r3, #2 //r3 = r3 * 4, т.е r3 = (i * sizeof(int))

ldr r2, [r7, #12] //r2 = array

add r3, r3, r2 //r3 = r3 + r2, т.е r3 = (array + (i * sizeof(int))) - получили адрес i-го элемента массива

mov r1, r3 //r1 = r3, в r1 адрес i-го элемента массива, r1 - это второй аргумент функции scanf

movwr0, #:lower16:.LC0//во вторые 16 бит r0 записывается вторая половина LCO ("%d")

movt r0, #:upper16:.LC0//в первые 16 бит r0 записывается первая половина LCO ("%d"), r0 - первый аргумент функции scanf

```
bl      __isoc99_scanf      //вызов функции scanf с сохранением адреса возврата в lr                    //scanf("%d", array + i);
ldr     r3, [r7, #8]    //r3 = i
adds    r3, r3, #1      //r3 = r3 + 1, т.е i = i + 1                                                  //i++
str     r3, [r7, #8]    //сохранение значения из r3 (i) на стеке
.L9:
ldr     r3, [r7, #4]    //r3 = size
ldr     r2, [r7, #8]    //r2 = i
cmp     r2, r3          //r2 < r3, т.е i < size                                                       //i < size;
blt     .L10            //если i < size, то переход в тело цикла
ldr     r3, [r7, #4]    //если i >= size, то r3 = size
lsls    r3, r3, #2      //r3 = r3 * 4, т.е r3 = (size * sizeof(int))
ldr     r2, [r7, #12]   //r2 = array
add     r3, r3, r2      //r3 = r3 + r2, т.е r3 = (array + (size * sizeof(int))), получили в r3 конец массива
mov     r1, r3          //r1 = r3, в r1 адрес конца массива, r1 - второй аргумент функции bubble_sort
ldr     r0, [r7, #12]   //r0 = array
bl      bubble_sort     //вызов функции bubble_sort с сохранением адреса возврата в lr                //bubble_sort(array, array + size);
mov     r3, #0          //r3 = 0
mov     r4, #0          //r4 = 0
strd    r3, [r7, #16]   //сохранение значения из r3 (i) на стеке  //long long i = 0;
b .L11                  //переход к проверке условия выхода из цикла
.L12:
ldr     r3, [r7, #16]   //r3 = i
lsls    r3, r3, #2      //r3 = r3 * 4, т.е r3 = (i * sizeof(int))
ldr     r2, [r7, #12]   //r2 = array
```

add r3, r3, r2 //r3 = r3 + r2, т.е r3 = (array + (i * sizeof(int))) - получили адрес i-го элемента массива

ldr r3, [r3] //r3 = *r3,т.е r3 = array[i]

mov r1, r3 //r1 = r3, r1 = array[i], r1 - второй аргумент функции printf

movwr0, #:lower16:.LC0//во вторые 16 бит r0 записывается вторая половина LCO ("%d")

movt r0, #:upper16:.LC0//в первые 16 бит r0 записывается первая половина LCO ("%d"), r0 - первый аргумент функции printf

bl printf //вызов функции printf с сохранением адреса возврата в lr //printf("%d", array[i]);

ldrd r3, [r7, #16] //загрузка двух 32-битных слов, r3 = i, r4 = 0(в нашем случае)

adds r3, r3, #1 //r3 = r3 + 1,т.е i = i + 1, суффикс s команды adds позволяет менять флаги //i++

adc r4, r4, #0 //если из младшего слова 64-битного числа i есть перенос, то прибавим его к старшему слову

strd r3, [r7, #16] //сохраняем значения из r3 (i) на стеке

.L11:

ldr r3, [r7, #4] //r3 = size

asr r4, r3, #31 //арифметический сдвиг вправо на 31 бит значения size, получим 0 (в нашем случае), r4 = 0, старшее 32-битное слово в i

ldrd r1, [r7, #16] //загрузка двух 32-битных слов, r1 = i, r2 = 0

cmp r1, r3 //сравниваем младшие 32-битные слова перменных i и size

sbcs r3, r2, r4 //r3 = r2 - r4 с переносом, т.е сравниваются старшие слова переменных size и i путем вычитания с переносом

blt .L12 //если i < size, то переходим в тело цикла //i < size;

movs r3, #0 //r3 = 0

mov r0, r3 //r0 = r3

movwr3, #:lower16:__stack_chk_guard //во вторые 16 бит r3 записывается вторая половина __stack_chk_guard (защита стека)

movt r3, #:upper16:__stack_chk_guard //в первые 16 бит r3 записывается первая половина __stack_chk_guard (защита стека)

ldr r2, [r7, #28] //получим в r2 значение __stack_chk_guard, которое сохраняли в начале программы

ldr r3, [r3] //r3 = *r3, получим в r3 значение __stack_chk_guard, которое должно быть

cmp r2, r3 //сравним их

beq .L14 //если они равны, то все хорошо, можно завершать программу

bl __stack_chk_fail//а иначе проверим стек на ошибки

.L14:

adds r7, r7, #36 //вернем зарезервированную память

mov sp, r7 //sp = r7

@ sp needed

pop {r4, r7, pc} //вернем данные со стека, которые сохранили в начале программы, обратно в регистры //}

.size main, .-main//директива компилятора для подсчета размера функции

.ident "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609" //директива совместимости на уровне исходного кода

.section .note.GNU-stack,"",%progbits //указание линовщику пометить стек и данные, как неисполняемые

-O1:

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, 1
.eabi_attribute 34, 1
.eabi_attribute 18, 4
.file   "main.c"
.text
.align 2
.global     swap
.syntax unified
.thumb
.thumb_func
.type   swap, %function
swap:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 0, uses_anonymous_args = 0
@ link register save eliminated.
                    //не выделяется память на стеке при входе в функцию
ldr     r3, [r0]    //операции выполняются сразу на регистрах, без перегона в стек
ldr     r2, [r1]
str     r2, [r0]
str     r3, [r1]
bx      lr
.size   swap, .-swap
.align 2
.global     bubble_sort
```

```
.syntax unified
.thumb
.thumb_func
.type  bubble_sort, %function
bubble_sort:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 0, uses_anonymous_args = 0
@ link register save eliminated.
subs   r1, r1, #4
cmp    r1, r0
bne    .L12
bx     lr
.L6:
ldr    r2, [r3]
ldr    r4, [r3, #4]
cmp    r2, r4
itt    gt              //используетя условное исполнение команд на основе
```
флагов состояния процессора(пропуск 2-х следующих команд, если r2 <= r4 (*in_iter <= *(in_iter + 1)) (в системе команд THUMB сокращает длину команд, путем исключения битов суффикса, в системе команд ARM эта команда не имеет никакого эффекта, т.к все команды полноразмерные, т.е суффиксы дают эффект)

```
strgt  r4, [r3]        //swap не вызывается, заменен более коротким кодом,
```
выполняющим ту же задачу

```
strgt  r2, [r3, #4]
adds   r3, r3, #4
cmp    r3, r1
bne    .L6
.L7:
```

```
subs   r1, r1, #4
cmp   r0, r1
beq    .L2
b .L10
.L12:
push   {r4}
.L10:
cmp   r0, r1
it      ne      //используется условное исполнение команд (пропуск следующей команды, если r0 == r1 (in_iter == out_iter))
movne     r3, r0
bne    .L6      //если in_iter != out_iter, то входим в тело цикла
b .L7
.L2:
ldr     r4, [sp], #4
bx      lr
.size   bubble_sort, .-bubble_sort
.align 2
.global     main
.syntax unified
.thumb
.thumb_func
.type   main, %function
main:
@ args = 0, pretend = 0, frame = 8
@ frame_needed = 0, uses_anonymous_args = 0
push   {r4, r5, r6, r7, r8, lr}       //больше данных держится на регистрах
sub    sp, sp, #8
movwr3, #:lower16:__stack_chk_guard
```

```
movt  r3, #:upper16:__stack_chk_guard
ldr   r3, [r3]
str   r3, [sp, #4]
add   r1, sp, #8
movs  r3, #0
str   r3, [r1, #-8]!      //операция с постинкрементированием
movwr0, #:lower16:.LC0
movt  r0, #:upper16:.LC0
bl    __isoc99_scanf
ldr   r0, [sp]
lsls  r0, r0, #2
bl    malloc
mov   r6, r0
ldr   r1, [sp]
cmp   r1, #0        //если size меньше или равен 0, то можно не пытаться входить в цикл
ble   .L14
mov   r5, r0
movs  r4, #0
movwr7, #:lower16:.LC0
movt  r7, #:upper16:.LC0
.L15:
mov   r1, r5
mov   r0, r7
bl    __isoc99_scanf
adds  r4, r4, #1
ldr   r1, [sp]
adds  r5, r5, #4
cmp   r1, r4
```

```
bgt     .L15
.L14:
add     r1, r6, r1, lsl #2
mov     r0, r6
bl      bubble_sort
ldr     r3, [sp]
cmp     r3, #0
ble     .L16
subs    r6, r6, #4
movs    r4, #0
movs    r5, #0
movwr7, #:lower16:.LC0
movt    r7, #:upper16:.LC0
mov     r8, #1
.L17:
ldr     r2, [r6, #4]!
mov     r1, r7
mov     r0, r8
bl      __printf_chk            //использование защищенных версий
```
стандартных библиотечных функций
```
adds    r4, r4, #1
adc     r5, r5, #0
ldr     r2, [sp]
asrs    r3, r2, #31
cmp     r4, r2
sbcs    r3, r5, r3
blt     .L17
.L16:
movs    r0, #0
```

```
movwr3, #:lower16:__stack_chk_guard
movt  r3, #:upper16:__stack_chk_guard
ldr     r2, [sp, #4]
ldr     r3, [r3]
cmp   r2, r3
beq     .L18
bl      __stack_chk_fail
.L18:
add     sp, sp, #8
@ sp needed
pop     {r4, r5, r6, r7, r8, pc}
.size   main, .-main
.section     .rodata.str1.4,"aMS",%progbits,1
.align 2
.LC0:
.ascii  "%d\000"
.ident "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
.section     .note.GNU-stack,"",%progbits
```

Отличия от листинга без комплексной оптимизации (-O0):

1. При входе в подпрограммы не создается отдельный фрейм на стеке;
2. Задействовано больше регистров, а операции с данными при возможности выполняются сразу на регистрах, без сохранений на стек;
3. Не вызываются некоторые функции, а заменяются вставкой в код более коротких реализаций, выполняющих ту же самую функциональность;
4. Проверяется возможность не входить в цикл и избегается вход в цикл при возможности;
5. Используются защищенные версии некоторых стандартных библиотечных функций;
6. Используются команды с условным исполнением;

7. Используются выражения с инкрементированием операндов.

-O2:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O1;
2. Более агрессивный инлайнинг функций: был полностью заинлайнен bubble_sort();
3. Если инструкции независимы, то не обязательно выполняются в логическом порядке, например:

   movw r0, #:lower16:.LC0

   str r6, [r1, #-8]!

   movt r0, #:upper16:.LC0

   хотя, как в O0 и O1, логичнее и привычнее:

   str r3, [r1, #-8]! //команда не “вклинивается” посреди двух следующих команд

   movw r0, #:lower16:.LC0

   movt r0, #:upper16:.LC0

-O3:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O2.

-Os:

```
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, 4
.eabi_attribute 34, 1
.eabi_attribute 18, 4
.file   "main.c"
.text
.align 1            //установка выравнивания – выравнивание на 2 байта
.global     swap
.syntax unified
.thumb
.thumb_func
.type  swap, %function
swap:
@ args = 0, pretend = 0, frame = 0
@ frame_needed = 0, uses_anonymous_args = 0
@ link register save eliminated.
ldr     r3, [r0]
ldr     r2, [r1]
str     r2, [r0]
str     r3, [r1]
bx      lr
.size   swap, .-swap
.align 1
.global     bubble_sort
.syntax unified
.thumb
.thumb_func
.type  bubble_sort, %function
bubble_sort:
@ args = 0, pretend = 0, frame = 0
```

```
@ frame_needed = 0, uses_anonymous_args = 0
push   {r4, lr}
.L8:
subs   r1, r1, #4
cmp    r1, r0
beq    .L10
mov    r3, r0
.L5:
ldm    r3, {r2, r4}
adds   r3, r3, #4
cmp    r2, r4
ittgt
strgt  r4, [r3, #-4]
strgt  r2, [r3]
cmp    r1, r3          //проверка условия выхода из внутреннего цикла
bne    .L5             //после полного прохода внутреннего цикла, возвращаемся обратно к началу функции bubble_sort, чтобы проверить условие выхода из внешнего цикла
b .L8
.L10:
pop    {r4, pc}
.size  bubble_sort, .-bubble_sort
.section    .text.startup,"ax",%progbits
.align 1
.global     main
.syntax unified
.thumb
.thumb_func
.type  main, %function
```

```
main:
@ args = 0, pretend = 0, frame = 8
@ frame_needed = 0, uses_anonymous_args = 0
push   {r0, r1, r4, r5, r6, r7, r8, lr}
add    r1, sp, #8
ldr    r5, .L19
movs   r4, #0
ldr    r0, .L19+4
str    r4, [r1, #-8]!
ldr    r3, [r5]
ldr    r6, .L19+4
str    r3, [sp, #4]
bl     __isoc99_scanf
ldr    r0, [sp]
lsls   r0, r0, #2
bl     malloc
mov    r8, r0
.L12:
ldr    r1, [sp]
cmp    r4, r1
bge    .L17
add    r1, r8, r4, lsl #2
mov    r0, r6
bl     __isoc99_scanf
adds   r4, r4, #1
b      .L12
.L17:
add    r1, r8, r1, lsl #2
mov    r0, r8
```

```
bl      bubble_sort
ldr     r4, .L19+4
movs    r6, #0
movs    r7, #0
.L14:
ldr     r2, [sp]
cmp     r6, r2
asr     r3, r2, #31
sbcs    r3, r7, r3
bge     .L18
lsls    r3, r6, #2
mov     r1, r4
movs    r0, #1
ldr     r2, [r8, r3]
bl      __printf_chk
adds    r6, r6, #1
adc     r7, r7, #0
b .L14
.L18:
ldr     r2, [sp, #4]
movs    r0, #0
ldr     r3, [r5]
cmp     r2, r3
beq     .L16
bl__stack_chk_fail
.L16:
add     sp, sp, #8
@ sp needed
pop     {r4, r5, r6, r7, r8, pc}
```

```
.L20:
.align 2
.L19:
.word __stack_chk_guard
.word .LC0
.size   main, .-main
.section    .rodata.str1.1,"aMS",%progbits,1
.LC0:
.ascii  "%d\000"
.ident "GCC: (Ubuntu/Linaro 5.4.0-6ubuntu1~16.04.9) 5.4.0 20160609"
.section    .note.GNU-stack,"",%progbits
```

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутствуют почти все оптимизации уровня -O2, кроме тех, которые увеличивают размер бинарника;
2. Изменена структура двойного цикла: избегается операции декрементирования внешнего итерирующего указателя, что уменьшает размер кода;
3. Установлено выравнивание на 2 байта.

-Ofast:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O3;

-Og:

Отличия от листинга без комплексной оптимизации (-O0):

1. Используются почти все оптимизации уровня -O1, кроме подстановки кода функций, а так же на стеке сохраняется текущая база кадра вызывающей функции, что сохраняет возможность просмотра стека вызовов.

3. ВЫВОДЫ:

1. Понакомились с программной архитектурой ARM;
2. Проанализировали ассемблерные листинги программы для архитектуры ARM;
3. Сопоставили команды языка Си с машинными командами;
4. Продемонстрировали использование ключевых особенностей архитектуры ARM на конкретных участках ассемблерного кода;
5. Описали и объяснили оптимизационнные преобразования, выполненные компилятором на различных уровнях комплексной оптимизации.