МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №3
по курсу «ЭВМ и периферийные устройства»

ВВЕДЕНИЕ В АРХИТЕКТУРУ x86/x86-64

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

Новосибирск, 2018

1. ЦЕЛИ РАБОТЫ:

1. Знакомство с программной архитектурой x86/x86-64;
2. Анализ ассемблерного листинга программы для архитектуры x86/x86-64;

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;
}
```

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

При компиляции с оптимизациями добавляется ключ -O{0, 1, 2, 3, s, fast, g), например 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.

2. Для вышеприведенной программы были сгенерированны ассемблерные листинги для архитектуры x86 и архитектуры x86-64, используя различные уровни комплексной оптимизации.

   Команды генерирования ассемблерных листингов:

   x86-64: gcc -S main.c

   x86: gcc -S -m32 main.c

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

3. Листинг для архитектуры x86-64 на ассемблере с описаниями назначения команд с точки зрения реализации сортировки пузырьком:

   -O0:

```
  .file"main.c"                    //имя компилируемого файла
.text
```

```
   .globl   swap                //идентификатор swap имеет глобальную область видимости, виден линковщику
   .type    swap, @function     //swap - это функция
swap:
.LFB2:
   .cfi_startproc //отладочная информация //void swap(int *left, int *right){
    pushq  %rbp                 //кладем на стек текущую базу кадра
    .cfi_def_cfa_offset 16      //отладочная информация
   .cfi_offset 6, -16           //отладочная информация
   movq   %rsp, %rbp            //теперь база указывает на свое старое значение и тем самым разделяет фреймы функций
   .cfi_def_cfa_register 6      //отладочная информация
   movq   %rdi, -24(%rbp)       //загрузим left во фрейм
   movq   %rsi, -32(%rbp)       //загрузим right во фрейм
   movq   -24(%rbp), %rax       //left -> rax                  //left;
   movl   (%rax), %eax          //значение по rax -> eax       //eax = *left;
   movl   %eax, -4(%rbp)        //загрузим tmp во фрейм        //int temp = eax;
   movq   -32(%rbp), %rax       //right -> rax                 //right;
   movl   (%rax), %edx          //значение по rax -> edx       //edx = *right;
   movq   -24(%rbp), %rax       //left -> rax                  //left;
   movl   %edx, (%rax)          //edx -> в значение по rax     //*left = *right;
   movq   -32(%rbp), %rax       //right -> rax                 //right;
   movl   -4(%rbp), %edx        //tmp -> edx                   //tmp = *right
   movl   %edx, (%rax)          //tmp -> в значение по rax     //*right = tmp;
   nop                          //нет операции
   popq   %rbp          //удаляем фрейм, снимая со стека указатель на него
   .cfi_def_cfa 7, 8    //отладочная информация
   ret                  //возвращаем уппавление вызвавшей функции   // }
   .cfi_endproc         //отладочная информация
```

```
.LFE2:
   .size     swap, .-swap      //отладочная информация
   .globl    bubble_sort       //идентификатор bubble_sort - глобальный, виден линковщику
   .type     bubble_sort, @function   //bubble_sort - функция
bubble_sort:
.LFB3:
   .cfi_startproc              //отладочная информация //void bubble_sort(int *begin, int *end){
   pushq    %rbp               //кладем на стек текущую базу кадра
   .cfi_def_cfa_offset 16      //отладочная информация
   .cfi_offset 6, -16          //отладочная информация
   movq     %rsp, %rbp         //теперь база указывает на свое старое значение и тем самым разделяет фреймы функций
   .cfi_def_cfa_register 6     //отладочная информация
   subq     $32, %rsp          //резервируем 32 байта памяти на фрейме
   movq     %rdi, -24(%rbp) //загрузим begin во фрейм
   movq     %rsi, -32(%rbp) //загрузим end во фрейм
   movq     -32(%rbp), %rax//end -> rax                    //end;
   subq     $4, %rax           //rax = rax - 4              //end - 1;
   movq     %rax, -16(%rbp)//загрузим out_iter во фрейм    //int  *out_iter  = end - 1;
   jmp .L3                     //прыгаем на сравнение во внешнем цикле алгоритма
.L7:
   movq     -24(%rbp), %rax    //begin -> rax          //begin;
   movq     %rax, -8(%rbp)     //загрузим in_iter во фрейм
                                                      //int *in_iter = begin;
   jmp .L4                     //прыгаем во внутренний цикл
```

```
.L6:
   movq    -8(%rbp), %rax  //in_iter -> rax                  //in_iter
   movl    (%rax), %edx    //rax -> edx                      //*in_iter;
   movq    -8(%rbp), %rax  //in_iter -> rax                  //in_iter;
   addq    $4, %rax        //сдвинем in_iter на 1 ячейку массива вправо
                                                             //in_iter + 1;
   movl    (%rax), %eax    //значение по (in_iter + 1) -> eax
                                                             //*(in_iter + 1);
   cmpl    %eax, %edx      //сравним значения, которые лежат в in_iter и в
(in_iter + 1)                                                //if(*in_iter > *(in_iter + 1))
   jle   .L5               //если в (in_iter + 1) меньшее значение,чем в
iter,то продолжаем этот блок,а иначе переходим к изменению in_iter
   movq    -8(%rbp), %rax  //in_iter -> rax                  //in_iter;
   leaq4(%rax), %rdx       //(in_iter + 1) -> rdx           //in_iter + 1
   movq    -8(%rbp), %rax  //in_iter -> rax                  //in_iter
   movq    %rdx, %rsi      //передаем (in_iter + 1) в swap
   movq    %rax, %rdi      //передаем in_iter в swap
   call swap               //вызываем swap
                                                             //swap(in_iter, in_iter + 1);
.L5:
   addq    $4, -8(%rbp)    //сдвигаем in_iter на 1 ячейку вправо
                                                             //in_iter++
.L4:
   movq    -8(%rbp), %rax  //in_iter -> rax                  //in_iter
   cmpq    -16(%rbp), %rax//rax срвнивается с out_iter       //in_iter != out_iter;
   jne  .L6                //если не равны, то прыгаем в тело цикла
   subq    $4, -16(%rbp)   //иначе сдвигаем out_iter на одну ячеку влево
                                                             //out_iter--
.L3:
```

```
  movq    -16(%rbp), %rax      //out_iter -> rax            //out_iter
  cmpq    -24(%rbp), %rax      //rax сравнивается с  begin
                                                          //out_iter != begin;
  jne  .L7                     //если не равны, то прыгаем на подготовку
ко внутреннему циклу
  nop                  // иначе нет операции
  leave                // выбрасываем последний кадр стека
  .cfi_def_cfa 7, 8    //отладочная информация
  ret                  //возвращаем управление вызывающей функции // }
  .cfi_endproc         //отладочная информация
.LFE3:
  .size    bubble_sort, .-bubble_sort    //отладочная информация
  .section .rodata                       //выбор секции файла
.LC0:
  .string  "%d"             //строковый литерал
  .text
  .globl   main             //идентификатор main - глобальный, виден
линковщику
  .type    main, @function  //main - имя функции
main:
.LFB4:
  .cfi_startproc            //отладочная информация        //int main(){
  pushq    %rbp             //кладем на стек текущую базу кадра
  .cfi_def_cfa_offset 16    //отладочная информация
  .cfi_offset 6, -16        //отладочная информация
  movq     %rsp, %rbp       //теперь база указывает на свое старое
значение и тем самым разделяет фреймы функций
  .cfi_def_cfa_register 6//отладочная информация
  subq     $32, %rsp        //зарезервируем 32 байта на стеке
```

```
movq    %fs:40, %rax      //получаем канарейку (защита стека)
movq    %rax, -8(%rbp)  //записываем значение канарейки
xorl%eax, %eax // обнуляем eax
movl    $0, -32(%rbp)     //запишем 0 в size              //int size = 0;
leaq-32(%rbp), %rax       //адрес size -> rax             //&size;
movq    %rax, %rsi        //адрес size - второй аргумент функции scanf
                                                          //"%d"
movl    $.LC0, %edi       //форматная строка - первый арумент функции scanf
movl    $0, %eax          //обнулили переменную, куда произойдет возврат значения scanf
call __isoc99_scanf  //вызов scanf                        //scanf("%d", &size);
movl    -32(%rbp), %eax     //size -> eax                 //size;
cltq                        //eax -> rax
salq$2, %rax                //умножим size на 4           //size * sizeof(int);
movq    %rax, %rdi          //передаем size * 4 в malloc
call malloc   //вызов malloc   //int *array = (int*)malloc(size * sizeof(int));
movq    %rax, -16(%rbp)     //запишем адрес array во фрейм
movl    $0, -28(%rbp)       //обнуляем итератор i     //int i = 0;
jmp .L9                     //прыгаем в цикл
.L10:
movl    -28(%rbp), %eax     //i -> eax
cltq                        //eax -> rax                //i;
leaq0(,%rax,4), %rdx//вычислим смещение относительно начала массива array и -> rdx  // i * 4;
movq    -16(%rbp), %rax     //array -> rax              //array;
addq    %rdx, %rax //получаем нужную позицию в массиве array, сохраняем в rax    //(array + i);
movq    %rax, %rsi          //rax -> rsi (передача аргумента)
```

```
  movl    $.LC0, %edi          //форматная строка - первый аргумент функции scanf (передача аргумента)
  movl    $0, %eax             //обнулили переменную, куда произойдет возврат значения scanf
  call __isoc99_scanf          //вызов scanf
                                                    //scanf("%d", array + i);
  addl    $1, -28(%rbp)//увеличили итератор i на 1//i++;
.L9:
  movl    -32(%rbp), %eax      //size -> eax          //size;
  cmpl    %eax, -28(%rbp)//сравниваем eax с i       //i < size;
  jl    .L10                   //если i < size, то считываем значение
  movl    -32(%rbp), %eax      //size -> eax          //size;
  cltq                         //eax -> rax
  leaq0(,%rax,4), %rdx         //вычислим смещение относительно начала массива array и -> rdx // size * 4;
  movq    -16(%rbp), %rax      //array -> rax         //array;
  addq    %rax, %rdx //получаем нужную позицию в массиве array, сохраняем в rdx
                               //array + (size * 4);
  movq    -16(%rbp), %rax      //array -> rax         // array;
  movq %rdx, %rsi      //rdx -> rsi (передача 2-го аргумента)
  movq    %rax, %rdi  //rax -> rdi (передача 1-го аргумента)
  call bubble_sort//вызов функции bubble_sort
                                                    //bubble_sort(array, array+size);
  movq    $0, -24(%rbp)    //обнуляем итератор i    //long long i = 0;
  jmp .L11                 //переходим к циклу
.L12:
  movq    -24(%rbp), %rax//i -> rax
  leaq0(,%rax,4), %rdx     //вычислим смещение относительно начала массива array и -> rdx          //i * 4;
```

```
  movq    -16(%rbp), %rax//array -> rax                  //array;
  addq    %rdx, %rax      //получаем нужную позицию в массиве array,
сохраняем в rax                                          //(array + i);
  movl    (%rax), %eax//получаем значение, лежащее по rax -> eax
                                                         //array[i]
  movl    %eax, %esi  //eax -> esi (передача 2-го аргумента)
  movl    $.LC0, %edi//форматная строка - первый аргумент printf
                                                         //"%d";
  movl    $0, %eax    //в eax будет возвращен результат работы printf
  call printf         //вызов printf                   //printf("%d", array[i]);
  addq    $1, -24(%rbp)   //увеличиваем i на 1         // i++;
.L11:
  movl    -32(%rbp), %eax     //size -> eax             // size;
  cltq                        //eax -> rax
  cmpq    -24(%rbp), %rax     //сравниваем i и rax //i < size;
  jg   .L12                   //если i < size, то переходим в тело цикла
  movl    $0, %eax            //0 -> eax
  movq    -8(%rbp), %rcx      //получаем канарейку
  xorq    %fs:40, %rcx        //сравниваем с ранее записанным
значением
  je   .L14                   //если все нормально(они равны), то
завершаем программу
  call __stack_chk_fail       //иначе проверяем стек на ошибки
.L14:
  leave                       //удаляем данные с кадра
  .cfi_def_cfa 7, 8           //отладочная информация
  ret                         //возвращаем управление операционной
системе                                                  // }
  .cfi_endproc                //отладочная информация
```

```
.LFE4:
   .size      main, .-main              //отладочная информация
   .ident     "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
   .section .note.GNU-stack,"",@progbits      //выбор секции файла
```

-O1:

```
.file   "main.c"
   .text
   .globl   swap
   .type    swap, @function
swap:
.LFB38:
   .cfi_startproc
   movl     (%rdi), %eax     //не выделяется память на стеке при входе в функцию
   movl     (%rsi), %edx     //все операции копирования переменных и указателей происходят прямо на регистрах
   movl     %edx, (%rdi)     //не совершается огромное число бесполезных обращений к стеку
   movl     %eax, (%rsi)
   ret
   .cfi_endproc
.LFE38:
   .size    swap, .-swap
   .globl   bubble_sort
   .type    bubble_sort, @function
bubble_sort:
.LFB39:
   .cfi_startproc
   subq     $4, %rsi
```

```
    cmpq    %rdi, %rsi
    jne  .L11
    rep ret
.L9:
    movl    (%rax), %edx    //swap замененен вставкой более оптимальной последовательности команд
    movl    4(%rax), %ecx
    cmpl    %ecx, %edx
    jle   .L5
    movl    %ecx, (%rax)
    movl    %edx, 4(%rax)
.L5:
    addq    $4, %rax
    cmpq    %rsi, %rax
    jne  .L9
.L7:
    subq    $4, %rsi
    cmpq    %rsi, %rdi
    je    .L2
.L11:
    movq    %rdi, %rax
    cmpq    %rsi, %rdi
    jne  .L9
    jmp .L7
.L2:
    rep ret
    .cfi_endproc
.LFE39:
    .size    bubble_sort, .-bubble_sort
```

```
    .section .rodata.str1.1,"aMS",@progbits,1
.LC0:
    .string  "%d"
    .text
    .globl   main
    .type    main, @function
main:
.LFB40:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    pushq    %rbx
    .cfi_def_cfa_offset 24
    .cfi_offset 3, -24
    subq     $24, %rsp //в rsp хранится начало выделенной памяти, обращения к стеку происходят в сторону роста адресов
    .cfi_def_cfa_offset 48
    movq     %fs:40, %rax
    movq     %rax, 8(%rsp)
    xorl%eax, %eax
    movl     $0, 4(%rsp)
    leaq4(%rsp), %rsi
    movl     $.LC0, %edi
    call __isoc99_scanf
    movslq  4(%rsp), %rdi
    salq$2, %rdi
    call malloc
    movq     %rax, %rbp
```

```
  movl    4(%rsp), %eax
  testl%eax, %eax //сравнение size с нулем
  jle  .L13          //если size меньше или равен 0, то можно не пытаться входить в цикл
  movl    $0, %ebx
.L14:
  movslq  %ebx, %rax //цикл со считыванием данных в массив организован намного компактнее
  leaq0(%rbp,%rax,4), %rsi
  movl    $.LC0, %edi
  movl    $0, %eax
  call __isoc99_scanf
  addl    $1, %ebx
  movl    4(%rsp), %eax
  cmpl    %ebx, %eax
  jg   .L14
.L13:
  cltq
  leaq0(%rbp,%rax,4), %rsi
  movq    %rbp, %rdi
  call bubble_sort
  cmpl    $0, 4(%rsp)
  jle  .L15
  movl    $0, %ebx
.L16:
  movl    0(%rbp,%rbx,4), %edx
  movl    $.LC0, %esi
  movl    $1, %edi
  movl    $0, %eax
```

```
    call __printf_chk       //вызывает защищенную версию printf
    addq    $1, %rbx
    movslq  4(%rsp), %rax
    cmpq    %rbx, %rax
    jg  .L16
.L15:
    movl    $0, %eax
    movq    8(%rsp), %rcx
    xorq    %fs:40, %rcx
    je  .L17
    call __stack_chk_fail
.L17:
    addq    $24, %rsp
    .cfi_def_cfa_offset 24
    popq    %rbx
    .cfi_def_cfa_offset 16
    popq    %rbp
    .cfi_def_cfa_offset 8
    ret
    .cfi_endproc
.LFE40:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
    .section .note.GNU-stack,"",@progbits
```

Отличия от листинга без комплексной оптимизации (-O0):

1. Обращение к элементам на стеке происходит в сторону роста адресов относительно указателя на голову стека, так как кадр на стеке не формируется;
2. При входе в подпрограммы не создается отдельный фрейм на стеке;

3. Все операции с данными при возможности выполняются сразу на регистрах, без постоянных перегонов на стек;
4. Не вызываются некоторые функции, а заменяются вставкой в код более оптимальных реализаций;
5. Проверяется возможность не входить в цикл и избегается вход в цикл при возможности;
6. Используются защищенные версии некоторых стандартных библиотечных функций;
7. Использование rep ret для выхода из подпрограмм (нужно для процессоров AMD, которые могут предсказывать переход по ветвлениям).

-O2:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O1;
2. Использование регистра r12.

-O3:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O2.

-Os:

```
.file   "main.c"
   .section .text.unlikely,"ax",@progbits
.LCOLDB0:
   .text
.LHOTB0:
   .globl   swap
   .type    swap, @function
swap:
.LFB20:
   .cfi_startproc
```

```
    movl    (%rdi), %eax
    movl    (%rsi), %edx
    movl    %edx, (%rdi)
    movl    %eax, (%rsi)
    ret
    .cfi_endproc
.LFE20:
    .size   swap, .-swap
    .section .text.unlikely
.LCOLDE0:
    .text
.LHOTE0:
    .section .text.unlikely
.LCOLDB1:
    .text
.LHOTB1:
    .globl  bubble_sort
    .type   bubble_sort, @function
bubble_sort:
.LFB21:
    .cfi_startproc
.L8:
    subq    $4, %rsi
    cmpq    %rdi, %rsi   //проверяем условие выхода из внешнего цикла
    je  .L10
    movq    %rdi, %rax
.L5:
    movl    (%rax), %edx
    movl    4(%rax), %ecx
```

```
  cmpl    %ecx, %edx
  jle  .L4
  movl    %ecx, (%rax)
  movl    %edx, 4(%rax)
.L4:
  addq    $4, %rax
  cmpq    %rax, %rsi
  jne  .L5
  jmp .L8       //после полного прохода внутреннего цикла, возвращаемся обратно к началу функции bubble_sort, чтобы проверить условие выхода из внешнего цикла
.L10:
  ret
  .cfi_endproc
.LFE21:
  .size    bubble_sort, .-bubble_sort
  .section .text.unlikely
.LCOLDE1:
  .text
.LHOTE1:
  .section .rodata.str1.1,"aMS",@progbits,1
.LC2:
  .string  "%d"
  .section .text.unlikely
.LCOLDB3:
  .section .text.startup,"ax",@progbits
.LHOTB3:
  .globl   main
  .type    main, @function
```

```
main:
.LFB22:
    .cfi_startproc
    pushq    %r12
    .cfi_def_cfa_offset 16
    .cfi_offset 12, -16
    pushq    %rbp
    .cfi_def_cfa_offset 24
    .cfi_offset 6, -24
    movl     $.LC2, %edi
    pushq    %rbx
    .cfi_def_cfa_offset 32
    .cfi_offset 3, -32
    xorl%ebx, %ebx
    subq     $16, %rsp
    .cfi_def_cfa_offset 48
    leaq4(%rsp), %rsi
    movl     $0, 4(%rsp)
    movq     %fs:40, %rax
    movq     %rax, 8(%rsp)
    xorl%eax, %eax
    call __isoc99_scanf
    movslq  4(%rsp), %rdi
    salq$2, %rdi
    call malloc
    movq     %rax, %rbp
    movq     %rax, %r12
.L12:
    movslq  4(%rsp), %rax
```

```
    cmpl    %eax, %ebx
    jge  .L18
    movq    %r12, %rsi
    movl    $.LC2, %edi
    xorl%eax, %eax
    call __isoc99_scanf
    incl%ebx          //инкременитруем итератор цикла с помощью унарной операции
    addq    $4, %r12
    jmp.L12
.L18:
    leaq0(%rbp,%rax,4), %rsi
    movq    %rbp, %rdi
    xorl%ebx, %ebx
    call bubble_sort
.L14:
    movslq  4(%rsp), %rax
    cmpq    %rax, %rbx
    jge  .L19
    movl    0(%rbp,%rbx,4), %edx
    movl    $.LC2, %esi
    movl    $1, %edi
    xorl%eax, %eax
    incq    %rbx
    call __printf_chk
    jmp.L14
.L19:
    xorl%eax, %eax
    movq    8(%rsp), %rcx
```

```
  xorq    %fs:40, %rcx
  je    .L16
  call __stack_chk_fail
.L16:
  addq    $16, %rsp
  .cfi_def_cfa_offset 32
  popq    %rbx
  .cfi_def_cfa_offset 24
  popq    %rbp
  .cfi_def_cfa_offset 16
  popq    %r12
  .cfi_def_cfa_offset 8
  ret
  .cfi_endproc
.LFE22:
  .size    main, .-main
  .section .text.unlikely
.LCOLDE3:
  .section .text.startup
.LHOTE3:
  .ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
  .section .note.GNU-stack,"",@progbits
```

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутствуют почти все оптимизации уровня -O2;
2. Изменена структура двойного цикла: избегается операции декрементирования внешнего итерирующего указателя, что уменьшает размер кода;
3. Использование унарного incl вместо бинарного addl при прибавлении единицы, что так же уменьшает размер кода.

-Ofast:

Отличия от листинга без комплексной оптимизации (-O0):

1. Присутсвуют все оптимизации уровня -O3;
2. Используется 32-битный регистр r13d вместо 64-битного r12, где это возможно.

-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.

Отличия ассемблерных листингов для архитектур x86 и x86-64:

```
.file   "main.c"
   .text
   .globl   swap
   .type    swap, @function
swap:
.LFB2:
   .cfi_startproc
   pushl    %ebp
   .cfi_def_cfa_offset 8
   .cfi_offset 5, -8
   movl     %esp, %ebp
   .cfi_def_cfa_register 5
   subl$16, %esp
   movl     8(%ebp), %eax
   movl     (%eax), %eax
   movl     %eax, -4(%ebp)
   movl     12(%ebp), %eax
```

```
    movl    (%eax), %edx
    movl    8(%ebp), %eax
    movl    %edx, (%eax)
    movl    12(%ebp), %eax
    movl    -4(%ebp), %edx
    movl    %edx, (%eax)
    nop
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE2:
    .size   swap, .-swap
    .globl  bubble_sort
    .type   bubble_sort, @function
bubble_sort:
.LFB3:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    subl$16, %esp
    movl    12(%ebp), %eax
    subl$4, %eax
    movl    %eax, -8(%ebp)
    jmp .L3
```

```
.L7:
    movl    8(%ebp), %eax
    movl    %eax, -4(%ebp)
    jmp .L4
.L6:
    movl    -4(%ebp), %eax
    movl    (%eax), %edx
    movl    -4(%ebp), %eax
    addl    $4, %eax
    movl    (%eax), %eax
    cmpl    %eax, %edx
    jle   .L5
    movl    -4(%ebp), %eax
    addl    $4, %eax
    pushl   %eax
    pushl   -4(%ebp)
    call swap
    addl    $8, %esp
.L5:
    addl    $4, -4(%ebp)
.L4:
    movl    -4(%ebp), %eax
    cmpl    -8(%ebp), %eax
    jne  .L6
    subl$4, -8(%ebp)
.L3:
    movl    -8(%ebp), %eax
    cmpl    8(%ebp), %eax
    jne  .L7
```

```
    nop
    leave
    .cfi_restore 5
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE3:
    .size   bubble_sort, .-bubble_sort
    .section .rodata
.LC0:
    .string "%d"
    .text
    .globl  main
    .type   main, @function
main:
.LFB4:
    .cfi_startproc
    leal 4(%esp), %ecx
    .cfi_def_cfa 1, 0
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    .cfi_escape 0x10,0x5,0x2,0x75,0
    movl    %esp, %ebp
    pushl   %ecx
    .cfi_escape 0xf,0x3,0x75,0x7c,0x6
    subl$36, %esp
    movl    %gs:20, %eax
    movl    %eax, -12(%ebp)
```

```
    xorl%eax, %eax
    movl    $0, -36(%ebp)
    subl$8, %esp
    leal -36(%ebp), %eax
    pushl   %eax
    pushl   $.LC0
    call __isoc99_scanf
    addl    $16, %esp
    movl    -36(%ebp), %eax
    sall $2, %eax
    subl$12, %esp
    pushl   %eax
    call malloc
    addl    $16, %esp
    movl    %eax, -28(%ebp)
    movl    $0, -32(%ebp)
    jmp .L9
.L10:
    movl    -32(%ebp), %eax
    leal 0(,%eax,4), %edx
    movl    -28(%ebp), %eax
    addl    %edx, %eax
    subl$8, %esp
    pushl   %eax
    pushl   $.LC0
    call __isoc99_scanf
    addl    $16, %esp
    addl    $1, -32(%ebp)
.L9:
```

```
    movl    -36(%ebp), %eax
    cmpl    %eax, -32(%ebp)
    jl  .L10
    movl    -36(%ebp), %eax
    leal 0(,%eax,4), %edx
    movl    -28(%ebp), %eax
    addl    %edx, %eax
    subl$8, %esp
    pushl   %eax        //второй аргумент функции bubble_sort - конецмассива передается через стек
    pushl   -28(%ebp)   //первый аргумент - начало массива - аналогично передается через стек
    call bubble_sort  //вызов bubble_sort
    addl    $16, %esp   //возвращение позиции указателя вершины стека после работы функции
    movl    $0, -24(%ebp)    //обнуляем вторую половину(младшие разряды) 64-битного long long int итератора
    movl    $0, -20(%ebp)    //обнуляем первую половину(старшие разряды) 64-битного long long int итератора
    jmp .L11
.L12:
    movl    -24(%ebp), %eax
    leal 0(,%eax,4), %edx
    movl    -28(%ebp), %eax
    addl    %edx, %eax
    movl    (%eax), %eax
    subl$8, %esp
    pushl   %eax
    pushl   $.LC0
```

```
    call printf
    addl    $16, %esp
    addl    $1, -24(%ebp)   //прибавляем 1 к младшему 32-битному слову 64-битного числа
    adcl$0, -20(%ebp)   //если есть перенос, то прибавим его к старшему 32-битному слову 64-битного числа
.L11:
    movl    -36(%ebp), %eax     //size -> eax
    cltd            //eax -> представлеятся в виде двух слов, где в edx все биты заполнены знаковым битом
    cmpl    -20(%ebp), %edx     //сравниваем старшее 32-битное слово 64-битного числа с нулями(в нашем случае)
    jg  .L12
    cmpl    -20(%ebp), %edx
    jl  .L16
    cmpl    -24(%ebp), %eax //сравниваем младшее 32-битное слово 64-битного числа с size
    ja  .L12            //переход в тело цикла, если size оказался больше
.L16:
    movl    $0, %eax
    movl    -12(%ebp), %ecx
    xorl%gs:20, %ecx
    je  .L15
    call __stack_chk_fail
.L15:
    movl    -4(%ebp), %ecx
    .cfi_def_cfa 1, 0
    leave
    .cfi_restore 5
```

```
    leal -4(%ecx), %esp
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE4:
    .size    main, .-main
    .ident    "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
    .section .note.GNU-stack,"",@progbits
```

1. Используются 32-битные регистры и 32-битные команды;
2. Все операции с 64 битными числами происходят в несколько этапов: резервируется два 32-битных слова; сравнивается так же два 32-битных слова поочередно, в зависимости от результата сравнения предыдущего слова; при операции сложения с 64-битным числом используется команда adcl для возможного переноса разряда в старшее 32-битное слово;
3. Аргументы для вызванной функции от вызывающей функции передаются через стек, а не в регистрах;
4. После возвращения из вызванной функции, указатель вершины стека возвращается в то состояние, которое было до вызова функции.

3. ВЫВОДЫ:

1. Познакомились с программной архитектурой x86/x86-64;
2. Проанализировали ассемблерный листинг программы для архитектуры x86/x86-64;
3. Сопоставьте команды языка Си с машинными командами;
4. Описали и объяснили оптимизационнные преобразования, выполненные компилятором на различных уровнях комплексной оптимизации;
5. Сравнили различия в программах для архитектуры x86 и архитектуры x86-64.