Задание 1

Напишем программу program1a.c с функцией simple_function(), вызываемой из main(). Данная функция создаёт целочисленную переменную с и присваивает ей значение 5. В программе program1b.c модернизируем функцию simple_function() так, чтобы она возвращала значение с. Сравним ассемблерные листинги: program1a_64bit.s и program1b_64bit.s. Они отличаются лишь одной пятнадцатой строкой. В первой программе команда nop говорит процессору ничего не делать, а во второй, значение с перемещается в регистр %eax для того, чтобы вернуть его вызывающему коду (См. Рис. 1 и 2).

```
 5  simple_function:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr64
 9      pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      movl    $5, -4(%rbp)
15      nop
16      popq    %rbp
17      .cfi_def_cfa 7, 8
18      ret
19      .cfi_endproc
```

Рисунок 1: Ф-я simple_function() из program1a_64bit.s

```
 5  simple_function:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr64
 9      pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      movl    $5, -4(%rbp)
15      movl    -4(%rbp), %eax
16      popq    %rbp
17      .cfi_def_cfa 7, 8
18      ret
19      .cfi_endproc
```

Рисунок 2: Ф-я simple_function() из program1b_64bit.s

```
24  main:
25  .LFB1:
26      .cfi_startproc
27      endbr64
28      pushq   %rbp
29      .cfi_def_cfa_offset 16
30      .cfi_offset 6, -16
31      movq    %rsp, %rbp
32      .cfi_def_cfa_register 6
33      subq    $16, %rsp
34      movl    $5, -4(%rbp)
35      movl    -4(%rbp), %eax
36      movl    %eax, %edi
37      call    simple_function
38      movl    $0, %eax
39      leave
40      .cfi_def_cfa 7, 8
41      ret
42      .cfi_endproc
```

Рисунок 3: Ф-я main() из program1c_64bit.s

В программе program1c.c значение с в simple_function() передаётся из функции main(), а после просто возвращается обратно. Ассемблерный листинг этой программы отличается от двух предыдущих. Теперь число 5 помещается в место на стеке внутри main(), после оно перемещается в регистр %edi (См. Рис. 3, строки 34-36). А уже с ним производятся операции в simple_function().

В программе program1d.c мы передаём несколько значений в simple_function() из функции main() и возвращаем их сумму обратно. Ассемблерный листинг получается очень похожим на предыдущую программу. Теперь мы помещаем на стек три значения, а потом перемещаем их в регистры (См. Рис. 4, строки 40-47). После функция simple_function() работает с этими регистрами.

Теперь будем сравнивать ассемблерный листинг, созданный на 32 битной архитектуре с тем, что мы получали ранее.

Сравнивая файлы program1a_64bit.s и program1a_32bit.s, program1b_64bit.s и program1b_32bit.s мы можем увидеть (за исключением некоторых системных вызовов) лишь то, что в 32 битном листинге у регистров вместо префикса «r», стоит префикс «e» (См. Рис. 5 и 6). В двух последних программах есть уже более значительные отличия (См. Рис. 4 и 7). В них мы передаём в функцию значение переменной (одной или нескольких). В 64 битной архитектуре значение каждой переменной мы просто клали в отдельный 64 битный регистр, а после, внутри функции, доставали необходимые нам значения из этих регистров. В 32 битной архитектуре мы не можем себе такого позволить, вероятно ввиду небольшого числа самих регистров, и поэтому мы передаём значения переменных в функцию работая непосредственно со стеком.

```
30  main:
31  .LFB1:
32      .cfi_startproc
33      endbr64
34      pushq   %rbp
35      .cfi_def_cfa_offset 16
36      .cfi_offset 6, -16
37      movq    %rsp, %rbp
38      .cfi_def_cfa_register 6
39      subq    $16, %rsp
40      movl    $5, -12(%rbp)
41      movl    $10, -8(%rbp)
42      movl    $15, -4(%rbp)
43      movl    -4(%rbp), %edx
44      movl    -8(%rbp), %ecx
45      movl    -12(%rbp), %eax
46      movl    %ecx, %esi
47      movl    %eax, %edi
48      call    simple_function
49      movl    $0, %eax
50      leave
51      .cfi_def_cfa 7, 8
52      ret
53      .cfi_endproc
```

Рисунок 4: Ф-я main() из program1d_64bit.s

```
24  main:
25  .LFB1:
26      .cfi_startproc
27      endbr64
28      pushq   %rbp
29      .cfi_def_cfa_offset 16
30      .cfi_offset 6, -16
31      movq    %rsp, %rbp
32      .cfi_def_cfa_register 6
33      movl    $0, %eax
34      call    simple_function
35      movl    $0, %eax
36      popq    %rbp
37      .cfi_def_cfa 7, 8
38      ret
39      .cfi_endproc
```

Рисунок 5: Ф-я main() из program1a_64bit.s

```
28  main:
29  .LFB1:
30      .cfi_startproc
31      endbr32
32      pushl   %ebp
33      .cfi_def_cfa_offset 8
34      .cfi_offset 5, -8
35      movl    %esp, %ebp
36      .cfi_def_cfa_register 5
37      call    __x86.get_pc_thunk.ax
38      addl    $_GLOBAL_OFFSET_TABLE_, %eax
39      call    simple_function
40      movl    $0, %eax
41      popl    %ebp
42      .cfi_restore 5
43      .cfi_def_cfa 4, 4
44      ret
45      .cfi_endproc
```

Рисунок 6: Ф-я main() из program1a_32bit.s

Задание 2

В первых двух программах (program2a_64bit.c и program2a_32bit.c, program2b_64bit.c и program2b_32bit.c) все отличия заключаются только в вышеупомянутой замене префиксов в именах регистров. В ассемблерных листингах третьей программы program2c.c мы можем заметить одну интересную особенность. В обоих листингах при заполнении массива используется стек, причём заполнение осуществляется сверху вниз. Но на 64 битной архитектуре (См. Рис 8 и 9) при этом задействуется указатель на начало стека со сдвигом в отрицательную сторону (вверх). А на 32 битной архитектуре мы используем указатель на текущее положение на стеке, который мы предварительно передвинули, со сдвигом в положительную сторону (вниз).

```
30  main:
31  .LFB1:
32      .cfi_startproc
33      endbr32
34      pushl   %ebp
35      .cfi_def_cfa_offset 8
36      .cfi_offset 5, -8
37      movl    %esp, %ebp
38      .cfi_def_cfa_register 5
39      subl    $16, %esp
40      call    __x86.get_pc_thunk.ax
41      addl    $_GLOBAL_OFFSET_TABLE_, %eax
42      movl    $5, -12(%ebp)
43      movl    $10, -8(%ebp)
44      movl    $15, -4(%ebp)
45      pushl   -4(%ebp)
46      pushl   -8(%ebp)
47      pushl   -12(%ebp)
48      call    simple_function
49      addl    $12, %esp
50      movl    $0, %eax
51      leave
52      .cfi_restore 5
53      .cfi_def_cfa 4, 4
54      ret
55      .cfi_endproc
```

Рисунок 7: Ф-я main() из program1d_32bit.s

```
 5  main:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr64
 9      pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      subq    $32, %rsp
15      movq    %fs:40, %rax
16      movq    %rax, -8(%rbp)
17      xorl    %eax, %eax
18      movl    $1, -32(%rbp)
19      movl    $2, -28(%rbp)
20      movl    $3, -24(%rbp)
21      movl    $4, -20(%rbp)
22      movl    $5, -16(%rbp)
23      movl    $0, %eax
24      movq    -8(%rbp), %rdx
25      xorq    %fs:40, %rdx
26      je  .L3
27      call    __stack_chk_fail@PLT
```

Рисунок 8: Ф-я main() из program2c_64bit.s

```
 5  main:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr32
 9      pushl   %ebp
10      .cfi_def_cfa_offset 8
11      .cfi_offset 5, -8
12      movl    %esp, %ebp
13      .cfi_def_cfa_register 5
14      andl    $-16, %esp
15      subl    $32, %esp
16      call    __x86.get_pc_thunk.ax
17      addl    $_GLOBAL_OFFSET_TABLE_, %eax
18      movl    %gs:20, %eax
19      movl    %eax, 28(%esp)
20      xorl    %eax, %eax
21      movl    $1, 8(%esp)
22      movl    $2, 12(%esp)
23      movl    $3, 16(%esp)
24      movl    $4, 20(%esp)
25      movl    $5, 24(%esp)
26      movl    $0, %eax
27      movl    28(%esp), %edx
28      xorl    %gs:20, %edx
29      je  .L3
30      call    __stack_chk_fail_local
```

Рисунок 9: Ф-я main() из program2c_32bit.s

Задание 3

Если просто описать структуру, то в ассемблерном листинге мы увидим лишь стандартный минимальный набор команд внутри функции main(). Но если мы инициализируем её поля (См. Program3b.c), то увидим отдельный блок в ассемблерном коде (См. Рис 10). Он будем одинаковым и на 32bit, и на 64bit, за исключением некоторых системных вызовов. Поэтому рассмотрим листинг, полученный на 64 битной архитектуре, так как он менее громоздкий.

```
 8  struct1:
 9      .long   1
10      .long   2
11      .byte   107
12      .zero   3
13      .long   15
14      .text
15      .globl  main
16      .type   main, @function
```

Рисунок 10: Структура struct1 из program3b_64bit.s

Рассмотрим как поля структуры описываются в ассемблерном листинге на примере структуры struct1 из файла program3b_64bit.s (См. Рис. 10).

Эта структура содержит три поля типа int и одно поле типа char (См. Рис. 12). Как видно из Рис. 10, int-овые поля описаны в строках 9, 10 и 13. Описание поля типа char занимает две строки — 11 и 12. Значение необходимого нам символа записывается в виде ASCII кода. В данном случае, код символа „k“ — 107.

```
17  main:
18  .LFB0:
19      .cfi_startproc
20      endbr64
21      pushq   %rbp
22      .cfi_def_cfa_offset 16
23      .cfi_offset 6, -16
24      movq    %rsp, %rbp
25      .cfi_def_cfa_register 6
26      movl    struct1(%rip), %eax
27      addl    $5, %eax
28      movl    %eax, struct1(%rip)
29      movl    4+struct1(%rip), %eax
30      addl    $2, %eax
31      movl    %eax, 4+struct1(%rip)
32      movb    $121, 8+struct1(%rip)
33      movl    12+struct1(%rip), %eax
34      addl    $10, %eax
35      movl    %eax, 12+struct1(%rip)
36      movl    $0, %eax
37      popq    %rbp
38      .cfi_def_cfa 7, 8
39      ret
40      .cfi_endproc
```

Рисунок 11: Ф-я main() из program3b_64bit.s

```
 3  struct example
 4  {
 5      int a;
 6      int b;
 7      char c;
 8      int d;
 9  };
10
11  struct example struct1 = {1, 2, 'k', 15};
```

Рисунок 12: Структура example и объявление структуры struct1 из program3b.c

Операции с полями структуры происходят внутри функции main() (См. Рис. 11). Как мы можем видеть struct1 в коде используется как указатель на адрес на стеке. Поэтому мы просто присваиваем каждому полю необходимые значения, прибавляя к адресу struct1 нужное количество байт. Так как прибавляемые значения положительные, то мы можем сделать вывод о том, что стек заполняется сверху вниз. Также в виду вышесказанного мы можем понять, что все поля структуры расположены в памяти в виде непрерывного массива данных.

Можно заметить, что прибавляемое значение всегда одно и то же, но ведь типы имеют разные размеры: int — 2 байта, char — 1 байт. Вероятно это связано с более высокой скоростью доступа к каждому четвёртому байту в памяти. Также именно этим объясняется

строка 12 на Рис. 10, в которой описывается количество нулей. Логично предположить, что это количество нулевых байтов, которые остаются при записи значения типа char.

Добавим статический массив в поля структуры struct1 (См. Рис. 14). Как и в предыдущей программе внутри main() мы поменяем некоторые значения полей, а теперь ещё и изменим несколько элементов в массиве. Листинги полученные на 32-bit и 64-bit также не имеют принципиальных отличий. Но теперь приведём в пример 32 битный листинг программы.

```
21  main:
22  .LFB0:
23      .cfi_startproc
24      endbr32
25      pushl   %ebp
26      .cfi_def_cfa_offset 8
27      .cfi_offset 5, -8
28      movl    %esp, %ebp
29      .cfi_def_cfa_register 5
30      call    __x86.get_pc_thunk.ax
31      addl    $_GLOBAL_OFFSET_TABLE_, %eax
32      movl    struct1@GOTOFF(%eax), %edx
33      addl    $5, %edx
34      movl    %edx, struct1@GOTOFF(%eax)
35      movl    struct1@GOTOFF(%eax), %edx
36      addl    $2, %edx
37      movl    %edx, 4+struct1@GOTOFF(%eax)
38      movb    $121, 8+struct1@GOTOFF(%eax)
39      movl    $10, 20+struct1@GOTOFF(%eax)
40      movl    $15, 28+struct1@GOTOFF(%eax)
41      movl    $0, %eax
42      popl    %ebp
43      .cfi_restore 5
44      .cfi_def_cfa 4, 4
45      ret
46      .cfi_endproc
```

Рисунок 13: Ф-я main() из program3c_32bit.s

```
3   struct example
4   {
5       int a;
6       int b;
7       char c;
8       int d[5];
9   };
```

Рисунок 14: Структура example из program3c.c

Сначала разберёмся со строчками 30, 31 (См. Рис. 13). Специальная функция __x86.get_pc_thunk.ax загружает свой обратный адрес, который является адресом инструкции addl, которая непосредственно следует за ней в регистр %eax. Затем мы добавляем к этому адресу значение константы _GLOBAL_OFFSET_TABLE_, которая вне зависимости от позиции в коде представляет разницу между между адресом инструкции, использующей _GLOBAL_OFFSET_TABLE_, и адресом глобальной таблицы смещений. Таким образом, %eax теперь указывает на глобальную таблицу смещений.

Символ @GOTOFF обращается к самой переменной относительно базы GOT (как удобный, но произвольный способ привязки). Таким образом с помощью данной команды мы по сути получаем адрес необходимой нам переменной. Однако если мы посмотрим на этот же код на 64 битной архитектуре, то мы увидим что то значительно более простое (См. Рис. 15). Здесь мы вместо всех этих действий с GOT просто загружаем поля структуры struct1 с фиксированного смещения от счётчика программы. PC-относительная адресация была добавлена вместе с расширениями 64-bit в архитектуру x86.

```
35      movl    %eax, 4+struct1(%rip)
36      movb    $121, 8+struct1(%rip)
37      movl    $10, 20+struct1(%rip)
38      movl    $15, 28+struct1(%rip)
```

Рисунок 15: Отрывок из ф-и main() из program3c_64bit.s

Теперь разберёмся в принципиальных моментах. Так как доступ к определённому элементу массива, находящемуся в поле структуры осуществляется так же как и доступ к

любому другому полю, то мы можем сделать вывод о том, что массив, так же как и все остальные поля структуры ,лежит в памяти непрерывно.

Теперь в программе program3d.c мы создадим небольшую структуру с двумя полями типа int и передадим ее в качестве аргумента в функцию, где попытаемся изменить одно из её полей. Результат нам известен. В структуру передастся экземпляр структуры, и соответственно все операции в функции будут проводиться с этим экземпляром, не меняя переданную структуру. В этом мы ещё раз удостоверимся просто посмотрев на вывод (убрав комментарии в файле program3d.c). Разберёмся в причинах. Сначала посмотрим на 64 битный листинг (См. Рис. 16 и 17).

```
14  simple_function:
15  .LFB0:
16      .cfi_startproc
17      endbr64
18      pushq   %rbp
19      .cfi_def_cfa_offset 16
20      .cfi_offset 6, -16
21      movq    %rsp, %rbp
22      .cfi_def_cfa_register 6
23      movq    %rdi, -8(%rbp)
24      movl    -8(%rbp), %eax
25      addl    $10, %eax
26      movl    %eax, -8(%rbp)
27      nop
28      popq    %rbp
29      .cfi_def_cfa 7, 8
30      ret
31      .cfi_endproc
```

Рисунок 16: Ф-я simple_function() из program3d_64bit.s

```
36  main:
37  .LFB1:
38      .cfi_startproc
39      endbr64
40      pushq   %rbp
41      .cfi_def_cfa_offset 16
42      .cfi_offset 6, -16
43      movq    %rsp, %rbp
44      .cfi_def_cfa_register 6
45      movq    struct1(%rip), %rax
46      movq    %rax, %rdi
47      call    simple_function
48      movl    $0, %eax
49      popq    %rbp
50      .cfi_def_cfa 7, 8
51      ret
52      .cfi_endproc
```

Рисунок 17: Ф-я main() из program3d_64bit.s

```
14  simple_function:
15  .LFB0:
16      .cfi_startproc
17      endbr32
18      pushl   %ebp
19      .cfi_def_cfa_offset 8
20      .cfi_offset 5, -8
21      movl    %esp, %ebp
22      .cfi_def_cfa_register 5
23      call    __x86.get_pc_thunk.ax
24      addl    $_GLOBAL_OFFSET_TABLE_, %eax
25      movl    8(%ebp), %eax
26      addl    $10, %eax
27      movl    %eax, 8(%ebp)
28      nop
29      popl    %ebp
30      .cfi_restore 5
31      .cfi_def_cfa 4, 4
32      ret
33      .cfi_endproc
```

Рисунок 18: Ф-я simple_function() из program3d_32bit.s

Внутри функции main() (См. Рис. 17) мы передаём адрес структуры struct1 в регистр %rax, а после в регистр %rdi, с которым мы и будем работать внутри вызываемой функции. Внутри simple_function() мы извлекаем из %rdi значение необходимого нам поля и записываем его в стек, место для которого выделяется сразу после объявления этой функции. Таким образом мы записываем всю структуру целиком в стек локальных переменных функции simple_function(). После мы изменяем уже значение на стеке (См. Рис. 16, строки 23-26). По завершению функции стек её

локальный переменных будет уничтожен и все изменения не повлияют на изначальную структуру.

Посмотрим на 32 битный листинг (См. Рис. 18 и 19). Здесь всё несколько более замысловато. За основу возьмём тот факт, что сама функция изменяет лишь свой экземпляр структуры, но никак не влияет на переданную ей структуру. А также то, что при вызове функции, она формирует свой стек локальных переменных. Посмотрим на Рис. 19. В строчках 49 и 50 мы кладём в стек поля структуры. Теперь взглянем на Рис. 18. Здесь, в строчках 25-27 функции simple_function(), мы перекладываем какое-то значение из стека в регистр %eax, увеличиваем его значение на 10, что и прописано в коде этой функции, а после возвращаем значение регистра в то же место на стеке. Руководствуясь вышеупомянутыми фактами, мы можем сказать, что этот стек не тот, с которым взаимодействовала функция main(). Иначе мы бы действительно поменяли поля структуры. Отсюда я могу сделать единственный вывод, в строчках 18, 21, 23 и 24 происходит копирование структуры в стек локальных переменных функции simple_function(). А дальнейшие операции сразу же происходят с копией нашей структуры.

```
38  main:
39  .LFB1:
40      .cfi_startproc
41      endbr32
42      pushl   %ebp
43      .cfi_def_cfa_offset 8
44      .cfi_offset 5, -8
45      movl    %esp, %ebp
46      .cfi_def_cfa_register 5
47      call    __x86.get_pc_thunk.ax
48      addl    $_GLOBAL_OFFSET_TABLE_, %eax
49      pushl   4+struct1@GOTOFF(%eax)
50      pushl   struct1@GOTOFF(%eax)
51      call    simple_function
52      addl    $8, %esp
53      movl    $0, %eax
54      leave
55      .cfi_restore 5
56      .cfi_def_cfa 4, 4
57      ret
58      .cfi_endproc
```

Рисунок 19: Ф-я main() из program3d_32bit.s

Теперь добавим структуру в возвращаемые значения функции (См. Program3e.c).

```
36  main:
37  .LFB1:
38      .cfi_startproc
39      endbr64
40      pushq   %rbp
41      .cfi_def_cfa_offset 16
42      .cfi_offset 6, -16
43      movq    %rsp, %rbp
44      .cfi_def_cfa_register 6
45      movq    struct1(%rip), %rax
46      movq    %rax, %rdi
47      call    simple_function
48      movq    %rax, struct1(%rip)
49      movl    $0, %eax
50      popq    %rbp
51      .cfi_def_cfa 7, 8
52      ret
53      .cfi_endproc
```

Рисунок 20: Ф-я main() из program3e_64bit.s

```
14  simple_function:
15  .LFB0:
16      .cfi_startproc
17      endbr64
18      pushq   %rbp
19      .cfi_def_cfa_offset 16
20      .cfi_offset 6, -16
21      movq    %rsp, %rbp
22      .cfi_def_cfa_register 6
23      movq    %rdi, -8(%rbp)
24      movl    -8(%rbp), %eax
25      addl    $10, %eax
26      movl    %eax, -8(%rbp)
27      movq    -8(%rbp), %rax
28      popq    %rbp
29      .cfi_def_cfa 7, 8
30      ret
31      .cfi_endproc
```

Рисунок 21: Ф-я simple_function() из program3e_64bit.s

Рассмотрим 64 битный листинг. Передача структуры в функцию осуществляется точно также, как и в предыдущем примере. Как мы можем видеть, возврат значения осуществляется при помощи регистра %rax. В функции simple_function() мы перемещаем изменённую структуру в этот регистр (См. Рис. 21, строка 27). А в функции main() мы присваиваем значение регистра структуре struct1 (См. Рис. 20, строка 48).

На 32-bit передача структуры происходит так же как и раньше, однако в отличие от 64-bit возврат значения осуществляется при помощи стека, поэлементным присвоением значений каждому полю структуры.

Задание 4

Функция со структурой в аргументе уже была рассмотрена в предыдущем задании. Поэтому теперь рассмотрим функцию с указателем на структуру в аргументе. Сначала 64 битный листинг.

```
38  main:
39  .LFB1:
40      .cfi_startproc
41      endbr64
42      pushq   %rbp
43      .cfi_def_cfa_offset 16
44      .cfi_offset 6, -16
45      movq    %rsp, %rbp
46      .cfi_def_cfa_register 6
47      leaq    struct1(%rip), %rdi
48      call    simple_function
49      movl    $0, %eax
50      popq    %rbp
51      .cfi_def_cfa 7, 8
52      ret
53      .cfi_endproc
```

Рисунок 22: Ф-я main() из program4b_64bit.s

```
14  simple_function:
15  .LFB0:
16      .cfi_startproc
17      endbr64
18      pushq   %rbp
19      .cfi_def_cfa_offset 16
20      .cfi_offset 6, -16
21      movq    %rsp, %rbp
22      .cfi_def_cfa_register 6
23      movq    %rdi, -8(%rbp)
24      movq    -8(%rbp), %rax
25      movl    (%rax), %eax
26      leal    10(%rax), %edx
27      movq    -8(%rbp), %rax
28      movl    %edx, (%rax)
29      nop
30      popq    %rbp
31      .cfi_def_cfa 7, 8
32      ret
33      .cfi_endproc
```

Рисунок 23: Ф-я simple_function() из program4b_64bit.s

Как и ранее мы используем регистр %rdi для передачи структуры в функцию (См. Рис. 22). Только теперь мы используем leaq вместо movq. Операция leaq загружает адрес struct1 в регистр %rdi (строка 47). В функции simple_function() мы также работаем с памятью, передавая адрес в стек локальных переменных функции (строка 23). После мы присваиваем этот адрес ещё и регистру %rax (строка 24). В строке 27 (См. Рис. 23) мы прибавляем 10 в первому полю структуры, используя операцию leal и записываем адрес в регистр %edx. А в строке 28 мы меняем значение первого поля структуры, используя его адрес.

Теперь посмотрим на 32 битный листинг. Функция simple_function() принципиально не имеет отличий, от уже приведённого примера на другой архитектуре. А в функции main() (См. Рис. 24) отличие есть. В предыдущем примере адрес в функцию передавался с помощью регистра. Здесь же мы отправляем полученный адрес на стек (строка 52). И в функции simple_function() извлекаем его оттуда.

Сейчас рассмотрим передачу структуры в функцию по ссылке. Для этого придётся перейти на язык C++. Посмотрев на 64 битные листинги мы можем увидеть, что функции simple_function() посимвольно идентичны в программах

program4b_64bit.s и program4c_64bit.s. В функции main() есть небольшие отличия. В последней программе (См. Рис. 25) выделяется место на стеке в строчке 54. А после (в строках 56-57) происходят несколько перемещений одной и той же величины. Мне кажется что всё это является недостатком данного уровня оптимизации и совершенно не важно для корректной работы программы. 32 битные листинги также не имею отличий, не считая уже названного ненужного выделения памяти на стеке. Отсюда мы можем сделать вывод о том, что ссылки и указатели внутри работают совершенно одинаково в случае передачи значения в функцию.

```
40  main:
41  .LFB1:
42      .cfi_startproc
43      endbr32
44      pushl   %ebp
45      .cfi_def_cfa_offset 8
46      .cfi_offset 5, -8
47      movl    %esp, %ebp
48      .cfi_def_cfa_register 5
49      call    __x86.get_pc_thunk.ax
50      addl    $_GLOBAL_OFFSET_TABLE_, %eax
51      leal    struct1@GOTOFF(%eax), %eax
52      pushl   %eax
53      call    simple_function
54      addl    $4, %esp
55      movl    $0, %eax
56      leave
57      .cfi_restore 5
58      .cfi_def_cfa 4, 4
59      ret
60      .cfi_endproc
```

Рисунок 24: Ф-я main() из program4b_32bit.s

Задание 5

Теперь мы будем проверять всё, что уже разобрали на прочность. Создадим большую структуру со статическим массивом среди ей полей (См. Рис. 26) и передадим её в функцию simple_function() (См. program5a.c).

```
45  main:
46  .LFB1523:
47      .cfi_startproc
48      endbr64
49      pushq   %rbp
50      .cfi_def_cfa_offset 16
51      .cfi_offset 6, -16
52      movq    %rsp, %rbp
53      .cfi_def_cfa_register 6
54      subq    $16, %rsp
55      leaq    struct1(%rip), %rax
56      movq    %rax, -8(%rbp)
57      movq    -8(%rbp), %rax
58      movq    %rax, %rdi
59      call    _Z15simple_functionR7example
60      movl    $0, %eax
61      leave
62      .cfi_def_cfa 7, 8
63      ret
64      .cfi_endproc
```

Рисунок 25: Ф-я main() из program4c_64bit.s

```
3   struct example
4   {
5       int a;
6       int b;
7       char c;
8       int d[10];
9   };
10
11  struct example struct1 = {1, 2, 'k', {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}};
```

Рисунок 26: Структура example и объявление структуры struct1 из program5a.c

```
47  main:
48  .LFB1:
49      .cfi_startproc
50      endbr64
51      pushq   %rbp
52      .cfi_def_cfa_offset 16
53      .cfi_offset 6, -16
54      movq    %rsp, %rbp
55      .cfi_def_cfa_register 6
56      pushq   %rbx
57      .cfi_offset 3, -24
58      subq    $56, %rsp
59      movq    %rsp, %rax
60      movq    struct1(%rip), %rcx
61      movq    8+struct1(%rip), %rbx
62      movq    %rcx, (%rax)
63      movq    %rbx, 8(%rax)
64      movq    16+struct1(%rip), %rcx
65      movq    24+struct1(%rip), %rbx
66      movq    %rcx, 16(%rax)
67      movq    %rbx, 24(%rax)
68      movq    32+struct1(%rip), %rcx
69      movq    40+struct1(%rip), %rbx
70      movq    %rcx, 32(%rax)
71      movq    %rbx, 40(%rax)
72      movl    48+struct1(%rip), %edx
73      movl    %edx, 48(%rax)
74      call    simple_function
75      addq    $56, %rsp
76      movl    $0, %eax
77      movq    -8(%rbp), %rbx
78      leave
79      .cfi_def_cfa 7, 8
80      ret
81      .cfi_endproc
```

Рисунок 27: Ф-я main() из program5a_64bit.s

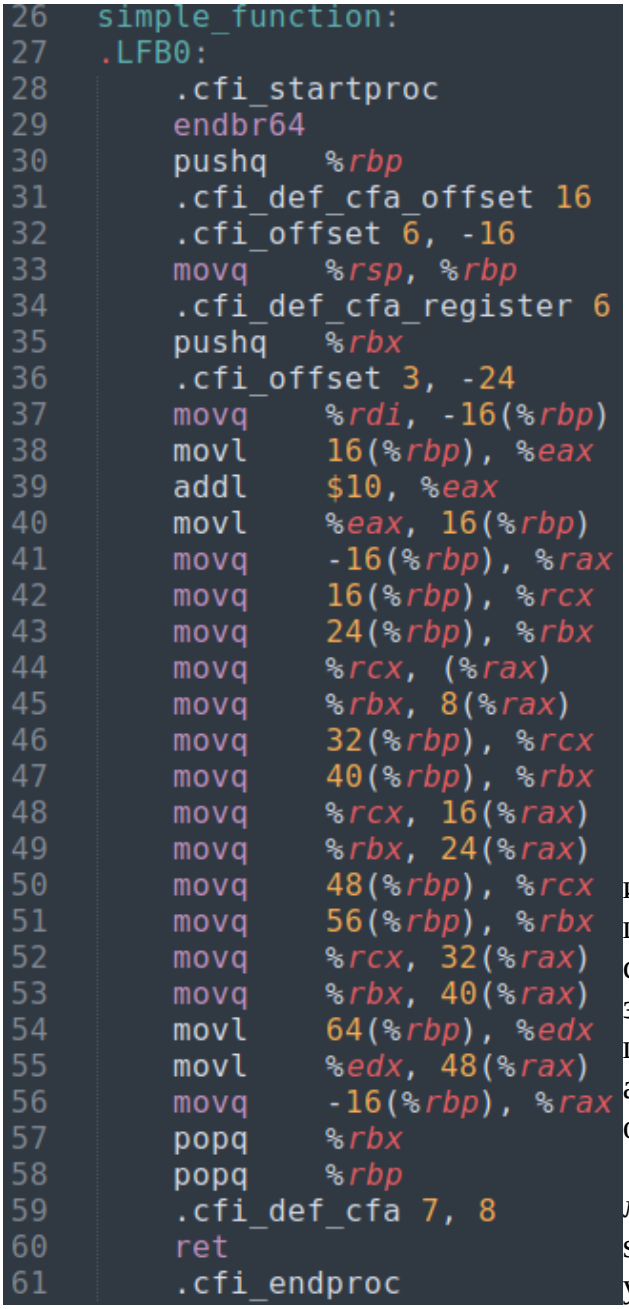
```
50  main:
51  .LFB1:
52      .cfi_startproc
53      endbr32
54      pushl   %ebp
55      .cfi_def_cfa_offset 8
56      .cfi_offset 5, -8
57      movl    %esp, %ebp
58      .cfi_def_cfa_register 5
59      call    __x86.get_pc_thunk.ax
60      addl    $_GLOBAL_OFFSET_TABLE_, %eax
61      pushl   48+struct1@GOTOFF(%eax)
62      pushl   44+struct1@GOTOFF(%eax)
63      pushl   40+struct1@GOTOFF(%eax)
64      pushl   36+struct1@GOTOFF(%eax)
65      pushl   32+struct1@GOTOFF(%eax)
66      pushl   28+struct1@GOTOFF(%eax)
67      pushl   24+struct1@GOTOFF(%eax)
68      pushl   20+struct1@GOTOFF(%eax)
69      pushl   16+struct1@GOTOFF(%eax)
70      pushl   12+struct1@GOTOFF(%eax)
71      pushl   8+struct1@GOTOFF(%eax)
72      pushl   4+struct1@GOTOFF(%eax)
73      pushl   struct1@GOTOFF(%eax)
74      call    simple_function
75      addl    $52, %esp
76      movl    $0, %eax
77      leave
78      .cfi_restore 5
79      .cfi_def_cfa 4, 4
80      ret
81      .cfi_endproc
```

Рисунок 28: Ф-я main() из progra5a_32bit.s

Теперь посмотрим на 64 битный листинг (См. Рис. 27). По сравнению с тем, что было, когда мы передавали в функцию небольшую структуру, состоящую из двух полей типа int, отличия есть. Ранее мы предавали структуру, используя регистр, из которого внутри функции и извлекали необходимые нам данные. Сейчас же мы передаём структуру используя стек. Мы последовательно загружаем каждые 8 байт структуры в регистры %rcx и %rbx, а их уже отправляем в стек. Листинг самой функции simple_function() отличается незначительно: раньше мы сначала извлекали из переданного регистра необходимые данные, а сейчас сразу же начинаем работать со стеком.

Рассмотрим 32 битный листинг. Отличий с тем, что было, когда структура была небольшой практически нет, за исключением того, что теперь нам нужно передать в стек больше элементов. Но есть различие с 64 битной архитектурой. Здесь мы передаём на стек каждый четвёртый байт структуры, начиная с её конца. А в 64-bit, мы передаём на стек каждый восьмой байт. Похоже что это происходит лишь из за того, что 64-bit архитектура работает с 64 битными регистрами с префиксом r, которые в два раза вместительнее, нежели 32 битные с префиксом e.

Теперь добавим большую структуру в возвращаемые значения функции. В архитектуре 64-bit изменились как функция main(), так и функция simple_function(). Ранее мы возвращали небольшую структуру из двух полей типа int, поэтому мы просто записывали структуру в регистр, а после извлекали из него данные. Теперь же мы не можем так поступить, а от того снова будем пользоваться стеком.



```
26  simple_function:
27  .LFB0:
28      .cfi_startproc
29      endbr64
30      pushq   %rbp
31      .cfi_def_cfa_offset 16
32      .cfi_offset 6, -16
33      movq    %rsp, %rbp
34      .cfi_def_cfa_register 6
35      pushq   %rbx
36      .cfi_offset 3, -24
37      movq    %rdi, -16(%rbp)
38      movl    16(%rbp), %eax
39      addl    $10, %eax
40      movl    %eax, 16(%rbp)
41      movq    -16(%rbp), %rax
42      movq    16(%rbp), %rcx
43      movq    24(%rbp), %rbx
44      movq    %rcx, (%rax)
45      movq    %rbx, 8(%rax)
46      movq    32(%rbp), %rcx
47      movq    40(%rbp), %rbx
48      movq    %rcx, 16(%rax)
49      movq    %rbx, 24(%rax)
50      movq    48(%rbp), %rcx
51      movq    56(%rbp), %rbx
52      movq    %rcx, 32(%rax)
53      movq    %rbx, 40(%rax)
54      movl    64(%rbp), %edx
55      movl    %edx, 48(%rax)
56      movq    -16(%rbp), %rax
57      popq    %rbx
58      popq    %rbp
59      .cfi_def_cfa 7, 8
60      ret
61      .cfi_endproc
```

Рисунок 29: Ф-я simple_function() из program5b_64bit.s

```
 99      call    simple_function
100      addq    $56, %rsp
101      movq    -96(%rbp), %rax
102      movq    -88(%rbp), %rdx
103      movq    %rax, struct1(%rip)
104      movq    %rdx, 8+struct1(%rip)
105      movq    -80(%rbp), %rax
106      movq    -72(%rbp), %rdx
107      movq    %rax, 16+struct1(%rip)
108      movq    %rdx, 24+struct1(%rip)
109      movq    -64(%rbp), %rax
110      movq    -56(%rbp), %rdx
111      movq    %rax, 32+struct1(%rip)
112      movq    %rdx, 40+struct1(%rip)
113      movl    -48(%rbp), %eax
114      movl    %eax, 48+struct1(%rip)
115      movl    $0, %eax
116      movq    -24(%rbp), %rsi
117      xorq    %fs:40, %rsi
118      je  .L5
119      call    __stack_chk_fail@PLT
```

Рисунок 30: Отрывок из ф-и main() из program5b_64bit.s

Внутри функции simple_function() мы изменяем структуру и записываем её на стек с шагом в 8 байт (См. Рис. 29). А в main() мы считываем со стека структуру и изменяем значения начальной структуры struct1 так же с шагом в 8 байт (См. Рис. 30). На 32 битной архитектуре не наблюдается принципиальных отличий, кроме уже вышеупомянутых.

Если мы поместим большую структуру в локальные переменные функции simple_function(), то не произойдёт ничего удивительного. На обоих архитектурах поля структуры будут последовательно записываться на стек локальных переменных функции. А после со структурой на стеке будут осуществлены необходимые операции, заданные нами.

Если же мы изменим размеры структуры, пусть в ней снова будут два поля типа int, и поместим эту структуру в локальные переменные функции simple_function(), то результат останется таким же. Структура запишется в стек и будет нужным образом изменена.

Задание 6

Создадим рекурсивную функцию factorial() с одним аргументом, вычисляющую факториал числа, и возвращающую значение факториала в вызывающий код. Сравним листинг, полученный на разных архитектурах. Сразу же в глаза бросается структура кода. В обоих листингах теперь используются метки внутри вызываемой функции. Основная логика идентична в обоих программах.

Как мы видим на обоих архитектурах происходит работа со стеком. Посмотрим на его размер.

```
 5  factorial:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr64
 9      pushq   %rbp
10      .cfi_def_cfa_offset 16
11      .cfi_offset 6, -16
12      movq    %rsp, %rbp
13      .cfi_def_cfa_register 6
14      subq    $32, %rsp
15      movl    %edi, -20(%rbp)
16      cmpl    $1, -20(%rbp)
17      jne .L2
18      movl    $1, %eax
19      jmp .L3
20  .L2:
21      movl    -20(%rbp), %eax
22      subl    $1, %eax
23      movl    %eax, %edi
24      call    factorial
25      movl    -20(%rbp), %edx
26      imull   %edx, %eax
27      movl    %eax, -4(%rbp)
28      movl    -4(%rbp), %eax
29  .L3:
30      leave
31      .cfi_def_cfa 7, 8
32      ret
33      .cfi_endproc
```

Рисунок 31: Ф-я factorial() из program6_64bit.s

```
 5  factorial:
 6  .LFB0:
 7      .cfi_startproc
 8      endbr32
 9      pushl   %ebp
10      .cfi_def_cfa_offset 8
11      .cfi_offset 5, -8
12      movl    %esp, %ebp
13      .cfi_def_cfa_register 5
14      subl    $24, %esp
15      call    __x86.get_pc_thunk.ax
16      addl    $_GLOBAL_OFFSET_TABLE_, %eax
17      cmpl    $1, 8(%ebp)
18      jne .L2
19      movl    $1, %eax
20      jmp .L3
21  .L2:
22      movl    8(%ebp), %eax
23      subl    $1, %eax
24      subl    $12, %esp
25      pushl   %eax
26      call    factorial
27      addl    $16, %esp
28      movl    8(%ebp), %edx
29      imull   %edx, %eax
30      movl    %eax, -12(%ebp)
31      movl    -12(%ebp), %eax
32  .L3:
33      leave
34      .cfi_restore 5
35      .cfi_def_cfa 4, 4
36      ret
37      .cfi_endproc
```

Рисунок 32: Ф-я factorial() из program6_32bit.s

Сначала для 64 битной архитектуры (См. Рис. 31). В строчке 14 мы отнимаем от регистра %rsp 32, тем самым, выделяя на стеке 32 байта. После мы лишь правильно завершаем работу со стеком, используя операцию leave в строке 30.

В 32-bit мы сначала выделяем 24 байта на стеке в строчке 14 (См. Рис. 32), а после аллоцируем ещё 12 байт в строке 24. Также мы освобождаем 16 байт стека в 27 строке.

Если мы посмотрим на функцию main() на обоих архитектурах, то также заметим, что память в 64-bit выделяется только один раз и больше размер стека не изменяется до конца выполнения функции. В 32-bit в main() происходит постоянное взаимодействие с регистром %esp: мы то выделяем, то освобождаем память.

Резюмируя, можно сказать, что работа со стеком устроена намного эффективнее на 64 битной архитектуре, нежели на 32 битной. Так как частое и многократное выделение и перевыделение памяти занимают очень много времени.