

Лабораторная работа 4 (0100 = 4)

Модули и функции

Цель работы: изучить стандартные соглашения о вызовах и их соответствие платформам, научиться комбинировать функции на C/C++ и ассемблере. Каждый из модулей в заданиях Л4 реализуется либо на чистом ассемблере, либо на чистом C/C++ (без ассемблерных вставок).

Задание Л4.№1

Разработайте ассемблерную функцию $f1()$, вычисляющую целое выражение от двух целых аргументов (таблица Л4.1), а также головную программу на языке C/C++, использующую разработанную функцию.

Бонус +1 балл, если вычисление производится одной командой *lea*.

Чем отличается и в чём схожа передача целочисленных параметров в System V amd64 psABI и Microsoft 64?

Вариант целочисленных выражений для расчёта: Вариант 1 $f1(x, y) = -7 + x + 8y$

Отличия в передаче целочисленных параметров

Особенность	System V AMD64 ABI	Microsoft x64 Calling Convention
Передача целочисленных параметров	Первые 6 целочисленных параметров в регистрах RDI, RSI, RDX, RCX, R8, R9	Первые 4 целочисленных параметра в регистрах RCX, RDX, R8, R9

Реализация:

В силу ограничений выбранных мной инструментов для выполнения ЛР, я провёл небольшой анализ встретившихся проблем и особенностей:

1. Ассемблер/вставки

В средах www.jdoodle.com и godbolt.org я не обнаружил возможности каким-либо образом добавить отдельные asm-файлы, поэтому для запуска всего ассемблерного кода было принято использовать обёртки `inline asm`, вопреки цели работы. Но чтобы не остаться в стороне, привожу примерные решения, которые я бы использовал в обычном режиме.

2. Соглашение о вызовах (Calling convention)

- **Microsoft x64 (MSVC):**
 - Параметры передаются в регистрах:
 - 1-й параметр — RCX
 - 2-й параметр — RDX
 - 3-й параметр — R8
 - 4-й параметр — R9
 - Возврат результата — в RAX.
 - Стек выравнивается по 16 байтам, при вызове выделяется теневое пространство (shadow space) — 32 байта под параметры, даже если они в регистрах.
- **System V amd64 (gcc/clang):**
 - Параметры передаются в регистрах:
 - 1-й параметр — RDI
 - 2-й параметр — RSI
 - 3-й параметр — RDX
 - 4-й параметр — RCX
 - 5-й параметр — R8
 - 6-й параметр — R9
 - Возврат результата — в RAX.
 - Нет теневого пространства, но стек тоже выравнивается по 16 байтам.

3. Поддержка inline asm

- **x64 MSVC:**
 - `inline asm` не поддерживается.
 - поэтому часто используют `__declspec(naked)` или отдельные asm-файлы, либо использовать `intrinsics`, которые в моих инструментах тоже не заработали.
 - Эти ограничения связано с архитектурой и компилятором.
- **System V amd64 (gcc/clang):**
 - Поддерживается полноценный `inline asm` в C-коде.
 - Можно писать ассемблер прямо внутри функции, использовать операнды, связывать с C-переменными.

4. Синтаксис ассемблера

- **MSVC:**
 - Используется синтаксис MASM (Intel-синтаксис).
 - Пример: `lea rax, [rcx + rdx*8 - 7]`
- **GCC/Clang:**
 - Используется синтаксис GAS (AT&T-синтаксис) по умолчанию, где регистры пишутся с % и порядок операндов обратный.
 - Пример: `lea (%rdi, %rsi, 8), %rax`
 - Но можно переключить на Intel-синтаксис через директивы.

Реализация функции f1 на ассемблере (System V amd64)

Файл f1.S:

```
1. .global f1
2. f1:
3.     lea     rdi, [rdi + rsi*8 - 7]  # rdi = x + 8*y - 7
4.     mov     rax, rdi               # возвращаем результат в rax
5.     ret
```

f1.asm (Microsoft x64)

Файл f1.asm:

```
1. ; f1.asm ; Функция: int64_t f1(int64_t x, int64_t y)
2. ; Вычисляет: f1 = -7 + x + 8*y
3. ; Microsoft x64 calling convention:
4. ; RCX = x, RDX = y
5. ; Возврат результата: RAX
6.
7. PUBLIC f1
8. .CODE
9. f1 PROC
10.     lea rax, [rcx + rdx*8 - 7] ; rax = x + 8*y - 7
11.     ret
12. f1 ENDP
13. END
```

Или

Листинг для asm inline:

```
1. __declspec(naked) int64_t f1(int64_t x, int64_t y) {
2.     __asm {
3.         ; В Microsoft x64 параметры:
4.         ; RCX = x, RDX = y lea rax, [rcx + rdx*8 - 7]
5.         ; rax = x + 8*y - 7
6.         ret
7.     }
8. }
```

Что получилось:

```
Задание №1
=====
SystemInfo
=====
ОС: Linux
Архитектура процессора: x86_64 (64-бит)
Compiler: GCC
Version: 13.2.1
=====
f1 (
x= 0000000000000005 101 5 +5 +0x0.0000000000005p-1022 +2.470328e-323 +0.00
,
y= 0000000000000003 11 3 +3 +0x0.0000000000003p-1022 +1.482197e-323 +0.00
) =
0000000000000016 10110 22 +22 +0x0.0000000000016p-1022 +1.086944e-322 +0.00
=====
```

Рис. 1: Результат выполнения в System V amd64 (gcc, linux)

Листинг:

Файл task4_1.c:

```
1. #ifndef task4_1_H
2. #define task4_1_H
3.
4. #include <stdio.h>
5. #include <stdint.h>
6.
7. void print16(void *p);
8. void print32(void *p);
9. void print64(void *p);
10.
11. void printSystemInfo();
12.
13. int64_t f1(int64_t x, int64_t y);
14.
15. void run_task4_1()
16. {
17.     printf("\nЗадание №1\n");
18.     printf("=====");
19.     printSystemInfo();
20.     printf("=====\n");
21.
22.     int64_t x = 5;
23.     int64_t y = 3;
24.     int64_t result = f1(x, y);
25.
26.     printf("f1 (\nx= ");
27.     print64(&x);
28.     printf(", \ny= ");
29.     print64(&y);
30.     printf(") =\n");
31.     print64(&result);
32.
33.     printf("\n=====\n");
34. }
35.
36. int64_t f1(int64_t x, int64_t y) {
37.     int64_t result;
38.     __asm__ (
39.         "lea (%1, %2, 8), %0\n\t" // result = x + 8*y
40.         "sub $7, %0" //
41.         : "=r" (result) // output operand
42.         : "r" (x), "r" (y) // input operands
43.     );
44.     return result;
45. }
46.
47. #endif
```

Задание Л4.№2

Разработайте ассемблерную функцию $f2()$, вычисляющую выражение от двух чисел с плавающей запятой двойной точности x и y (таблица Л4.2), используя команды AVX $vsubsd$ и $vdivsd$ или их SSE-аналоги $subsd$ и $divsd$, а также головную программу на языке C/C++.

Чем отличается и в чём схожа передача параметров с плавающей запятой в System V amd64 psABI и Microsoft 64 вообще? Различается ли реализация $f2()$?

Вариант целочисленных выражений для расчёта: Вариант 1 $f2(x, y) = x - y$

Отличия в передаче параметров с плавающей запятой

Особенность	System V AMD64 ABI	Microsoft x64 Calling Convention
Регистры для плавающих параметров	Первые 8 параметров с плавающей запятой передаются в регистрах XMM0–XMM7	Первые 4 параметра с плавающей запятой передаются в регистрах XMM0–XMM3
Обработка аргументов с плавающей точкой	Для variadic-функций (например, printf) требуется установить регистр AL в количество использованных векторных регистров (XMM0-XMM7).	---

```
Задание №2
=====
SystemInfo
=====
ОС: Linux
Архитектура процессора: x86_64 (64-бит)
Compiler: GCC
Version: 13.2.1
=====
f2 (
x= 4014000000000000 0 4617315517961601024 +4617315517961601024 +0x1.4p+2 +5.000000e+00 +5.00
,
y= 4008000000000000 0 4613937818241073152 +4613937818241073152 +0x1.8p+1 +3.000000e+00 +3.00
) =
4000000000000000 0 4611686018427387904 +4611686018427387904 +0x1p+1 +2.000000e+00 +2.00
=====
```

Рис. 2: результат выполнения вставки

Листинг:

Файл task4_2.c:

```
1. #ifndef task4_2_H
2. #define task4_2_H
3.
4. #include <stdio.h>
5. #include <stdint.h>
6.
7. void print16(void *p);
8. void print32(void *p);
9. void print64(void *p);
10.
11. void printSystemInfo();
12.
13. double f2(double x, double y);
14.
15. void run_task4_2()
16. {
17.     printf("\nЗадание №2\n");
18.     printf("=====");
19.     printSystemInfo();
20.     printf("=====\n");
21.
22.     double x = 5.0, y = 3.0;
23.     double result = f2(x, y);
24.
25.     printf("f2 (nx= ");
26.     print64(&x);
27.     printf(", ny= ");
28.     print64(&y);
29.     printf(") =\n");
30.     print64(&result);
31.
32.     printf("\n===== \n");
33. }
34.
35. double f2(double x, double y) {
36.     double result;
37.     __asm__ (
38.         "vsubsd %2, %1, %0"
39.         : "=x" (result)    // output in xmm register
40.         : "x" (x), "x" (y) // inputs in xmm registers
41.         );
42.     return result;
43. }
44.
45. #endif
```

Код f2.asm (Microsoft x64):

```
1. ; f2.asm
2. ; Функция double f2(double x, double y) = x - y
3. ; Параметры: x в xmm0, y в xmm1
4. ; Результат: xmm0
5.
6. .code
7. PUBLIC f2
8.
9. f2 PROC
10.     vsubsd xmm0, xmm0, xmm1 ; xmm0 = xmm0 - xmm1
11.     ret
12. f2 ENDP
13.
14. END
```

Задание Л4.№3

Разработайте программу, целиком написанную на ассемблере, которая печатает группу, номер и состав команды при помощи функции *puts()* библиотеки *libc* (аналогично заданию Л1.№1).

Обратите внимание, что для вложенного вызова функции мало передать параметры через соответствующие регистры — необходимо полностью соблюсти соглашение, соответствующее платформе.

Используйте то, что стартовый код *libc* также соблюдает это соглашение, и до вызова *main()* (по обоим 64-битным соглашениям) выравнивает стек на 16 байт (верхняя пунктирная линия 16х на рис. Л4.1). После вызова *main()* на стек ложится адрес возврата (рис. Л4.1, а). Состояние стека перед вызовом *puts()* для двух соглашений показано на рис. Л4.1, б) и в).

Чем отличаются и в чём совпадают требования соглашений System V amd64 psABI и Microsoft 64 в случае функции с постоянным числом параметров — такой, как *puts()*, а также как разработанные выше *f1()* и *f2()*?

1. Общие требования к соглашениям System V amd64 psABI и Microsoft 64 для функций с фиксированным числом параметров

Особенность	System V AMD64 psABI	Microsoft x64 calling convention
Передача параметров через регистры	Первые 6 целочисленных параметров — в регистрах RDI, RSI, RDX, RCX, R8, R9; параметры с плавающей точкой — в XMM0–XMM7	Первые 4 параметра (целочисленные и указатели) — в RCX, RDX, R8, R9; параметры с плавающей точкой — в XMM0–XMM3
Возврат значения	Целочисленные и указатели — через RAX; плавающая точка — через XMM0	
Теневое пространство (shadow space)	Отсутствует	Перед вызовом функции вызывающая сторона выделяет 32 байта в стеке (4 * 8 байт) под параметры (даже если они передаются в регистрах), чтобы функция могла использовать это место для сохранения параметров
Сохранение регистров	Callee сохраняет RBX, RBP, R12–R15; Caller сохраняет RDI, RSI, RDX, RCX, R8–R11	Callee сохраняет RBX, RBP, RDI, RSI, R12–R15; Caller сохраняет RAX, RCX, RDX, R8–R11
Выравнивание стека	Перед вызовом call указатель стека RSP должен быть выровнен по 16-байтной границе. Например, перед вызовом puts() может потребоваться корректировка стека (например, sub rsp, 8).	После выделения shadow space и других данных стек также должен быть выровнен по 16-байтной границе перед call. Например, выделение 32 байт shadow space + 8 байт для выравнивания (итого sub rsp, 40).
Стек после вызова	Если параметры передаются через стек, вызывающий код отвечает за их очистку	

2. Особенности вызова puts() (один параметр — указатель на строку)

Аспект	System V AMD64 psABI	Microsoft x64 calling convention
Передача параметра	В регистре RDI	В регистре RCX
Возврат значения	Регистр RAX для возврата значения (количество выведенных символов или ошибка).	
Очистка стека	для puts() это не актуально, так как параметр один и передаётся через регистр	
Теневое пространство	Отсутствует	Обязательно выделение 32 байт теневого пространства
Параметры в стеке	Нет (если параметр помещается в регистр)	Нет (параметры в регистре, но резервируется теневое пространство)


```
Active code page: 65001
\gnu-asm-main\Задания\ЛР>chcp 65001
\gnu-asm-main\Задания\ЛР>lr4_3
Группа: ПИН-31Д
Номер: 17
Выполнил: Понкращенко Д.Б.
```

Рис. 3: выполнение ассемблерной программы на локальном Windows, amd64 системе.

Листинг:

Файл lr4_3.S:

```
1. # file: lr4_3.S
2. # Ассемблерная программа, вызывающая puts() для вывода строки
3. # Используется System V amd64 psABI (Linux/macOS)
4.
5. .section .data
6. msg:
7.     .asciz "Группа: ПИН-31Д\nНомер: 17\nВыполнил: Понкращенко Д.Б.\n"
8.
9. .section .text
10. .global main
11. .extern puts
12.
13. main:
14.     sub rsp, 8           # rsp выровнен по 16 байтам при входе в main
15.
16.     lea msg(%rip), %rdi  # Первый параметр - адрес строки в RDI
17.     call puts            # вызов puts
18.
19.     movl $0, %eax        # код возврата 0
20.     ret
```

Файл lr4_3.asm:

```
1. ; lr4_3.asm
2. ; Ассемблерная программа, вызывающая puts() для вывода строки
3. ; Microsoft x64 calling convention (Windows)
4.
5. .data
6. msg db "Группа: ПИН-31Д", 10, "Номер: 17", 10, "Выполнил: Понкращенко Д.Б.", 0
7.
8. .code
9. public main
10. extern puts: proc
11.
12. main PROC
13.     sub rsp, 40          ; Выделяем 32 байта shadow space + 8 байт для выравнивания
14.
15.     lea rcx, msg         ; Первый параметр - адрес строки в RCX
16.     call puts            ; Вызов puts
17.
18.     add rsp, 40          ; Восстанавливаем rsp
19.
20.     mov eax, 0           ; Возвращаем 0
21.     ret
22. main ENDP
23. end
```

Задание Л4.№4.

Разработайте на языке C/C++ программу, которая:

а) включает пять локальных для *main()* переменных:

- *i16* — 16-битное целое *short/unsigned short*;
- *i32* — 32-битное целое *int/unsigned int*;
- *i64* — 64-битное целое *long long/unsigned long long*;
- *f32* — 32-битное число с плавающей запятой *float*;
- *f64* — 64-битное число с плавающей запятой *double*;

б) запрашивает значения *i16*, *i32*, *i64*, *f32*, *f64* одним вызовом *scanf()*;

в) печатает значения *i16*, *i32*, *i64*, *f32*, *f64* одним вызовом *printf()*;

Убедитесь, что ввод-вывод работает корректно.

Разработайте аналогичную программу, целиком реализованную на ассемблере. Форматные строки и вообще вызовы *scanf()* и *printf()* должны совпадать с C/C++-прототипом. Локальные переменные на ассемблере, естественно, не имеют имён, но в документации имена используются и здесь совпадают с прототипом.

Обдумайте, как разместить пять локальных переменных *i16*, *i32*, *i64*, *f32*, *f64* в четырёх 64-битных стековых словах с учётом выравнивания — то есть адрес переменной должен быть кратен как минимум её размеру. Изобразите фрагмент стека с переменными аналогично рис. Л4.1; укажите, где какая переменная.

Поместятся ли в регистры все параметры *scanf()*? Какие параметры *scanf()* будут стековыми? Изобразите стек перед вызовом *scanf()*. Расположение переменных друг относительно друга должно совпасть с разработанным ранее.

Этот и последующие рисунки для Л4.№4 делайте для укороченного пролога/эпилога (только уменьшение *rsp* в прологе и увеличение в эпилоге; локальные переменные адресуются относительно *rsp*). Изобразите стек перед вызовом *printf()*, причём расположение АВ из *main()* и переменных должно совпасть с АВ и переменными перед вызовом *scanf()*. Сколько стековых слов (и, соответственно, сколько байтов) необходимо зарезервировать в прологе, чтобы хватило и на переменные, и на *scanf()*, и на *printf()*, и были бы соблюдены все требования соглашения? Исправьте изображения стека перед вызовами *scanf()* и *printf()* с учётом сделанных выводов. Рассчитайте смещения переменных *i16*, *i32*, *i64*, *f32*, *f64*, а также стековых параметров *scanf()* и *printf()* относительно *rsp*.

Реализуйте по разработанным изображениям стека программу.

Чем отличаются и в чём совпадают соглашения System V amd64 psABI и Microsoft 64 в случае функции с переменным числом параметров, как *scanf()* и *printf()*?

Особенности соглашений System V amd64 и Microsoft 64 для функций с переменным числом параметров

Особенность	System V AMD64 psABI	Microsoft x64 calling convention
Обработка аргументов с плавающей точкой	Для variadic-функций (например, <i>printf</i>) требуется установить регистр AL в количество использованных векторных регистров (XMM0-XMM7).	---
Передача параметров через регистры	Первые 6 целочисленных параметров — в регистрах RDI, RSI, RDX, RCX, R8, R9; параметры с плавающей точкой — в XMM0–XMM7	Первые 4 параметра (целочисленные и указатели) — в RCX, RDX, R8, R9; параметры с плавающей точкой — в XMM0–XMM3

```

Задание №4
=====
SystemInfo
=====
ОС: Linux
Архитектура процессора: x86_64 (64-бит)
Compiler: GCC
Version: 13.2.1
=====
10 100 1000 2.3 5.6
i16=10 i32=100 i64=1000 f32=2.300000 f64=5.600000
=====

```

Рис. 4: выполнение task4_4.c GCC Linux

```

10 100 1000 2.3 5.6
i16: 10, i32: 100, i64: 1000, f32: 2.300000, f64: 5.600000
|

```

Рис. 5: выполнение lr4_4.S System V AMD64 ABI (Linux/macOS)

Листинг:

Файл task4_4.c:

```

13. void run_task4_4()
14. {
20.     short i16;
21.     int i32;
22.     long long i64;
23.     float f32;
24.     double f64;
25.
26.     // Форматная строка для scanf: %hd (short), %d (int), %lld (long long), %f (float), %lf
27.     scanf("%hd %d %lld %f %lf", &i16, &i32, &i64, &f32, &f64);
28.
29.     // Форматная строка для printf – выводим в том же порядке
30.     printf("i16=%hd i32=%d i64=%lld f32=%f f64=%lf\n", i16, i32, i64, f32, f64);
33. }

```

Файл lr4_4.S:

```

1. # file: lr4_4.S
2. # Программа с пятью локальными переменными, вводом scanf и выводом printf
3. # System V AMD64 ABI (Linux/macOS), укороченный пролог (адресация от rsp)
4.
5.     .section .data
6. format_in: .string "%hd %d %lld %f %lf" # Формат ввода: short, int, long long, float, double
7. format_out: .string "i16=%hd i32=%d i64=%lld f32=%f f64=%lf\n" # Формат вывода
8.
9.     .section .text
10.    .global main
11.    .extern scanf
12.    .extern printf
13.
14. main:
15.
16.    # Резервируем место на стеке для локальных переменных
17.    subq    $40, %rsp    # Резервируем 40 байт (2 + 4 + 8 + 4 + 8 = 26 байт и 14 выравнивание)
18.
19.    # Адреса локальных переменных относительно rsp:
20.    # i16: 24(%rsp) (выравнивание 2)
21.    # i32: 20(%rsp) (выравнивание 4)
22.    # i64: 16(%rsp) (выравнивание 4)
23.    # f32: 8(%rsp) (выравнивание 8)
24.    # f64: 0(%rsp) (выравнивание 8)
25.
26.    lea     format_in(%rip), %rdi    # 1-й аргумент: форматная строка
27.    lea     24(%rsp), %rsi           # 2-й: адрес i16
28.    lea     20(%rsp), %r8            # 5-й: адрес f32
29.    lea     16(%rsp), %rdx           # 3-й: адрес i32
30.    lea     8(%rsp), %r9             # 6-й: адрес f64
31.    lea     0(%rsp), %rcx            # 4-й: адрес i64
32.
33.    call    scanf                    # Вызов scanf

```

```

34.
35.     leaq    format_out(%rip), %rdi    # 1-й аргумент: форматная строка
36.     movswl  24(%rsp), %rsi           # 2-й: i16
37.     movss   20(%rsp), %xmm0          # 5-й: f32 (загружаем как float)
38.     cvtss2sd %xmm0, %xmm0            # Конвертируем float в double
39.     movl    16(%rsp), %rdx           # 3-й: i32
40.     movsd   8(%rsp), %xmm1           # 6-й: f64
41.     movq    0(%rsp), %rcx            # 4-й: i64
42.     movl    $2, %eax                 # Указываем количество XMM-аргументов (2)
43.
44.     call    printf                   # Вызов printf
45.
46.     movq    %rbp, %rsp               # Восстанавливаем указатель стека
47.     movl    $0, %eax                 # Возвращаем 0
48.     ret

```

Файл lr4_4.asm:

```

1. ; lr4_4.asm
2. ; Программа с пятью локальными переменными, вводом scanf и выводом printf
3. ; Microsoft x64 calling convention (Windows), укороченный пролог (адресация от rsp)
4. .data
5. format_in db "%hd %d %lld %f %lf", 0
6. format_out db "i16=%hd i32=%d i64=%lld f32=%f f64=%lf", 10, 0
7.
8. .code
9. public main
10. extern scanf: proc
11. extern printf: proc
12.
13. main PROC
14.     sub rsp, 32 ; Резервируем 32 байт
15.
16.     ; Адреса локальных переменных относительно rsp:
17.     ; i16: rsp+0
18.     ; i32: rsp+2
19.     ; i64: rsp+6
20.     ; f32: rsp+14
21.     ; f64: rsp+18
22.
23.     ; Передача параметров в scanf
24.     lea rcx, format_in                ; форматная строка - 1-й параметр в rcx
25.     lea rdx, [rsp]                   ; i16 - 2-й параметр в rdx
26.     lea r8, [rsp+2]                   ; i32 - 3-й параметр в r8
27.     lea r9, [rsp+6]                   ; i64 - 4-й параметр в r9
28.     lea xmm0, dword ptr [rsp+14]      ; f32 - 5-й параметр в xmm0
29.     lea xmm1, qword ptr [rsp+18]      ; f64 - 6-й параметр в xmm1
30.
31.     call scanf
32.
33.     ; Передача параметров в printf
34.     lea rcx, format_out                ; форматная строка - 1-й параметр в rcx
35.     movsx rвч, word ptr [rsp]          ; i16 -> rdx
36.     mov r8, dword ptr [rsp+2]          ; i32 -> r8
37.     mov r9, qword ptr [rsp+6]          ; i64 -> r9
38.     movss xmm0, dword ptr [rsp+14]     ; f32 -> xmm0
39.     movsd xmm1, qword ptr [rsp+18]     ; f64 -> xmm1
40.
41.     call printf
42.
43.     add rsp, 32 ; Освобождаем стек
44.     mov eax, 0 ; Возвращаем 0
45.     ret
46. main ENDP
47. end

```

Задание Л4.№5.

Разработайте программу, целиком реализованную на ассемблере, аналогичную заданию Л4.№4, но с классическим прологом/эпилогом (локальные переменные адресуются относительно *rbp*).

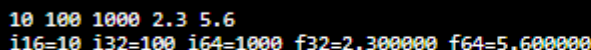
Для этого также изобразите стек перед вызовами *scanf()* и *printf()* с учётом копии старого значения *rbp*. Укажите стрелкой, на какое место в стеке указывает *rbp* после классического пролога (какой адрес хранится в *rbp*).

Что изменилось по сравнению с соответствующими рисунками Л4.№4?

Рассчитайте смещения переменных *i16*, *i32*, *i64*, *f32*, *f64*, а также стековых параметров *scanf()* и *printf()* не только относительно *rsp* (положительные смещения), но и относительно *rbp* (отрицательные).

Реализуйте по разработанным изображениям стека программу; локальные переменные *i16*, *i32*, *i64*, *f32*, *f64* должны адресоваться через *rbp*.

Какие преимущества классический пролог имеет перед укороченным?



```
10 100 1000 2.3 5.6
i16=10 i32=100 i64=1000 f32=2.300000 f64=5.600000
```

Рис. 6: выполнение Ir4_5.S System V AMD64 ABI (Linux/macOS)

Листинг:

Файл Ir4_5.S:

```
1. # file: Ir4_5.S
2. # Программа с пятью локальными переменными, вводом scanf и выводом printf
3. # System V AMD64 ABI (Linux/macOS) , классический пролог (rbp фиксирован)
4.
5.     .section .data
6. format_in: .string "%hd %d %lld %f %lf" # Формат ввода: short, int, long long, float, double
7. format_out: .string "i16: %hd, i32: %d, i64: %lld, f32: %f, f64: %lf\n" # Формат вывода
8.
9.     .section .text
10.    .global main
11.    .extern scanf
12.    .extern printf
13.
14. main:
15.     # Пролог
16.     pushq %rbp                # Сохраняем базовый указатель
17.     movq %rsp, %rbp          # Устанавливаем новый базовый указатель
18.
19.     # Резервируем место на стеке для локальных переменных
20.     subq $32, %rsp           # Резервируем 32 байта (2 + 4 + 8 + 4 + 8 = 26 байт)
21.
22.     # Вызов scanf
23.     lea -2(%rbp), %rsi        # 2-й: адрес i16
24.     lea -6(%rbp), %rdx        # 3-й: адрес i32
25.     lea -14(%rbp), %rcx       # 4-й: адрес i64
26.     lea -18(%rbp), %r8        # 5-й: адрес f32
27.     lea -26(%rbp), %r9        # 6-й: адрес f64
28.
29.     lea format_in(%rip), %rdi # 1-й аргумент: форматная строка
30.     call scanf
31.
32.     # Вызов printf
33.     movq -2(%rbp), %rsi        # 2-й: i16 (знаковое расширение до 32 бит)
34.     movq -6(%rbp), %rdx        # 3-й: i32
35.     movq -14(%rbp), %rcx       # 4-й: i64
36.     movsd -18(%rbp), %xmm0     # 5-й: f32 (загружаем как float)
37.     cvtss2sd %xmm0, %xmm0     # Конвертируем float в double для printf
38.     movsd -26(%rbp), %xmm1    # 6-й: f64
39.
40.     movl $2, %eax             # Указываем количество XMM-аргументов (2)
41.
42.     lea format_out(%rip), %rdi # 1-й аргумент: форматная строка
43.     call printf
44.
45.     # Эпилог
```

```

46.    movq    %rbp, %rsp          # Восстанавливаем указатель стека
47.    popq    %rbp
48.
49.    movl     $0, %eax           # Возвращаем 0
50.    ret

```

Файл lr4_5.asm:

```

1.  ; lr4_5.asm
2.  ; Программа с пятью локальными переменными, вводом scanf и выводом printf
3.  ; Microsoft x64 calling convention (Windows), классический пролог (rbp фиксирован)
4.
5.  .data
6.  format_in db "%hd %d %lld %f %lf", 0
7.  format_out db "i16=%hd i32=%d i64=%lld f32=%f f64=%lf", 10, 0
8.
9.  .code
10. public main
11. extern scanf: proc
12. extern printf: proc
13.
14. main PROC
15.     push rbp                    ; Сохраняем базовый указатель
16.     mov rbp, rsp              ; Устанавливаем новый базовый указатель
17.     sub rsp, 32               ; Выделяем 32 байт на стеке
18.
19.     ; Передача параметров в scanf
20.     lea rcx, format_in        ; форматная строка - 1-й параметр в rcx
21.     lea rdx, [rbp - 2]        ; i16 - 2-й параметр в rdx
22.     lea r8, [rbp - 6]         ; i32 - 3-й параметр в r8
23.     lea r9, [rbp - 14]        ; i64 - 4-й параметр в r9
24.     lea xmm0, dword ptr [rbp - 18] ; f32 - 5-й параметр в xmm0
25.     lea xmm1, qword ptr [rbp - 26] ; f64 - 6-й параметр в xmm1
26.
27.     call scanf
28.
29.     ; Передача параметров в printf
30.     lea rcx, format_out        ; форматная строка printf в rcx
31.     movsx rdx, word ptr [rbp - 2] ; i16 -> rdx
32.     mov r8, dword ptr [rbp - 6] ; i32 -> r8
33.     mov r9, qword ptr [rbp - 14] ; i64 -> r9
34.     movss xmm0, dword ptr [rbp - 18] ; f32 -> xmm0
35.     movsd xmm1, qword ptr [rbp - 26] ; f64 -> xmm1
36.
37.     call printf
38.
39.     mov rsp, rbp              ; Восстанавливаем указатель стека
40.     pop rbp                  ; Восстанавливаем базовый указатель
41.     mov eax, 0               ; Возвращаем 0
42.     ret
43. main ENDP
44. end


```

Задание Л3.№6.

Бонус +2 балла для пар, обязательное для троек.

Опишите на C/C++ функцию с восемью параметрами типа *int/unsigned*, которая печатает свои параметры и возвращает результат, равный восьмому параметру.

Разработайте головную программу на ассемблере, вызывающую эту функцию.



```
Parameters: 1 2 3 4 5 6 7 8
Returned value: 1
```

Рис. 7: выполнение головной программы на ассемблере в связке с Си (Microsoft x64)

Листинг:

Файл task4_6.c:

```
1. #include <stdio.h>
2.
3. int func8 (int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8) {
4.     printf("Params: %d %d %d %d %d %d %d %d\n", a1, a2, a3, a4, a5, a6, a7, a8);
5.     return a8;
6. }
```

Файл lr4_6.S:

```
1. # lr4_6.S
2. # Головная программа, вызывающая func8
3. # System V AMD64 ABI, классический пролог (rbp)
4.
5.     .extern func8
6.     .extern printf
7.
8.     .section .data
9. format_res:
10.    .string "Returned value: %u\n"
11.
12.    .section .text
13.    .global main
14.
15. main:
16.    pushq    %rbp
17.    movq     %rsp, %rbp
18.
19.    # Передаем параметры func8 (8 целочисленных)
20.    # В System V первые 6 параметров – в rdi, rsi, rdx, rcx, r8, r9
21.    # Параметры 7 и 8 – на стеке (правый налево)
22.
23.    subq     $16, %rsp    # место для 7 и 8 параметров
24.
25.    movl     $1, %rdi     # a1
26.    movl     $2, %rsi     # a2
27.    movl     $3, %rdx     # a3
28.    movl     $4, %rcx     # a4
29.    movl     $5, %r8      # a5
30.    movl     $6, %r9      # a6
31.
32.    movl     $7, 8(%rsp)   # a7 (7-й параметр на стеке)
33.    movl     $8, 0(%rsp)   # a8 (8-й параметр на стеке)
34.
35.    call     func8
36.
37.    # Возвращаемое значение в rax (%rax)
38.
39.    movl     %rax, %rsi    # 2-й параметр printf (значение)
40.    leaq     format_res(%rip), %rdi    # 1-й параметр printf (формат)
41.    call     printf
42.
43.    movl     $0, %eax      # return 0
44.    popq     %rbp
45.
46.    ret
```

Файл Ir4_6.asm:

```
1. ; Ir4_6.asm
2. ; Головная программа, вызывающая func8
3. ; Microsoft x64 calling convention (Windows)
4.
5.     .data
6. format_res db "Returned value: %u", 10, 0
7.
8.     .code
9.     extern func8: proc
10.    extern printf: proc
11.    public main
12.
13. main PROC
14.     sub rsp, 72          ; теневое пространство + выравнивание + 4*8 данных стека
15.
16.     ; Первые 4 параметра в регистрах
17.     mov rcx, 1           ; a1
18.     mov rdx, 2           ; a2
19.     mov r8, 3            ; a3
20.     mov r9, 4            ; a4
21.
22.     ; Параметры 5-8 на стеке, слева направо
23.     mov dword ptr [rsp+32], 5 ; a5 (5-й параметр на стеке)
24.     mov dword ptr [rsp+40], 6 ; a6 (6-й параметр на стеке)
25.     mov dword ptr [rsp+48], 7 ; a7 (7-й параметр на стеке)
26.     mov dword ptr [rsp+56], 8 ; a8 (8-й параметр на стеке)
27.
28.     call func8
29.
30.     ; Возвращаемое значение в eax
31.
32.     mov rdx, rax          ; 2-й параметр printf
33.     lea rcx, format_res   ; 1-й параметр printf
34.     call printf
35.
36.     add rsp, 72
37.     mov eax, 0
38.     ret
39.
40. main ENDP
41. end
```