

Курс: Архитектура Компьютера и Операционные Системы

Лекция 2: Адресация, стек и вызов функций

Лектор: Evgeny Sokolov

Дата: 07.11.2025

Содержание

1 Адресация памяти в x86-64	2
1.1 Синтаксис Scale-Index-Base (SIB)	2
1.2 Указание размера операнда	2
2 Инструкция LEA (Load Effective Address)	3
2.1 LEA как оптимизация компилятора	3
3 Работа со стеком и локальными переменными	4
3.1 Проблема: Callee-clobbered регистры	4
3.2 Решение 1: Сохранение на стеке	4
3.3 Решение 2: Callee-saved регистры	5
4 Фреймовые указатели (Frame Pointers)	5
4.1 Структура стека с RBP	6
5 Секции данных в ассемблере	7
5.1 Директивы ассемблера для данных	7
6 Флаги процессора и условные переходы	8
7 Взаимодействие ассемблера и C/C++	9
7.1 Позиционно-независимый код (PIC) и RIP-адресация	9
7.2 Оптимизация хвостового вызова (TCO)	9
7.3 Вызов функций C (scanf / printf)	10
8 Синтаксисы ассемблера: Intel vs. AT&T	11

1 Адресация памяти в x86-64

Продолжаем изучение [низкоуровневый язык программирования, близкий к машинному коду \(Ассемблер\)](#). Ключевой темой является работа с памятью. В прошлый раз мы установили, что для обращения к памяти (разыменования) используется синтаксис с квадратными скобками.

1.1 Синтаксис Scale-Index-Base (SIB)

Общий синтаксис адресации памяти в 64-битном режиме (в Intel-синтаксис) выглядит следующим образом:

$$[rbase + rindex \times scale + displacement]$$

где:

- **rbase** — базовый регистр.
- **rindex** — регистр-индекс.
- **scale** — множитель (масштаб) для индекса. Допустимые значения: $scale \in \{1, 2, 4, 8\}$.
- **displacement** — константное смещение (сдвиг).

Этот синтаксис был разработан для удобной работы с массивами и структурами. Например, **rbase** может хранить адрес начала массива, **rindex** — индекс элемента, **scale** — размер одного элемента (e.g., 8 байт для `uint64_t`), а **displacement** — сдвиг до нужного поля внутри структуры.

```

1 ; * (uint64_t*)(rax + 8 * rdx) = rcx
2 ; (rax = base, rdx = index, 8 = scale)
3 mov [rax + rdx * 8], rcx
4
5 ; * (uint64_t*)(rbx + rbp + 32) = rax
6 ; (rbx = base, rbp = index, 1 = scale (default), 32 = displacement)
7 mov [rbx + rbp + 32], rax

```

Листинг 1 – Примеры SIB-адресации

1.2 Указание размера операнда

В листинг 1 ассемблер мог угадать размер операции (64 бита) по размеру регистра **rcx** или **rax**. Однако при работе с константами возникает неоднозначность.

```

1 mov [rax], 0 ; OSHIBKA: Neizvesten razmer: 1, 2, 4 ili 8 bayt?

```

Листинг 2 – Неоднозначность размера

Компилятор ассемблера не знает, какой размер данных вы намереваетесь записать. Для явного указания размера используются специальные директивы:

- **BYTE PTR** — 8 бит (1 байт).
- **WORD PTR** — 16 бит (2 байта).
- **DWORD PTR** — 32 бита (4 байта).
- **QWORD PTR** — 64 бита (8 байт).

```

1 ; * (uint32_t*)rax = 0
2 mov DWORD PTR [rax], 0

```

Листинг 3 – Явное указание размера (32 бита)

Итоги раздела

- Адресация SIB ($[base + index \times scale + disp]$) — основной механизм доступа к памяти.
- $scale$ ограничен значениями $\{1, 2, 4, 8\}$.
- При неоднозначности (например, при записи константы) размер операции нужно указывать явно (e.g., DWORD PTR).

2 Инструкция LEA (Load Effective Address)

Инструкция Load Effective Address, инструкция загрузки вычисленного адреса (LEA) — один из самых полезных и часто используемых инструментов в Ассемблер.

Определение: LEA (Load Effective Address)

Инструкция **lea** вычисляет адрес, используя синтаксис SIB, но **не разыменовывает** его. Вместо этого она записывает вычисленный адрес в регистр-приемник.

```

1 ; MOV: Prochitat' 8 bayt po adresu [rax] i polozhit' v rdx
2 ; rdx = * (uint64_t*)rax
3 mov rdx, [rax]
4
5 ; LEA: Vychislit' adres (v etom sluchae prosto rax) i polozhit' v rdx
6 ; rdx = rax
7 lea rdx, [rax]

```

Листинг 4 – Сравнение MOV и LEA

Основное применение LEA — это вычисление адресов, но благодаря своей способности выполнять сложение и умножение (на 1, 2, 4, 8), она стала мощным инструментом для арифметических вычислений.

```

1 ; rdx = rax + 4 * rbx + 16
2 lea rdx, [rax + rbx * 4 + 0x10]

```

Листинг 5 – LEA для вычисления адреса

2.1 LEA как оптимизация компилятора

Компиляторы часто используют LEA для выполнения простых арифметических операций, так как LEA часто выполняется быстрее, чем инструкции умножения (такие как imul). Например, для компиляции функции $a * 3$:

```

1 uint64_t Mul3(uint64_t a) {
2     return a * 3;
3 }

```

Листинг 6 – C++ код для умножения на 3

Компилятор (`g++ -O2`) сгенерирует следующий код (листинг 7), используя `LEA` вместо умножения. В Linux (System V AMD64 ABI) первый аргумент (`a`) передается в регистре `rdi`, а возвращаемое значение — в `rax`.

$$a \times 3 = a \times (1 + 2) = a + a \times 2$$

Этот паттерн идеально ложится в SIB-адресацию: $[rdi + rdi \times 2]$.

```

1 0000000000000000 <Mul3(unsigned long)>:
2   0: f3 0f 1e fa endbr64 ; Zashchitnaya instruktsiya
3   4: 48 8d 04 7f lea rax,[rdi+rdi*2] ; rax = rdi + rdi * 2
4   8: c3 ret

```

Листинг 7 – Результат компиляции Mul3 (objdump)

Итоги раздела

- `lea` вычисляет адрес, но не читает память.
- Это мощный инструмент для компактных арифметических вычислений, часто используемый компиляторами.

3 Работа со стеком и локальными переменными

У процессора ограниченное количество регистров. При вызове функции (инструкция `call`) возникает проблема: как сохранить значения локальных переменных, если вызываемая функция может перезаписать ("испортить") регистры?

3.1 Проблема: Callee-clobbered регистры

Согласно соглашениям о вызовах (Calling Conventions), большинство регистров (как `rax`, `rdx`, `rdi` и т.д.) являются *callee-clobbered* — вызываемая функция (*callee*) имеет право изменять их без восстановления.

Рассмотрим код, где мы храним `a` и `b` в регистрах:

```

1 mov rax, 1 ; a = 1
2 mov rdx, 2 ; b = 2
3 call f ; f()
4 add rax, rdx ; ??? (rdx mozhet byt' isporchen funktsiey f)
5 ret

```

Листинг 8 – Проблема сохранения локальных переменных

После возврата из `f`, мы не можем полагаться на то, что в `rdx` все еще лежит 2.

3.2 Решение 1: Сохранение на стеке

Основной механизм для сохранения локальных переменных — это *стековый фрейм*. Мы можем "зарезервировать" место на стеке, сдвинув указатель стека *Stack Pointer*, регистр-указатель на вершину стека (*RSP*), и сохранить туда наши значения.

```

1 mov rax, 1 ; a = 1
2 mov rdx, 2 ; b = 2
3
4 sub rsp, 16 ; Rezerviruem 16 bayt na steke
5 mov [rsp + 8], rdx ; Sokhranyaem b (po smeshcheniyu 8)
6 mov [rsp], rax ; Sokhranyaem a (na vershinu steka)

```

```

7
8 call f ; f()
9
10 ; VOSSTANOVLENIE
11 mov rax, [rsp] ; Vosstanavlivаем а
12 mov rdx, [rsp + 8] ; Vosstanavlivаем б
13 add rsp, 16 ; Освобождаем место на стеке
14
15 add rax, rdx ; Теперь безопасно
16 ret

```

Листинг 9 – Использование стека для локальных переменных

3.3 Решение 2: Callee-saved регистры

Некоторые регистры, напротив, являются *callee-saved* (например, `rbx`, `rbp`). Это означает, что если вызываемая функция хочет их использовать, она *обязана* сохранить их значение (обычно на стеке) и восстановить перед выходом (`ret`). Компиляторы используют это для оптимизации.

Рассмотрим C++ код:

```

1 uint64_t f();
2 uint64_t g();
3 uint64_t Sum() {
4     return f() + g();
5 }

```

Листинг 10 – C++ код Sum()

Чтобы вычислить `g()`, нужно сначала вызвать `f()`, но результат `f()` (который вернется в `rax`) будет перезаписан результатом `g()`. Компилятор (листинг 11) решает эту проблему, сохраняя результат `f()` в callee-saved регистре `rbx`.

```

1 <Sum():>
2     push rbx ; 1. Сохраняет старое значение rbx
3     call <f()> ; 2. Вызывает f(). Результат в rax
4     mov rbx, rax ; 3. Спрятывает результат f() в rbx
5     call <g()> ; 4. Вызывает g(). Результат в rax
6     add rax, rbx ; 5. rax = rax + rbx (результат g() + результат f())
7     pop rbx ; 6. Восстанавливает старое значение rbx
8     ret

```

Листинг 11 – Дизассемблированный код Sum() (g++ -O2)

Примечание

В листинг 11 мы видим `call` на адрес вроде `<Sum()>+0xa` (в реальном `objdump` это часто `call 0`). Это *релокация*. На этапе компиляции адрес функции `f` еще неизвестен. Компилятор оставляет "дырку" (часто 0), а компоновщик (linker) на финальном этапе сборки подставляет в это место реальный адрес функции.

4 Фреймовые указатели (Frame Pointers)

В листинг 9 мы вручную двигали `RSP` (`sub rsp, 16`) и обращались к переменным относительно `RSP` (`[rsp + 8]`). Это работает, но усложняет отладку и трассировку стека.

Определение: Фреймовый указатель (RBP)

Base Pointer, регистр-указатель на базу стекового фрейма (RBP) (Base Pointer) — это регистр, который по соглашению используется для хранения адреса *начала* текущего **стековый фрейм**. Это обеспечивает "стабильный" якорь для доступа к локальным переменным, даже если RSP постоянно движется (например, при push/pop).

Для использования RBP применяется стандартный **пролог** (в начале функции) и **эпилог** (в конце).

```

1 f:
2 ; --- PROLOG ---
3 push rbp ; 1. Sokhranit' RBP predydushchey funktsii na stek
4 mov rbp, rsp ; 2. Zapomnit' tekushchuyu vershinu steka kak bazu (nachalo)
5 ; nashego freyma. Teper' RBP stabilen.
6
7 ; --- Telo funktsii ---
8 ; Mesto dlya lokal'nykh peremennых vydelyaetsya zdes'
9 ; sub rsp, 32 ; (vydelit' 32 bayta)
10 ; Dostup k peremennym idet otnositel'no RBP:
11 ; mov [rbp - 8], rax
12
13 ; --- EPILOG ---
14 mov rsp, rbp ; 1. Osvoobodit' use lokal'nye peremennye, vernuu rsp k baze
15 pop rbp ; 2. Vosstanovit' RBP predydushchey funktsii
16 ret

```

Листинг 12 – Стандартный пролог и эпилог функции

4.1 Структура стека с RBP

Когда каждая функция использует этот пролог, значения RBP на стеке образуют **односвязный список**. Каждое сохраненное значение RBP указывает на RBP предыдущей (вызвавшей) функции.

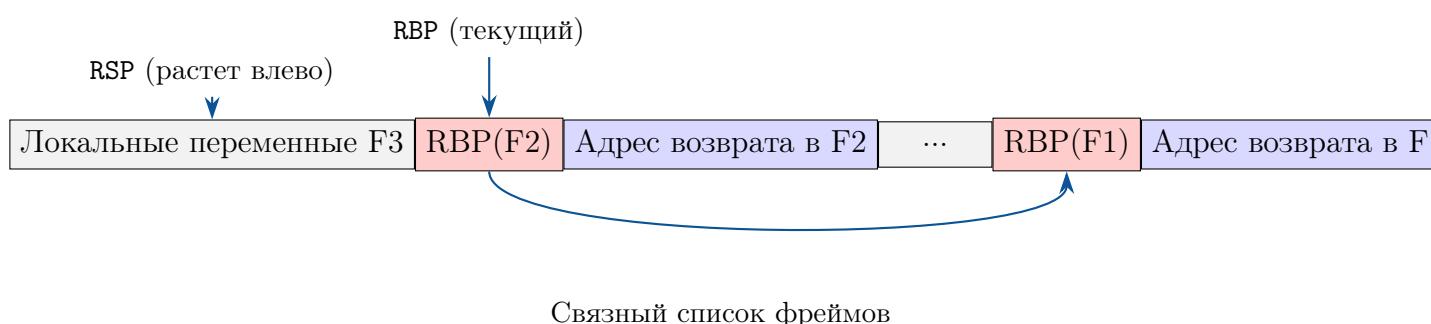


Рис. 1 – Структура стека при использовании фреймовых указателей (RBP)

Это позволяет отладчикам и другим инструментам легко "разматывать" стек (stack unwinding) и строить трассировку вызовов (stack trace).

Примечание

Использование RBP как фреймового указателя — это *соглашение*. Оно требует одного лишнего регистра и нескольких инструкций в прологе/эпилоге. Современ-

ные компиляторы (`g++ -O2`) по умолчанию часто отключают фреймовые указатели (`-fomit-frame-pointer`) для оптимизации. Вместо этого они генерируют специальную отладочную информацию (DWARF), которая позволяет разматывать стек, зная только **Instruction Pointer, регистр-указатель на следующую инструкцию (RIP)**.

5 Секции данных в ассемблере

Ассемблерный код и данные не хранятся в перемешку. Они организованы в секции, которые сообщают операционной системе, как их следует загружать в память.

- **.text** — Код (инструкции). [cite: 216] Загружается с правами **Read-Only** и **Execute (RX)**. [cite: 499]
- **.rodata** — Данные только для чтения. [cite: 217] (e.g., строковые литералы, константы). Загружаются с правами **Read-Only (R)**. [cite: 501]
- **.data** — Инициализированные данные. [cite: 217] (e.g., глобальные переменные с начальным значением). Загружаются с правами **Read-Write (RW)**. [cite: 505]
- **.bss** — Неинициализированные данные. [cite: 218] (e.g., `int x;`). Эти данные *не хранятся* в исполняемом файле, файл хранит только их размер. При загрузке ОС выделяет память и *обнуляет* ее. [cite: 507]

5.1 Директивы ассемблера для данных

Мы можем явно указать, в какую секцию помещать байты, с помощью директив.

```

1 ; Db"yavlyayem sektsiyu .rodata
2 .section .rodata
3
4 .global s1 ; Delaem metku s1 vidimoy dlya linkera
5 s1:
6     .byte 0x48, 0x65 ; 'H', 'e'
7     .ascii "ll" ; 'l', 'l'
8     .asciz "o!" ; 'o', '!', i nulevoy bayt (terminator)

```

Листинг 13 – Пример секции `.rodata` (строка "Hello!")

В листинг 13 метка `s1` указывает на строку "Hello!\0".

Другие директивы для инициализации данных в секциях `.data` или `.rodata`:

```

1 .data
2     val1: .byte 0x10 ; 1 bayt
3     val2: .short 0x1234 ; 2 bayta
4     val3: .long 0x12345678 ; 4 bayta
5     val4: .quad 0x1122334455667788 ; 8 bayt
6
7     ; Zapolnit' 32 bayta znacheniem 0xFF
8     buffer: .space 32, 0xFF
9
10 .bss
11     ; Zarezervirovat' 1024 bayta (budut obnuleny)
12     big_buffer: .skip 1024

```

Листинг 14 – Директивы `.data` и `.bss`

Примечание

Все есть байты. В конечном счете, ассемблер просто транслирует мнемоники инструкций в байты в секции `.text`. Можно написать функцию, используя только директиву `.byte`, если знать машинные коды.

6 Флаги процессора и условные переходы

Большинство арифметических и логических инструкций (ALU) изменяют специальный регистр флагов (EFLAGS/RFLAGS). Условные переходы (`jcc`) анализируют эти флаги.

Основные флаги, интересующие нас:

- **Carry Flag, флаг переноса (беззнаковое переполнение) (CF)** (Carry Flag) — Установлен, если произошел перенос/заём из старшего бита (индикатор **беззнакового** переполнения). [cite: 313, 545]
- **Zero Flag, флаг нуля (результат равен нулю) (ZF)** (Zero Flag) — Установлен, если результат операции равен нулю. [cite: 314, 546]
- **Sign Flag, флаг знака (установлен старший бит результата) (SF)** (Sign Flag) — Установлен, если старший бит результата равен 1 (индикатор **отрицательного** числа в знаковой интерпретации). [cite: 315, 546]
- **Overflow Flag, флаг переполнения (знаковое переполнение) (OF)** (Overflow Flag) — Установлен, если произошло **знаковое** переполнение (e.g., 100+100 дало отрицательный результат в 8-битном знаковом представлении). [cite: 316, 547]

```

1 # 0b0001 + 0b0111 = 0b1000 (1 + 7 = 8)
2 # Rezul'tat (8) imet bit znaka (SF=1).
3 # Proizoshlo znakovoe perepolnenie (1+7 != -8) (OF=1).
4 # Flagi: SF, OF

5
6 # 0b1001 + 0b0111 = 0b0000 (s perenosom) (9 + 7 = 16)
7 # Rezul'tat 0 (ZF=1).
8 # Proizoshlo bezznakovoe perepolnenie (CF=1).
9 # Flagi: ZF, CF

10
11 # 0b0010 - 0b0011 = 0b1111 (s zaemom) (2 - 3 = -1)
12 # Rezul'tat -1 (SF=1).
13 # Proizoshel bezznakovy zaem (CF=1).
14 # Flagi: CF, SF

```

Листинг 15 – Примеры установки флагов (4-битная арифметика)

Инструкция `cmp` (compare) — это, по сути, `sub`, которая не сохраняет результат, а только устанавливает флаги.

После `cmp` (или `add`, `sub`, `and...`) используются инструкции условного перехода `jcc`:

- `je / jz` — Jump if Equal / Jump if Zero (проверяет `ZF=1`).
- `jne / jnz` — Jump if Not Equal / Jump if Not Zero (`ZF=0`).
- `js` — Jump if Sign (`SF=1`).
- `ja` — Jump if Above (беззнаковое "больше") (проверяет `CF=0` и `ZF=0`).
- `jg` — Jump if Greater (знаковое "больше") (проверяет `SF=OF` и `ZF=0`).

7 Взаимодействие ассемблера и C/C++

Можно смешивать код на C/C++ и Ассемблер в одной программе, если соблюдать соглашения.

7.1 Позиционно-независимый код (PIC) и RIP-адресация

При попытке получить доступ к глобальной переменной (e.g., из C++ или секции `.data`) возникает проблема:

```

1 .data
2 my_var: .quad 123
3
4 .text
5 ; OSHIBKA: Ne budet rabotat' v souremennykh OS
6 mov rax, [my_var]

```

Листинг 16 – Наивный доступ к глобальной переменной

Проблема в том, что в современных ОС из соображений безопасности (ASLR – Address Space Layout Randomization) программа загружается в память по *случайному* адресу. [cite: 587-589] Мы не знаем абсолютный адрес `my_var` на этапе компиляции. [cite: 583]

Решение — **Position-Independent Code**, позиционно-независимый код (PIC) (PIC). Код не должен полагаться на абсолютные адреса, а только на *относительные*.

Определение: RIP-относительная адресация

В 64-битном режиме можно адресовать данные *относительно указателя инструкции* (RIP). Так как RIP всегда указывает на следующую исполняемую инструкцию, а `my_var` находится на *неизменном* расстоянии от этой инструкции (весь код и данные сдвигаются вместе), этот сдвиг остается константой. [cite: 593-594]

```

1 extern c ; Ob'yavlyuem metku 'c' vneshney (opredelena v C++)
2
3 .text
4 GetC:
5     ; Korrektno: zagruzit' znachenie po adresu [rip + smeshchenie do 'c']
6     mov rax, [c+rip]
7     ret

```

Листинг 17 – Корректный доступ к глобальной переменной (PIC)

7.2 Оптимизация хвостового вызова (TCO)

Рассмотрим функцию-обертку, которая просто вызывает другую функцию и немедленно возвращает ее результат.

```

1 MyFuncWrapper:
2     ; ... podgotovka argumentov ...
3     call OtherFunc ; 1. Zapisat' adres vozvrata (A) na stek
4     ret ; 2. Snyat' adres (A) so steka i pereyti na nego

```

Листинг 18 – Неоптимальный хвостовой вызов

Здесь `call` кладет на стек адрес возврата (в `MyFuncWrapper`), а `ret` немедленно его снимает. Это лишняя работа.

Tail Call Optimization, оптимизация хвостового вызова (TCO) (TCO) — это замена `call + ret` на один `jmp`.

```

1 MyFuncWrapper:
2     ; ... подготавка аргументов ...
3     jmp OtherFunc ; Передача управления OtherFunc

```

Листинг 19 – Оптимизированный хвостовой вызов (TCO)

Когда `OtherFunc` выполнит `ret`, она вернет управление не в `MyFuncWrapper`, а тому, кто вызвал `MyFuncWrapper` (т.к. его адрес возврата все еще лежит на вершине стека). [cite: 608] Компиляторы (-O1 и выше) активно применяют эту оптимизацию.

7.3 Вызов функций C (`scanf` / `printf`)

Пользоваться вводом-выводом C++ (`iostream`) из [Ассемблер](#) почти невозможно из-за name mangling (искажения имен). [cite: 638-640] Гораздо проще использовать функции из C `<stdio.h>`, такие как `scanf` и `printf`. [cite: 643]

При этом нужно строго соблюдать два правила соглашения о вызовах (ABI): **1. Аргументы:** Первые 6 целочисленных аргументов/указателей передаются через регистры (именно в таком порядке): `RDI, RSI, RDX, RCX, R8, R9`.

2. Выравнивание стека: Перед инструкцией `call RSP` (указатель стека) **должен быть выровнен по 16-байтной границе**. [cite: 602]

Примечание

Ловушка выравнивания: Когда нашу функцию `main` вызывают, `RSP` уже выровнен по 16-байтной границе. Но инструкция `call` (которая вызвала `main`) помещает на стек 8-байтный адрес возврата. [cite: 603] Это означает, что *внутри* нашей функции `main RSP не выровнен` (он равен $16N + 8$). Перед тем, как мы сами сделаем `call` (например, `call scanf`), мы должны "скомпенсировать" эти 8 байт, например, `sub rsp, 8`. [cite: 604, 733]

```

1 .intel_syntax noprefix
2
3 .section .rodata
4 ; Formatnaya stroka dlya chteniya ("%lld")
5 read_fmt: .asciz "%lld"
6 ; Formatnaya stroka dlya zapisi ("%lld\n")
7 write_fmt: .asciz "%lld\n"
8
9 .text
10 .global main
11 main:
12     ; --- PROLOG ---
13     ; Vydelyaem 8 bayt dlya peremennoy 'n'
14     ; I zaocno VYRAVNIVAE stek (rsp byl 16N+8, stal 16N)
15     sub rsp, 8
16
17     ; --- Vyzov scanf ---
18     ; scanf("%lld", &n);
19     ; &n teper' = adres [rsp]
20
21     ; Arg 1 (RDI): Adres formatnoy stroki

```

```

22    lea rdi, [read_fmt+rip]
23    ; Arg 2 (RSI): Adres, kuda pisat' rezul'tat (vershina steka)
24    mov rsi, rsp
25
26    ; Dlya variadic funktsiy (kak scanf) nuzhno obnulit' rax
27    xor rax, rax
28    call scanf
29
30    ; --- Vyzov printf ---
31    ; printf("%lld\n", n + 1);
32    ; Zagruzhaem 'n' so steka
33    mov rsi, [rsp]
34    ; Uvelichivaem
35    add rsi, 1
36
37    ; Arg 1 (RDI): Adres formatnoy stroki
38    lea rdi, [write_fmt+rip]
39    ; Arg 2 (RSI): Znachenie (n + 1)
40    ; (uzhe v rsi)
41
42    xor rax, rax
43    call printf
44
45    ; --- EPILOG ---
46    ; "return 0;" 
47    ; Po ABI, my vozvrashchaem znachenie iz main cherez RAX
48    xor rax, rax
49
50    ; Osvobozhdaem mesto na stekte
51    add rsp, 8
52    ret

```

Листинг 20 – Пример: чтение числа (n) и вывод (n+1) на Assembler

Итоги раздела

- Для доступа к глобальным данным используйте **RIP-относительную адресацию** ([my_var+rip]).
- `call func + ret` можно заменить на `jmp func` (TCO).
- При вызове функций С (e.g., `printf`) стек **должен быть выровнен по 16 байт до** инструкции `call`.
- Аргументы передаются через `RDI`, `RSI`, `RDX`, `RCX`...
- `scanf` ожидает *указатель* (адрес) в `RSI`, `printf` — *значение*.
- Возвращаемое значение из `main` — это то, что лежит в `RAX` в момент `ret`.

8 Синтаксисы ассемблера: Intel vs. AT&T

Существует два доминирующих синтаксиса x86 Ассемблер.

- Intel-синтаксис:** (Используется в этой лекции, в документации Intel, Microsoft).

- **AT&T-синтаксис (GNU):** (Используется по умолчанию в objdump и gcc). [cite: 688]

Ключевые отличия:

Таблица 1 – Сравнение синтаксисов Intel и AT&T (GNU)

Аспект	Intel (мы)	AT&T (GNU)
Порядок operandов	<code>mov rax, rbx</code> (Приемник, Источник)	<code>mov %rbx, %rax</code> (Источник, Приемник)
Регистры	<code>rax, rbx</code>	<code>%rax, %rbx</code> (с префиксом %)
Константы	<code>16, 0x10</code>	<code>\$16, \$0x10</code> (с префиксом \$)
Адресация	<code>[rax + rbx * 4 + 32]</code>	<code>32(%rax, %rbx, 4)</code>
Размер	<code>DWORD PTR [rax]</code>	<code>movl \$0, (%rax)</code> (суффикс l/q/w/b)

Примечание

Полезно уметь читать оба синтаксиса. В objdump можно включить [Intel-синтаксис](#) с помощью флага -M intel.

