Курс: *Архитектура компьютера и ОС*

Лекция 7: Представление данных, сборка и основы ассемблера

Лектор: Имя Фамилия

Дата: 07.11.2025

Содержание

1 Представление целых чисел

В прошлых лекциях мы обсуждали представление текстовых данных. Теперь рассмотрим, как в памяти кодируются целые числа.

1.1 Беззнаковые числа

С беззнаковыми (unsigned) числами всё просто. Они представляются напрямую своим двоичным эквивалентом. Если у нас есть N бит, мы можем представить числа от 0 до $2^N - 1$. Например, для 3-битного числа:

- `000` $\rightarrow 0$
- `001` $\rightarrow 1$
- `010` $\rightarrow 2$
- `111` $\rightarrow 7$

1.2 Знаковые числа: Прямой код (Sign-Magnitude)

Первая и самая прямолинейная идея для представления знаковых чисел — использовать один бит (обычно старший) для кодирования знака, а остальные биты — для кодирования абсолютного значения (величины).

Например, для 3-битного числа (1 бит на знак, 2 на значение):

- `001` $\rightarrow +1$
- `010` $\rightarrow +2$
- `101` $\rightarrow$ -1
- `110` $\rightarrow$ -2

У этого подхода есть два существенных недостатка:

1. **Проблема двух нулей:** Существует два представления для нуля: `000` ($+0$) и `100` (-0). Это избыточно и усложняет проверки.
2. **Сложная арифметика:** Обычный двоичный сумматор "ломается". Сложение $+1$ (`001`) и -1 (`101`) в лоб даст `110`, что равно -2, а не 0. Для выполнения арифметических операций требуются сложные проверки знаков.

1.3 Знаковые числа: Дополняющий код (Two's Complement)

Современные компьютеры решают эти проблемы, используя Дополняющий код.

Определение: Дополняющий код

Дополняющий код — это способ представления знаковых чисел, основанный на арифметике по модулю 2^N, где N — количество бит.

- Положительные числа (и 0) представляются так же, как и беззнаковые (в диапазоне от 0 до $2^{N-1} - 1$).

- Отрицательные числа x (в диапазоне от -2^{N-1} до -1) представляются как беззнаковое число $2^N + x$.

Рассмотрим 3-битные числа (модуль $2^3 = 8$):

- `000` $\to 0$
- `001` $\to 1$
- `010` $\to 2$
- `011` $\to 3$
- `100` $\to 4$ (или $4 - 8 = -4$)
- `101` $\to 5$ (или $5 - 8 = -3$)
- `110` $\to 6$ (или $6 - 8 = -2$)
- `111` $\to 7$ (или $7 - 8 = -1$)

Преимущества дополняющего кода:

- **Один ноль:** Значение `000` уникально.
- **Простая арифметика:** Обычный двоичный сумматор корректно работает как для знаковых, так и для беззнаковых чисел.

Примечание

Пример арифметики: Сложим 1 (`001`) и -2 (`110`) как знаковые.

$$\texttt{001} + \texttt{110} = \texttt{111}$$

Результат `111` в дополняющем коде — это -1. Сложение работает.
Теперь сложим 1 (`001`) и 6 (`110`) как беззнаковые.

$$\texttt{001} + \texttt{110} = \texttt{111}$$

Результат `111` в беззнаковом коде — это 7. Сложение также работает.

1.3.1 Получение отрицательного числа

Практическое правило для получения представления числа $-x$ из x в дополняющем коде:

1. Инвертировать все биты x (операция ~x, побитовое НЕ).
2. Прибавить к результату 1.

Формула: $-x = \sim x + 1$.

Пример: Найти представление -3 (для 3-битного числа).

1. Берём 3: `011`
2. Инвертируем (~): `100`
3. Прибавляем 1: `100` + 1 = `101`

Результат `101` — это -3, что совпадает с нашей таблицей.

2 Выравнивание данных в памяти

Определение: Выравнивание данных (Data Alignment)

Выравнивание данных — это ограничение, согласно которому данные определённого типа и размера должны размещаться в памяти по адресам, кратным некоторой степени двойки.
Например, 8-байтный `int64_t` должен иметь адрес, который делится на 8 (т.е. `address % 8 == 0`).

2.1 Зачем нужно выравнивание?

Выравнивание — это не просто прихоть компилятора, а требование, диктуемое аппаратным обеспечением (процессором).

- **Эффективность:** Процессоры читают данные из памяти не по одному байту, а "блоками"(например, по 4, 8 или 16 байт). Если 8-байтовое число "пересекает"границу такого блока (например, начинается с адреса 4 и заканчивается на 11), процессору придётся выполнить два чтения из памяти вместо одного.
- **Корректность:** На некоторых архитектурах (не x86) обращение по невыровненному адресу может привести к немедленному падению программы (аппаратному прерыванию). На x86 это "всего лишь"приводит к сильному замедлению.
- **Атомарность:** Операции чтения/записи по выровненным адресам, как правило, атомарны (неделимы). Невыровненная запись (например, 8 байт) может быть выполнена процессором как две отдельные записи по 4 байта.

Примечание

Проблема неатомарности особенно важна при работе с разделяемой памятью (shared memory). Представим, что два процесса (например, полученные через `fork()` с памятью `mmap(MAP_SHARED)`) работают с одним 8-байтным числом по невыровненному адресу.
Процесс А пишет новое значение. Он может успеть записать первые 4 байта, но не вторые. В этот момент Процесс Б читает это число и видит "мусор"— половину старого значения и половину нового.

Из-за этих требований компиляторы (C, C++, Rust, Go) автоматически вставляют "пропуски"(padding) в структуры, а стандартные аллокаторы (`malloc`, `operator new`) возвращают память, выровненную по максимальному требованию для стандартных типов (например, 16 байт на x86-64).

3 Процесс сборки программы

Рассмотрим, почему в C/C++ принято разделять код на заголовочные файлы (`.h`) и файлы реализации (`.cpp`).

3.1 Препроцессор и `#include`

Первый этап сборки — Препроцессинг. Директивы, начинающиеся с `#`, обрабатываются на этом этапе.

Директива `#include "header.h"` — это простая текстовая операция. Она заменяет эту строку содержимым файла `header.h`. Это можно проверить, запустив компилятор с флагом `-E`:

```
# gcc -E main.c
```

Листинг 1 – Запуск только препроцессора

3.1.1 Проблема многократного включения

Если один `.h` файл включается несколько раз (например, `a.h` и `b.h` оба включают `common.h`, а `main.cpp` включает `a.h` и `b.h`), мы получим дублирование кода и ошибки компиляции.

Для решения этой проблемы используются Страж включения:

- **Классический способ (Стражи):**

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

// ... soderzhimoe zagolovka ...

#endif // MY_HEADER_H
```

Листинг 2 – Использование ifndef/define

- **Современный способ:**

```
#pragma once

// ... soderzhimoe zagolovka ...
```

Листинг 3 – Использование pragma once

Оба способа гарантируют, что препроцессор включит тело файла только один раз.

3.2 Единицы трансляции и ускорение сборки

Основная причина разделения кода на `.h` и `.cpp` — это **ускорение сборки** больших проектов за счёт параллелизма.

Определение: Единица трансляции (Translation Unit)

Единица трансляции — это один `.c` или `.cpp` файл после того, как препроцессор "вклеил"в него содержимое всех `#include`.

Процесс сборки можно разбить на два этапа:

1. **Компиляция (Compilation):** Компилятор (например, `gcc -c`) *независимо и параллельно* обрабатывает каждую единицу трансляции, превращая её в Объектный файл (<code>.o</code>). Этот этап включает синтаксический анализ, оптимизации (<code>-O2</code>) и генерацию машинного кода.
2. **Линковка (Linking):** Линковщик (компоновщик) берёт все <code>.o</code> файлы и "сшивает"их в один исполняемый файл. Этот этап, как правило, последовательный, но он выполняется быстрее, чем полная перекомпиляция всего проекта.

Если мы меняем один `.cpp` файл, нам нужно перекомпилировать только его, а затем быстро перелинковать проект. Если бы весь код был в одном файле, любое изменение требовало бы полной перекомпиляции.

3.3 Объектные файлы и символы

Объектный файл (<code>.o</code>) — это "полуфабрикат". Он содержит машинный код, но в нём ещё нет информации о том, где находятся функции и переменные из *других* <code>.o</code> файлов.

Связь между файлами осуществляется через **символы**. С помощью утилиты `nm` можно посмотреть таблицу символов объектного файла.

```
# nm main.o
0000000000000000 T main
                 U isEven
                 U printf
                 U scanf
```

Листинг 4 – Анализ символов с помощью nm

- **T (Text):** Символ *определён* (defined) в этом файле. Здесь определён `main`.
- **U (Undefined):** Символ *используется*, но не определён. Линковщик должен будет найти его в другом <code>.o</code> файле или библиотеке.

3.4 Линковка и релокации

Когда компилятор генерирует `main.o` и видит вызов `isEven()`, он не знает адреса этой функции. Вместо адреса он оставляет "дырку"— специальную запись, называемую Релокация.

Задача линковщика:

1. Найти `main.o`, у которого `isEven` помечен как `U`.
2. Найти другой <code>.o</code> файл (например, `even.o`), у которого `isEven` помечен как `T`.
3. "Заполнить дырку"(выполнить релокацию) в `main.o`, подставив реальный адрес `isEven` из `even.o`.

Если линкощик не может найти определение для `U`-символа (или находит *несколько* определений), он выдаёт ошибку ("Undefined reference"или "Multiple definition").

Примечание

Поскольку <code>.o</code> файлы содержат только машинный код и таблицу символов (а не C++ или C код), они языково-независимы. Это позволяет компоновать программу из частей, написанных на разных языках (например, скомпилировать функцию на Rust, а вызвать её из C).

3.5 Формат ELF

В Linux исполняемые файлы и объектные файлы хранятся в формате ELF (Executable and Linkable Format). Он состоит из **секций**, которые сообщают загрузчику ОС, как создать образ процесса в памяти.

Основные секции:

- `.text`: Исполняемый код (инструкции центральный процессор (CPU)). Загружается с правами "чтение + исполнение".

- `.rodata`: (Read-Only Data) Константные данные, например, строковые литералы (<code>"Hello«/code>). Загружается с правами "только чтение".
- `.data`: Инициализированные глобальные и статические переменные (<code>int x = 10;</code>). Загружается с правами "чтение + запись".
- `.bss`: Неинициализированные глобальные и статические переменные (<code>int y;</code>). Эта секция *не занимает места в файле*, она просто говорит загрузчику: "выдели X байт памяти и заполни их нулями".

4 Особенности C++: Имена и переменные

4.1 Искажение имён (Name Mangling)

В C++ можно объявлять функции с одинаковыми именами, но разными аргументами (перегрузка) или в разных пространствах имён:

```
void f();
void f(int);
namespace A { void f(); }
```

Линковщик C не справился бы с этим, так как он видит только один символ `f`. Компилятор C++ решает эту проблему, кодируя полную сигнатуру функции в имя символа. Этот процесс называется Искажение имён (Name Mangling).

Например, `A::f()` может превратиться в `_Z1A1fv`.

4.2 Extern C

Чтобы C++ мог вызывать функции из C (или из ассемблера, который следует C-соглашениям) или наоборот, нужно отключить Искажение имён (Name Mangling). Для этого используется extern "C":

```
// Ob'yavlyaem, chto eta funktsiya ispol'zuet C ABI
// (bez iskazheniya imen)
extern "C" void my_c_function(int x);
```

4.3 Глобальные переменные: `extern` против `static`

Как и в случае с функциями, глобальные переменные нужно *объявлять* (в `.h`) и *определять* (в `.cpp`).

- **Правильный способ (Общая переменная):**

```
// Govorim kompilyatoru, chto peremennaya *gde-to* sushchestvuet
extern int shared_value;
```

```
// Vydelyaem pamyat' i zadaem znachenie
int shared_value = 123;
```

Все `.cpp` файлы, включившие `def.h`, будут ссылаться на *одну и ту же* копию `shared_value`.

- **Неправильный способ (Локальные копии)**

```
// 'static' v global'noy oblasti vidimosti
// delaet peremennuyu lokal'noy dlya edinitsy translyatsii
static int value = 0;
```

Если `main.cpp` и `other.cpp` включат `static.h`, *каждый* из них получит свою *собственную, независимую* копию `value`. Линкер не выдаст ошибки, но программа будет работать некорректно.

5 Основы ассемблера и архитектуры

5.1 Архитектура фон Неймана

Современные процессоры в основном следуют Архитектура фон Неймана .

Ключевая особенность — единый блок памяти, в котором хранятся и данные, и инструкции (код) программы. Процессор выполняет инструкции последовательно, используя RIP (Instruction Pointer) (Instruction Pointer) для отслеживания адреса текущей инструкции.

5.2 Регистры и память

Доступ к оперативной памяти (RAM) — медленная операция (порядка 100 ns). Чтобы процессор не простаивал, он содержит Регистр — сверхбыстрые ячейки памяти.

В архитектуре x86-64 (которую мы используем) есть 16 64-битных регистров общего назначения (<code>RAX</code>, <code>RCX</code>, <code>RDX</code>, <code>RSI</code>, <code>RDI</code>, <code>R8</code>...<code>R15</code> и т.д.).

Два регистра имеют особо важное значение:

- **RIP (Instruction Pointer):** Указатель на инструкцию.
- **RSP (Stack Pointer):** Указатель на вершину стека.

5.3 Стек и вызовы функций

Для реализации вызовов функций используется стек.

1. `call f` (Вызов функции):
   - Процессор помещает адрес *следующей* за `call` инструкции (адрес возврата) на вершину стека (<code>push return_addr</code>).
   - Процессор совершает безусловный переход на адрес функции `f` (<code>jmp f</code>).
2. `ret` (Возврат из функции):
   - Процессор снимает адрес возврата с вершины стека (<code>pop return_addr</code>).
   - Процессор совершает безусловный переход на этот адрес (<code>jmp return_addr</code>).

5.4 Соглашение о вызовах (ABI)

Как функции передают аргументы и возвращают значения? Процессор об этом "не знает". Это определяется программным соглашением — ABI (Application Binary Interface).

Для Linux x86-64 (System V ABI) действуют следующие правила:

Определение: Соглашение о вызовах (x86-64 System V ABI)

- **Передача аргументов (целочисленных):**
  - 1-й аргумент: RDI
  - 2-й аргумент: RSI

- 3-й аргумент: <code>RDX</code>
- 4-й аргумент: <code>RCX</code>
- 5-й аргумент: <code>R8</code>
- 6-й аргумент: <code>R9</code>

- **Передача аргументов (7-й и далее):**
  - Передаются через стек. Вызывающая сторона (caller) кладёт их на стек в обратном порядке *до* выполнения инструкции `call`.
- **Возвращаемое значение:**
  - RAX

Примечание

Аргументы на стеке. Поскольку `call` кладёт на стек адрес возврата, внутри вызываемой функции (callee) аргументы, переданные через стек, оказываются смещены:

- `[rsp]` — Адрес возврата (положен инструкцией `call`)
- `[rsp+8]` — 7-й аргумент
- `[rsp+16]` — 8-й аргумент
- и т.д.

(Здесь `[addr]` означает "прочитать 8 байт из памяти по адресу `addr`").

6 Практика: написание функций на ассемблере

Мы будем использовать синтаксис Intel. Файл `.S` должен начинаться с директив:

```
.intel_syntax noprefix # Ustanavlivaem sintaksis
.text # Nachalo sektsii koda
```

Чтобы сделать функцию `my_func` видимой для линковщика (C/C++), её нужно объявить глобальной:

```
.global my_func
my_func:
    # ... instruktsii ...
    ret
```

6.1 Пример 1: Возврат константы

```
// C++: extern "C" long constant();
```

```
.global constant
constant:
    mov rax, 42 # Vozvrashchaemoe znachenie - v RAX
    ret
```

6.2 Пример 2: Identity (аргумент -> возврат)

```
// C++: extern "C" long identity(long x);
```

```
.global identity
identity:
    # 1-y argument 'x' prihodit v RDI
    mov rax, rdi # Peremeshchaem RDI v RAX
    ret
```

6.3 Пример 3: Сложение (два аргумента)

```
// C++: extern "C" long add(long x, long y);
```

```
.global add
add:
    # x v RDI, y v RSI
    add rdi, rsi # Skladyvaem: RDI = RDI + RSI
    mov rax, rdi # Peremeshchaem rezul'tat (v RDI) v RAX
    ret
```

6.4 Пример 4: Условный переход (If/Else)

```
/* C++: extern "C" long select(long cond, long a, long b);
 * if (cond == 0) return b;
 * else return a;
 */
```

```
.global select
select:
    # cond v RDI, a v RSI, b v RDX

    # Proveryaem RDI na nol'.
    # Operatsiya 'add' menyaet RFLAGS, v t.ch. Zero Flag (ZF)
    add rdi, 0

    # jz (Jump if Zero) - perehod, esli ZF=1 (rezul'tat byl 0)
    jz .L_return_b

.L_return_a:
    # cond != 0
    mov rax, rsi # return a
    ret

.L_return_b:
    # cond == 0
    mov rax, rdx # return b
    ret
```

6.5 Пример 5: Цикл (Sum)

```
/* C++: extern "C" long sum(long n);
 * long s = 0;
 * for (long i = n; i > 0; i--) { s += i; }
 * return s;
 */
```

```
.global sum
sum:
    # n v RDI
    xor rax, rax # rax (summa) = 0

.L_loop_start:
    add rax, rdi # summa += n
    add rdi, -1 # n--

    # 'add rdi, -1' (n--):
    # - Esli n > 0, perenosa (carry) ne budet.
    # - Esli n == 0, to 0 + (-1) daet perenos (borrow).

    # jnc (Jump if No Carry) - prygnut', esli n > 0
    jnc .L_loop_start

    # n == 0, tsikl zavershen
    ret
```

Итоги раздела

Итоги раздела "Ассемблер":

- Код на ассемблере — это прямое представление машинных инструкций (мнемоники).
- Взаимодействие с C/C++ происходит через ABI (Application Binary Interface) (соглашение о вызовах).
- В Linux x86-64 аргументы передаются через регистры (`RDI`, `RSI` и т.д.), а возвращаемое значение — через `RAX`.
- Аргументы, не поместившиеся в регистры (7-й и далее), передаются через стек и доступны по адресу `[rsp+8]`, `[rsp+16]` и т.д.
- Управление потоком (if, loop) реализуется через RFLAGS и инструкции условных переходов (`jz`, `jnc` и др.).