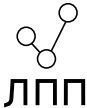


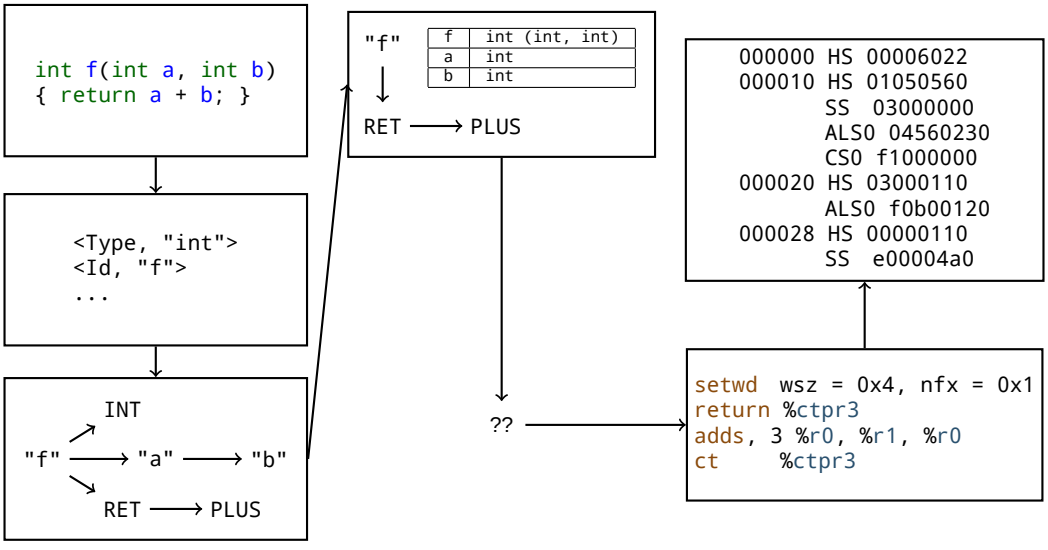
Проектирование компиляторов

Лекция 5.
Язык ассемблера и работа программы.

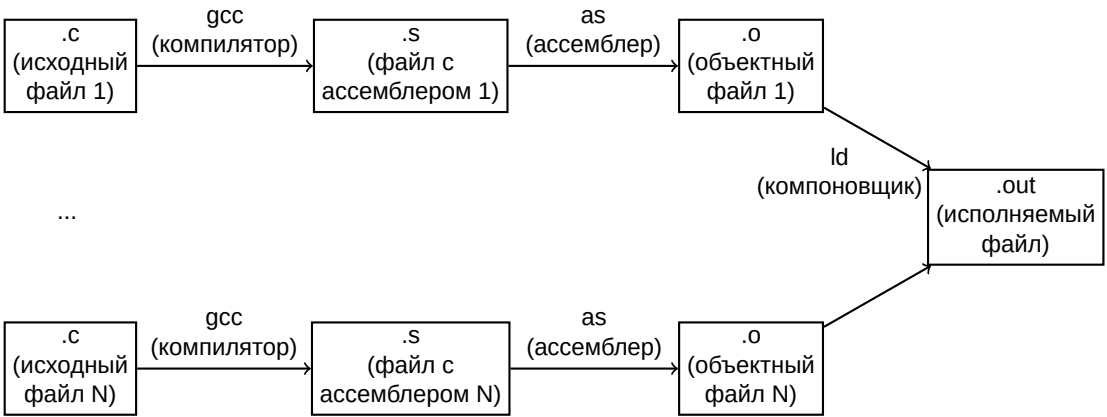
Маркин А. Л.
master@lab-pp.ru

2023

Компиляция программы



Сборка программы из нескольких модулей



Ассемблер

Ассемблер

Язык ассемблера — специальный низкоуровневый язык, близкий к целевой аппаратной платформе.

Часто инструкции ассемблера повторяют мнемоники из системы команд целевой платформы, а также имеют схожие обозначения регистров и структуру аргументов.

Однако, бывают и языки ассемблера, претендующие на некоторую универсальность. На практике же добиться универсальности низкоуровневого языка невозможно.

Ассемблер

Примеры программы на языках ассемблера для различных систем команд:

e2k

```
{
  setwd wsz = 0x8, nfx = 0x1, dbl = 0x0
  setbn rsz = 0x3, rbs = 0x4, rcur = 0x0
  disp  %ctpr1, puts
  getsp,0       _f32s,_lts1 0xfffffff0, %dr2
}
{
  nop 3
  addd,0 0x0, [ _f64,_lts0 .LC.2.1 ], %dr0
}
{
  call  %ctpr1, wbs = 0x4
}
{
  nop 5
  return        %ctpr3
  addd,3        0x0, 0x0, %dr0
}
{
  ct    %ctpr3
}
```

amd64

```
pushq   %rbp
movq    %rsp, %rbp
subq    $16, %rsp
movl    $0, -4(%rbp)
leaq    .L.str(%rip), %rdi
movb    $0, %al
callq   printf@PLT
xorl    %eax, %eax
addq    $16, %rsp
popq    %rbp
retq
```

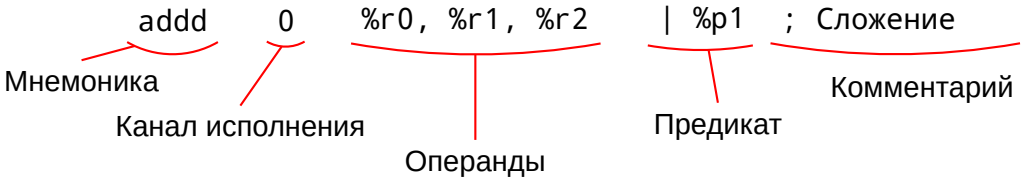
Risc-V

```
addi    sp, sp, -16
sw      ra, 12(sp)
sw      s0, 8(sp)
addi    s0, sp, 16
li      a0, 0
sw      a0, -16(s0)
sw      a0, -12(s0)
lui     a0, %hi(.L.str)
addi    a0, a0, %lo(.L.str)
call    printf
lw      a0, -16(s0)
lw      ra, 12(sp)
lw      s0, 8(sp)
addi    sp, sp, 16
ret
```

Ассемблер

Примеры программы на языках ассемблера, претендующих на универсальность:

Структура инструкции ассемблера



- Мнемоника — наглядное имя операции
- Канал исполнения — номер исполняющего устройства, в которое отправится инструкция
- Операнды — аргументы инструкции
- Предикат — логический регистр или выражение, разрешающее исполнение инструкции

Типы данных в языках ассемблера

Типом данных можно считать возможные типы операндов, принимаемых на вход инструкциями. Один из возможных типов операндов — регистры.

Регистр — ячейка внутренней памяти процессора. Типы регистров бывают:

- Целочисленные
- Плавающие
- Логические
- Адреса
- Упакованные (векторные)
- Любые другие, которые разработчик процессора посчитал необходимым

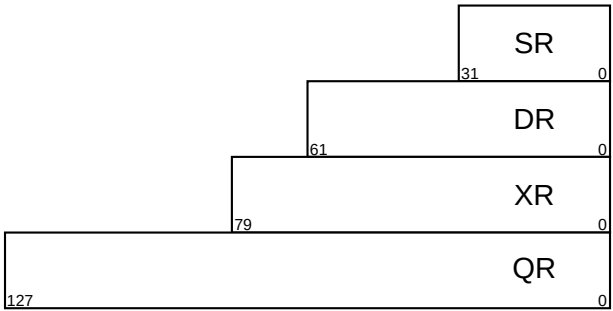
Типы данных в языках ассемблера

Доступ к данным из регистров является наиболее быстрым, поэтому для программиста выгодно иметь большое их количество. Однако для аппаратуры большое количество регистров сложнее в реализации.

- x86-64: 16 целочисленных регистров, 16 плавающих регистров
- ARM: 31 целочисленный регистр, 32 плавающих
- Sparc: 31 целочисленный регистр, 32 плавающих
- RISC V: 31 целочисленный регистр, 32 плавающих
- Itanium: 128 целочисленных регистров, 128 плавающих, 64 предикатных регистра
- Эльбрус: 256 смешанных регистров, 32 предикатных регистра

Типы данных в языках ассемблера

Помимо типа данных регистра, операнд также определяется и размером этого регистра:



Одинарный (Single)

Двойной (Double)

Расширенный (eXtended)

Квадро (Quadro)

Типы данных в языках ассемблера

Размер тип данных и его размер, с которым работает инструкция кодируется в самой инструкции:

- `adds` — сложение 32-битных целых чисел
- `addd` — сложение 64-битных целых чисел
- `addq` — сложение 128-битных целых чисел
- `fadds` — сложение 32-битных плавающих чисел
- `faddd` — сложение 32-битных плавающих чисел

Системные регистры

Помимо регистров общего назначения, бывают также и **системные регистры**. Это заранее определённые регистры, регулирующие работу процессора:

- `IP` — Адрес текущей команды.
- `NIP` — Адрес следующей команды.
- `PSR` — Регистр состояния процессора. Содержит флаги, управляющие работой процессора. В частности, в регистре содержится признак привилегированного режима.
- `FPFR` — Статусный регистр вещественной арифметики. Содержит тонкие настройки выполнения плавающих операций.

Объектные и исполняемые файлы

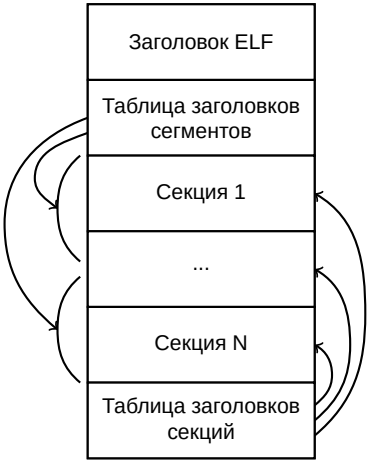
Формат объектных файлов

Elf (Executable and Linkable Format) — формат объектного файла.
Объектные файлы чаще всего бывают трёх видов:

- **Переадресуемые** — содержат код для связывания.
- **Исполняемые** — содержат код, исполняемый на целевой платформе.
- **Разделяемые объекты** — файлы, которые подходят для статического, так и для динамического связывания.

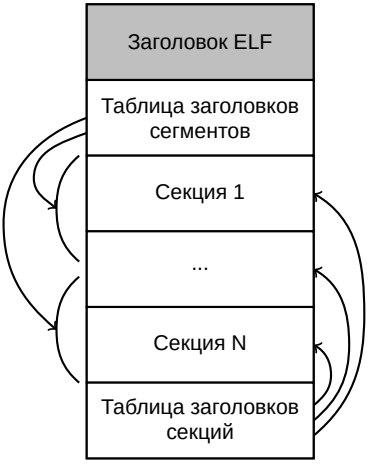
Объектные файлы содержат в себе код, предназначенный для исполнения на целевой платформе и являются двоичным представлением программы.

Структура объектного файла



Объектный файл состоит из заголовка и данных самого файла. Данные файла разделены на различные секции, которые могут объединяться в сегменты.

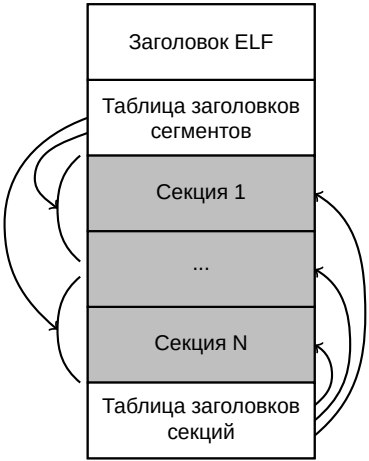
Структура объектного файла



Заголовок ELF содержит в себе данные об объектном файле: тип, версия формата, архитектура процессора, виртуальный адрес точки входа, размеры и смещения остальных частей файла.

```
$ readelf -h a.out
ELF Header:
 Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
 Class:                             ELF32
 Data:                              2's complement, little endian
 Version:                           1 (current)
 OS/ABI:                            UNIX - System V
 ABI Version:                       0
 Type:                              EXEC (Executable file)
 Machine:                           MCST Elbrus general purpose hardwa
...
```

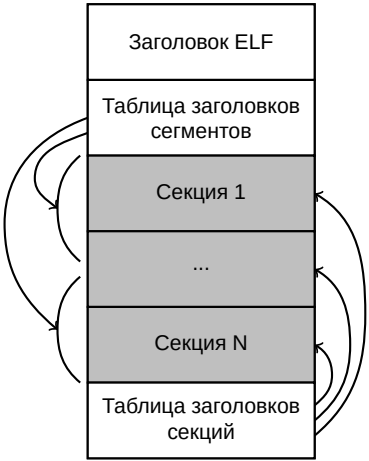
Структура объектного файла



Секции содержат непосредственно информацию, необходимую для исполнения программы. Она включает в себя код, данные и другую информацию.

```
$ readelf -S a.out
...
 [13] .text         PROGBITS     0000000000001080 00001080
    0000000000000168 0000000000000000 AX       0     0     16
...
 [23] .data         PROGBITS     0000000000004020 00003020
    0000000000000010 0000000000000000 WA       0     0     8
...
```

Структура объектного файла

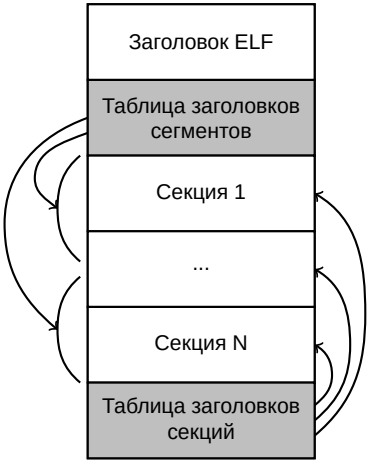


Чаще всего в программе присутствуют следующие секции:

- **Исполняемого кода** — `.text`
- **Изменяемых данных** — `.data`
- **Неизменяемых данных** — `.rodata`
- **Изменяемых неопределённых данных** — `.bss`

Эти секции заполнятся данными из кода программы и называются **пользовательскими**. Некоторые другие секции также считаются пользовательскими.

Структура объектного файла



Таблицы заголовков содержат ссылки на секции и сегменты, а также другую необходимую для работы программы информацию.

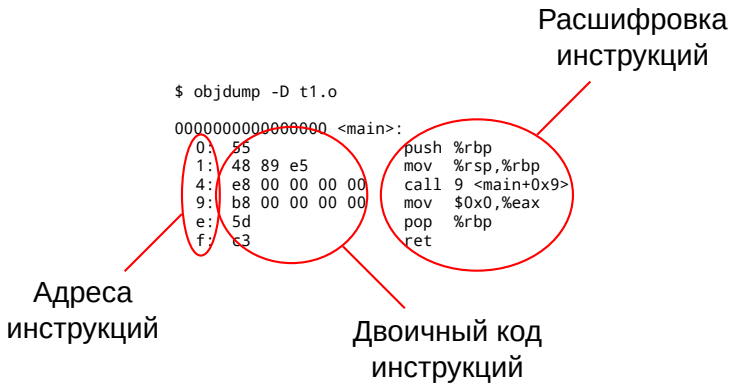
Сегменты — набор из нуля и более секций. Сегмент содержит информацию для операционной системы о том как загружать страницы секций в память.

Содержимое объектного файла

В объектном файле бывают и **не-пользовательские** секции. Они создаются компилятором. Несколько примеров:

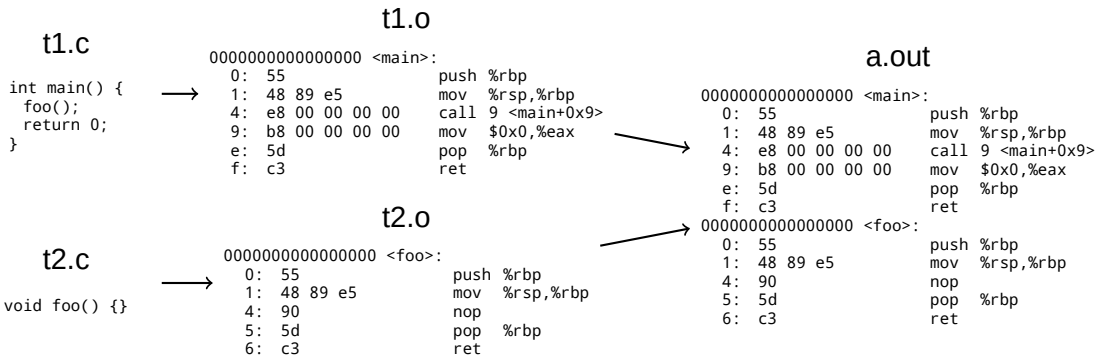
- `.dynamic` — секция с данными для динамического связывания.
- `.got` — глобальная таблица смещений (global offset table).
- `.plt` — таблица связывания процедур (procedure linking table).
- `.symtab` — таблица символов.

Содержимое объектного файла



Содержимое объектного файла

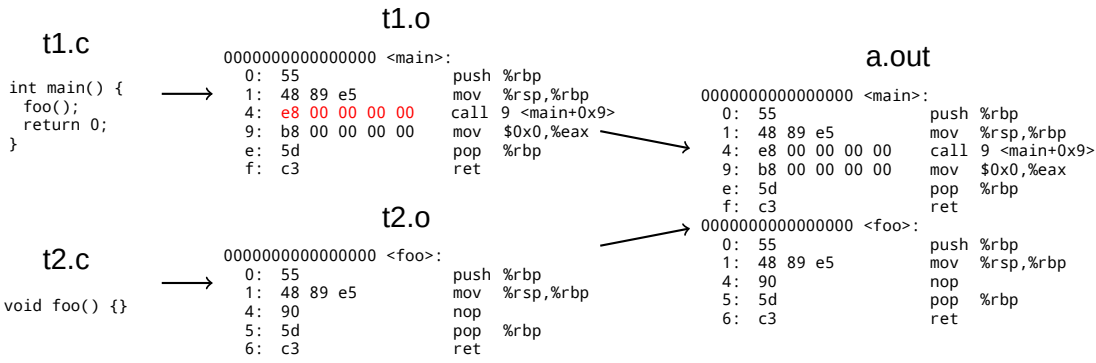
Будет ли работать программа, если код из объектных файлов слить в один файл (`cat t1.o t2.o > a.out`)?



Содержимое объектного файла

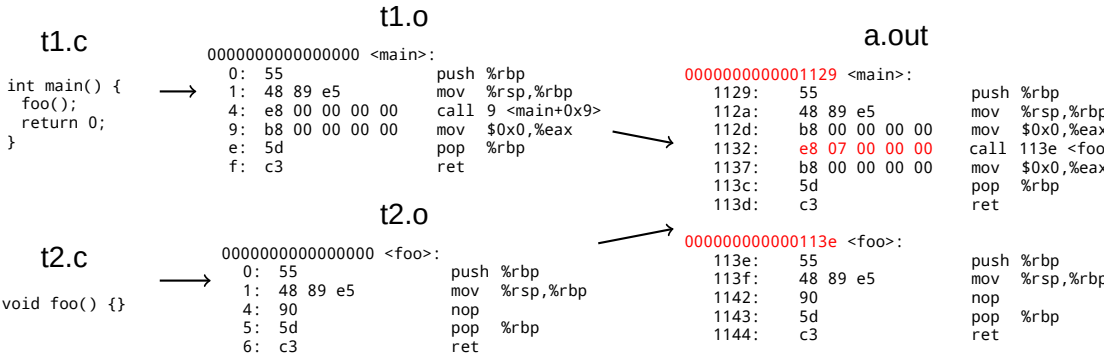
`e8` — код инструкции `call`.

`00 00 00 00` — значение для вычисления адреса вызываемой функции.



Связывание

Для получения правильного исполняемого файла необходимо произвести **связывание**: `ld t1.o t2.o -o a.out`.



Связывание

Связывание позволило присвоить адреса функциям и выставить правильный аргумент для инструкции вызова:

`07 00 00 00` — значение для вычисления адреса вызываемой функции.

a.out

```
0000000000001129 <main>:
    1129:    55                  push %rbp
    112a:    48 89 e5            mov  %rsp,%rbp
    112d:    b8 00 00 00 00      mov  $0x0,%eax
    1132:    e8 07 00 00 00      call 113e <foo>
    1137:    b8 00 00 00 00      mov  $0x0,%eax
    113c:    5d                  pop  %rbp
    113d:    c3                  ret

000000000000113e <foo>:
    113e:    55                  push %rbp
    113f:    48 89 e5            mov  %rsp,%rbp
    1142:    90                  nop
    1143:    5d                  pop  %rbp
    1144:    c3                  ret
```

Связывание

Про динамическое связывание

Загрузка исполняемого файла

Загрузка исполняемого файла в память

Что происходит когда мы запускаем исполняемый файл `./a.out`?

Операционная система загружает исполняемый файл в память при помощи специальной программы — **загрузчика** (loader).

Основной задачей загрузчика является перенести сегменты из файла в память и передать управление на точку входа программы.

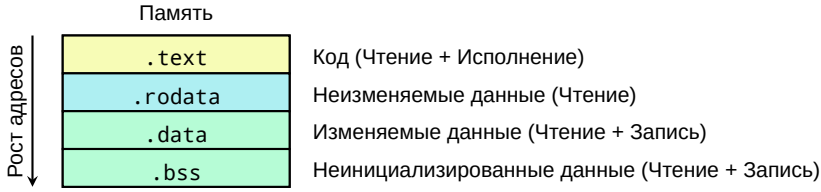
Загрузка исполняемого файла в память

В общих чертах загрузчик производит следующие действия:

1. Проверяет, что файл действительно является исполняемым. Для этого необходимо прочесть «магические» числа в начале файла.
2. Читает заголовок файла, из которого получает информацию о таблице сегментов, а потом и о самих сегментах.
3. Переносит все необходимые сегменты в память. В том числе заполняет нулями `.bss` секцию.
4. В процессе формирует различную служебную информацию, необходимую для работы программы. Записывает аргументы программы и переменные среды.
5. Передаёт управление на точку старта.

Загрузка исполняемого файла в память

Секции программы в памяти после загрузки:



Работа программы

Работа программы

Работа программы начинается из «точки старта». Эта точка определена в заголовке исполняемого файла в поле `e_entry`.

Работа программы

Работа программы начинается из «точки старта». Эта точка определена в заголовке исполняемого файла в поле `e_entry`.

Обычно точкой входа в программу обычно является функция `_start`. Её реализация системно-зависима, и обычно находится в библиотеке `crt0`. Данная функция нужна для:

- Инициализации регистров, стека, памяти.
- Настройки языковой среды исполнения, в том числе вызова глобальных функций-контсрукторов.
- Запуска функции `main`.
- Запуска глобальных функций-деструкторов, завершения исполнения программы.

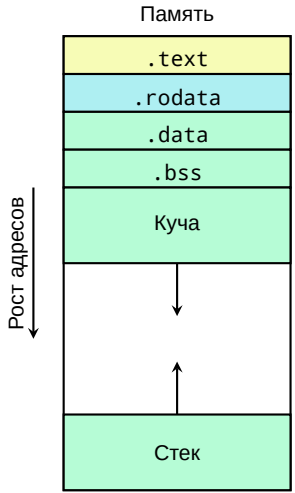
Работа программы

При работе программы программисту доступно два вида памяти:

Стек — область памяти, используемая процедурами для хранения локальных данных. Память на нём выделяется при старте процедуры и очищается после её завершения.

Куча — область памяти, используемая для хранения данных, чьё время жизни превышает время жизни процедуры. Память на ней выделяется пользователем при помощи функций вида `malloc` и `free`.

Работа программы



Стек и куча растут навстречу друг другу. В большинстве современных систем стек растёт вниз.

Некоторые процессоры позволяют выбирать режим роста стека, однако на практике стек, растущий вверх практически не встречается.

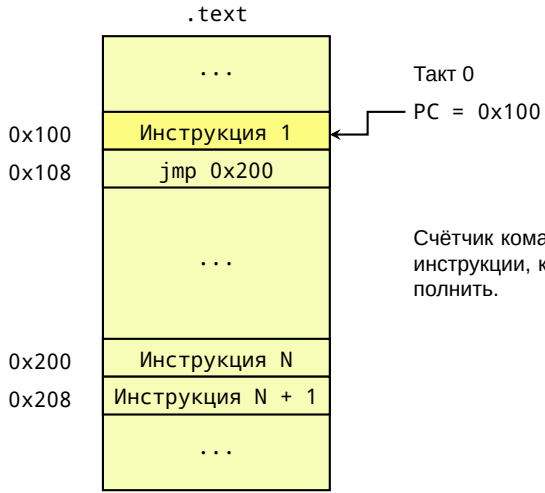
Работа программы

Счётчик команд — специальный регистр, который хранит адрес инструкции, которую следует начать выполнять.

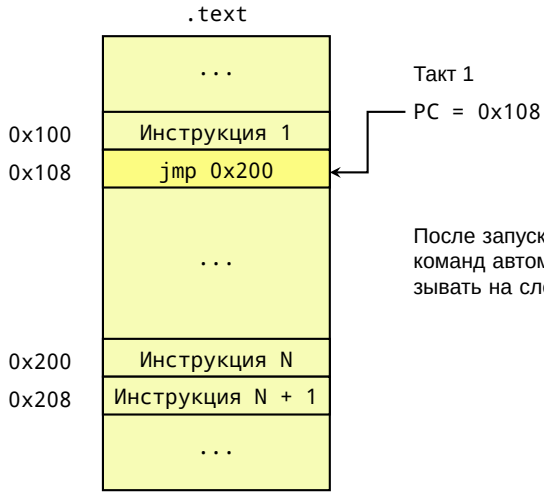
Чаще всего обозначается как **PC** (Program Counter) или как **IP** (Instruction Pointer).

Счётчик команд автоматически переключается на следующую команду. Также его может изменить инструкция передачи управления.

Работа программы

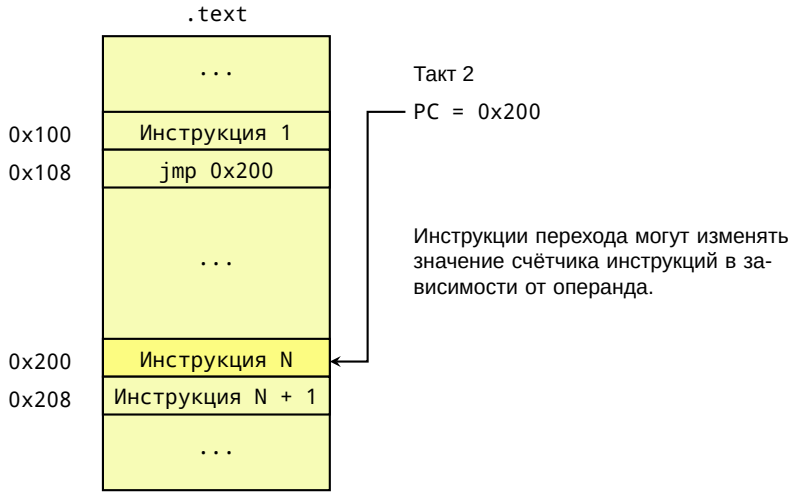


Работа программы



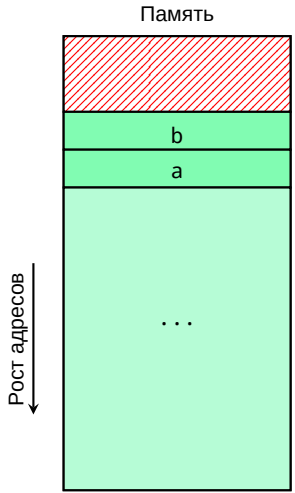
После запуска инструкции счётчик команд автоматически начинает указывать на следующую команду.

Работа программы



Процедурный механизм

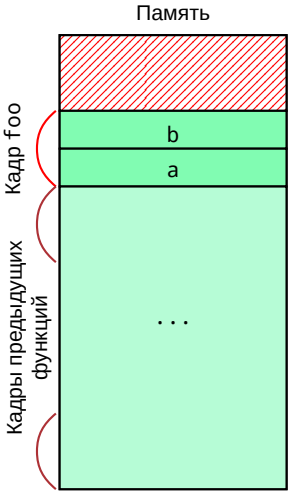
Локальные данные функции



Стек содержит локальные данные функции:

```
void foo() {
  int64 a;
  double b;
// ...
}
```

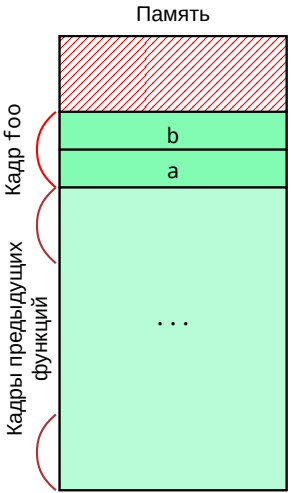
Локальные данные функции



Область стека, в которой лежат данные, принадлежащие конкретной функции называется **кадром стека** (stack frame).

```
void foo() {
  int64 a;
  double b;
// ...
}
```

Локальные данные функции



Границы кадра активной функции описываются двумя значениями:

- **Указатель кадра** (frame pointer) — значение, указывающее на начало выделенного кадра активной функции.
- **Указатель стека** (stack pointer) — значение, указывающее на вершину стека.

Расположение данных в памяти

Локальные данные процедуры

Механизм вызова функций

Как передавать аргументы

Механизм вызова функций

Что ещё расположено на стеке

Механизм вызова функций

Соглашения о вызовах

Литература

- *Э. Таненбаум* Архитектура компьютера.
- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- *J. R. Levine* Linkers & Loaders.
- *K. D. Cooper, L. Torczon* Engineering a compiler.
- *D. Drysdale* How programs get run: ELF binaries https://lwn.net/Articles/631631/
- *A. Aiken* CS143 Compilers. Stanford lectures.
- Executable and Linkable Format (ELF)