Булева алгебра

- **Предложена Джорджем Булем в XIX веке**
  - Алгебраическое представление одной из логик
    - Кодирует “Истина” как 1 и “Ложь” как 0

И (And)

- **A&B = 1 когда оба A=1 and B=1**

&	0	1
0	0	0
1	0	1

ИЛИ (Or)

- **A|B = 1 когда либо A=1, либо B=1**

|	0	1
0	0	1
1	1	1

НЕ(Not)

- **~A = 1 when A=0**

~	
0	1
1	0

Исключающее ИЛИ (Xor)

- **A^B = 1 когда либо A=1, либо B=1, но не оба**

^	0	1
0	0	1
1	1	0

Операции сдвига в Си

- **Сдвиг влево: `x << y`**
  - Сдвигает вектор битов x влево на y позиций
    - Вытолкнутые слева биты теряются
    - Заполняет нулями справа
- **Сдвиг вправо: `x >> y`**
  - Сдвигает вектор битов x вправо на y позиций
    - Вытолкнутые справа биты теряются
  - Логический сдвиг
    - Заполняет нулями справа
  - Арифметический сдвиг
    - Повторяет вправо наиболее значимый бит
- **Неопределённый результат**
  - Сдвиг на величину меньше 0 или больше размера слова

Аргумент x	01100010
<< 3	00010*000*
Логич. >> 2	*00*011000
Ариф. >> 2	*00*011000

Аргумент x	10100010
<< 3	00010*000*
Логич. >> 2	*00*101000
Ариф. >> 2	*11*101000

Соответствие знаковых и беззнаковых

Биты	Знаковое		Беззнаковое
0000	0		0
0001	1		1
0010	2		2
0011	3	=	3
0100	4		4
0101	5		5
0110	6		6
0111	7		7
1000	-8		8
1001	-7		9
1010	-6		10
1011	-5	$+/- 2^4$	11
1100	-4		12
1101	-3		13
1110	-2		14
1111	-1		15

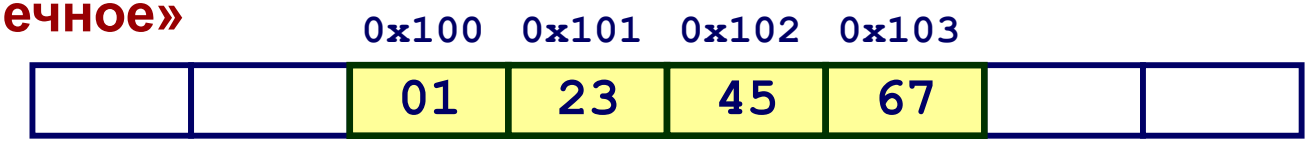
Порядок байт в слове

- **В каком порядке располагаются в памяти байты многобайтового слова?**
- **Соглашения**
  - «Тупоконечники»: Sun, PPC Mac, Internet
    - Наименее значимый байт имеет наибольший адрес
  - «Остроконечники»: x86
    - Наименее значимый байт имеет наименьший адрес

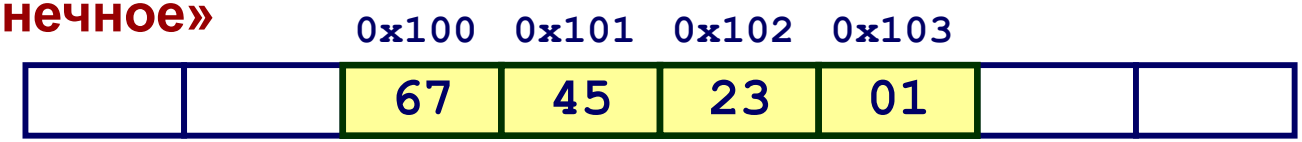
Примеры упорядочения байт

- **«Тупоконечное»**
  - Наименее значимый байт имеет наибольший адрес
- **«Остроконечное»**
  - Наименее значимый байт имеет наименьший адрес
- **Пример**
  - Переменная x
    - имеет 4-байтовое представление 0x01234567
    - Расположена по адресу &x - 0x100

«Тупоконечное»



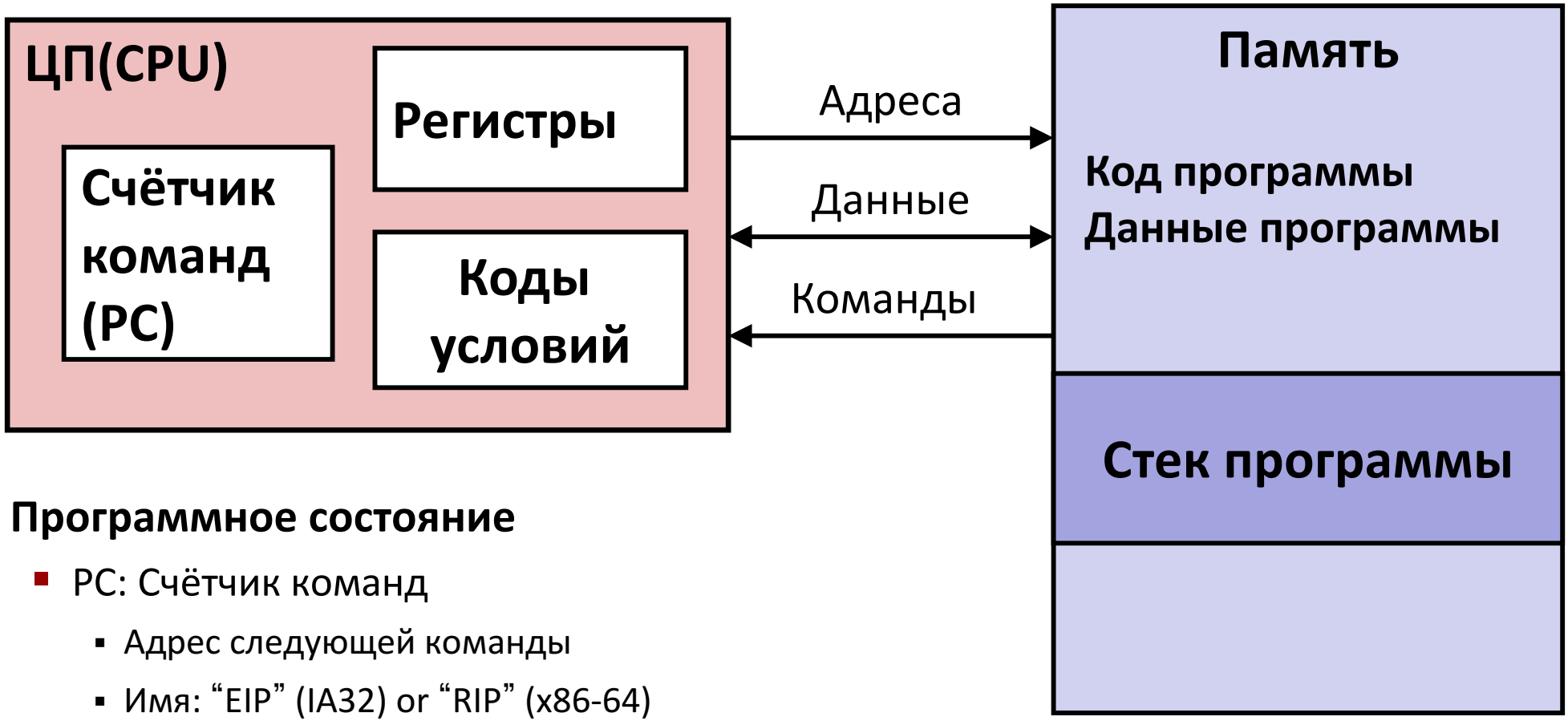
«Остроконечное»



Плавающая точка IEEE

- **Стандарт IEEE 754**
  - Принят в 1985 как единый стандарт арифметики с плавающей точкой
    - До этого множество уникальных стандартов
  - Поддерживается всеми основными CPU/FPU
- **В основе - вопросы вычислений**
  - Удачно стандартизованы
    - округления,
    - переполнения,
    - потеря значимости
  - Сложно сделать быстрым в аппаратуре
    - При создании стандарта численные аналитики доминировали над разработчиками аппаратуры
  - Есть реализации «с отклонениями»
- **Есть версии и альтернативы**

Программная модель ассемблера

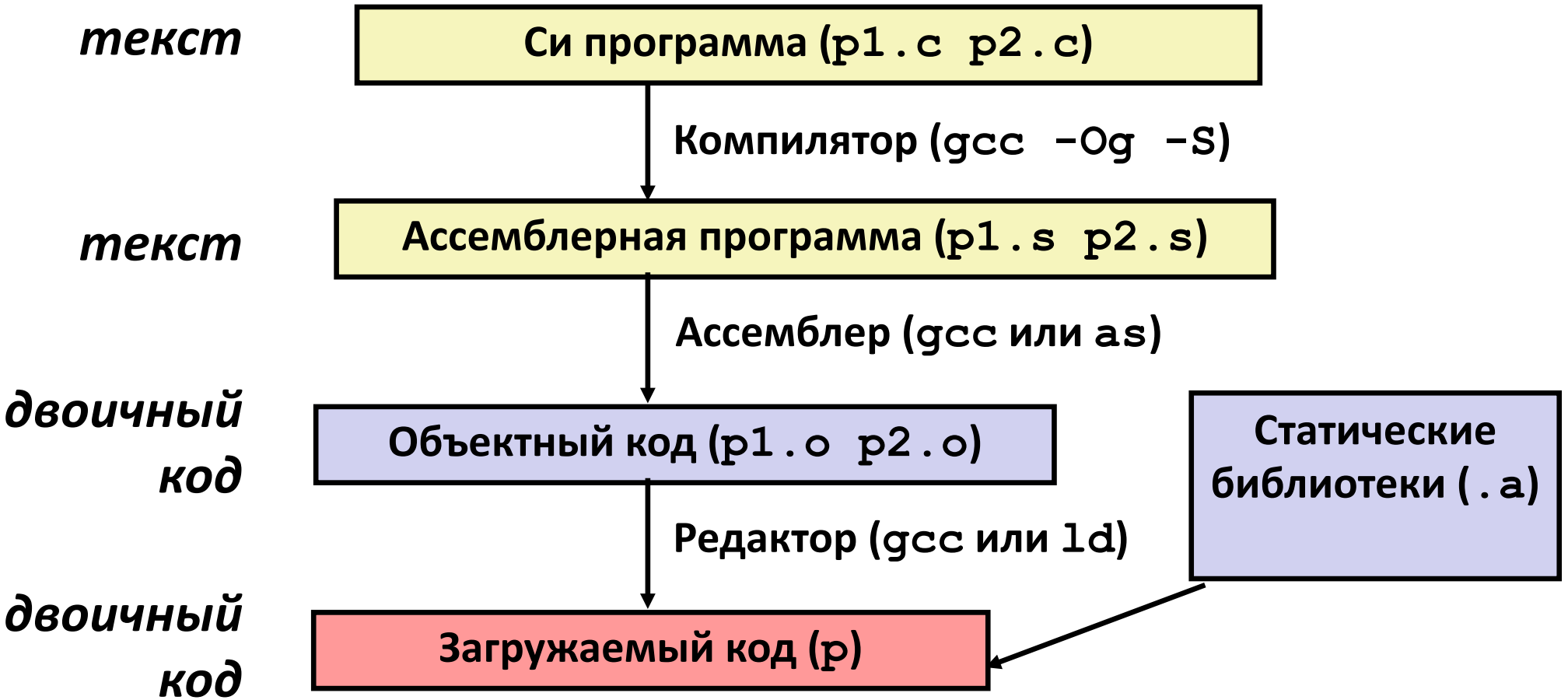


- **Программное состояние**
  - PC: Счётчик команд
    - Адрес следующей команды
    - Имя: “EIP” (IA32) or “RIP” (x86-64)
  - Набор регистров
    - Наиболее используемые данные
  - Коды условий
    - Хранят информацию о состоянии самой последней арифм. команды
    - Используются для условных переходов

- **Память**
  - Массив адресуемых байт
  - Код, данные пользователя
  - Включая стек для поддержки процедур

Трансляция Си кода в объектный

- Код в файлах `p1.c p2.c`
- Компилируем командой: `gcc –Og p1.c p2.c -o p`
  - Используем минимальную оптимизацию (`–Og`)
  - Помещаем результирующий двоичный код в файл `p`



Дизассемблирование объектного кода

Дизассемлированный код

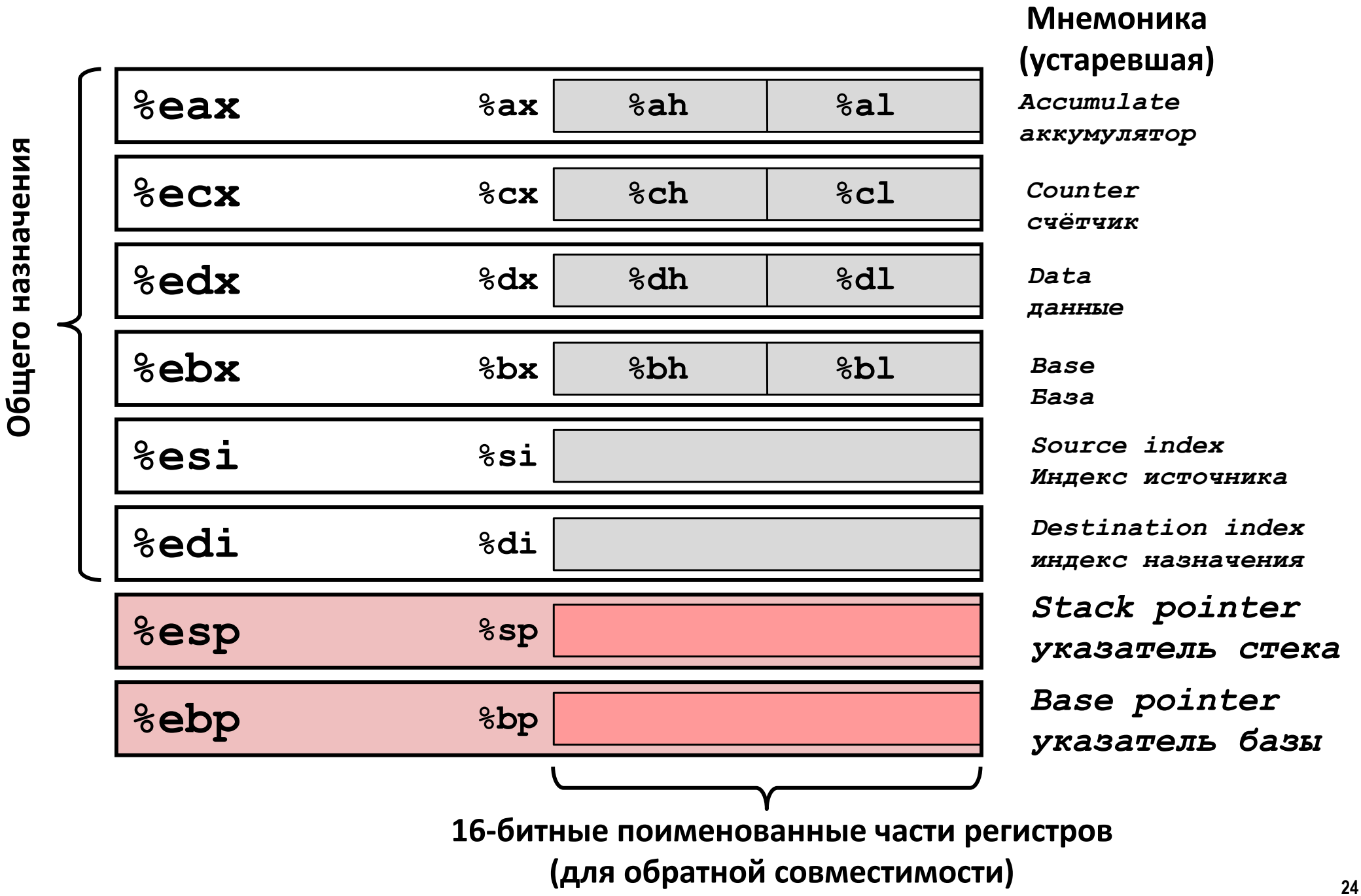
```
0000000000400595 <sumstore>:
  400595:  53                   push   %rbx
  400596:  48 89 d3             mov    %rdx,%rbx
  400599:  e8 f2 ff ff ff       callq  400590 <plus>
  40059e:  48 89 03             mov    %rax,(%rbx)
  4005a1:  5b                   pop    %rbx
  4005a2:  c3                   retq
```

- **Дизассемблер**

  `objdump -d sum`

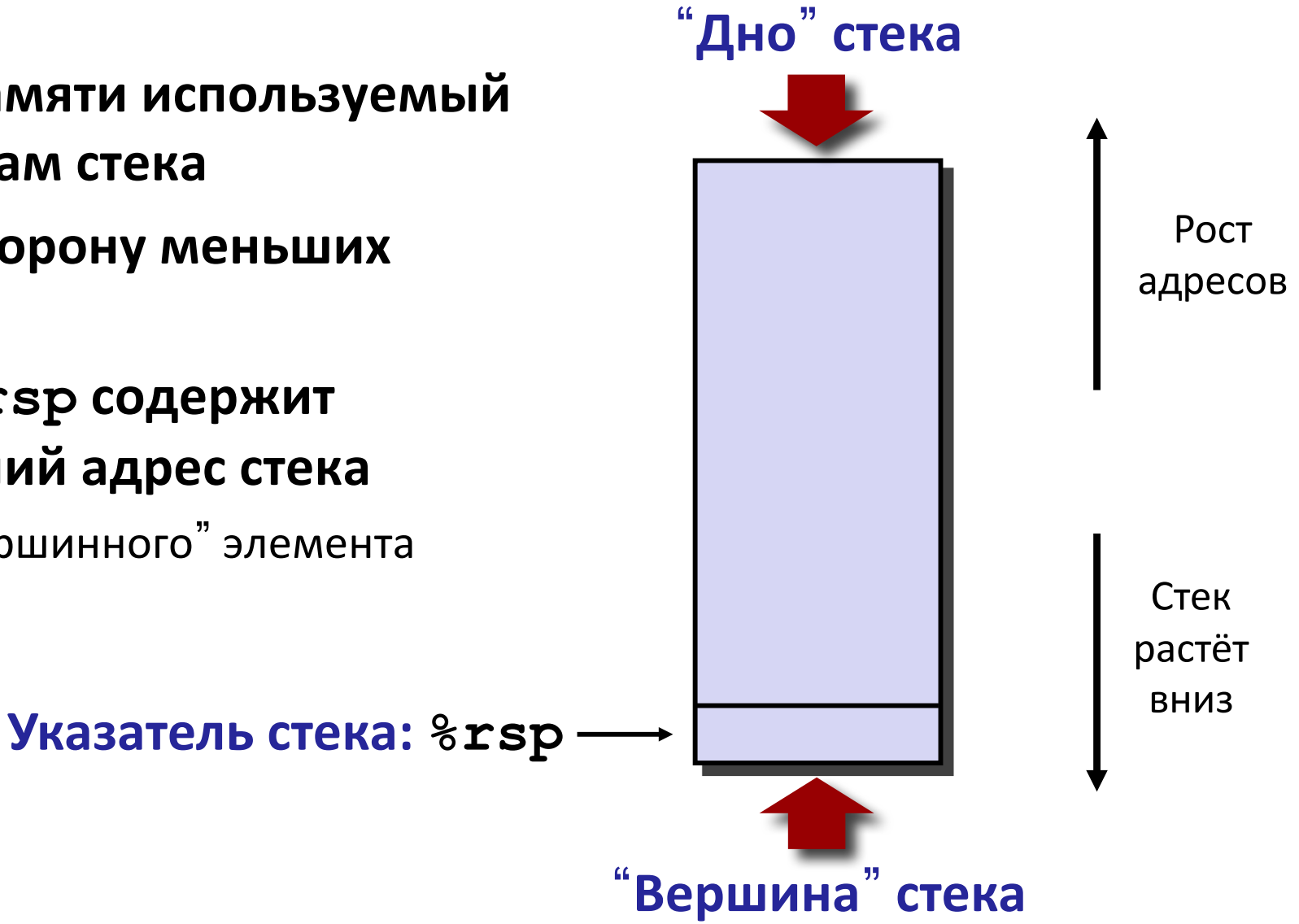
  - Полезный инструмент для анализа объектного кода
  - Анализирует битовые последовательности наборов команд
  - Приблизительно воссоздаёт ассемблерный код
  - Может обрабатывать файлы `a.out` (загрузочные) или `.o`

Почти история: регистры IA32



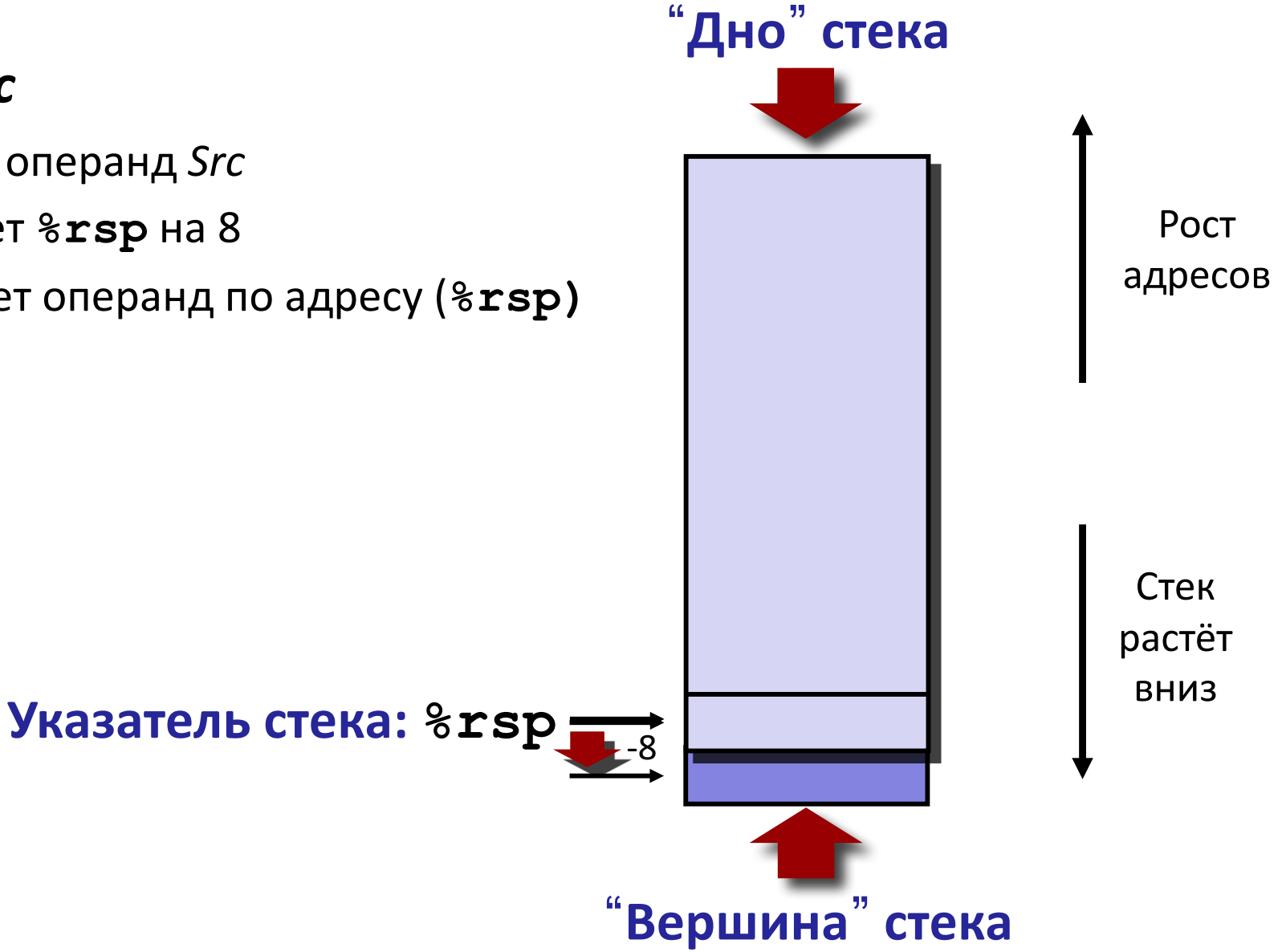
Стек x86-64

- **Участок памяти используемый по правилам стека**
- **Растёт в сторону меньших адресов**
- **Регистр `%rsp` содержит наименьший адрес стека**
  - адрес “вершинного” элемента



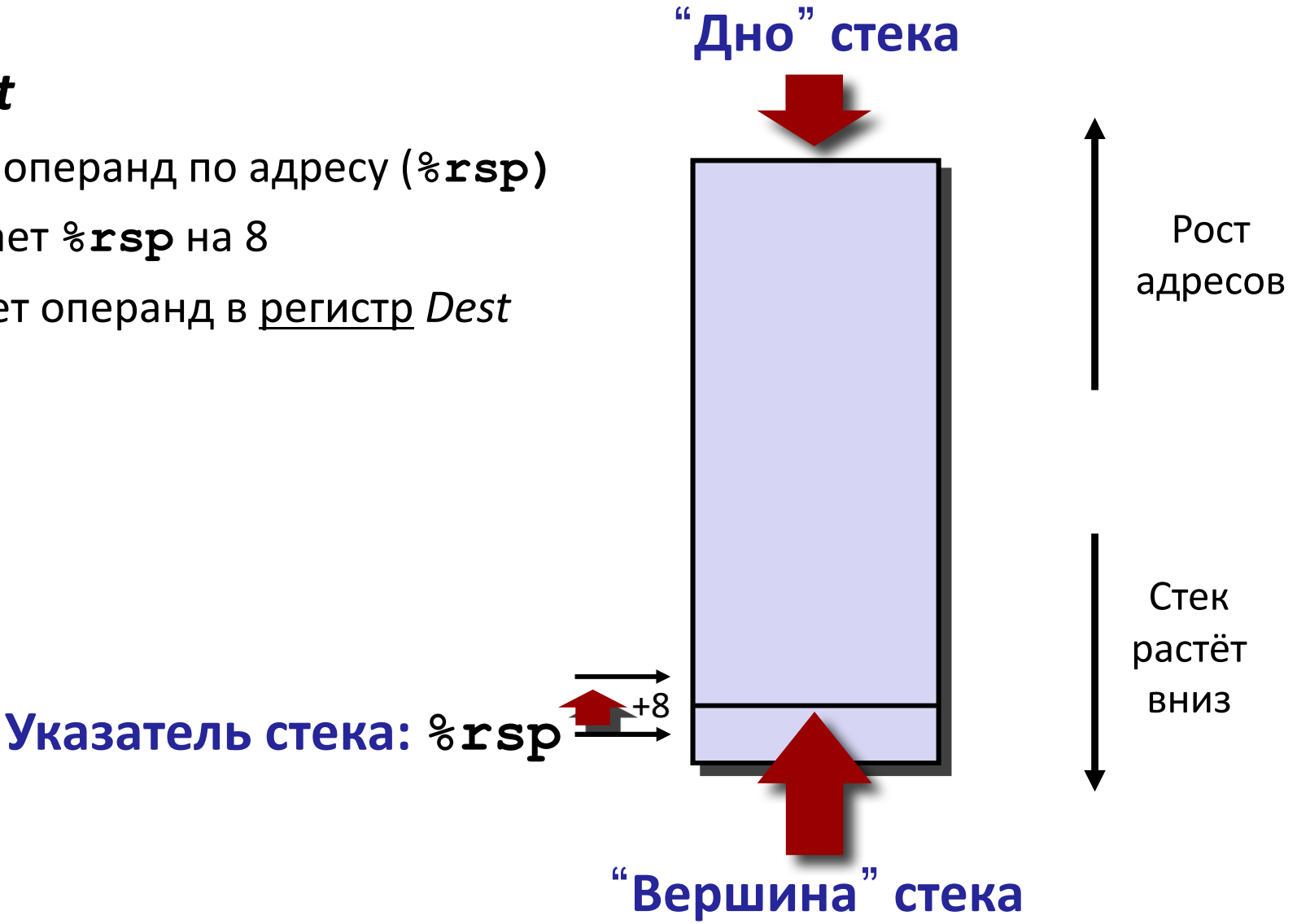
Стек x86-64: Вталкивание

- **`pushq` *Src***
  - Выбирает операнд *Src*
  - Уменьшает **`%rsp`** на 8
  - Записывает операнд по адресу (**`%rsp`**)



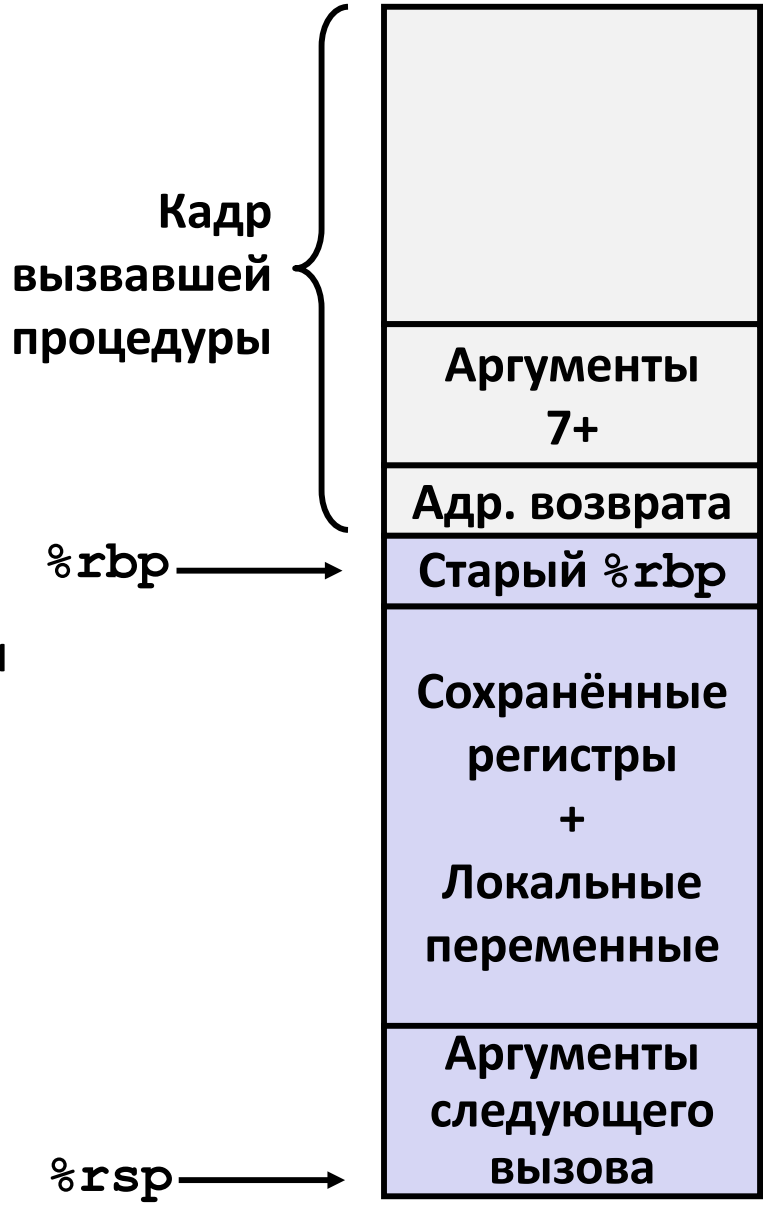
Стек x86-64: Выталкивание

- **`popq` *Dest***
  - Выбирает операнд по адресу (**`%rsp`**)
  - Увеличивает **`%rsp`** на 8
  - Записывает операнд в регистр *Dest*



Сводка: процедуры x86-64

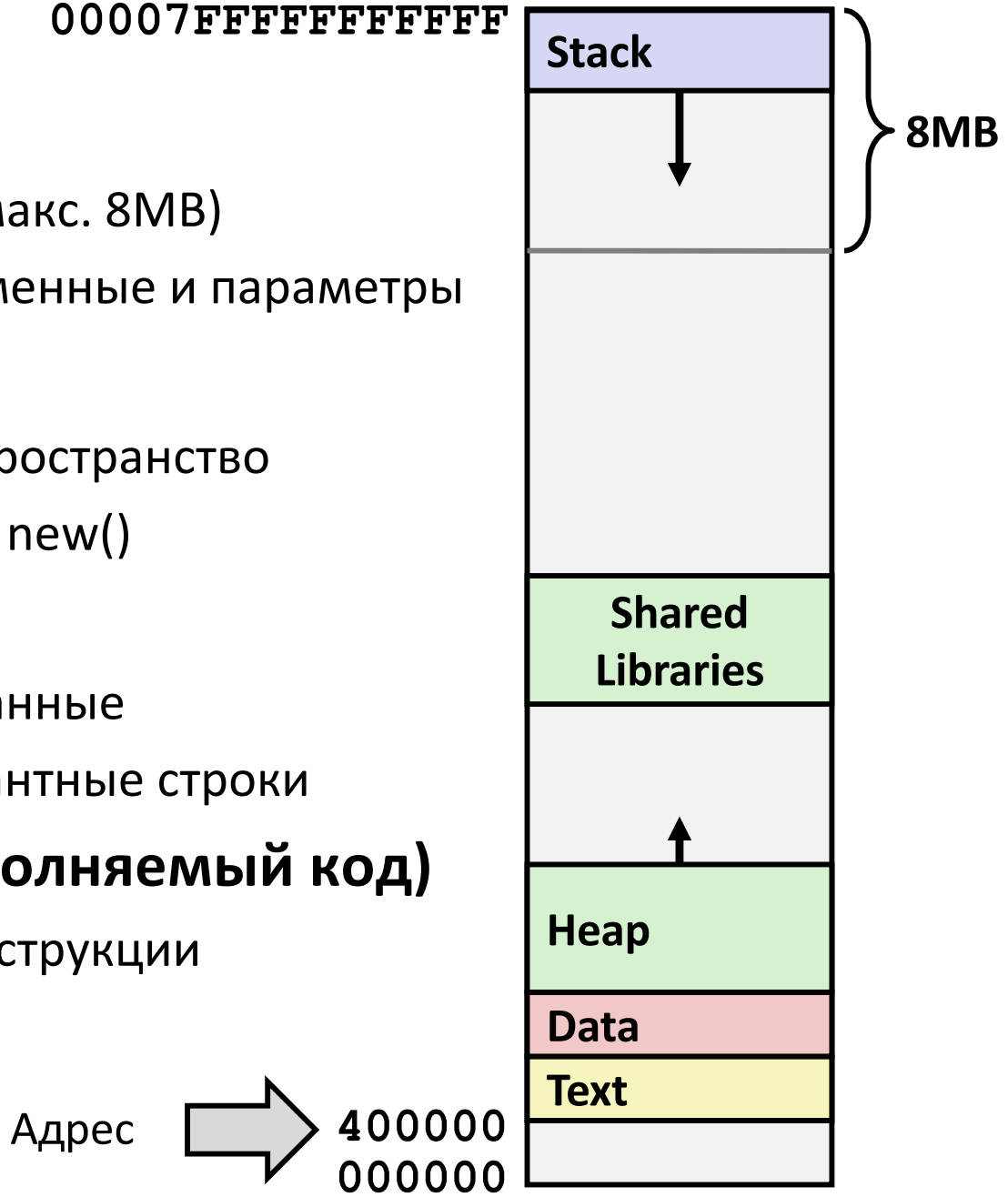
- **Важно!**
  - Стек – походящая структура данных для вызова процедур и возврата из них
    - если P вызывает Q, то возврат из Q раньше чем из P
- **(Взаимная) рекурсия реализуется обычными правилами вызова**
  - Значения безопасно хранятся в своём кадре и регистрах сохраняемых вызванной
  - Аргументы функции – на вершине стека
  - Результат возвращается в **`%rax`**
- **`Указатели – адреса значений`**
  - **В стеке или среди глобальных переменных**



Распределение памяти
x86-64 Linux

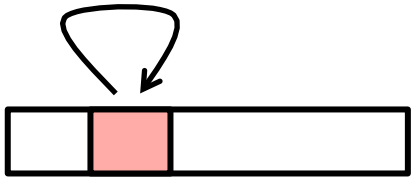
Не в масштабе

- **Stack (стек)**
  - Стек времени исполнения (макс. 8MB)
  - Например, локальные переменные и параметры
- **Heap (куча)**
  - Динамически занимаемое пространство
  - При вызове malloc(), calloc(), new()
- **Data (данные)**
  - Статически размещаемые данные
  - Например, массивы и константные строки
- **Text и Shared Libraries (исполняемый код)**
  - Исполняемые машинные инструкции
  - Только чтение

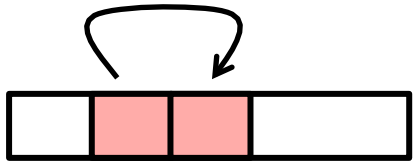


Локальность

- **Принцип локальности : Программы чаще обращаются к данным и командам, чьи адреса близкие к тем которые недавно использовались**



- **ВременнАя локальность:**
  - Недавно использованные ячейки вероятно вскоре будут использованы вновь



- **Пространственная локальность:**
  - К ячейкам с ближайшими адресами обращения произойдут в ближайшее время

Пример локальности

- **Вопрос: Вы можете переставить циклы так, чтобы функция сканировала 3-мерный массив `a` также эффективно как одномерный? С хорошей пространственной локальностью.**

```
int sum_array_3d(int a[M][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < M; i++)
        for (j = 0; j < N; j++)
            for (k = 0; k < N; k++)
                sum += a[k][i][j];
    return sum;
}
```

Пример иерархии хранения данных

Диаграмма быстродействия памяти