

Компьютерные основы программирования
Представление программ
часть 1

Лекция 4, 9 марта 2017

Лектор: Чуканова Ольга
Владимировна
Кафедра информатики
602 АК
ovcha@mail.ru

Машинный уровень

- Краткая история изделий Интел
- Си, ассемблер, машинный код
- Основы ассемблера: регистры, операнды, пересылки
- Немного об x86-64
- Полная адресация, вычисление адреса (leal)
- Арифметические операции
- Управление: флаги условий
- Условные переходы и пересылки
- Циклы
- Операторы переключения

Процессоры Intel x86

- Подавляющее доминирование на рынках настольных ПК, ноутбуков, серверов
- Эволюционное конструирование
 - Обрато совместимы до 8086, выпущенного в 1978
 - Функции добавляются с ходом времени
- Complex instruction set computer (CISC)
 - Много команд во многих форматах
 - Но, лишь немногие встречаются в Linux программам
 - Трудно конкурировать по быстродействию с Reduced Instruction Set Computers (RISC)
 - Однако, Intel сделал именно это!
 - В терминах быстродействия. Но не энергопотребления.

Вехи эволюции Intel x86

<i>Имя</i>	<i>Дата</i>	<i>Транзисторов</i>	<i>МГц</i>
■ 8086	1978	29K	5-10
<ul style="list-style-type: none">▪ 1-й 16-битный процессор. Основа для IBM PC & DOS▪ 1MB адресного пространства			
■ 386	1985	275K	16-33
<ul style="list-style-type: none">▪ 1-й 32-битный процессор, известен как IA32▪ Добавлена “плоская адресация”▪ Способен исполнять Unix▪ 32-битный Linux/gcc не использует более поздние команды			
■ Pentium 4F	2004	125M	2800-3800
<ul style="list-style-type: none">▪ 1-й 64-битный процессор, известен как x86-64			
■ Core i7	2008	731M	2667-3333
<ul style="list-style-type: none">▪ Современные машины			

Микропроцессоры Intel

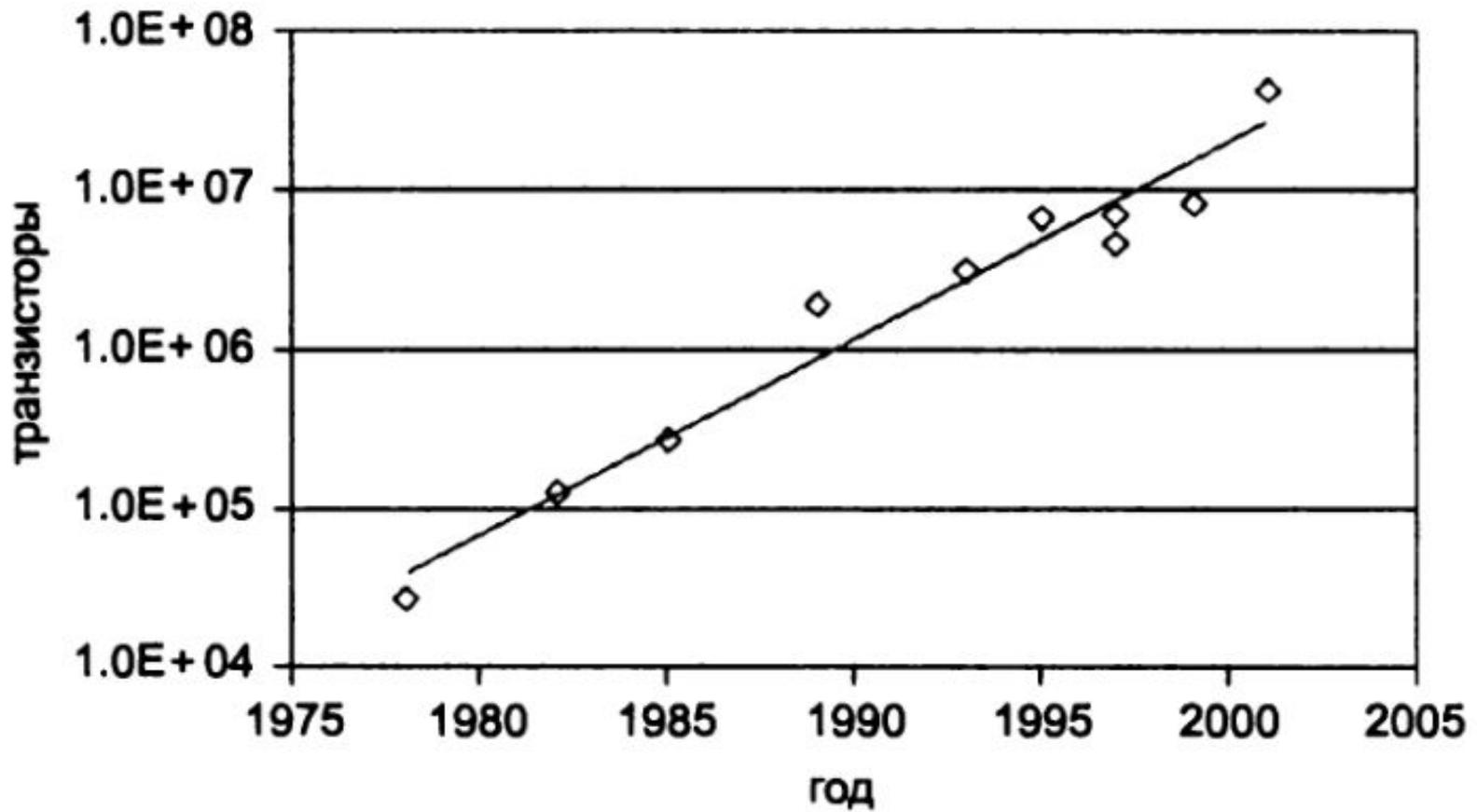


Рис. 3.1. Закон Мура

Процессоры Intel x86 : Обзор

Архитектуры	Процессоры
X86-16	8086
	286
X86-32/IA32	386
	486
MMX	Pentium
	Pentium MMX
SSE	Pentium III
SSE2	Pentium 4
SSE3	Pentium 4E
X86-64 / EM64t	Pentium 4F
SSE4	Core 2 Duo
	Core i7

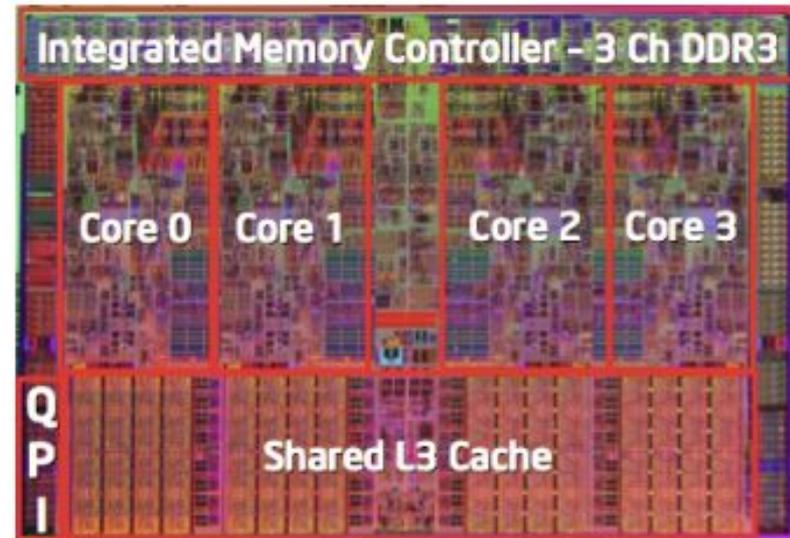


IA: часто переопределяется как последняя архитектура Intel

Процессоры Intel x86, далее

■ Эволюция машин

■ 386	1985	0.3M
■ Pentium	1993	3.1M
■ Pentium/MMX	1997	4.5M
■ PentiumPro	1995	6.5M
■ Pentium III	1999	8.2M
■ Pentium 4	2001	42M
■ Core 2 Duo	2006	291M
■ Core i7	2008	731M



■ Добавленные возможности

- Команды обработки данных мультимедиа(multimedia)
 - Параллельные операции с 1, 2, 4-байтовыми данными, целыми и с плавающей точкой
- Команды для более эффективной условной обработки

■ Эволюция Linux/GCC

- Два главных шага: 1) 32-битный 386. 2) 64-битный x86-64

x86 Clones: Advanced Micro Devices (AMD)

■ Сначала

- AMD догоняла Intel
- Немного медленнее, много дешевле

■ Потом

- Привлекла лучших аппаратчиков из Digital Equipment Corp. И других компаний
- Создала Opteron: сильного соперника Pentium 4
- Разработала x86-64, собственное 64-битное расширение

64-битный Intel

- Intel попытался радикально сменить IA32 на IA64
 - Полностью другая архитектура (Itanium)
 - Исполнение кода IA32, только как устаревшего
 - Разочаровывающее быстродействие
- AMD вышла с эволюционным решением
 - x86-64 (сейчас известна как “AMD64”)
- Intel нарушил обязательство сосредоточения на IA64
 - Трудно признавать ошибки и превосходство AMD
- 2004: Intel анонсировал EM64T расширение к IA32
 - 64-битная технология расширенной памяти
 - Почти идентична x86-64!
- Сейчас все “low-end” процессоры x86 содержат x86-64
 - Однако большинство кода - всё ещё в 32-битном режиме

Определения

- **Архитектура(системы команд)**, также instruction set architecture: ISA. Часть конструкции процессора которую надо знать для понимания ассемблерного кода.
 - Примеры: описание набора команд, регистры.
- **Микроархитектура**: Реализация архитектуры.
 - Примеры: размер кеша и частота ядра.

Классификация архитектур системы команд

Классификация по составу и сложности команд

- Архитектуры с полным набором команд: **CISC** (Complex Instruction Set Computer)
- Архитектуры с сокращенным набором команд: **RISC** (Reduced Instruction Set Computer)
- Архитектуры с параллелизмом на уровне команд: **ILP** (Instruction Level Parallelism)

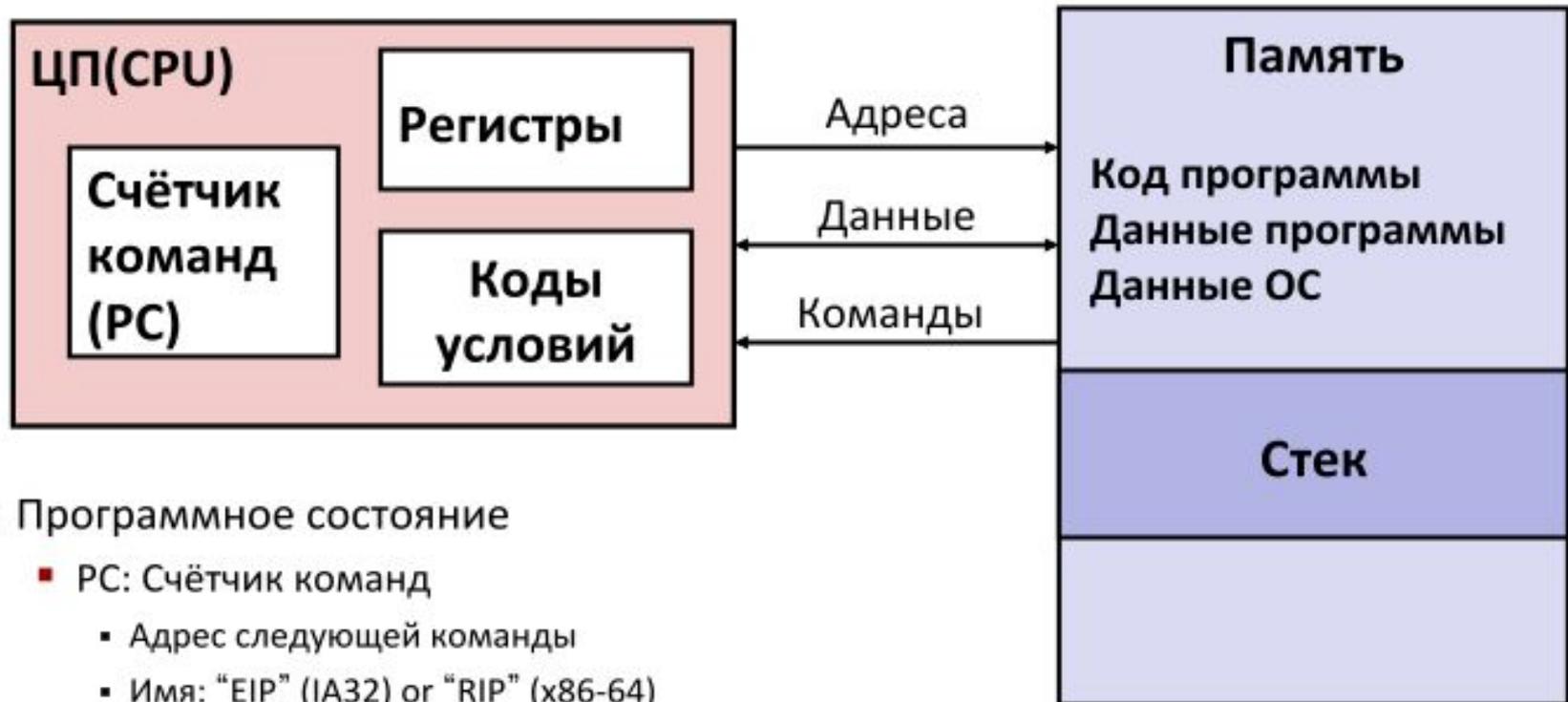
Классификация архитектур системы команд

Классификация по месту хранения операндов

- Регистровая
- Стековая
- С выделенным доступом к памяти
- Комбинированная

Выбор той или иной архитектуры влияет на принципиальные моменты: сколько адресов будет содержать адресная часть команд, какова длина этих адресов, насколько просто будет происходить доступ к операндам и какой, в конечном итоге, будет длина команд

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 адресованных байт
- Код, данные пользователя, (некоторые) данные ОС
- Включая стек для поддержки процедур

Форматы команд

Операционная часть	Адресная часть
---------------------------	-----------------------

- **Длина команды**

Для ускорения выборки из памяти желательно, чтобы команда была как можно короче, а ее длина была равна или кратна ширине шины данных

- **Количество адресов в команде**

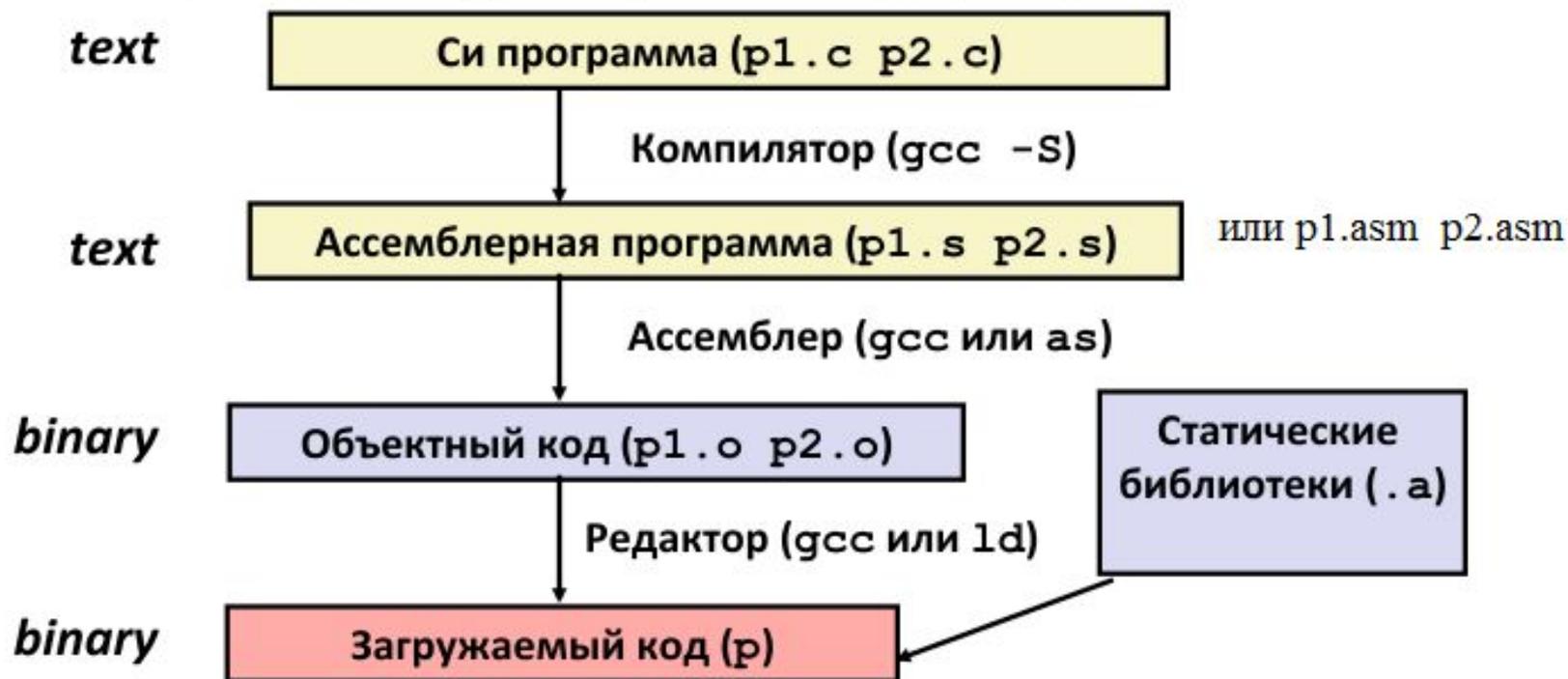
Четырехадресные, трехадресные, двухадресные, одноадресных, нульадресные

- **Разрядность полей команды**

- **Способы адресации**

Трансляция Си кода в объектный

- Код в файлах `p1.c p2.c`
- Компилируем командой: `gcc -O1 p1.c p2.c -o p`
 - Используем минимальную оптимизацию (`-O1`)
 - Помещаем результирующий двоичный код в файл `p`



Компиляция в ассемблер

Си - код

```
int sum(int x, int y)
{
    int t = x+y;
    return t;
}
```

Некоторые компиляторы используют команду "leave"

Сгенерированный код ассемблера IA32

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    popl %ebp
    ret
```

Получается командой

```
/usr/local/bin/gcc -O1 -S code.c
```

Создаёт файл code.s

```
sum proc    near
    push    bp
    mov     bp,sp
    mov     ax, word ptr [bp+4]
    add     ax, word ptr [bp+6]
    pop     bp
    ret
```

Типы данных ассемблера

- “Целые” 1-, 2-, или 4- байтные данные
 - Значения данных
 - Адреса (нетипизованные, байтные указатели)
- 4-, 8-, или 10- байтные данные с плавающей точкой
- Никаких сложных типов как массивы или структуры
 - Просто байты расположенные в памяти рядом

Операции ассемблера

- Выполняют арифметические операции с данными в регистрах или в памяти
- Передают данные между памятью и регистрами
 - Загружают данные из памяти в регистры
 - Выгружают данные из регистров в память
- Передача управления
 - Безусловные переходы
 - Условные переходы
 - Вызов процедур и возврат из них

Объектный код

Код `sum`

```
0x401040 <sum>:  
  0x55  
  0x89  
  0xe5  
  0x8b  
  0x45  
  0x0c  
  0x03  
  0x45  
  0x08  
  0x5d  
  0xc3
```

- **Всего 11 байт**
- **Каждая команда 1, 2, or 3 bytes**
- **Старт по адресу 0x401040**

■ Ассемблер

- Транслирует `.s` into `.o`
- Кодирует команды двоичным кодом
- Почти готовый к загрузке код
- Межфайловых связей кода

■ Редактор (связей)

- Выполняет ссылки между файлами
- Компонует со статическими библиотеками
 - Например, код для `malloc`, `printf`
- Некоторые библиотеки *связываются динамически*
 - Связывание происходит при исполнении

Форматы команд

Операционная часть	Адресная часть
---------------------------	-----------------------

- **Длина команды**

Для ускорения выборки из памяти желательно, чтобы команда была как можно короче, а ее длина была равна или кратна ширине шины данных

- **Количество адресов в команде**

Четырехадресные, трехадресные, двухадресные, одноадресных, нульадресные

- **Разрядность полей команды**

- **Способы адресации**

Пример машинных команд

```
int t = x+y;
```

```
addl 8(%ebp), %eax
```

Аналогично выражению:

`x += y`

Более точно:

```
int eax;
```

```
int *ebp;
```

```
eax += ebp[2]
```

```
0x80483ca: 03 45 08
```

■ Си код

- Сложить два знаковых

■ Ассемблер

- Сложить 2 4-байтных целых
 - “Long” слова в манере GCC
 - те-же команды и для signed и для unsigned

■ Операнды:

x: Регистр **%eax**

y: Память **M[%ebp+8]**

t: Регистр **%eax**

– Возвращает в **%eax**

■ Объектный код

- 3-байтные команды
- по адресу **0x80483ca**

Дизассемблирование объектного кода

Дизассемлированный код

```
080483c4 <sum>:  
80483c4: 55          push    %ebp  
80483c5: 89 e5      mov     %esp, %ebp  
80483c7: 8b 45 0c   mov     0xc(%ebp), %eax  
80483ca: 03 45 08   add     0x8(%ebp), %eax  
80483cd: 5d        pop     %ebp  
80483ce: c3        ret
```

■ Дизассемблер

`objdump -d p`

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

Вариант дизассемблирования

Объектный код

Дизассемблированный код

0x401040:

0x55

0x89

0xe5

0x8b

0x45

0x0c

0x03

0x45

0x08

0x5d

0xc3

Dump of assembler code for function sum:

```
0x080483c4 <sum+0>:      push   %ebp
0x080483c5 <sum+1>:      mov    %esp,%ebp
0x080483c7 <sum+3>:      mov    0xc(%ebp),%eax
0x080483ca <sum+6>:      add   0x8(%ebp),%eax
0x080483cd <sum+9>:      pop   %ebp
0x080483ce <sum+10>:     ret
```

- С отладчиком gdb

```
gdb p
```

```
disassemble sum
```

- Дизассемблируемая процедура

```
x/11xb sum
```

- Просмотреть 11 байт начиная с `sum`

Что удастся дизассемблировать?

- Всё, что имеет смысл исполняемых команд
- Дизассемблер только принимает предложенные байты и воспроизводит их как ассемблерный код

```
1  int simple(int *xp, int y)
2  {
3      int t = *xp + y;
4      *xp = t;
5      return t;
6  }
```

```
.file      "simple.c"
.version   "01.01"
gcc2_compiled.:
.text
    .align 4
.globl simple
    .type simple,@function
simple:
    pushl   %ebp
    movl    %esp,%ebp
    movl    8(%ebp),%eax
    movl    (%eax),%edx
    addl    12(%ebp),%edx
    movl    %edx,(%eax)
    movl    %edx,%eax
    movl    %ebp,%esp
    popl    %ebp
    ret
.Lfel:
```

```
1 int simple(int *xp, int y)
2 {
3     int t = *xp + y;
4     *xp = t;
5     return t;
6 }
```

```
1 simple:
2     pushl   %ebp           Сохранить указатель фрейма
3     movl   %esp, %ebp     Установить новый указатель фрейма
4     movl   8(%ebp), %eax   Взять xp
5     movl   (%eax), %edx   Извлечь *xp
6     addl   12(%ebp), %edx  Прибавить y, чтобы получить t
7     movl   %edx, (%eax)   Сохранить t по адресу *xp
8     movl   %edx, %eax     Установить t как возвращаемое значение
9     movl   %ebp, %esp    Переустановить указатель стека
10    popl   %ebp           Переустановить указатель фрейма
11    ret                Возврат результата
```

`mov bp,sp ;установить новый указатель`

```
1 int simple(int *xp, int y)
2 {
3     int t = *xp + y;
4     *xp = t;
5     return t;
6 }
```

1 значение по адресу xp
2 {
3 значение по адресу *xp прибавить y, чтобы получить t
4 сохранить t по адресу xp
5 как возвращаемое значение
6 следующее значение указателя

```
simple proc near
    push bp ;сохранить предыдущее значение указателя
    mov bp,sp ;установить новый указатель
    mov si, word ptr [bp] ; извлечь адрес xp
    mov dx, word ptr [si] ; извлечь значение по адресу - *xp
    add dx, word ptr [bp+6] ;прибавить y, чтобы получить t
    mov word ptr [si], dx ;сохранить t по адресу xp
    mov ax, dx ;установить t как возвращаемое значение
    pop bp ; восстановить предыдущее значение указателя
    ret ; возврат
```