

Юрий Анатольевич Богоявленский, заведующий кафедрой
Информатики и математического обеспечения, к.т.н., доцент:
ybgv@cs.petrSU.ru, <https://cs.petrSU.ru>

Веб страница дисциплины

<https://cs.petrSU.ru/~chistyak/architecture/>

Лекция 8. Формирование кадра стека функции при ее вызове по соглашениям ABI Intel386 Architecture

План лекции

Схема организации функций по соглашениям о вызовах ABI.....	2
Исходное состояние стека.....	4
Работает вызывающая функция.....	4
Выполнение команды pusha. Состояние 0.....	4
Помещение параметров в стек. Состояние 1.....	5
Вызов функции командой call. Состояние 2.....	6
Работает вызванная функция f1.....	6
Выполнение первой обязательной команды пролога pushl %ebp.	
Состояние 3.....	6
Выполнение второй обязательной команды пролога movl %esp,%ebp.	
Состояние 4.....	8
Резервирование памяти для локальных переменных в кадре стека.	
Состояние 5.....	9
Примеры адресного доступа к параметрам и локальным переменным	
.....	11
Вложенный в функцию f1 вызов функции f2. Состояние 6.....	11
Возврат из вызванной функции в вызывающую.....	13
Выполнение первой обязательной команды эпилога movl %ebp, %esp. Возврат в	
состояние 4.....	14
Выполнение второй обязательной команды эпилога popl %ebp. Возврат в состояние 2	15
Работает вызывающая функция. Очистки стека от параметров.....	16
Выводы.....	17

Схема организации функций по соглашениям о вызовах ABI

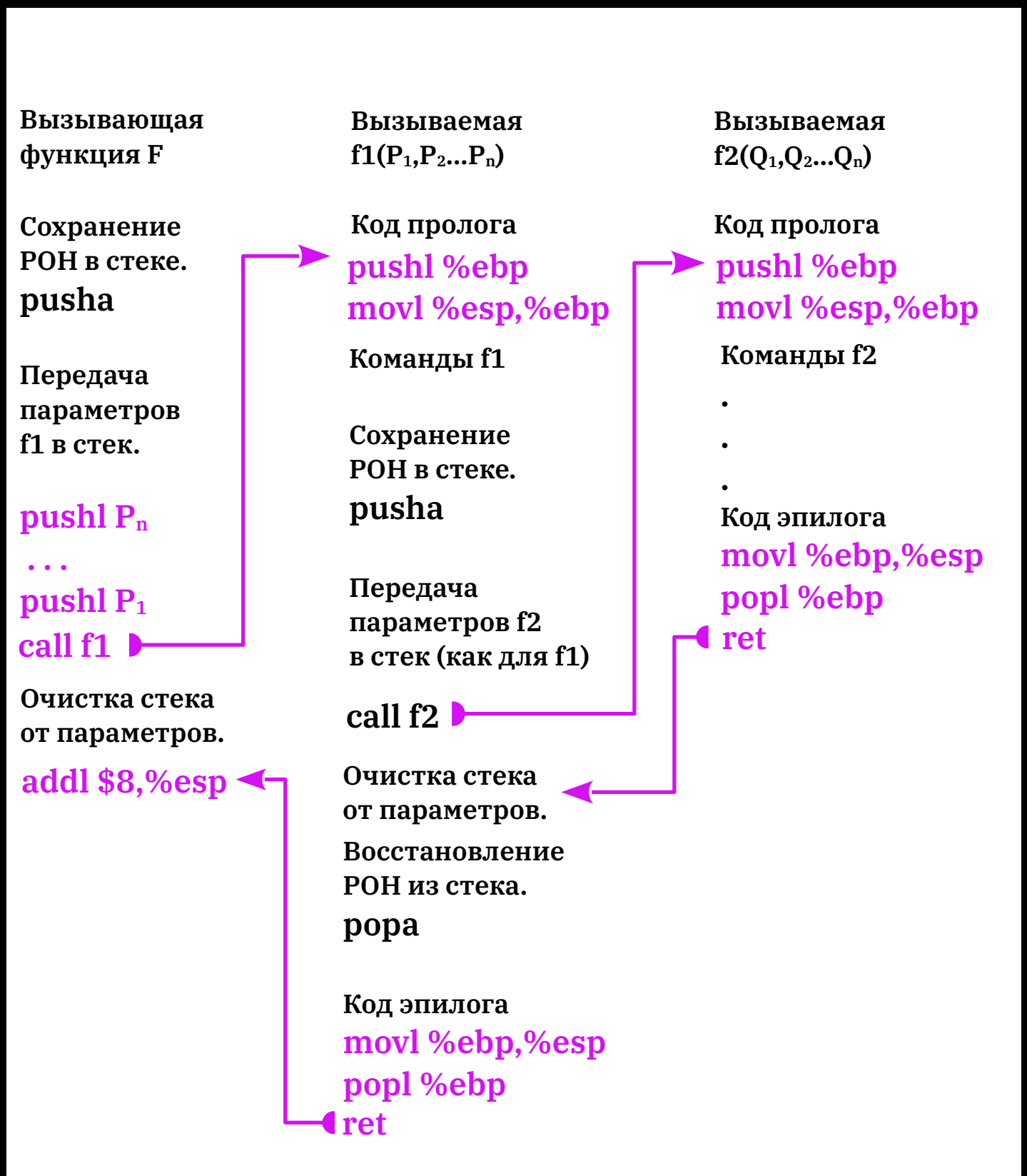


Рис. 1. Схема организации функций по соглашениям о вызовах ABI

При вызове функции согласно общей схеме на Рис 1. в архитектурном стеке формируется принадлежащим ей структура данных, называемая кадром стека времени выполнения (a frame on the run-time stack).

Опишем последовательность изменения состояний стека при формировании этого кадра.

До вызова функции стек находится в исходном состоянии:

Исходное состояние стека

Исходное состояние	В стеке	Операнд для доступа
	Вершина стека	1 0(%esp)

Работает вызывающая функция

Выполнение команды pusha. Состояние 0

Выполняется команда pusha сохранения в стеке значений регистров общего назначения %eax, %ecx, %edx, %ebx, %esp, %ebp, %esi, %edi.

NB. В стек записывается значение %esp, которое он имел ДО выполнения команды pusha. Стек переходит в состояние 0:

Состояние 0	В стеке	Операнд для доступа
	Вершина стека	%edi 0(%esp)
	%esi	4(%esp)
	%ebp	8(%esp)
	%esp	12(%esp)
	%ebx	16(%esp)
	%edx	20(%esp)
	%ecx	24(%esp)
	%eax	28(%esp)

Старший адрес

1

32(%esp)

Итак, если используется команда `pusha` то значения регистров общего назначения входят в состав данных, создаваемых в кадре стека.

Помещение параметров в стек. Состояние 1

Выполняются команды записи параметров в стек в порядке обратном определенному в функции $f(P_1, P_2 \dots P_n)$ языка C:

`pushl Pn`

`pushl Pn-1`

`...`

`pushl P1`

После их выполнения стек переходит в состояние 1:

Состояние 1	В стеке	Операнд для доступа
Вершина стека	P1	0(%esp)
	P2	4(%esp)
	...	...
	PN	4*(N-1)(%esp)
	Восемь PОН функции F	Не показаны.
Старший адрес	1	Не показан.

Здесь и далее не приводятся формы операндов элемента стека в режиме регистровой адресации для PОН.

NBNB. Если параметр не 4-байтовое слово, а некоторая структура данных, например массив, то в стек нужно помещать 4-байтовый адрес этой структуры, например первого элемента массива.

Итак, значения параметров входят в состав данных, создаваемых в кадре стека.

Вызов функции командой `call`. Состояние 2

Выполняется команда вызова функции:

<адрес команды `call`> `call f1`

<адрес следующей команды> следующая команда

Команда `call` производит два действия:

1. Запись в стек значения <адрес следующей команды>, т. н. адреса возврата, который будем обозначать `RetAddr`. Это эквивалентно выполнению команды `pushl RetAddr`
2. Передача управления на точку входа в `f1` — `<%eip> = <адрес f1>`

Стек переходит в состояние 2:

Состояние 2	В стеке	Операнд для доступа
Вершина стека	<code>RetAddr</code> функции <code>F</code>	<code>0(%esp)</code>
	<code>P1</code>	<code>4(%esp)</code>
	<code>P2</code>	<code>8(%esp)</code>
	...	...
	<code>PN</code>	<code>4+4*(N-1)(%esp)</code>
	Восемь <code>POH</code> функции <code>F</code>	Не показаны.
Старший адрес	1	Не показан.

Итак, значение `RetAddr` входит в состав данных, создаваемых в кадре стека.

Работает вызванная функция `f1`

Выполнение первой обязательной команды пролога `pushl %ebp`. Состояние 3

Команда `pushl %ebp` помещает в стек значение регистра `%ebp` которое было в вызывающей функции. Будем обозначать его `caller%ebp`.

Это делается, чтобы обеспечить в коде вызываемой функции `f1` многократный адресный доступа к данным, созданным в стеке при вызове функции как для выполнения операция над параметрами, так и для других целей.

По соглашениям **ABI** этот доступ организован с помощью режима адресации через базовый регистр `%ebp` для всех вложенных вызовов функций, в то числе и самой вызывавшей функции.

Говорят, что значение `%ebp` является базой кадра стека.

Поэтому значение `caller%ebp` любой вызывающей функции сохраняется в стеке вызываемой функции командой пролога `pushl %ebp` и при возврате из вызываемой функции в вызывающую восстанавливается соответствующей командой кода эпилога.

Таким образом, значение `caller%ebp` вызывающей функции также входит в состав данных, создаваемых в кадре стека при вызове `f1`.

После выполнения команды `pushl %ebp` стек переходит в состояние 3, в котором базой кадра стека пока является регистр `%esp`.

Состояние 3	В стеке	Операнд для доступа
Вершина стека	<code>caller%ebp</code> функции F	<code>0(%esp)</code>
	<code>RetAddr</code> функции F	<code>4(%esp)</code>
	P1	<code>8(%esp)</code>
	P2	<code>12(%esp)</code>
	...	...
	PN	<code>4+4+4*(N-1)(%esp)</code>

Старший адрес	Восемь PОН функции F	Не показаны.
	1	Не показан.

NB. Значение `%esp` сейчас нельзя изменять (например для вызова функции из `f1`), в т.ч. командами `pushl` и `popl` т.к. потеряем значение базы кадра стека.

Выполнение второй обязательной команды пролога `movl %esp,%ebp`. Состояние 4

Команда `movl %esp,%ebp` присваивает регистру `%ebp` значение базы кадра стека функции `f1`. Ее надо выполнить сразу после первой команды пролога и после этого регистр `%ebp` можно использовать в командах тела функции как базу для адресного доступа к данным кадра стека функции `f1` и к локальным переменным в стеке же (см. чуть ниже), если программист их создаст.

NBNB. Т.о. значение регистра `%ebp` в теле вызываемой функции изменять нельзя!

После выполнения команды `movl %esp, %ebp` стек переходит в состояние 4:

Состояние 4 Вершина стека	В стеке	Операнд для доступа
	caller%ebp функции F	0(%esp), 0(%ebp)
	RetAddr функции F	4(%esp), 4(%ebp)
	P1	8(%esp), 8(%ebp)
	P2	12(%esp), 12(%ebp)

Старший адрес	...	...
	PN	$4+4+4^*(N-1)(\%esp)$ $4+4+4^*(N-1)(\%ebp)$
	Восемь PОН функции F	Не показаны.
	1	Не показан.

NBNB. Сразу после выполнения второй команды пролога оба регистра — `%esp` и `%ebp` можно использовать как базовые для адресного доступа к элементам кадра стека, прежде всего, к параметрам.

Однако регистр `%esp` нужен нам для других целей, прежде всего для возможного вызова других функций из текущей функции — `f1`, для создания переменных, локальных в `f1`, возможно для чего-то еще.

Главное — теперь МОЖНО использовать `%esp` для любых целей т. к. в качестве базового регистра для адресного доступа к параметрам в кадре стека по соглашениям ABI используется регистр `%ebp`.

Итак, после выполнения кода пролога регистр `%esp` можно использовать для любых целей. Обычно это:

- резервирование в стеке памяти для переменных, локальных в `f1`;
- вызов другой функции с созданием ее кадра стека (на Рис.1 — `f2`) из кода текущей функции `f1`.

Резервирование памяти для локальных переменных в кадре стека. Состояние 5

Память для локальных переменных `f1` в стеке резервируется соответствующим уменьшением значения регистра

%esp.

Отметим, что после выполнения кода эпилога значения этих переменных становятся недоступными.

Память для двух локальные 4-х байтовых переменных резервируется выполнением после кода пролога команды:

```
subl $8, %esp
```

после чего стек переходит в состояние 5:

Состояние 5		В стеке	Операнд для доступа
		Начало кадра стека (stack frame) f1	
Вершина стека	Локальная.2 функции f1	- 8(%ebp)	
	Локальная.1 функции f1	- 4(%ebp)	
	caller%ebp функции F	0(%ebp)	
	RetAddr функции F	4(%ebp)	
	P1	8(%ebp)	
	P2	12(%ebp)	
	...	...	
	PN	4+4+4*(N-1)(%ebp)	
	Восемь PОН функции F	Не показаны.	
		Конец кадра стека (stack frame) f1	
Старший адрес	1	Не показан.	

NBNB. В этом состоянии ОБЕСПЕЧЕН многократный адресный доступ к параметрам и локальным переменным f1 в стеке через базовый регистр %ebp как к

элементам одномерного массива. При этом регистр `%esp` уже не нужен для работы `f1` и может, как уже сказано, использоваться для любых целей.

Примеры адресного доступа к параметрам и локальным переменным

Запись значения первого параметра в регистр `%edx`:

```
movl 8(%ebp), %edx
```

Вычисление суммы значений второго параметра и регистра `%edx`:

```
addl 12(%ebp), %edx
```

Запись содержимое регистра `%eax` в локальную переменную 1:

```
movl %eax, -4(%ebp)
```

Вычисление суммы значений локальной переменной 2 и регистра `%edx`:

```
addl -8(%ebp), %edx
```

NB. Память в стеке для локальных переменных **БУДЕТ НЕДОСТУПНА** после выхода из функции.

NB. Локальные переменные не будут удалены при создании кадра стека при вложенном вызове другой функции из `f1`, их значения могут быть использованы в `f1` после возврата из этой другой функции.

Вложенный в функцию `f1` вызов функции `f2`. Состояние 6

Отметим, что при вложенном вызове вызываемая функция будет формировать свой кадр стека, который будет расположен в стеке «выше» (в направлении младших адресов ОП) кадра стека вызывающей функции.

Таким образом В таблице ниже показано состояние 6 стека после вызова функции `f2` и выполнения команд ее пролога. Кадры стека `f1` и `f2` выделены рамками разных цветов.

**Состояние 6 стека
после выполнения
кода пролога f2**

Вершина стека

В стеке	Операнд для доступа
Начало кадра стека (stack frame) f2	
Локальная . 2 функции f2	- 8 (%ebp)
Локальная . 1 функции f2	- 4 (%ebp)
caller%ebp функции f1	0 (%ebp)
RetAddr функции f1	4 (%ebp)
Q1	8 (%ebp)
Q2	12 (%ebp)
. . .	. . .
Qm	$4 + 4 + 4 * (N - 1) (\%ebp)$
Восемь P0H функции f1	Не показаны.
Конец кадра стека (stack frame) f2	

Старший адрес

Начало кадра стека (stack frame) f1	
Локальная . 2 функции f1	- 8 (%ebp)
Локальная . 1 функции f1	- 4 (%ebp)
caller%ebp функции F	0 (%ebp)
RetAddr функции F	4 (%ebp)
P1	8 (%ebp)
P2	12 (%ebp)
...	...
PN	$4 + 4 + 4 * (N - 1) (\%ebp)$
Восемь P0N функции F	Не показаны.
Конец кадра стека (stack frame) f1	
1	Не показан.

Возврат из вызванной функции в вызывающую

Проницательный студент уже подозревает, что код эпилога выполняет действия, обратные коду пролога, убирая из стека кадр вызванной функции, вместе с командами очистки стека от параметров в вызывающей функции.

Рассмотрим процесс возврата из функции f1 в предположении, что возврат из функции f2 уже выполнен и стек находится в состоянии 5.

Состояние 5

В стеке	Операнд для доступа
Начало кадра стека (stack frame) f1	

Вершина стека

Локальная.2
функции f1

- 8(%ebp)

Локальная.1
функции f1

- 4(%ebp)

caller%ebp
функции F

0(%ebp)

RetAddr
функции F

4(%ebp)

P1

8(%ebp)

P2

12(%ebp)

...

...

PN

$4 + 4 + 4 * (N - 1)(\%ebp)$

Восемь PОН
функции F

Не показаны.

Конец кадра стека (stack frame) f1

Старший адрес

1

Не показан.

Выполнение первой обязательной команды эпилога `movl %ebp, %esp`. Возврат в состояние 4

Регистр `%esp` получает значение, которое он имел до выполнения команды выделения памяти в кадре стека локальным переменным `subl $8, %esp` значения которых теперь становятся недоступными, а стек возвращается из состояния 5 в состояние 4, в котором на вершине стека находится значение `caller%ebp`:

Состояние 4

В стеке

Операнд для доступа

Вершина стека	caller%ebp функции F	0(%esp), 0(%ebp)
	RetAddr функции F	4(%esp), 4(%ebp)
	P1	8(%esp), 8(%ebp)
	P2	12(%esp), 12(%ebp)
	...	...
	PN	4+4+4*(N-1)(%esp) 4+4+4*(N-1)(%ebp)
Старший адрес	Восемь PОН функции F	Не показаны.
	1	Не показан.

Выполнение второй обязательной команды эпилога `popl %ebp`. Возврат в состояние 2

Команда `popl %ebp` извлекает из стека значение `caller%ebp` восстанавливая тем самым значение регистра `%ebp`, которое он имел в вызывавшей функции, т. е. восстанавливая базу кадра стека этой функции. Стек переходит из состояния 4 в состояние 2, в котором на его вершине находится значение адреса возврата `RetAddr`:

Состояние 2	В стеке	Операнд для доступа
	RetAddr функции F	0(%esp)
Вершина стека	P1	4(%esp)
	P2	8(%esp)
	...	...
	PN	4+4*(N-1)(%esp)
	Восемь	Не показаны.

Старший адрес

РОН функции F	
1	Не показан.

Именно это и нужно для выполнения команды `ret`, которая должна всегда следовать за второй командой эпилога.

Команда `ret` извлекает из стека значение адреса возврата в вызывающую функцию `RetAddr` и, помещая его в регистр `%eip`, возвращает тем самым управление вызывающей функции. При этом стек переходит в состояние 1 и содержит только параметры и РОН.

Состояние 1
Вершина стека

В стеке	Операнд для доступа
P1	$0(\%esp)$
P2	$4(\%esp)$
...	...
PN	$4 * (N - 1)(\%esp)$
Восемь РОН функции F	Не показаны.
Старший адрес	1
	Не показан.

Работает вызывающая функция. Очистки стека от параметров

Выполняется команда очистки стека от параметров:

`addl $L,%esp`

(`L` — константа, равная суммарной длине в байтах всех параметров — для 2 параметров это 8). Стек переходит в состояние 0 в котором содержит только значения 8 регистров общего назначения.

Состояние 0

В стеке	Операнд для
---------	-------------

Вершина стека

	доступа
%edi	0(%esp)
%esi	4(%esp)
%ebp	8(%esp)
%esp	12(%esp)
%ebx	16(%esp)
%edx	20(%esp)
%ecx	24(%esp)
%eax	28(%esp)
Старший адрес	1
	32(%esp)

Выполняется команда **popa** восстановления значений **РОН** бывших в вызывающей функции в стек переходит в исходное состояние:

Исходное состояние

Вершина стека

В стеке	Операнд для доступа
1	0(%esp)

Выводы

- Перед помещением параметров функции в стек целесообразно поместить значения 8 **РОН** не входящей в соглашение **АВІ** командой **pusha**.
- По соглашениям о связях **АВІ** команды передачи параметров в стек, команда **call**, команды пролога и, по желанию программиста, команды резервирования памяти для локальных переменных, создают в стеке для каждой вызываемой функции отдельную, принадлежащую ей структуру данных — **кадр стека времени выполнения** (a frame on the run-time stack).
- Команда **call** производит два действия:

1. Запись в стек значения `<адрес след. команды>`, т. н. адреса возврата, который будем обозначать `RetAddr`. Это эквивалентно выполнению команды `pushl RetAddr`

2. Передача управления на точку входа в вызываемую функцию `f1` — `<%eip> = <адрес f1>`

- По соглашениям **ABI** адресный доступ к данным кадра стека организован с помощью режима адресации через базовый регистр `%ebp` для всех вложенных вызовов функций, в том числе и самой вызывавшей функции. Говорят, что значение `%ebp` является базой кадра стека.

- Первая команда пролога `pushl %ebp` помещает в стек значение регистра `%ebp` - базы кадра стека вызывающей функции.

- Вторая команда пролога `movl %esp,%ebp` присваивает регистру `%ebp` значение базы кадра стека вызываемой функции. Ее надо выполнить сразу после первой команды пролога и после этого регистр `%ebp` можно использовать в командах тела функции как базу для адресного доступа к данным кадра стека вызываемой функции и к локальным переменным в стеке.

- Операнды команд для адресного доступа к параметрам функции и локальным переменным в кадре стека задаются в виде `смещение (%ebp)`, где смещение выбирается из таблицы состояния стека 5.

- После формирования кадра стека регистр `%esp` больше не нужен для работы вызванной функции, что обеспечивает возможность вложенных вызовов других функций с формированием их собственных кадров стека.

- В каждый момент времени существует столько кадров стека, сколько функций не выполнили код эпилога и команду `ret`.

- Код эпилога обеспечивает деактивацию кадра стека функции, восстановление базы кадра стека вызывающей функции и подготовку к возврату в нее командой `ret`.

- После выполнения кода эпилога значения локальных переменных в стеке становятся недоступными.

- Глубина вложенности вызовов функций ограничивается объемом памяти выделенным архитектурному стеку.

- Очистки стека от параметров выполняется в вызывающей функции командой `addl $L,%esp`, следующей за командой `call`, где `L` — константа, равная суммарной длине в байтах всех параметров.