Низкоуровневое программирование

Лекция 3

Процедуры и функции

Соглашения о вызовах

Функции в языке ассемблера

В языке ассемблера нет явного определения понятия функции. Де-факто, функция – это участок кода, который начинается с метки, и заканчивается инструкцией `ret`[1].

Для вызова функции используется инструкция `call`. Инструкция сохраняет в стек текущее значение регистра RIP и затем производит переход на указанный адрес.

Для возврата из функции используется инструкция `ret` (return). Она достает из вершины стека значение и сохраняет его в RIP (прыжок обратно в место вызова функции).

Кроме инструкций перехода `j*` и `ret`, нет никакого иного способа изменить значение RIP из программы!

```
a:
  add rcx,1
  ret


main:
  xor rcx, rcx
  call a
  …
  ret
```

[1] *технически, функция может также заканчиваться на JMP или системный вызов завершения процесса*

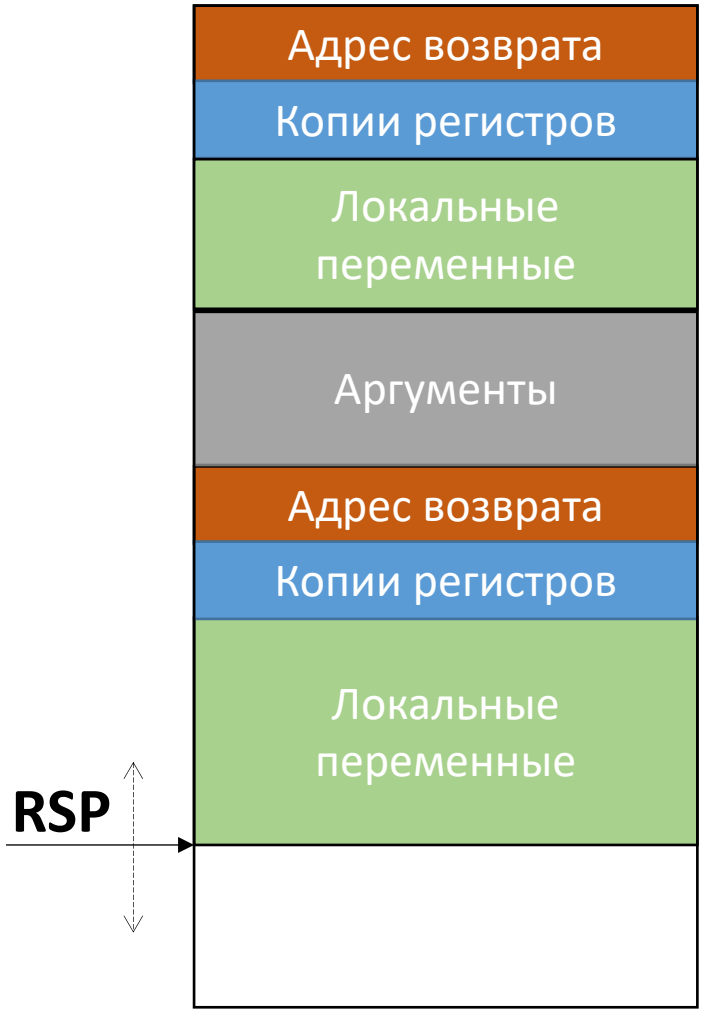
Стек вызовов

Стек вызовов (программный стек или просто стек) – область памяти, предназначенная для хранения *локальных переменных* и вспомогательных данных, необходимых для осуществления вызовов функций.

Указатель на вершину стека хранится в регистре **RSP**.

Стек растет вниз.
Вычитание из RSP увеличивает стек.
Прибавление к RSP уменьшает стек.

Прибавление/убавление значения к RSP не означает, что предыдущие значения будут стерты из памяти, но *память ниже RSP может быть изменена* ОС или средой выполнения (см. след. лекции).



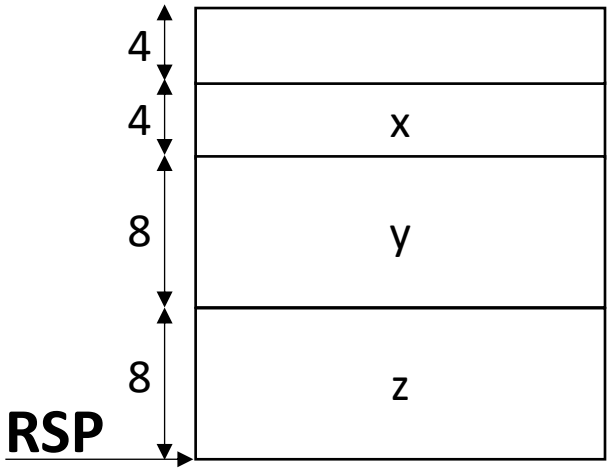
Локальные переменные

Локальные переменные хранятся на стеке. Выделение и освобождение памяти для переменных производится путем вычитания/прибавления к RSP соответствующего значения.

Примечание: обычно высчитывается общий размер локальных переменных в блоке кода, который вычитается из RSP при входе и прибавляется при выходе. Этот размер округляется в большую сторону до размера регистра (4 для x86, 8 для x86-64).

```
{
    int x = 0;
    long long y = 1;
    double z = 1.0;
    …
}
```

```
sub rsp, 24
mov dword[rsp+16], 0
mov qword[rsp+8], 1
fld1
fstp qword[rsp]
…
add rsp, 24
```



Кадры стека

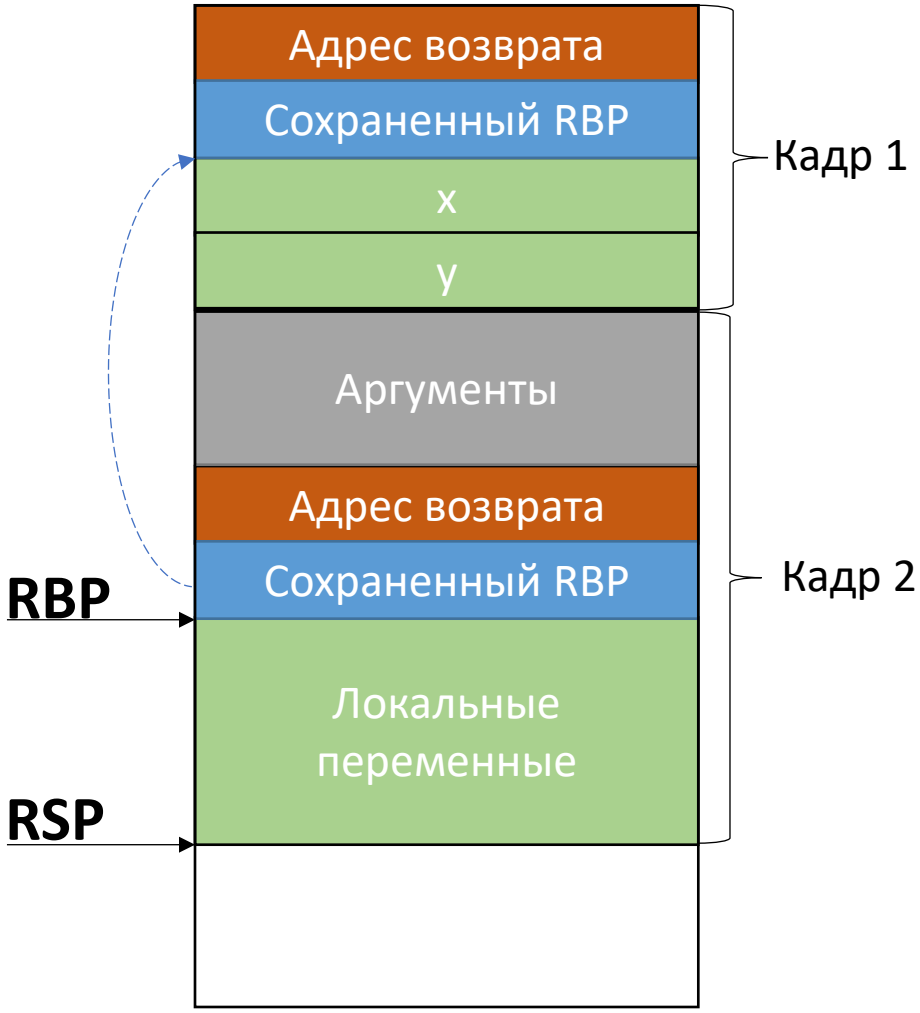
Кадром стека называется область, содержащая аргументы, локальные переменные, сохраненные значения регистров и адрес возврата выполняющейся функции.

Т.к. значение RSP может меняться в ходе работы функции, вводится понятие **указателя кадра стека**, играющего роль «точки отчета», относительно которой адресуются элементы кадра стека.

Для хранения указателя кадра стека используется регистр **RBP**(Base Pointer).

Обычно RBP указывает на свою сохраненную копию.Цепочка сохраненных указателей кадров организует кадры стека с связный список.*

**если указатель кадра стека используется*

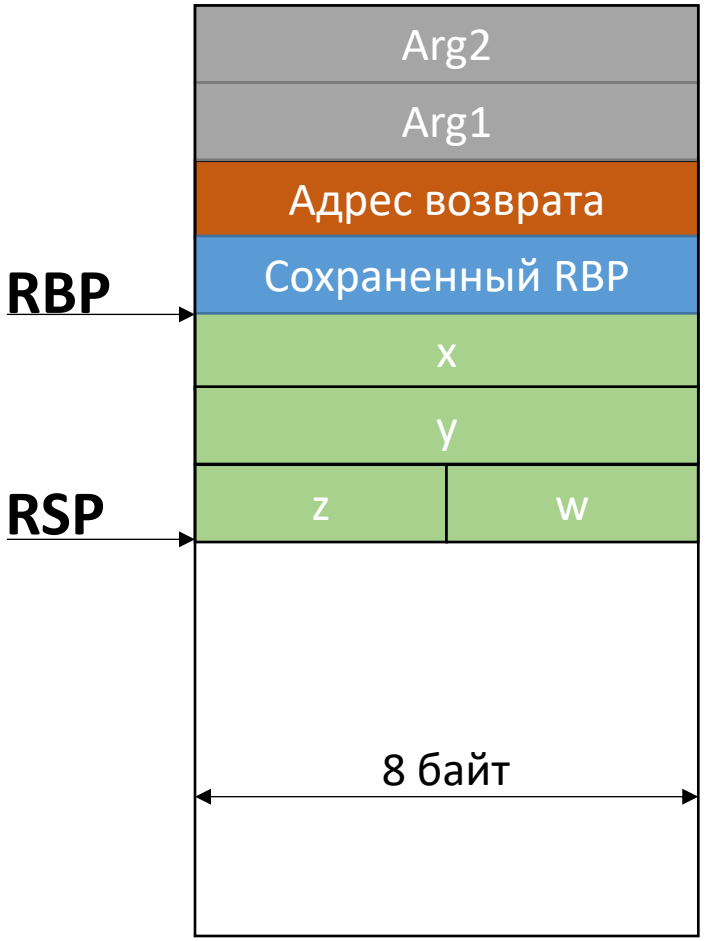


Адресация относительно RBP

Адресация локальных переменных и аргументов производится относительно регистра RBP/EBP.

Обычно:

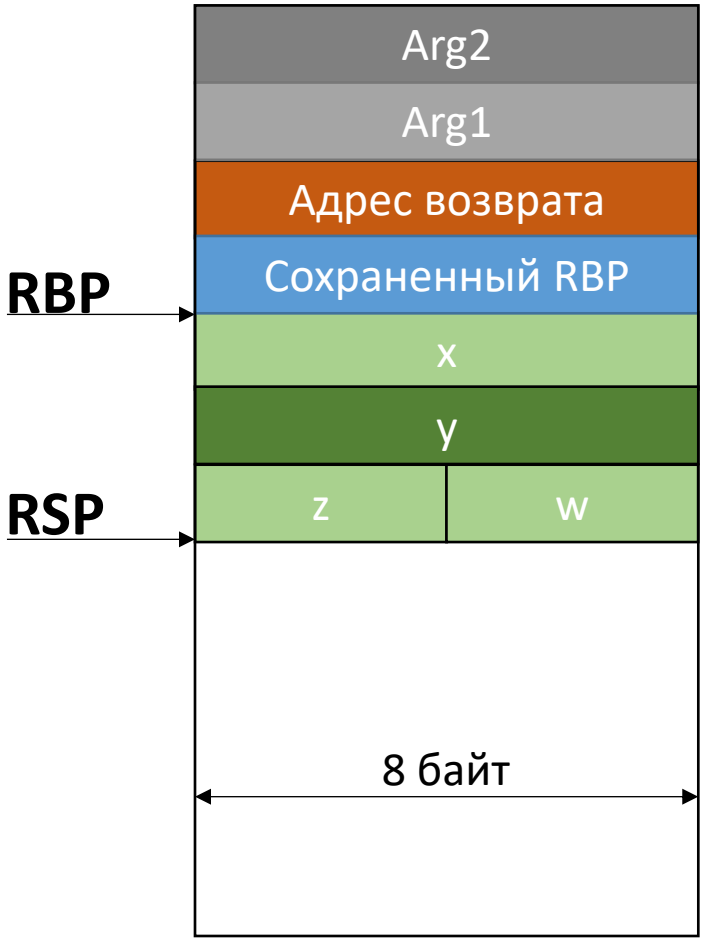
- локальные переменные располагаются внизу относительно RBP (**отрицательное смещение**).
- аргументы располагаются вверху относительно RBP (**положительное смещение**).



Адресация относительно RBP

```
y = 2*arg1 + arg2
```

```
mov rax, [rbp+16]
lea rax,[2*rax]
add rax, [rbp+24]
mov [rbp-16],rax
```



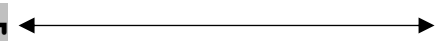
Пролог и эпилог

Кадр стека создается перед выполнением тела функции и уничтожается после его завершения.

Часть функции, в которой происходит инициализация кадра стека, называется **прологом**. В прологе происходит сохранение регистра RBP на стек, установка нового значения RBP, сохранение значений других регистров (если необходимо) и выделение места под локальные переменные.

Часть, в которой происходит уничтожение кадра стека, называется **эпилогом**. В эпилоге происходит освобождение места, занятого локальными переменными, восстановление значений сохраненных регистров (если таковые были) и выход из функции.

```
a:
  push rbp
  mov rbp, rsp
  push rbx ; Пролог
  sub rsp, 24 ; Локальные переменные
  …
  mov rbx, [rbp-8]
  mov rsp, rbp; Эпилог
  pop rbp ;
  ret
```



```
mov rbx, [rbp-8]
leave
ret
```

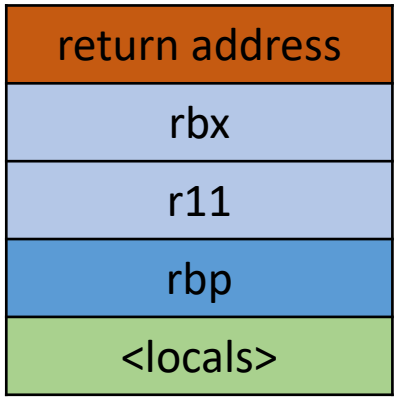
Примечание: `mov rbp, rsp` позволяет не выполнять `add rsp, 24`

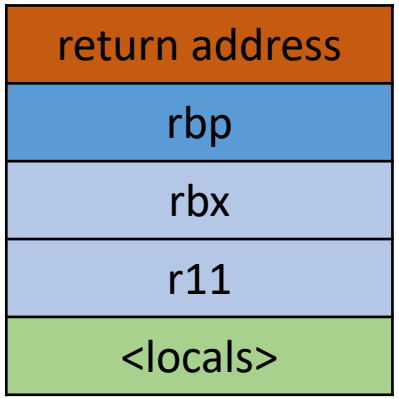
Примечания

- Если функция не работает со стеком, то устанавливать указатель кадра необязательно. В этом случае RBP можно рассматривать, как регистр общего назначения.
- Если кадр стека функции фиксирован (не происходит изменения RSP в середине функции), то устанавливать указатель кадра необязательно.
- Если функции нужно сохранить значения регистров (почему – см. далее), то сохраненные значения могут располагаться как выше, так и ниже копии RBP.
- *Сохранение регистров ниже копии RBP является более предпочтительным вариантом.* В частности, отладчики и анализаторы кода предпочитают, чтобы адрес возврата и сохраненный RBP находились рядом.

```
push rbx
push r11
push rbp
mov rbp, rsp
sub rsp, LSIZE
…
leave
pop r11
pop rbx
ret
```

```
push rbp
mov rbp, rsp
sub rsp, LSIZE+16
mov [rbp-16],r11
mov [rbp-8],rbp
…
mov rbx, [rbp-8]
mov r11, [rbp-16]
leave
ret
```





Передача параметров и возврат результата

Передача параметров в функцию может производиться разными способами.

Передача параметров и возврат результата могут производиться как в регистрах, так и на стеке.

Сам по себе язык ассемблера не ограничивает программиста в выборе способа передачи.

```
int a(int b, int c)
{
    return b+c;
}


int main()
{
   int x = a(12, 14)
}
```

```
a:
    mov eax, [esp+8]
    mov ebx, [esp+12]
    add eax, ebx
    ret

main:
    sub esp, 8
    mov [esp+4], 14 ; arg 2
    mov [esp], 12   ; arg 1
    call a
    add esp, 8
    ret
```

Соглашения о вызовах

Соглашение о вызовах (calling convention) – набор правил, регламентирующих вызов подпрограммы.

Соглашение о вызовах определяет правила передачи аргументов и возврата результата из подпрограммы, а также состояние регистров и стека до и после вызова.

Соглашения о вызовах являются частью более широкого набора правил, называемого **ABI** (Application Binary Interface), который в настоящее время устанавливается ОС. ABI регламентирует форматы исполняемых файлов, вызовы функций, правила взаимодействия с ОС и пр.

Т.к. в настоящее время на настольных ПК в основном используются 2 архитектуры набора команд (x86-32 и x86-64) и 2 семейства ОС (Windows и UNIX-подобные), возникает 4 возможных комбинации соглашений.

Подробнее

Регистры и вызов функции

Относительно состояния после вызова функции регистры делятся на изменяемые и неизменяемые.

Изменяемые регистры (volatile, caller-saved) могут свободно изменяться *вызываемой* функцией. Значения изменяемых регистров до и после функции может отличаться, поэтому *вызывающая* должна сохранить значения этих регистрах до вызова функции (если эти значения важны).

Неизменяемые регистры (nonvolatile, callee-saved) должны иметь одинаковое значение до и после вызова функции. Если вызываемая функция собирается изменить значение неизменяемого регистра, она должна сохранять его значение в прологе и восстановить его в эпилоге.

Арифметические флаги регистра FLAGS – всегда изменяемые.

Соглашение cdecl (x86)

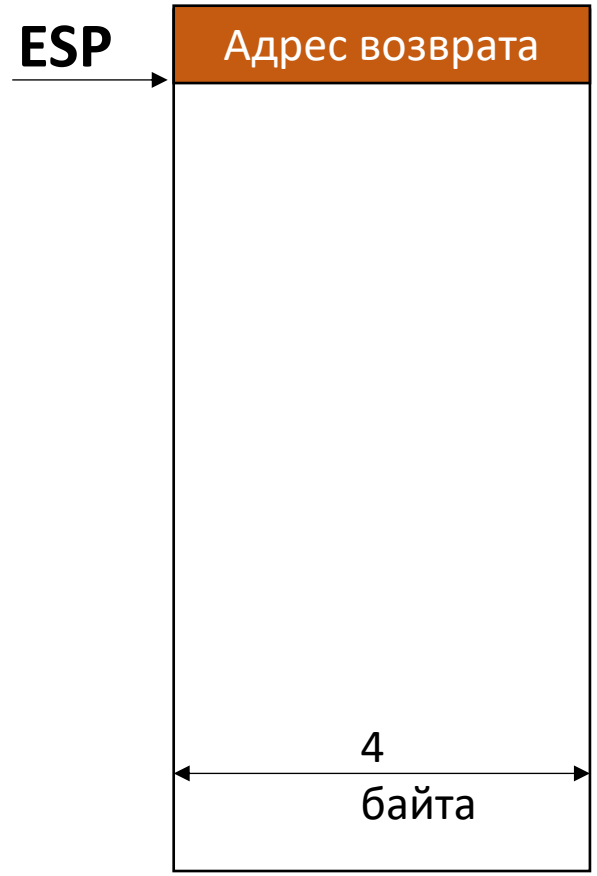
Соглашение **cdecl** является стандартным при сборке программ для x86-32.

- Вершина стека выравнивается по 4 байтам. Если размер аргумента или переменной не кратен 4, на стеке под него выделяется место, кратное 4 (лишние байты при этом не используются).
- Аргументы передаются **на стеке в обратном порядке**.
- После завершения вызова аргументы из стека убирает **вызывающая** функция.
- Изменяемые регистры: EAX, ECX, EDX, XMM0-7. Остальные регистры – неизменяемые.
- Стек x87 должен быть пуст в момент входа в функцию, в момент выхода должен содержать только возвращаемое значение, если оно есть.
- Целочисленный результат возвращается в регистре EAX или паре регистров EDX:EAX, вещественный – в ST0.
- *При сборке под Windows:* к имени функции добавляется префикс _.

Вызов cdecl-функции (Windows x86-32)

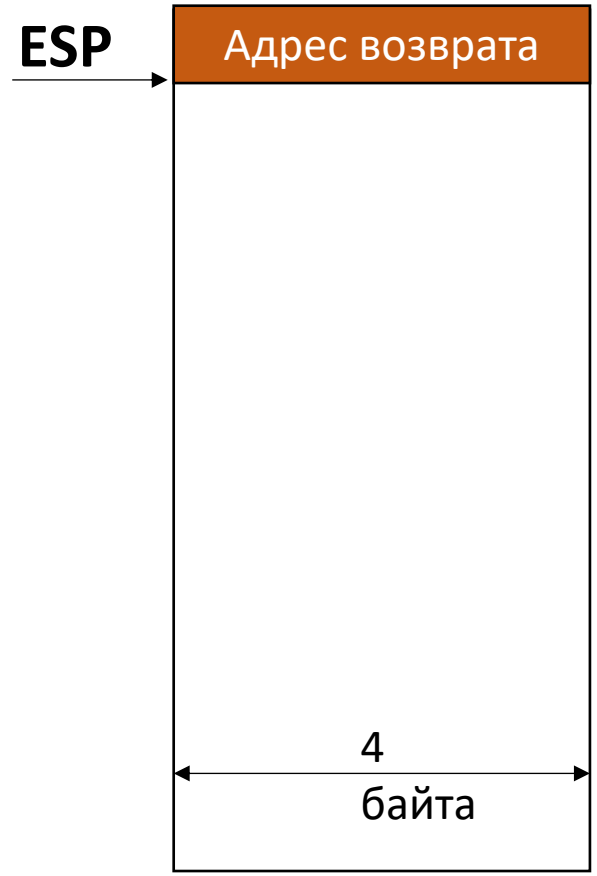
```
int fma(int a, int b, int c)
{
    return a*b+c;
}
```

```
void f()
{
   int x = 16;
   int z = fma(x, 4, 1);
   /*…*/
}
```



Вызов cdecl-функции (Windows x86-32)

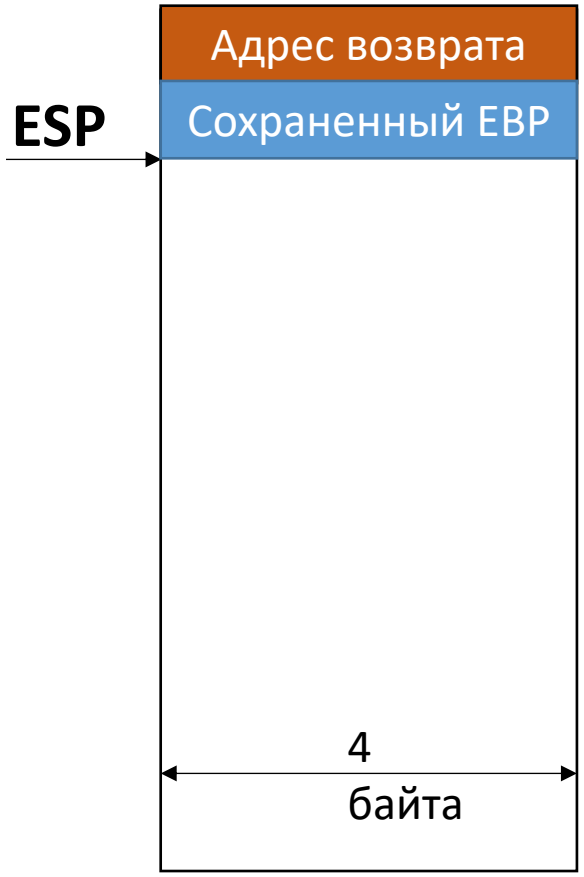
```
_f:
    push ebp          <-- EIP
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

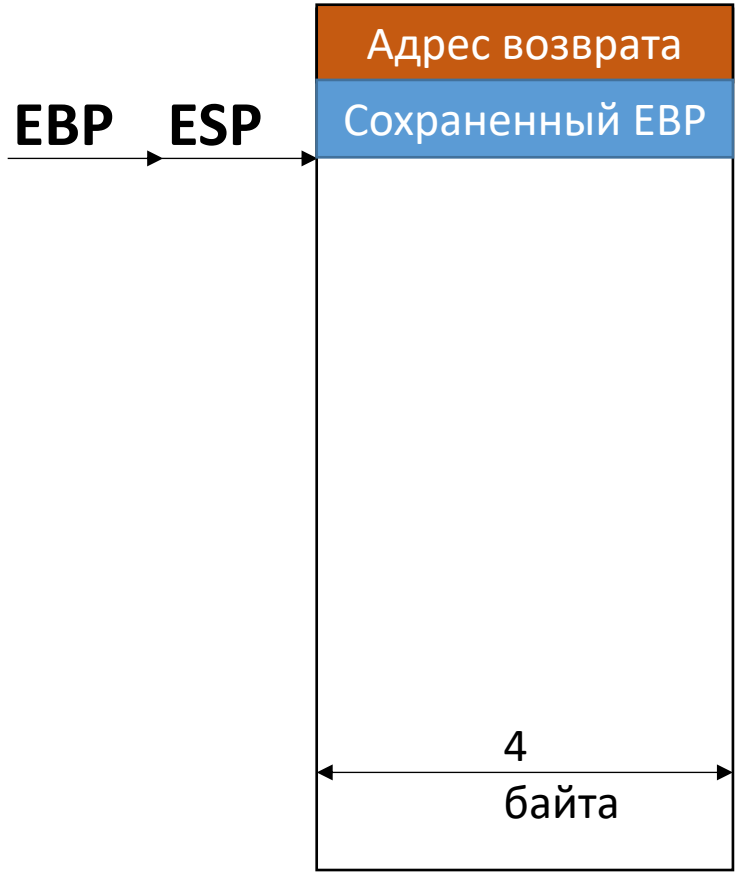
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ;…
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

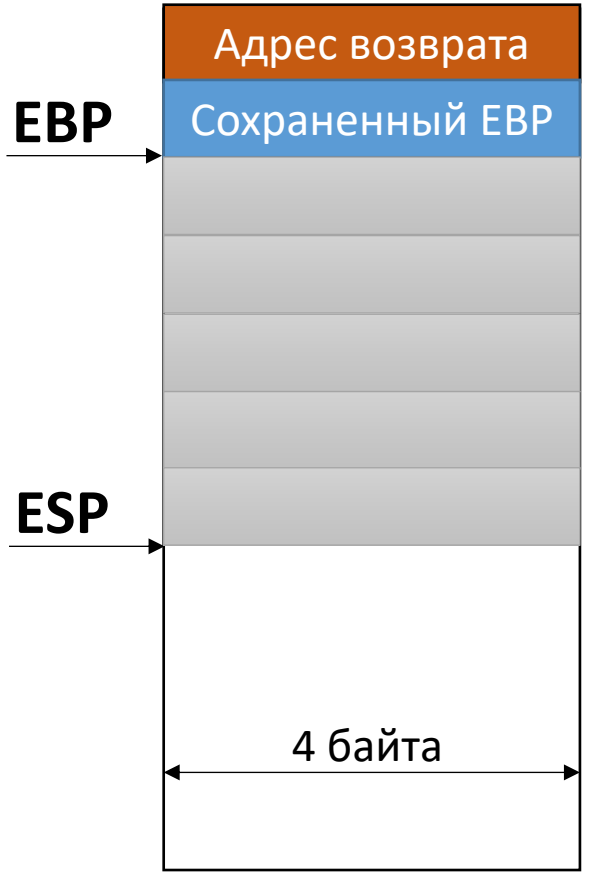
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

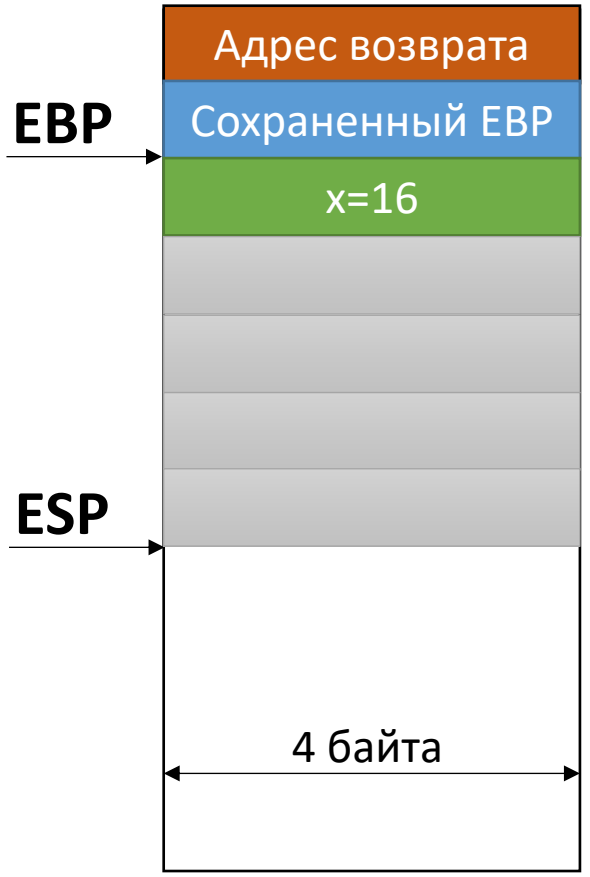
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

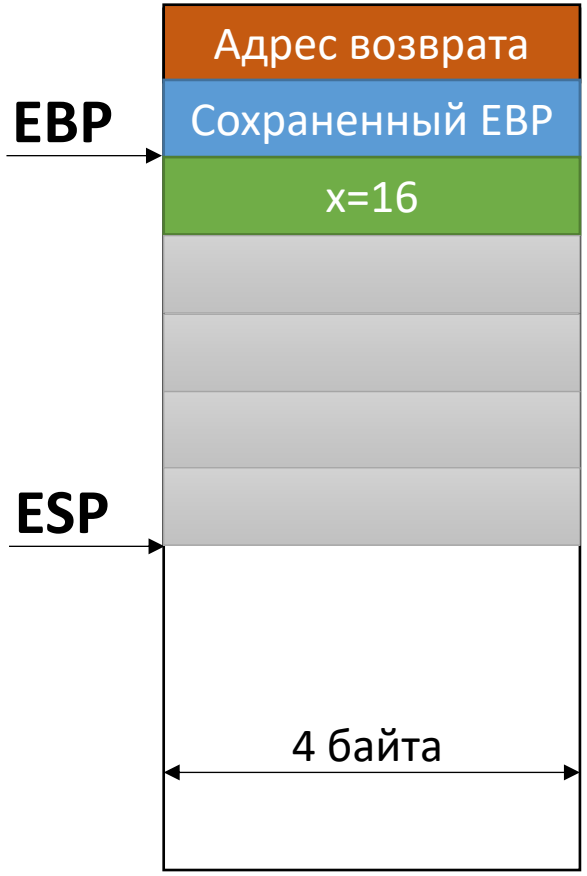
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

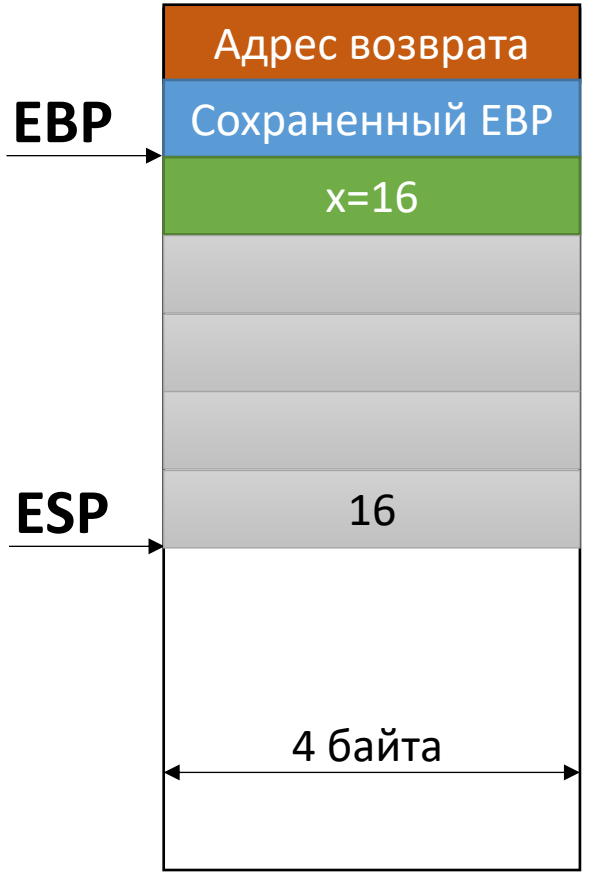
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

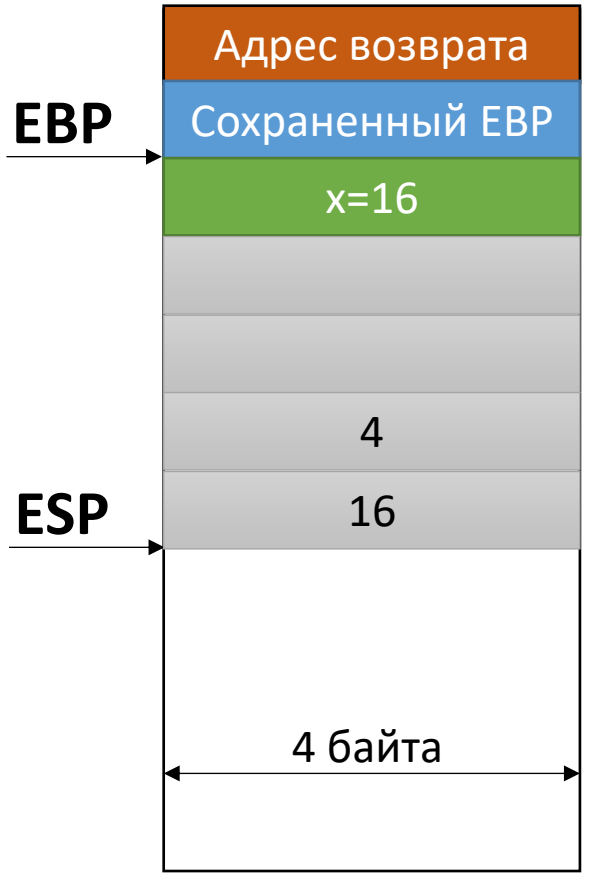
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

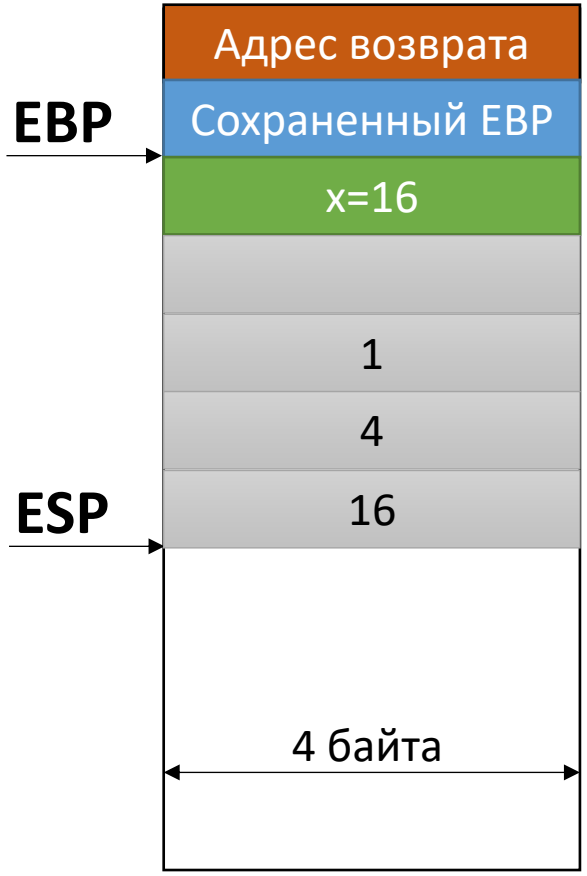
EIP



Вызов cdecl-функции (Windows x86-32)

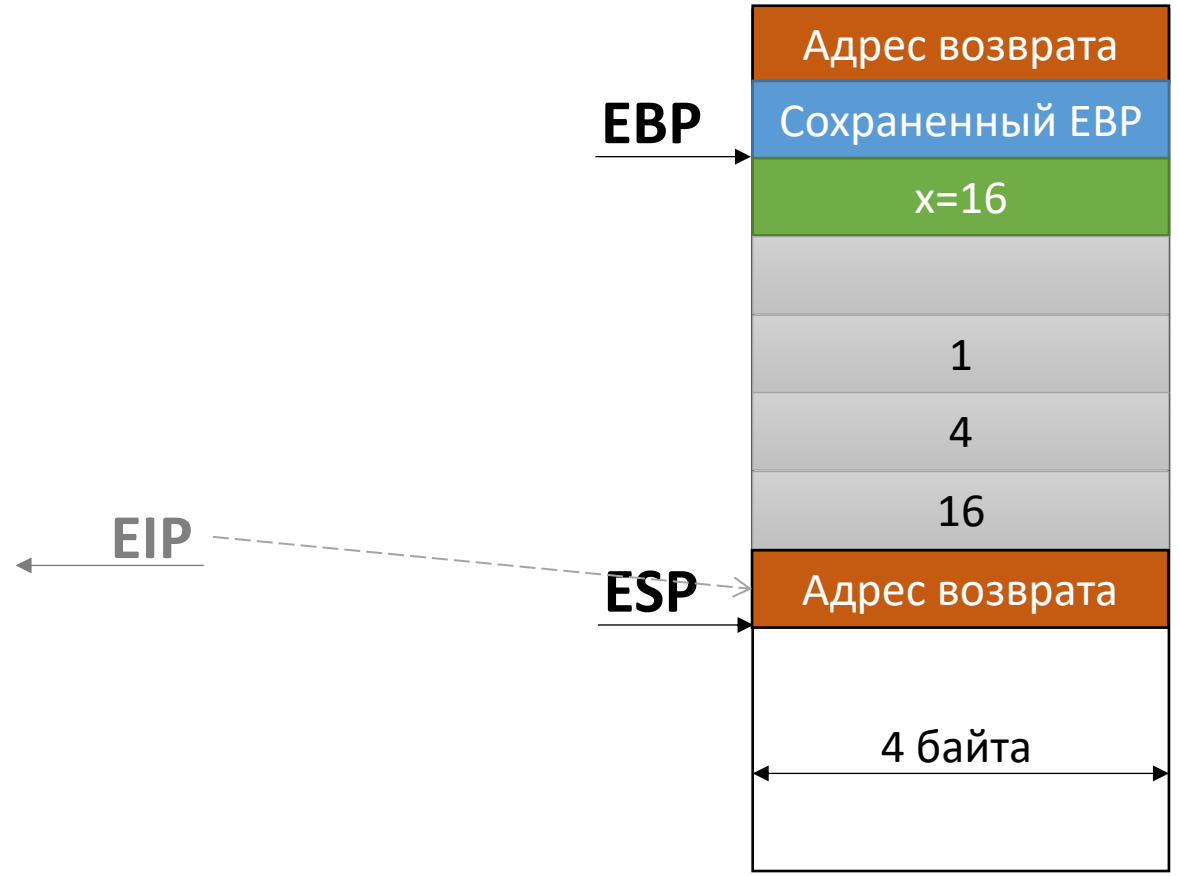
```
_f:
    push ebp
    mov ebp, esp
    sub esp, 20
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```



Вызов cdecl-функции (Windows x86-32)

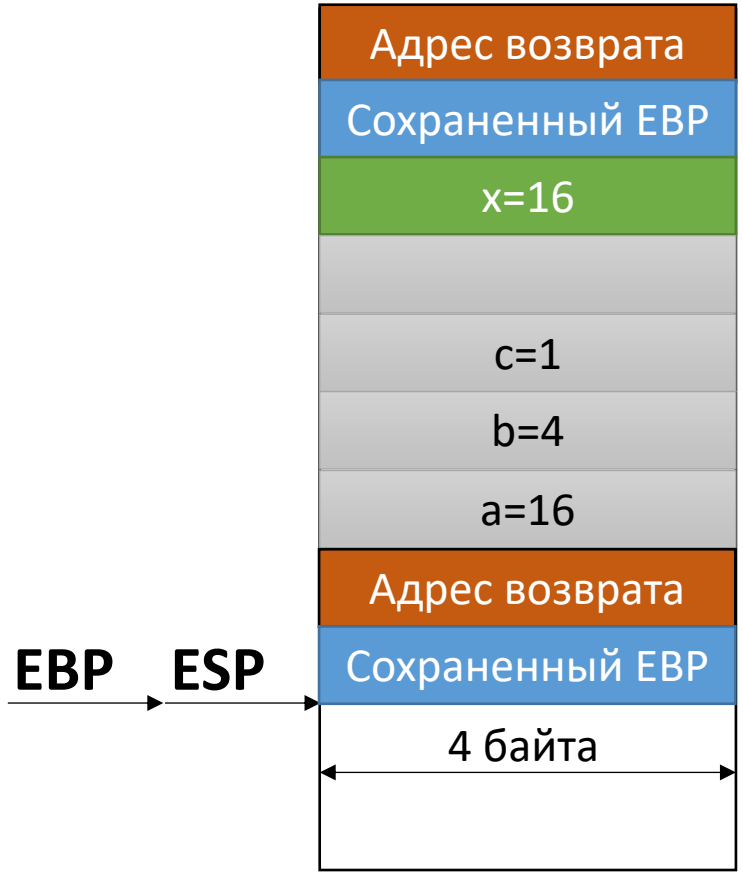
```
int fma(int a, int b, int c)


_fma:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret
```

EIP

Вызов cdecl-функции (Windows x86-32)

```
int fma(int a, int b, int c)


_fma:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret
```

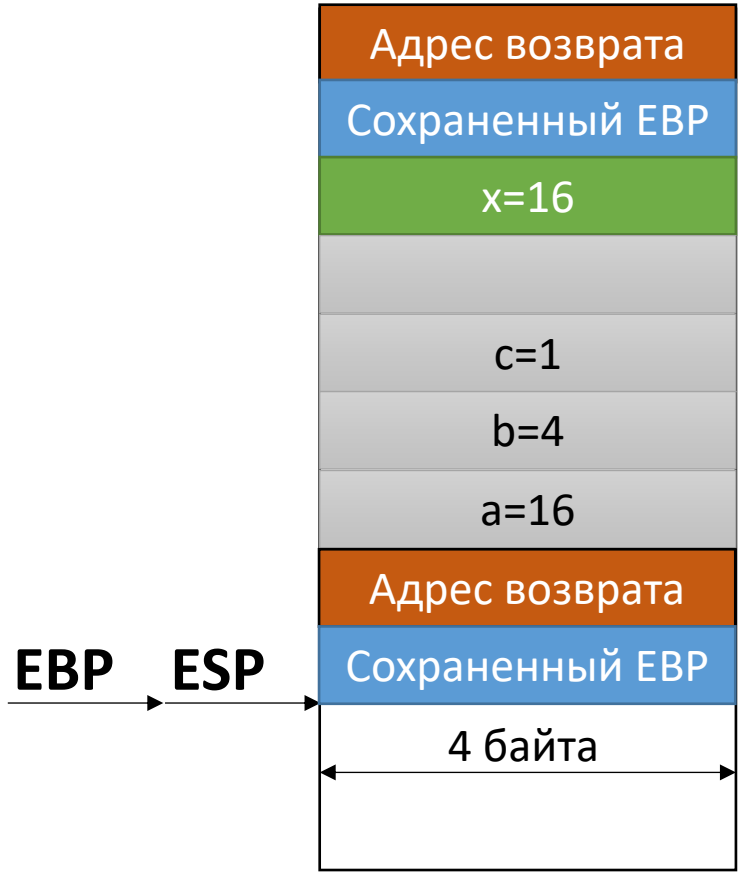
EIP



Вызов cdecl-функции (Windows x86-32)

```
int fma(int a, int b, int c)


_fma:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]     ;a
    imul eax, dword [ebp + 12]   ;a*b
    add eax, dword [ebp + 16]    ;a*b+c
    leave
    ret
```

EIP

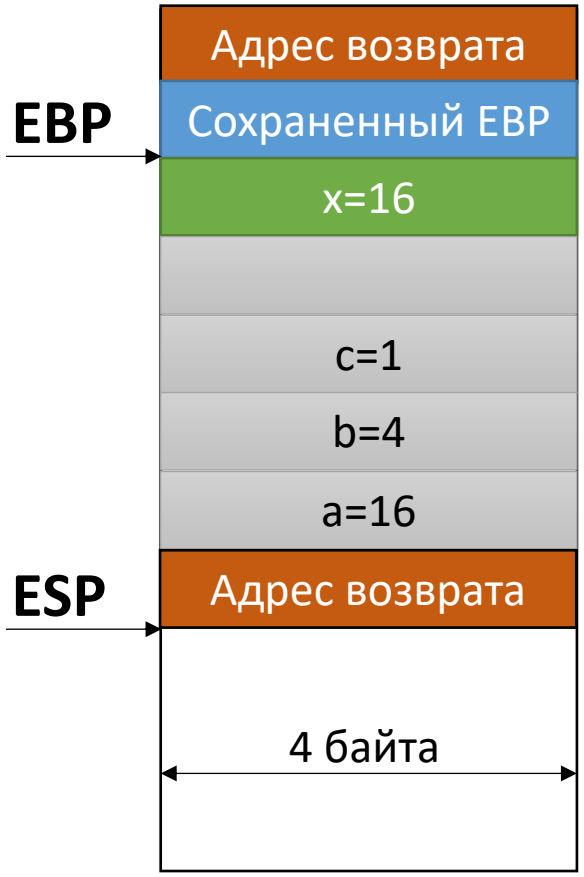


Вызов cdecl-функции (Windows x86-32)

```
int fma(int a, int b, int c)
```

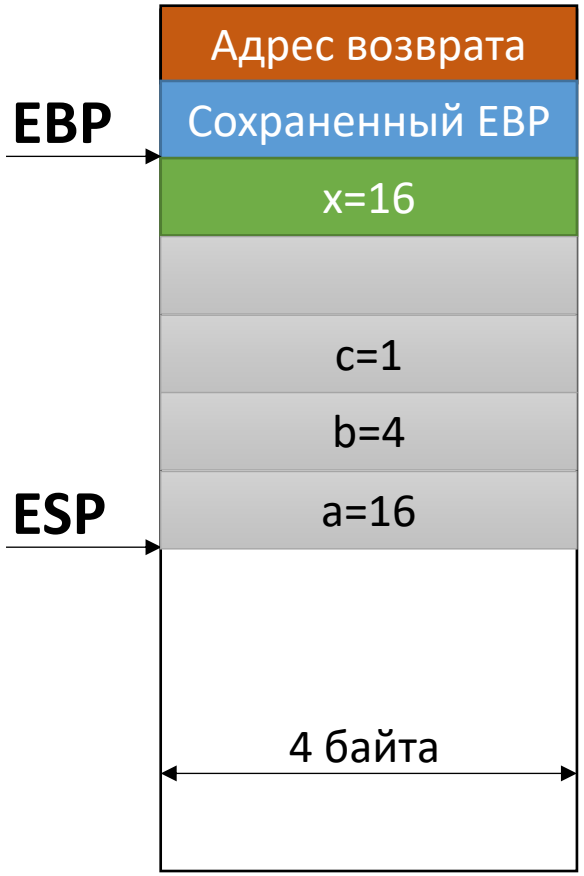
```
_fma:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret
```

EIP



Вызов cdecl-функции (Windows x86-32)

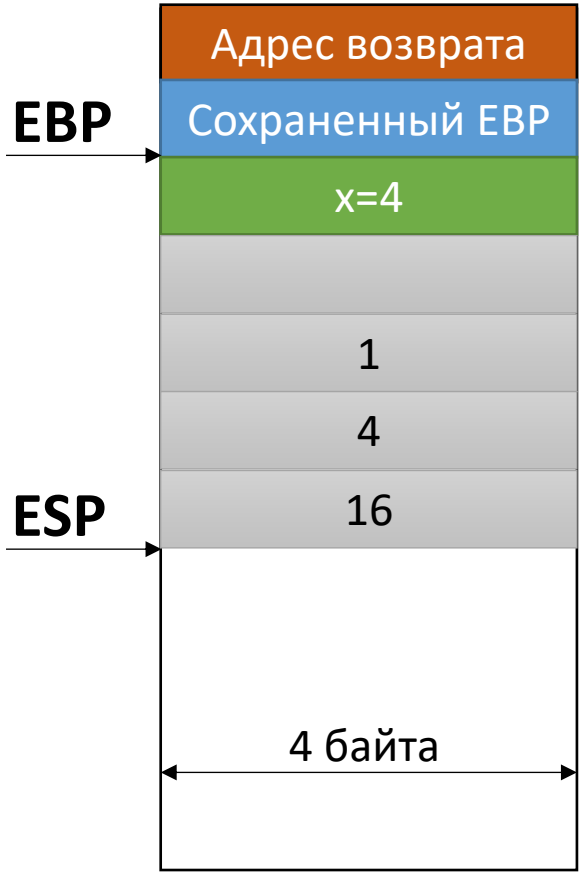
```
int fma(int a, int b, int c)


_fma:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret       ;результат в eax
```



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

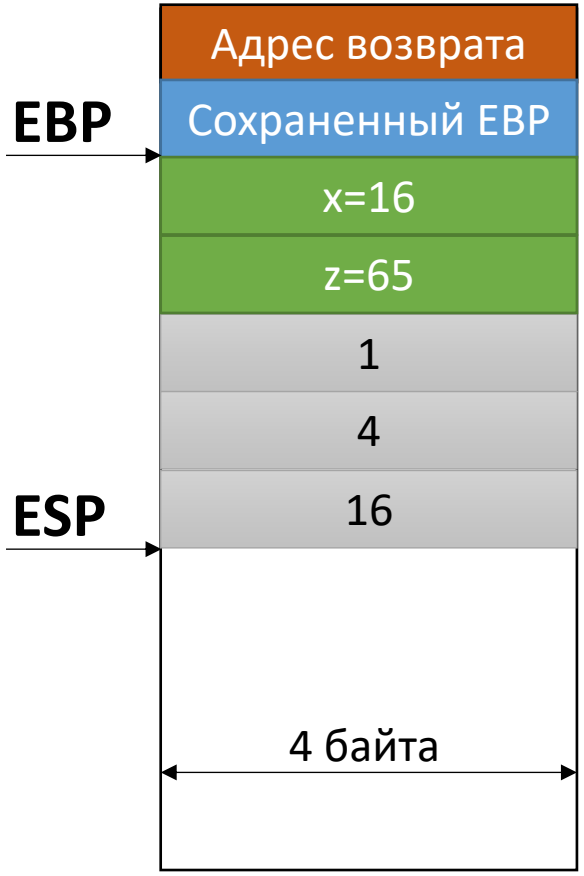
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

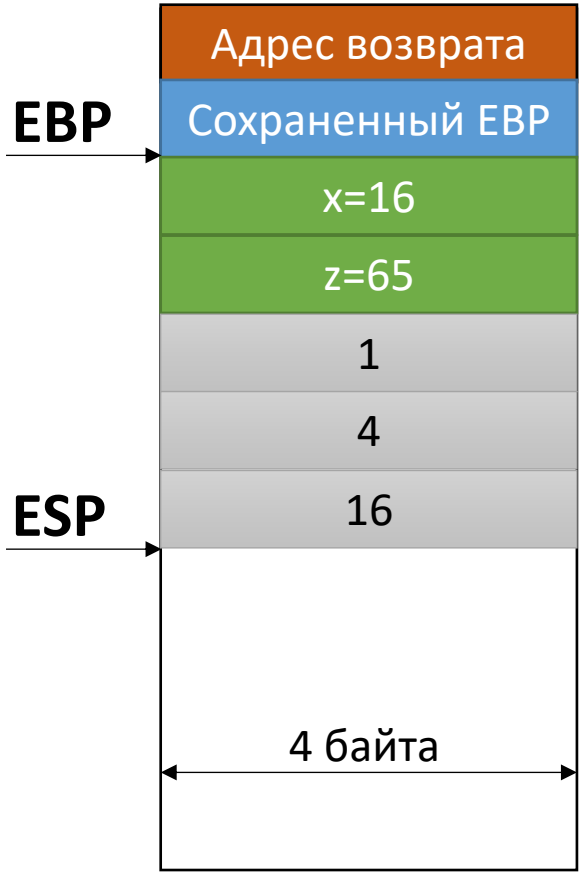
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

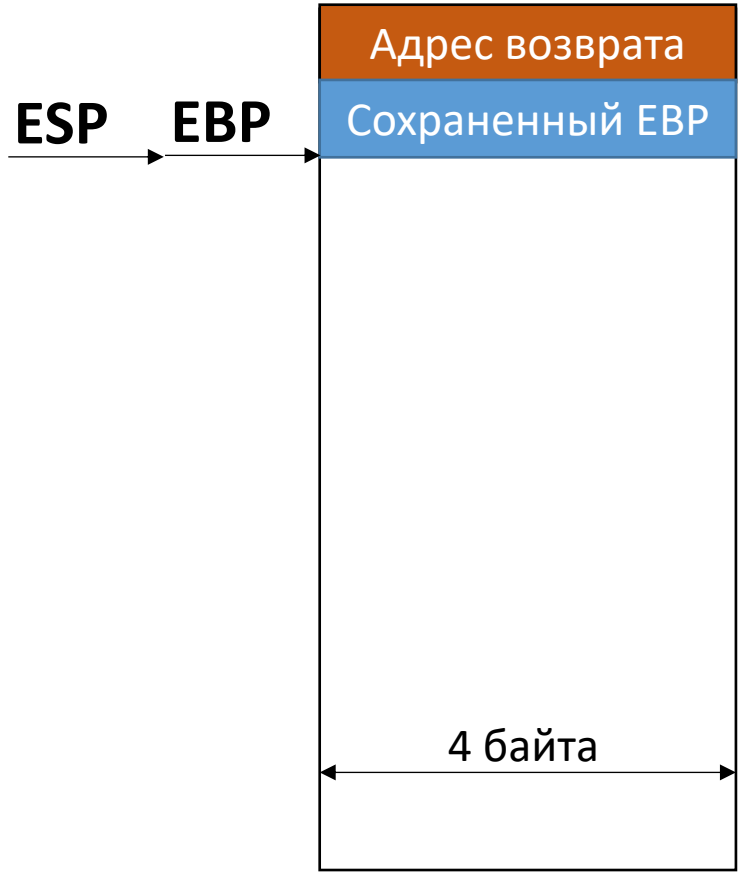
EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

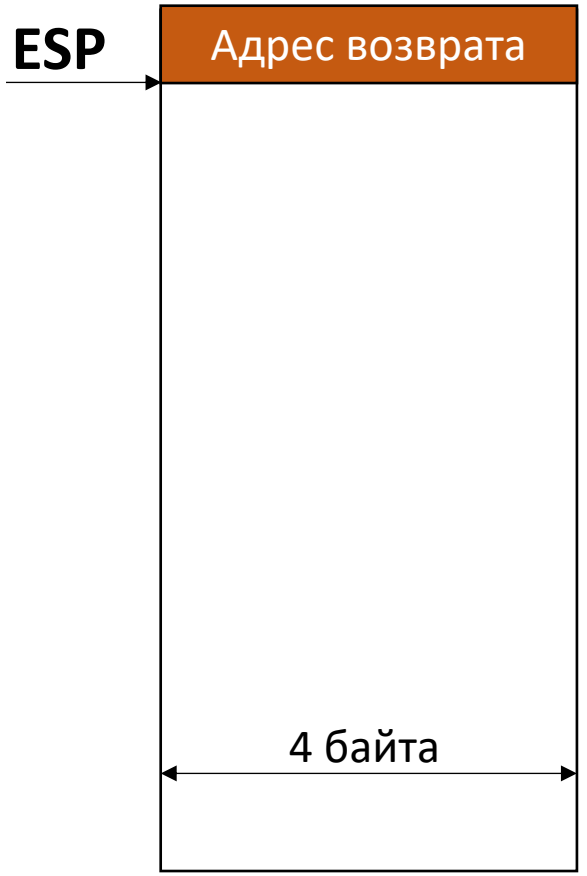
EIP



Вызов cdecl-функции (Windows x86-32)

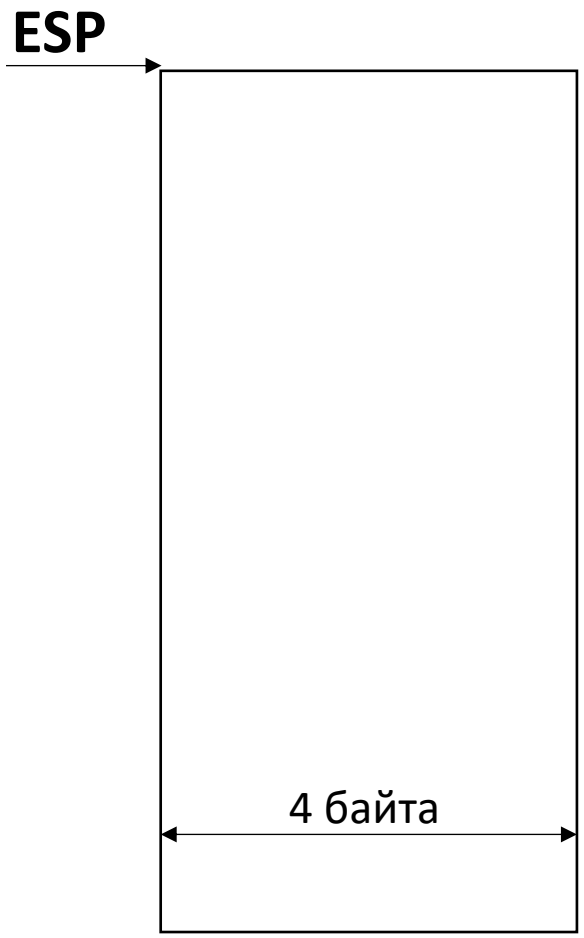
```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

EIP



Вызов cdecl-функции (Windows x86-32)

```
_f:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call fma
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```



Структуры в ассемблере

С точки зрения языка ассемблера структура эквивалентна группе переменных. Объекты эквивалентны структурам.

Адресация полей структуры осуществляется путем смещения относительно адреса начала структуры.

Хотя типов в языке ассемблера нет, для объявления структур есть специальный синтаксис. Данный синтаксис служит для облегчения работы с полями – размер структуры и смещения полей относительно начала структуры высчитываются автоматически. *Объявлять структуры таким образом необязательно.*

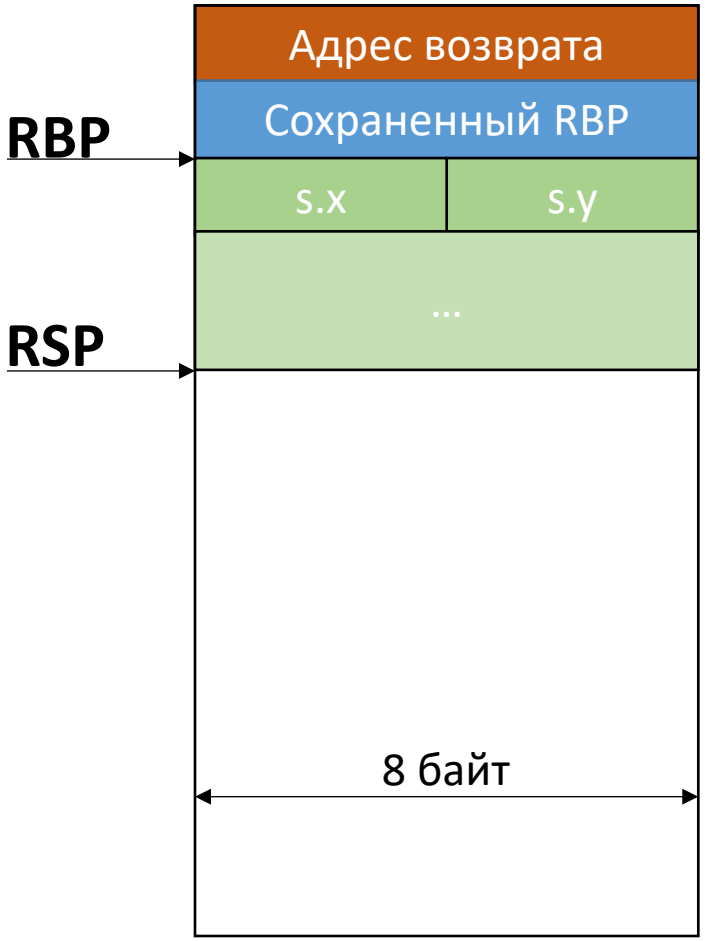
```
struct S{
    int x;
    int y;
};
```

```
struc S
    .x: resd 1
    .y: resd 1
endstruc
```

Структуры в ассемблере

```
struct S{
    int x;
    int y;
};

int main(){
    S s1;
    /*др. локальные*/
    s1.x = 10;
    s2.y = 20;
    …
}
```

```
struc S
    .x: resd 1
    .y: resd 1
endstruc

main:
    push    rbp
    mov     rbp, rsp
    sub     rsp, …
    lea     rax, [rbp-S_size]
    mov     DWORD[rax+S.x], 10
    mov     DWORD[rax+S.y], 20
    …
    xor     eax, eax
    leave
    ret
```



Простые структуры

Под простыми структурами в рамках предмета понимаются структуры или объекты:

- объявленные в языке C либо в рамках близкого к C подмножества языка C++;
- имеющие тривиальный конструктор копирования и деструктор, и все поля которых являются фундаментальными типами или простыми структурами.

Копирование простой структуры сводится к побайтовому копированию, а ее уничтожение не требует никаких дополнительных действий.

Возврат структур и объектов в cdecl

В случае сборки для Windows:

- Если результатом является простая структура размером менее 8 байт с тривиальным обычным конструктором и оператором присваивания, то результат возвращается в паре регистров EDX:EAX.
- Иначе, вызывающая функция создает буфер для результата, и передает указатель на буфер, как первый аргумент. Вызываемая функция записывает результат в буфер и возвращает указатель на буфер в EAX.

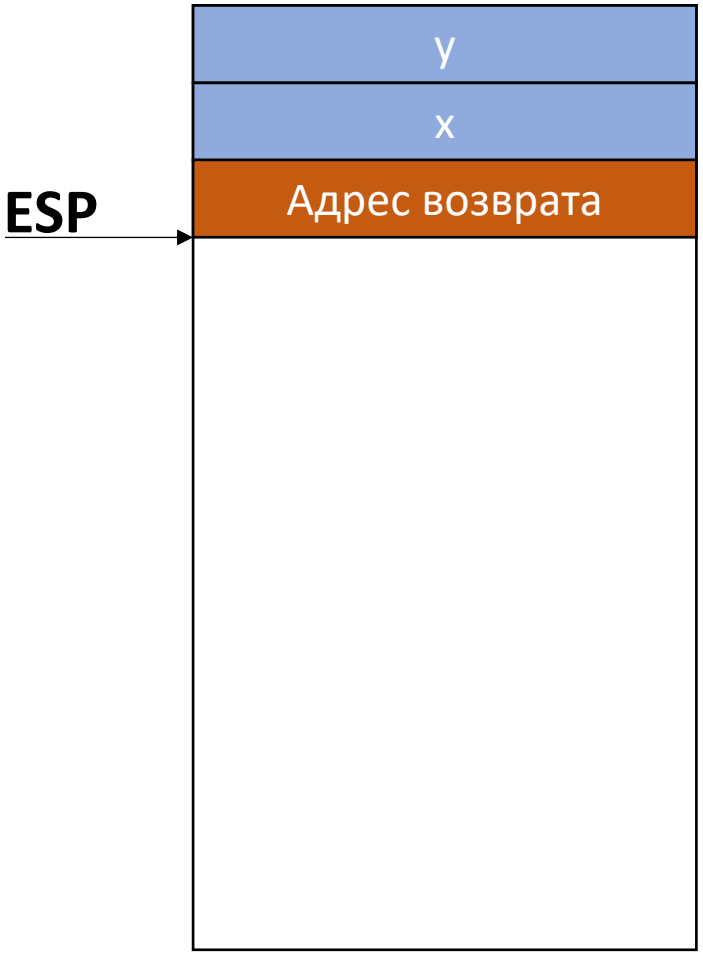
В случае сборки для Linux и др. UNIX-like систем:

- Если результатом является любая структура, то вызывающая функция создает буфер для результата, и передает указатель на буфер, как первый *скрытый* аргумент. Вызываемая функция записывает результат в буфер и возвращает указатель на буфер в EAX. *Указатель на скрытый аргумент убирает со стека вызываемая функция.*

Соглашение cdecl (Windows x86)

```
struct small{
    int x, y;
};

auto foo(int x, int y){
    return small{x, y};
}
```

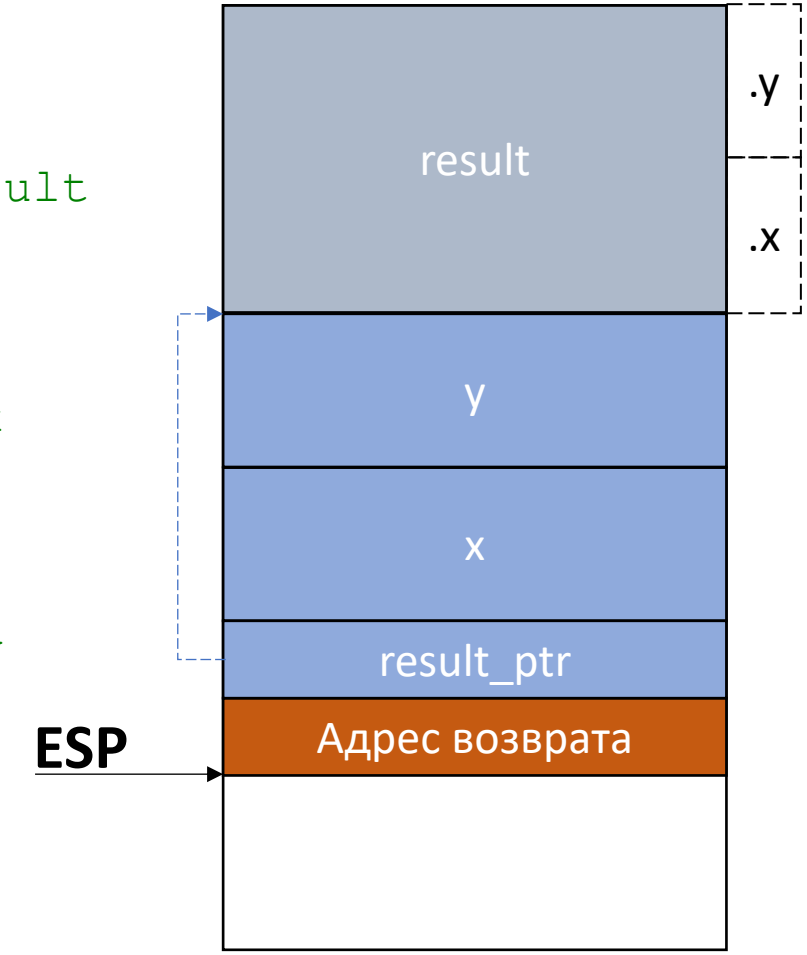
```
_foo:
    mov eax, [esp+4]   ; result.x=x
    mov edx, [esp+8]   ; result.y=y
    ret                ; return result
```



Соглашение cdecl (Windows x86)

```
struct large{
    long long x, y;
};

auto bar(long long x,
         long long y){
    return large{x, y};
}
```

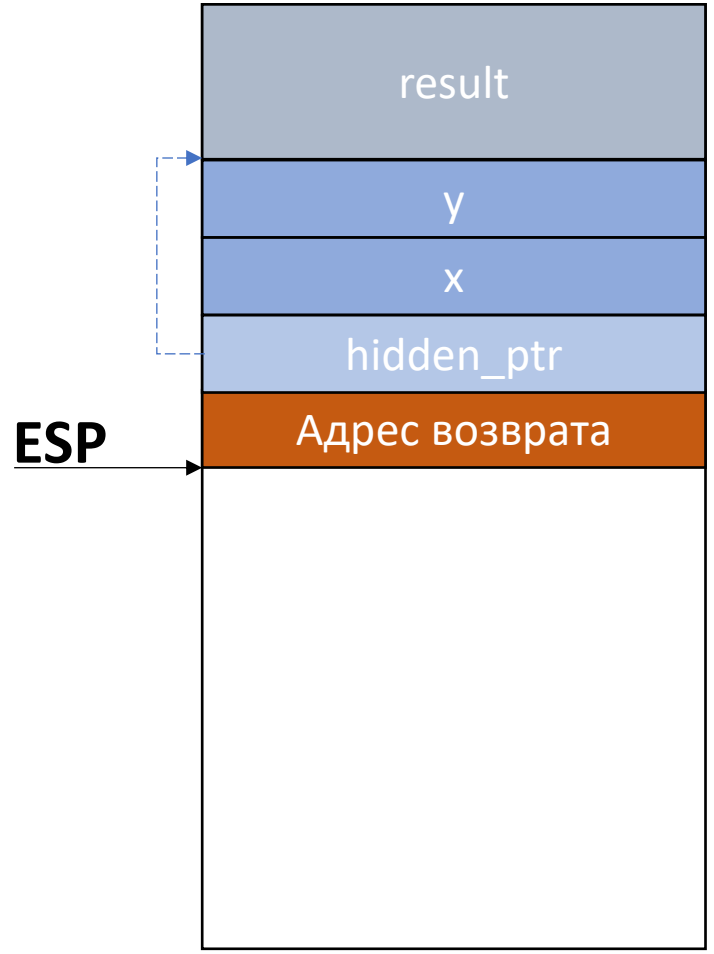
```
_bar:
    mov   eax, [esp+4] ; eax = &result
    mov   ecx, [esp+8]
    mov   edx, [esp+12]
    mov   [eax],ecx
    mov   [eax+4], edx  ;eax->x = x
    mov   ecx, [esp+16]
    mov   edx, [esp+24]
    mov   [eax+8], ecx
    mov   [eax+12], edx ;eax->y = y
    ret
```



Соглашение cdecl (UNIX-like x86)

```
struct small{
    int x, y;
};

auto foo(int x, int y){
    return small{x, y};
}
```

```
foo:
    mov eax, [esp+4]   ;eax = &result
    mov edx, [esp+8]
    mov [eax], edx     ;eax->x = x
    mov edx, [esp+12]
    mov [eax+4], edx   ;eax->y = y
    ret 4
```



Соглашение stdcall (x86)

Соглашение **stdcall** используется в WinAPI.

Данное соглашение аналогично cdecl, за исключением того, что *аргументы со стека убирает вызываемая функция*.

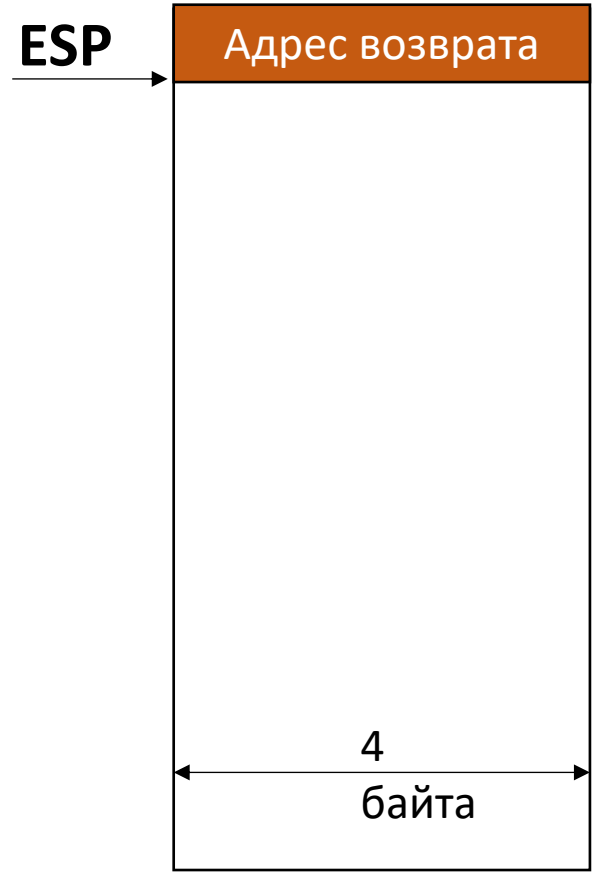
Для удаления аргументов используется инструкция `ret N`, где `N` – общий размер аргументов. После восстановления адреса возврата данная инструкция увеличивает значение регистра ESP на N.

При использовании этого соглашения к имени функции добавляется префикс _ и постфикс *@<общий размер аргументов>* (исключение – функции WinAPI, у которых отсутствует префикс _).

Вызов stdcall-функции (Windows x86-32)

```
int __stdcall fma(int a, int b, int c)
{
    return a*b+c;
}
```

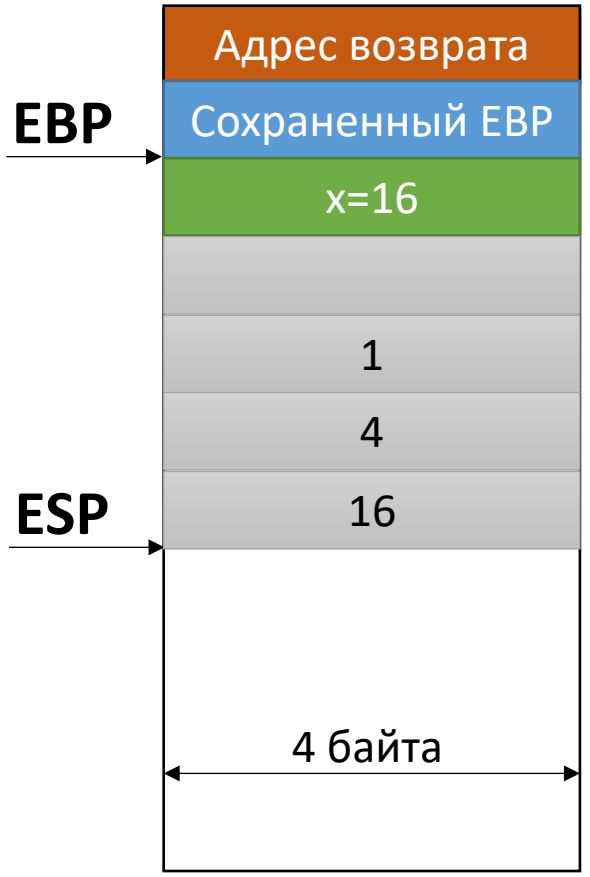
```
void __stdcall f()
{
   int x = 16;
   int z = fma(x, 4, 1);
   /*…*/
}
```



Вызов stdcall-функции (Windows x86-32)

```
_f@0:
    push ebp
    mov ebp, esp
    sub esp, 20
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 8], 1
    call _fma@12
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

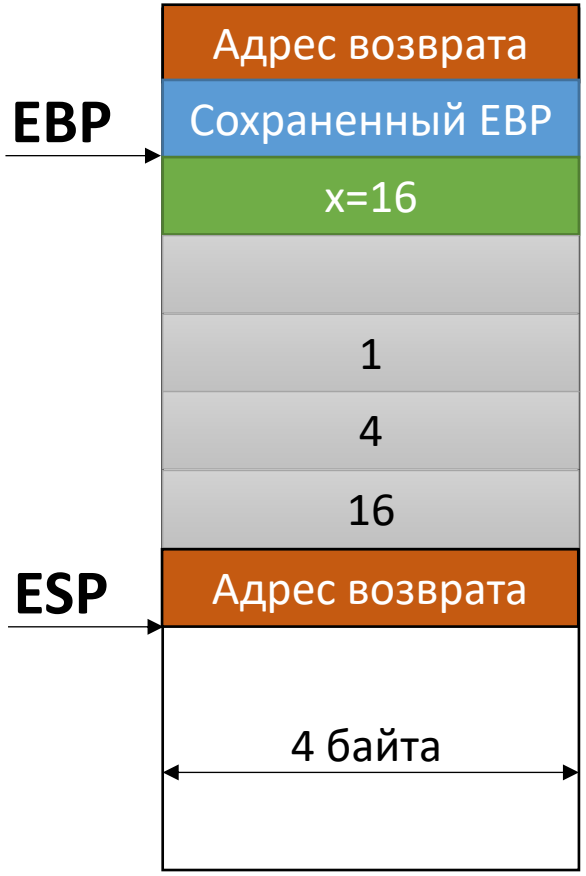
EIP



Вызов stdcall-функции (Windows x86-32)

```
int fma(int a, int b, int c)


_fma@12:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret 12
```

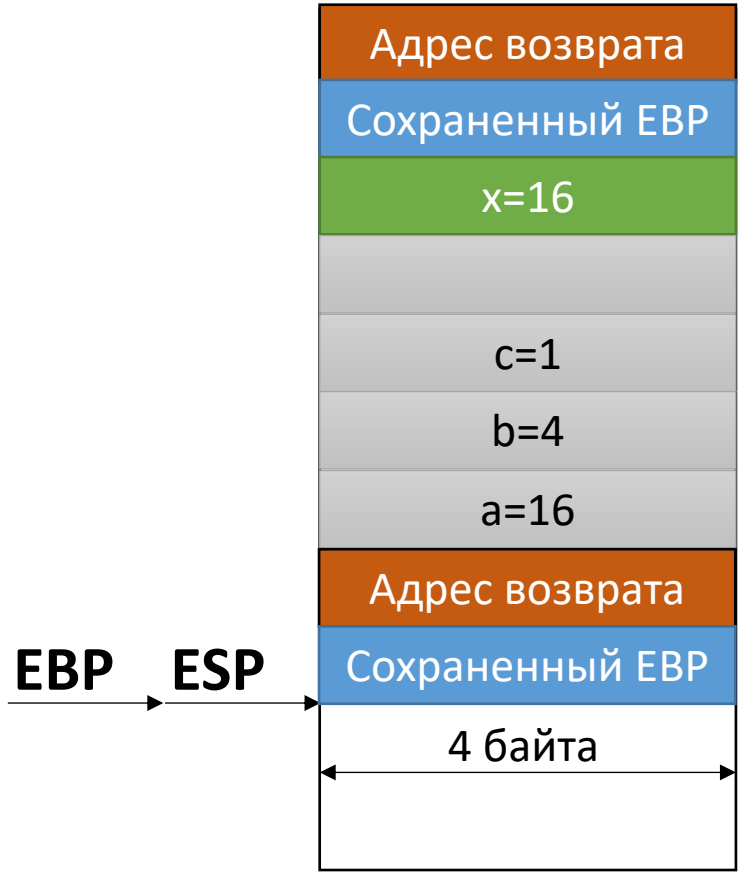
EIP



Вызов stdcall-функции (Windows x86-32)

```
int fma(int a, int b, int c)


_fma@12 :
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret 12
```

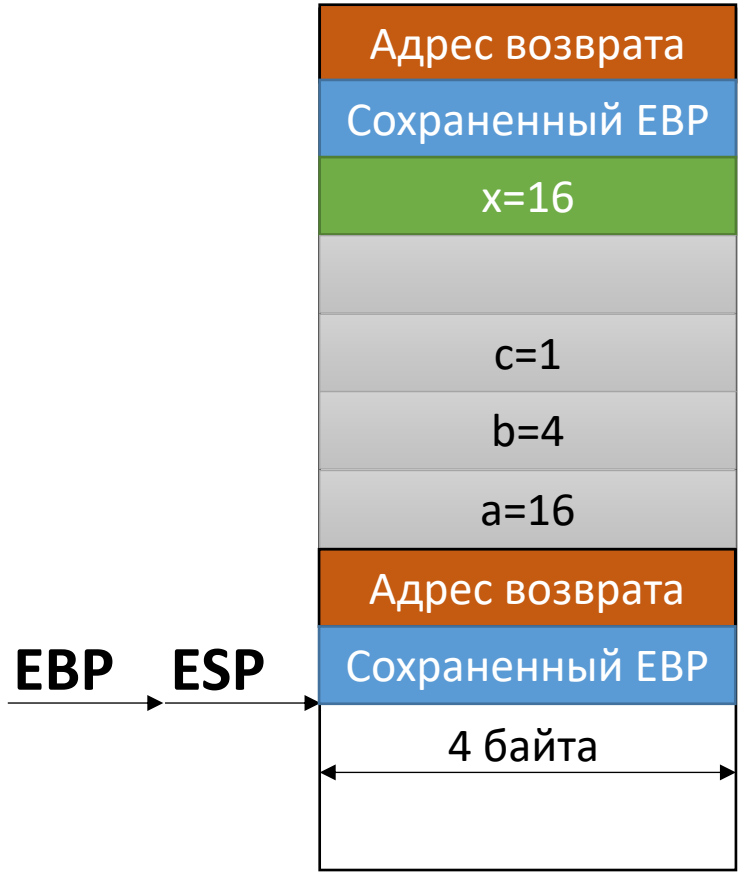
EIP



Вызов stdcall-функции (Windows x86-32)

```
int fma(int a, int b, int c)


_fma@12 :
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]      ;a
    imul eax, dword [ebp + 12]    ;a*b
    add eax, dword [ebp + 16]     ;a*b+c
    leave
    ret 12
```

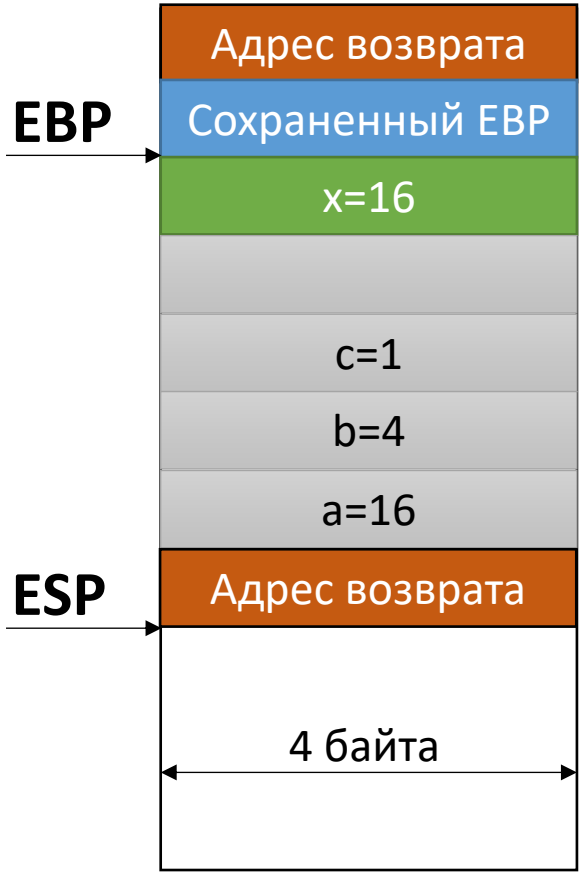
EIP



Вызов stdcall-функции (Windows x86-32)

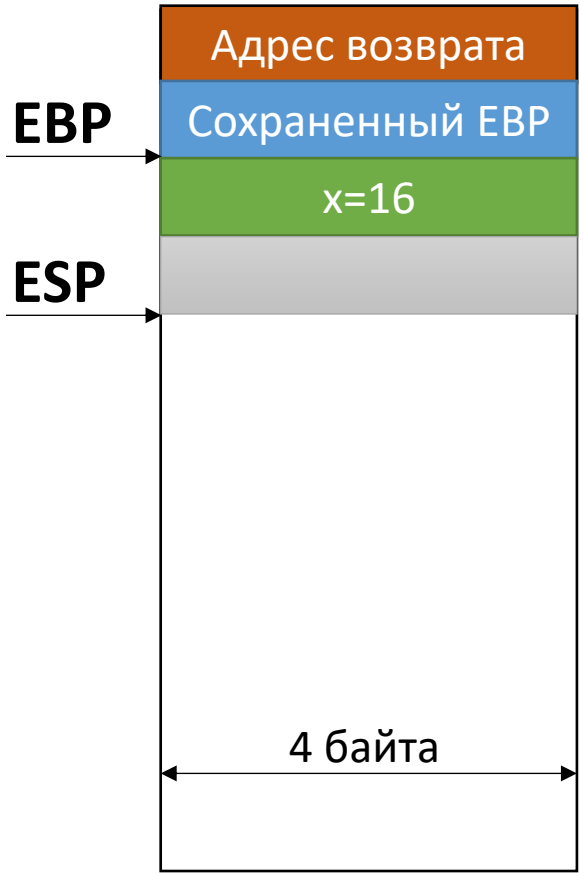
```
int fma(int a, int b, int c)


_fma@12 :
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret 12
```

EIP



Вызов stdcall-функции (Windows x86-32)

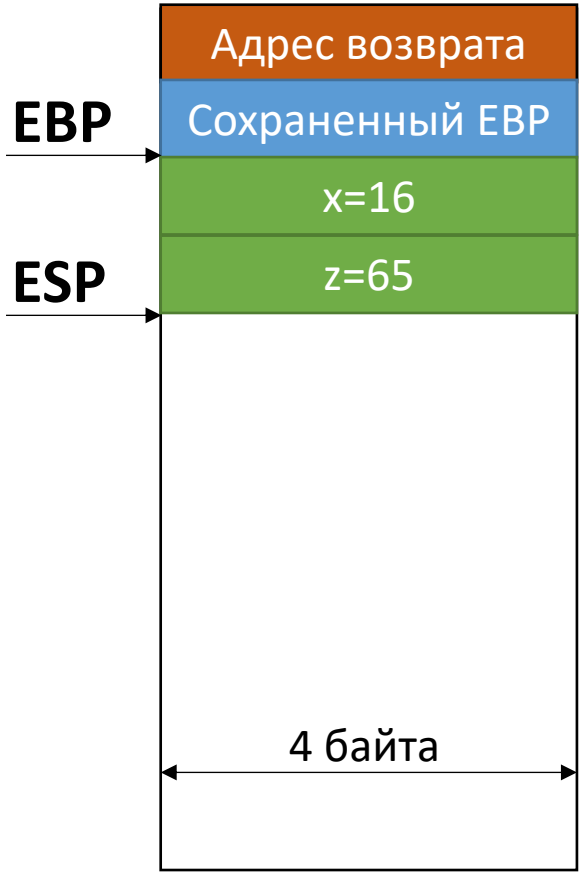
```
int fma(int a, int b, int c)


_fma@12:
    push ebp
    mov ebp, esp
    mov eax, dword [ebp + 8]
    imul eax, dword [ebp + 12]
    add eax, dword [ebp + 16]
    leave
    ret 12      ;результат в eax
```



Вызов stdcall-функции (Windows x86-32)

```
_f@0:
    push ebp
    mov ebp, esp
    sub esp, 24
    mov dword [ebp - 4], 16 ;x
    mov eax, dword [ebp - 4]
    mov dword [esp], eax
    mov dword [esp + 4], 4
    mov dword [esp + 4], 1
    call _fma@12
    mov dword [ebp - 8], eax ; z
    ; …
    xor eax, eax
    mov esp, ebp
    pop ebp
    ret
```

EIP



Вызов методов класса (x86)

При сборке для Windows для вызова методов класса используется соглашение **thiscall.** Данное соглашение аналогично stdcall, с той разницей, что указатель this передается в регистре ECX. Кроме того, если метод возвращает структуру, то она всегда возвращается через буфер на стеке (не через регистры).

Если метод класса имеет переменное число аргументов, используется соглашение cdecl с this в качестве первого аргумента.

При сборке для UNIX-like систем для вызова методов класса используется соглашение cdecl. Указатель this считается первым аргументом. Если метод возвращает структуру, то указатель на буфер является первым *скрытым* аргументом, this является вторым аргументом. Указатель this, как нескрытый параметр, удаляется вызывающей функцией.

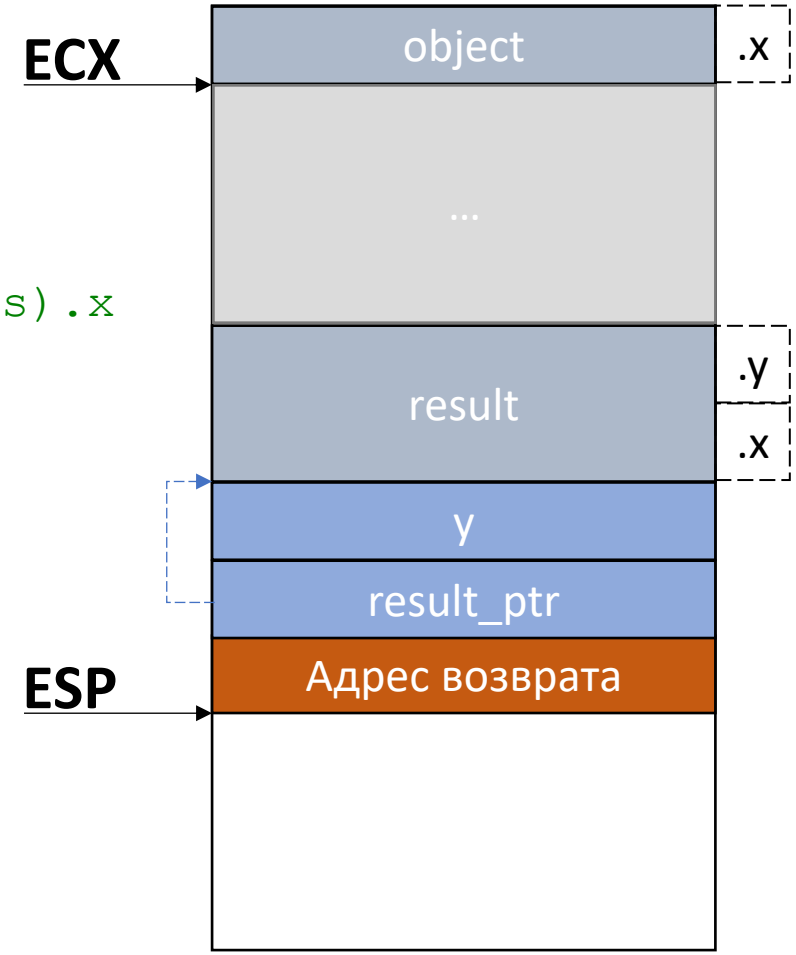
В любом случае, имя метода подвергается искажению, правила которого – свои для каждого компилятора (см. след. лекцию).

Стоит посмотреть: соглашения fastcall, pascal, vectorcall

Вызов методов класса (Windows x86)

```
struct small{
    int x, y;
};

class Cls{
public:
    int x;
    auto method(int y){
        return small{x, y};
    }
};
```

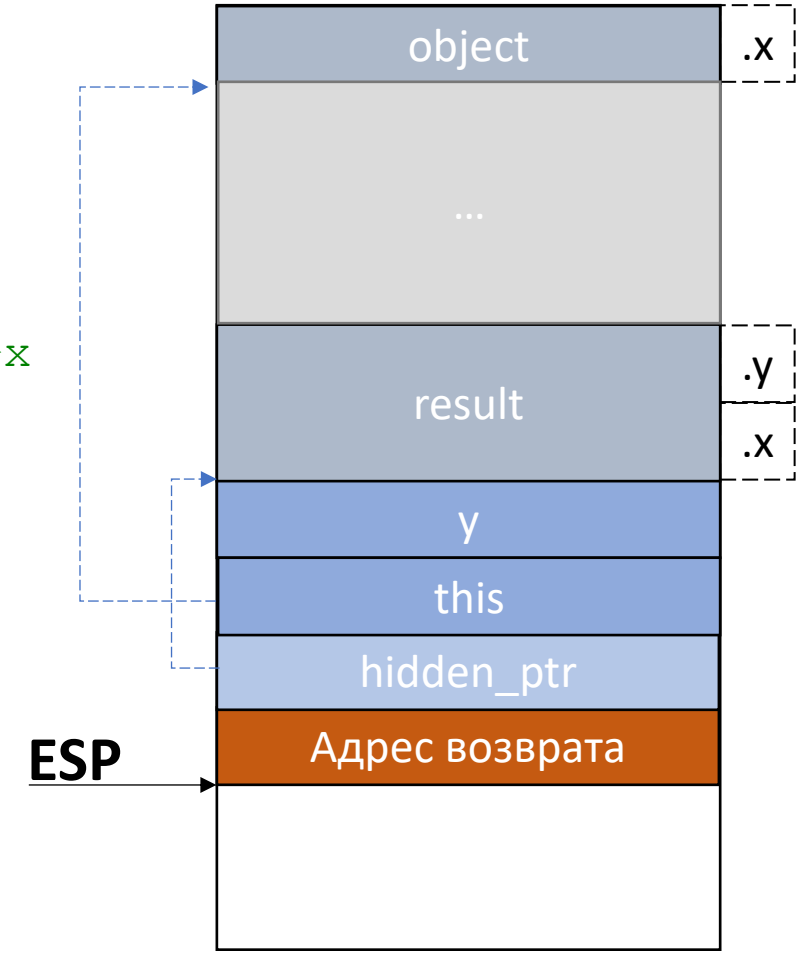
```
<method>:
    mov eax, [esp+4] ;eax=&result
    mov ecx, [ecx]   ;ecx = (*this).x
    mov edx, [esp+8] ;edx=y
    mov [eax], ecx
    mov [eax+4], edx
    ret 8
```



Вызов методов класса (UNIX-like x86)

```
struct small{
    int x, y;
};

class Cls{
public:
  int x;
  auto method(int y){
    return small{x, y};
  }
};
```

```
<method>:
    mov eax, [esp+4] ; eax = &result
    mov edx, [esp+8] ; edx = this
    mov edx, [edx]
    mov [eax], edx   ; result.x=this->x
    mov edx, [esp+12]
    mov [eax+4], edx ; result.y = y
    ret 4
```



Соглашение Microsoft x64

Соглашение используется при построении программ для ОС Windows.

- Целочисленные аргументы в RCX, RDX, R8-9; вещественные в XMM0-3. *Пятый* и далее аргументы - через стек в обратном порядке. Если размер аргумента не кратен 8, для него резервируется место, кратное 8.
- После завершения вызова аргументы из стека убирает вызывающая функция**.**
- Если у функции переменное число аргументов, и вещественное значение передается в XMM-регистре, оно должно также дублироваться в целочисленном регистре *без преобразования*.
- Простые структуры размером менее 8 байт передаются в регистре общего назначения. Если нет свободных регистров или структура не является простой – структура копируется отдельно, указатель на нее передается, как аргумент.
- Перед адресом возврата должно резервироваться **shadow space** длиной 32 байта. Аргументы на стеке располагаются после shadow space.
- Возвращаемое значение – в RAX или XMM0 (если вещественное).
- Простые структуры размером менее 8 байт с тривиальным обычным конструктором возвращаются в RAX, иначе – через буфер на стеке (передается 1-ым параметром), указатель на буфер возвращается в RAX.
- Изменяемые регистры: RAX, RCX, RDX, R8-R11, XMM0-5; остальные – неизменяемые.
- Вершина стека выравнивается по 8 байтам.
- **При вызове функции вершина стека должна быть выровнена по границе 16 байт**.

Соглашение Microsoft x64

```
int f(int a, double b,
      float c, double* d,
      int e, double f )
{
    int l1;
    float l3;
    l1 = (a + b)*c;
    l3 = e * (*d) - f;
    return l1 + l3;
}
```

a->ECX
b->XMM1
c->XMM2
d->R9,
e, f-> стек

```
f:
    cvtss2sd  xmm0,xmm2
    cvtsi2sd  xmm3,ecx
    addsd     xmm3,xmm1
    mulsd     xmm3,xmm0
    cvtsi2sd  xmm0,dword[rsp+40]
    cvtsd2si  eax,xmm3
    mulsd     xmm0, [r9]
    cvtsi2ss  xmm1,eax
    subsd     xmm0, [rsp+48]
    cvtpd2ps  xmm0,xmm0
    addss     xmm1,xmm0
    cvtss2si eax,xmm1
    ret
```

Соглашение System V

Соглашение используется при построении программ для UNIX-like ОС (в т.ч. Linux).

- Целочисленные аргументы передаются в RDI, RSI, RDX, RCX, R8-9; вещественные в XMM0-7. Те, что не поместились – через стек в обратном порядке. Если размер аргумента не кратен 8, для него резервируется место, кратное 8. Аргументы из стека убирает вызывающая функция**.**
- Если у функции переменное число аргументов, *в регистре AL должно передаваться число занятых XMM-регистров.*
- Возвращаемое значение – в RAX или XMM0 (если вещественное).
- Неизменяемые регистры: RBX, RBP, R12-15; остальные – изменяемые.
- После RSP находится **красная зона** длиной 128 байт.
- Вершина стека выравнивается по 8 байтам.
- **Вершина стека в момент вызова должна быть выровнена по границе 16 байт.**
- Простые структуры, размером менее 16 байт, могут передаваться в 1 или 2 регистрах. Часть, имеющая в составе целочисленные поля, передается в регистре общего назначения. Часть, имеющая в составе *только* вещественные числа, передается в XMM-регистре. Если свободных регистров нет, структура передается на стеке. Если структура не является простой – см. Microsoft x64.
- Возврат простых структур производится по тем же правилам, что и передача в роли аргумента с использованием регистров RAX, RDX, XMM0 и XMM1.

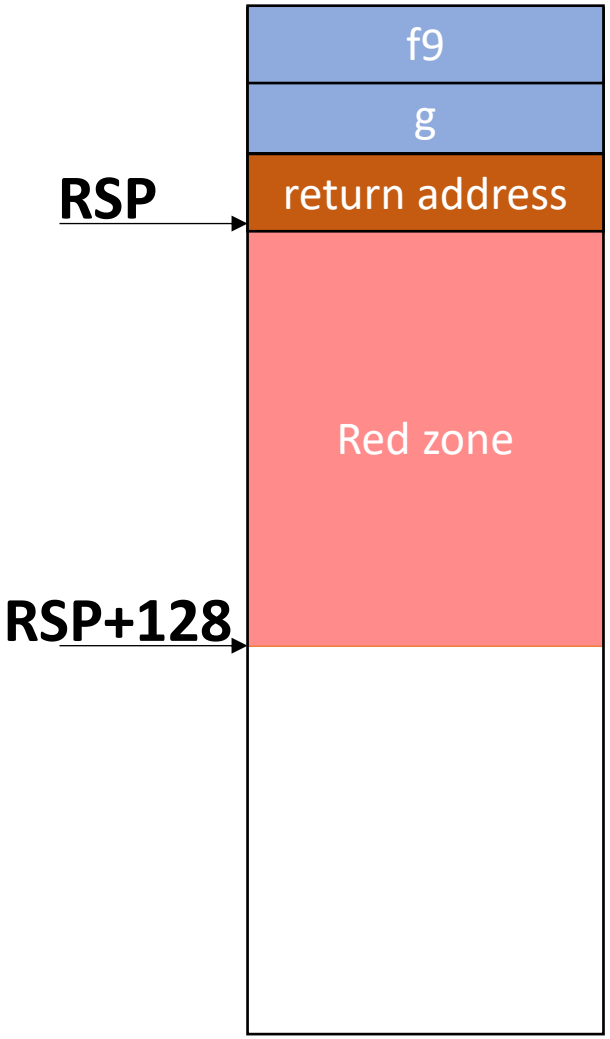
Красная зона

В ABI System V 128 байт, находящиеся ниже текущей вершины стека, являются **красной зоной** – ОС не может изменять значения в ней.

Если функция не вызывает другие функции (т.н. **leaf function, листовая функция**), то она может не тратить инструкции на выделение памяти на стеке. Вместо этого данные могут сохраняться в пределах красной зоны.

Листовой функции не требуется устанавливать собственный указатель кадра стека -> отсутствуют пролог и эпилог, а адресация аргументов и локальных переменных происходит относительно RSP.

Соглашение System V x64




```
int f(int a, long b, short c,
      char* d, int e, bool f,
      char g, float f1, float f2,
      float f3, float f4, float f5,
      float f6, double f7, double f8,
      double f9
)
{
       int l1= a+b+c-*d-f-g;
       float l2 =f1+f2+f3-f4-f5-f6;
       float l3= f7-f8-f9;
       return l1 -l2 + l3;
}
```

a->EDI, b->RSI, c->DX, d->RCX, e->R8d, f->R9b, g->стек,
f1-f8 -> XMM0-7, f9-> стек

```
f: movaps   xmm8, xmm1
   movsx    edx, dx
   movzx    r9d, r9b
   movsx    eax, BYTE[rsp+8]
   add      r9d, eax
   sub      edx, r9d
   movsx    eax, BYTE[rcx]
   sub      edx, eax
   add      edx, edi
   add      edx, esi
   pxor     xmm1, xmm1
   cvtsi2ss xmm1, edx
   addss    xmm0, xmm8
   addss    xmm0, xmm2
   subss    xmm0, xmm3
   subss    xmm0, xmm4
   subss    xmm0, xmm5
   subss    xmm1, xmm0
   subsd    xmm6, xmm7
   subsd    xmm6, [rsp+16]
   cvtsd2ss xmm6, xmm6
   addss    xmm1, xmm6
   cvttss2si eax, xmm1
   ret
```

Требования языка C для функций с переменным числом аргументов

Языки C/C++ предъявляют дополнительные требования к передаче значений в vararg-функции.

- значения (unsigned) char/short расширяются до (unsigned) int;
- значения типа float расширяются до double.

Как следствие, вызов vararg-функций из стандартной библиотеки языка C должен подчиняться данным правилам – **в первую очередь это касается printf()**.

(cdecl) https://godbolt.org/z/qWc4YjPcr;

(Win64) https://godbolt.org/z/6KrdnWz4o;

(Sys V) https://godbolt.org/z/MYE7he3xq.