Факультатив по программированию на языке C

Занятие 8
Язык ассемблера

План занятий

№	Тема	Описание
1	Введение в курс	Языки программирования. Основы работы с Linux.
2	Основы языка С	Написание и компиляция простейших программ с использованием gcc. Правила написания кода.
3	Компиляция	Разбиение программы на отдельные файлы. Make файлы. Компиляция.
4	Ввод данных. Библиотеки	Работа со вводом/выводом. Статические и динамические библиотеки.
5	Хранение данных. Память	Хранение процесса в памяти компьютера. Виртуальная память, сегментация. Секции программы.
6	Устройство памяти.	Elf файлы. Указатели и массивы. Типы данных. Gdb и отладка
7	Аллокация памяти	Аллокация памяти. Битовые операции – сдвиги, логические операции. Битовые поля. Перечисления. Static переменные. Inline функции.
8	Язык ассемблера	Язык ассемблера. Вызов функции. Безопасные функции. Макросы
9	Архиватор	Программирование архиватора

Дерево языка

Стандартный ввод и вывод

arch/x86/boot/printf.c

```
int printf(const char* fmt, ...)
{
   char printf_buf[1024];
   va_list args;
   int printed;
   va_start(args, fmt);
   printed = vsprintf(printf_buf, fmt, args);
   va_end(args);
   puts(printf_buf);
   return printed;
}
```

Переменное число аргументов

```c
#include <stdio.h>
#include <stdarg.h>
double sum_all(int num, ...)
{
    double sum = 0.0, s;
    va_list argptr;
    va_start(argptr, num);
    for (int i = 0; i < num; i++) {
        s = va_arg(argptr, double);
        sum += s;
    }
    va_end(argptr);
    return sum;
}

int main(void)
{
    double S;
    S = sum_all(5, 1.0, 0.2, 0.03, 0.004, 0.0005);
    printf("Sum = %f\n", S);
    return 0;
}
```

```
Sum = 1.234500
```

Оптимизация кода

```
int summ(int x, int y)
{
    return x + y;
}

double summ(double x, double y)
{
    return x + y;
}
```

Как оптимизировать код на C++?

Код см. в репозитории

Оптимизация кода

```
template <typename T>
T summ(T x, T y)
{
    return x + y;
}
```

Как оптимизировать код на С?

Код см. в репозитории

Макросы

```
#define PI 3.14159
#define circleArea(r) (PI*r*r)
```

Макросы - это препроцессорные "функции" , т.е. лексемы, созданные с помощью директивы #define, которые принимают параметры подобно функциям.

Макросы

№	Команда	Действие
1	ADD(x,y)	Y = X + Y
2	SUB(x,y)	Y = Y – X
3	MOV(x,y)	Положить X -> Y
4	INC(x)	X = X + 1
5	CMP(x,y)	Y – X, установка флагов
6	JNE(L)	Переход, если НЕ равно
7	PUSH(x)	Положить в стек из X
8	POP(x)	Достать из стека и положить в X

Напишем основные ассемблерные команды в виде макросов

Макросы

```
#define ADD(x,y) y = x + y
#define SUB(x,y) y = y - x
#define MOV(x,y) y = x
#define INC(x)   x++
#define JNE(L) if (CMP_FLAG != 0) goto L;
#define PRINT(x) printf("%d\n", x)
```

Макросы

```
#define PUSH(x)\
for (int i = stack_length-1; i > 0; i--)\
{\
    stack[i] = stack[i - 1];\
}\
    stack[0] = x;\
    esp++;
```

```
#define POP(x)\
for (int i = 0; i < stack_length - 1; i++)\
{\
    x = stack[0];\
    stack[i] = stack[i + 1];\
    esp--;\
}
```

```
#define CMP(x,y)          \
    if(x < y)    CMP_FLAG = -1;\
    if(x > y)    CMP_FLAG = 1;\
    if (x == y)  CMP_FLAG = 0;
```

Макросы

```c
#include <stdio.h>

#define DEF_SUM(type) type sum_##type (type a, type b) { \
     type result = a + b; \
     return result; \
}

DEF_SUM(int)
DEF_SUM(float)
DEF_SUM(double)

int main()
{
    printf("%d\n", sum_int(1, 2));
    printf("%lf\n", sum_float(2.4, 6.3));
    printf("%lf\n", sum_double(1.43434, 2.546656));
    return 0;
}
```

Макросы

```c
int sum_int(int a, int b) { int result = a + b; return result; }
float sum_float(float a, float b) { float result = a + b; return result; }
double sum_double(double a, double b) { double result = a + b; return result; }

int main()
{
    printf("%d\n", sum_int(1, 2));
    printf("%lf\n", sum_float(2.4, 6.3));
    printf("%lf\n", sum_double(1.43434, 2.546656));
    return 0;
}
```

```
gcc -E macros.c -o macros.ss
```

Макросы

Вернемся к примеру с аллокатором памяти

```
#ifdef DEBUG
      #define info(... ) safe_printf(__VA_ARGS__)
#else
      #define info(... )
#endif
```

```
info("Allocated:\t from %p to %p\n", memory_location,
memory_location + numbytes);
```

Makefile примера

```
DFLAGS=-Wall -g -Werror -DDEBUG
RFLAGS=-Wall -Werror -O3

all: clean
	gcc -c $(RFLAGS) *.c
	gcc *.o $(RFLAGS) -o main
debug: clean
	gcc -c $(DFLAGS) *.c
	gcc *.o $(DFLAGS) -o main
lib: clean
	gcc -c main.c -o main.o
	gcc -shared -fpic -o *.so *lib.c
	gcc *.o *.so -Wl,-rpath,. -o main
clean:
	rm -f *.o
clean_all:
	rm -f *.o
	rm -f *so
```

-D отвечает за декларацию макроса, идущего сразу после

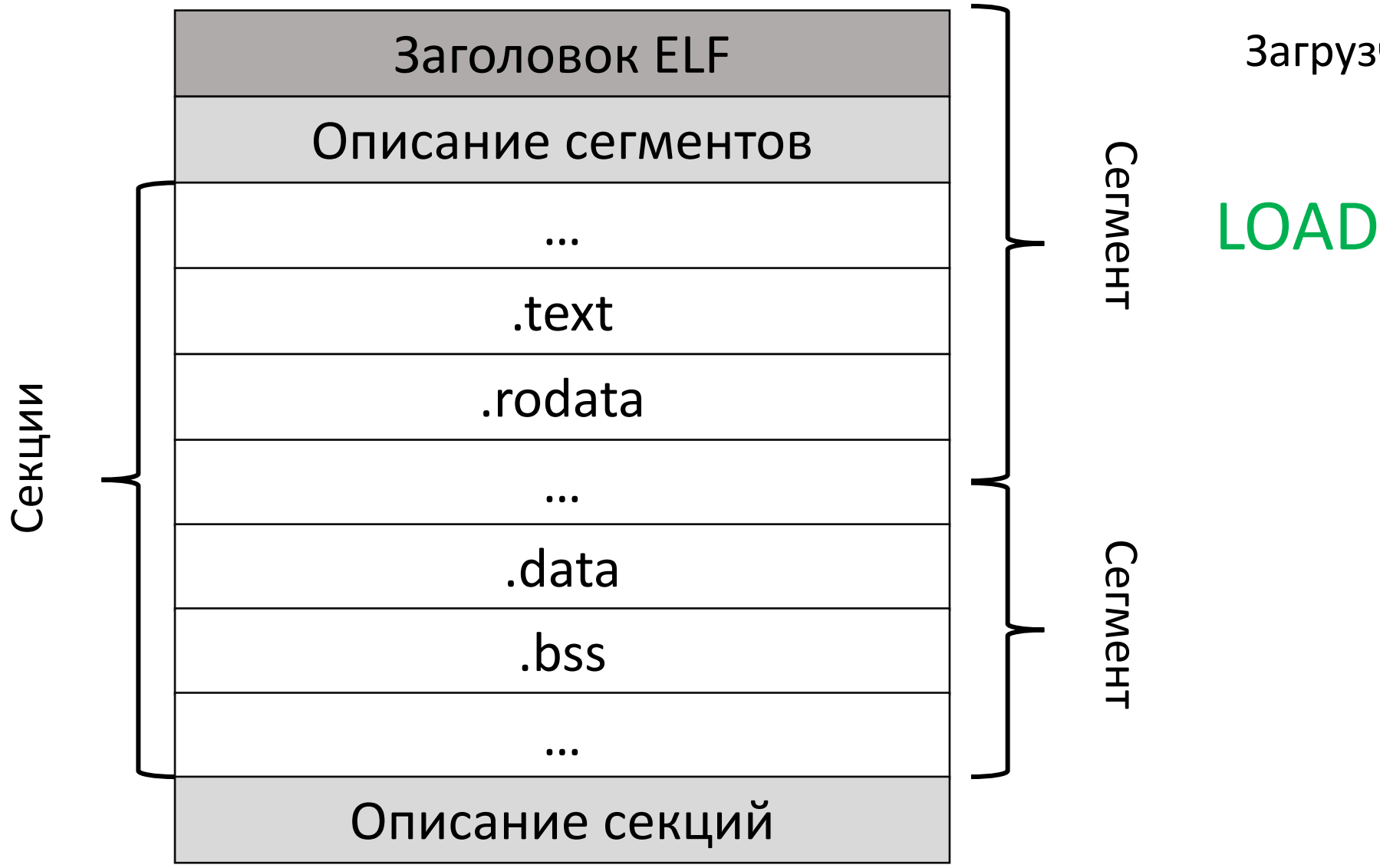
```
-DDEBUG == #define DEBUG
```

Перерыв ☺

Напоминание

Компоновщик

Загрузчик



Кратко об ассемблере

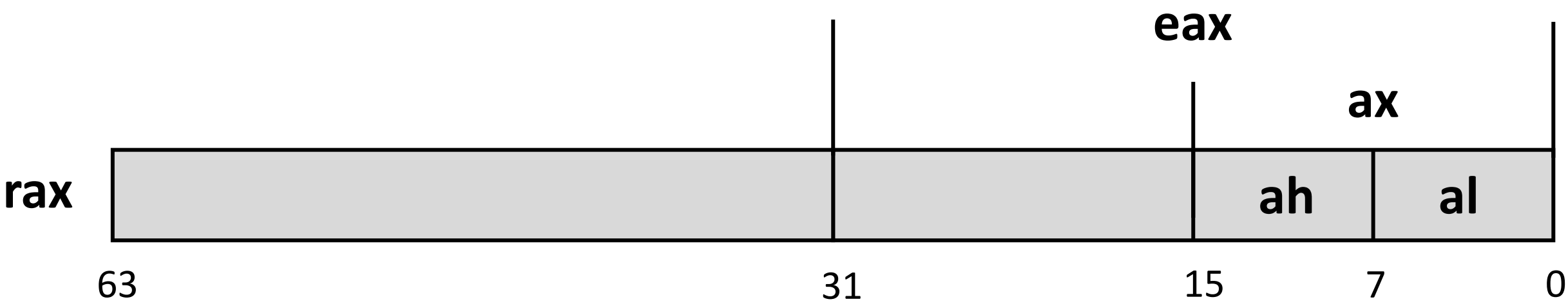
Регистр	Назначение
%eax	хранение результатов промежуточных вычислений
%ebx	хранения адреса (указателя) на некоторый объект в памяти
%ecx	счетчик
%edx	хранения результатов промежуточных вычислений
%esp	содержит адрес вершины стека
%ebp	указатель базы кадра стека
%esi	индекс источника
%edi	индекс приёмника

Команда	Назначение
mov *источник, назначение*	копирование *источника* в *назначение*
lea *источник*, *назначение*	помещает адрес *источника* в *назначение*
add *источник*, приёмник	*приёмник* = *приёмник* + *источник*
sub *источник*, приёмник	*приёмник* = *приёмник* - *источник*
push *источник*	поместить в стек
pop *назначение*	извлечь из стека
cmp *операнд_2*, *операнд_1*	*операнд_1* – *операнд_2* и устанавливает флаги
jle *метка*	Переход если <=

Регистры

64-bit register	Lowest 32-bits	Lowest 16-bits	Lowest 8-bits
rax	eax	ax	al
rbx	ebx	bx	bl
rcx	ecx	cx	cl
rdx	edx	dx	dl
rsi	esi	si	sil
rdi	edi	di	dil
rbp	ebp	bp	bpl
rsp	esp	sp	spl

Регистры



Системные вызовы

```
#
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The __x64_sys_*() stubs are created on-the-fly for sys_*() system calls
#
# The abi is "common", "64" or "x32" for this file.
#
0       common  read                    sys_read
1       common  write                   sys_write
2       common  open                    sys_open
3       common  close                   sys_close
```

https://github.com/torvalds/linux/blob/master/**arch/x86/entry/syscalls/syscall_64.tbl**

Системные вызовы

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count
2	sys_open	const char *filename	int flags	int mode
3	sys_close	unsigned int fd		

https://blog.rchapman.org/posts/Linux_System_Call_Table_for_x86_64/

Классический пример

```c
#include <unistd.h>

int main()
{
    char str[] = "Hello, MIET!\n";
    write(1, str, sizeof(str) - 1);
    _exit(0);
}
```

Вместо стандартных функций мы используем напрямую системные вызовы

Классический пример

```
 .global _start
        .text
_start:
        # write(1, message, 12)
        mov     $1, %rax                # системный вызов 1
        mov     $1, %rdi                # связываем с файловым дескриптором 1
        mov     $message, %rsi          # передаем адрес строки
        mov     $12, %rdx               # размер строки
        syscall                         # системный вызов

        # exit(0)
        mov     $60, %rax               # системный вызов 1
        mov     $0, %rdi                # передаем 0 в качетсе аргумента
        syscall                         # системный вызов
message:
        .ascii  "Hello, MIET\n"
```

Проанализируем программу - `readelf -e ex1`

```
gcc -c ex1.s
ld ex1.o -o ex1
./ex1
```

Классический пример

```
 .global _start
        .text
_start:
        # write(1, message, 12)
        mov     $1, %rax                # системный вызов 1
        mov     $1, %rdi                # связываем с файловым дескриптором 1
        mov     $message, %rsi          # передаем адрес строки
        mov     $12, %rdx               # размер строки
        syscall                         # системный вызов

        # exit(0)
        mov     $60, %rax               # системный вызов 60
        mov     $0, %rdi                # передаем 0 в качестве аргумента
        syscall                         # системный вызов

       .data
message:
        .ascii  "Hello, MIET\n"
```

```
gcc -c ex1.s && ld ex1.o -o ex1 && readelf -e ex1
```

Данные

.byte — размещает каждое выражение как 1 байт;
.short — 2 байта;
.long — 4 байта;
.quad — 8 байт;
.ascii — разместить строку без нуль символа;
.string — разместить строку с нуль символом;

Самостоятельное задание

%rax	System call	%rdi	%rsi	%rdx
0	sys_read	unsigned int fd	char *buf	size_t count
1	sys_write	unsigned int fd	const char *buf	size_t count
2	sys_open	const char *filename	int flags	int mode
3	sys_close	unsigned int fd		

Добавьте ввод сообщение с помощью системного вызова read

Возможное решение

```
 .global _start
        .text
_start:
        mov     $0, %rax                # системный вызов 0
        mov     $0, %rdi                # связываем с файловым дескриптором 0
        mov     $x, %rsi                # передаем адрес строки
        mov     $5, %rdx                # размер строки
        syscall                         # системный вызов

        mov     $1, %rax                # системный вызов 1
        mov     $1, %rdi                # связываем с файловым дескриптором 1
        mov     $x, %rsi                # передаем адрес строки
        mov     $5, %rdx                # размер строки
        syscall                         # системный вызов

        mov     $60, %rax               # системный вызов 60
        mov     $0, %rdi                # передаем 0 в качестве аргумента
        syscall                         # системный вызов

        .data
x:      .long    10
```

Ветвление

jcc метка

Мнемоника	Английское слово	Смысл	Тип операндов
`e`	equal	равенство	любые
`n`	not	инверсия условия	любые
`g`	greater	больше	со знаком
`l`	less	меньше	со знаком
`a`	above	больше	без знака
`b`	below	меньше	без знака

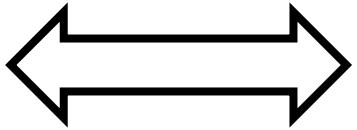
Пример ветвления

```
_start:

        movq    $y, %rbx
        movq    (%rbx), %rax
        cmp     $5, %rax
        jne N_EQ
        #Действие 1

N_EQ:
        #Действие 2


        .data
y:
    .quad 6
```



Модернизируйте программу, следующим образом:

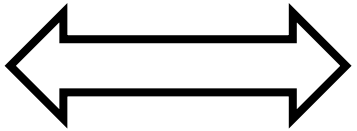
```
if(y==5)
    {
        #Вывод на экран «equal»
    }
        #Выход из программы
```

Циклы

```
_start:

        movq     $0, %rbx
N_EQ:
       #Действие

        add      $1, %rbx
        cmp      $5, %rbx
        jne N_EQ
```



Модернизируйте программу, следующим образом:

```
eax=0;
do
{
    #Вывод на экран «no equal»
    eax = eax + 1;
}
while(eax != 5)
#Вывод на экран «equal»
#Выход из программы
```

Ассемблерные вставки

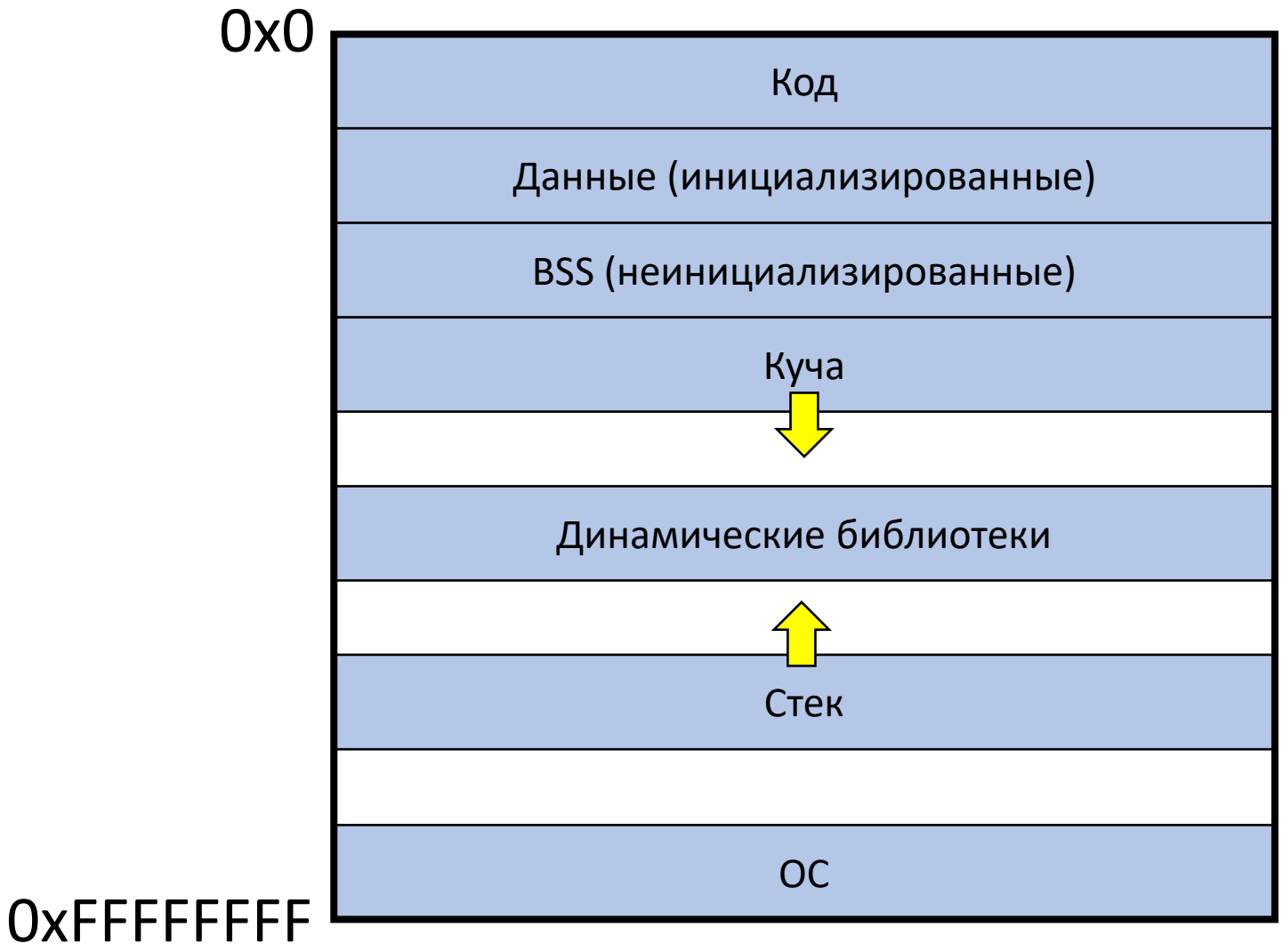
```c
#include <stdio.h>

int main()
{
    int sum = 0, x = 1, y = 2;
    asm("add %1, %0"
    : "=r" (sum)
    : "r" (x), "0" (y)); // sum = x + y;
    printf("sum = %d, x = %d, y = %d \n", sum, x, y); // sum = 3, x = 1, y = 2
    return 0;
}
```

Как выиграть в игру?

```
sab@LAPTOP-B03PIUAN:.../Secret$ ./secret
Enter secret number:
123
You are so wrong!
```

Секции программы



Напоминание: ассемблер (32-bit x86)

Регистр	Назначение
%eax	хранение результатов промежуточных вычислений
%ebx	хранения адреса (указателя) на некоторый объект в памяти
%ecx	счетчик
%edx	хранения результатов промежуточных вычислений
%esp	содержит адрес вершины стека
%ebp	указатель базы кадра стека
%esi	индекс источника
%edi	индекс приёмника

Команда	Назначение
mov *источник, назначение*	копирование *источника* в *назначение*
lea *источник, назначение*	помещает адрес *источника* в *назначение*
add *источник*, приёмник	*приёмник* = *приёмник* + *источник*
sub *источник*, приёмник	*приёмник* = *приёмник* - *источник*
push *источник*	поместить в стек
pop *назначение*	извлечь из стека
cmp *операнд_2, операнд_1*	*операнд_1 – операнд_2* и устанавливает флаги
jle *метка*	Переход если <=

Вызов функции

```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

Вызов функции

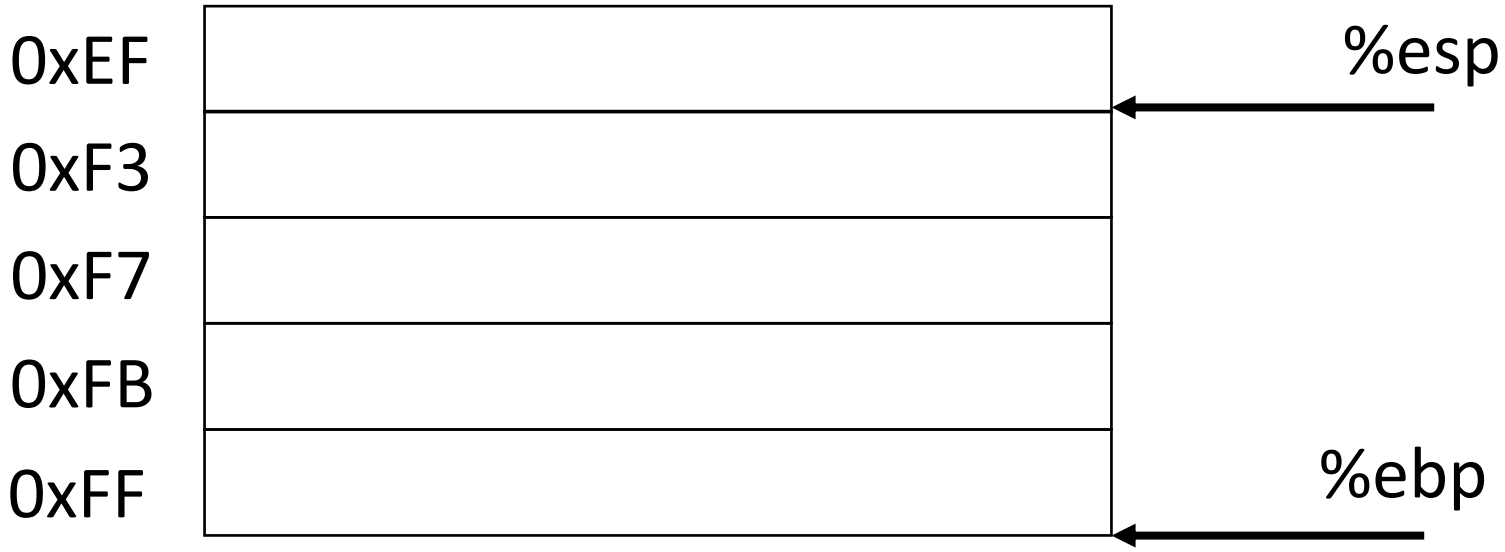
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

Стек - напоминание



Функции ассемблера

`leave`	prepares the stack for leaving a function. Equivalent to:
	`mov %ebp, %esp`
	`pop %ebp`

`call addr <fname>`	switches active frame to callee function. Equivalent to:
	`push %eip`
	`mov addr, %eip`

`ret`	restores active frame to caller function. Equivalent to: `pop %eip`

Вызов функции

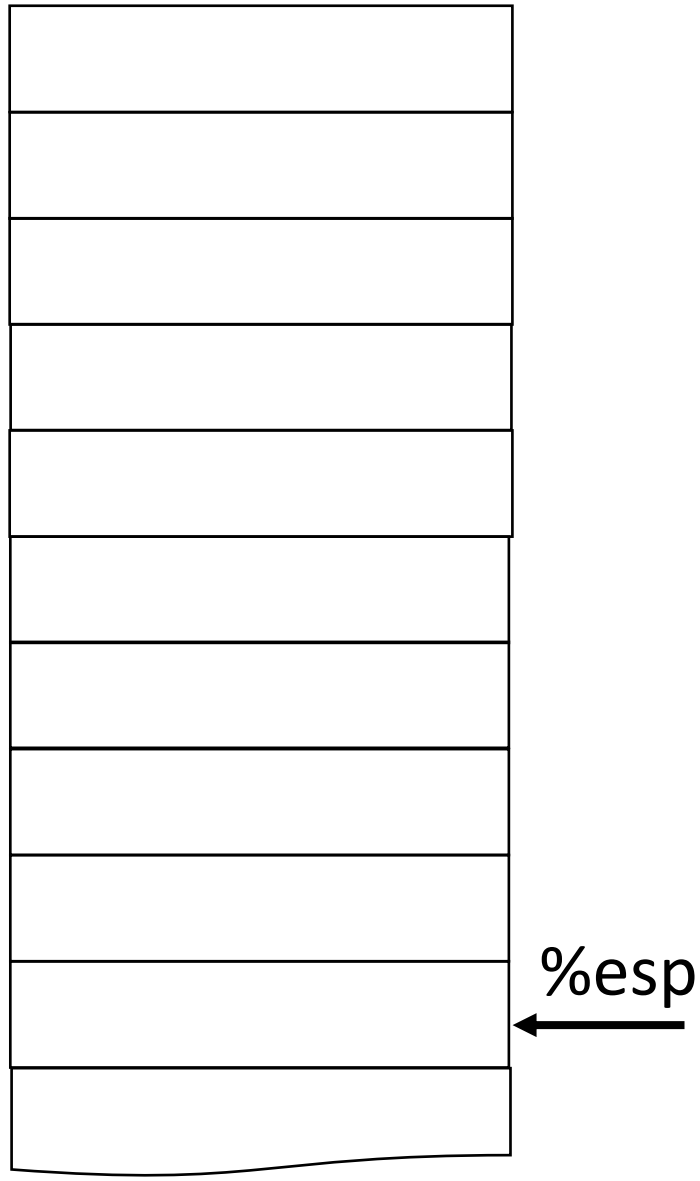
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd07c
%ebp	0x0
%eip	0x565561c5



Вызов функции

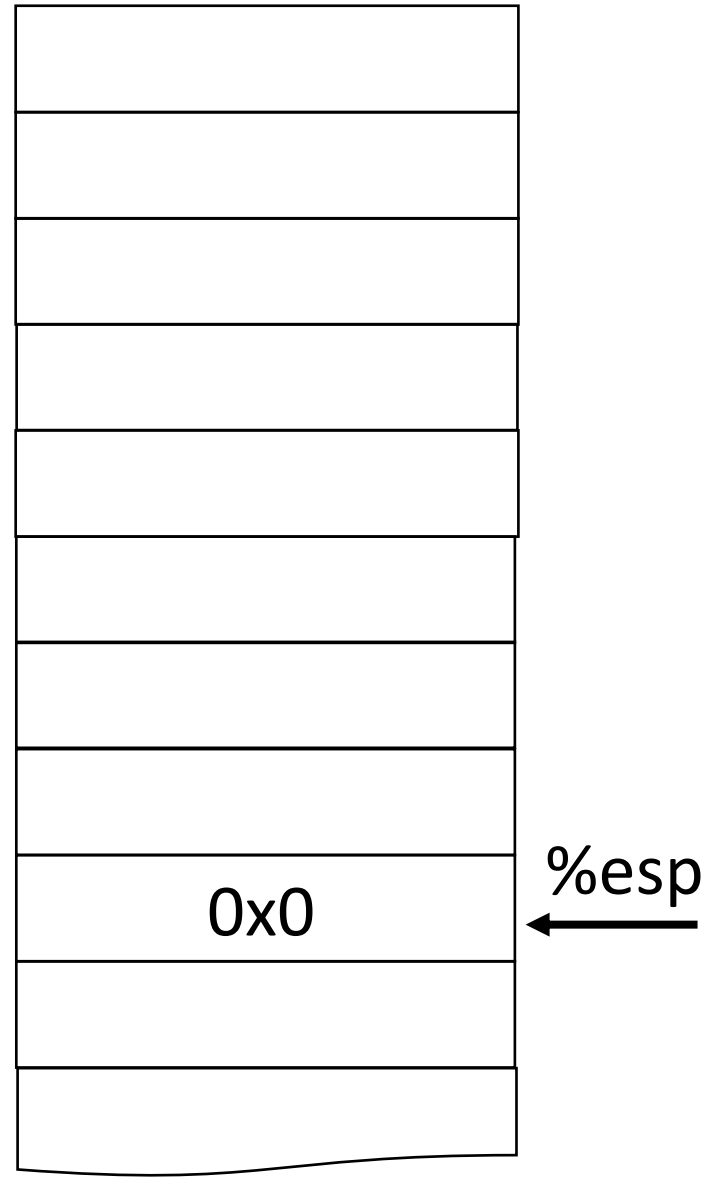
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd078
%ebp	0x0
%eip	0x565561c9



Вызов функции

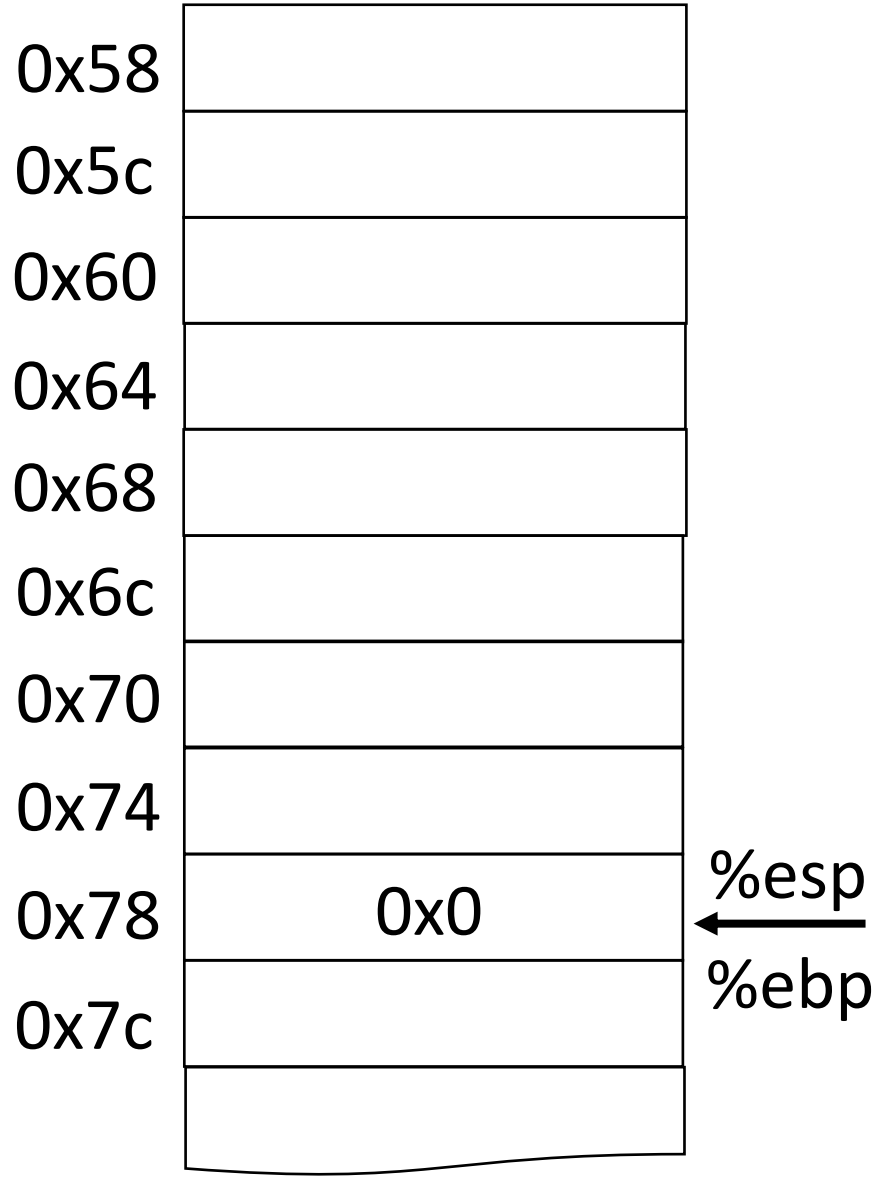
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd078
%ebp	0xffffd078
%eip	0x565561ca



Вызов функции

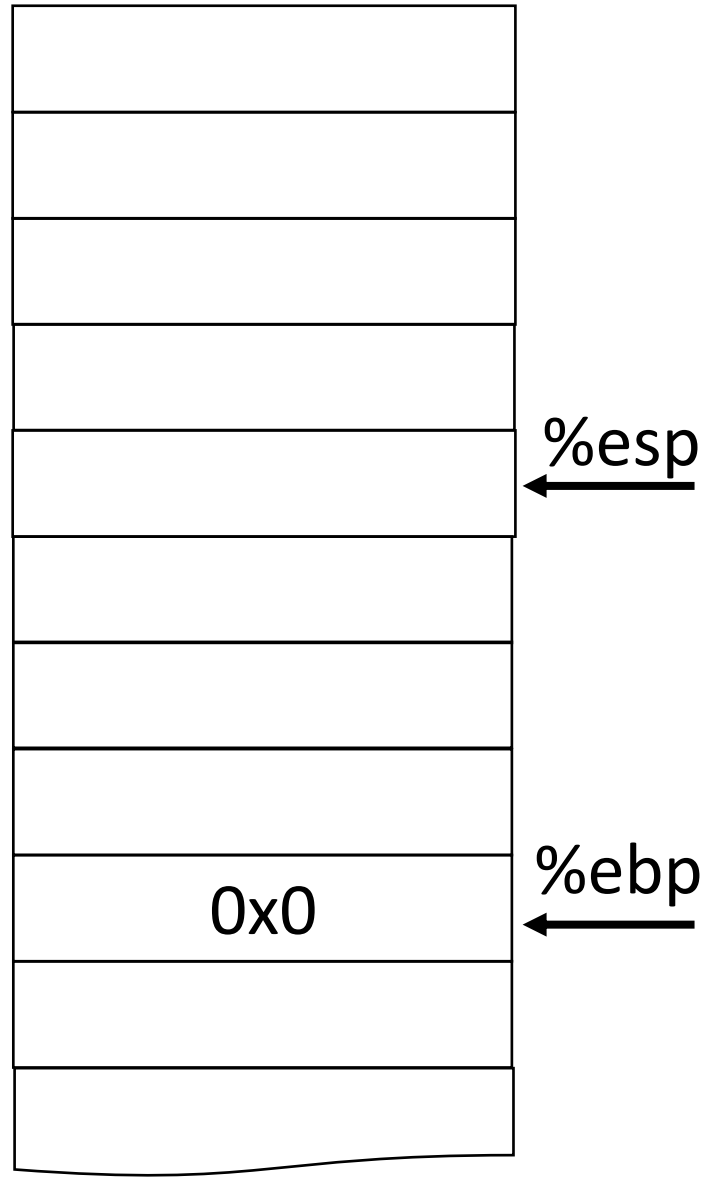
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd068
%ebp	0xffffd078
%eip	0x565561cc



Вызов функции

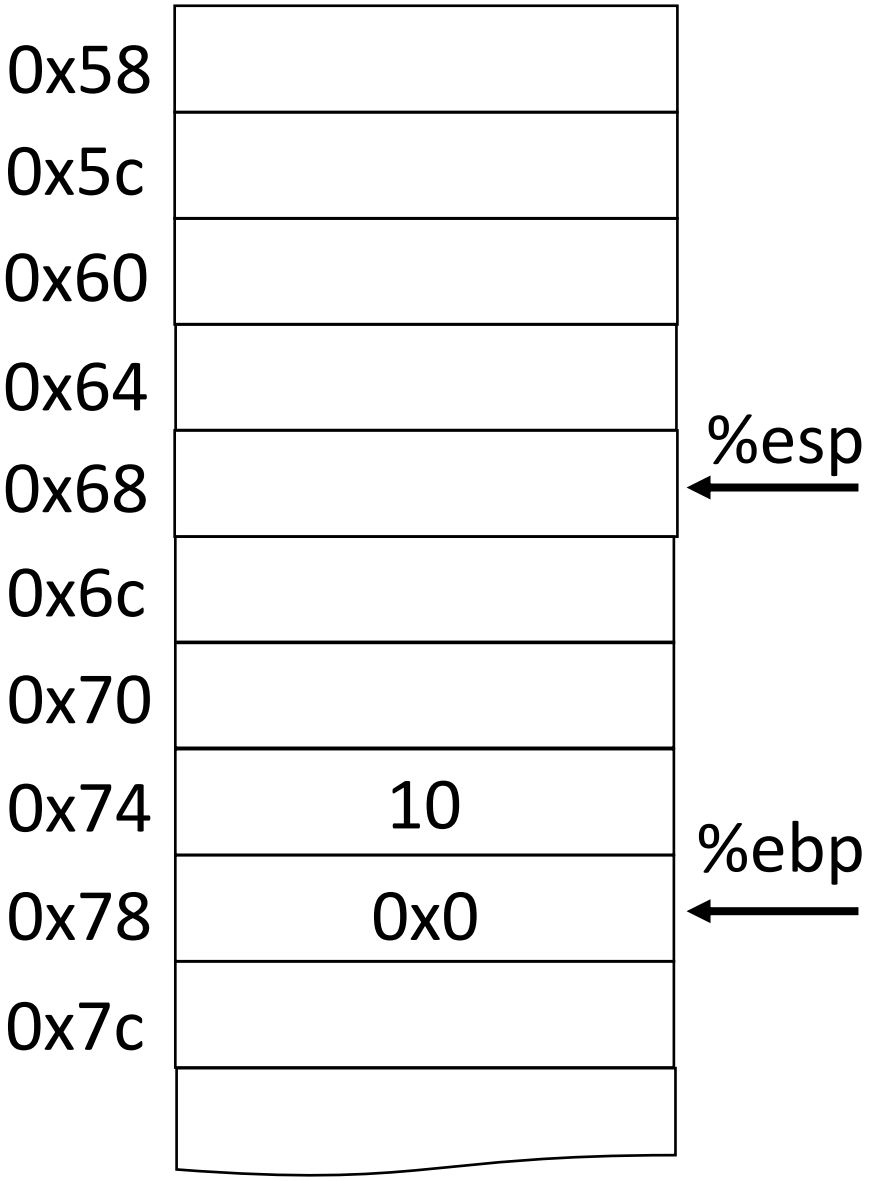
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd068
%ebp	0xffffd078
%eip	0x565561d9



Вызов функции

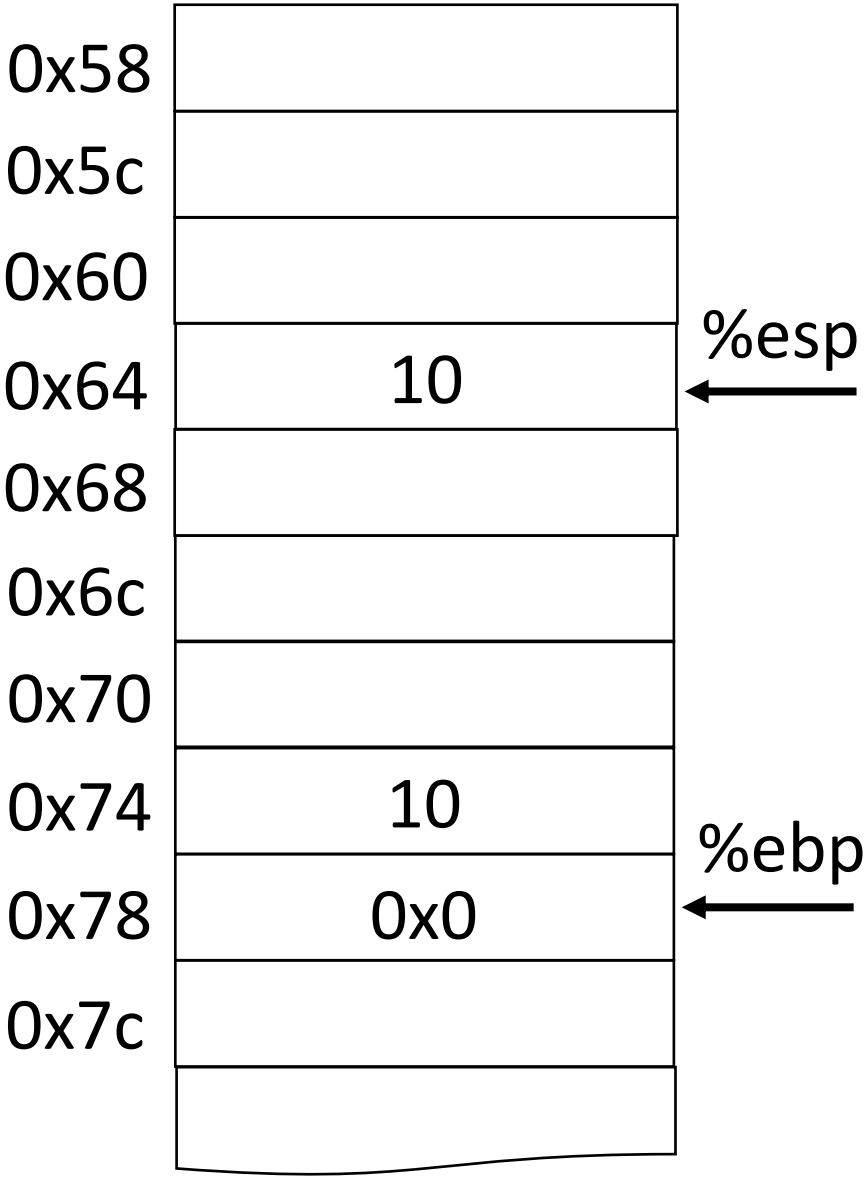
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd064
%ebp	0xffffd078
%eip	0x565561df



Вызов функции

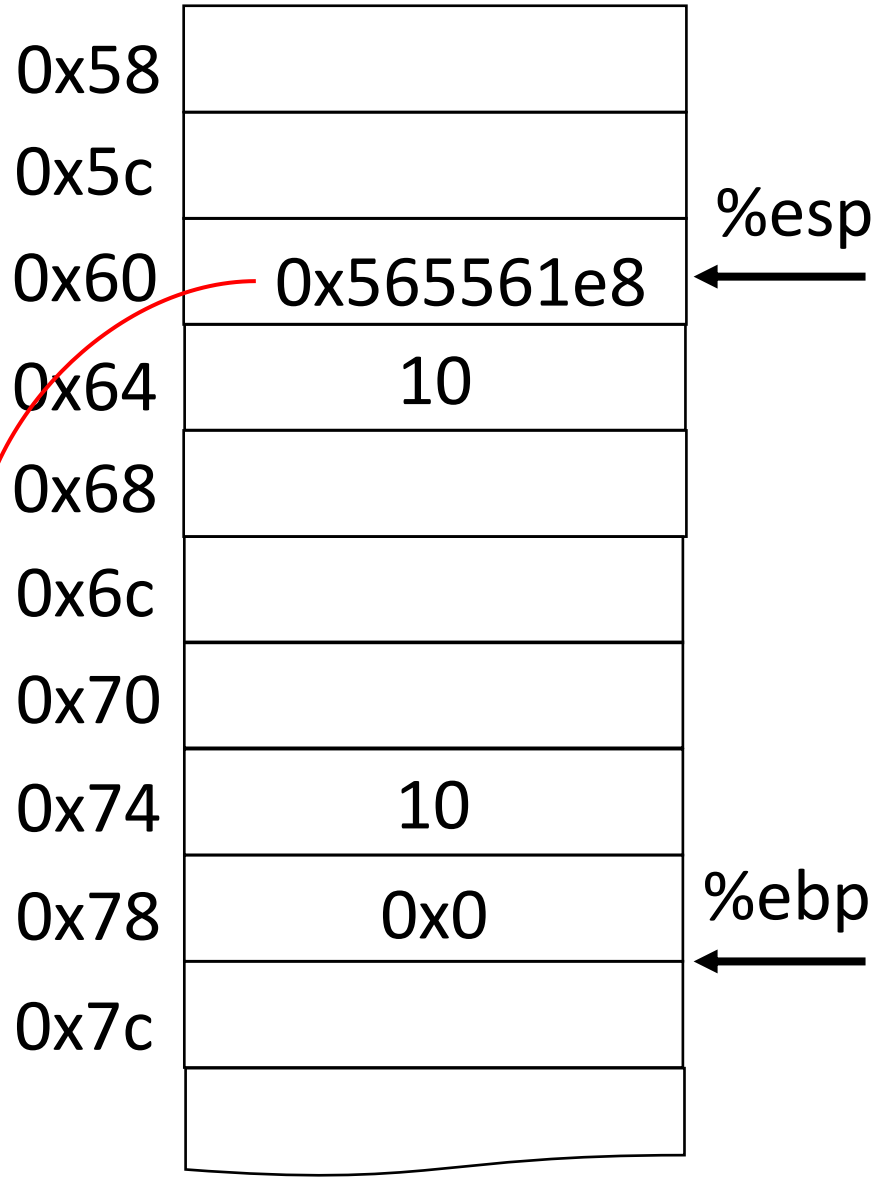
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd060
%ebp	0xffffd078
%eip	**0x565561b1**



Вызов функции

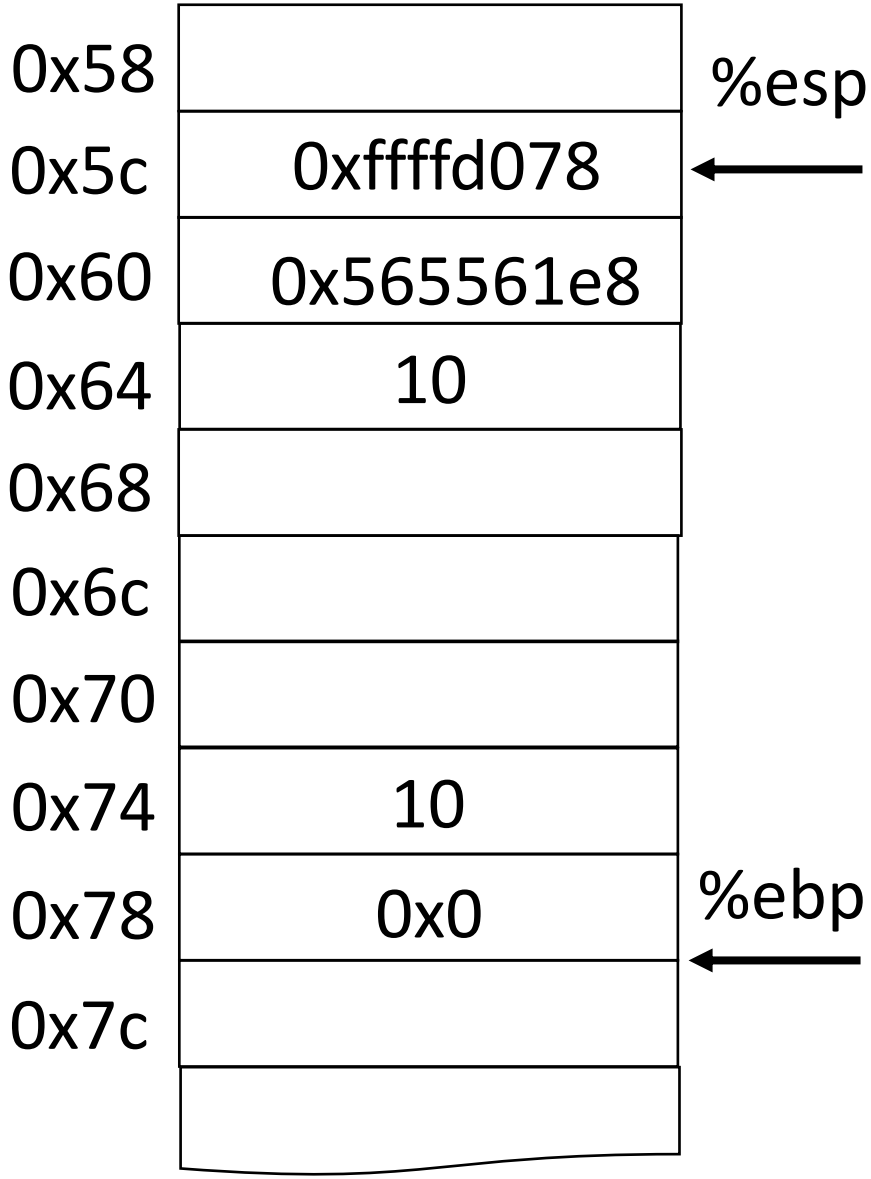
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd05c
%ebp	0xffffd078
%eip	0x565561b1



Вызов функции

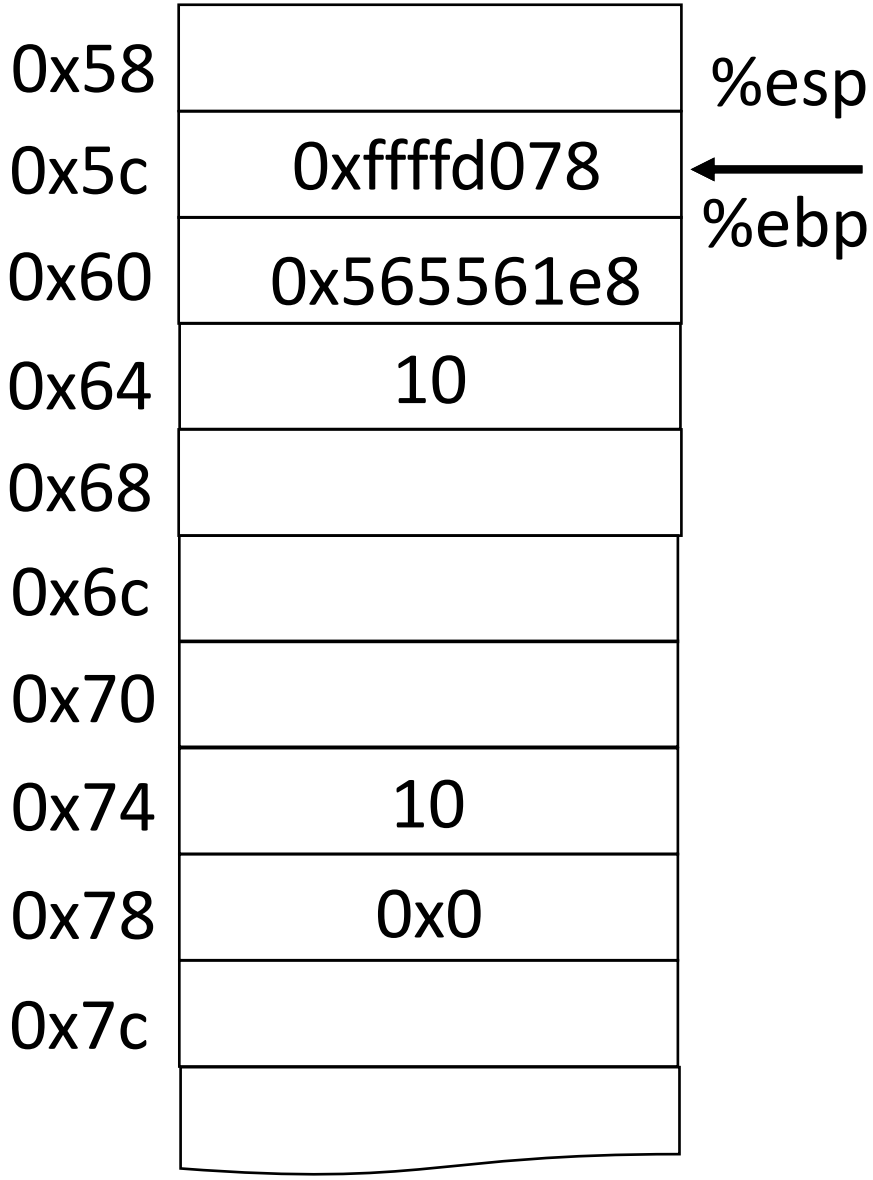
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd05c
%ebp	0xffffd05c
%eip	0x565561b2



Вызов функции

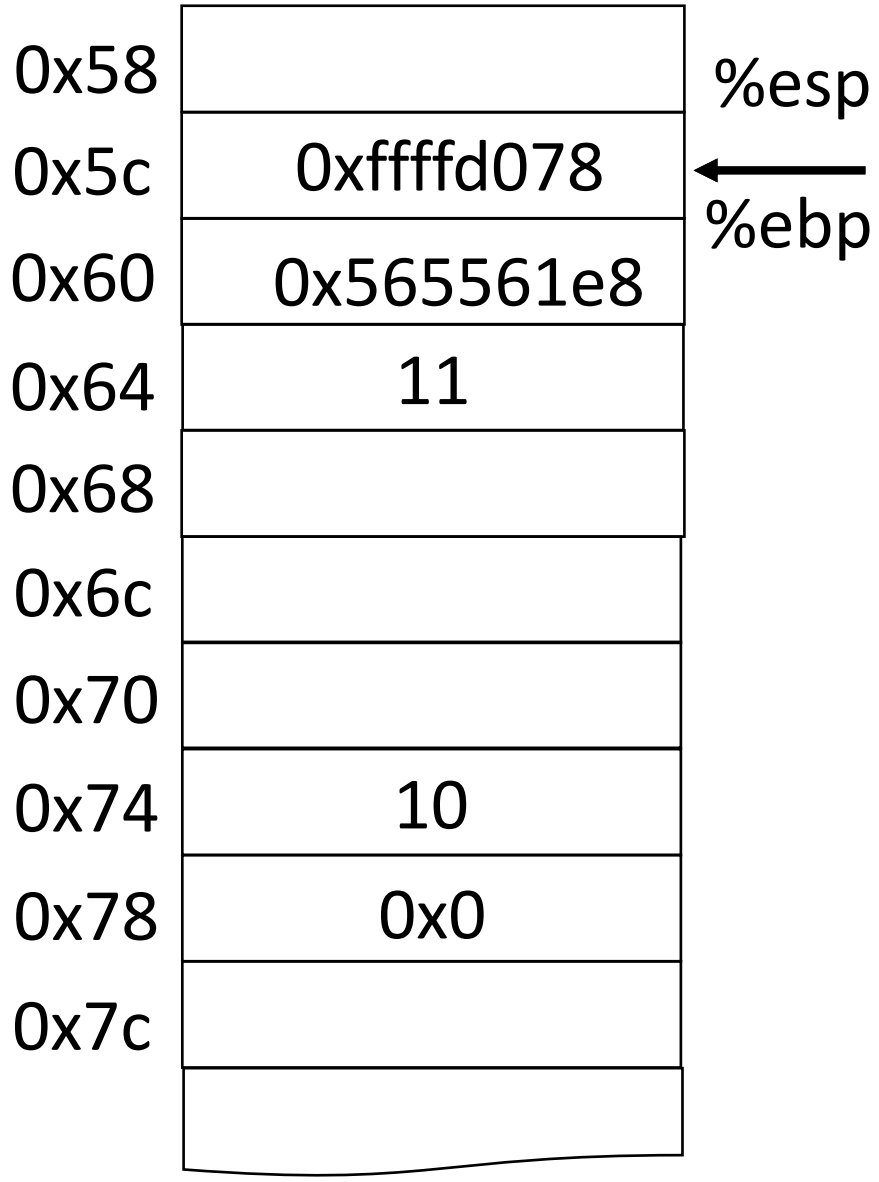
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd05c
%ebp	0xffffd05c
%eip	0x565561b2



Вызов функции

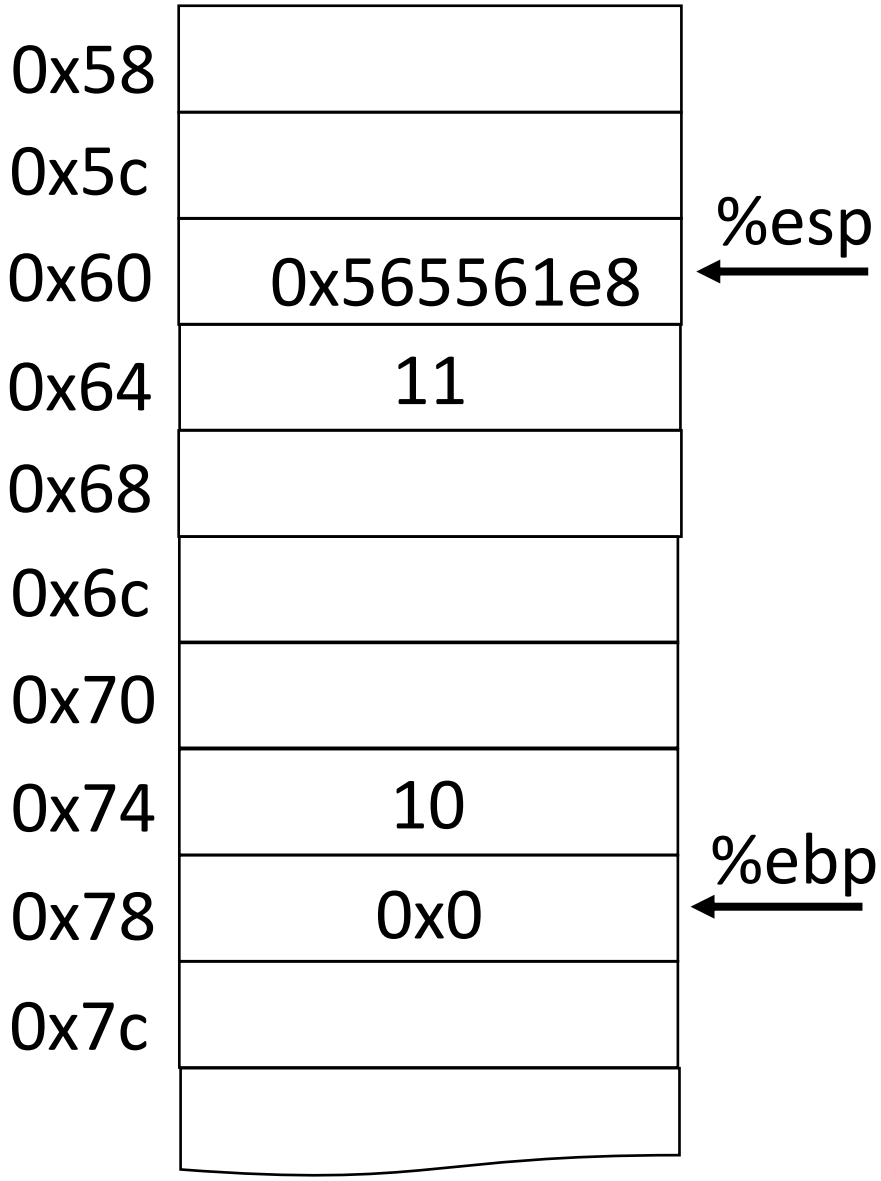
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd060
%ebp	0xffffd078
%eip	0x565561b2



Вызов функции

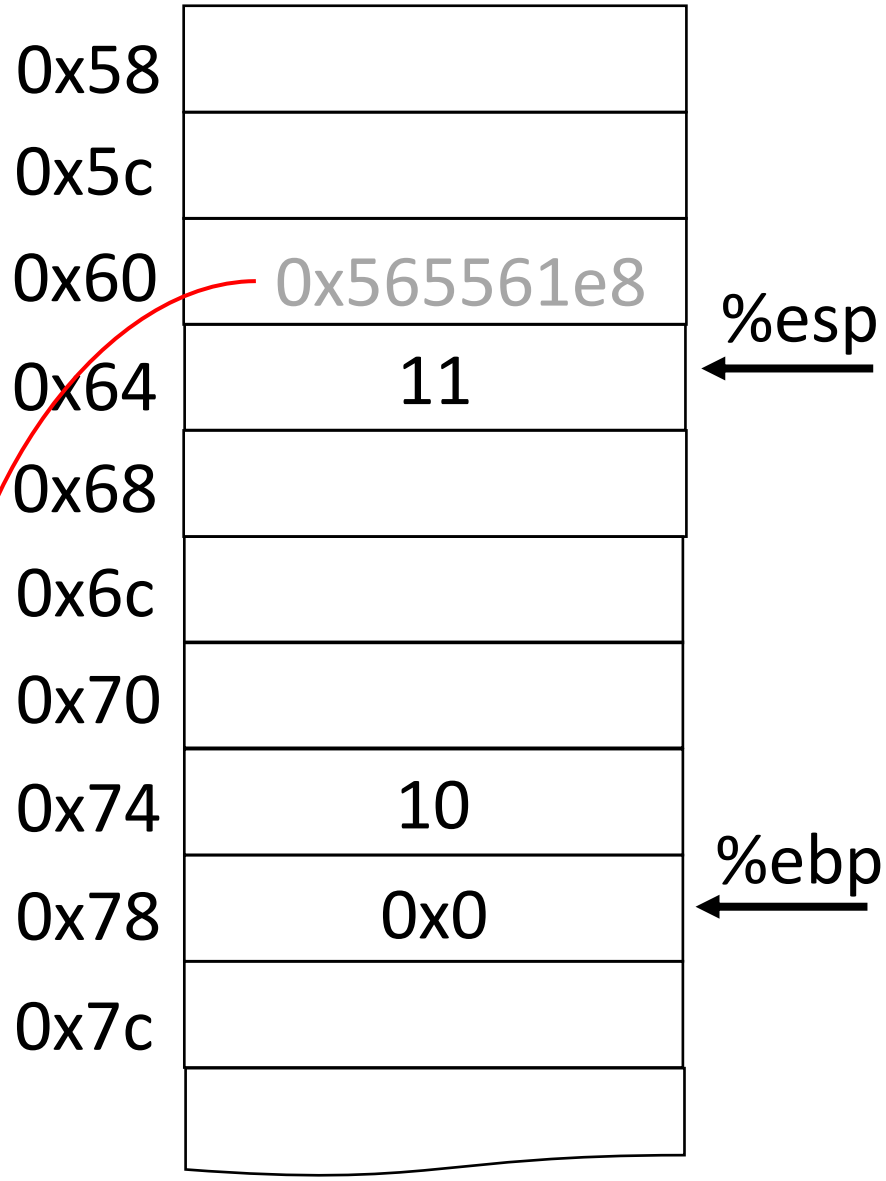
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd064
%ebp	0xffffd078
%eip	0x565561e8



Вызов функции

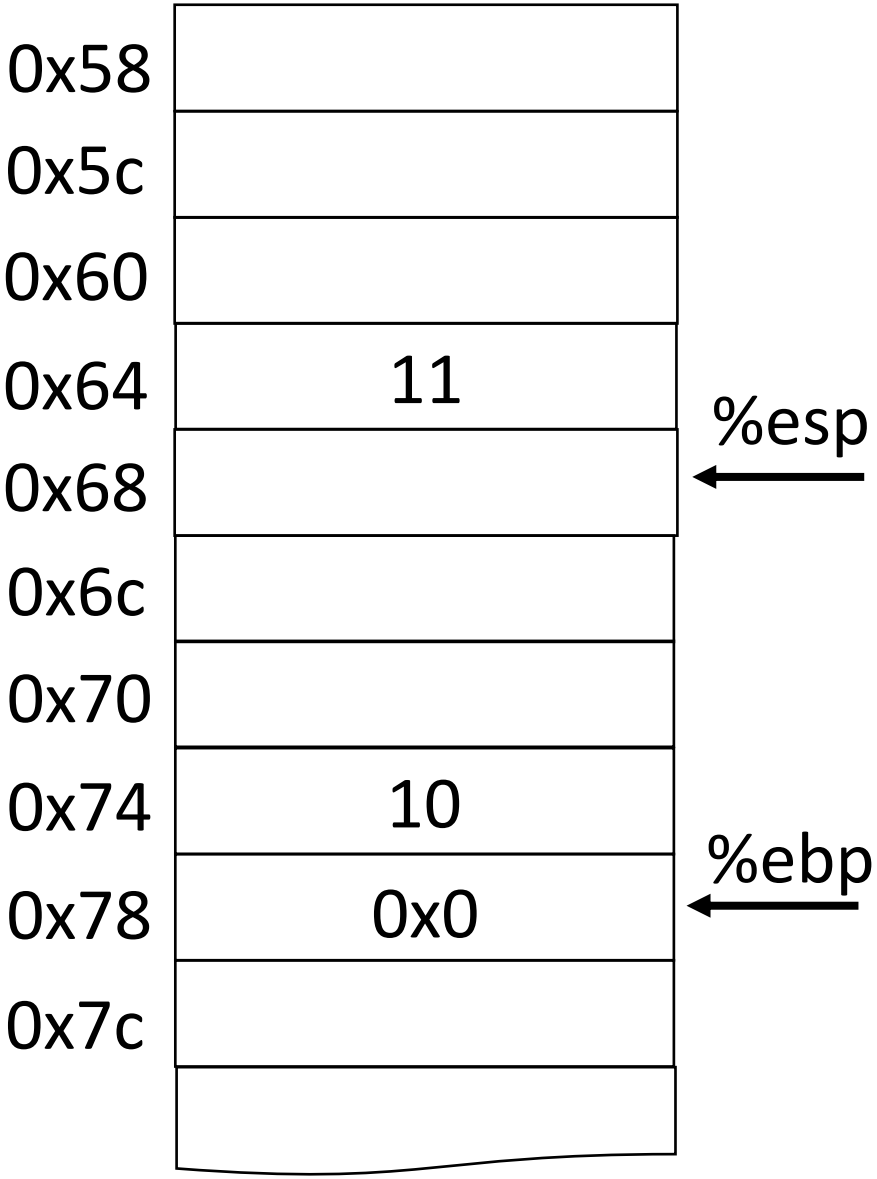
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd068
%ebp	0xffffd078
%eip	0x565561e8



Вызов функции

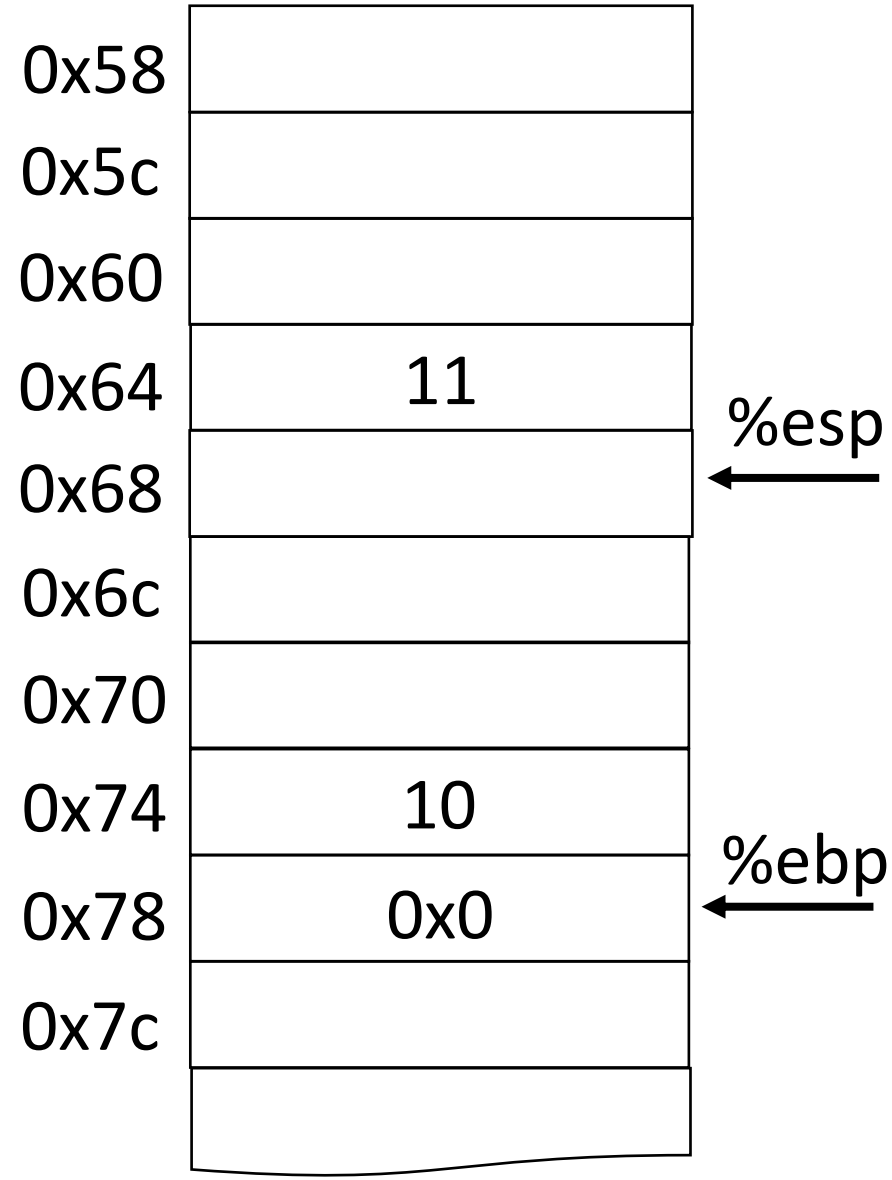
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd068
%ebp	0xffffd078
%eip	0x565561eb



Вызов функции

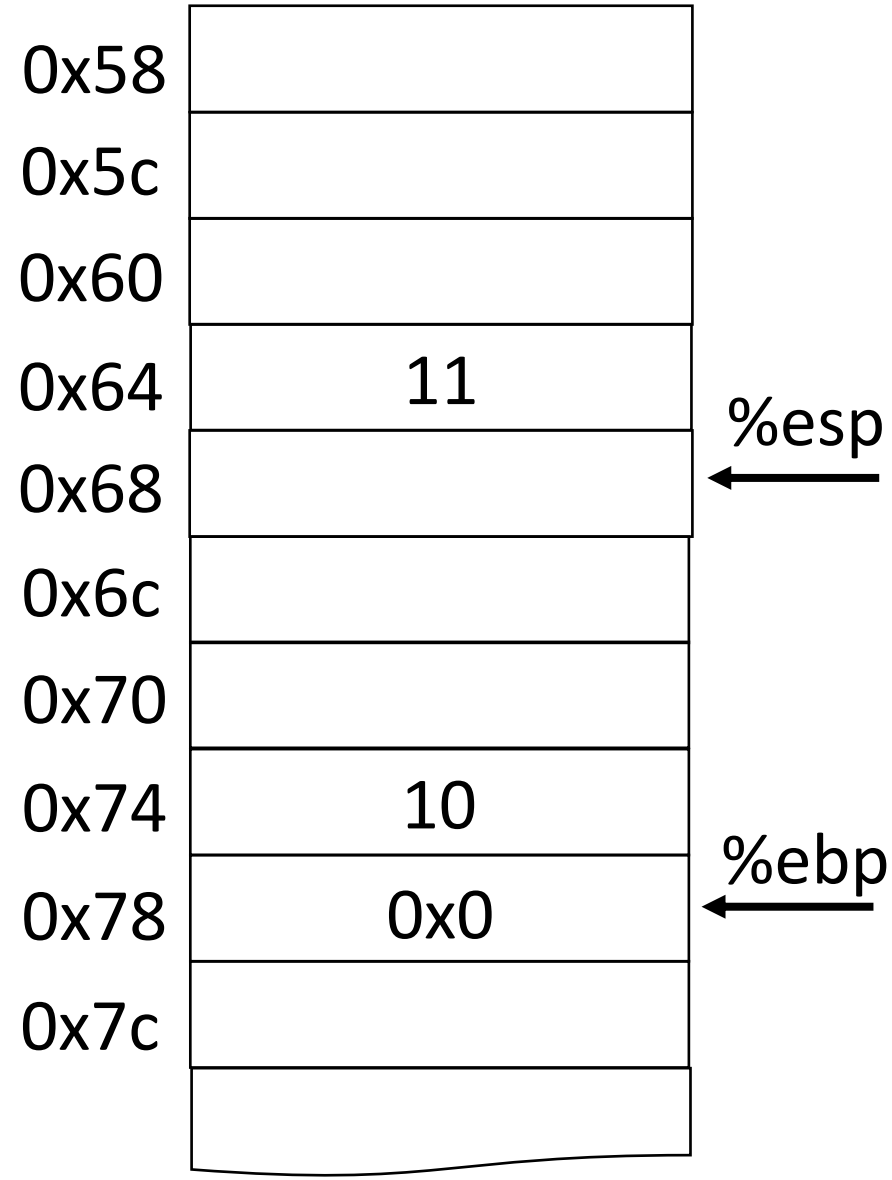
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd068
%ebp	0xffffd078
%eip	0x565561eb



Вызов функции

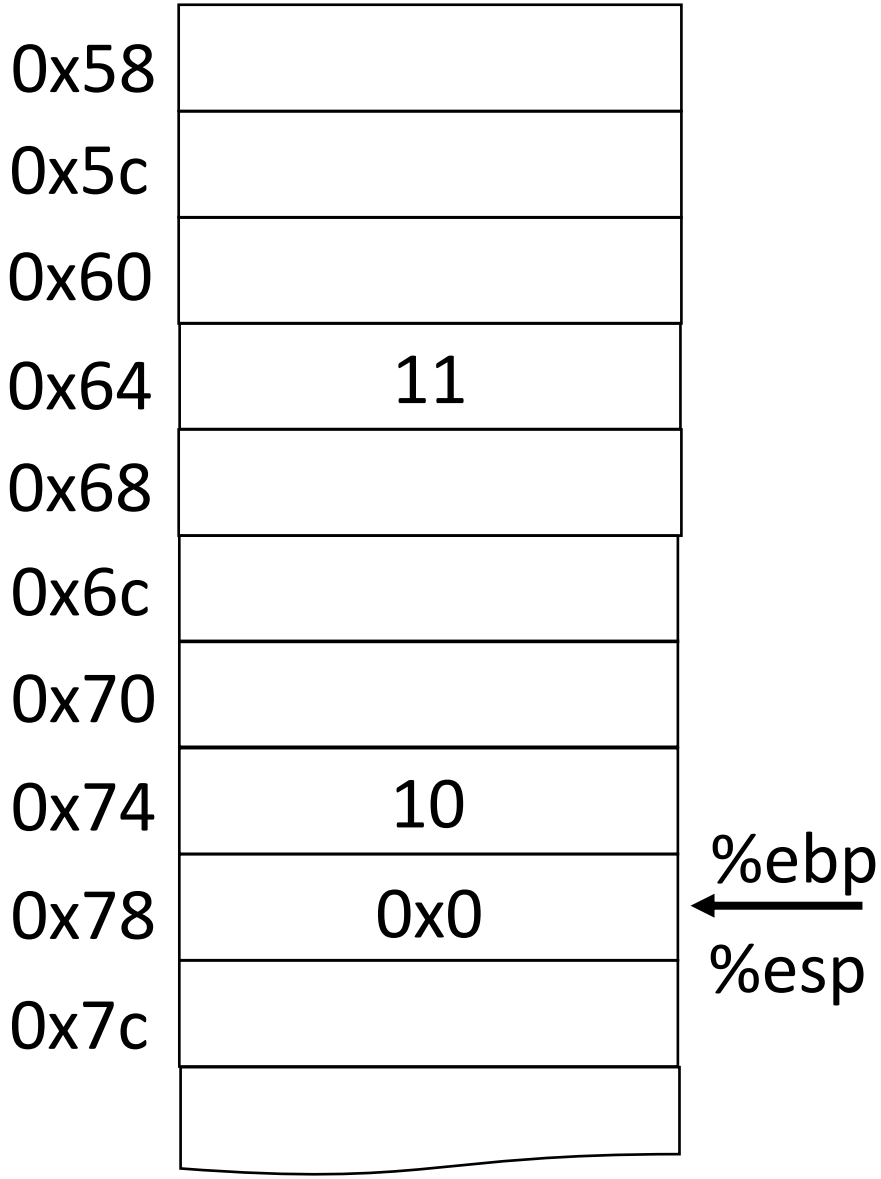
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        mov %ebp, %esp
        pop %ebp
        ret
```

%esp	0xffffd078
%ebp	0xffffd078
%eip	0x565561eb



Вызов функции

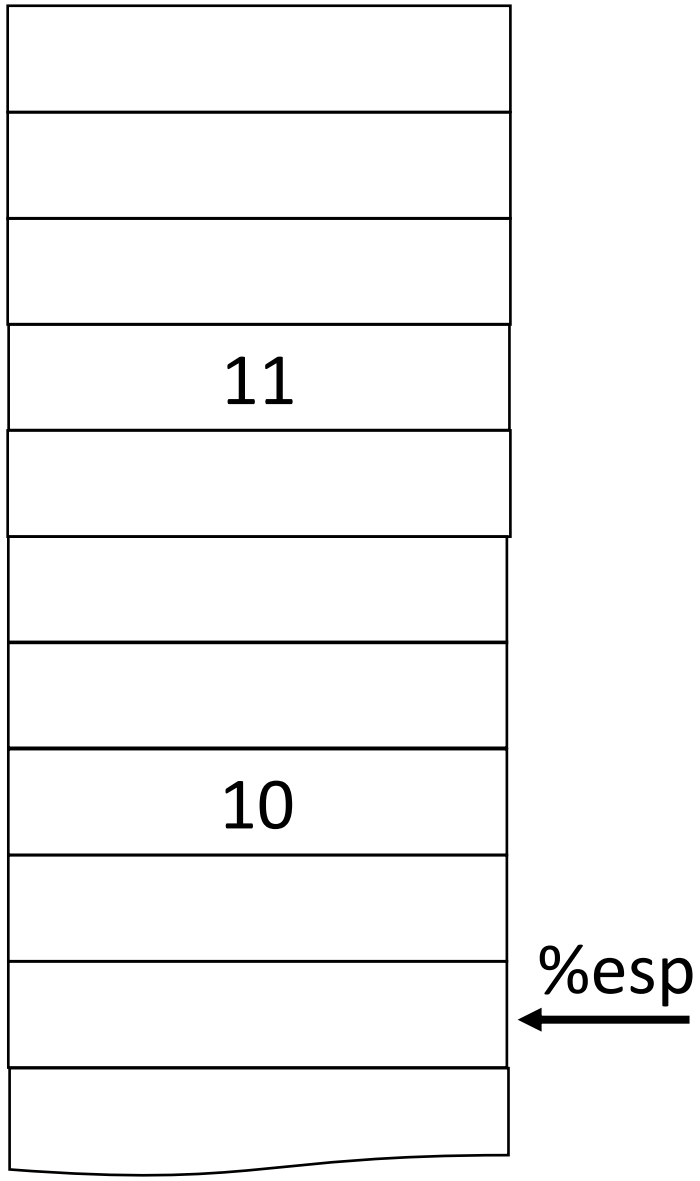
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        mov %ebp, %esp
        pop %ebp
        ret
```

%esp	0xffffd07c
%ebp	0x0
%eip	0x565561eb



Вызов функции

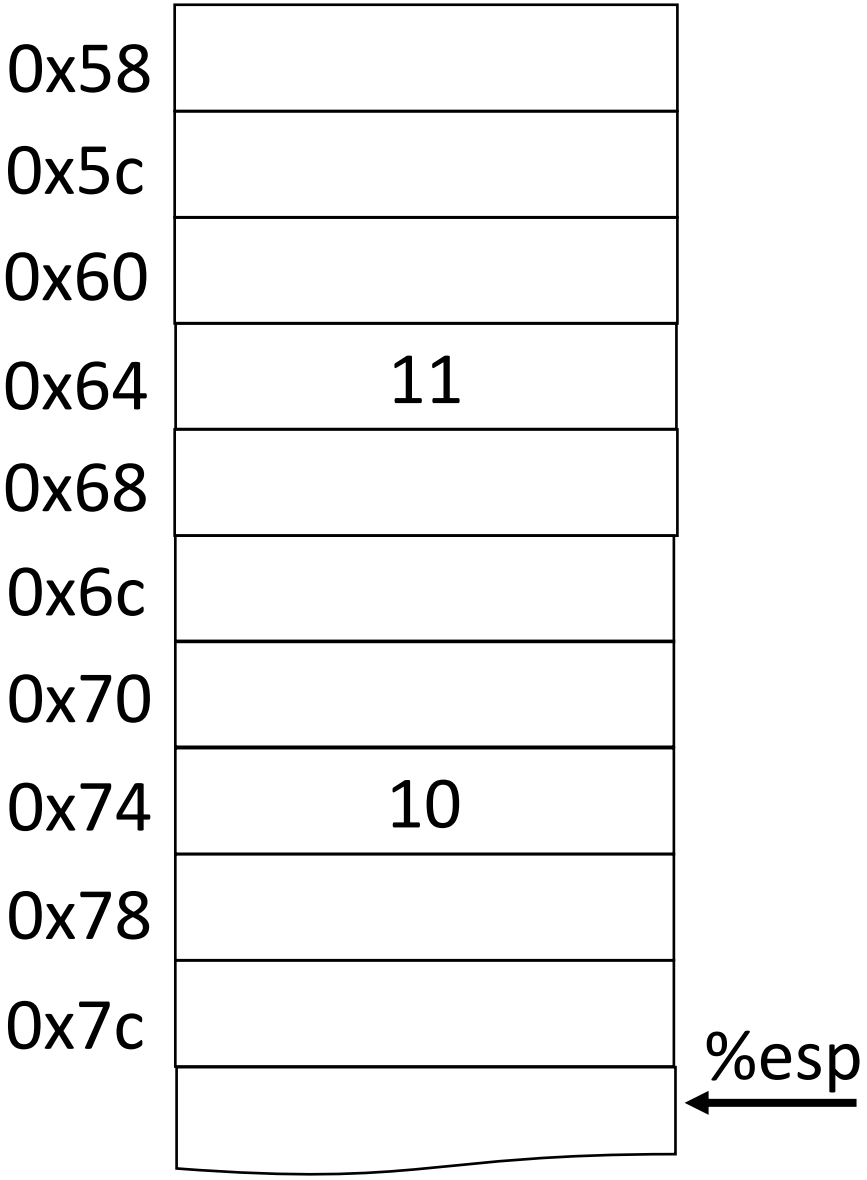
```c
#include <stdio.h>

void func(int x)
{
    x++;
}

int main()
{
int x = 10;
func(x);
return 0;
}
```

```
func:
        pushl   %ebp
        movl    %esp, %ebp
        addl    $1, 8(%ebp)
        popl    %ebp
        ret
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $10, -4(%ebp)
        pushl   -4(%ebp)
        call    func
        addl    $4, %esp
        movl    $0, %eax
        leave
        ret
```

%esp	0xffffd080
%ebp	0x0
%eip	0xf7de6ee5



Еще один небольшой пример

```c
#include <stdio.h>
int main(void)
{
  char x[2];
  scanf("%s", x);
  return 0;
}
```

Алгоритм действий

1. Проверяем порядок следования байт

```
readelf secret -a
```

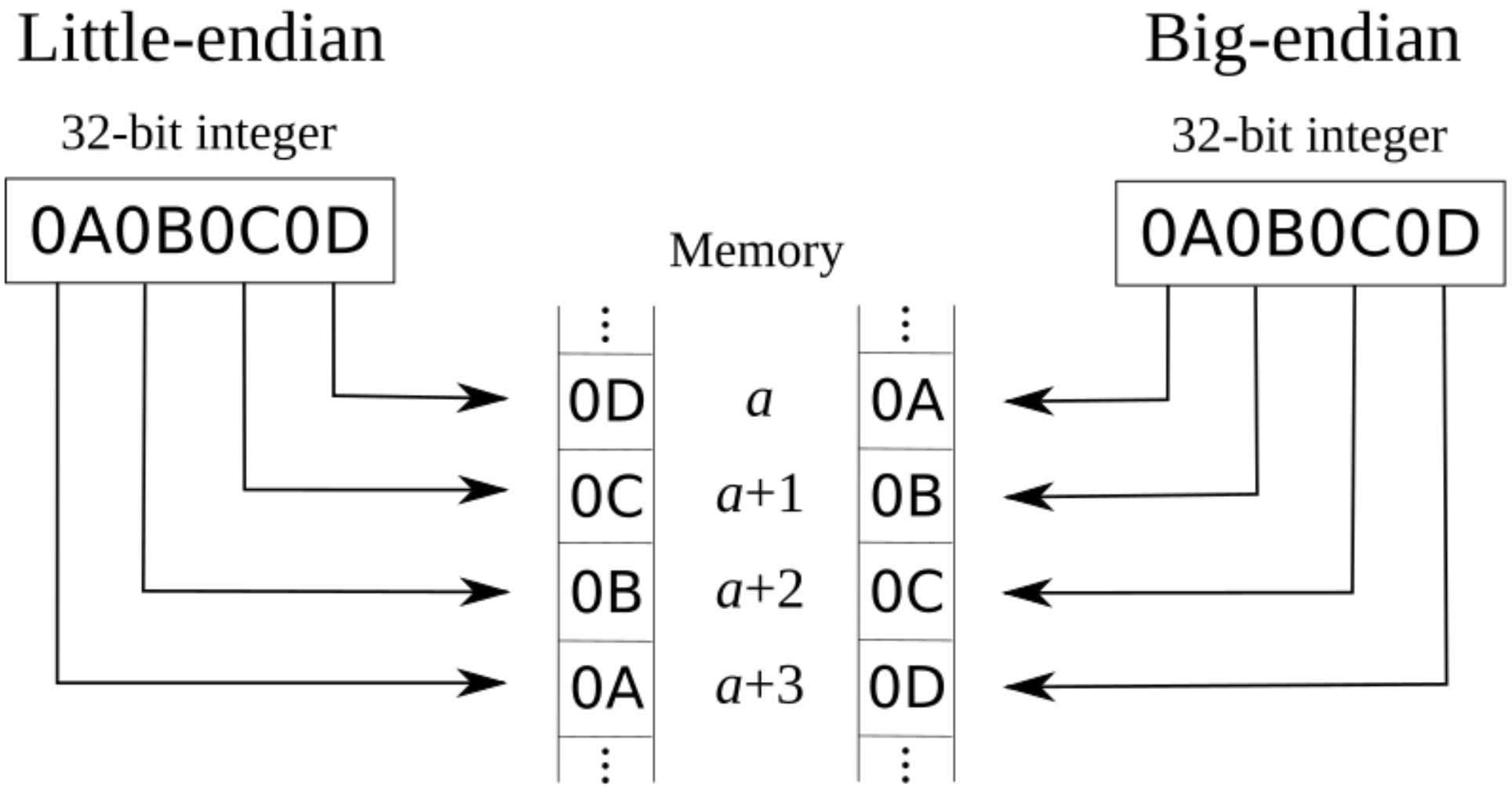
2. Анализируем ассемблерный код

```
objdump secret -d
```

3. Анализируем запущенную программу

```
gdb secret
```

Напоминание



Попытка взлома

```
#include <stdio.h>

char ebuff[] =
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /*first 10 bytes of junk*/
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /*next 10 bytes of junk*/
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /*following 10 bytes of junk*/
"\x31\x32\x33\x34\x35\x36\x37\x38\x39\x30" /*last 10 bytes of junk*/
"\xda\x06\x40\x00\x00\x00\x00\x00" /*address of endGame (little endian)*/
;

int main(void) {
    int i;
    for (i = 0; i < sizeof(ebuff); i++) { /*print each character*/
        printf("%c", ebuff[i]);
    }
    return 0;
}
```

Попытка взлома

```
gcc -o exp exp.c
./exp > exploit
./secret < exploit
 echo $?
```

Мораль!

Instead of:	Use:
`gets(buf)`	`fgets(buf, 12, stdin)`
`scanf("%s", buf)`	`scanf("%12s", buf)`
`strcpy(buf2, buf)`	`strncpy(buf2, buf, 12)`
`strcat(buf2, buf)`	`strncat(buf2, buf, 12)`
`sprintf(buf, "%d", num)`	`snprintf(buf, 12, "%d", num)`

Спасибо за внимание!