Sécurité des logiciels

Assembly language, part 4/4

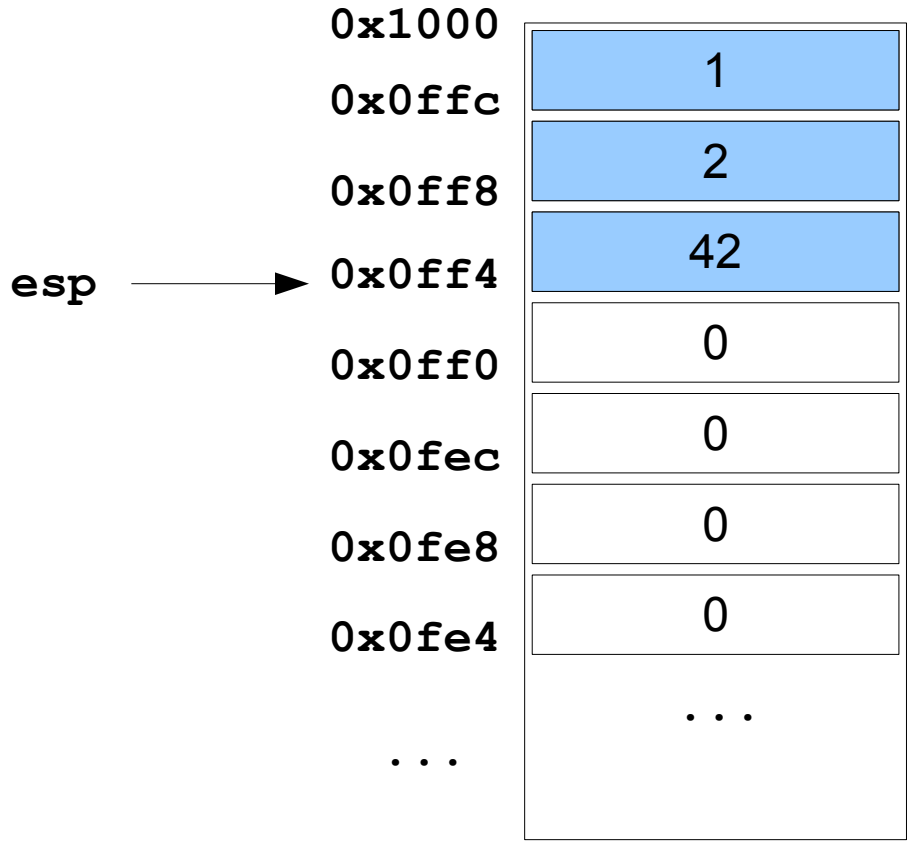
Samuel Thibault <samuel.thibault@u-bordeaux.fr>
Pieces from Emmanuel Fleury <emmanuel.fleury@u-bordeaux.fr>

Calling a function

Calling a function

Call instruction

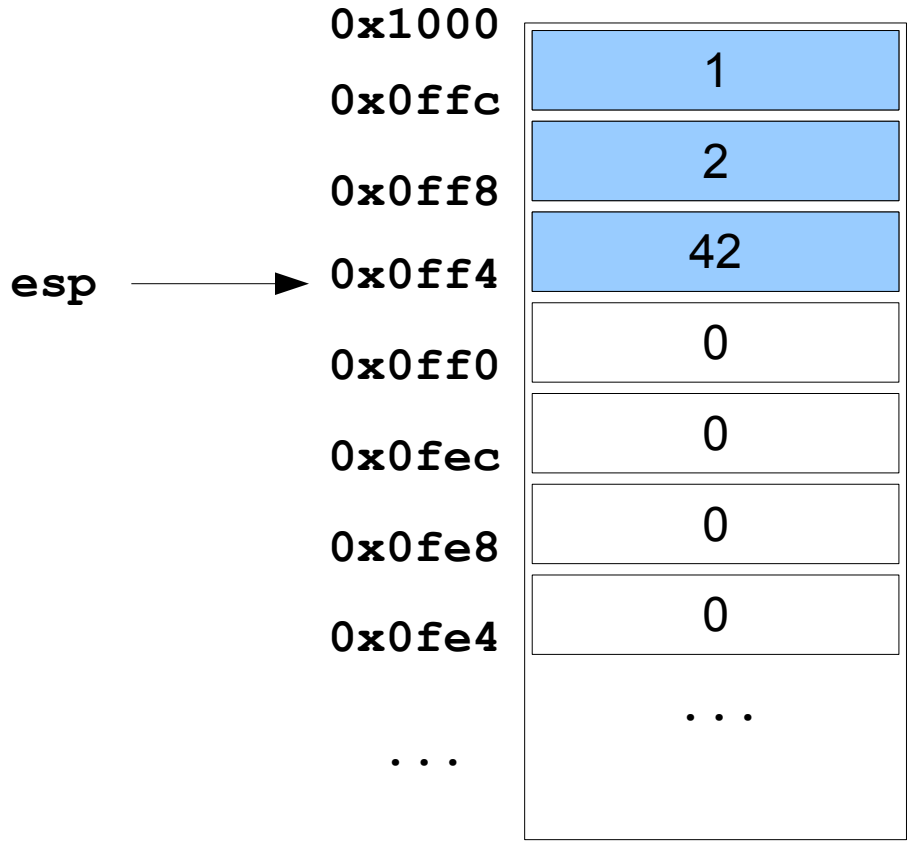
```
call f
```



Calling a function

Call instruction

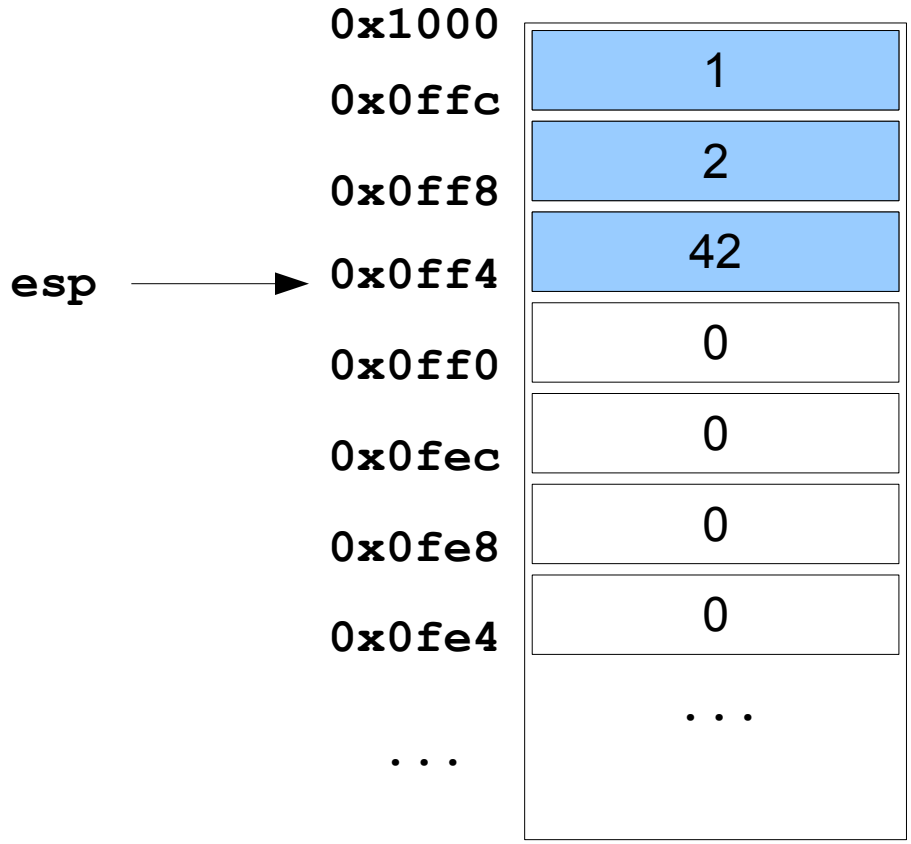
```
0x0810: call f
<=>
0x0810: pushl %eip+5
        jmp f
```



Calling a function

Call instruction

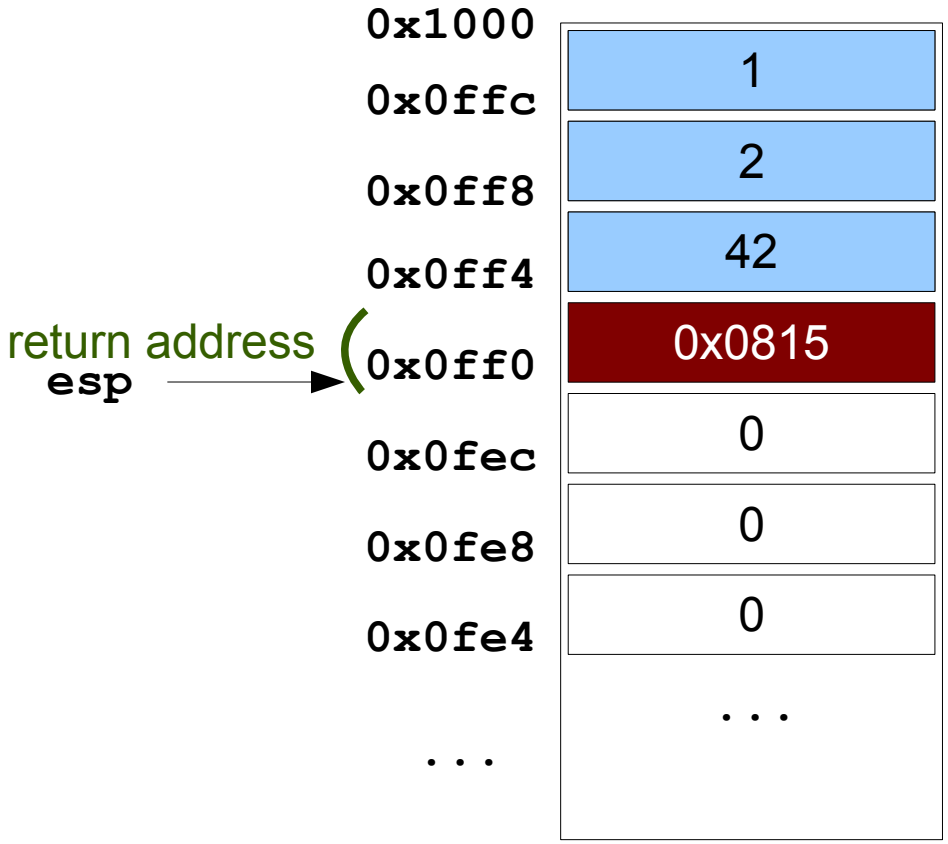
```
0x0810: call f
<=>
0x0810: pushl %eip+5
        jmp f
```



Calling a function

Call instruction

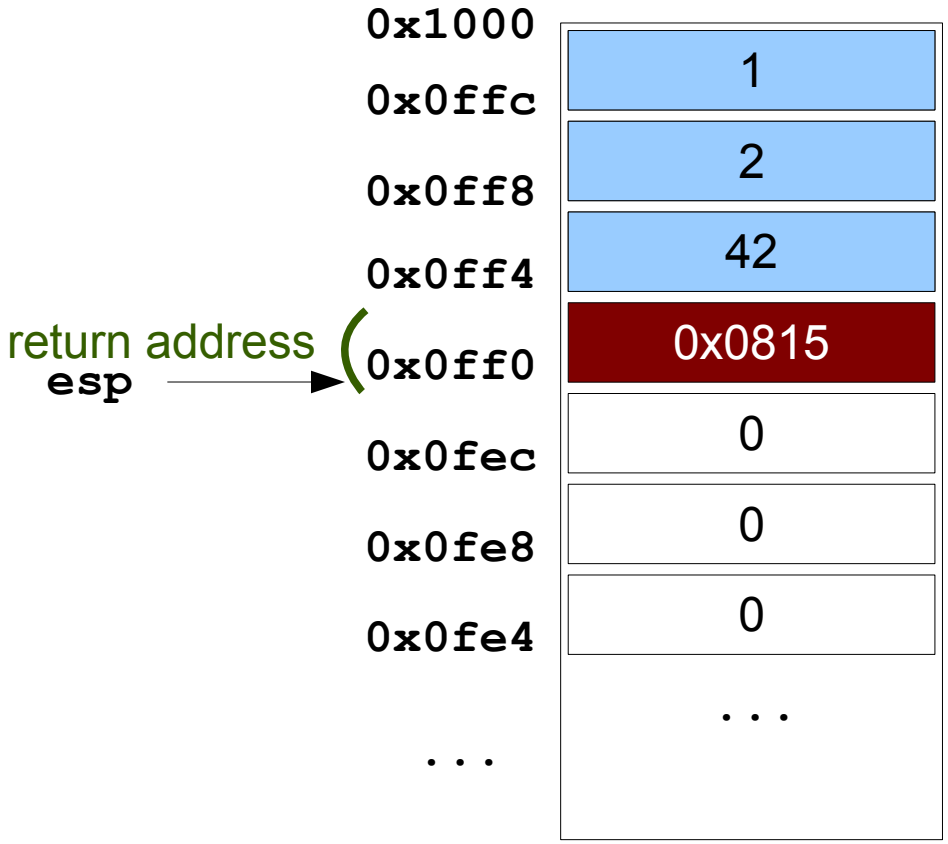
```
0x0810: call f
<=>
0x0810: pushl %eip+5
        jmp f
```



Calling a function

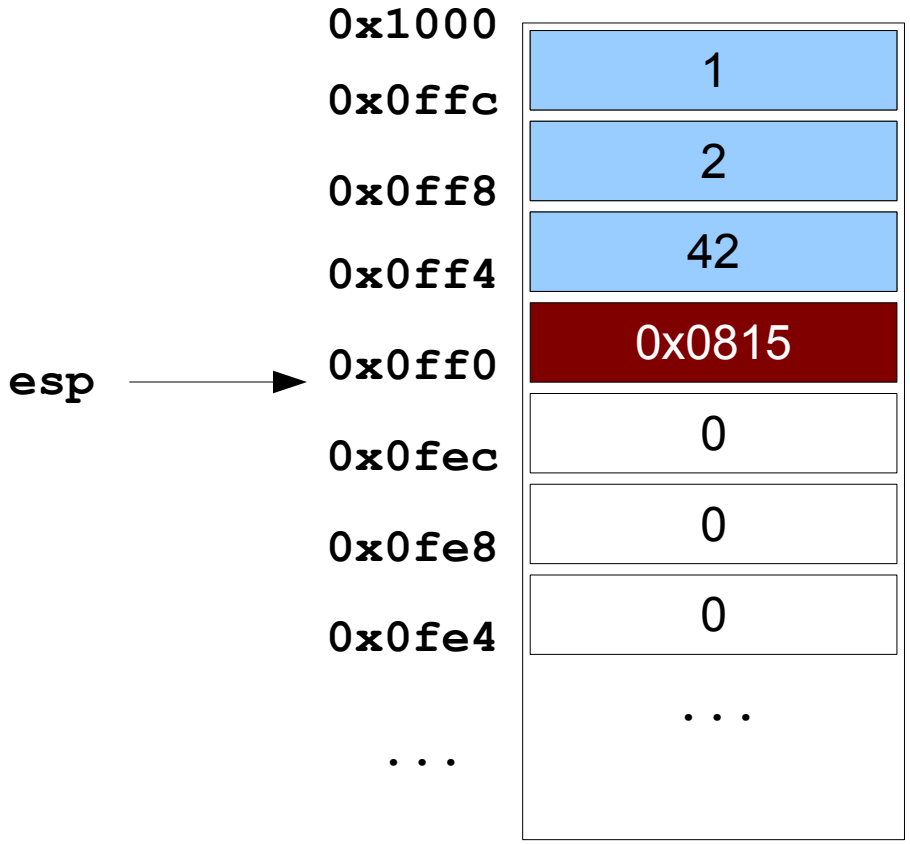
Call instruction

```
0x0810: call f
<=>
0x0810: pushl %eip+5
        jmp f
```



Inside a function

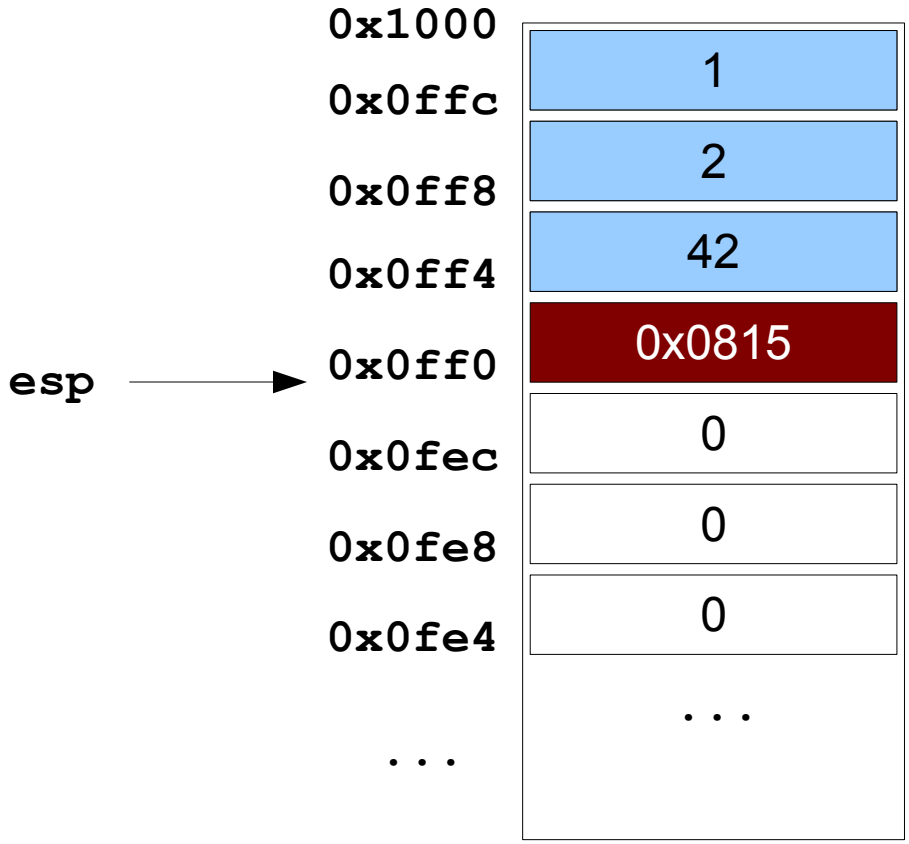
```
0x07a0: f:   movl $0, %eax
0x07a5:      ret
```



Returning from a function

Ret instruction

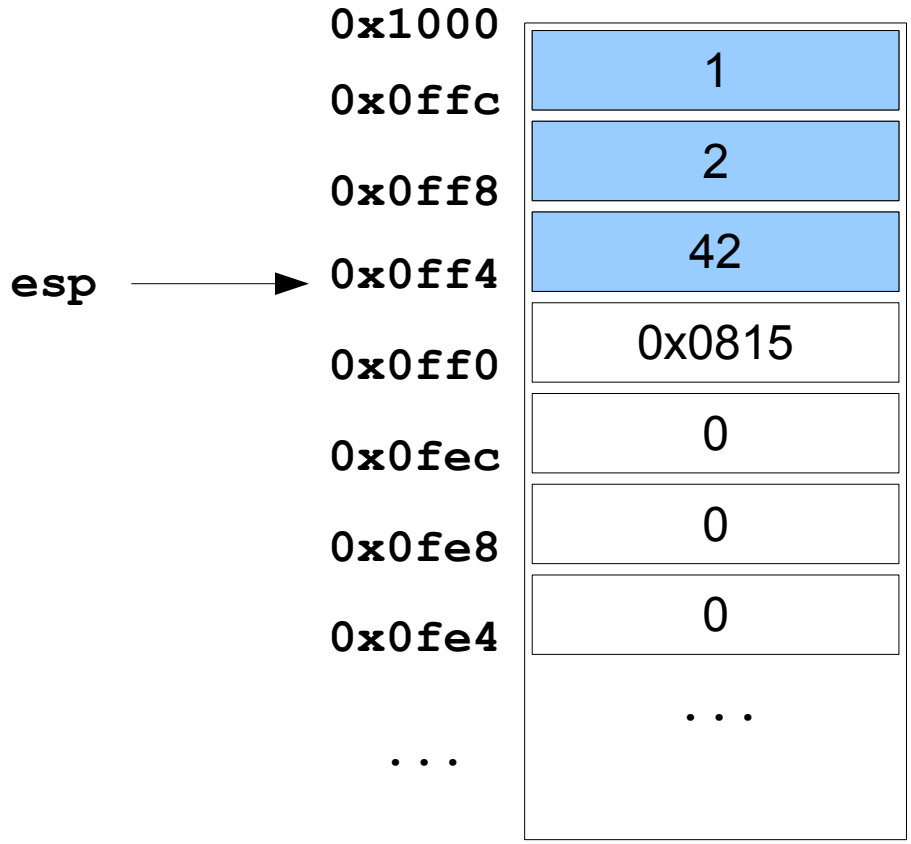
```
0x07a5: ret
<=>
0x07a5: popl %eip
<=>
0x07a5: jmpl (%esp)
        addl $4,%esp
```



Returning from a function

Ret instruction

```
0x07a5: ret
<=>
0x07a5: popl %eip
<=>
0x07a5: jmpl (%esp)
        addl $4,%esp
```



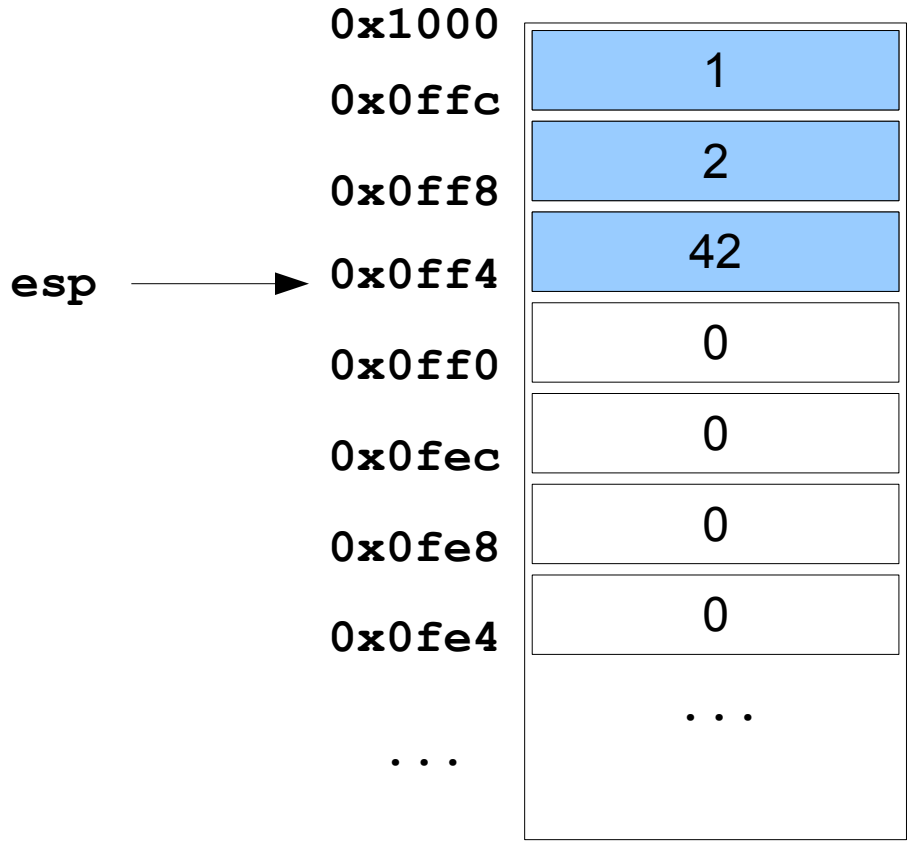
Passing parameters
(32bit version)

Calling a function

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
```

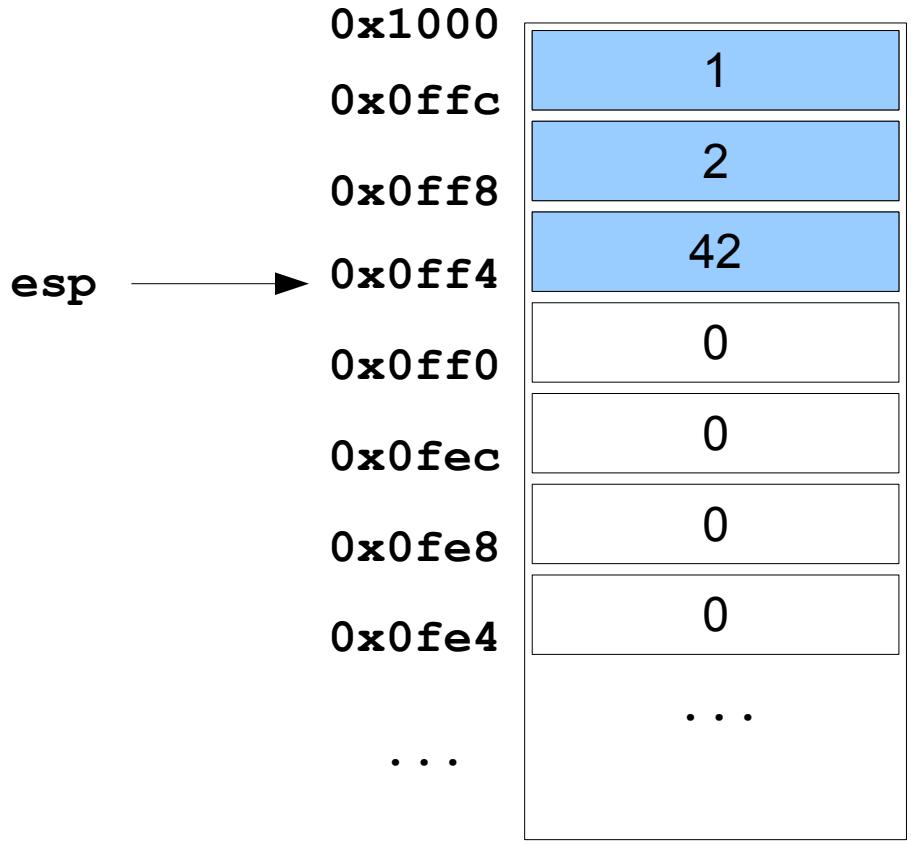


Calling a function

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
```

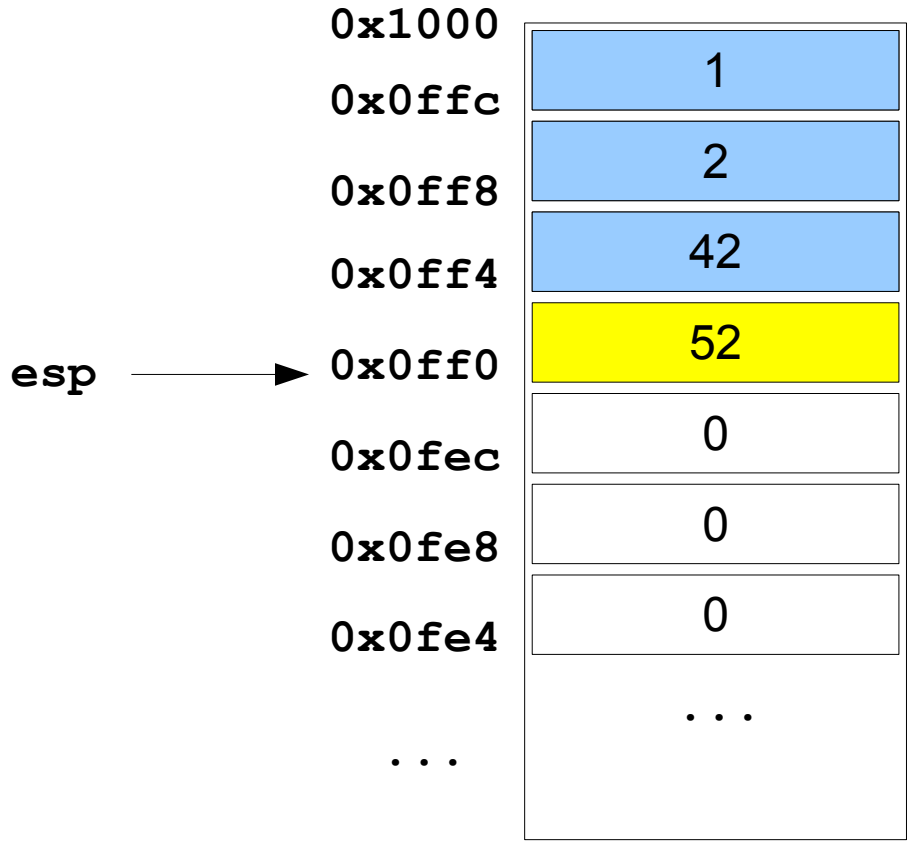


Calling a function

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
```

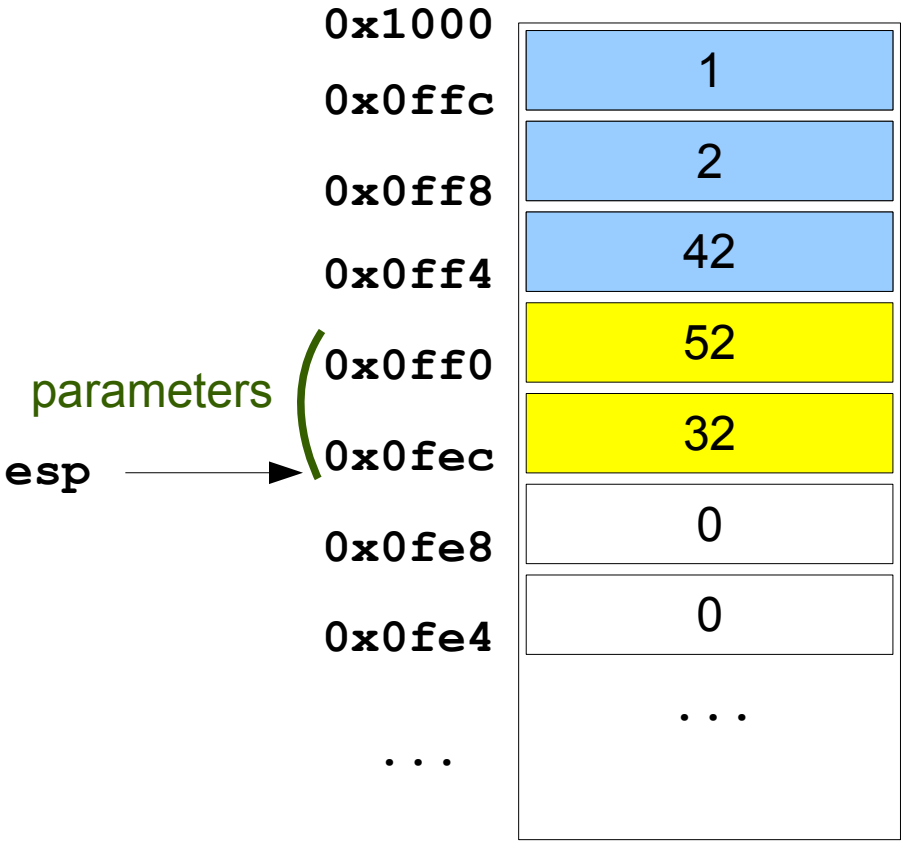


Calling a function

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
```



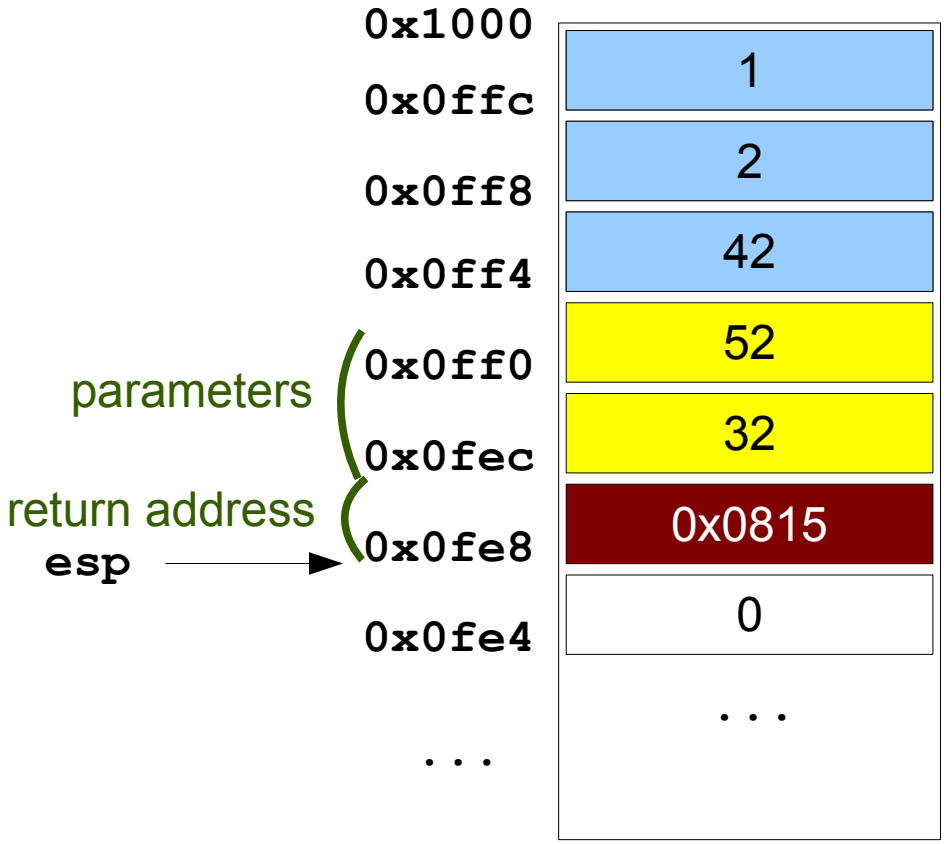
Calling a function

Call instruction

- x = f(32,52)

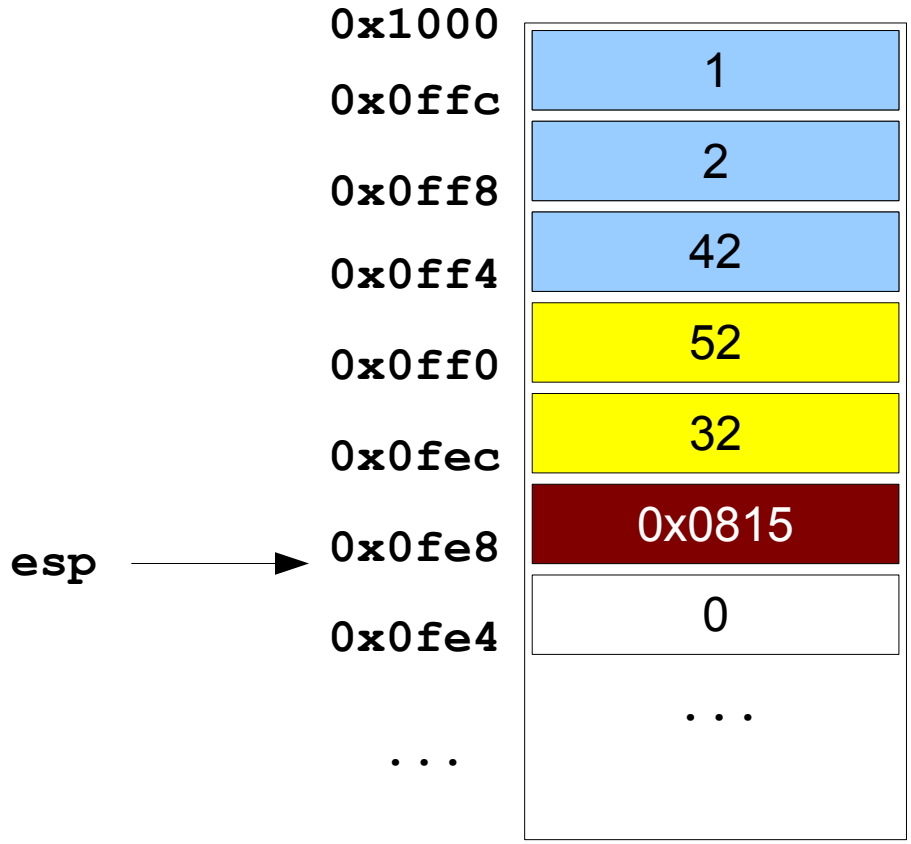
```
pushl $52
pushl $32
call f
```

I.e. push in reverse order



Inside the function

```
f: movl 4(%esp),%eax
   addl 8(%esp),%eax
   ret
```



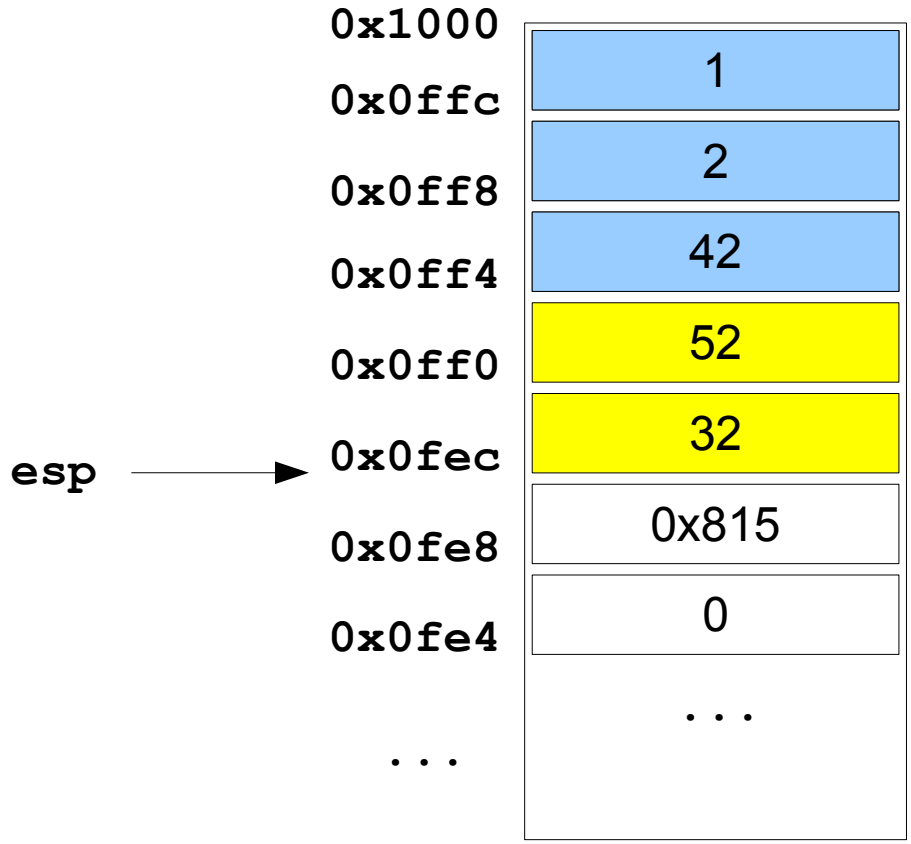
Back to the caller

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f

; result in %eax
```

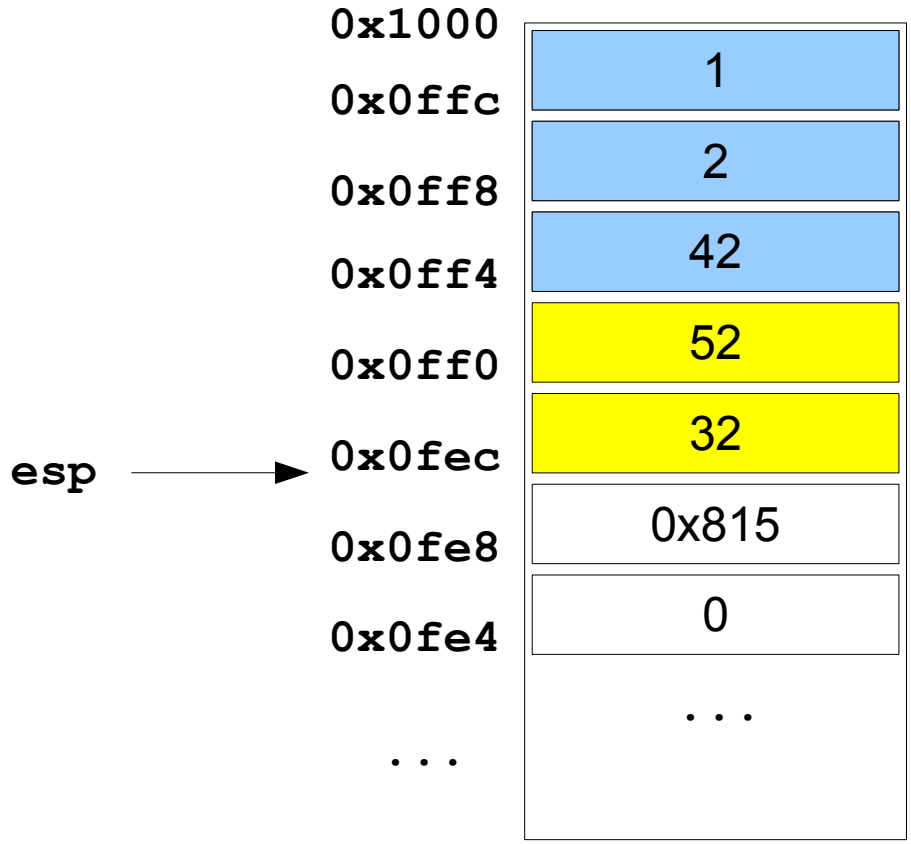


Back to the caller

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
addl $8, %esp
; result in %eax
```



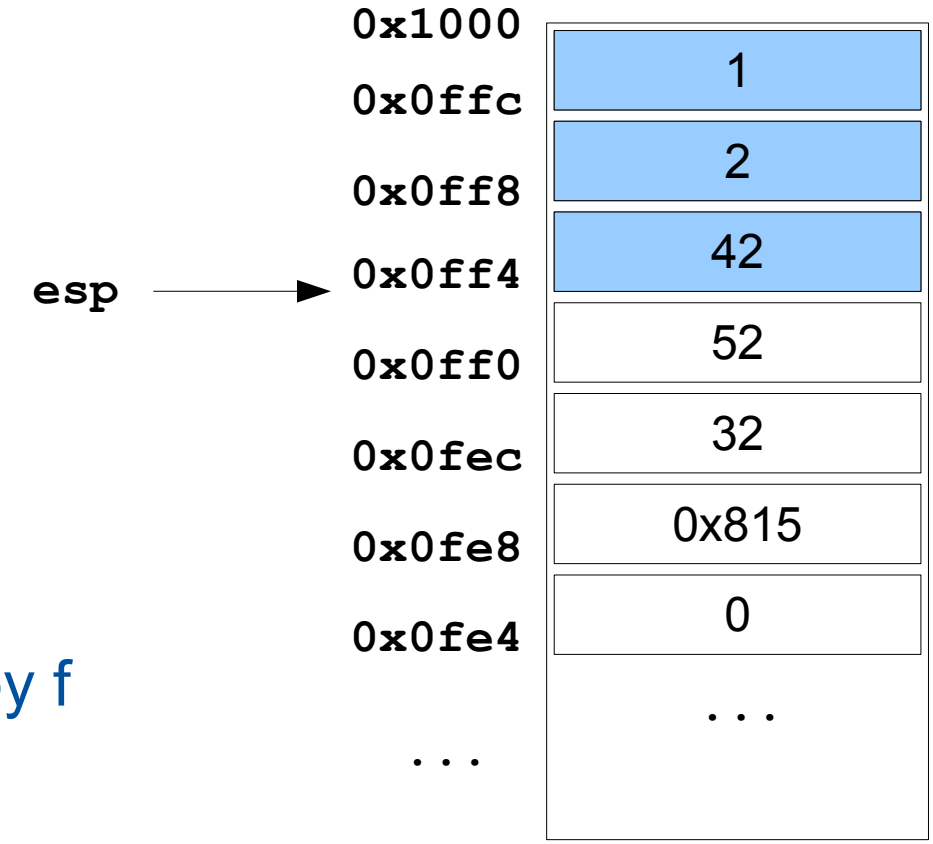
Back to the caller

Call instruction

- x = f(32,52)

```
pushl $52
pushl $32
call f
addl $8, %esp
; result in %eax
```

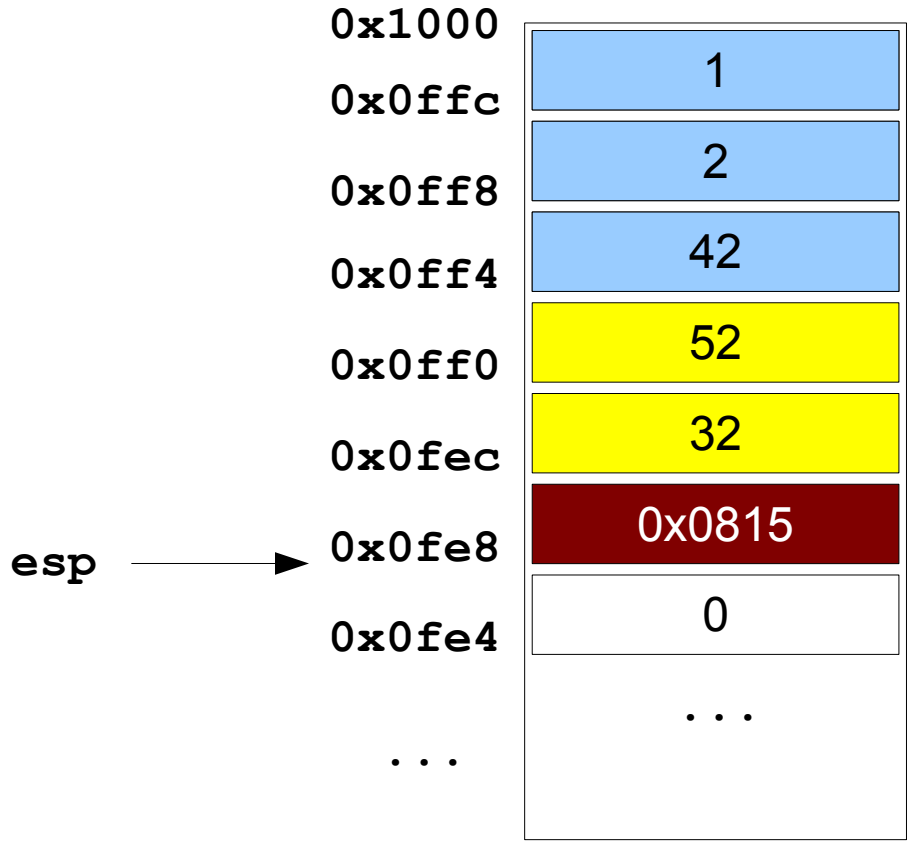
(On Windows this is to be done by f by using **`ret $8`**)



Local variables

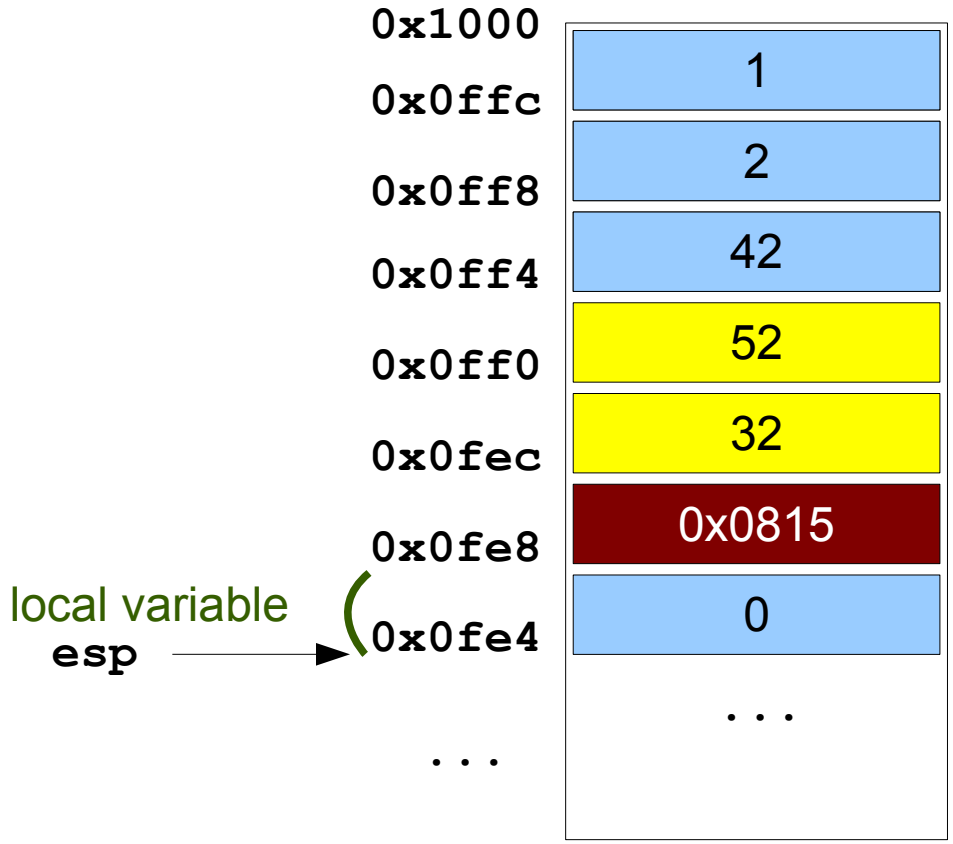
Inside the function

```
f:  subl $4,%esp
    movl 8(%esp),%eax
    addl 12(%esp),%eax
    movl %eax,(%esp)
    ...
    ret
```



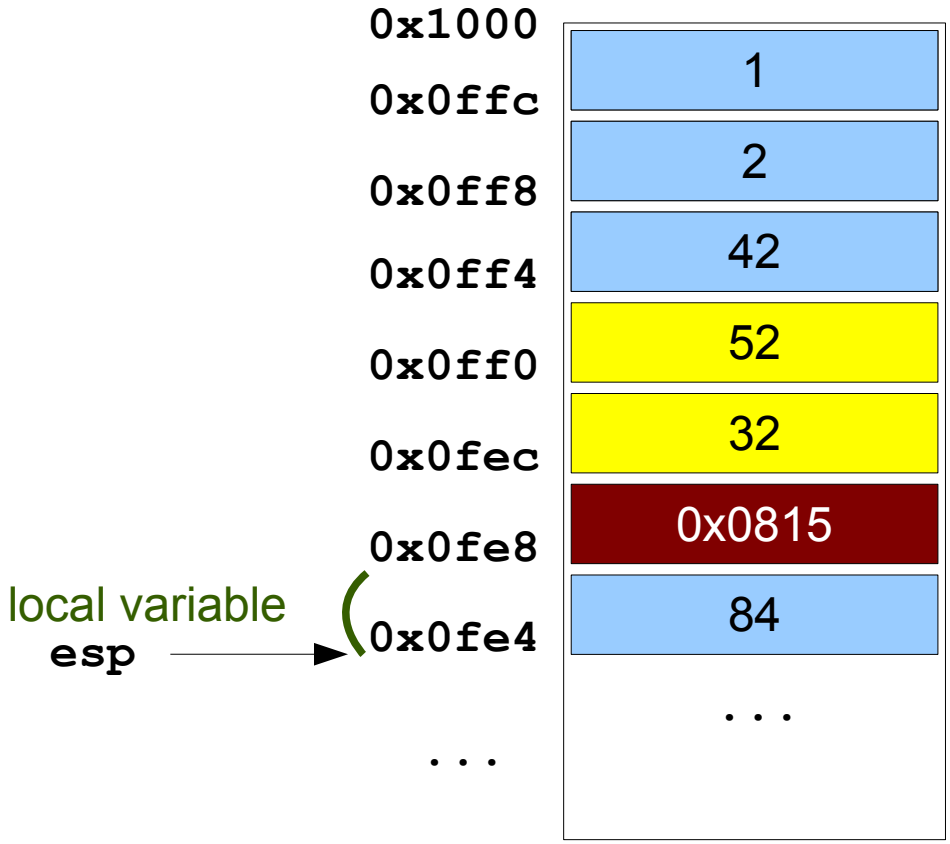
Inside the function

```
f:  subl $4,%esp
    movl 8(%esp),%eax
    addl 12(%esp),%eax
    movl %eax,(%esp)
    ...
    ret
```



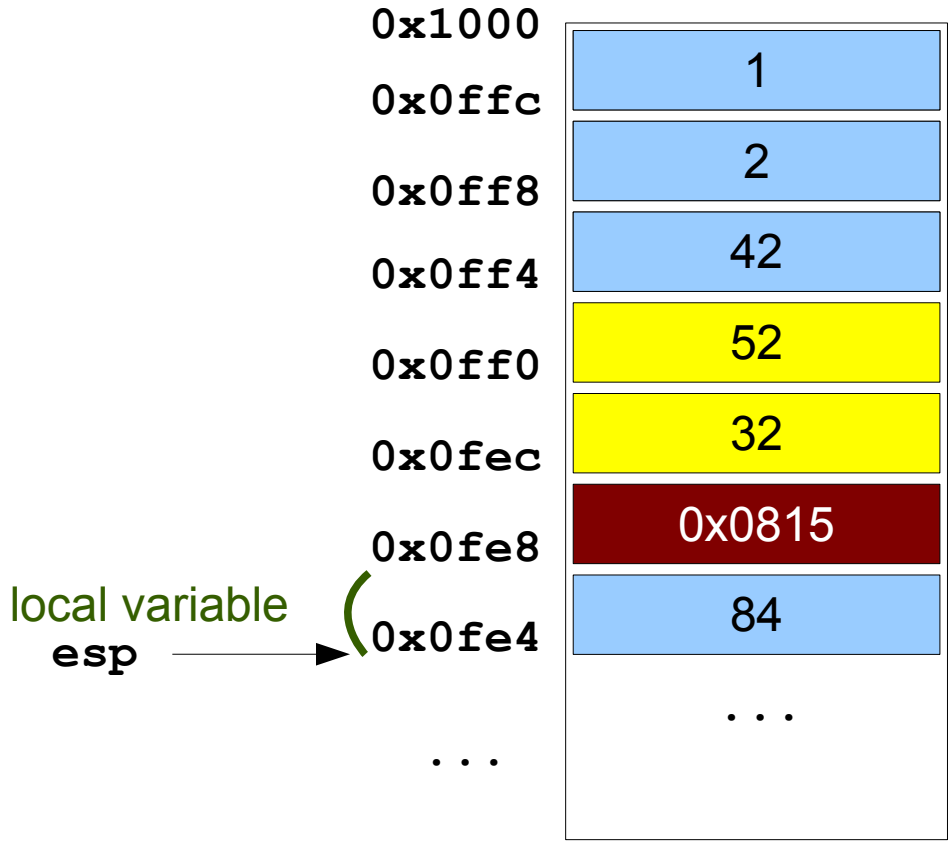
Inside the function

```
f:  subl $4,%esp
    movl 8(%esp),%eax
    addl 12(%esp),%eax
    movl %eax,(%esp)
    ...
    ret
```



Inside the function

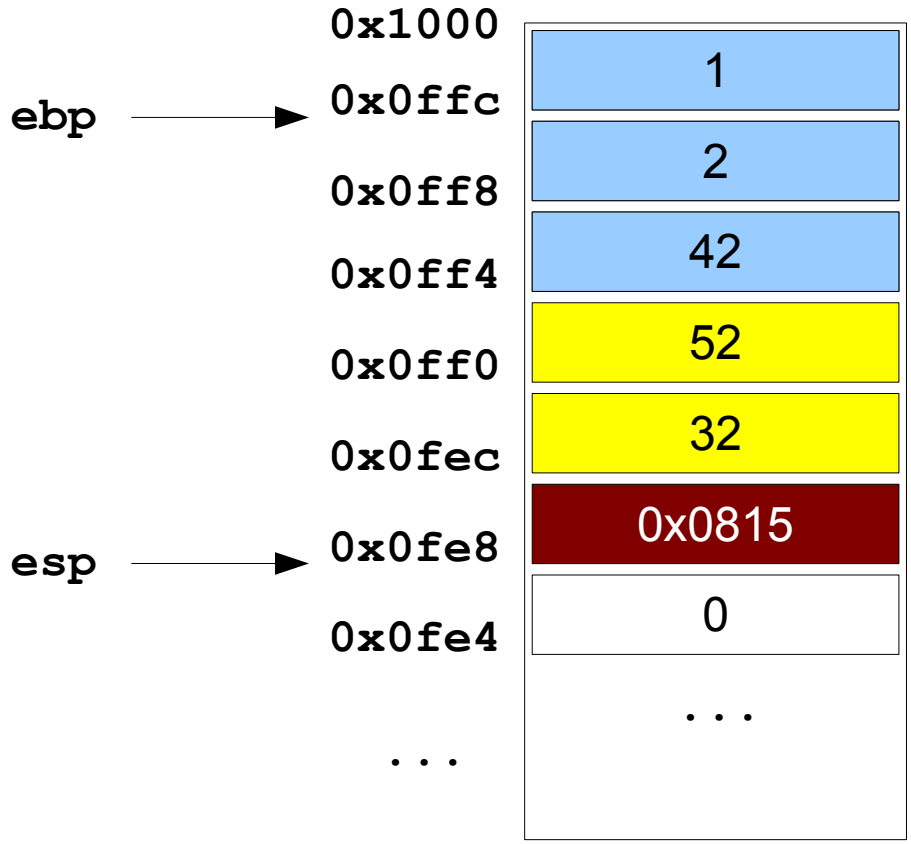
```
f:  subl $4,%esp
    movl 8(%esp),%eax
    addl 12(%esp),%eax
    movl %eax,(%esp)
    ...
    addl $4,%esp
    ret
```



Base Pointer (bp)

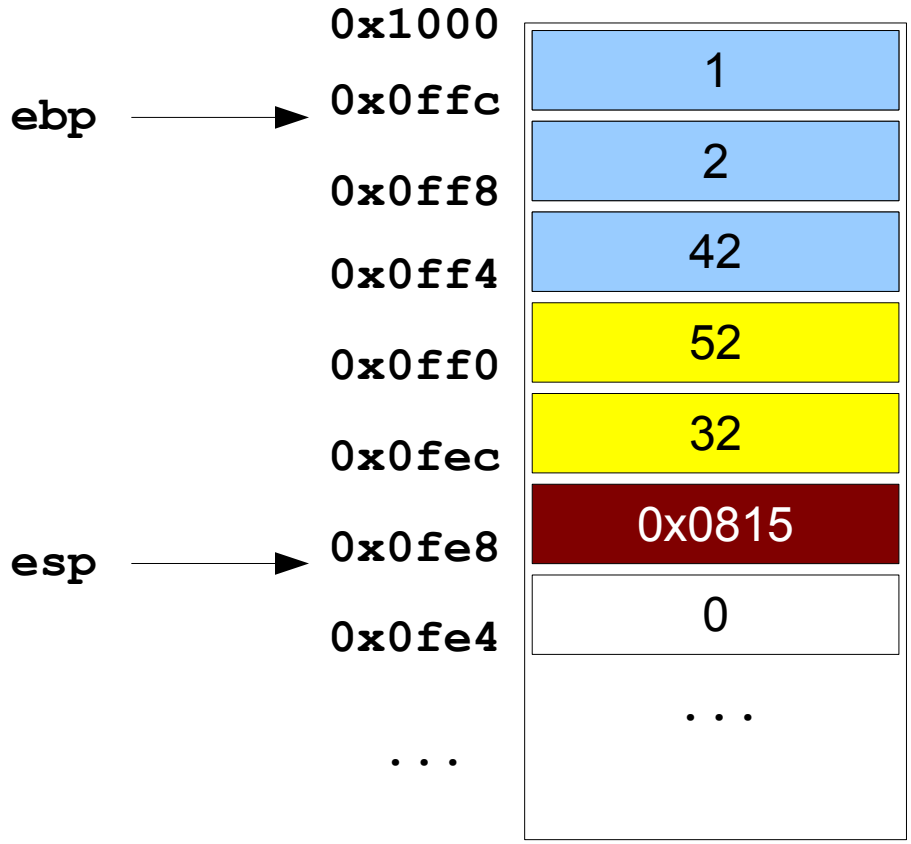
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   ...
```



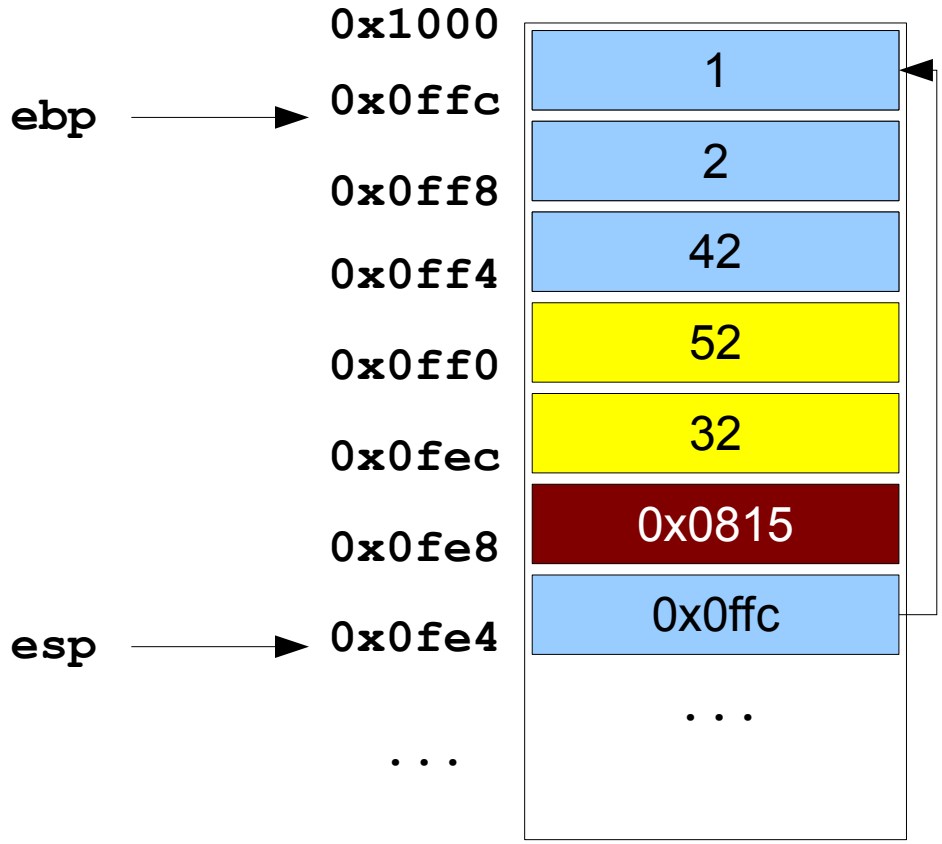
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   ...
```



Inside the function

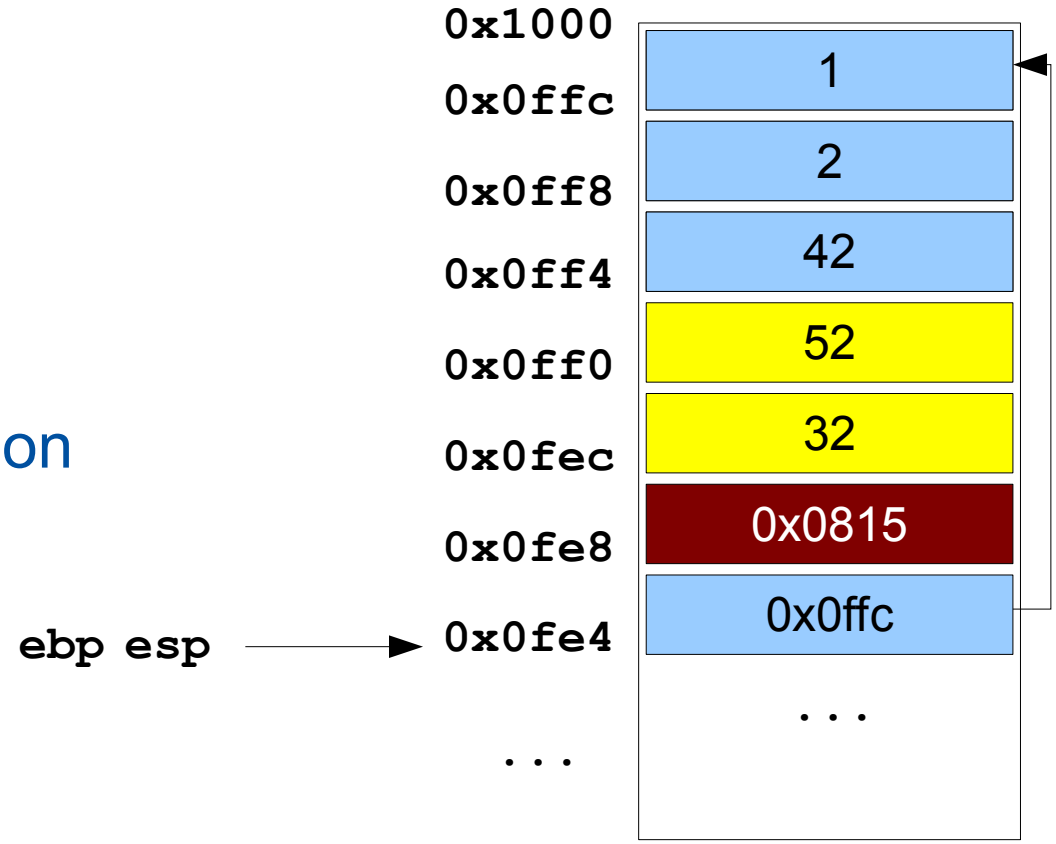
```
f: pushl %ebp
   movl %esp,%ebp
   ...
```



Inside the function

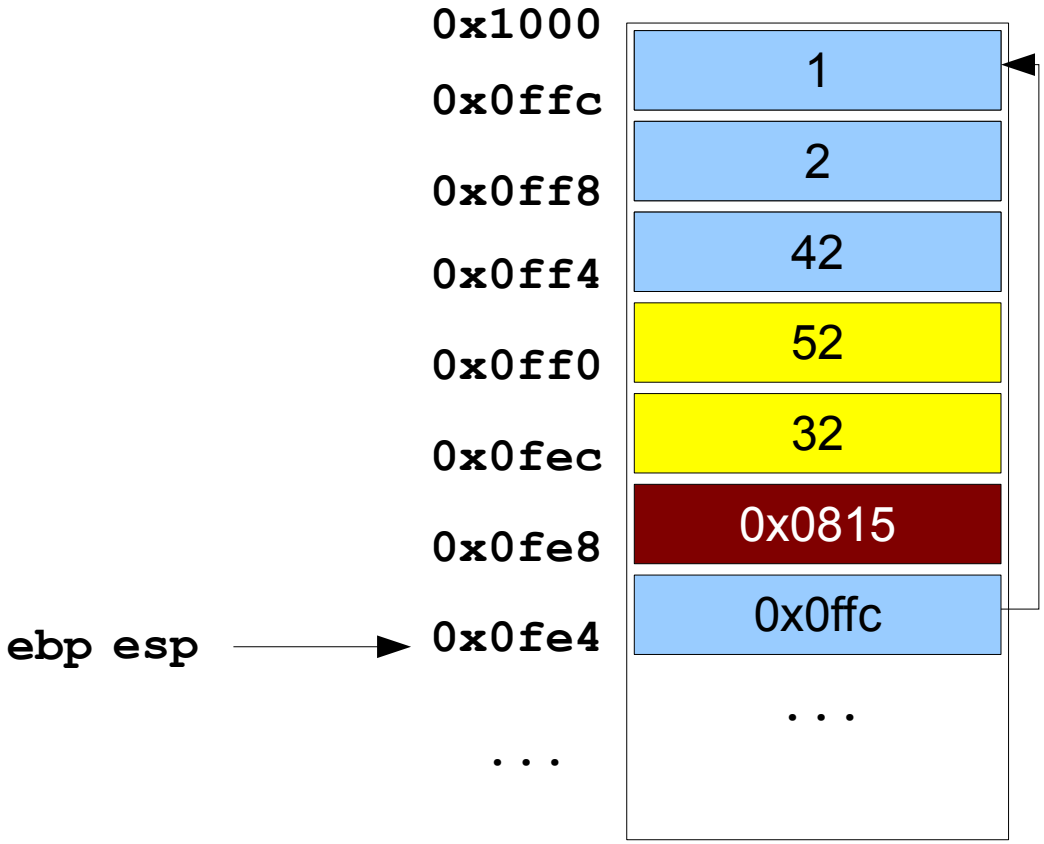
```
f: pushl %ebp
   movl %esp,%ebp
   ...
```

Also available as one-instruction **enter**



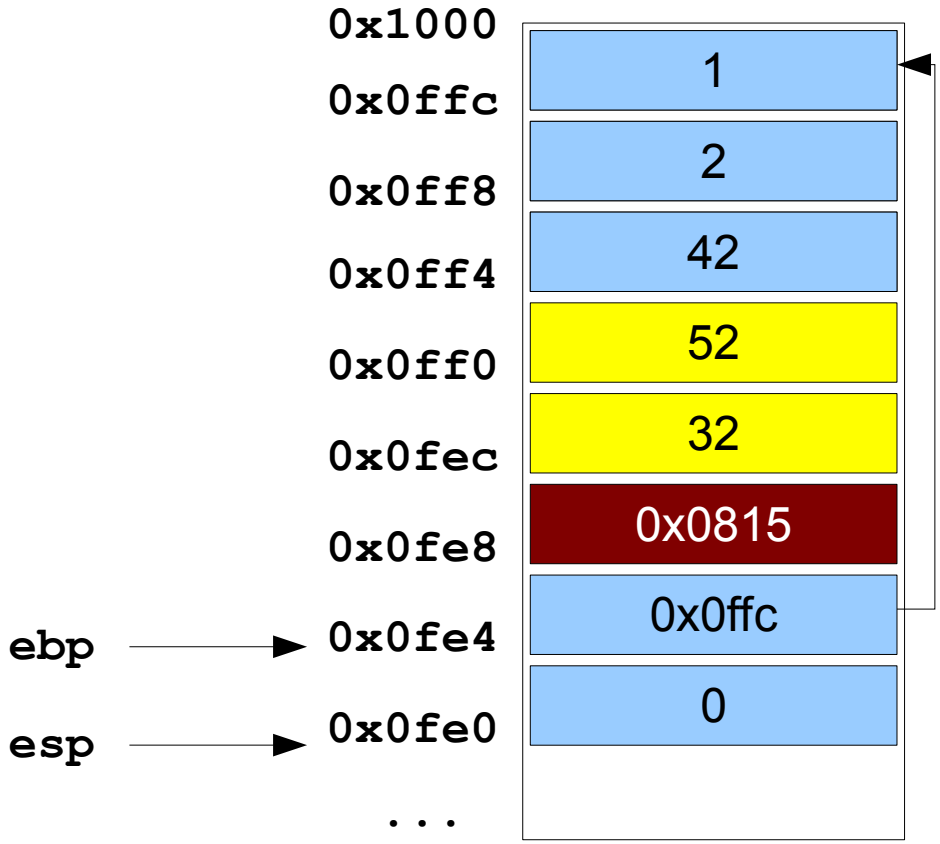
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   ret
```



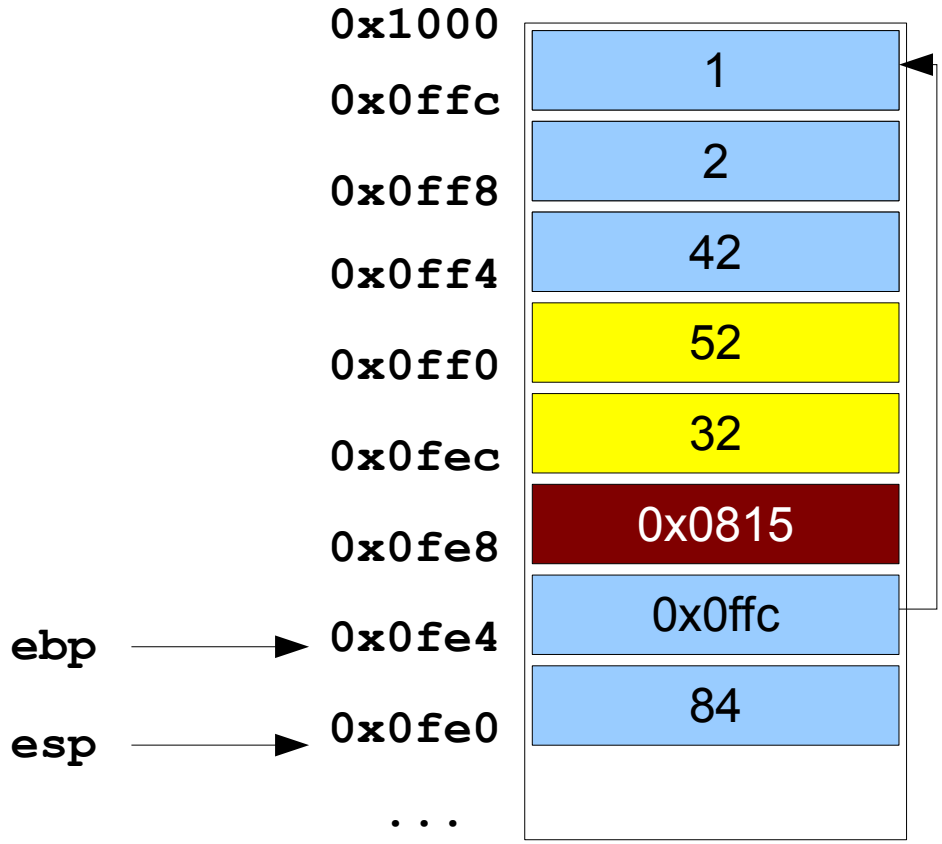
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   ret
```



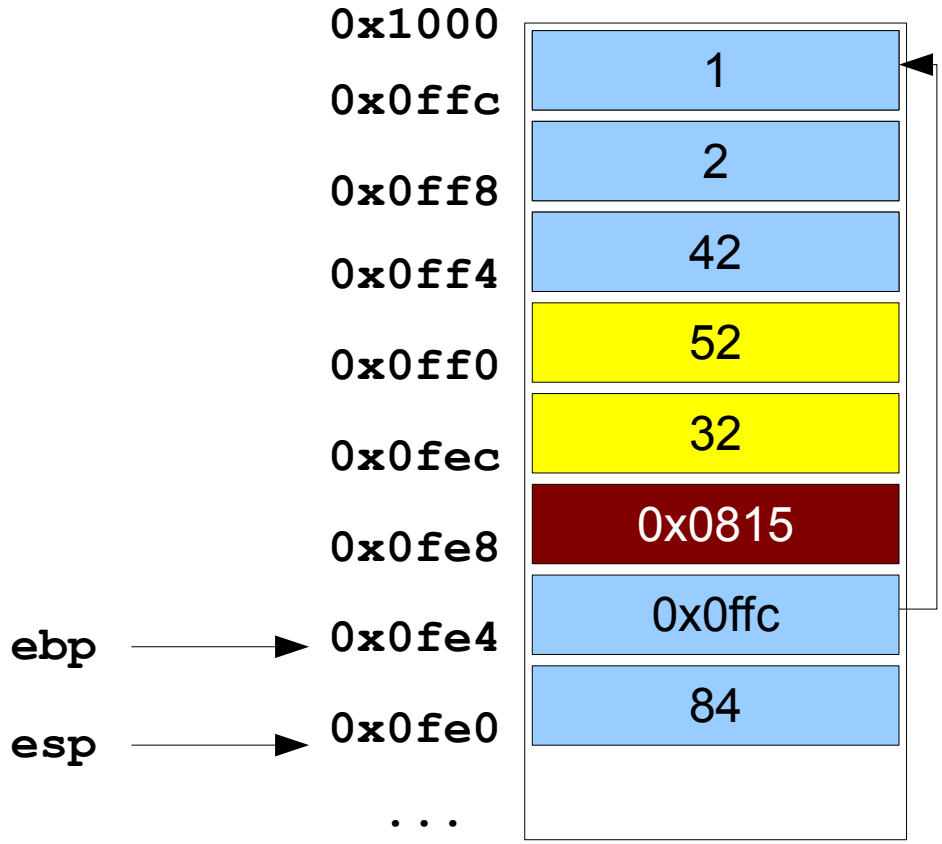
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   ret
```



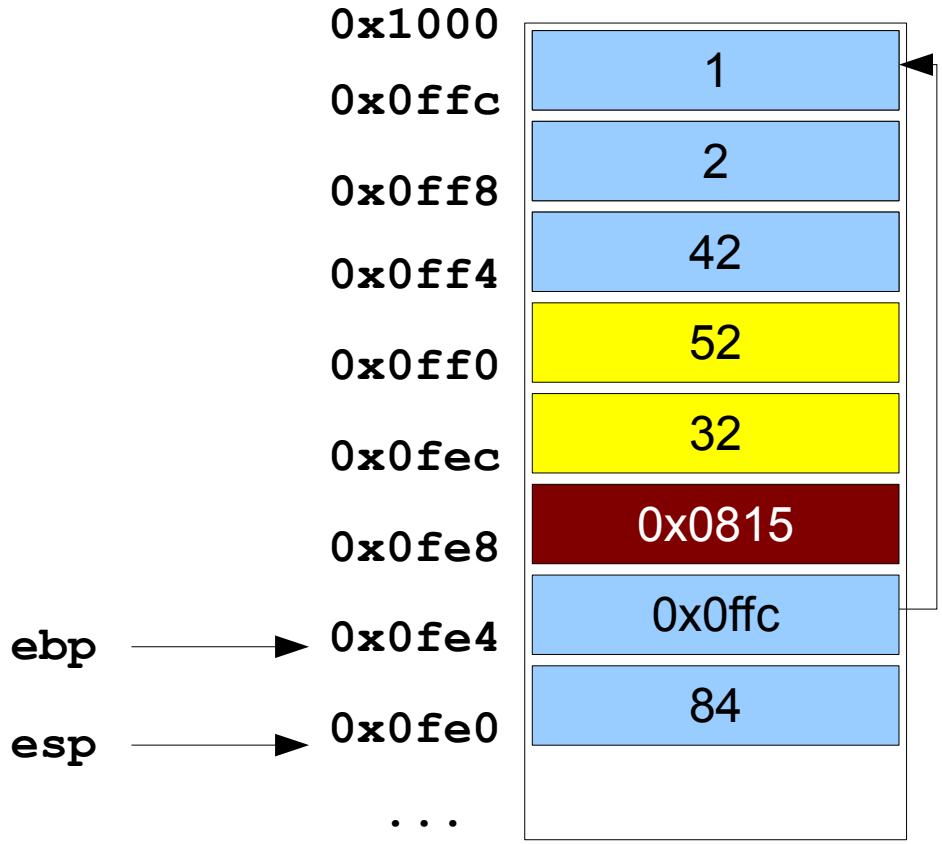
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   ret
```



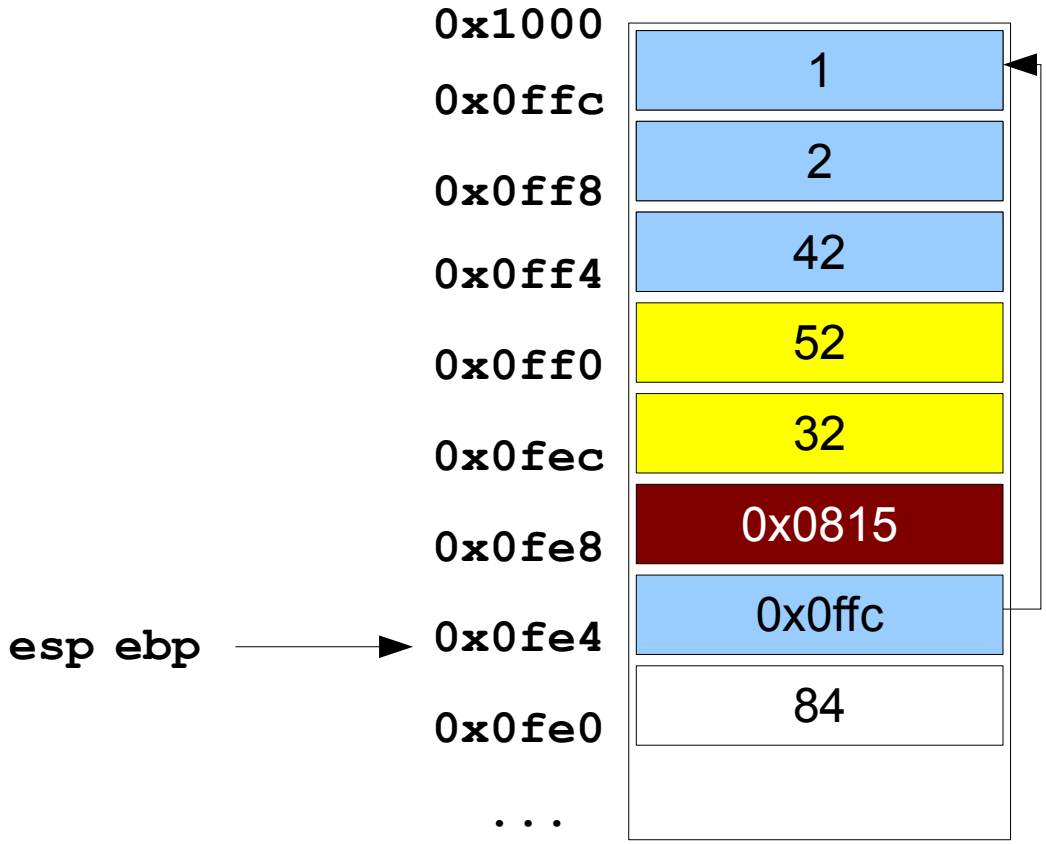
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```



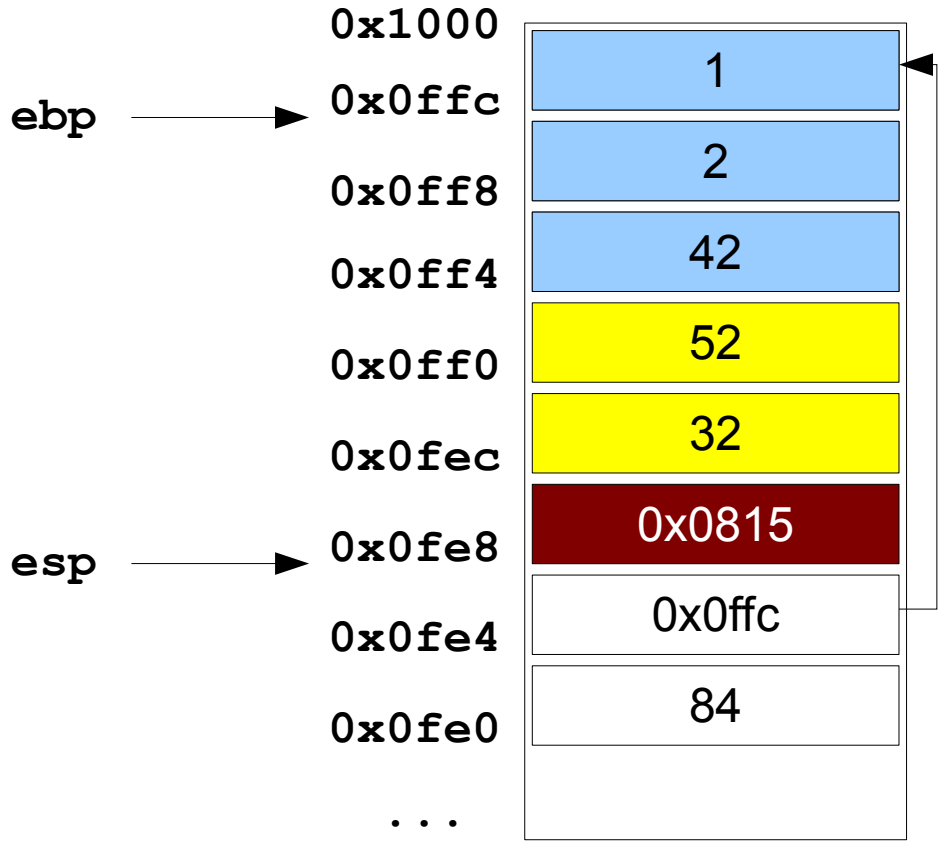
Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```



Inside the function

```
f: pushl %ebp
   movl %esp,%ebp
   subl $4, %esp
   movl 8(%ebp),%eax
   addl 12(%ebp),%eax
   movl %eax,-4(%ebp)
   ...
   movl -4(%ebp), %eax
   movl %ebp, %esp
   popl %ebp
   ret
```



Also known as one-instruction **leave**

Summing it up

32bit Conventions

Calling a function

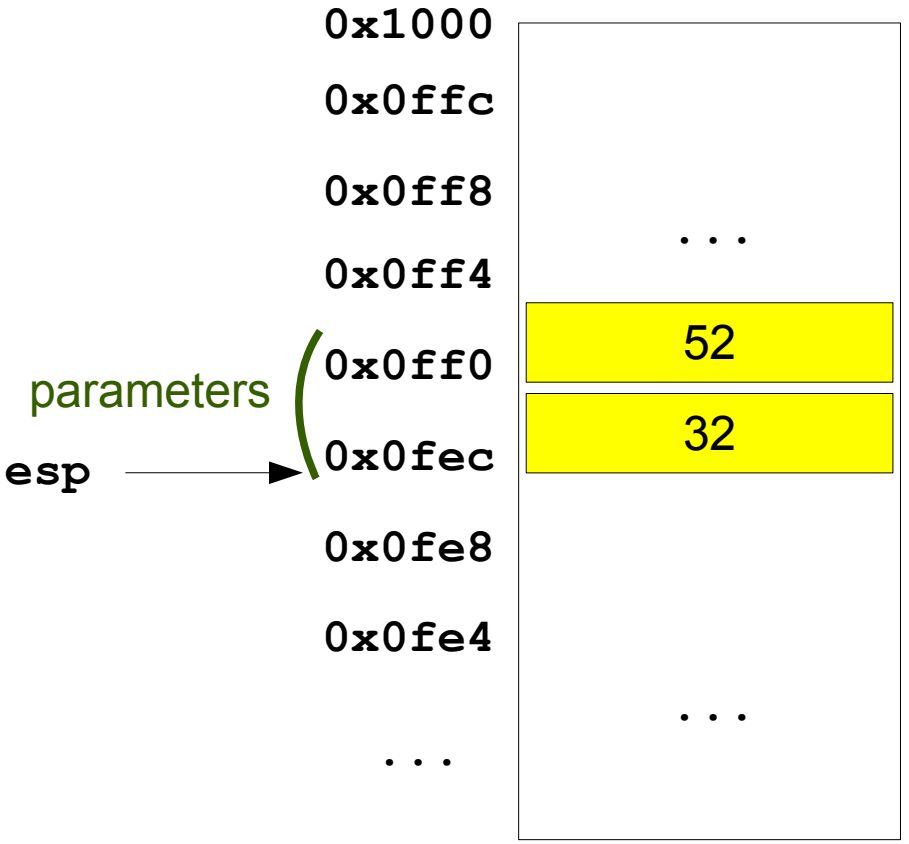
f(32, 52)

On function call, on the stack:

- parameters

```
call f
```

Which means pushing them in reverse order...



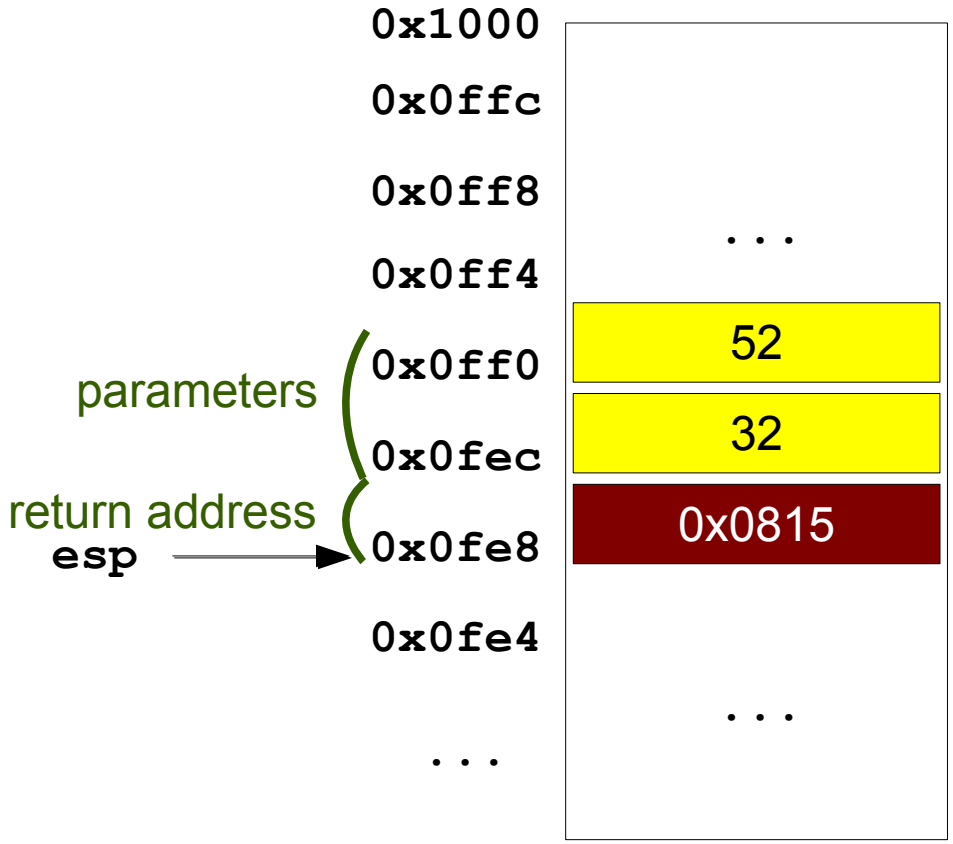
Function entry

f(32, 52)

On function entry, on the stack:

- parameters
- return address

```
f: ...
   ret
```



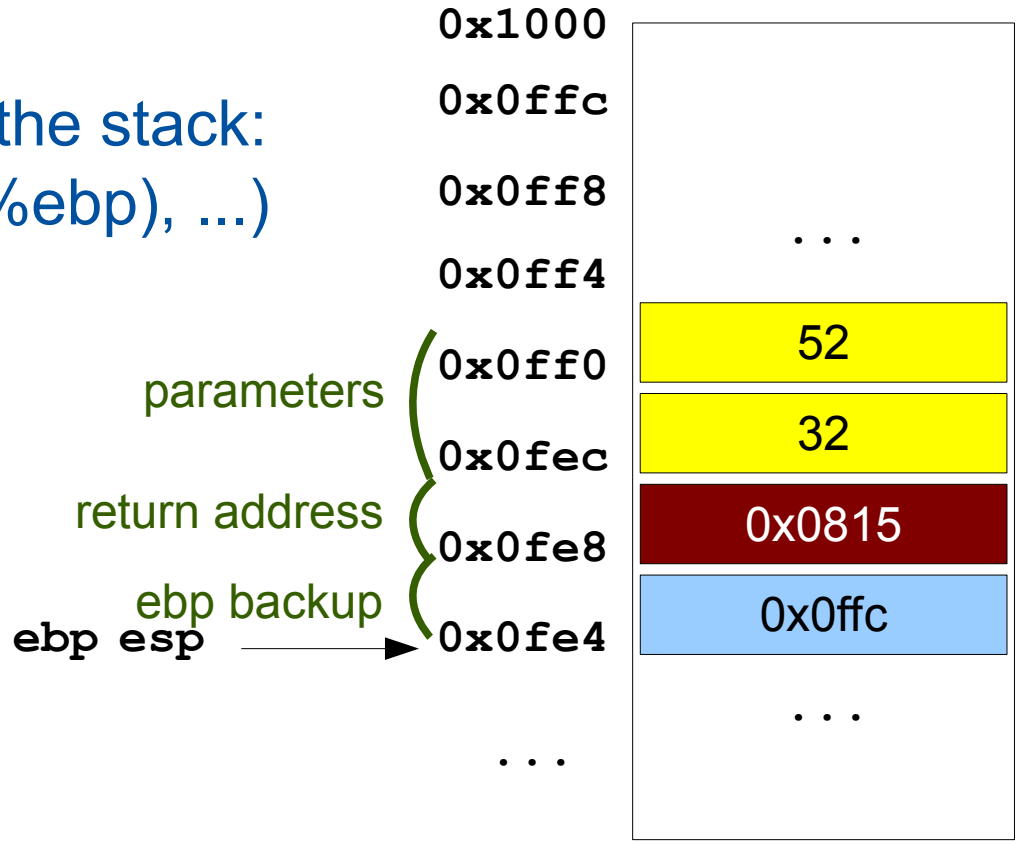
Ebp setup

f(32, 52)

After ebp setup (optional), on the stack:

- parameters (8(%ebp), 12(%ebp), ...)
- return address
- ebp backup

```
f: pushl %ebp
   movl %esp, %ebp
   ...
   ret
```



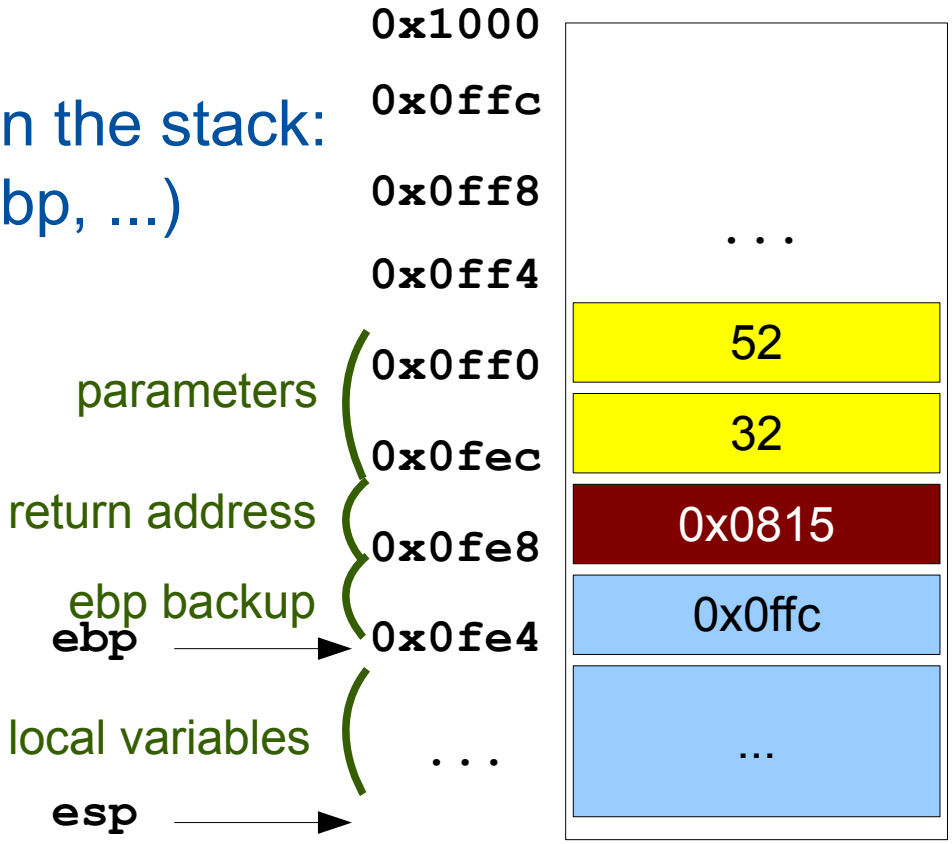
Now the real meat!

f(32, 52)

After local variables allocation, on the stack:

- parameters (8(%ebp), 12(%ebp, ...)
- return address
- ebp backup
- local variables (-4(%ebp), ...)

```
f: pushl %ebp
   movl %esp, %ebp
   subl $42, %esp
   ...
   ret
```



Learn this slide by heart

The ABI

What is an ABI?

Function call interoperability between application and libraries

→ Need to agree on

- Location of return address
- Content of the stack (parameters, return address)
- Location of returned value
- Volatile registers (see next slides)

- SystemV ABIs (Linux, BSD, MacOS)
  - i386, some variants: **cdecl**, stdcall, fastcall
  - amd64 (~= i386 fastcall)
- Microsoft x32/x64 ABIs

ABI vs API?

In C,

- .h headers provide the API
  - C types
  - Function/variables names
  - I.e. what the C caller should respect
- Interpretation of .h for a given platform provides the ABI
  - Type sizes
  - Calling convention
  - Symbol names
  - I.e. what the assembly language caller should respect

Vocabulary

- Function call: when foo() calls bar()
  - foo() is the **caller** function (*appelant*)
  - bar() is the **callee** function (*appelé*)
- **Local variable** (aka **automatic variable**): a variable whose scope is not getting outside of the function
- **Parameters** (aka **arguments**): Data set by the caller for the callee
- **Return value**: Data set by the callee for the caller
- **Call stack** (aka **backtrace**): The chain of functions that have been currently called
  - e.g. main() → foo() → bar()

SystemV ABI

32bit

- Return address in (%esp)
- Parameters in 4(%esp), 8(%esp), ...
- Return value in
  - %eax for ≤ 32bit integers
  - st(0) for float

64bit

- Return address in (%rsp)
- ≤ 64bit integer parameters in %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Float/Double parameters in %xmm0, %xmm1, ...
- struct members split in registers
- Otherwise, on the stack
- Return value in
  - %rax for ≤ 64bit integers
  - %xmm0 for float/double

A horrible 64bit example

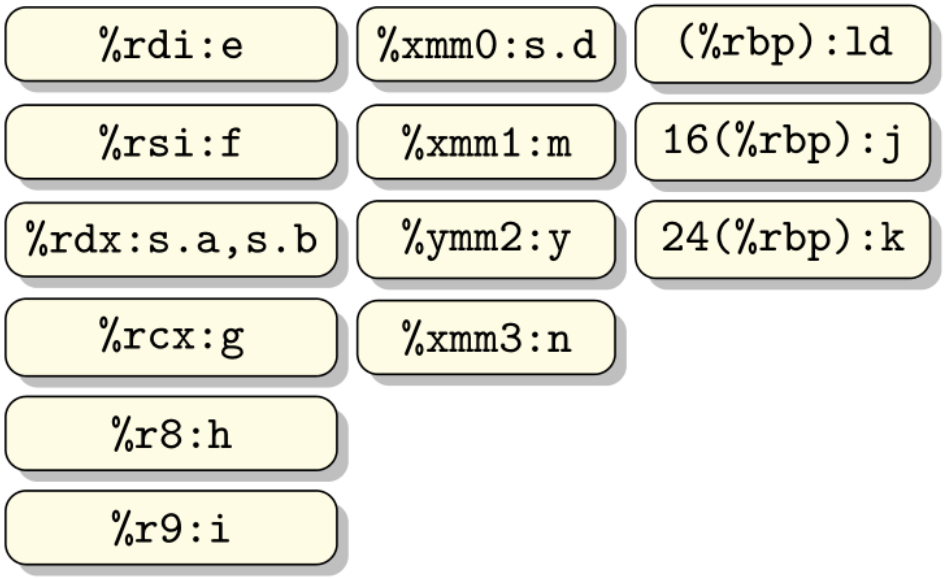
```
typedef struct {
  int a, b;
  double d;
} structparm;

structparm s;

int e,f,g,h,i,j,k;
long double ld;
double m, n;
__m256 y;

extern void
func (int e, int f, structparm s, int g, int h,
      long double ld, double m, __m256 y, double n,
      int i, int j, int k);

func (e, f, s, g, h, ld, m, y, n, i, j, k);
```



Case of returning a structure

```
struct mys foo(int a, float b)
```

- The caller is in charge of providing the memory space of the structure
- Pointer passed as additional argument before **`a`**
- Callee eventually sets %eax to that pointer
- (32bit) Callee drops the additional argument with **`ret $4`**

So, actually becomes

```
struct mys *foo(struct mys *ret, int a, float b)
```

Volatile?

- Caller may want to keep values in registers across function call
  - Avoid having to store in a local variable
- But callee could want to use registers too!

→ Meet halfway: volatile / non-volatile

Volatile registers: callee can use it at will

- eax, ecx, edx
- aka **caller-saved**

Non-volatile registers: callee has to save/restore it

- ebx, edi, esi, ebp
- aka **callee-saved**

References


Michael Matz, Jan Hubicka, Andreas Jaeger, and Mark Mitchell.
System V Application Binary Interface: AMD64 Architecture Processor Supplement, September 2010.
Version 0.99.5.


Santa Cruz Operation, Inc.
System V Application Binary Interface: i386 Architecture Processor Supplement, fourth edition, March 1997.