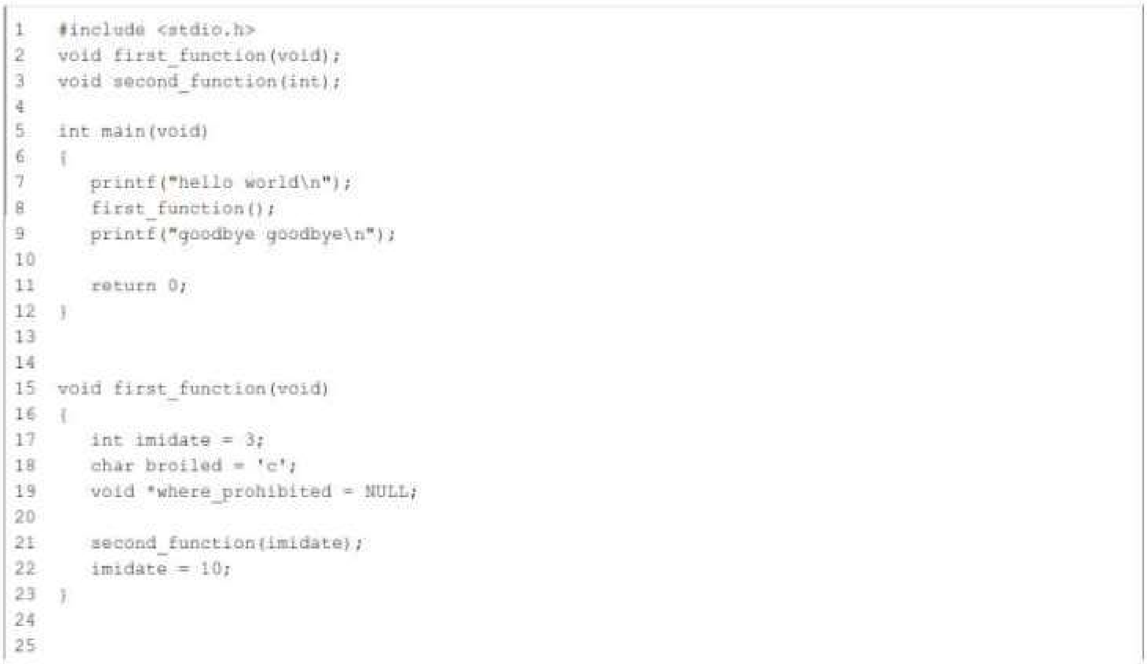
Stack Frames And The Stack

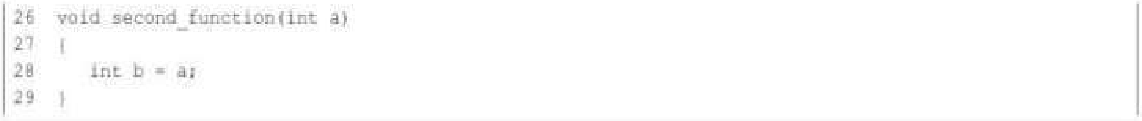
You just learned about the memory layout for a process. One section of this memory layout is called the stack, which is a collection of stack frames. Each stack frame represents a function call. As functions are called, the number of stack frames increases, and the stack grows. Conversely, as functions return to their caller, the number of stack frames decreases, and the stack shrinks. In this section, we learn what a stack frame is. A very detailed explanation here, but we'll go over what's important for our purposes.

A program is made up of one or more functions which interact by calling each other. Every time a function is called, an area of memory is set aside, called a stack frame, for the new function call. This area of memory holds some crucial information, like:

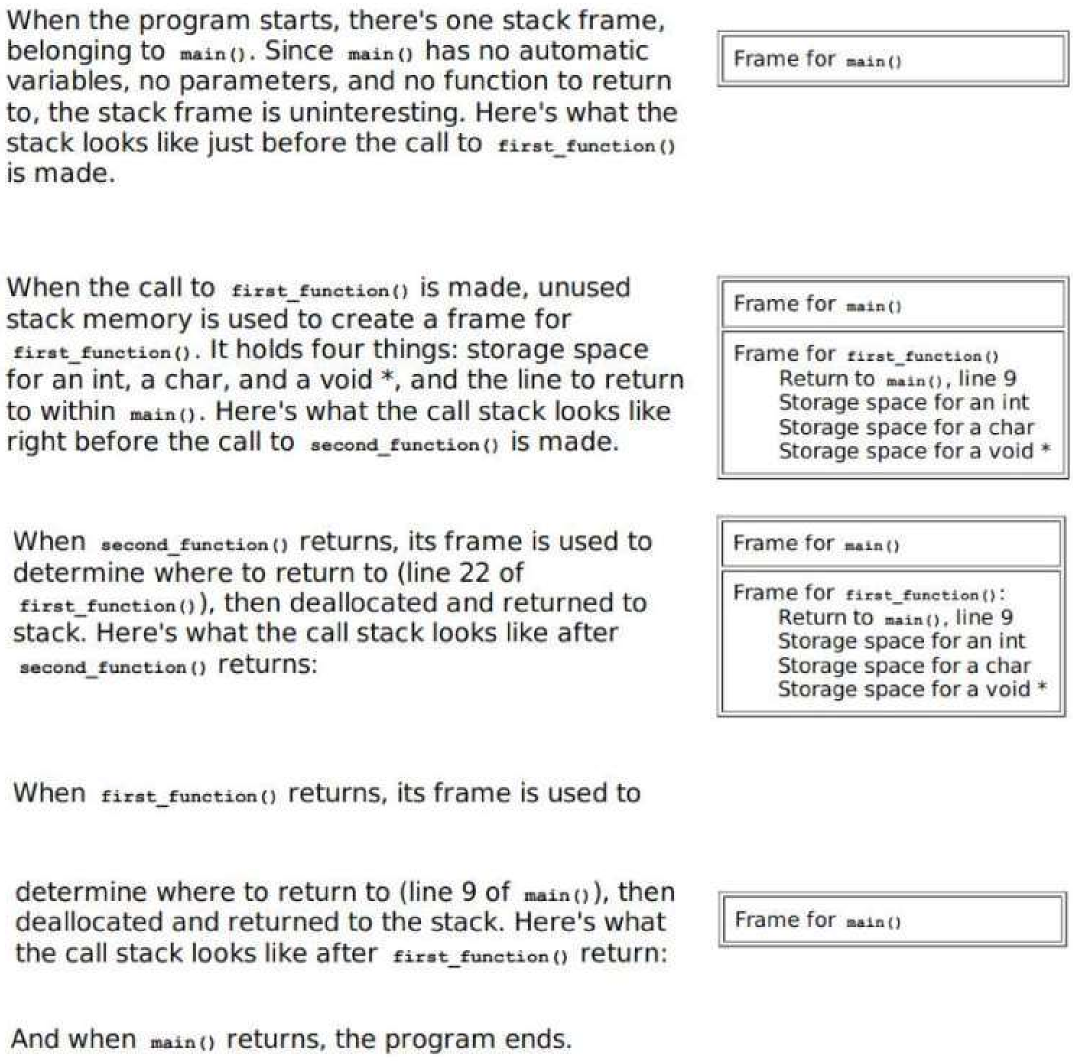
1. Storage space for all the automatic variables for the newly called function.
2. The **line number** of the calling function to return to when the called function returns.
3. The arguments, or parameters, of the called function.



```c
#include <stdio.h>
void first_function(void);
void second_function(int);

int main(void)
{
   printf("hello world\n");
   first_function();
   printf("goodbye goodbye\n");

   return 0;
}


void first_function(void)
{
   int imidate = 3;
   char broiled = 'c';
   void *where_prohibited = NULL;

   second_function(imidate);
   imidate = 10;
}


```



```c
void second_function(int a)
{
   int b = a;
}
```



When the program starts, there's one stack frame, belonging to `main()`. Since `main()` has no automatic variables, no parameters, and no function to return to, the stack frame is uninteresting. Here's what the stack looks like just before the call to `first_function()` is made.

Frame for `main()`

When the call to `first_function()` is made, unused stack memory is used to create a frame for `first_function()`. It holds four things: storage space for an int, a char, and a void *, and the line to return to within `main()`. Here's what the call stack looks like right before the call to `second_function()` is made.

Frame for `main()`

Frame for `first_function()`
Return to `main()`, line 9
Storage space for an int
Storage space for a char
Storage space for a void *

When `second_function()` returns, its frame is used to determine where to return to (line 22 of `first_function()`), then deallocated and returned to stack. Here's what the call stack looks like after `second_function()` returns:

Frame for `main()`

Frame for `first_function()`:
Return to `main()`, line 9
Storage space for an int
Storage space for a char
Storage space for a void *

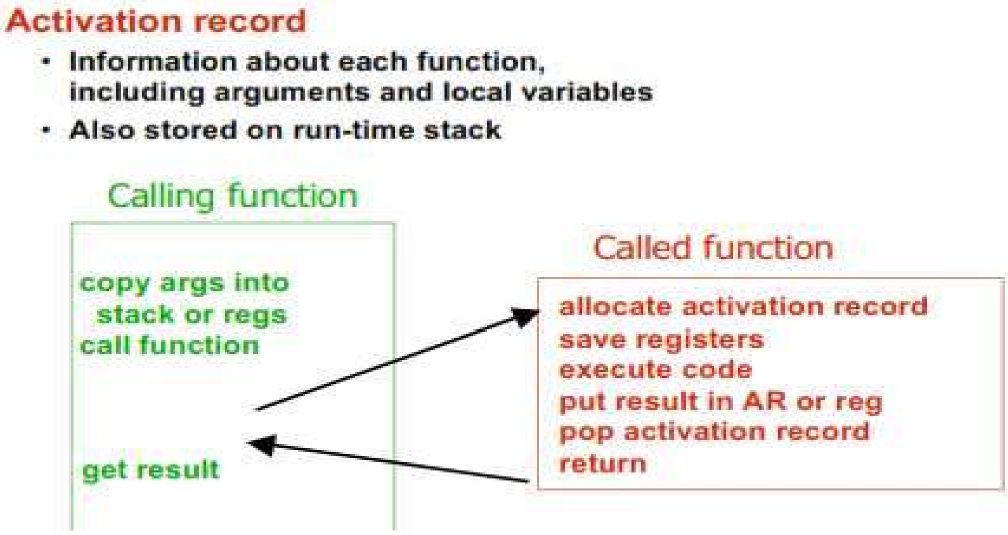
When `first_function()` returns, its frame is used to

determine where to return to (line 9 of `main()`), then deallocated and returned to the stack. Here's what the call stack looks like after `first_function()` return:

Frame for `main()`

And when `main()` returns, the program ends.

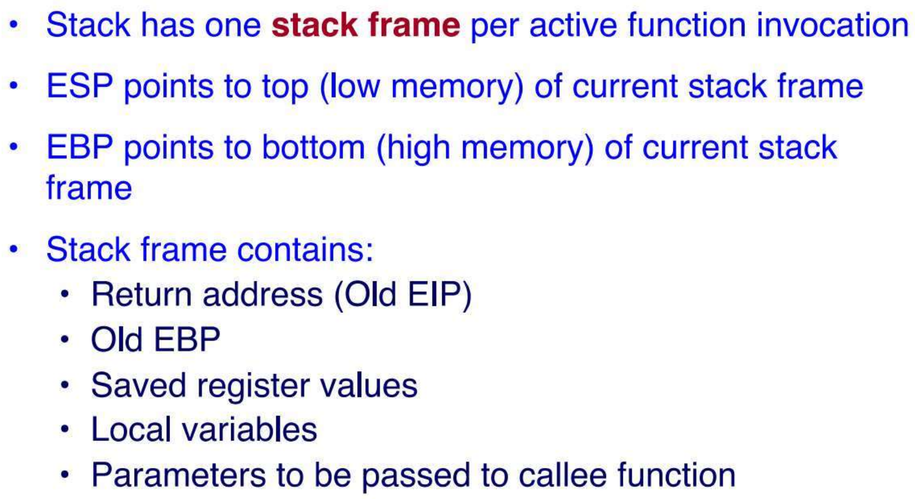
Now what is an activation record (AR)….



Activation record

- Information about each function, including arguments and local variables
- Also stored on run-time stack

Summary of Intel Architecture's function handling



- Stack has one **stack frame** per active function invocation
- ESP points to top (low memory) of current stack frame
- EBP points to bottom (high memory) of current stack frame
- Stack frame contains:
  - Return address (Old EIP)
  - Old EBP
  - Saved register values
  - Local variables
  - Parameters to be passed to callee function

Explanation on x86 Registers(reg)

The main tools to write programs in x86 assembly are the processor registers. The registers are like variables built in the processor. Using registers instead of memory to store values makes the process faster and cleaner.

General registers

General registers are the ones we use most of the time, most of the instructions perform on these registers. They all can be broken down into 16 and 8-bit registers.

32 bits: EAX EBX ECX EDX
16 bits: AX BX CX DX
8 bits: AH AL BH BL CH CL DH DL
The "H" and "L" suffixes on the 8-bit registers stand for high byte and low byte.

EAX, AX, AH, AL:
Called the Accumulator register. It is used for I/O port access, arithmetic, interrupt calls, etc...
EBX, BX, BH, BL:
Called the Base register. It is used as a base pointer for memory access. Gets some interrupt return values.
ECX, CX, CH, CL:
Called the Counter register. It is used as a loop counter and for shifts. Gets some interrupt values.
EDX, DX, DH, DL:
Called the Data register. It is used for I/O port access, arithmetic, and some interrupt calls.

Segment registers

Segment registers hold the segment address of various items. They are only available in 16 values. They can only be set by a general register or special instructions.

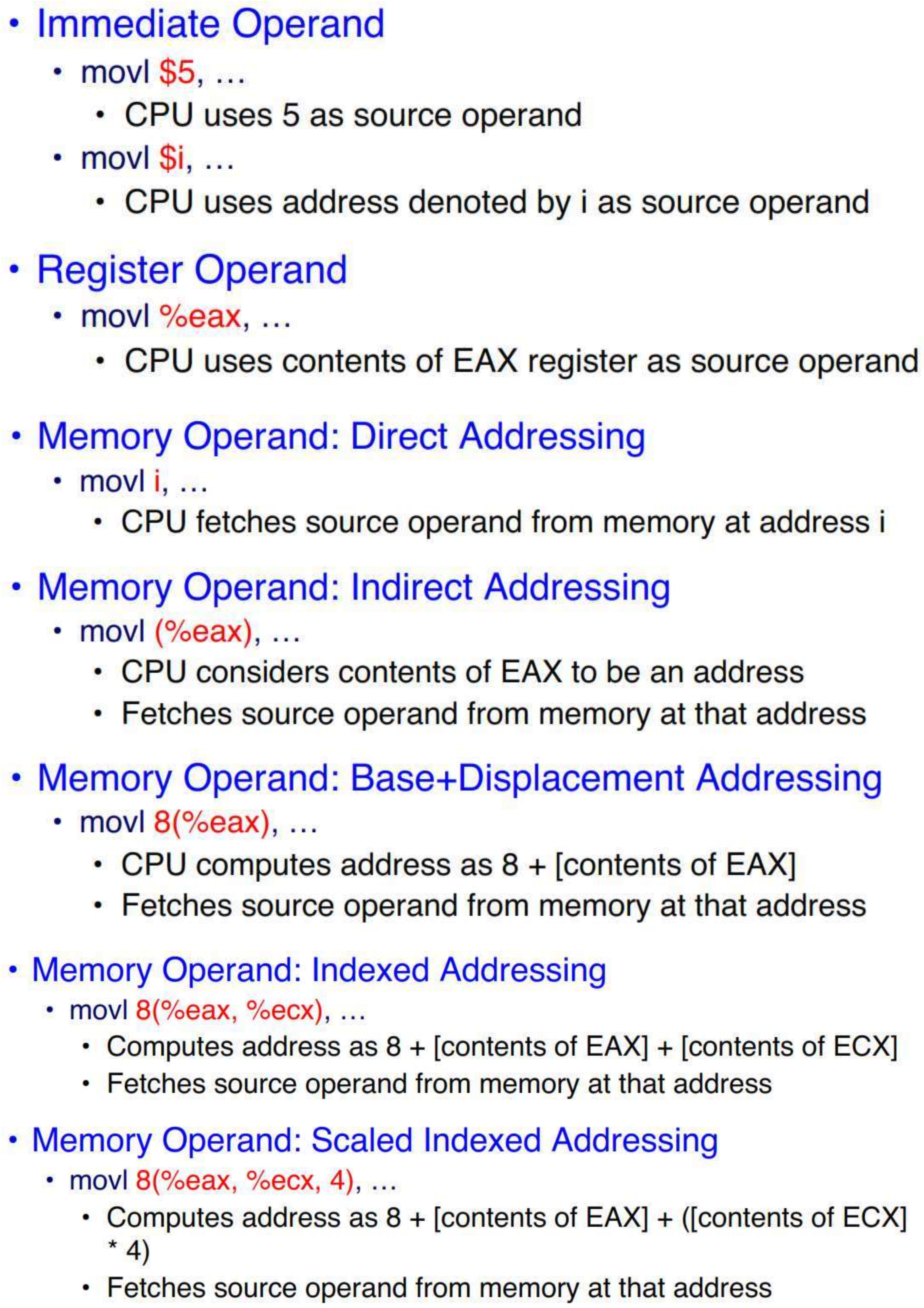
CS:
Holds the Code segment in which your program runs.
Changing its value might make the computer hang.
DS:
Holds the Data segment that your program accesses.
Changing its value might give erroneous data.
ES, FS, GS:
These are extra segment registers available for
far pointer addressing like video memory and such.
SS:
Holds the Stack segment your program uses. Sometimes has the same value as DS. Changing its value can give unpredictable results, mostly data related.

Indexes and pointers

ESI and EDI: Index register
Used for string, and memory array copying.
EBP: Stack Base pointer register
Holds the base address of the stack.
ESP: Stack pointer register
Holds the top address of the stack.
EIP: Instruction Pointer
Holds the offset of the next instruction.

The ESI, EDI, EBX, EBP, ESP are call-preserved whereas EAX, ECX and EDX are not call-preserved. Call-preserved registers are respected by C library function and their values persist through the C library function calls.

Examples of Operands



- Immediate Operand
  - movl $5, …
    - CPU uses 5 as source operand
  - movl $i, …
    - CPU uses address denoted by i as source operand
- Register Operand
  - movl %eax, …
    - CPU uses contents of EAX register as source operand
- Memory Operand: Direct Addressing
  - movl i, …
    - CPU fetches source operand from memory at address i
- Memory Operand: Indirect Addressing
  - movl (%eax), …
    - CPU considers contents of EAX to be an address
    - Fetches source operand from memory at that address
- Memory Operand: Base+Displacement Addressing
  - movl 8(%eax), …
    - CPU computes address as 8 + [contents of EAX]
    - Fetches source operand from memory at that address
- Memory Operand: Indexed Addressing
  - movl 8(%eax, %ecx), …
    - Computes address as 8 + [contents of EAX] + [contents of ECX]
    - Fetches source operand from memory at that address
- Memory Operand: Scaled Indexed Addressing
  - movl 8(%eax, %ecx, 4), …
    - Computes address as 8 + [contents of EAX] + ([contents of ECX] * 4)
    - Fetches source operand from memory at that address

A Simple Example

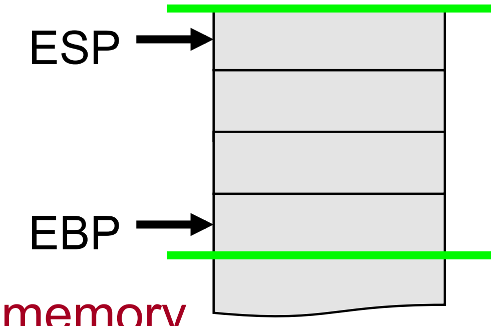
```
int add3(int a, int b, int c)
{
   int d;
   d = a + b + c;
   return d;
}
```

```
/* In some calling function */

   …
   x = add3(3, 4, 5);
   …
```

Trace of a Simple Example 1

```
x = add3(3, 4, 5);
```

Low memory



High memory

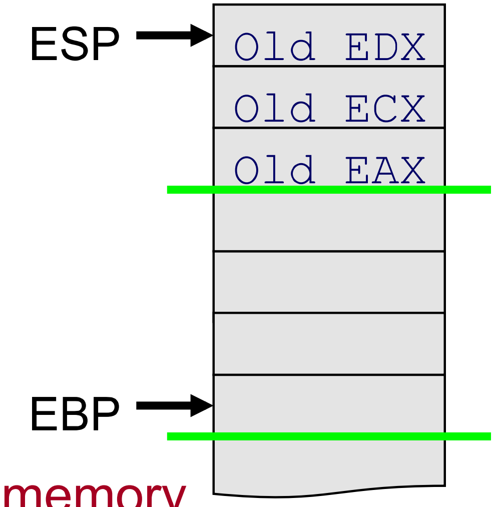
Trace of a Simple Example 2

```
x = add3(3, 4, 5);
```

Low memory

# Save caller-save registers if necessary

```
pushl %eax
pushl %ecx
pushl %edx
```



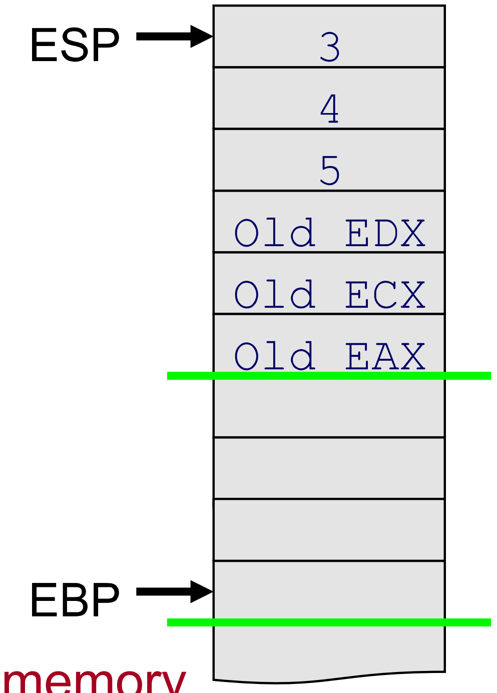
High memory

Trace of a Simple Example 3

```
x = add3(3, 4, 5);
```

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
```



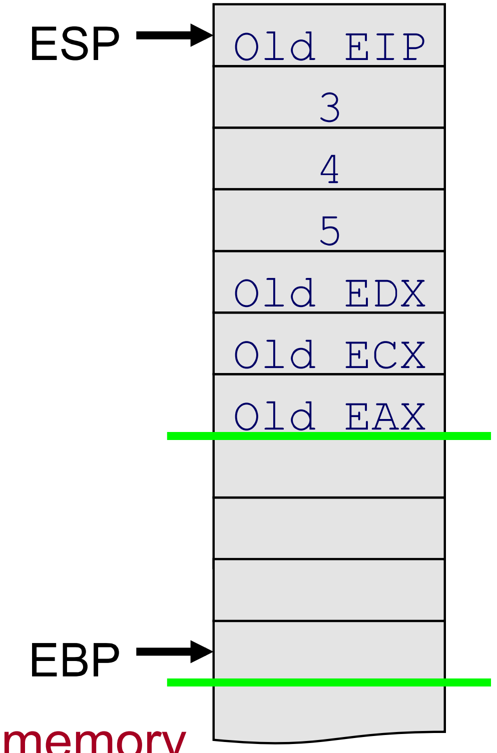
High memory

Trace of a Simple Example 4

```
x = add3(3, 4, 5);
```

Low memory

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
```



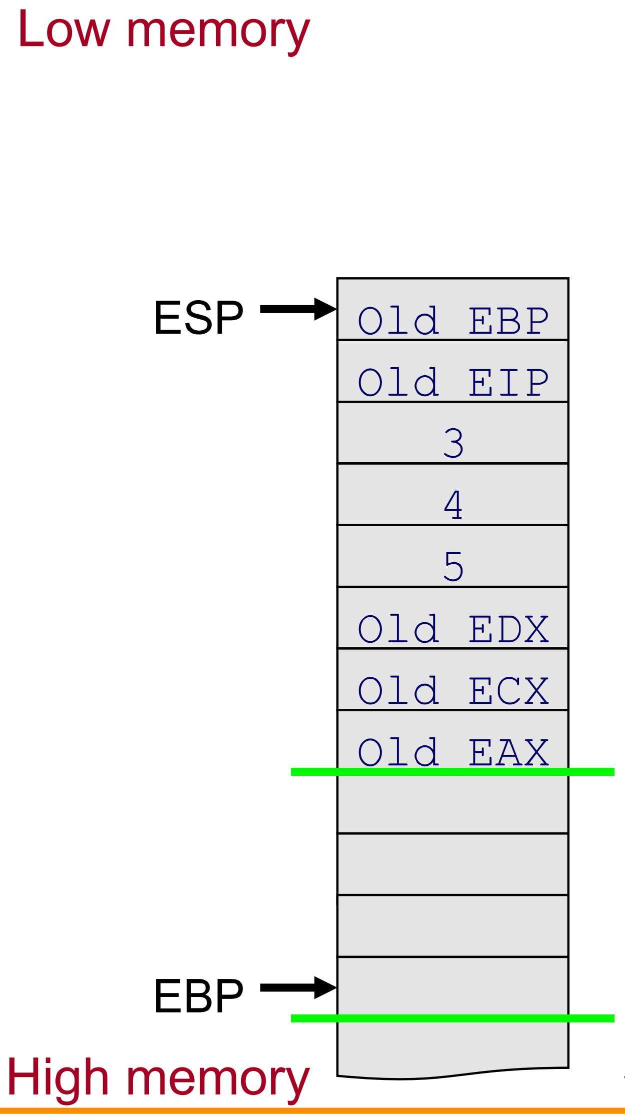
High memory

Trace of a Simple Example 5

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
```

Prolog

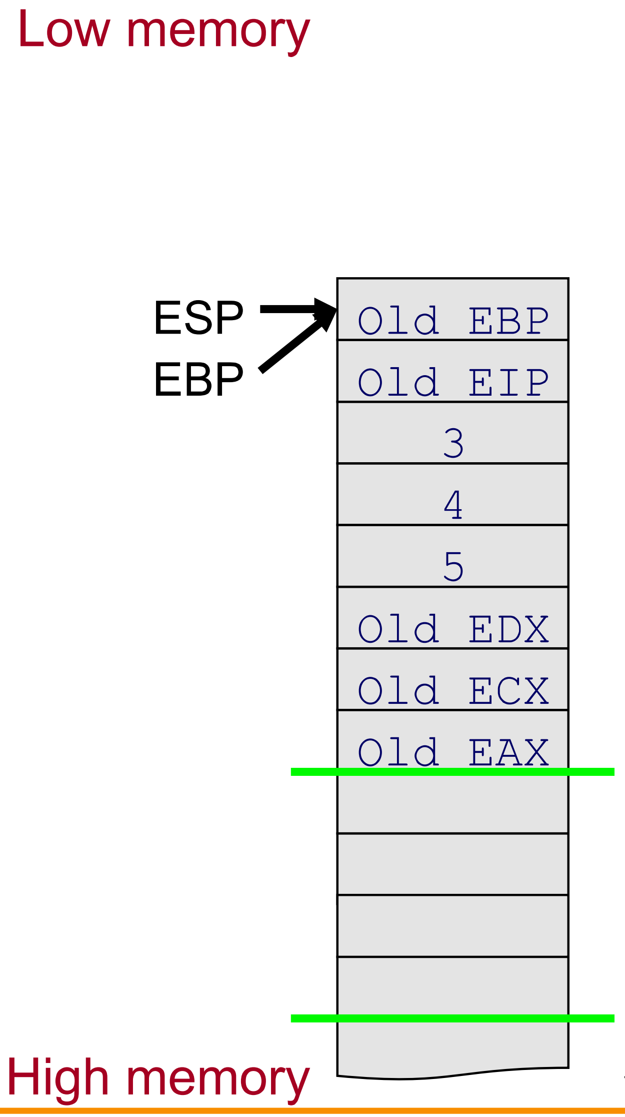


Trace of a Simple Example 6

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
```

Prolog



Low memory

ESP

EBP

Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

Old EAX

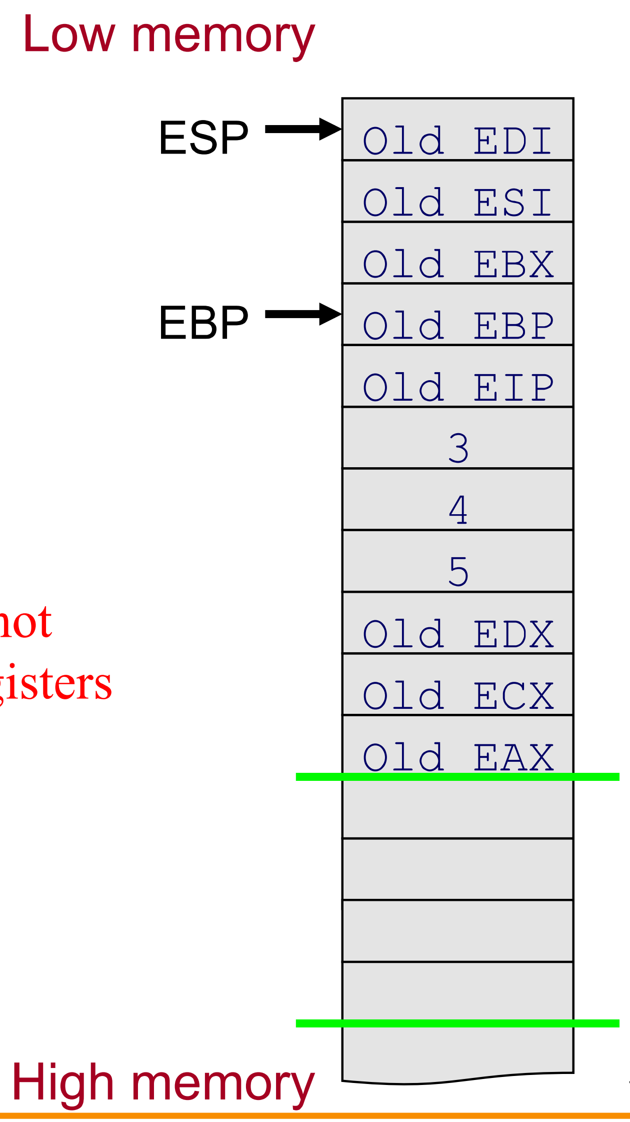
High memory

Trace of a Simple Example 7

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
```

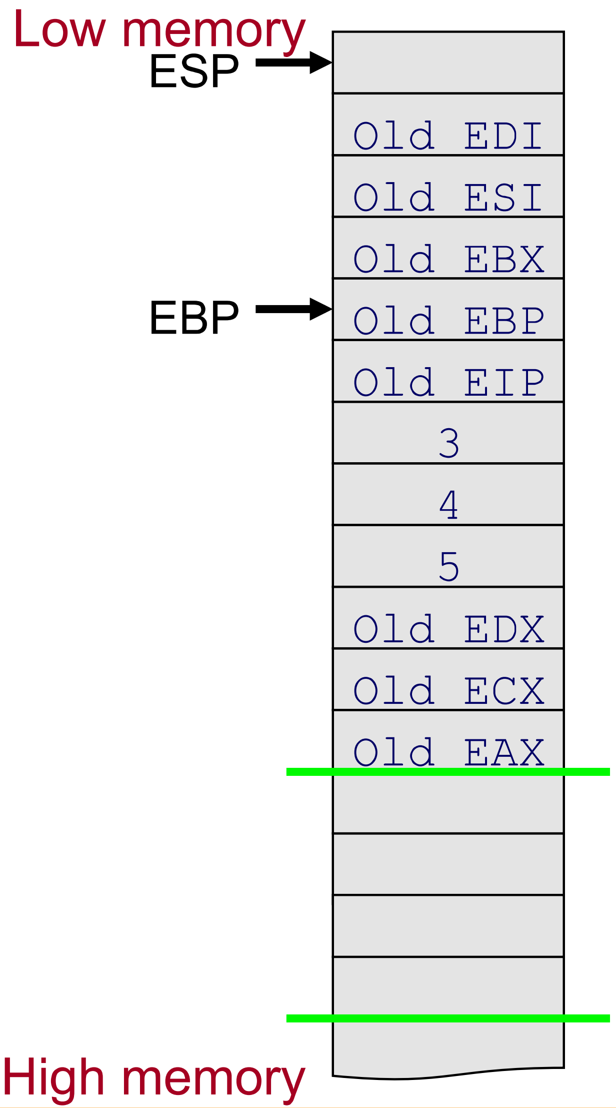
Unnecessary here; add3 will not change the values in these registers



Trace of a Simple Example 8

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
```



Trace of a Simple Example 9

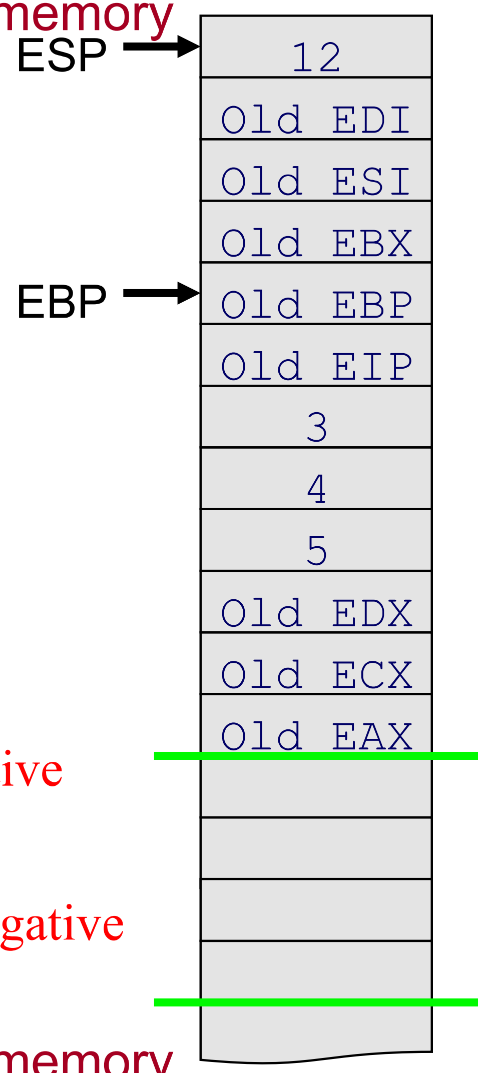
```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Save old EBP
pushl %ebp
# Change EBP
movl %esp, %ebp
# Save caller-save registers if necessary
pushl %ebx
pushl %esi
pushl %edi
# Allocate space for local variable
subl $4, %esp
# Perform the addition
movl 8(%ebp), %eax
addl 12(%ebp), %eax
addl 16(%ebp), %eax
movl %eax, -16(%ebp)
```

Access params as positive offsets relative to EBP

Access local vars as negative offsets relative to EBP

Low memory



ESP

EBP

12

Old EDI

Old ESI

Old EBX

Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

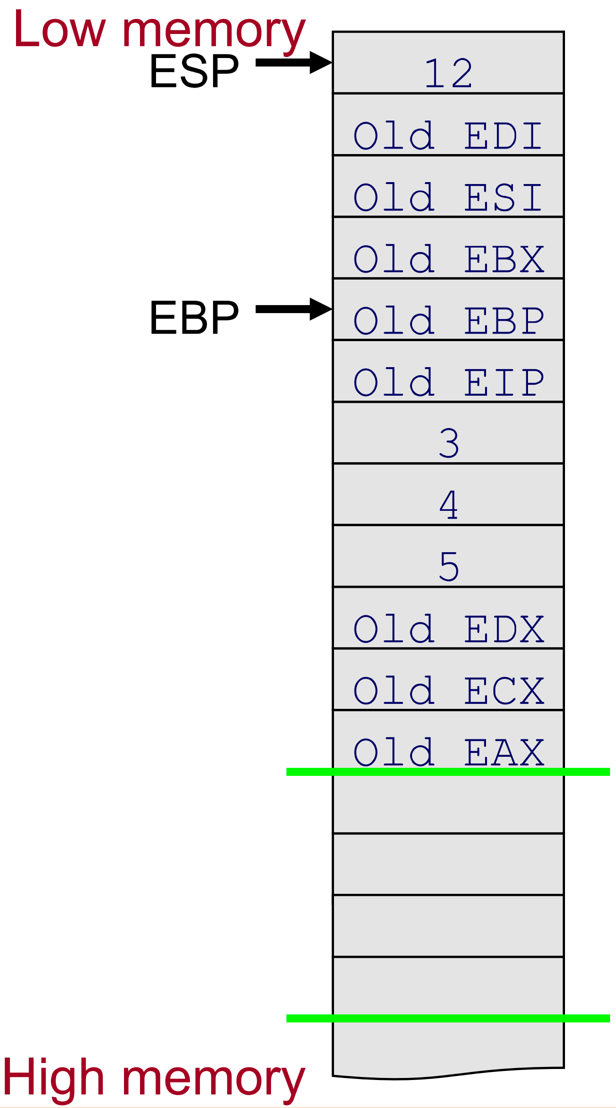
Old EAX

High memory

Trace of a Simple Example 10

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
```



Low memory

ESP → 12

Old EDI

Old ESI

Old EBX

EBP → Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

Old EAX

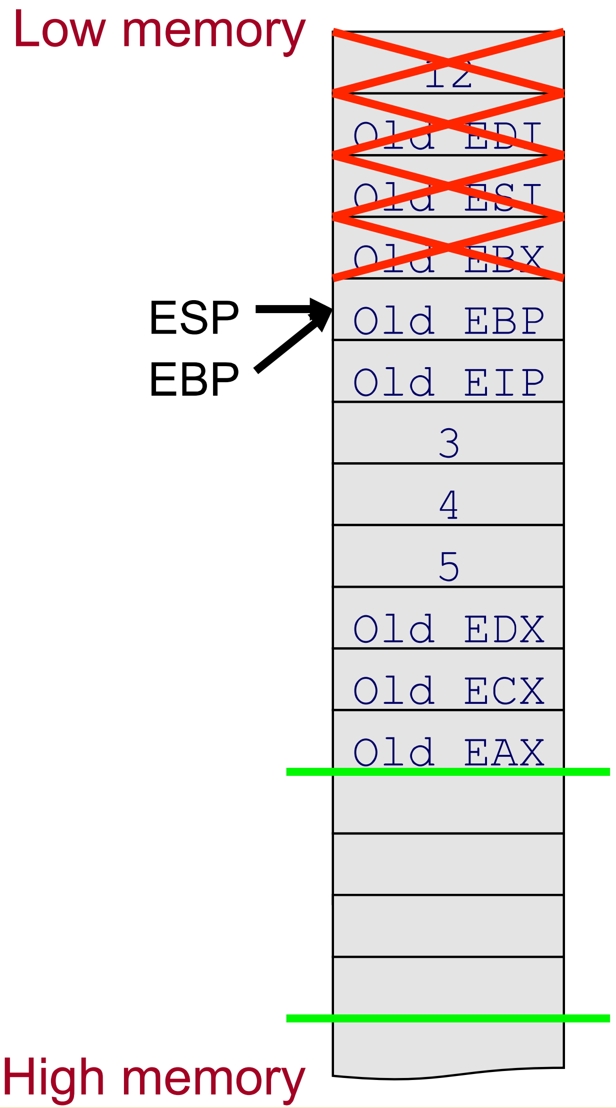
High memory

Trace of a Simple Example 11

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
```

Epilog



Low memory

~~12~~

~~Old EDI~~

~~Old ESI~~

~~Old EBX~~

ESP EBP → Old EBP

Old EIP

3

4

5

Old EDX

Old ECX

Old EAX

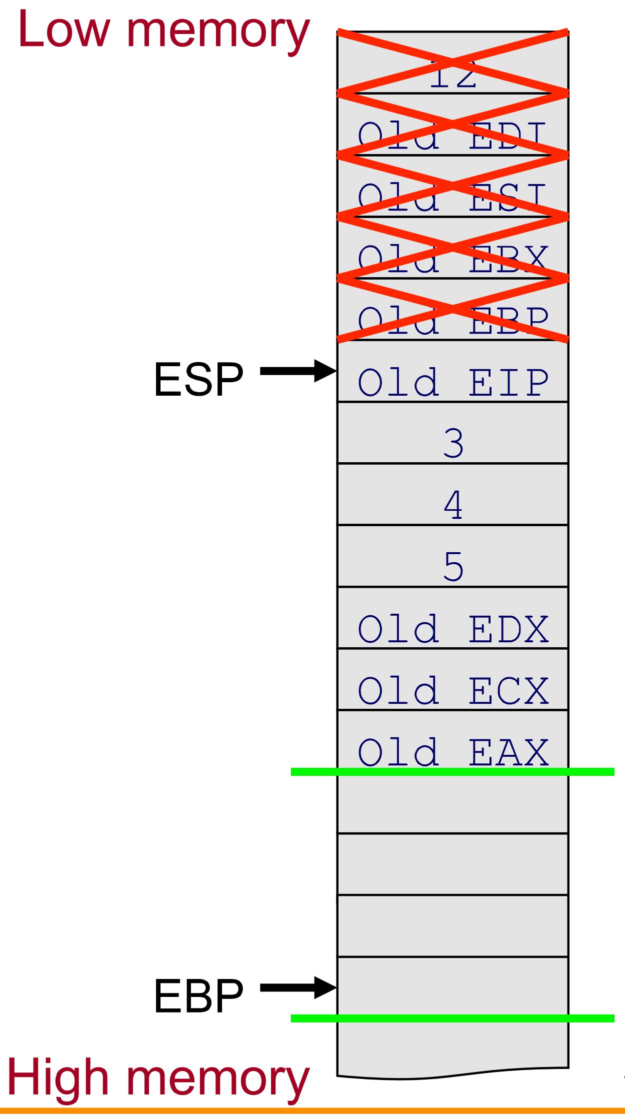
High memory

Trace of a Simple Example 12

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
```

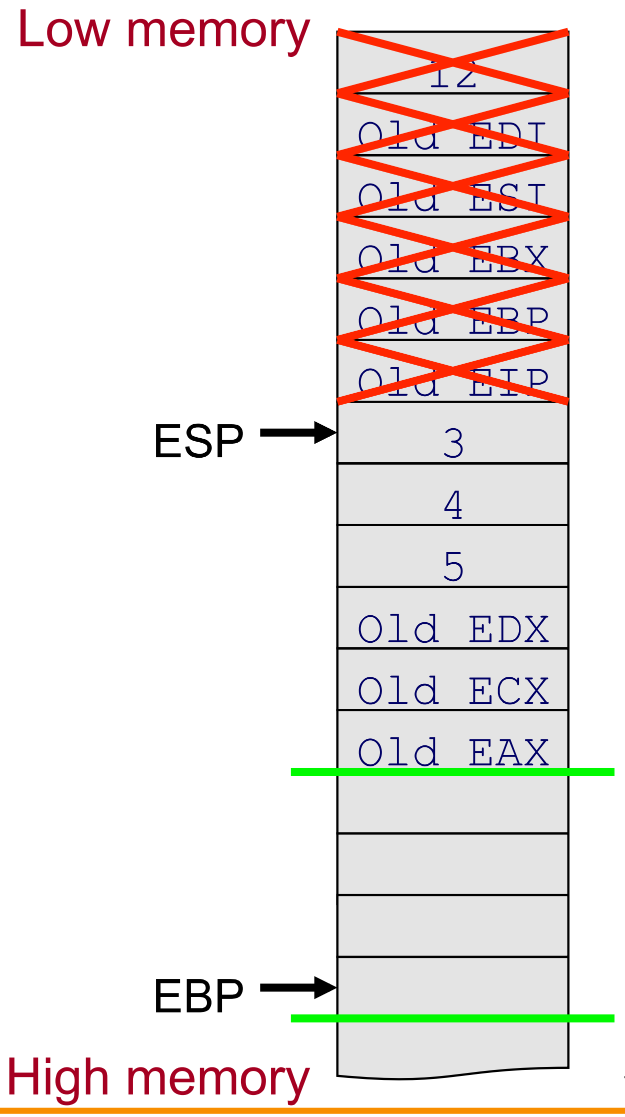
Epilog



Trace of a Simple Example 13

```
int add3(int a, int b, int c) {
  int d;
  d = a + b + c;
  return d;
}
```

```
# Copy the return value to EAX
movl -16(%ebp), %eax
# Restore callee-save registers if necessary
movl -12(%ebp), %edi
movl -8(%ebp), %esi
movl -4(%ebp), %ebx
# Restore ESP
movl %ebp, %esp
# Restore EBP
popl %ebp
# Return to calling function
ret
```



Low memory

~~12~~

~~Old EDI~~

~~Old ESI~~

~~Old EBX~~

~~Old EBP~~

~~Old EIP~~

ESP

3

4

5

Old EDX

Old ECX

Old EAX

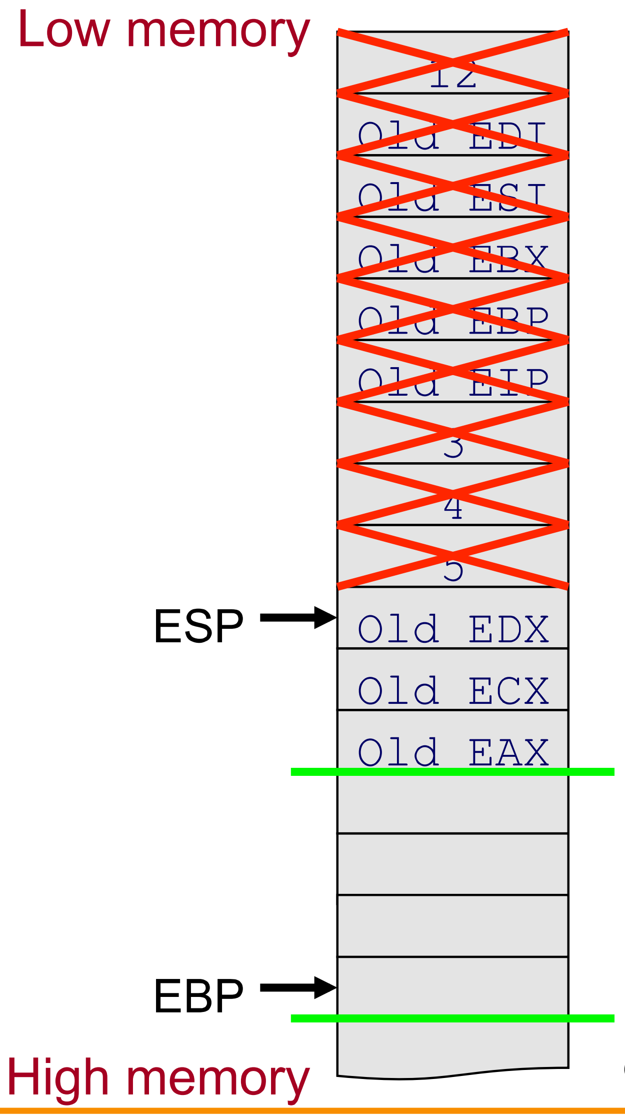
EBP

High memory

Trace of a Simple Example 14

```
x = add3(3, 4, 5);
```

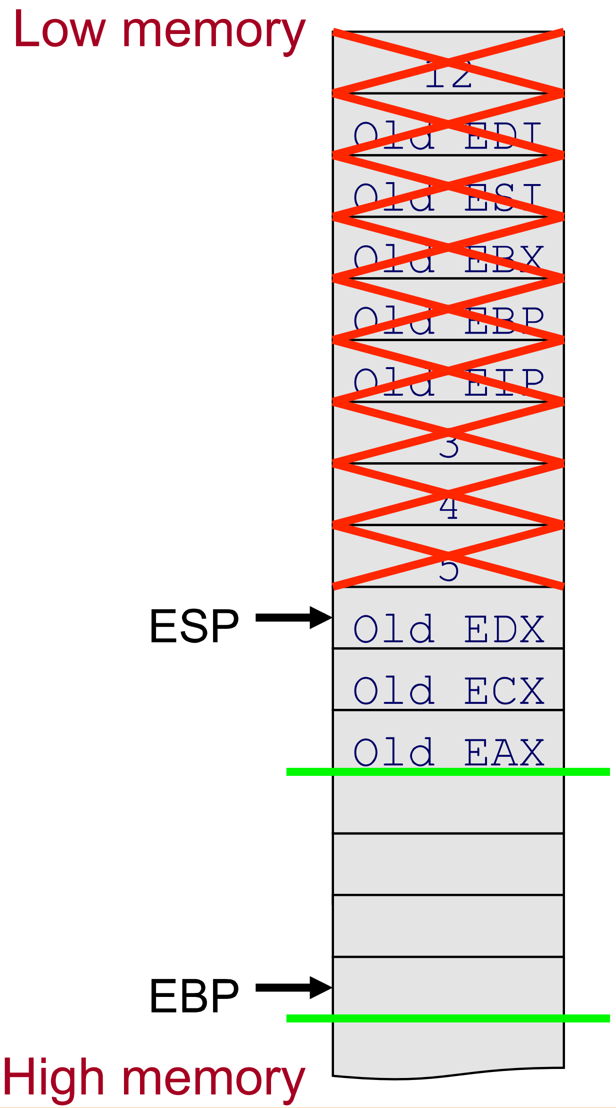
```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl $12, %esp
```



Trace of a Simple Example 15

```
x = add3(3, 4, 5);
```

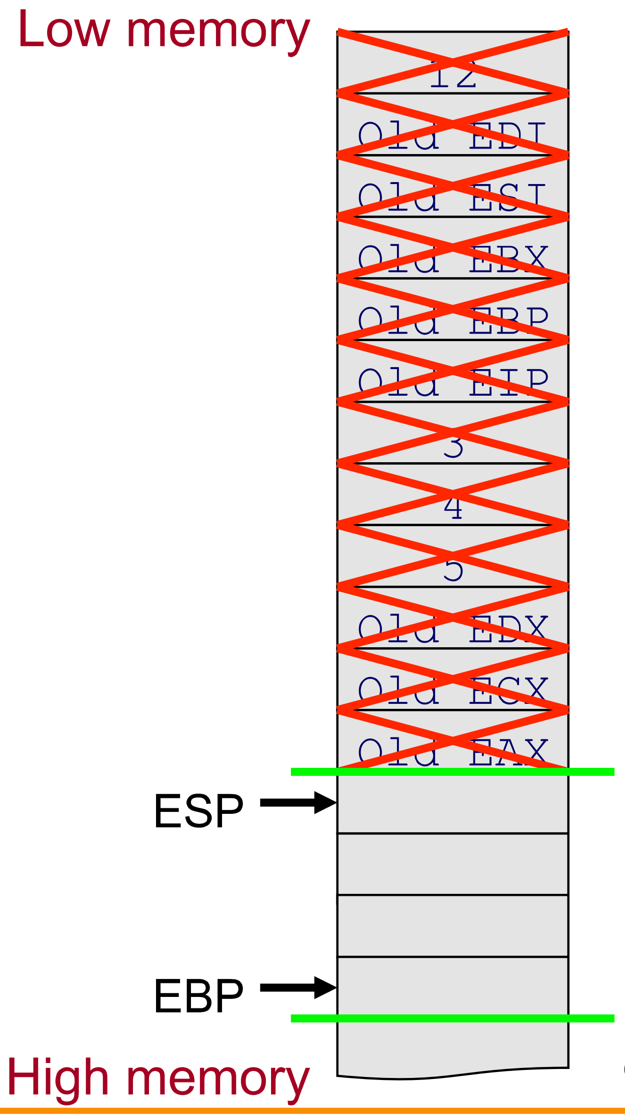
```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
```



Trace of a Simple Example 16

```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
```

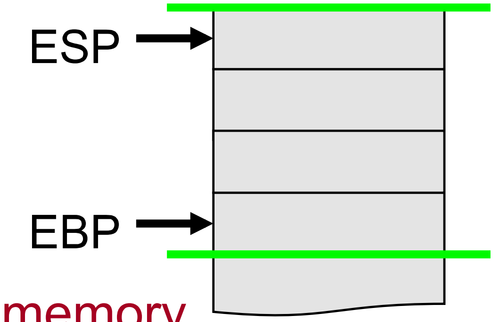


Trace of a Simple Example 17

```
x = add3(3, 4, 5);
```

```
# Save caller-save registers if necessary
pushl %eax
pushl %ecx
pushl %edx
# Push parameters
pushl $5
pushl $4
pushl $3
# Call add3
call add3
# Pop parameters
addl %12, %esp
# Save return value
movl %eax, wherever
# Restore caller-save registers if necessary
popl %edx
popl %ecx
popl %eax
# Proceed!
…
```

Low memory



High memory

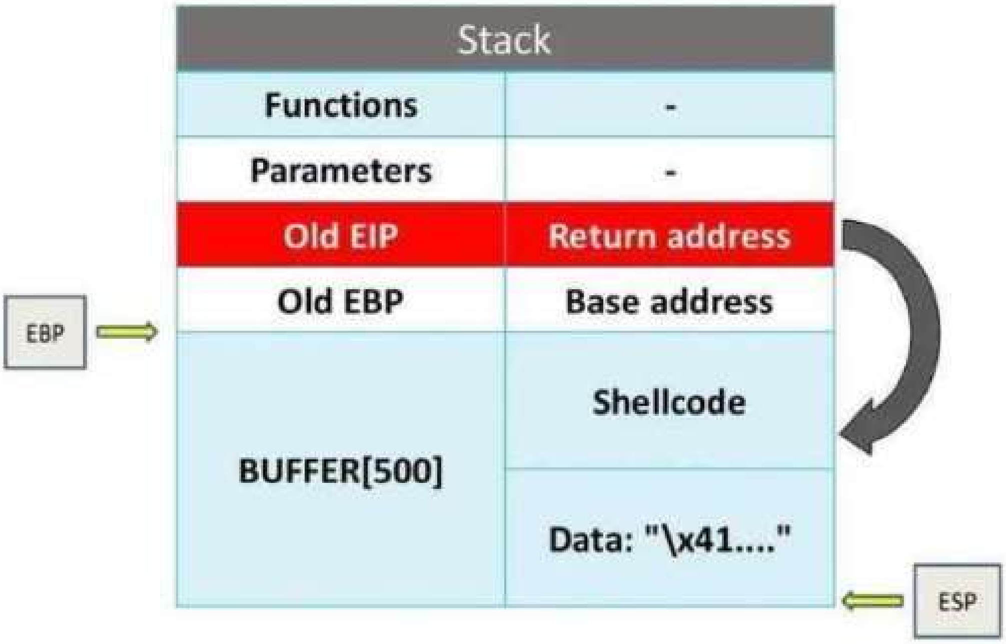
Summary

- Calling and returning
  - Call instruction: push EIP onto stack and jump
  - Ret instruction: pop stack to EIP
- Passing parameters
  - Caller pushes onto stack
  - Callee accesses as positive offsets from EBP
  - Caller pops from stack

Summary (cont.)

- Storing local variables
  - Callee pushes on stack
  - Callee accesses as negative offsets from EBP
  - Callee pops from stack
- Handling registers
  - Caller saves and restores EAX, ECX, EDX if necessary
  - Callee saves and restores EBX, ESI, EDI if necessary
- Returning values
  - Callee returns data of integral types and pointers in EAX

How the buffer overflow is exploited….

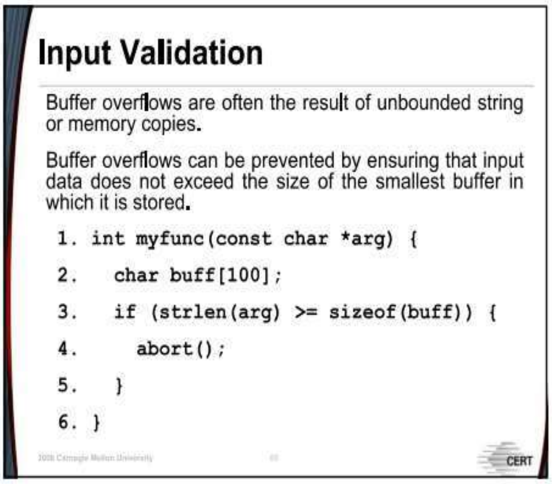


Assume a `buffer` of length 500 defined in a function. Now it is overflowed in such a way that it has some `random data`, followed by the `shellcode` (malicious code) and then the `Return address` which points to the shellcode.

So after the function gets executed, the instruction pointed by the `Return address` gets executed and this is how our shellcode gets executed.

This is pretty much how Buffer Overflow happens.

Mitigations….

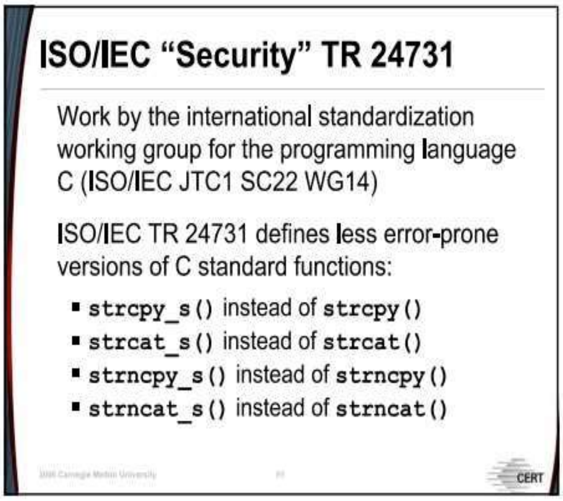


Input Validation

Buffer overflows are often the result of unbounded string or memory copies.

Buffer overflows can be prevented by ensuring that input data does not exceed the size of the smallest buffer in which it is stored.

```
int myfunc(const char *arg) {
  char buff[100];
  if (strlen(arg) >= sizeof(buff)) {
    abort();
  }
}
```



ISO/IEC "Security" TR 24731

Work by the international standardization working group for the programming language C (ISO/IEC JTC1 SC22 WG14)

ISO/IEC TR 24731 defines less error-prone versions of C standard functions:

- `strcpy_s()` instead of `strcpy()`
- `strcat_s()` instead of `strcat()`
- `strncpy_s()` instead of `strncpy()`
- `strncat_s()` instead of `strncat()`