Offensive Software Exploitation

Summer 2020

Ali Hadi
@binaryz0ne

Intro. to Software Exploitation

Prepared by:

Ali Hadi

Shadi Naif, @ShadiNaif

Part #2

Outline

- Introduction
- CPU Instructions & Registers
- Functions, High Level View
- Stacks and Stack Frames
- Memory Addressing
- Managing Stack Frames
- Functions, Low Level View
  - Understanding the Process
  - Call Types
  - Assembly Language
  - General Trace
  - Code Optimizations
  - Stack Reliability

Software Exploitation Intro.

- A program is made of a set of rules following a certain execution flow that tells the computer what to do.
- Exploiting the program (Goal):
  - Getting the computer to do what you want it to do, even if the program was designed to prevent that action *[The Art of Exploitation, 2nd Ed]*.
- First documented attack 1972 (US Air Force Study).
- Even with the new mitigation techniques, software today is still exploited!

What is needed?

- To understand software exploitation, we need a well understanding of:
  - Computer Languages
  - Operating Systems
  - Architectures

What will be covered?

- What we will cover is:
  - CPU Registers
  - How Functions Work
  - Memory Management for the IA32 Architecture
  - A glance about languages: Assembly and C

- Why do I need those?
  - Because most of the security holes come from memory corruption!

CPU Instructions & Registers

- The CPU contains many registers depending on its model & architecture.
- In this lecture, we are interested in three registers: EBP, ESP, and EIP which is the instruction pointer.
- Instruction is the lowest execution term for the CPU, while Statement is a high-level term that is compiled and then loaded as one or many instructions.
- Assembly language is the human friendly representation of the instructions machine code.

CPU Registers Overview

16 Bits	32 Bits	64 Bits	Description
AX	EAX	RAX	Accumulator
BX	EBX	RBX	Base Index
CX	ECX	RCX	Counter
DX	EDX	RDX	Data
BP	EBP	RBP	Base Pointer
SP	ESP	RSP	Stack Pointer
IP	EIP	RIP	Instruction Pointer
SI	ESI	RSI	Source Index Pointer
DI	EDI	RDI	Destination Index Pointer

- Some registers can be accessed using there lower and higher words. For example, AX register; lower word AL and higher word AH can be accessed separately.
- The above is not the complete list of CPU registers.

Functions: High Level View

```
void myfun2(char *x) {
  printf(“You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf(“No arguments!\n");
}
```

Functions: High Level View

```c
void myfun2(char *x) {
    printf("You entered: %s\n", x);
}

void myfun1(char *str) {
    char buffer[16];
    strcpy(buffer, str);
    myfun2(buffer);
}

int main(int argc, char *argv[]) {
    if (argc > 1)
        myfun1(argv[1]);
    else printf("No arguments!\n");
}
```

A function consist of:

- Name
- Parameters (or arguments)
- Body
- Local variable definitions
- Return value type

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions: High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

PUSH position into the Stack

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun1(argv[1]);
```

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

PUSH position into the Stack

`myfun2(buffer);`

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

myfun2(buffer);
myfun1(argv[1]);

Functions: High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
      myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun2(buffer);`
`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

`myfun2(buffer);`
`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

POP Position out of the Stack

`myfun2(buffer);`

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

```
myfun1(argv[1]);
```

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
    myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

POP Position out of the Stack

`myfun1(argv[1]);`

Functions: High Level View

```
void myfun2(char *x) {
   printf("You entered: %s\n", x);
}

void myfun1(char *str) {
   char buffer[16];
   strcpy(buffer, str);
   myfun2(buffer);
}

int main(int argc, char *argv[]) {
   if (argc > 1)
       myfun1(argv[1]);
   else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
      myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

Current Statement

Saved Return Positions

Functions: High Level View

```
void myfun2(char *x) {
  printf("You entered: %s\n", x);
}

void myfun1(char *str) {
  char buffer[16];
  strcpy(buffer, str);
  myfun2(buffer);
}

int main(int argc, char *argv[]) {
  if (argc > 1)
     myfun1(argv[1]);
  else printf("No arguments!\n");
}
```

A stack is the best structure to trace the program execution

End of Execution

Saved Return Positions

Stack & Stack Frames

- There is no physical stack inside the CPU. Instead; the CPU uses the main memory to represent a logical structure of a stack

- The operating system reserves a contiguous raw memory space for the stack

- This stack is logically divided into many Stack Frames

- The stack and all stack frames are represented in the memory upside-down

Stack & Stack Frames – Cont.

- A stack frame is represented by two pointers:
  - Base pointer (saved in EBP register):

    the memory address that is equal to (EBP-1) is the first memory location of the stack frame.

  - Stack pointer (saved in ESP register):

    the memory address that is equal to (ESP) is the top memory location of the stack frame.

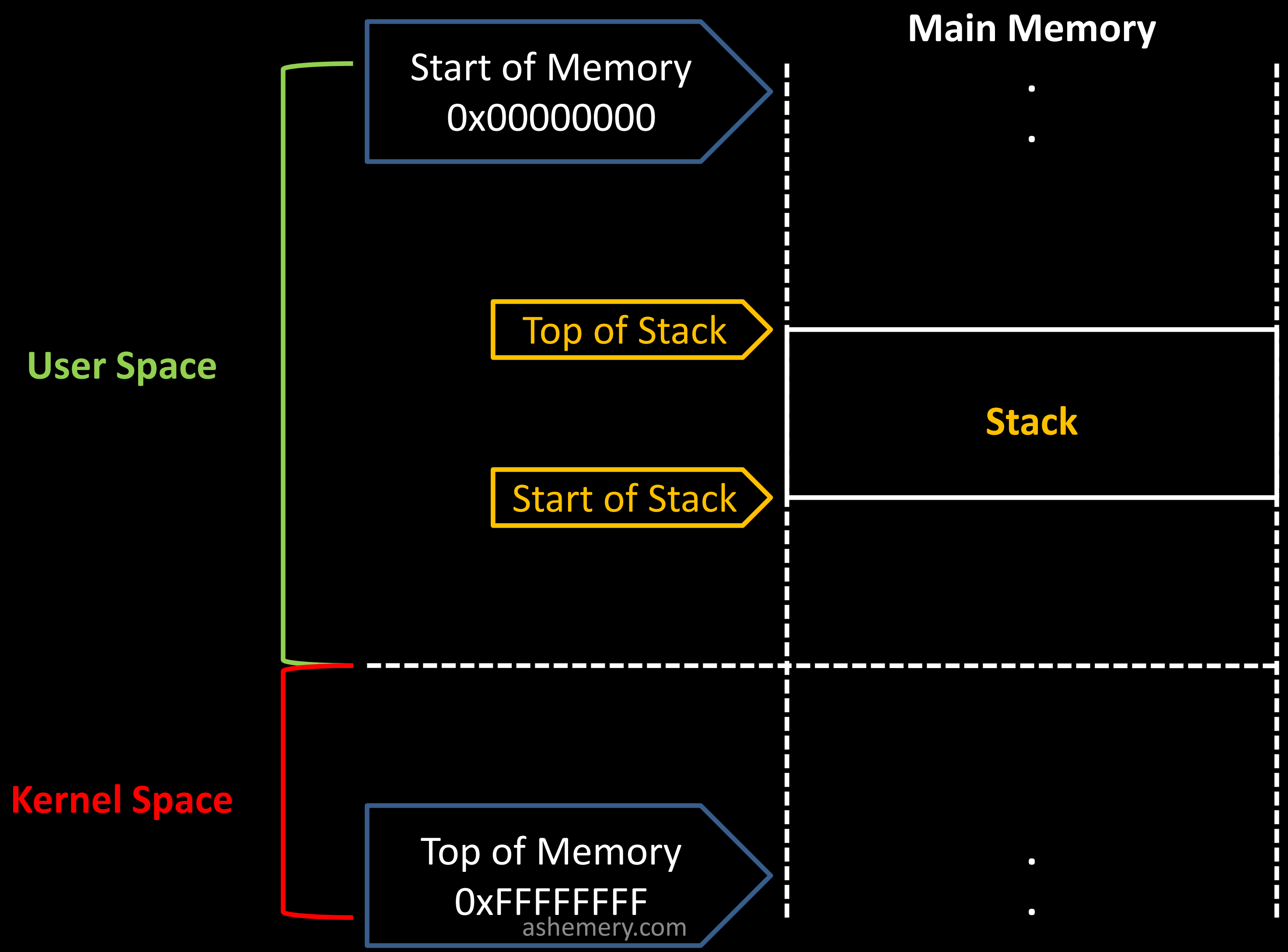
Stack & Stack Frames – Cont.

- When Pushing or Popping values, ESP register value is changed (the stack pointer moves)

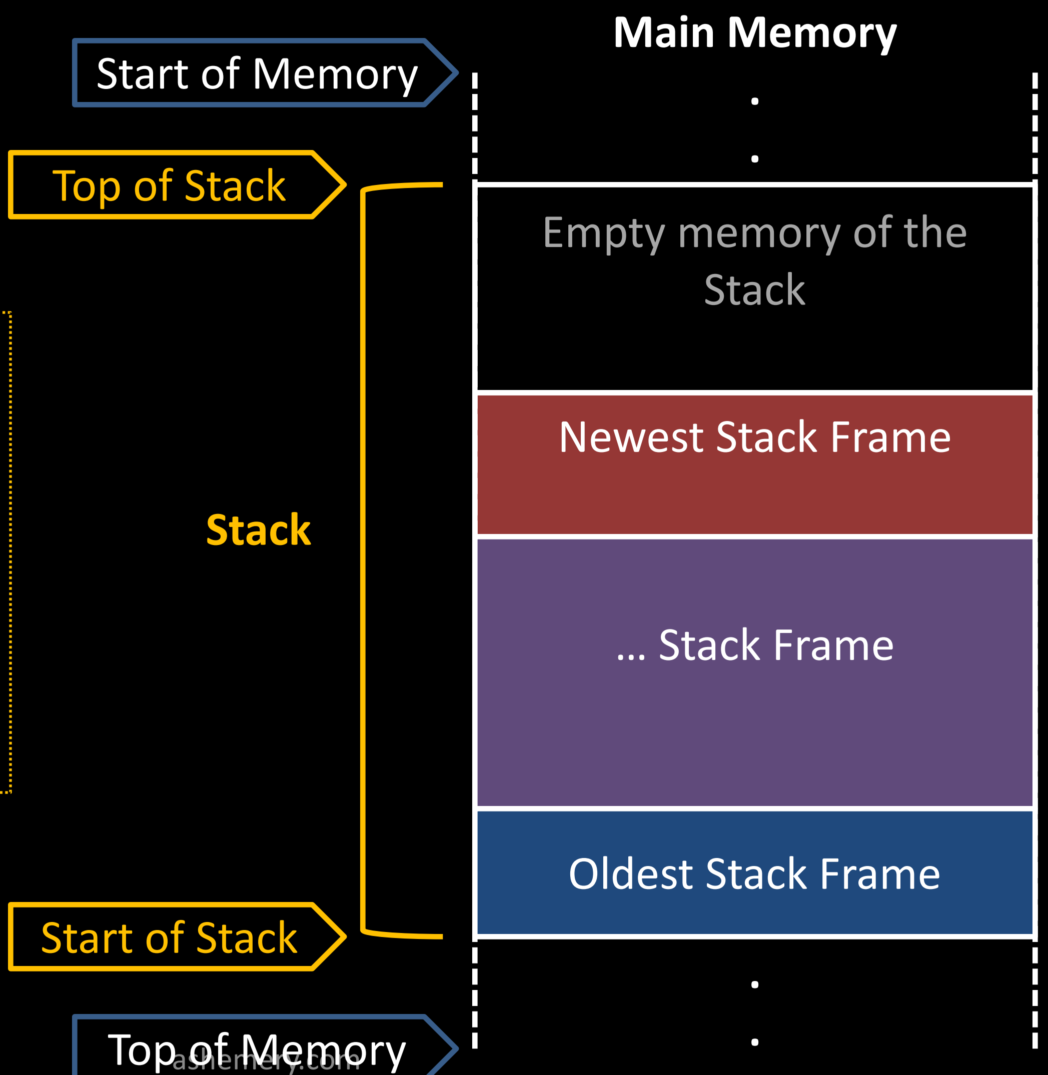
- Base Pointer (value of EBP) never changes unless the current Stack Frame is changed.

- The stack frame is empty when EBP value = ESP value.

Memory Addressing



Stack & Stack Frames inside the Main Memory



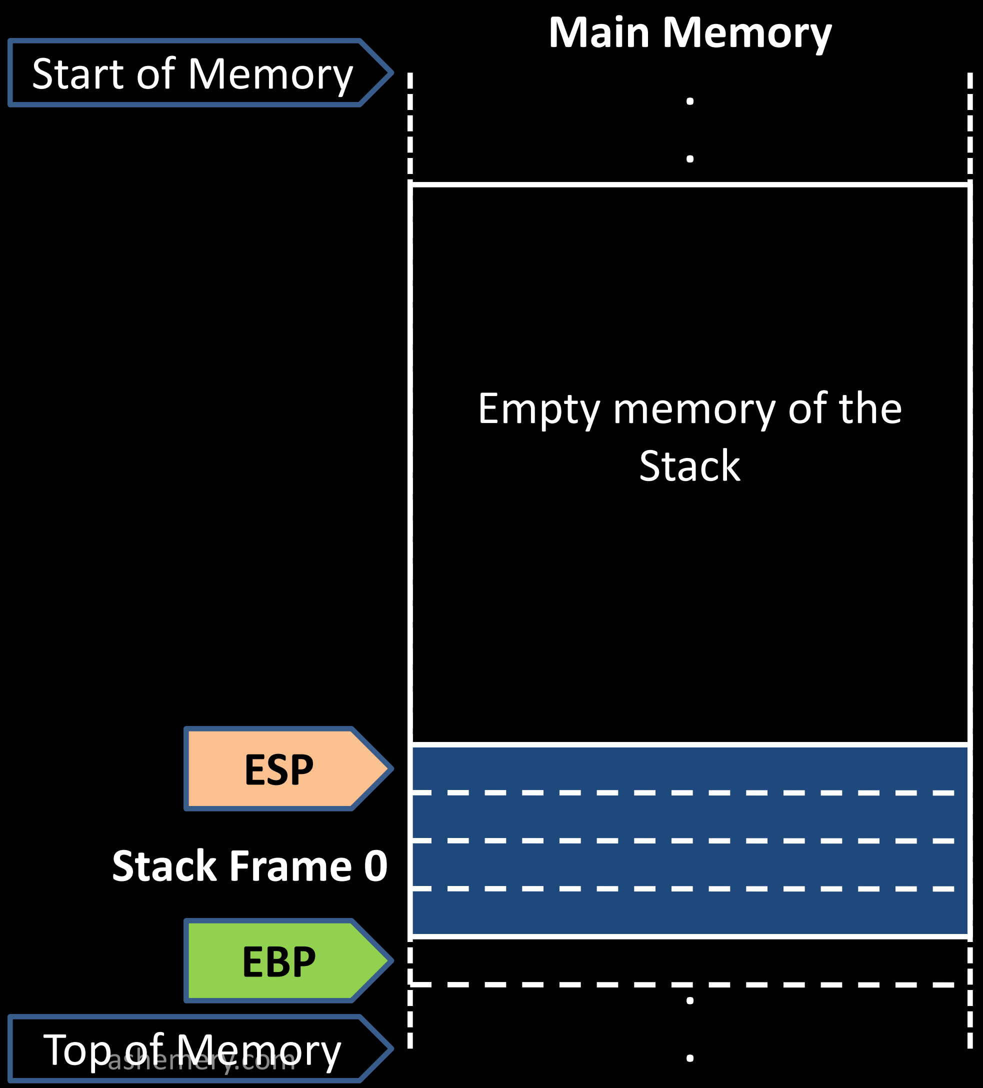
Note 1:
The newest stack frame is indexed as **Stack Frame 0**, the older one **Stack Frame 1**, And the Oldest Stack Frame is indexed **Stack Frame (count-1)**

Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

ESP points to the top of the current Stack Frame. And it points to the top of the **Stack** as well.

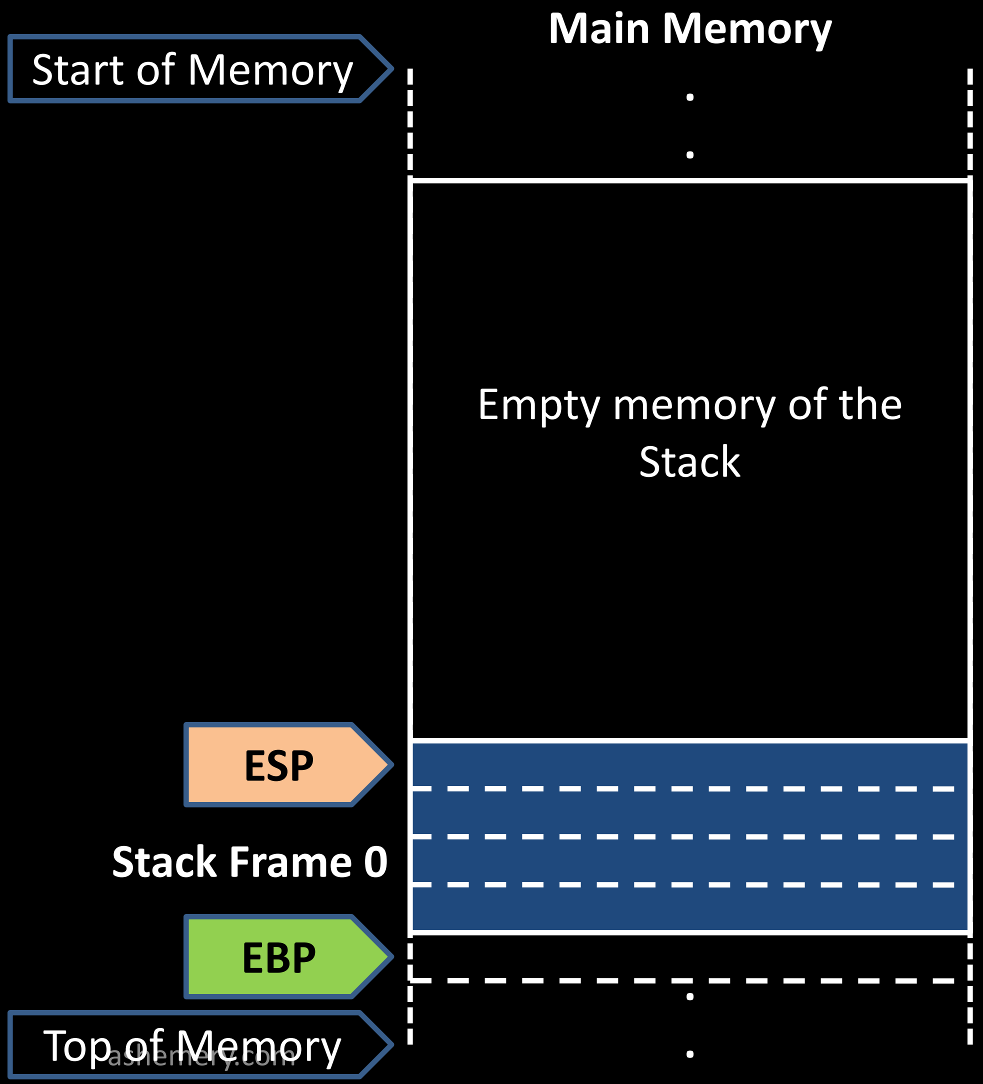
Whenever a function is called, a new Stack Frame is created. Local variables are also allocated in the bottom of the created Stack Frame.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

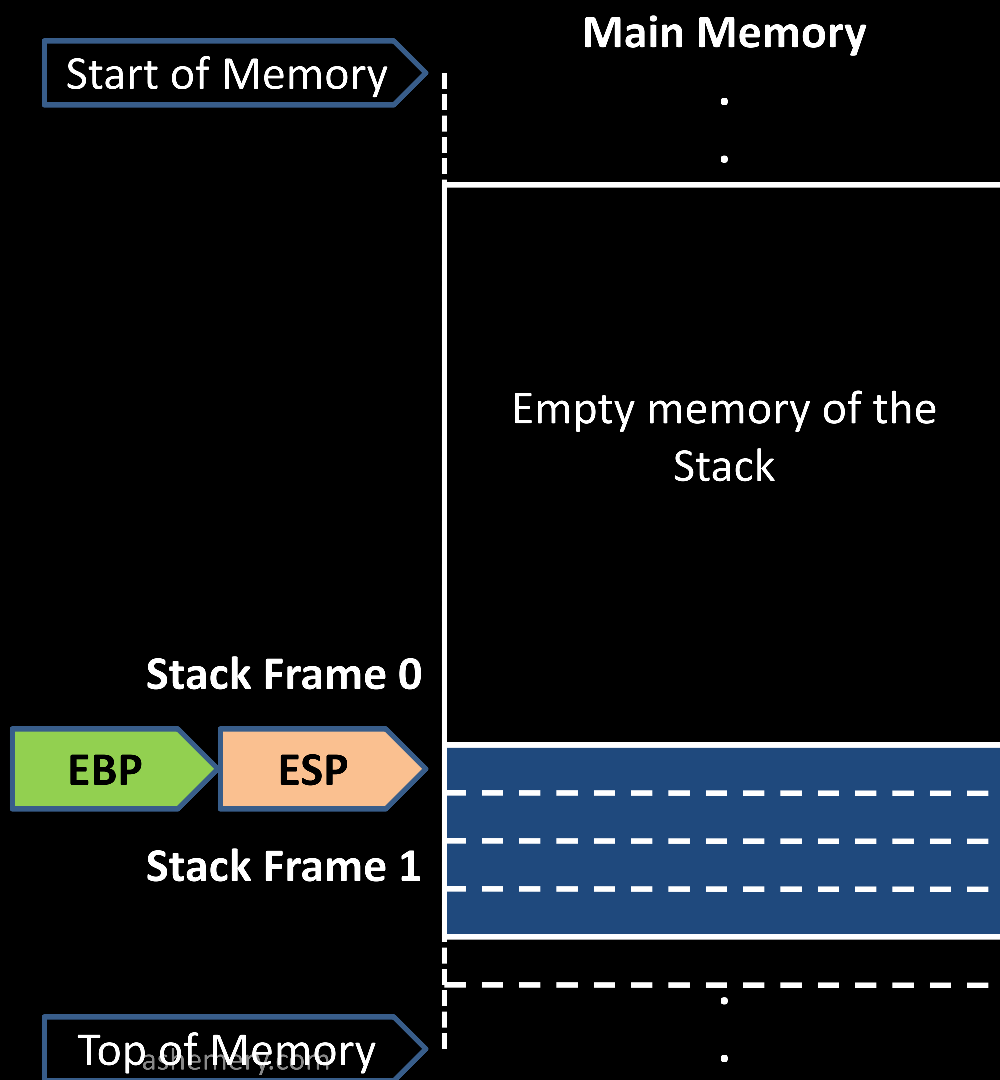
To create a new Stack Frame, simply change EBP value to be equal to ESP.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

Now EBP = ESP, this means that the Newest Stack Frame is empty. The previous stack frame now is indexed as Stack Frame 1

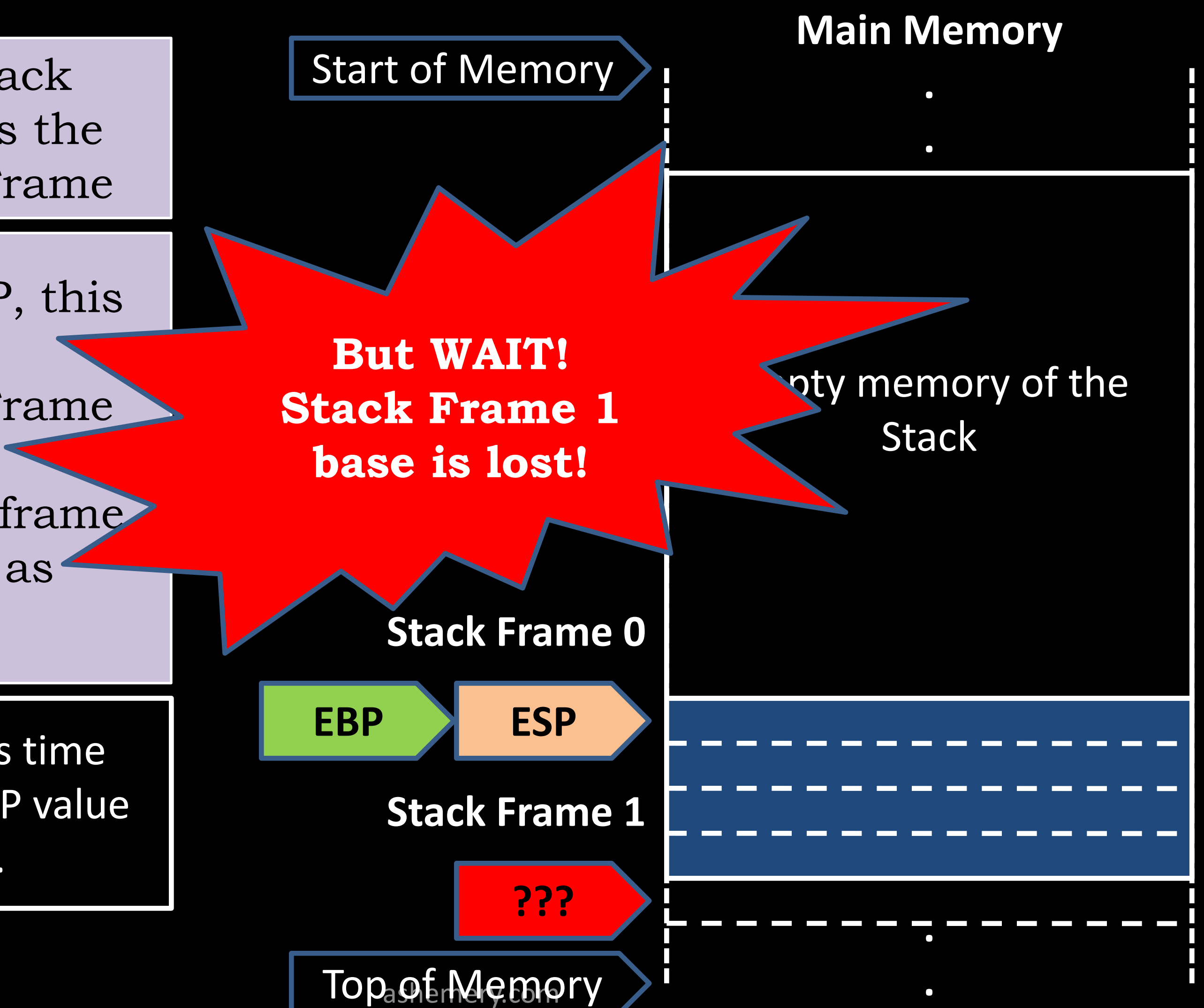


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

Now EBP = ESP, this means that the Newest Stack Frame is empty. The previous stack frame now is indexed as Stack Frame 1

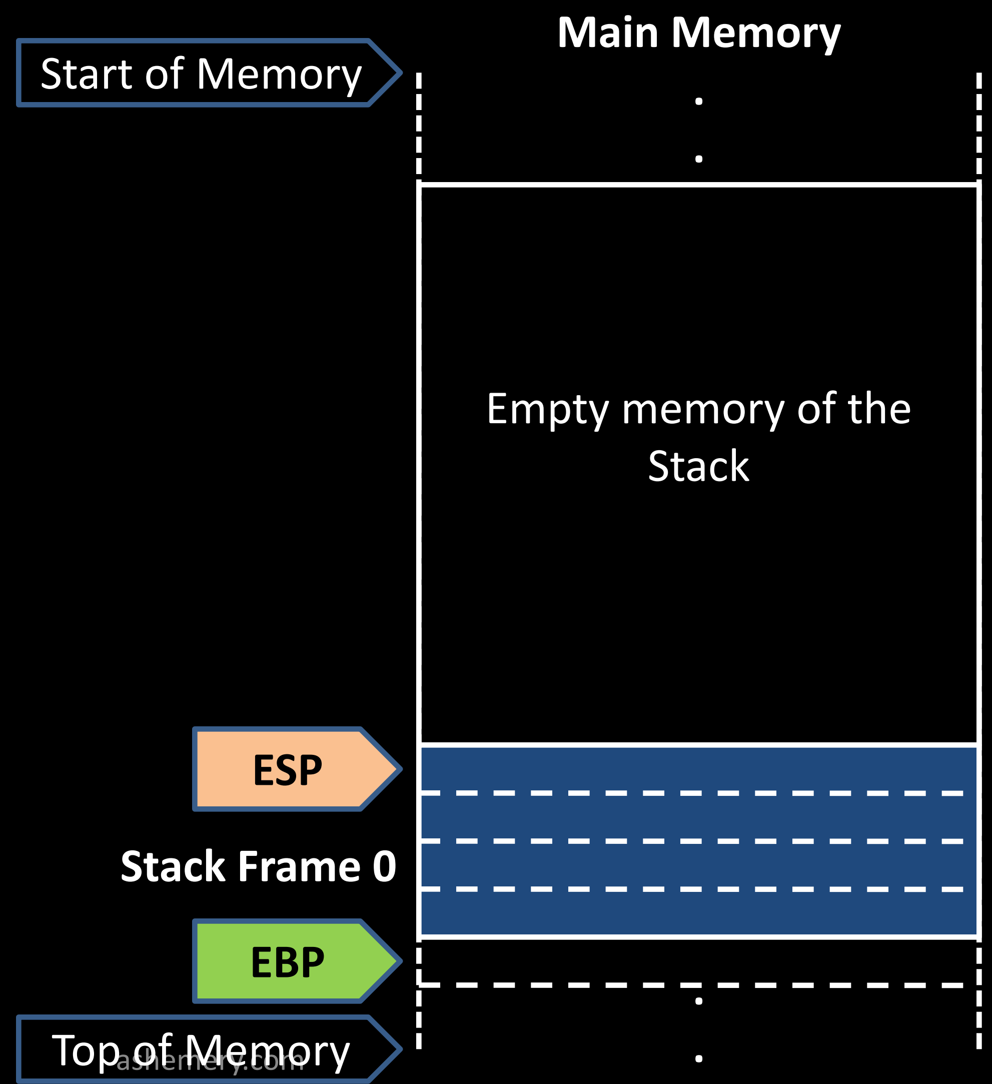
Let's try again. This time we should save EBP value before changing it.



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

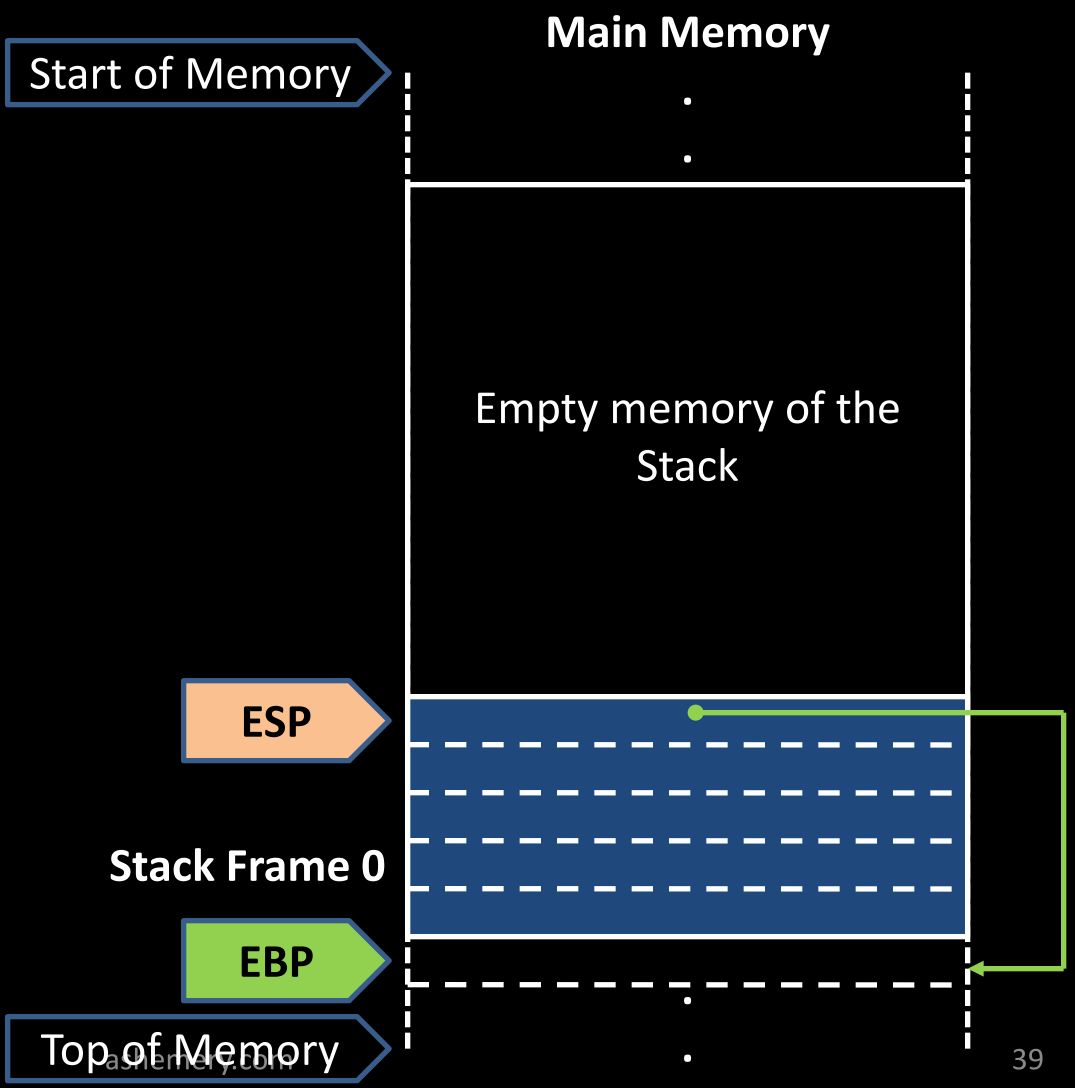


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

Now change the value of EBP.



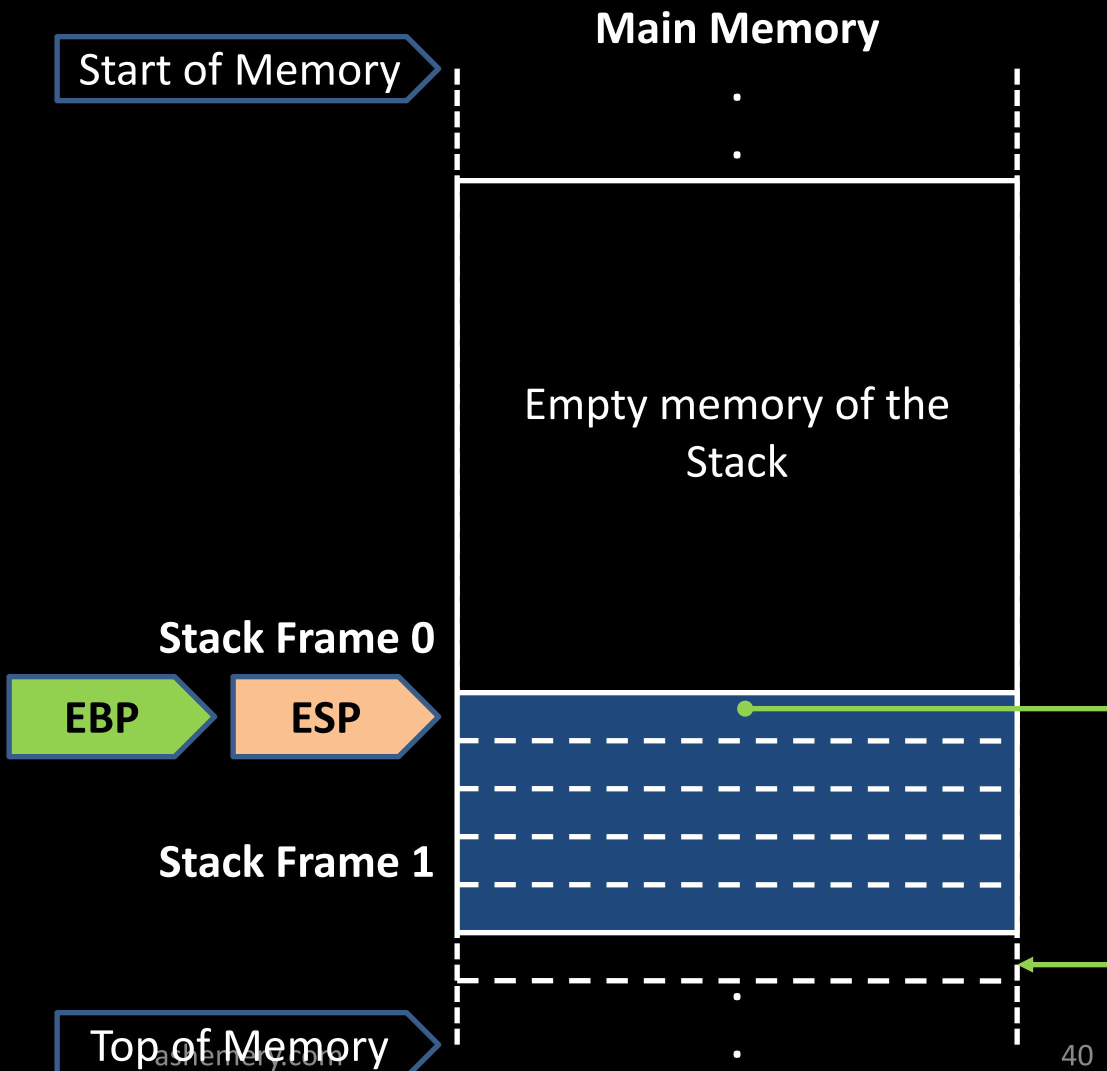
Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

First, PUSH value of EBP to save it.

Now change the value of EBP.

PROLOGUE is:
Creating new Stack Frame then allocating space for local variables.

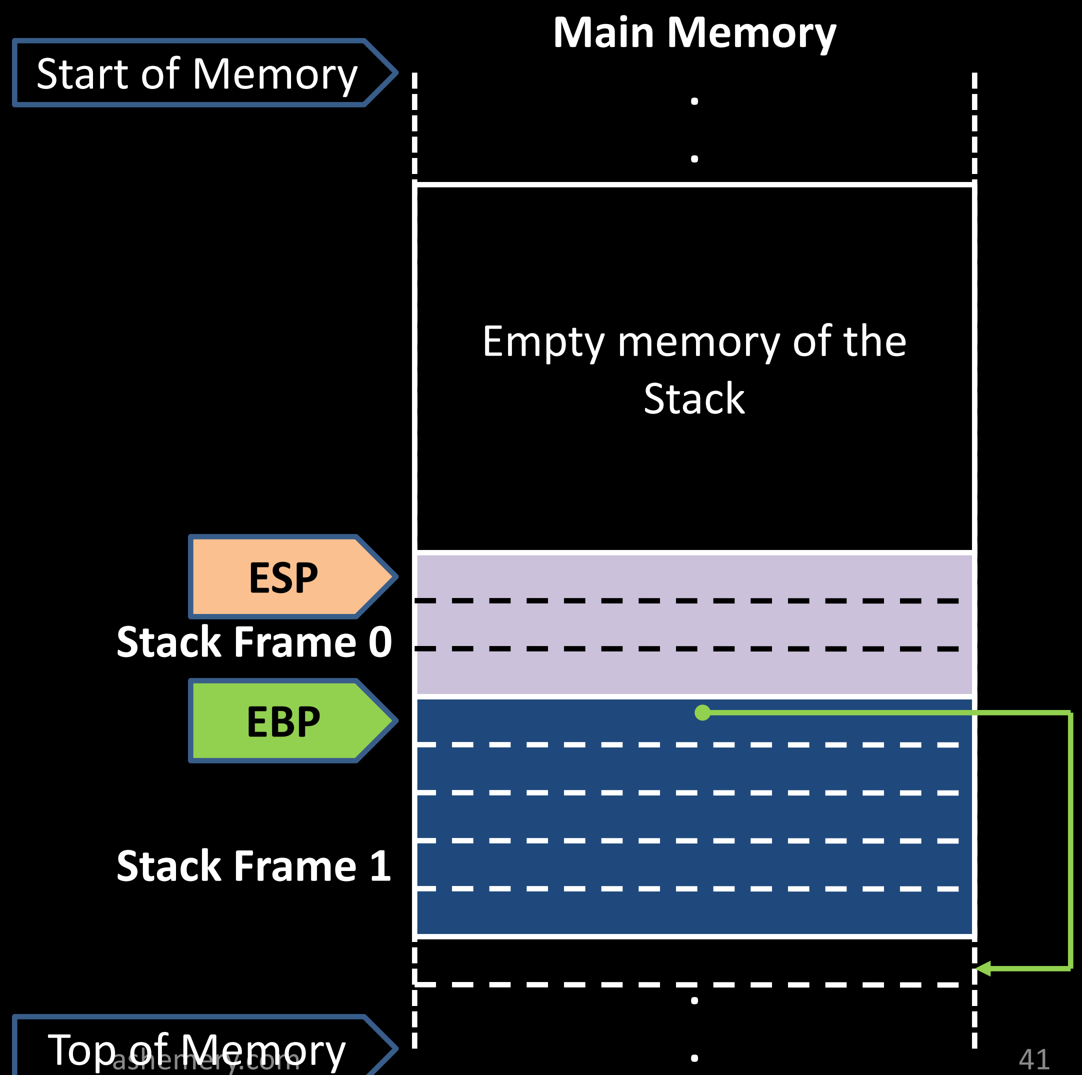


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

PUSH and POP operations affect ESP value only.

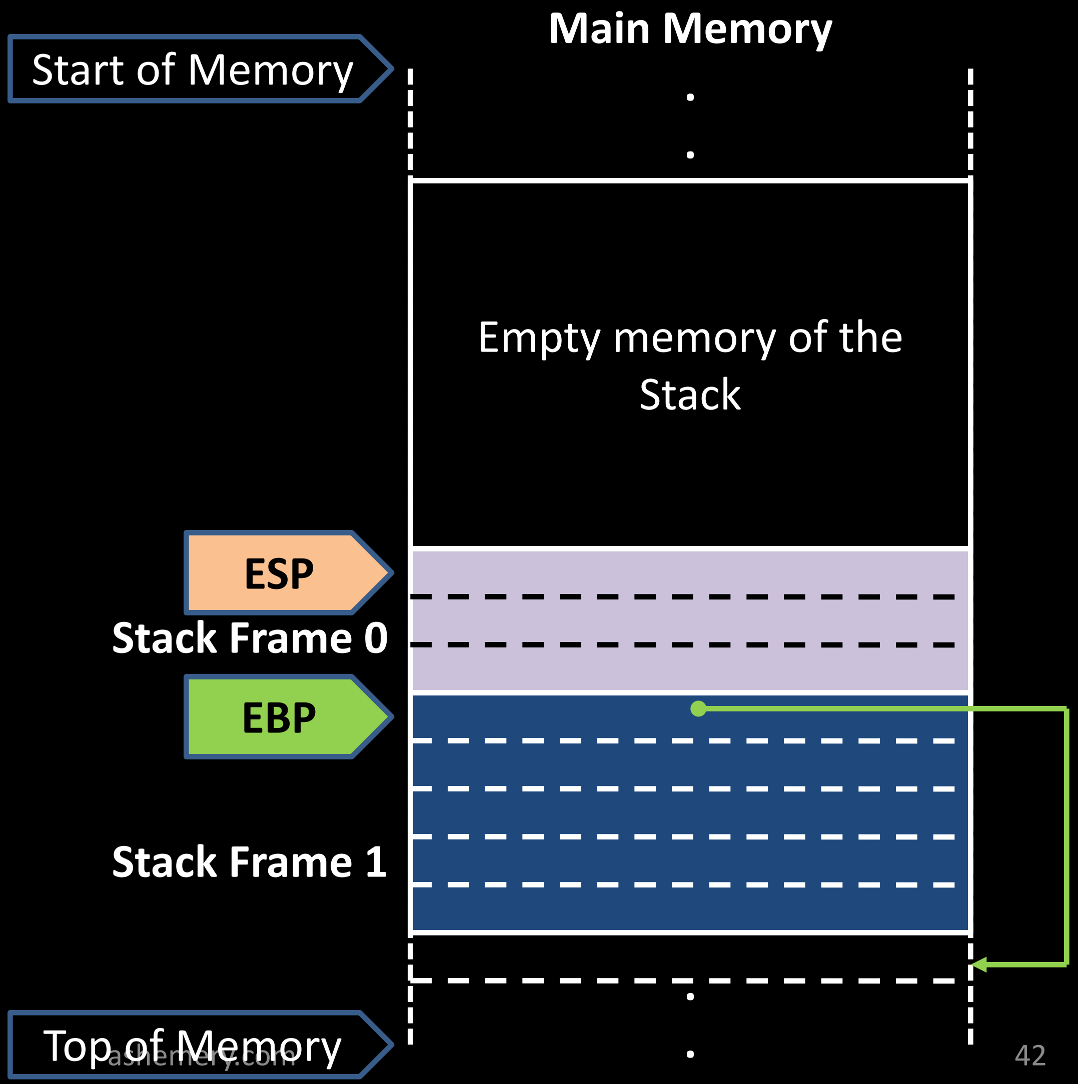
We don't need to save ESP value for the previous stack frame, because it is equal to the current EBP value



Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

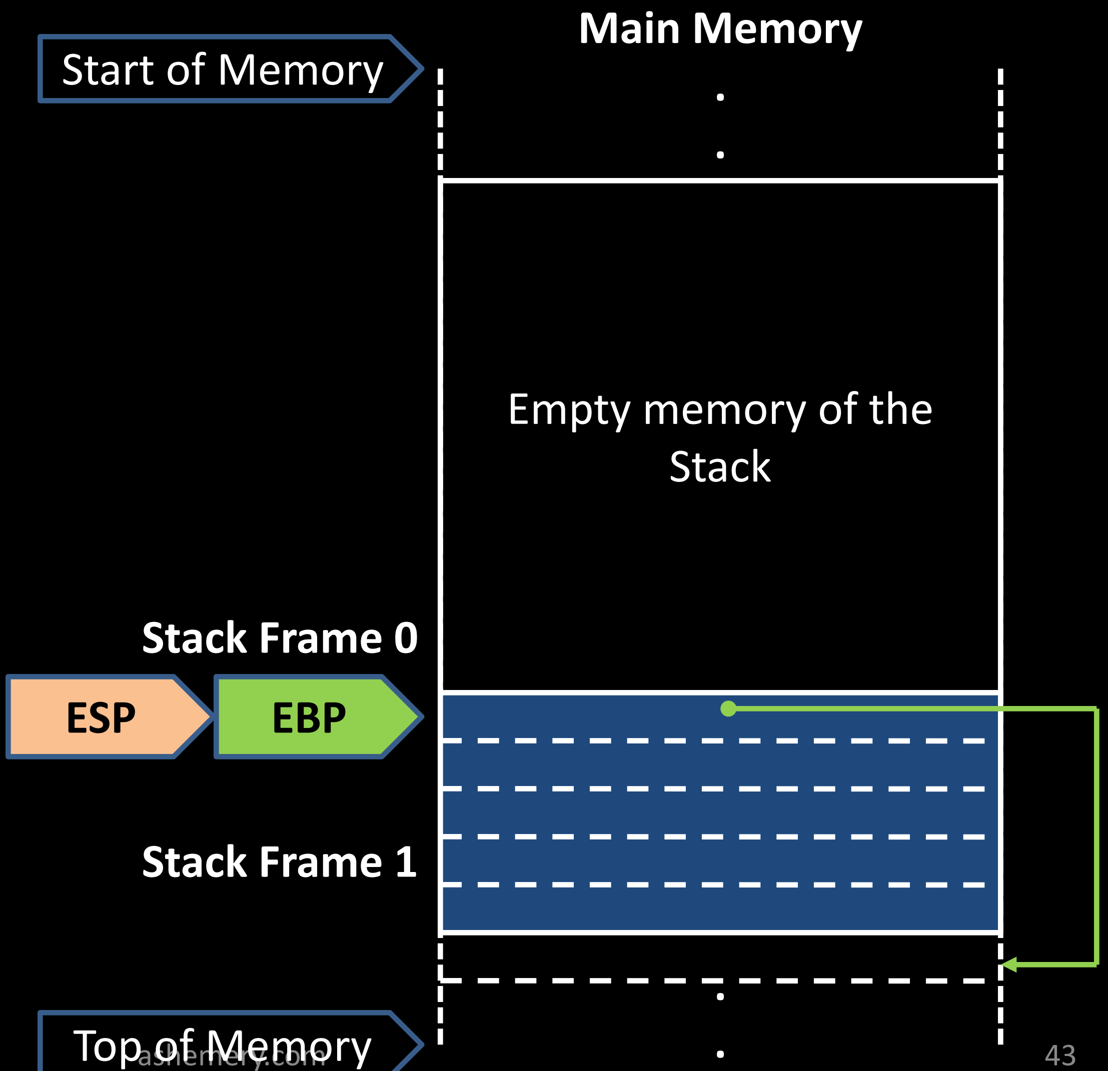


Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

To delete the current Stack Frame and return back to the previous one, we should POP out the top value from the **Stack** into EBP.



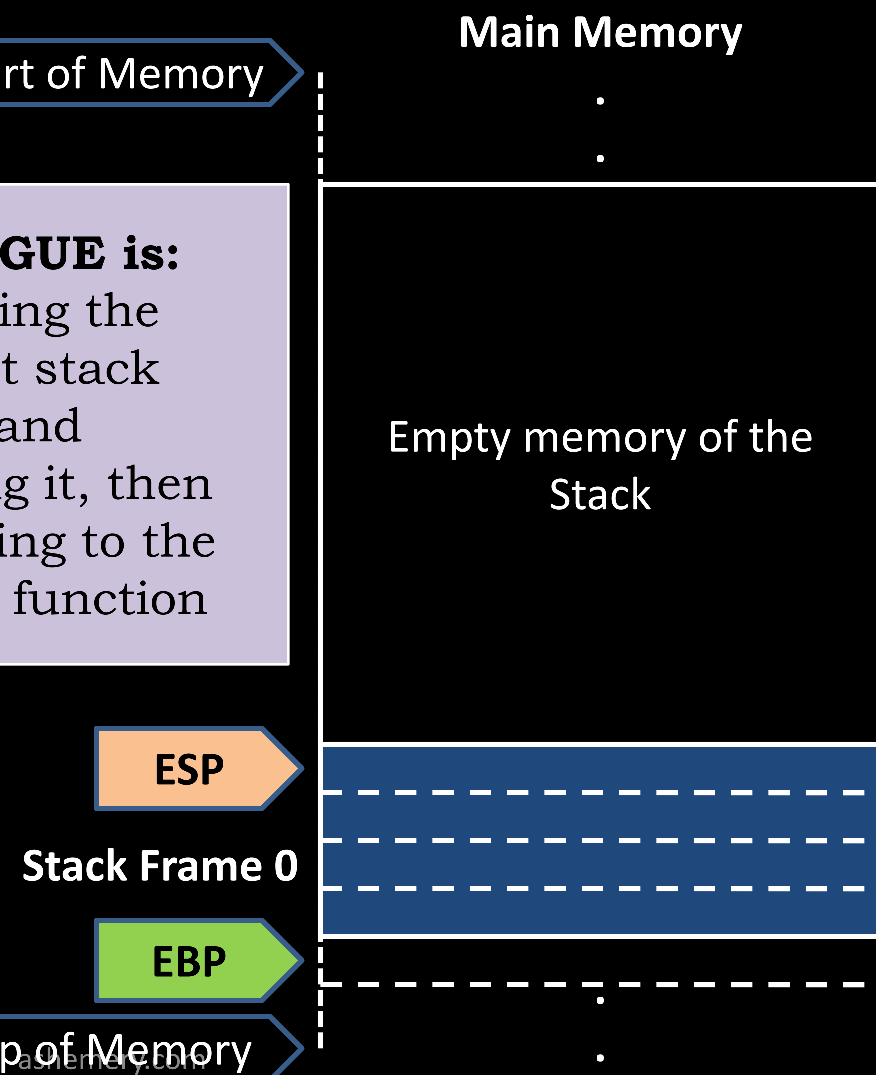
Managing Stack Frames

The Current Stack Frame is always the Newest Stack Frame

To empty out the current Stack Frame, ESP value should be set to the same value of EBP

To delete the current Stack Frame and return back to the previous one, we should POP out the top value from the **Stack** into EBP.

EPILOGUE is: Emptying the current stack frame and deleting it, then returning to the calling function



Functions: Low Level View
- Understanding the Process -

A simple function call in a high level language is not a simple operation as it seems.

`add(x, y);`

PUSH arguments (if any)	PUSH arguments (if any)
Call the function	PUSH EIP
	Jump to function’s first instruction
PROLOGUE	PUSH EBP
	Set EBP = ESP
	PUSH local variables (if any)
Execute the function	Execute the function
EPILOGUE	POP out all local variable
	POP EBP
	POP EIP
POP arguments	POP arguments

Functions: Low Level View
- Understanding the Process -

Each PUSH operation must be reversed by a POP operation somewhere in the execution

Performing (PUSH arguments) is done by the caller function. Arguments are pushed in a reverse order.

Performing (POP arguments) can be done by the caller or the callee function. This is specified by the (call type) of the callee function

Return value of the callee is saved inside EAX register while executing the function's body

- PUSH arguments (if any)
- PUSH EIP
- Jump to function's first instruction
- PUSH EBP
- Set EBP = ESP
- PUSH local variables (if any)
- Execute the function
- POP out all local variable
- POP EBP
- POP EIP
- POP arguments

Functions, Low Level View
- Call Types -

- Programming languages provide a mechanism to specify the call type of the function.
- (Call Type) is not (Return Value Type).
- The caller needs to know the call type of the callee to specify how arguments should be passed and how Stack Frames should be cleaned.
- There are many call types; two of them are commonly used in most programming languages:
  - cdecl: the default call type for C functions. The caller is responsible of cleaning the stack frame.
  - stdcall: the default call type for Win32 APIs. The callee is responsible of cleaning the stack frame.

Other(s)

- Some call types use deferent steps to process the function call. For example, fastcall send arguments within Registers not by the stack frame. (Why?)

Functions: Low Level View
- Assembly Language -

Each of these steps are processed by one or many instructions.

As like as other programming languages; assembly provides many ways to perform the same operation. Therefore, the disassembled code can vary from one compiler to another.

Now we are going to introduce the default way for performing each of these steps using assembly language.

- PUSH arguments (if any)
- PUSH EIP
- Jump to function's first instruction
- PUSH EBP
- Set EBP = ESP
- PUSH local variables (if any)
- Execute the function
- POP out all local variable
- POP EBP
- POP EIP
- POP arguments

Functions: Low Level View
- Assembly Language -

	cdecl	stdcall	
caller	`push <arg2>` `push <arg1>`	`push <arg2>` `push <arg1>`	PUSH arguments (if any)
caller	`call <callee>`	`call <callee>`	PUSH EIP Jump to function’s first instruction

callee	`push ebp`
	`mov ebp, esp`
	`push <default value of local variable>`
		`push ebp`
		`mov ebp, esp`
		`push <default value of local variable>`
			PUSH EBP
			Set EBP = ESP
			PUSH local variables (if any)

			Execute the function

callee	`pop ecx`
	`pop ebp`
	`ret`
		`pop ecx`
		`pop ebp`
		`ret <args size>`
			POP out all local variable
			POP EBP
			POP EIP

caller	`pop ecx` `pop ecx`		POP arguments

Functions: Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

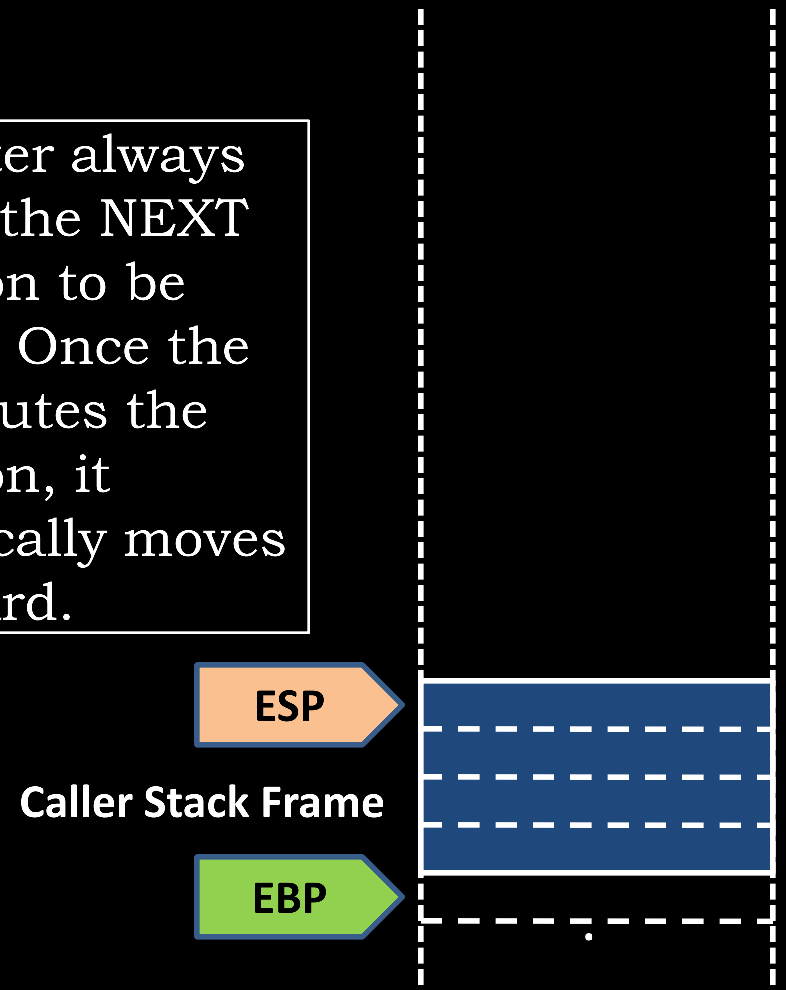
```
push    ebp
mov     ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

EIP

EIP register always points to the NEXT instruction to be executed. Once the CPU executes the instruction, it automatically moves EIP forward.



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

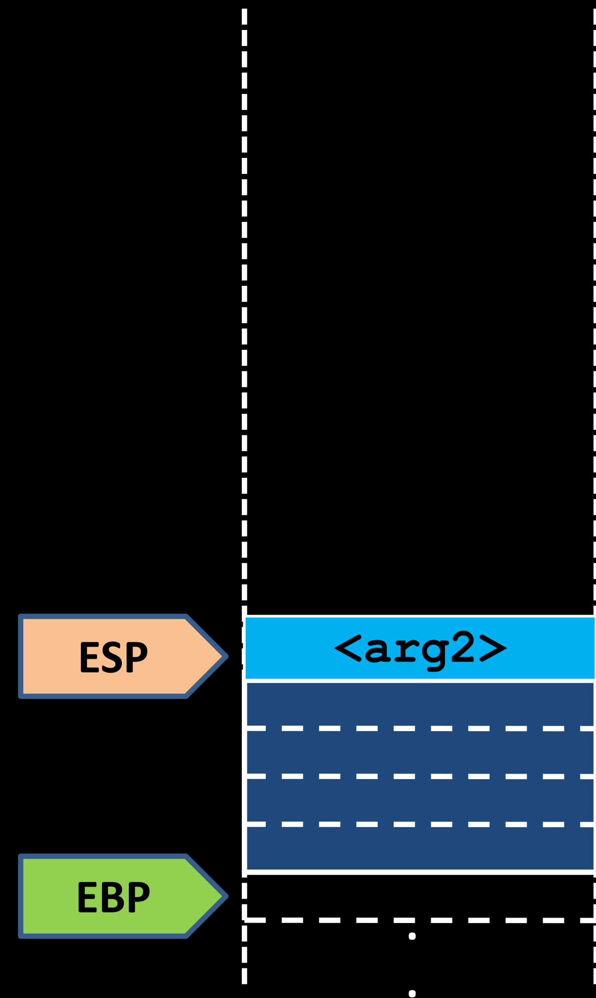
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

EIP



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

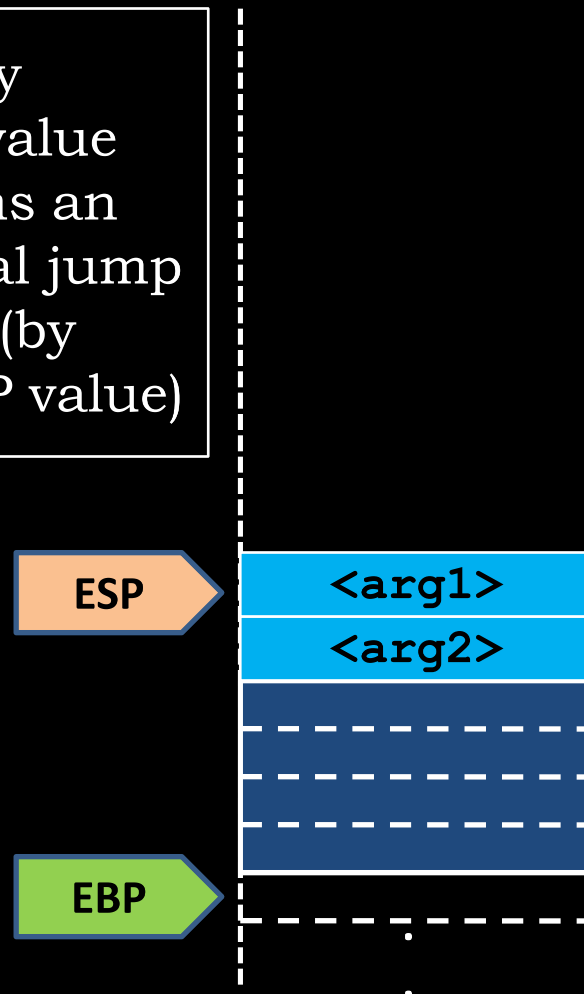
EIP

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

(**call**) actually pushes EIP value then performs an unconditional jump to the callee (by changing EIP value)



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

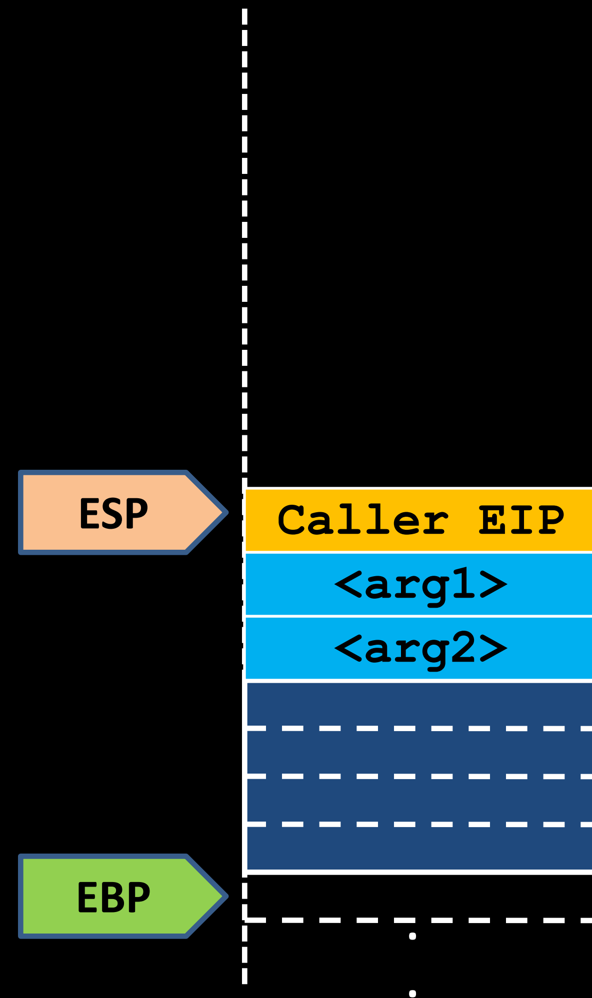
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

EIP

```
pop    ecx
pop    ebp
ret
```

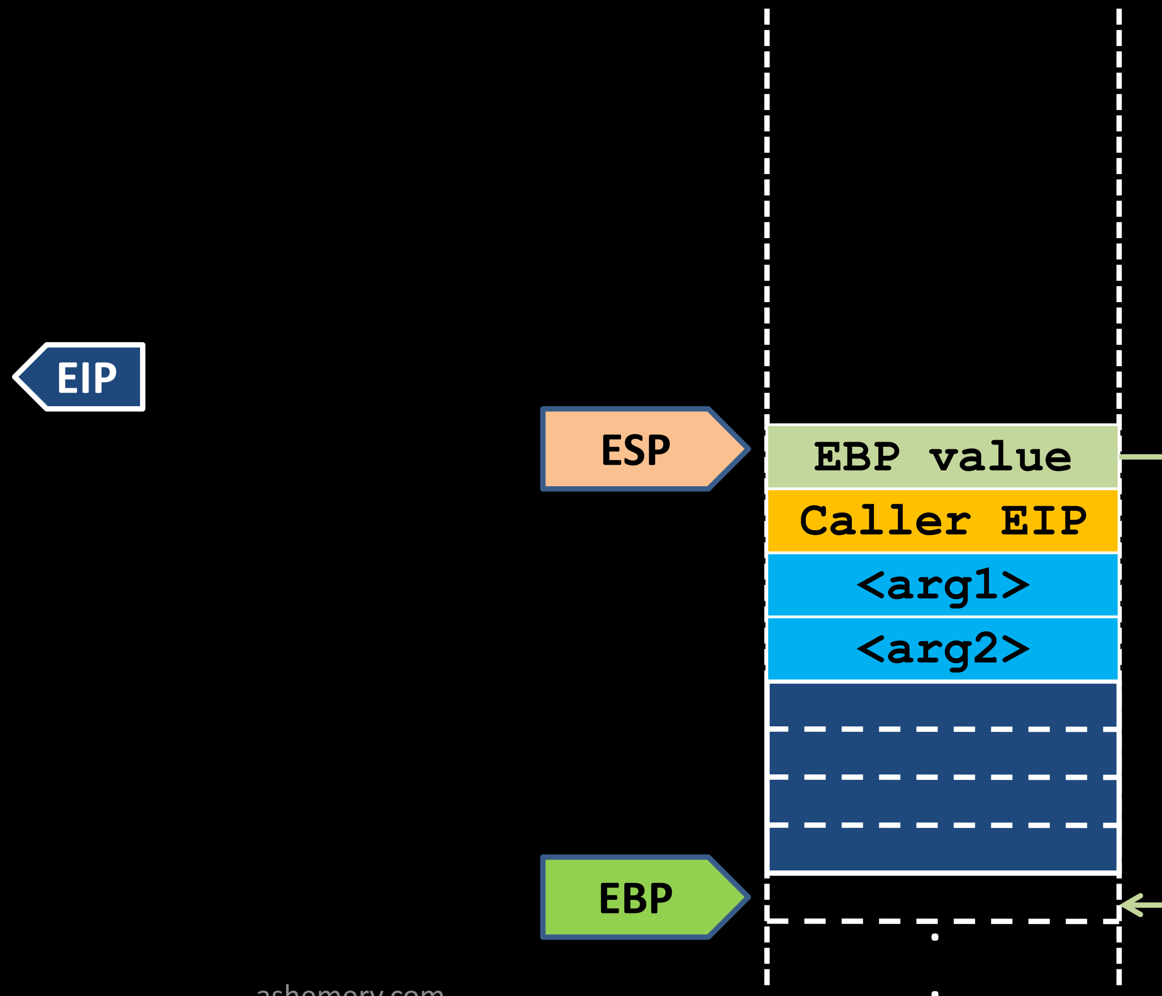
```
pop    ecx
pop    ecx
```



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>

call   <callee>

push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>

pop    ecx
pop    ebp
ret

pop    ecx
pop    ecx
```



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

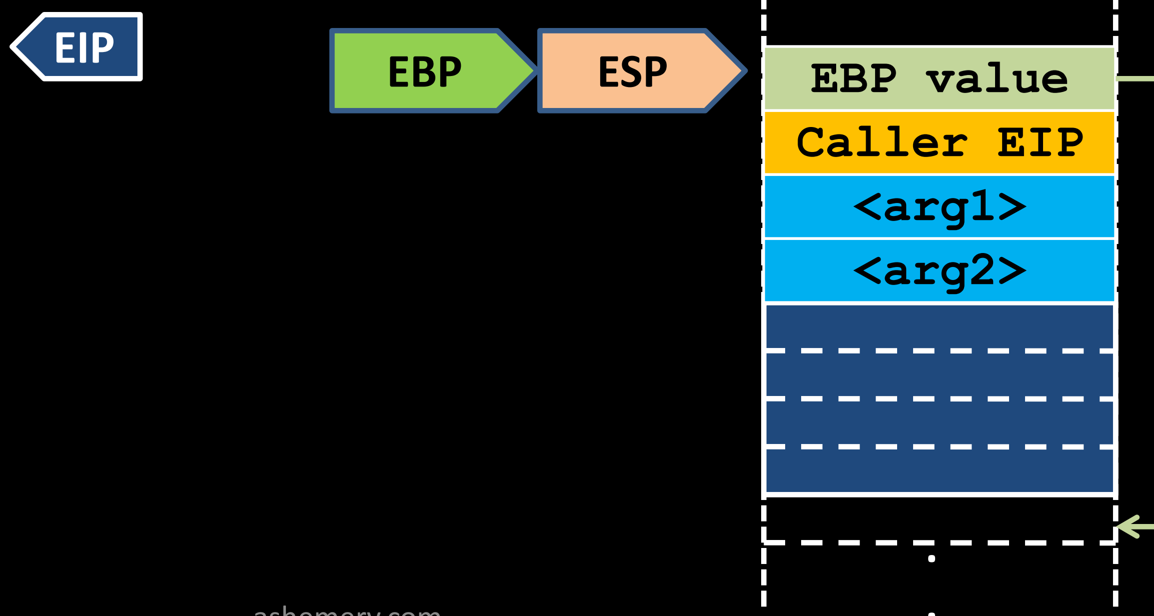
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

Let's say we have one local variable of type int.



Functions: Low Level View - General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

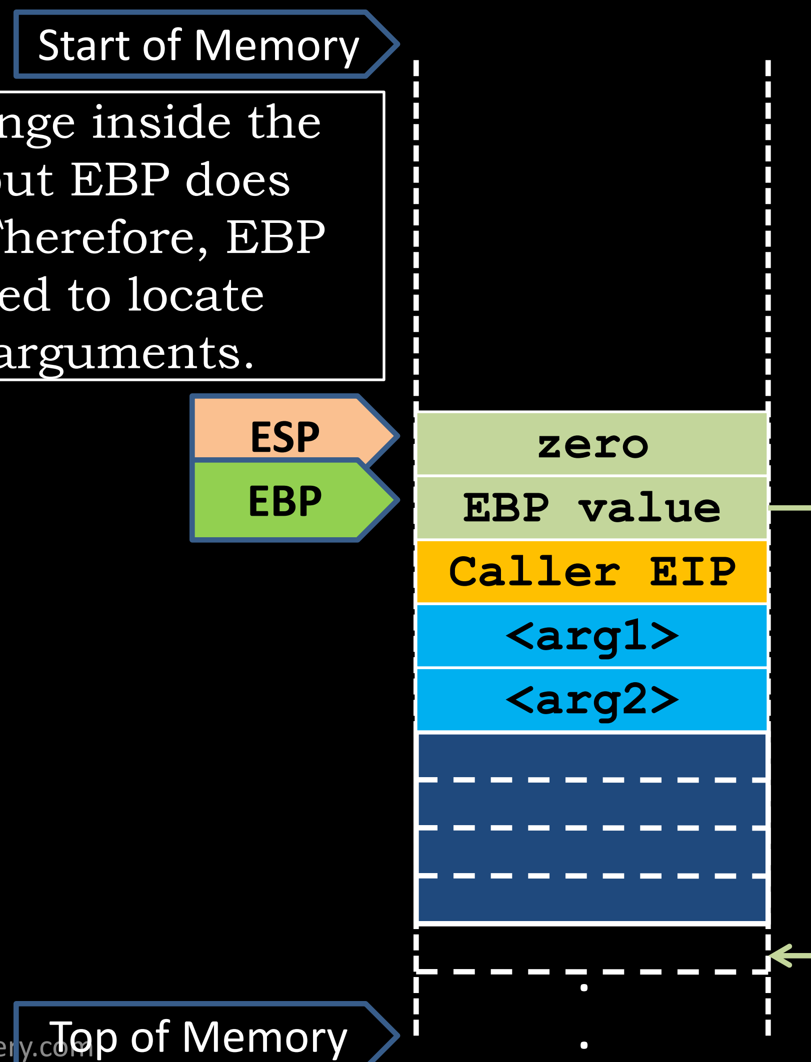
```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push    <default
        value of
        local
        variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

ESP may change inside the callee body, but EBP does not change. Therefore, EBP location is used to locate variable and arguments.



Functions: Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

```
call    <callee>
```

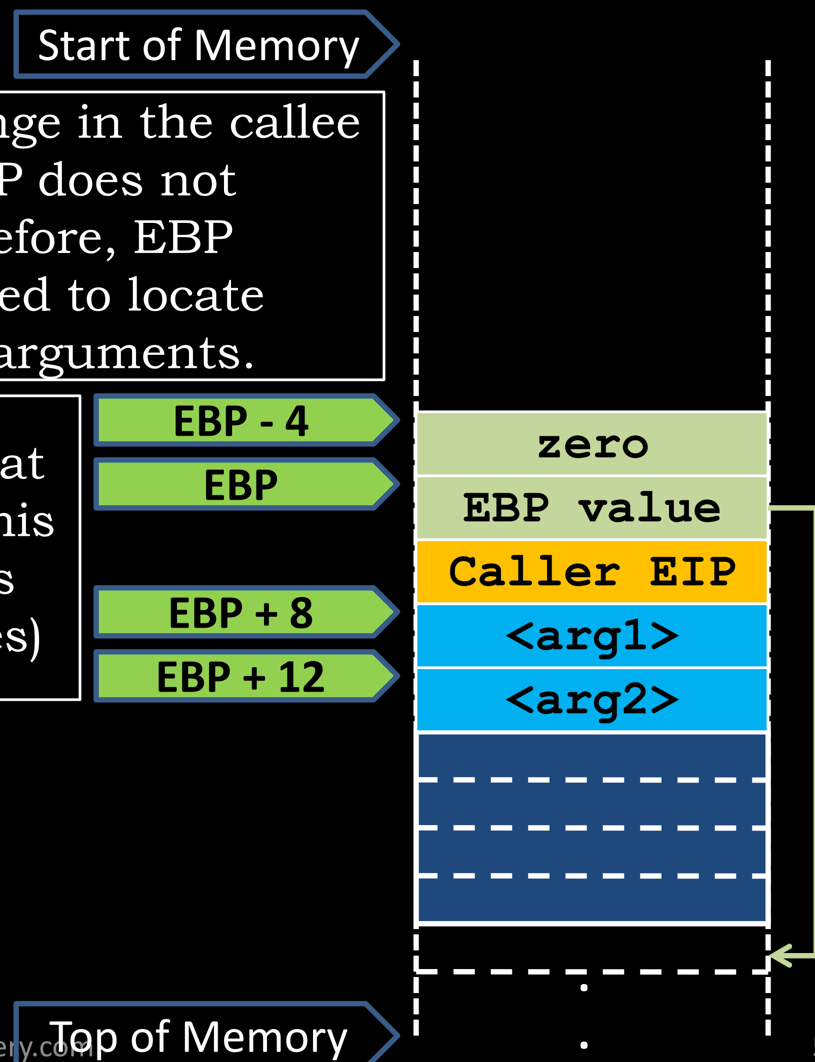
```
push    ebp
mov     ebp, esp
push    <default
        value of
        local
        variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

ESP can change in the callee body, but EBP does not change. Therefore, EBP location is used to locate variable and arguments.

Remember that each row of this stack graph is 32bits (4 bytes)



Functions: Low Level View
- General Trace -

cdecl

```
push    <arg2>
push    <arg1>
```

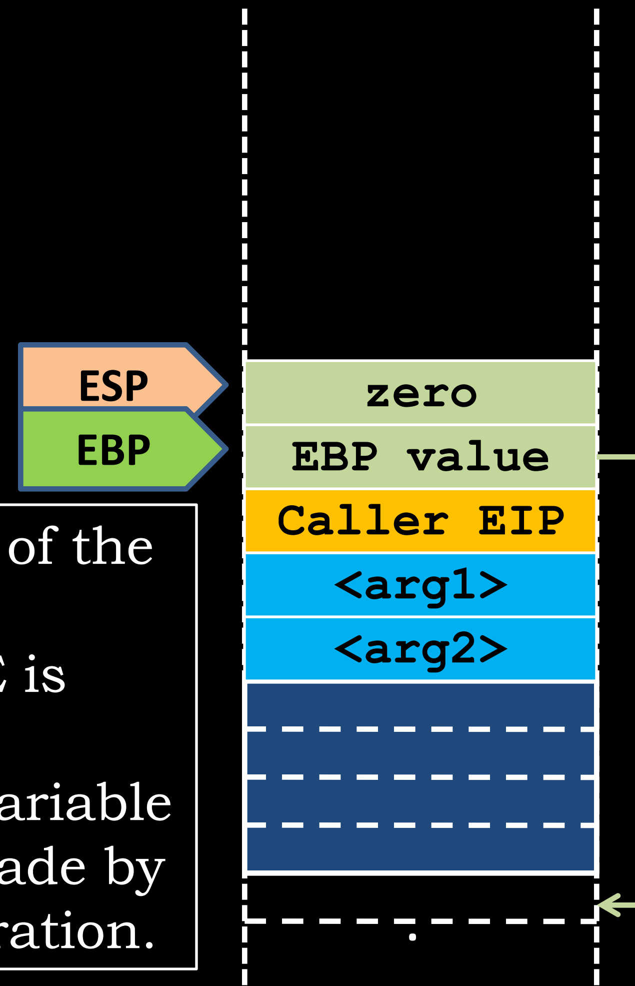
```
call    <callee>
```

```
push    ebp
mov     ebp, esp
push    <default
         value of
         local
         variable>
```

```
pop     ecx
pop     ebp
ret
```

```
pop     ecx
pop     ecx
```

At the end of the callee, EPILOGUE is processed. Cleaning variable space is made by a POP operation.



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

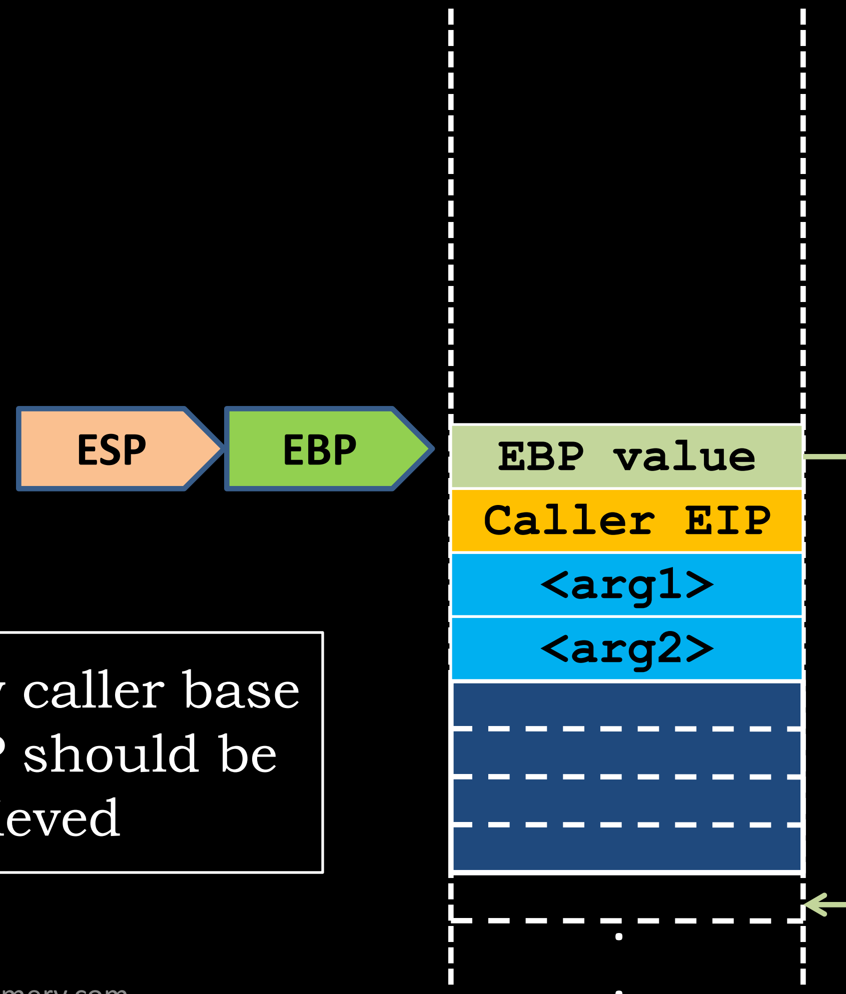
```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

EIP

```
pop    ecx
pop    ecx
```

Now caller base EBP should be retrieved



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

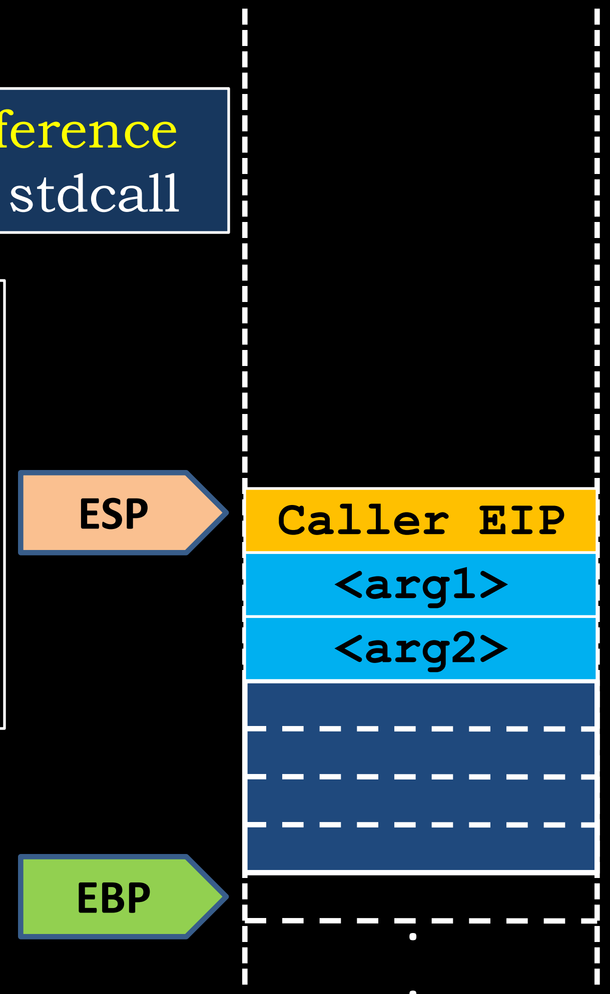
```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

Here comes the deference between cdecl and stdcall

`ret` instruction simply pops a value from the stack and save it in EIP, that should direct the execution back to the caller

EIP



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

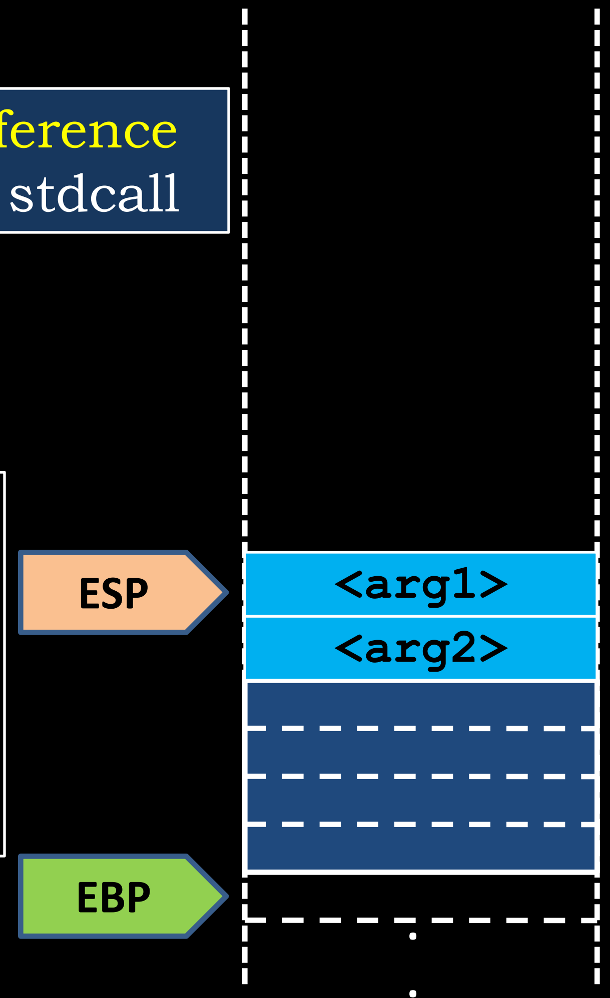
```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

Here comes the deference between cdecl and stdcall

Now the caller is responsible of cleaning the stack from passed arguments using POP operations.

EIP



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

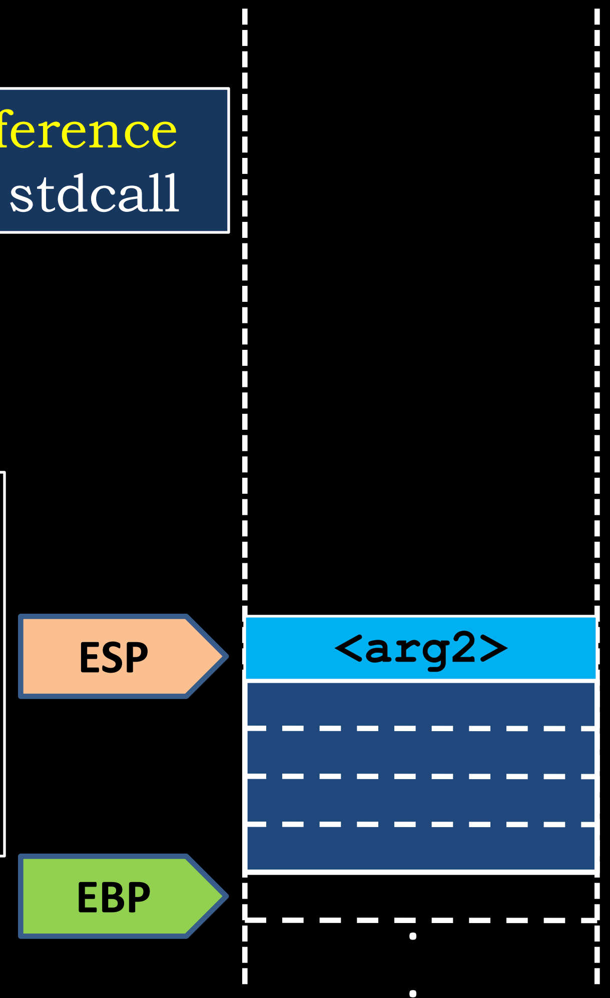
```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

EIP

Here comes the deference between cdecl and stdcall

Now the caller is responsible of cleaning the stack from passed arguments using POP operations.



Functions: Low Level View
- General Trace -

cdecl

```
push   <arg2>
push   <arg1>
```

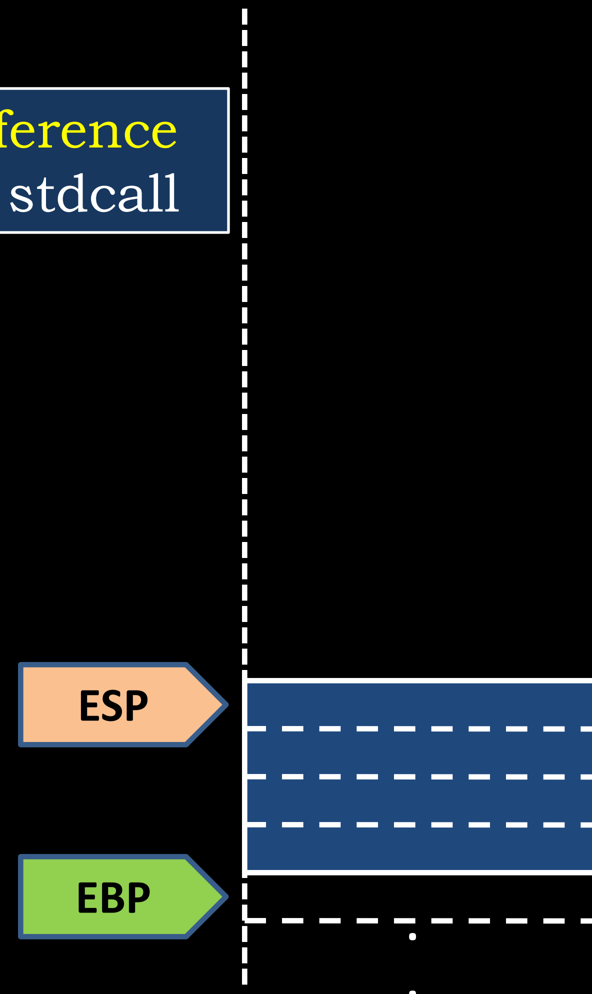
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop    ecx
pop    ebp
ret
```

```
pop    ecx
pop    ecx
```

Here comes the deference between cdecl and stdcall



Functions: Low Level View
- General Trace -

stdcall

```
push   <arg2>
push   <arg1>
```

```
call   <callee>
```

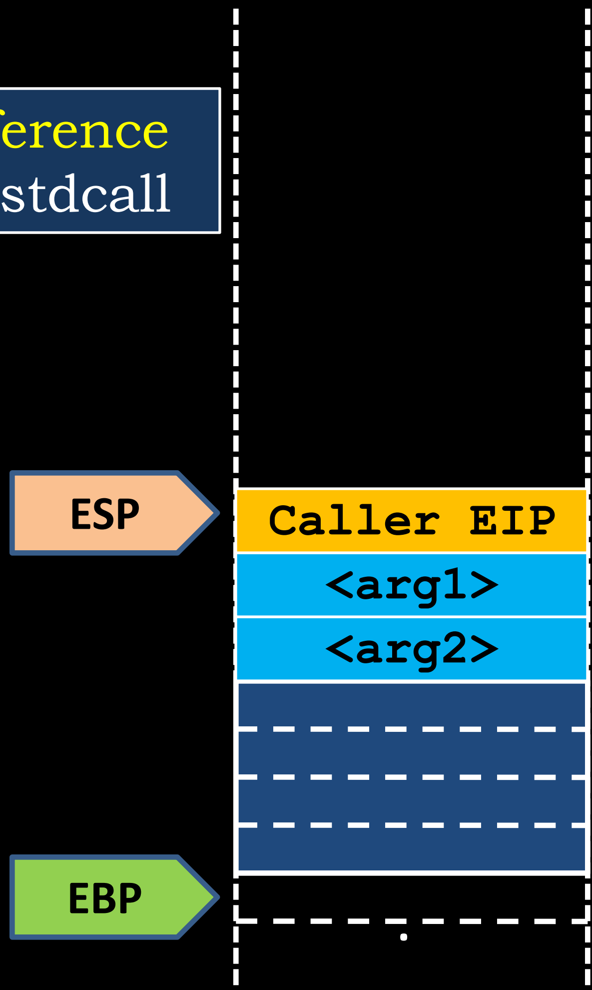
```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop    ecx
pop    ebp
ret    <args size>
```

EIP

Here comes the deference between cdecl and stdcall

`ret` instruction proceeded by an integer value will add that value to ESP <u>immediately after</u> performing POP EIP



Functions: Low Level View
- General Trace -

stdcall

```
push   <arg2>
push   <arg1>
```

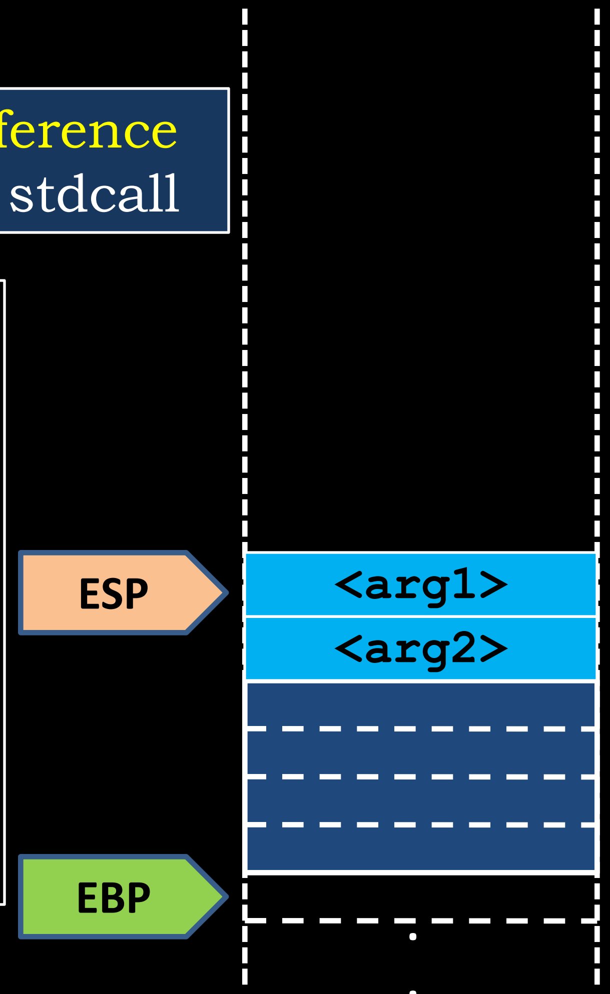
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret    <args size>
```

Here comes the deference between cdecl and stdcall

Now EIP is changed, but the CPU did not finish executing the instruction. It will add <args size> value to ESP.
In this example, we have two 32bits arguments (8 bytes)



Functions: Low Level View
- General Trace -

stdcall

```
push   <arg2>
push   <arg1>
```

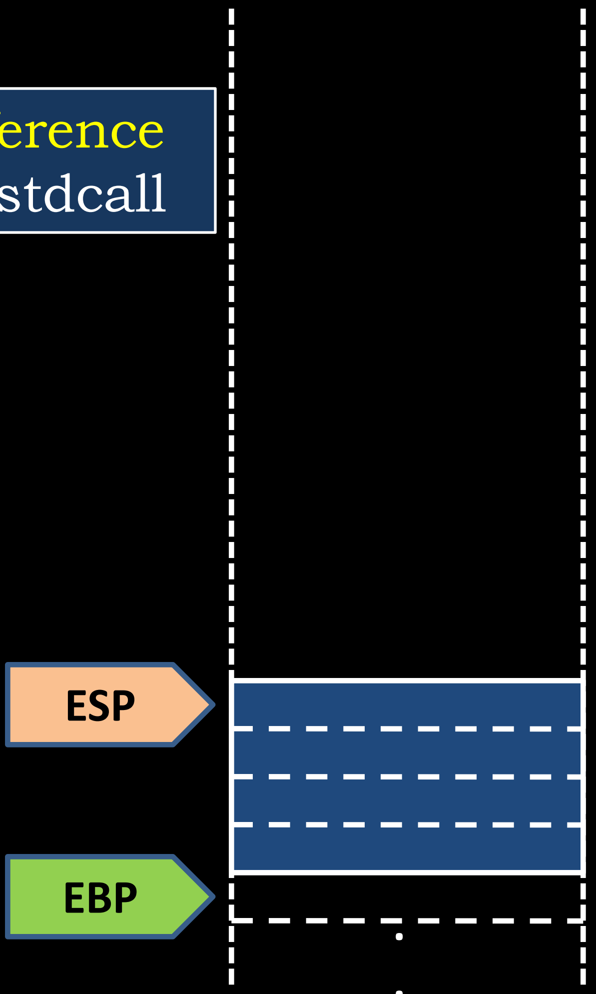
```
call   <callee>
```

```
push   ebp
mov    ebp, esp
push   <default
        value of
        local
        variable>
```

```
pop    ecx
pop    ebp
ret    <args size>
```

Here comes the deference between cdecl and stdcall

The stack has been cleaned by the callee. Now execution is back to the caller.



Functions: Low Level View
- Code Optimization -

- Compilers do not generate the default code like previous example. They use intelligent methods to optimize the code to be smaller and faster.
- For example, instructions `mov` and `xor` can be used to set EAX register to zero, but xor is smaller as a code byte. Therefore, compilers use `xor` instead of `mov` for such scenarios:
  - mov eax, 0 → code bytes: `B8 00 00 00 00`
  - xor eax, eax → code bytes: `3C 00`
- Discussing code optimization is out of the scope of this course, but we are going to discuss few tricks that you will see in the code generated by GCC for our examples.

Functions: Low Level View
- Code Optimization -

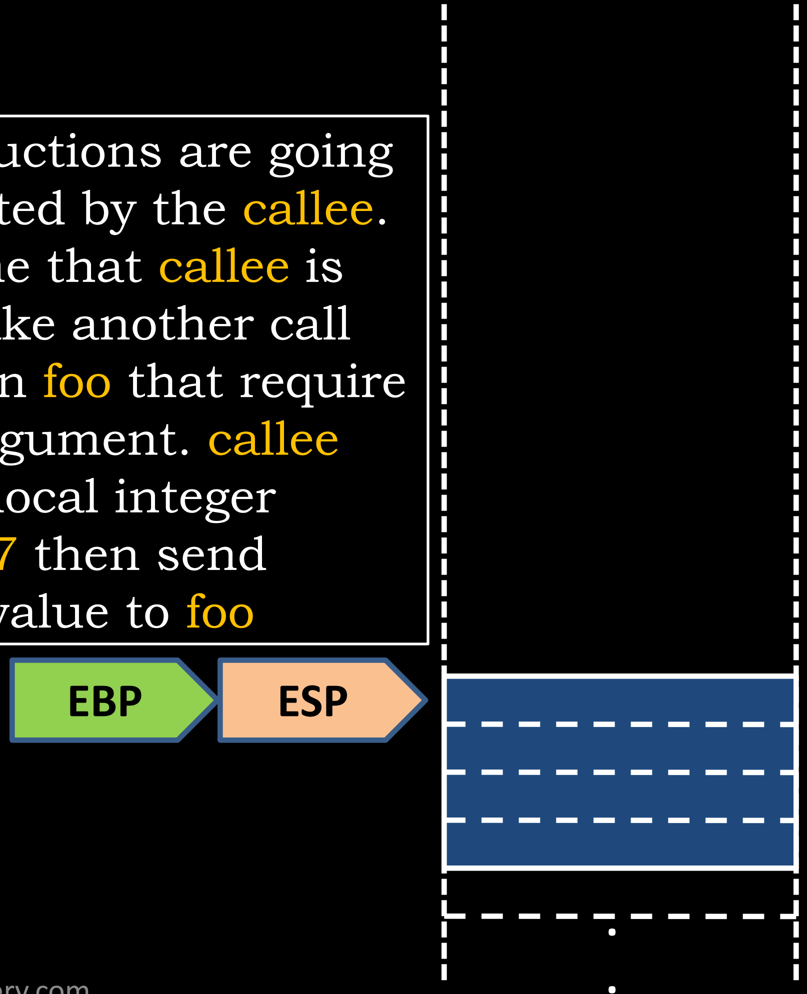
cdecl

```
push   ebp
mov    ebp, esp
push  <default
       value of
       local
       variable>
```

```
pop    ecx
pop    ebp
ret
```

EIP

These instructions are going to be executed by the callee. Let's assume that callee is going to make another call to a function foo that require 1 integer argument. callee will set it's local integer variable to 7 then send double it's value to foo



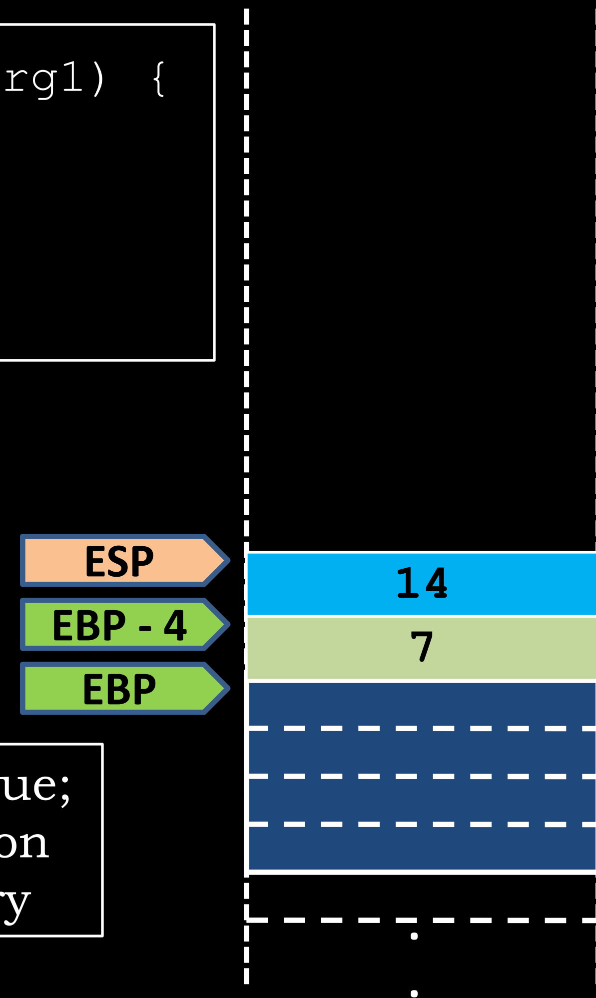
Functions: Low Level View
- Code Optimization -

cdecl

```
push    ebp
mov     ebp, esp
push    0

mov     [ebp-4], 7
mov     ecx, [ebp-4]
add     ecx, ecx

push    ecx

call    <foo>

pop     ecx

pop     ecx
pop     ebp
ret
```

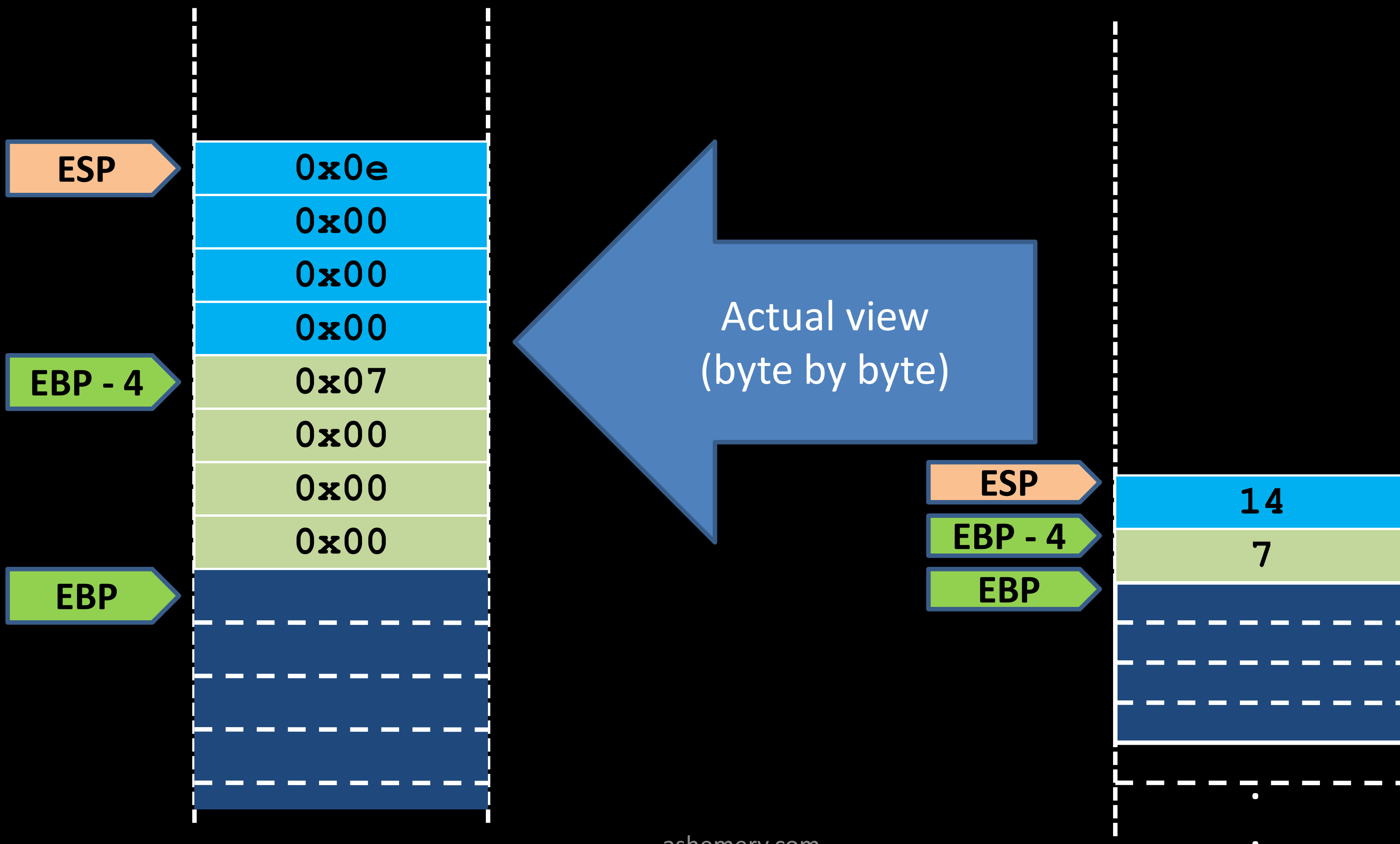
EIP

```
void callee(int arg1) {
  int v1;
  v1 = 7;
  foo(v1*2);
};
```



Before we continue; let's take a look on the stack memory

Functions: Low Level View - Hint about Endianness -



Functions: Low Level View
- Hint about Endianness -

Start of Memory

Pointer	little-endian	big-endian	Pointer
ESP	0x0e	0x00	ESP
	0x00	0x00	
	0x00	0x00	
	0x00	0x0e	
EBP - 4	0x07	0x00	EBP - 4
	0x00	0x00	
	0x00	0x00	
	0x00	0x07	
EBP			EBP

In little-endian architect (like intel processors); multi-byte values are filled starting from the least significant byte. In big-endian (like SPARC processors) they are filled in a reverse order (starting from most significant byte).

Top of Memory

Functions: Low Level View
- Code Optimization -

cdecl

```
push    ebp
mov     ebp, esp
push    0
```

```
mov     [ebp-4], 7
mov     ecx, [ebp-4]
add     ecx, ecx
```

```
push    ecx
```

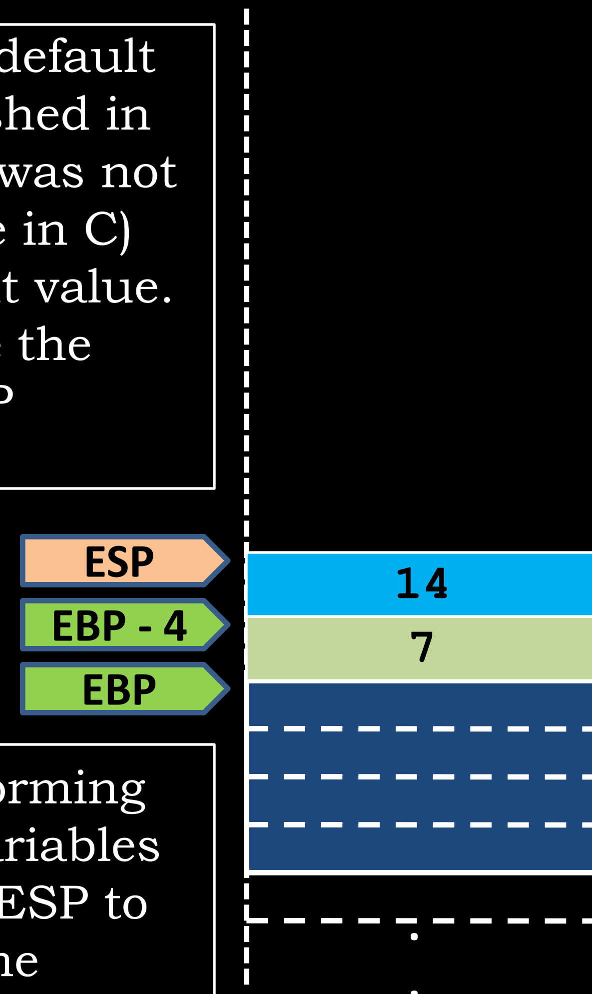
```
call    <foo>
```

```
pop     ecx
```

```
pop     ecx
pop     ebp
ret
```

EIP

We can see that the default value 0 that was pushed in the epilogue section was not used. Compilers (like in C) do not push a default value. Instead; they reserve the space by moving ESP register



Also, instead of performing POP to clean local variables space; we can move ESP to empty the stack frame

Functions: Low Level View
- Code Optimization -

cdecl

```
push    ebp
mov     ebp, esp
sub     esp, 4
```

```
mov     [ebp-4], 7
mov     ecx, [ebp-4]
add     ecx, ecx
```

```
push    ecx
```

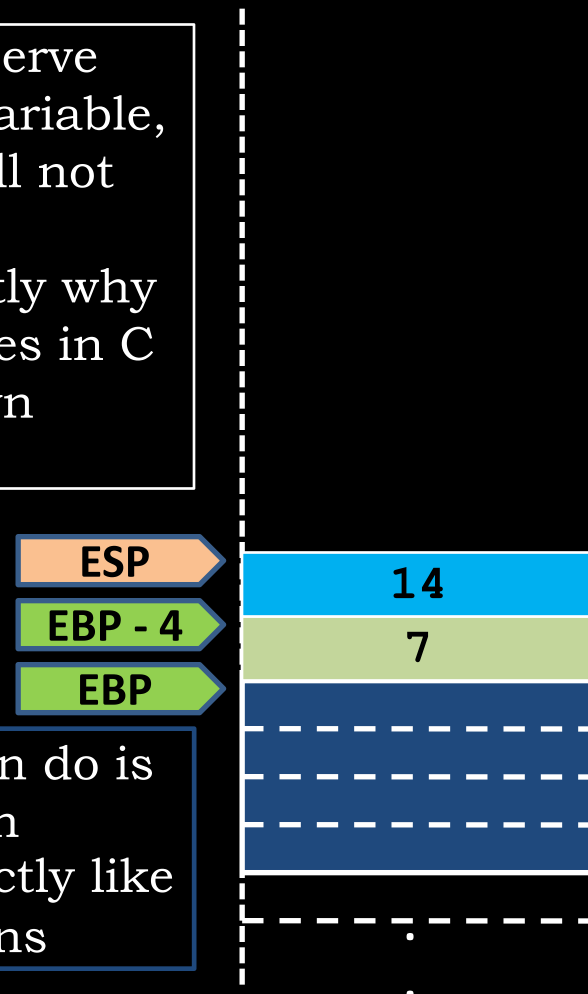
```
call    <foo>
```

```
pop     ecx
```

```
mov     esp, ebp
pop     ebp
ret
```

EIP

ESP will move to reserve space for the local variable, but that space is still not initialized.
Now you know exactly why uninitialized variables in C will contain unknown values (rubbish) ;)



Another thing we can do is using the instruction `leave` which do exactly like these two instructions

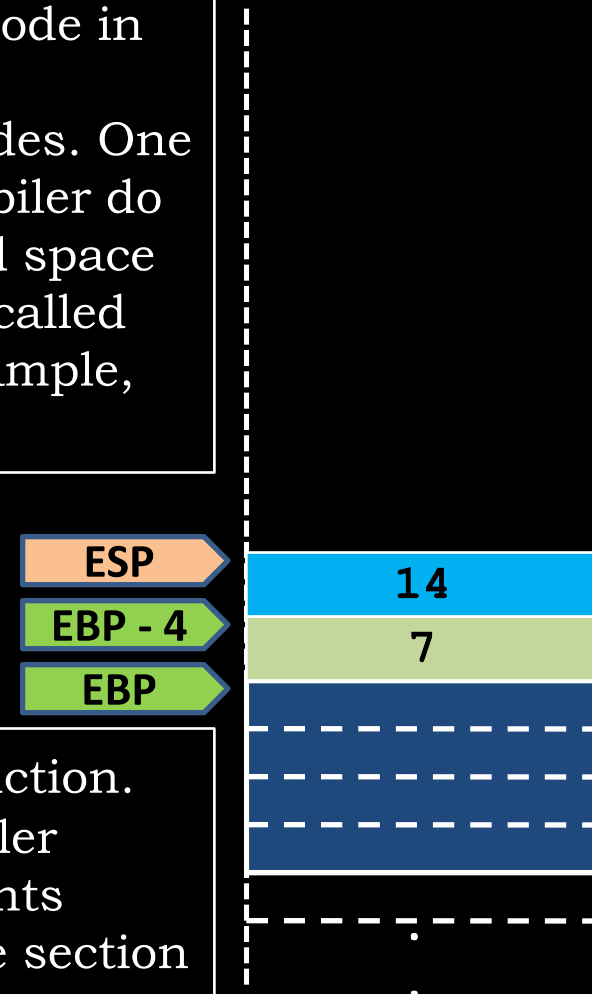
Functions: Low Level View
- Code Optimization -

cdecl

```
push    ebp
mov     ebp, esp
sub     esp, 4

mov     [ebp-4], 7
mov     ecx, [ebp-4]
add     ecx, ecx

push    ecx

call    <foo>

pop     ecx

leave
ret
```

Compilers read the code in many passes before generating object-codes. One of the thing the compiler do is calculating needed space for all arguments of called functions. In our example, `foo` needs 4 bytes.

EIP



`push` is a slow instruction. Therefore, the compiler reserves the arguments space in the epilogue section

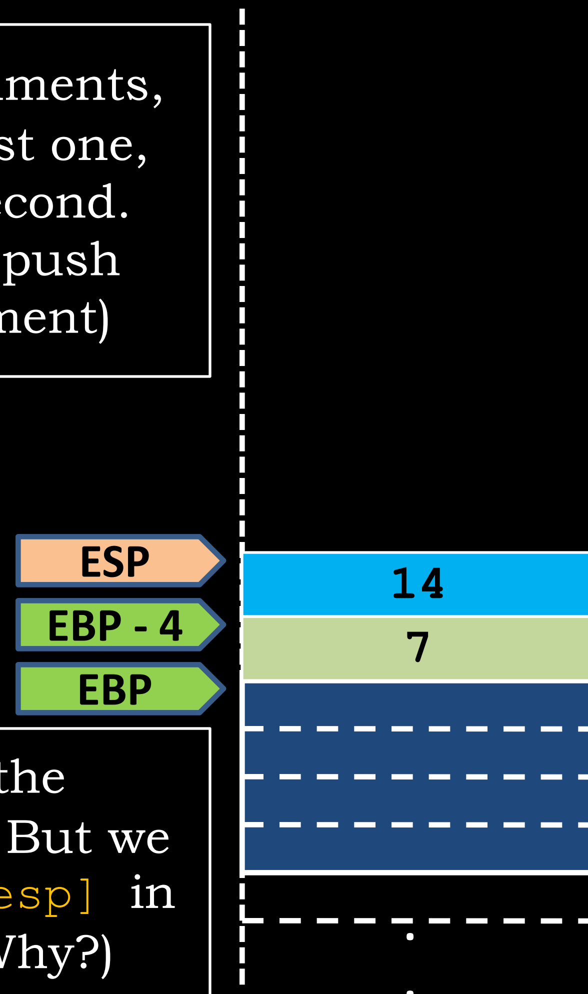
Functions: Low Level View
- Code Optimization -

cdecl

```
push   ebp
mov    ebp, esp
sub    esp, 8

mov    [ebp-4], 7
mov    ecx, [ebp-4]
add    ecx, ecx

mov    [ebp-8], ecx

call   <foo>

leave
ret
```

EIP

If `foo` takes two arguments, then EBP-8 is the first one, and EBP-12 is the second. (same as performing push for 2nd then 1st argument)



`[ebp-8]` is for sure the argument to passed. But we can replace it with `[esp]` in this scenario only. (Why?)

Functions: Low Level View
- Code Optimization -

cdecl

```
push   ebp
mov    ebp, esp
sub    esp, 8

mov    [ebp-4], 7
mov    ecx, [ebp-4]
add    ecx, ecx

mov    [esp], ecx

call   <foo>

leave
ret
```

The same result

cdecl

```
push   ebp
mov    ebp, esp
push   0

mov    [ebp-4], 7
mov    ecx, [ebp-4]
add    ecx, ecx

push   ecx

call   <foo>

pop    ecx

pop    ecx
pop    ebp
ret
```

Functions: Low Level View
- Example from GCC -

```
void myfun1(char *str) {
push    ebp
mov    ebp,esp
char buffer[16];
sub     esp, 0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>
}
leave
ret
```

The function myfun1 require 16 bytes for the local array.

strcpy require 8 bytes for it's arguments

myfun2 require 4 bytes for it's arguments

The compiler made a reservation for 24 bytes (0x18) which is 16 for array + 8 for **maximum** arguments space

Functions: Low Level View
- Example from GCC -

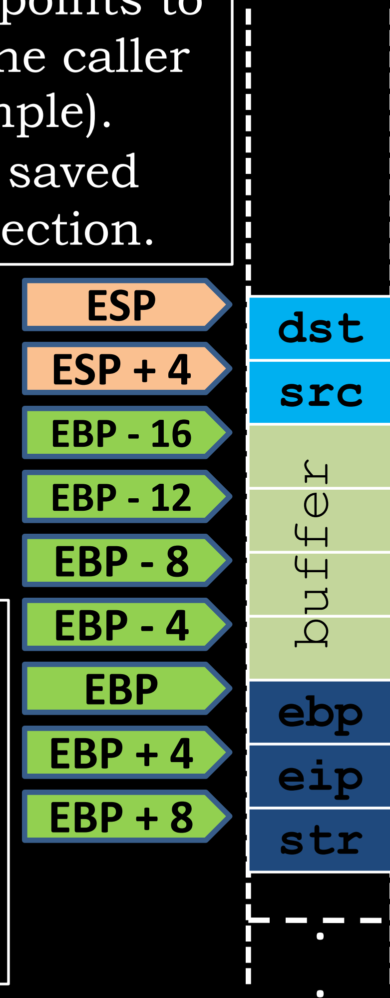
```
void myfun1(char *str) {
push    ebp
mov     ebp,esp
char buffer[16];
sub     esp,0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>
}
leave
ret
```

EIP

By default, EBP+4 points to the saved EIP of the caller (`main` in this example). EBP points to the saved EBP by epilogue section.

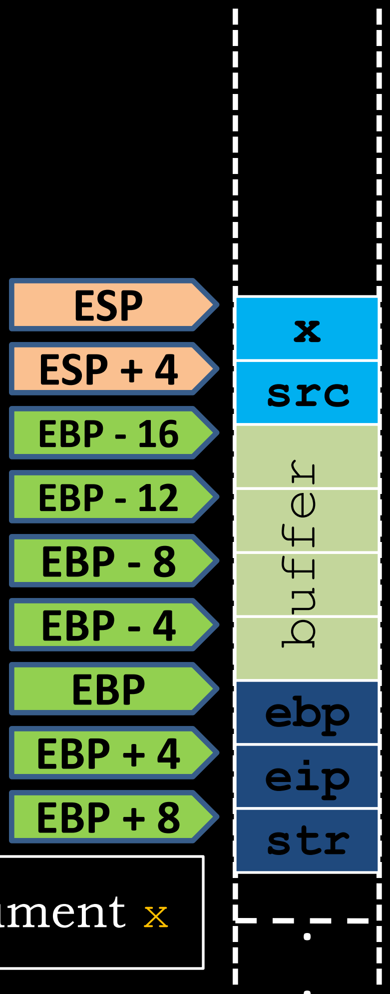
strcpy takes two arguments, destination `dst` then source `src`.

EBP+8 is the sent value by the caller `main` to the callee `myfun1` that is named `str` in this code.



Functions: Low Level View
- Example from GCC -

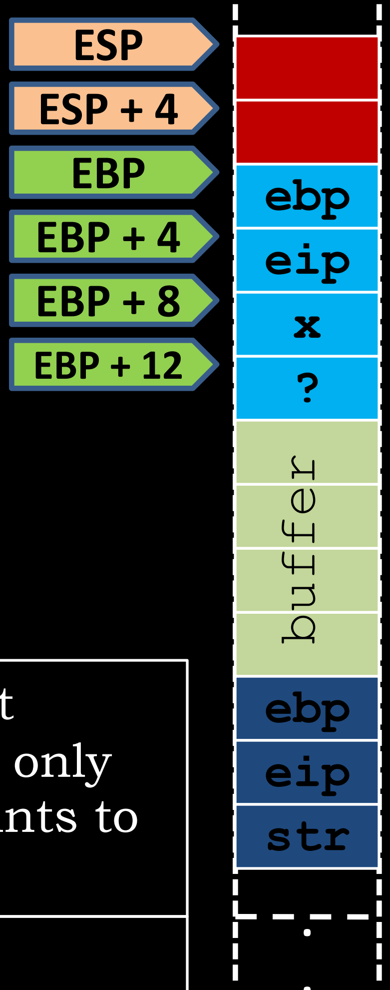
```
void myfun1(char *str) {
push    ebp
mov     ebp,esp
char buffer[16];
sub     esp,0x18
strcpy(buffer, str);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80482c4 <strcpy@plt>
myfun2(buffer);
lea     eax,[ebp-16]
mov     DWORD PTR [esp],eax
call    0x80483b4 <myfun2>   EIP
}
leave
ret
```



myfun2 takes one argument x

Functions: Low Level View
- Example from GCC -

```
void myfun2(char *x) {
push    ebp
mov     ebp,esp
sub     esp,0x8
printf(" You entered: %s\n", x);
mov     eax,DWORD PTR [ebp+8]
mov     DWORD PTR [esp+4],eax
mov     DWORD PTR [esp],0x8048520
call    0x80482d4 <printf@plt>   EIP
}
leave
ret
```



EBP+8 points to the first argument sent to the current function. EBP+12 points is the second and so on. But only one argument used by `myfun2`. Therefore, EBP+12 points to an irrelevant location as `myfun2` can see.

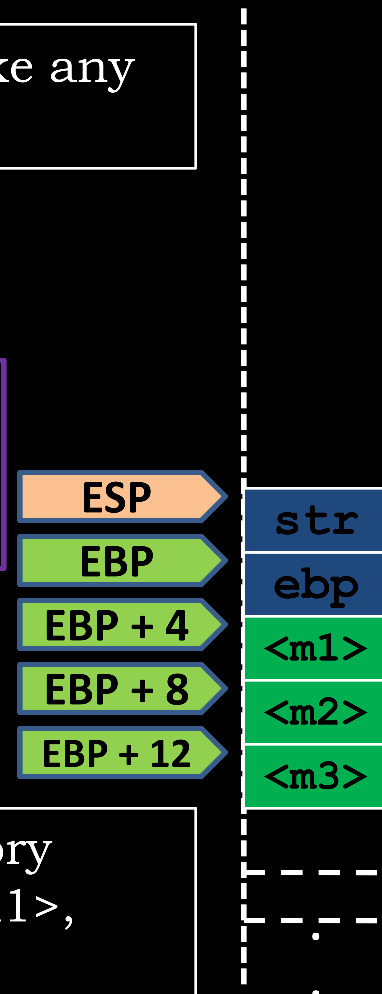
Can you guess what is currently saved in [EBP+12] ?

Functions: Low Level View
- Example from GCC -

```
int main(int argc, char *argv[]){
push    ebp
mov     ebp,esp
sub     esp,0x4
if (argc > 1)
cmp     DWORD PTR [ebp+8],0x1
jle     0x8048412
myfun1(argv[1]);
mov     eax,DWORD PTR [ebp+12]
add     eax,0x4
mov     eax,DWORD PTR [eax]
mov     DWORD PTR [esp],eax
call    0x80483cf <myfun1>     EIP
jmp     0x804841e
else printf("No arguments!\n");
mov     DWORD PTR [esp],0x8048540
call    0x80482d4 <printf@plt>
}
leave
ret
```

main is a function like any other function.

Can you tell what these instructions do?



What do these memory locations contain <m1>, <m2>, and <m3>?

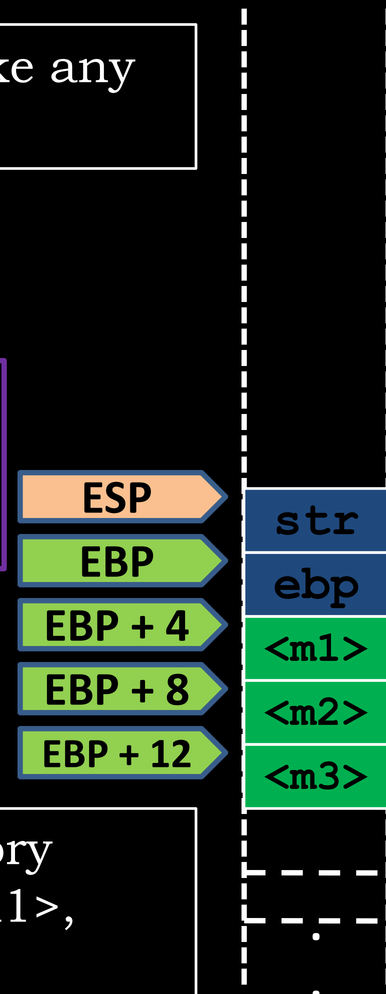
Functions: Low Level View
- Example from GCC -

```
int main(int argc, char *argv[]){
push    ebp
mov     ebp,esp
sub     esp,0x4
if (argc > 1)
cmp     DWORD PTR [ebp+8],0x1
jle     0x8048412
myfun1(argv[1]);
mov     eax,DWORD PTR [ebp+12]
add     eax,0x4
mov     eax,DWORD PTR [eax]
mov     DWORD PTR [esp],eax
call    0x80483cf <myfun1>
jmp     0x804841e
else printf("No arguments!\n");
mov     DWORD PTR [esp],0x8048540
call    0x80482d4 <printf@plt>
}
leave
ret
```

EIP

main is a function like any other function.

Can you tell what these instructions do?



What do these memory locations contain <m1>, <m2>, and <m3>?

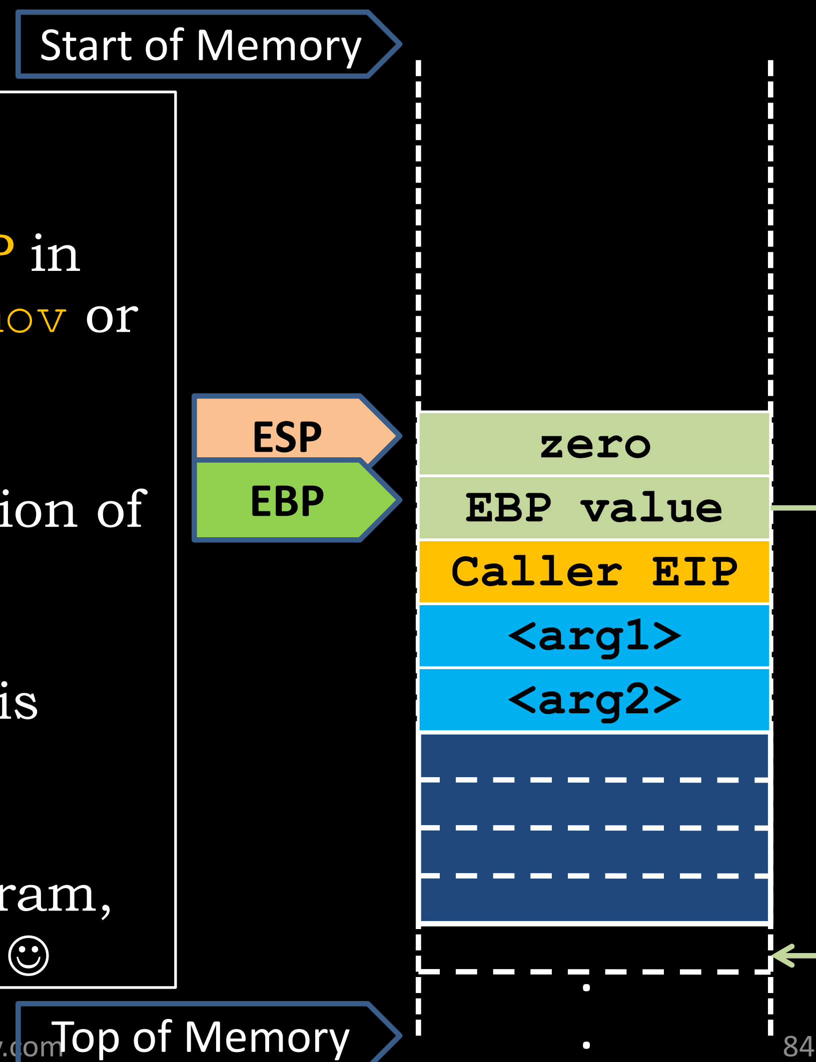
Assignment #1

Functions: Low Level View
- Stack Reliability -

<u>Think with me please:</u>

- What if we can locate Caller EIP in the stack and change it using `mov` or any other instruction?
- What if the new value is a location of another block of code?
- What if the other block of code is harmful (security wise)?

Bad for the user running the program, but good for the Exploit developer ☺



References

- Open Security Training, Introductory Intel x86: (Architecture, Assembly, Applications, & Alliteration) by Xeno Kovah, http://www.opensecuritytraining.info/IntroX86.html
- “Professional Assembly Language by Blum, page. 163
- Learned about the basic hardware registers and how they’ re used
- Learned about how the stack is used
- Saw how C code translates to assembly
- Learned basic usage of compilers, disassemblers, and debuggers so that assembly can easily be explored
- Learned about Intel vs AT&T asm syntax