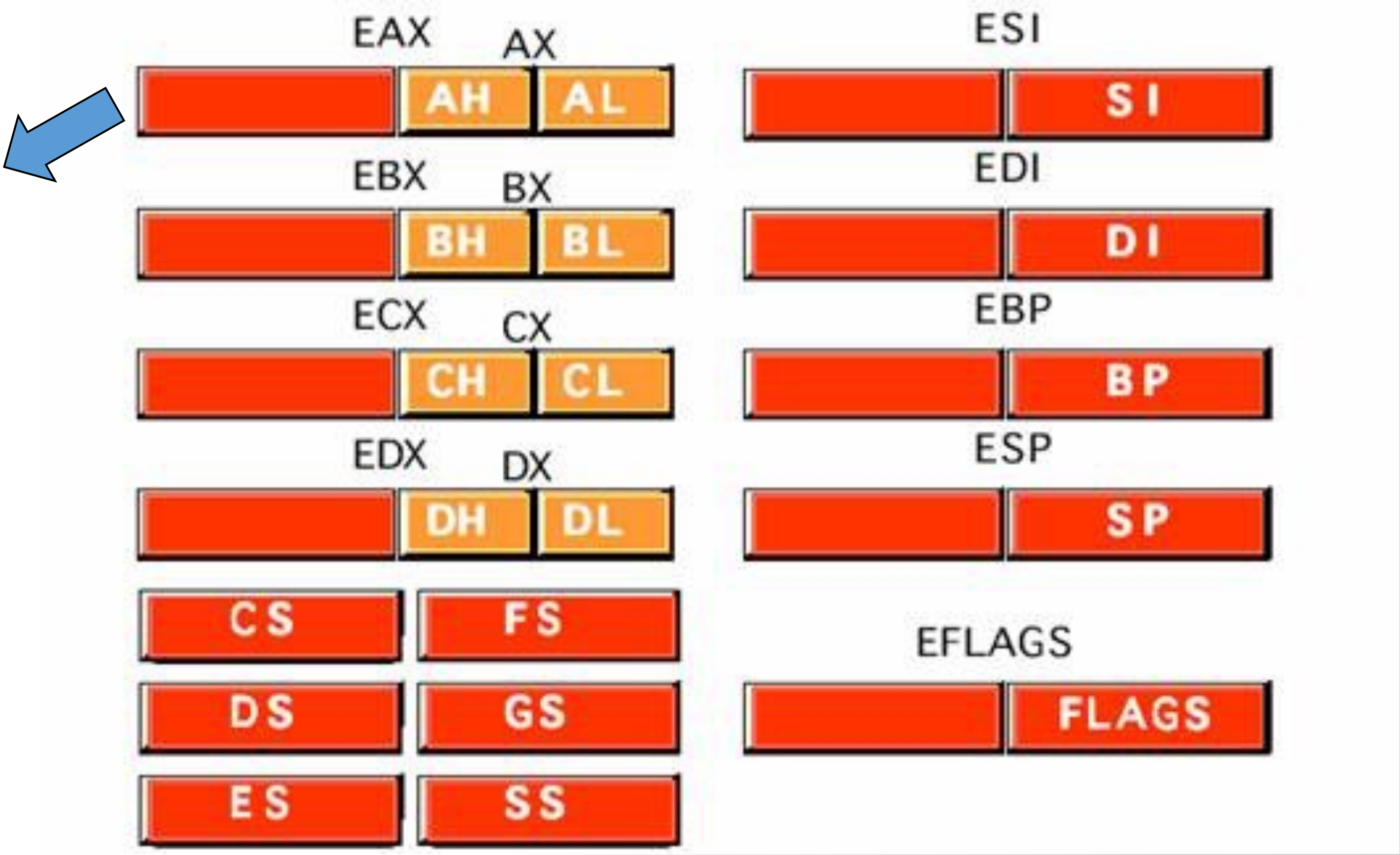
Class 3

CSC 472/583 Topics of Software Security
X86 Assembly & Stack

Dr. Si Chen (schen@wcupa.edu)

General-purpose Registers

- The **eight** 32-bit general-purpose data registers are used to hold operands for logical and arithmetic operations, operands for address calculations and memory pointers

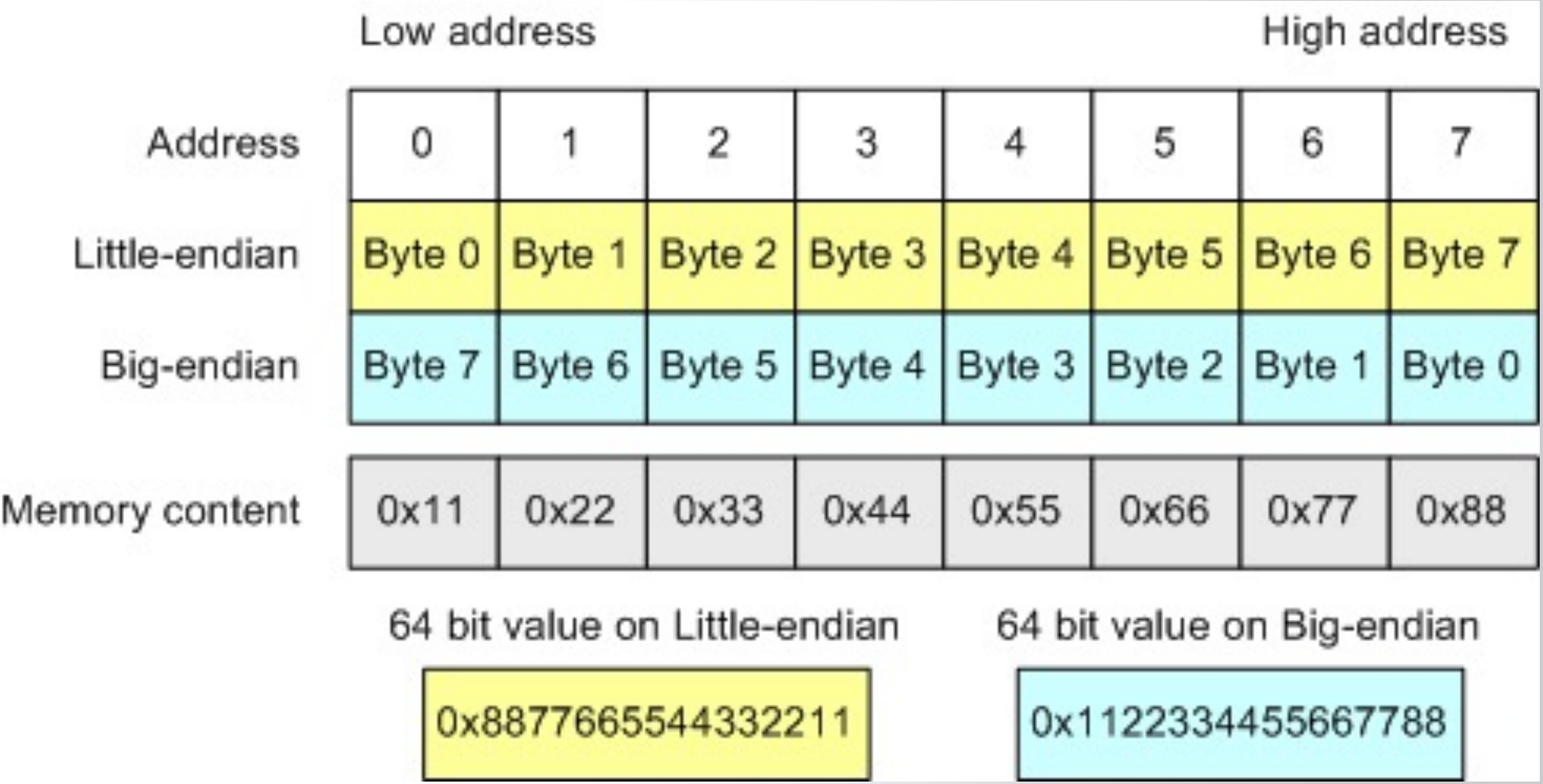


4 Bytes

- ## Register
- + `esp` `ebp` `esi` `edi` - *DWORD (32-bit)*
- + `sp` `bp` `si` `di` - WORD (16-bit) - *rarely used*
- + \[esp, ebp\] - *mark the range of stack frame*
- + esi, edi - *used as buffer pointer, some instruction will directly handle esi, edi*

- ## Other Register
- + `eip` - *Program counter, pointing to the current line*
- + `eflags` - *cannot change the value directly, store the instruction result*
- + `cs` `ss` `ds` `es` `fs` `gs` - *segment register*

Byte Order



Low address

High address

Address	0	1	2	3	4	5	6	7
Little-endian	Byte 0	Byte 1	Byte 2	Byte 3	Byte 4	Byte 5	Byte 6	Byte 7
Big-endian	Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0
Memory content	0x11	0x22	0x33	0x44	0x55	0x66	0x77	0x88

64 bit value on Little-endian

0x8877665544332211

64 bit value on Big-endian

0x1122334455667788

X86 ASM

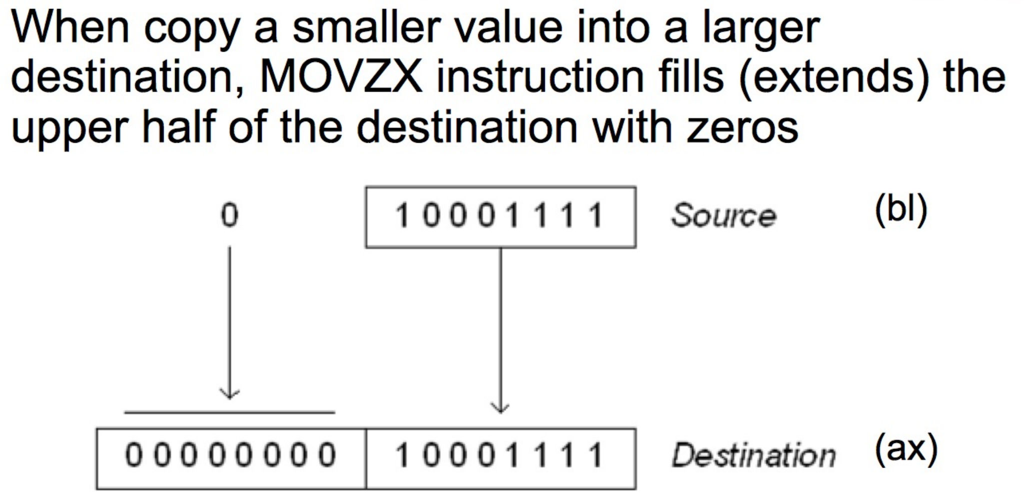
MOV

- Move **reg/mem** value to **reg/mem**
  - mov A, B is "Move B to A" (A=B)
  - Same data size

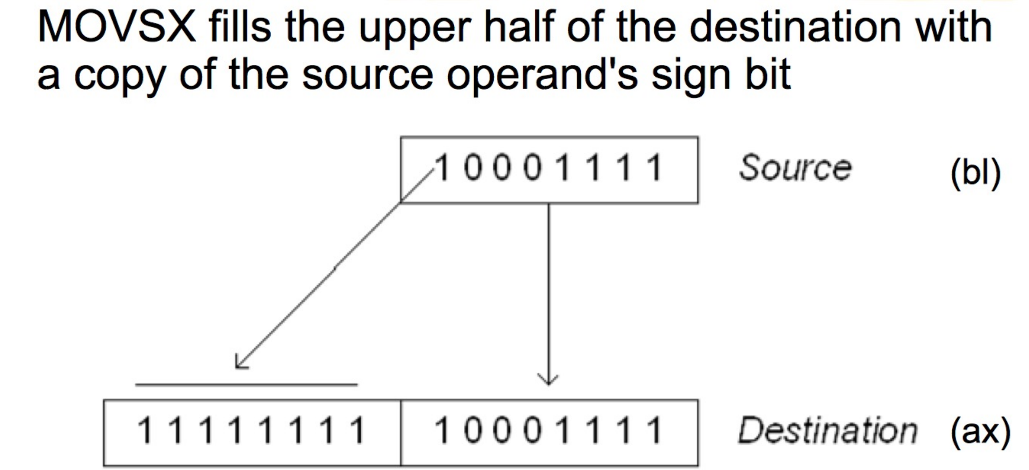
```
mov eax, 0x1337
mov bx, ax
mov [esp+4], bl
```

MOVZX / MOVSX

- From small register to large register
- Zero-extend (MOVZX) / sign-extend (MOVSX)
- Example: movzx ebx, al



When copy a smaller value into a larger destination, MOVZX instruction fills (extends) the upper half of the destination with zeros



MOVSX fills the upper half of the destination with a copy of the source operand's sign bit

More About Memory Access

- mov ebx, [esp + eax * 4] **Intel**
- mov (%esp, %eax, 4), %ebx **AT&T**
- mov BYTE [eax], 0x0f
  You must indicate the data size: BYTE/WORD/DWORD

ADD / SUB

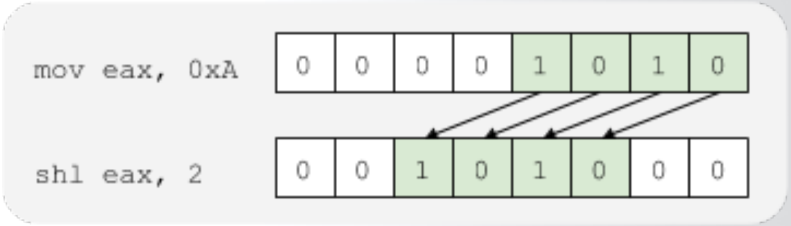
- ADD / SUB
- Normallly "reg += reg" or "reg += imm"
- Data size should be equal
  - ADD eax, ebx
  - sub eax, 123
  - sub eax, BL ; Illegal

INC / DEC

- **inc, dec** — Increment, Decrement
- The **inc** instruction increments the contents of its operand by one.
  The **dec** instruction decrements the contents of its operand by one.
- *Syntax*
  inc <reg>
  inc <mem>
  dec <reg>
  dec <mem>
- *Examples*
  DEC EAX — subtract one from the contents of EAX.
  INC DWORD PTR [var] — add one to the 32-bit integer stored at location *var*

SHL / SHR / SAR

- Shift logical left / right
- Shift arithmetic right
- Common usage: **SHL eax, 2** (when calculate memory address)

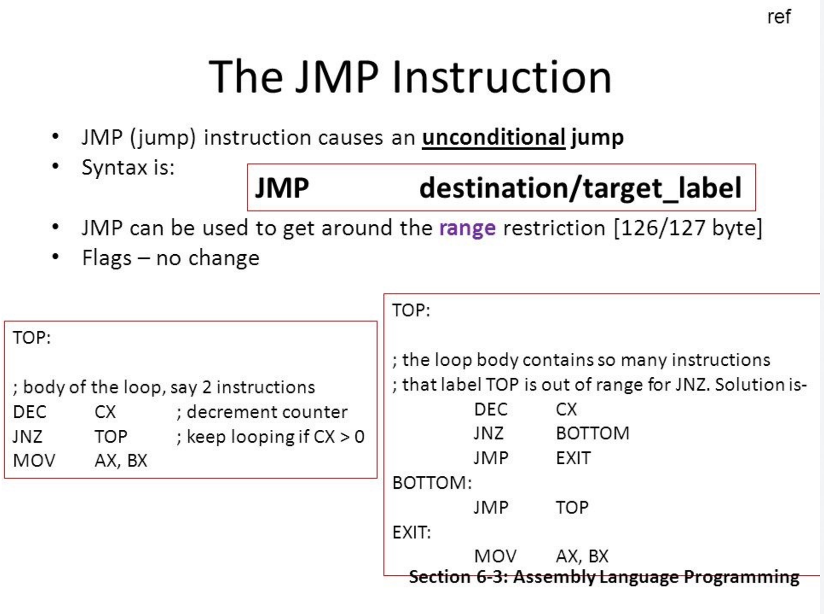


```
mov eax, 0xA    0 0 0 0 1 0 1 0

shl eax, 2      0 0 1 0 1 0 0 0
```

Jump

- Unconditional jump: jmp
- Conditional jump: je/jne
  and ja/jae/jb/jbe/jg/jge/jl/jle ...
- Sometime with ”cmp A, B” -- compare these two values and set eflags
- Conditional jump is decided by some of the eflags bits.



ref

The JMP Instruction

- JMP (jump) instruction causes an **unconditional jump**
- Syntax is:

```
JMP        destination/target_label
```

- JMP can be used to get around the **range** restriction [126/127 byte]
- Flags – no change

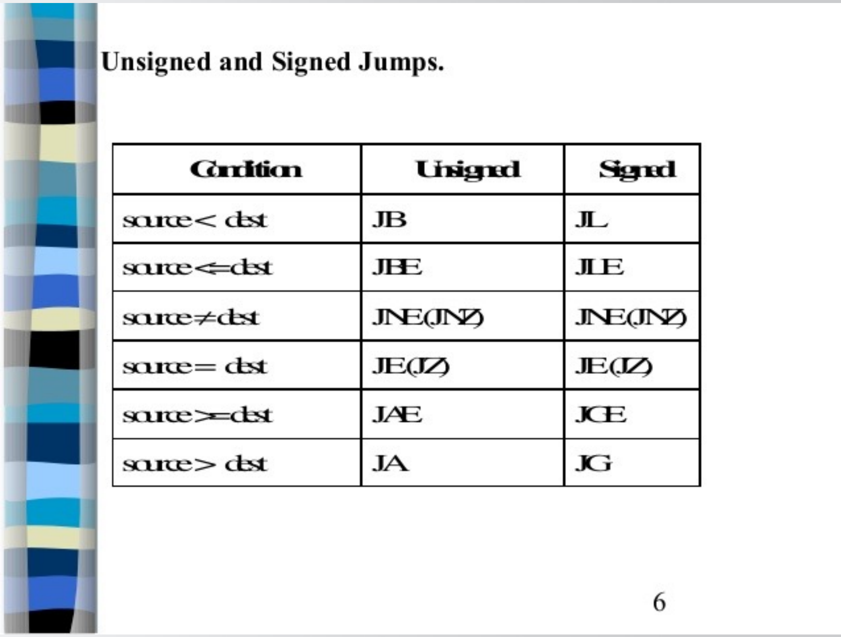
```
TOP:

; body of the loop, say 2 instructions
DEC     CX      ; decrement counter
JNZ     TOP     ; keep looping if CX > 0
MOV     AX, BX
```

```
TOP:

; the loop body contains so many instructions
; that label TOP is out of range for JNZ. Solution is-
         DEC      CX
         JNZ      BOTTOM
         JMP      EXIT
BOTTOM:
         JMP      TOP
EXIT:
         MOV      AX, BX
```

Section 6-3: Assembly Language Programming



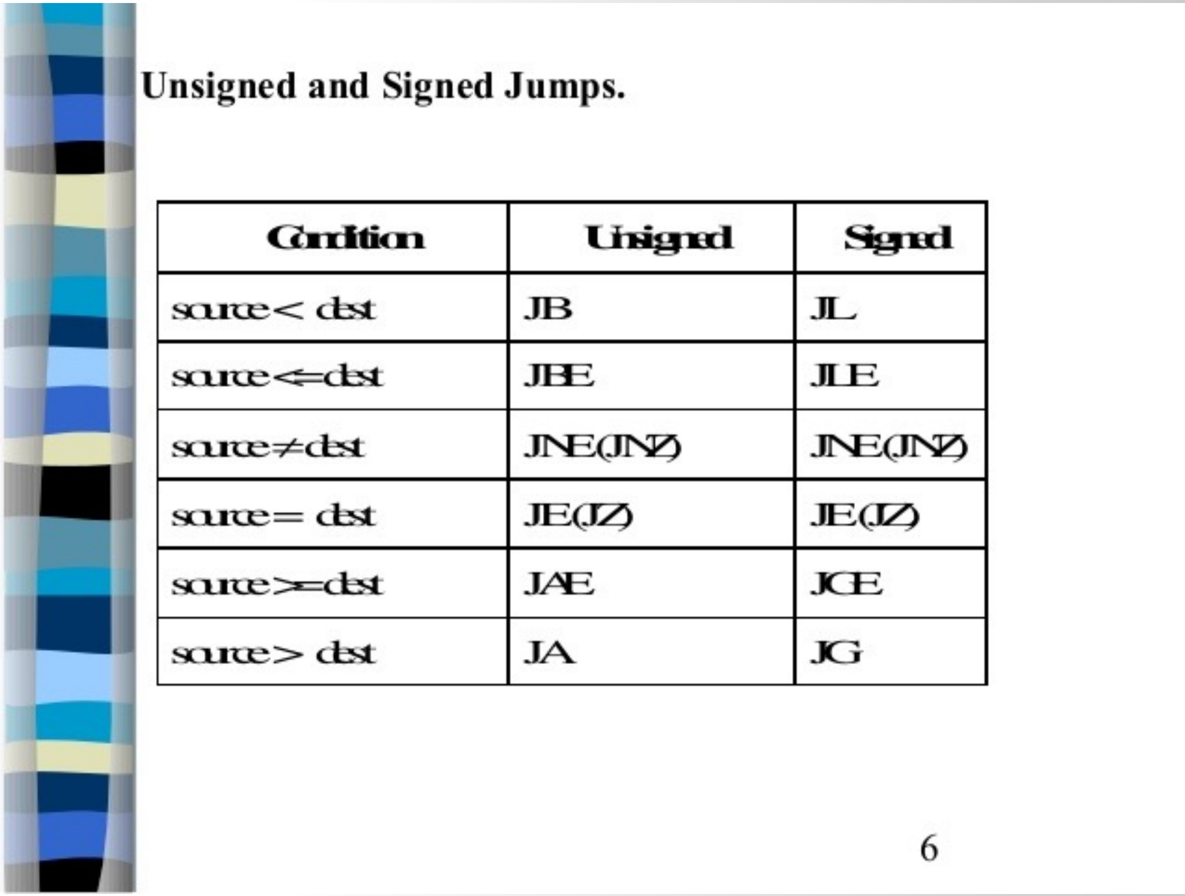
Unsigned and Signed Jumps.

Condition	Unsigned	Signed
source< dst	JB	JL
source<=dst	JBE	JLE
source≠dst	JNE(JNZ)	JNE(JNZ)
source= dst	JE(JZ)	JE(JZ)
source>=dst	JAE	JGE
source> dst	JA	JG

6

Jump

- ja/jae/jb/jbe are unsigned comparison
- jg/jge/jl/jle are signed comparison



Unsigned and Signed Jumps.

Condition	Unsigned	Signed
source< dest	JB	JL
source<=dest	JBE	JLE
source≠dest	JNE(JNZ)	JNE(JNZ)
source= dest	JE(JZ)	JE(JZ)
source>=dest	JAE	JGE
source> dest	JA	JG

CMP

- **cmp** — Compare
- Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately. This instruction is equivalent to the sub instruction, except the result of the subtraction is discarded instead of replacing the first operand. *Syntax*

  ```
  cmp <reg>,<reg>
  cmp <reg>,<mem>
  cmp <mem>,<reg>
  cmp <reg>,<con>
  ```

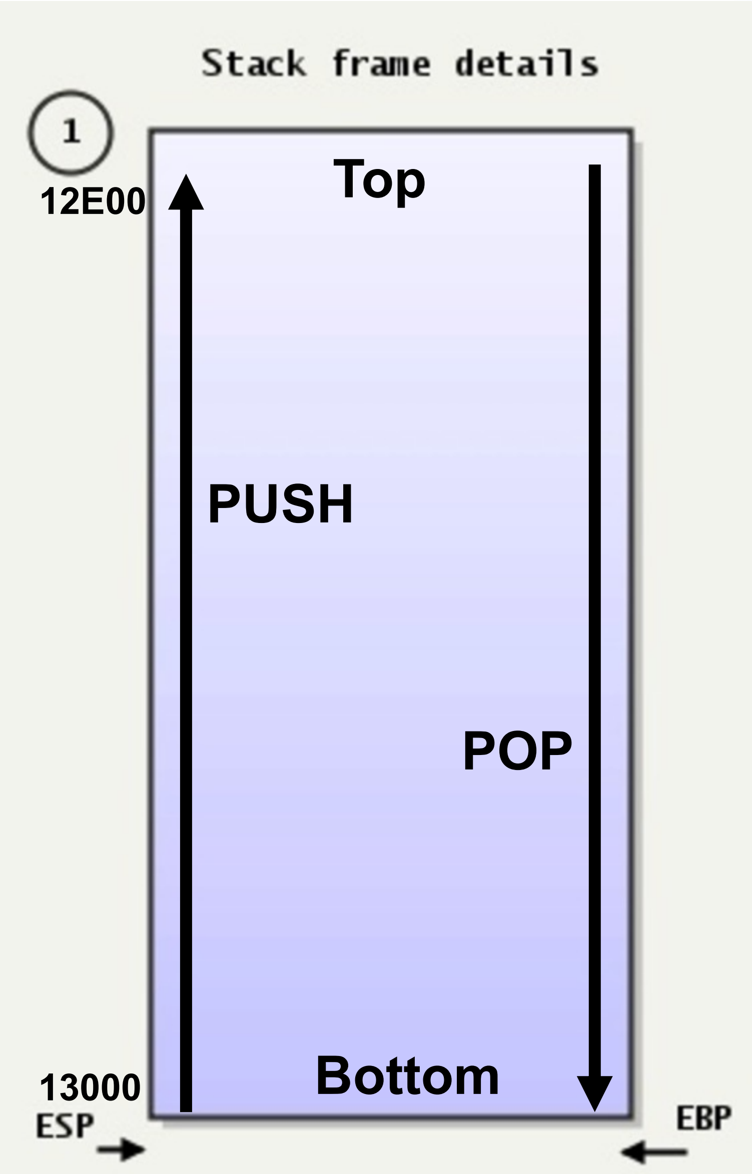
- *Example*

  ```
  cmp DWORD PTR [var], 10
  jeq loop
  ```

- If the 4 bytes stored at location *var* are equal to the 4-byte integer constant 10, jump to the location labeled *loop*.

Stack

The Stack



Stack:

- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.

Properties:

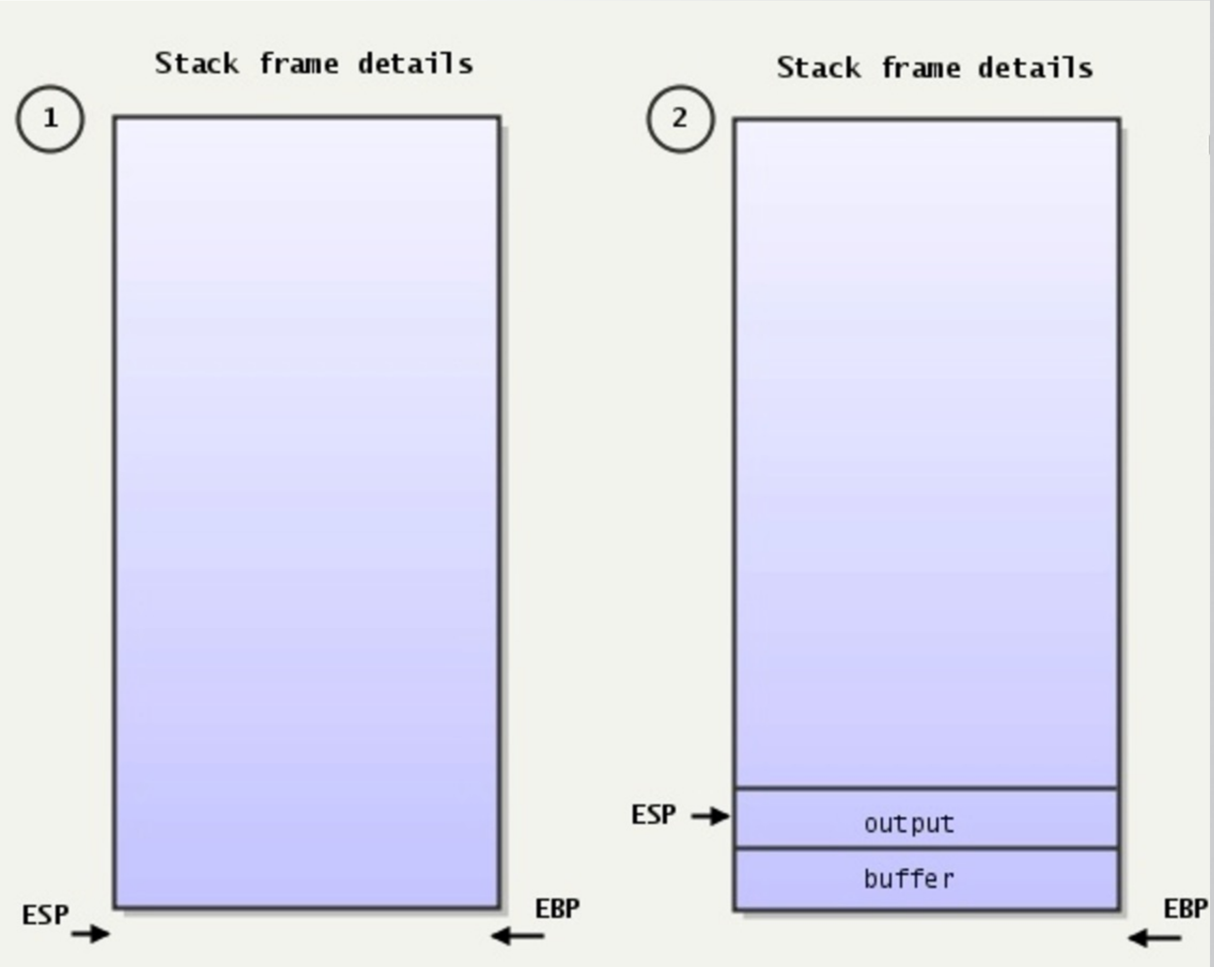
- the stack grows and shrinks as functions **push and pop** local **variables**
- there is no need to manage the memory yourself, variables are allocated and freed **automatically**
- the **stack has size limits**
- stack variables only exist while the function that created them, is running

EBP—Pointer to data on the stack
ESP—Stack pointer

The Stack

Stack:

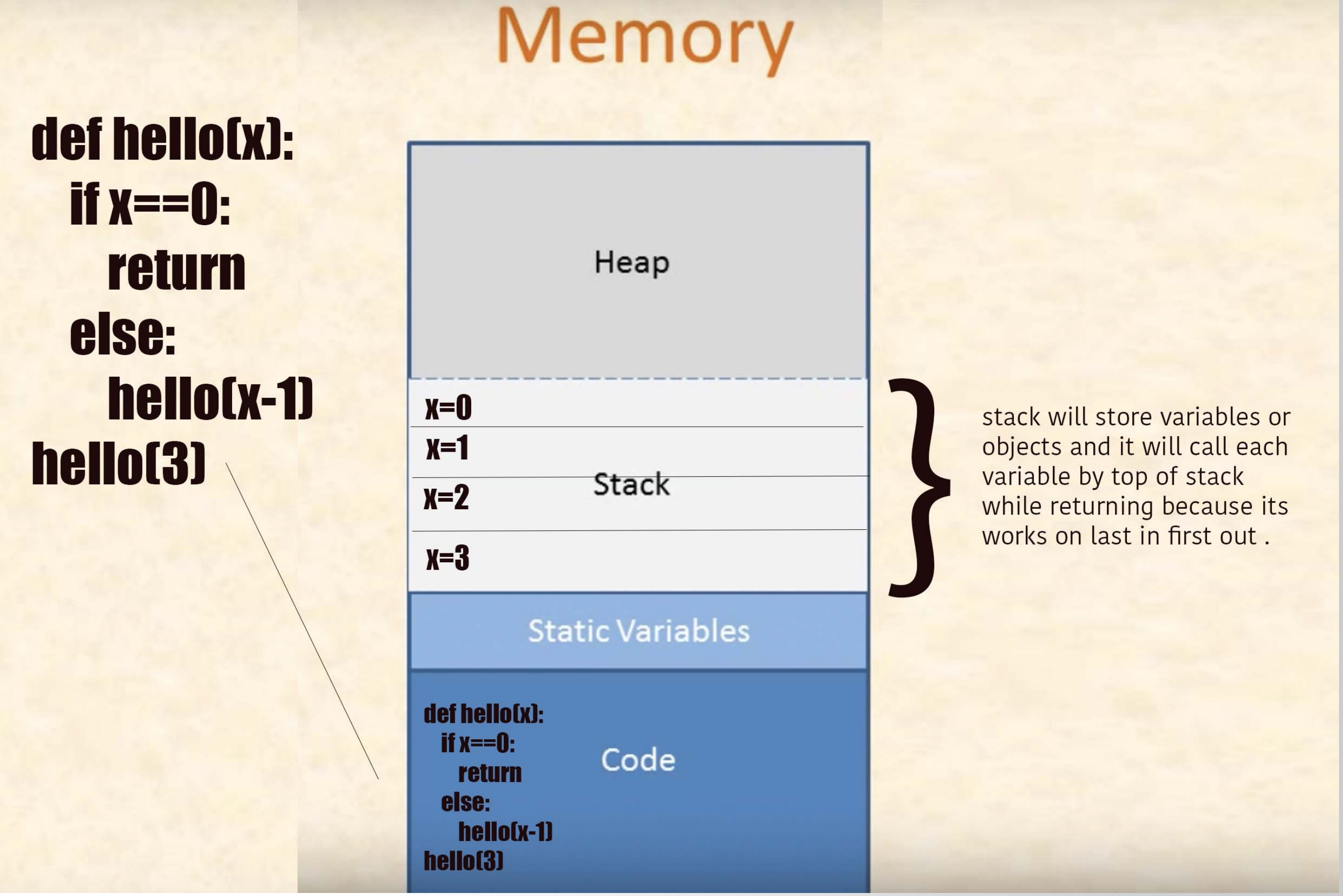
- A special region of your computer's memory that **stores temporary variables** created by each functions
- The stack is a "**LIFO**" (last in, first out) data structure
- Once a stack variable is freed, that region of memory becomes available for other stack variables.



Stack Frame

Stack Frame

- A stack frame is **a frame of data that gets pushed onto the stack**.
- In the case of a **call stack**, a stack frame would represent **a function call and its argument data**.

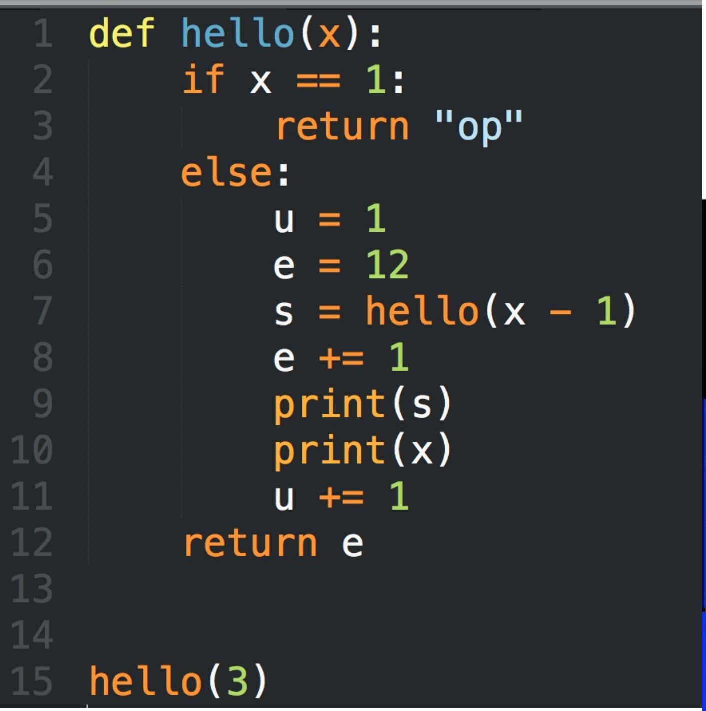


```
def hello(x):
    if x==0:
        return
    else:
        hello(x-1)
hello(3)
```

```
def hello(x):
    if x==0:
        return
    else:
        hello(x-1)
hello(3)
```

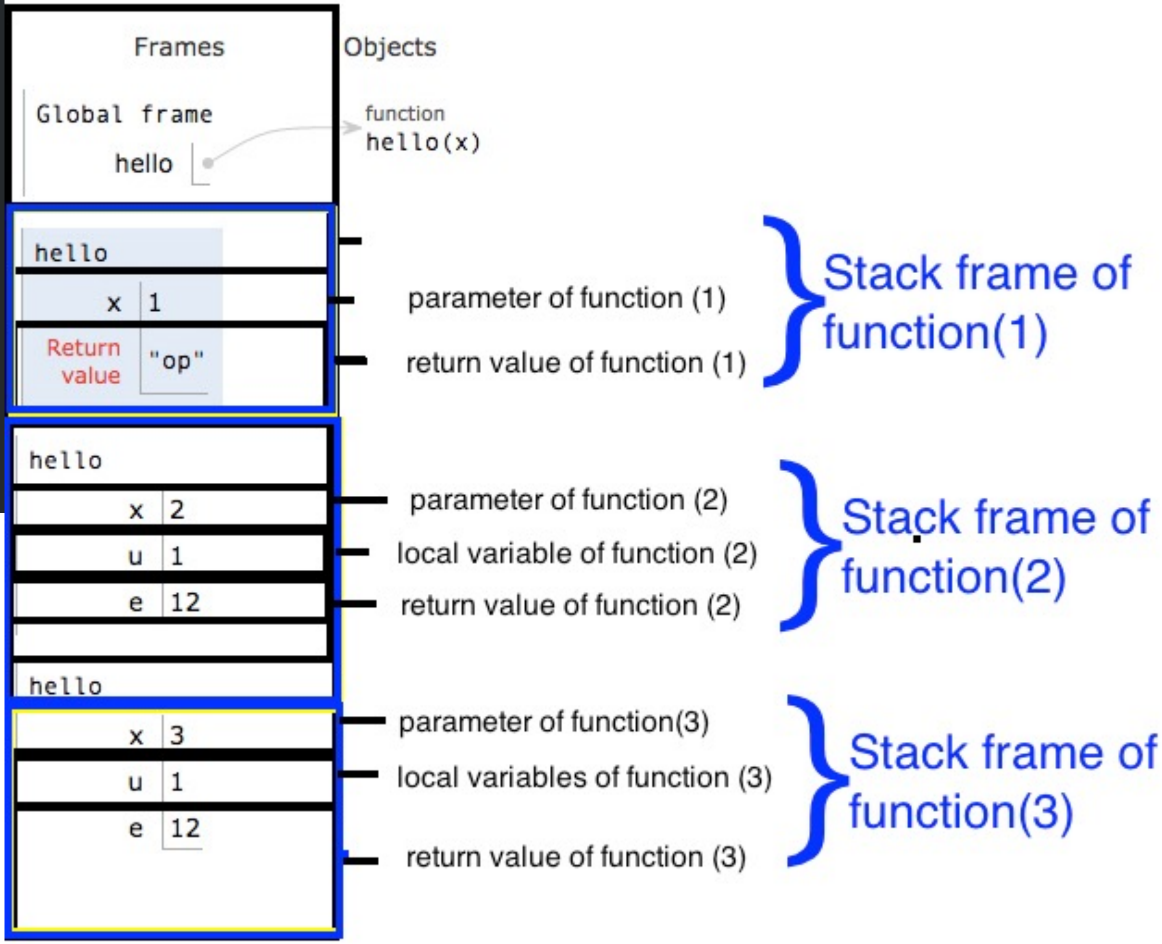
stack will store variables or objects and it will call each variable by top of stack while returning because its works on last in first out .

Stack Frame



```python
def hello(x):
    if x == 1:
        return "op"
    else:
        u = 1
        e = 12
        s = hello(x - 1)
        e += 1
        print(s)
        print(x)
        u += 1
    return e


hello(3)
```



parameter of function (1)

return value of function (1)

Stack frame of function(1)

parameter of function (2)

local variable of function (2)

return value of function (2)

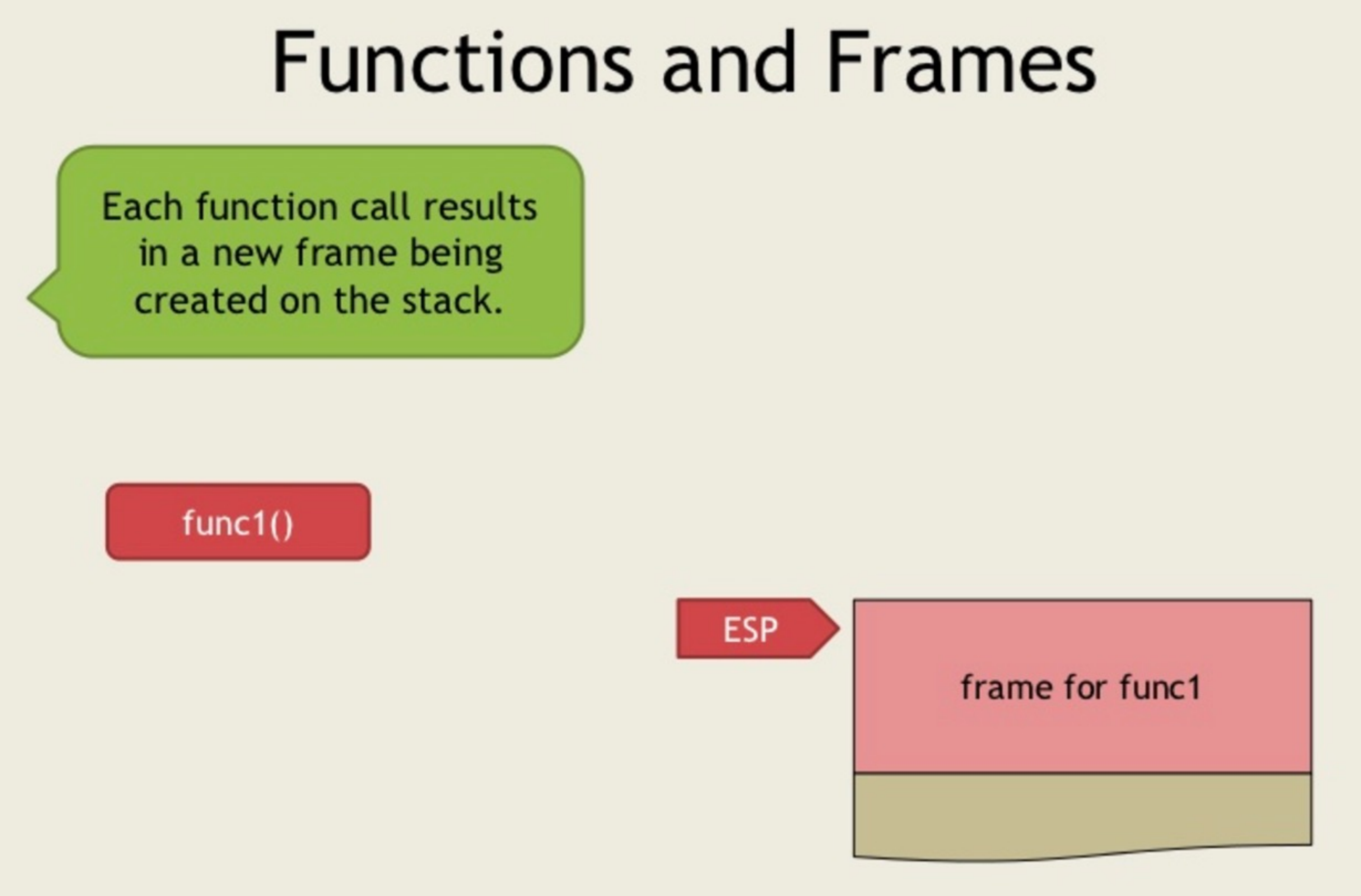
Stack frame of function(2)

parameter of function(3)

local variables of function (3)

return value of function (3)

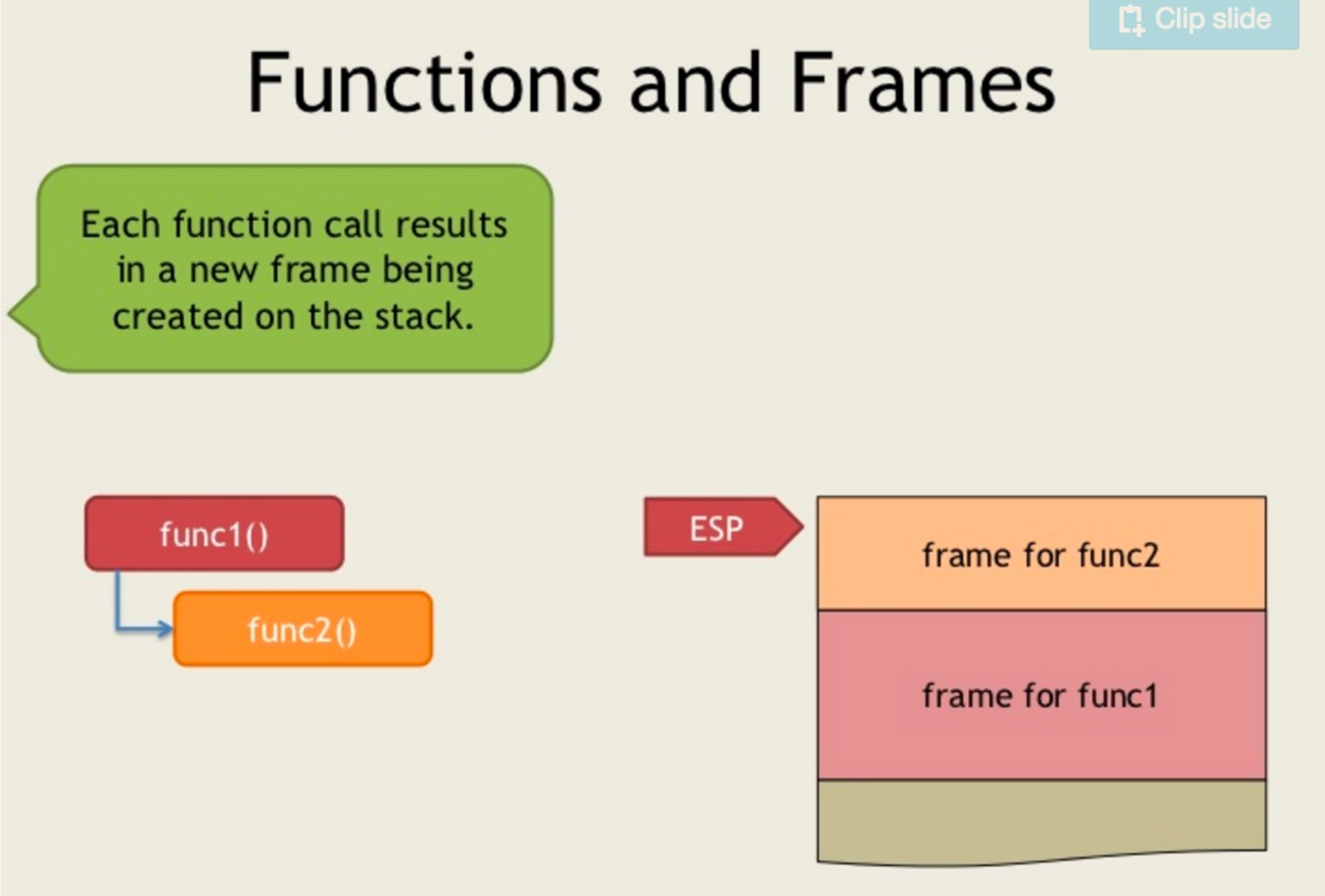
Stack frame of function(3)



Functions and Frames

Each function call results in a new frame being created on the stack.

https://www.slideshare.net/saumilshah/how-functions-work-7776073



Functions and Frames

Each function call results in a new frame being created on the stack.

func1()

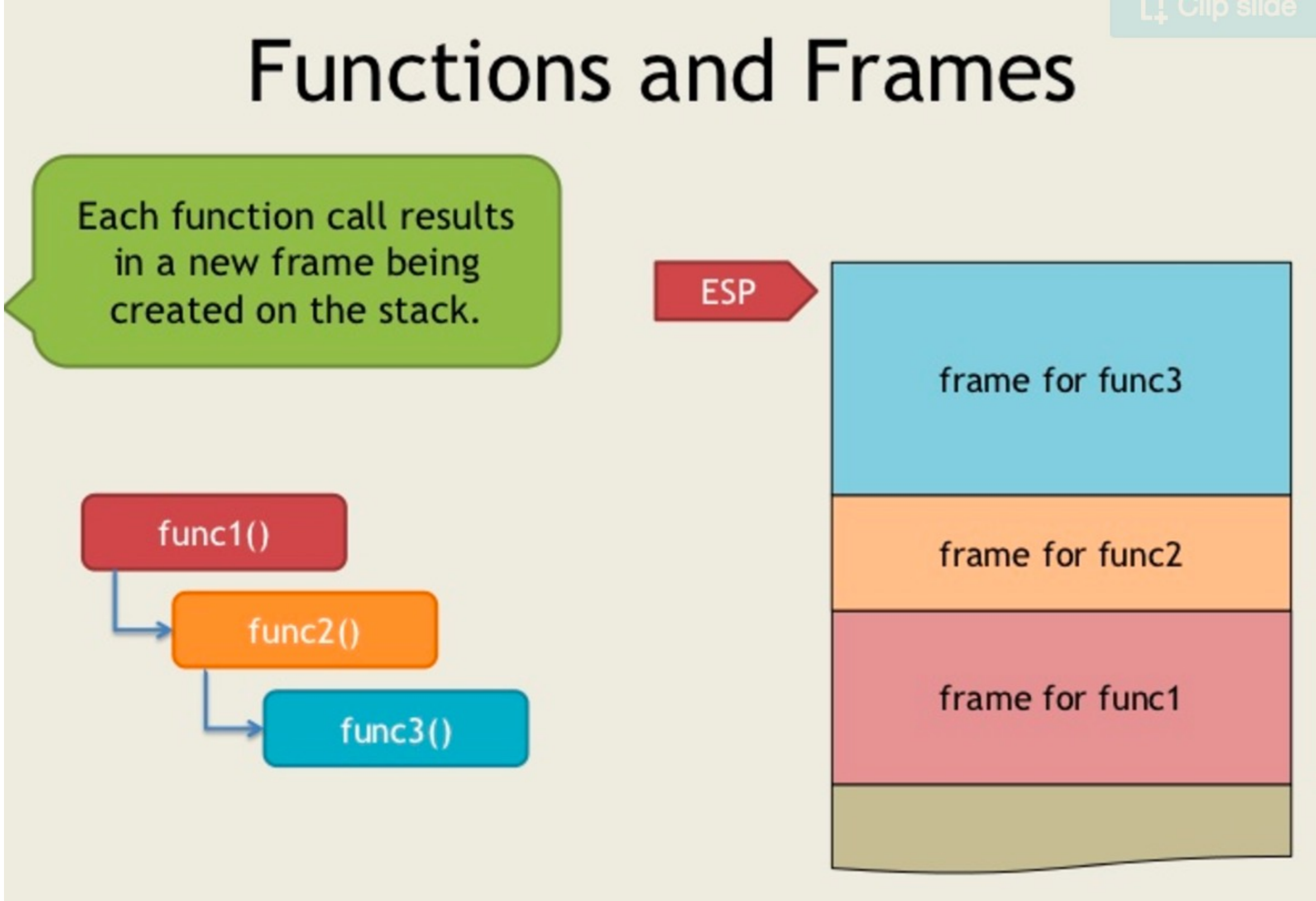
func2()

ESP

frame for func2

frame for func1

https://www.slideshare.net/saumilshah/how-functions-work-7776073



Functions and Frames

Each function call results in a new frame being created on the stack.

func1()

func2()

func3()

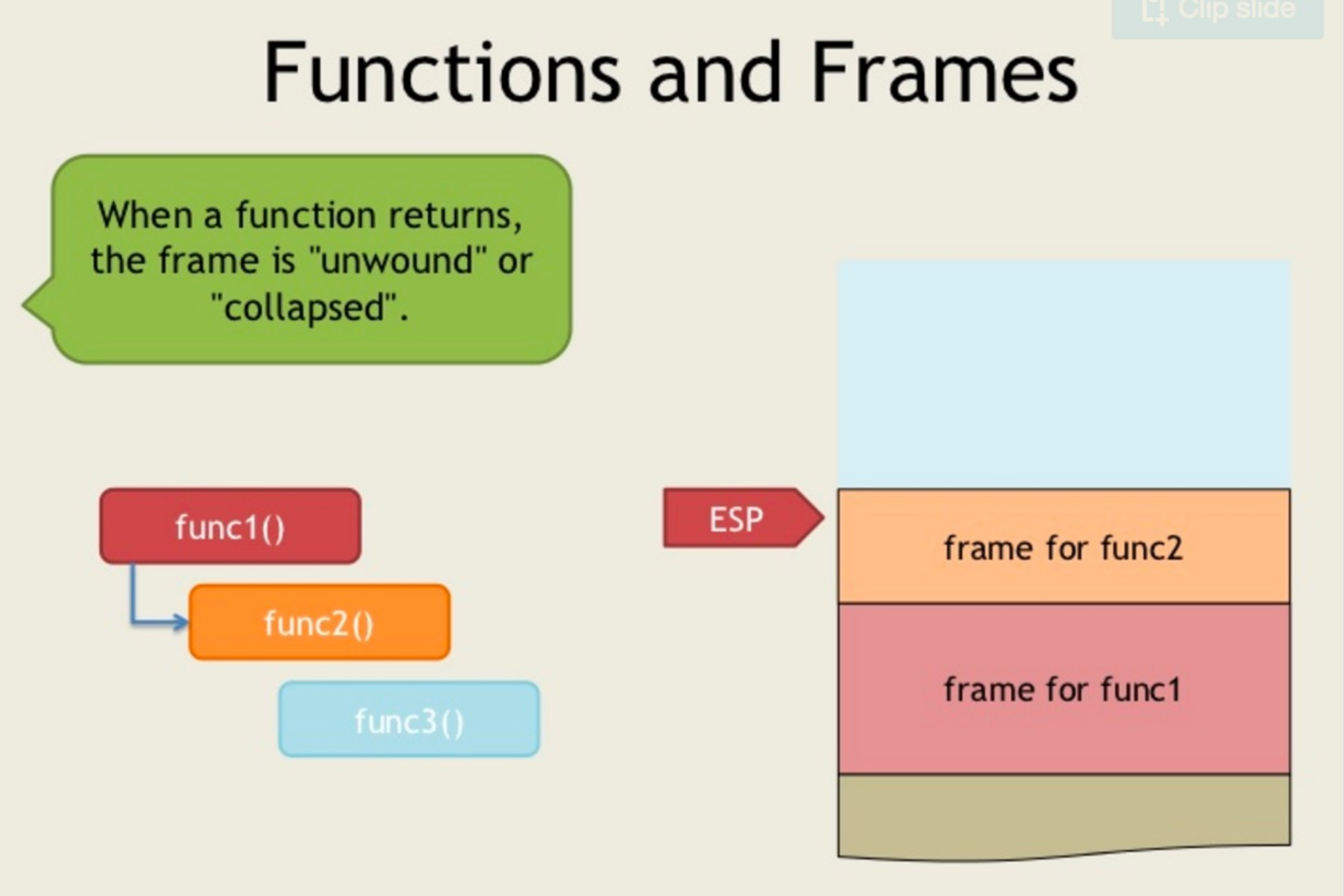
ESP

frame for func3

frame for func2

frame for func1

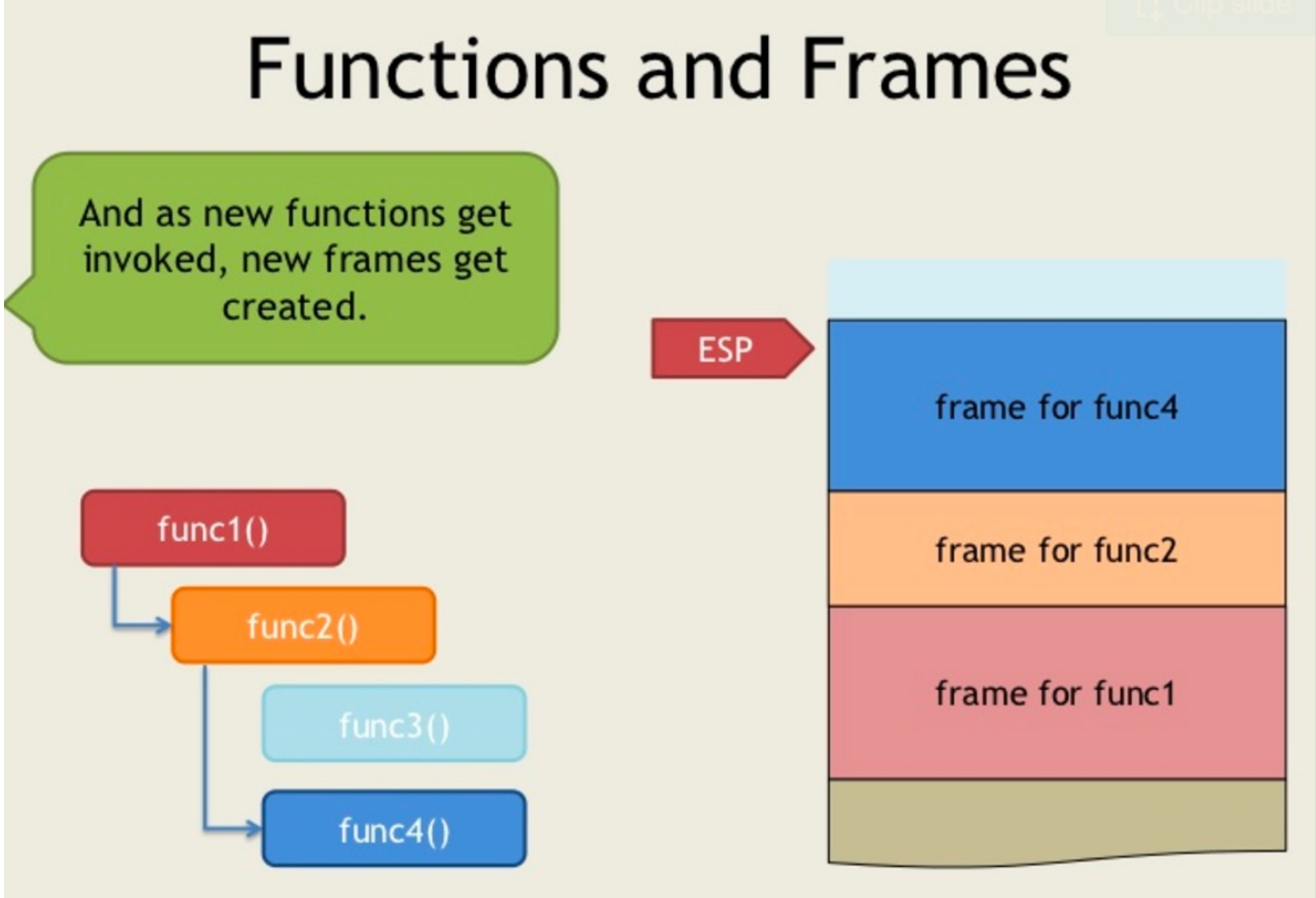
https://www.slideshare.net/saumilshah/how-functions-work-7776073



Functions and Frames

When a function returns, the frame is "unwound" or "collapsed".

https://www.slideshare.net/saumilshah/how-functions-work-7776073

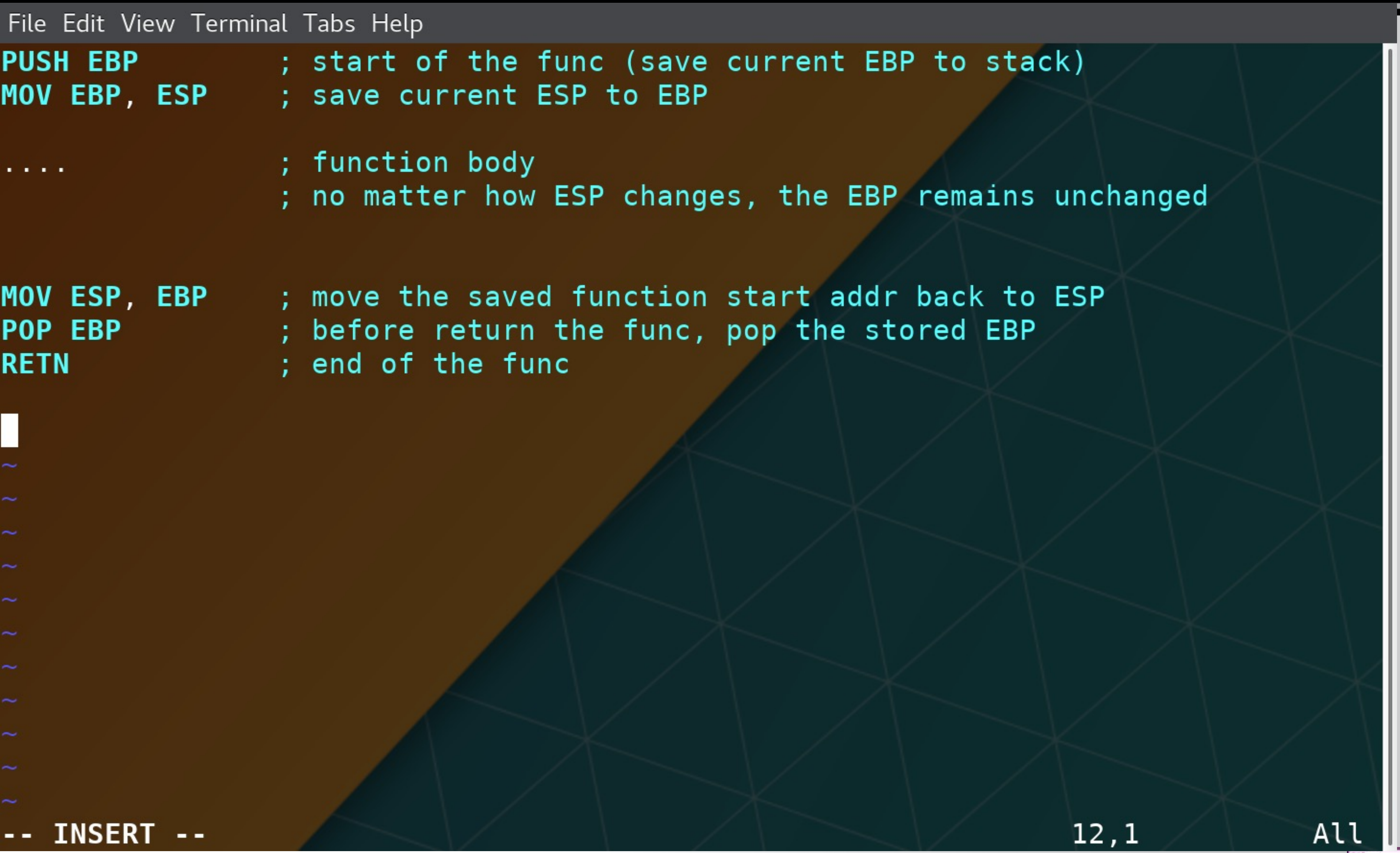


Functions and Frames

And as new functions get invoked, new frames get created.

https://www.slideshare.net/saumilshah/how-functions-work-7776073

Stack Frame



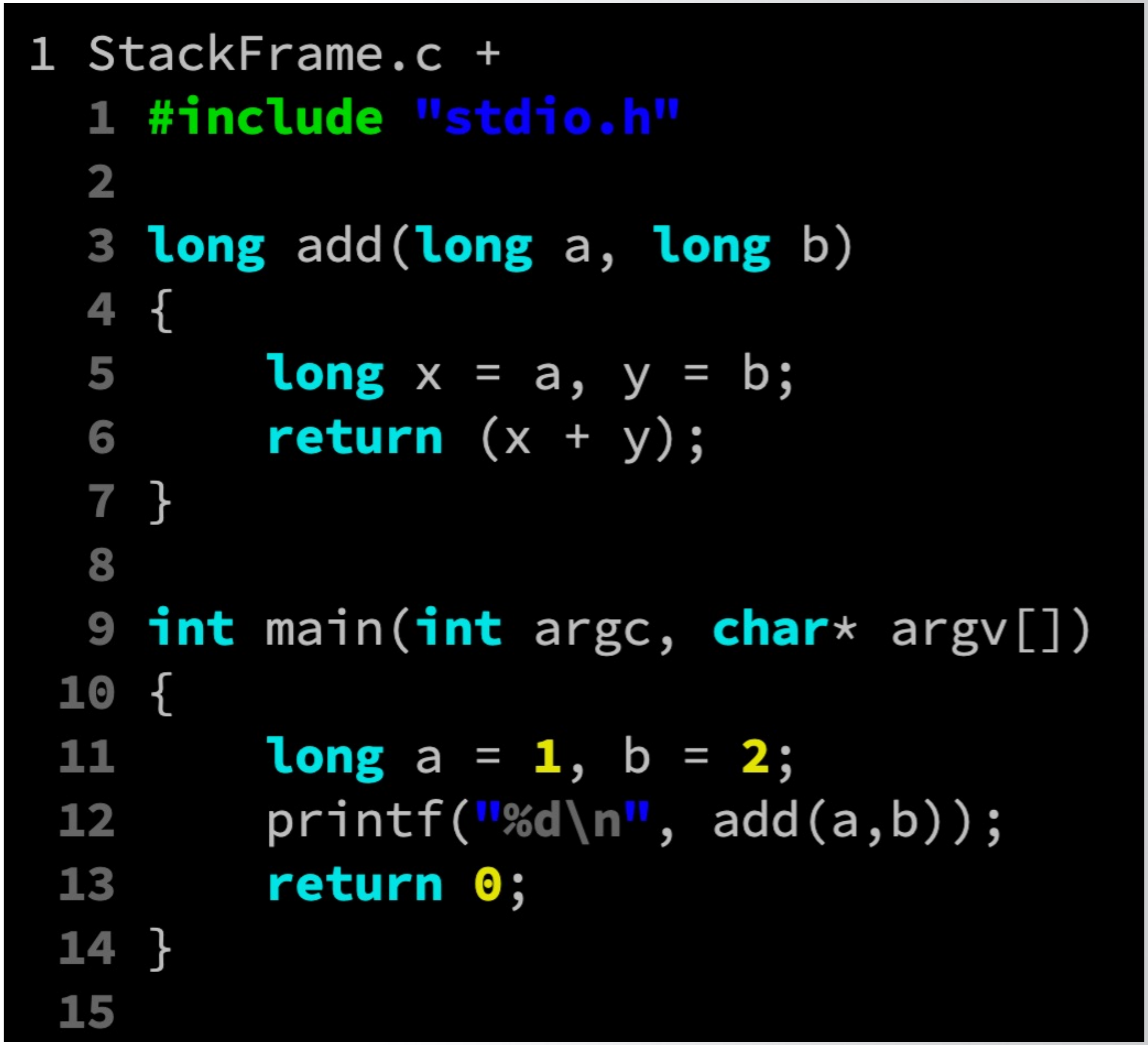
```
PUSH EBP          ; start of the func (save current EBP to stack)
MOV EBP, ESP      ; save current ESP to EBP

....              ; function body
                  ; no matter how ESP changes, the EBP remains unchanged


MOV ESP, EBP      ; move the saved function start addr back to ESP
POP EBP           ; before return the func, pop the stored EBP
RETN              ; end of the func


~
~
~
~
~
~
~
~
~
~
~
-- INSERT --                                                 12,1          All
```

StackFrame.c



```c
#include "stdio.h"

long add(long a, long b)
{
	long x = a, y = b;
	return (x + y);
}

int main(int argc, char* argv[])
{
	long a = 1, b = 2;
	printf("%d\n", add(a,b));
	return 0;
}

```

Q & A