III. STRUCTURED ASSEMBLY LANGUAGE PROGRAMMING TECHNIQUES

Modular Programming

(Recursive Functions)

Recursive Function

- a subprogram which calls itself

- define base case
- define recursive case

Recursive Function

- Countdown:
  - Countdown(n)
    - print(n) , print(n-1) , print(n-2) , … , print(2) , print(1), print(0)
    - print(n) , Countdown(n-1)
  - Countdown(0) = print(0)

Recursive Functions

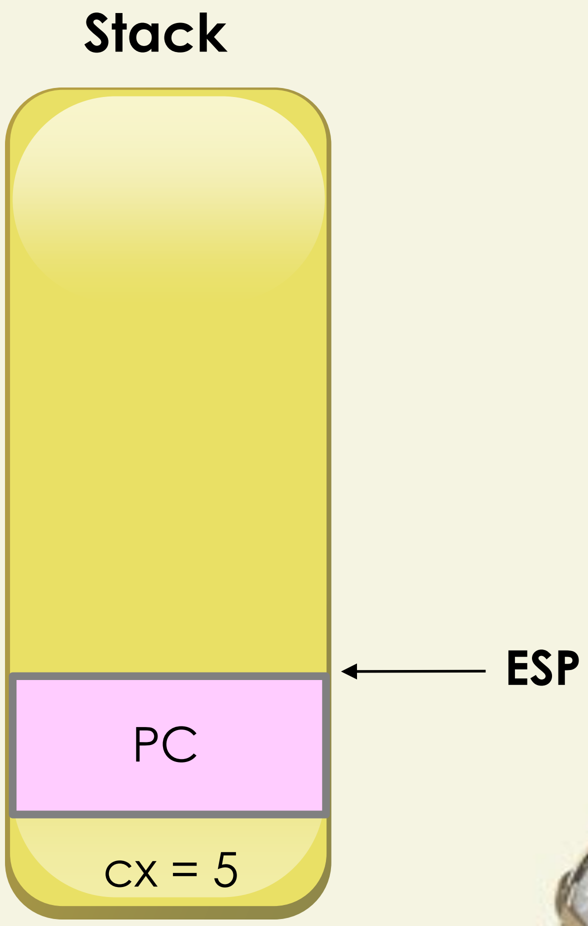
```
void countDown (int n) {
    if (n == 0)
     printf(“%i”,n);
    else{
     printf(“%i”,n);
     countDown(n-1);
    }
}
```

Recursive Functions

mov cx, 5

push cx

call countDown



Recursive Functions

mov cx, 5

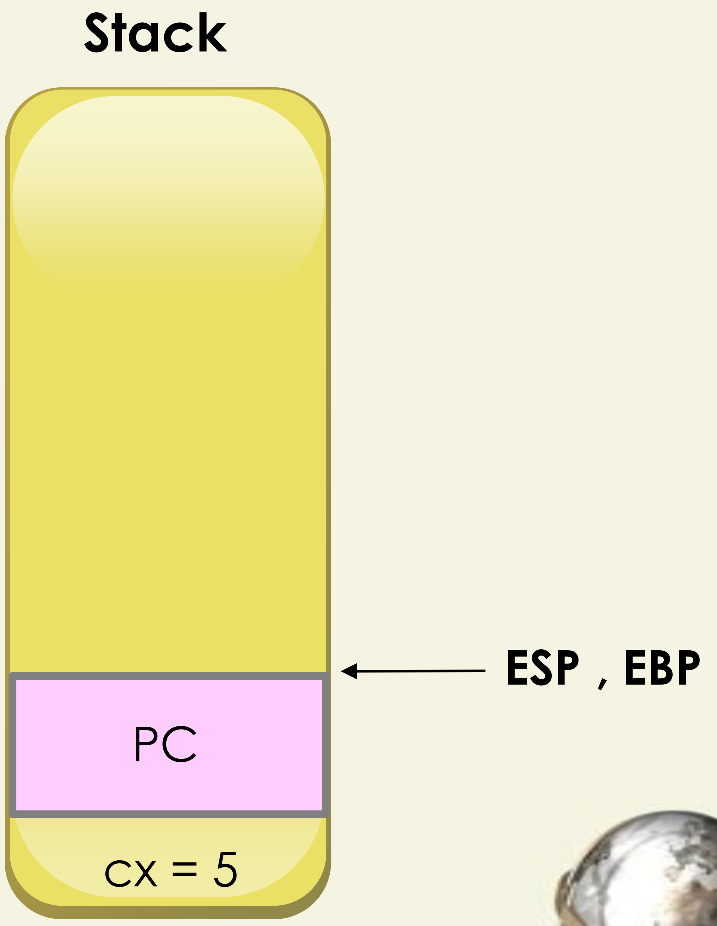
push cx

call countDown

. . .

countDown:

mov ebp, esp



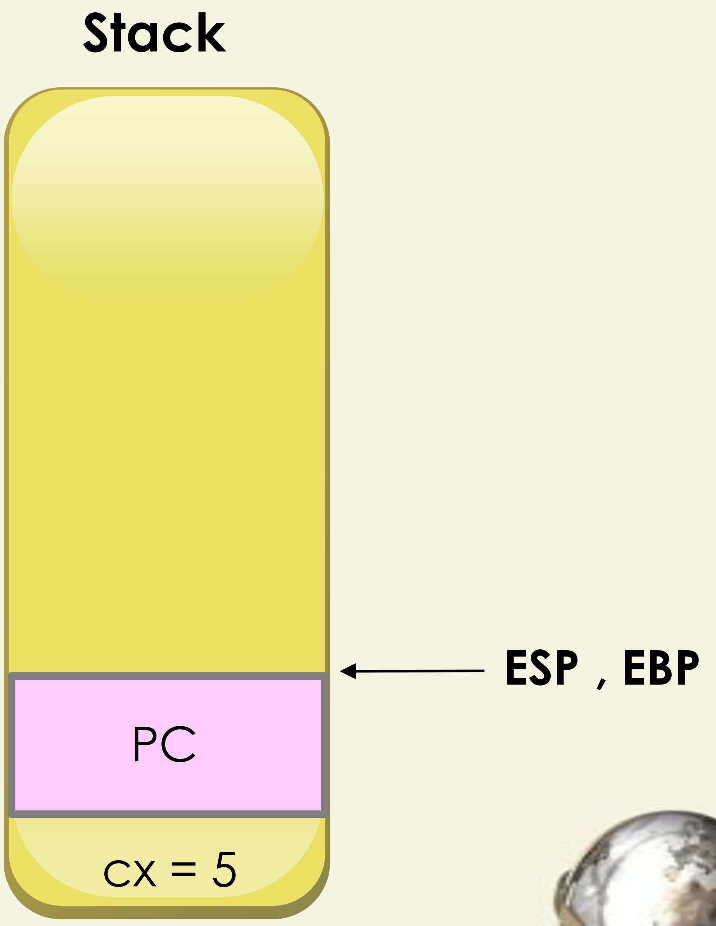
Recursive Functions

countDown:

mov ebp, esp

cmp [ebp+4], 0

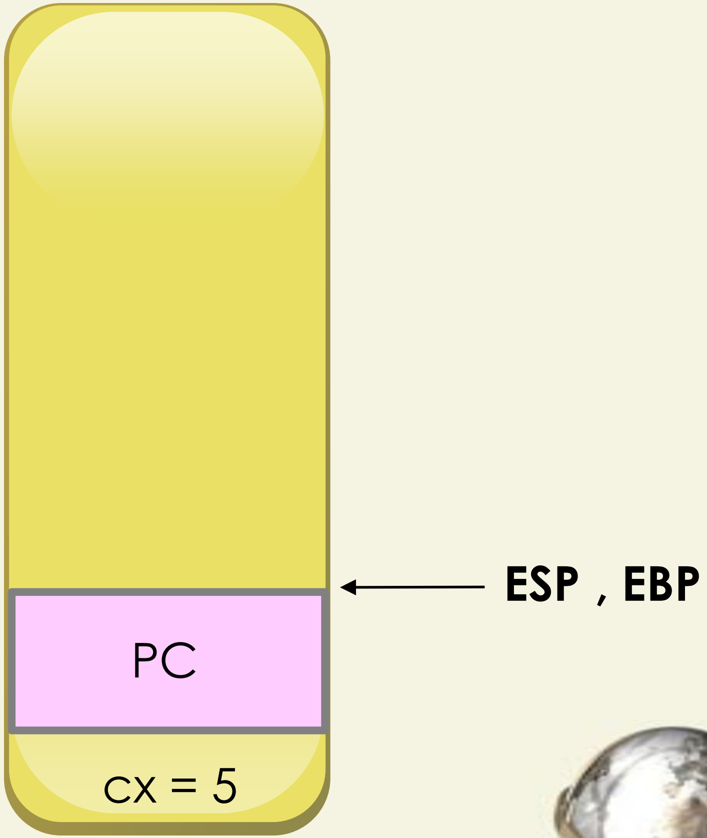
jl countEnd



Recursive Functions

```
countDown:
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
add [ebp+4], 30h
mov eax, 4
mov ebx, 1
lea ecx, [ebp+4]
mov edx, 1
int 80h
sub [ebp+4], 30h
```

Stack

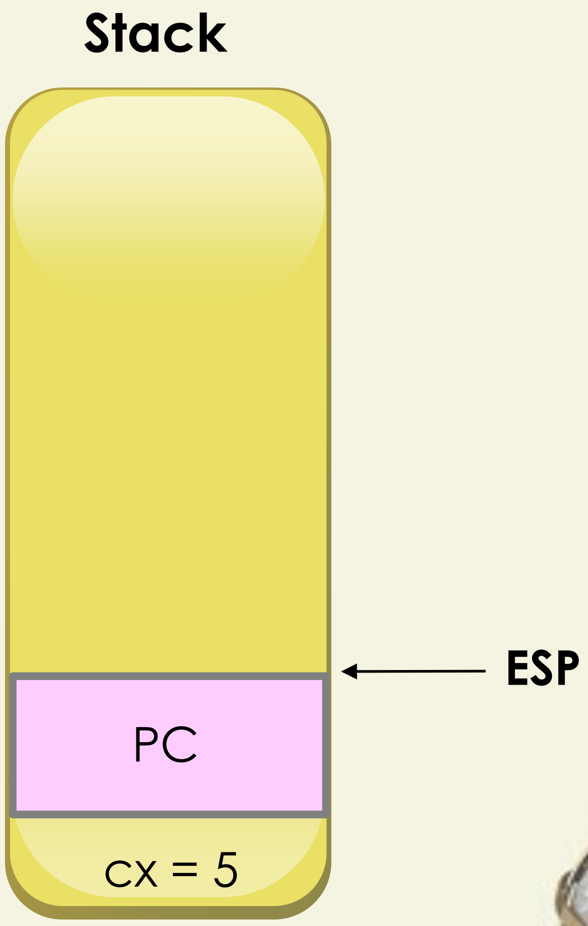


Recall

mov cx, 5

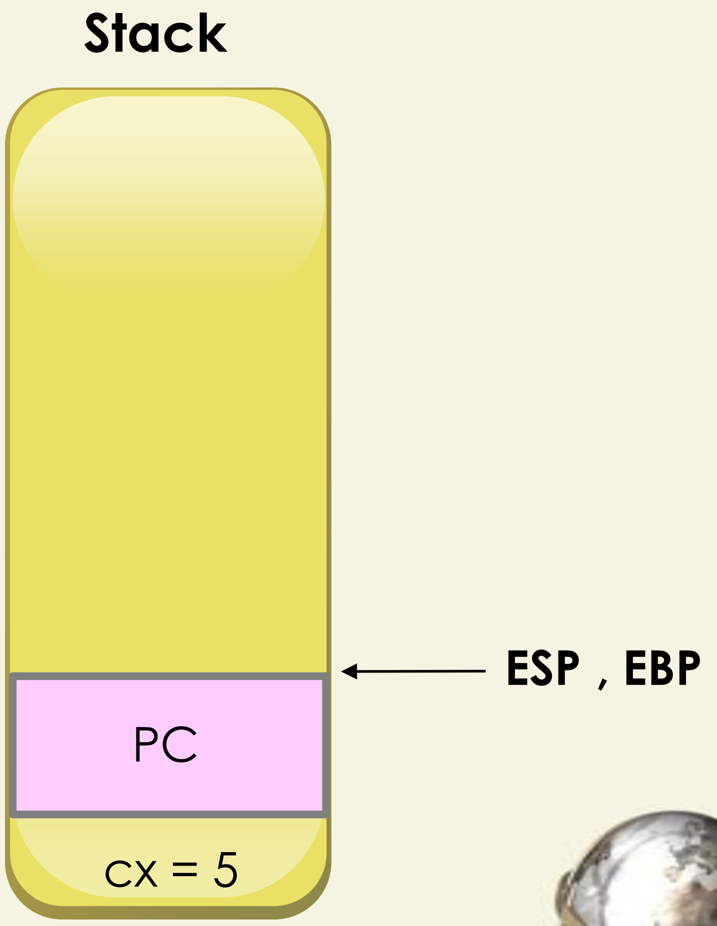
push cx

call countDown



Recursive Functions

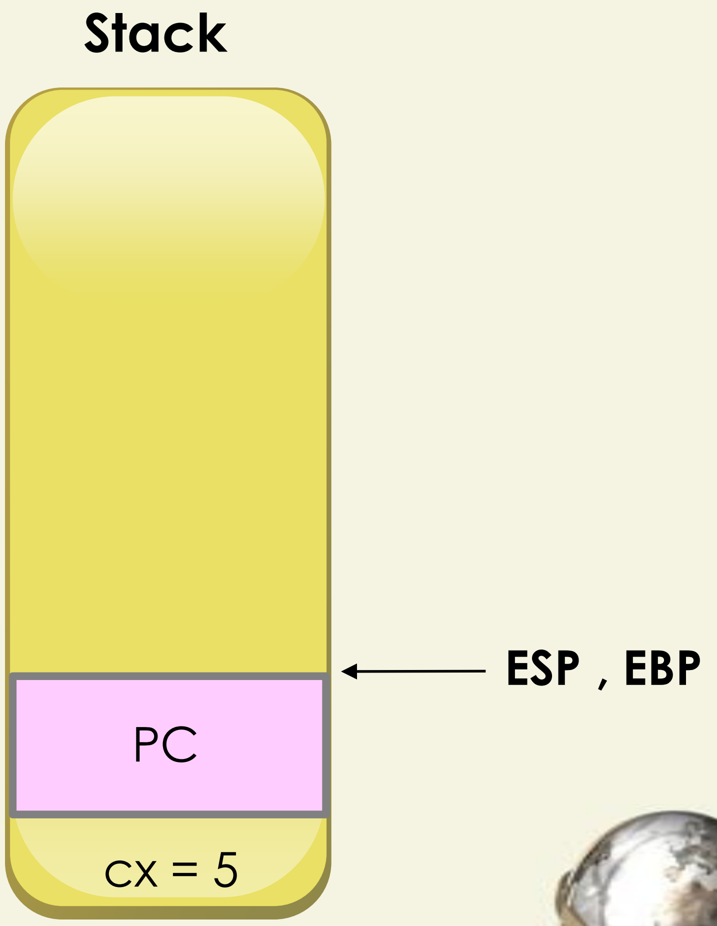
```
countDown:
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
add [ebp+4], 30h
mov eax, 4
mov ebx, 1
lea ecx, [ebp+4]
mov edx, 1
int 80h
sub [ebp+4], 30h
mov cx, [ebp+4]
```



Recursive Functions

countDown:

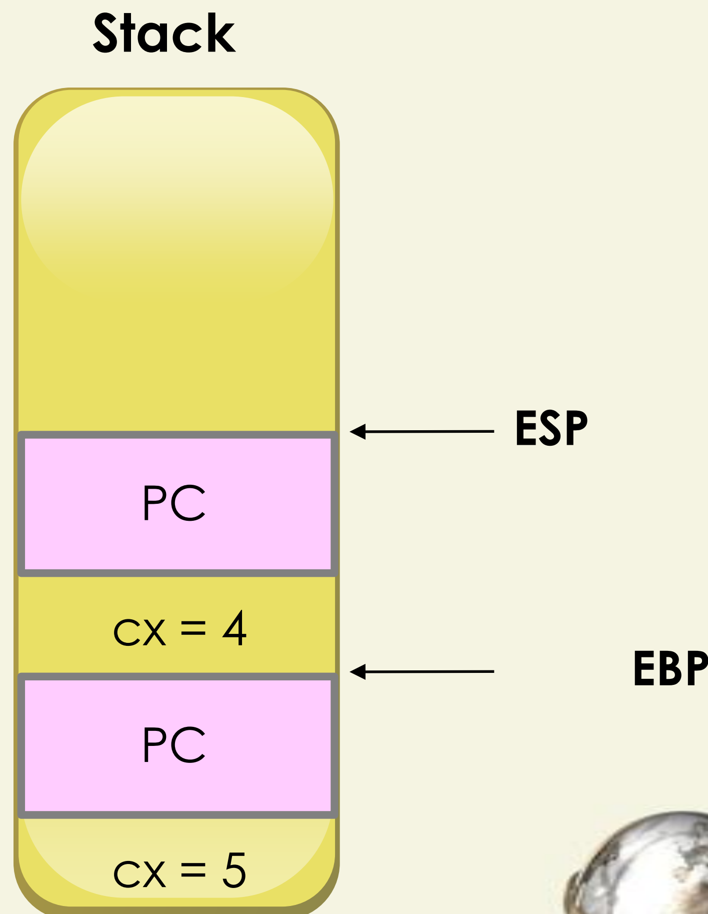
```
. . .
mov eax, 4
mov ebx, 1
lea ecx, [ebp+4]
mov edx, 1
int 80h
sub [ebp+4], 30h
mov cx, [ebp+4]
dec cx
```



Recursive Functions

countDown:

```
. . .
mov eax, 4
mov ebx, 1
lea ecx, [ebp+4]
mov edx, 1
int 80h
sub [ebp+4], 30h
mov cx, [ebp+4]
dec cx
push cx
call countDown
```



Recursive Functions

countDown:

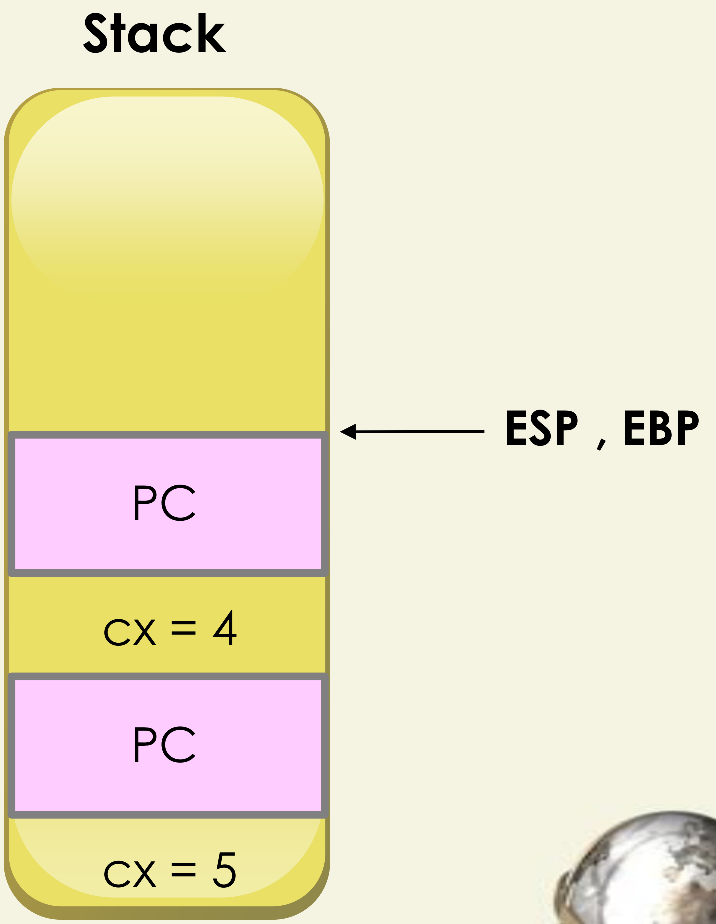
mov ebp, esp

cmp [ebp+4], 0

jl countEnd

. . .

call countDown



Recursive Functions

countDown:

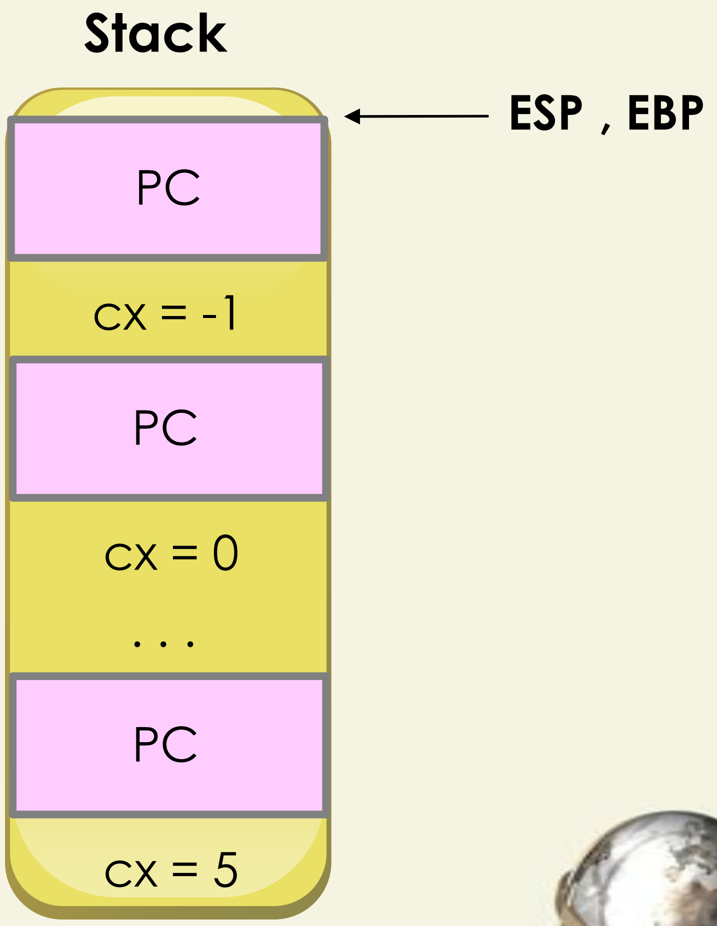
mov ebp, esp

cmp [ebp+4], 0

jl countEnd

. . .

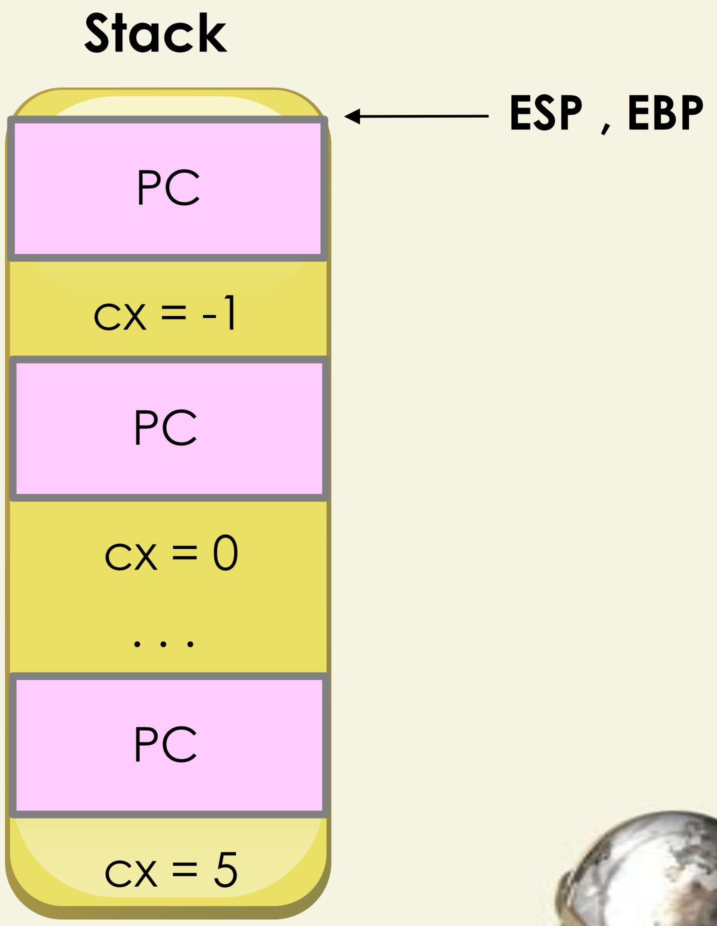
call countDown



Recursive Functions

countDown:

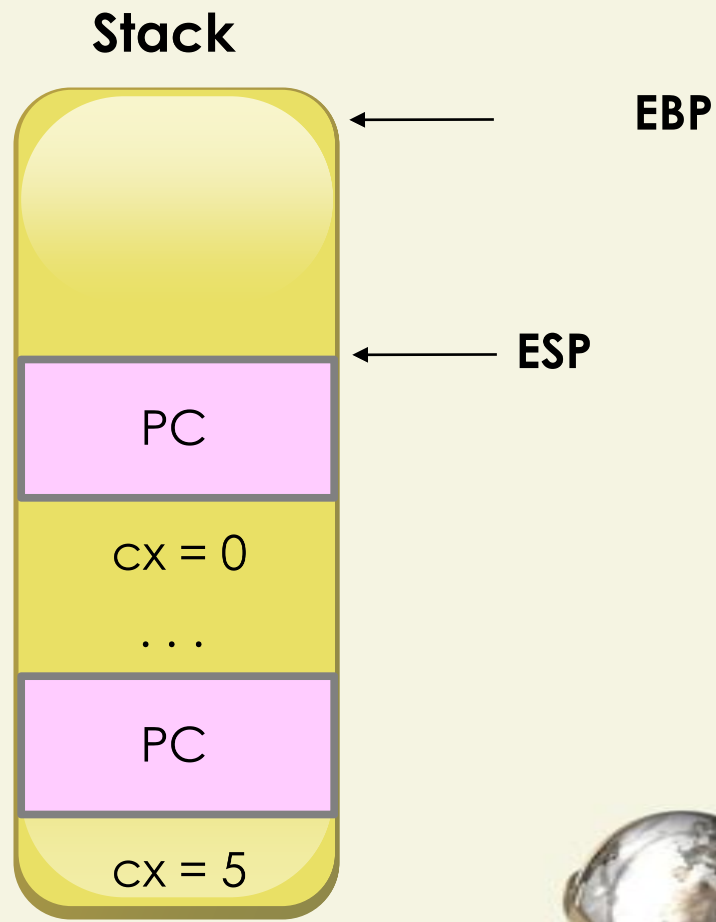
```
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
. . .
call countDown

countEnd:
ret 2
```



Recursive Functions

countDown:
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
. . .
call countDown

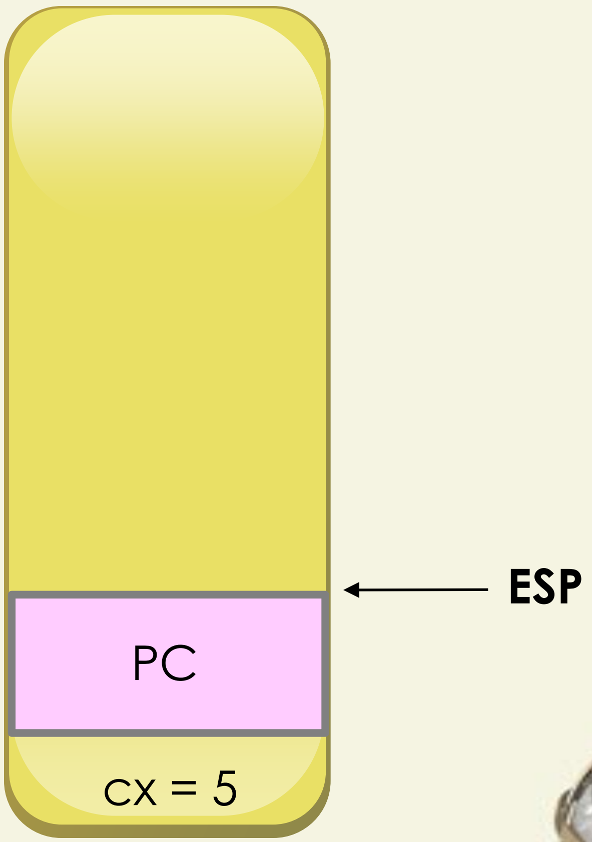
countEnd:
ret 2



Recursive Functions

```
countDown:
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
. . .
call countDown

countEnd:
ret 2
```

Stack



Recursive Functions

```
mov cx, 5
push cx
call countDown
. . .

countDown:
mov ebp, esp
cmp [ebp+4], 0
jl countEnd
add [ebp+4], 30h
```

```
mov eax, 4
mov ebx, 1
lea ecx, [ebp+4]
mov edx, 1
int 80h
sub [ebp+4], 30h
mov cx, [ebp+4]
dec cx
push cx
call countDown
countEnd:
ret 2
```

Recursive Functions

- product of X and Y by recursive addition

- $X * Y = X + [\, X * (Y - 1) \,]$
- $X * 1 = X$
- $X * 0 = 0$

Recursive Functions

```
int product (int x, int y) {
    if (y == 1) return x;
    if (y == 0) return 0;
    return (x + product(x, y-1));
}
```

Example: prod = product(4,3);

Recursive Functions

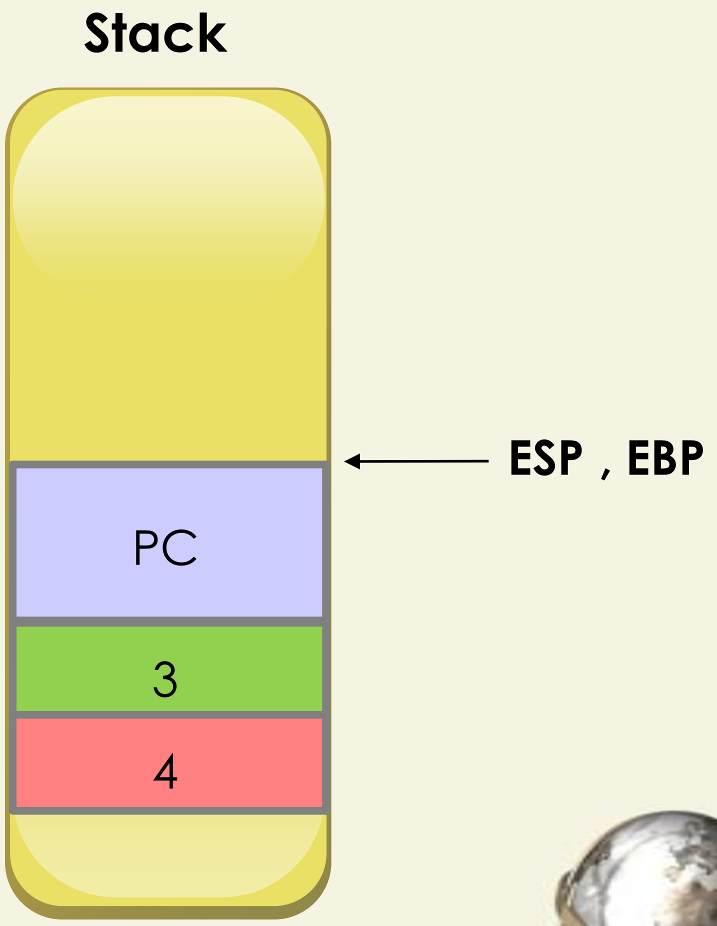
sub esp, 2

push word[x]

push word[y]

call product

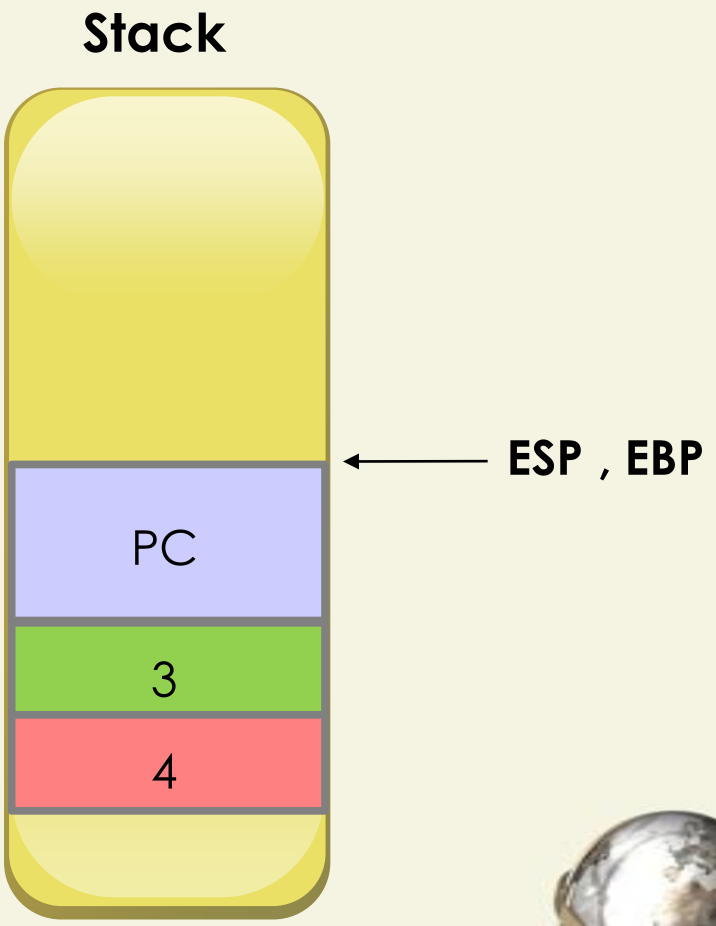
pop word[prod]



Recursive Functions

product:

mov ebp, esp



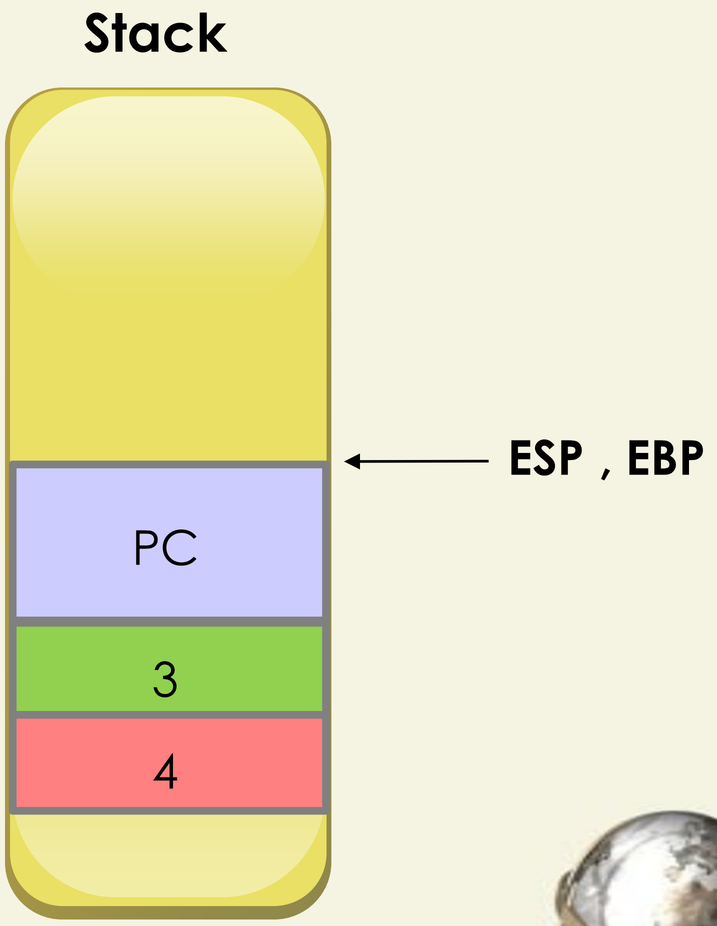
Recursive Functions

product:

mov ebp, esp

cmp word[ebp+4], 1

je return_x



Recursive Functions

product:

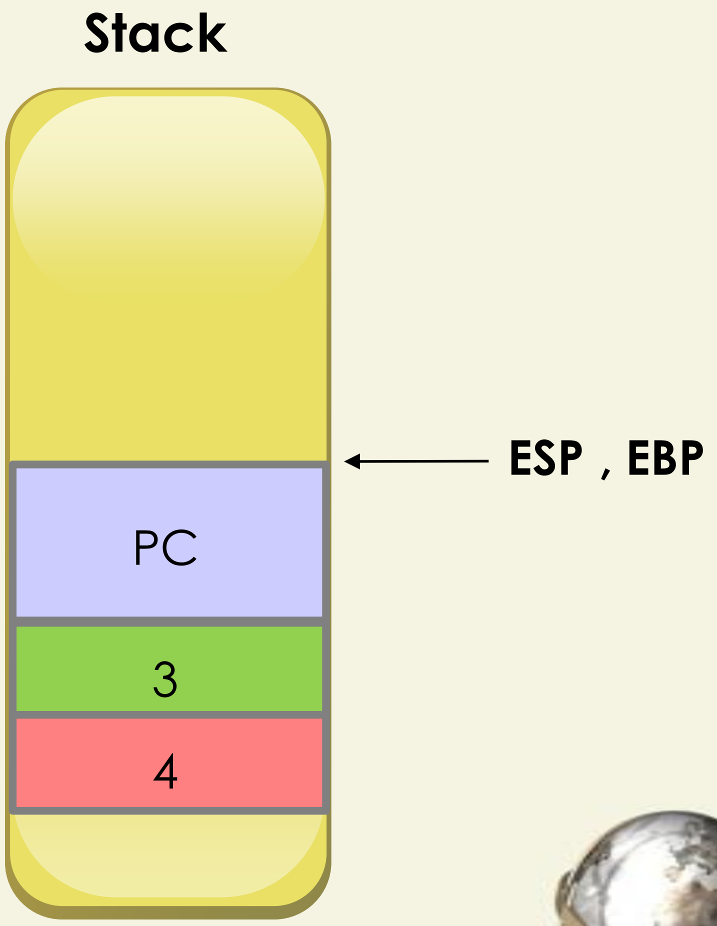
mov ebp, esp

cmp word[ebp+4], 1

je return_x

cmp word[ebp+4], 0

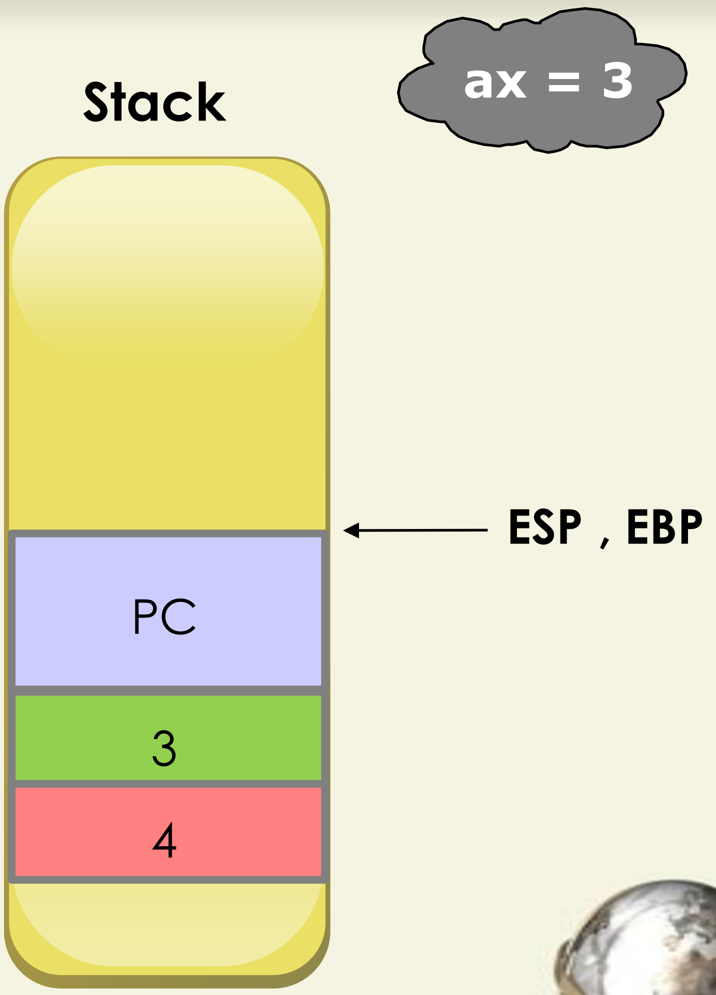
je return_0



Recursive Functions

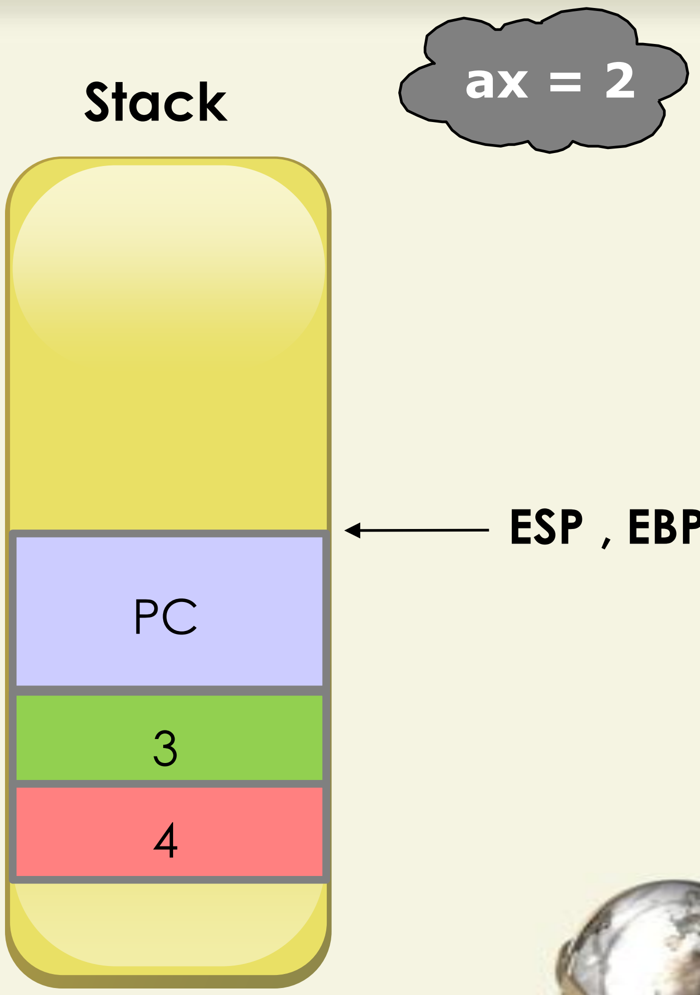
product:

```
mov ebp, esp
cmp word[ebp+4], 1
je return_x
cmp word[ebp+4], 0
je return_0
mov ax, [ebp+4]
```



Recursive Functions

```
product:
mov ebp, esp
cmp word[ebp+4], 1
je return_x
cmp word[ebp+4], 0
je return_0
mov ax, [ebp+4]
dec ax
```



Recursive Functions

product:

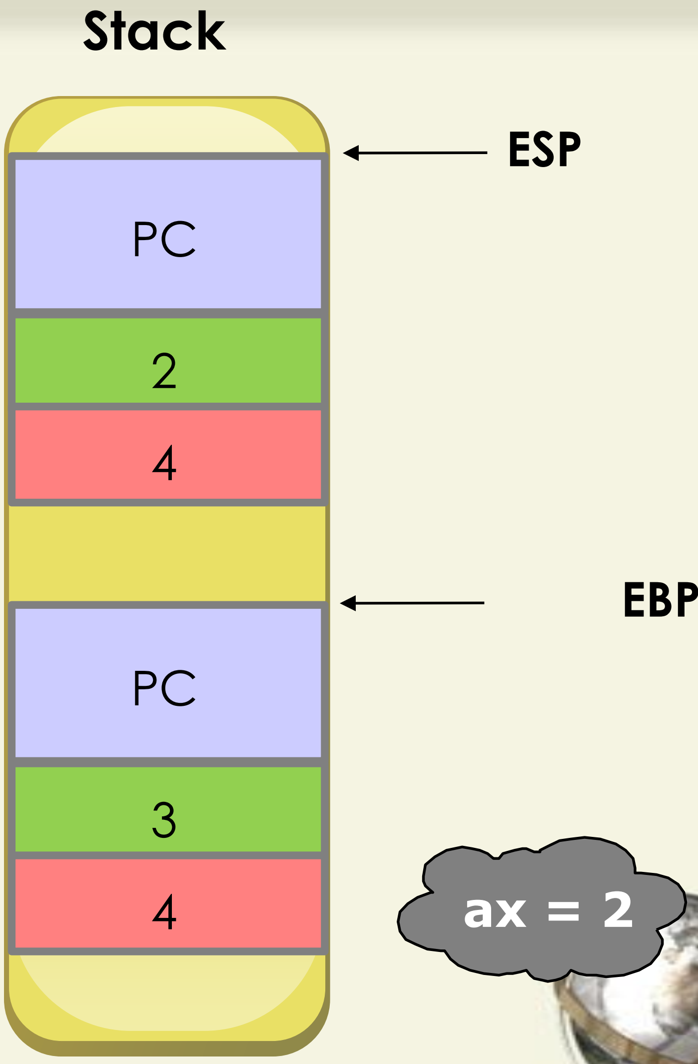
; recursive call

sub esp, 2

push word[ebp+6]

push ax

call product



Recursive Functions

product:

mov ebp,esp

cmp word[ebp+4], 1

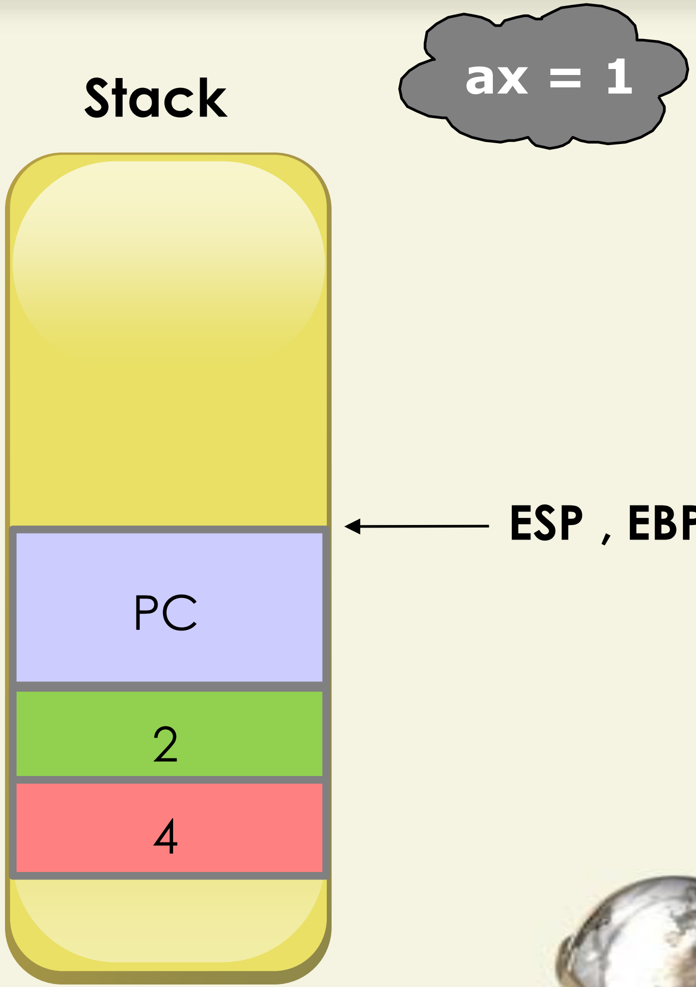
je return_x

cmp word[ebp+4], 0

je return_0

mov ax, [ebp+4]

dec ax



Recursive Functions

product:

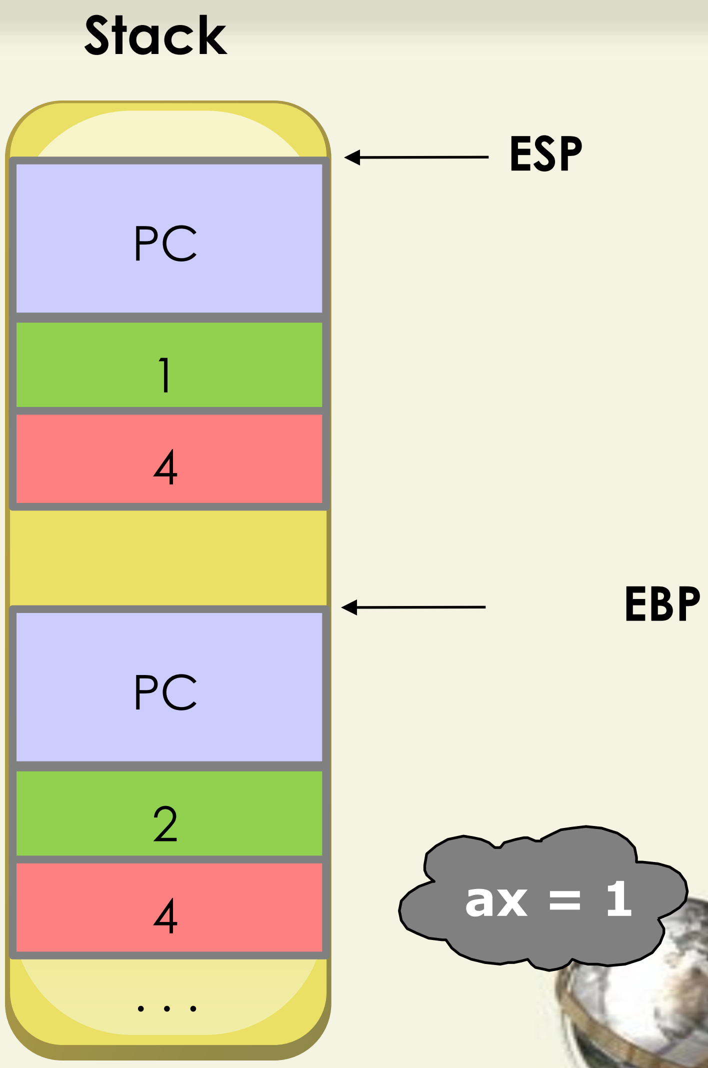
; recursive call

sub esp, 2

push word[ebp+6]

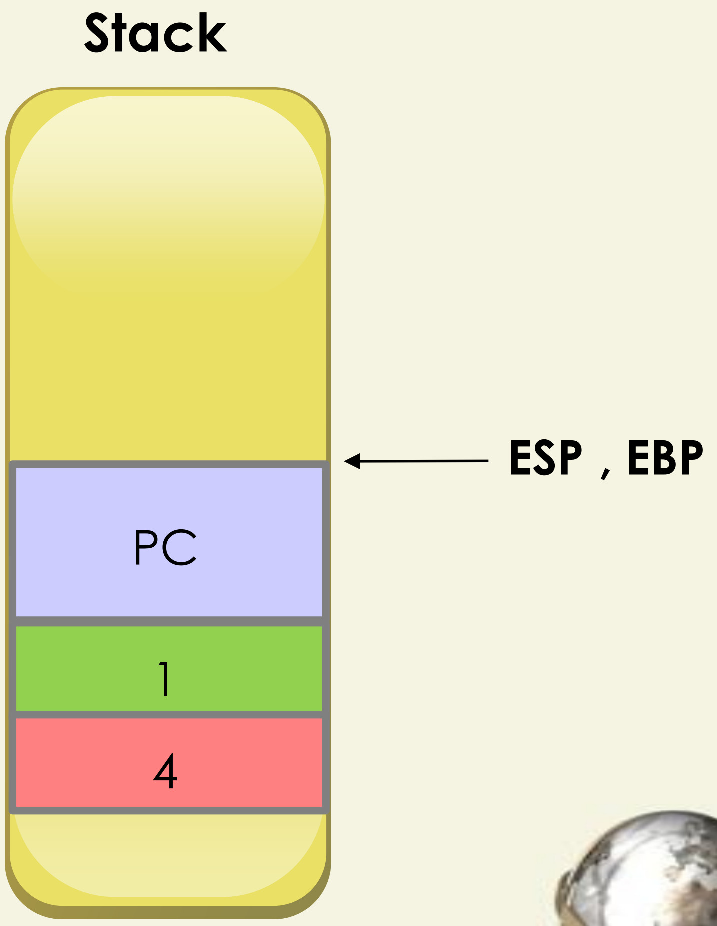
push ax

call product



Recursive Functions

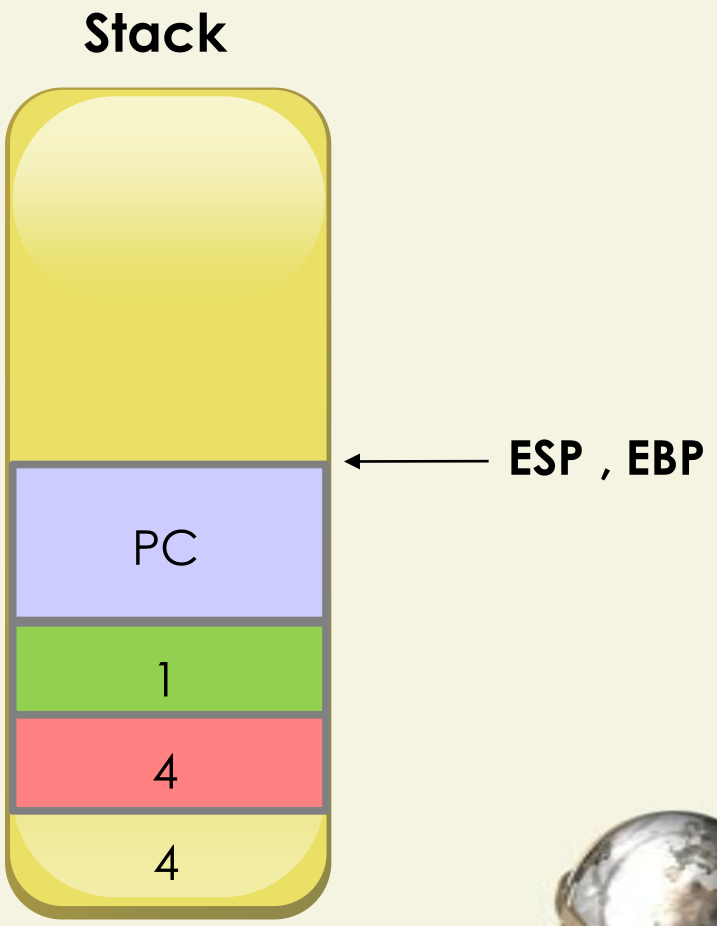
```
product:
mov ebp,esp
cmp word[ebp+4], 1
je return_x
cmp word[ebp+4], 0
je return_0
mov ax, [ebp+4]
dec ax
```



Recursive Functions

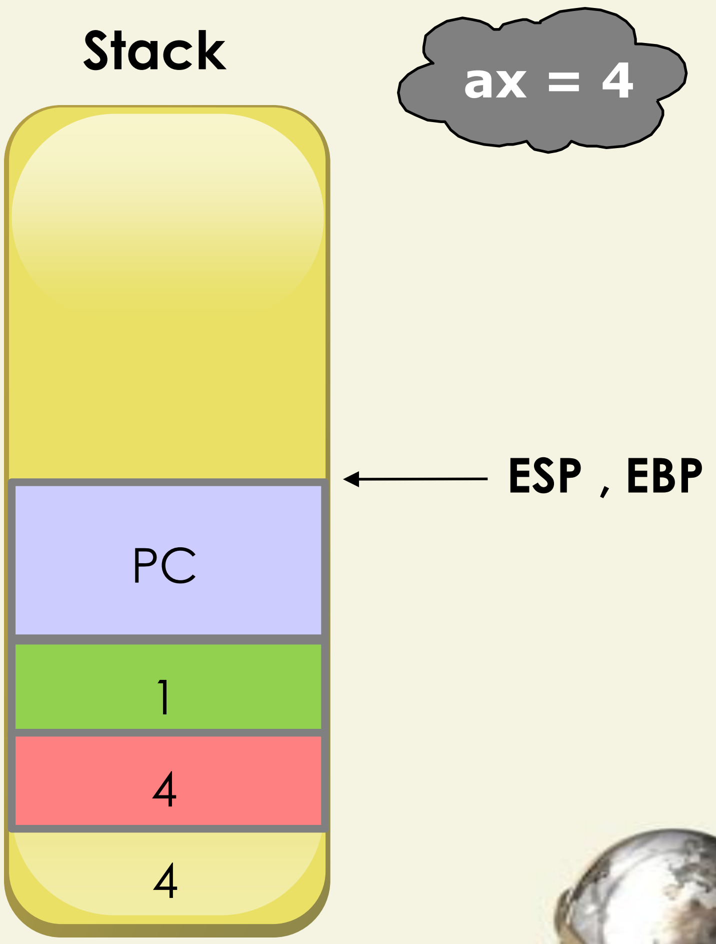
product:

```
mov ebp,esp
cmp word[ebp+4], 1
je return_x
cmp word[ebp+4], 0
je return_0
mov ax, [ebp+4]
dec ax
```



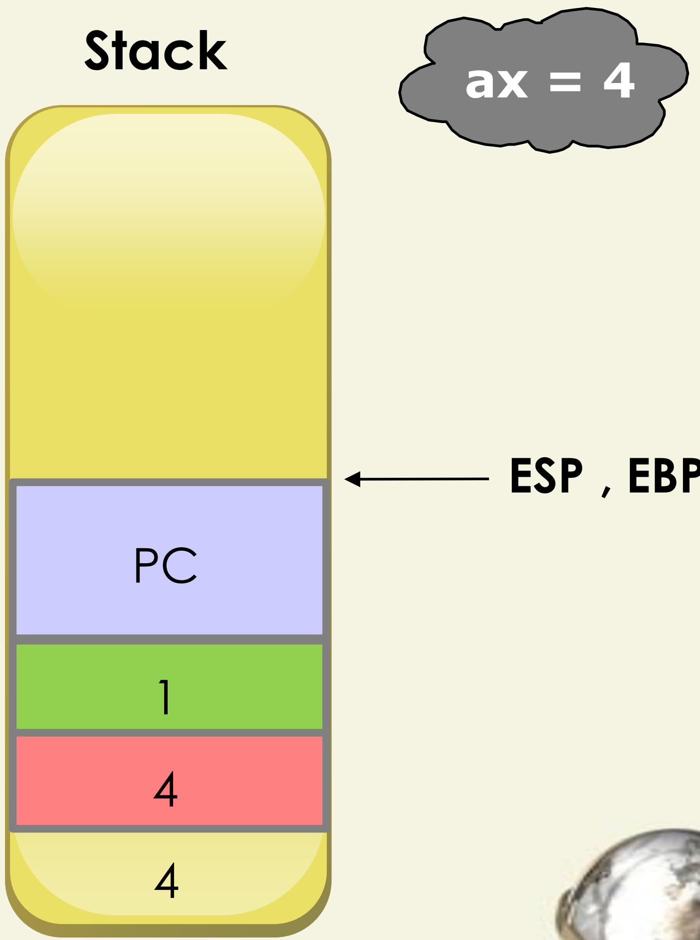
Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
```



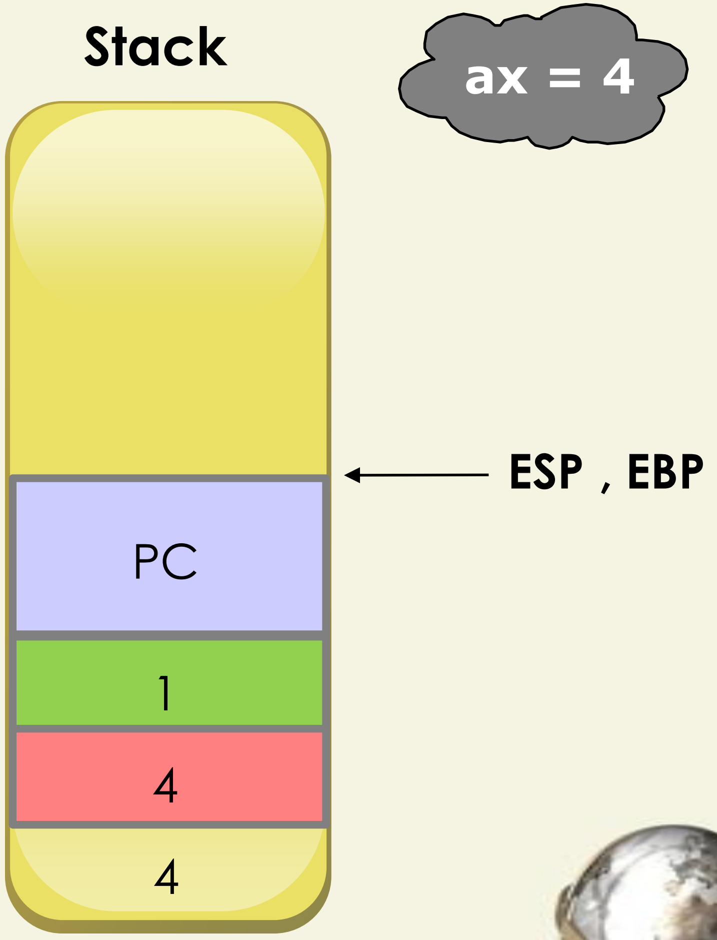
Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
```



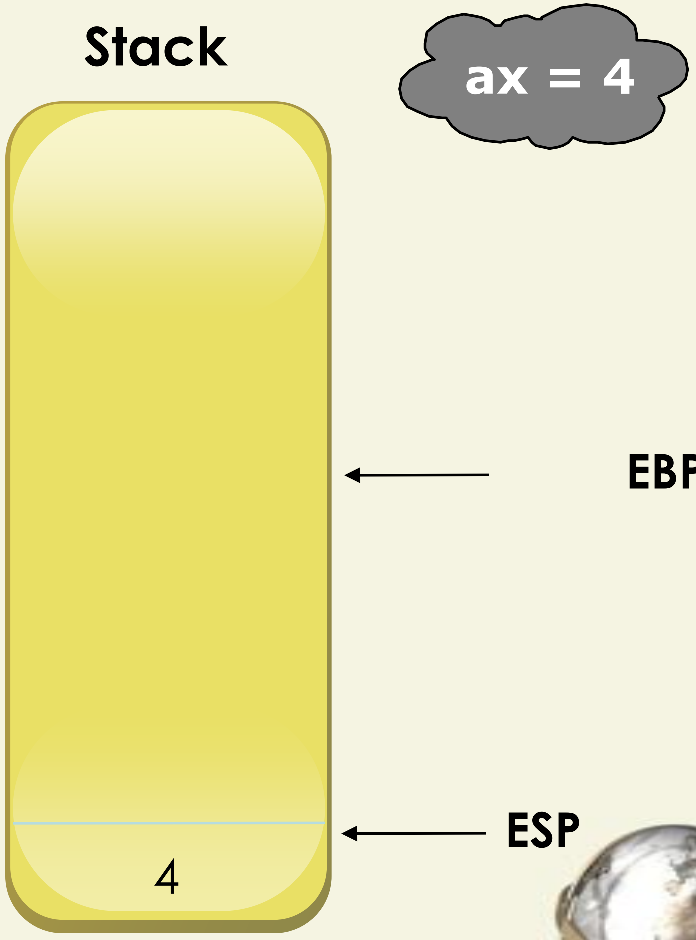
Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```



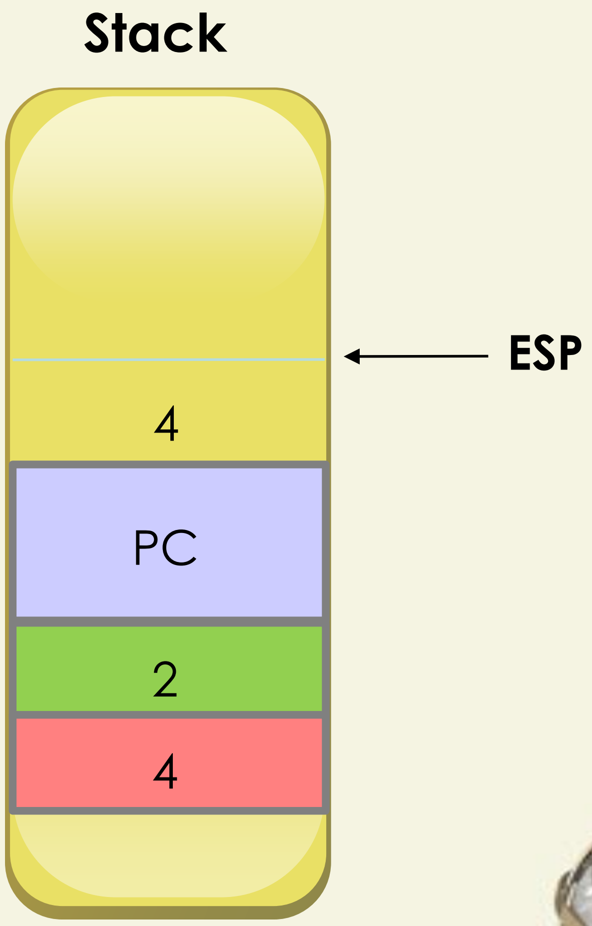
Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```



Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```



Recursive Functions

product:

; recursive call

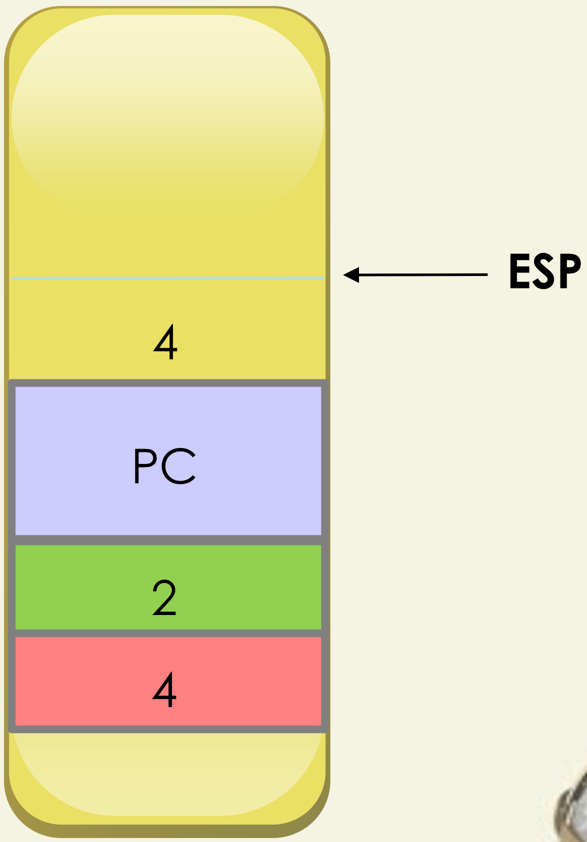
sub esp, 2

push word[ebp+6]

push ax

call product

Stack



Recursive Functions

product:

; recursive call

sub esp, 2

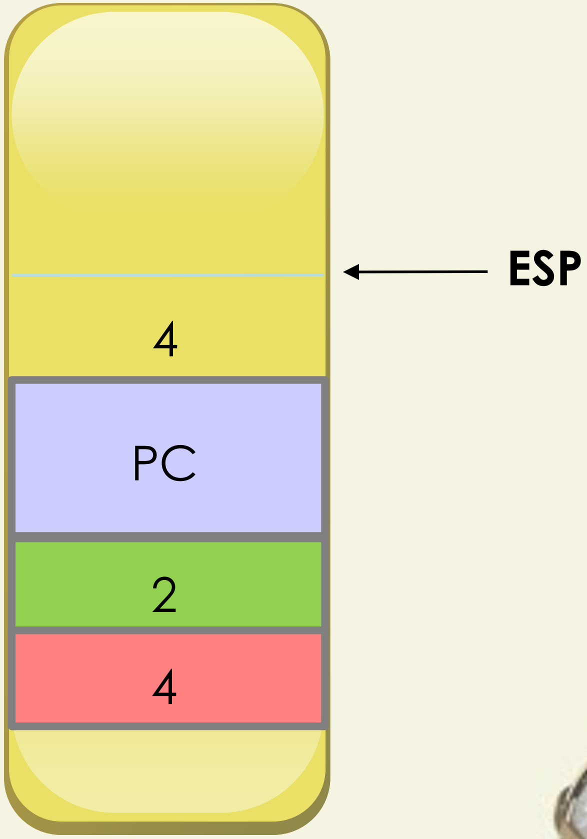
push word[ebp+6]

push ax

call product

pop bx

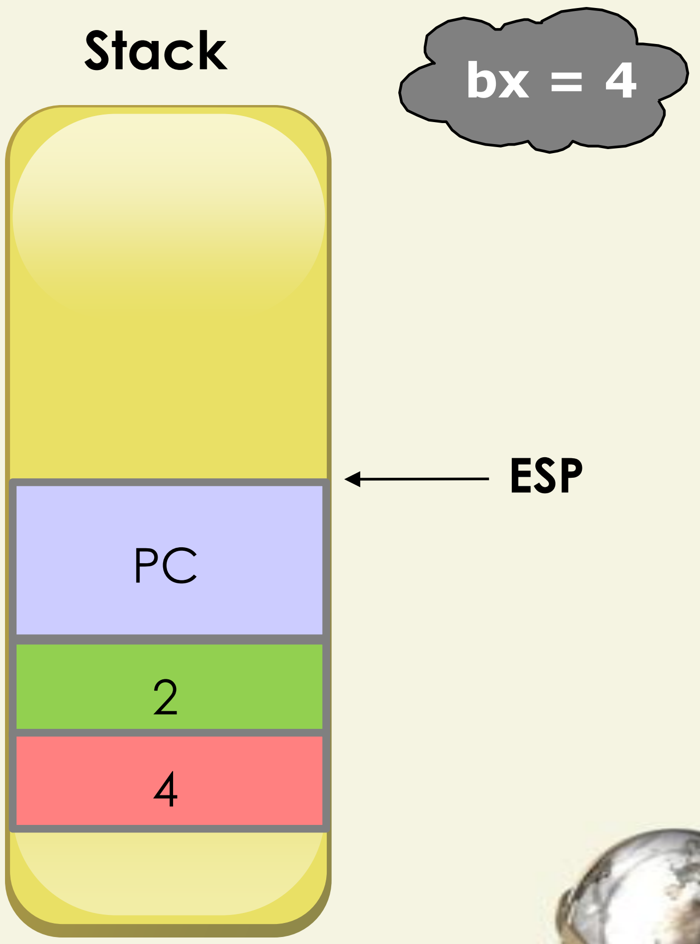
Stack



Recursive Functions

product:

```
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
```



Recursive Functions

product:

; recursive call

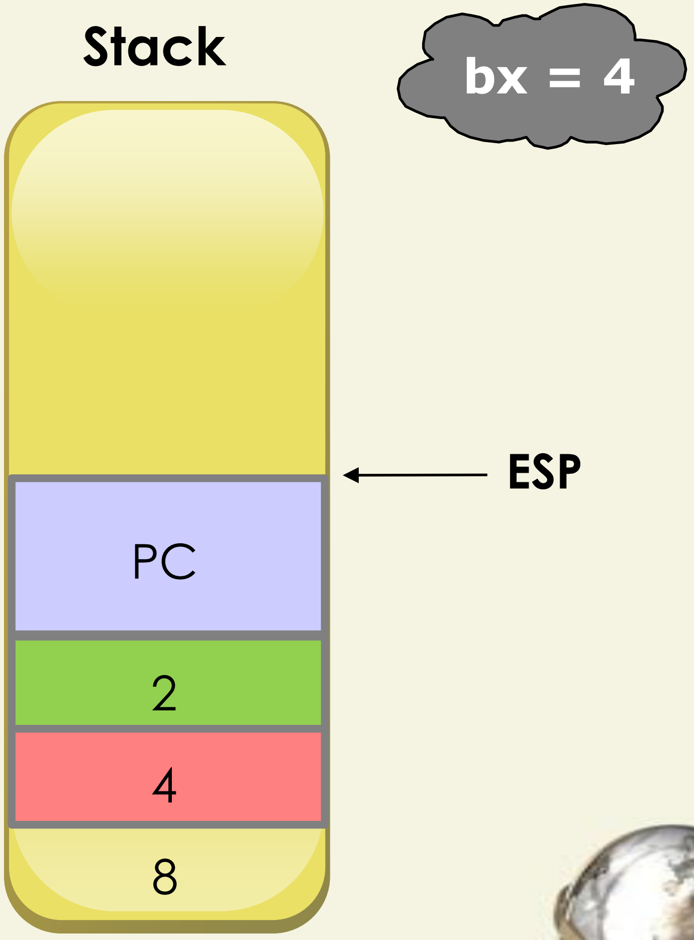
sub esp, 2

push word[ebp+6]

push ax

call product

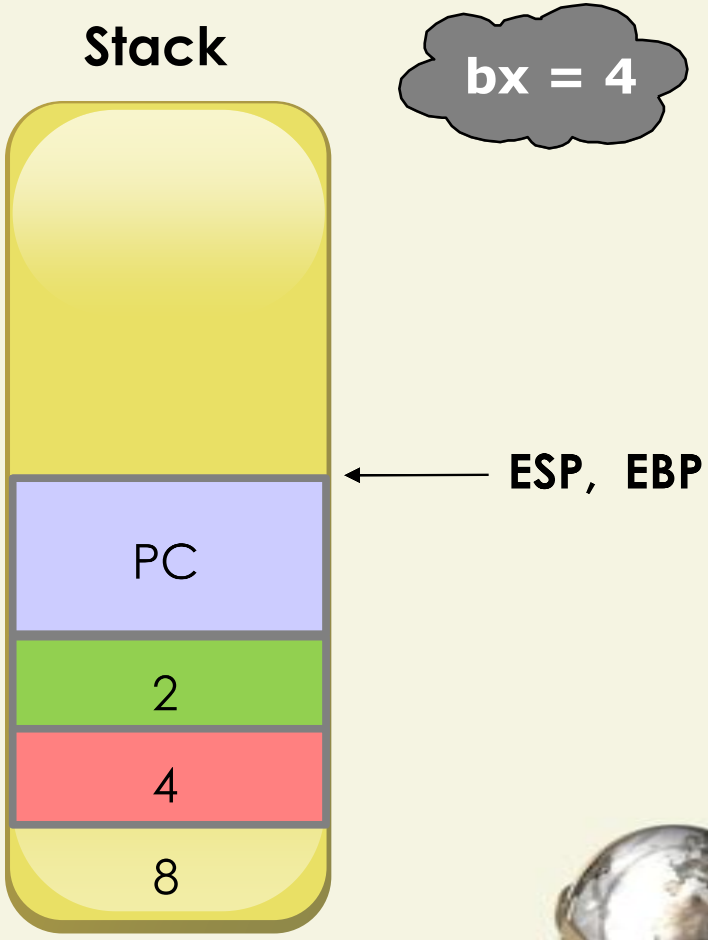
pop bx



Recursive Functions

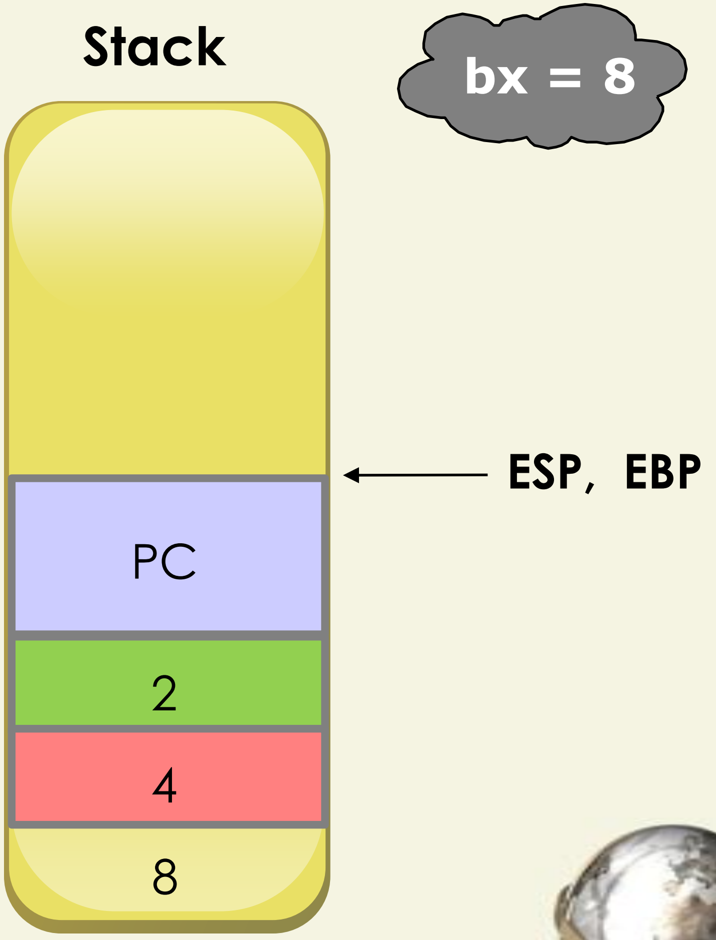
product:

; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]



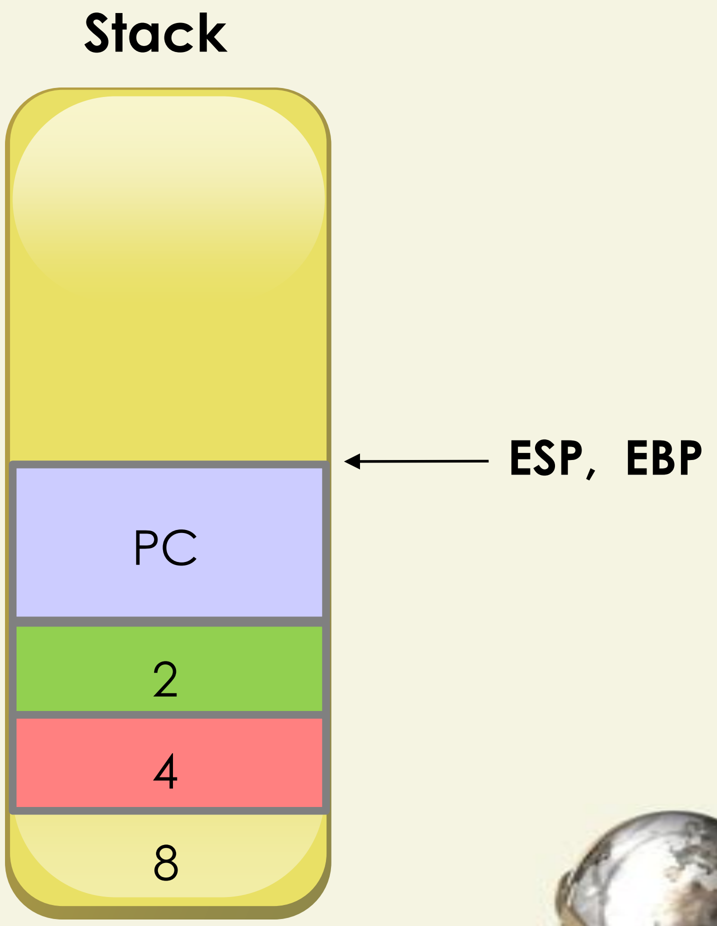
Recursive Functions

```
product:
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
```



Recursive Functions

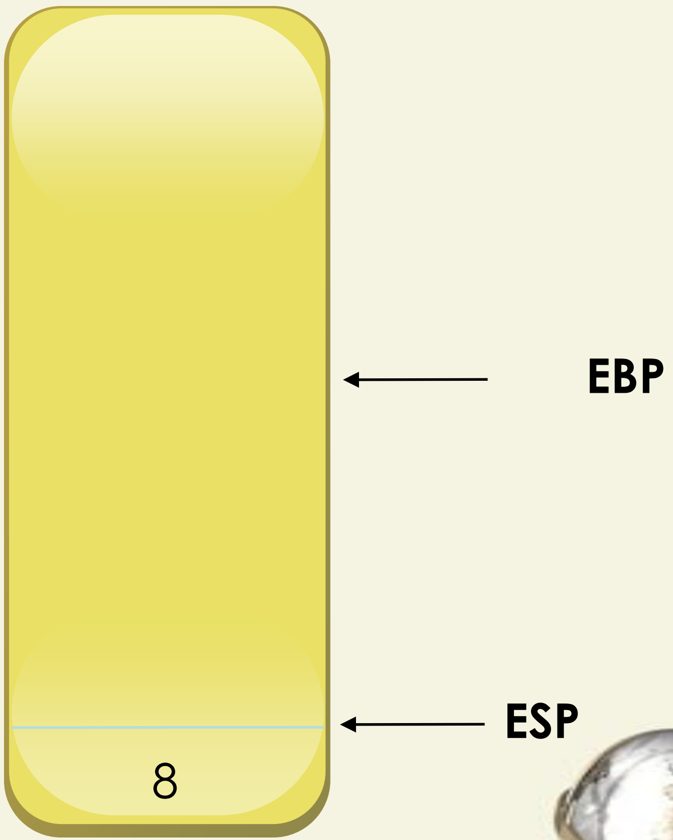
```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```



Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```

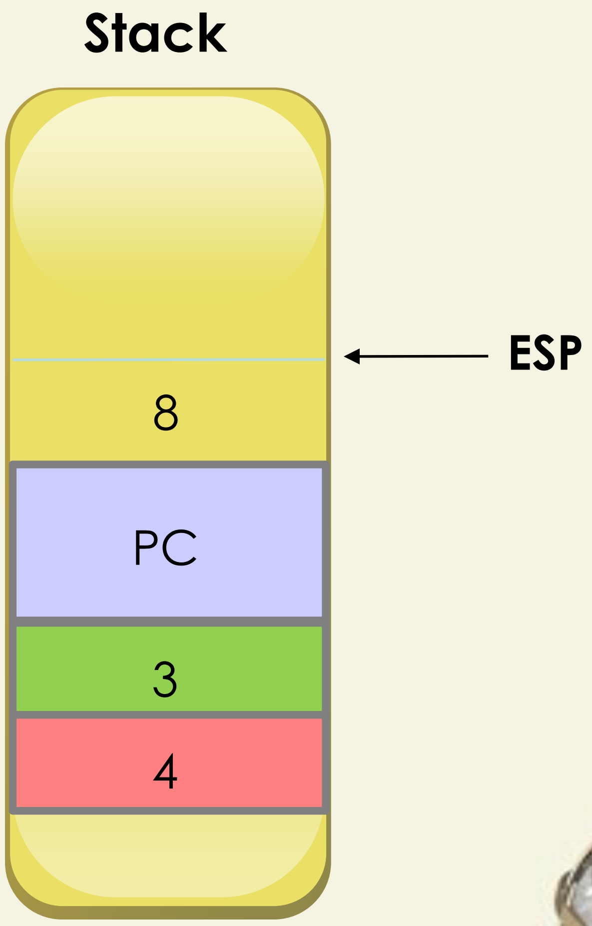
Stack



Recursive Functions

product:

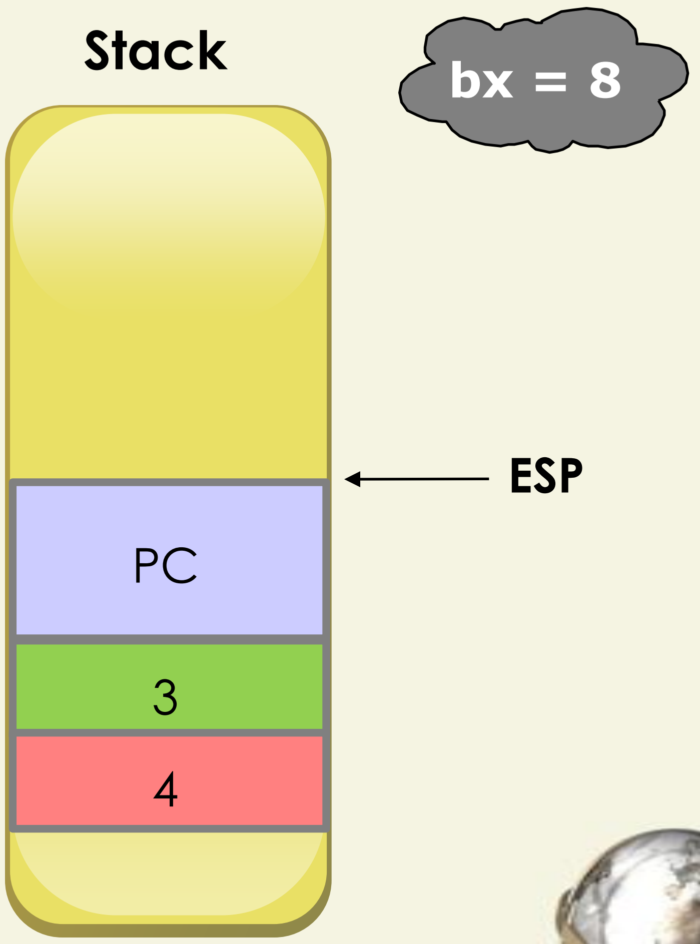
```
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
```



Recursive Functions

product:

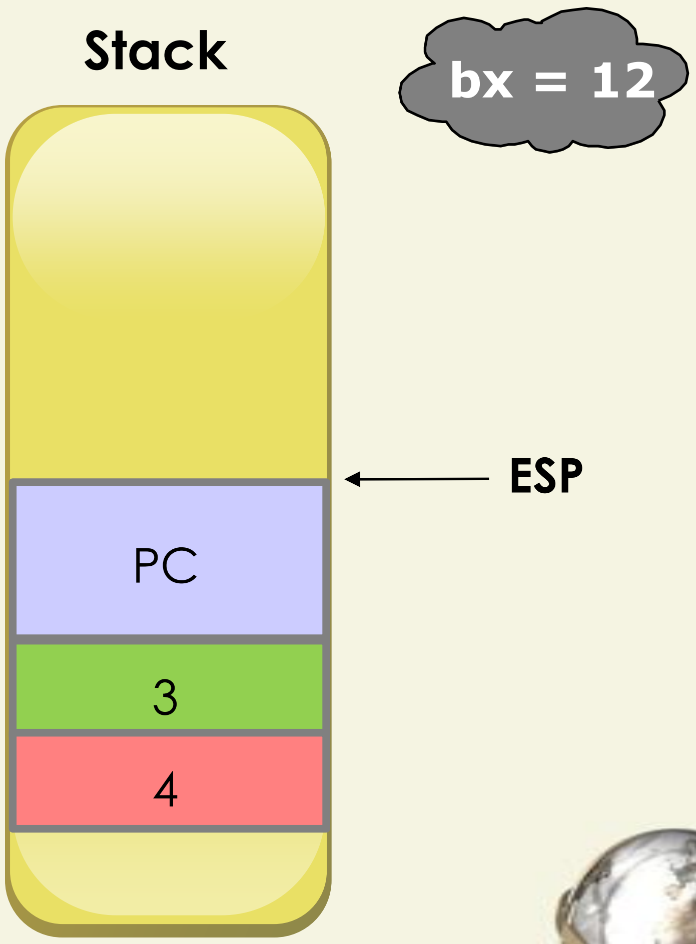
```
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
```



Recursive Functions

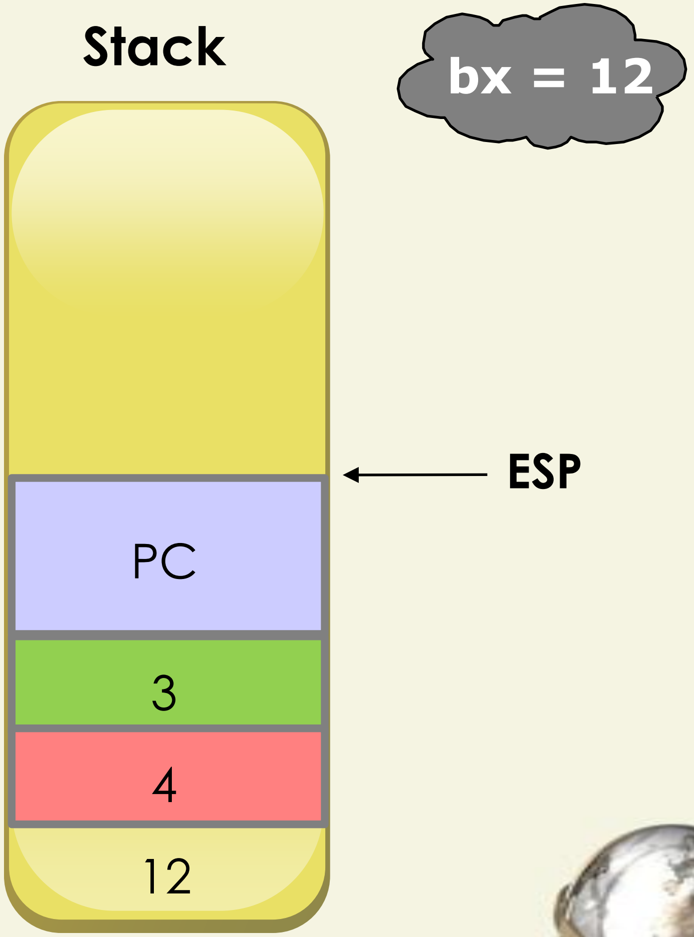
product:

```
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
```



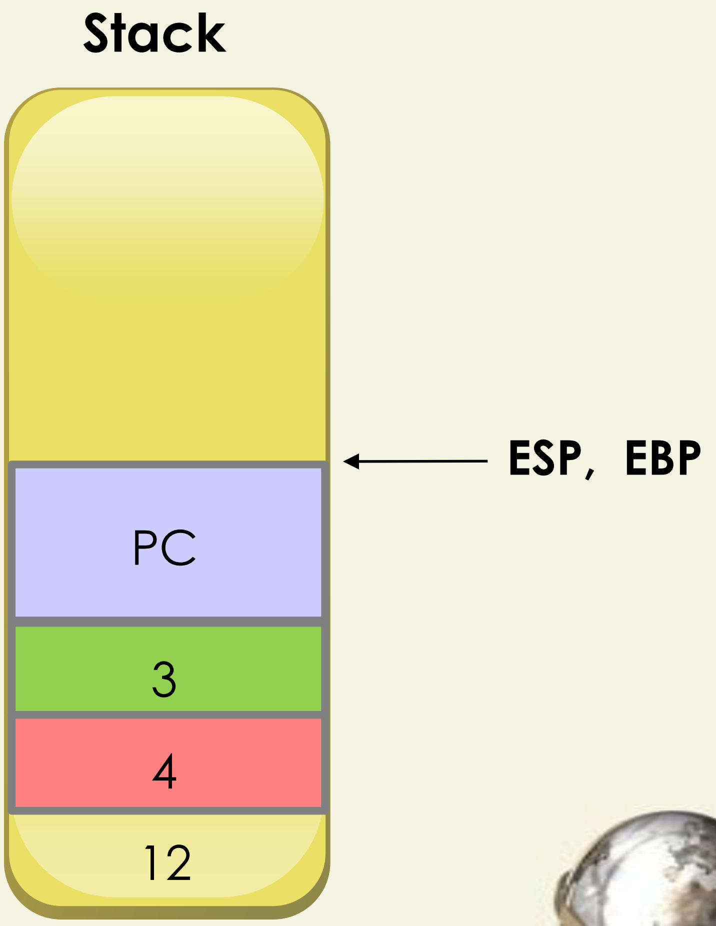
Recursive Functions

```
product:
; recursive call
sub esp, 2
push word[ebp+6]
push ax
call product
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
```



Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```



Recursive Functions

```
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```

Recursive Functions

```
product:
mov ebp, esp
cmp word[ebp+4], 1
je return_x
cmp word[ebp+4], 0
je return_0
mov ax, [ebp+4]
dec ax
sub esp, 2
push word[ebp+6]
push ax
call product
```

```
pop bx
mov ebp, esp
add bx, [ebp+6]
mov [ebp+8], bx
jmp exit
return_x:
  mov ax, [ebp+6]
  mov [ebp+8], ax
  jmp exit
return_0:
  mov word[ebp+8], 0
exit:
  ret 4
```

Recursive Functions

- define base case
- define recursive case

- Factorial:
  - X!
    - x * (x-1) * (x-2) * … * 2 * 1
    - x * (x-1)!
  - 1! = 1
  - 0! = 1

Recursive Functions

```
int factorial (int x) {
if (x == 1) return 1;
if (x == 0) return 1;
return (x*factorial(x-1));
}

f = factorial (n);
```

```
; function call

sub esp, 2
push word [n]
call factorial
pop word [f]
```

Recursive Functions

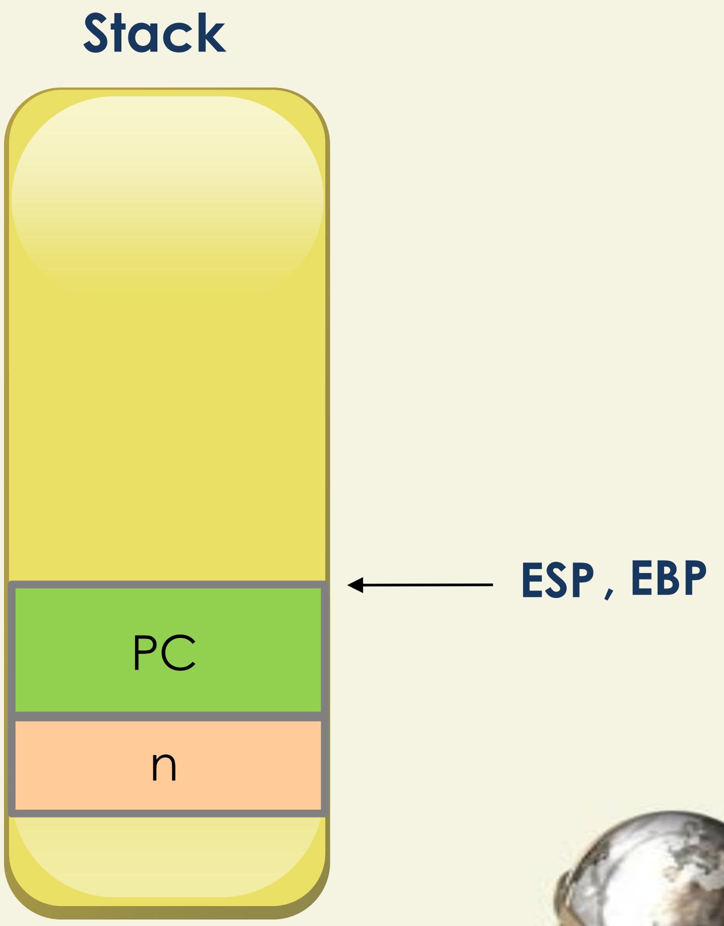
; function call

sub esp, 2

push word [n]

call factorial

pop word [f]



Recursive Functions

; function

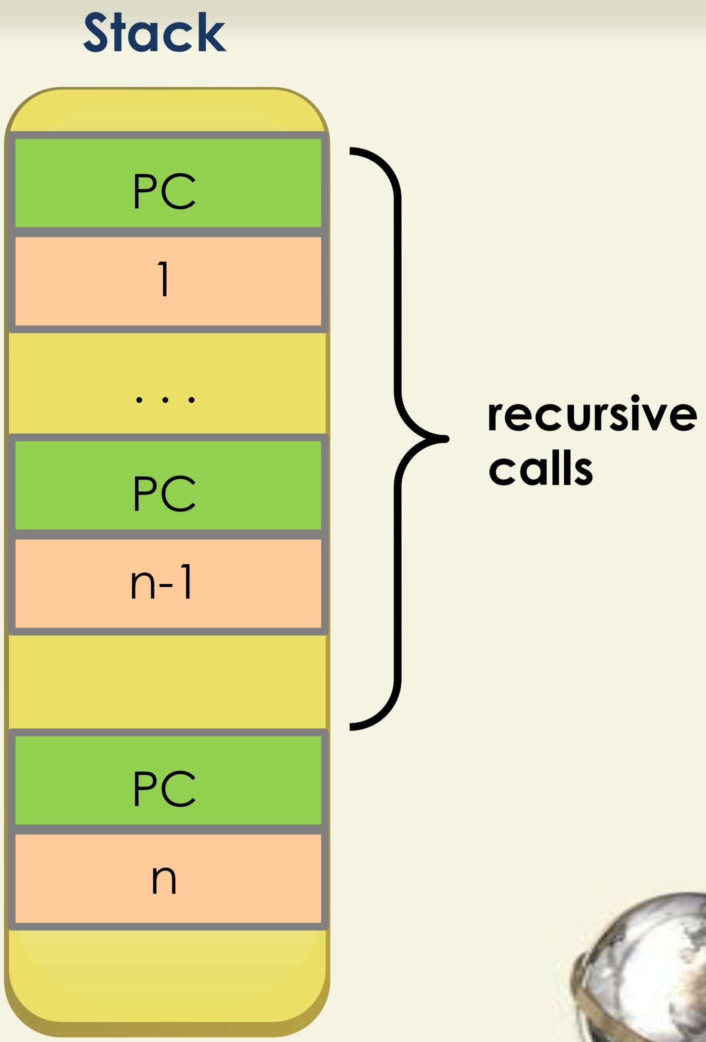
factorial:

mov ebp, esp

...

; recursive call

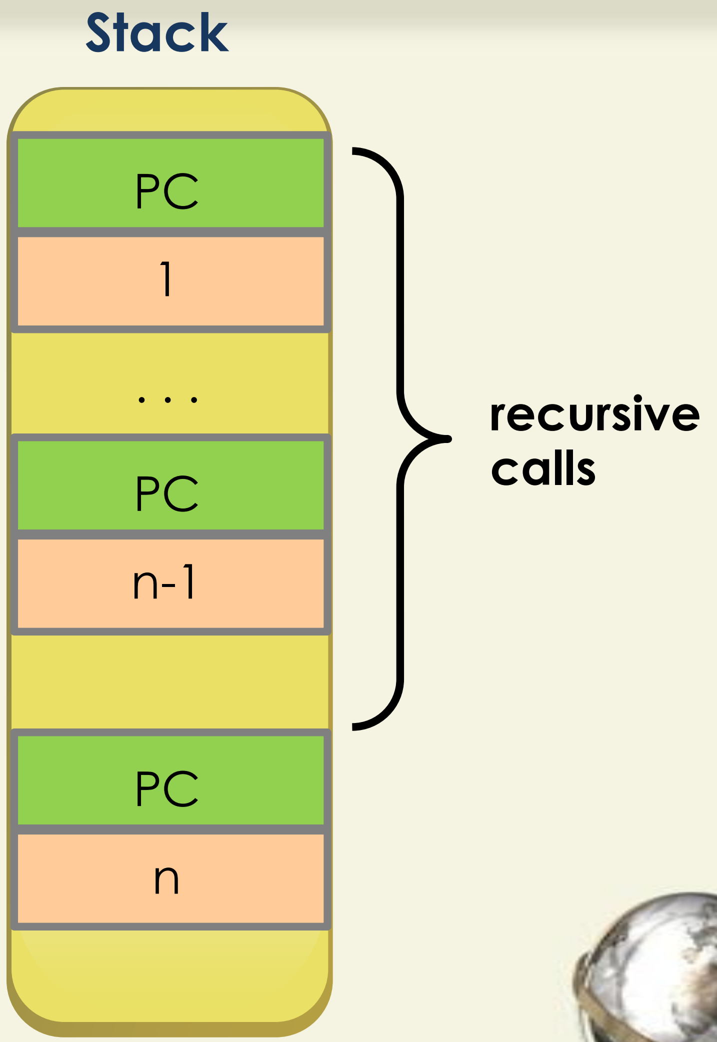
...



Recursive Functions

factorial:

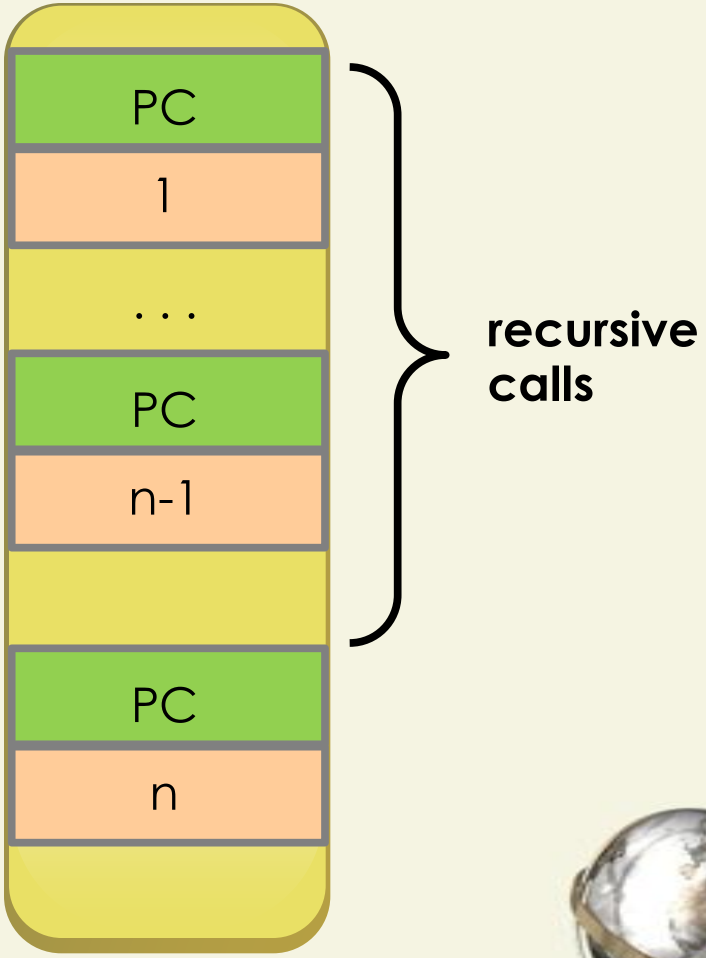
```
mov ebp, esp
cmp [ebp + 4], 1
je factorial_end
cmp [ebp + 4], 0
je factorial_end
```



Recursive Functions

```
factorial:
mov ebp, esp
. . .
mov cx, [ebp+4]
dec cx
sub esp, 2
push cx
call factorial
```

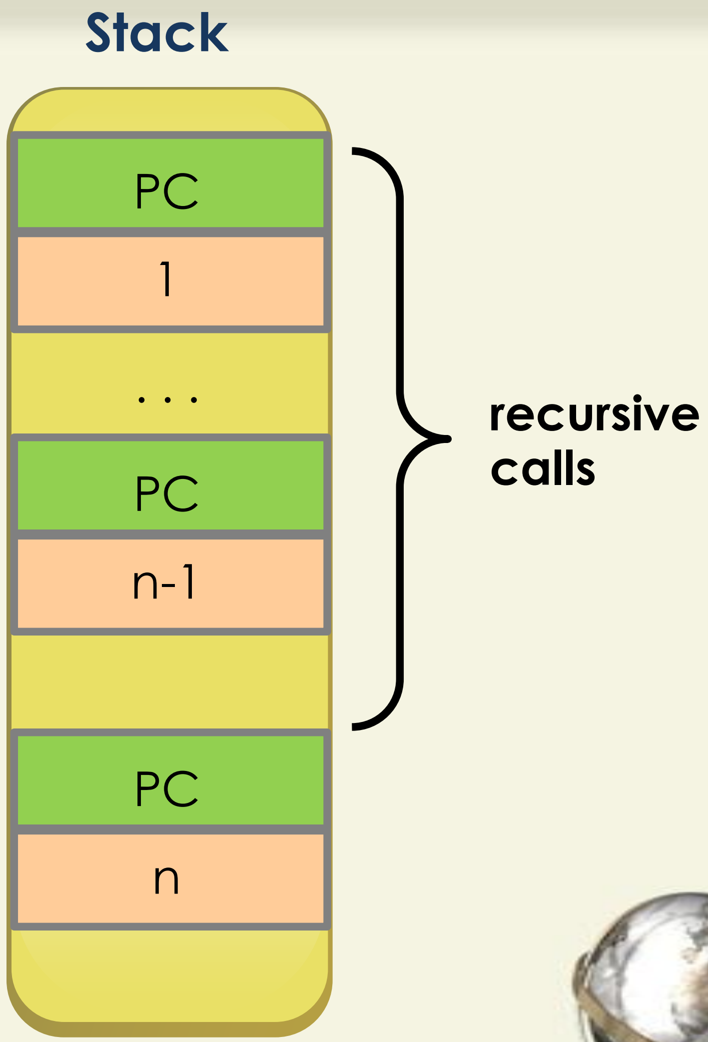
Stack



Recursive Functions

factorial:

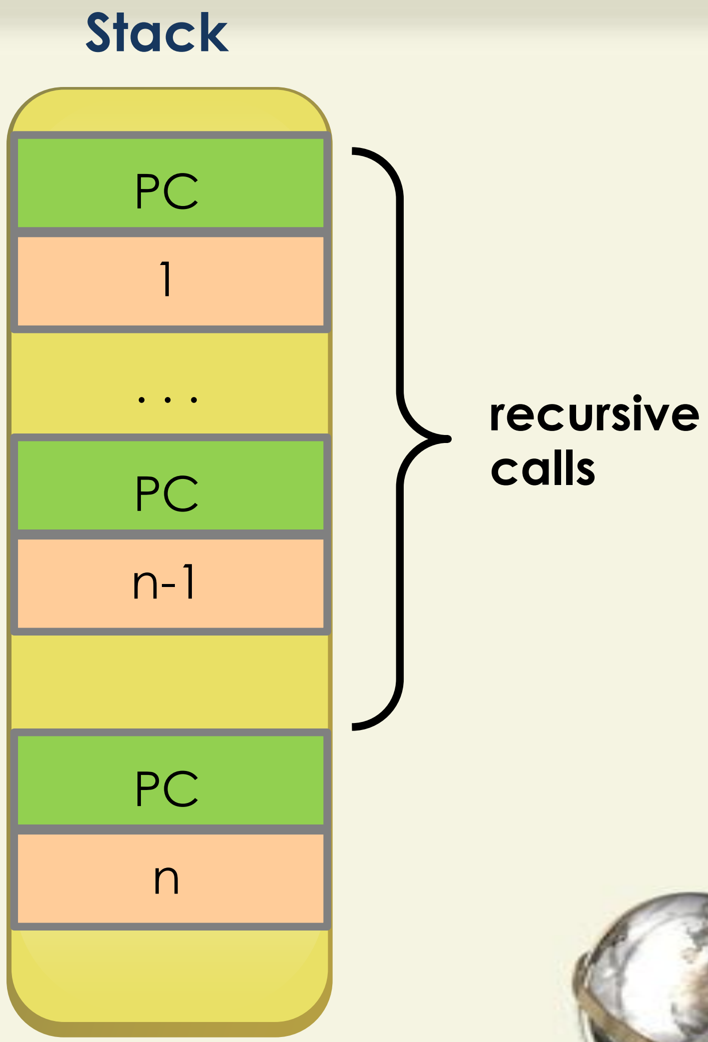
```
mov ebp, esp
cmp [ebp + 4], 1
je factorial_end
cmp [ebp + 4], 0
je factorial_end
```



Recursive Functions

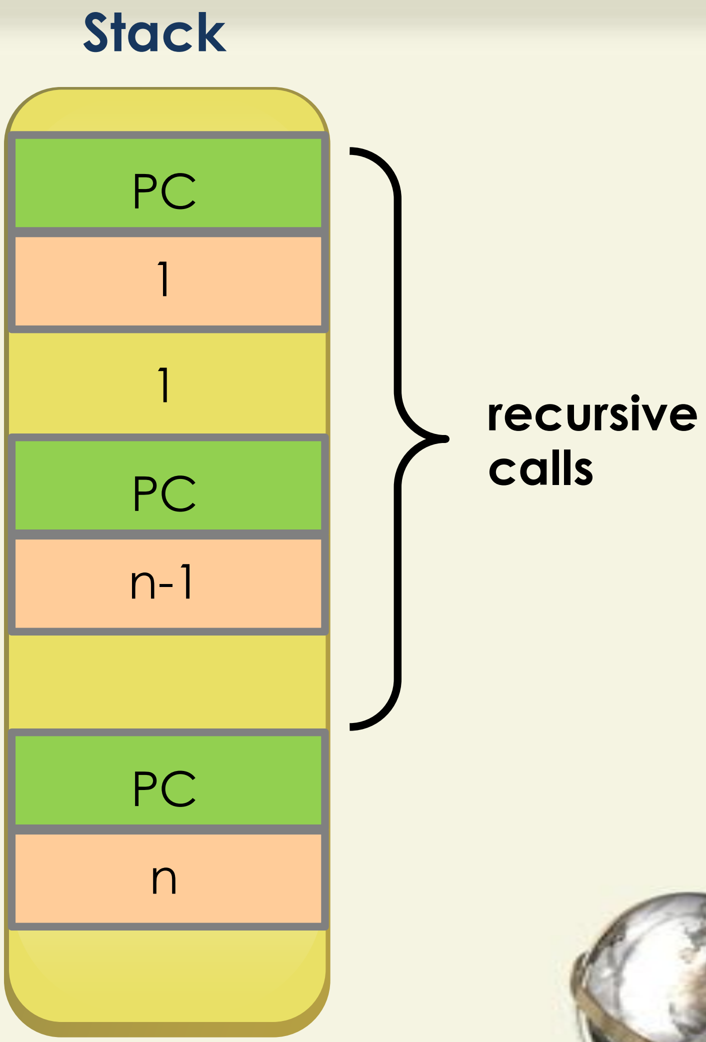
factorial:

```
. . .
mov cx, [ebp+4]
dec cx
sub esp, 2
push cx
call factorial
. . .
factorial_end:
mov word [ebp + 6], 1
ret 2
```



Recursive Functions

```
factorial:
. . .
mov cx, [ebp+4]
dec cx
sub esp, 2
push cx
call factorial
. . .
factorial_end:
mov word [ebp + 6], 1
ret 2
```



Recursive Functions

factorial:

. . .

mov cx, [ebp+4]

dec cx

sub esp, 2

push cx

call factorial

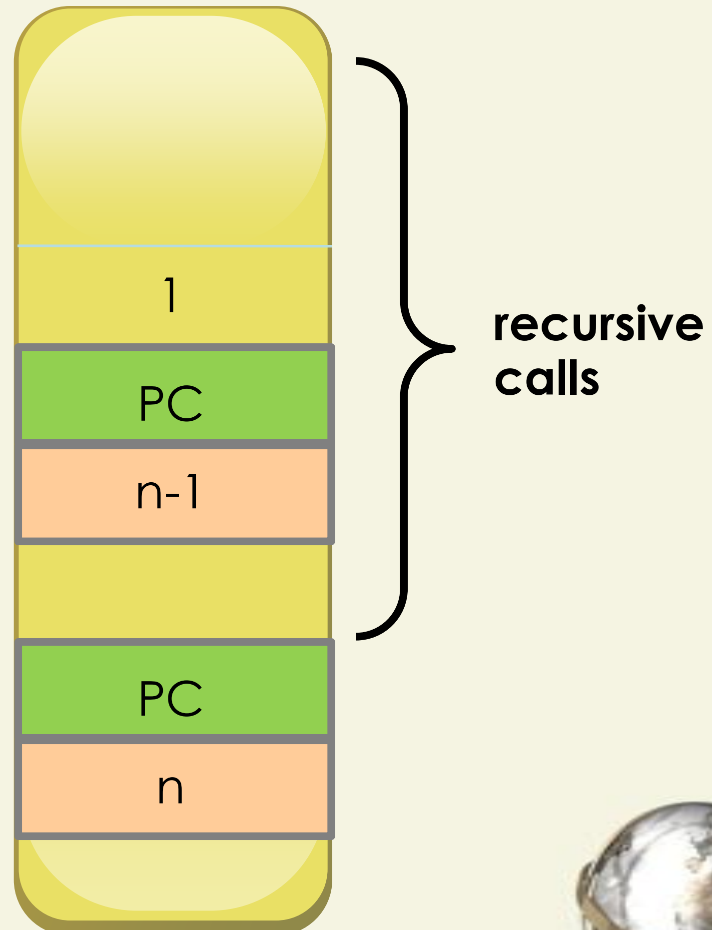
. . .

factorial_end:

mov word [ebp + 6], 1

ret 2

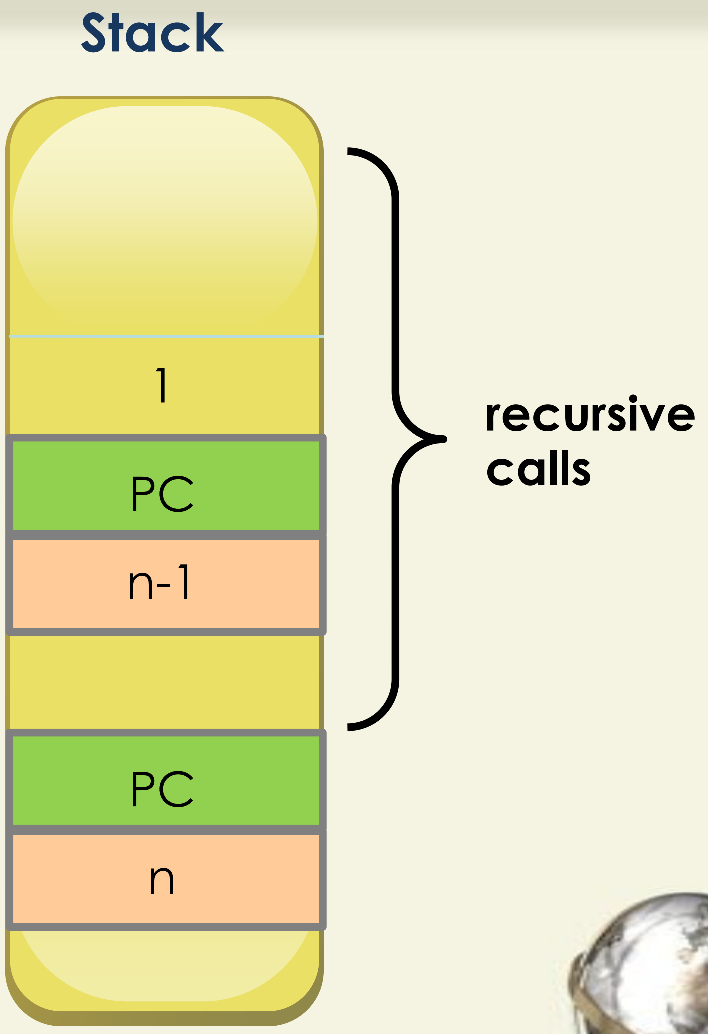
Stack



Recursive Functions

factorial:

```
. . .
mov cx, [ebp+4]
dec cx
sub esp, 2
push cx
call factorial
pop cx
. . .
```

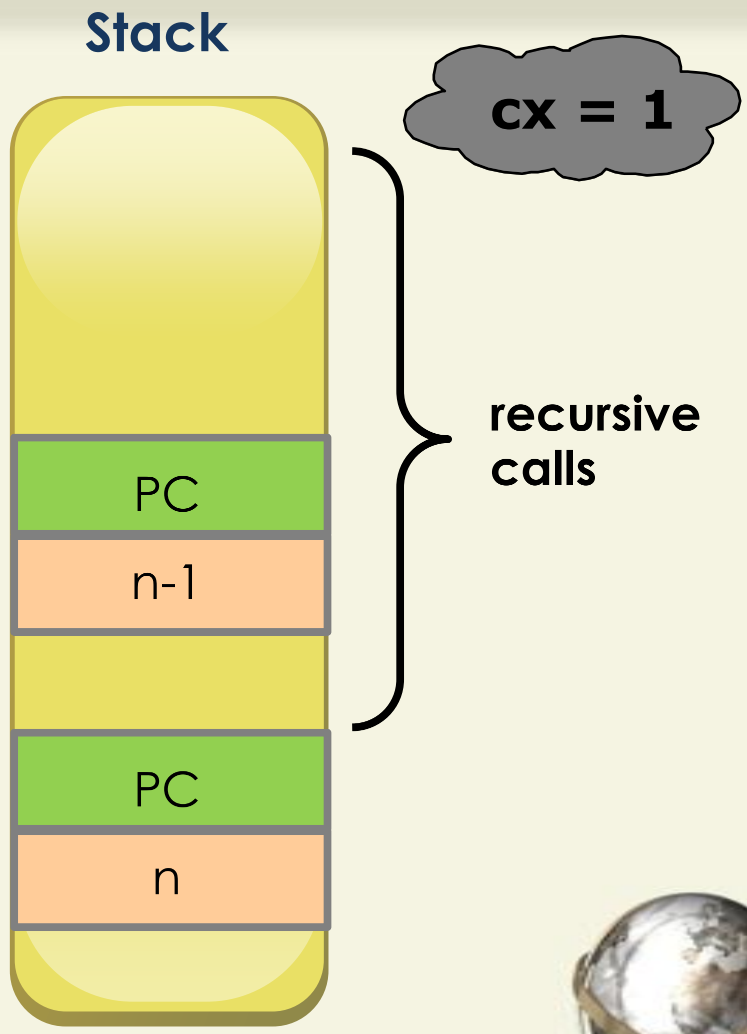


Recursive Functions

factorial:

. . .

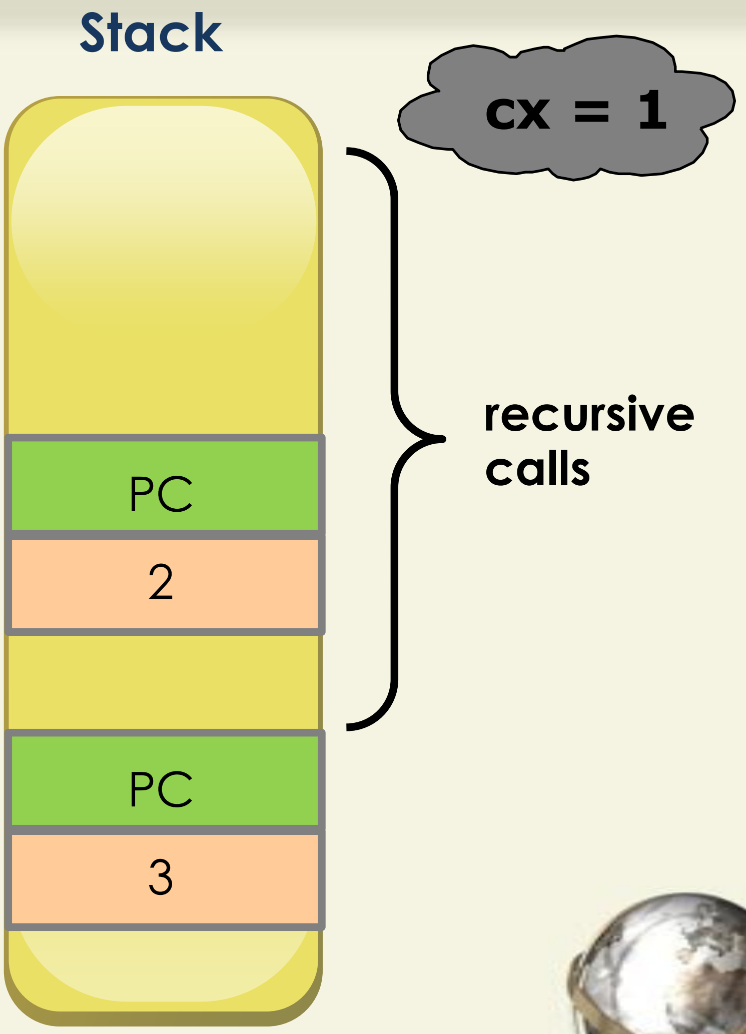
pop cx



Recursive Functions

factorial:

```
. . .
pop cx
mov ebp, esp
mov ax, cx
mul word [ebp + 4]
mov word [ebp + 6], ax
ret 2
. . .
```



Recursive Functions

factorial:

. . .

pop cx

mov ebp, esp

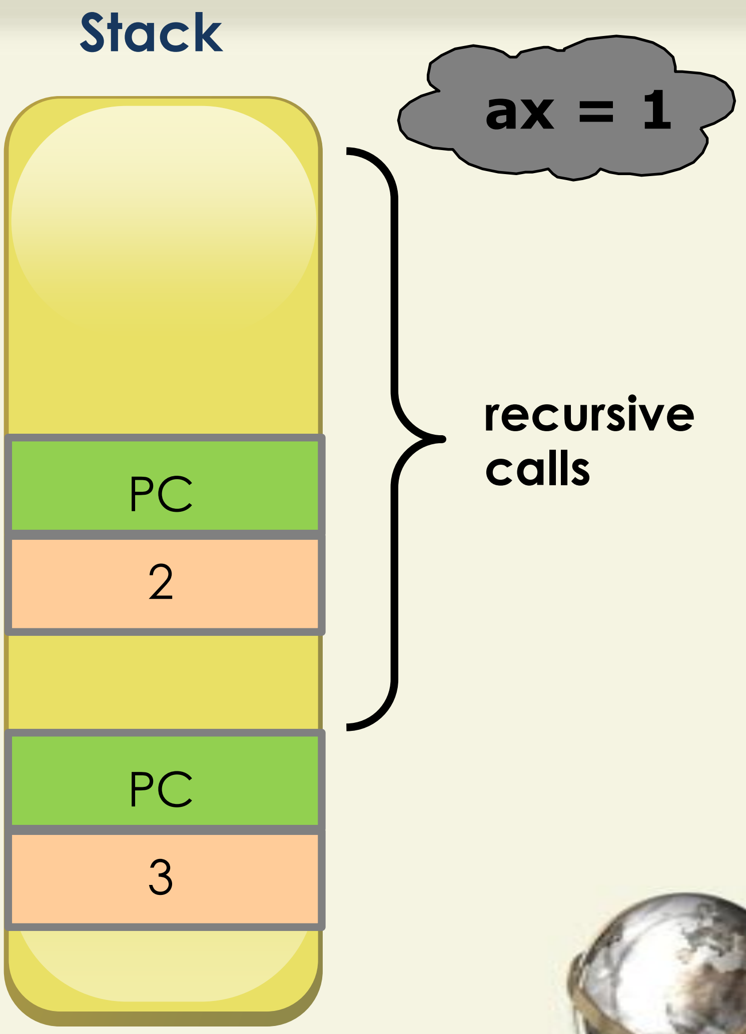
mov ax, cx

mul word [ebp + 4]

mov word [ebp + 6], ax

ret 2

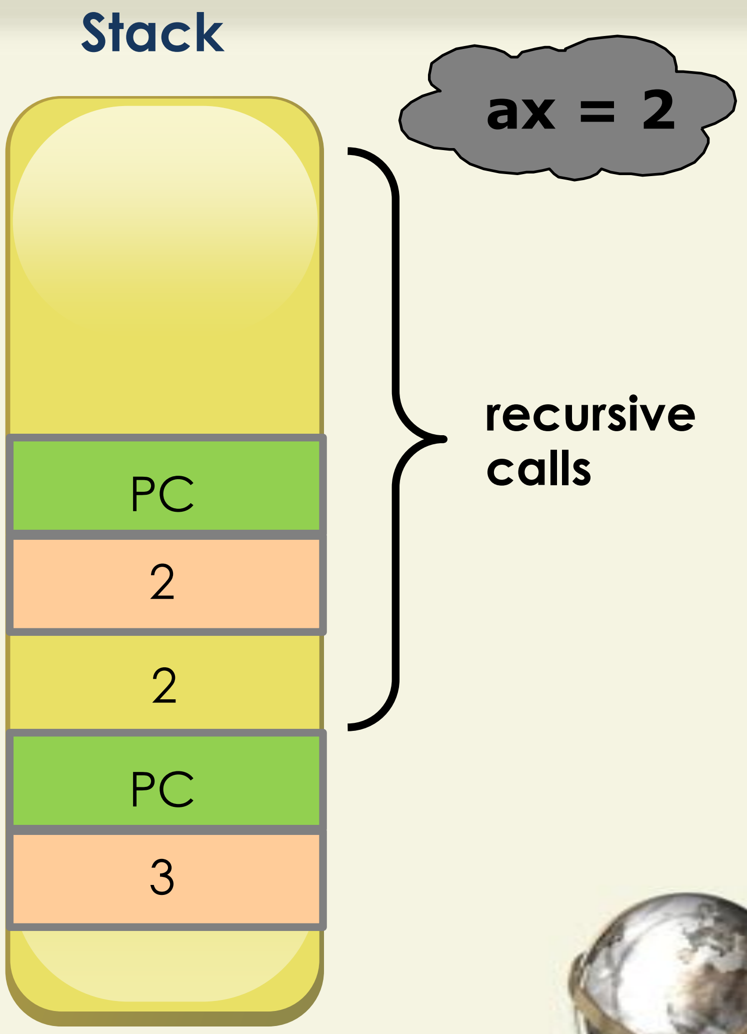
. . .



Recursive Functions

factorial:

```
. . .
pop cx
mov ebp, esp
mov ax, cx
mul word [ebp + 4]
mov word [ebp + 6], ax
ret 2
. . .
```



Recursive Functions

factorial:

. . .

pop cx

mov ebp, esp

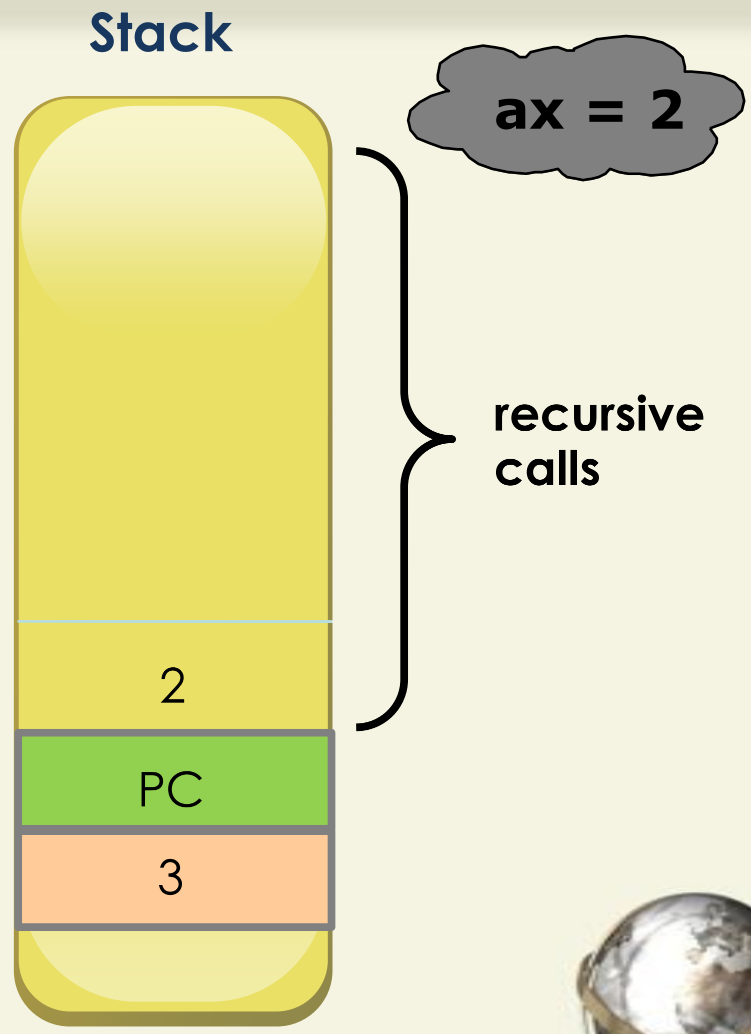
mov ax, cx

mul word [ebp + 4]

mov word [ebp + 6], ax

ret 2

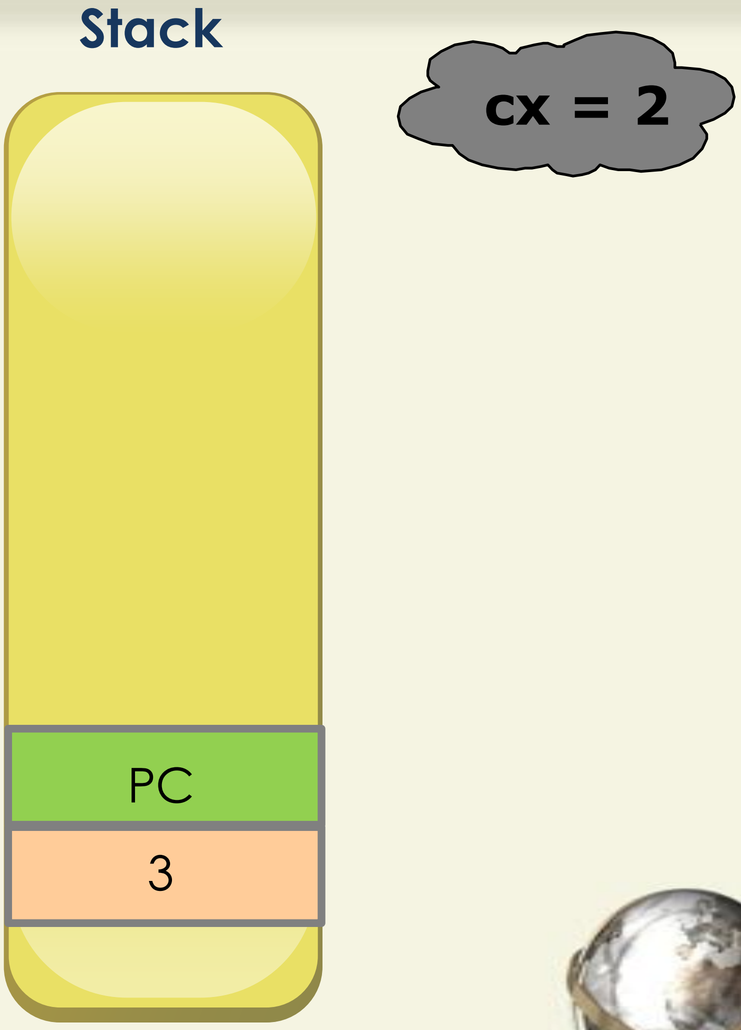
. . .



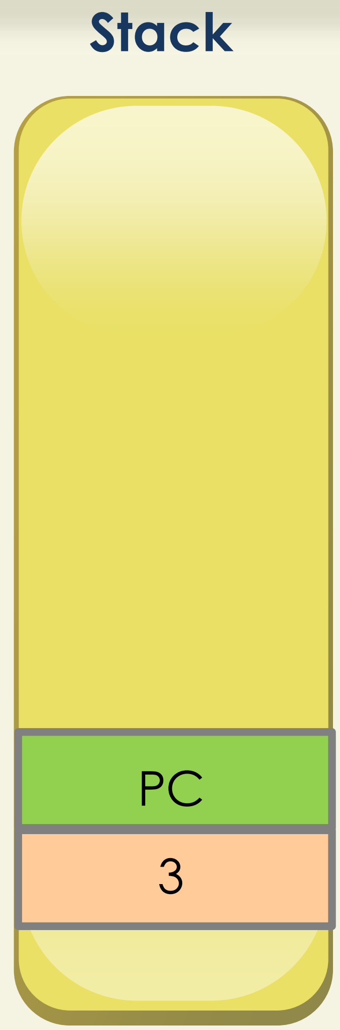
Recursive Functions

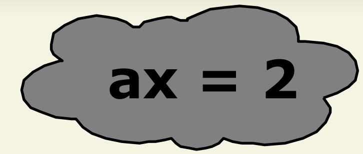
factorial:

```
. . .
pop cx
mov ebp, esp
mov ax, cx
mul word [ebp + 4]
mov word [ebp + 6], ax
ret 2
. . .
```



Recursive Functions





factorial:

. . .

pop cx

mov ebp, esp

mov ax, cx

mul word [ebp + 4]

mov word [ebp + 6], ax

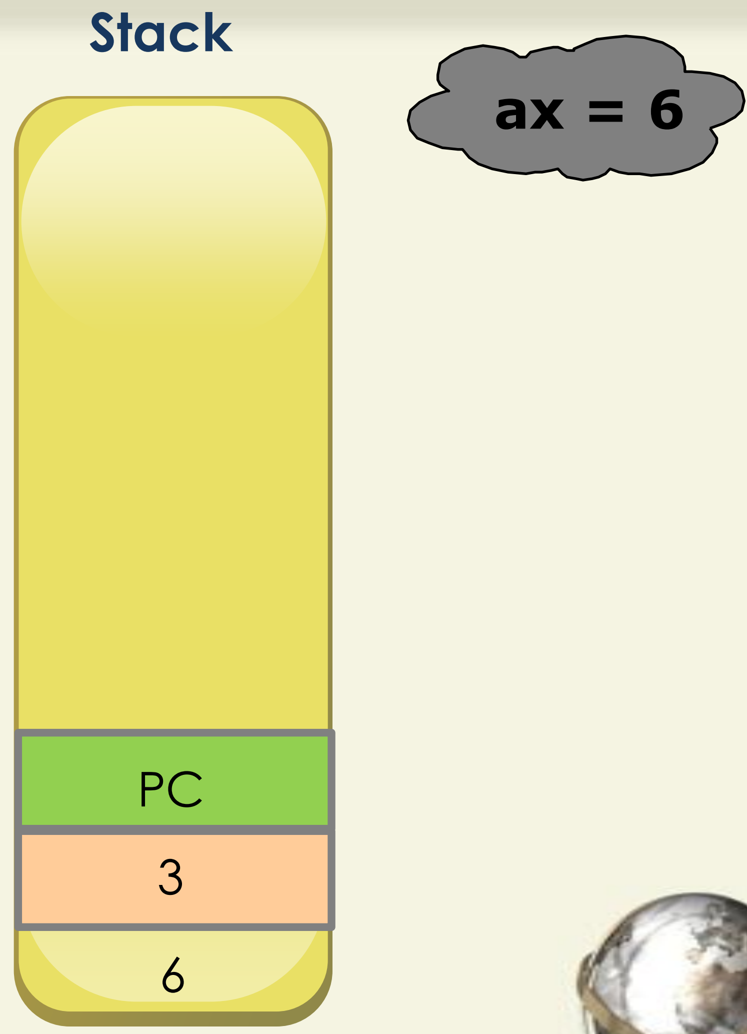
ret 2

. . .

Recursive Functions

factorial:

```
. . .
pop cx
mov ebp, esp
mov ax, cx
mul word [ebp + 4]
mov word [ebp + 6], ax
ret 2
. . .
```



Recursive Functions

```
factorial:
mov ebp, esp
cmp [ebp + 4], 1
je factorial_end
cmp [ebp + 4], 0
je factorial_end
mov cx, [ebp+4]
dec cx
sub esp, 2
push cx
call factorial
pop cx
mov ebp, esp
mov ax, cx
mul word [ebp + 4]
mov word [ebp + 6], ax
ret 2

factorial_end:
    mov word [ebp + 6], 1
    ret 2
```