

Introduction to 8086 Assembly

Lecture 9

Introduction to Subprograms



Indirect addressing

```
segment .data
I1:  dd 111
segment .text
    :
    mov eax, I1
    call print_int
    call print_nl

    mov eax, [I1]
    call print_int
    call print_nl

    mov ecx, I1
    mov eax, [ecx]
    call print_int
    call print_nl
```

indirect.asm



Indirect addressing

```
segment .data
I1:  dd 111
segment .text
    :
    mov eax, I1
    call print_int
    call print_nl

    mov eax, [I1]
    call print_int
    call print_nl

    mov ecx, I1
    mov eax, [ecx]
    call print_int
    call print_nl
```

indirect.asm



Indirect addressing

```
segment .data  
I1:  dd 111  
segment .text  
:  
    mov eax, I1  
    call print_int  
    call print_nl  
  
    mov eax, [I1]  
    call print_int  
    call print_nl  
  
    mov ecx, I1  
    mov eax, [ecx]  
    call print_int  
    call print_nl
```

indirect.asm



Indirect addressing

```
segment .data
```

```
I1: dd 111  
    dd 222  
    dd 444
```

```
segment .text
```

```
:
```

```
    mov ecx, I1
```

```
    mov eax, [ecx]  
    call print_int  
    call print_nl
```

indirect2.asm

indirect2.asm (cont.)

```
    mov eax, [ecx+1]  
    call print_int  
    call print_nl
```

```
    mov eax, [ecx+4]  
    call print_int  
    call print_nl
```

```
    mov eax, [ecx+8]  
    call print_int  
    call print_nl
```



Indirect addressing

```
segment .data
```

```
I1: dd 111  
    dd 222  
    dd 444
```

```
segment .text
```

```
:
```

```
    mov ecx, I1
```

```
    mov eax, [ecx]
```

```
    call print_int
```

```
    call print_nl
```

indirect2.asm

indirect2.asm (cont.)

```
    mov eax, [ecx+1]
```

```
    call print_int
```

```
    call print_nl
```

```
    mov eax, [ecx+4]
```

```
    call print_int
```

```
    call print_nl
```

```
    mov eax, [ecx+8]
```

```
    call print_int
```

```
    call print_nl
```

How does the assembler do this?



mov eax, [ecx]

mov ax, [ecx]

mov al, [ecx]



How to implement subprograms?

- Subprogram
- function
- subroutine
- procedure
- routine
- method
- callable

```
void print_salam(void);  
  
int main() {  
  
    print_salam();  
  
}  
  
void print_salam() {  
    printf("Salaaaaam!\n");  
}
```



How to implement subprograms?

```
void print_salam(void);

int main() {

    print_salam();

}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

```
segment .data
msg: db "Salaaaaam!", 10, 0

segment .text
:
:
print_salam:
    mov eax, msg
    call print_string
```



How to implement subprograms?

```
void print_salam(void);

int main() {

    print_salam();

}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

```
segment .data
msg: db "Salaaaaam!", 10, 0

segment .text
:
jmp print_salam

:
print_salam:
    mov eax, msg
    call print_string
```



How to implement subprograms?

```
void print_salam(void);

int main() {

    print_salam();

}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

```
segment .data
msg: db "Salaaaaam!", 10, 0

segment .text
:
jmp print_salam
I1:
:

print_salam:
    mov eax, msg
    call print_string
```



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

```
segment .data
msg: db "Salaaaaam!", 10, 0

segment .text
:
jmp print_salam
I1:   :
print_salam:
    mov eax, msg
    call print_string
```

A red arrow points from the label 'I1:' to the word 'return address'.



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

simplefunc1.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0

segment .text
:
jmp print_salam
I1:   :
print_salam:
    mov eax, msg
    call print_string

    jmp I1
```

A red arrow points from the label 'I1:' to the text 'return address'.



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

What's wrong?

segment .data simplefunc1.asm

msg: db "Salaaaaam!", 10, 0

segment .text

:

jmp print_salam

I1:

:

return address

print_salam:

mov eax, msg

call print_string

jmp I1



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

simplefunc2.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0

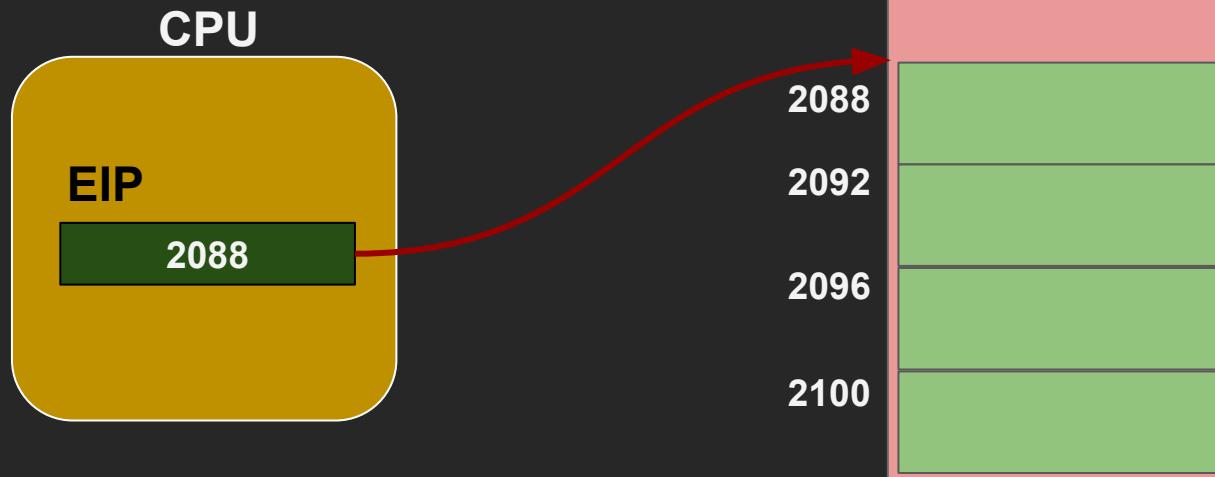
segment .text
:
jmp print_salam
I1: :
jmp print_salam
I2: :
print_salam:
    mov eax, msg
    call print_string
    jmp ?
```

A red arrow points from the label I1: to the text "return address". Another red arrow points from the label I2: to the start of the "print_salam" function.



Looking closer at the jmp command

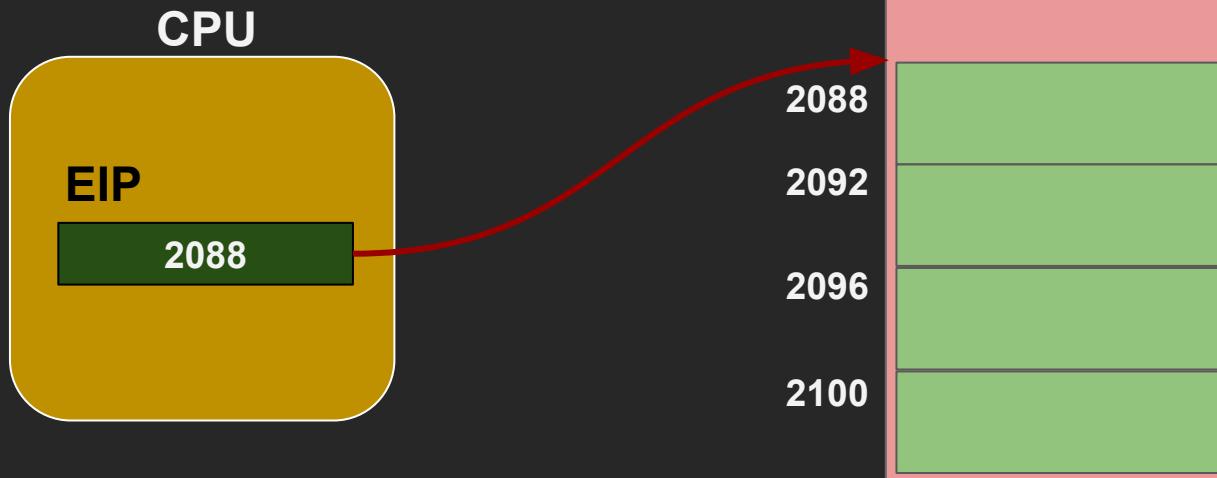
The **instruction pointer**
(program counter) IP, EIP, RIP





Looking closer at the jmp command

jmp labell

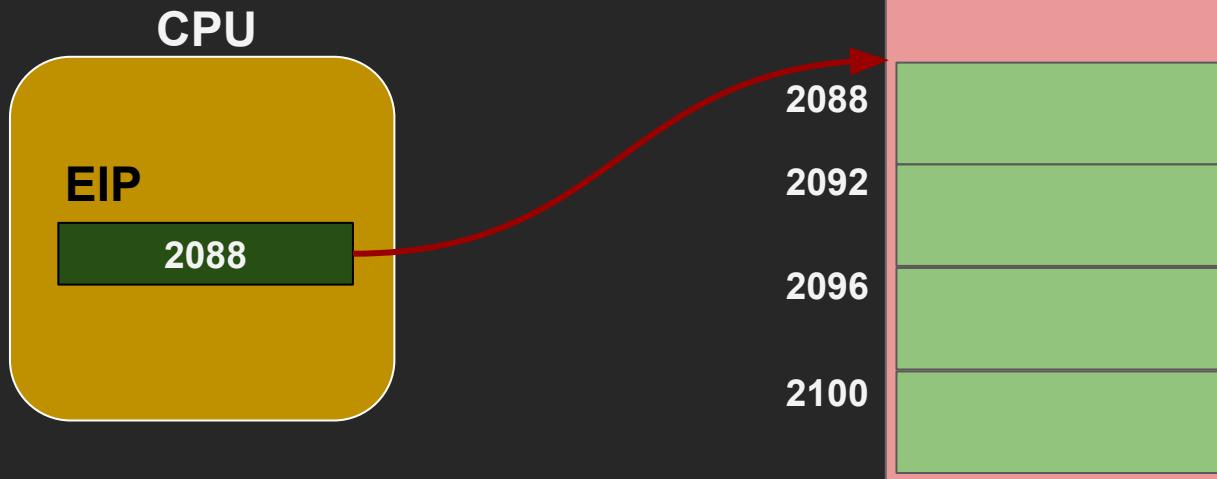




Looking closer at the jmp command

jmp labell

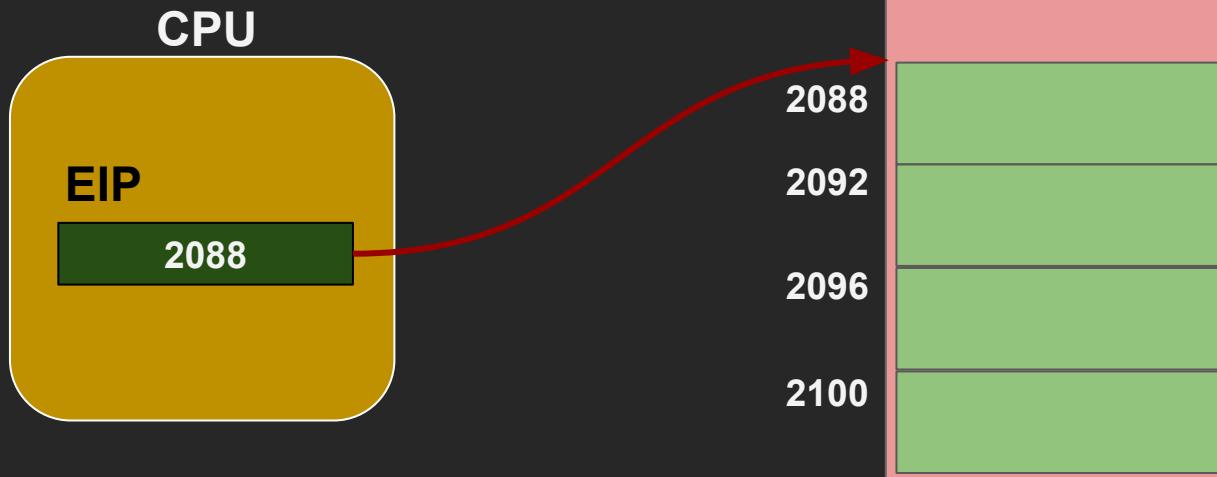
How are mov and jmp similar?





Looking closer at the jmp command

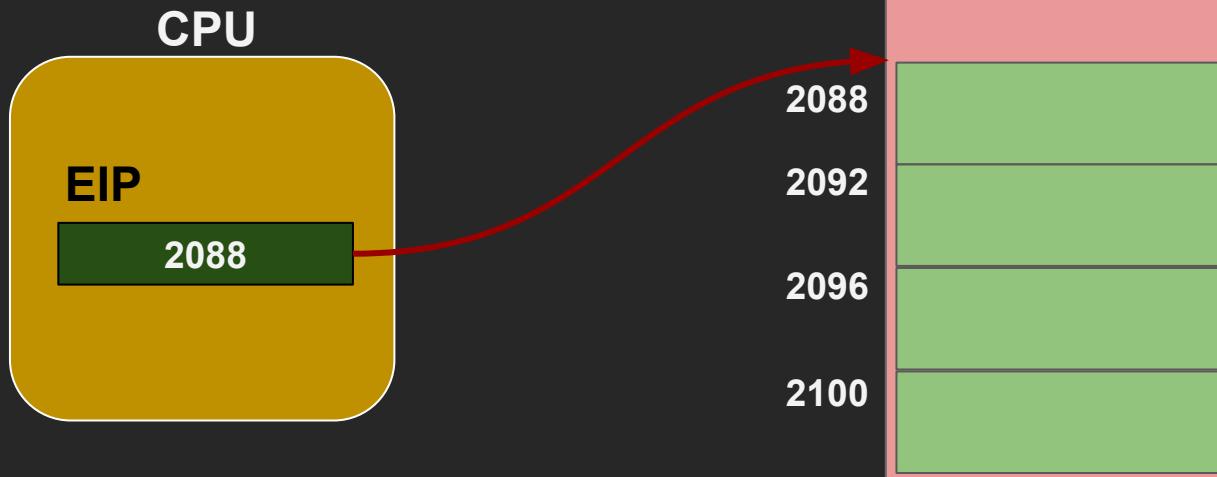
jmp label1
(mov EIP, label1)





Looking closer at the jmp command

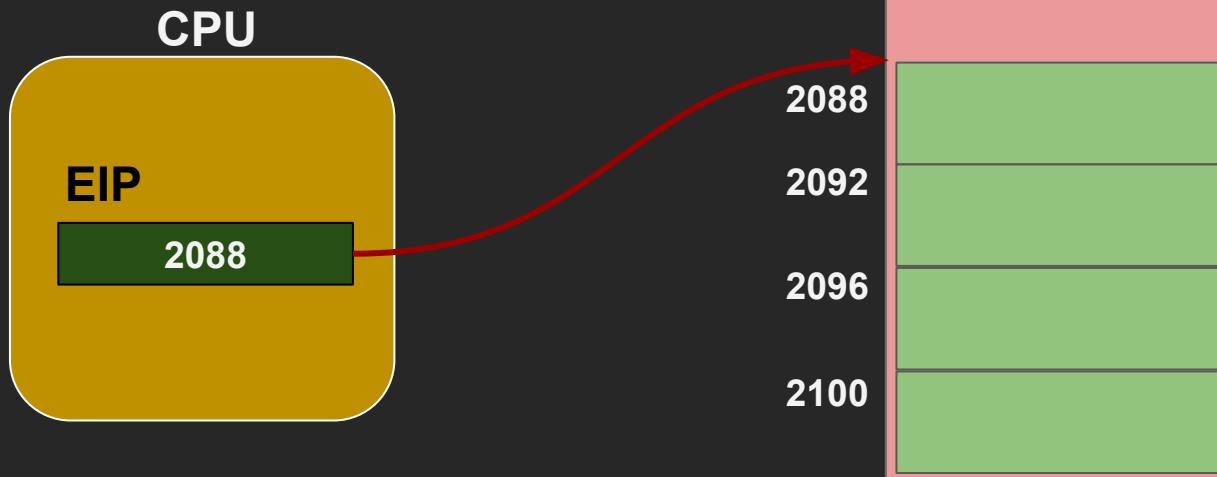
```
mov EAX, label1  
(mov EIP, EAX)
```





Looking closer at the jmp command

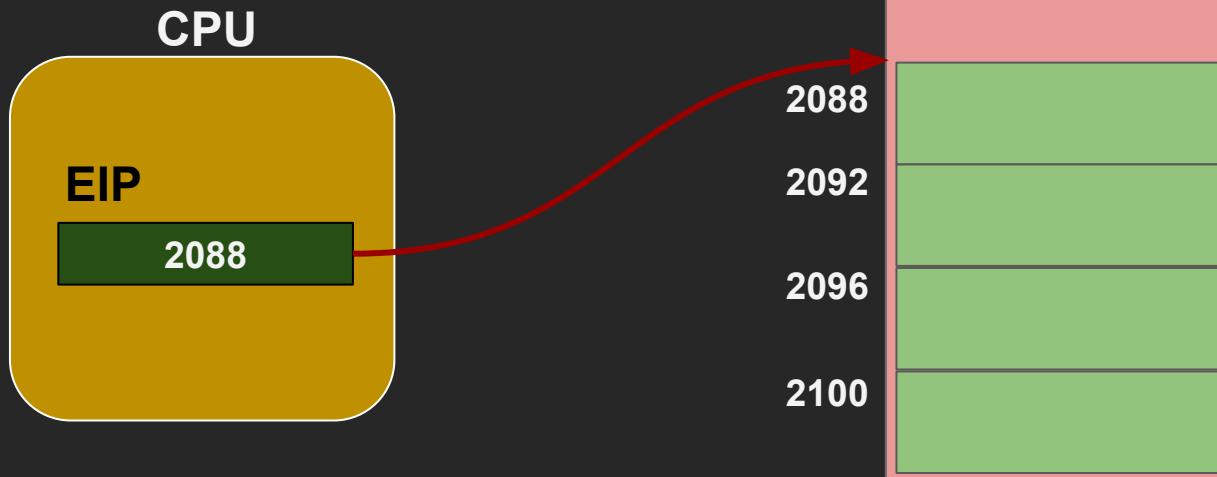
```
mov EAX, label1  
(mov EIP, EAX)
```





Looking closer at the jmp command

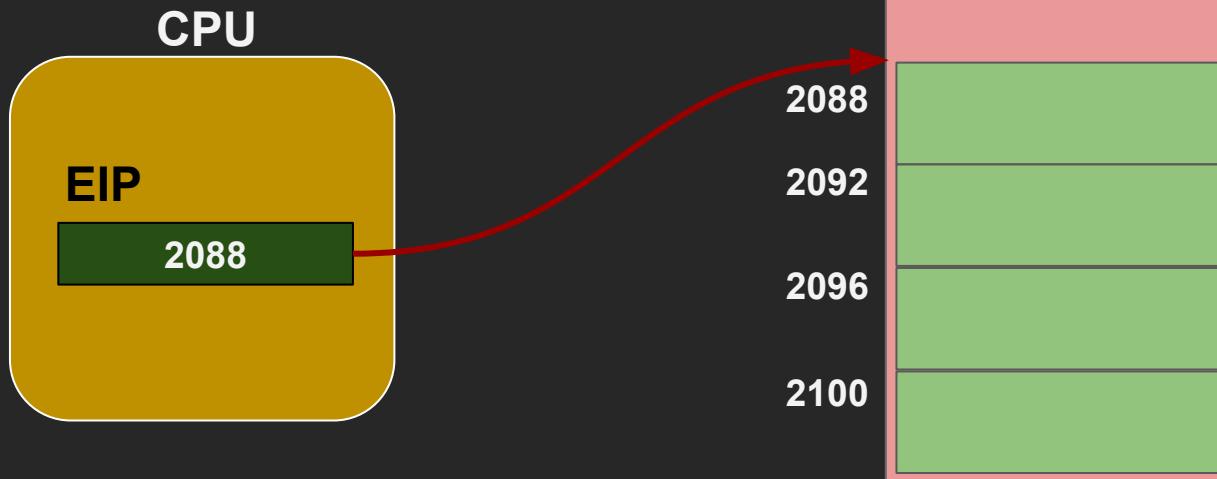
```
mov EAX, label1  
jmp EAX      (mov EIP, EAX)
```





Looking closer at the jmp command

```
(mov EIP, [l1])  
jmp [l1]
```





Indirect jump

Direct Jump: **jmp l1**

Indirect Jump: **mov eax, l1**
 jmp eax



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

simplefunc3.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
jmp print_salam
I1: → return address
jmp print_salam
I2: →
:
print_salam:
    mov eax, msg
    call print_string
    jmp ?
```



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

simplefunc3.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
jmp print_salam
I1: → return address
jmp print_salam
I2: →
:
print_salam:
    mov eax, msg
    call print_string
    jmp edx
```

The assembly code shows the implementation of the `print_salam` function. It starts with a data section containing a string "Salaaaaam!" followed by a null terminator and a carriage return. The text section begins with a jump instruction to the `print_salam` label. Two instances of `jmp print_salam` are shown, with red arrows pointing from them to the label, indicating they are part of the same function. The `print_salam` label is followed by a block of assembly instructions: `mov eax, msg`, `call print_string`, and `jmp edx`.



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

simplefunc3.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
mov edx, I1
jmp print_salam
I1:
mov edx, I2
jmp print_salam
I2:
:
print_salam:
    mov eax, msg
    call print_string
    jmp edx
```

return address

```
graph TD; I1((I1)) --> J1[jmp print_salam]; I2((I2)) --> J2[jmp print_salam]; PS[print_salam] --> J3[jmp edx]; RA((return address)) --> J1;
```



How to implement subprograms?

```
void print_salam(void);

int main() {
    print_salam();
}

void print_salam() {
    printf("Salaaaaam!\n");
}
```

Limitations?

simplefunc3.asm

```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
mov edx, I1
jmp print_salam
I1:
mov edx, I2
jmp print_salam
I2:
:
print_salam:
    mov eax, msg
    call print_string
    jmp edx
```

return address

```
graph TD; J1[jmp print_salam] --> I1[I1:]; J2[jmp print_salam] --> I2[I2:]; J3[jmp edx] --> PS[print_salam:]; RA[return address] --> J3;
```



K. N. Toosi
University of Technology

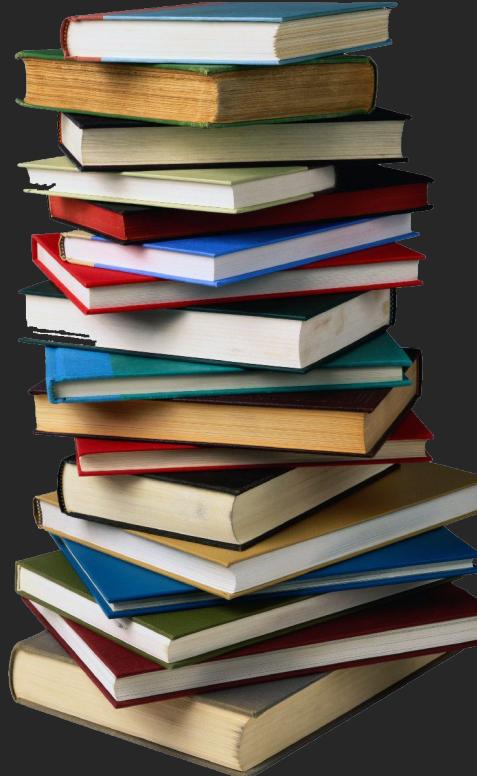
The stack



<http://freepngimg.com/png/25783-coin-stack-transparent-image>



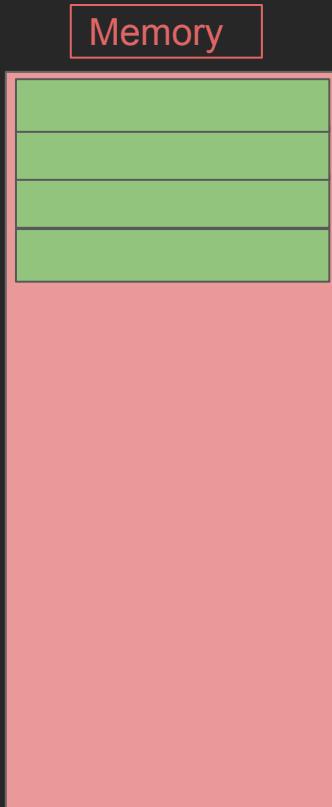
<https://pixabay.com/en/plate-stack-tableware-plate-stack-629970/>



<http://carbon.materialwitness.co/book-stack/>



The stack



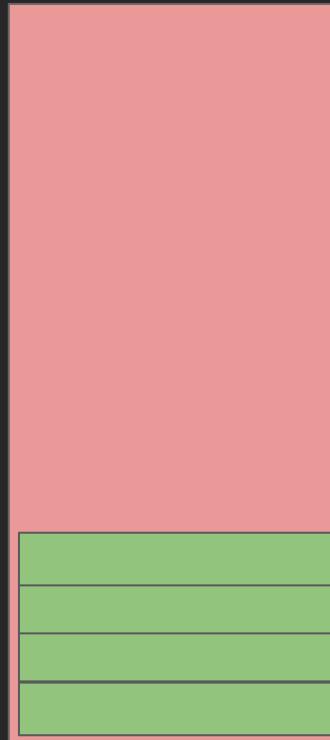


Implementing the stack

Stack Segment



Stack Segment



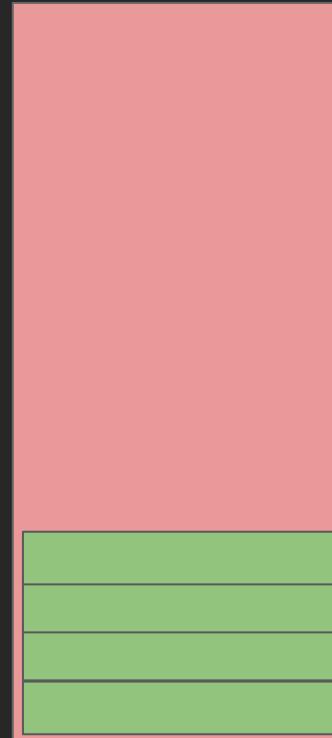


Implementing the stack

Stack Segment



Stack Segment



x86

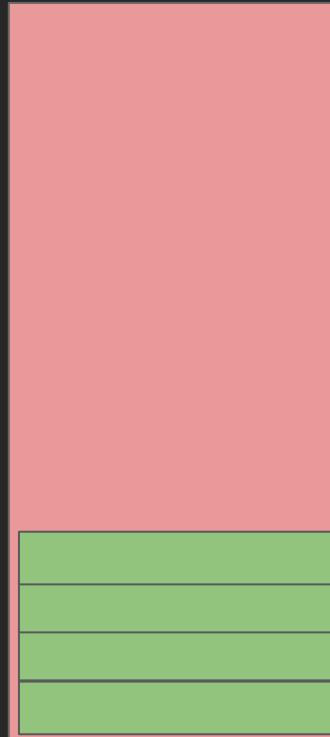


Implementing the stack

Stack Segment



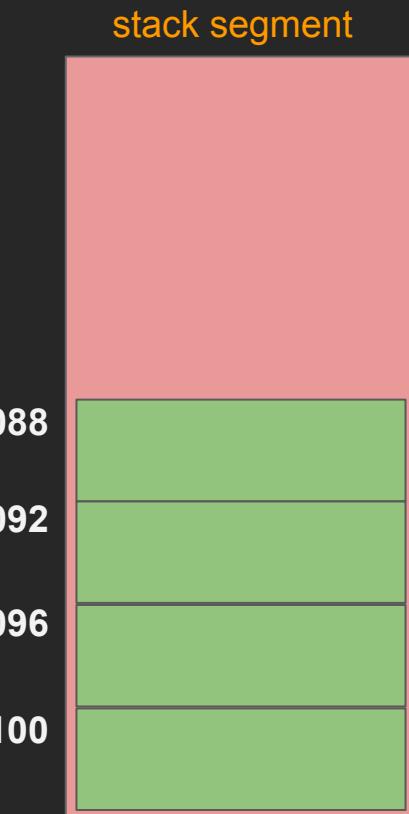
Stack Segment



x86
(why?)

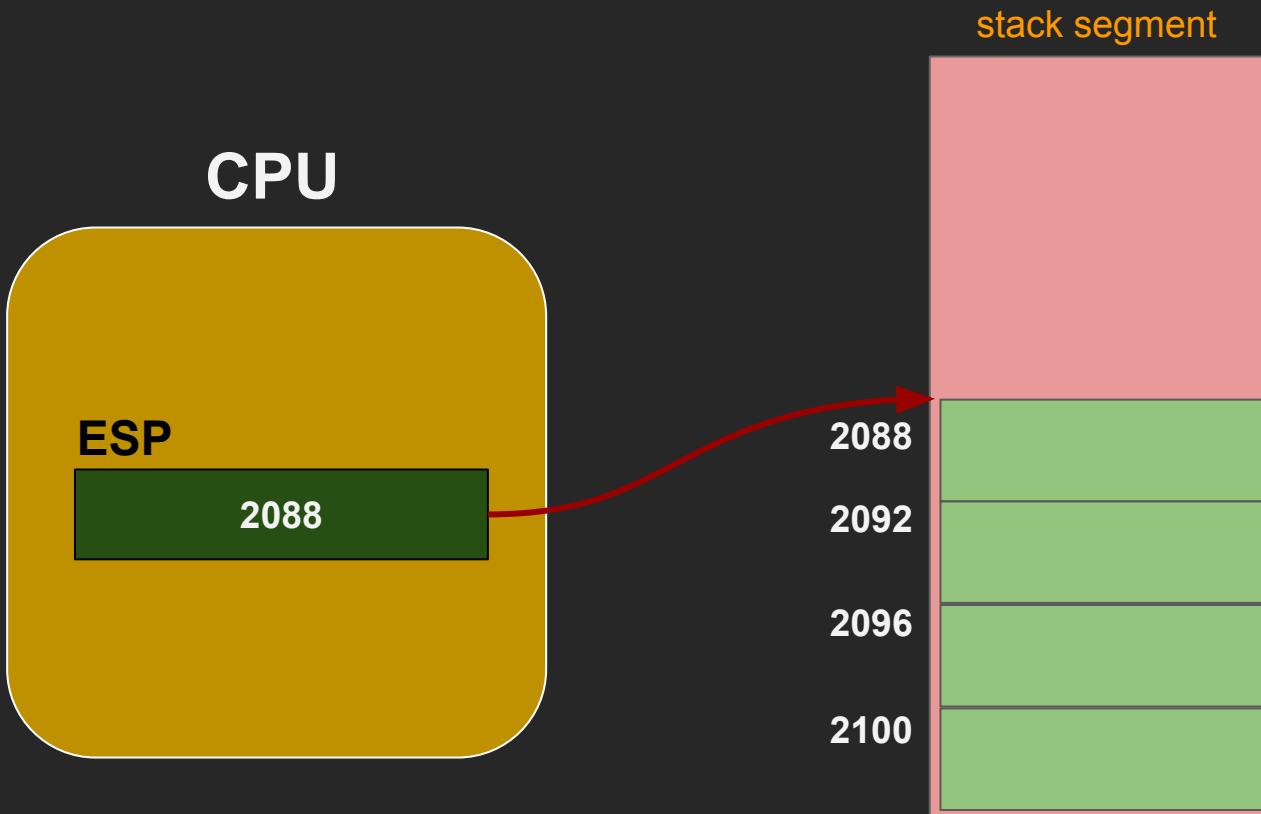


Implementing the stack



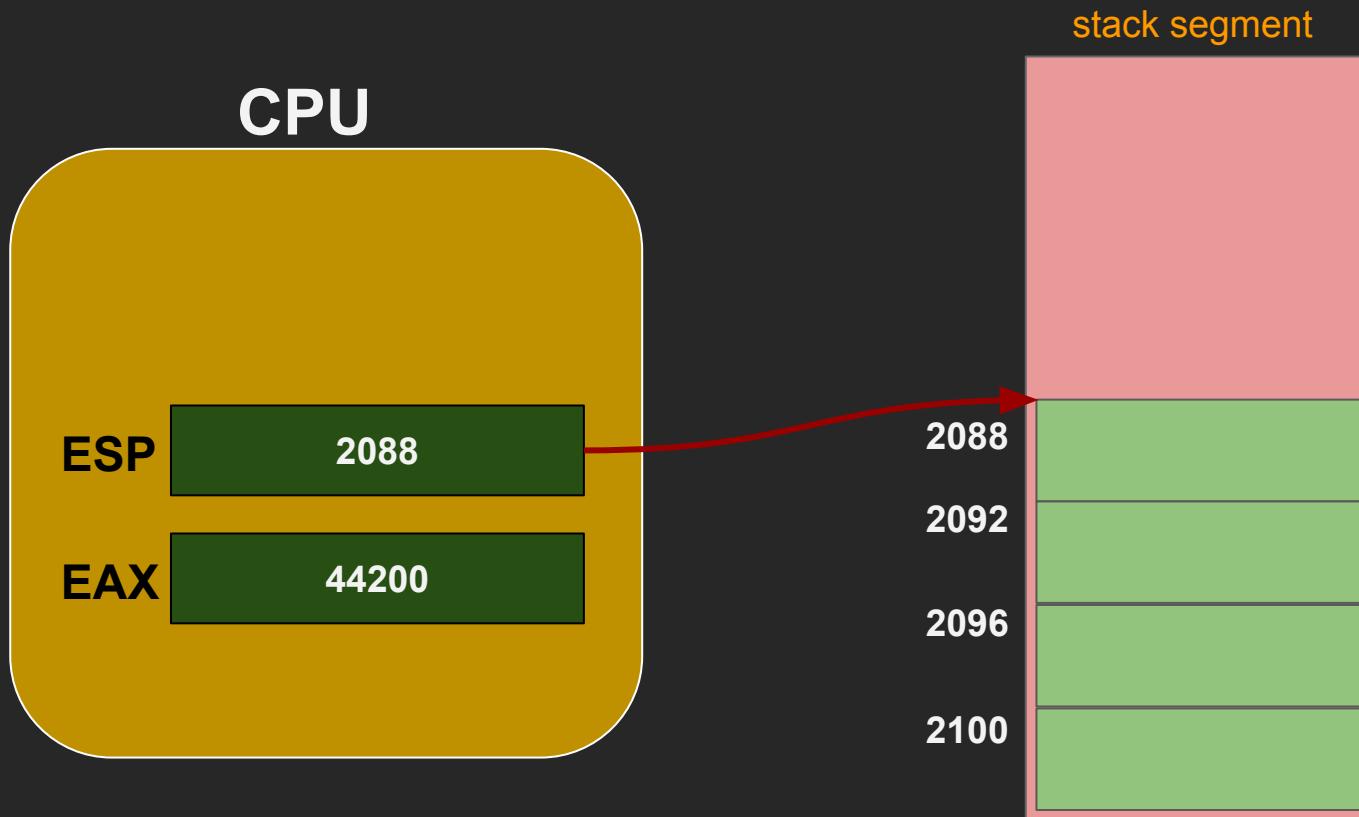


Stack Pointer (SP, ESP, RSP)



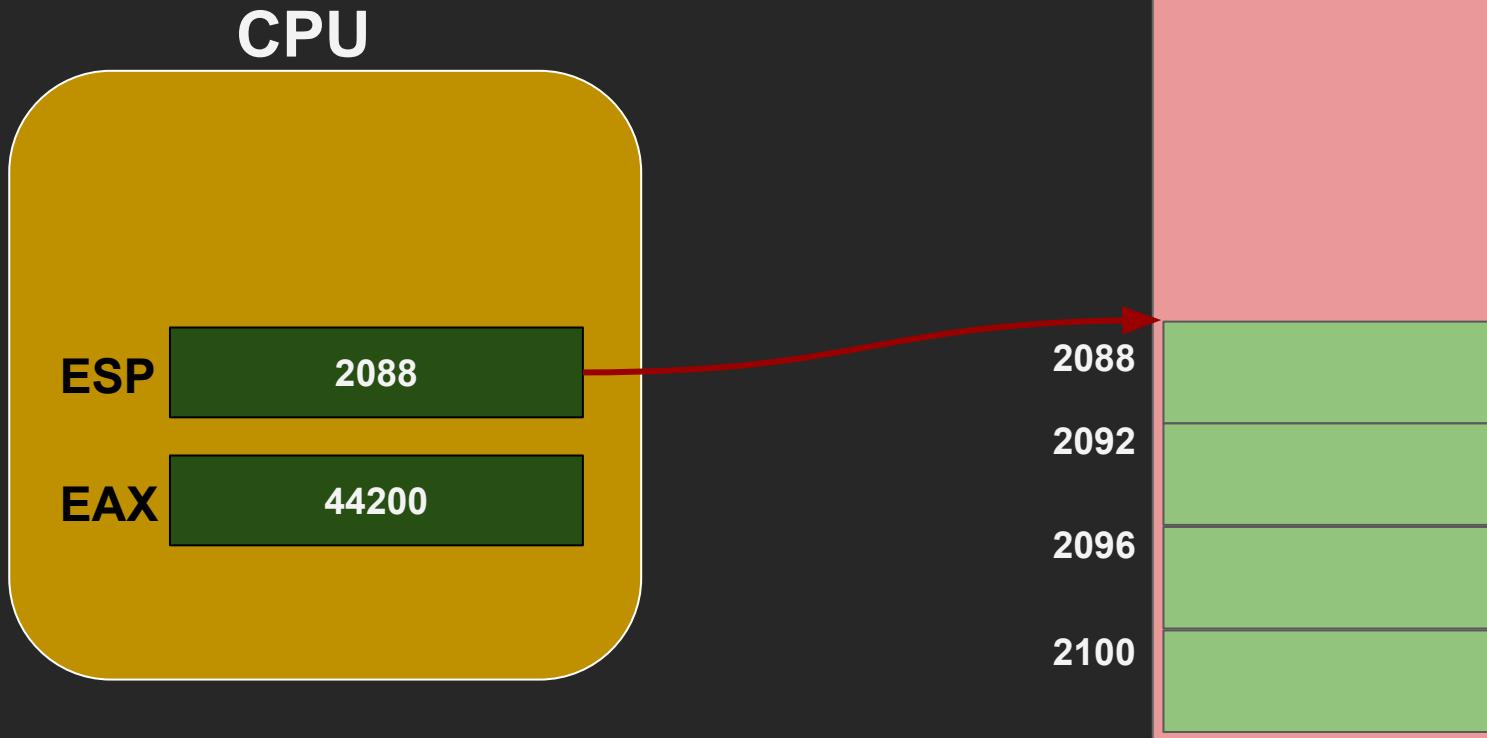


Pushing on the stack



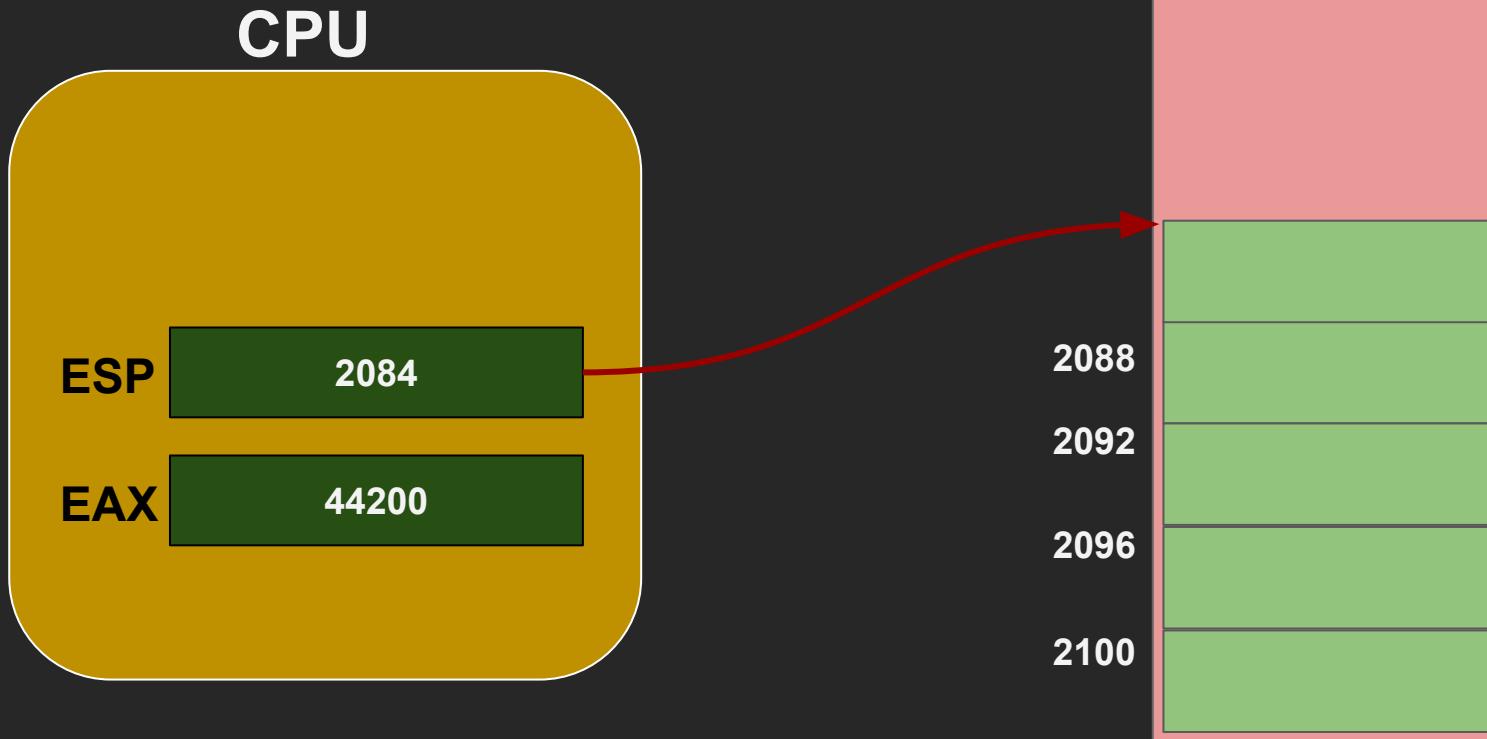


Push EAX on the stack



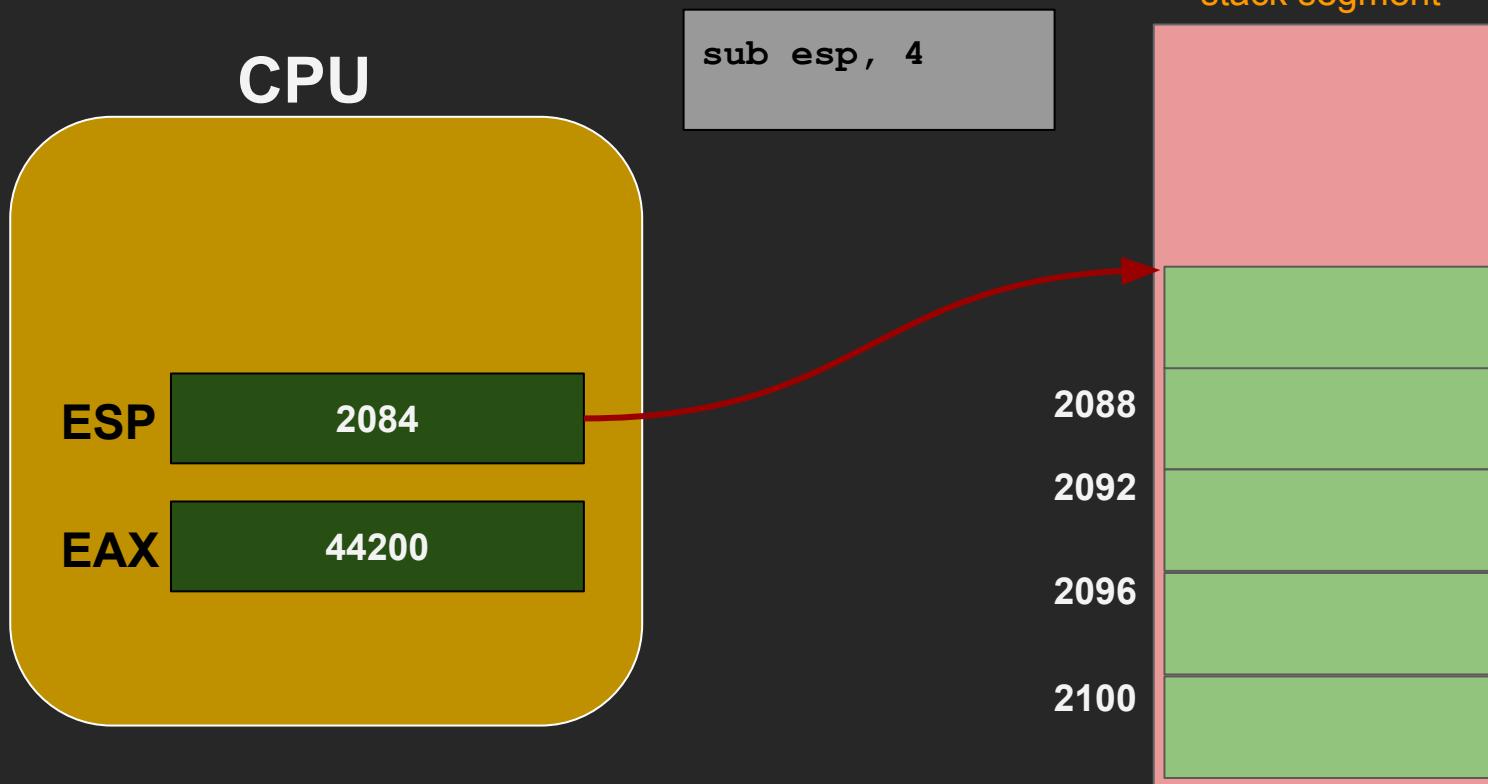


Push EAX on the stack



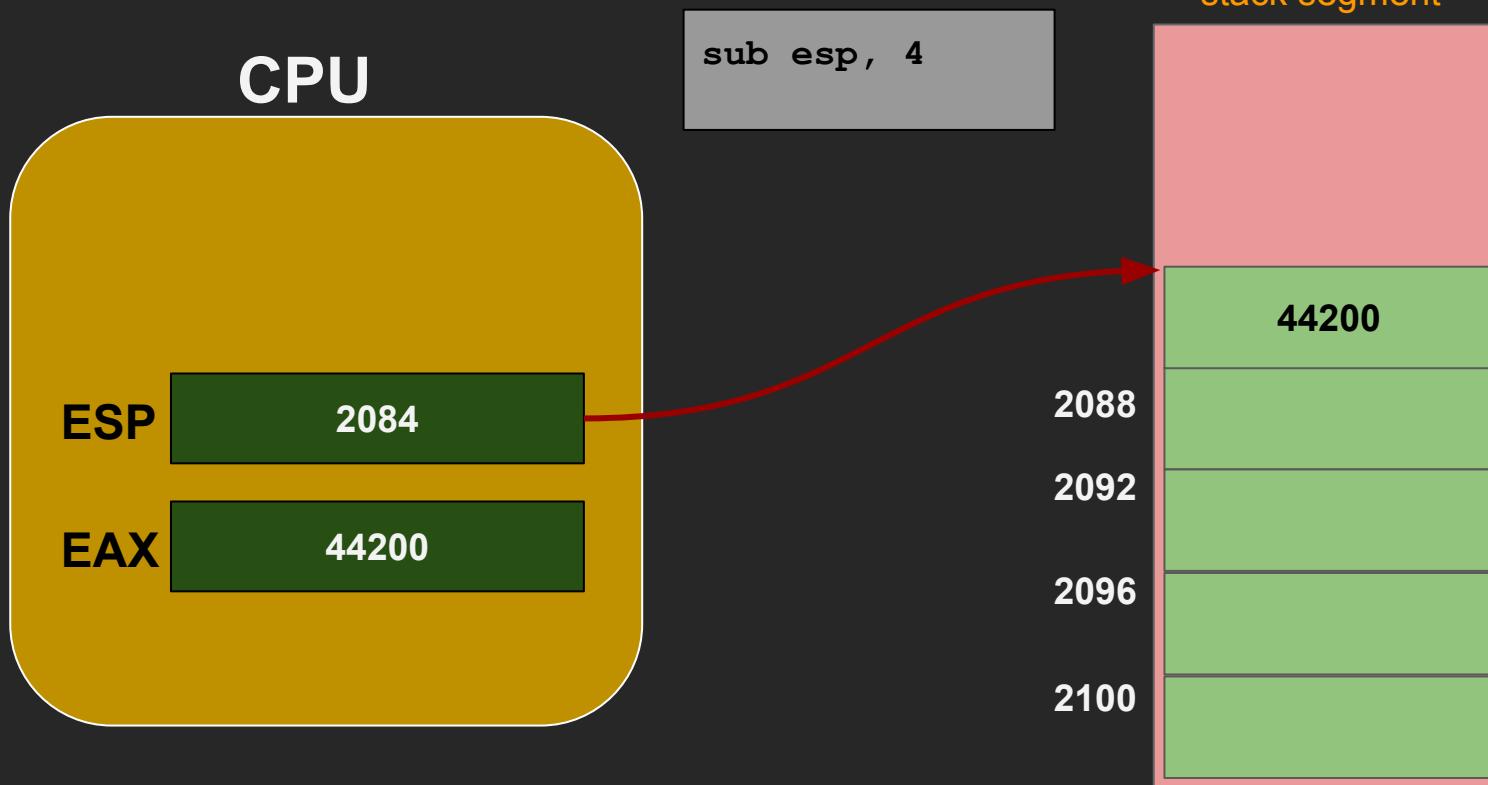


Push EAX on the stack



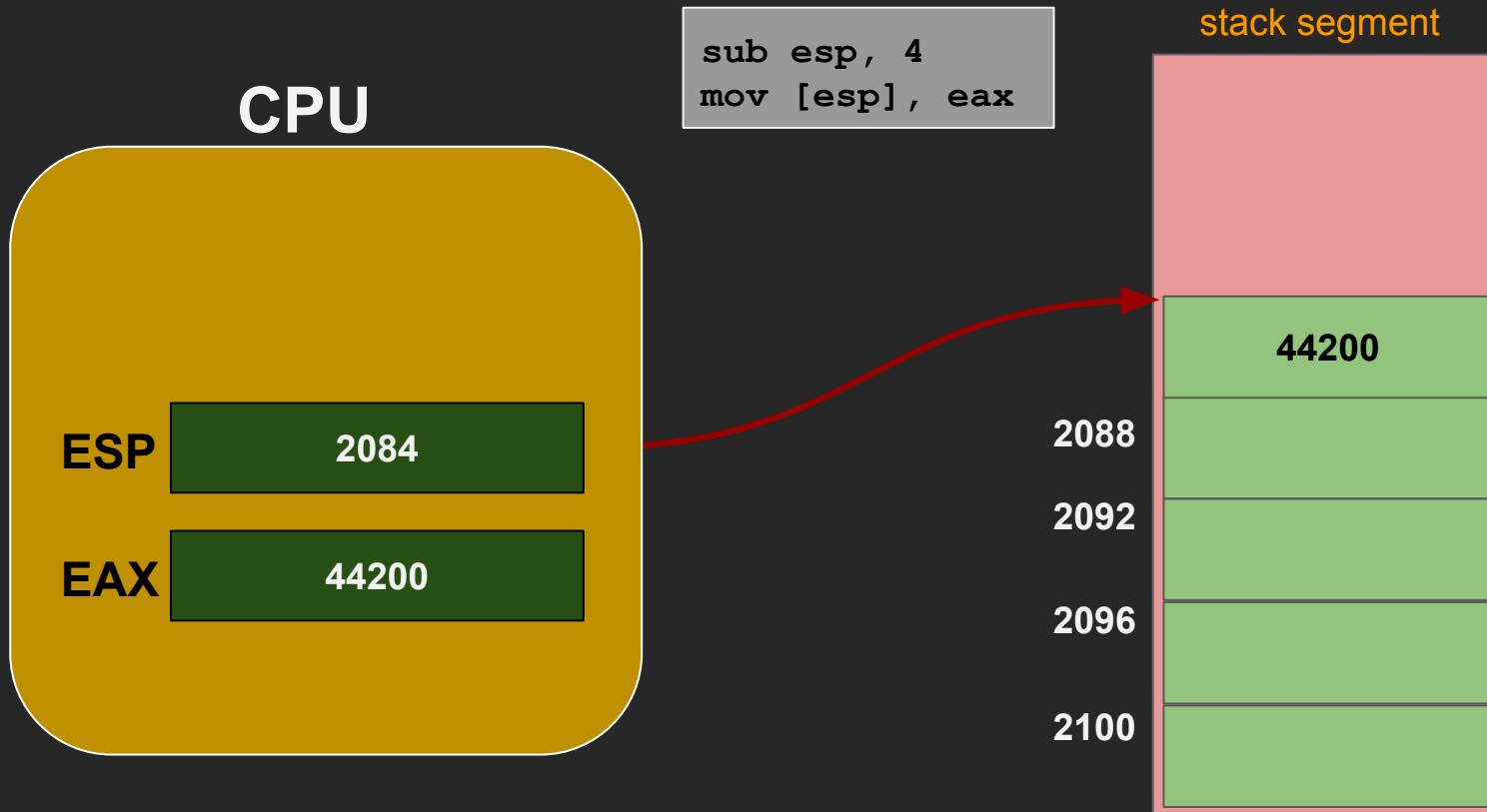


Push EAX on the stack



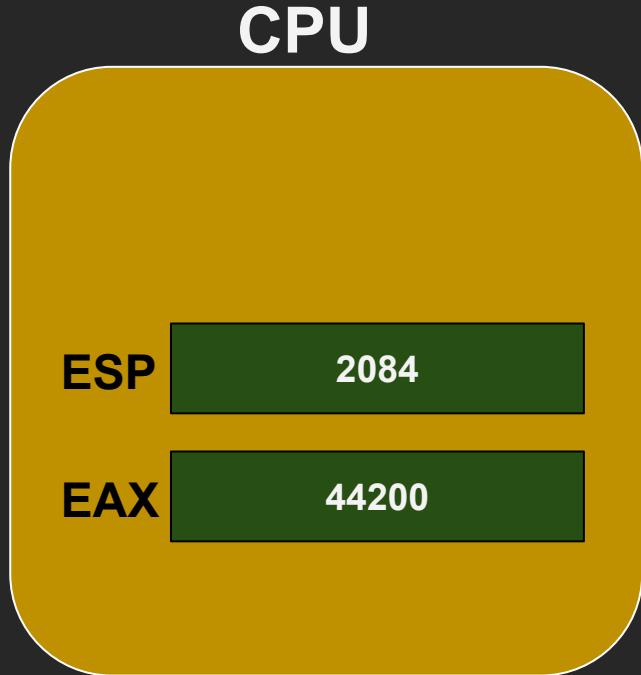


Push EAX on the stack

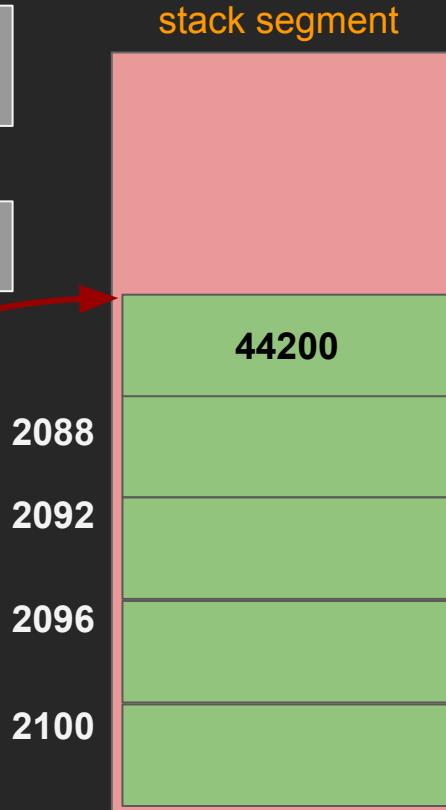




Push EAX on the stack

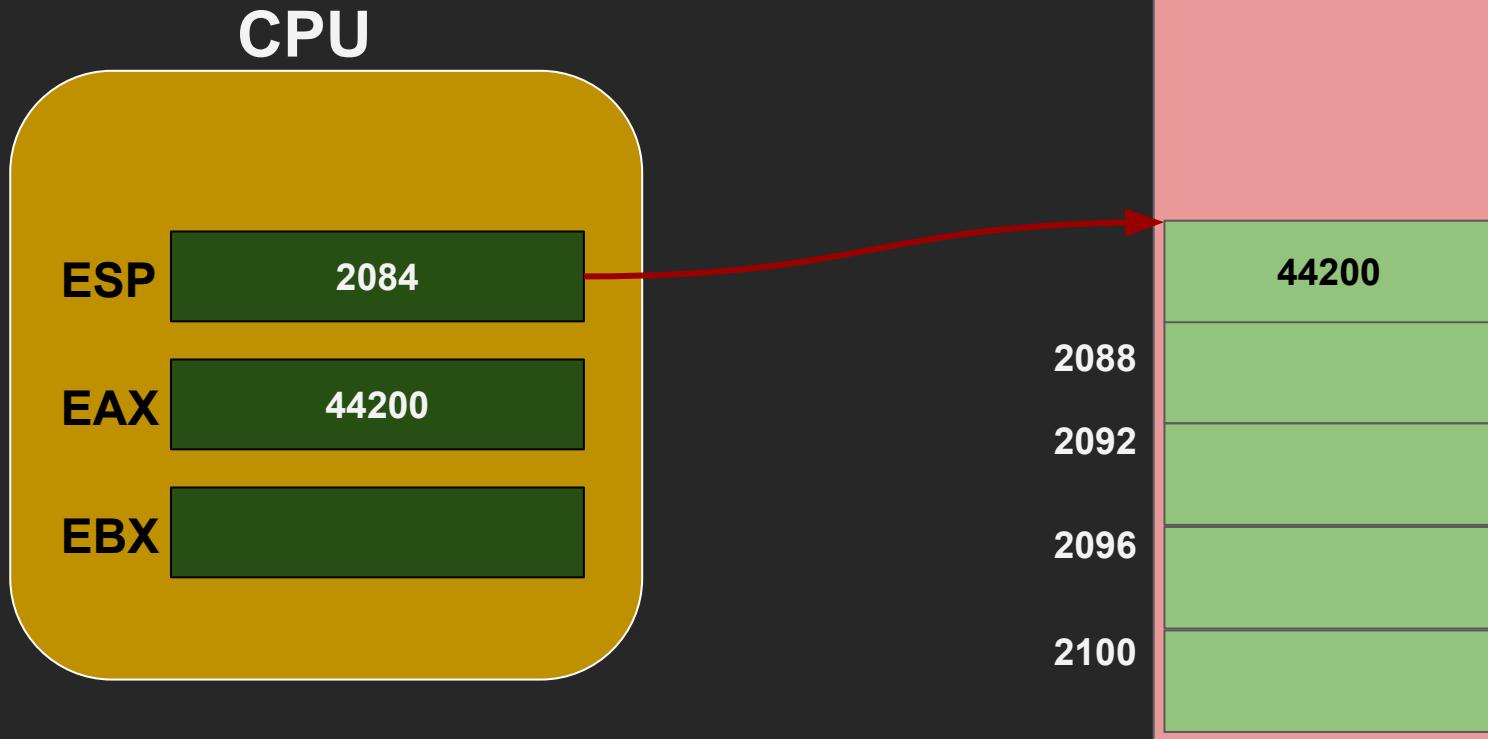


```
sub esp, 4  
mov [esp], eax  
|||  
push eax
```



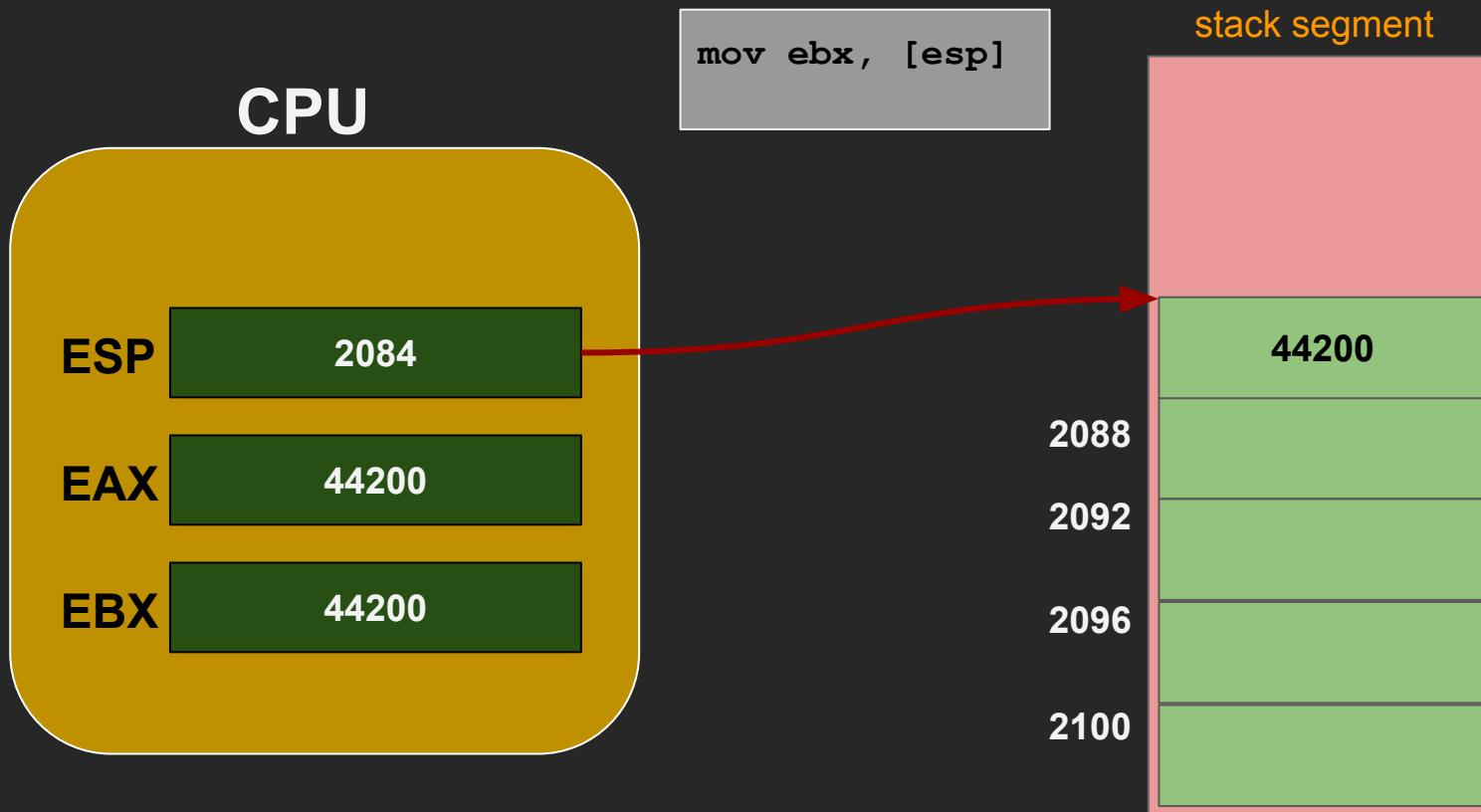


Pop into EBX



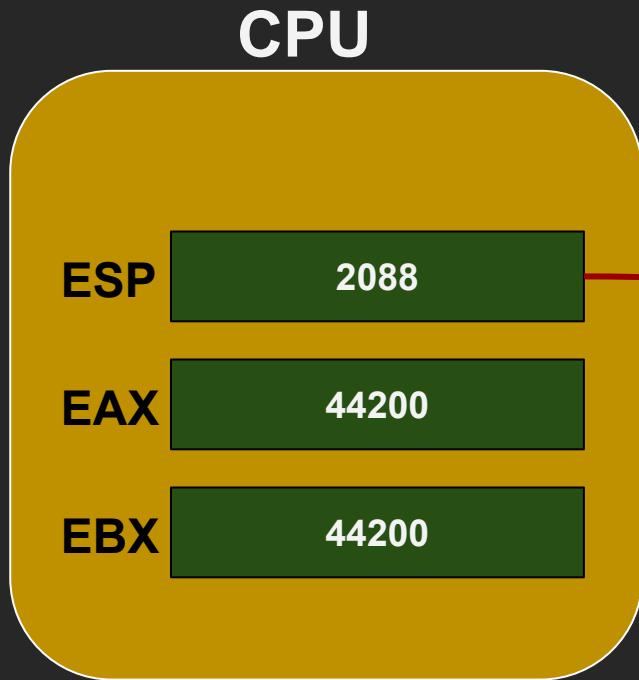


Pop into EBX

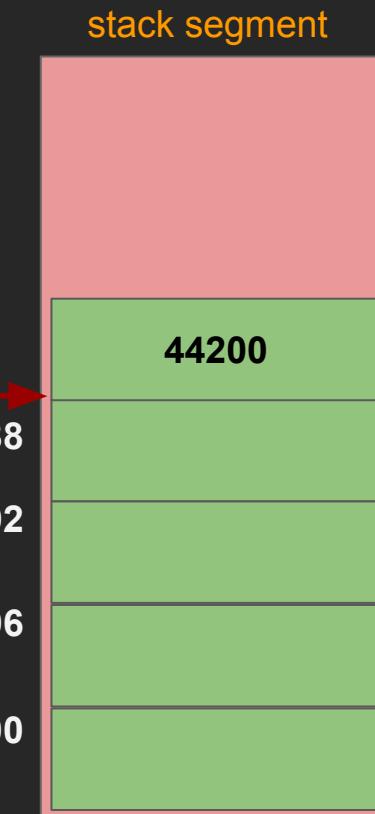




Pop into EBX

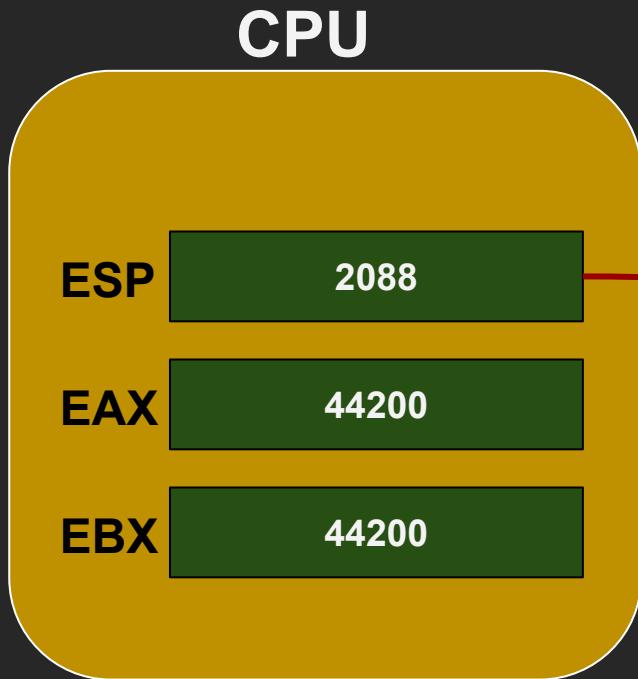


```
mov ebx, [esp]  
add esp, 4
```

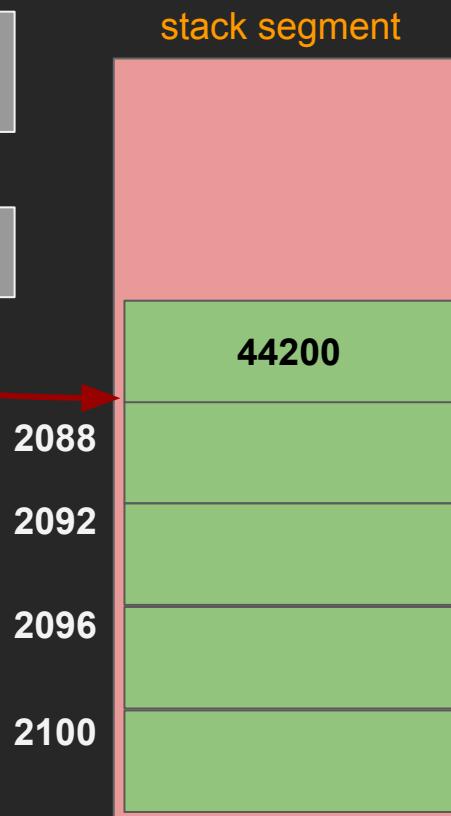




Pop into EBX

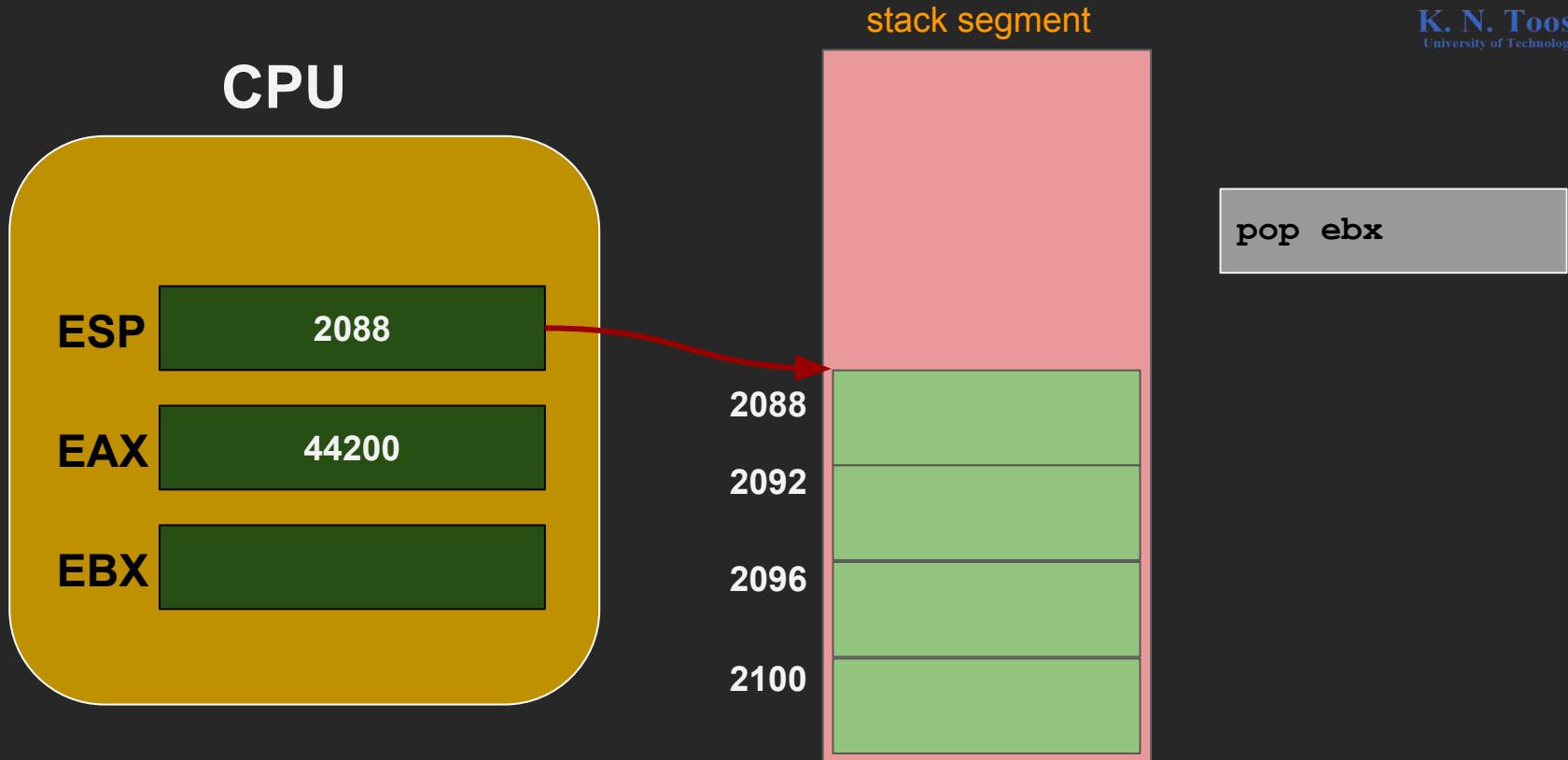


```
mov ebx, [esp]  
add esp, 4  
|||  
pop ebx
```



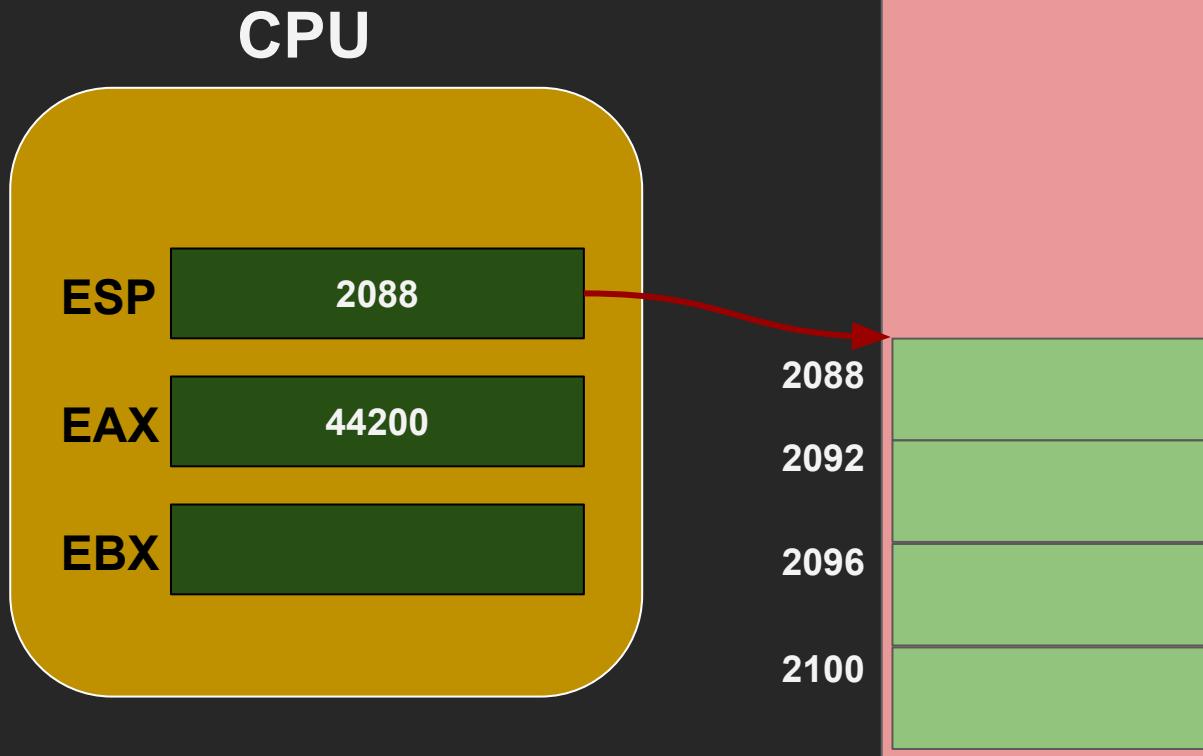


just pop 4 bytes (store nowhere)



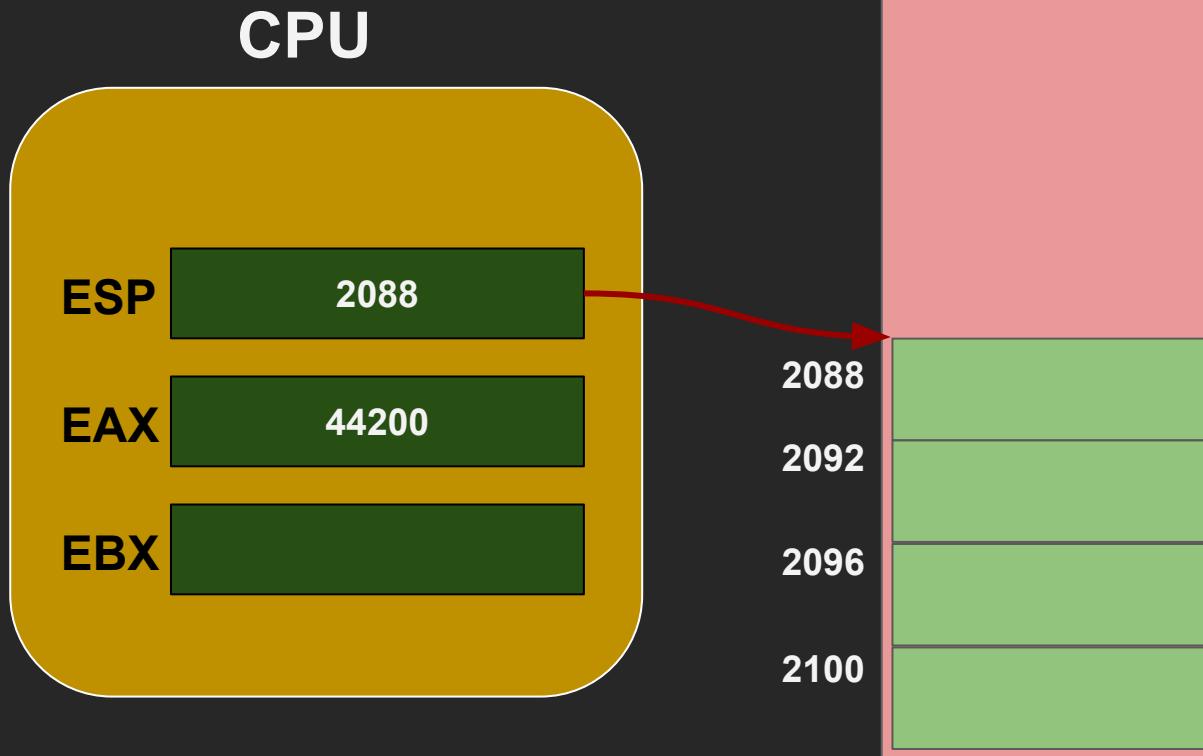


just pop 4 bytes (store nowhere)



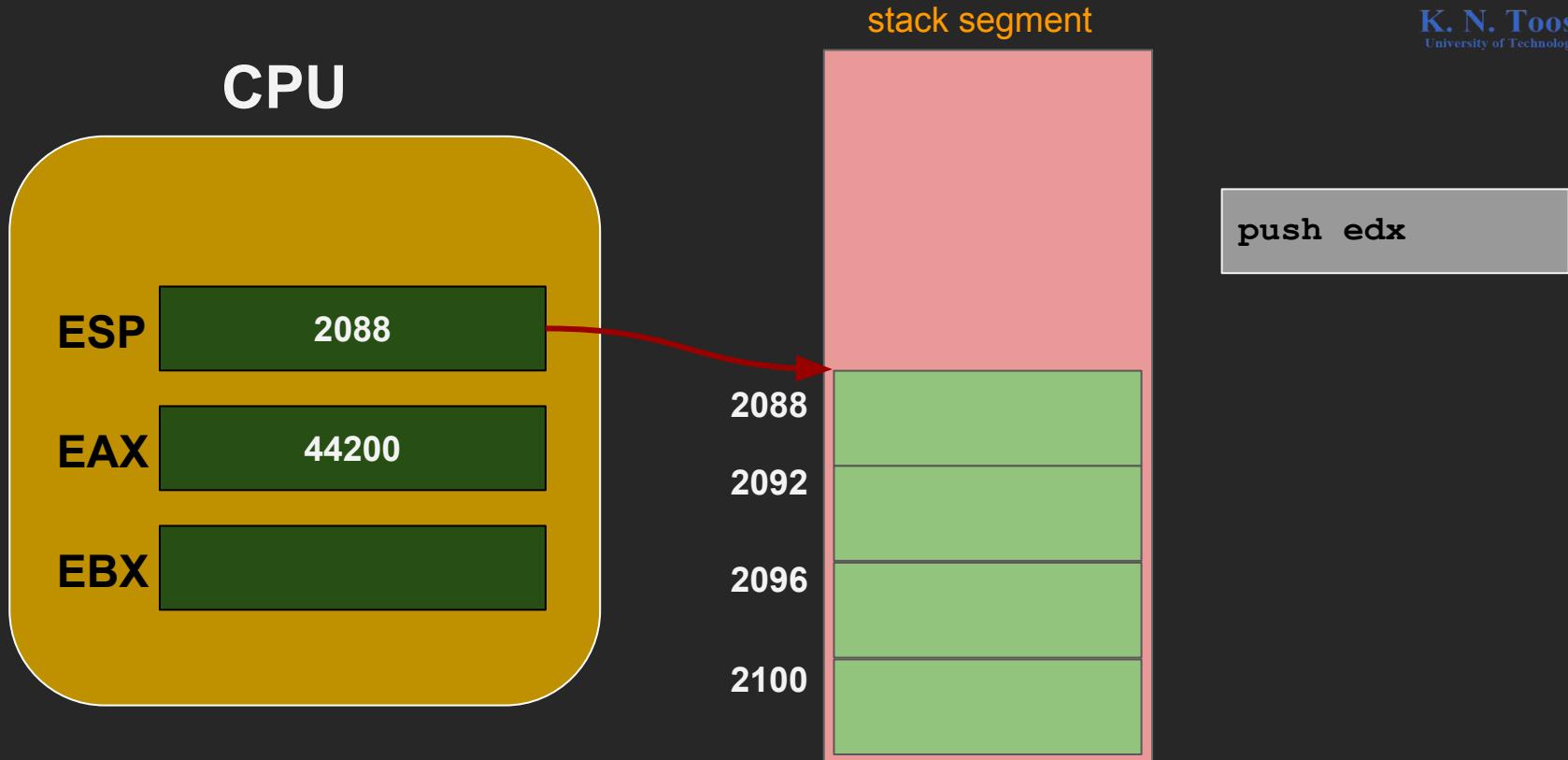


just pop 4 bytes (store nowhere)



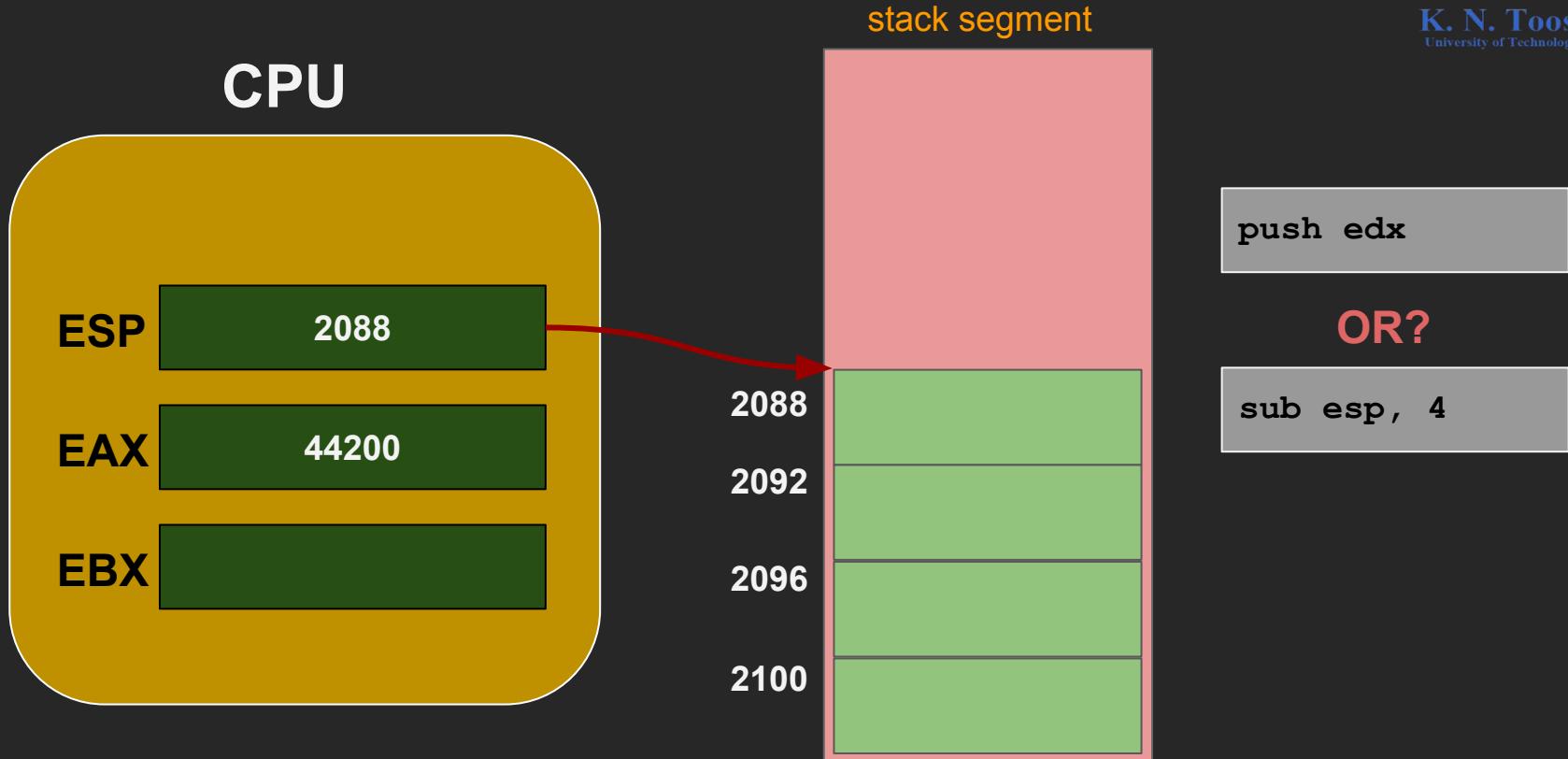


reserve memory on stack



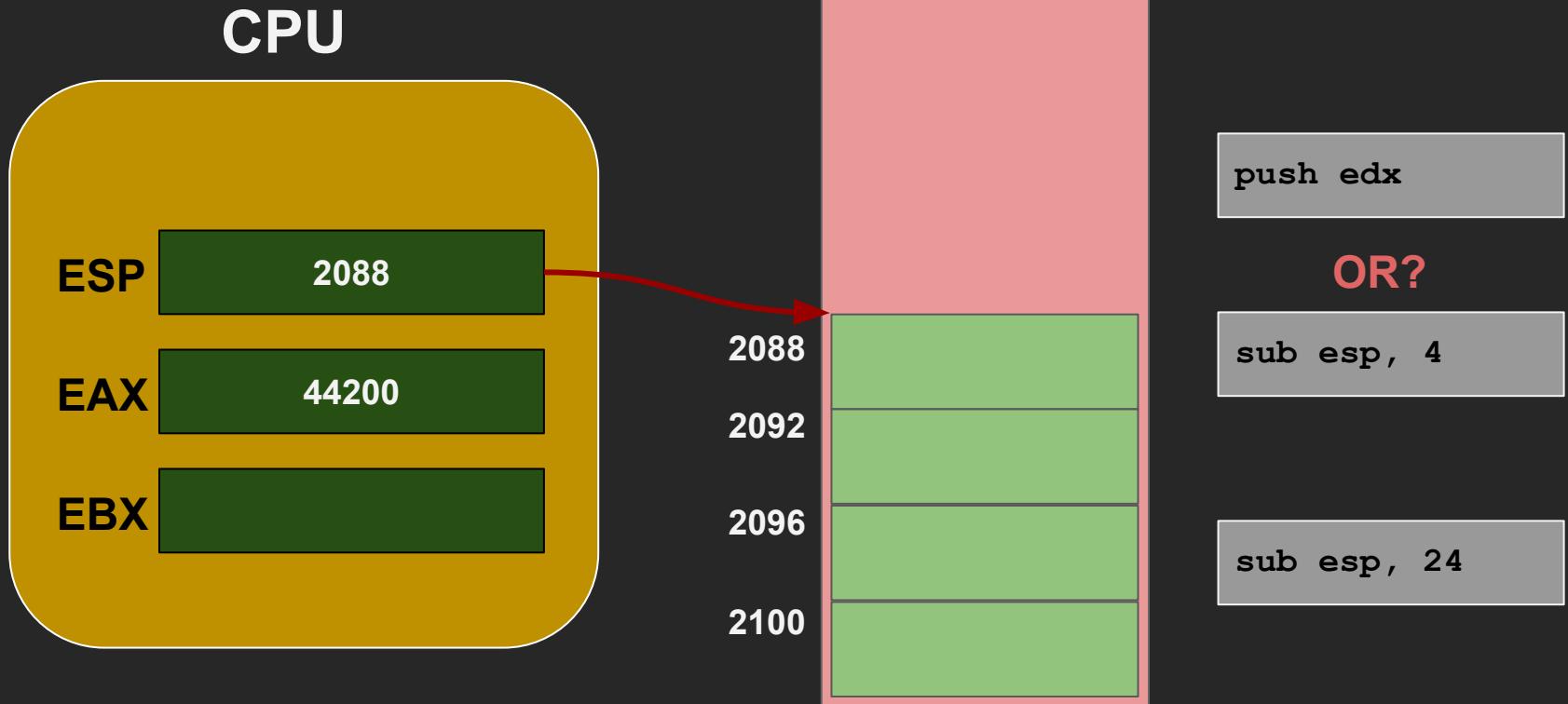


reserve memory on stack





reserve memory on stack





K. N. Toosi
University of Technology

Push and Pop

Push reg/mem/immed

Pop reg/mem

Practice



K. N. Toosi
University of Technology

push eax

push ebx

pop eax

pop ebx



pusha and popa

- 8086:
 - pusha: Push AX, CX, DX, BX, SP, BP, SI, DI
 - popa: Pop DI, SI, BP, BX, DX, CX, AX.
- 80386: (netwide assembler, what we use)
 - pusha, pushad: Push EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
 - popa, popad: Pop EDI, ESI, EBP, EBX, EDX, ECX, EAX.
- 80386: some other assemblers
 - pusha: Push AX, CX, DX, BX, SP, BP, SI, DI
 - pushad: Push EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
 - popa: Pop DI, SI, BP, BX, DX, CX, AX.
 - popad: Pop EDI, ESI, EBP, EBX, EDX, ECX, EAX
- 64 bit
 - no pusha/popa in 64-bit mode



pushf and popf

- push and pop FLAGS/EFLAGS register
- some assemblers use (pushf/pushfd/pushfq, etc.)



Back to subroutines

```
segment .data      simplefunc3.asm
msg:  db "Salaaaaam!", 10, 0
segment .text
    :
    mov edx, I1
    jmp print_salam
I1:
    mov edx, I2
    jmp print_salam
I2:
    :
print_salam:
    mov eax, msg
    call print_string
    jmp edx
```



Back to subroutines

segment .data simplefunc3.asm

msg: db "Salaaaaam!", 10, 0

segment .text

:

mov edx, I1

jmp print_salam

I1:

mov edx, I2

jmp print_salam

I2:

:

print_salam:

mov eax, msg

call print_string

jmp edx

segment .data simplefunc4.asm

msg: db "Salaaaaam!", 10, 0

segment .text

:

push I1

jmp print_salam

I1:

push I2

jmp print_salam

I2:

:

print_salam:

mov eax, msg

call print_string

??



Back to subroutines

segment .data simplefunc3.asm

msg: db "Salaaaaam!", 10, 0

segment .text

:

mov edx, I1

jmp print_salam

I1:

mov edx, I2

jmp print_salam

I2:

:

print_salam:

mov eax, msg

call print_string

jmp edx

segment .data simplefunc4.asm

msg: db "Salaaaaam!", 10, 0

segment .text

:

push I1

jmp print_salam

I1:

push I2

jmp print_salam

I2:

:

print_salam:

mov eax, msg

call print_string

pop edx

jmp edx



V N T

the CALL instruction

segment .data simplefunc3.asm

```
msg: db "Salaaaaam!", 10, 0  
segment .text
```

```
    :
```

```
    mov edx, I1
```

```
    jmp print_salam
```

I1:

```
    mov edx, I2
```

```
    jmp print_salam
```

I2:

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    jmp edx
```

segment .data simplefunc4.asm

```
msg: db "Salaaaaam!", 10, 0  
segment .text
```

```
    :
```

```
    push I1
```

```
    jmp print_salam
```

I1:

```
    push I2
```

```
    jmp print_salam
```

I2:

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    pop edx
```

```
    jmp edx
```

segment .data simplefunc5.asm

```
msg: db "Salaaaaam!", 10, 0  
segment .text
```

```
    :
```

```
    call print_salam
```

I1:

```
    call print_salam
```

I2:

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    pop edx
```

```
    jmp edx
```



V N T

the CALL instruction

segment .data simplefunc3.asm

```
msg: db "Salaaaaam!", 10, 0
segment .text
```

```
    :
```

```
    mov edx, I1
```

```
    jmp print_salam
```

I1:

```
    mov edx, I2
```

```
    jmp print_salam
```

I2:

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    jmp edx
```

segment .data simplefunc4.asm

```
msg: db "Salaaaaam!", 10, 0
segment .text
```

```
    :
```

```
    push I1
```

```
    jmp print_salam
```

I1:

```
    push I2
```

```
    jmp print_salam
```

I2:

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    pop edx
```

```
    jmp edx
```

segment .data simplefunc5.asm

```
msg: db "Salaaaaam!", 10, 0
segment .text
```

```
    :
```

```
call print_salam
```

```
call print_salam
```

```
    :
```

print_salam:

```
    mov eax, msg
```

```
    call print_string
```

```
    pop edx
```

```
    jmp edx
```

the *CALL* instruction



K. N. Toosi
University of Technology

CALL is merely a form of jump!

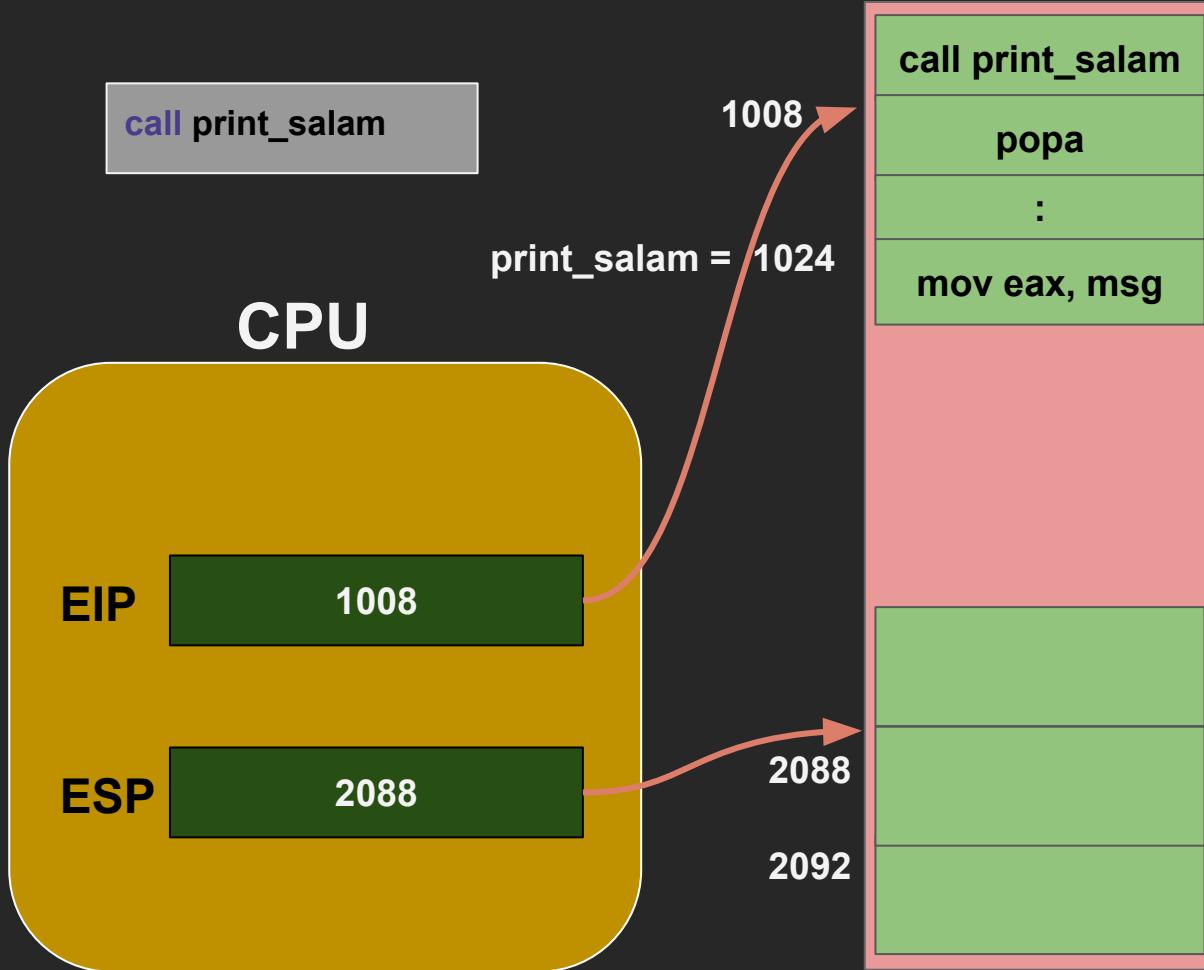


the *CALL* instruction

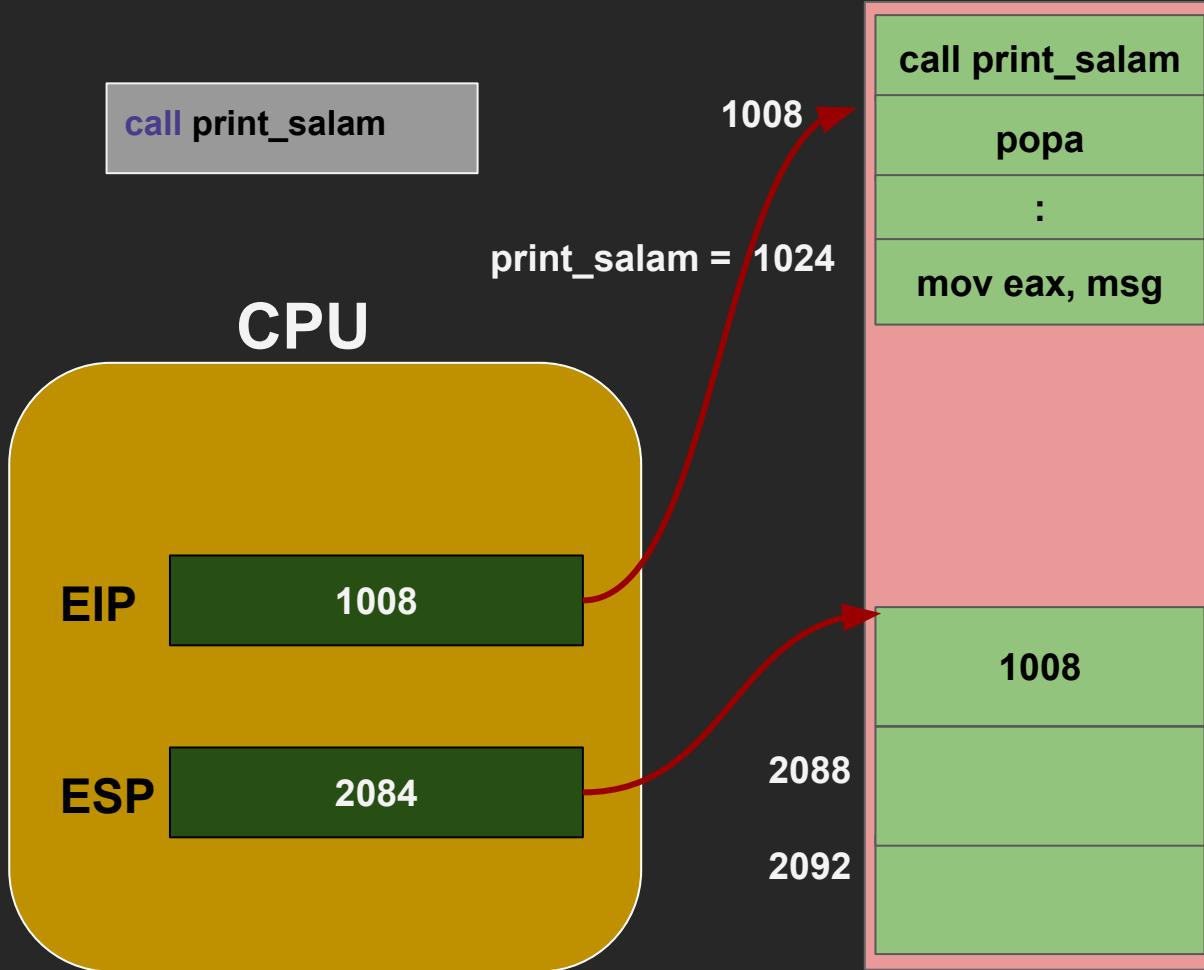
CALL is merely a form of jump!

`call label1`

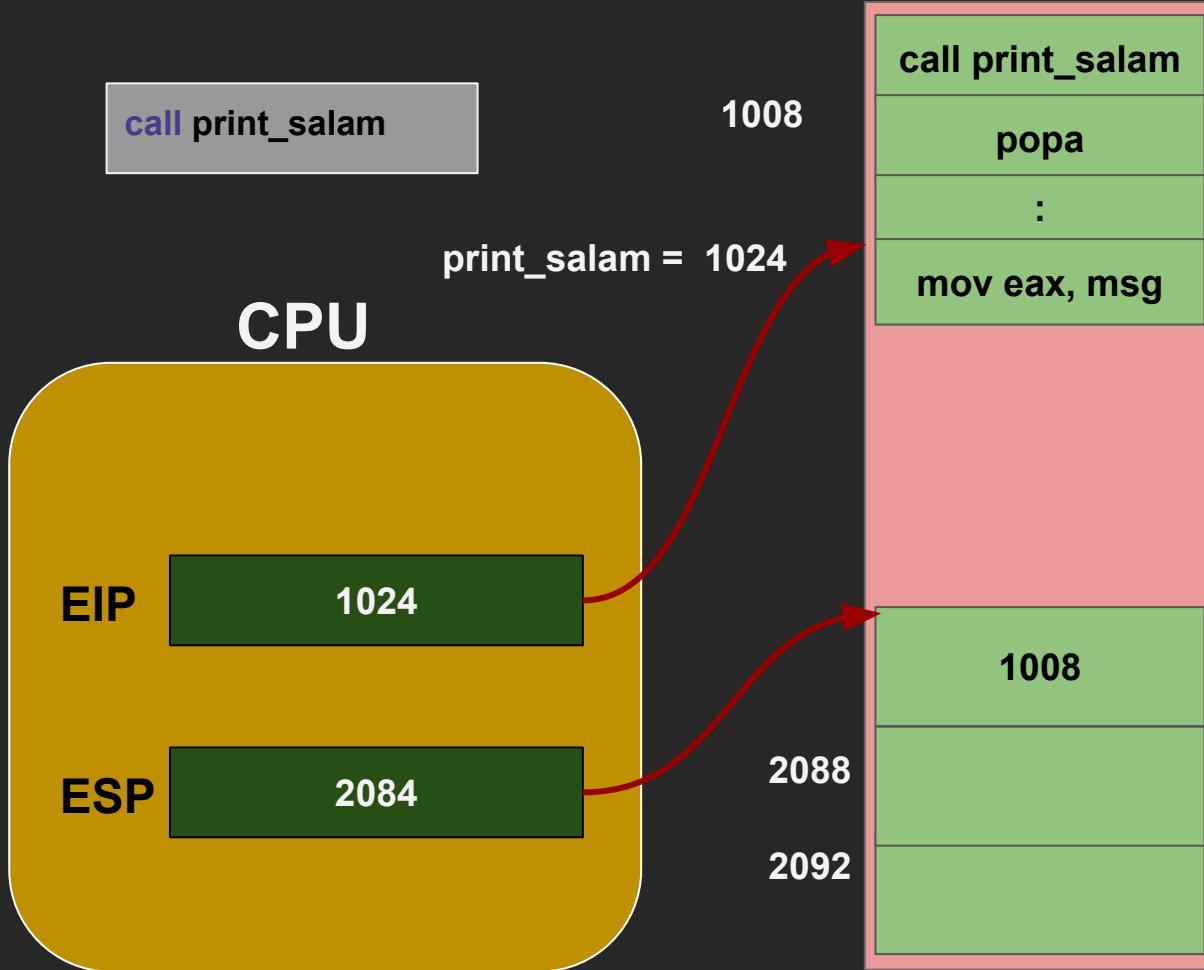
- Push return address on stack
- jump to `label1`



```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
call print_salam
popa
:
print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```



```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
call print_salam
popa
:
print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```



```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
call print_salam
popa
:
print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```



returning from a subroutine

```
segment .data      simplefunc5.asm
msg:   db "Salaaaaam!", 10, 0
segment .text
    :

    call print_salam

    call print_salam

    :

print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```



the RET instruction

```
segment .data      simplefunc5.asm
msg:  db "Salaaaaam!", 10, 0
segment .text
:
call print_salam

call print_salam

:
print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```

```
segment .data      simplefunc6.asm
msg:  db "Salaaaaam!", 10, 0
segment .text
:
call print_salam

call print_salam

:
print_salam:
    mov eax, msg
    call print_string
    ret
```



the RET instruction

```
segment .data      simplefunc5.asm
msg:  db "Salaaaaam!", 10, 0
segment .text
:
call print_salam

call print_salam

:
print_salam:
    mov eax, msg
    call print_string
    pop edx
    jmp edx
```

```
segment .data      simplefunc6.asm
msg:  db "Salaaaaam!", 10, 0
segment .text
:
call print_salam

call print_salam

:
print_salam:
    mov eax, msg
    call print_string
    ret
```



the RET instruction

RET is merely a form of jump!

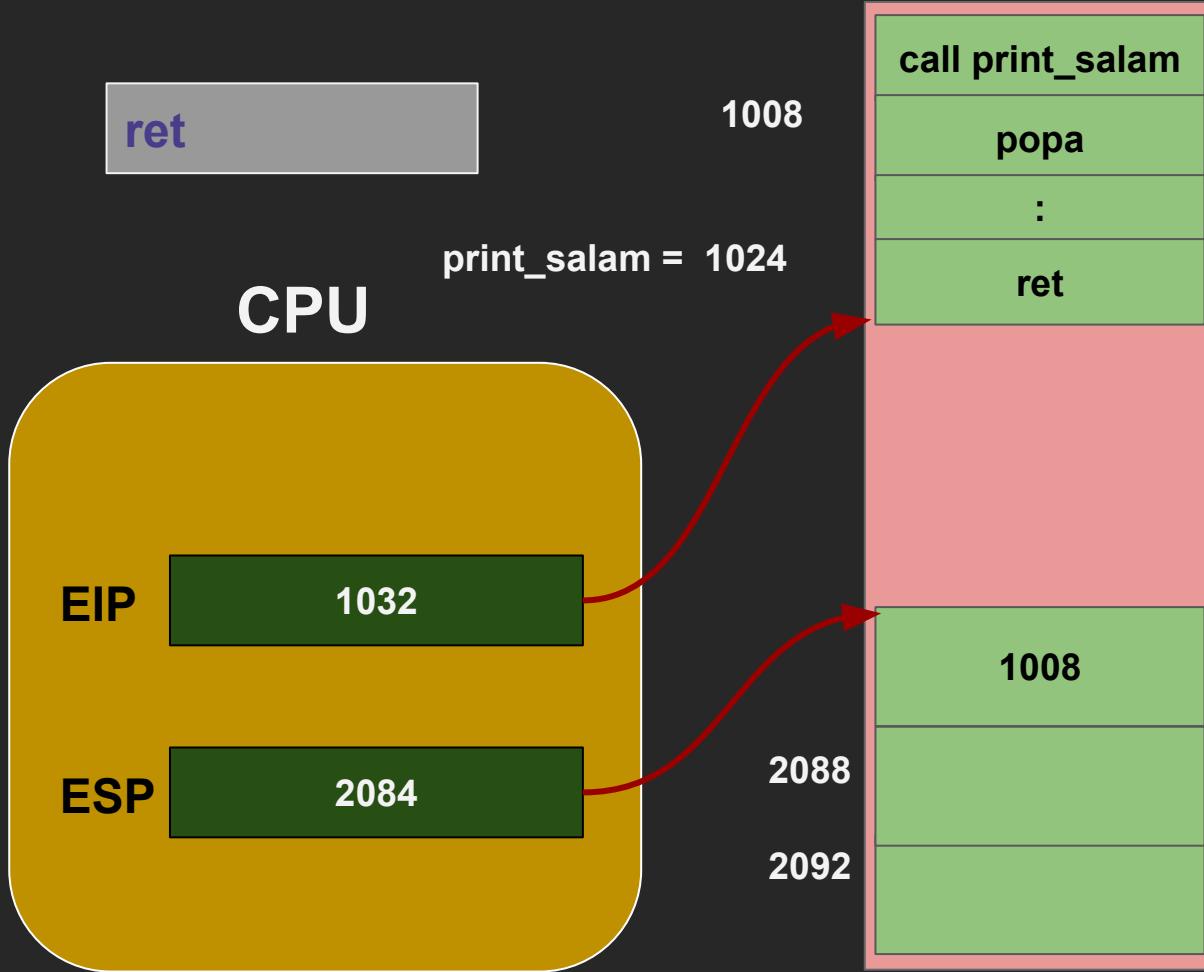


the RET instruction

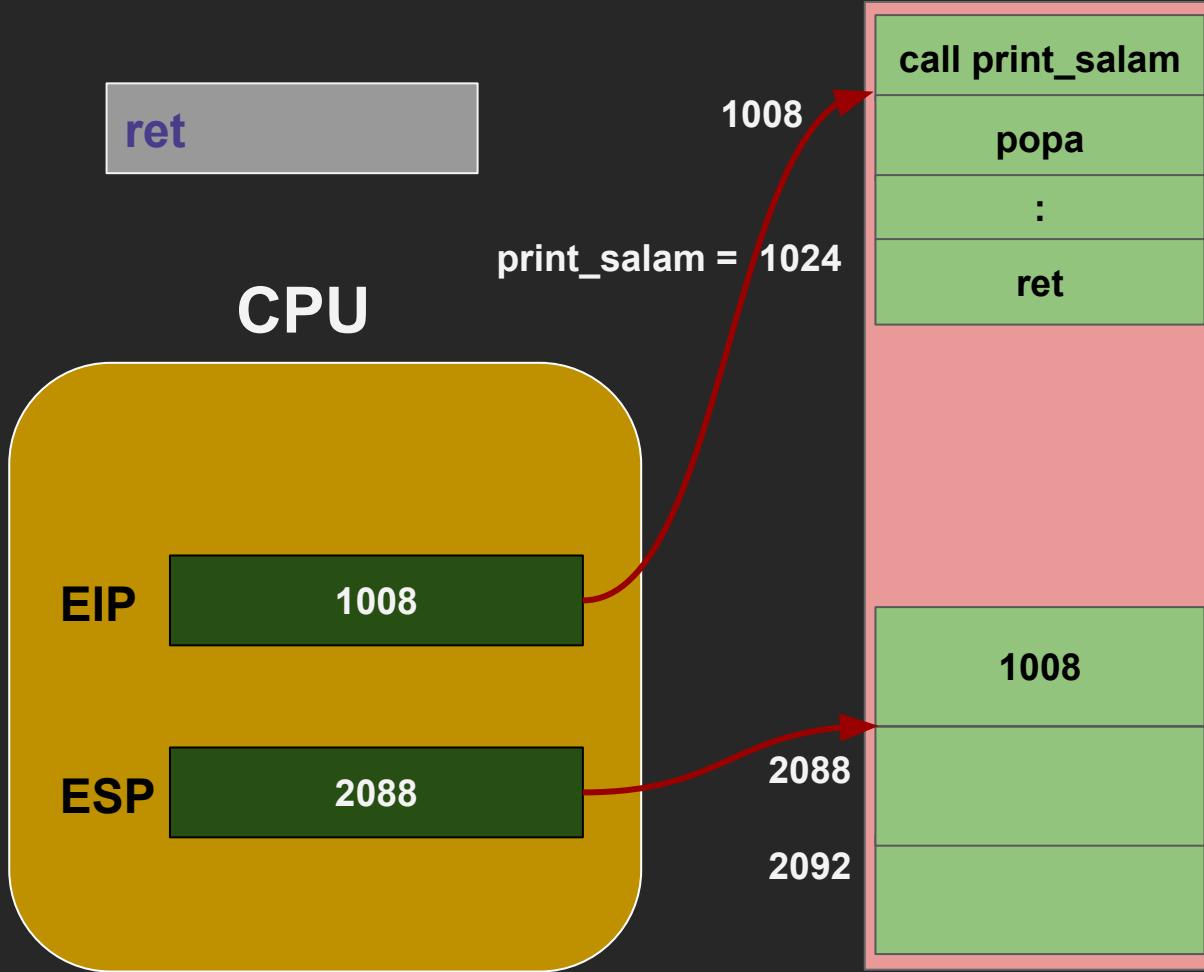
RET is merely a form of jump!

ret

- jump to the address stored on top of stack
- pop stack



```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
call print_salam
popa
:
print_salam:
    mov eax, msg
    call print_string
    ret
```



```
segment .data
msg: db "Salaaaaam!", 10, 0
segment .text
:
call print_salam
popa
:
print_salam:
    mov eax, msg
    call print_string
    ret
```



What else?

- parameters (arguments)
- local variables
- return values