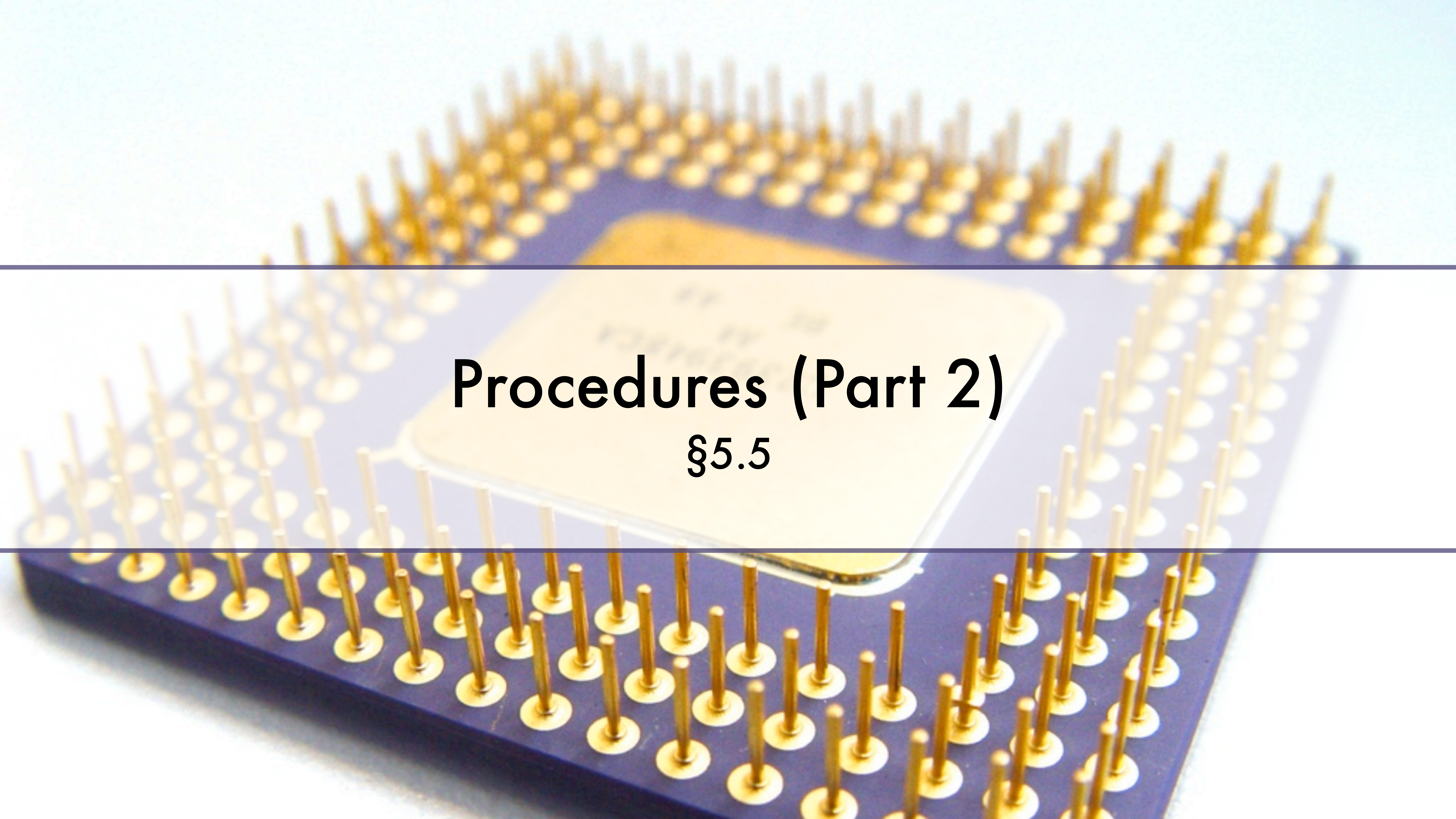
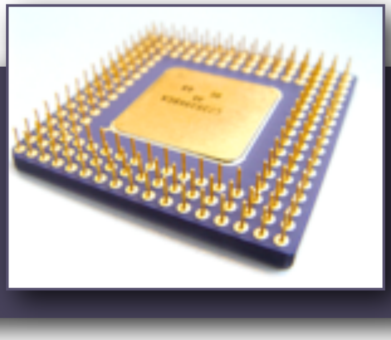


Procedures (Part 2)

§5.5

One Way to Implement a Stack



- `array DWORD 256 DUP(?)`
- `top_address DWORD (OFFSET array + SIZEOF array)`

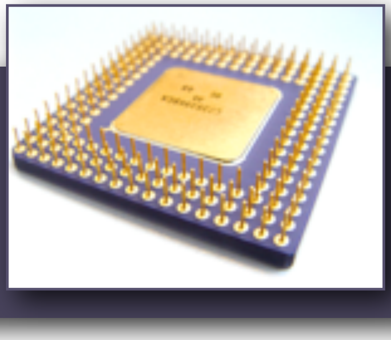
- *Push (push 32-bit value in EAX onto stack):*
  - `sub top_address, 4` *; Stack grows downward in memory!*
  - `mov esi, top_address`
  - `mov [esi], eax`

- *Pop (remove 32-bit top element, return in EAX):*
  - `mov esi, top_address`
  - `mov eax, [esi]`
  - `add top_address, 4` *; Omit this to implement Top*

This is essentially how the runtime stack works

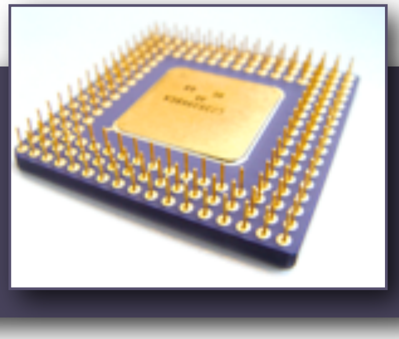
(but the top address is stored in ESP)

Topics Covered in Notes:

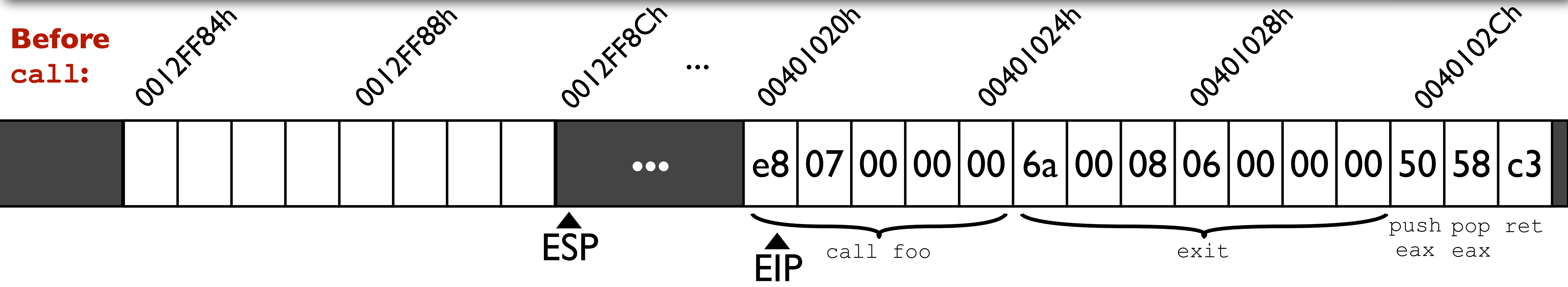


- PUSH instruction
- POP instruction
- CALL instruction
- RET instruction

Recall: Runtime Stack – How It's Used



Before call:



0012FF84h 0012FF88h 0012FF8Ch … 00401020h 00401024h 00401028h 0040102Ch

									•••	e8	07	00	00	00	6a	00	08	06	00	00	00	50	58	c3	

ESP

EIP

call foo

exit

push eax

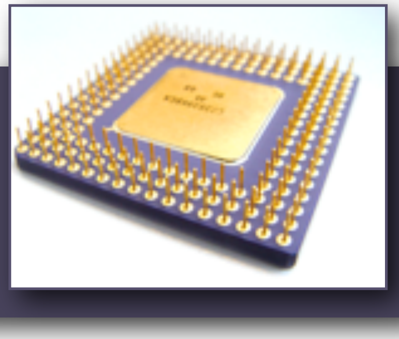
pop eax

ret

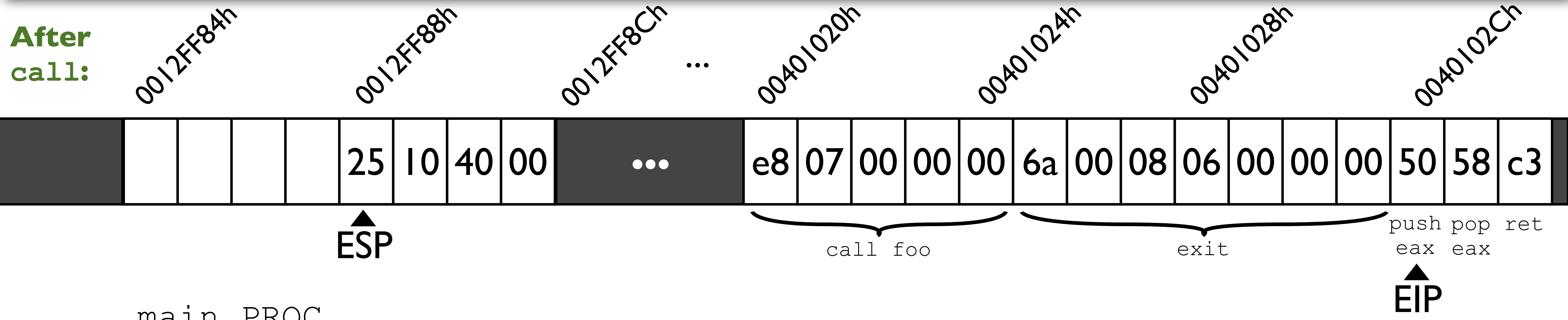
```
    main PROC
⇨       call foo
        exit
    main ENDP

    foo PROC
        push eax
        pop eax
        ret
    foo ENDP
```

- The `call` instruction will
  - Decrease ESP by 4
  - Store the address of the instruction *following* *`call`* at the memory address now in ESP
  - Set EIP to the memory address of the first instruction in the called procedure

Recall: Runtime Stack – How It's Used



After call:

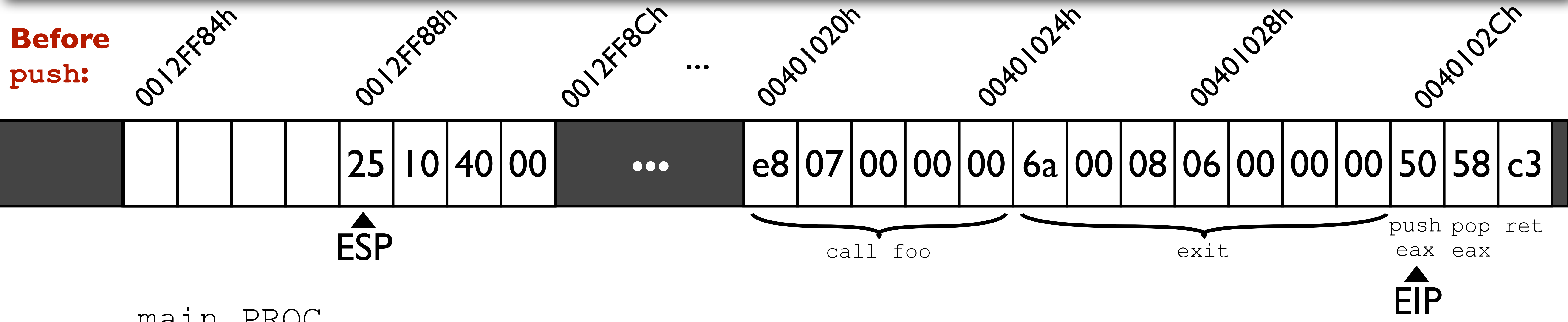


```
main PROC
    call foo
    exit
main ENDP

foo PROC
    push eax
    pop eax
    ret
foo ENDP
```

- The `call` instruction
  - Decrease ESP by 4
  - Store the address of the instruction *following* `call` at the memory address now in ESP
  - Set EIP to the memory address of the first instruction in the called procedure

Recall: Runtime Stack – How It’s Used

Before push:



0012FF84h 0012FF88h 0012FF8Ch … 00401020h 00401024h 00401028h 0040102Ch

				25	10	40	00	•••	e8	07	00	00	00	6a	00	08	06	00	00	00	50	58	c3

ESP

call foo

exit

push eax

pop eax

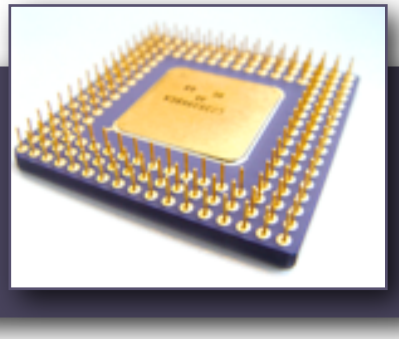
ret

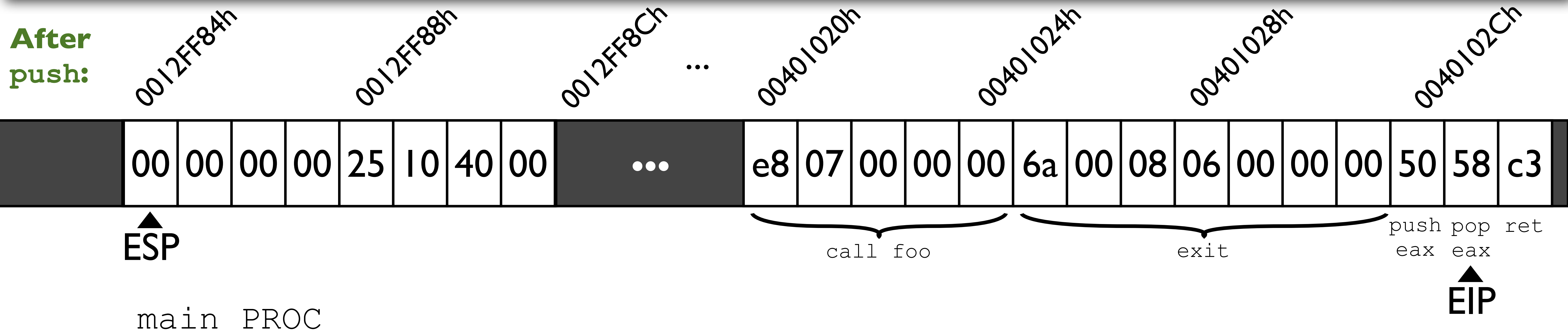
EIP

```
 main PROC
        call foo
        exit
 main ENDP

 foo PROC
⇨       push eax
        pop eax
        ret
 foo ENDP
```

- The `push` instruction will
  - Decrease ESP by 4
  - Store the value indicated at the address in ESP (we’ll assume EAX contains 00000000h)

Recall: Runtime Stack – How It's Used





After push:

0012FF84h 0012FF88h 0012FF8Ch … 00401020h 00401024h 00401028h 0040102Ch

00	00	00	00	25	10	40	00	•••	e8	07	00	00	00	6a	00	08	06	00	00	00	50	58	c3	

ESP

call foo

exit

push eax

pop eax

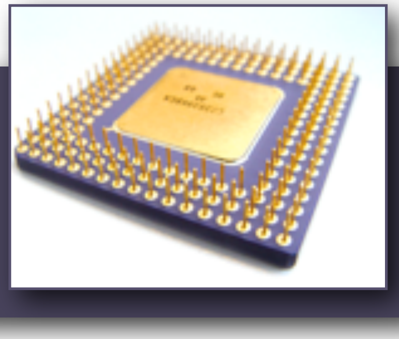
ret

EIP

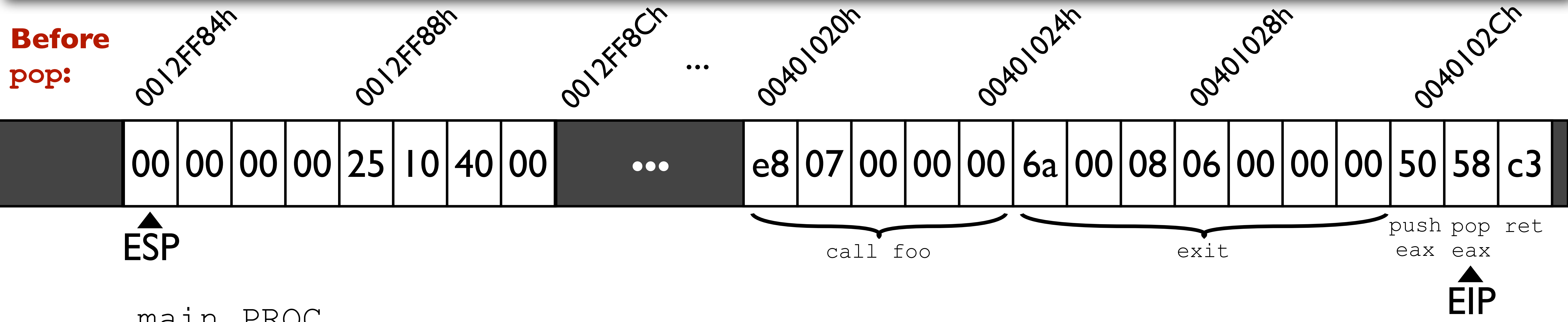
```
main PROC
    call foo
    exit
main ENDP

foo PROC
    push eax
    pop eax
    ret
foo ENDP
```

- The push instruction will
  - Decrease ESP by 4
  - Store the value indicated at the address in ESP (we'll assume EAX contains 00000000h)

Recall: Runtime Stack – How It’s Used



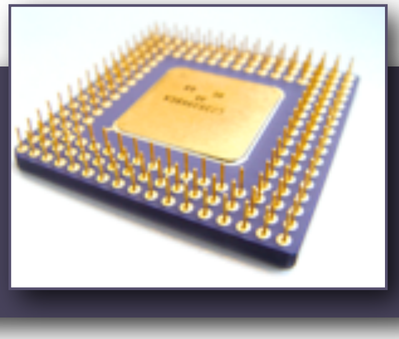
Before pop:



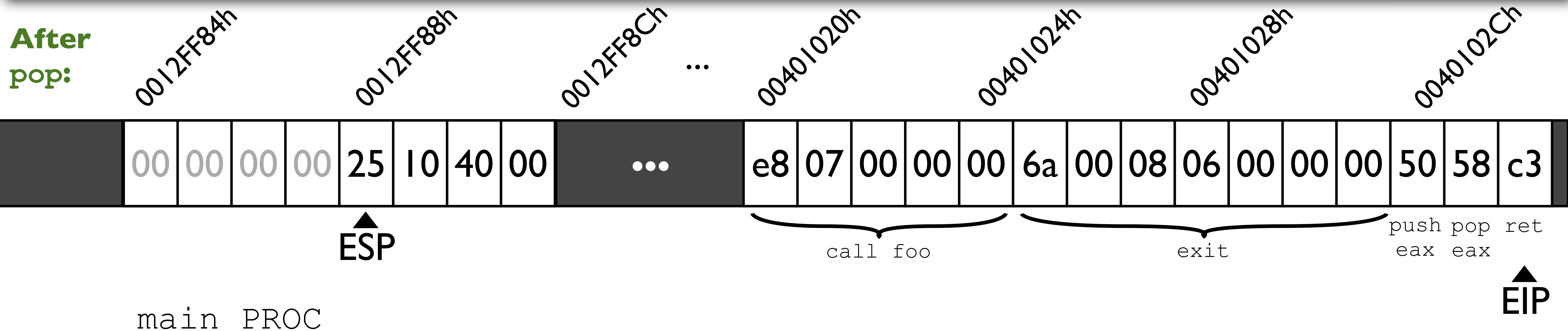
```
main  PROC
     call  foo
     exit
main  ENDP

foo  PROC
     push  eax
     pop  eax
     ret
foo  ENDP
```

- The pop instruction will
  - Load the value from the address given by ESP, copying it into the given register
  - Increase ESP by 4

Recall: Runtime Stack – How It’s Used



After pop:

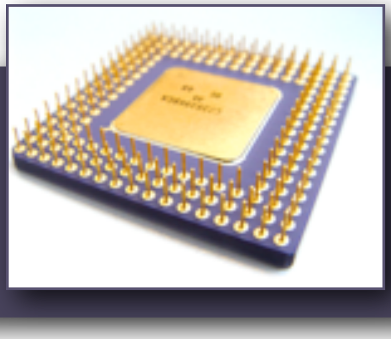


00 00 00 00 25 10 40 00 ••• e8 07 00 00 00 6a 00 08 06 00 00 00 50 58 c3

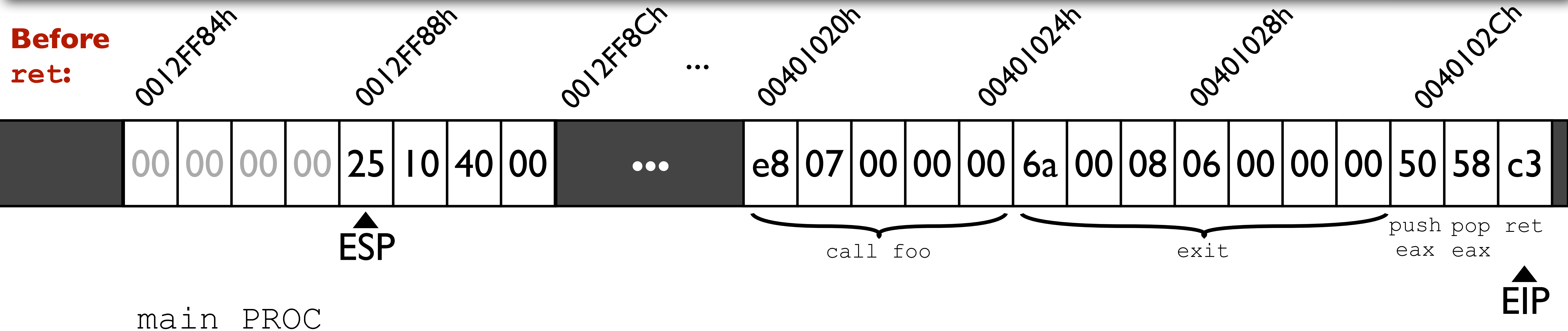
```
main PROC
    call foo
    exit
main ENDP

foo PROC
    push eax
    pop eax
⇨   ret
foo ENDP
```

- The pop instruction will
  - Load the value from the address given by ESP, copying it into the given register
  - Increase ESP by 4

Recall: Runtime Stack – How It's Used



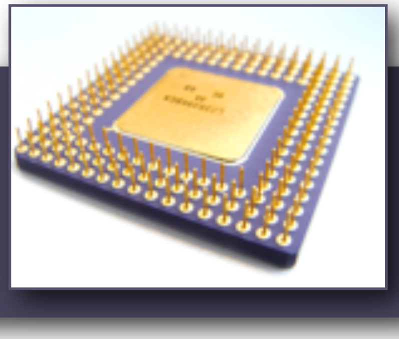
Before ret:



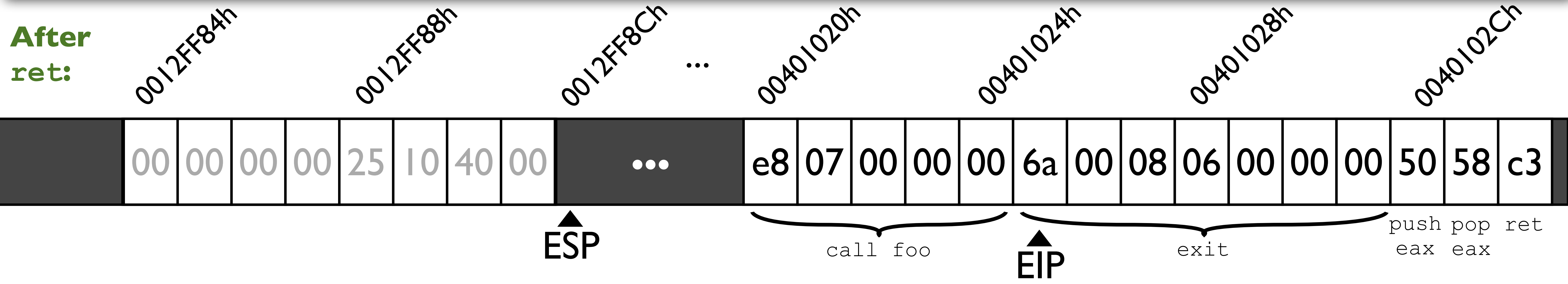
```
main PROC
    call foo
    exit
main ENDP

foo PROC
    push eax
    pop eax
    ret
foo ENDP
```

- The `ret` instruction will
  - Read the 32-bit value at ESP (in this example, 00401025h)
  - Increase ESP by 4
  - Set EIP to the value it just read (00401025h)

Recall: Runtime Stack – How It's Used



After ret:

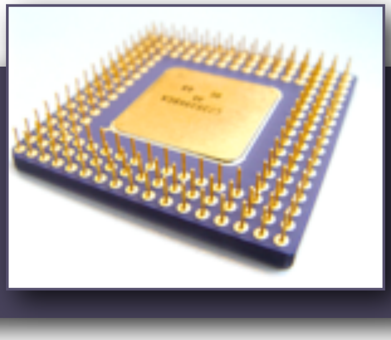


00 00 00 00 25 10 40 00 ••• e8 07 00 00 00 6a 00 08 06 00 00 00 50 58 c3

```
    main PROC
        call foo
⇨       exit
    main ENDP

    foo PROC
        push eax
        pop eax
        ret
    foo ENDP
```

- The `ret` instruction will
  - Read the 32-bit value at ESP (in this example, 00401025h)
  - Increase ESP by 4
  - Set EIP to the value it just read (00401025h)

Nested Procedure Calls



```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

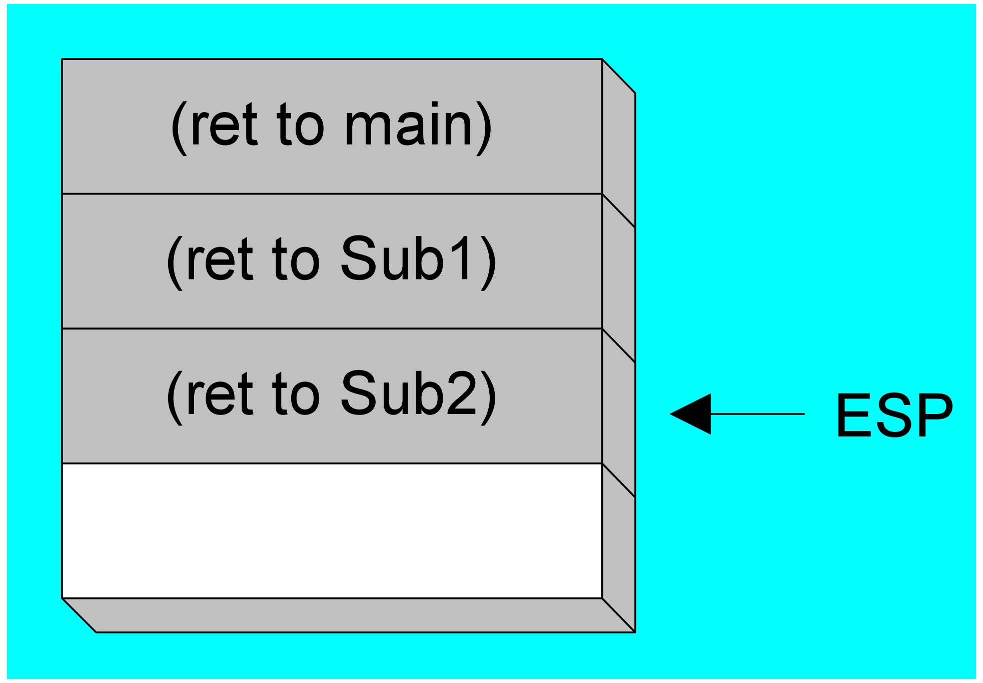
- *Nested procedure calls:*
  you call a procedure, and it calls other procedures before returning to you

- main calls Sub1
  Sub1 calls Sub2
  Sub2 calls Sub3

- Recall: Stacks are last-in-first-out (LIFO) data structures

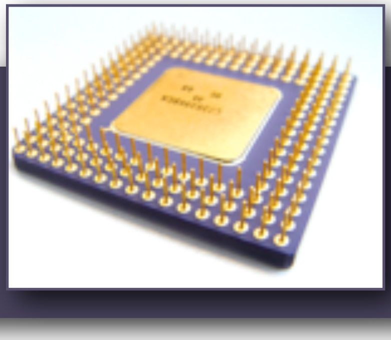
- You always want to return to the *last* procedure that CALLed

- The last procedure that CALLed will be the first address POPped

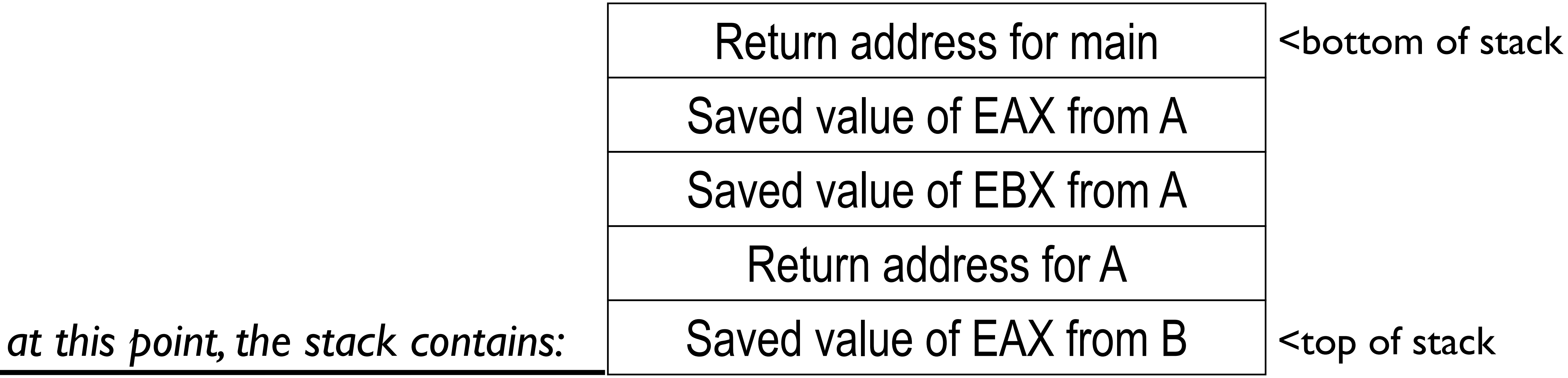


Irvine, Kip R. Assembly Language for x86 Processors 6/e, 2010.

Nested Procedure Calls



```
main PROC
    call A
    exit
main ENDP

A PROC
    push eax
    push ebx
    call B
    pop ebx
    pop eax
    ret
A ENDP

B PROC
    push eax
    pop eax
    ret
B ENDP
```



at this point, the stack contains:

Return address for main
Saved value of EAX from A
Saved value of EBX from A
Return address for A
Saved value of EAX from B