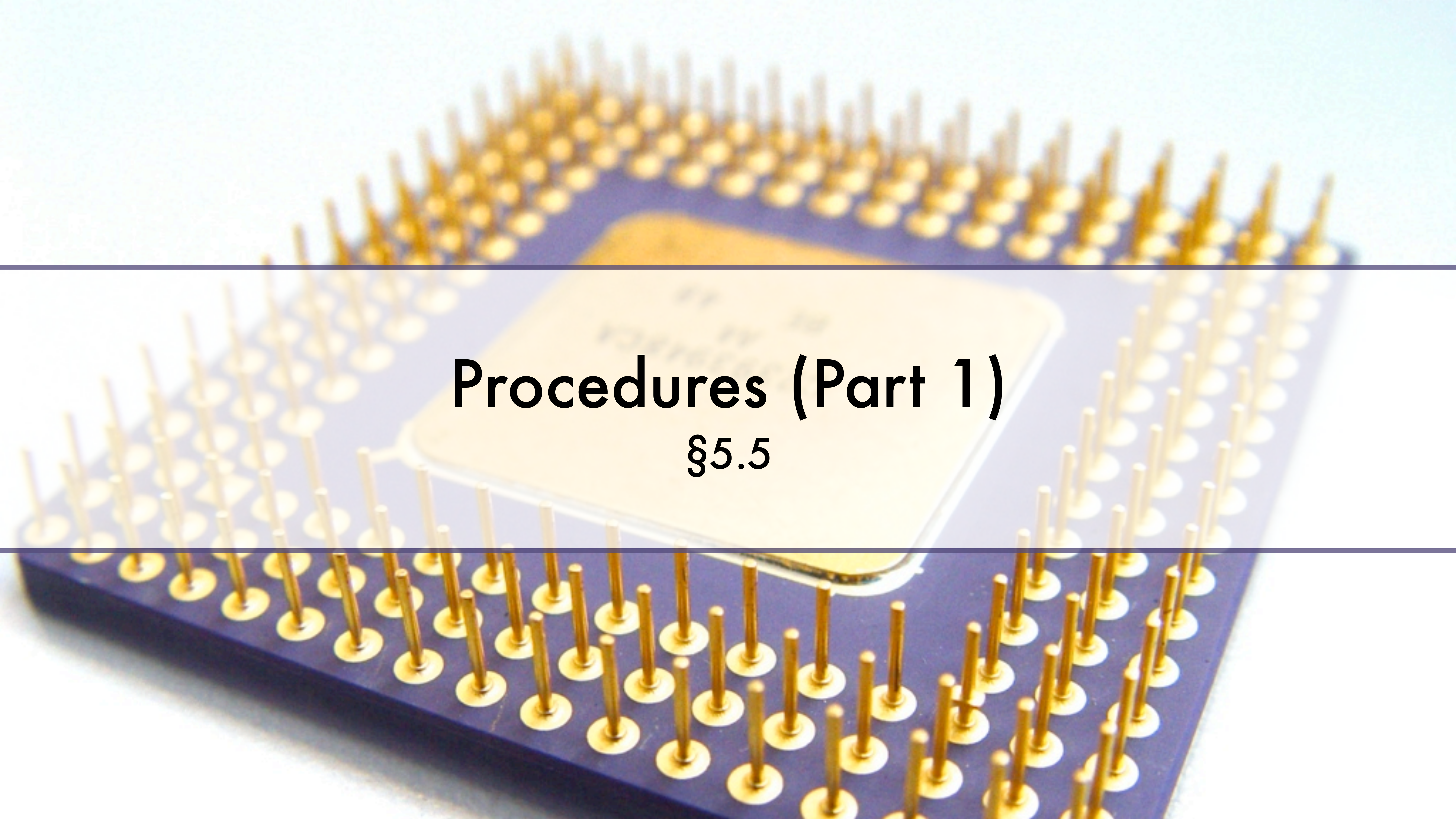
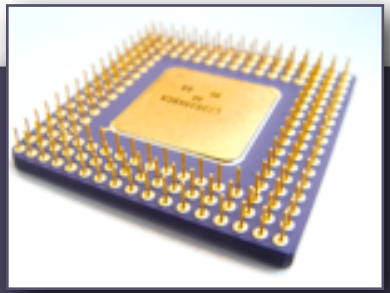


Procedures (Part 1)

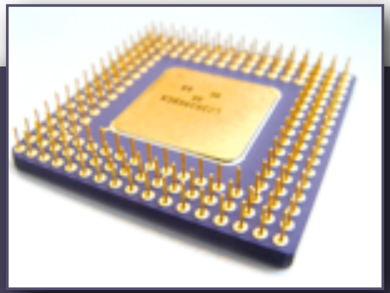
§5.5

Homework



- Meet in **labs** (2119/2122) on Monday
- **Homework 3** due Wednesday
- For next class (Monday, October 6):
  - Read **Section 4.1** (6/e pp. 94–103 **or** 7/e pp. 96–104)
    - Be able to explain and use these instructions: LAHF, SAHF, XCHG
  - Read 6th Edition: **Sections 5.4–5.5.2** (skip rest of §5.5) (pp. 157–168)
or 7th Edition: **Sections 5.1–5.2.4** (skip rest of §5.2) (pp. 140–150)
    - Be able to explain & use PUSHFD, PUSHAD, POPFD, & POPAD

Procedures



- *Procedures* are also called *subroutines* or *functions*

Name

Parameters

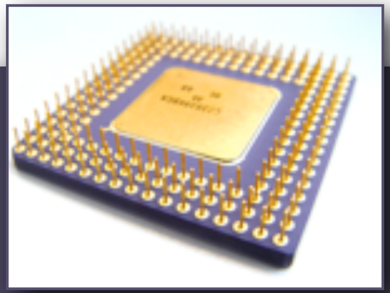
```
int sum(int x, int y) {
      return x + y;
}
```

Return value

- The variables *x* and *y* are called *parameters*
- When *calling* a function, as in sum(3,4), the values passed (3 and 4) are called *arguments*

- Java *methods* are defined in classes (so they're a bit different), but they receive arguments and return a value like procedures/subroutines/functions
  - Method calls are more complex than simple procedure calls

Procedures in Assembly/MASM



- Basic template (for now):

Name

Parameters

```
int sum(int x, int y) {
    return x + y;
}
```

Return value

Name

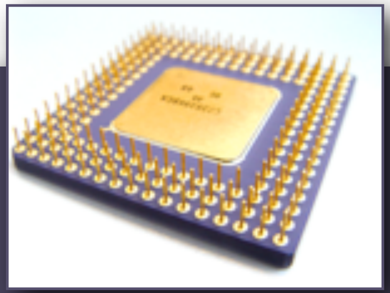
Pass arguments in registers

```
sum PROC
    add eax, ebx
    ret
sum ENDP
```

Put return value in EAX

RET instruction is **MANDATORY**

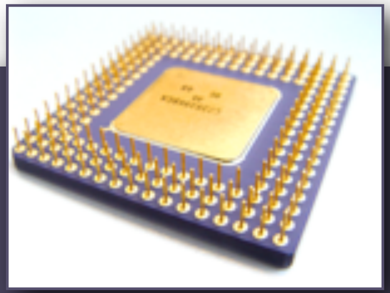
Defining Procedures, Part I



- Define the procedure using the PROC directive
  - *procedure_name* **PROC**

    ```
    procedure_name  PROC
            ...
            ret     ; Issue a RET instruction to return
    procedure_name  ENDP
    ```
- If arguments are required, pass them in registers
  - These are called *register parameters.*
  - The preferred way to pass arguments is using *stack parameters* (Chapter 8).
- To return a value, place it in EAX
- *Always* issue a RET instruction!
  - If you do not, your program will probably crash

Calling Procedures

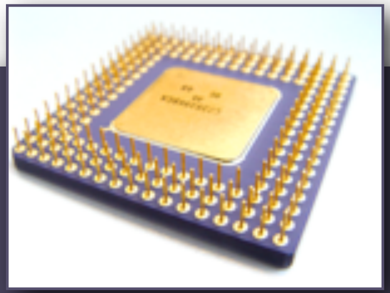


- Load arguments into registers
- Issue a `call` instruction
- If the procedure returns a value, load it from EAX

Example 1: Sum

```
INCLUDE Irvine32.inc
.code
main PROC
    mov eax, 3
    mov ebx, 2
    call Sum
    ; Now EAX contains 5
    exit
main ENDP
; ----------------------------------------
Sum PROC
; Adds signed or unsigned integer values
; Receives: EAX, EBX -- Values to add
; Returns:  EAX -- Sum
; ----------------------------------------
    add eax, ebx
    ret
Sum ENDP
end main
```

Documenting Procedures



- Document each procedure with:
  - A one-sentence description of what the procedure does
    - Don’t just restate the procedure name; paraphrase!
  - What arguments it expects in which registers
  - What value(s) it returns in which register(s) (if any)
  - Constraints on argument and return values (preconditions/postconditions)
    - E.g., “EAX must be nonzero”

```
; ----------------------------------------
Sum PROC
; Adds signed or unsigned integer values
; Receives: EAX, EBX -- Values to add
; Returns:  EAX -- Sum
; ----------------------------------------
```

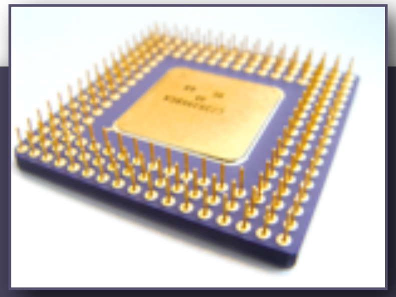
Example 2: WriteSmiley

```
INCLUDE Irvine32.inc
.data
emoticon BYTE ":-)", 0Dh, 0Ah, 0
.code
main PROC
    call WriteSmiley
    exit
main ENDP
; ----------------------------------------
WriteSmiley PROC
; Displays a happy emoticon
; Receives: None
; Returns: None
; ----------------------------------------
    mov edx, OFFSET emoticon
    call WriteString
    ret
WriteSmiley ENDP
end main
```

BAD

Modifies EDX but doesn't claim to return a value in EDX

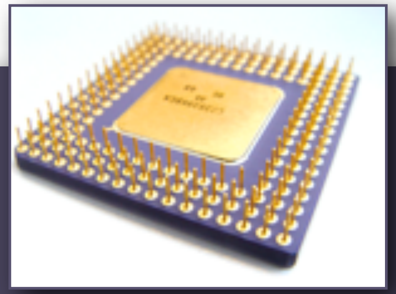
Defining Procedures, Part II



- If your procedure modifies any registers but does not return values in them,
  - Save their original values using the PUSH instruction
  - Before returning, restore values using POP
    - Pop registers in reverse order from what you pushed
    - Critical: must pop exactly the number of values pushed

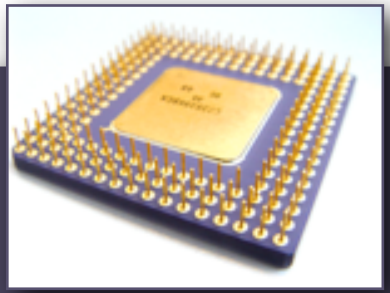
- ```
  procedure_name PROC
          push eax
          push ebx
          ; Now do stuff with EAX and EBX
          pop ebx
          pop eax
          ret
  procedure_name ENDP
  ```
  ```

Example 2: WriteSmiley



```
INCLUDE Irvine32.inc
.data
emoticon BYTE ":-)", 0Dh, 0Ah, 0
.code
main PROC
    call WriteSmiley
    exit
main ENDP
; ----------------------------------------
WriteSmiley PROC
; Displays a happy emoticon
; Receives: None
; Returns: None
; ----------------------------------------
    push edx
    mov edx, OFFSET emoticon
    call WriteString
    pop edx
    ret
WriteSmiley ENDP
end main
```

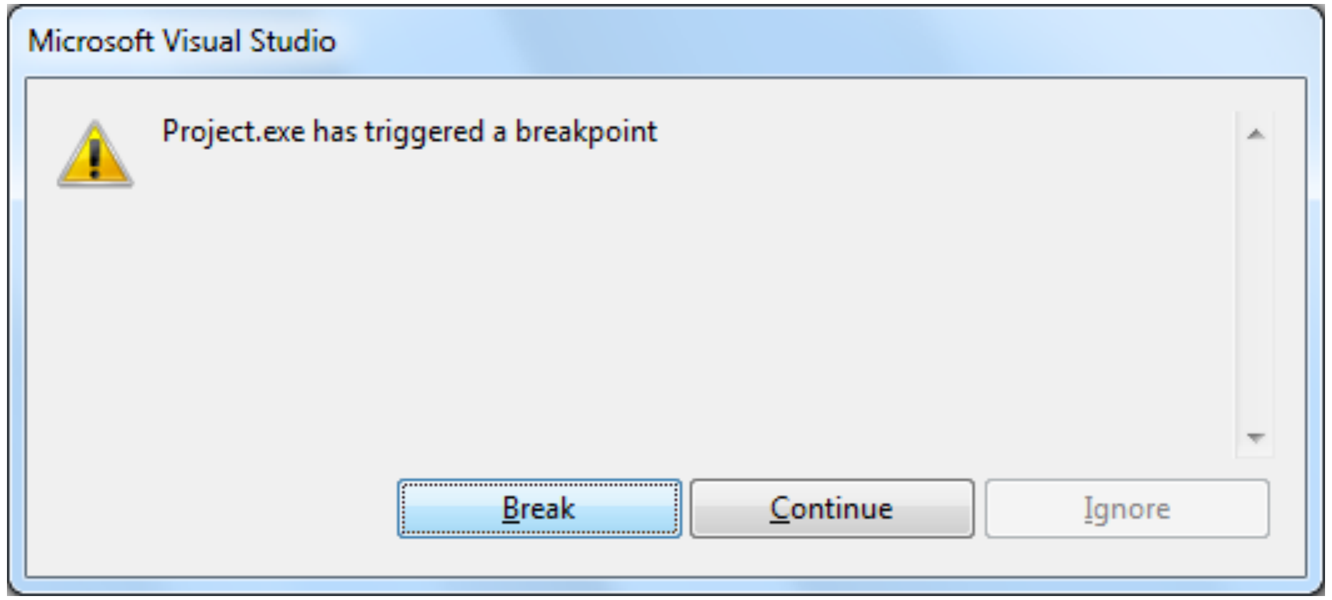
Be Careful



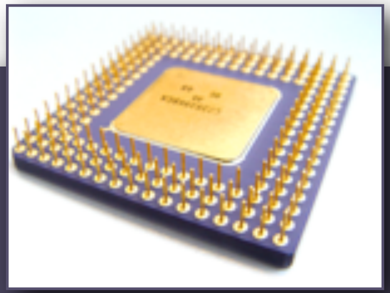
- **Q:** What is wrong with this code?

```
WriteIt PROC
      call WriteDec
WriteIt ENDP
```

- **A:** It does not issue a RET instruction. **BAD**



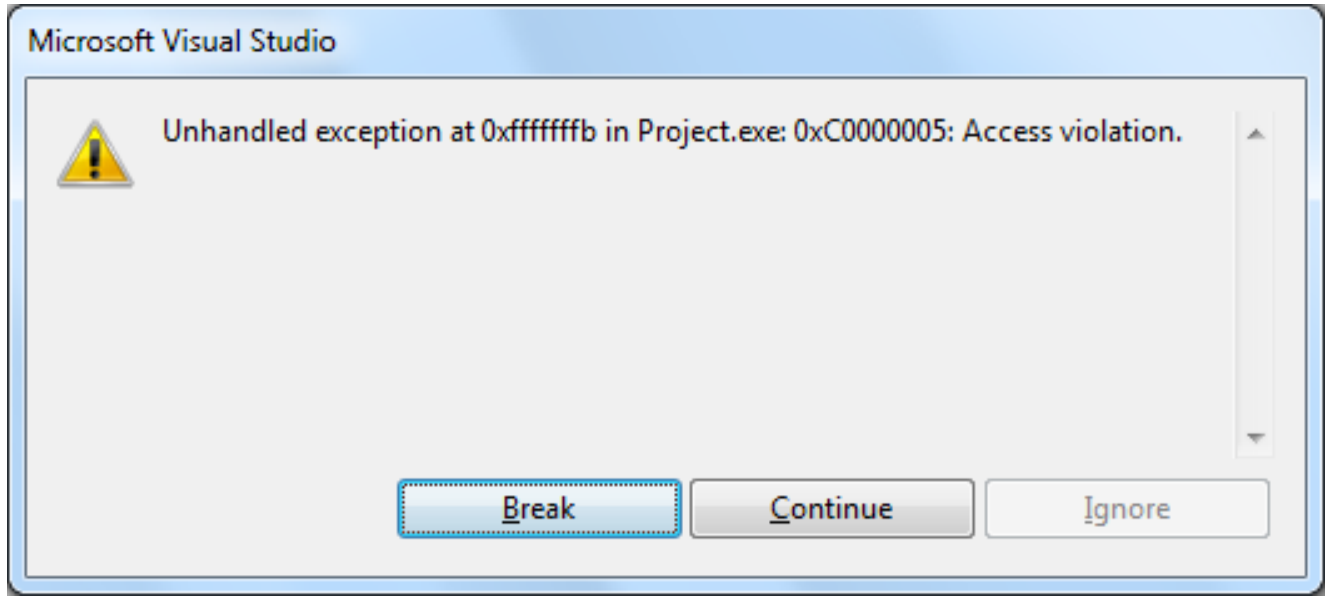
Be Careful



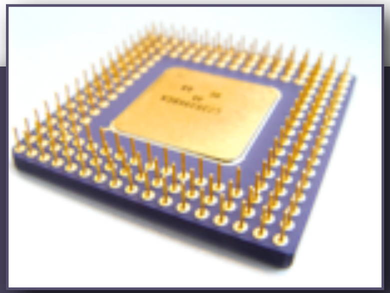
- **Q:** What is wrong with this code?

```
WriteIfPositive PROC
      push eax
      cmp eax, 0
      jle done
      call WriteDec
      pop eax
done: ret
WriteIfPositive ENDP
```

- **A:** If the argument is negative, it does not pop the stack. **BAD**



Labels

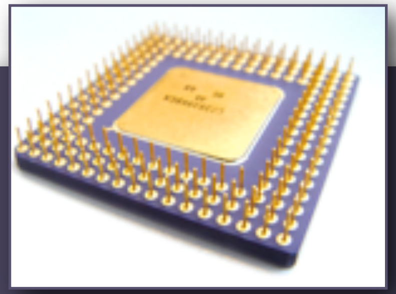


- Labels are local to a procedure, so the same label can be used in multiple procedures

```
foo PROC
        push eax
        jmp done   ; Refers to done in foo
  done: pop eax
        ret
foo ENDP

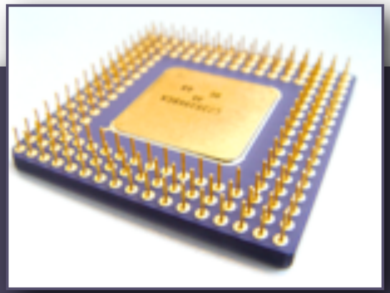
bar PROC
        jmp done   ; Refers to done in bar
  done: ret
bar ENDP
```

Summary



- Define procedures using PROC and ENDP
- Document purpose, arguments, return value
- Pass arguments in registers (for now)
- Return value (if any) in EAX
- Procedures *must* issue a RET instruction
- Save and restore register values using PUSH, POP
- Pop values in *reverse* order

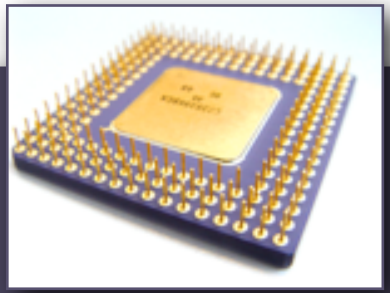
Exercises



1. What is wrong with the following?

```
; Adds two 32-bit integers
; Receives: EAX, EBX -- Values to add
; Returns: EAX -- Sum
sum PROC
    add eax, ebx
sum ENDP
```

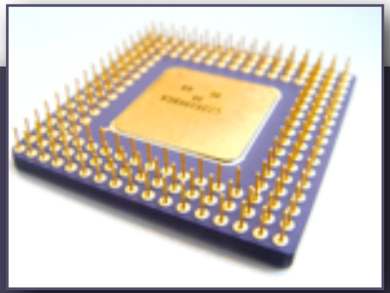
Exercises



2. What is wrong with the following?

```
; Subtracts 32-bit integers
; Receives: EAX, EBX -- Values to subtract
; Returns: EAX -- Difference (EAX-EBX)
sub PROC
    sub eax, ebx
    ret
sub ENDP
```

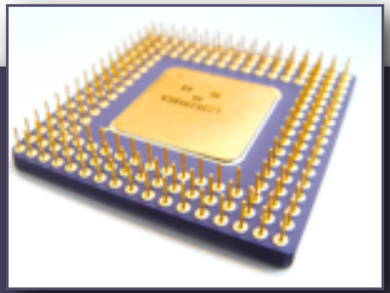
Exercises



3. What is wrong with the following?

```
; Doubles a 32-bit unsigned integer value
; Receives: EAX -- Value to double
; Returns: EAX -- 2*EAX
PROC double
    add eax, eax
    ret
END double
```

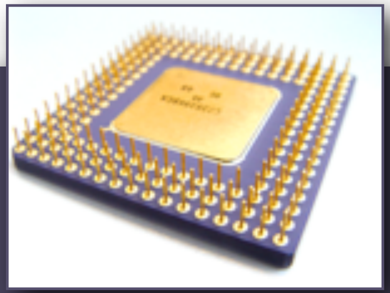
Exercises



4. What is wrong with the following?

```
; Displays an input value iff it is nonzero
; Receives: EAX -- 32-bit unsigned integer
; Returns: None
writeIfNonzero PROC
       mov ecx, eax    ; Copy input to ECX
       jecxz done
       call WriteDec   ; EAX ≠ 0; display it
done:  ret
writeIfNonzero ENDP
```

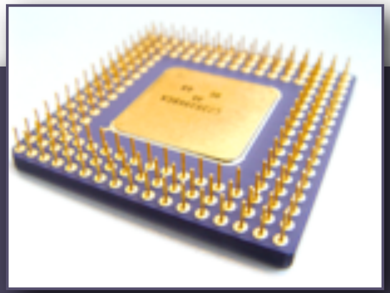
Exercises



5. What is wrong with the following?

```
; Displays an input value iff it is nonzero
; Receives: EAX -- 32-bit unsigned integer
; Returns: None
writeIfNonzero PROC
        push eax
        push ecx
        mov ecx, eax    ; Copy input to ECX
        jecxz done
        call WriteDec   ; EAX ≠ 0; display it
        pop ecx
        pop eax
done:   ret
writeIfNonzero ENDP
```

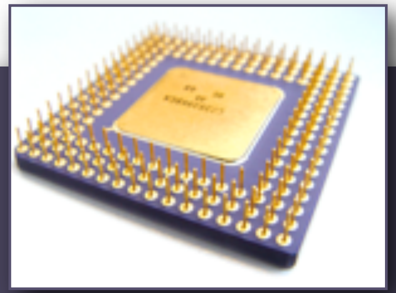
Exercises



6. What is wrong with the following?

```
; Displays an input value iff it is nonzero
; Receives: EAX -- 32-bit unsigned integer
; Returns: None
writeIfNonzero PROC
        push eax
        push ecx
        mov ecx, eax    ; Copy input to ECX
        jecxz done
        call WriteDec   ; EAX ≠ 0; display it
done:   pop eax
        pop ecx
        ret
writeIfNonzero ENDP
```

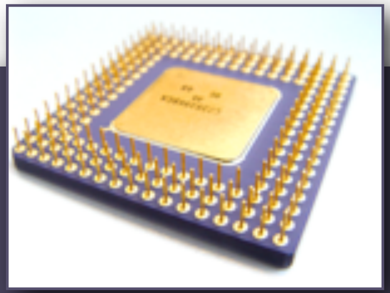
Exercises



7. How do you call this procedure to display the value 100?

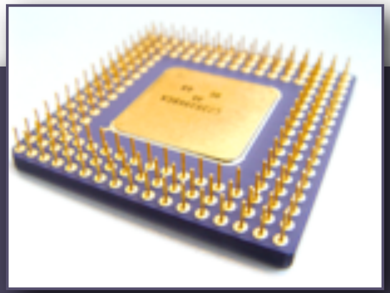
```
; Displays an input value iff it is nonzero
; Receives: EAX -- 32-bit unsigned integer
; Returns: None
writeIfNonzero PROC
        push eax
        push ecx
        mov ecx, eax    ; Copy input to ECX
        jecxz done
        call WriteDec   ; EAX ≠ 0; display it
done:   pop ecx
        pop eax
        ret
writeIfNonzero ENDP
```

Recall from COMP 2210: Stacks



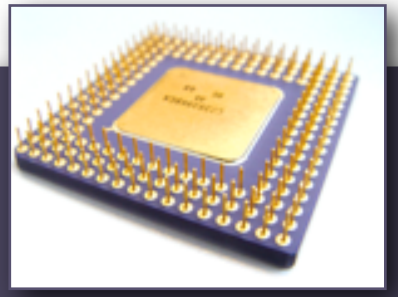
- A *stack* is an abstract data type with 3 operations:
  (sometimes more, e.g., *isEmpty*)
  - *push* adds an element to the stack
  - *pop* removes the most recently added element
  - *top* returns the most recently added element but does not remove it
- A stack is a last-in first-out (LIFO) structure since the element returned via *pop/top* is the last one (i.e., the most recent one) that was added

Runtime Stack



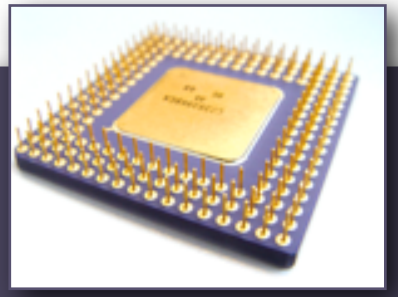
- The *runtime stack* (or just “the stack”)...
  - Consumes memory in a process’s *stack segment*
    - Recall: each process has *code*, *data*, and *stack* segments (maybe more)
  - Supported directly by the CPU
  - Grows *downward* in memory
  - ESP register contains the memory address of the top element
  - PUSH, POP, CALL, RET all affect the stack & change ESP
- Coming later (Chapter 8):
  - Procedure arguments can be passed on the stack
  - Local variables can be stored on the stack

Runtime Stack – Uses

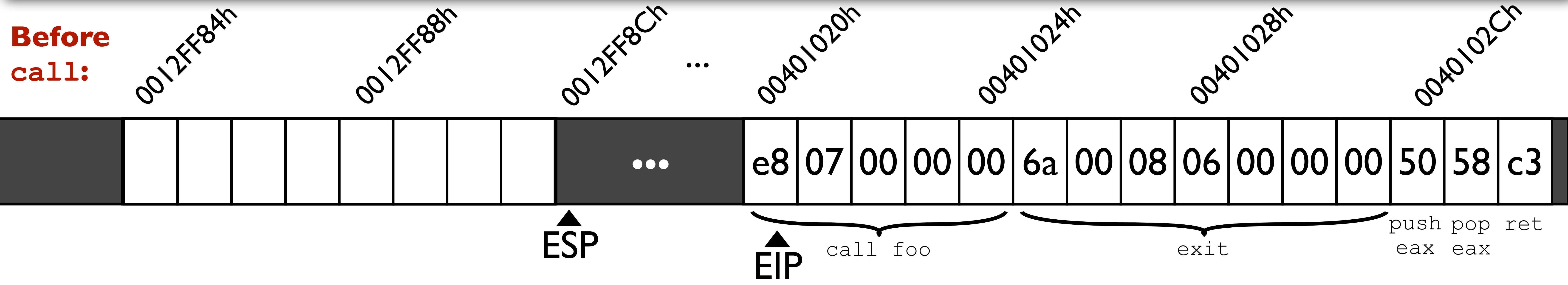


- The *runtime stack* is used for...
  - Saving register values (PUSH, POP instructions)
  - Saving the return address when a procedure is called (CALL instruction) and restoring EIP when a procedure finishes (RET instruction)
  - Passing procedure arguments (Chapter 8)
  - Storing local variables in a procedure (Chapter 8)
- Don't forget:
  - The runtime stack grows downward in memory!
  - ESP register contains the memory address of the top element
    - ESP = Extended Stack Pointer

Runtime Stack – How It's Used



Before call:

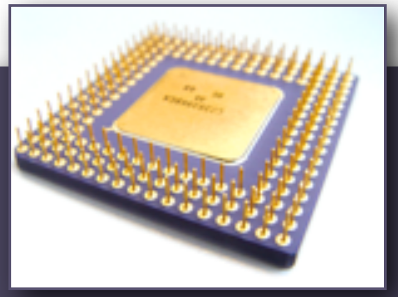



```
main PROC
     call foo
     exit
main ENDP

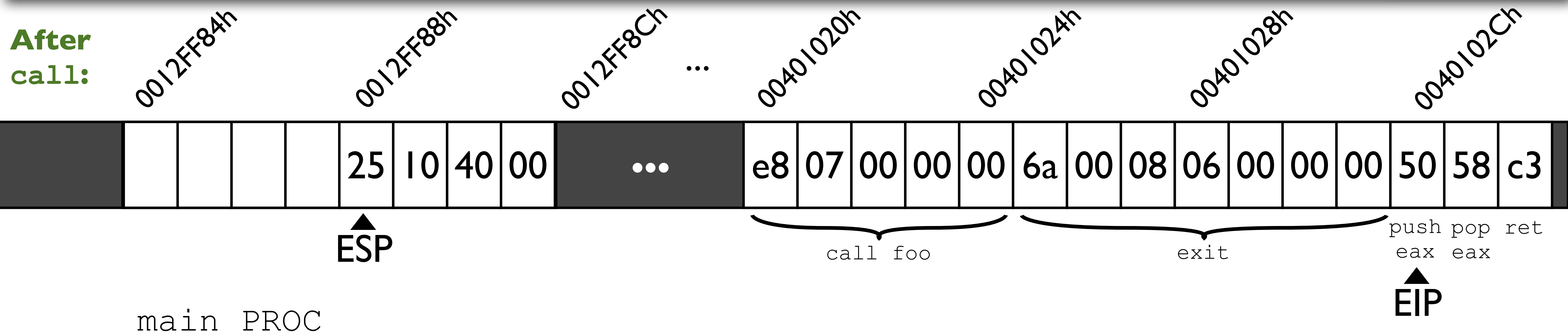
foo PROC
     push eax
     pop eax
     ret
foo ENDP
```

- 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

Runtime Stack – How It's Used



After call:

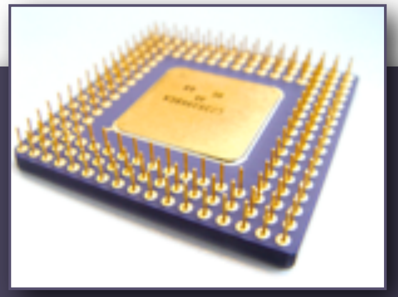



```
main PROC
    call foo
    exit
main ENDP

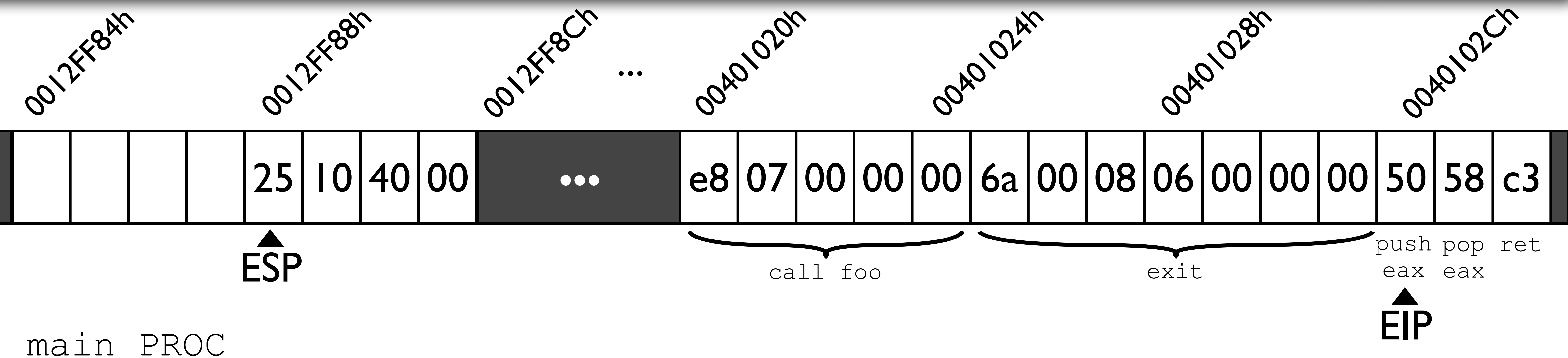
foo PROC
    push eax
    pop eax
    ret
foo ENDP
```

- 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

Runtime Stack – How It's Used



Before push:

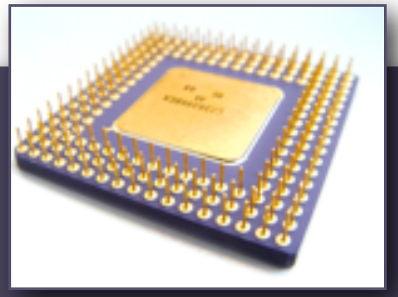



```
main PROC
     call foo
     exit
main ENDP

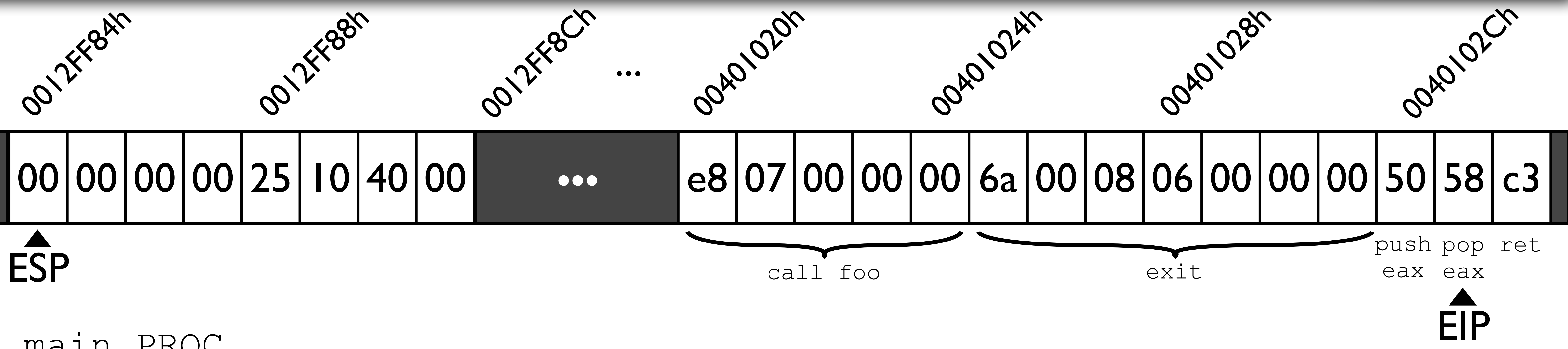
foo PROC
     push eax
     pop eax
     ret
foo ENDP
```

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

Runtime Stack – How It's Used



After push:

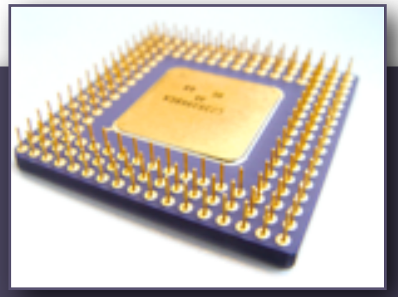


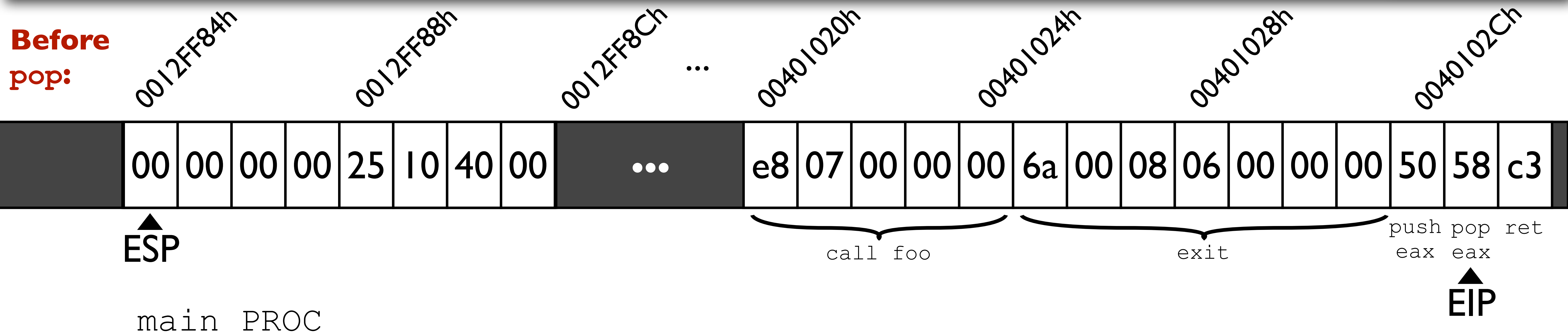

```
main PROC
     call foo
     exit
main ENDP

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)

Runtime Stack – How It's Used



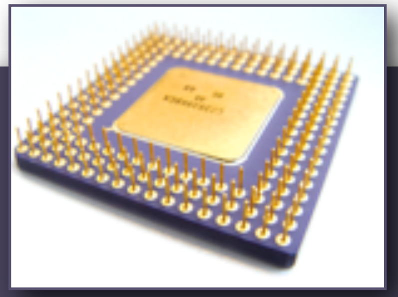


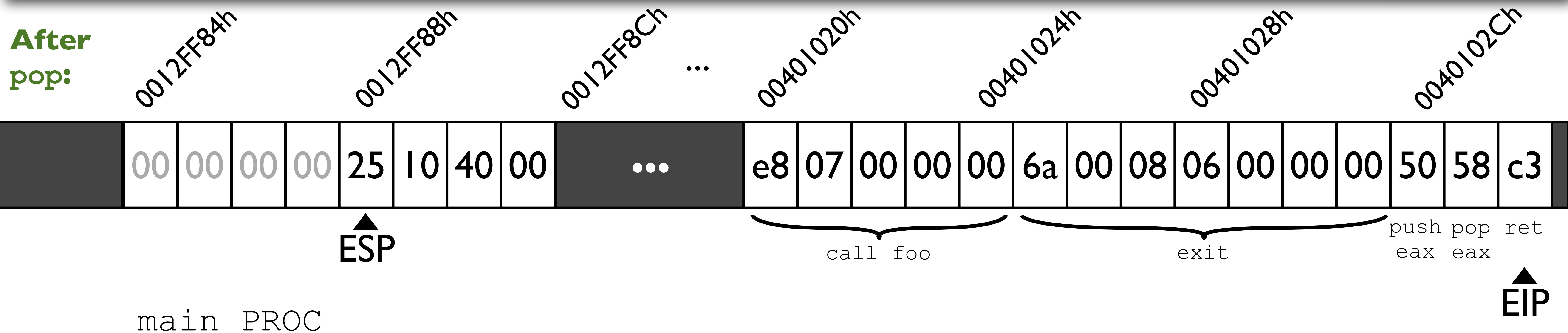

```
main PROC
     call foo
     exit
main ENDP

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

- The pop instruction will
  - Increase ESP by 4

Runtime Stack – How It's Used



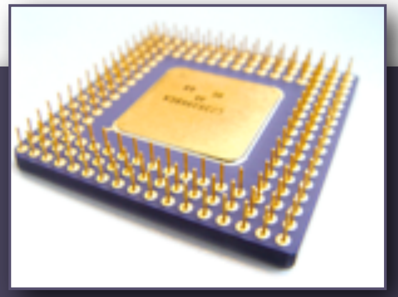



```
main PROC
     call foo
     exit
main ENDP

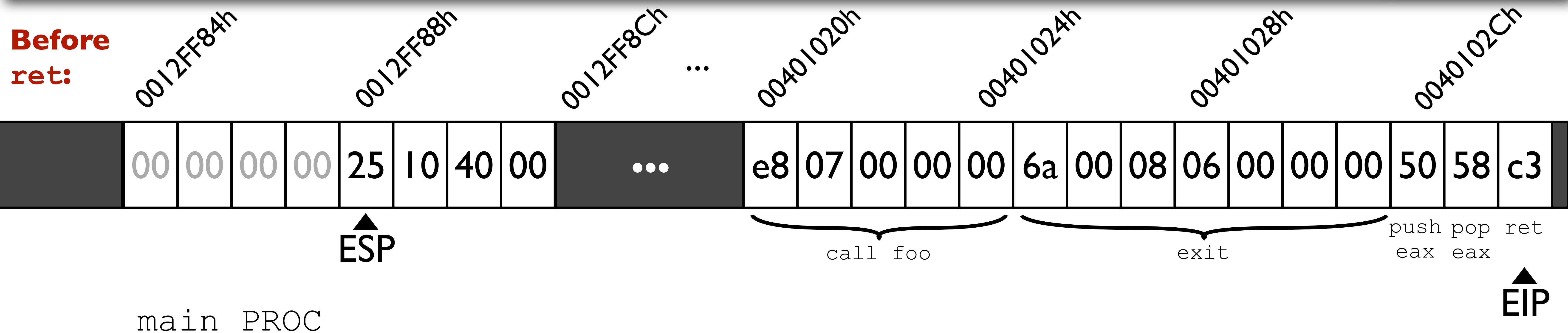
foo PROC
     push eax
     pop eax
     ret
foo ENDP
```

- The pop instruction will
  - Increase ESP by 4

Runtime Stack – How It's Used



Before ret:

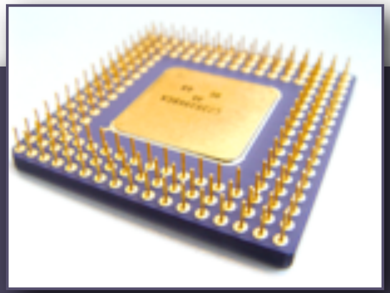


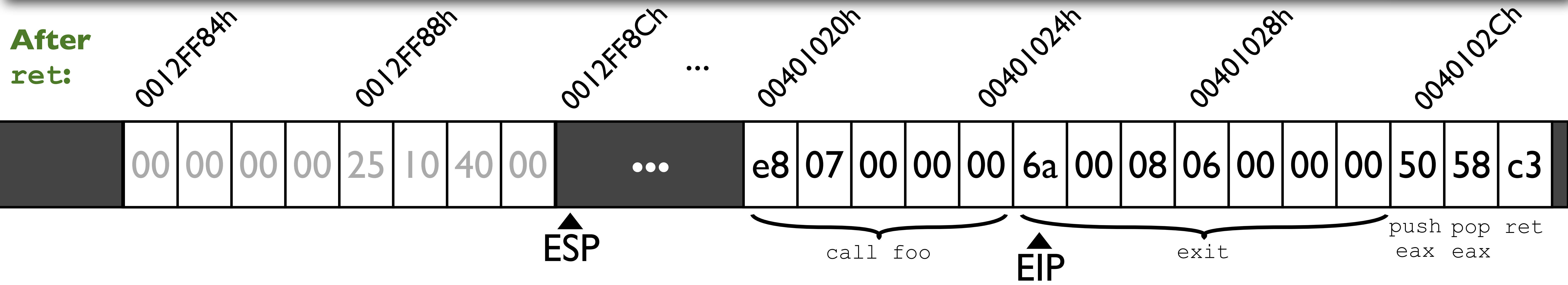

```
main PROC
     call foo
     exit
main ENDP

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)

Runtime Stack – How It's Used






```
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)