

250P: Computer Systems Architecture

Lecture 1: Introduction and x86 Instruction Set

Anton Burtsev
September, 2021

Class details

- Graduate
 - 45 students
- Instructor: Anton Burtsev
- Meeting time: 8:00pm-9:20pm (Mon/Wed)
 - Discussions
- 1 TA
 - Semen
- Web page
 - <https://www.ics.uci.edu/~aburtsev/250P/>

More details

- Several small homeworks
- Midterm
- Final
- Grades are curved
 - Homework: 50%, midterm exam: 25%, final exam: 25% of your grade.
 - You can submit late homework 3 days after the deadline for 60% of your grade

This course

- Book: Hennessy and Patterson's
 - Computer Architecture, A Quantitative Approach, 6th Edition
- Topics
 - Measuring performance/cost/power
 - Instruction level parallelism, dynamic and static
 - Memory hierarchy
 - Multiprocessors
 - Storage systems and networks

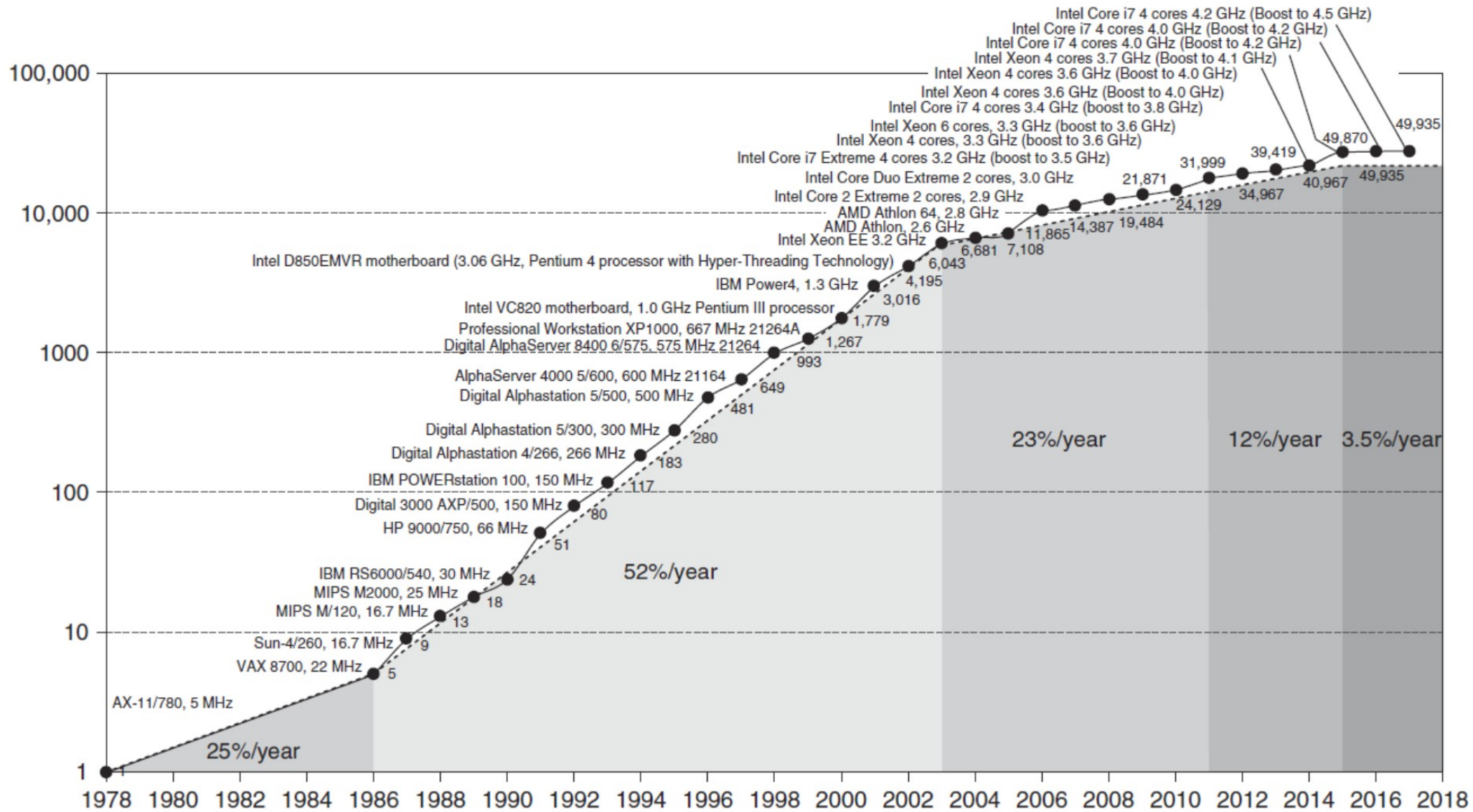
Course organization

- Lectures
 - High level concepts and abstractions
- Reading
 - Hennessy and Patterson
 - Bits of additional notes
- Homeworks

Computer technology

- Performance improvements:
 - Improvements in semiconductor technology
 - Feature size, clock speed
 - Improvements in computer architectures
 - Enabled by high-level language compilers, general operating systems
 - Lead to RISC architectures
- Together have enabled:
 - Lightweight computers
 - Productivity-based managed/interpreted programming languages

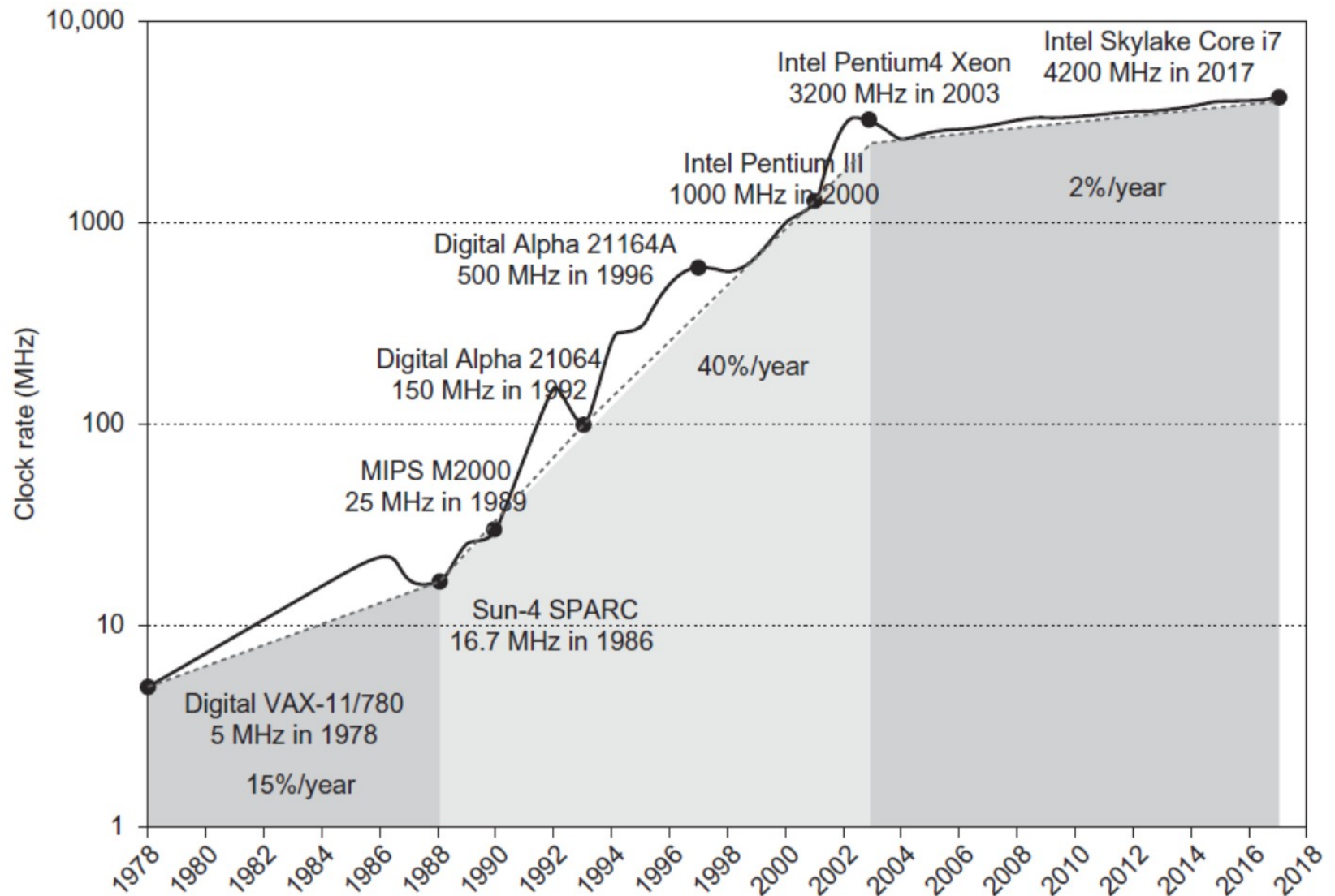
Single processor performance



Points to note

- The 52% growth per year is because of faster clock speeds and architectural innovations (led to 25x higher speed)
- Clock speed increases have dropped to 1% per year in recent years
- The 22% growth includes the parallelization from multiple cores
- End of Dennard scaling
- End of Moore's Law: transistors on a chip double every 18-24 months

Clock speed growth



Current trends in architecture

- Cannot continue to leverage Instruction-Level parallelism (ILP)
 - Single processor performance improvement ended in 2003
 - End of Dennard scaling
 - End of Moore's Law

Why does it matter to you?

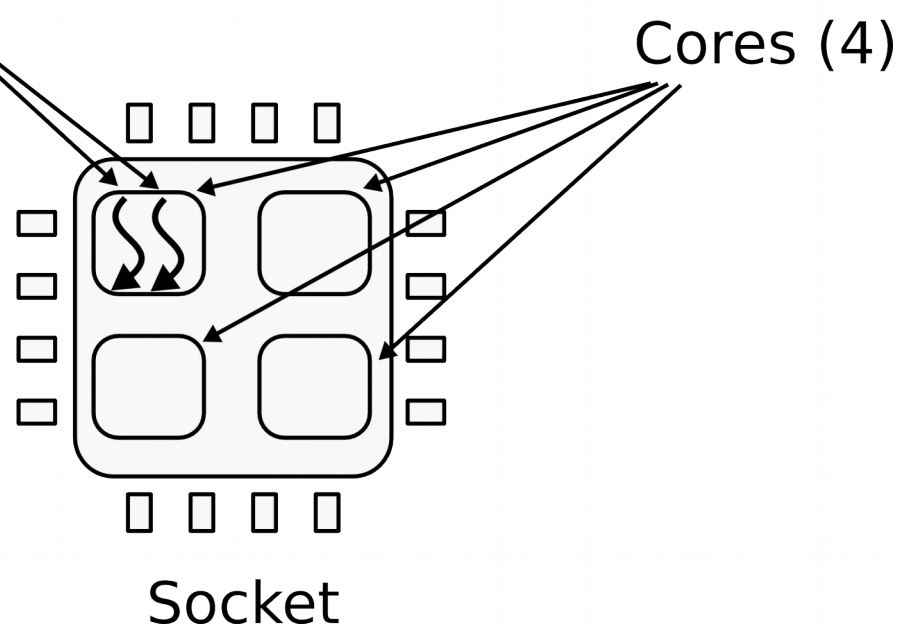
Basics of hardware and x86 instruction set

CPU

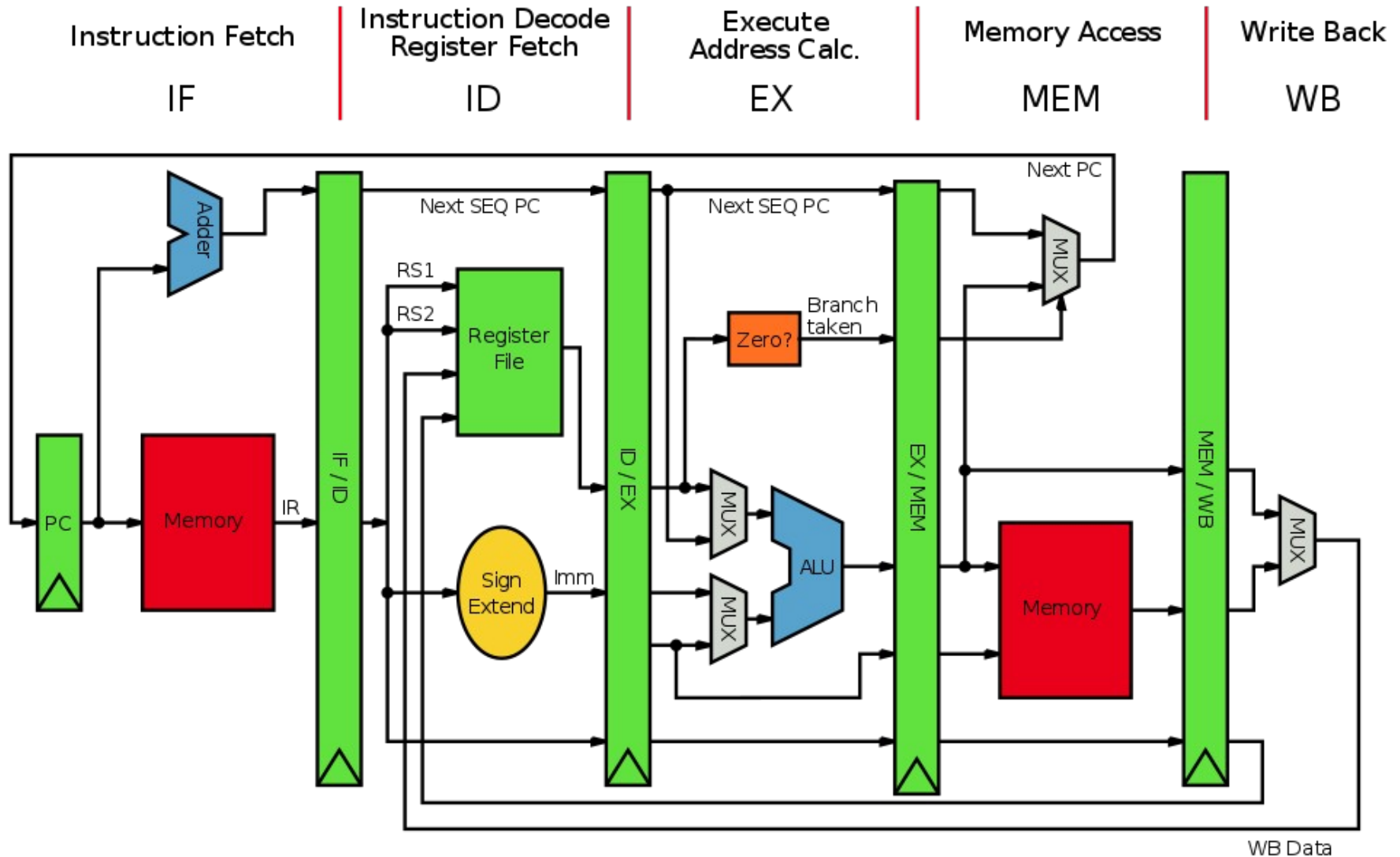
- 1 CPU socket
 - 4 cores
 - 2 logical (HT) threads each



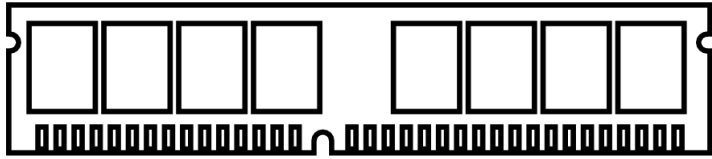
Hyper-Threading
(logical threads)



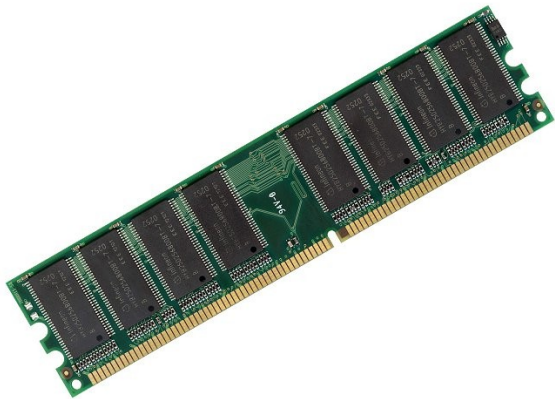
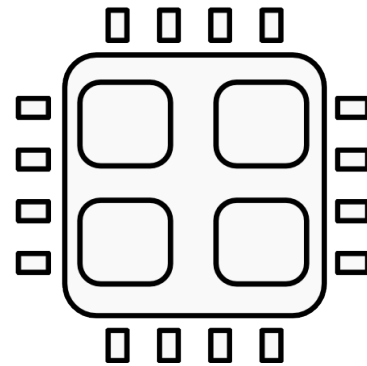
A simple 5-stage pipeline



Memory



Memory
Bus



Memory abstraction

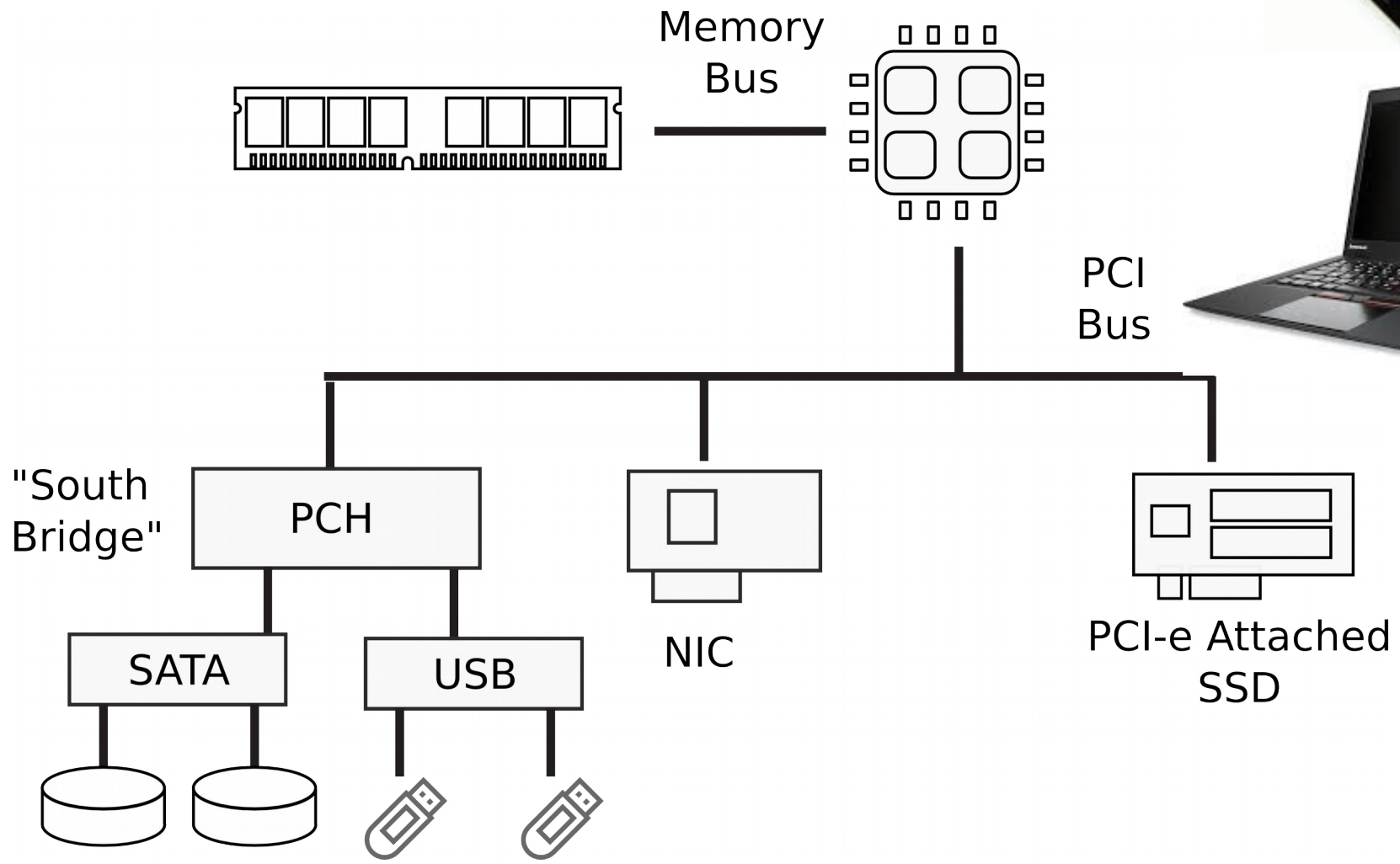
$\text{WRITE}(addr, value) \rightarrow \emptyset$

Store *value* in the storage cell identified by *addr*.

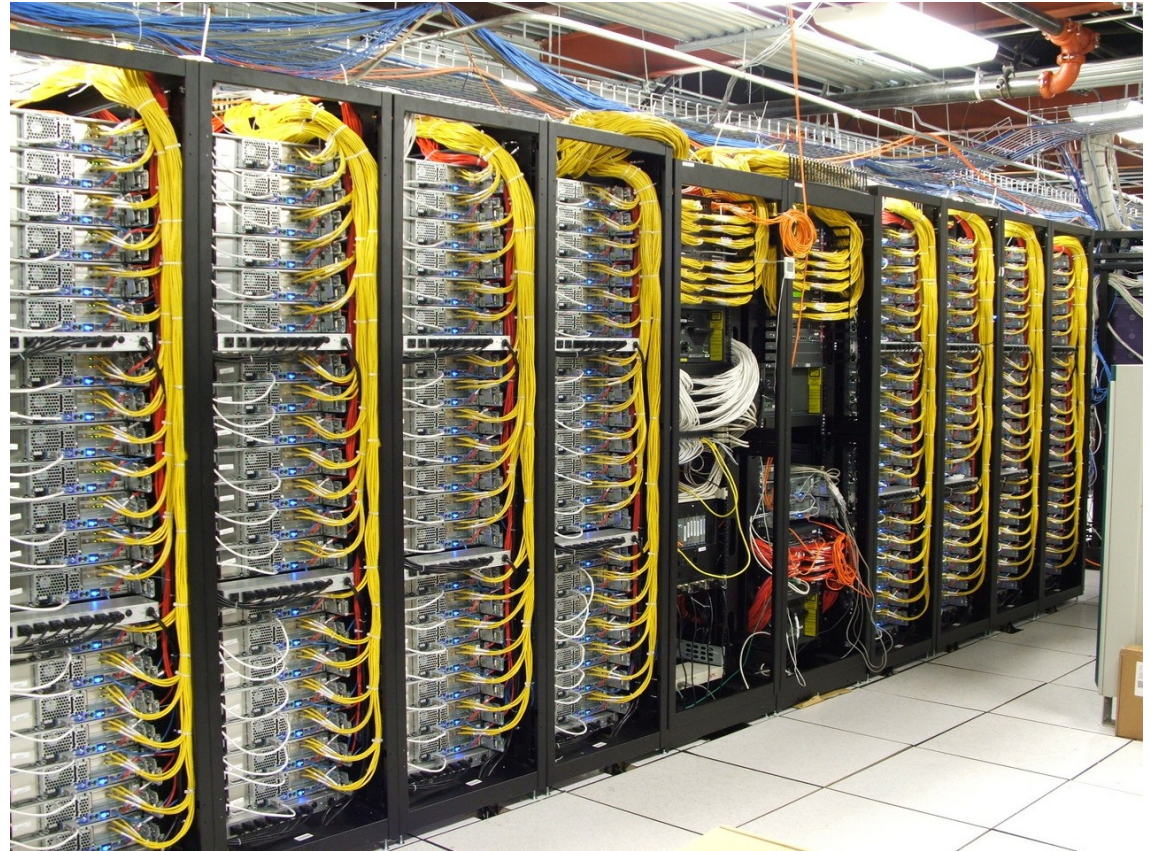
$\text{READ}(addr) \rightarrow value$

Return the *value* argument to the most recent WRITE call referencing *addr*.

I/O Devices



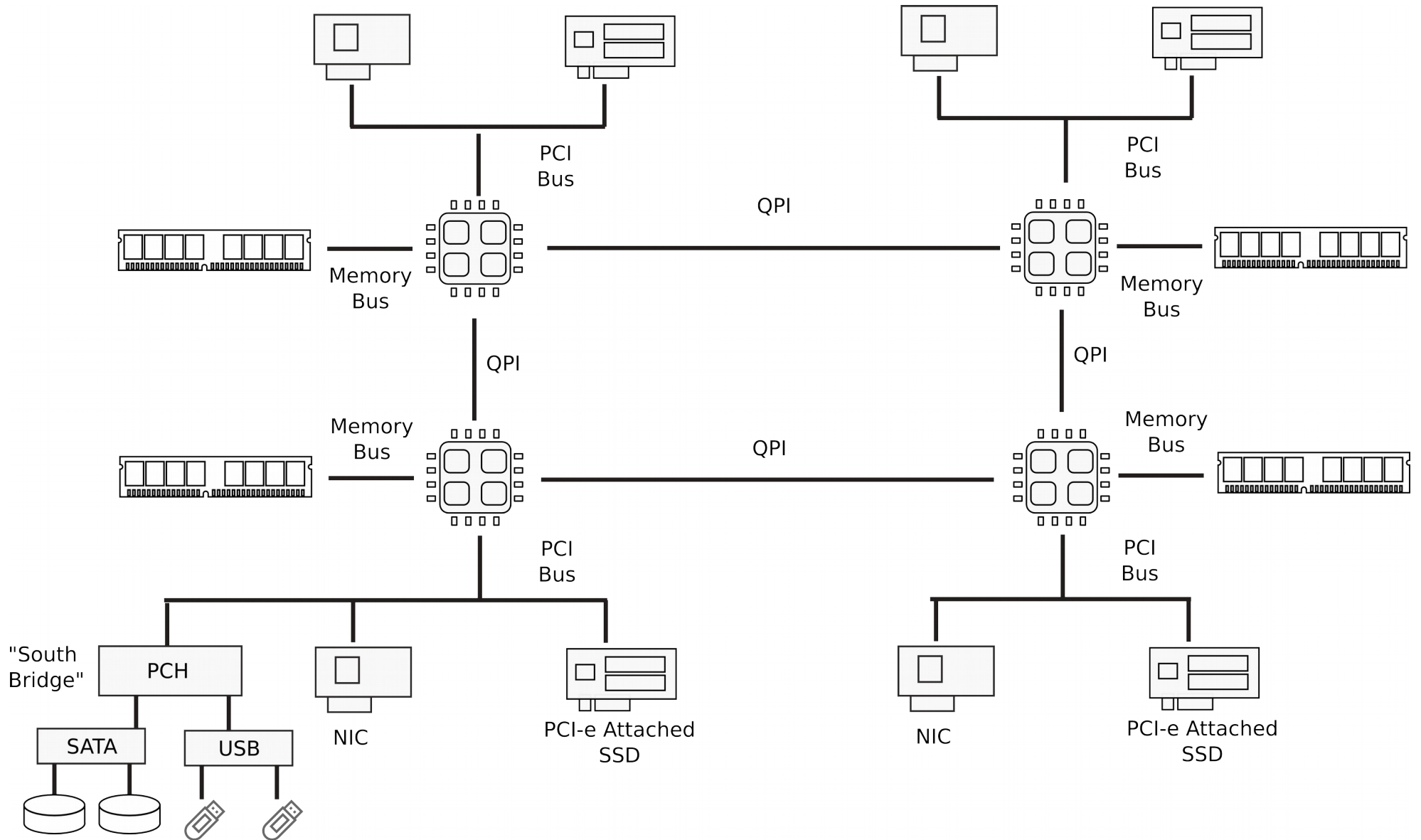
Dell R830 4-socket server



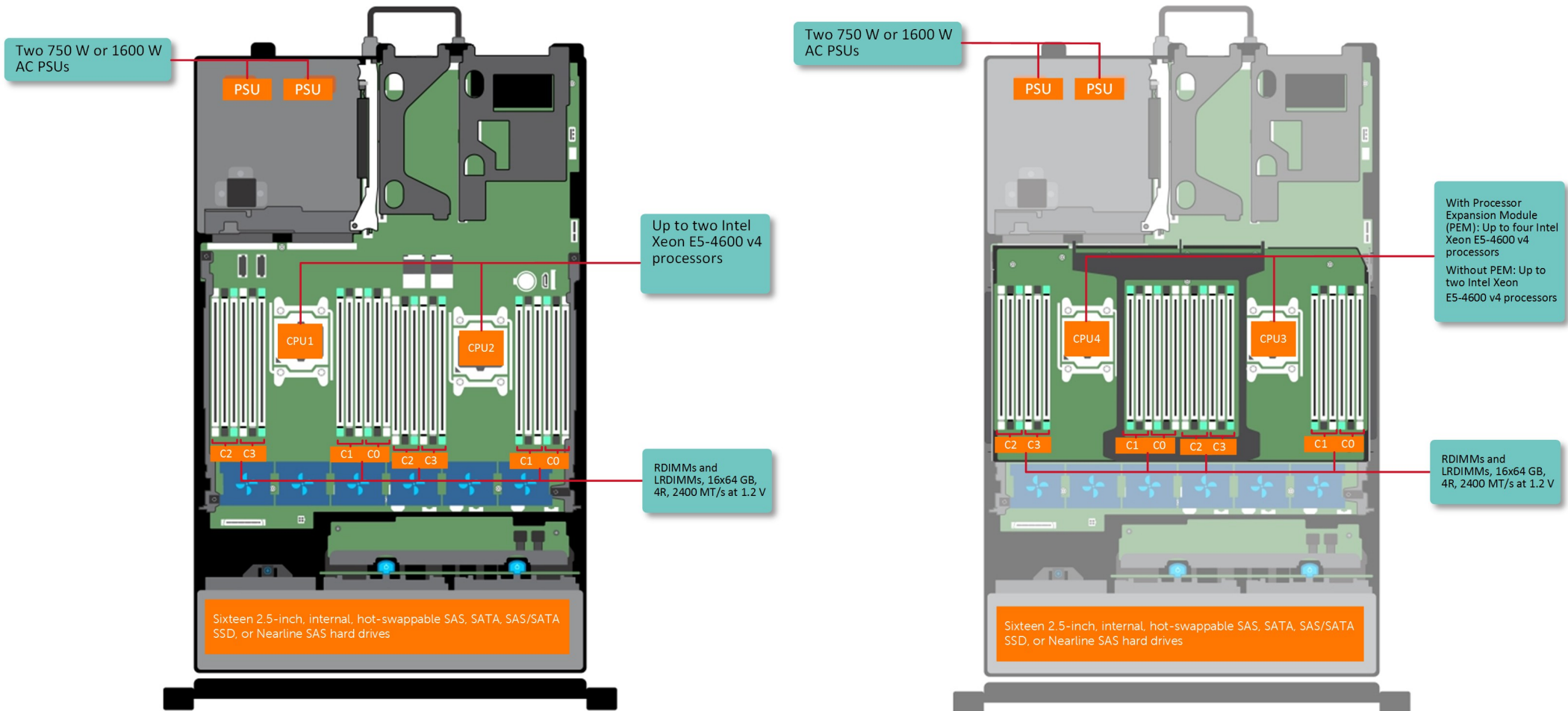
Dell Poweredge R830 System Server with 2 sockets on the main floor and 2 sockets on the expansion

http://www.dell.com/support/manuals/us/en/19/poweredge-r830/r830_om/supported-configurations-for-the-poweredge-r830-system?guid=guid-01303b2b-f884-4435-b4e2-57bec2ce225a&lang=en-us

Multi-socket machines



Dell R830 4-socket server



Dell Poweredge R830 System Server with 2 sockets on the main floor and 2 sockets on the expansion



http://www.dell.com/support/manuals/us/en/19/poweredge-r830/r830_om/supported-configurations-for-the-poweredge-r830-system?guid=guid-01303b2b-f884-4435-b4e2-57bec2ce225a&lang=en-us

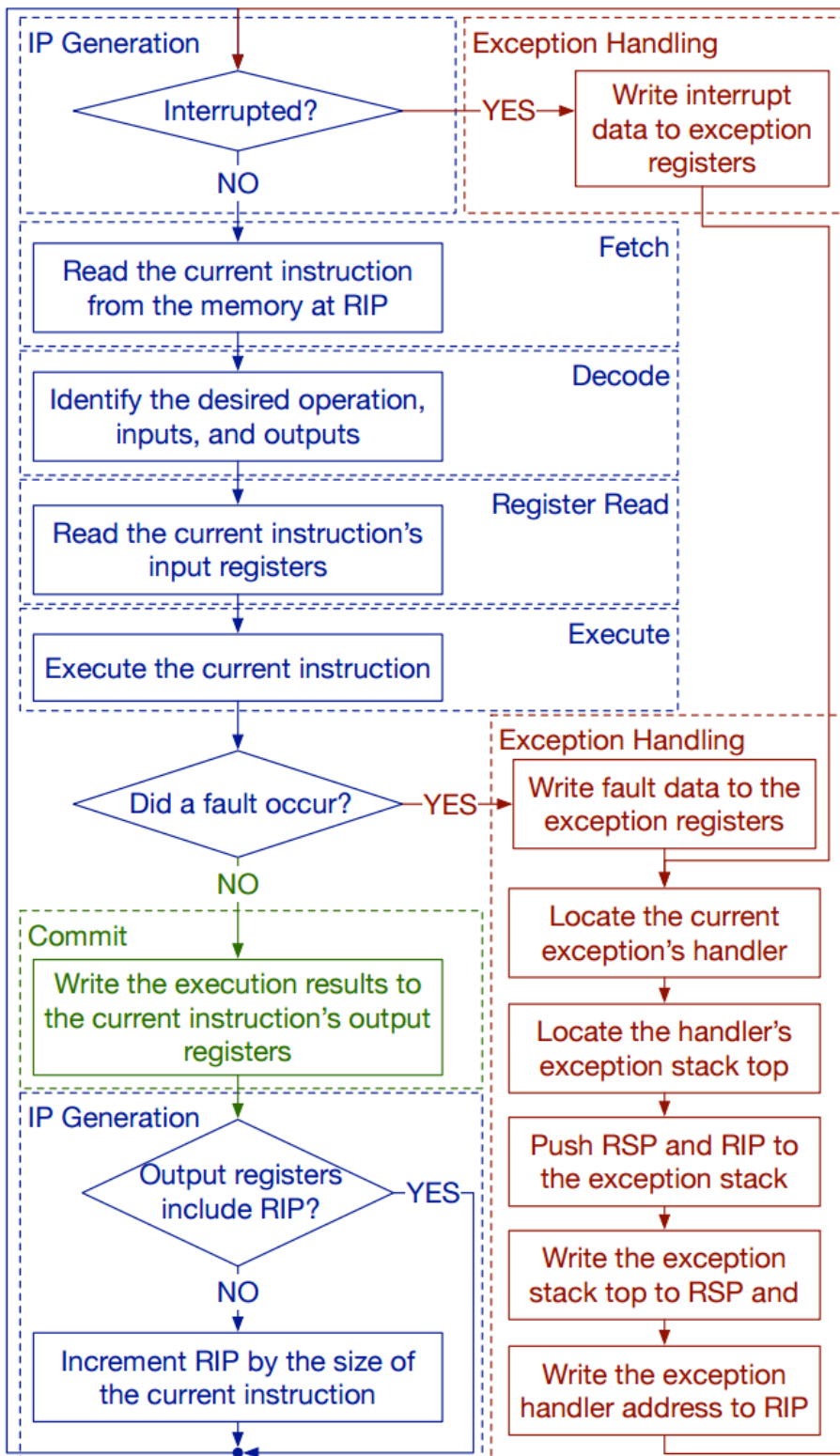
What does CPU do internally?

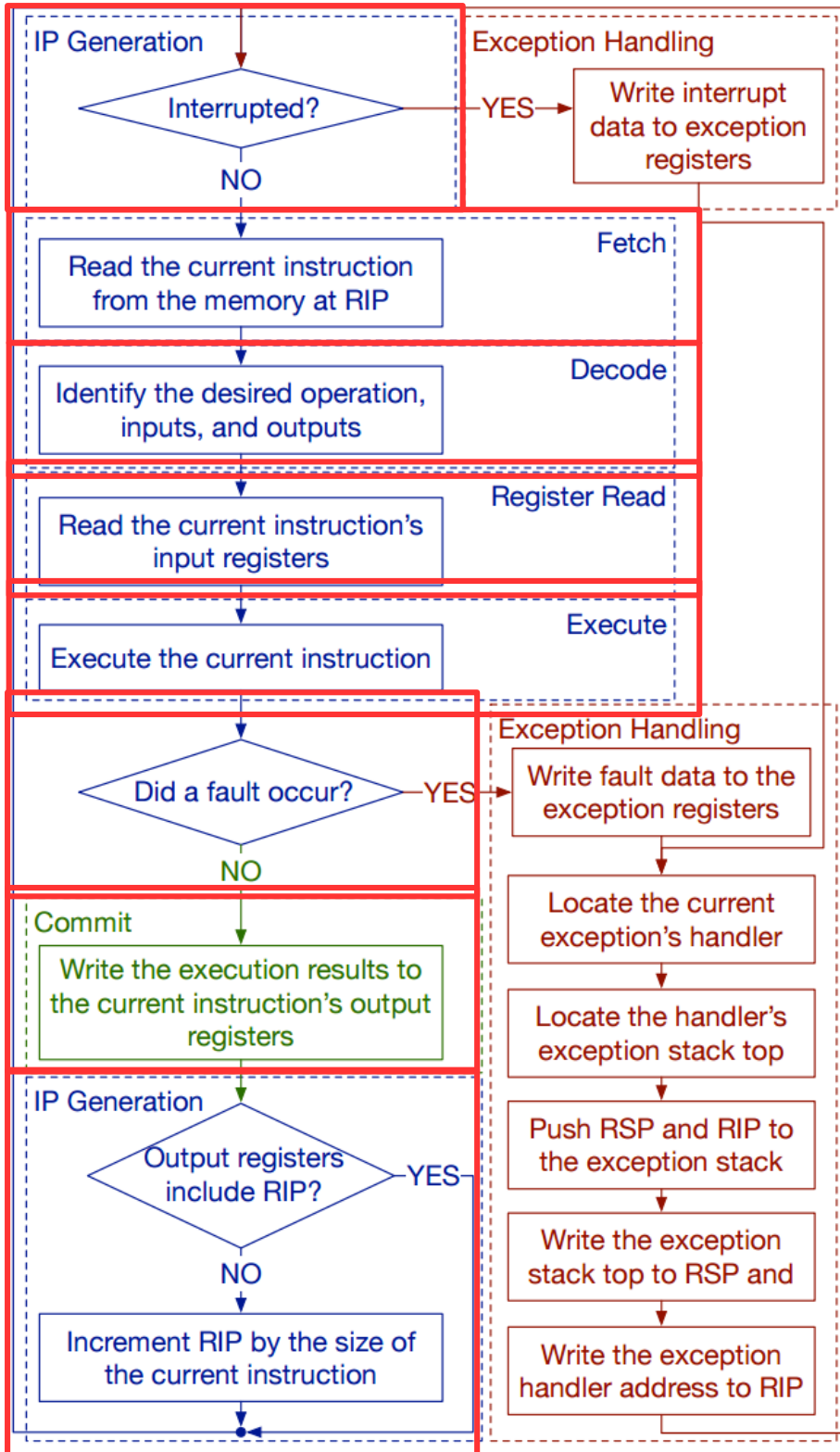
CPU execution loop

- CPU repeatedly reads instructions from memory
- Executes them
- Example

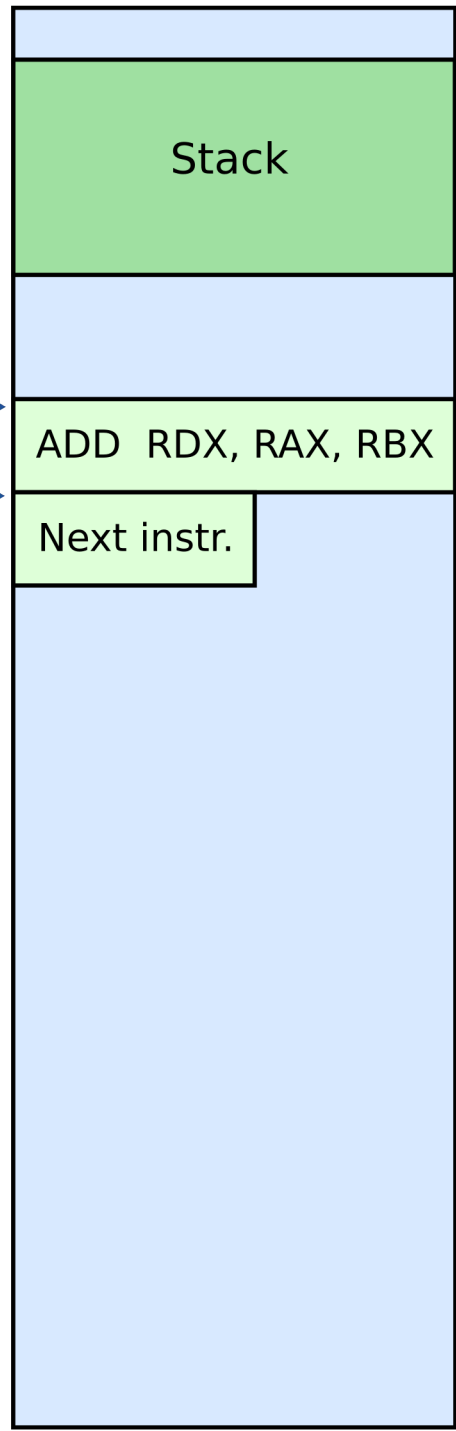
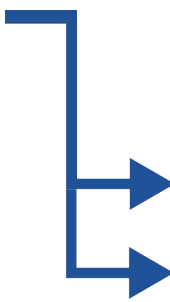
```
ADD EDX, EAX
```

```
// EDX = EAX + EDX
```





RSP
RIP



What are those instructions? (a brief introduction to x86 instruction set)

This part is based on David Evans' x86 Assembly Guide
<http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>

Note

- We'll be talking about **32bit x86** instruction set
 - The version of xv6 we will be using in this class is a 32bit operating system
 - You're welcome to take a look at the 64bit port

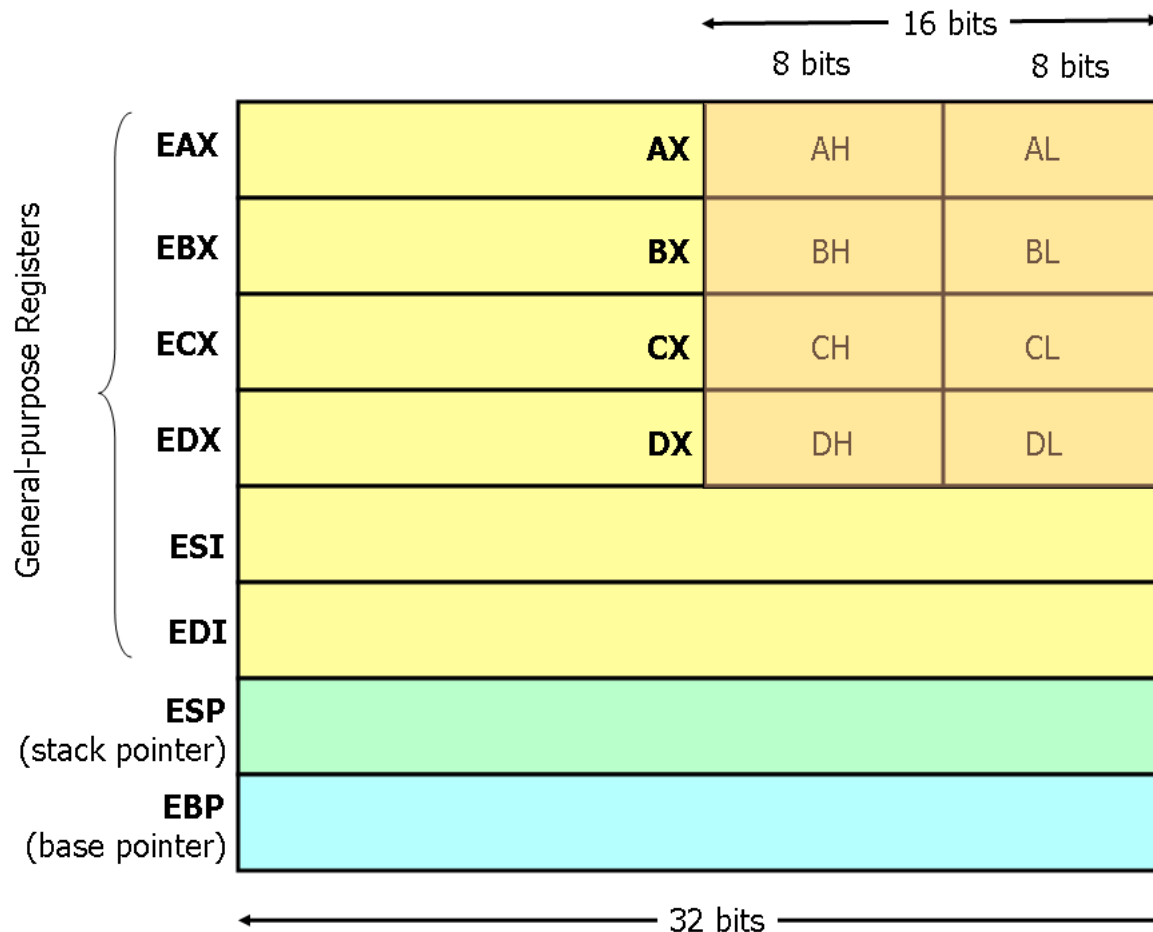
x86 instruction set

- The full x86 instruction set is large and complex
 - But don't worry, the core part is simple
 - The rest are various extensions (often you can guess what they do, or quickly look it up in the manual)

x86 instruction set

- Three main groups
 - Data movement (from memory and between registers)
 - Arithmetic operations (addition, subtraction, etc.)
 - Control flow (jumps, function calls)

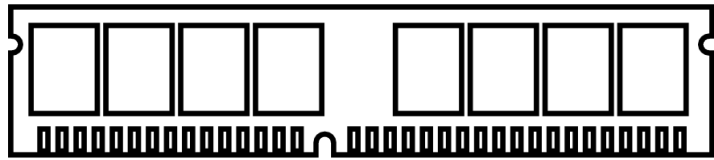
General registers



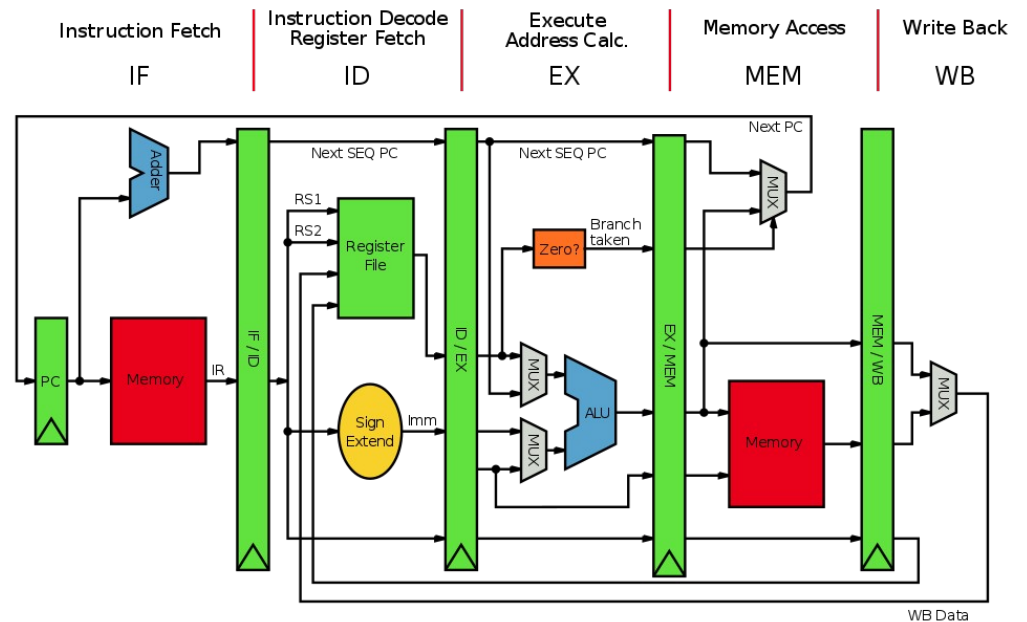
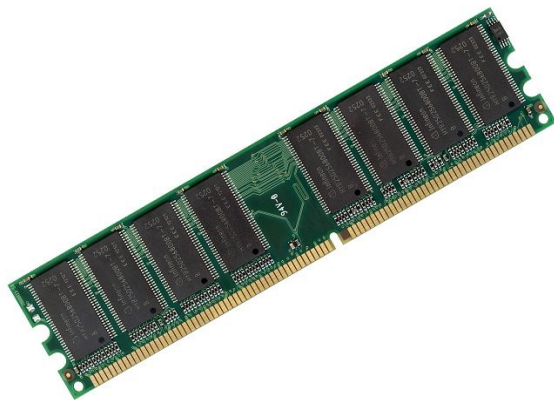
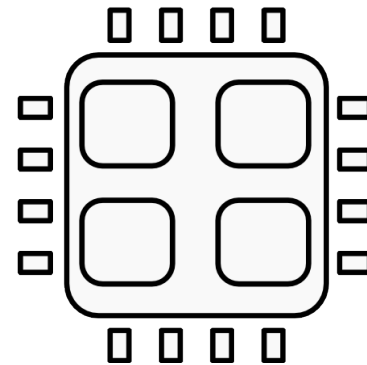
- 8 general registers
 - 32bits each
- Two (**ESP** and **EBP**) have a special role
- Others are more or less general
 - Used in arithmetic instructions, control flow decisions, passing arguments to functions, etc.

BTW, where are these registers?

Registers and Memory



Memory Bus



Data movement instructions

We use the following notation

- We use the following notation
- `<reg32>` Any 32-bit register (EAX,EBX,ECX,EDX,ESI,EDI,ESP, or EBP)
- `<reg16>` Any 16-bit register (AX, BX, CX, or DX)
- `<reg8>` Any 8-bit register (AH, BH, CH, DH, AL, BL, CL, or DL)
- `<reg>` Any register

- `<mem>` A memory address (e.g., `[eax]`, `[var + 4]`, or `dword ptr [eax+ebx]`)
- `<con32>` Any 32-bit constant
- `<con16>` Any 16-bit constant
- `<con8>` Any 8-bit constant
- `<con>` Any 8-, 16-, or 32-bit constant

mov instruction

- Copies the data item referred to by its second operand (i.e. register contents, memory contents, or a constant value) into the location referred to by its first operand (i.e. a register or memory).
 - Register-to-register moves are possible
 - Direct memory-to-memory moves are not
- Syntax

```
mov <reg>, <reg>
```

```
mov <reg>, <mem>
```

```
mov <mem>, <reg>
```

```
mov <reg>, <const>
```

```
mov <mem>, <const>
```

mov examples

```
mov eax, ebx ; copy the value in ebx into eax
```

```
mov byte ptr [var], 5 ; store 5 into the byte at location var
```

```
mov eax, [ebx] ; Move the 4 bytes in memory at the address  
; contained in EBX into EAX
```

```
mov [var], ebx ; Move the contents of EBX into the 4 bytes  
; at memory address var.  
; (Note, var is a 32-bit constant).
```

```
mov eax, [esi-4] ; Move 4 bytes at memory address ESI + (-4)  
; into EAX
```

```
mov [esi+eax], cl ; Move the contents of CL into the byte at  
; address ESI+EAX
```

mov: access to data structures

```
struct point {  
    int x;    // x coordinate (4 bytes)  
    int y;    // y coordinate (4 bytes)  
}  
  
struct point points[128]; // array of 128 points  
  
// load y coordinate of i-th point into y  
int y = points[i].y;  
  
; ebx is address of the points array, eax is i  
mov edx, [ebx + 8*eax + 4] ; Move y of the i-th  
                           ; point into edx
```

lea load effective address

- The lea instruction places the address specified by its second operand into the register specified by its first operand
 - The contents of the memory location are not loaded, only the effective address is computed and placed into the register
 - This is useful for obtaining a pointer into a memory region

lea vs mov access to data structures

- mov

```
// load y coordinate of i-th point into y
```

```
int y = points[i].y;
```

```
; ebx is address of the points array, eax is i
```

```
mov edx, [ebx + 8*eax + 4] ; Move y of the i-th point into edx
```

- lea

```
// load the address of the y coordinate of the i-th point into p
```

```
int *p = &points[i].y;
```

```
; ebx is address of the points array, eax is i
```

```
lea esi, [ebx + 8*eax + 4] ; Move address of y of the i-th point into esi
```

lea is often used instead of add

- Compared to add, lea can
 - perform addition with either two or three operands
 - store the result in any register; not just one of the source operands.
 - Examples

```
LEA EAX, [ EAX + EBX + 1234567 ]
```

```
    ; EAX = EAX + EBX + 1234567 (three operands)
```

```
LEA EAX, [ EBX + ECX ] ; EAX = EBX + ECX
```

```
    ; Add without overriding EBX or ECX with the result
```

```
LEA EAX, [ EBX + N * EBX ] ; multiplication by constant
```

```
    ; (limited set, by 2, 3, 4, 5, 8, and 9 since N is
```

```
    ; limited to 1,2,4, and 8).
```

Arithmetic and logic instructions

add Integer addition

- The add instruction adds together its two operands, storing the result in its first operand
 - Both operands may be registers
 - At most one operand may be a memory location
- Syntax

add <reg>, <reg>

add <reg>, <mem>

add <mem>, <reg>

add <reg>, <con>

add <mem>, <con>

add examples

```
add eax, 10 ; EAX ← EAX + 10
```

```
add BYTE PTR [var], 10 ; add 10 to the  
                        ; single byte stored at  
                        ; memory address var
```

sub Integer subtraction

- The sub instruction stores in the value of its first operand the result of subtracting the value of its second operand from the value of its first operand.
- Examples

```
sub al, ah ; AL ← AL - AH
```

```
sub eax, 216 ; subtract 216 from the value  
; stored in EAX
```

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
- 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
```

and, or, xor Bitwise logical and, or, and exclusive or

- These instructions perform the specified logical operation (logical bitwise and, or, and exclusive or, respectively) on their operands, placing the result in the first operand location
- Examples

```
and eax, 0fH ; clear all but the last 4  
              ; bits of EAX.
```

```
xor edx, edx ; set the contents of EDX to  
              ; zero.
```

shl, shr shift left, shift right

- These instructions shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros
- The shifted operand can be shifted up to 31 places. The number of bits to shift is specified by the second operand, which can be either an 8-bit constant or the register CL
 - In either case, shifts counts of greater than 31 are performed modulo 32.
- Examples

```
shl eax, 1 ; Multiply the value of EAX by 2
           ; (if the most significant bit is 0)
```

```
shr ebx, cl ; Store in EBX the floor of result of dividing
            ; the value of EBX by 2^n
            ; where n is the value in CL.
```

More instructions... (similar)

- Multiplication **imul**

```
imul eax, [var] ; multiply the contents of EAX by the  
                ; 32-bit contents of the memory location  
                ; var. Store the result in EAX.
```

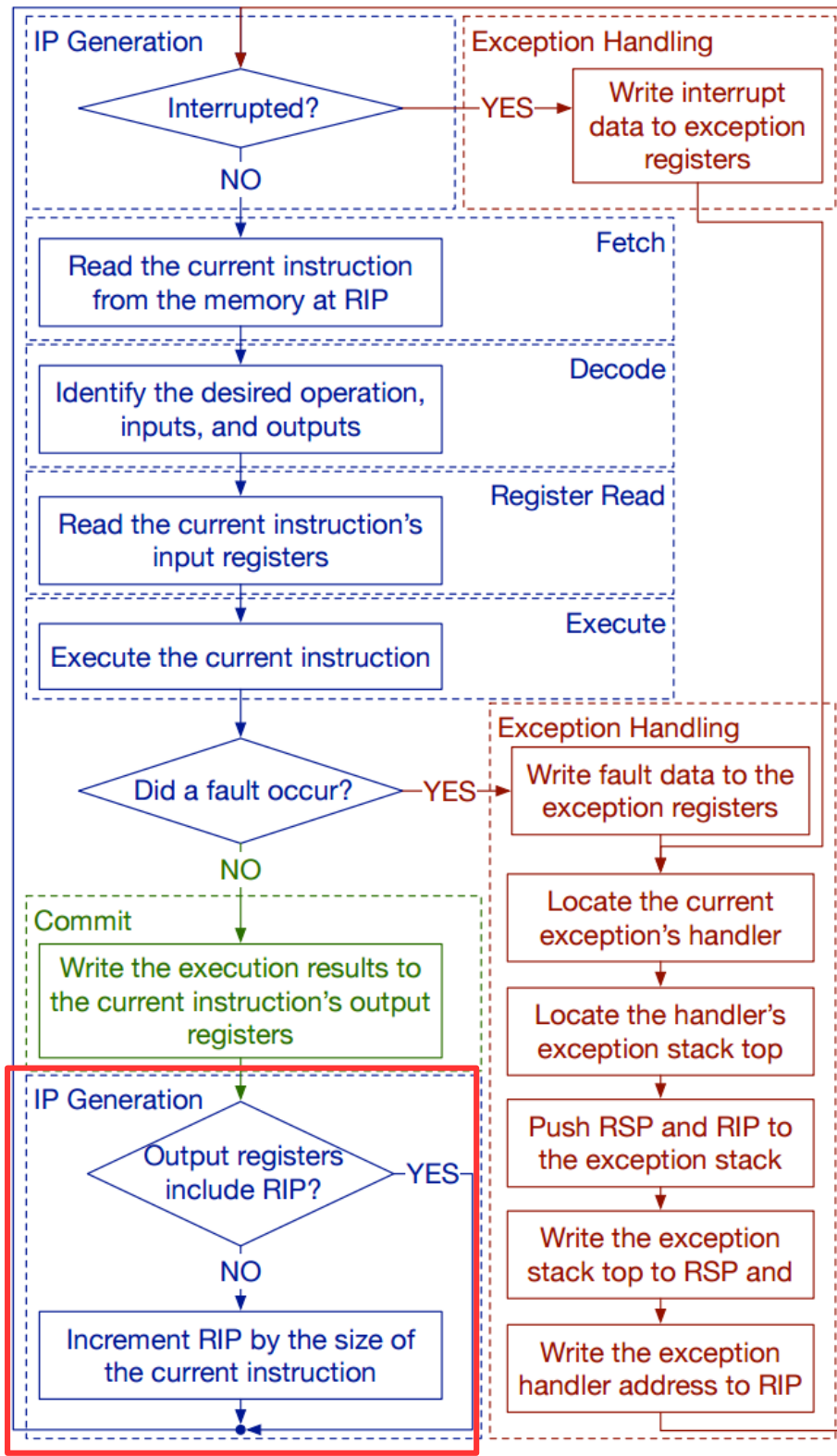
```
imul esi, edi, 25 ; ESI ← EDI * 25
```

- Division **idiv**
- **not** - bitwise logical not (flips all bits)
- **neg** - negation

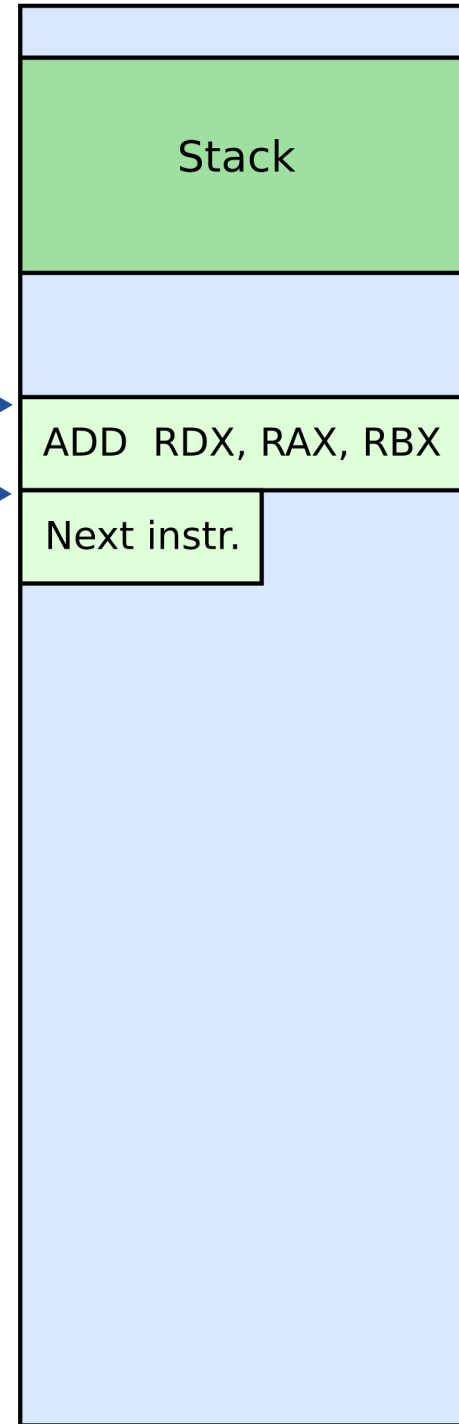
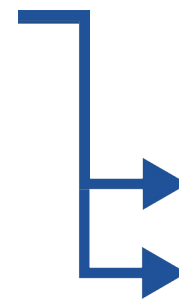
```
neg eax ; EAX ← - EAX
```

This is enough to do arithmetic

Control flow instructions



RSP
RIP



EIP instruction pointer

- EIP is a 32bit value indicating the location in memory where the current instruction starts (i.e., memory address of the instruction)
- EIP cannot be changed directly
 - Normally, it increments to point to the next instruction in memory
 - But it can be updated implicitly by provided control flow instructions

Labels

- `<label>` refers to a labeled location in the program text (code).
- Labels can be inserted anywhere in x86 assembly code text by entering a label name followed by a colon
- Examples

```
    mov esi, [ebp+8]
```

```
begin: xor ecx, ecx
```

```
    mov eax, [esi]
```

jump: jump

- Transfers program control flow to the instruction at the memory location indicated by the operand.
- Syntax

```
jmp <label>
```

- Example

```
begin:  xor ecx, ecx
```

```
...
```

```
jmp begin ; jump to instruction labeled  
          ; begin
```

jcondition: conditional jump

- Jumps only if a condition is true
 - The status of a set of condition codes that are stored in a special register (**EFLAGS**)
 - **EFLAGS** stores information about the last arithmetic operation performed for example,
 - Bit 6 of **EFLAGS** indicates if the last result was **zero**
 - Bit 7 indicates if the last result was **negative**
- Based on these bits, different conditional jumps can be performed
 - For example, the **jz** instruction performs a jump to the specified operand label if the result of the last arithmetic operation was **zero**
 - Otherwise, control proceeds to the next instruction in sequence

Conditional jumps

- Most conditional jump follow the comparison instruction (cmp, we'll cover it below)

- Syntax

`je <label>` (jump when equal)

`jne <label>` (jump when not equal)

`jz <label>` (jump when last result was zero)

`jg <label>` (jump when greater than)

`jge <label>` (jump when greater than or equal to)

`jl <label>` (jump when less than)

`jle <label>` (jump when less than or equal to)

- Example: if `EAX` is less than or equal to `EBX`, jump to the label `done`. Otherwise, continue to the next instruction

```
cmp eax, ebx
```

```
jle done
```

cmp: compare

- Compare the values of the two specified operands, setting the condition codes in EFLAGS
 - 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: if the 4 bytes stored at location `var` are equal to the 4-byte integer constant `10`, jump to the location labeled `loop`.

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

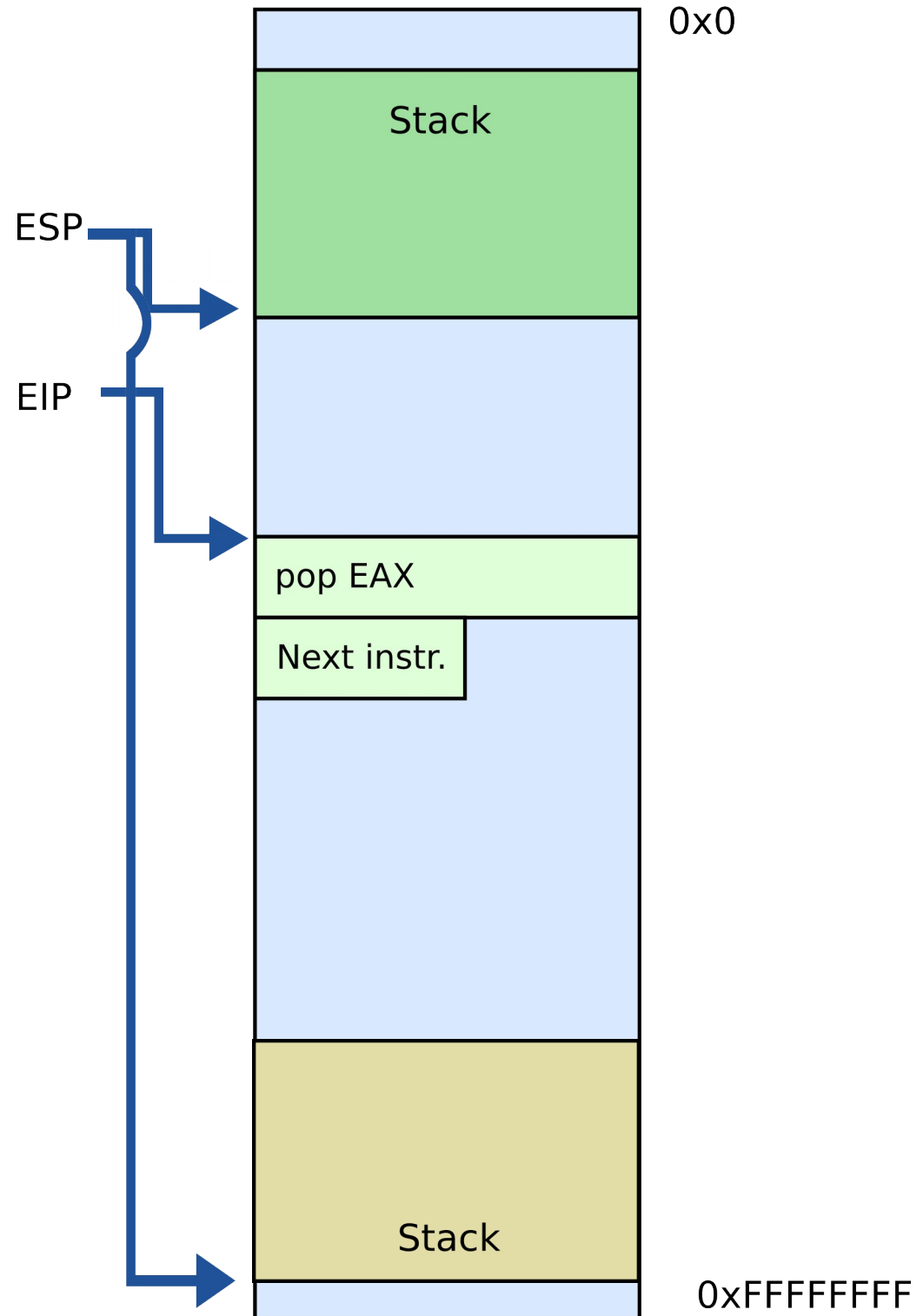
```
jeq loop
```

Stack and procedure calls

What is stack?

Stack

- It's just a region of memory
 - Pointed by a special register ESP
- You can change ESP
 - Get a new stack



Why do we need stack?

Calling functions

```
// some code...  
foo();  
// more code..
```

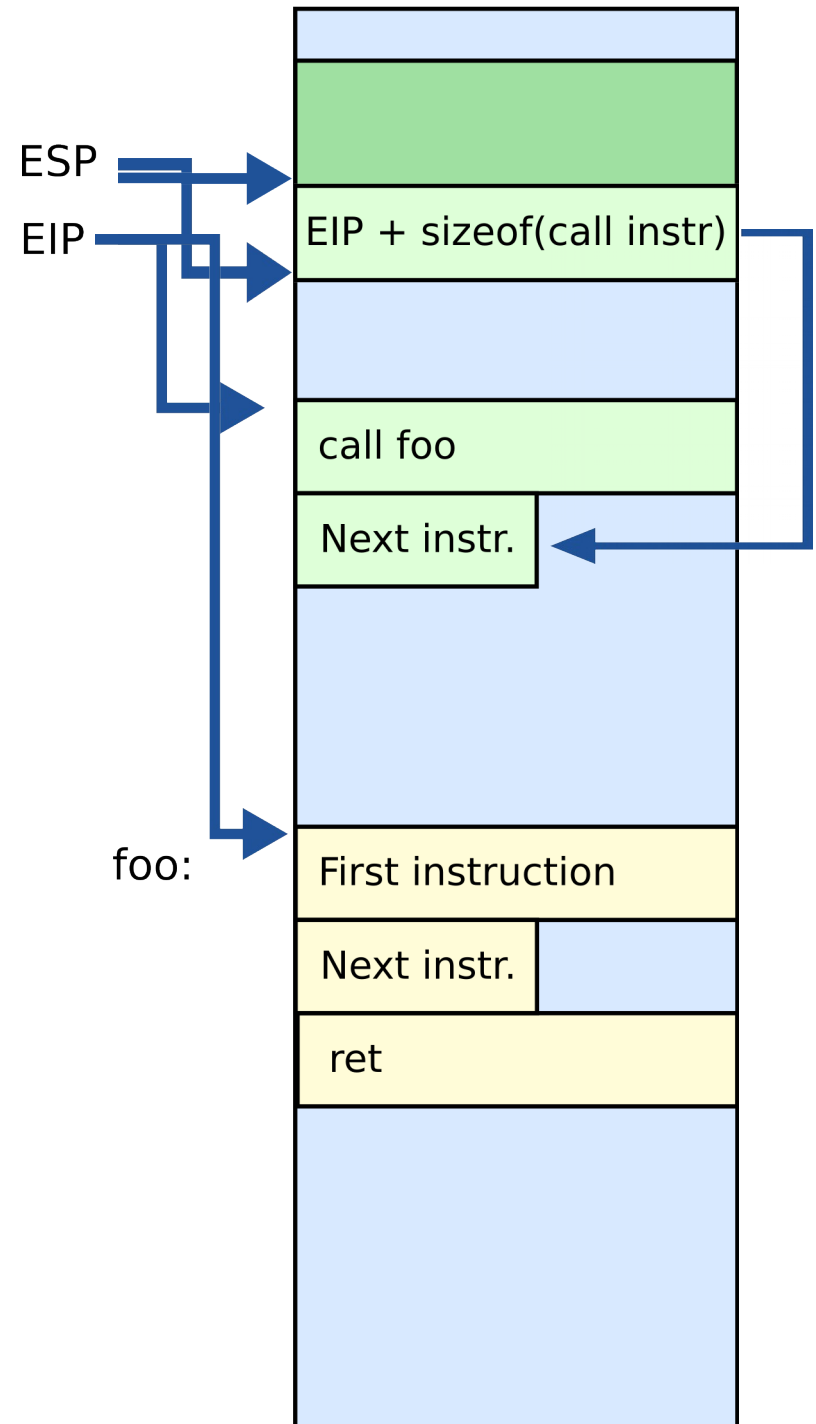
- Stack contains information for **how to return** from a subroutine
 - i.e., from foo()

- Functions can be called from different places in the program

```
if (a == 0) {  
    foo();  
    ...  
}  
else {  
    foo();  
    ...  
}
```

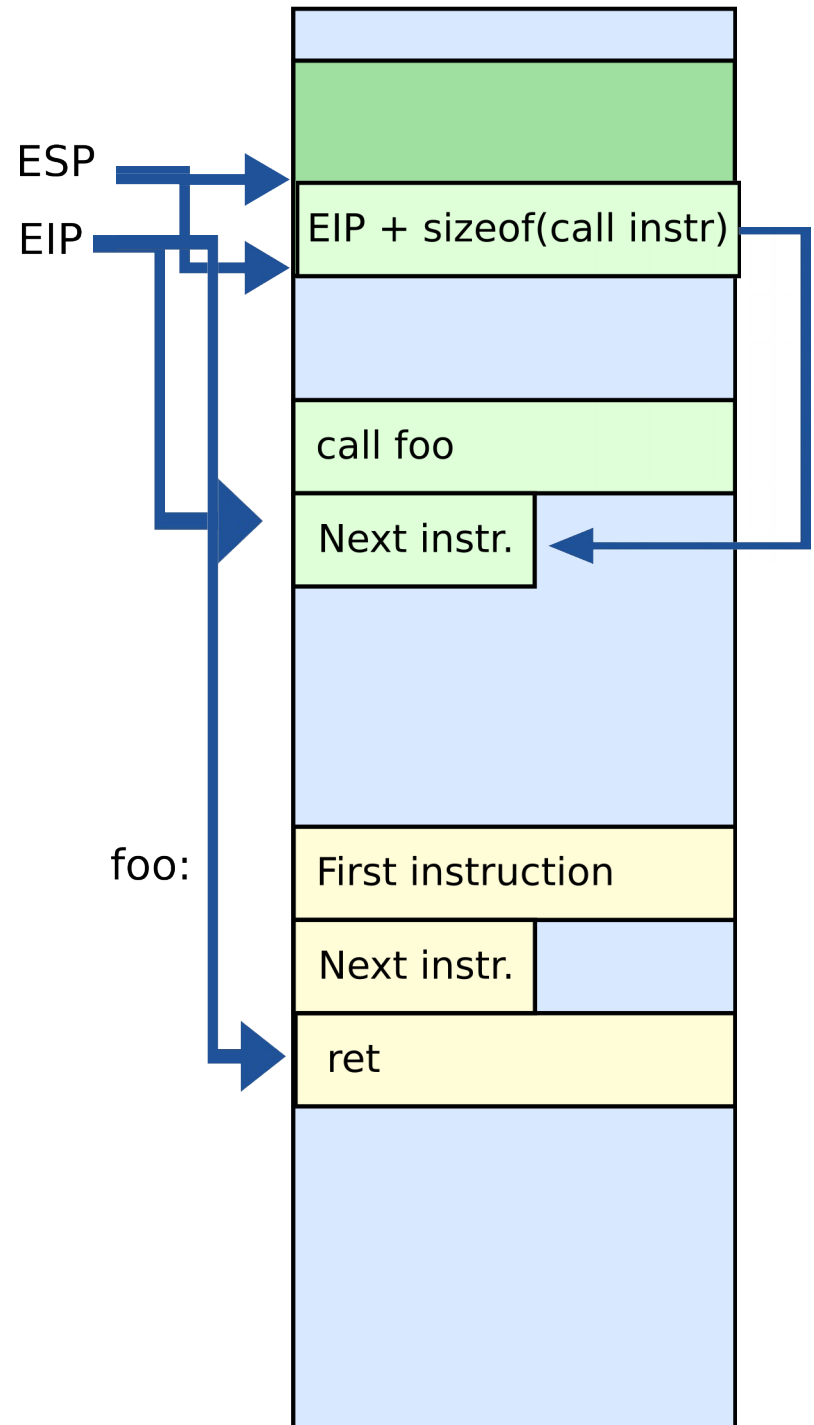
Stack

- Main purpose:
 - Store the return address for the current procedure
 - **Caller** pushes return address on the stack
 - **Callee** pops it and jumps



Stack

- Main purpose:
 - Store the return address for the current procedure
 - **Caller** pushes return address on the stack
 - **Callee** pops it and jumps



Call/return

- **CALL** instruction

- Makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack

```
push eip + sizeof(CALL); save return  
                                ; address
```

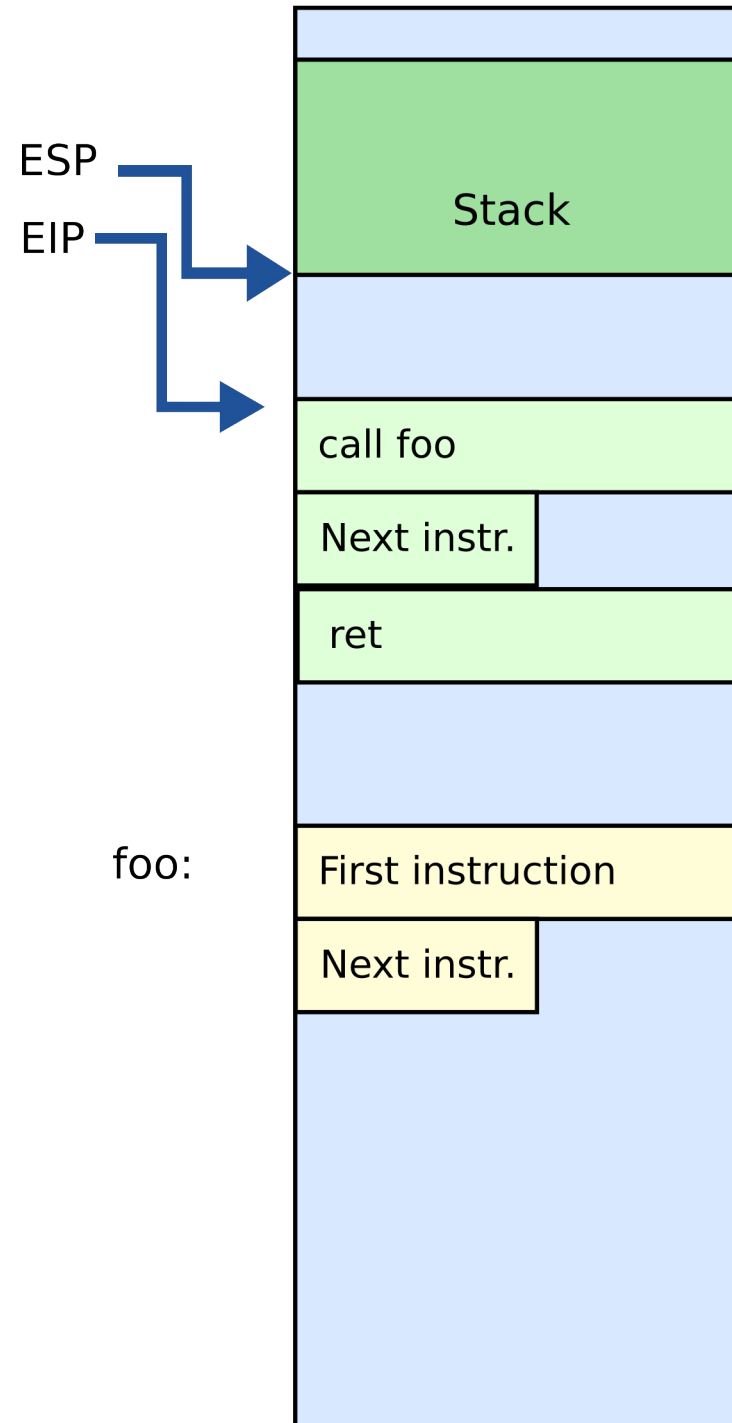
```
jmp _my_function
```

- **RET** instruction

- Pops off an address and jumps to that address

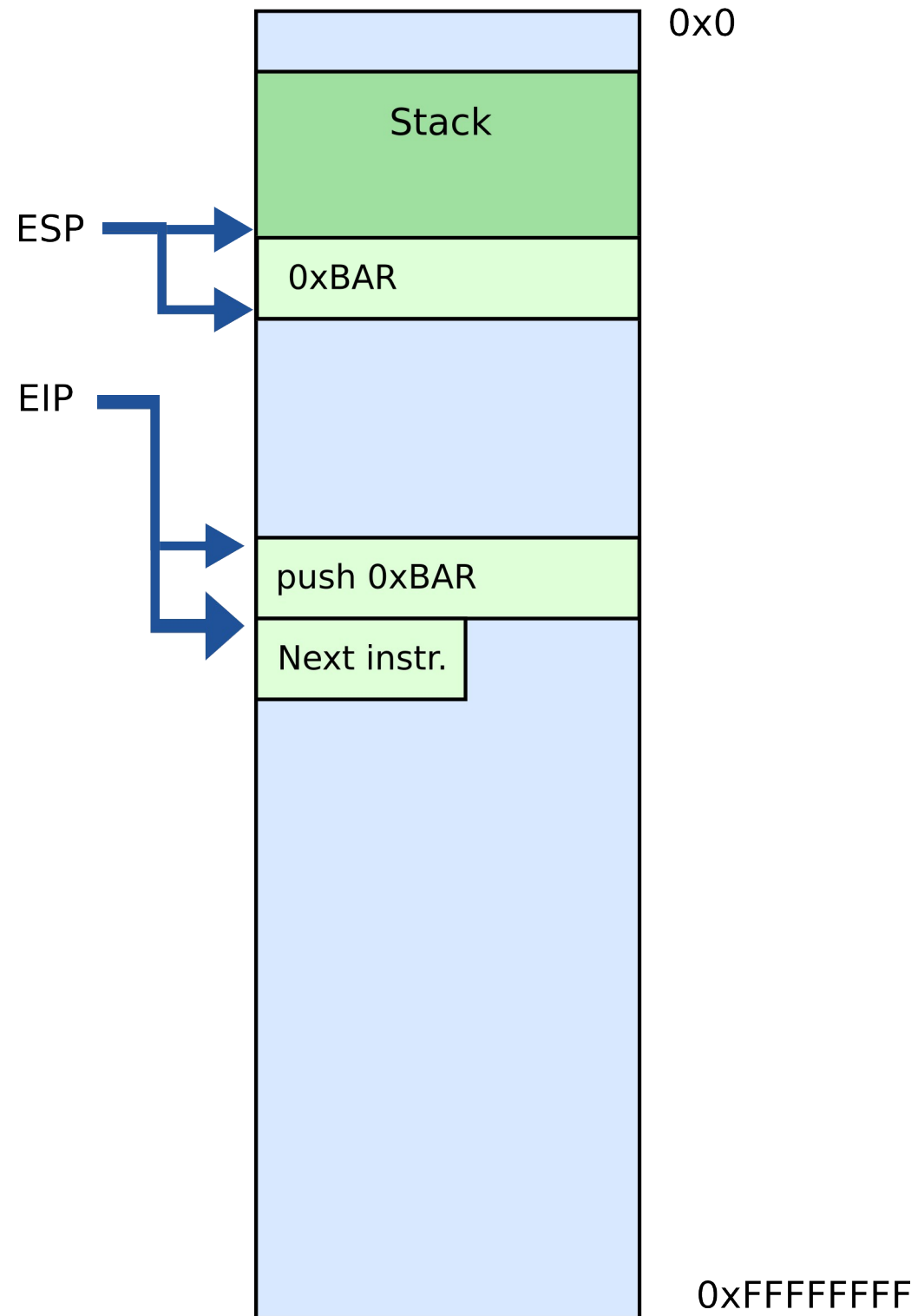
Stack

- Other uses:
 - Local data storage
 - Parameter passing
 - Evaluation stack
 - Register spill



Manipulating stack

- **ESP** register
 - Contains the memory address of the topmost element in the stack
- **PUSH** instruction
 - `push 0xBAR`
 - Subtract 4 from ESP
 - Insert data on the stack

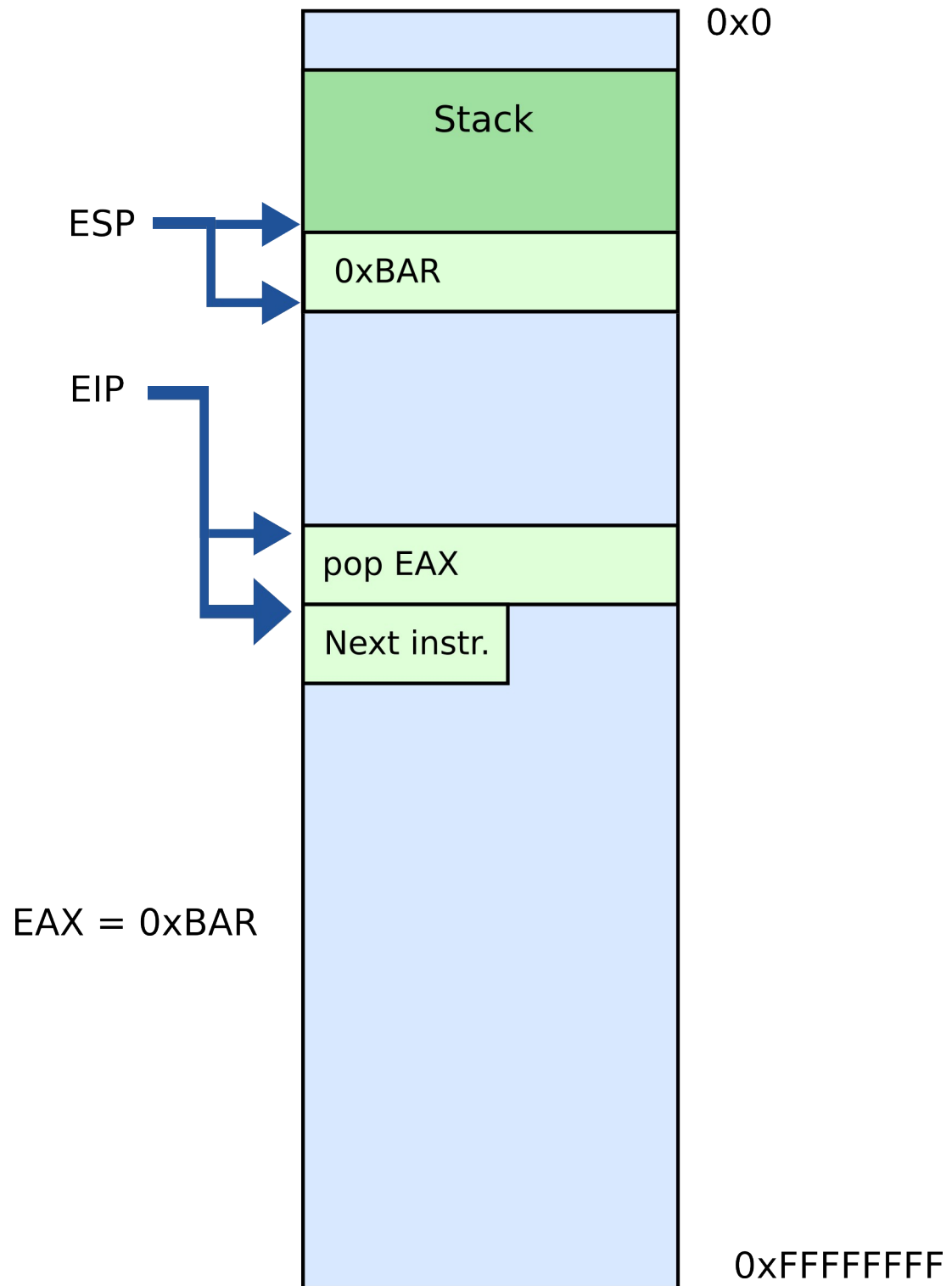


Manipulating stack

- **POP** instruction

`pop EAX`

- Removes data from the stack
- Saves in register or memory
- Adds 4 to ESP



Thank you!