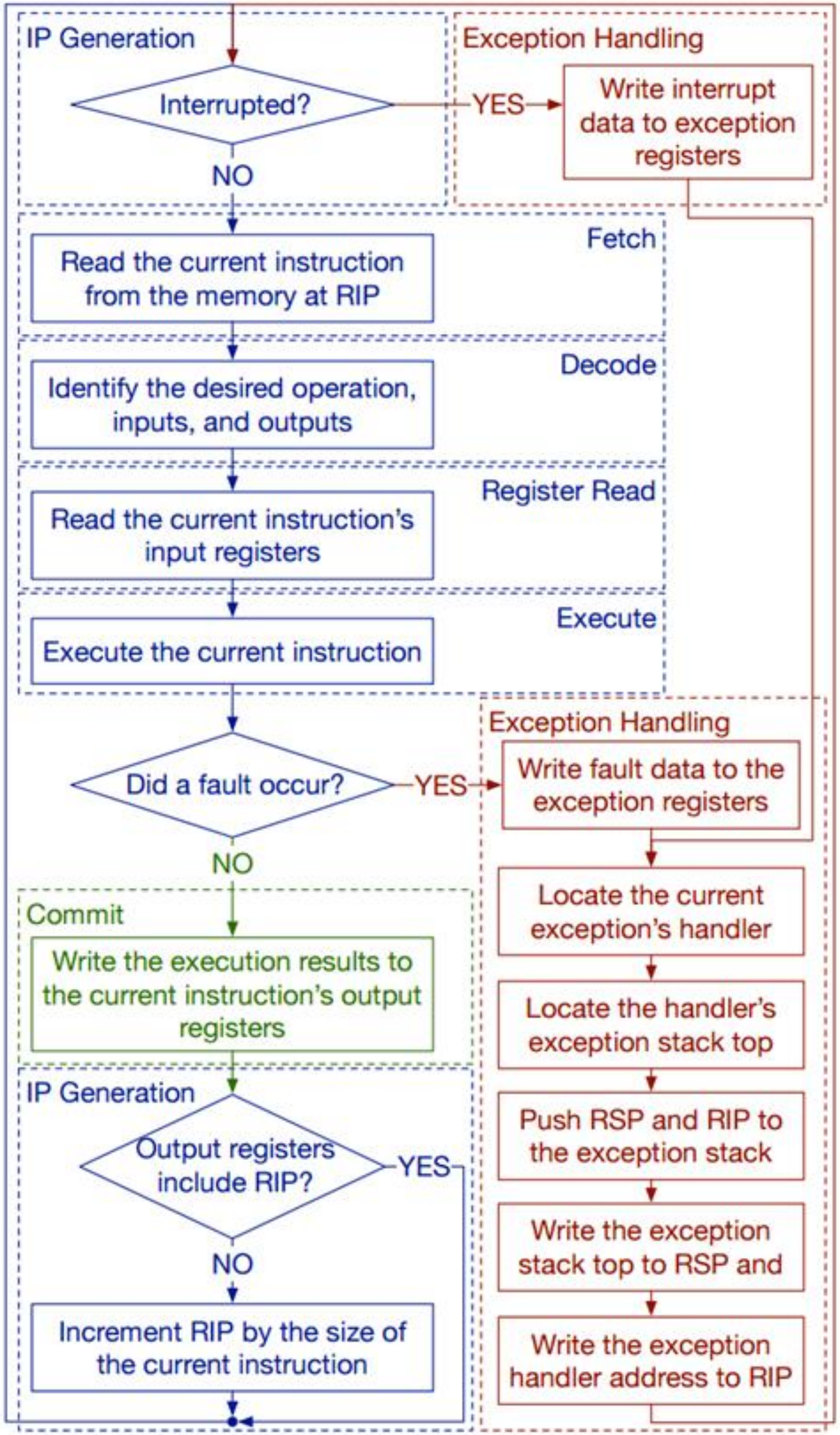
cs5460/6460 Operating Systems
Lecture 03: x86 instruction set

Anton Burtsev

January, 2026

How do CPUs work internally?

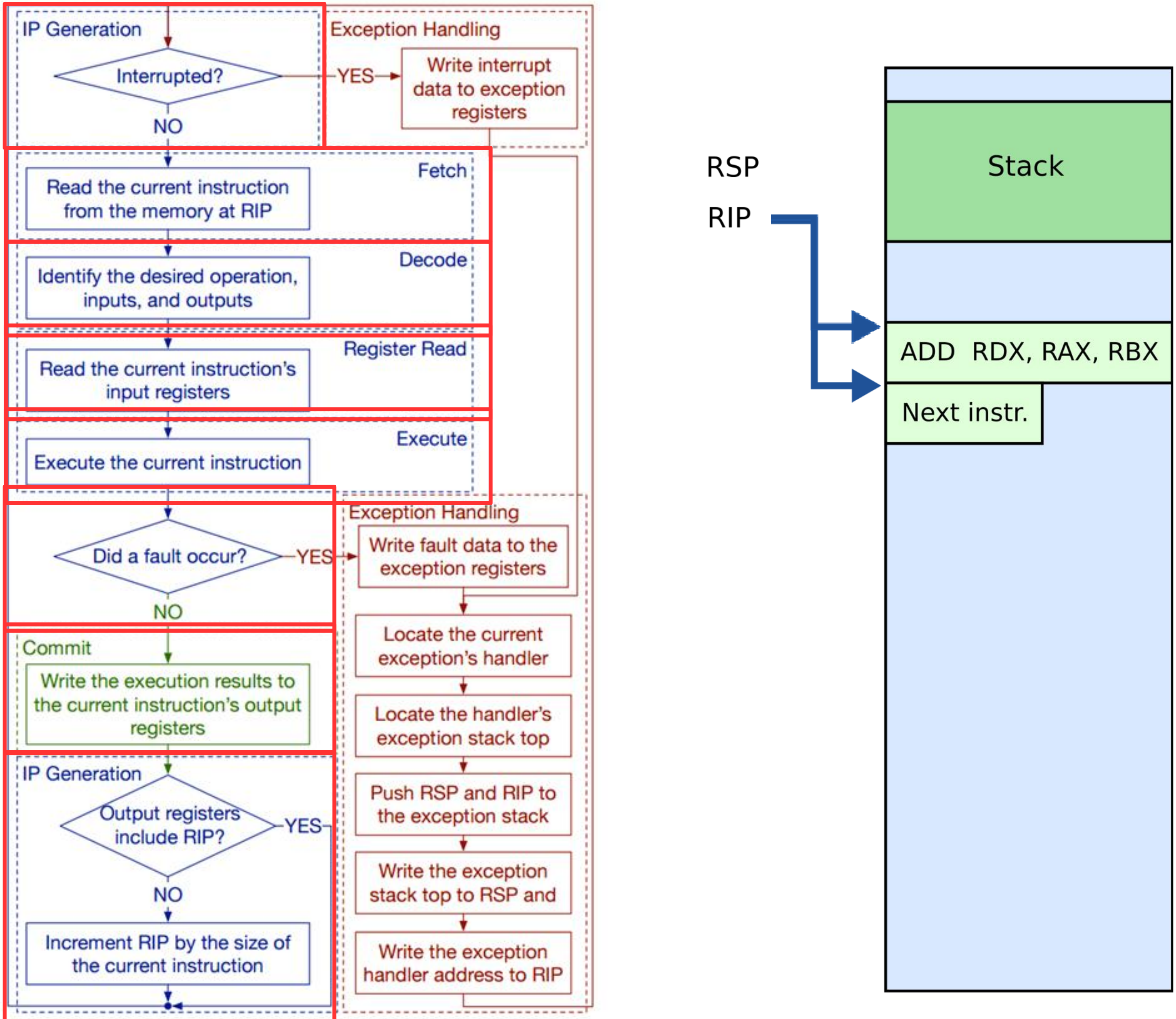


CPU execution loop

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

ADD RDX, RAX

// RDX = RAX + RDX


IP Generation
Interrupted?
YES
NO
Exception Handling
Write interrupt data to exception registers
Fetch
Read the current instruction from the memory at RIP
Decode
Identify the desired operation, inputs, and outputs
Register Read
Read the current instruction's input registers
Execute
Execute the current instruction
Did a fault occur?
YES
NO
Exception Handling
Write fault data to the exception registers
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
Write the exception stack top to RSP and
Write the exception handler address to RIP
Commit
Write the execution results to the current instruction's output registers
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction
RSP
RIP
Stack
ADD RDX, RAX, RBX
Next instr.

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

and it's adaptation for GNU/AT&T asm syntax by the Yale FLINT's group

https://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html

and *Computer Systems: A Programmer's Perspective* by Randal E. Bryant & David R. O'Hallaron (CS:APP)

Note

- We’ll be talking about 64bit x86 instruction set
- The version of xv6 we will be using in this class is a 64bit operating system

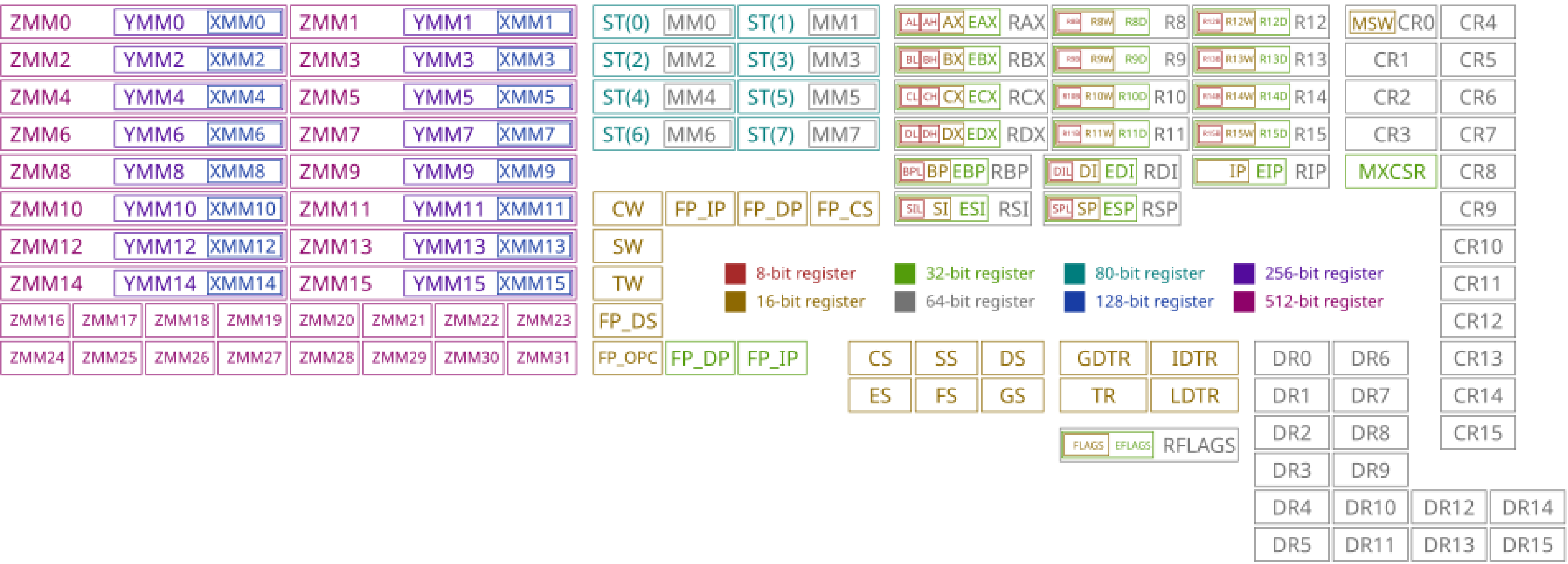
x86 instruction set

- 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)

x86 registers



https://en.wikipedia.org/wiki/X86#x86_registers

Figure 3.2 Integer registers. The low-order portions of all 16 registers can be accessed as byte, word (16-bit), double word (32-bit), and quad word (64-bit) quantities.

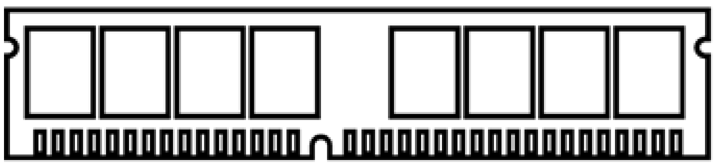
x86 registers

- 16 general registers
- 64 bits each
- Two (RSP and RBP) have a special role
- Others are more or less general
- Includes extra registers R8–R15
- Used in arithmetic instructions, control flow decisions, passing arguments to functions, etc.

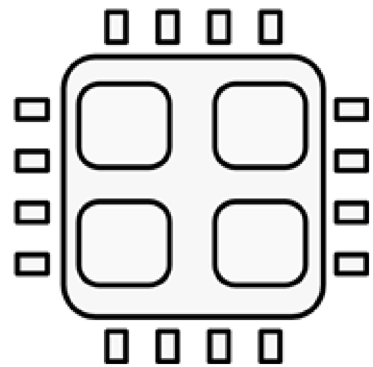
Source: CS:APP

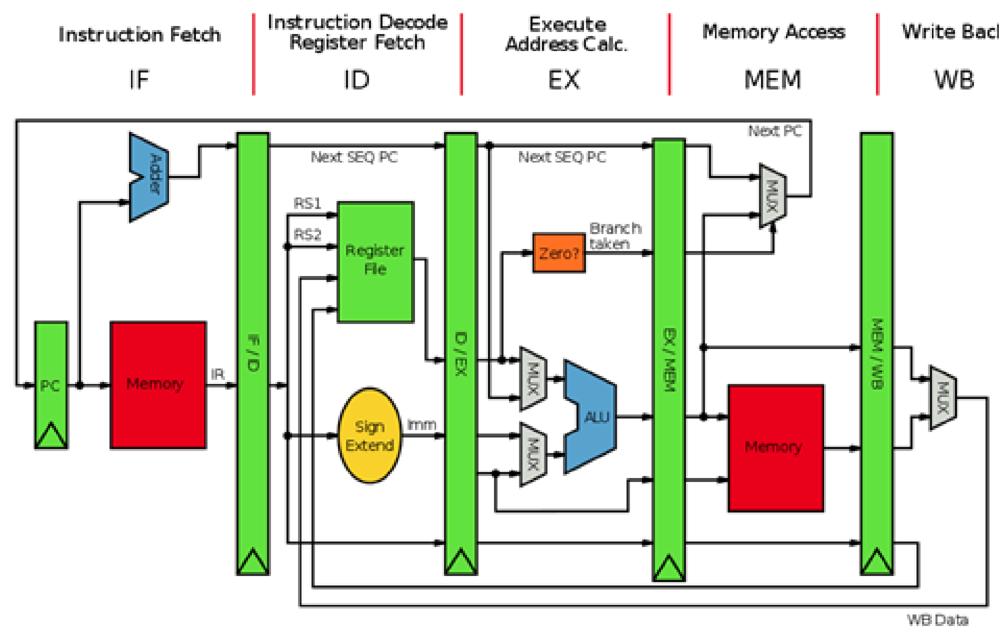
BTW, where are these registers?

Registers and Memory



Memory
Bus





Data movement instructions

We use the following notation

<reg64> Any 64-bit register (RAX..R15)

<reg32> Any 32-bit register (EAX..R15D)

<reg16> Any 16-bit register (AX..R15W)

<reg8> Any 8-bit register (AL..R15B)

<reg> Any register

<mem> A memory address (e.g., [rax], [var + 8], or qword ptr [rax+rbx])

<con64> Any 64-bit constant

<con32> Any 32-bit constant

<con16> Any 16-bit constant

<con8> Any 8-bit constant

<con> Any 8-, 16-, 32-, or 64-bit constant

mov instruciton

- 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 rax, rbx            ; copy the value in rbx into rax
mov byte ptr [var], 5 ; store 5 into the byte at location var
mov rax, [rbx]          ; Move the 8 bytes in memory at the address
                        ; contained in RBX into RAX
mov [var], rbx          ; Move the contents of RBX into the 8 bytes
                        ; at memory address var.
                        ; (Note, var is a label / address constant).
mov rax, [rsi-8]        ; Move 8 bytes at memory address RSI + (-8)
                        ; into RAX
mov [rsi+rax], cl       ; Move the contents of CL into the byte at
                        ; address RSI+RAX
```

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;

; rbx is address of the points array, rax is i
mov edx, [rbx + 8*rax + 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;

; rbx is address of the points array, eax is i

mov edx, [rbx + 8*rax + 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;

; rbx is address of the points array, eax is i

lea rsi, [rbx + 8*rax + 4] ; Move address of y of the i-th point

; into rsi

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 rax, [ rax + rbx + 1234567 ]
  ; rax = rax + rbx + 1234567 (three operands)
lea rax, [ rbx + rcx ] ; rax = rbx + rcx
  ; add without overriding rbx or rcx with the result
lea rax, [ rbx + n * rbx ] ; 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 rax, 10 ; RAX ← RAX + 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 rax, 216 ; subtract 216 from the value
             ; stored in RAX
```

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 rax ; subtract one from RAX
inc QWORD PTR [var] ; add one to the 64-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 rax, 0fH ; clear all but the last 4
             ; bits of RAX
xor rdx, rdx ; set the contents of RDX 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 63 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 63 are performed modulo 64.
- Examples

```
shl rax, 1  ; Multiply the value of RAX by 2
            ; (if the most significant bit is 0)
shr rbx, cl ; Store in RBX the floor of result of dividing
            ; the value of RBX by 2^n
            ; where n is the value in CL.
```

More instructions… (similar)

- Multiplication imul

```
imul rax, [var] ; multiply RAX by the
                ; 64-bit contents of the
                ; memory location var.
                ; Store result in RAX
imul rsi, rdi, 25 ; RSI ← RDI * 25
```

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

```
neg rax ; RAX ← - RAX
```

This is enough to do arithmetic

Poll Q1: What is inside rbx?

After we execute the mov instruction?

```
; rax = 2
; rbx = 3
mov rbx, rax
; what is the value of
; rbx here?
```

https://pollev.com/cs5460

Poll Q2: What is this instruction doing?

```
mov rbx, [rax]

; Is it writing memory? Or reading it?
```

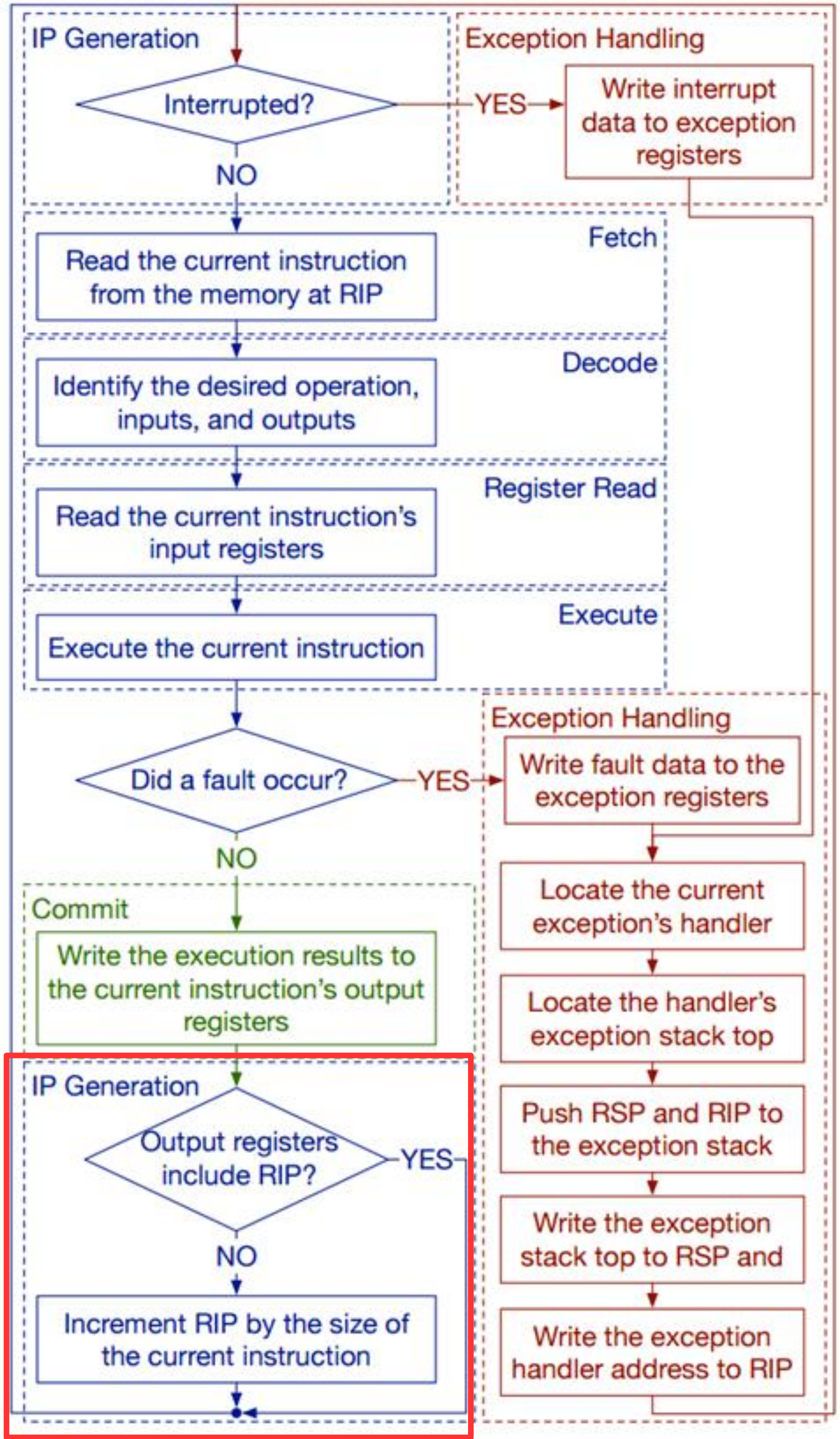
https://pollev.com/cs5460

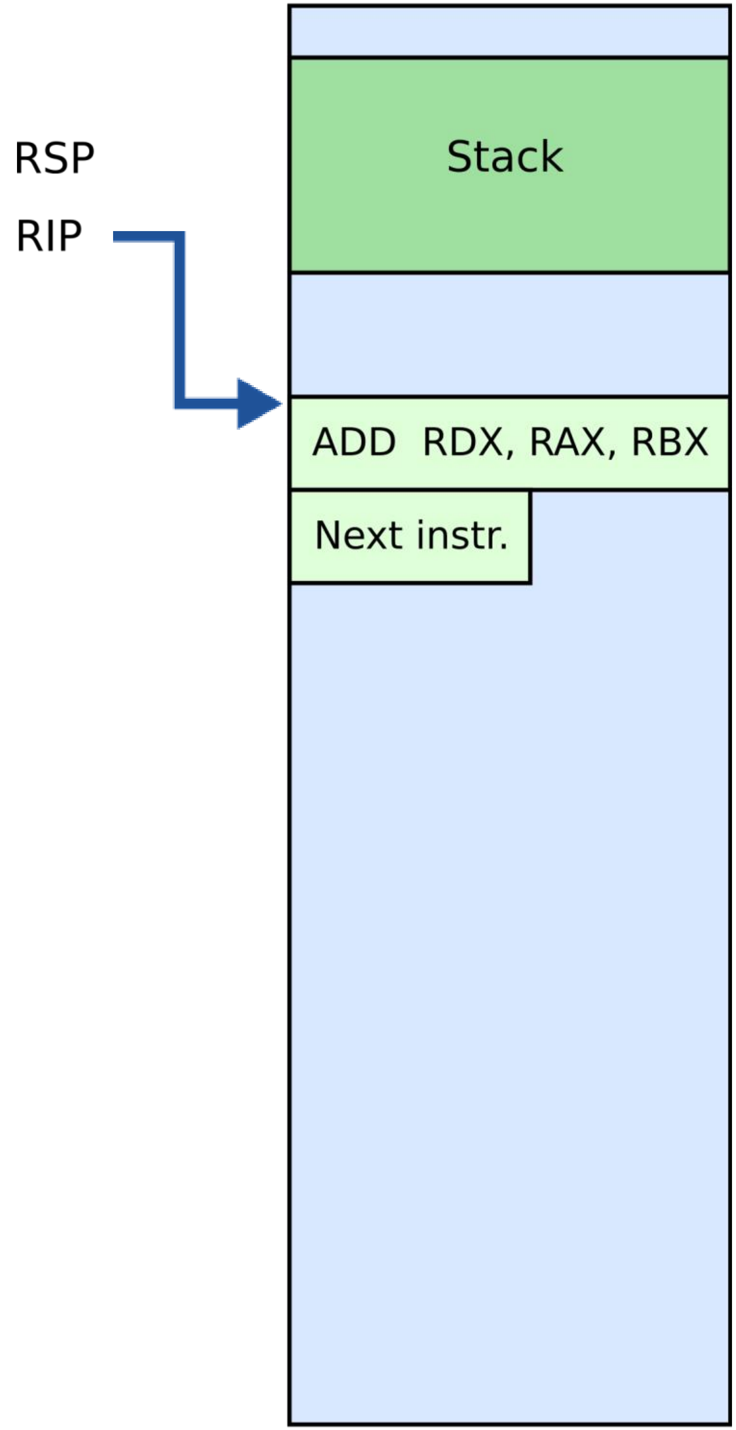
Poll Q3: Is this a legal instruction

```
mov [rbx], [rax]
```

https://pollev.com/cs5460

Control flow instructions


IP Generation
Interrupted?
YES
NO
Exception Handling
Write interrupt data to exception registers
Fetch
Read the current instruction from the memory at RIP
Decode
Identify the desired operation, inputs, and outputs
Register Read
Read the current instruction's input registers
Execute
Execute the current instruction
Did a fault occur?
YES
NO
Exception Handling
Write fault data to the exception registers
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
Write the exception stack top to RSP and
Write the exception handler address to RIP
Commit
Write the execution results to the current instruction's output registers
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction


RSP
RIP
Stack
ADD RDX, RAX, RBX
Next instr.

RIP instruction pointer

- RIP is a 64-bit value indicating the location in memory where the current instruction starts
  - (i.e., memory address of the instruction)
- RIP cannot be changed directly
  - Normally, it increments to point to the next instruction in memory
  - But it can be updated implicitly by 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 rsi, [rbp+8]
begin: xor rcx, rcx
    mov rax, [rsi]
```

jump: jump

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

```
jmp <label>
```

- Example

```
begin:  xor rcx, rcx
        ...
        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 (RFLAGS)
- RFLAGS stores information about the last arithmetic operation performedm for example,
- Bit 6 of RFLAGS 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

Instruction		Synonym	Jump condition	Description
jmp	*Label*		1	Direct jump
jmp	**Operand*		1	Indirect jump
je	*Label*	jz	ZF	Equal / zero
jne	*Label*	jnz	~ZF	Not equal / not zero
js	*Label*		SF	Negative
jns	*Label*		~SF	Nonnegative
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	Greater (signed >)
jge	*Label*	jnl	~(SF ^ OF)	Greater or equal (signed >=)
jl	*Label*	jnge	SF ^ OF	Less (signed <)
jle	*Label*	jng	(SF ^ OF) | ZF	Less or equal (signed <=)
ja	*Label*	jnbe	~CF & ~ZF	Above (unsigned >)
jae	*Label*	jnb	~CF	Above or equal (unsigned >=)
jb	*Label*	jnae	CF	Below (unsigned <)
jbe	*Label*	jna	CF | ZF	Below or equal (unsigned <=)

Figure 3.15 **The jump instructions.** These instructions jump to a labeled destination when the jump condition holds. Some instructions have "synonyms," alternate names for the same machine instruction.

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 RAX is less than or equal to RBX, jump to the label done. Otherwise, continue to the next instruction

```
cmp rax, rbx
jle done
```

cmp: compare

- Compare the values of the two specified operands, setting the condition codes in RFLAGS
- 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 8 bytes stored at location var are equal to the 8-byte integer constant 10, jump to the location labeled loop.

```
cmp QWORD PTR [var], 10
jeq loop
```

Stack and procedure calls

What is stack?

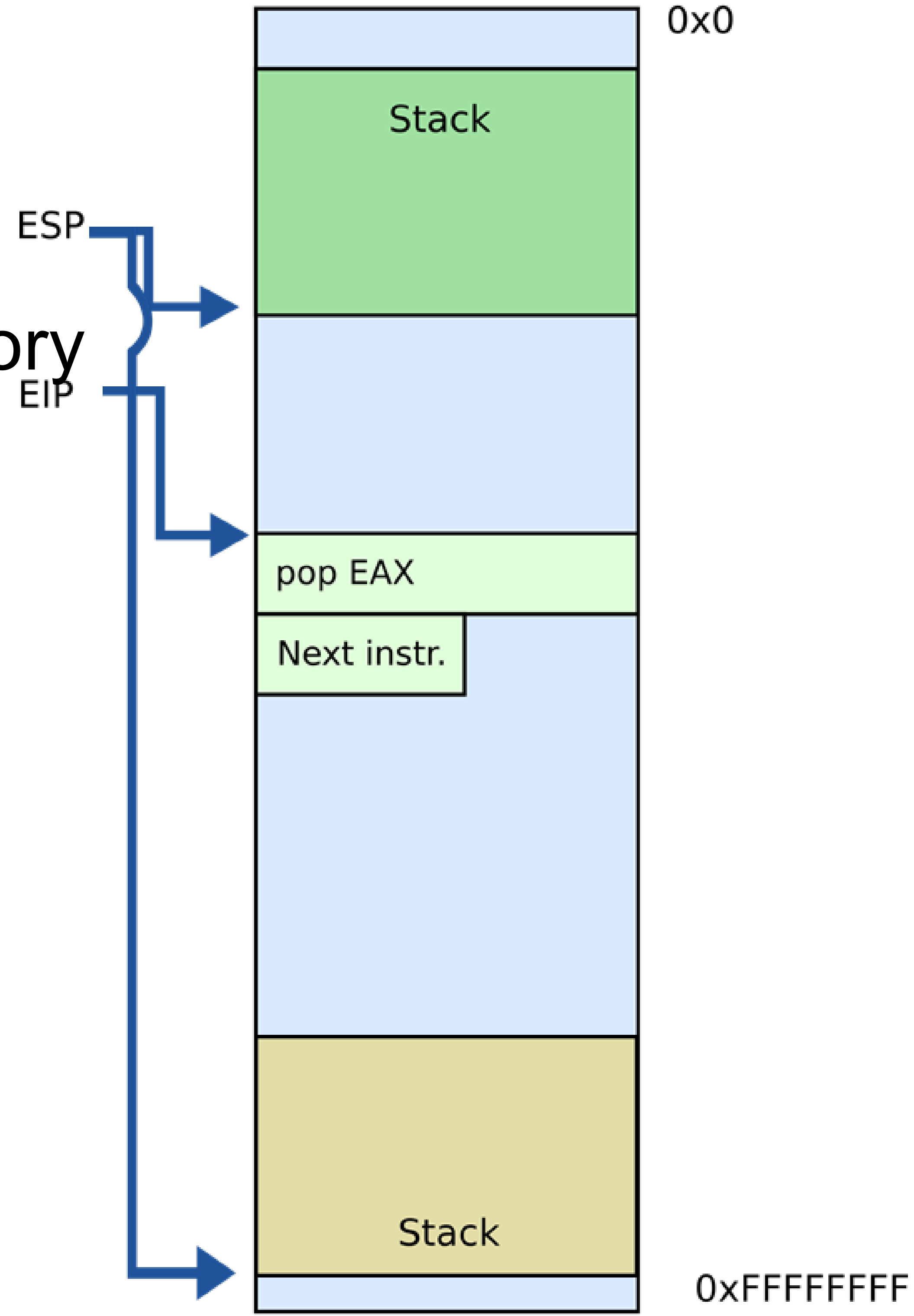
Stack

It's just a region of memory

Pointed by a special register RSP

You can change RSP

Get a new stack



Why do we need stack?

Calling functions

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

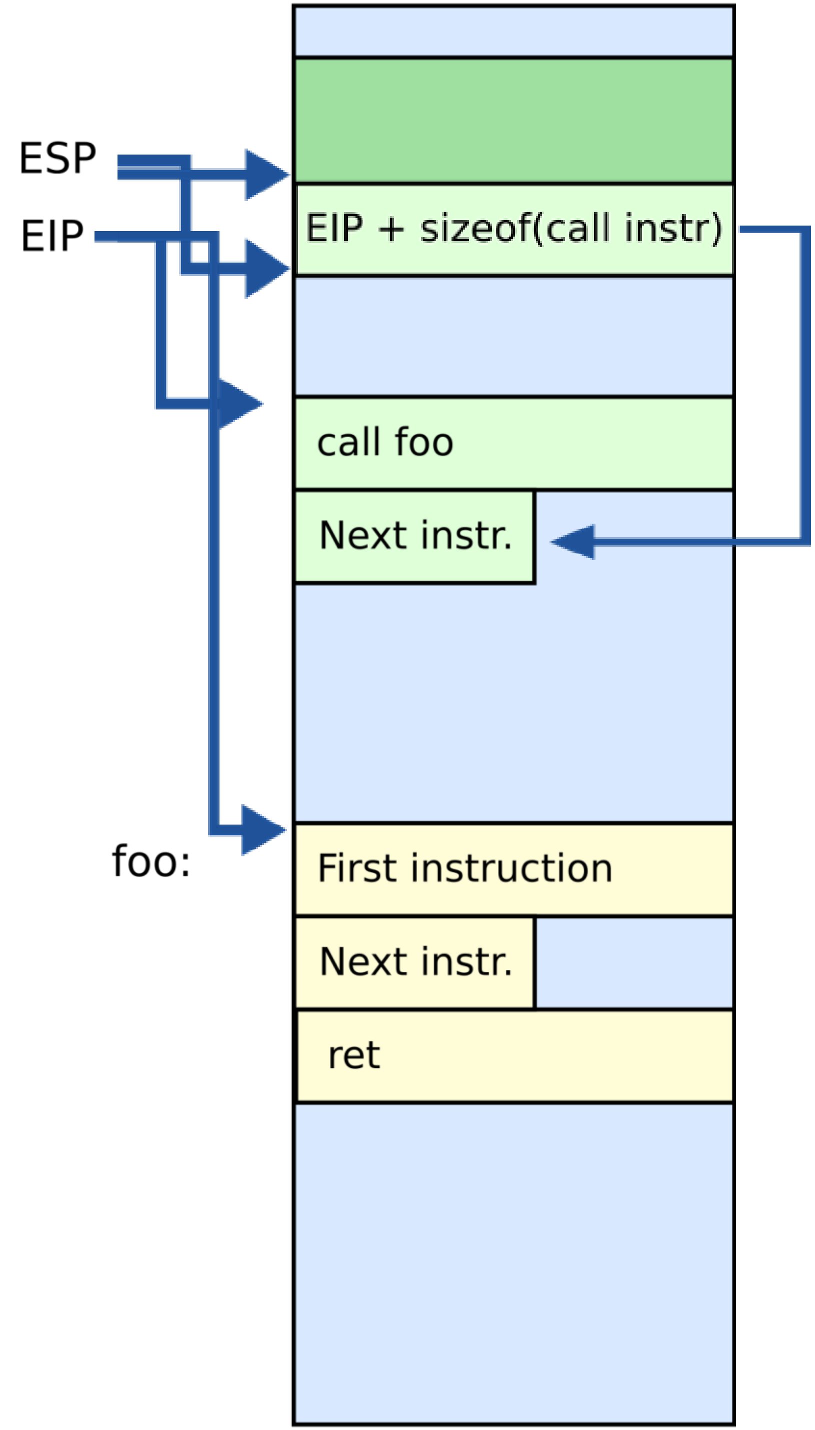
- Stack contains information for how to return from a subroutine
- i.e., from foo()

- Functions can be called from different places in the program

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

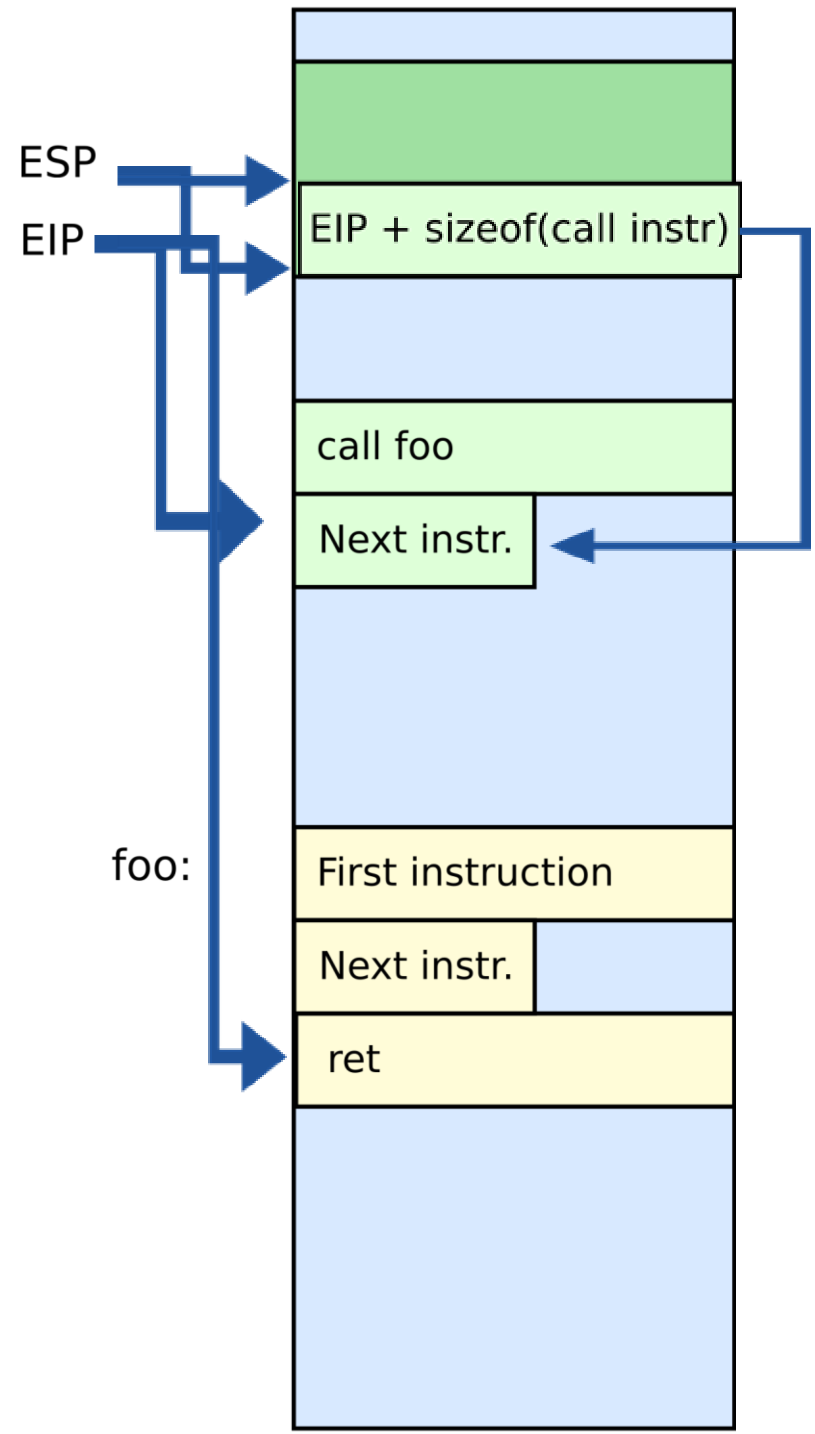
Stack

- 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

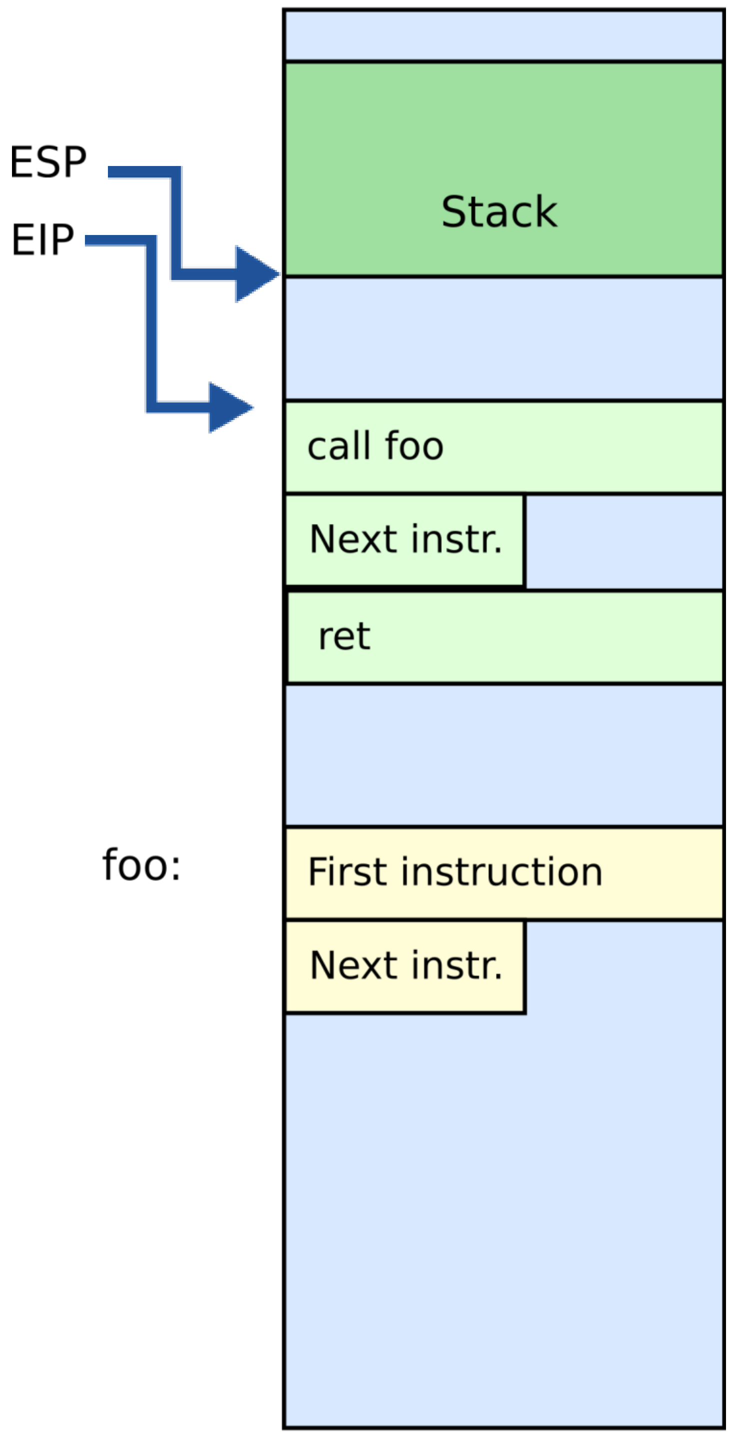
```
    push rip + sizeof(CALL) ; save return
                            ; address
  jmp _my_function
```

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

RSP register

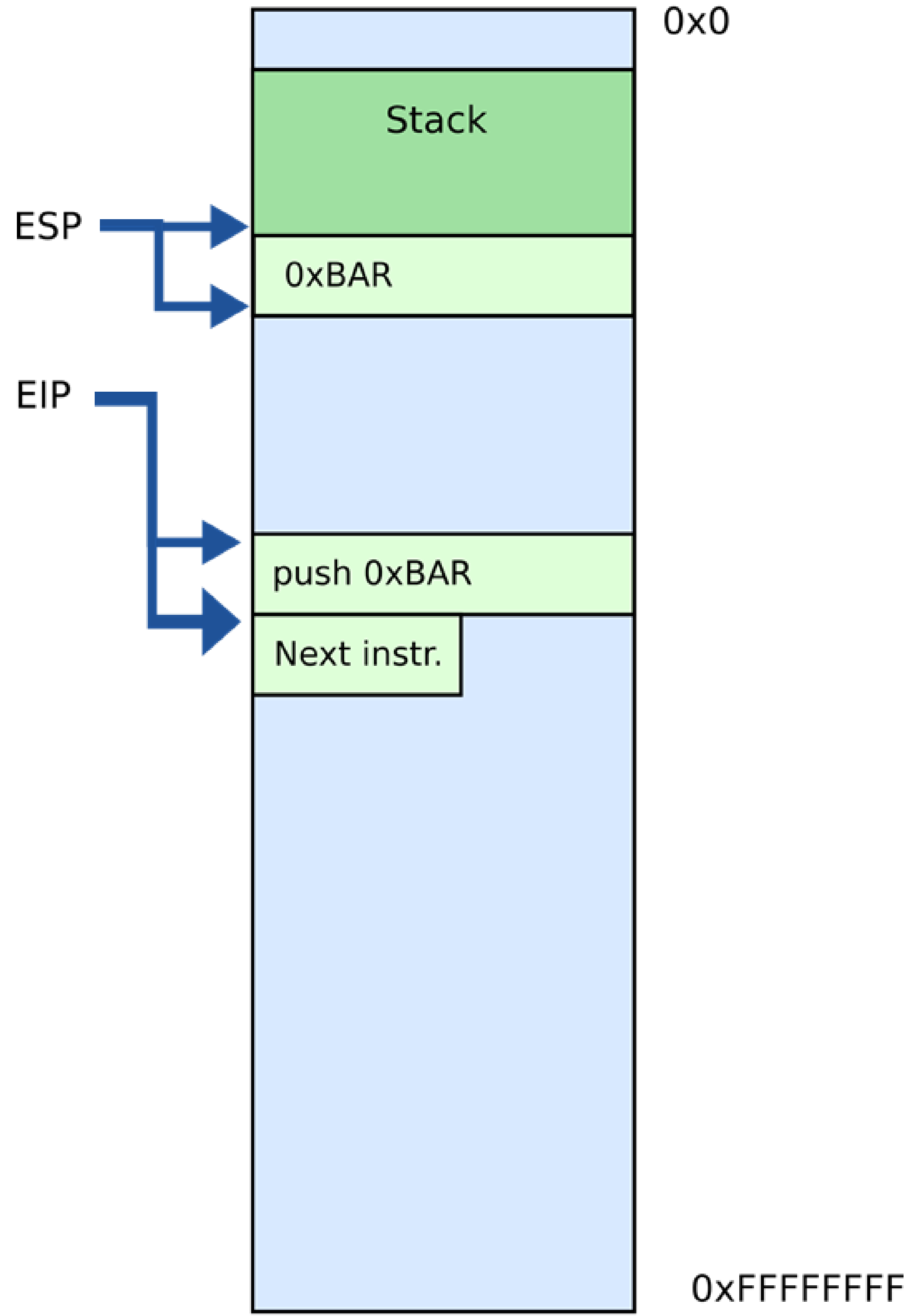
Contains the memory address of the topmost element in the stack

PUSH instruction

push 0xBAR

Subtract 8 from RSP

Insert data on the stack



Manipulating stack

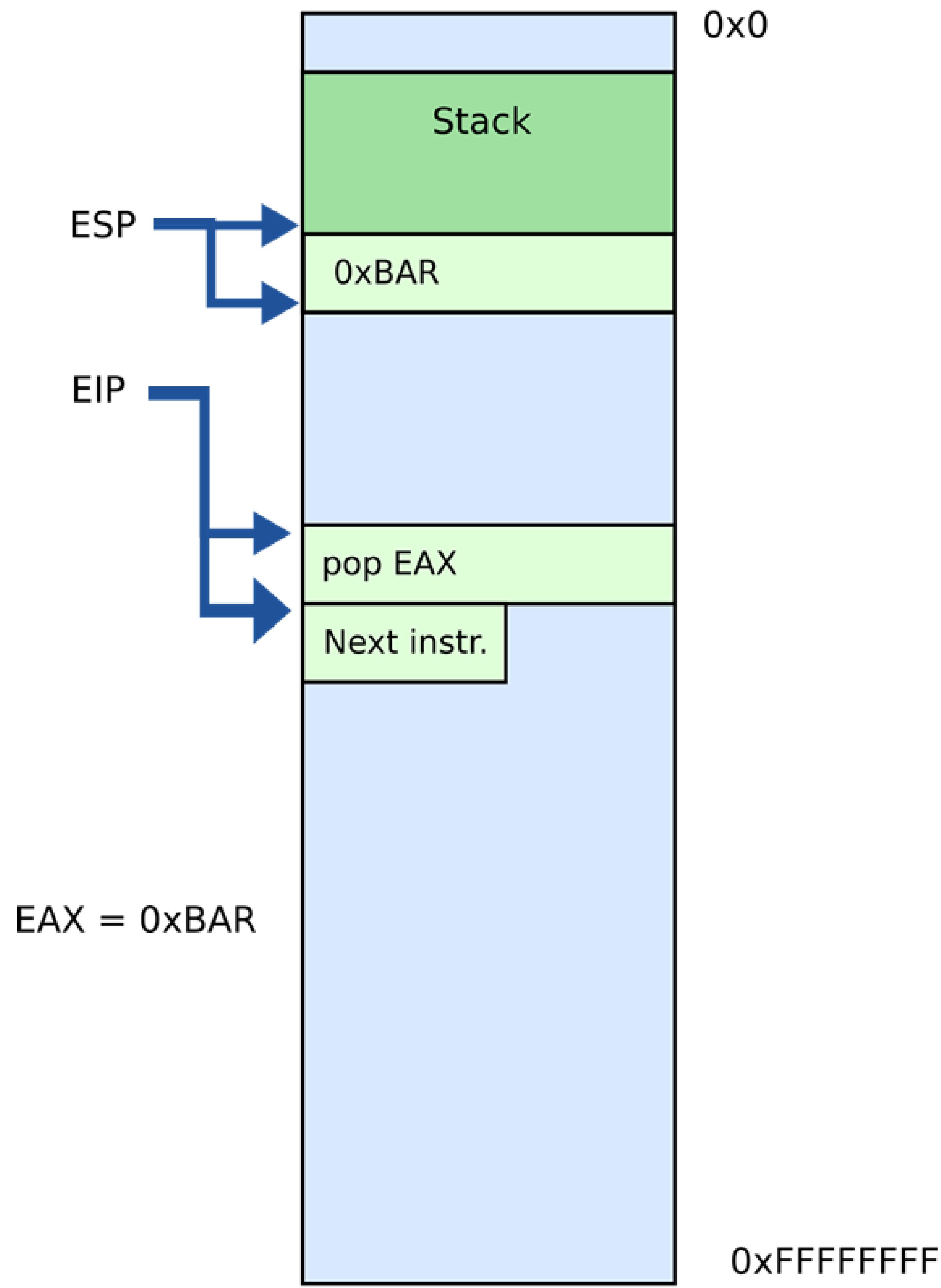
POP instruction

pop RAX

Removes data from the stack

Saves in register or memory

Adds 8 to RSP



Thank you!

Old 32bit ASM slides

We use the following notation

<reg32> Any 32-bit register (EAX,EBX,ECX,EDX,ESI,EDI,ESP,EBP)

<reg16> Any 16-bit register (AX, BX, CX, or DX)

<reg8> Any 8-bit register (AH, BH, CH, DH, AL, BL, CL, 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 instruciton

- 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 then 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 result in EAX
imul esi, edi, 25 ; ESI ← EDI * 25
```

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

```
neg eax ; EAX ← - EAX
```

This is enough to do arithmetic

Poll Q1: What is inside ebx?

- After we execute the mov instruction?

```
; eax = 2

; ebx = 3

mov ebx, eax

; what is the value of eax here?
```

What is inside ebx?

ebx is 3

0%

ebx is 2

0%

None of the above

0%

Poll Q2: What is this instruction doing?

```
mov ebx, [eax]

; Is it writing memory? Or reading it?
```

What is this instruciton mov ebx, [eax] doing?

Reading memory	0%
Wrriting memory	0%
None of the above	0%

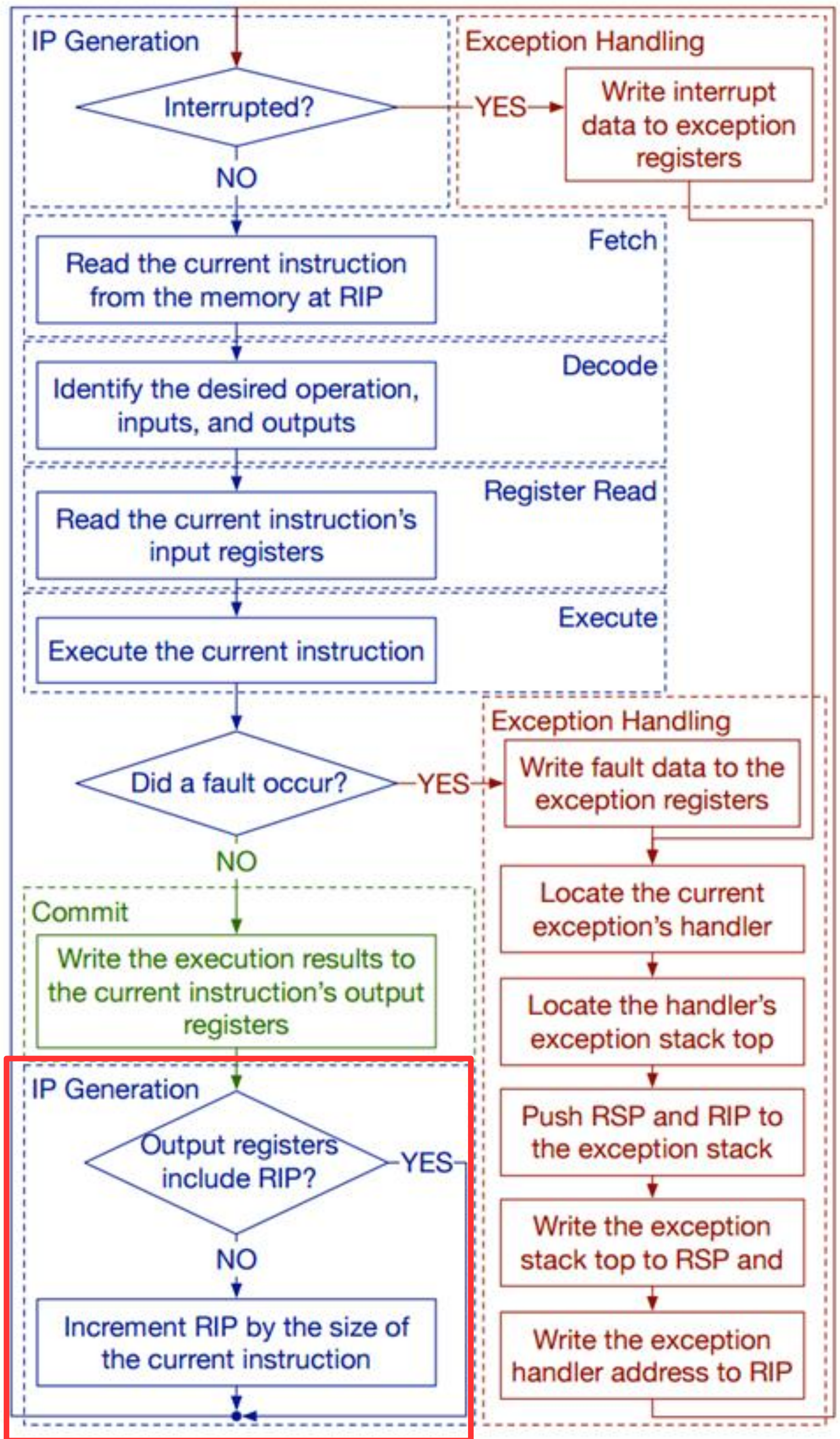
Poll Q3: Is this a legal instruction

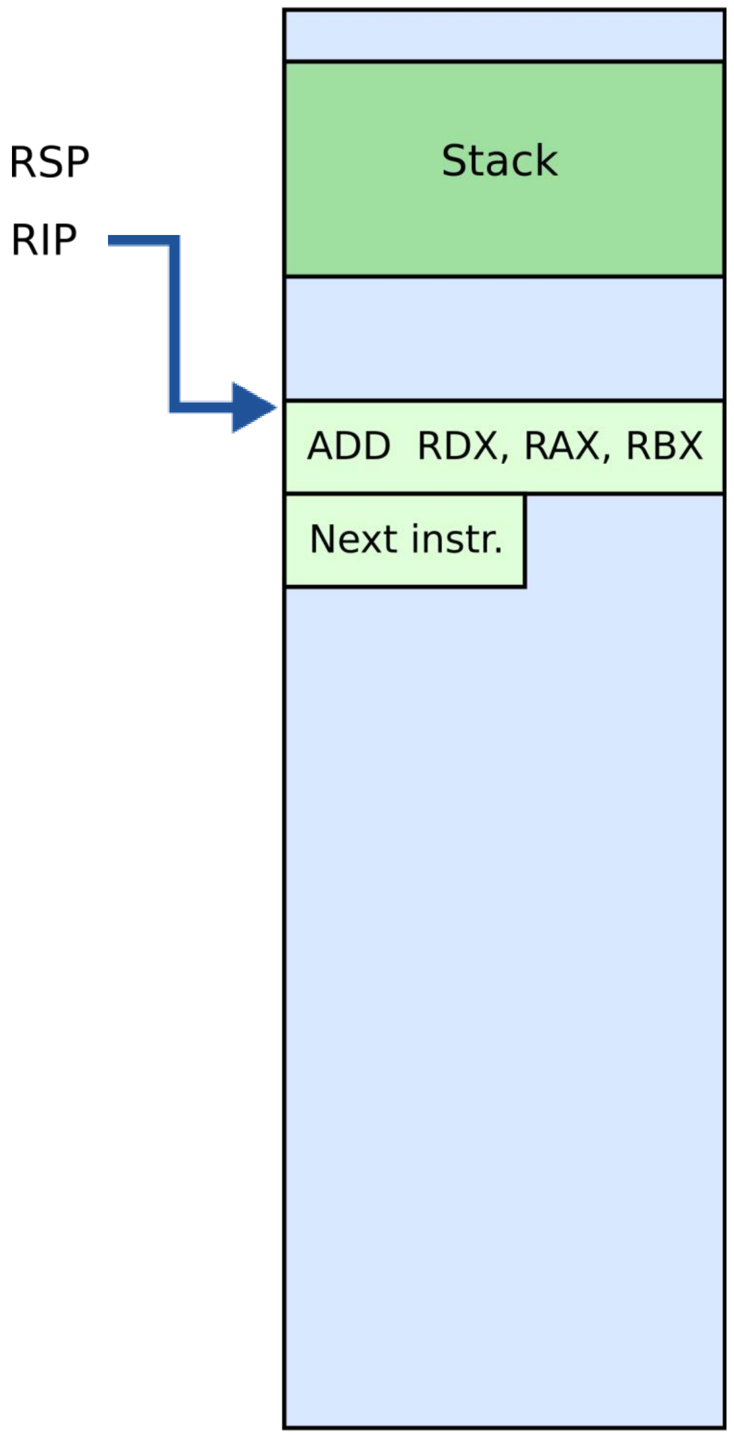
```
mov [ebx], [eax]
```

Is this a legal x86 instruction? mov [eax], [ebx]

Yes	0%
No	0%

Control flow instructions


IP Generation
Interrupted?
YES
NO
Exception Handling
Write interrupt data to exception registers
Fetch
Read the current instruction from the memory at RIP
Decode
Identify the desired operation, inputs, and outputs
Register Read
Read the current instruction's input registers
Execute
Execute the current instruction
Did a fault occur?
YES
NO
Exception Handling
Write fault data to the exception registers
Locate the current exception's handler
Locate the handler's exception stack top
Push RSP and RIP to the exception stack
Write the exception stack top to RSP and
Write the exception handler address to RIP
Commit
Write the execution results to the current instruction's output registers
IP Generation
Output registers include RIP?
YES
NO
Increment RIP by the size of the current instruction


RSP
RIP
Stack
ADD RDX, RAX, RBX
Next instr.

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 performedm 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
```

Poll Q1: What is inside ebx?

- After we execute the mov instruction?

```
; eax = 2

; ebx = 3

mov ebx, eax

; what is the value of ebx here?
```

What is inside ebx?

ebx is 3

0%

ebx is 2

0%

None of the above

0%

Poll Q2: What is this instruction doing?

```
mov ebx, [eax]

; Is it writing memory? Or reading it?
```

What is this instruciton mov ebx, [eax] doing?

Reading memory

0%

Wrriting memory

0%

None of the above

0%

Poll Q3: Is this a legal instruction

```
mov [ebx], [eax]
```

Is this a legal x86 instruction? mov [eax], [ebx]

Yes

0%

No

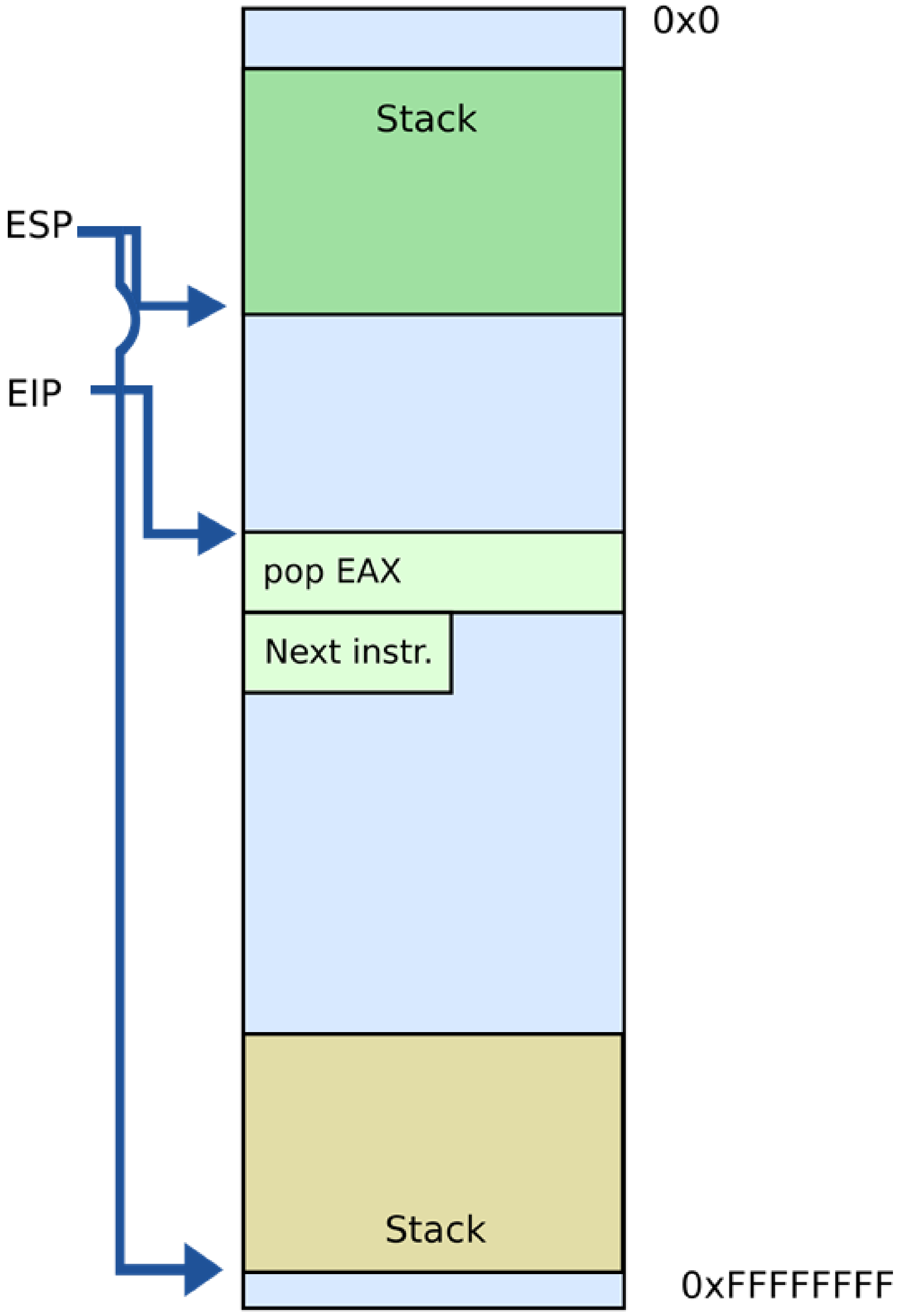
0%

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..
```

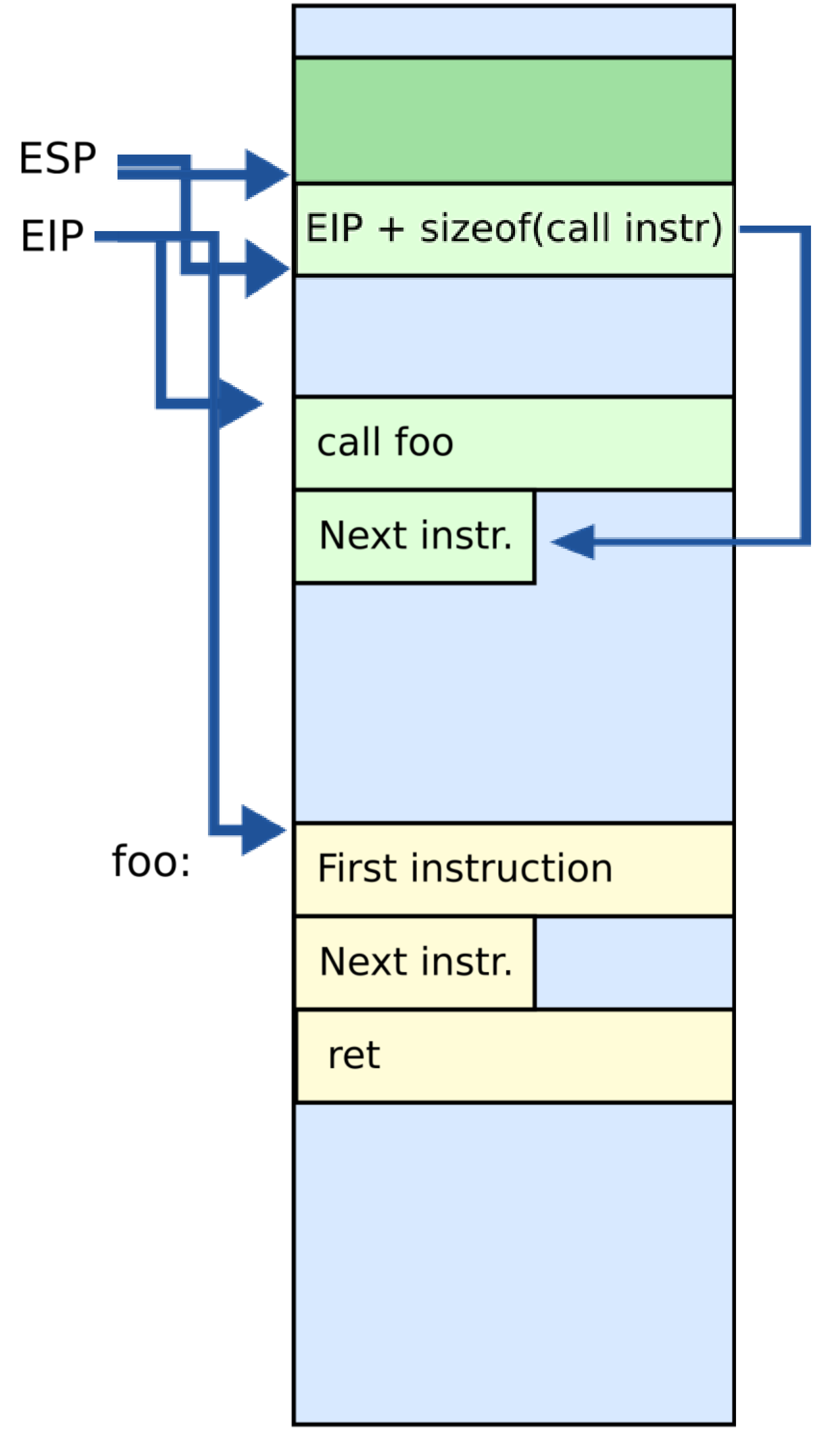
- Stack contains information for how to return from a subroutine
- i.e., from foo()

- Functions can be called from different places in the program

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

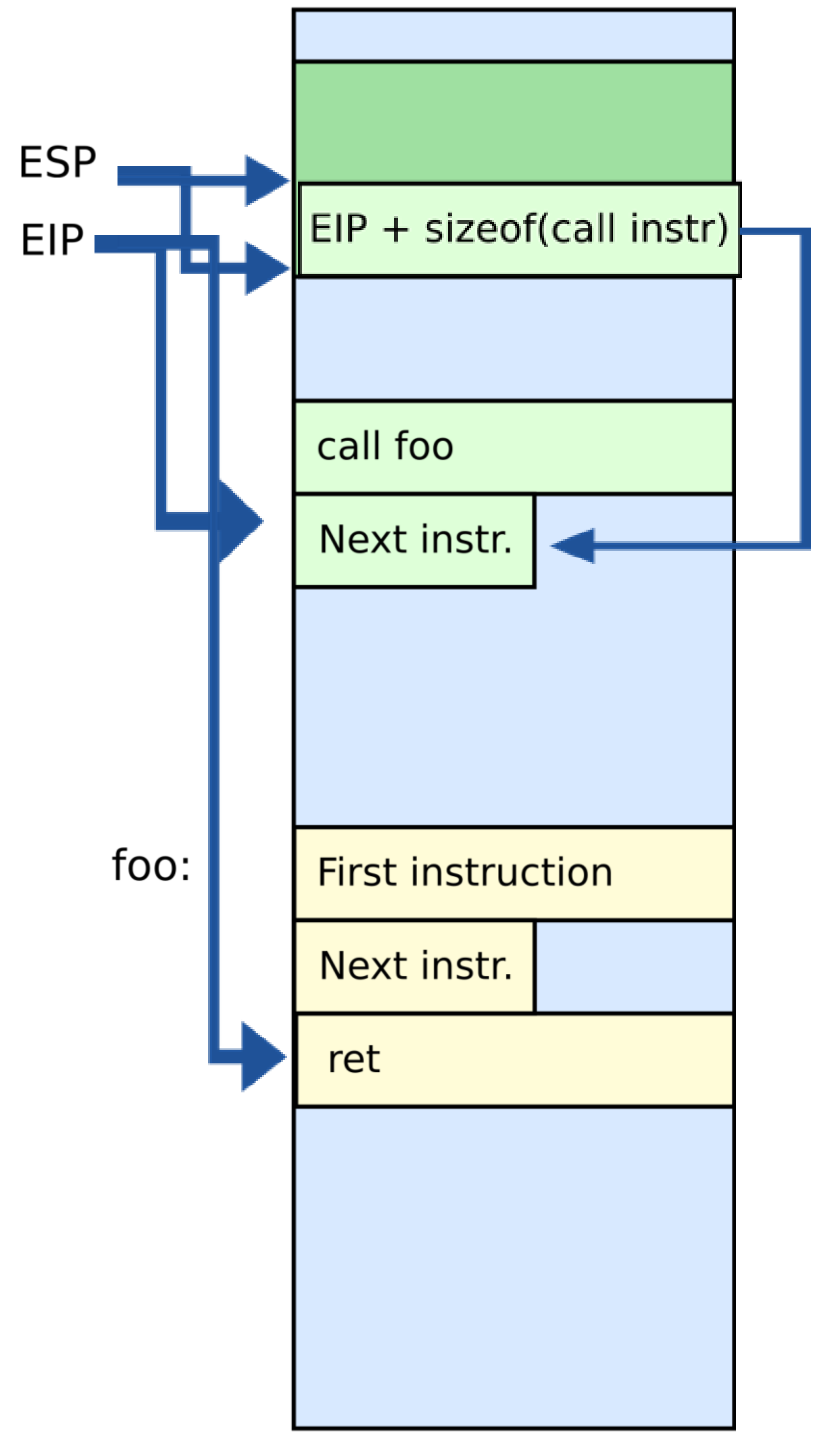
Stack

- 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

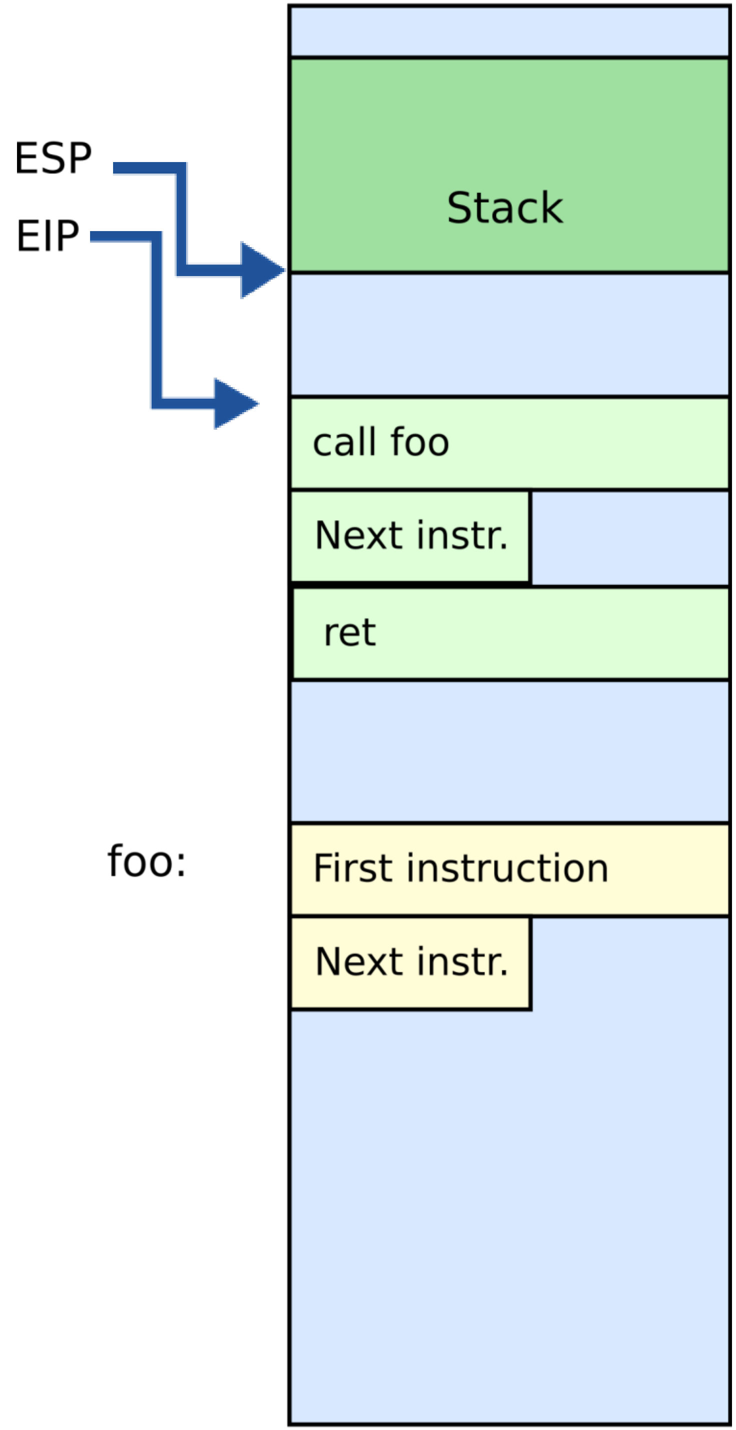
- CALL instruction
- Makes an unconditional jump to a subprogram and pushes the address of the next instruction on the stack

```
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

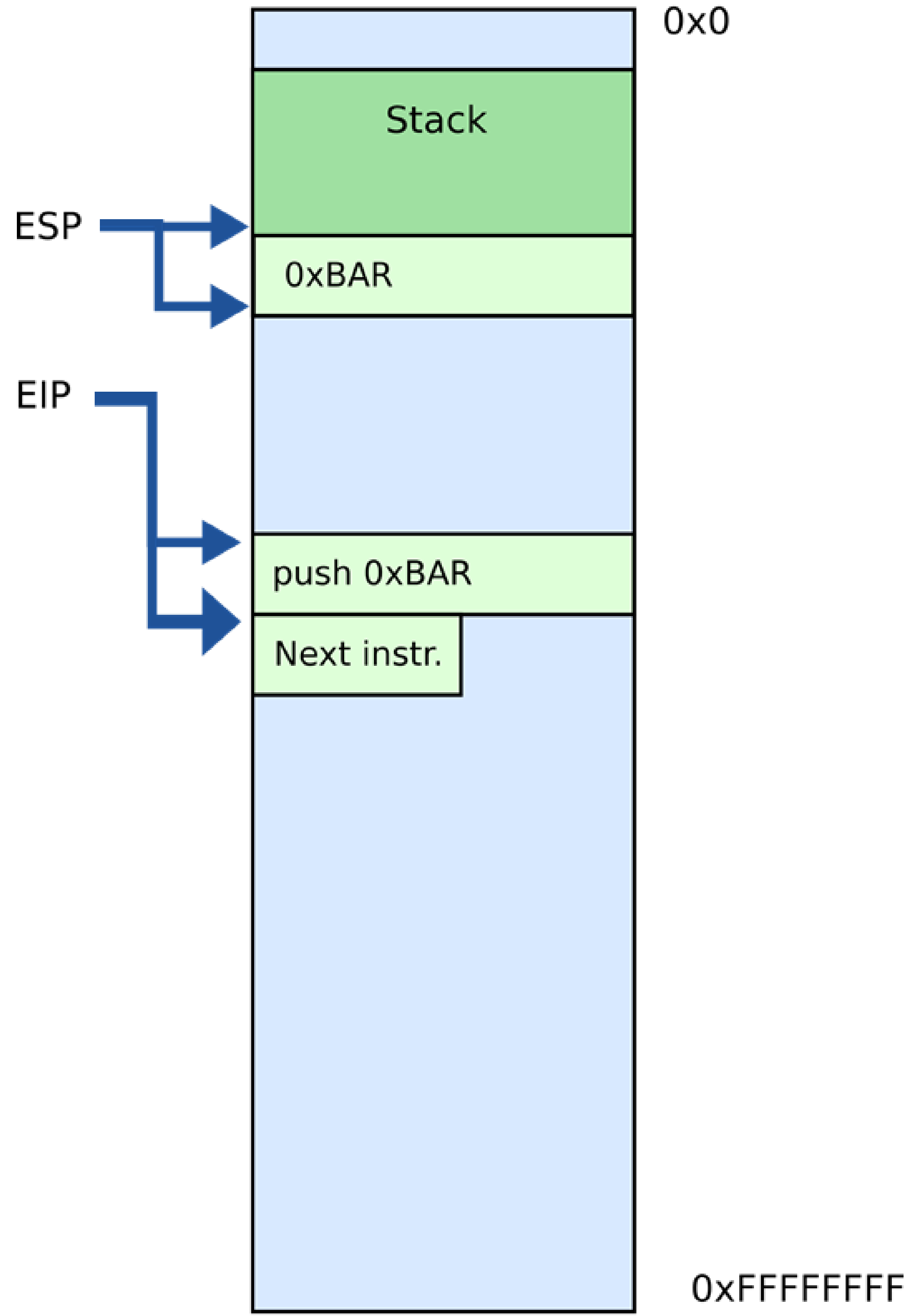


Manipulating stack

- 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

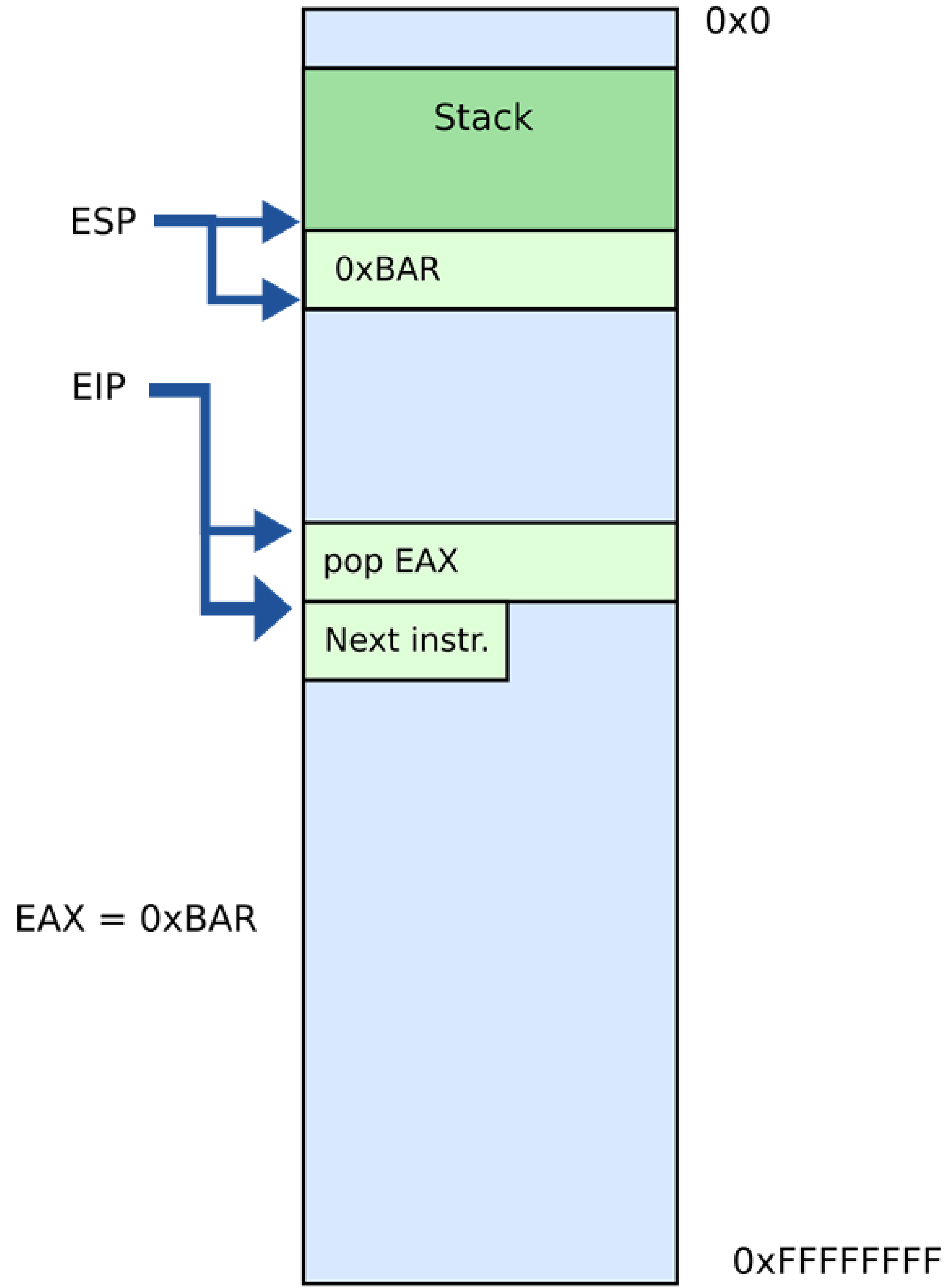


Manipulating stack

- POP instruction

pop EAX

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



Thank you!

Some examples