Machine Program: Procedure

Jinyang Li

Slides based on Tiger Wang

Roadmap: how does hardware execute a program?

- Where is data stored?
  - Instructions and (most) data are stored in memory
  - Temporary data (e.g. local variables, loop variables) stored in registers
- How does CPU execute a program?
  - Load an instruction from memory according to PC
  - Execute instruction (may access memory)
  - update PC
  - Repeat
- Modes of execution:
  - sequential:
    - PC is changed to point to the next instruction
  - control flow: jmp, conditional jmp
    - PC is changed to point to the jump target address
  - **Today→ procedure call**

Requirements of procedure calls?

```
P(…) {
  y = Q(x);
  y++;
}
```

```
int Q(int i)
{
  int t, z;
  ...
  return z;
}
```

1. Passing control

Requirements of procedure calls?

```
P(…) {
  y = Q(x);
  y++;
}
```

```
int Q(int i)
{
  int t, z;
  ...
  return z;
}
```

1. Passing control
2. Passing Arguments & return value

Requirements of procedure calls?

```
P(…) {
  y = Q(x);
  y++;
}
```

```
int Q(int i)
{
  int t, z;
  ...
  return z;
}
```

1. Passing control
2. Passing Arguments & return value
3. Allocate / deallocate local variables

How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

```
void f(){
    ..
    g(..)
L2: ..
}
```

```
void g(){
    ..
    h(..)
L3: ..
}
```

How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

Jump to f()

Remember where to come back

```
void f(){
    ..
    g(..)
L2: ..
}
```

L1

```
void g(){
    ..
    h(..)
L3: ..
}
```

How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

```
void f(){
    ..
    g(..)
L2: ..
}
```

```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to f()
Remember where to come back

Jump to g()
Remember where to come back

L2
L1

How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

Jump to f()
Remember where to come back

```
void f(){
    ..
    g(..)
L2: ..
}
```

Jump to g()
Remember where to come back

L3
L2
L1

```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to h()
Remember where to come back

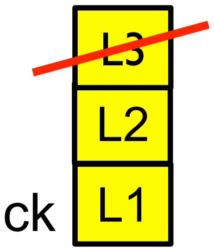
How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

Jump to f()
Remember where to come back

```
void f(){
    ..
    g(..)
L2: ..
}
```

Jump to g()
Remember where to come back




```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to L3
Forget L3

How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

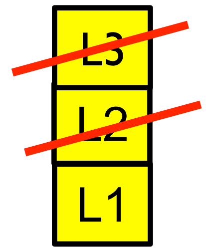
Jump to f()
Remember where to come back

```
void f(){
    ..
    g(..)
L2: ..
}
```

Jump to L2
Forget L2

```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to L3
Forget L3



How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

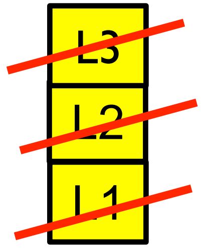
```
void f(){
    ..
    g(..)
L2: ..
}
```

```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to L1
Forget L1

Jump to L2
Forget L2

Jump to L3
Forget L3



How to transfer control for procedure calls?

```
void main(){
    ..
    f(..)
L1: ..
}
```

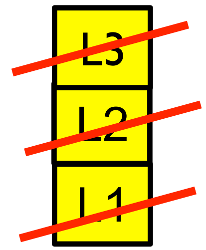
Jump to L1
Forget L1

```
void f(){
    ..
    g(..)
L2: ..
}
```

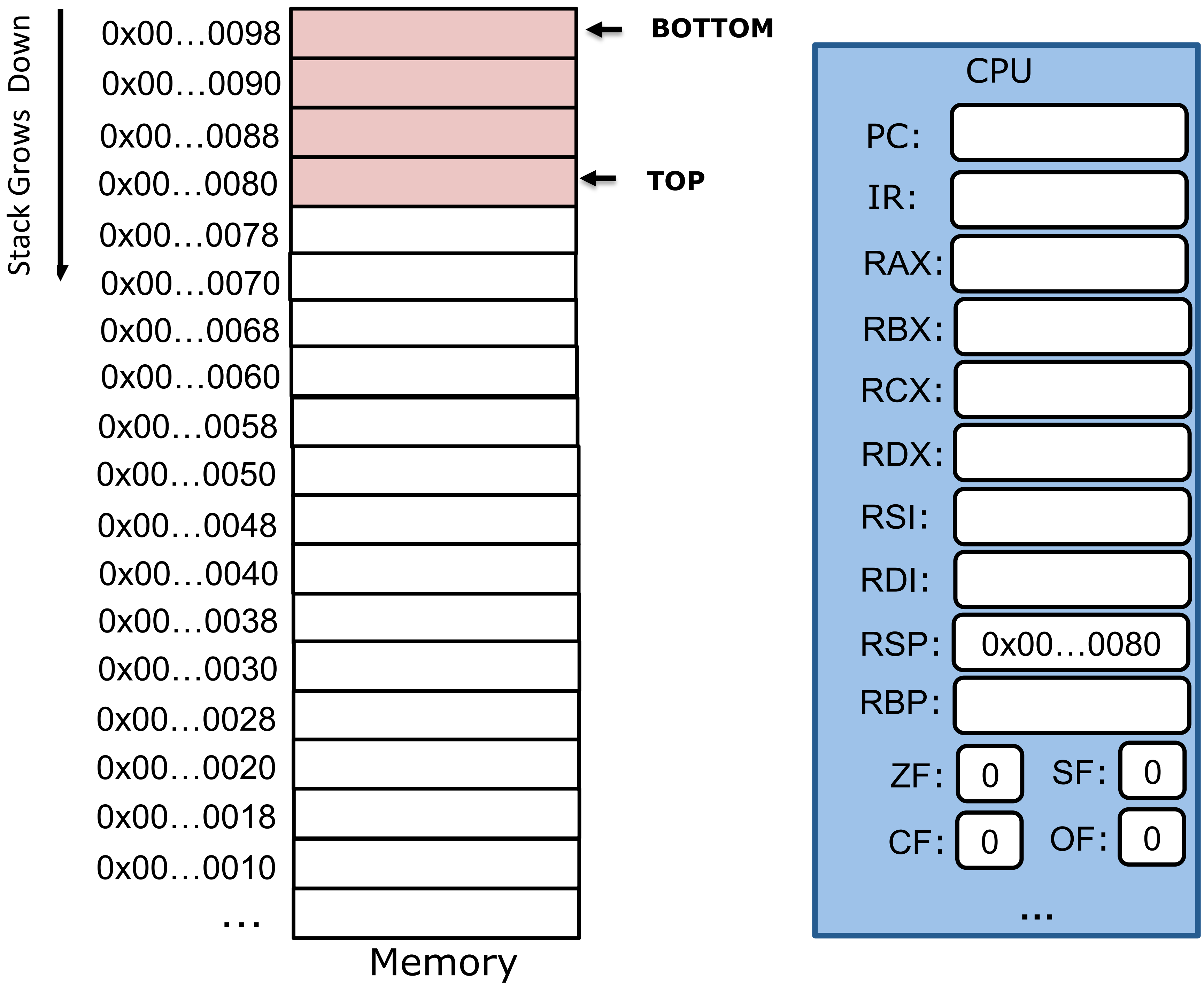
Jump to L2
Forget L2

```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to L3
Forget L3



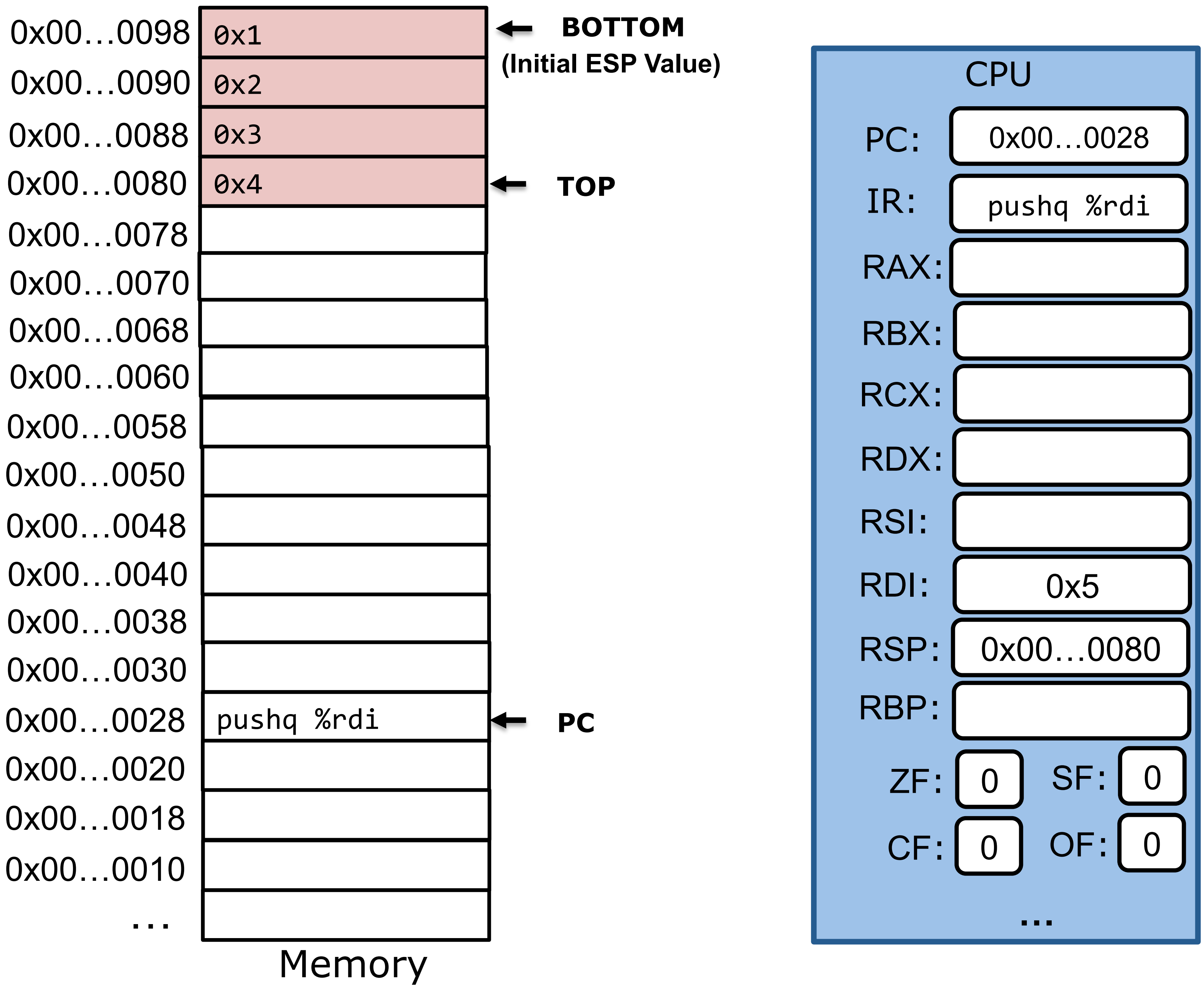
Stack

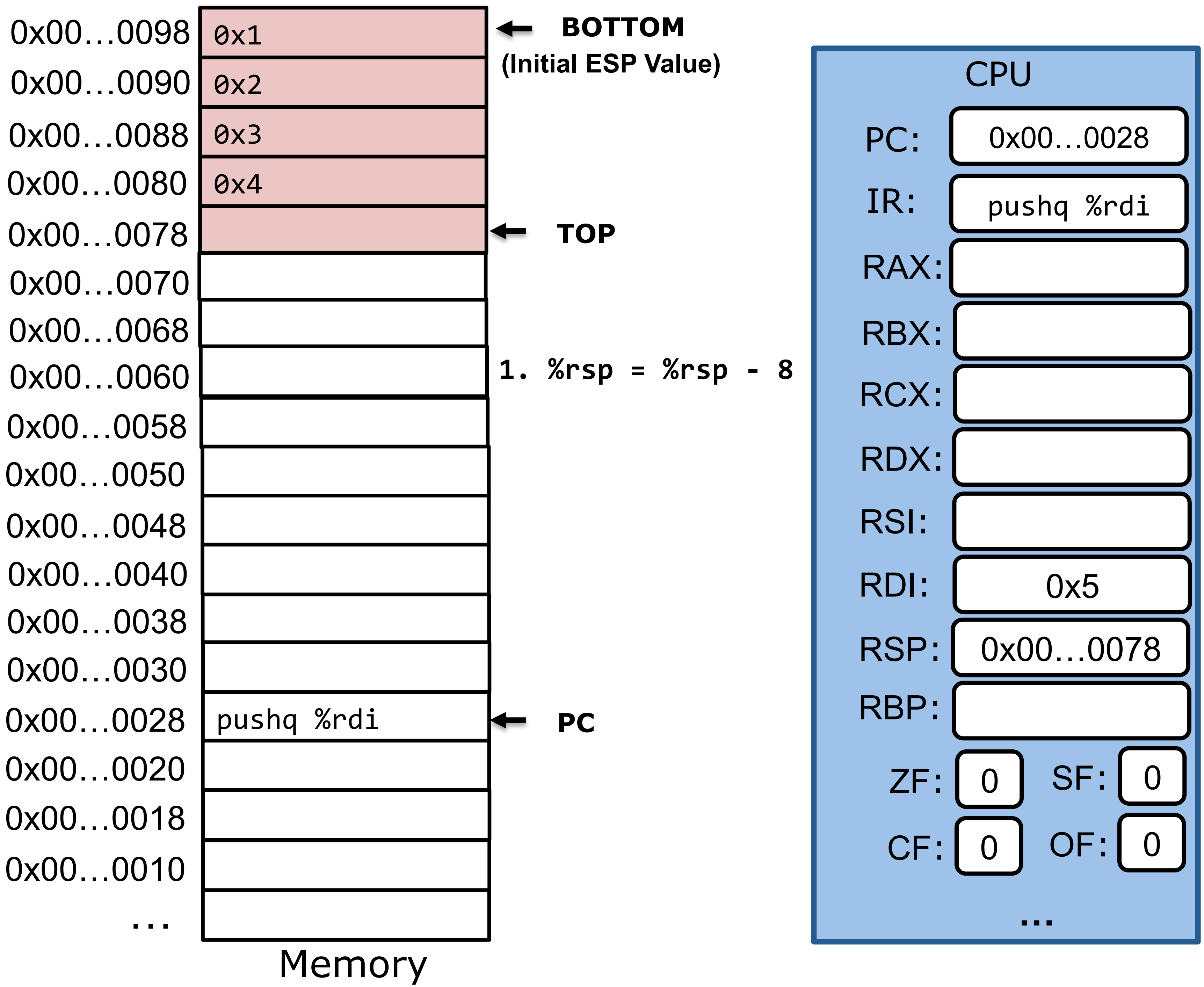

Stack Grows Down
0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
BOTTOM
TOP
Memory
CPU
PC:
IR:
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP: 0x00…0080
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…

Stack – push Instruction

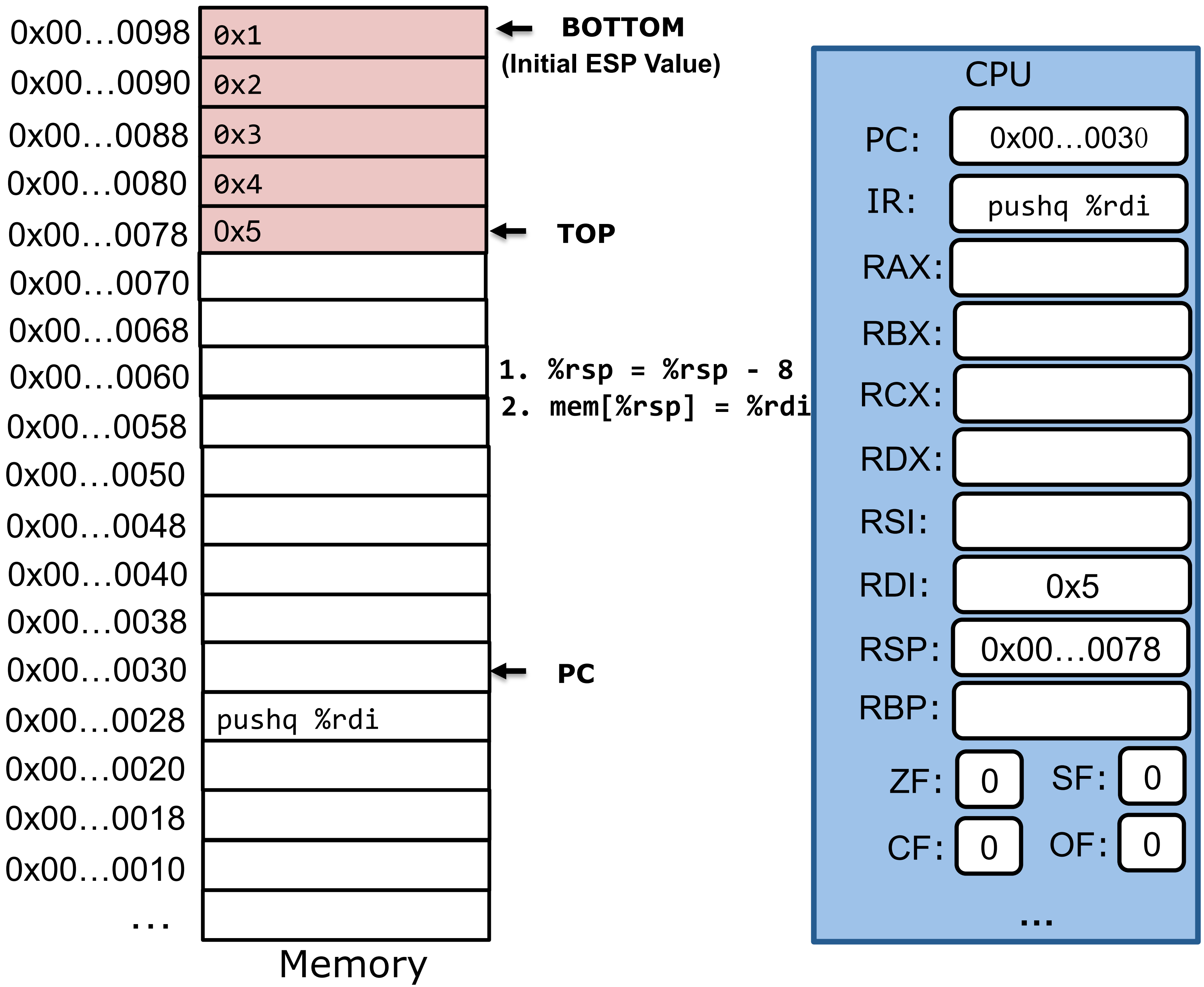
pushq `src`

- Decrement %rsp by 8
- Write operand at address given by `%rsp`


0x00…0098
0x1
BOTTOM
(Initial ESP Value)
0x00…0090
0x2
0x00…0088
0x3
0x00…0080
0x4
TOP
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
pushq %rdi
PC
0x00…0020
0x00…0018
0x00…0010
...
Memory
CPU
PC:
0x00…0028
IR:
pushq %rdi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
0x5
RSP:
0x00…0080
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...



0x00…0098
0x1
BOTTOM
(Initial ESP Value)
0x00…0090
0x2
0x00…0088
0x3
0x00…0080
0x4
0x00…0078
TOP
0x00…0070
0x00…0068
0x00…0060
1. %rsp = %rsp - 8
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
pushq %rdi
PC
0x00…0020
0x00…0018
0x00…0010
…
Memory
CPU
PC: 0x00…0028
IR: pushq %rdi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI: 0x5
RSP: 0x00…0078
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…

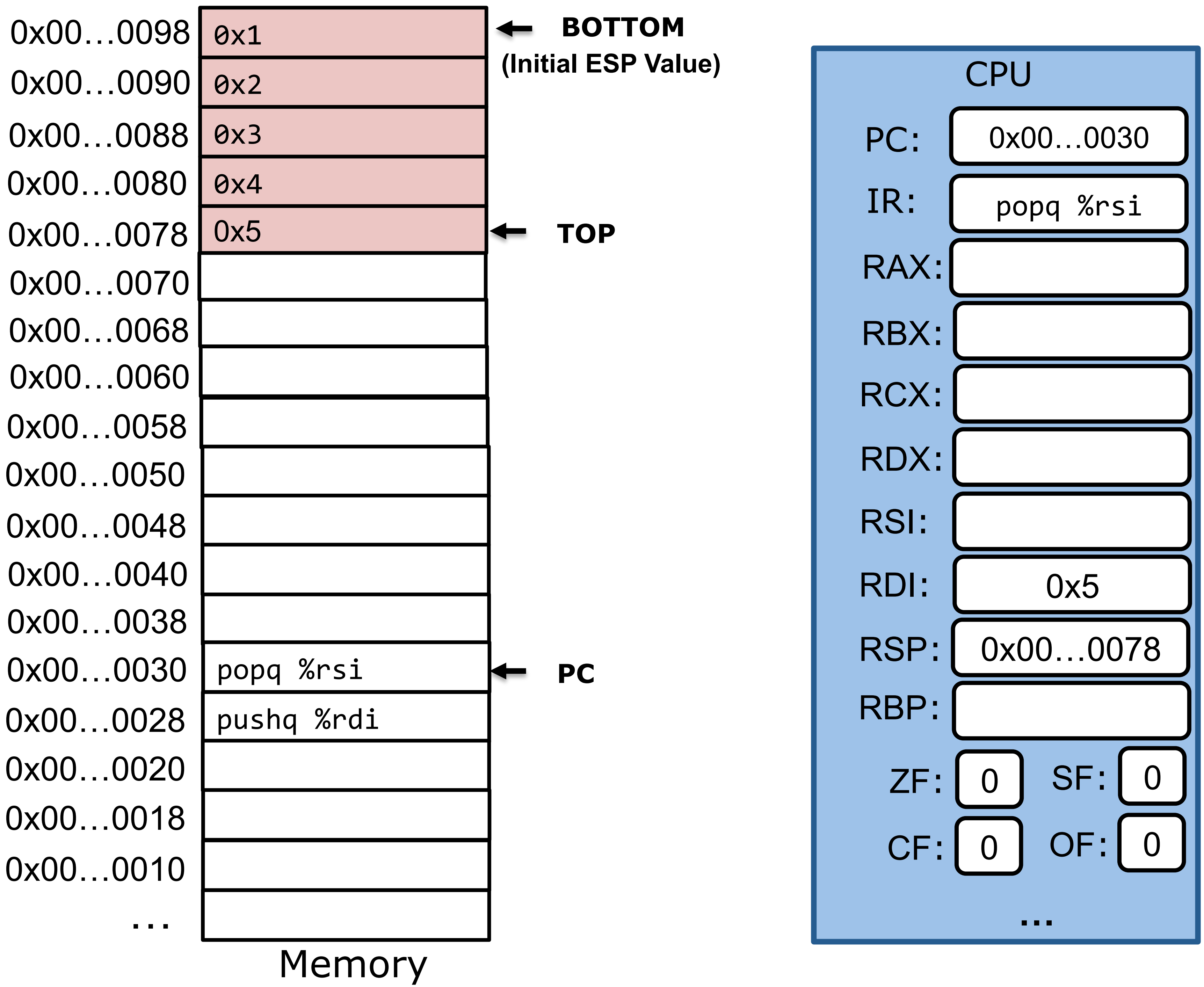


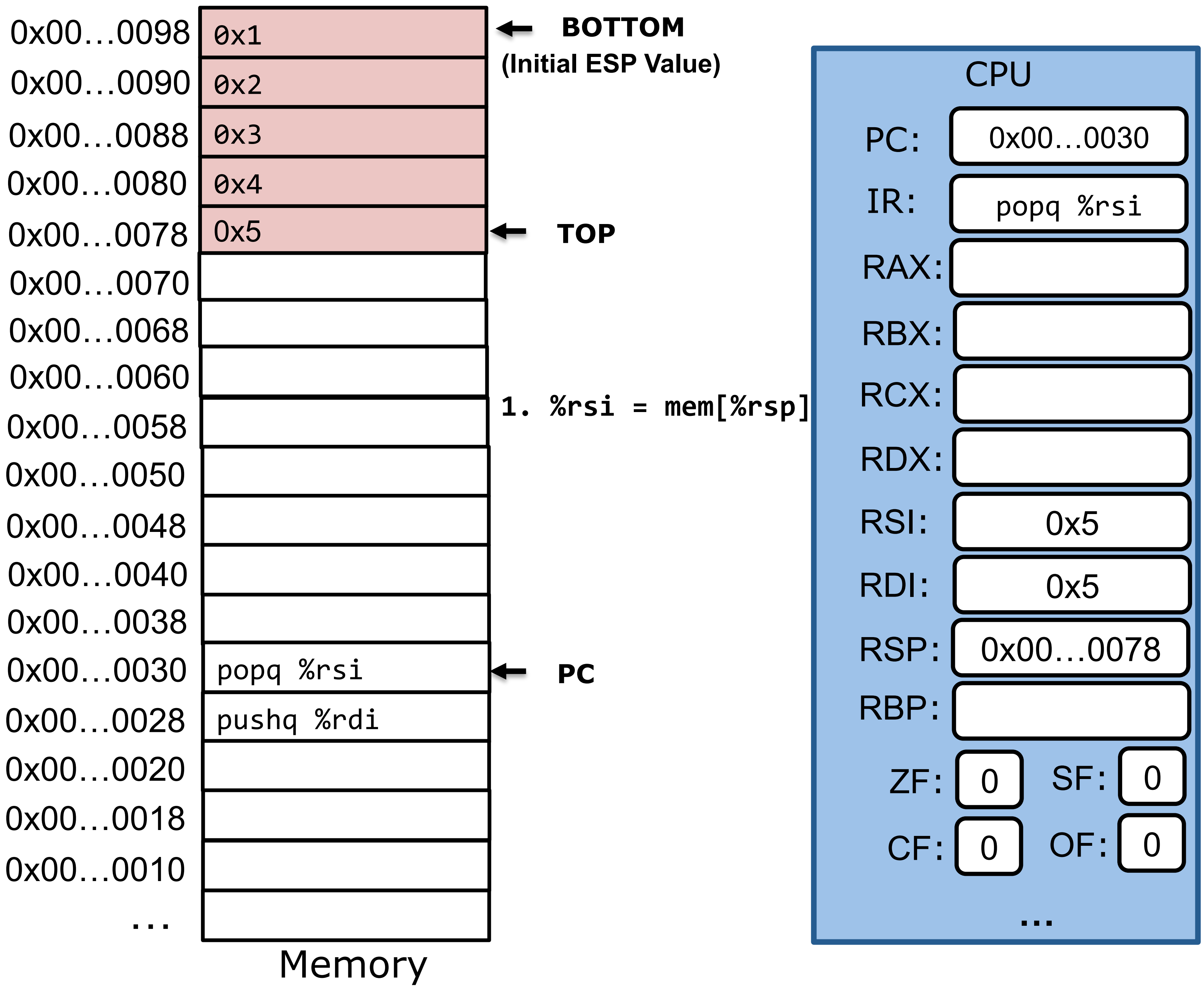
0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
0x1
0x2
0x3
0x4
0x5
pushq %rdi
Memory
BOTTOM
(Initial ESP Value)
TOP
1. %rsp = %rsp - 8
2. mem[%rsp] = %rdi
PC
CPU
PC: 0x00…0030
IR: pushq %rdi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI: 0x5
RSP: 0x00…0078
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…

Stack – pop Instruction

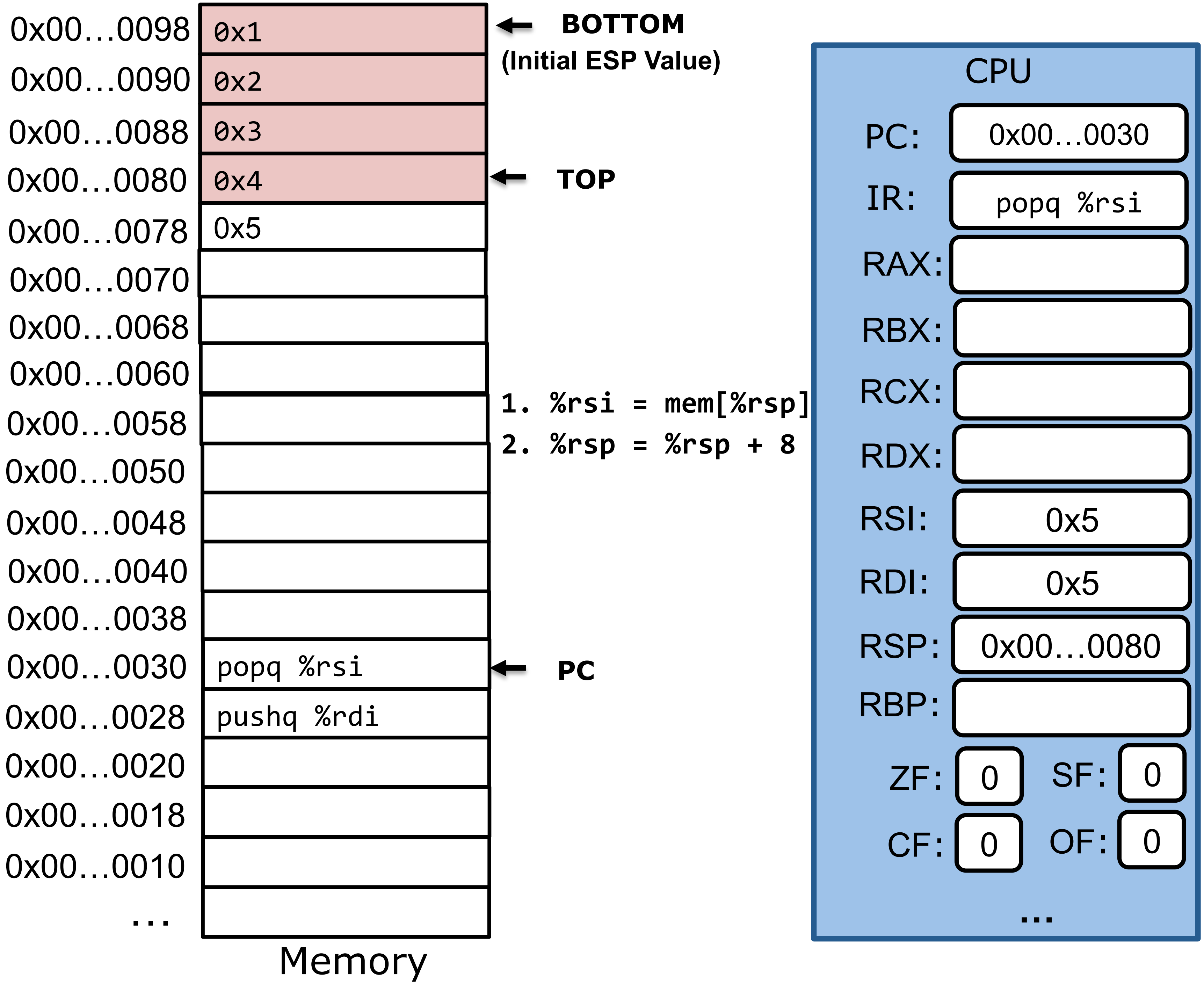
`popq` `dest`

- Store the value at address `%rsp` to dest
- Increment %rsp by 8


0x00…0098
0x1
BOTTOM
(Initial ESP Value)
0x00…0090
0x2
0x00…0088
0x3
0x00…0080
0x4
0x00…0078
0x5
TOP
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
popq %rsi
PC
0x00…0028
pushq %rdi
0x00…0020
0x00…0018
0x00…0010
...
Memory
CPU
PC: 0x00…0030
IR: popq %rsi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI: 0x5
RSP: 0x00…0078
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...



0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
0x1
0x2
0x3
0x4
0x5
popq %rsi
pushq %rdi
Memory
BOTTOM
(Initial ESP Value)
TOP
1. %rsi = mem[%rsp]
PC
CPU
PC: 0x00…0030
IR: popq %rsi
RAX:
RBX:
RCX:
RDX:
RSI: 0x5
RDI: 0x5
RSP: 0x00…0078
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…



0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
0x1
0x2
0x3
0x4
0x5
popq %rsi
pushq %rdi
Memory
BOTTOM
(Initial ESP Value)
TOP
1. %rsi = mem[%rsp]
2. %rsp = %rsp + 8
PC
CPU
PC: 0x00…0030
IR: popq %rsi
RAX:
RBX:
RCX:
RDX:
RSI: 0x5
RDI: 0x5
RSP: 0x00…0080
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
…

Control transfer from caller to callee

`call` `label(func name)`

- Push return address on stack
  - Current pc + 8
- Jump label
  - Change the pc to the address of the label

```
int add(int a, int b) {
  int c = a + b;
  return c;
}
```

```
int main() {

    int a = 0;
    int b = 2;
    int c = add(a, b);
    printf("%d\b", c);
    return 0;
}
```

Control transfer – call Instruction

`call` `label(func name)`

- Push the return address on stack
  - Return address points to the next instruction after **call**
- Jump label
  - Change the pc to label's value

```
add:
    leal (%rdi,%rsi), %eax
    ret
```

```
main:
    movl    $2, %esi
    movl    $0, %edi
    call    add
    movl    %eax, %edx
     ...
```

GCC -Og *.c

return address points to this instruction

Control transfer from callee back to caller

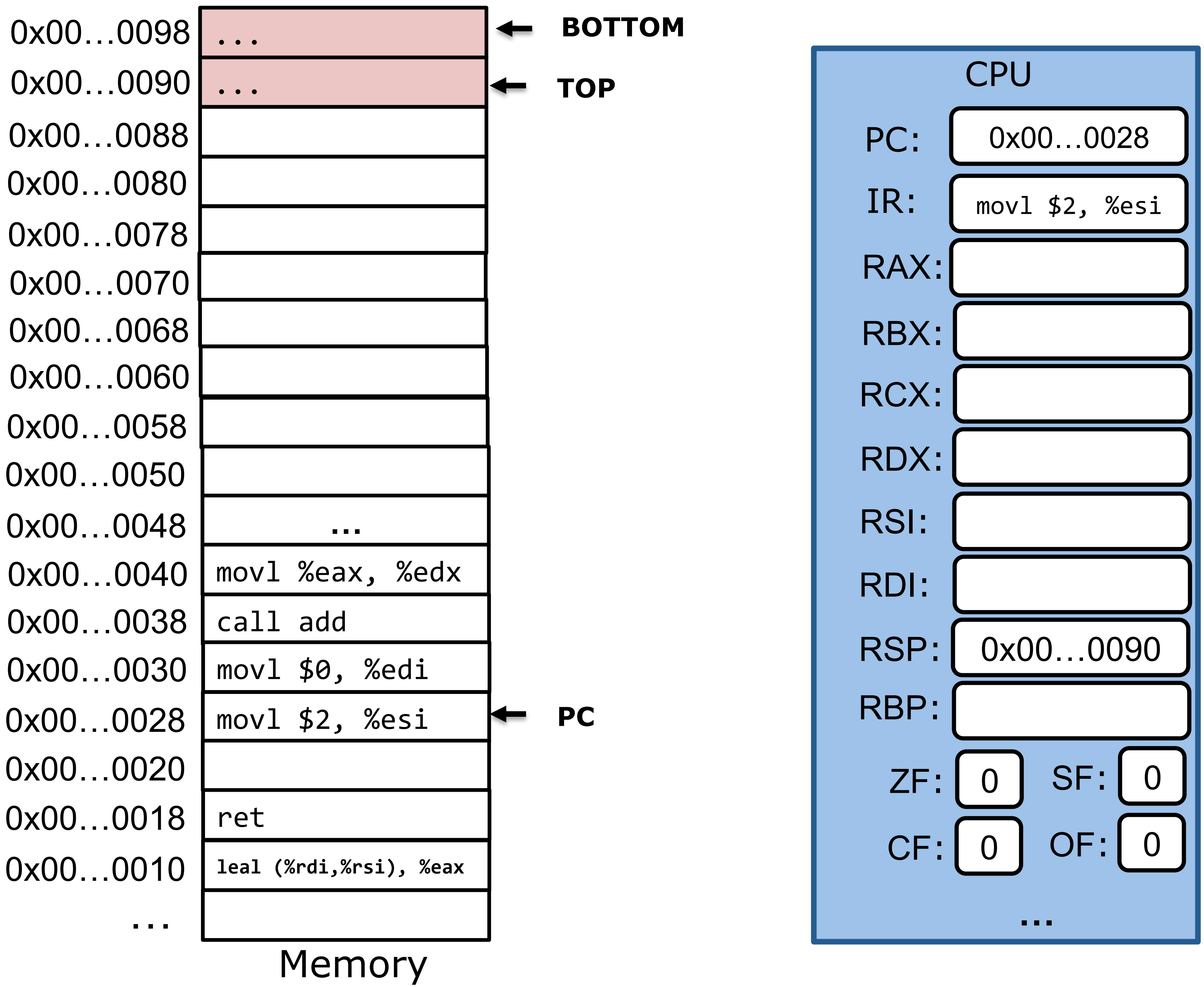
`ret`

- Pop 8 bytes from the stack to PC
  - pc = mem[%rsp]

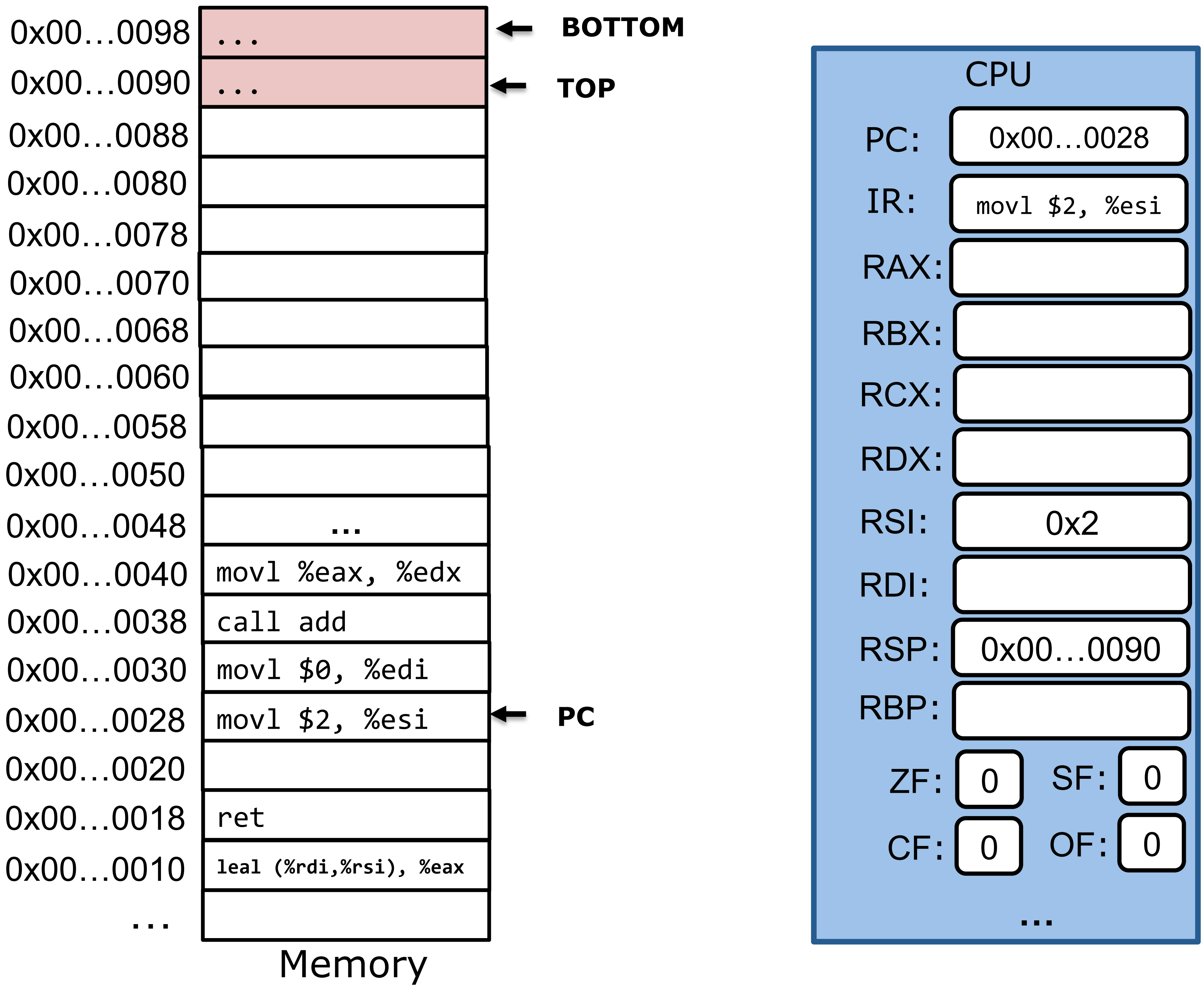
```
add:
    leal (%rdi,%rsi), %eax
    ret
```

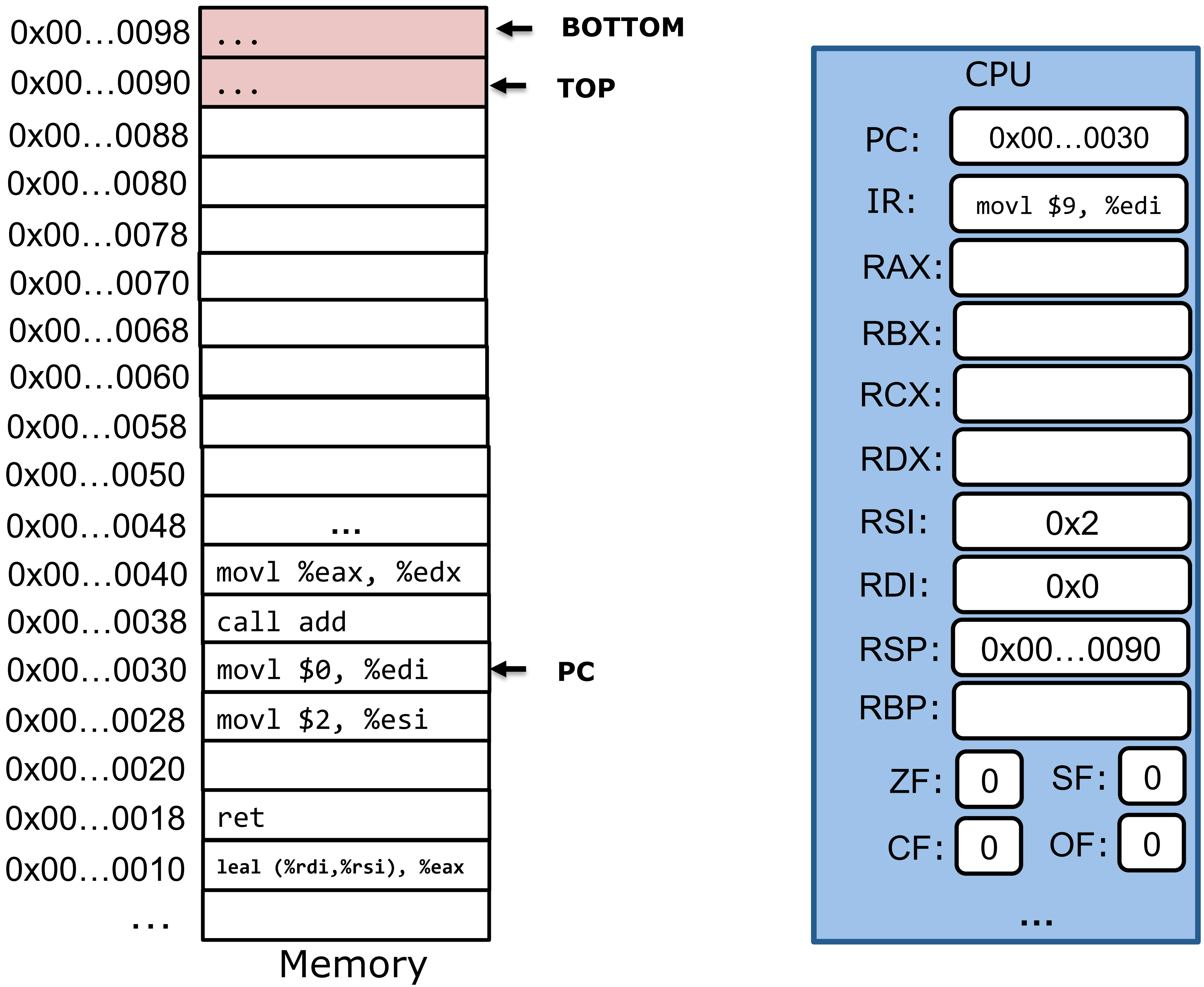
GCC -Og *.c

```
main:
    movl    $2, %esi
    movl    $0, %edi
    call    add
    movl    %eax, %edx
    ...
```

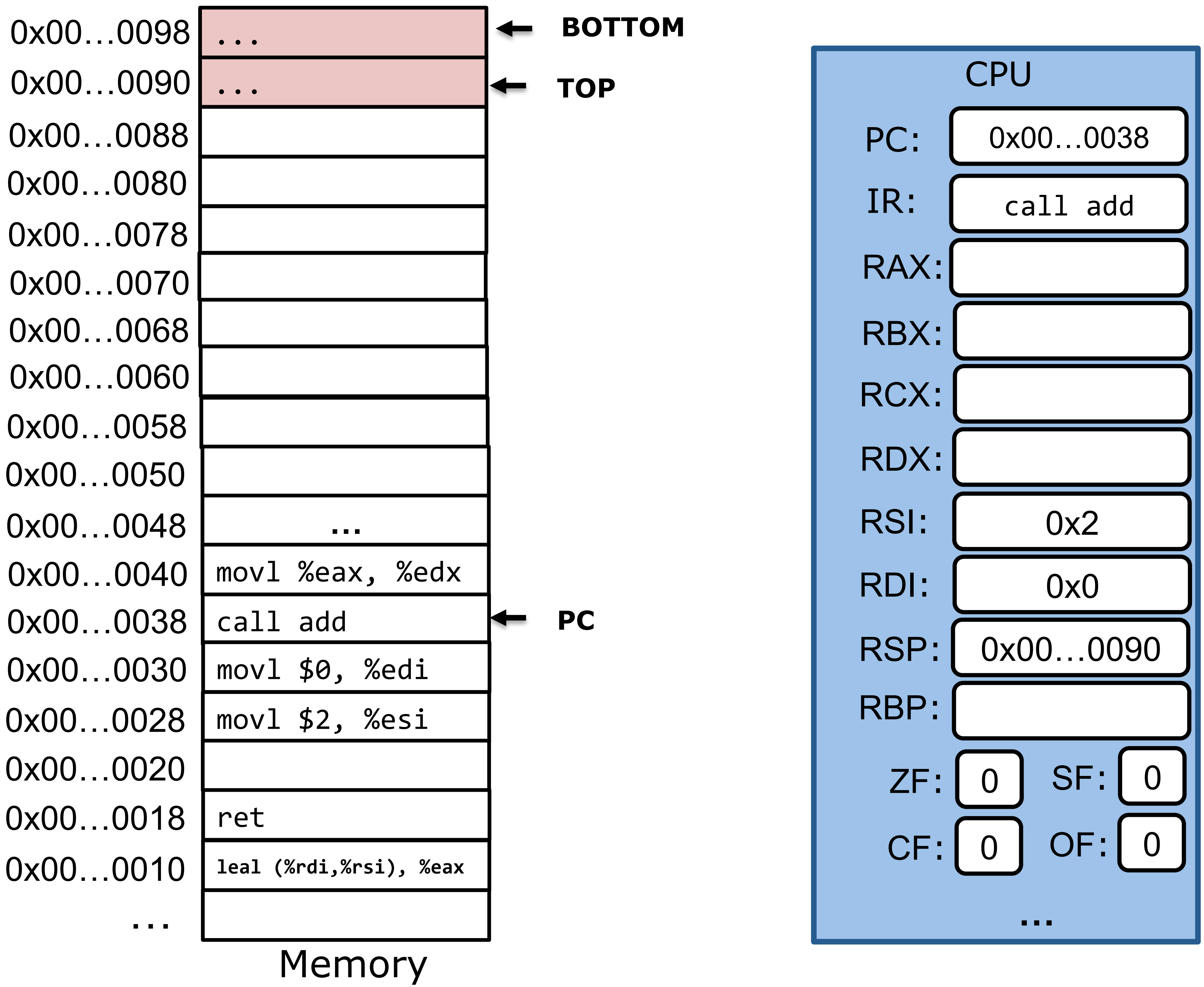


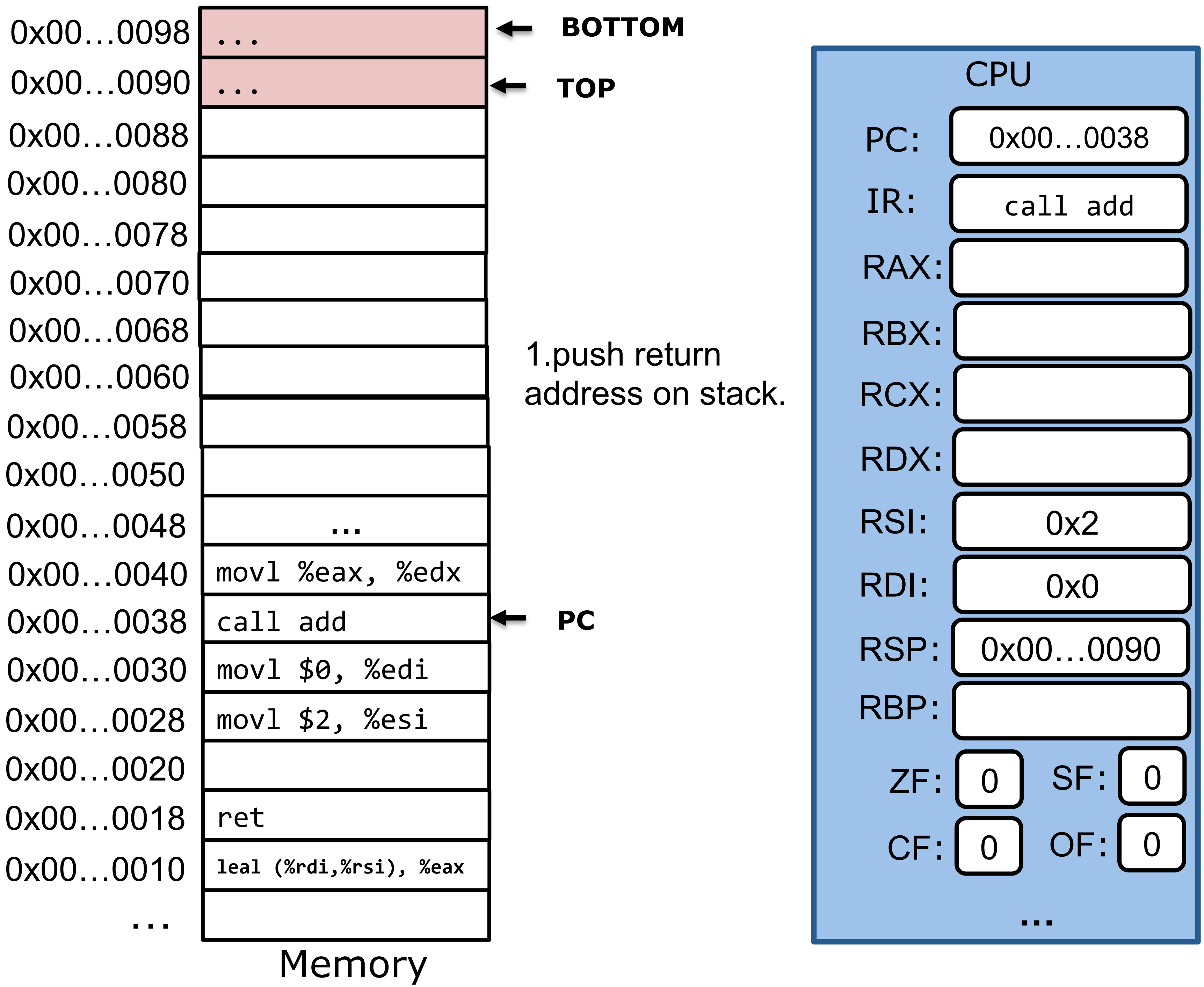
0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
...
...
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
BOTTOM
TOP
PC
Memory
CPU
PC: 0x00…0028
IR: movl $2, %esi
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP: 0x00…0090
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

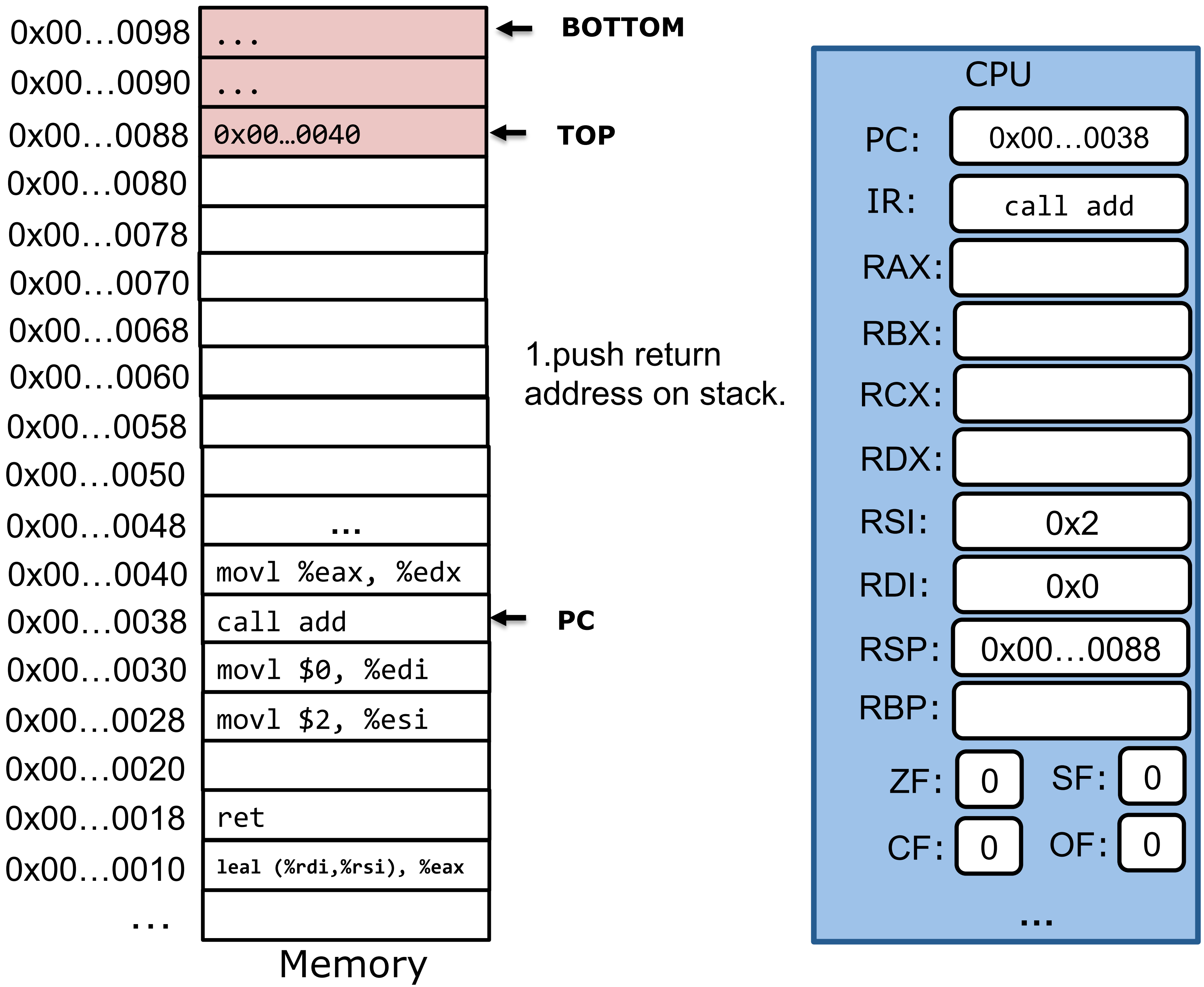

0x00…0098
...
BOTTOM
0x00…0090
...
TOP
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
...
0x00…0040
movl %eax, %edx
0x00…0038
call add
0x00…0030
movl $0, %edi
0x00…0028
movl $2, %esi
PC
0x00…0020
0x00…0018
ret
0x00…0010
leal (%rdi,%rsi), %eax
...
Memory
CPU
PC:
0x00…0028
IR:
movl $2, %esi
RAX:
RBX:
RCX:
RDX:
RSI:
0x2
RDI:
RSP:
0x00…0090
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...

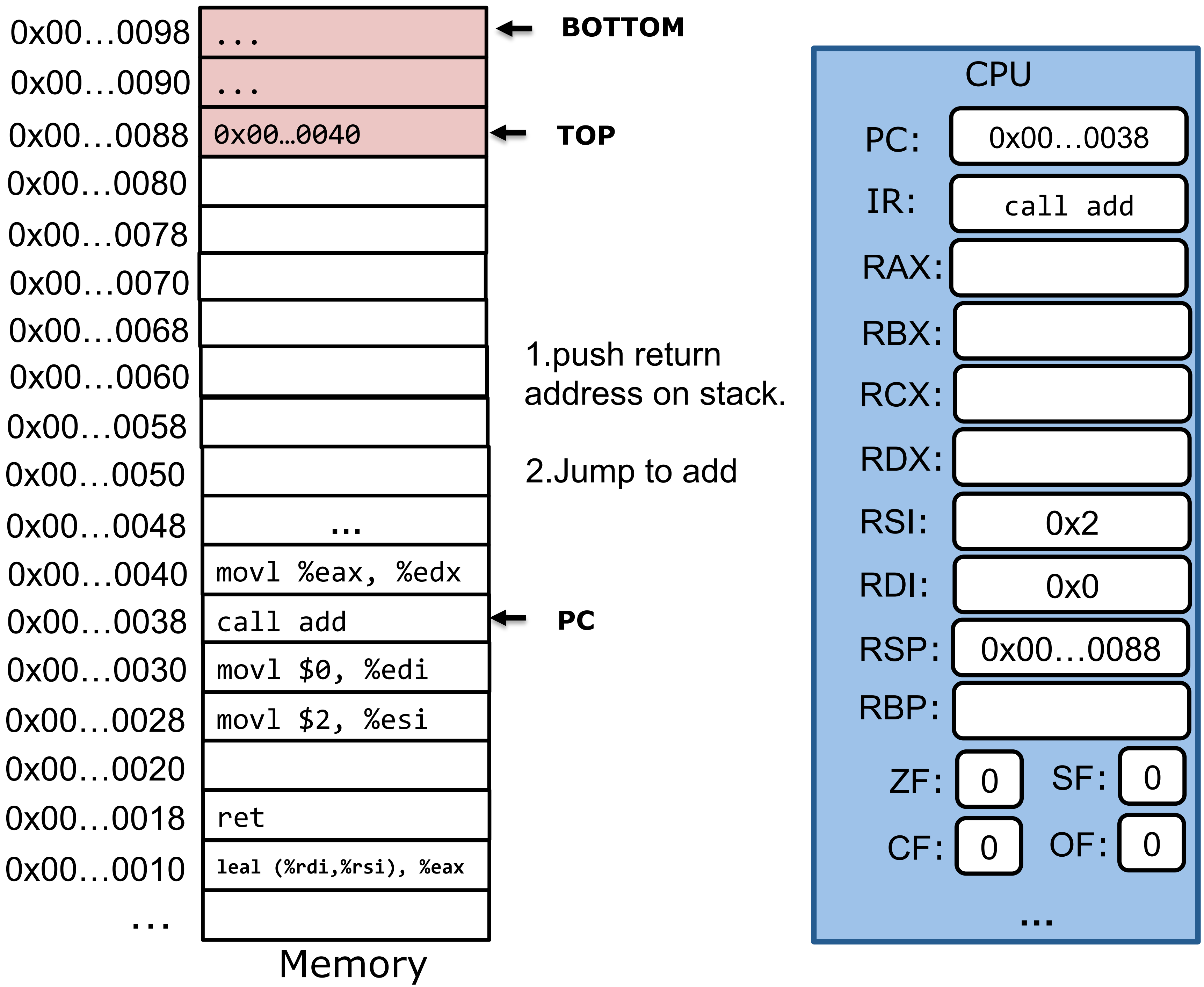


0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
...
...
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
BOTTOM
TOP
PC
Memory
CPU
PC: 0x00…0030
IR: movl $9, %edi
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0090
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

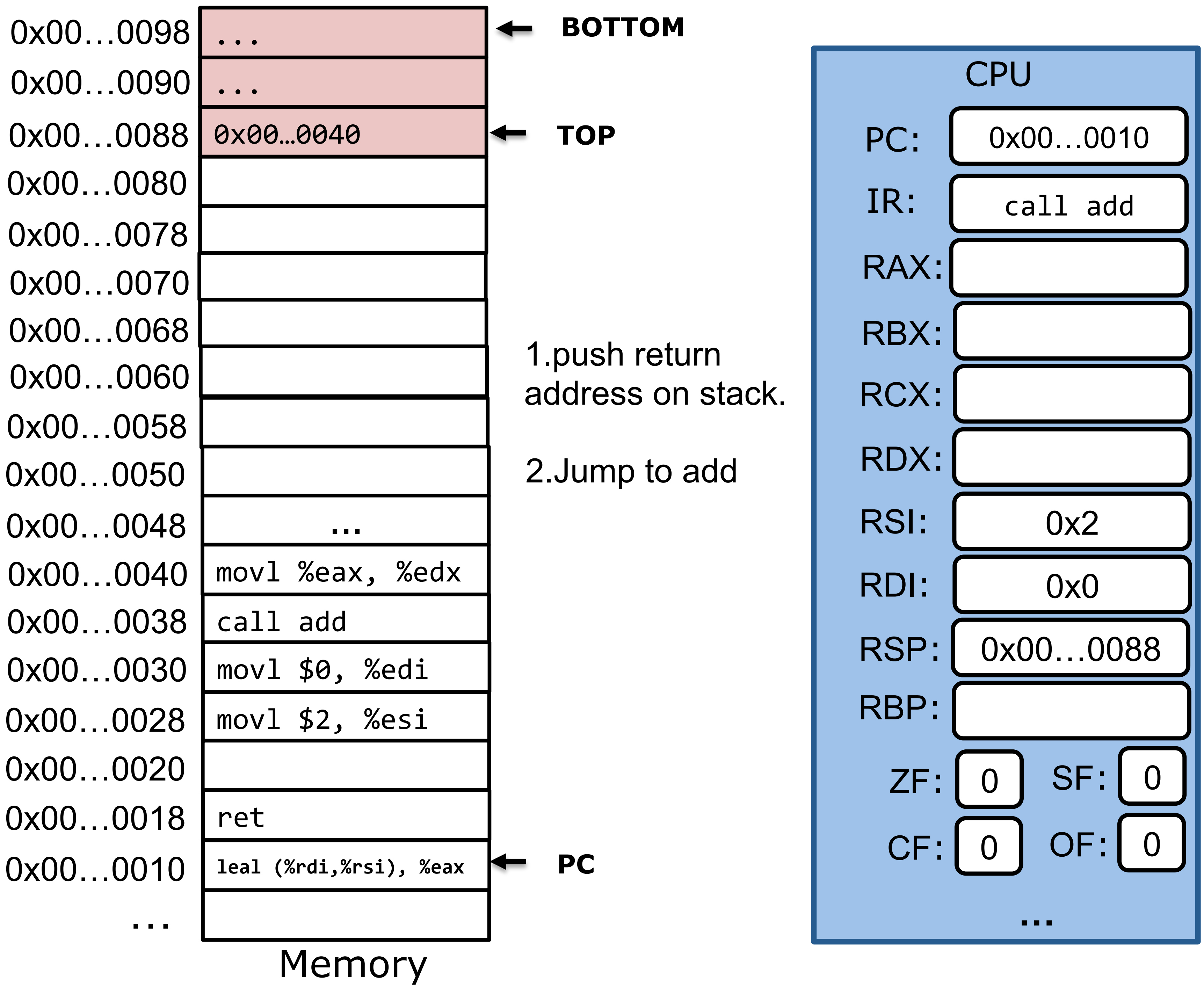

0x00…0098
...
BOTTOM
0x00…0090
...
TOP
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
...
0x00…0040
movl %eax, %edx
0x00…0038
call add
PC
0x00…0030
movl $0, %edi
0x00…0028
movl $2, %esi
0x00…0020
0x00…0018
ret
0x00…0010
leal (%rdi,%rsi), %eax
...
Memory
CPU
PC: 0x00…0038
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0090
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

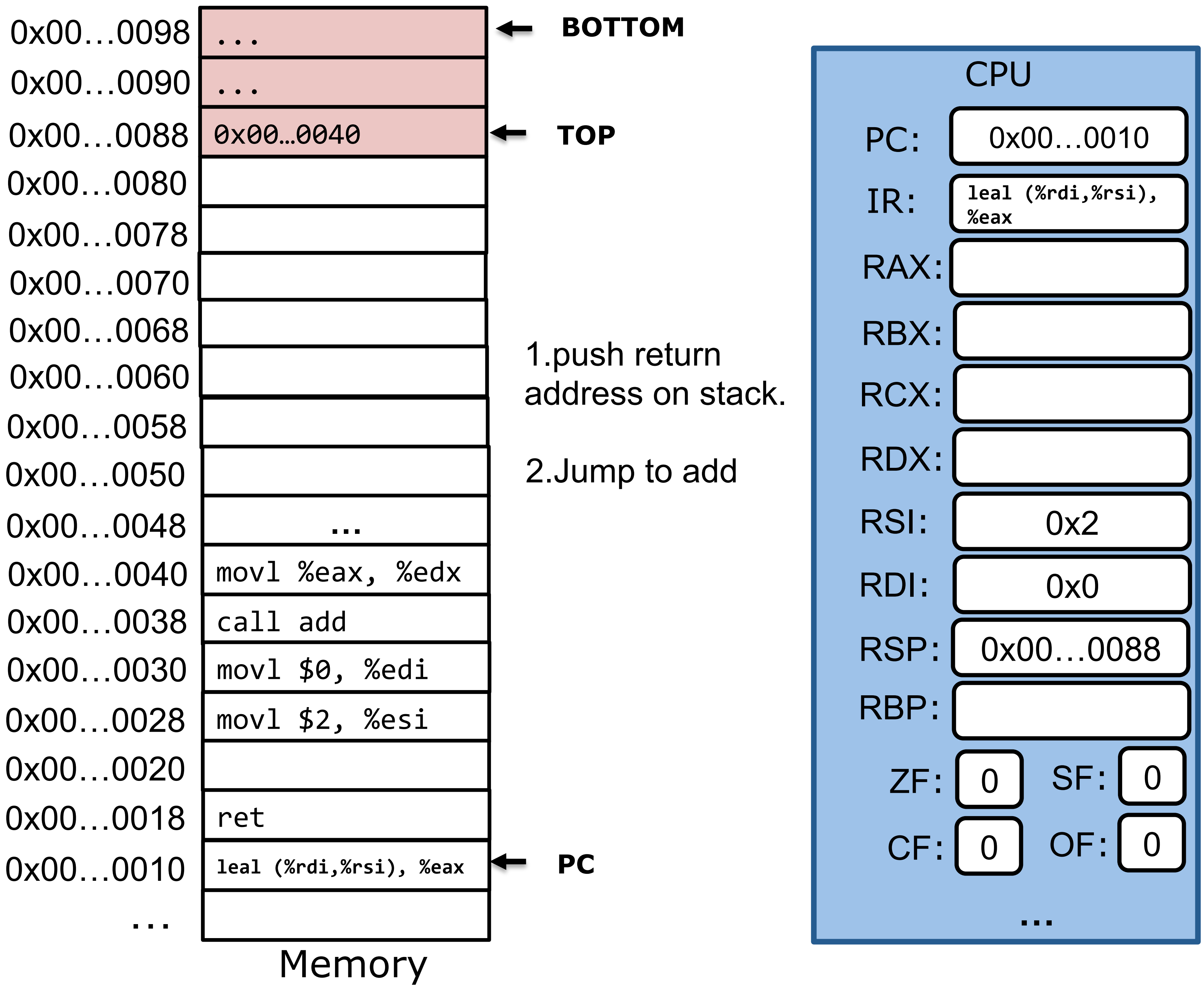

0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
...
...
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
Memory
BOTTOM
TOP
PC
1.push return address on stack.
CPU
PC: 0x00…0038
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0090
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...


0x00…0098
...
BOTTOM
0x00…0090
...
0x00…0088
0x00…0040
TOP
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
...
0x00…0040
movl %eax, %edx
0x00…0038
call add
PC
0x00…0030
movl $0, %edi
0x00…0028
movl $2, %esi
0x00…0020
0x00…0018
ret
0x00…0010
leal (%rdi,%rsi), %eax
...
Memory
1.push return address on stack.
CPU
PC: 0x00…0038
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

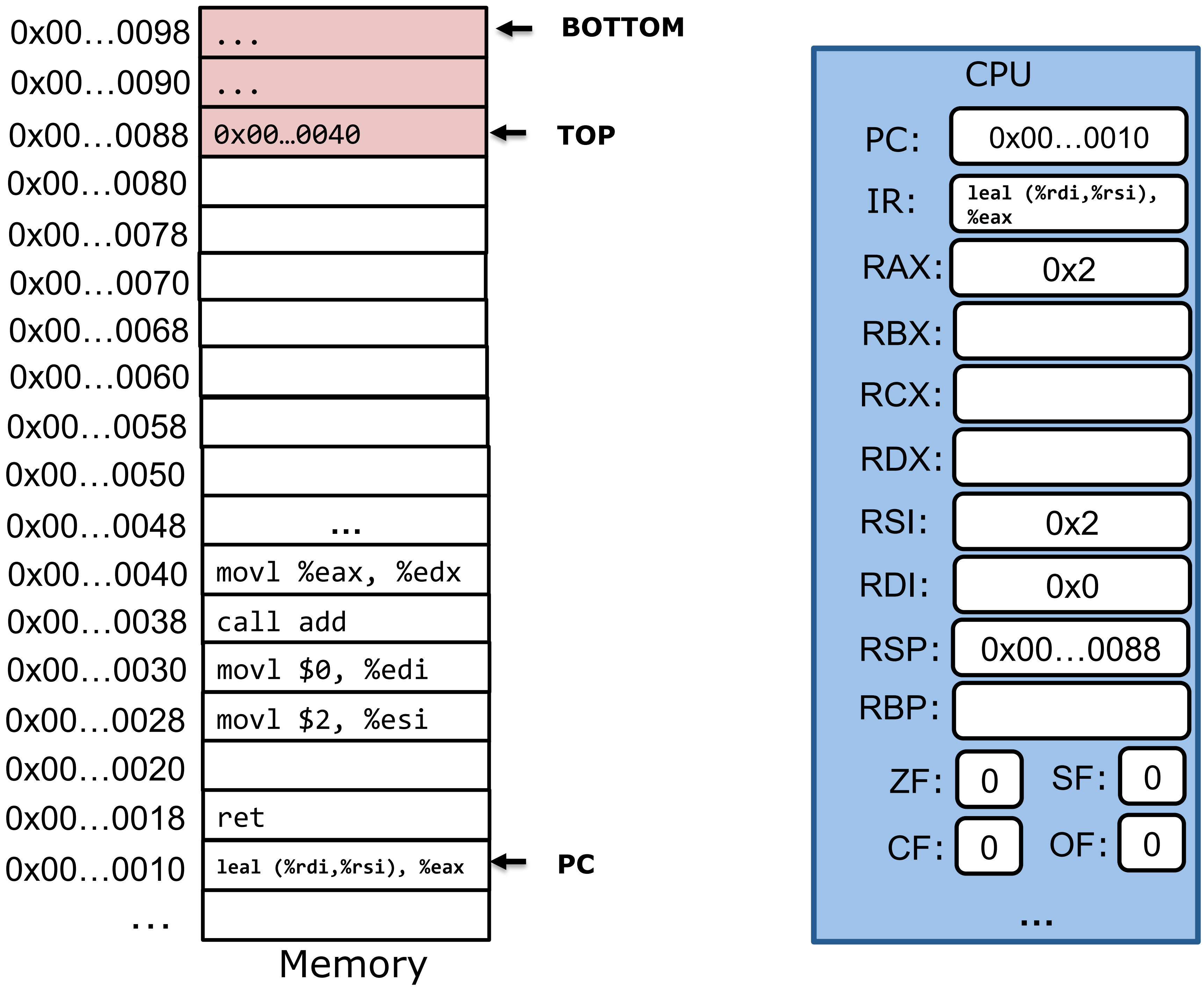


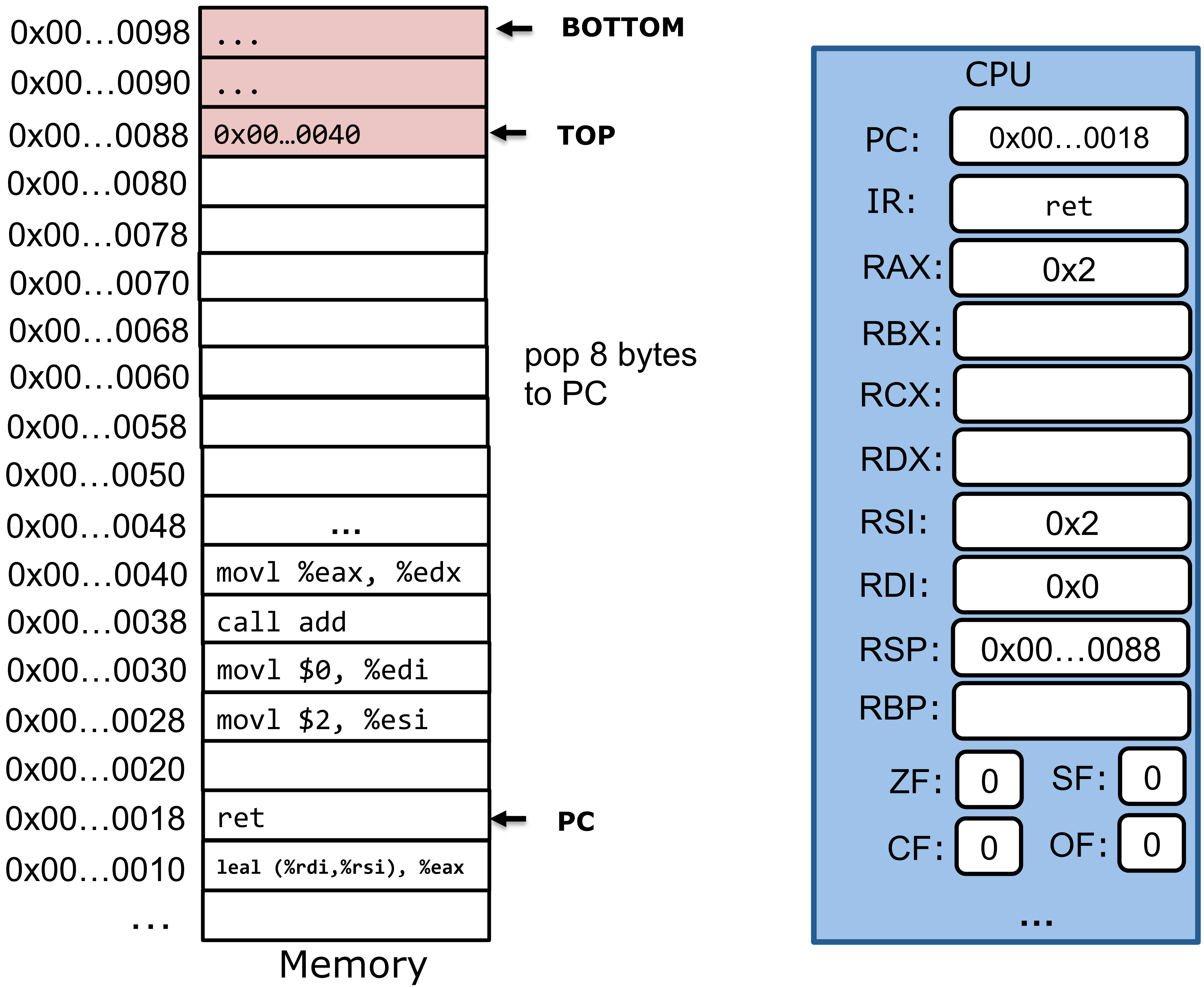
0x00…0098
...
BOTTOM
0x00…0090
...
0x00…0088
0x00…0040
TOP
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
...
0x00…0040
movl %eax, %edx
0x00…0038
call add
PC
0x00…0030
movl $0, %edi
0x00…0028
movl $2, %esi
0x00…0020
0x00…0018
ret
0x00…0010
leal (%rdi,%rsi), %eax
...
Memory
1.push return address on stack.
2.Jump to add
CPU
PC: 0x00…0038
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

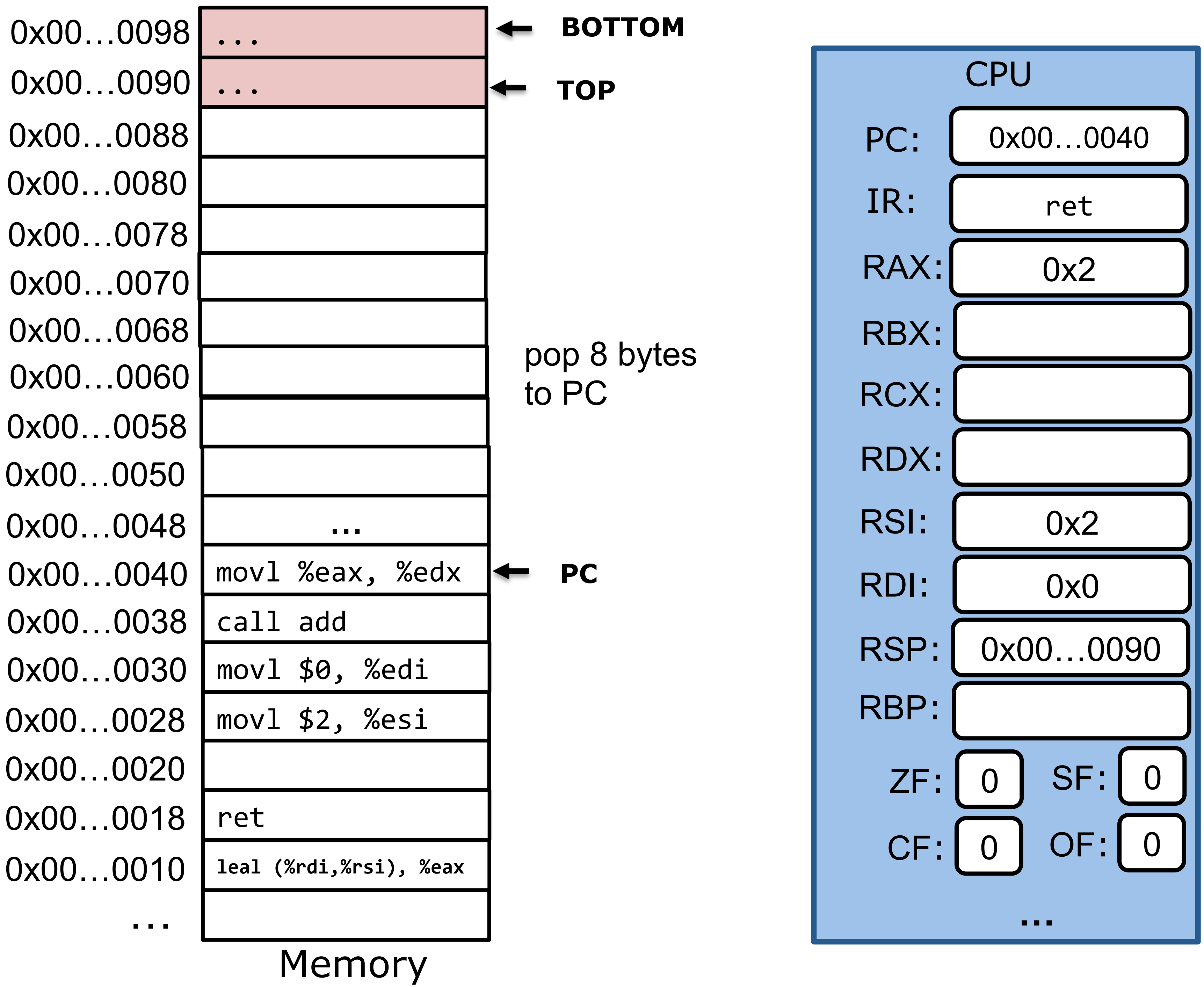

0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
...
0x00…0040
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
Memory
BOTTOM
TOP
PC
1.push return address on stack.
2.Jump to add
CPU
PC: 0x00…0010
IR: call add
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...



0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
...
0x00…0040
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
Memory
BOTTOM
TOP
PC
1.push return address on stack.
2.Jump to add
CPU
PC: 0x00…0010
IR: leal (%rdi,%rsi), %eax
RAX:
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...


0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
0x00…0040
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
Memory
BOTTOM
TOP
PC
CPU
PC: 0x00…0010
IR: leal (%rdi,%rsi), %eax
RAX: 0x2
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...


0x00…0098
0x00…0090
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
0x00…0040
0x00…0038
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
...
0x00…0040
...
movl %eax, %edx
call add
movl $0, %edi
movl $2, %esi
ret
leal (%rdi,%rsi), %eax
Memory
BOTTOM
TOP
pop 8 bytes
to PC
PC
CPU
PC: 0x00…0018
IR: ret
RAX: 0x2
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0088
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0



0x00…0098
...
BOTTOM
0x00…0090
...
TOP
0x00…0088
0x00…0080
0x00…0078
0x00…0070
0x00…0068
0x00…0060
0x00…0058
0x00…0050
0x00…0048
...
0x00…0040
movl %eax, %edx
PC
0x00…0038
call add
0x00…0030
movl $0, %edi
0x00…0028
movl $2, %esi
0x00…0020
0x00…0018
ret
0x00…0010
leal (%rdi,%rsi), %eax
...
Memory
pop 8 bytes to PC
CPU
PC: 0x00…0040
IR: ret
RAX: 0x2
RBX:
RCX:
RDX:
RSI: 0x2
RDI: 0x0
RSP: 0x00…0090
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0
...

Where to store function arguments and return values?

- Hardware does not dictate where arguments and return value are stored
  - It’s up to the software (compilers).
- Where to put arguments/return value?
  - Arguments and return value are like local variables
  - They are allocated when function is called, de-allocated when function returns.
  - Must do such allocation/de-allocation very fast

Where to store function arguments and return values?

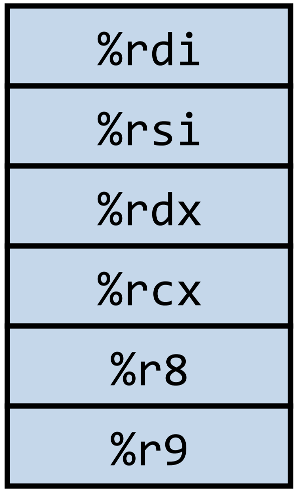
- Two possible designs:
  - Store everything on stack
  - Use registers ← Registers are much faster than memory but there are only a few of them

- The chosen design → the calling convention
  - All code on a computer system must obey the same convention
  - Otherwise, libraries won't work

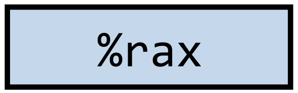
C/UNIX's calling convention

Registers

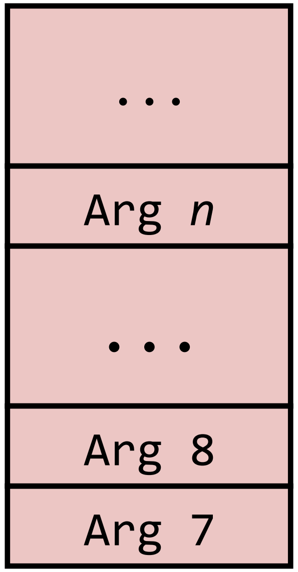
First 6 arguments



Return value



Stack



Only allocate stack space when needed

C's calling convention: args/return values

Registers

- First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %r9
- Return value: %rax

```
int add(int a, int b, int c, int d, int e, int f, int g, int h) {
  int r = a + b + c + d + e + f + g + h;
  return r;
}

int main() {

    int c = add(1, 2, 3, 4, 5, 6, 7, 8);
    printf("%d\b", c);
    return 0;
}
```

C's calling convention: args/return values

```
int add(int a, int b, int c, int d, int e, int f, int g, int h) {
  int r = a + b + c + d + e + f + g + h;
  return r;
}
```

```
main:
    pushq   $8
    pushq   $7
    movl    $6, %r9d
    movl    $5, %r8d
    movl    $4, %ecx
    movl    $3, %edx
    movl    $2, %esi
    movl    $1, %edi
    call    add
```

```
add:
    addl    %esi, %edi
    addl    %edi, %edx
    addl    %edx, %ecx
    addl    %r8d, %ecx
    addl    %r9d, %ecx
    movl    %ecx, %eax
    addl    8(%rsp), %eax
    addl    16(%rsp), %eax
    ret
```

8(%rsp) stores g

16(%rsp) stores h

what does (%rsp) store?

How to allocate/deallocate local variables?

Use registers whenever possible

Allocate local variables on the stack

- `subq  $0x8,%rsp` //allocate 8 bytes
- `movq $1,  8(%rsp)` //store 1 in the allocated 8 bytes

Calling convention: Caller vs. callee-save registers

- What can the caller assume about the content of a register across function calls?

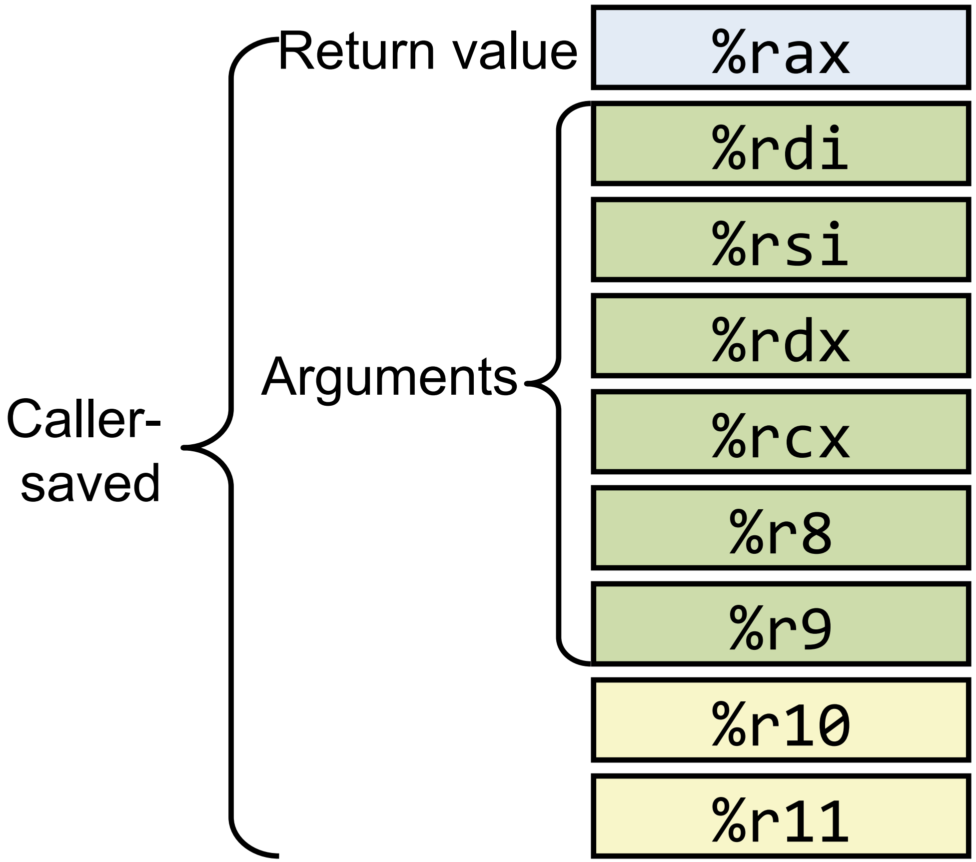
```
int foo() {
   int a;    // suppose a is stored in %r12
   a = .... // compute result of a

   int r = bar();

   int result = r + a; // does %r12 still store the value of a?
   return result;
}
```

Calling convention: register saving

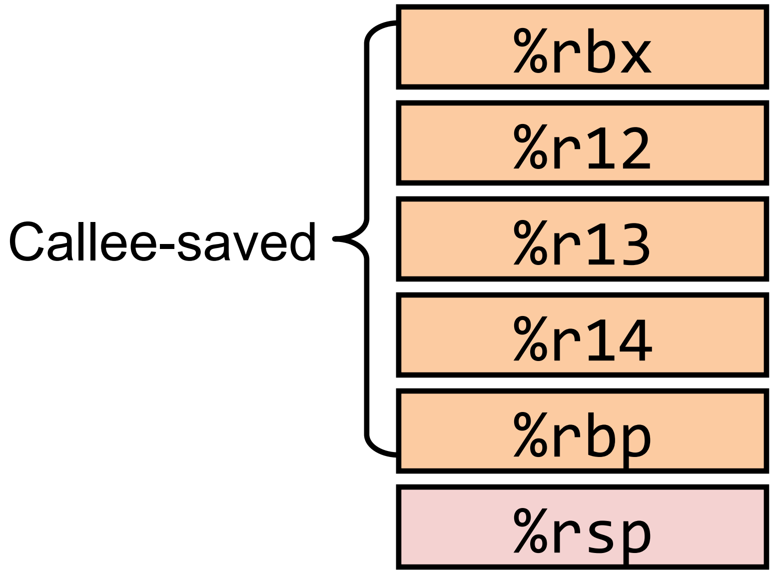
Some registers are "caller saved", others are "callee saved"

- Caller saved
  - Caller saves "caller saved" registers on stack before the call
- Callee saved
  - Callee saves "callee saved" registers on stack before using
  - Callee restores them before returning to caller

C' calling convention: Register Usage



Callee can directly use these registers



Caller can assume these registers are unchanged.

Example

```c
int add2(int a, int b)
{
  return a + b;
}


int add3(int a, int b, int c)
{
  int r = add2(a, b);
  r = r + c;
  return r;
}
```

```
add2:
    leal    (%rdi,%rsi), %eax
    ret


add3:
    pushq   %rbx
    movl    %edx, %ebx
    movl    $0, %eax
    call    add2
    addl    %ebx, %eax
    popq    %rbx
    ret
```

Registers

First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9

Return value: %rax

Example

```c
int add2(int a, int b)
{
  return a + b;
}


int add3(int a, int b, int c)
{
  int r = add2(a, b);
  r = r + c;
  return r;
}
```

```
add2:
    leal    (%rdi,%rsi), %eax
    ret
```

save %rbx (callee-save) before writing it

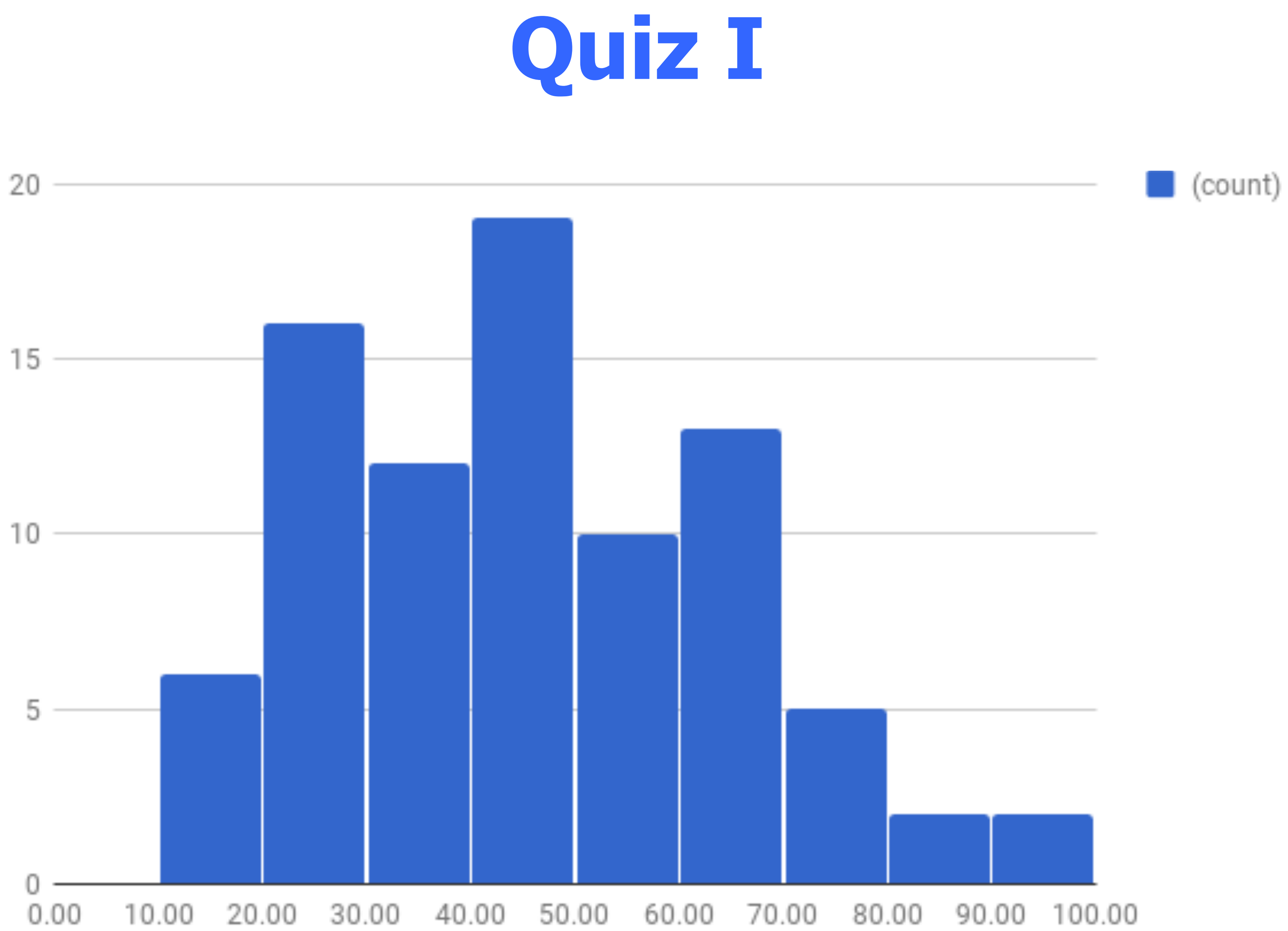
```
add3:
    pushq   %rbx
    movl    %edx, %ebx
    movl    $0, %eax
    call    add2
    addl    %ebx, %eax
    popq    %rbx      # restore %rbx before ret
    ret
```

r is saved to %ebx

Registers

First 6 Arguments: %rdi, %rsi, %rdx, %rcx, %r8, %9

Return value: %rax


Quiz I
(count)
20
15
10
5
0
0.00
10.00
20.00
30.00
40.00
50.00
60.00
70.00
80.00
90.00
100.00