Machine Program: Procedure

Jinyang Li

What we've learnt about how hardware runs a program?

- Basic hardware execution:
  - CPU fetch next instructions from memory according to %rip
  - Decode and execute instruction (e.g. mov, add)
  - CPU updates %rip to point to next instruction
- Modes of execution:
  - Sequential:
    - PC (%rip) is changed to point to the next instruction
  - Control flow: jmp, conditional jmp
    - PC (%rip) is changed to point to the jump target address

Today's lesson plan

- How x86 supports function call
  - Role of stack
  - Call / ret
  - Calling convention (where args/ret-vals are stored)

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: ..
}
```

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

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

Jump to f()
Remember where to come back

L1

How to transfer control for procedure calls?

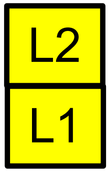
```
void main(){
    ..
    f(..)
L1: ..
}
```

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

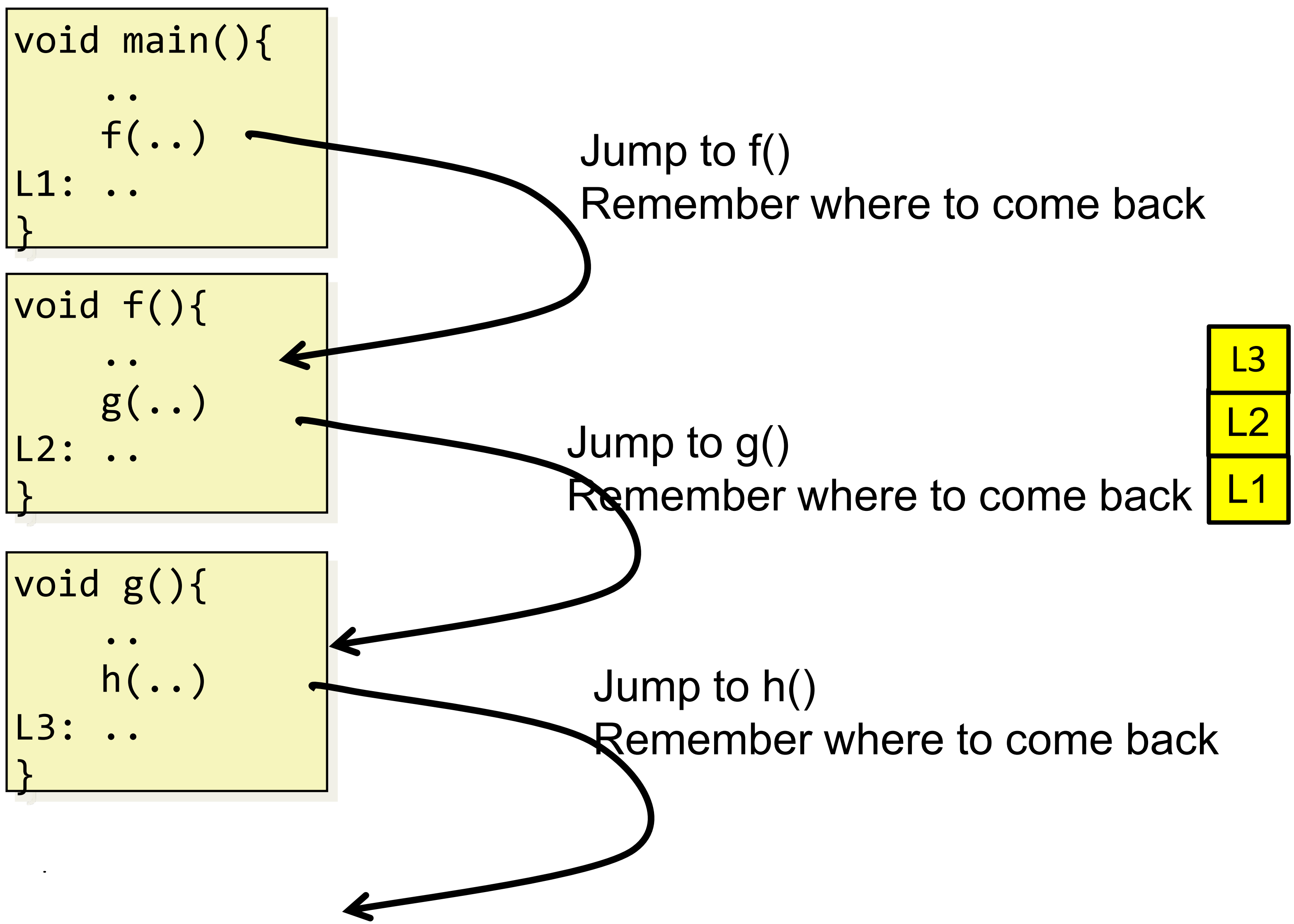
```
void g(){
    ..
    h(..)
L3: ..
}
```

Jump to f()
Remember where to come back

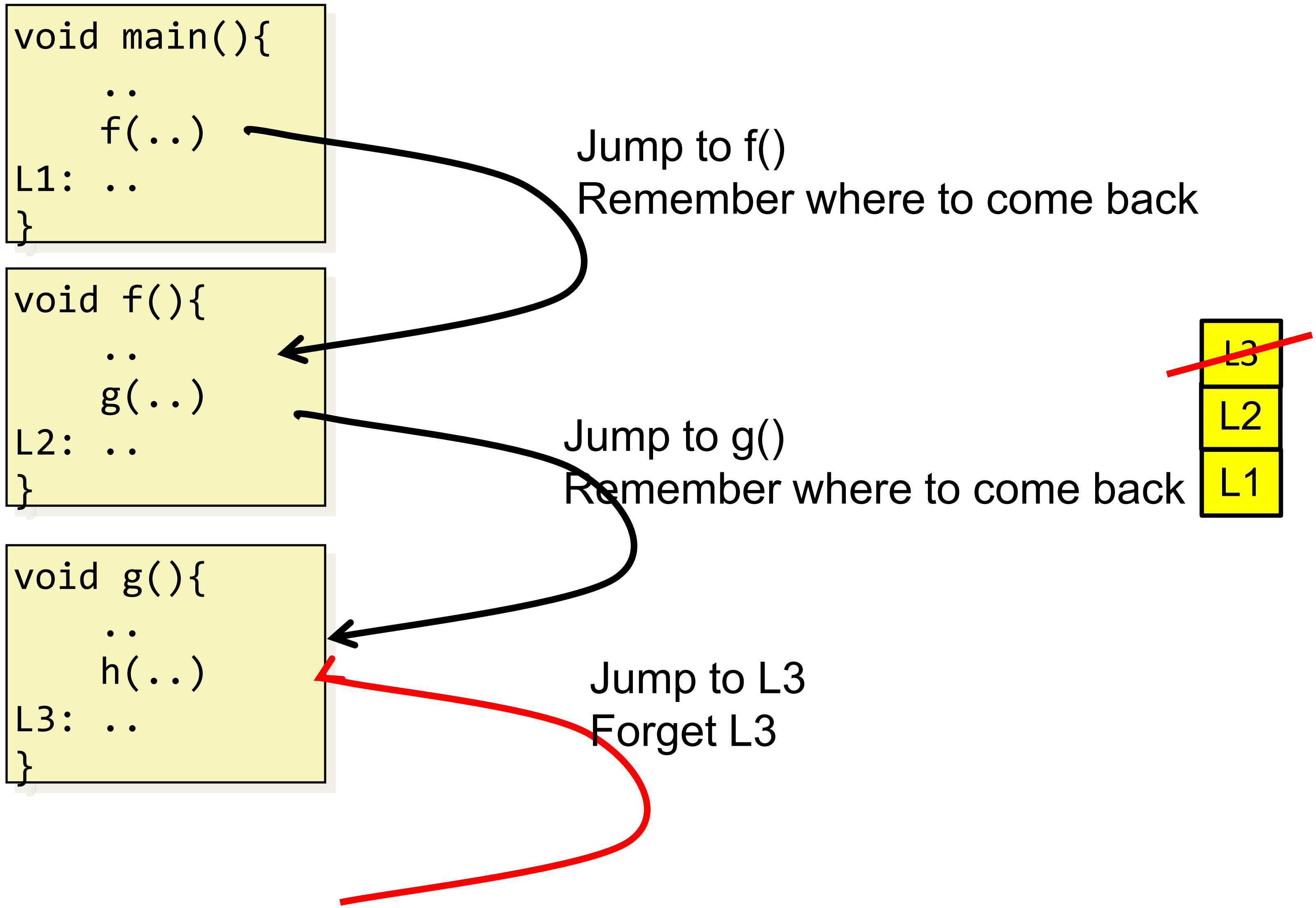
Jump to g()
Remember where to come back



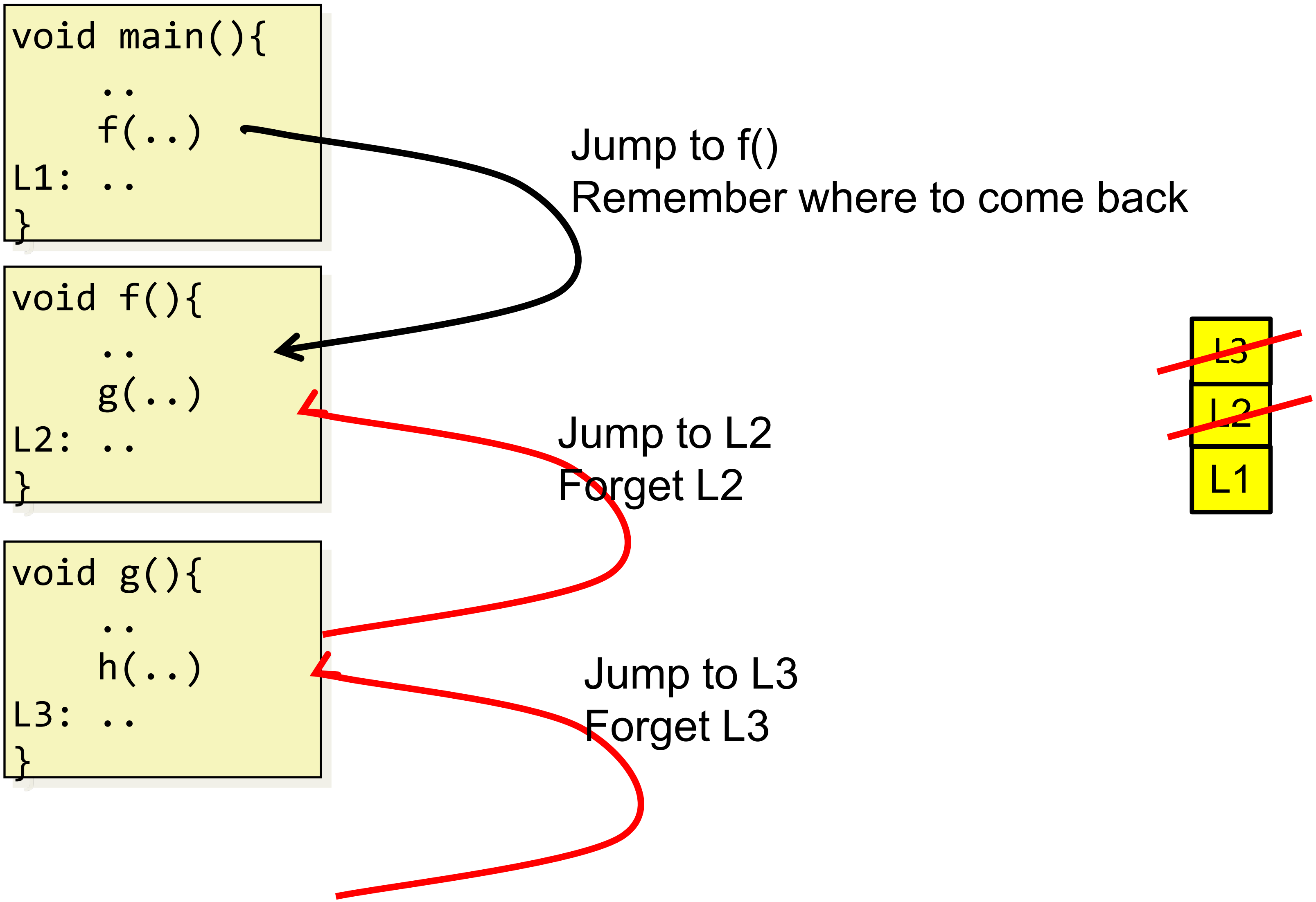
How to transfer control for procedure calls?



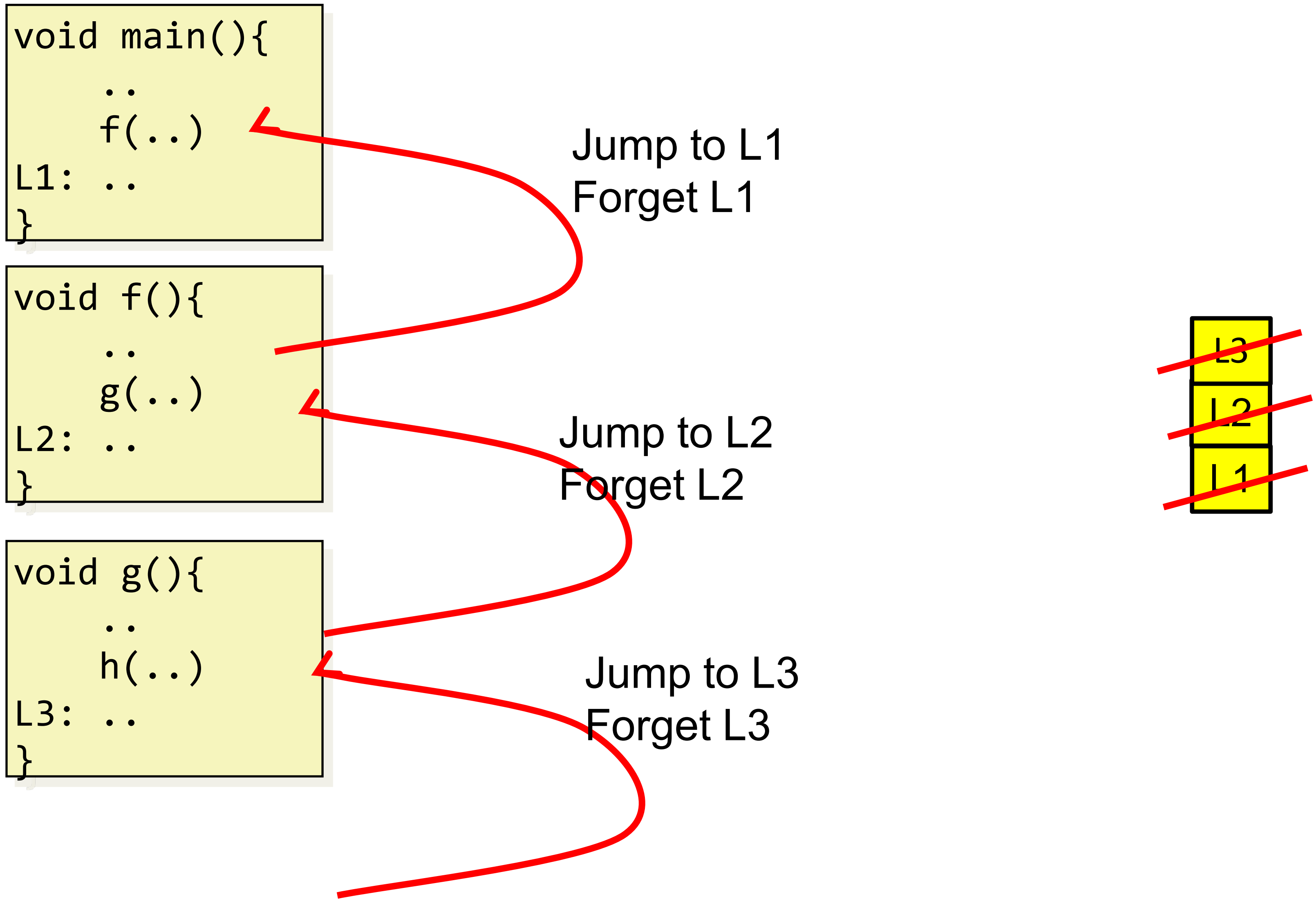
How to transfer control for procedure calls?



How to transfer control for procedure calls?



How to transfer control for procedure calls?

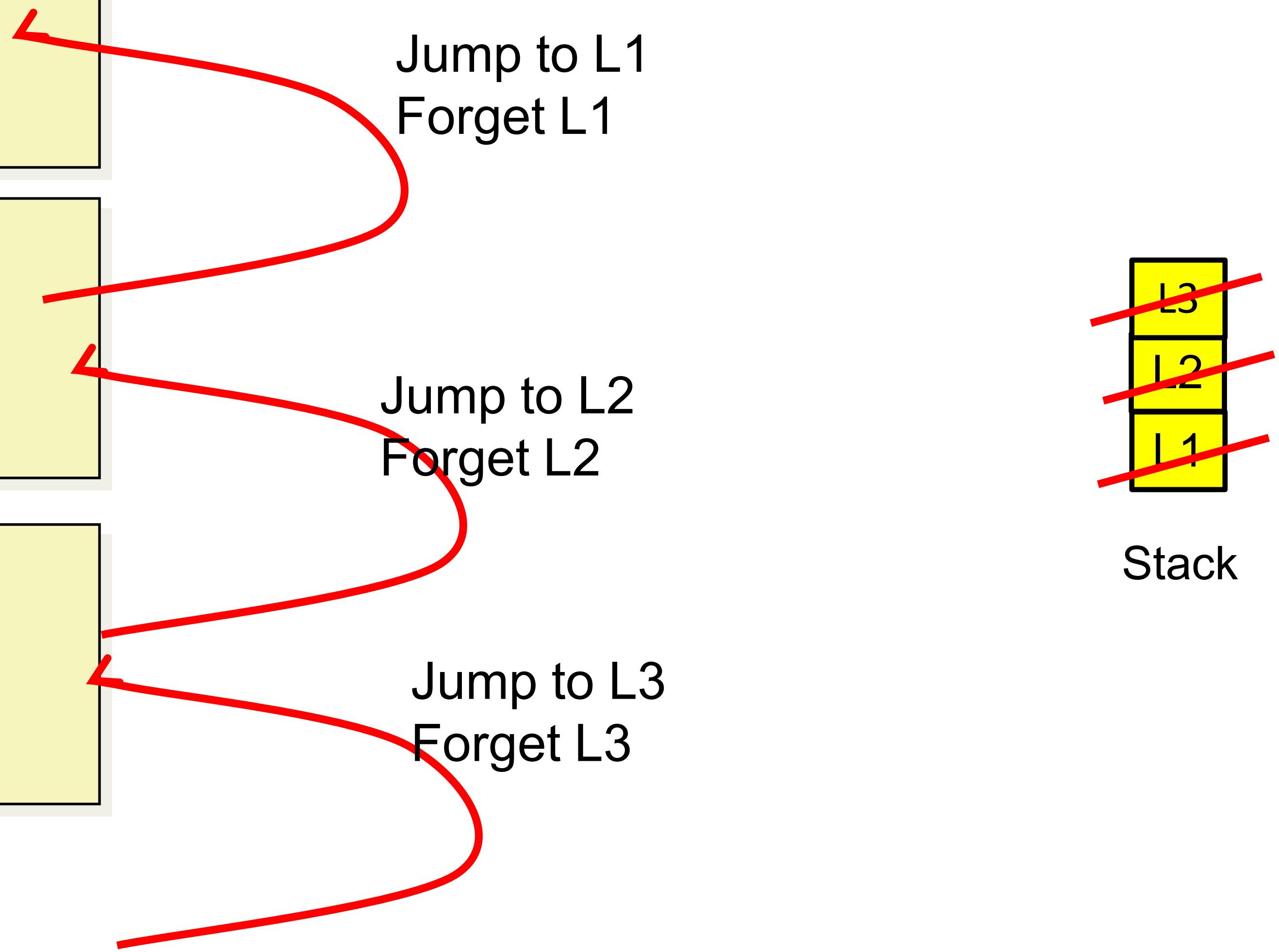


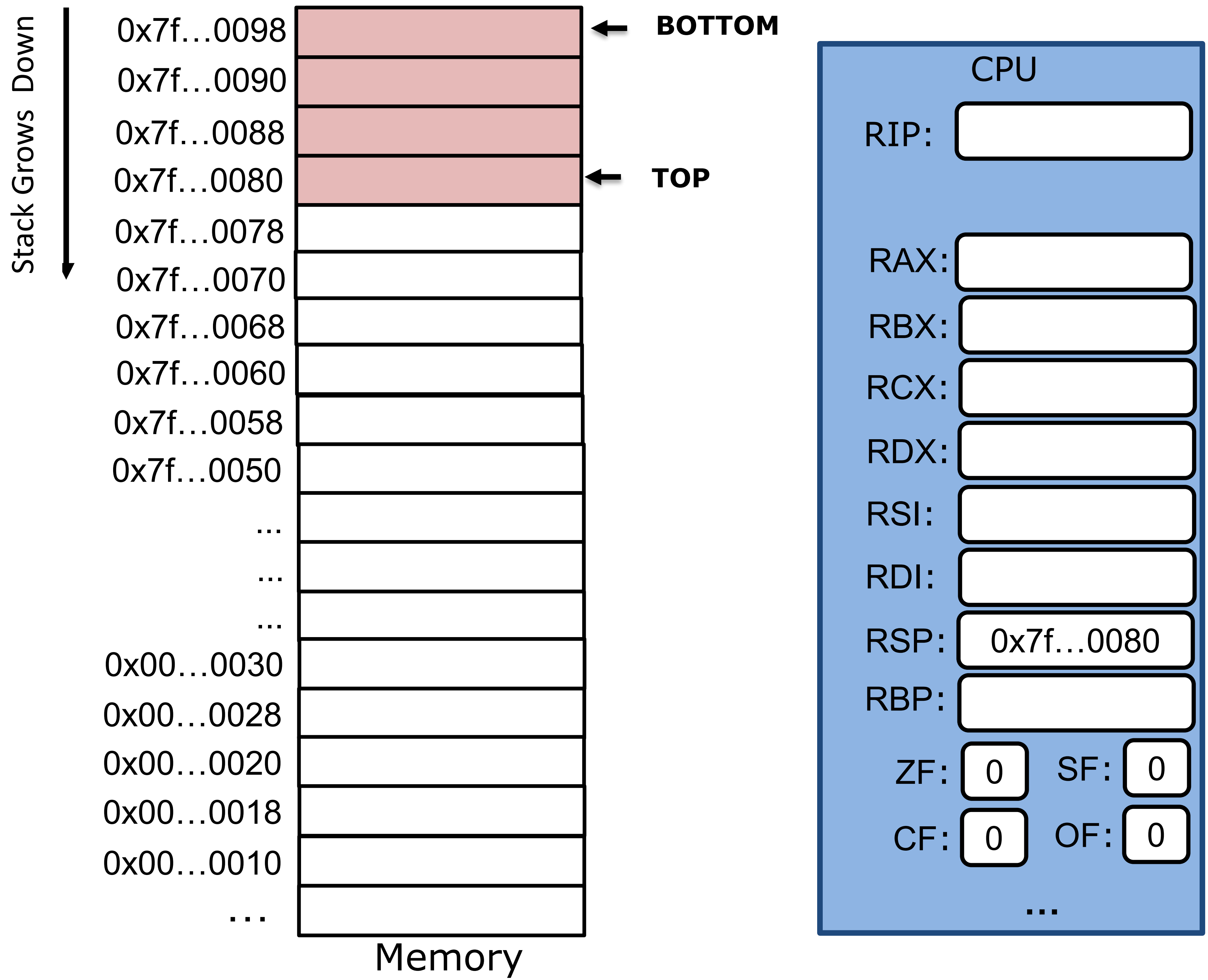
How to transfer control for procedure calls?

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

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

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



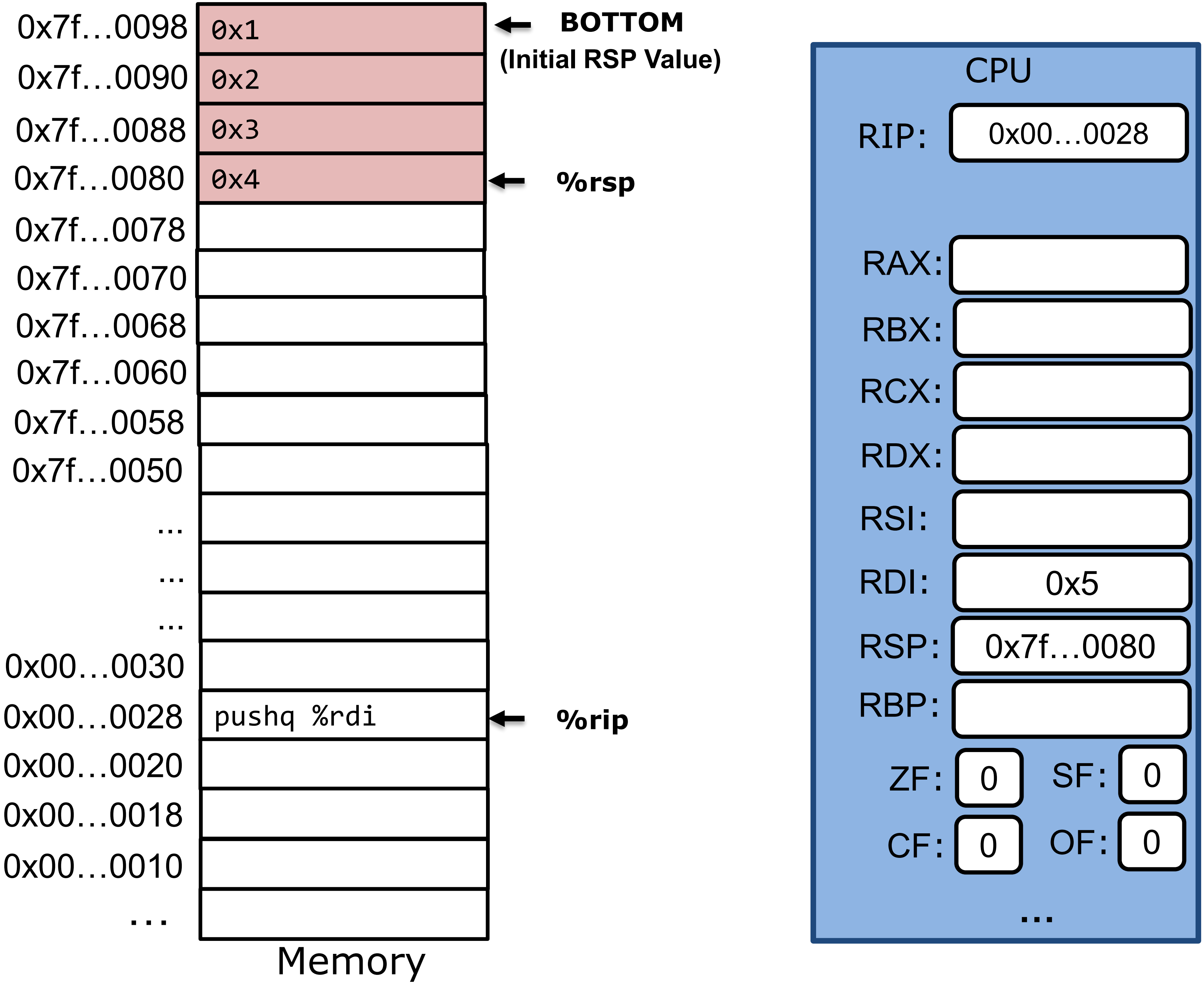


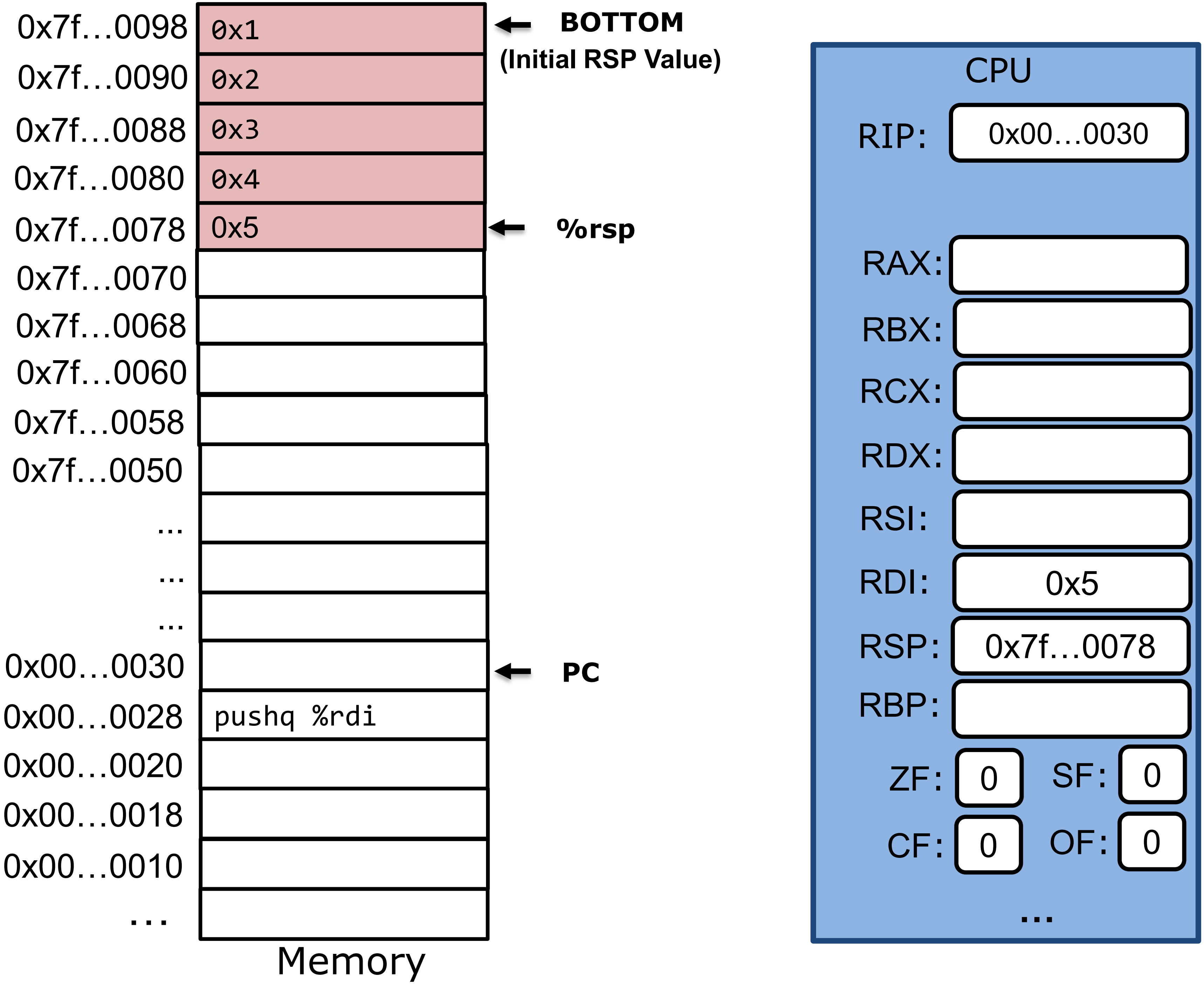
Stack Grows Down
0x7f…0098
0x7f…0090
0x7f…0088
0x7f…0080
0x7f…0078
0x7f…0070
0x7f…0068
0x7f…0060
0x7f…0058
0x7f…0050
…
…
…
0x00…0030
0x00…0028
0x00…0020
0x00…0018
0x00…0010
…
Memory
BOTTOM
TOP
CPU
RIP:
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…0080
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

Stack – push Instruction

`pushq` `src`

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


0x7f…0098
0x1
BOTTOM
(Initial RSP Value)
0x7f…0090
0x2
0x7f…0088
0x3
0x7f…0080
0x4
%rsp
0x7f…0078
0x7f…0070
0x7f…0068
0x7f…0060
0x7f…0058
0x7f…0050
...
0x00…0030
0x00…0028
pushq %rdi
%rip
0x00…0020
0x00…0018
0x00…0010
Memory
CPU
RIP: 0x00…0028
RAX:
RBX:
RCX:
RDX:
RSI:
RDI: 0x5
RSP: 0x7f…0080
RBP:
ZF: 0
SF: 0
CF: 0
OF: 0

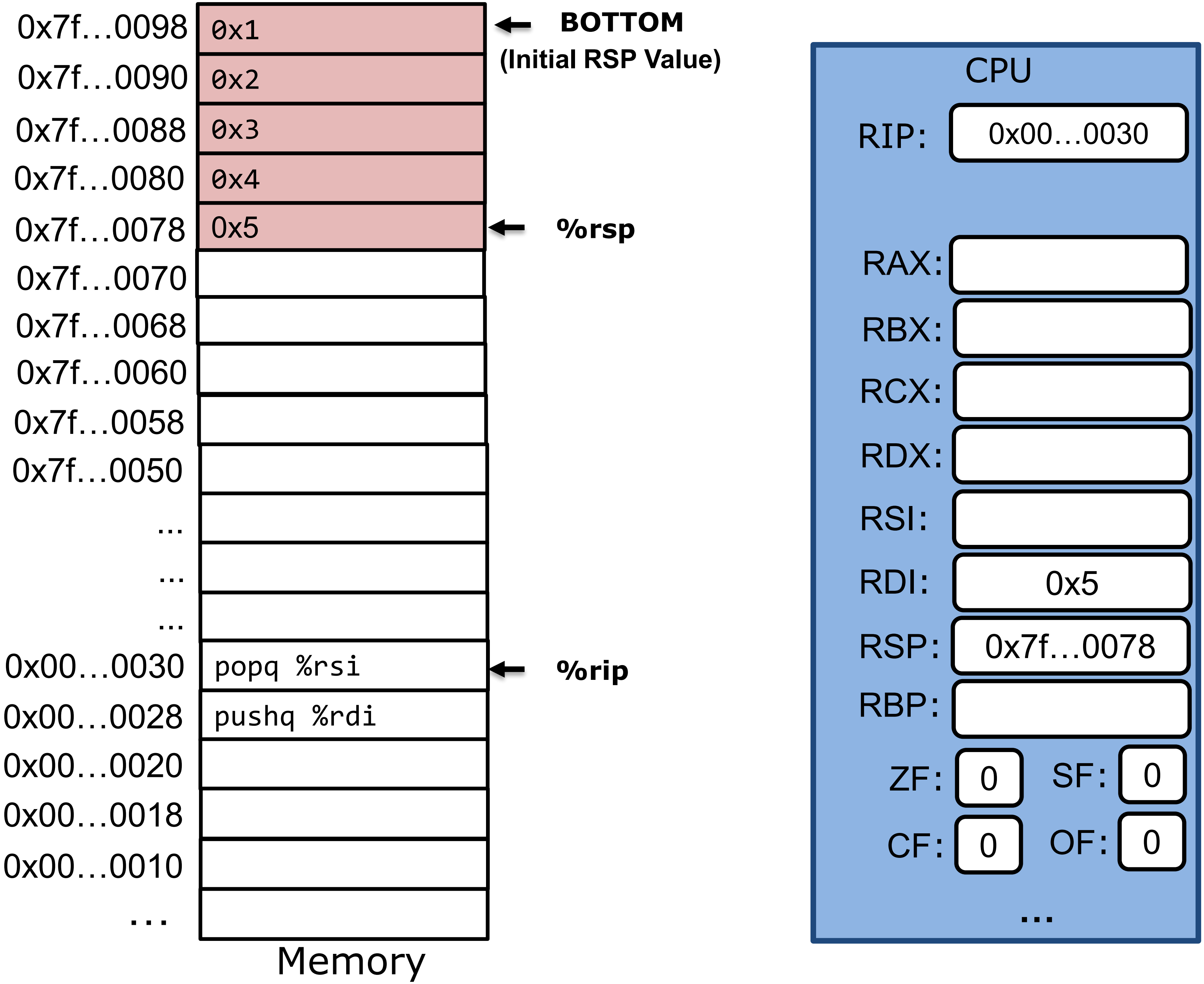


0x7f…0098
0x1
BOTTOM
(Initial RSP Value)
0x7f…0090
0x2
0x7f…0088
0x3
0x7f…0080
0x4
0x7f…0078
0x5
%rsp
0x7f…0070
0x7f…0068
0x7f…0060
0x7f…0058
0x7f…0050
…
…
…
0x00…0030
PC
0x00…0028
pushq %rdi
0x00…0020
0x00…0018
0x00…0010
…
Memory
CPU
RIP:
0x00…0030
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
0x5
RSP:
0x7f…0078
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

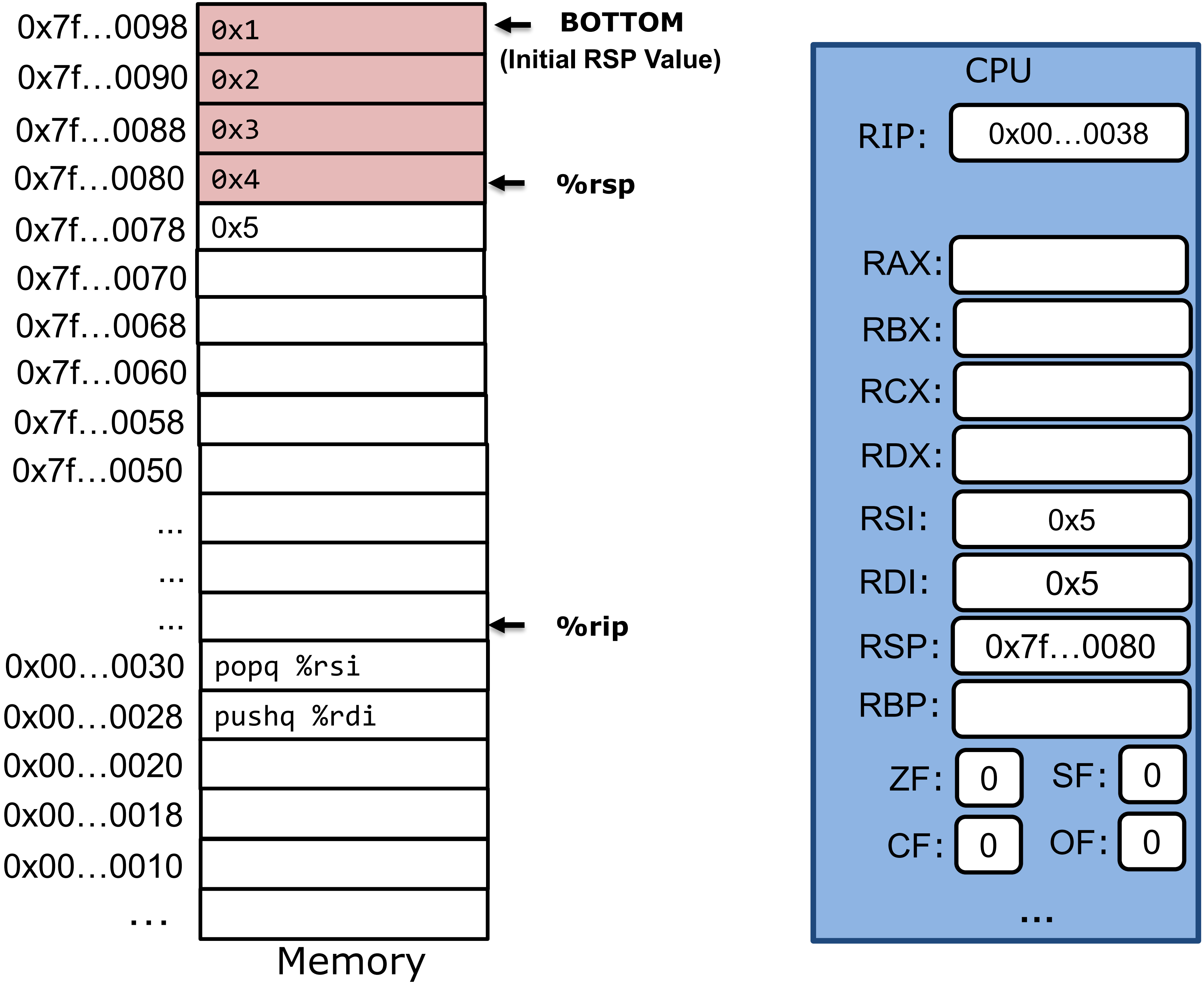
Stack – pop Instruction

popq `dest`

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



0x7f…0098
0x1
BOTTOM
(Initial RSP Value)
0x7f…0090
0x2
0x7f…0088
0x3
0x7f…0080
0x4
0x7f…0078
0x5
%rsp
0x7f…0070
0x7f…0068
0x7f…0060
0x7f…0058
0x7f…0050
…
…
…
0x00…0030
popq %rsi
%rip
0x00…0028
pushq %rdi
0x00…0020
0x00…0018
0x00…0010
…
Memory
CPU
RIP:
0x00…0030
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
0x5
RSP:
0x7f…0078
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…


0x7f…0098
0x1
BOTTOM
(Initial RSP Value)
0x7f…0090
0x2
0x7f…0088
0x3
0x7f…0080
0x4
%rsp
0x7f…0078
0x5
0x7f…0070
0x7f…0068
0x7f…0060
0x7f…0058
0x7f…0050
…
…
…
%rip
0x00…0030
popq %rsi
0x00…0028
pushq %rdi
0x00…0020
0x00…0018
0x00…0010
…
Memory
CPU
RIP:
0x00…0038
RAX:
RBX:
RCX:
RDX:
RSI:
0x5
RDI:
0x5
RSP:
0x7f…0080
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
…

call/ret : control transfer from caller to callee and vice versa

`call` `label`

- Push return address on stack
  - return_address = the instruction immediately after **`call`**
  - `%rsp=%rsp-8, mem[%rsp]=return_addr`
- Jump to the address of the label
  - Label points to the first instruction of the function

`ret`

- Pop 8 bytes from the stack to %rip
  - `%rip = mem[%rsp], %rsp = %rsp +8`

call/ret : control transfer from caller to callee and vice versa

```c
int count = 0;

void inc() {
  count++;
}

int main() {
    inc();
}
```

gcc -Og -S test.c

```
inc:
    addl $0x1, count(%rip)
    ret
main:
    ...
    call      add
    movl $0, %eax
    ...
```

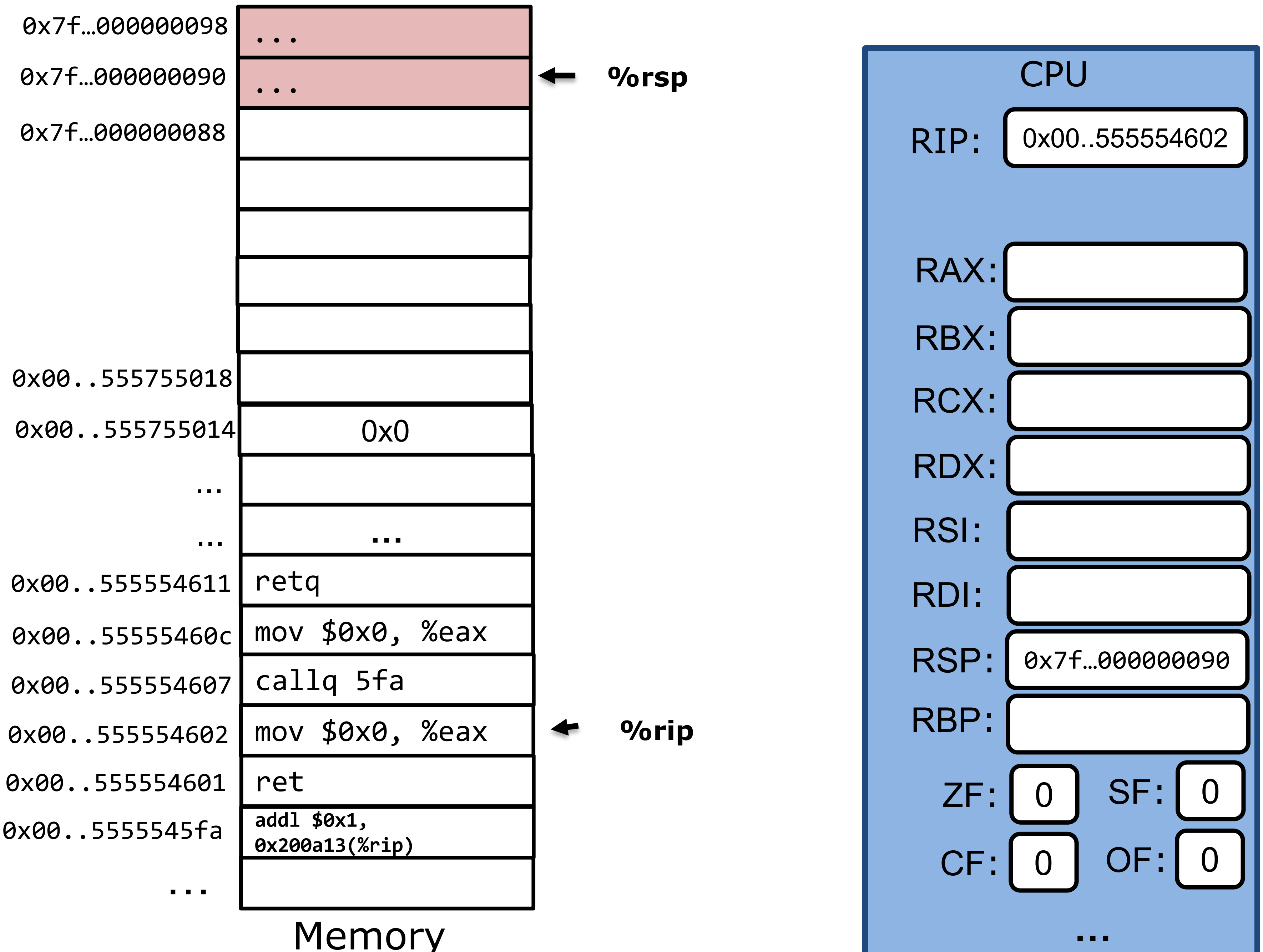
return address points to this instruction

Call instruction: control transfer from caller to callee

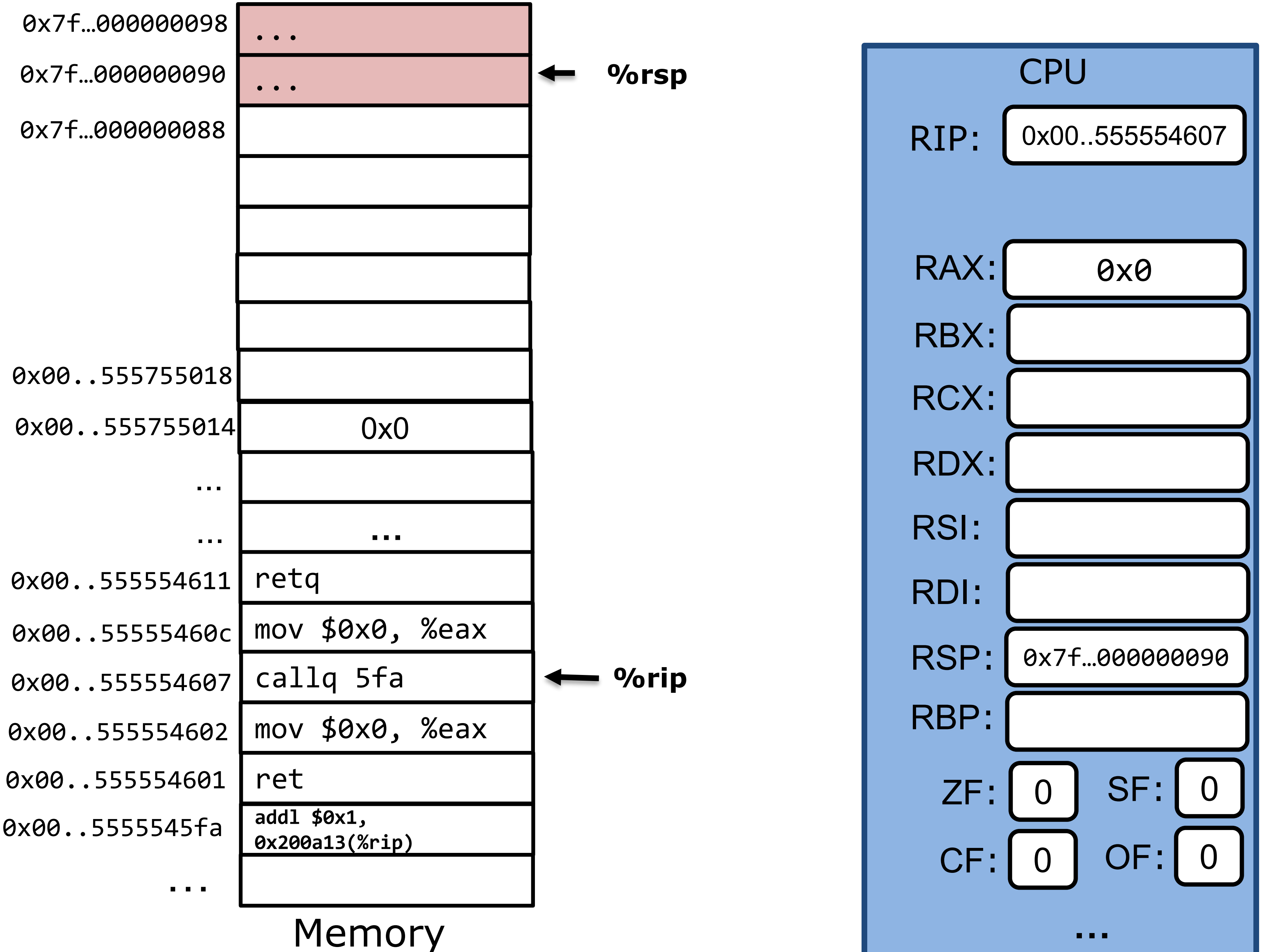
```
gcc -Og test.c
objdump -d a.out
```

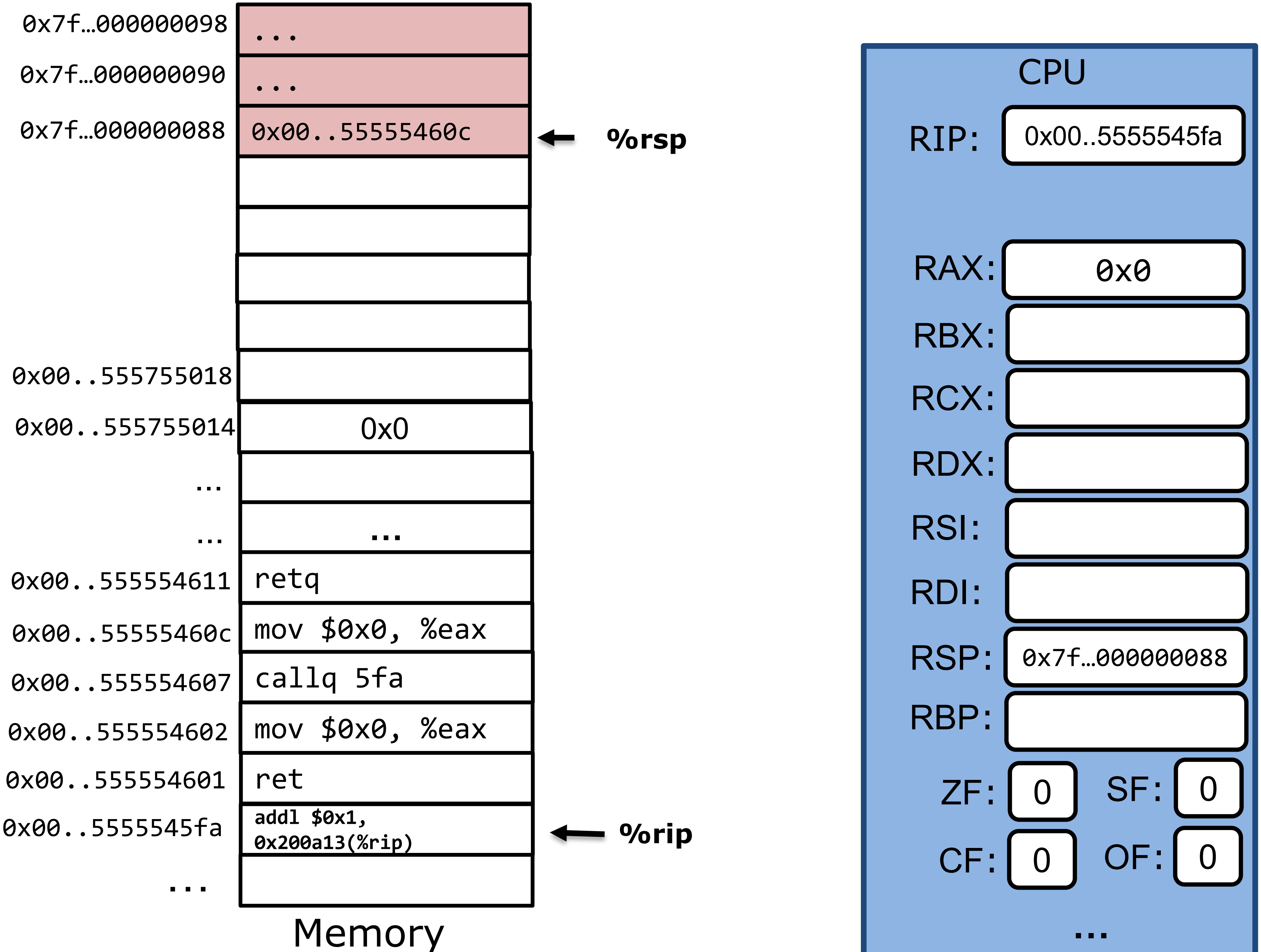
```
int count = 0;

void inc() {
  count++;
}

int main() {
    inc();
}
```

```
0000000000005fa <inc>:
             5fa: 83 05 13 0a 20 00 01  addl   $0x1, 0x200a13(%rip)
             601: c3                    retq

0000000000000602 <main>:
             602: b8 00 00 00 00   mov   $0x0,%eax
             607: e8 ee ff ff ff   callq 5fa <inc>
             60c: b8  00 00 00 00  mov   $0x0,%eax
             611: c3               retq
```

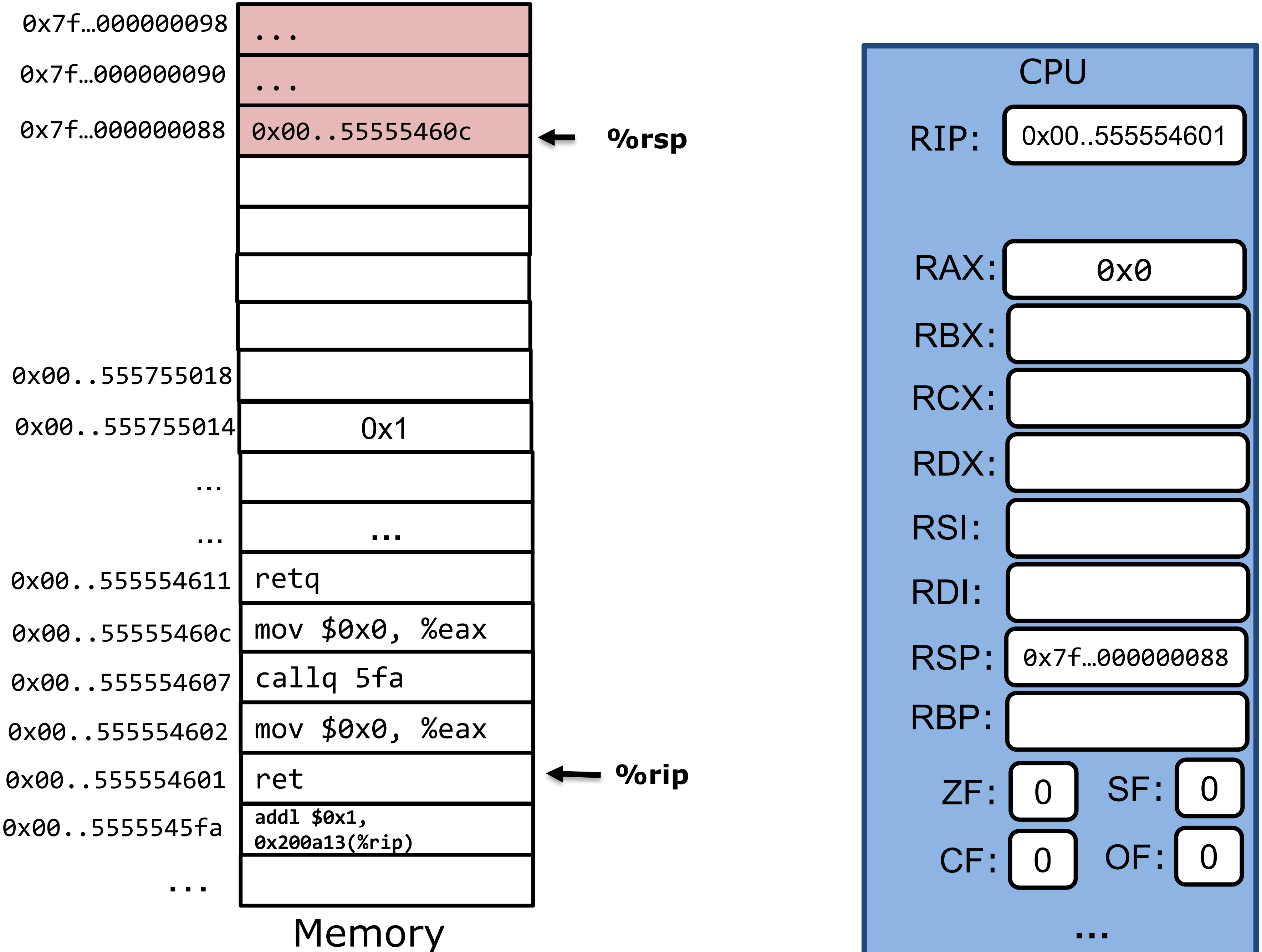


0x7f…000000098
...
0x7f…000000090
...
%rsp
0x7f…000000088
0x00..555755018
0x00..555755014
0x0
...
...
...
0x00..555554611
retq
0x00..55555460c
mov $0x0, %eax
0x00..555554607
callq 5fa
0x00..555554602
mov $0x0, %eax
%rip
0x00..555554601
ret
0x00..5555545fa
addl $0x1, 0x200a13(%rip)
...
Memory
CPU
RIP:
0x00..555554602
RAX:
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000090
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...

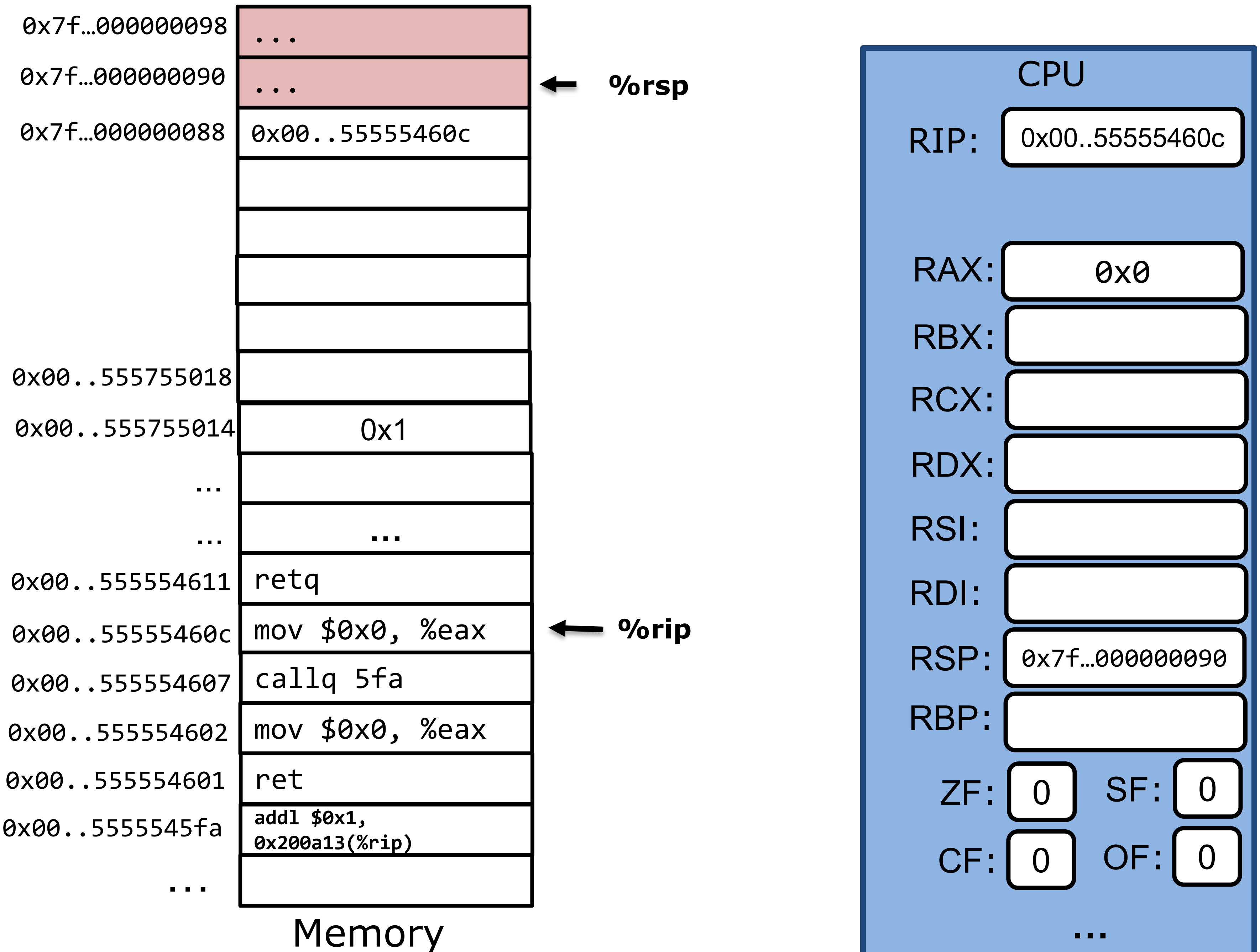

0x7f…000000098
...
0x7f…000000090
...
%rsp
0x7f…000000088
0x00..555755018
0x00..555755014
0x0
...
...
...
0x00..555554611
retq
0x00..55555460c
mov $0x0, %eax
0x00..555554607
callq 5fa
%rip
0x00..555554602
mov $0x0, %eax
0x00..555554601
ret
0x00..5555545fa
addl $0x1, 0x200a13(%rip)
...
Memory
CPU
RIP:
0x00..555554607
RAX:
0x0
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000090
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...



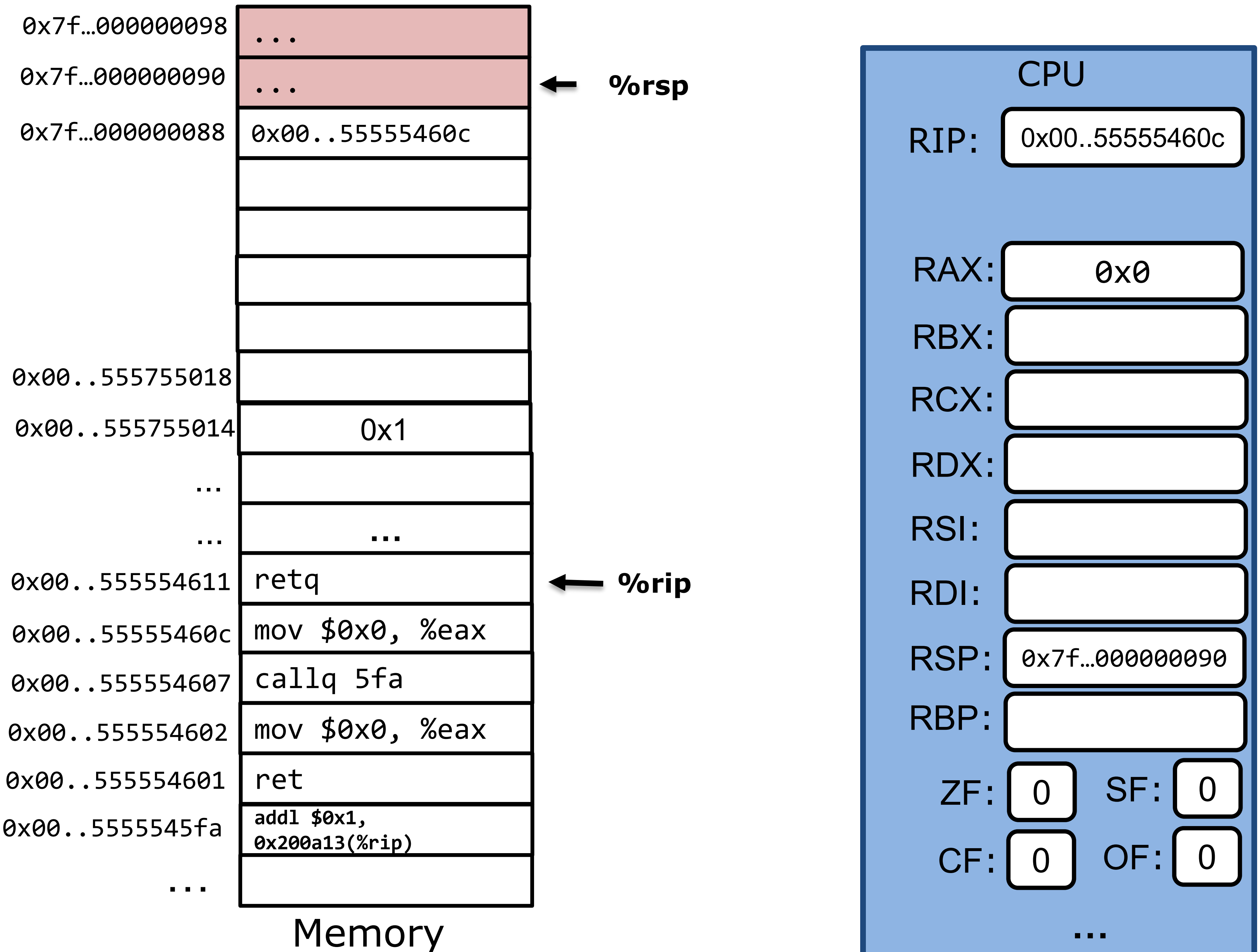
0x7f…000000098
...
0x7f…000000090
...
0x7f…000000088
0x00..55555460c
%rsp
0x00..555755018
0x00..555755014
0x0
...
...
...
0x00..555554611
retq
0x00..55555460c
mov $0x0, %eax
0x00..555554607
callq 5fa
0x00..555554602
mov $0x0, %eax
0x00..555554601
ret
0x00..5555545fa
addl $0x1, 0x200a13(%rip)
%rip
...
Memory
CPU
RIP:
0x00..5555545fa
RAX:
0x0
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000088
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...



0x7f…000000098
...
0x7f…000000090
...
0x7f…000000088
0x00..55555460c
%rsp
0x00..555755018
0x00..555755014
0x1
...
...
...
0x00..555554611
retq
0x00..55555460c
mov $0x0, %eax
0x00..555554607
callq 5fa
0x00..555554602
mov $0x0, %eax
0x00..555554601
ret
%rip
0x00..5555545fa
addl $0x1, 0x200a13(%rip)
...
Memory
CPU
RIP:
0x00..555554601
RAX:
0x0
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000088
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...



0x7f…000000098
...
0x7f…000000090
...
%rsp
0x7f…000000088
0x00..55555460c
0x00..555755018
0x00..555755014
0x1
...
...
...
0x00..555554611
retq
0x00..55555460c
mov $0x0, %eax
%rip
0x00..555554607
callq 5fa
0x00..555554602
mov $0x0, %eax
0x00..555554601
ret
0x00..5555545fa
addl $0x1,
0x200a13(%rip)
...
Memory
CPU
RIP:
0x00..55555460c
RAX:
0x0
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000090
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...



0x7f…000000098
...
0x7f…000000090
...
%rsp
0x7f…000000088
0x00..55555460c
0x00..555755018
0x00..555755014
0x1
...
...
...
0x00..555554611
retq
%rip
0x00..55555460c
mov $0x0, %eax
0x00..555554607
callq 5fa
0x00..555554602
mov $0x0, %eax
0x00..555554601
ret
0x00..5555545fa
addl $0x1, 0x200a13(%rip)
...
Memory
CPU
RIP:
0x00..55555460c
RAX:
0x0
RBX:
RCX:
RDX:
RSI:
RDI:
RSP:
0x7f…000000090
RBP:
ZF:
0
SF:
0
CF:
0
OF:
0
...

Where to store function arguments and return values?

- Hardware doesn't care where args/return vals are stored
  - It's a software convention
- Design consideration: where to put args and return vals?
  - Arguments and return value 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 on stack
  - Store in registers ← Registers are much faster than memory but there are only a few of them

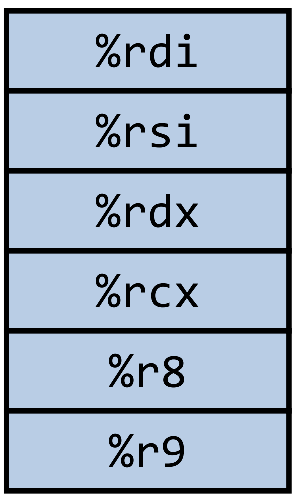
- The chosen design → the calling convention
  - All code on a computer system must obey the same convention
  - Otherwise, libraries won't work

C/UNIX/MacOS's calling convention

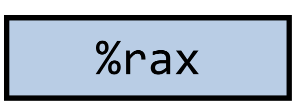
Windows have a different calling convention

Registers

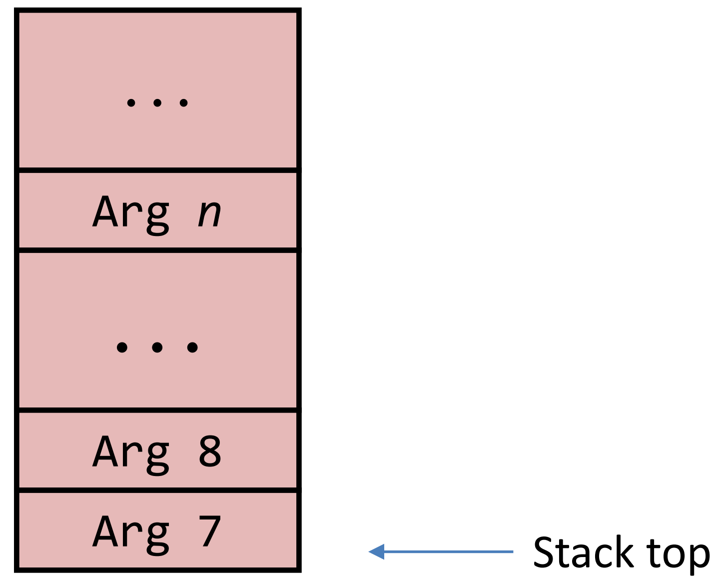
First 6 arguments



Return value



Stack



Only allocate stack space when needed

Calling convention: args, return vals

```
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;
}
```

```
main:
    pushl   $8
    pushl   $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    12(%rsp), %eax
    ret
```

8(%rsp) stores g

12(%rsp) stores h

what does (%rsp) store?

How to allocate/deallocate local vars?

- For primitive data types, use registers whenever possible
- Allocate local array/struct variables on the stack

```
int main() {
    int a[10];
    clear_array(a, 10);
    return 0;
}
```

```
main:
    subq    $48, %rsp
    movl    $10, %esi
    movq    %rsp, %rdi
    call    clear_array
    movl    $0, %eax
    addq    $48, %rsp
    ret
```

array allocation

array de-allocation

Calling convention: Caller vs. callee-save registers

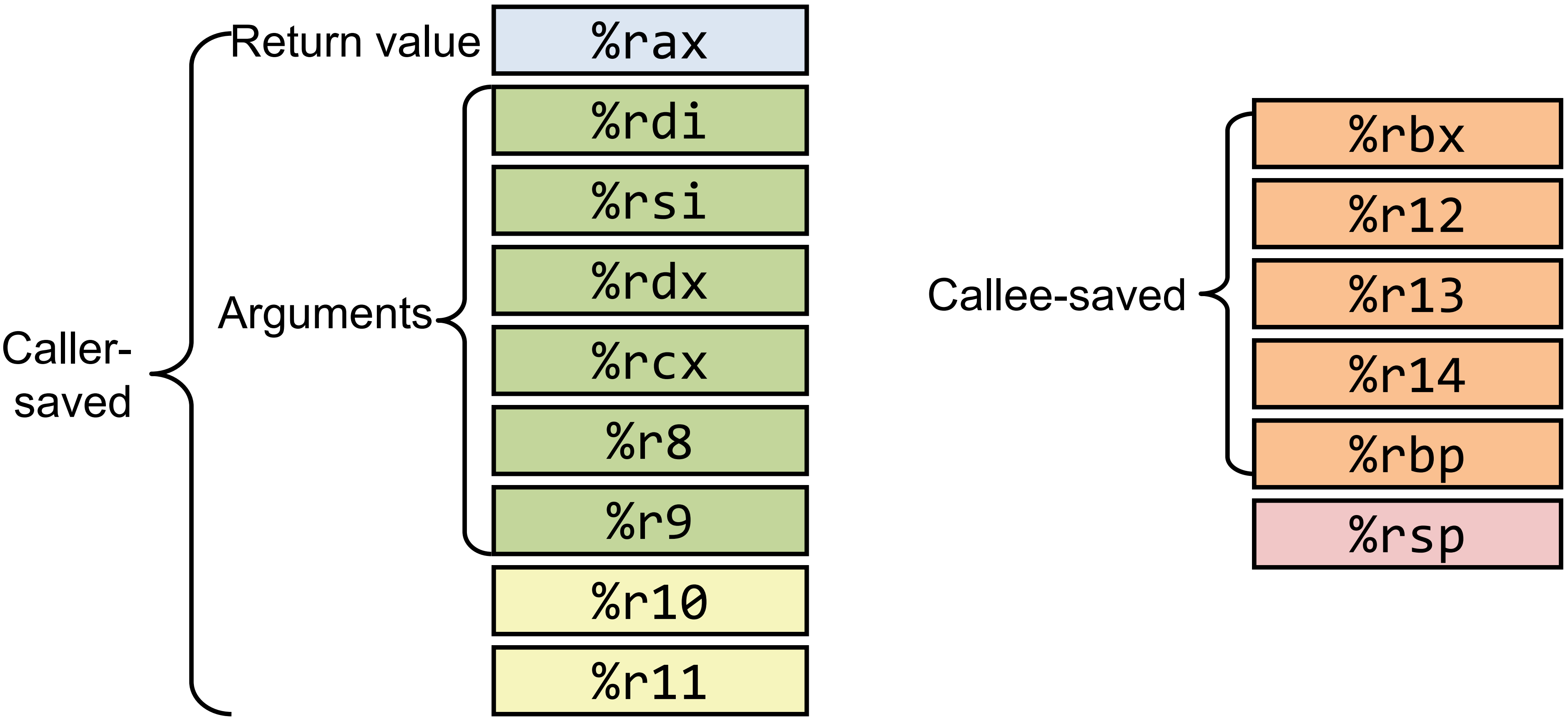
- What can the caller assume about the content of a register across function calls?

```
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

- Caller saved
    - If caller is going to need X's value after the call, it saves X on stack before the call and restores X after the call
  - Callee saved
    - If callee is going to use Y, it saves Y on stack before using and restores Y before returning to caller

Calling convention: Register saving



Callee can directly use these registers

Caller can assume callee-save registers are unchanged across function calls

Example

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

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

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

```
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, %r9

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 overwriting it

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

c is copied to %ebx, which is callee save

%rdx (contains c) is caller save, i.e. may be changed by add2

restore %rbx before ret

Registers

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

Return value: %rax

Summary

- Function call in x86
  - Stack (stores return-address, local variables)
    - Push, pop
  - Call/ret
    - Call saves return-address on stack, ret pops return-address from stack
  - UNIX calling convention
    - First 6 function arguments are stored in %rdi, %rsi, %rdx, %rcd, %r8, %r9
    - Return val is stored in %rax
    - Caller vs. callee save registers