

Machine Program: Procedure

Jinyang Li

Some slides are based on Tiger Wang's

What we've learnt (how hardware runs a program)

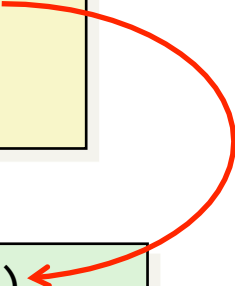
- Where are data and instructions stored?
 - memory: heap, stack, data, text
 - some local variables may reside in registers only
- Modes of execution:
 - Sequential:
 - PC (%rip) is changed to point to the next instruction
 - Control flow: jmp, conditional jmp
 - PC (%rip) is changed to point to the jump target address
 - **Today → procedure call**

Requirements of procedure calls?

```
P(...) {  
  y = Q(x);  
  y++;  
}
```

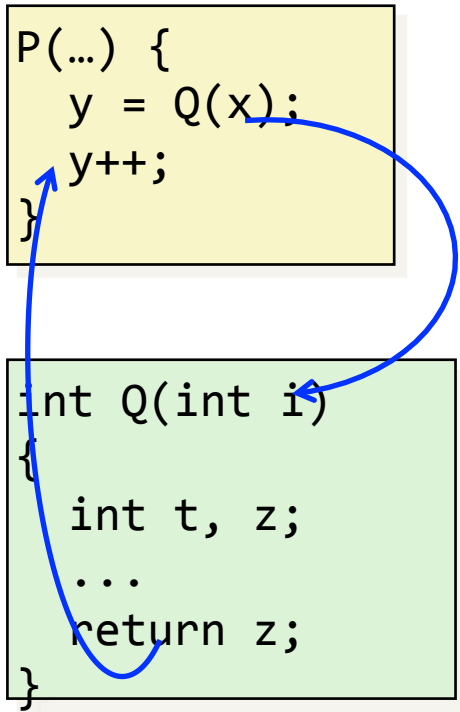
1. Passing control

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



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

L2

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

Jump to h()
Remember where to come back

L3
L2
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

Jump to L3
Forget L3



How to transfer control for procedure calls?

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

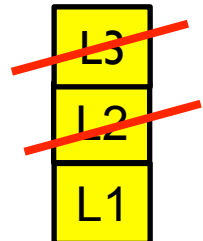
```
void f(){  
    ..  
    g(..)  
L2: ..  
}
```

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

Jump to f()
Remember where to come back

Jump to L2
Forget L2

Jump to L3
Forget L3



How to transfer control for procedure calls?

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

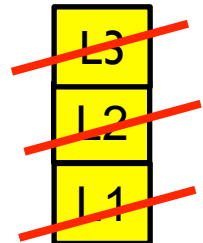
```
void f(){  
    ..  
    g(..)  
L2: ..  
}
```

```
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: ..  
}
```

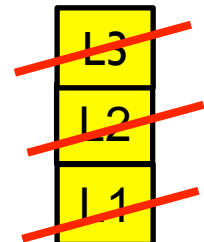
```
void f(){  
    ..  
    g(..)  
L2: ..  
}
```

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

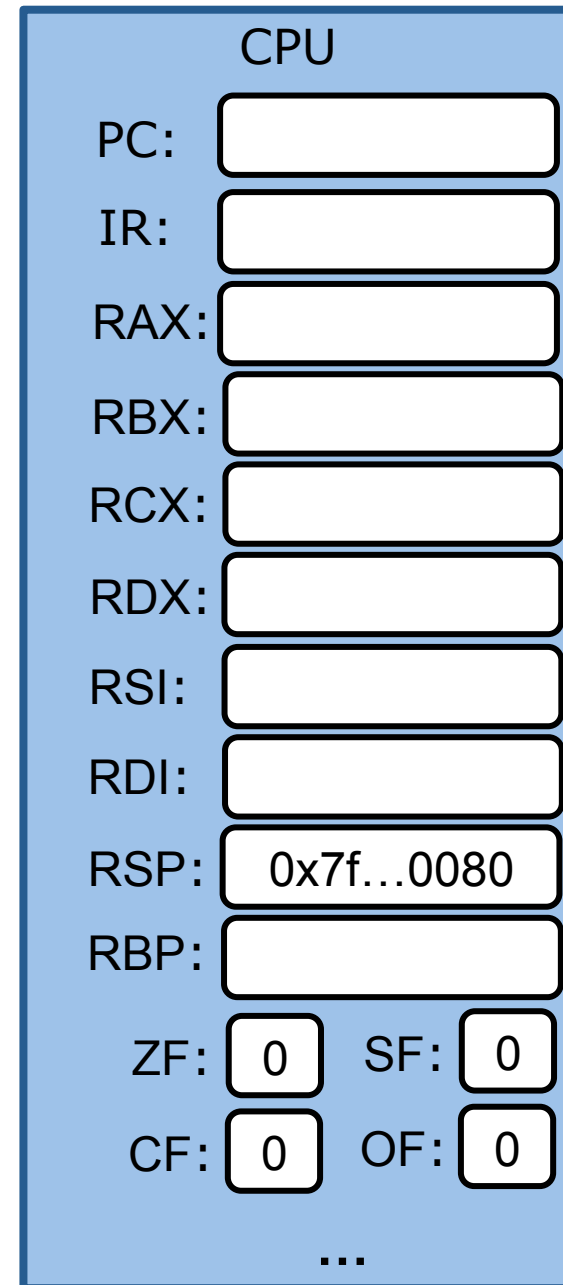
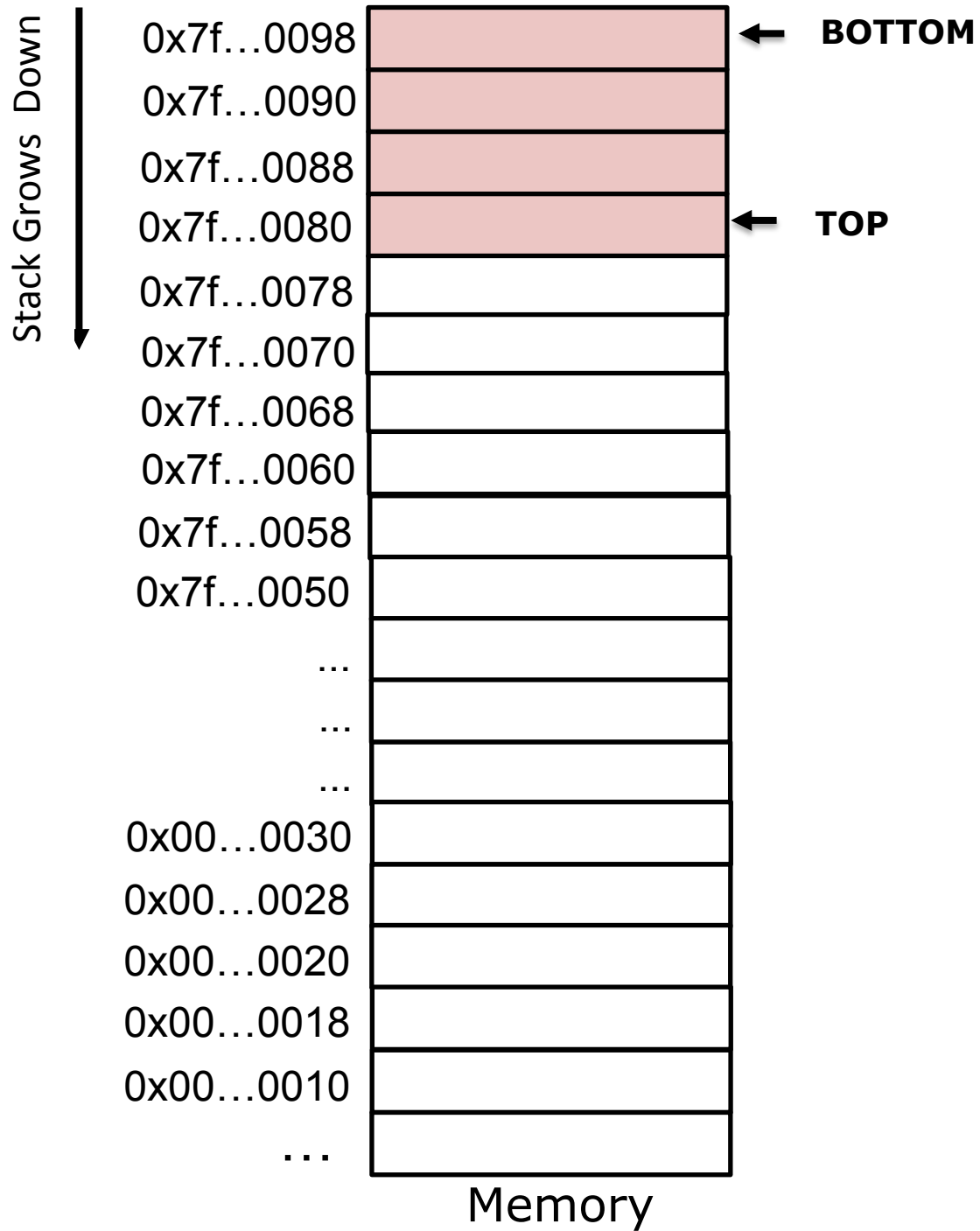
Jump to L1
Forget L1

Jump to L2
Forget L2

Jump to L3
Forget L3



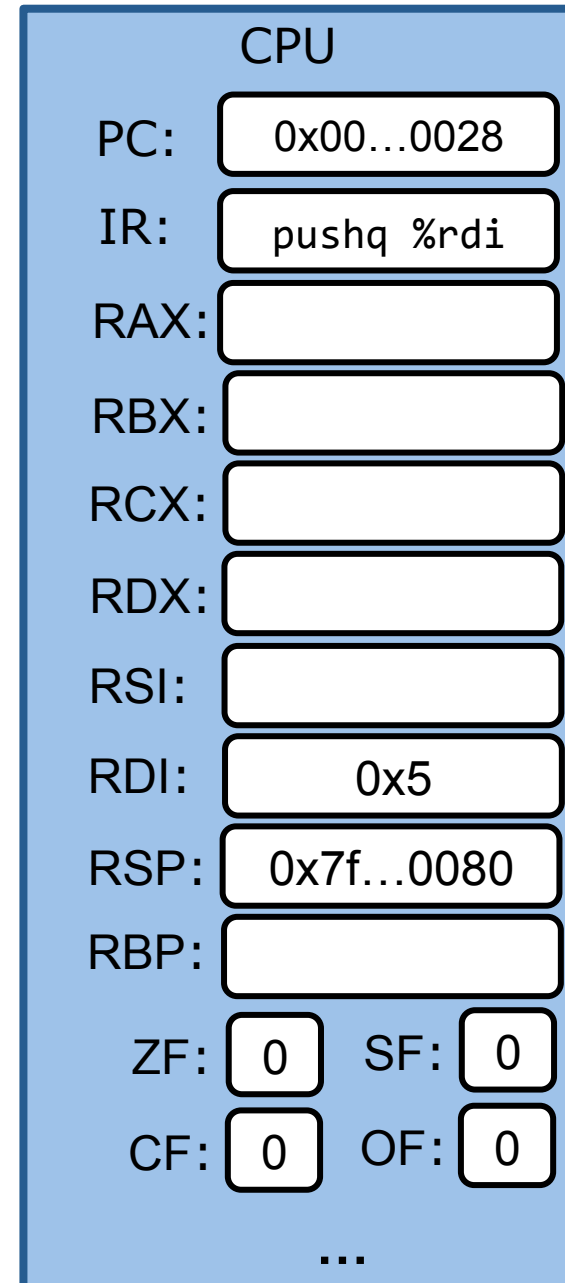
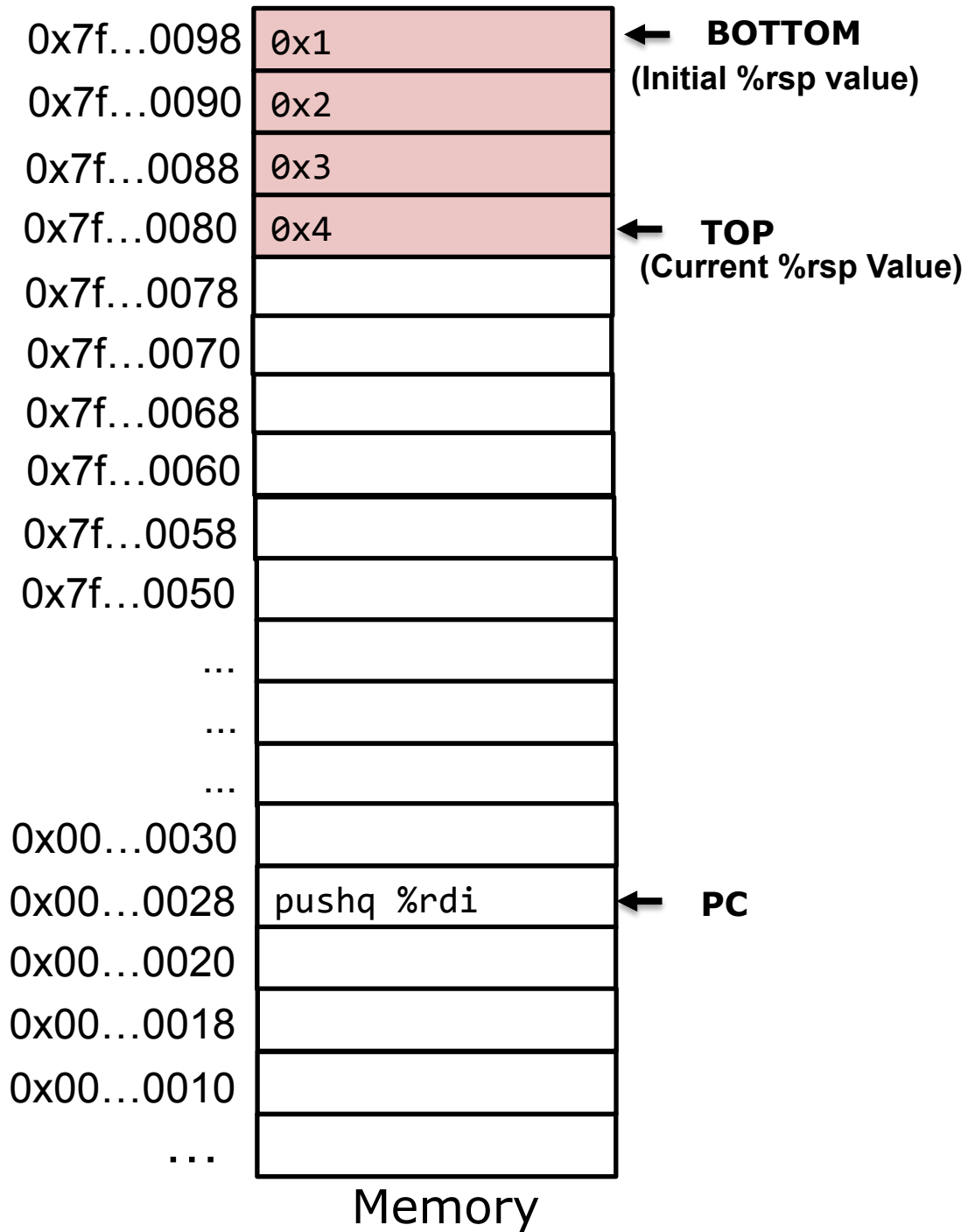
Stack

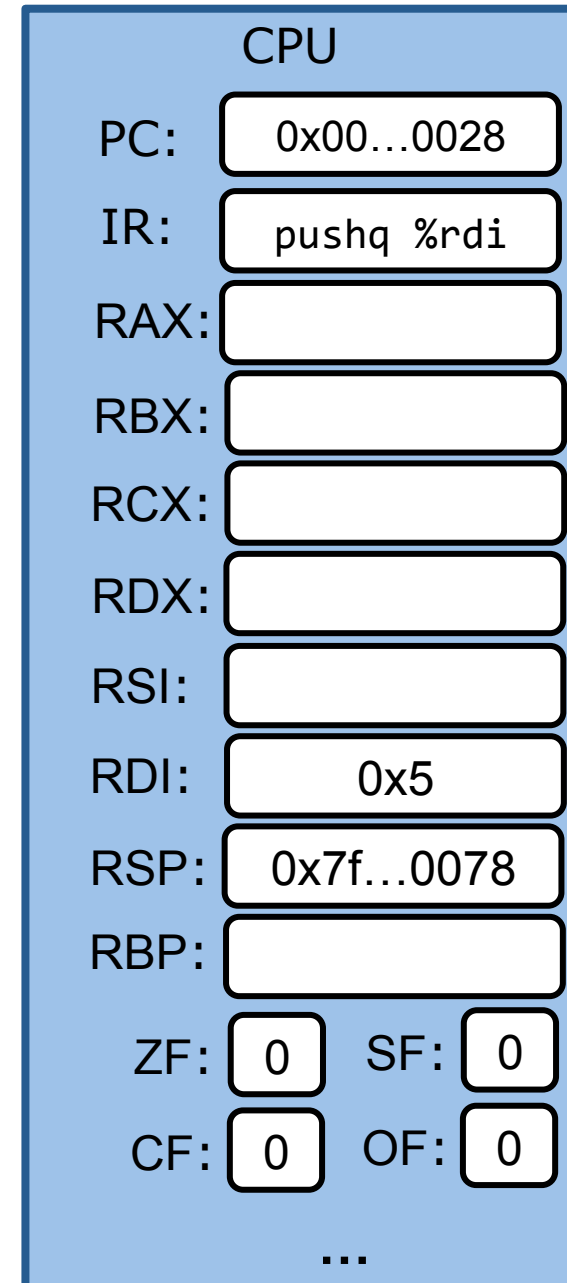
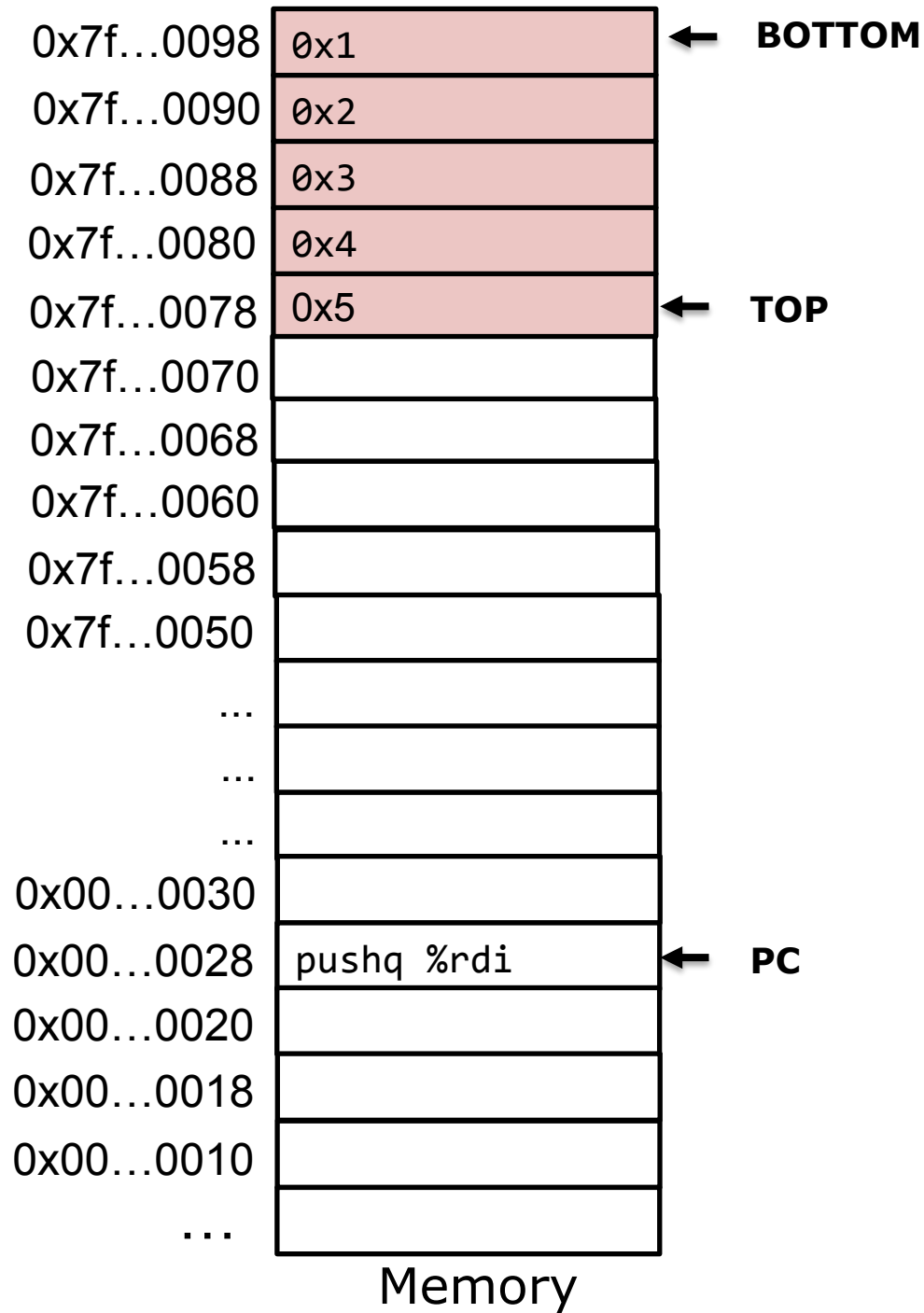


Stack – push Instruction

pushq src

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

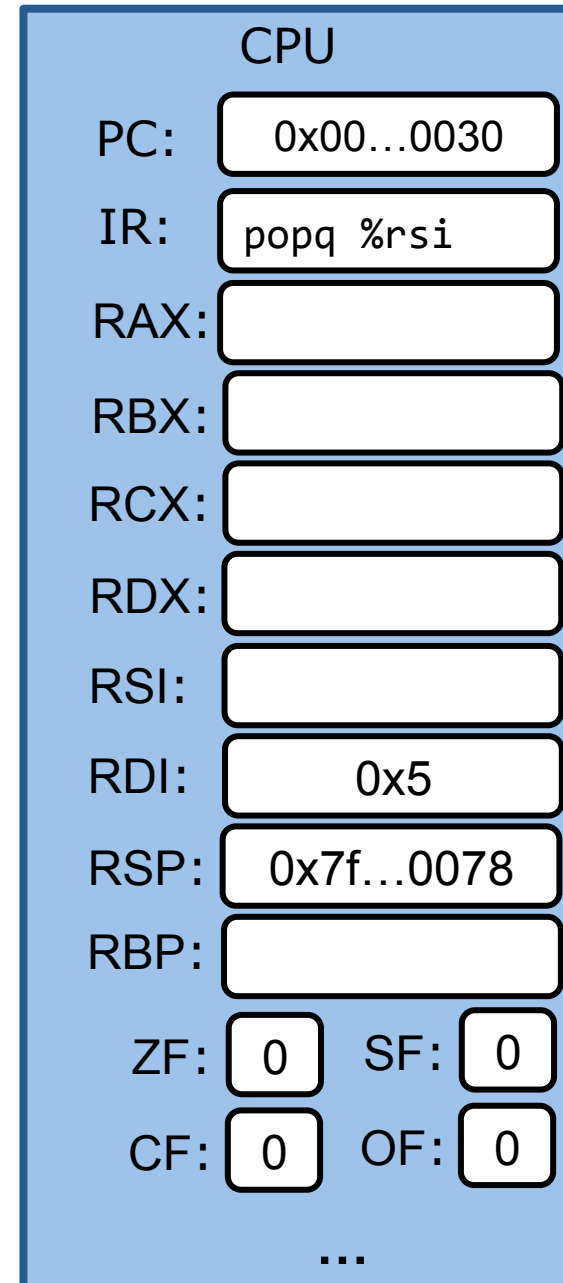
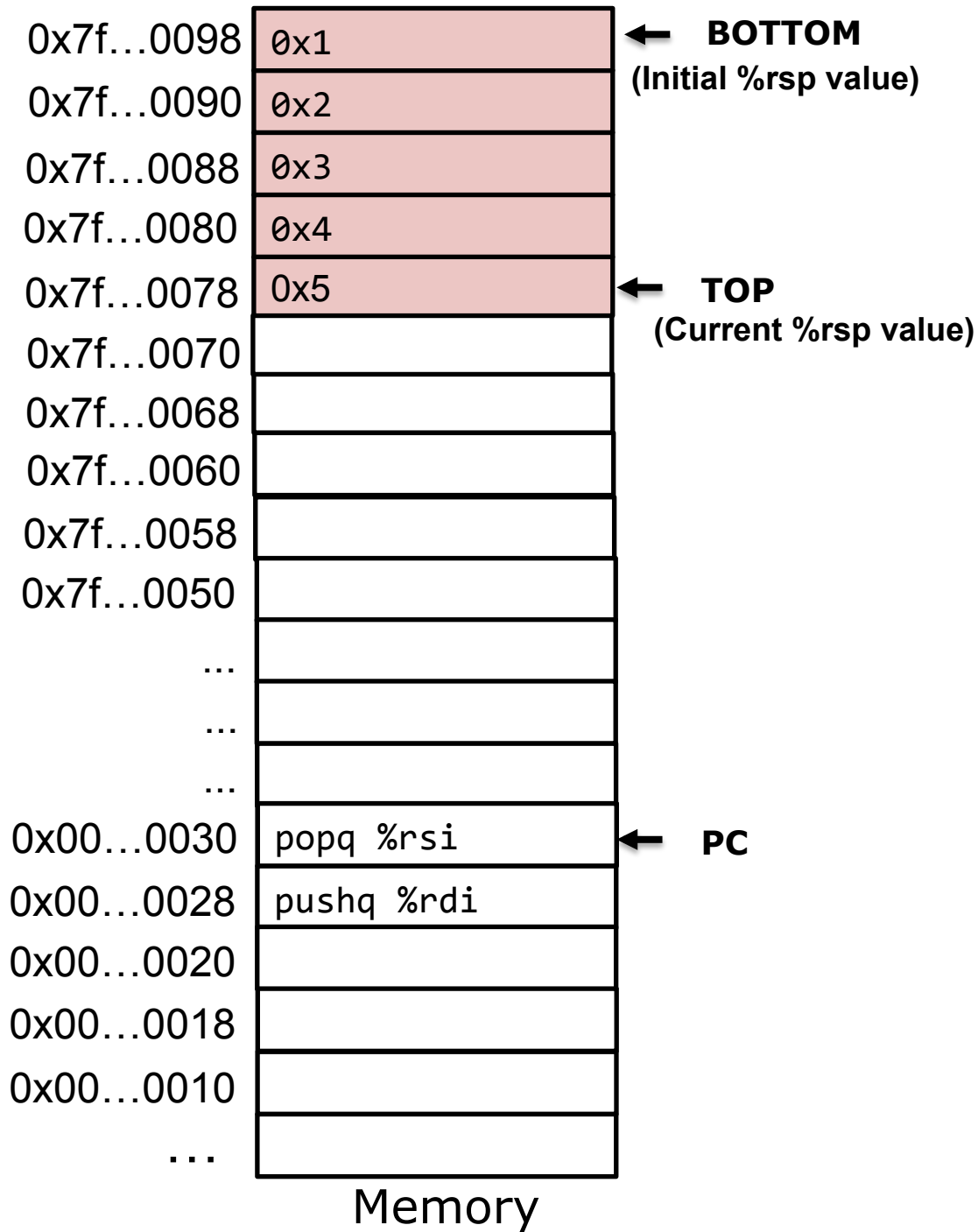


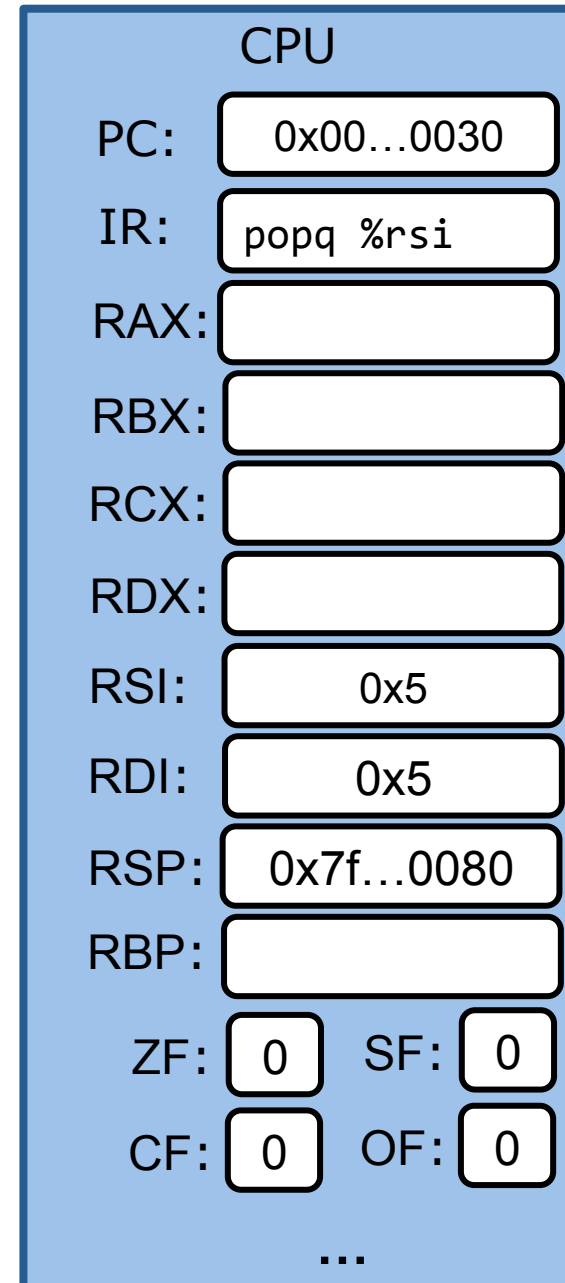
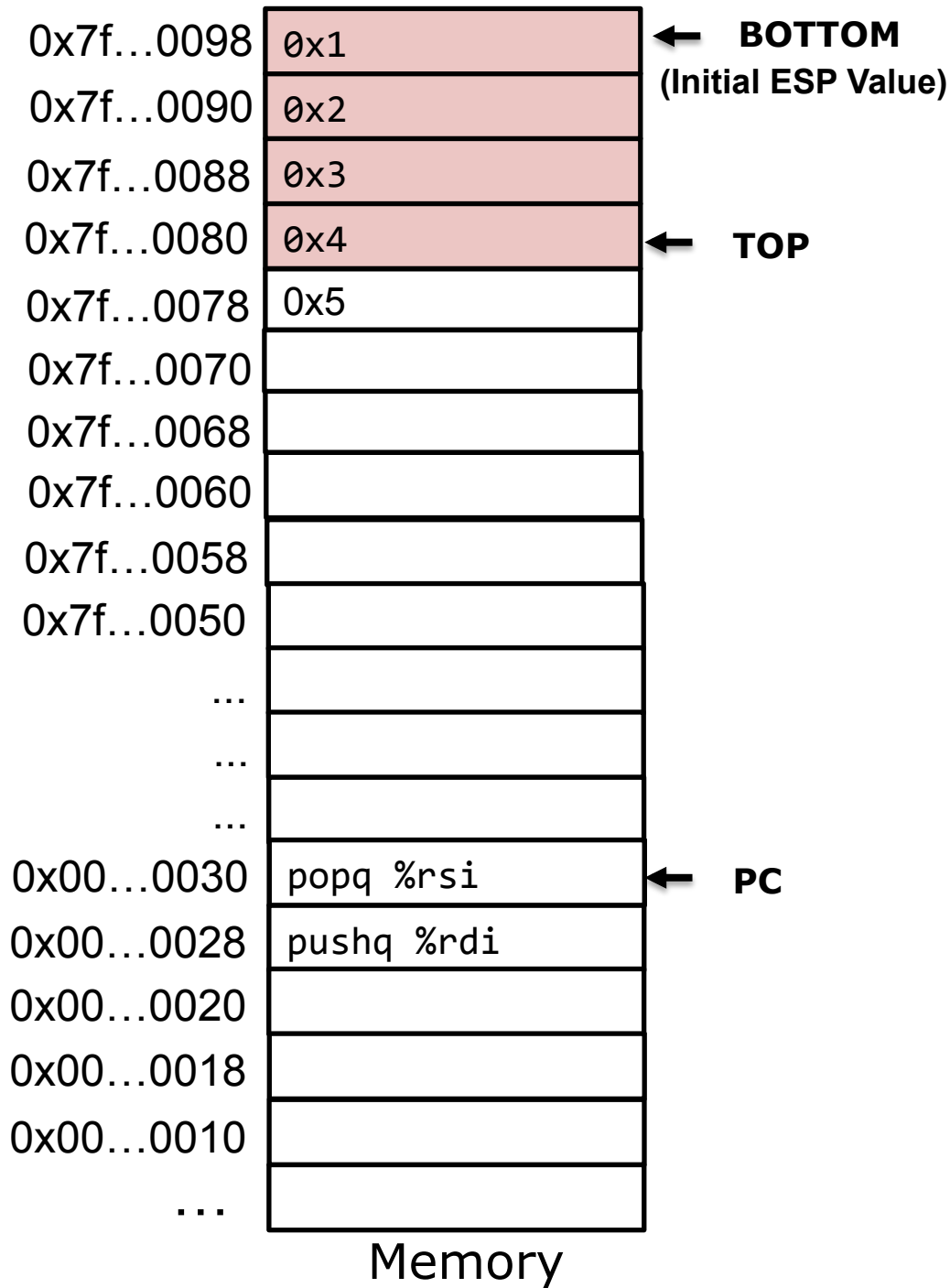


Stack – pop Instruction

popq dest

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





Call instruction: control transfer from caller to callee

call label

- Push return address on stack
 - “return address” points to instruction immediately after **call**
- Jump to the address of the label

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

```
int main() {  
    int c = add(0, 2);  
    printf("%d\n", c);  
    return 0;  
}
```

Call instruction: control transfer from caller to callee

call label

- Push return address on stack
 - “return address” points to instruction immediately after **call**
- Jump to the address of the label

```
gcc -Og -S main.c
```

add:

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

main:

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



return address points to this instruction

Call instruction: control transfer from caller to callee

call label

```
gcc main.c  
objdump -d a.out
```

0000000000400546 <add>:

400546: 8d 04 37

lea (%rdi,%rsi,1),%eax

400549: c3

retq

000000000040054a <main>:

40054a: 48 83 ec 08

sub \$0x8,%rsp

40054e: be 02 00 00 00

mov \$0x2,%esi

400553: bf 00 00 00 00

mov \$0x0,%edi

400558: e8 e9 ff ff ff

callq 400546 <add>

40055d: 89 c2

mov %eax,%edx

Ret instruction: control transfer from callee back to caller

ret

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

0000000000400546 <add>:

400546: 8d 04 37

lea (%rdi,%rsi,1),%eax

400549: c3

ret

000000000040054a <main>:

40054a: 48 83 ec 08

sub \$0x8,%rsp

40054e: be 02 00 00 00

mov \$0x2,%esi

400553: bf 00 00 00 00

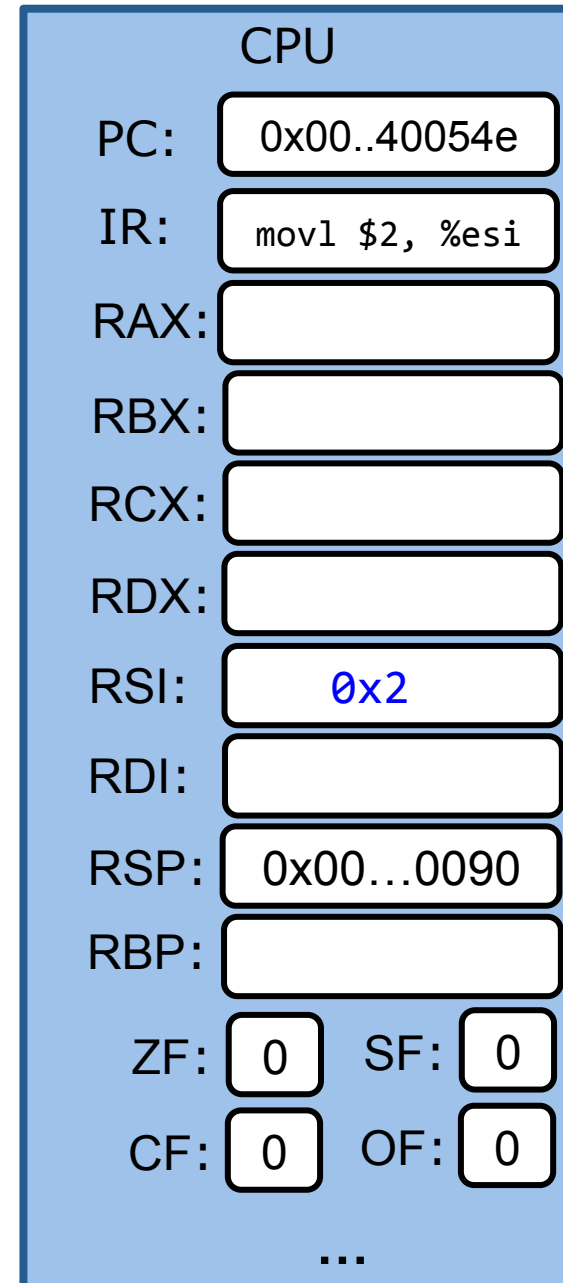
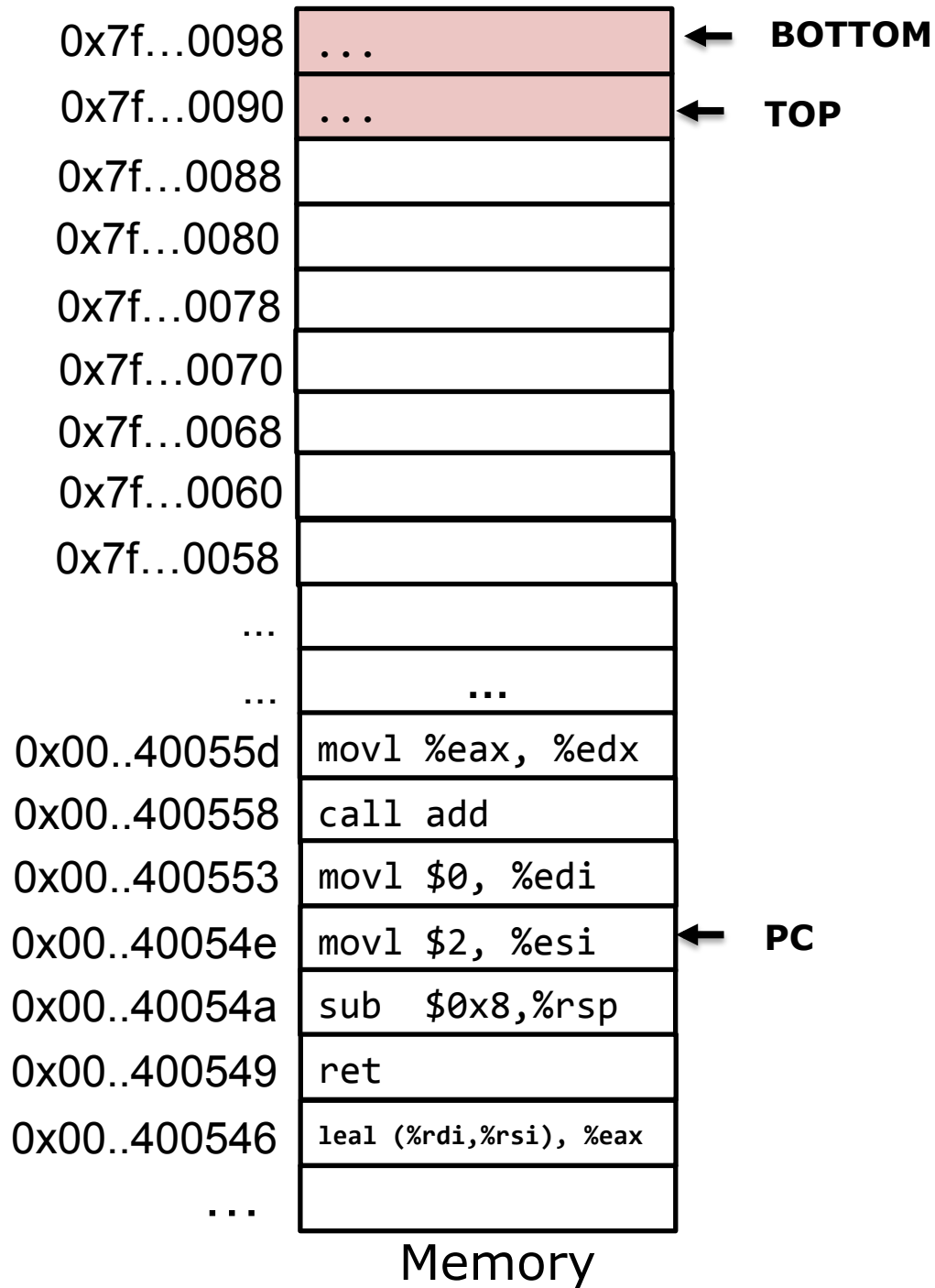
mov \$0x0,%edi

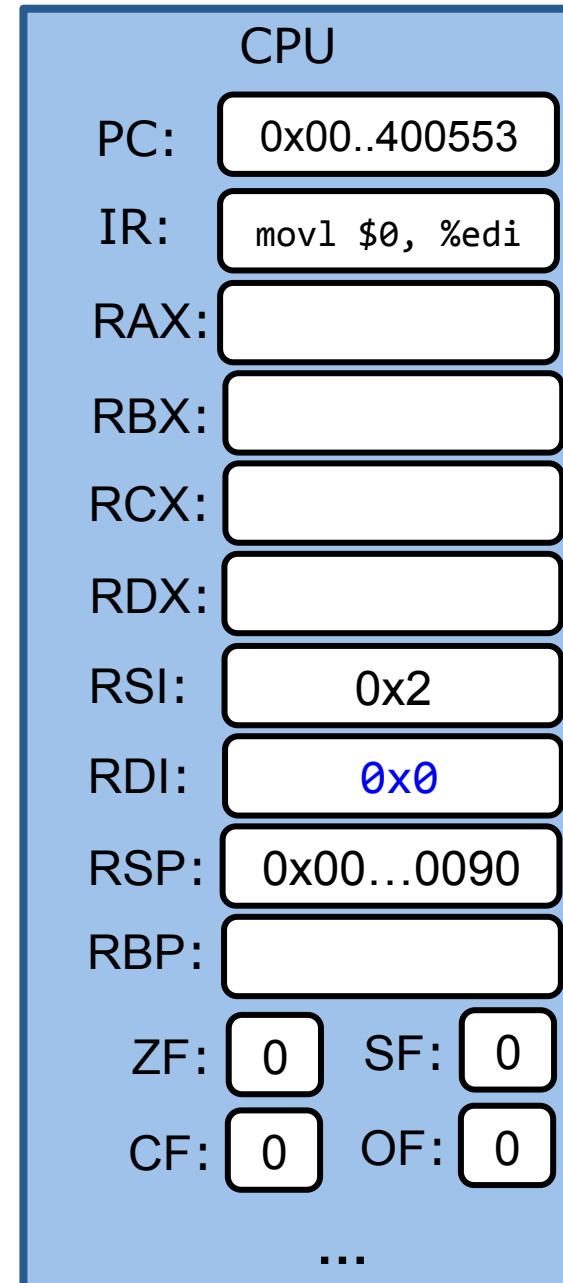
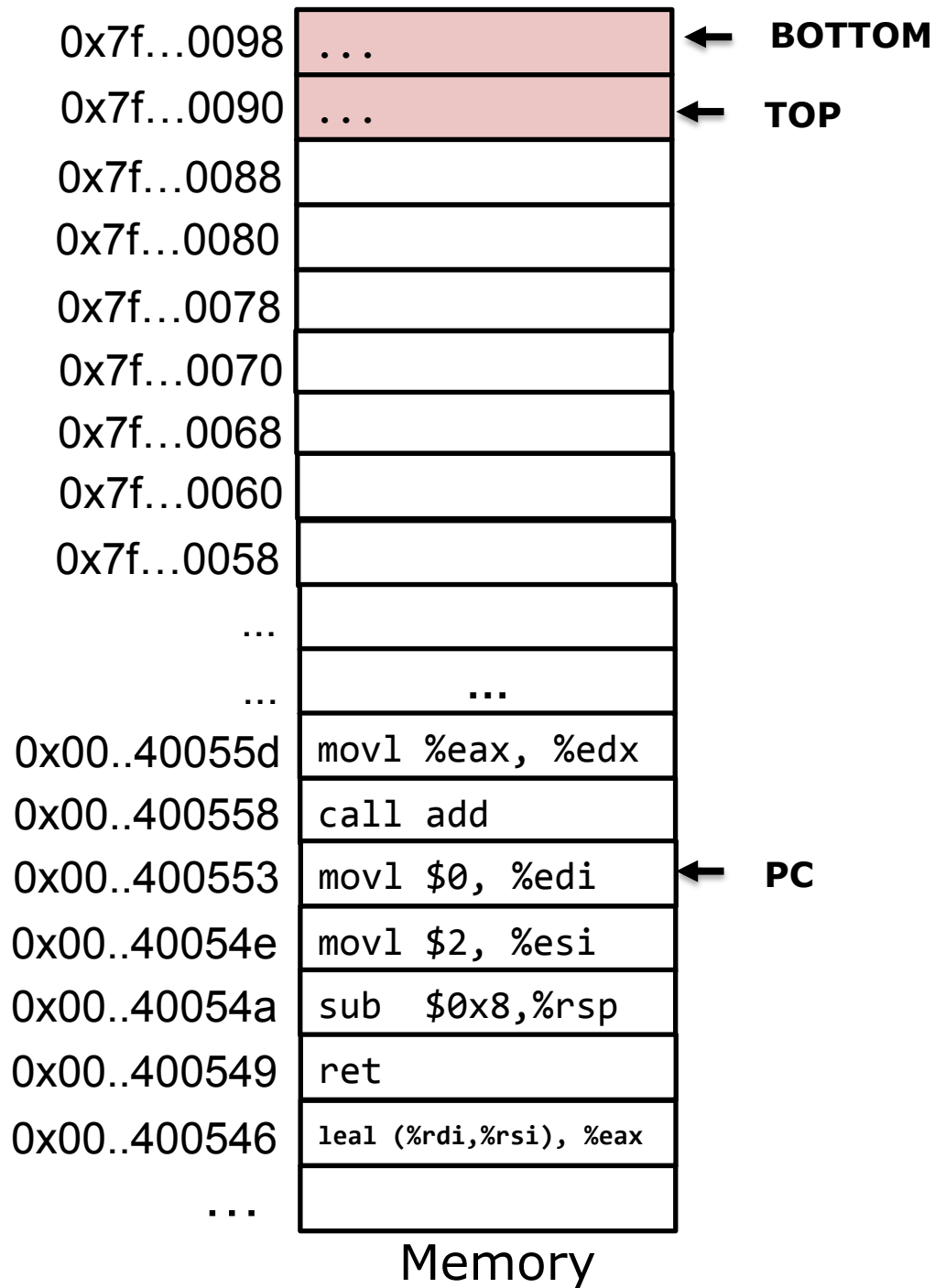
400558: e8 e9 ff ff ff

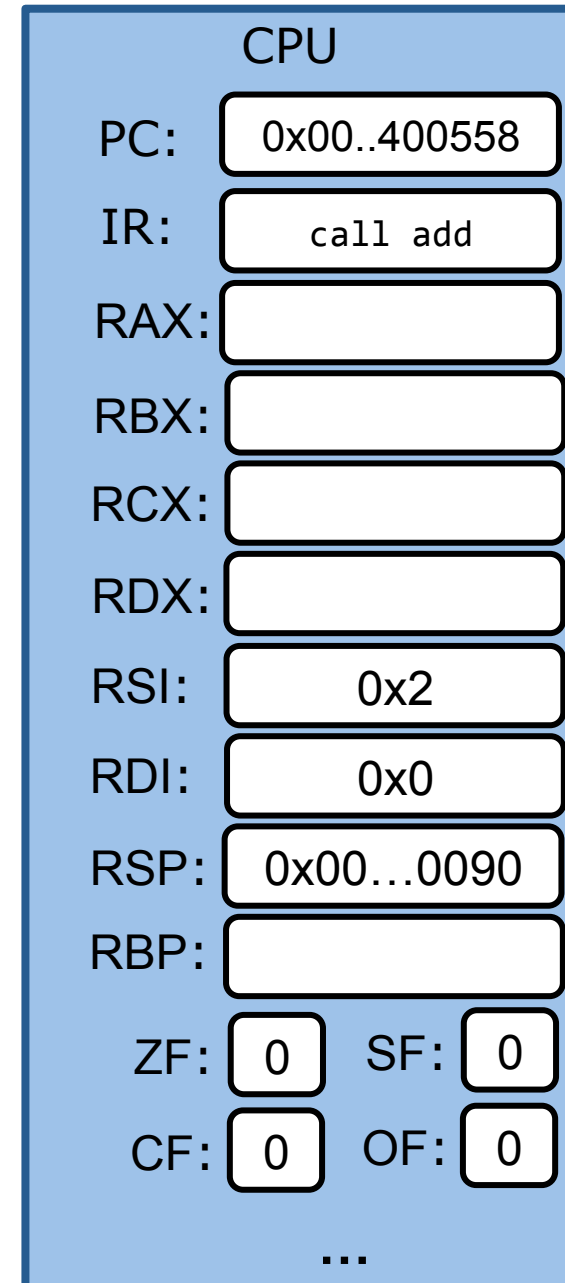
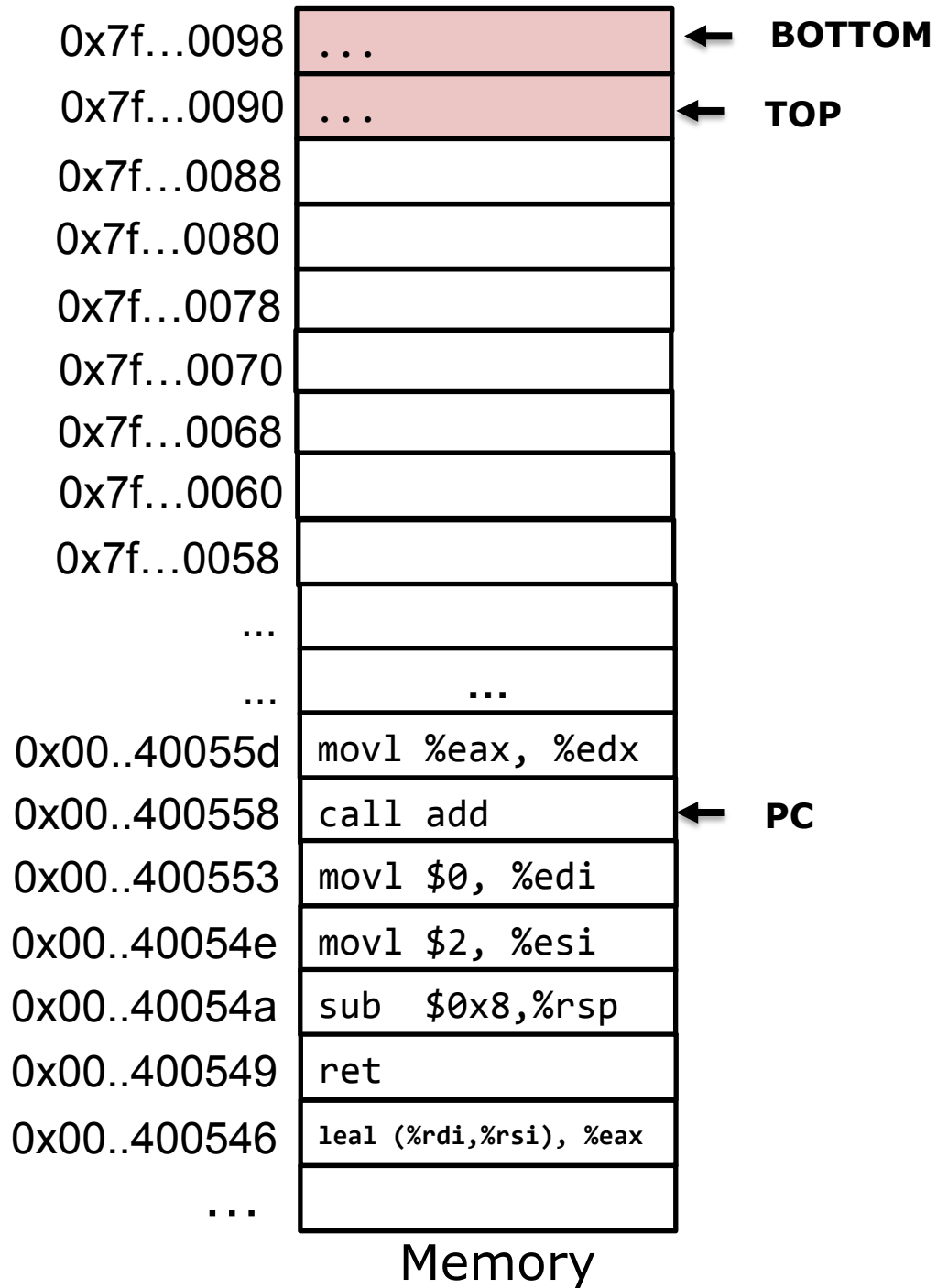
callq 400546 <add>

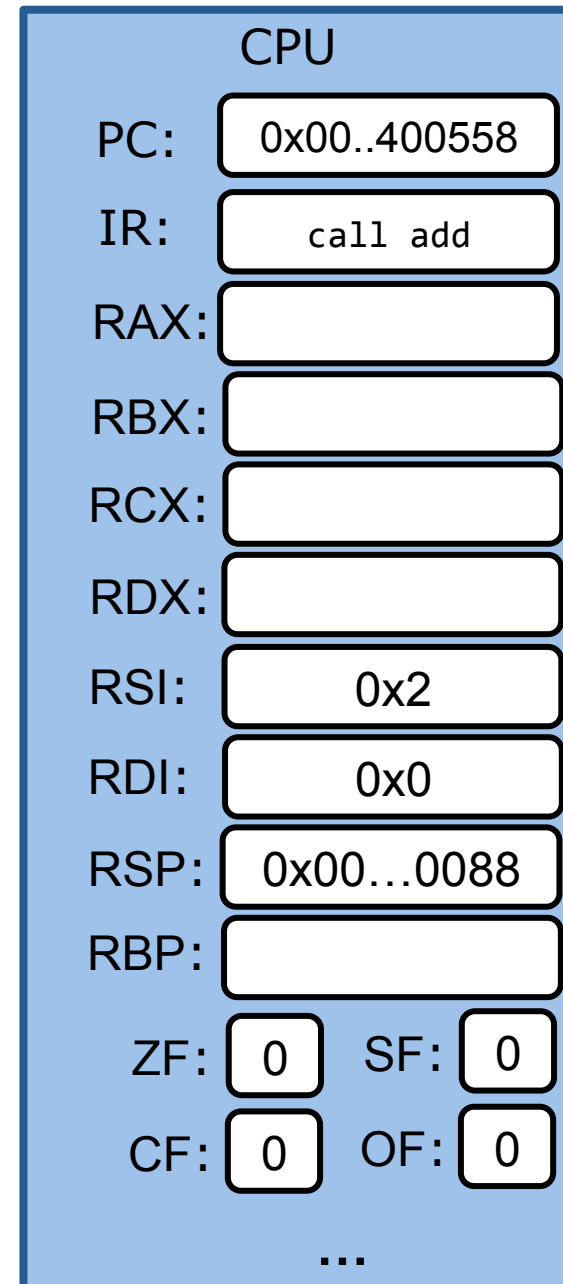
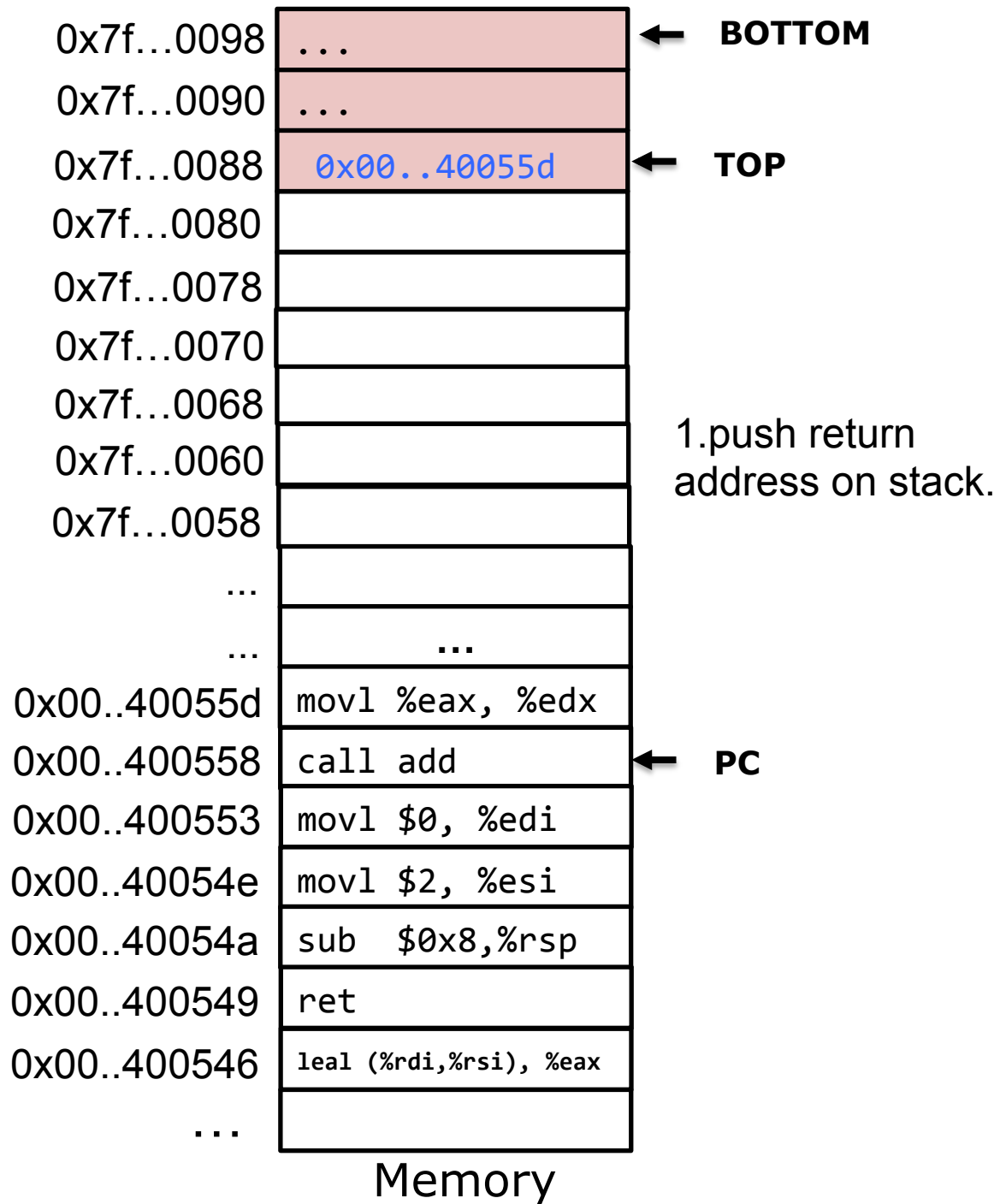
40055d: 89 c2

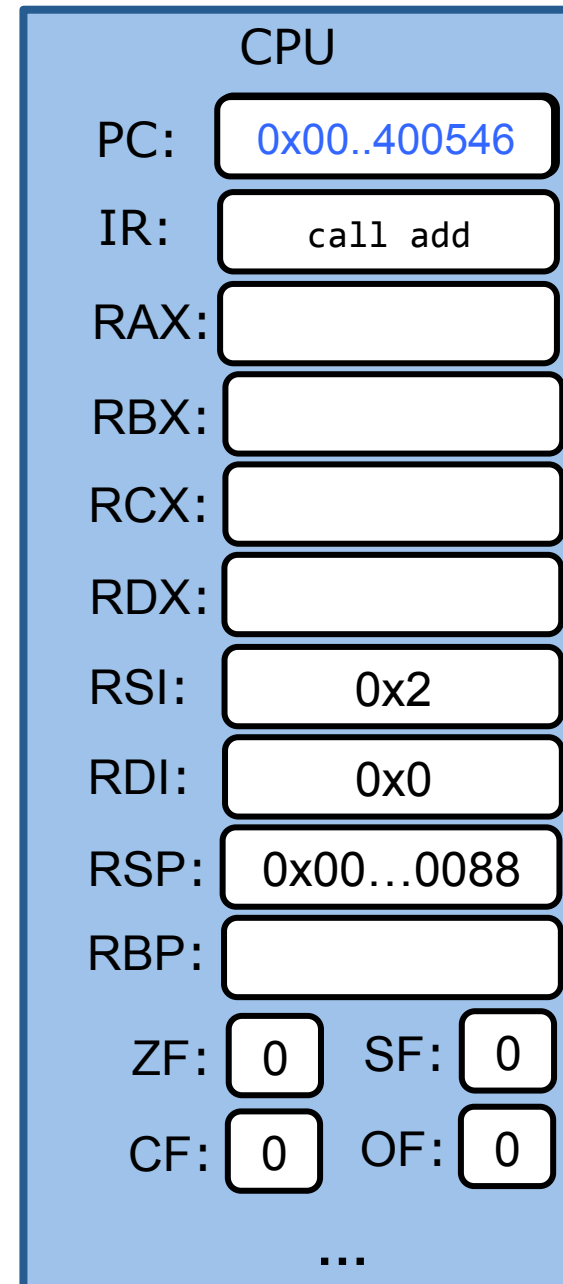
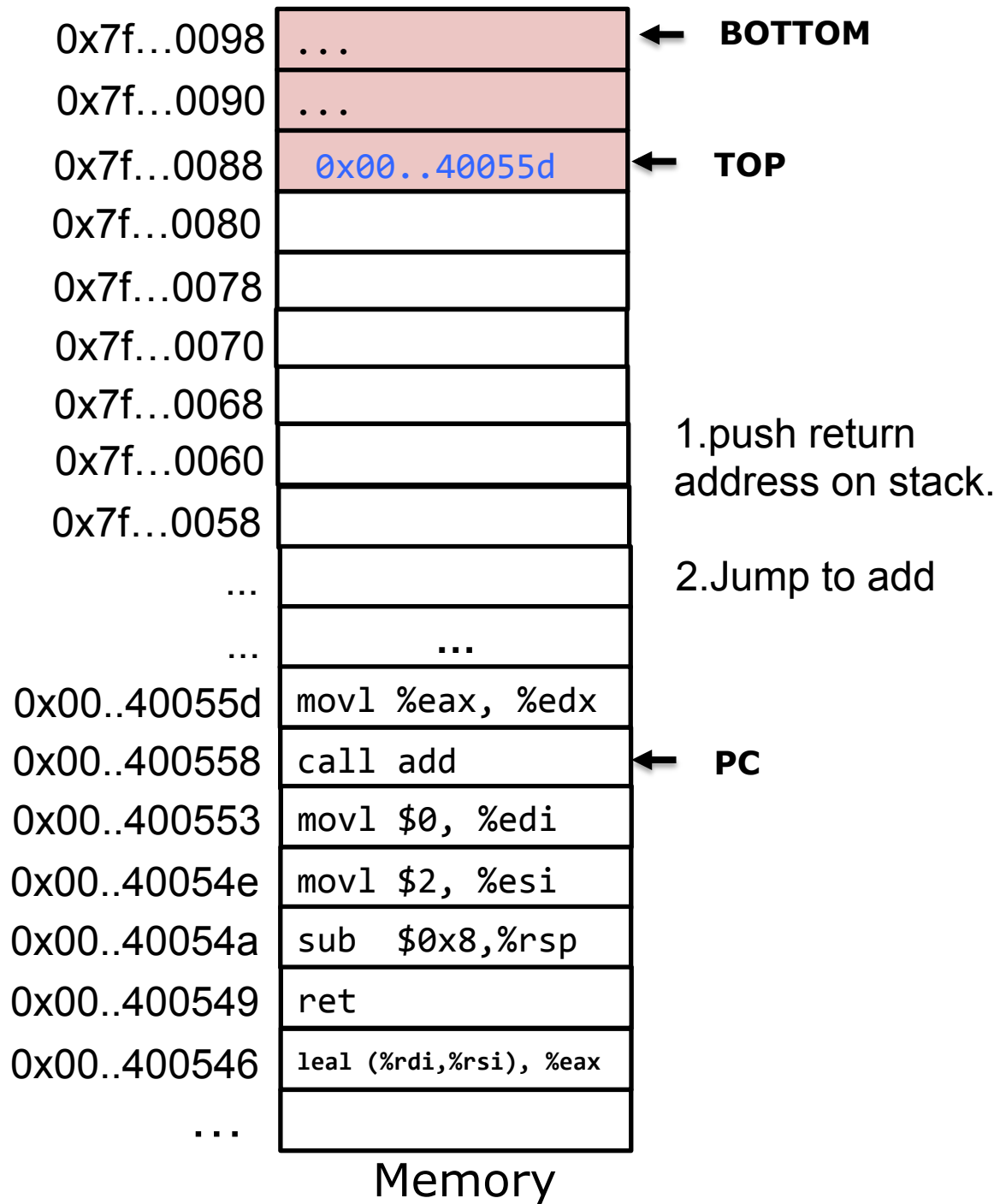
mov %eax,%edx

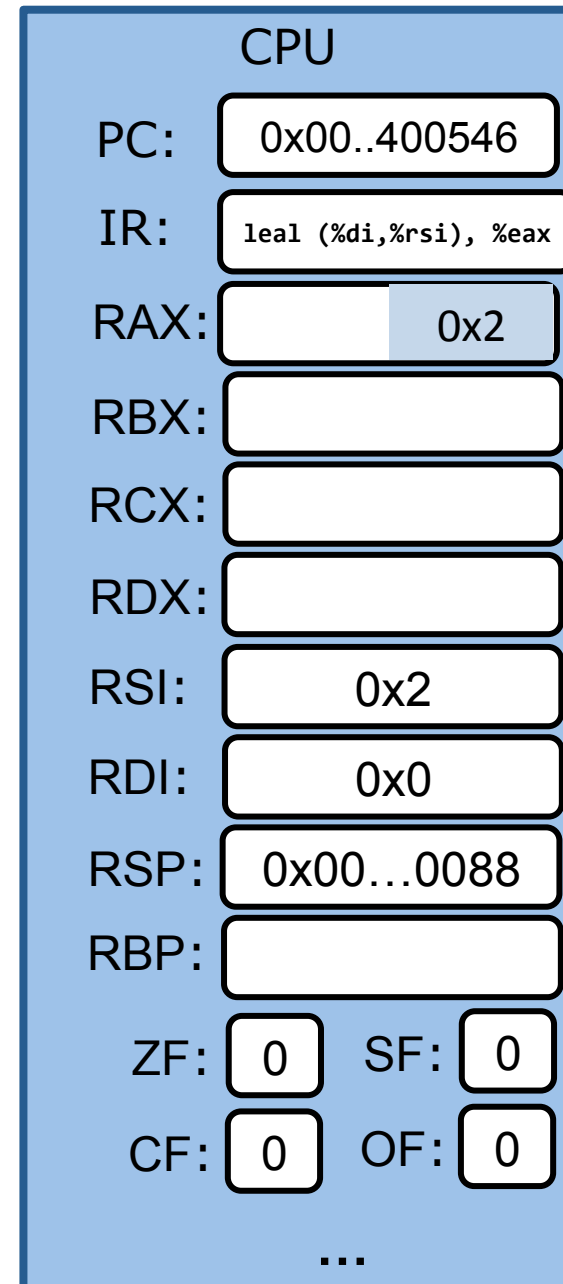
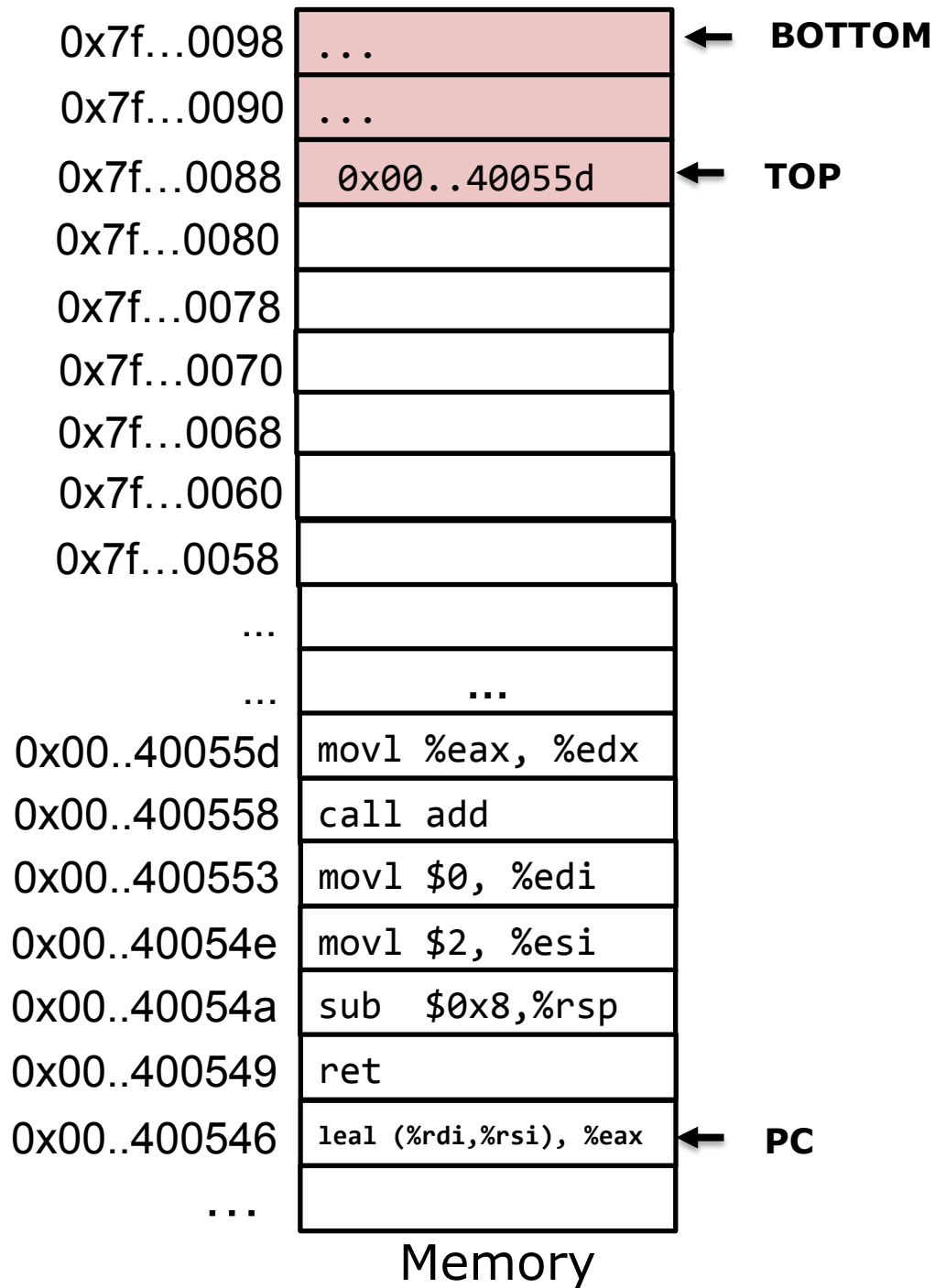


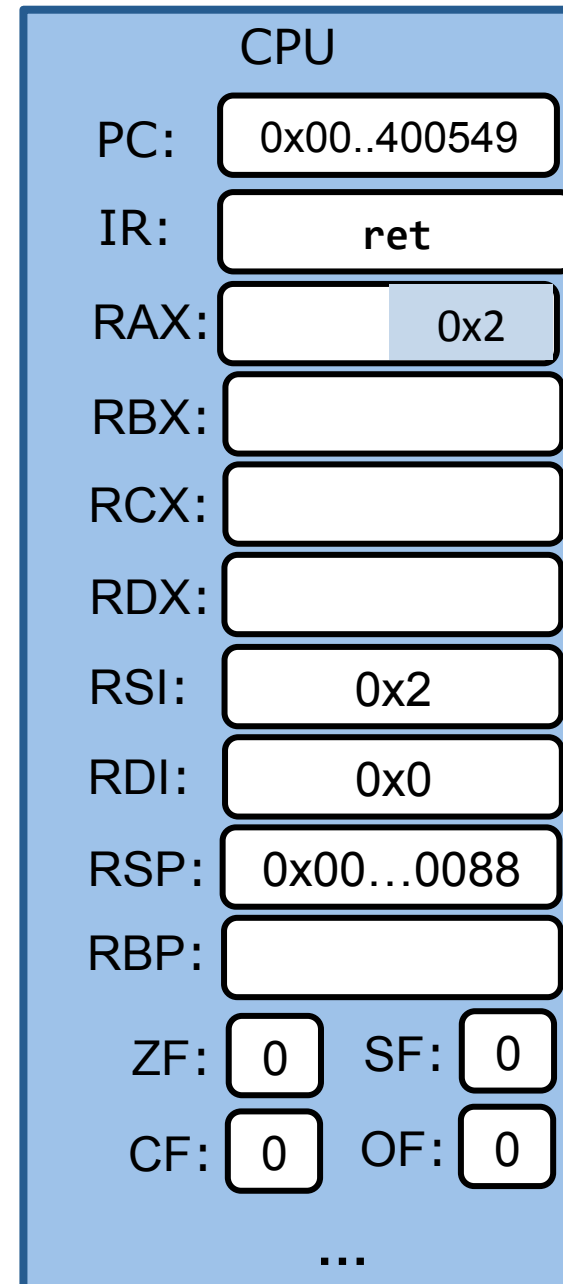
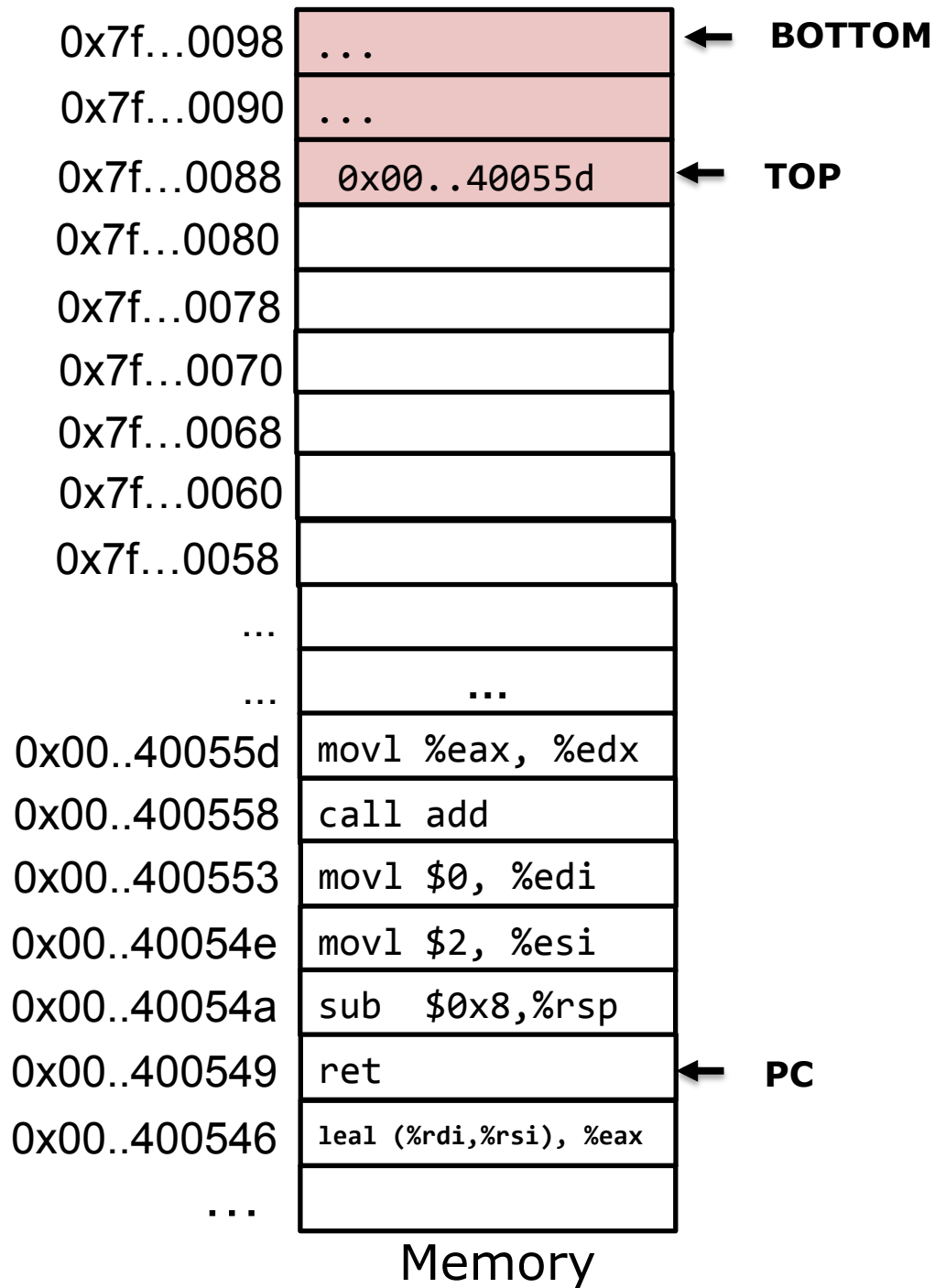


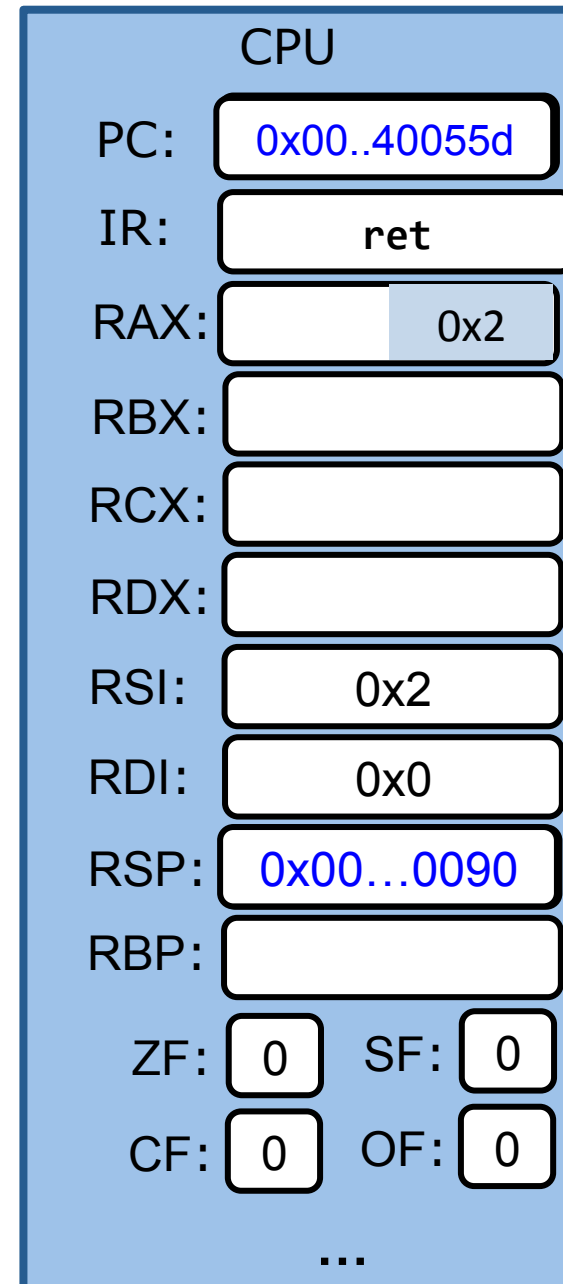
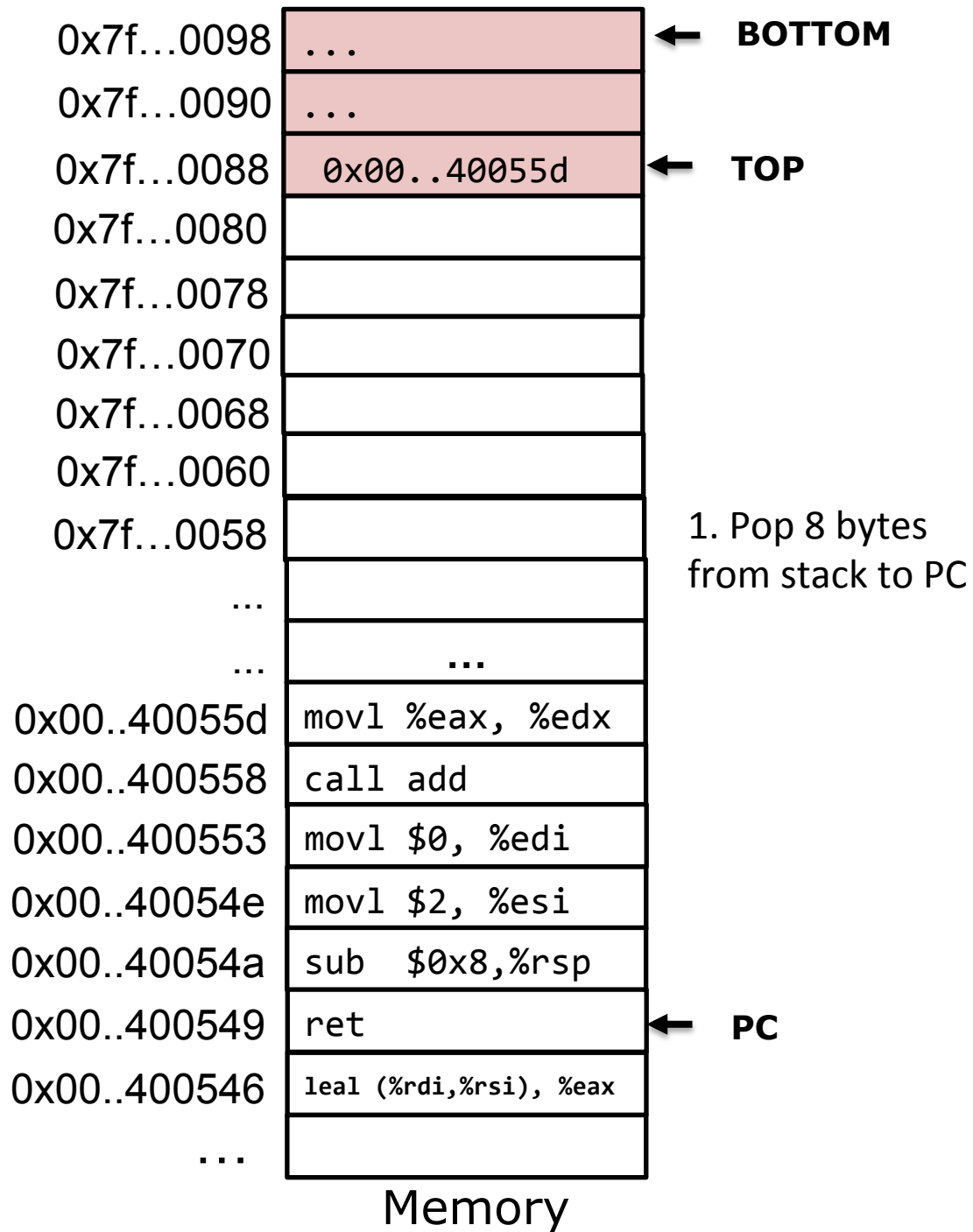













Where to store function arguments and return values?

- Hardware does not dictate where arguments and return value are stored
 - It's up to the software (compilers).
- Where to put arguments and return value?
 - 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
 - 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

%rdi
%rsi
%rdx
%rcx
%r8
%r9

Return value

%rax

Stack

...
Arg n
...
Arg 8
Arg 7

Only allocate stack space when needed

Calling convention: args, return vals

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\n", c);  
    return 0;  
}
```

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\n", c);  
    return 0;  
}
```

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 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

array
allocation

movl \$10, %esi

movq %rsp, %rdi

call clear_array

movl \$0, %eax

addq \$48, %rsp

array
de-allocation

ret

Calling convention: Caller vs. callee-save registers

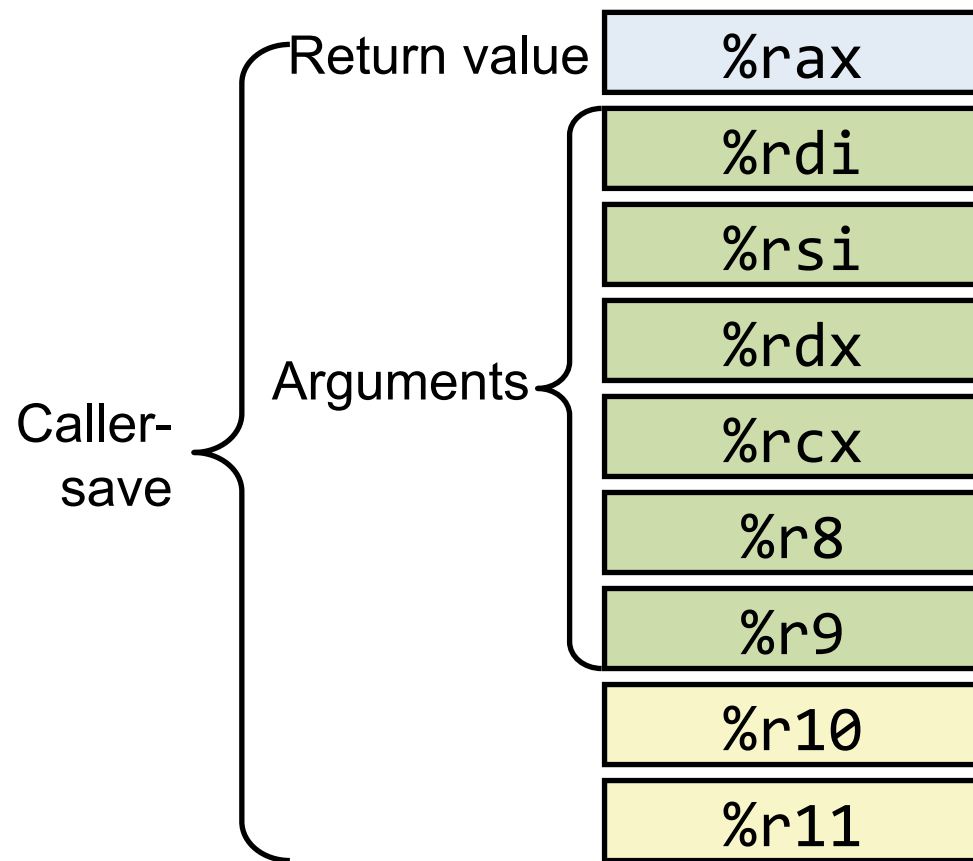
- What can the caller assume about the content of a register across function calls?

```
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;  
}
```

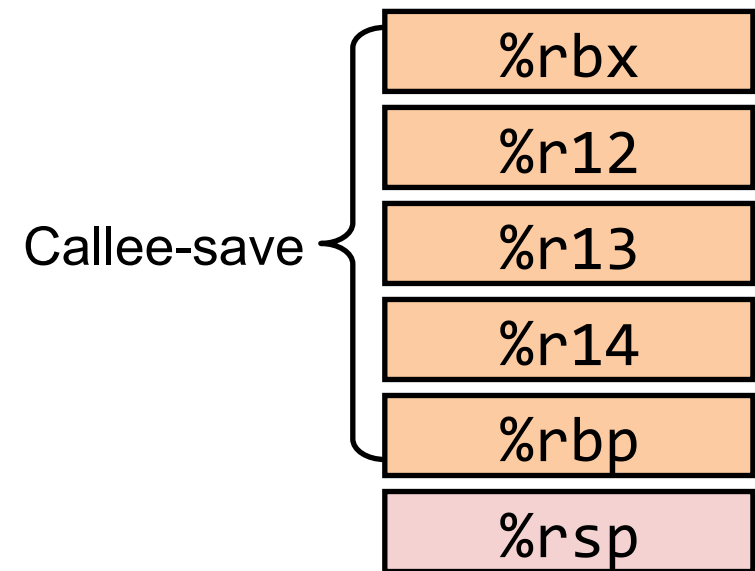
Calling convention: register saving

- Caller-save
 - 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-save
 - 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 these registers are unchanged.

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

```
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;
}
```

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

```
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*

restore %rbx before ret

Registers

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

Return value: %rax