CS107, Lecture 20
Assembly: Function Calls and the Runtime Stack

Reading: B&O 3.7

CS107 Topic 5

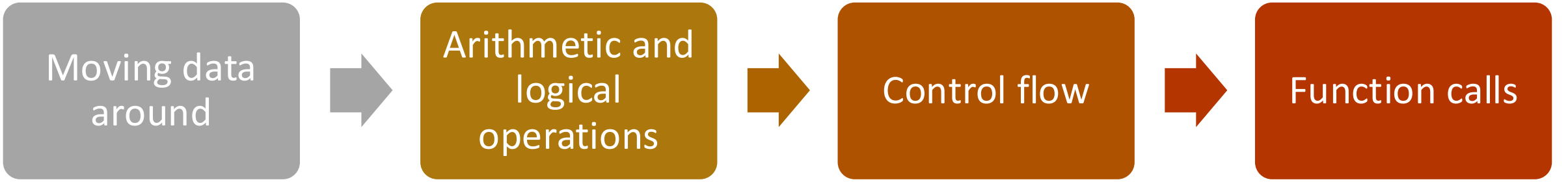
How does a computer interpret and execute C programs?

Why is answering this question important?

- Learning how our code is really translated and executed helps us write better code
- We can learn how to reverse engineer and exploit programs at the assembly level

assign5: find and exploit vulnerabilities in an ATM program, reverse engineer a program without seeing its code, and de-anonymize users given a data leak.

Learning Assembly



Reference Sheet: `cs107.stanford.edu/resources/x86-64-reference.pdf`
See more guides on Resources page of course website!

Learning Goals

- Learn how assembly calls functions and manages stack frames.
- Learn the rules of register use when calling functions.

Lecture Plan

- The Stack
- Calling Functions
  - Running another function's instructions
  - Parameters and return values
  - Trace: Calling a Function
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Lecture Plan

- **The Stack**
- Calling Functions
  - Running another function's instructions
  - Parameters and return values
  - Trace: Calling a Function
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Register Responsibilities

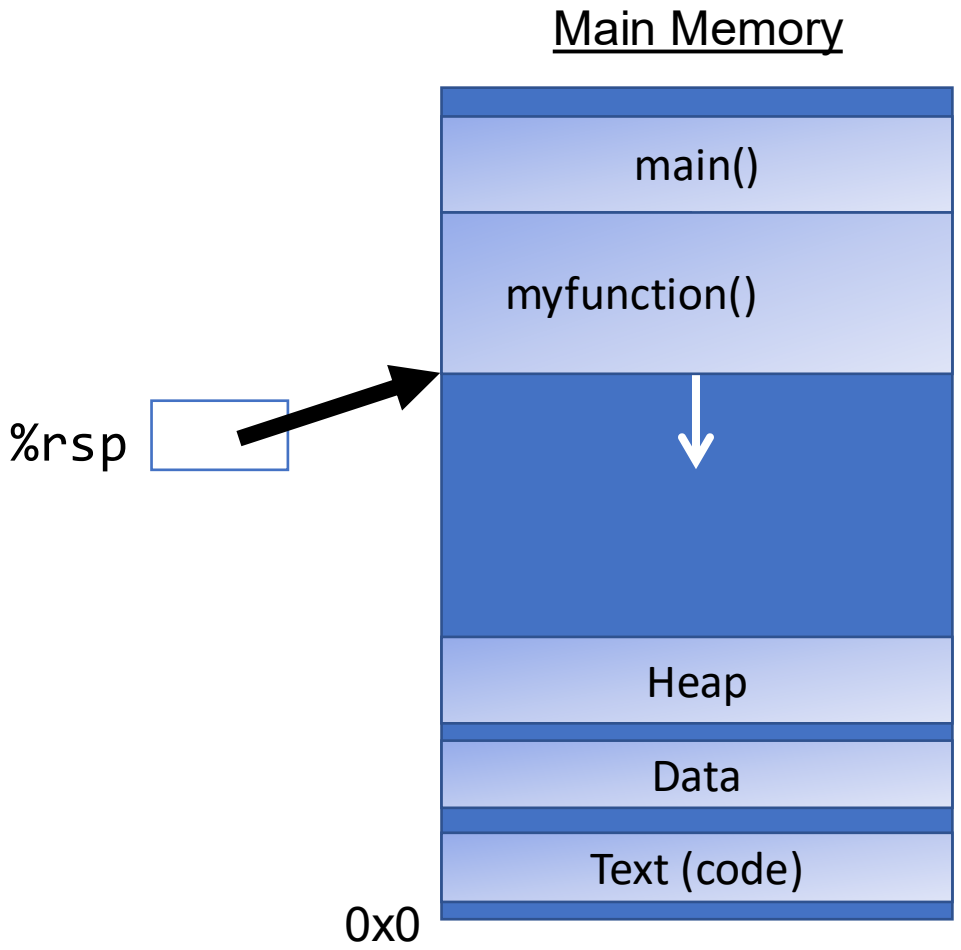
Some registers take on special responsibilities during program execution.

- **%rax** stores the return value
- **%rdi** stores the first parameter to a function
- **%rsi** stores the second parameter to a function
- **%rdx** stores the third parameter to a function
- **%rip** stores the address of the next instruction to execute
- **%rsp** stores the address of the current top of the stack

See the x86-64 Guide and Reference Sheet on the Resources webpage for more!

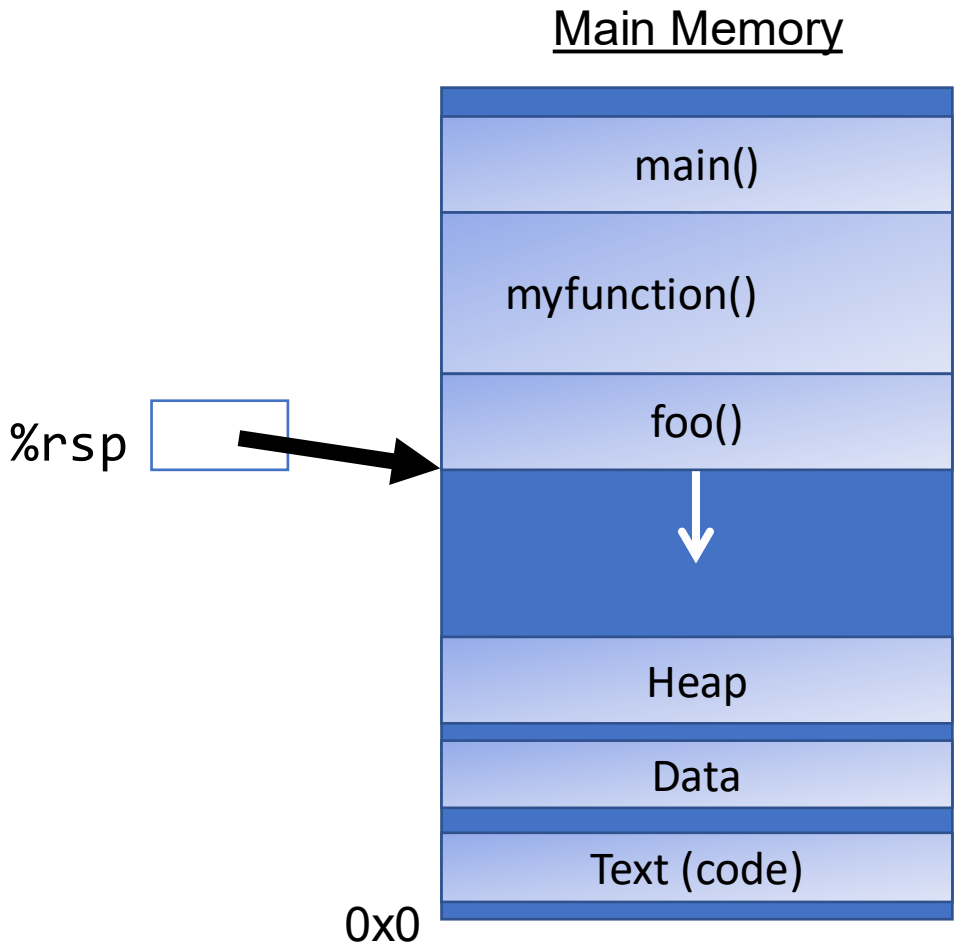
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



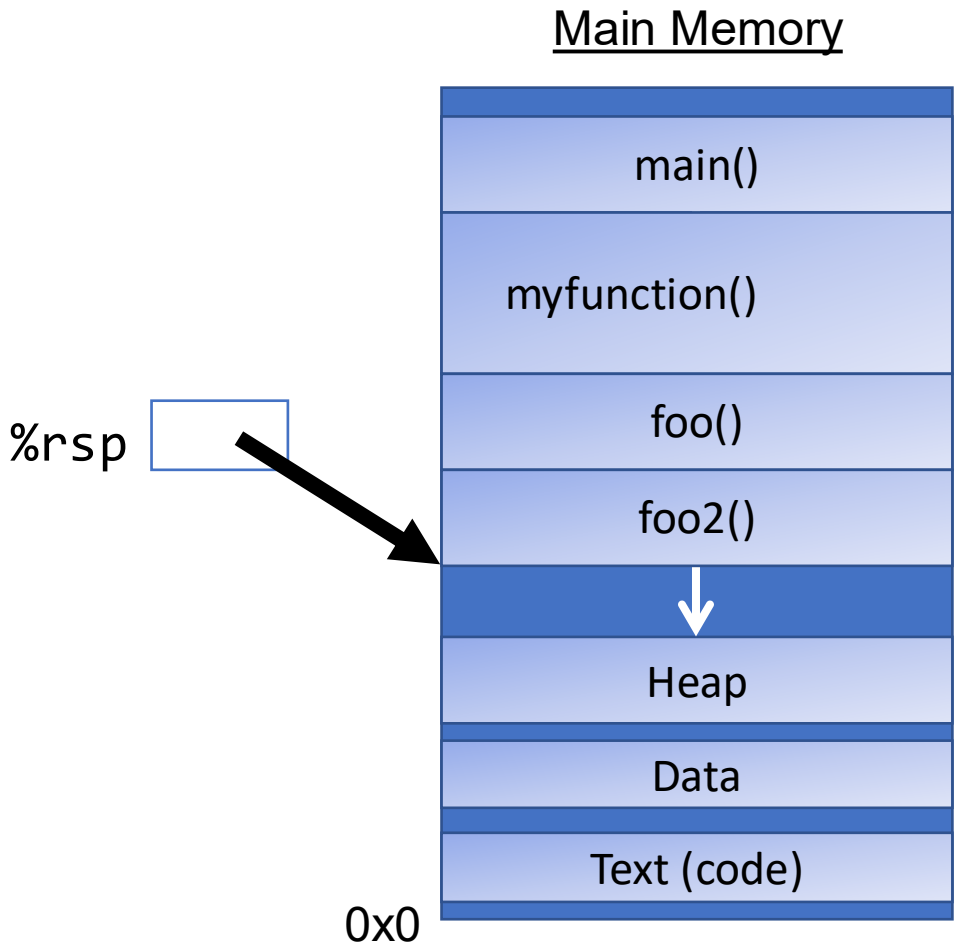
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



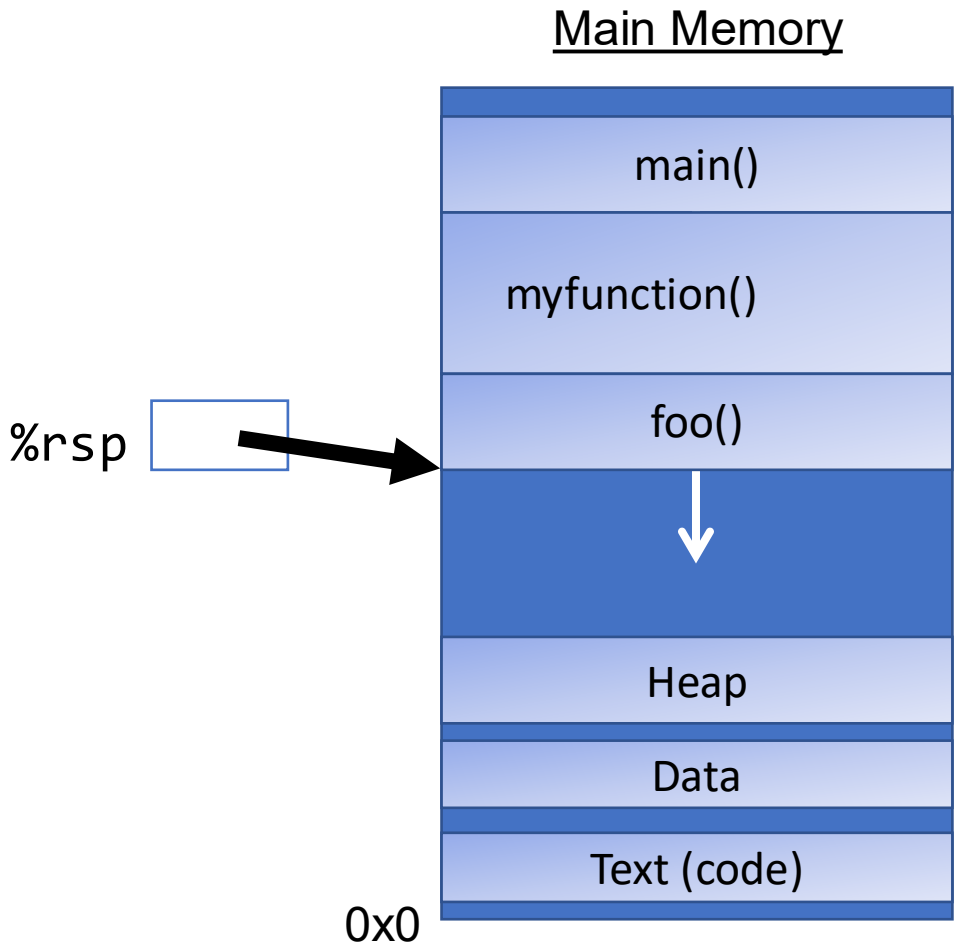
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



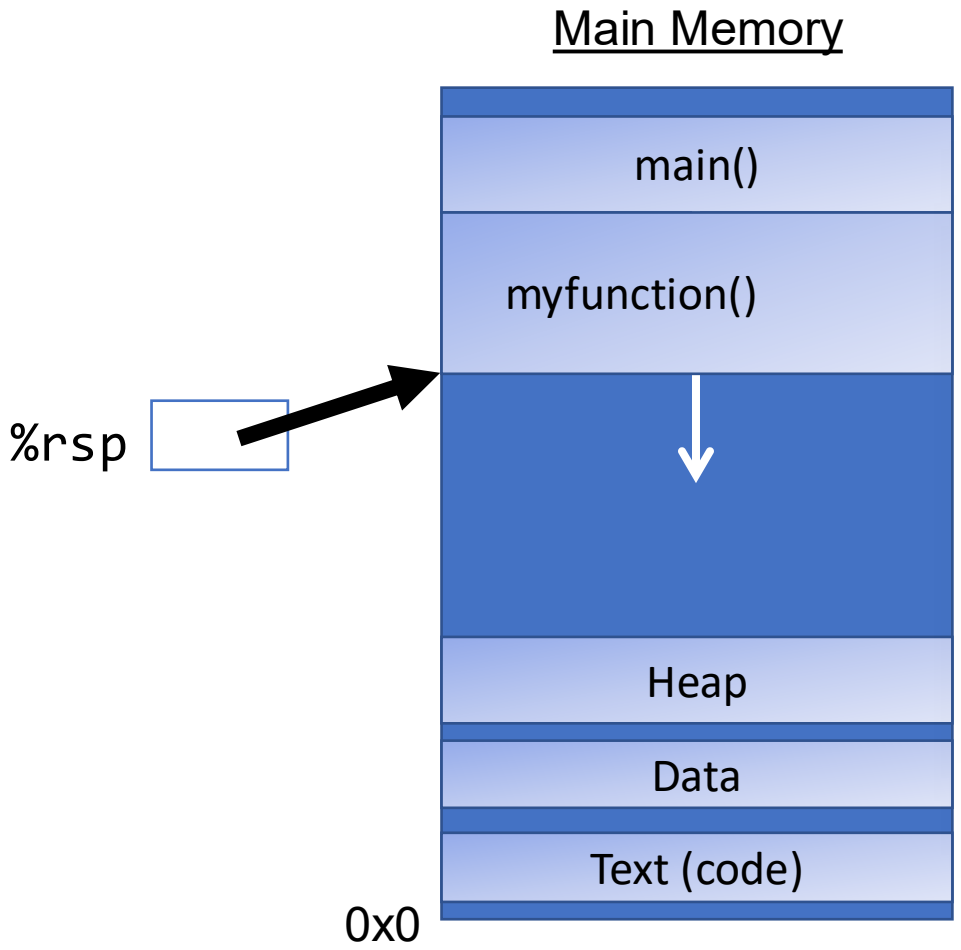
%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



%rsp

- **%rsp** is a special register that stores the address of the current "top" of the stack (the bottom in our diagrams, since the stack grows downwards).



Key idea: %rsp must point to the same place before a function is called and after that function returns, since stack frames go away when a function finishes.

The Stack

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons. Parameters are also stored in registers.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

The Stack

- So far, we've often seen local variables stored directly in registers, rather than on the stack as we'd expect. This is for optimization reasons. Parameters are also stored in registers.
- There are **three** common reasons that local data must be in memory:
  - We've run out of registers
    - There are a finite amount of registers, and we may run out of available registers.
    - Or, for parameters specifically, there are only 6 parameter registers; for parameters beyond 6, those are stored on the stack.
  - The '&' operator is used on it, so we must generate an address for it
  - They are arrays or structs (need to use address arithmetic)

How do you add/remove/copy things to/from the stack?

push

The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

push

The **push** instruction pushes the data at the specified source onto the top of the stack, adjusting **%rsp** accordingly.

Instruction	Effect
`pushq S`	`R[%rsp] ⟵ R[%rsp] - 8;` `M[R[%rsp]] ⟵ S`

- This behavior is equivalent to the following, but pushq is a shorter instruction:

```
subq $8, %rsp
movq  S, (%rsp)
```

pop

The **pop** instruction pops the topmost data from the stack and stores it in the specified destination, adjusting **%rsp** accordingly.

Instruction	Effect
popq D	D ⟵ M[R[%rsp]] R[%rsp] ⟵ R[%rsp] + 8;

- This behavior is equivalent to the following, but popq is a shorter instruction:

```
movq (%rsp), D
addq $8, %rsp
```

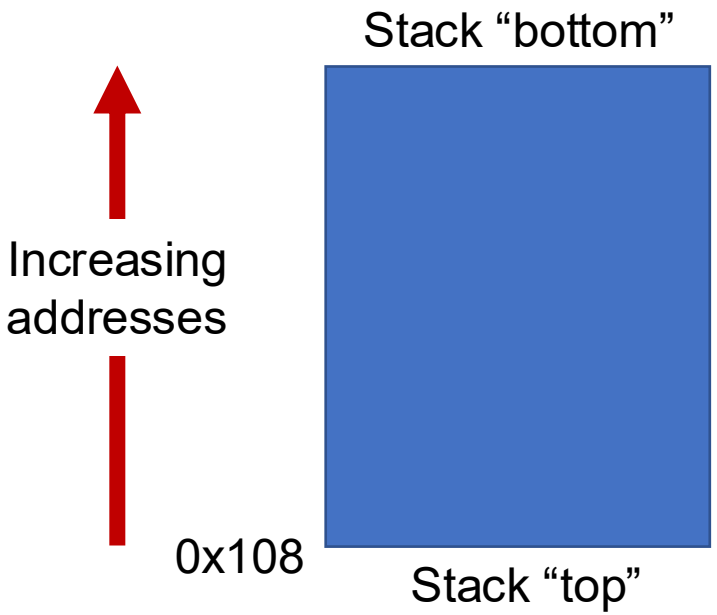
- **Note**: this *does not* remove/clear out the data! It just increments %rsp to indicate the next time the stack grows it can overwrite that location.

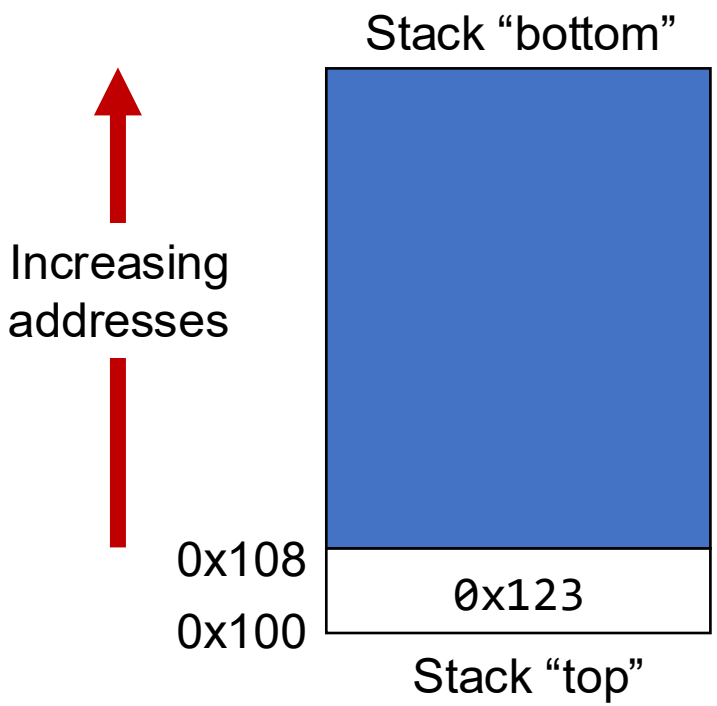
Example: push and pop

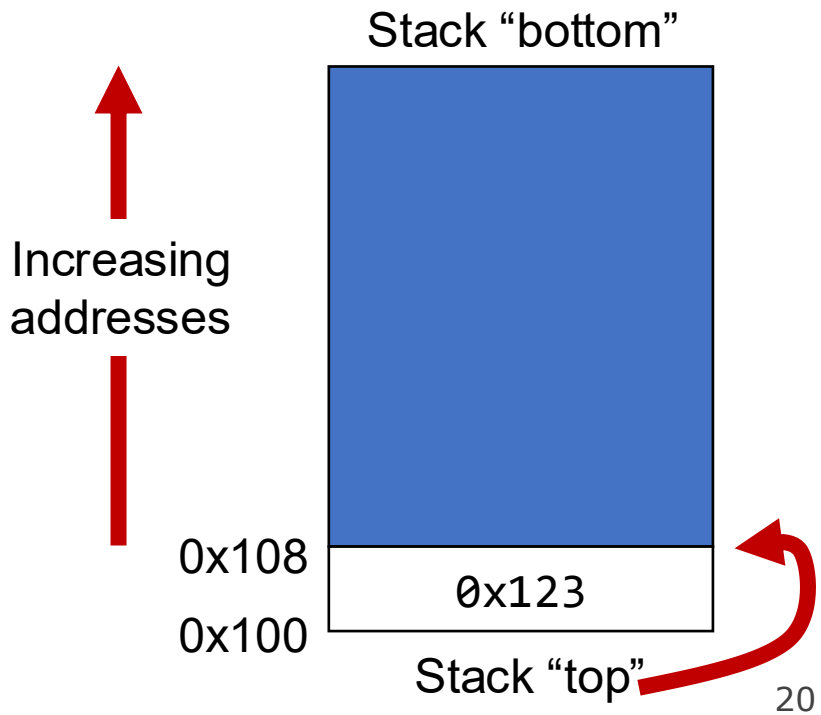
Initially	
%rax	0x123
%rdx	0
%rsp	0x108

pushq %rax	
%rax	0x123
%rdx	0
%rsp	0x100

popq %rdx	
%rax	0x123
%rdx	0x123
%rsp	0x108







The Stack

Another stack use method is to directly adjust where **%rsp** points to grow/shrink the stack.

```
0000000000401163 <main>:
  401163:    sub     $0x18,%rsp
  401167:    mov     %edi,0xc(%rsp)
  40116b:    mov     %rsi,(%rsp)
  ...
  401183:    add     $0x18,%rsp
  401187:    ret
```

The Stack

Another stack use method is to directly adjust where **%rsp** points to grow/shrink the stack.

```
0000000000401163 <main>:
  401163:    sub    $0x18,%rsp
  401167:    mov    %edi,0xc(%rsp)
  40116b:    mov    %rsi,(%rsp)
  ...
  401183:    add    $0x18,%rsp
  401187:    ret
```

Grow the stack by 24 bytes

The Stack

Another stack use method is to directly adjust where **%rsp** points to grow/shrink the stack.

```
0000000000401163 <main>:
   401163:    sub     $0x18,%rsp
   401167:    mov     %edi,0xc(%rsp)
   40116b:    mov     %rsi,(%rsp)
   ...
   401183:    add     $0x18,%rsp
   401187:    ret
```

Copy data onto the stack (in this new space)

The Stack

Another stack use method is to directly adjust where **%rsp** points to grow/shrink the stack.

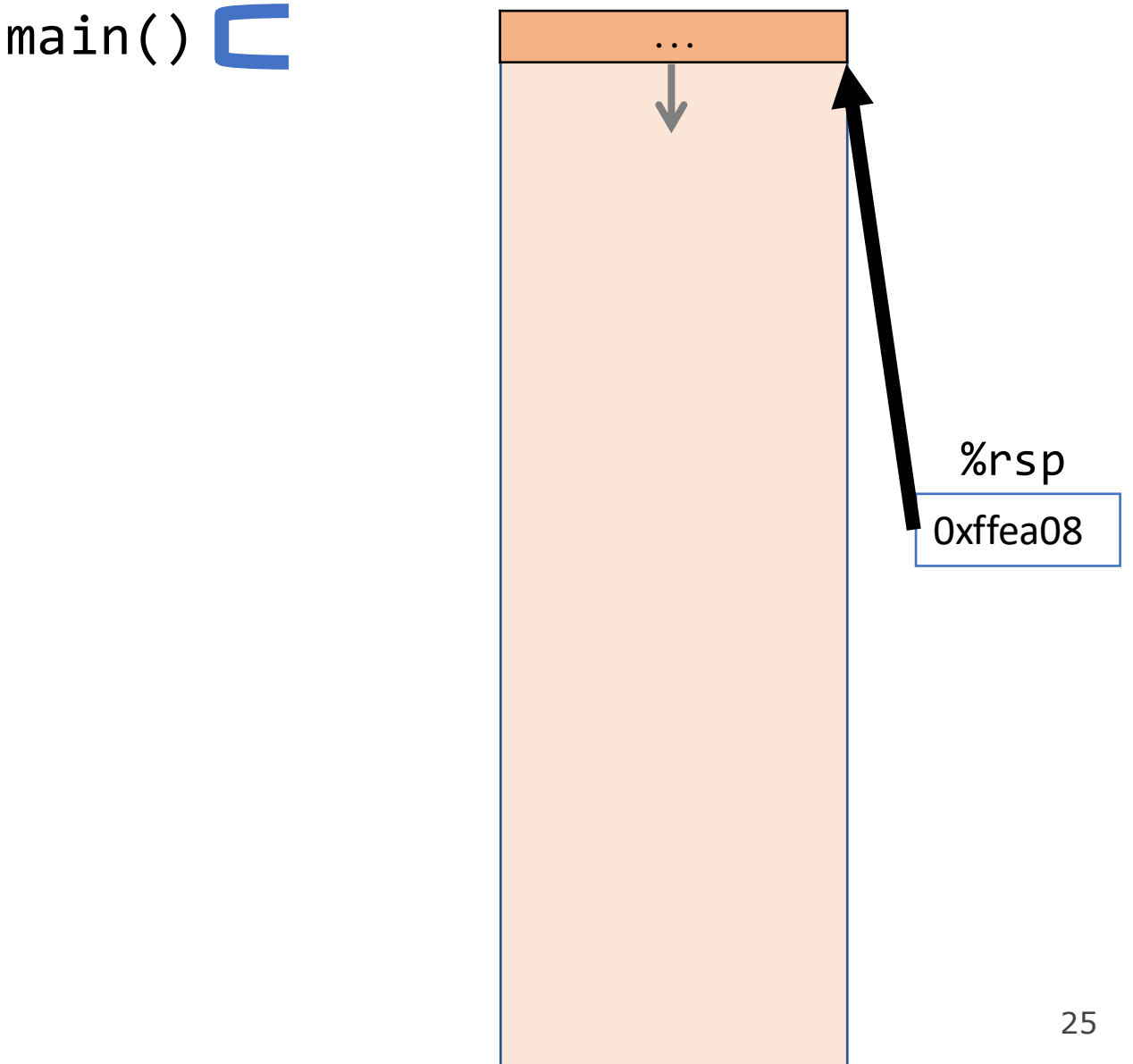
```
0000000000401163 <main>:
  401163:    sub     $0x18,%rsp
  401167:    mov     %edi,0xc(%rsp)
  40116b:    mov     %rsi,(%rsp)
  ...
  401183:    add     $0x18,%rsp
  401187:    ret
```

Before returning, shrink the stack back up by 24 bytes.

Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

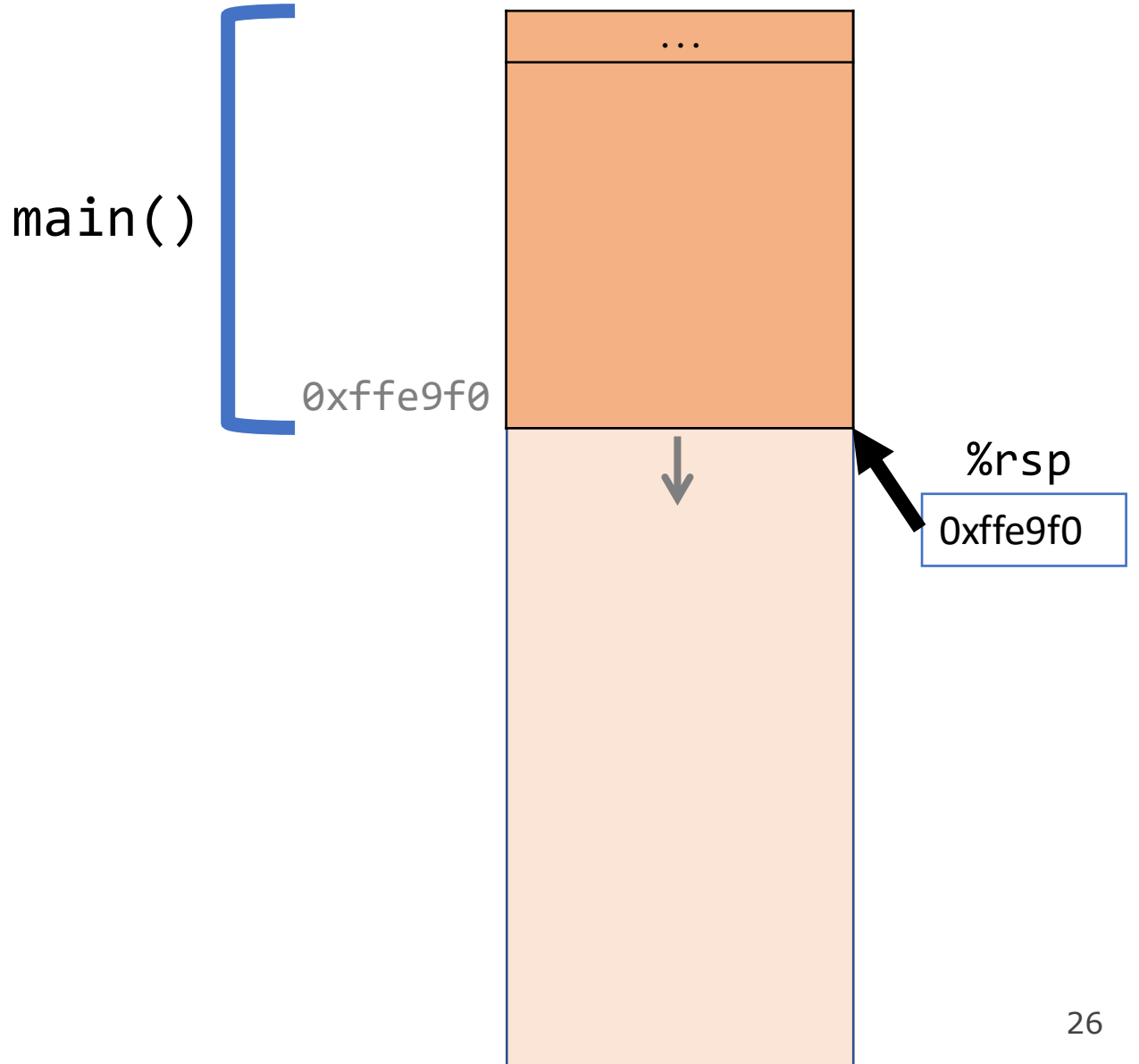
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

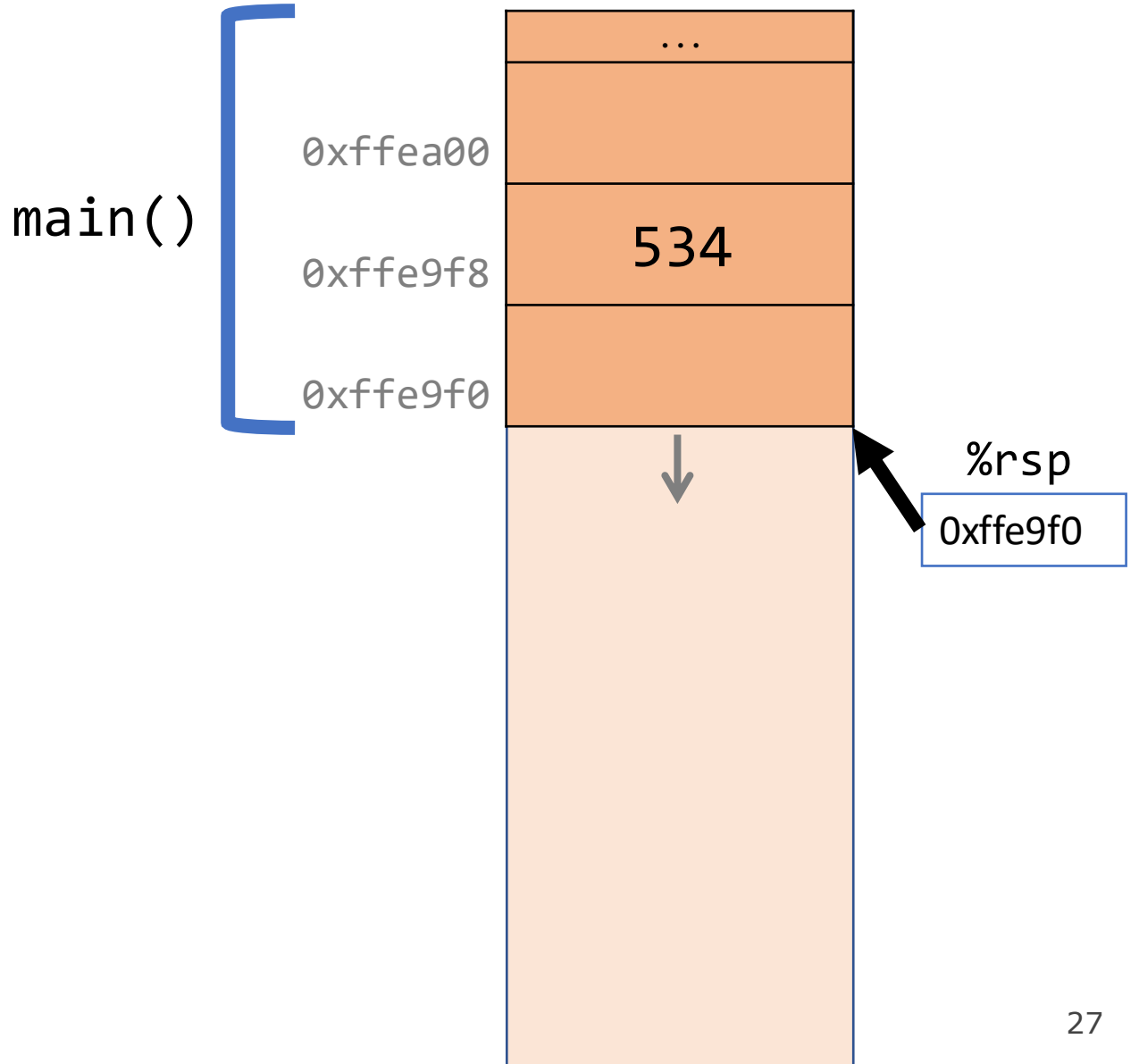
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

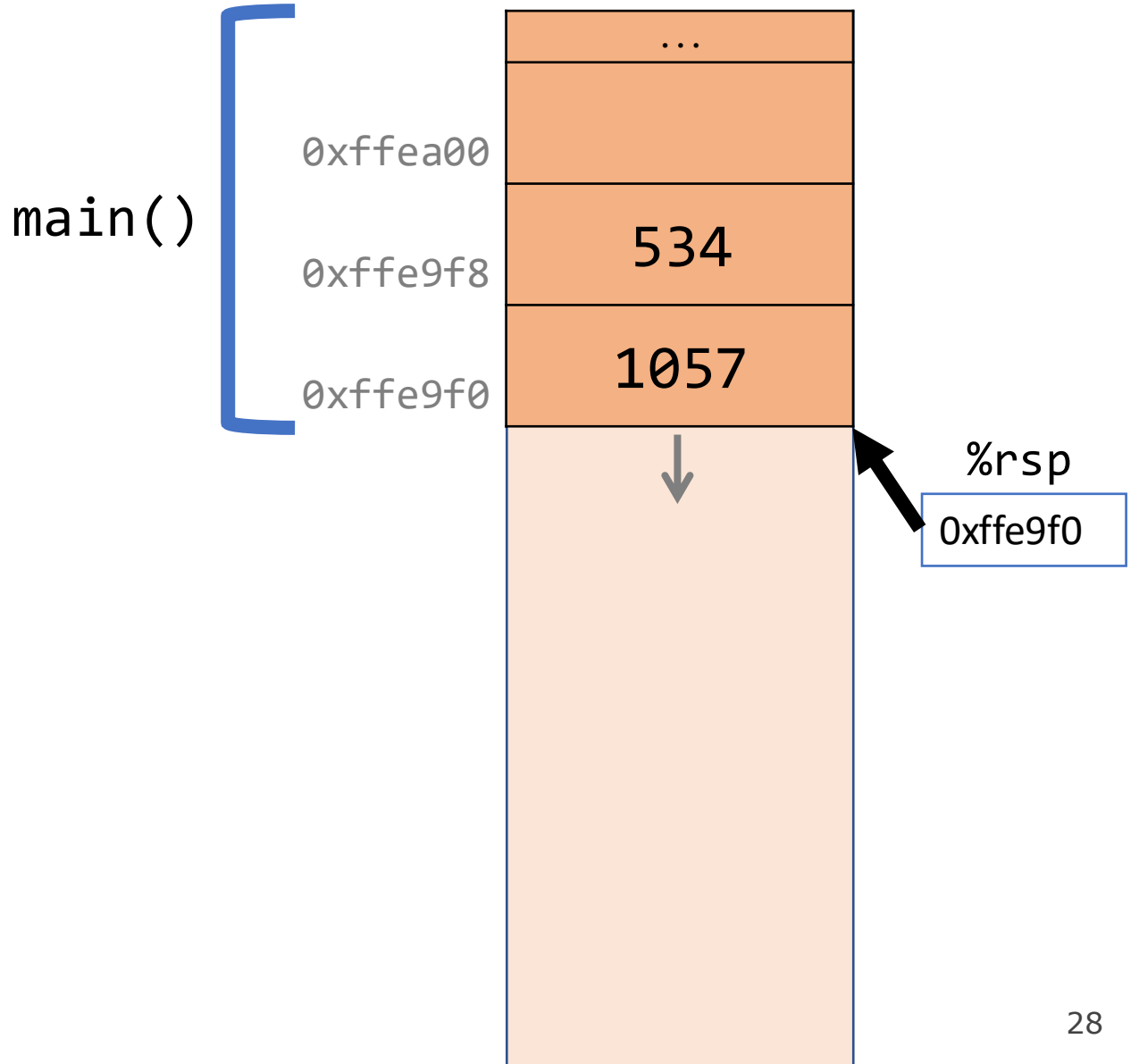
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

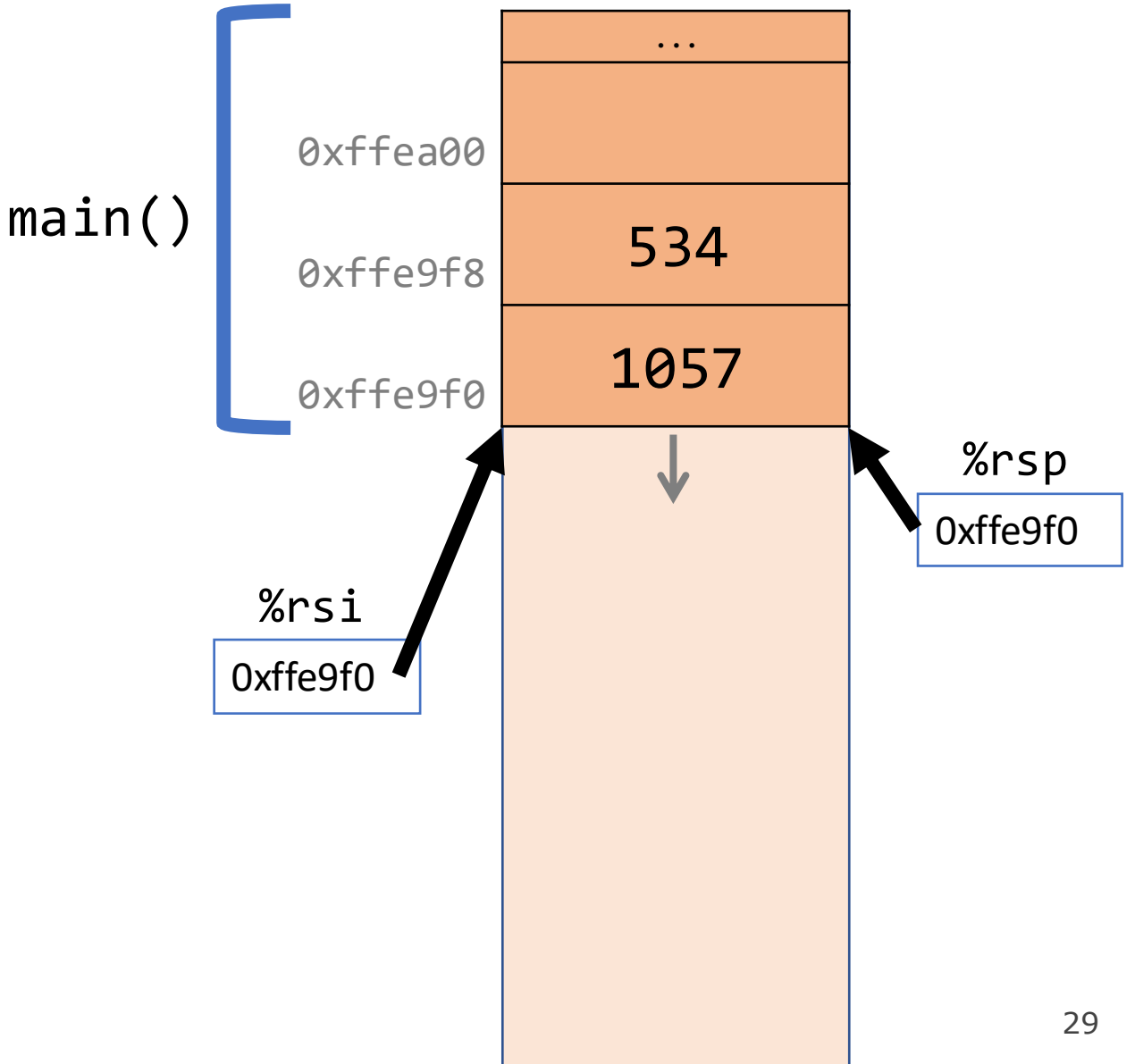
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

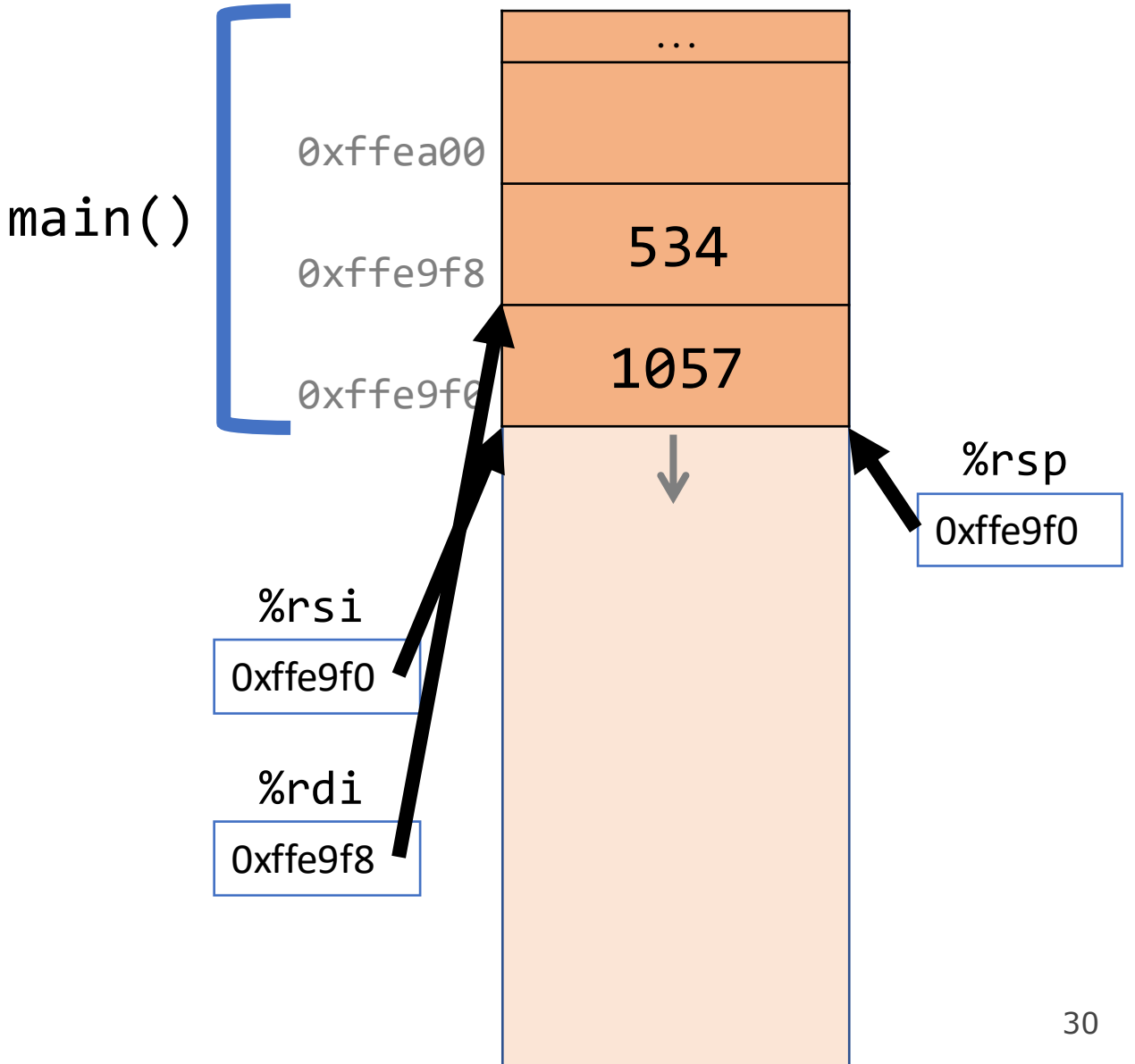
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

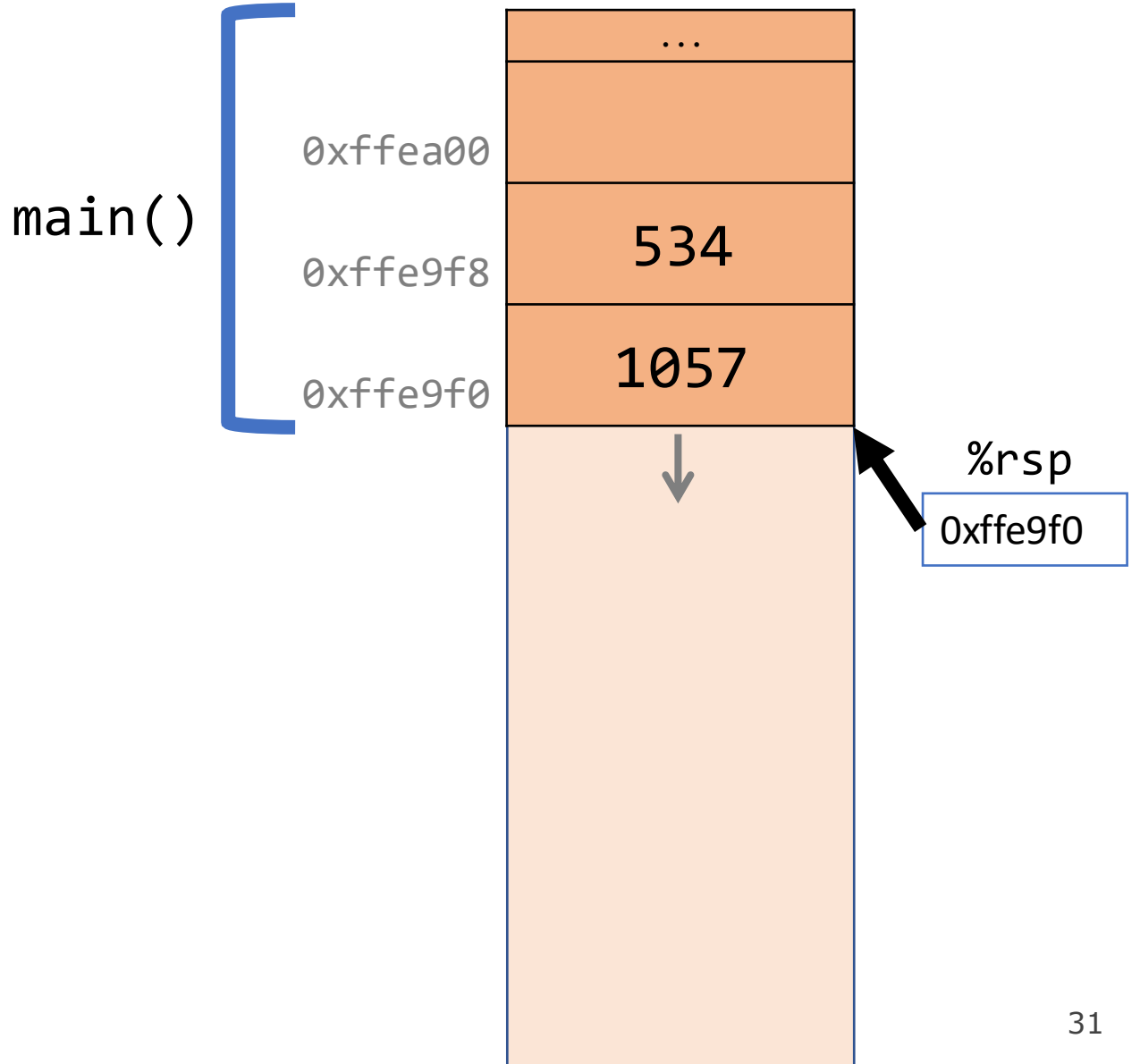
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

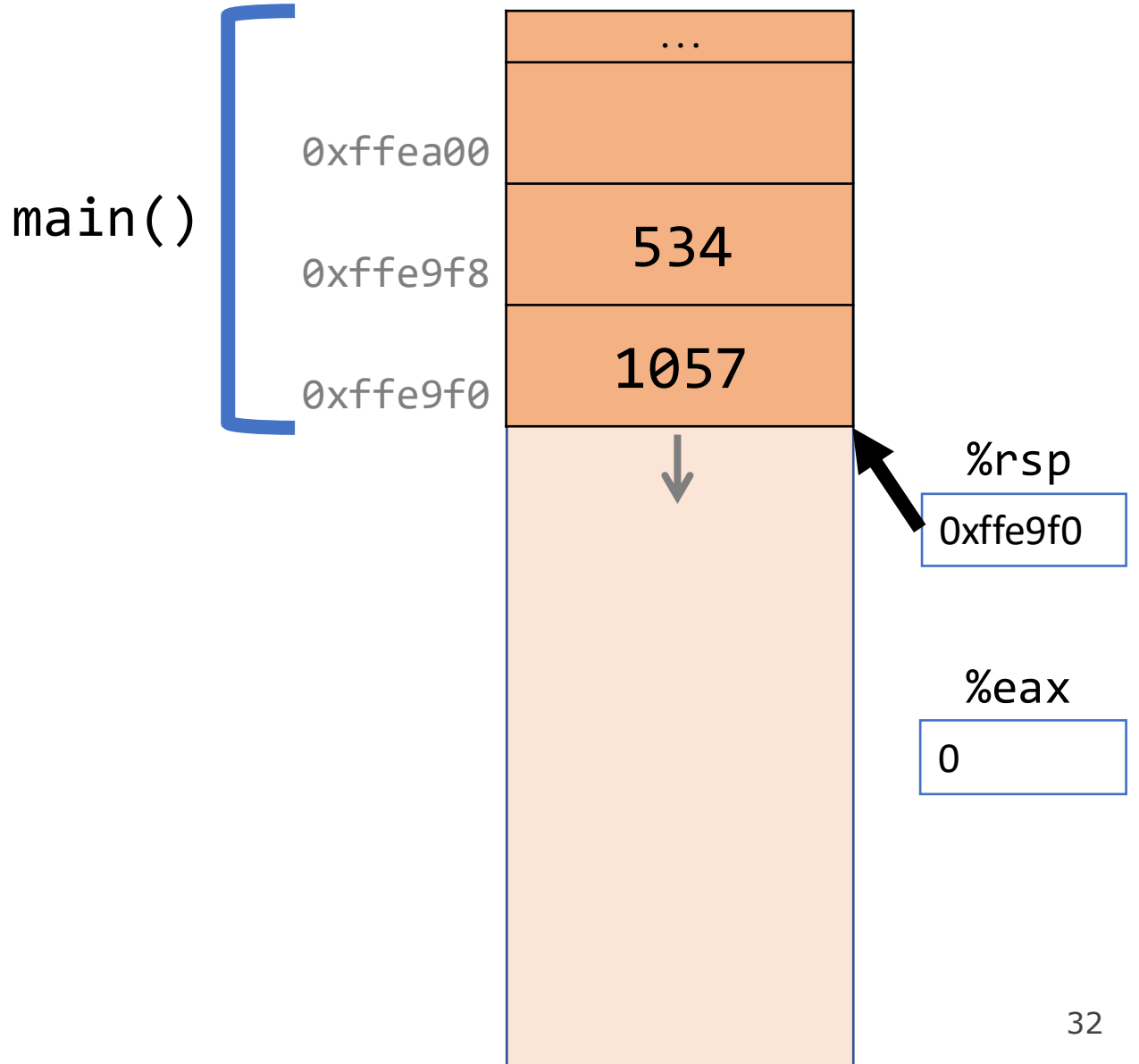
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

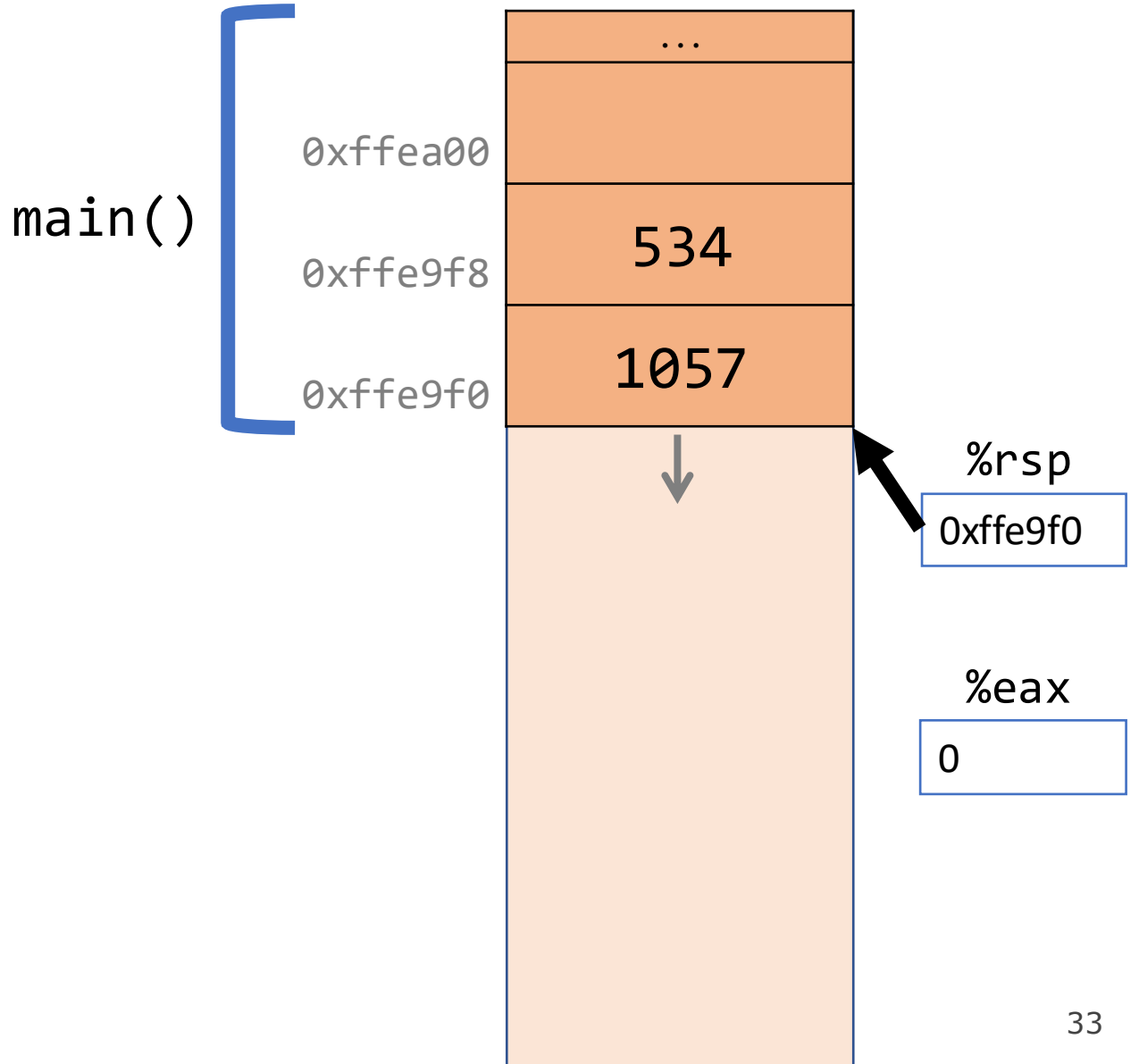
```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```



Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```

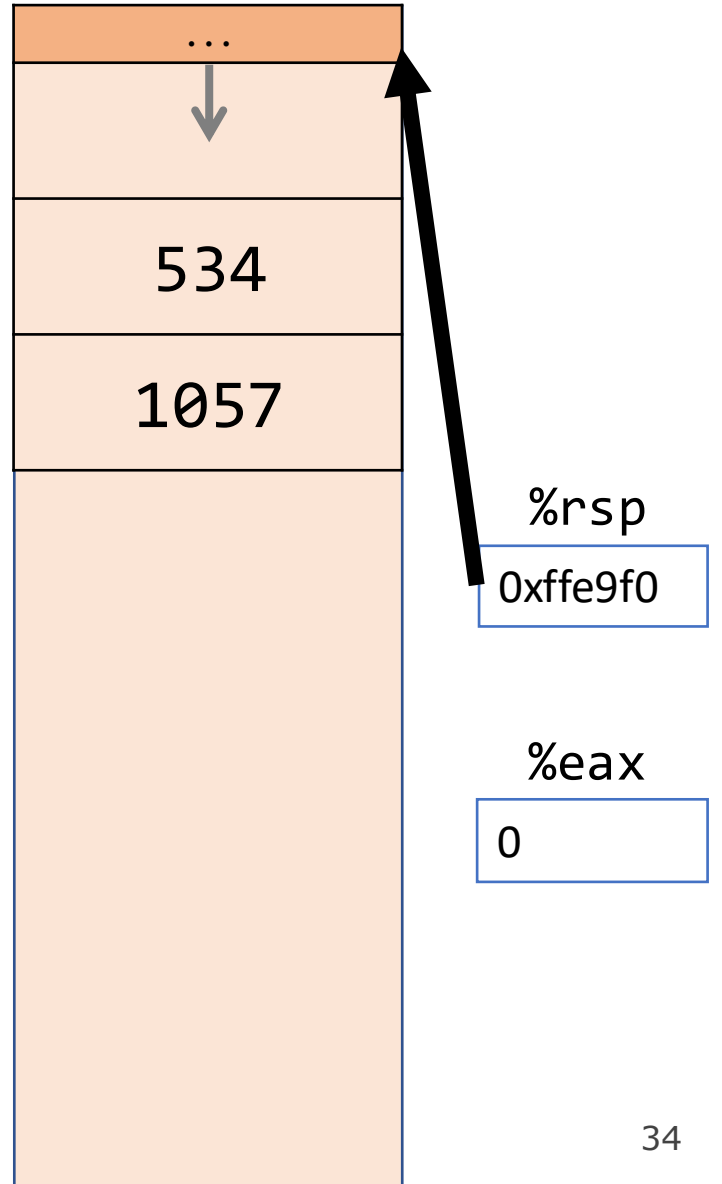


Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```

main()

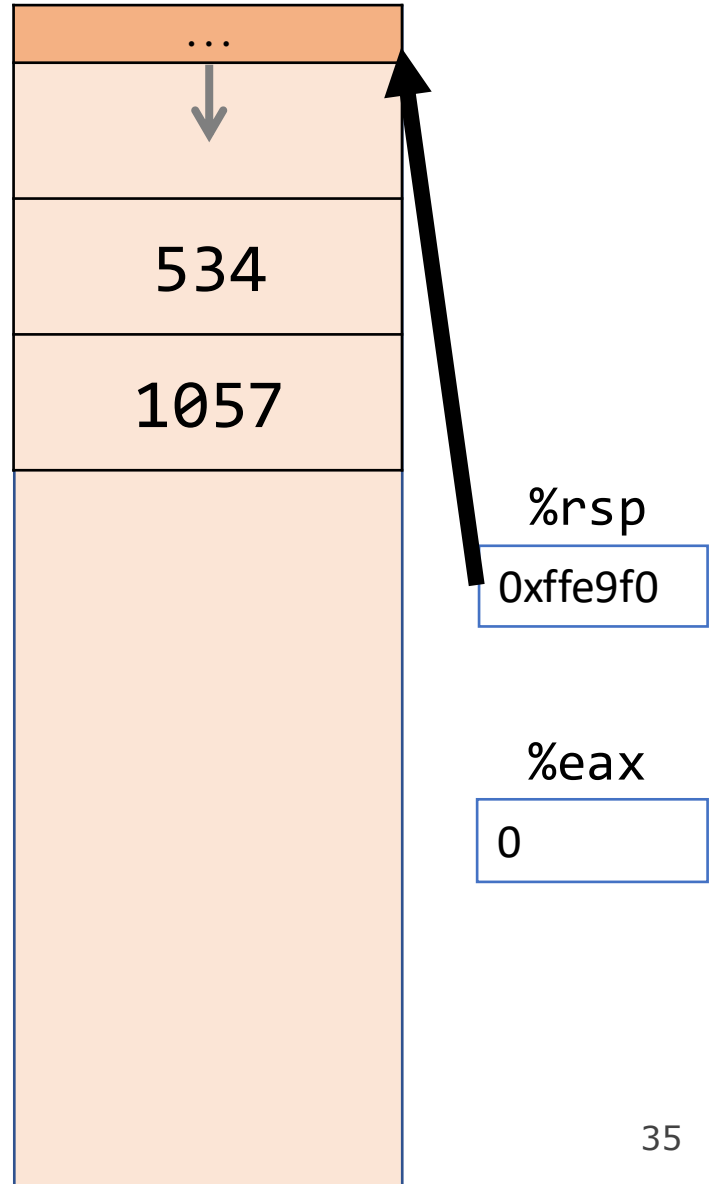


Using the Stack

```
int main(int argc, char *argv[]) {
    long arg1 = 534;
    long arg2 = 1057;
    long *ptr1 = &arg1;
    long *ptr2 = &arg2;
    ...
    return 0;
}
```

```
  subq $0x18, %rsp
  movq $0x216, 0x8(%rsp)
  movq $0x421, (%rsp)
  movq %rsp, %rsi
  leaq 0x8(%rsp), %rdi
  ...
  movl $0, %eax
  addq $0x18, %rsp
  ret
```

main()



Lecture Plan

- The Stack
- **Calling Functions**
  - Running another function's instructions
  - Parameters and return values
  - Trace: Calling a Function
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Calling Functions In Assembly

To call a function in assembly, we must do a few things:

- **Run another function's instructions** – %rip must be adjusted to execute the function being called and then resume the caller function afterwards.
- **Potentially pass parameters and receive a return value** – we must pass any parameters and receive any return value.

Terminology: **caller** function calls the **callee** function.

Lecture Plan

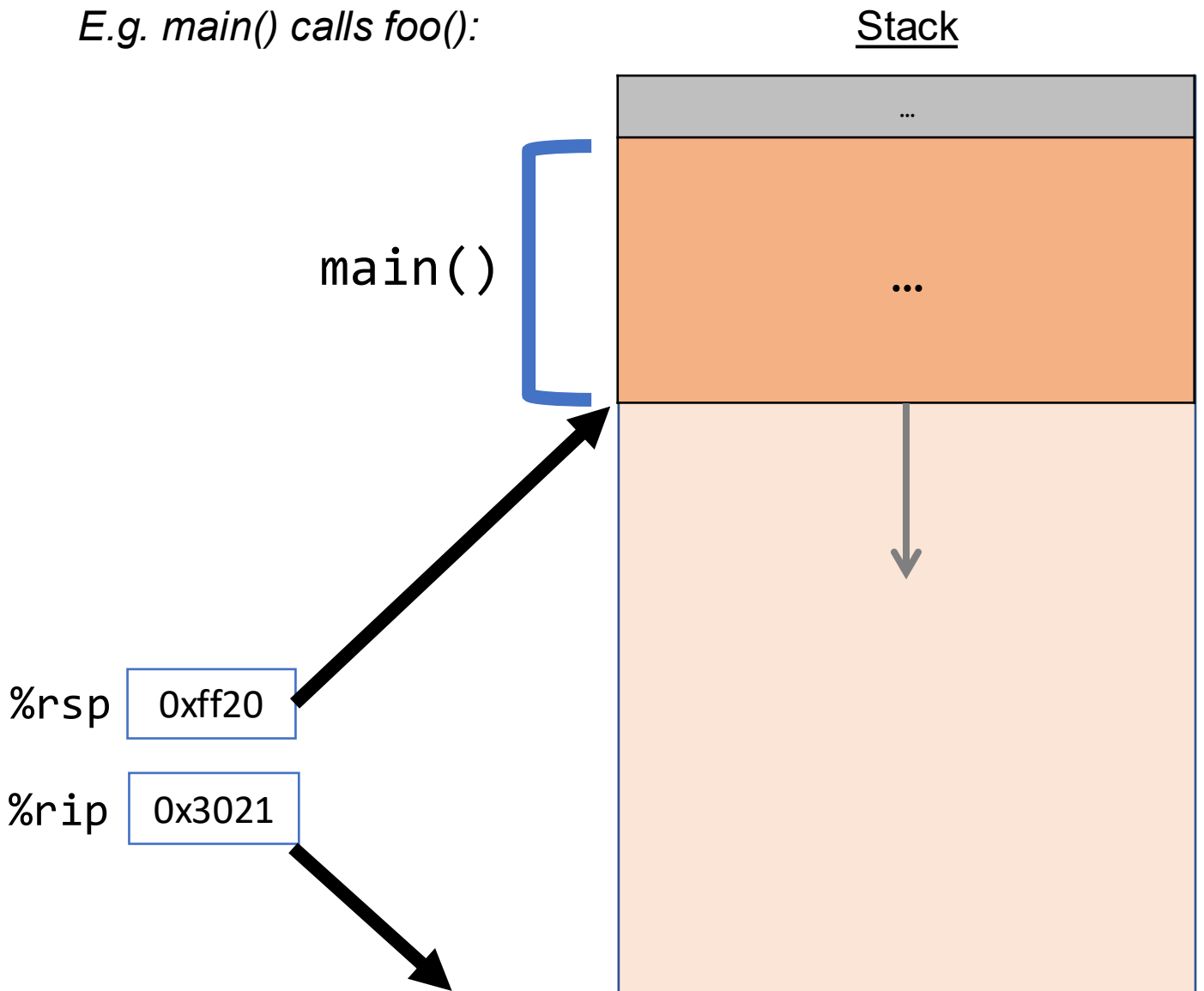
- The Stack
- **Calling Functions**
  - **Running another function's instructions**
  - Parameters and return values
  - Trace: Calling a Function
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

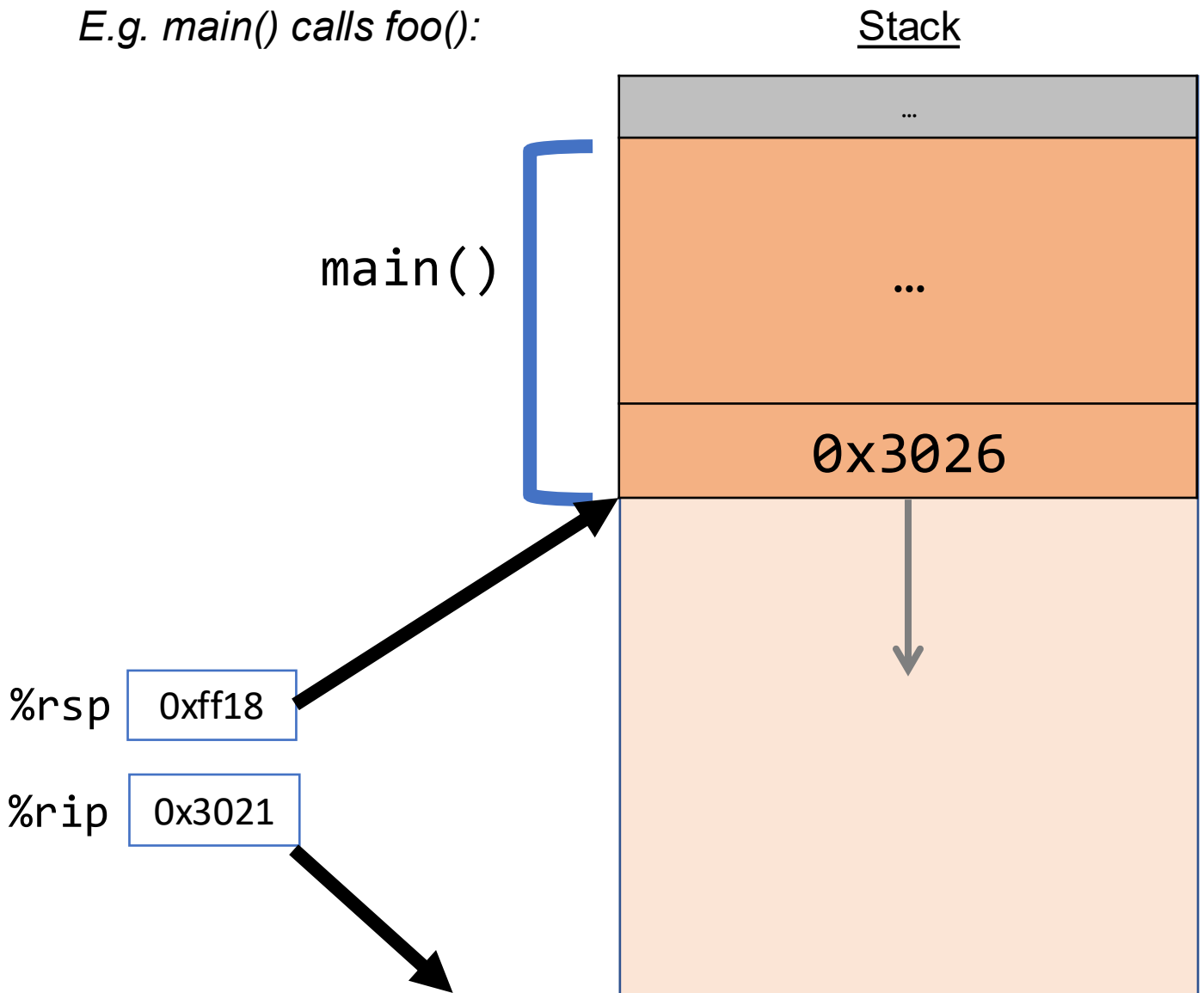
Solution: push the next value of %rip onto the stack. Then run the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

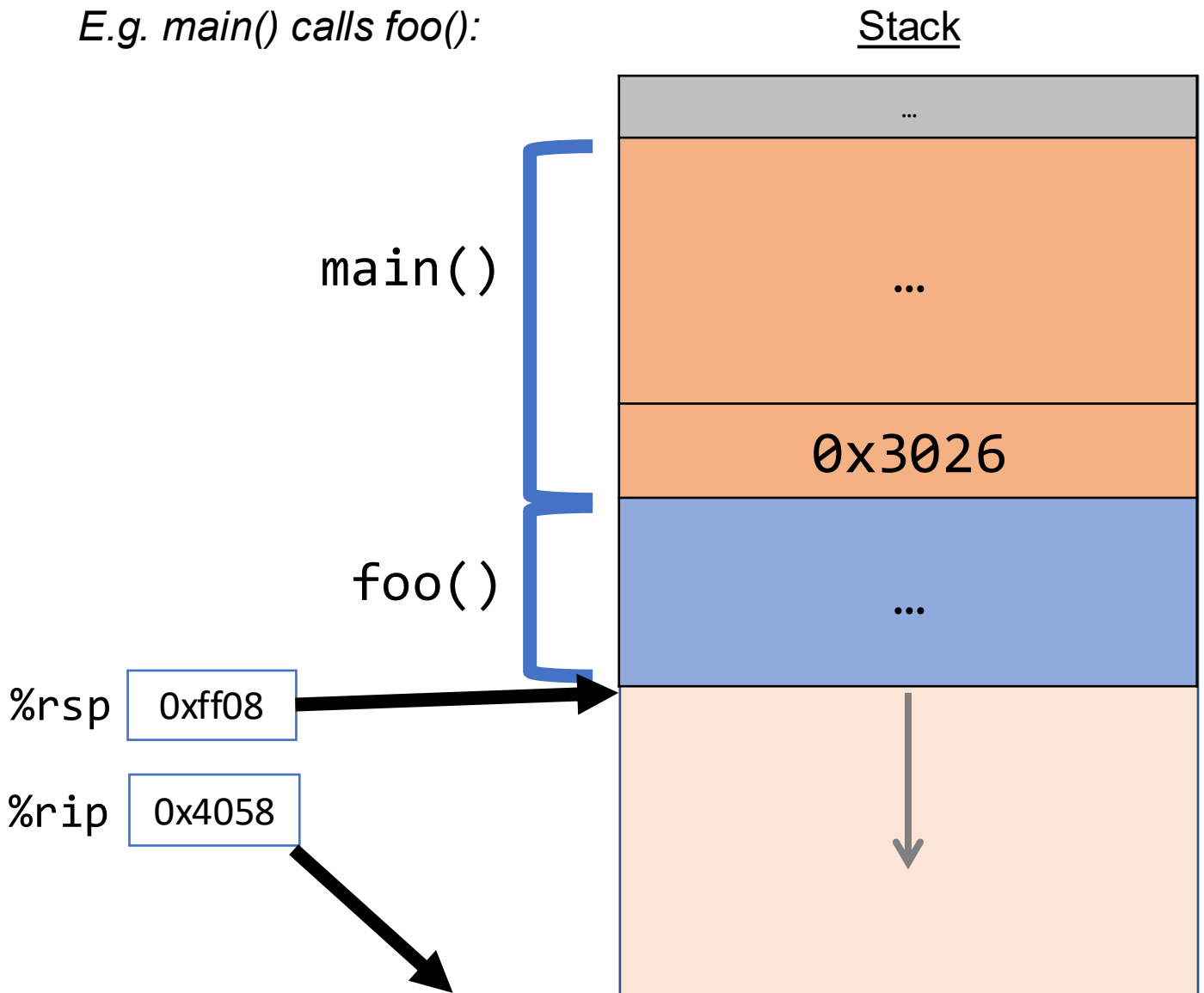
Solution: push the next value of %rip onto the stack. Then run the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

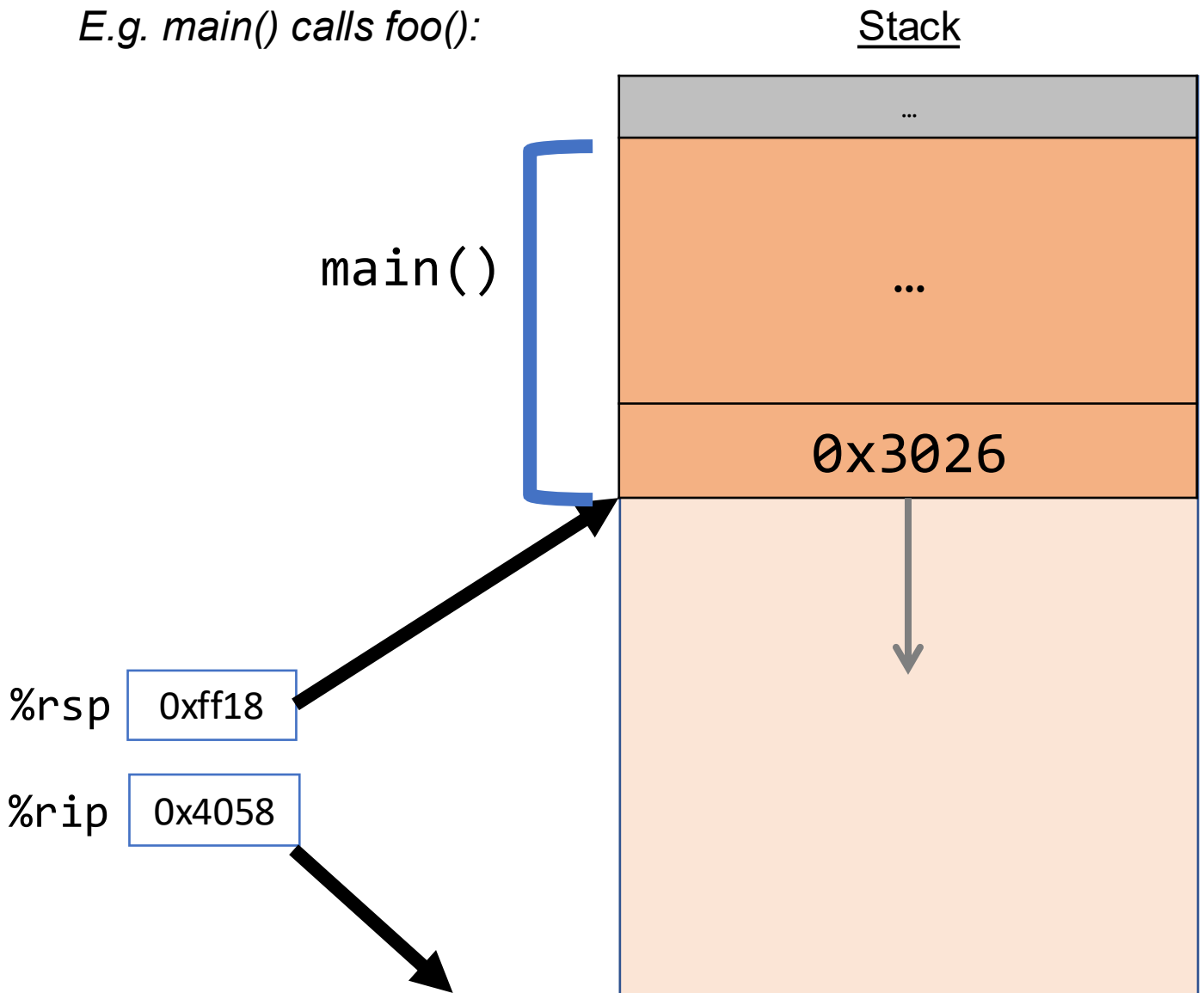
Solution: push the next value of %rip onto the stack. Then run the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

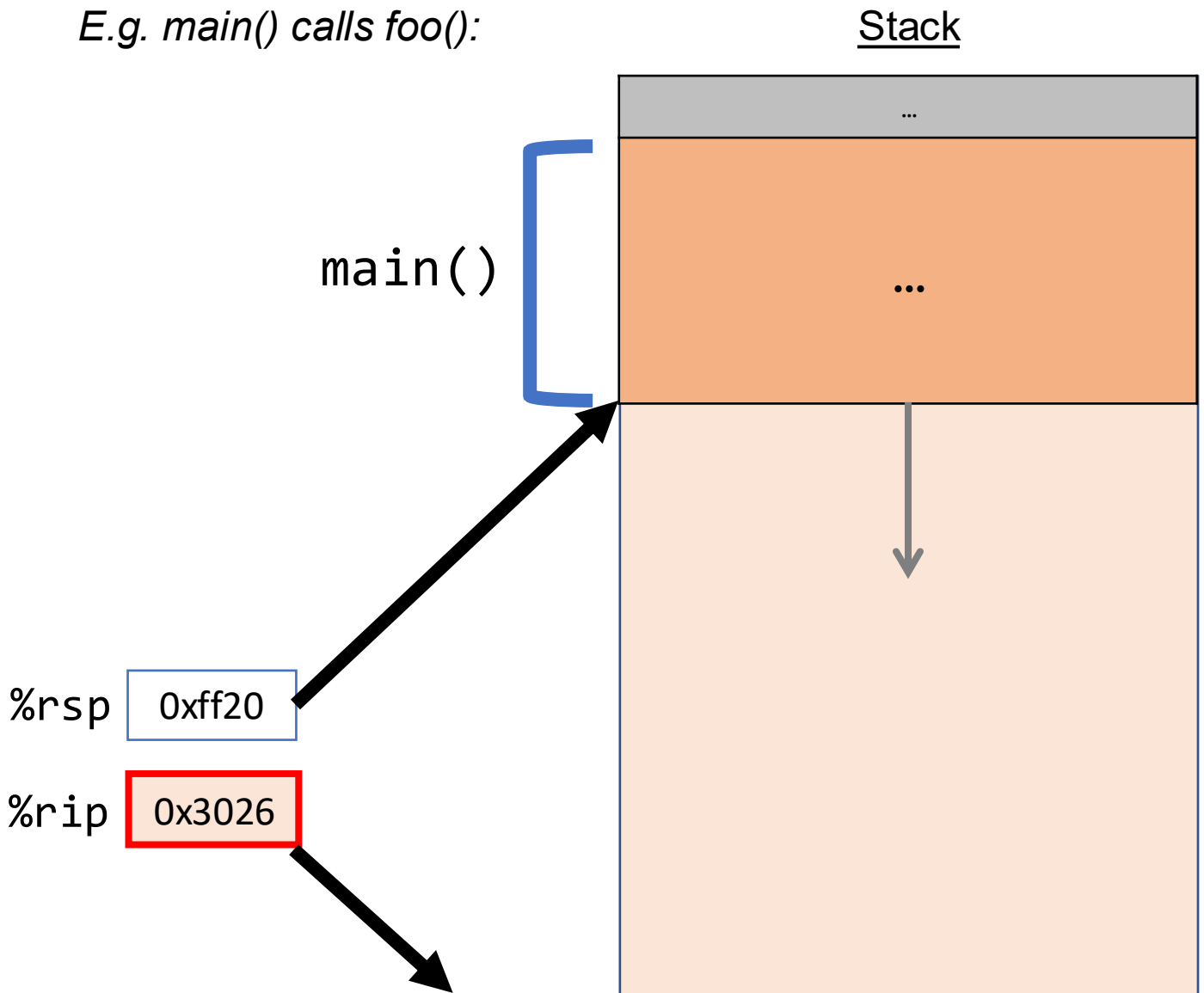
Solution: push the next value of %rip onto the stack. Then run the function. When it is finished, put this value back into %rip and continue executing.



Remembering Where We Left Off

Problem: %rip points to the next instruction to execute. To call a function, we must remember the *next* caller instruction to resume at after.

Solution: push the next value of %rip onto the stack. Then run the function. When it is finished, put this value back into %rip and continue executing.



Call And Return

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

The **ret** instruction pops this instruction address from the stack and stores it in %rip.

```
ret
```

The stored %rip value for a function is called its **return address.** It is the address of the instruction at which to resume the function's execution. (not to be confused with **return value**, which is the value returned from a function).

Registers

What does **call** do?

call pushes the next instruction address onto the stack and points %rip to another function's instructions.

Registers

What does **ret** do?

ret pops off the 8 bytes from the top of the stack and puts it into %rip, thus resuming execution in the caller.

ret is separate from the *return value* of the function (put in %rax).

Function Pointers

The **call** instruction pushes the address of the instruction immediately following the **call** instruction onto the stack and sets %rip to point to the beginning of the specified function's instructions.

```
call Label
call *Operand
```

- Why would we use **call** with a register instead of hardcoding the function name in the assembly? *When would we not know the function to call until we run the code?*
- Function pointers! E.g. qsort – qsort calls a function stored in a parameter register.

Practice: call and ret

In the assembly below, what will the value of %rip be and what value will be stored on top of the stack as a result of the **call** instruction executing?

```
000000000040112a <main>:
  40112a: 48 83 ec 08          sub    $0x8,%rsp
  40112e: bf 05 00 00 00       mov    $0x5,%edi
  401133: e8 ee ff ff ff       callq  401126 <foo>
  401138: 89 c2                mov    %eax,%edx
  ...
```

Respond on PollEv:
pollev.com/cs107

L20. What's in %rip, and what's on the top of the stack?

Practice: call and ret

In the assembly below, what will the value of %rip be and what value will be stored on top of the stack as a result of the **call** instruction executing?

```
000000000040112a <main>:
  40112a: 48 83 ec 08          sub    $0x8,%rsp
  40112e: bf 05 00 00 00       mov    $0x5,%edi
  401133: e8 ee ff ff ff       callq  401126 <foo>
  401138: 89 c2                mov    %eax,%edx
  ...
```

0x401138 stored on the stack (return address)**, 0x401126** put in %rip (address of foo's first instruction)

Lecture Plan

- The Stack
- **Calling Functions**
  - Running another function's instructions
  - **Parameters and return values**
  - Trace: Calling a Function
- Register Restrictions

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Parameters and Return

- There are special registers that store parameters and the return value.
- To call a function, we must put any parameters we are passing into the correct registers. (%rdi, %rsi, %rdx, %rcx, %r8, %r9, in that order)
- Parameters beyond the first 6 are put on the stack (pushed right before calling the function – the function will be written to know to look there for those params).
- If the caller expects a return value, it looks in %rax after the callee completes.

Lecture Plan

- The Stack
- Calling Functions
  - Running another function's instructions
  - Parameters and return values
  - **Trace: Calling a Function**
- Register Restrictions

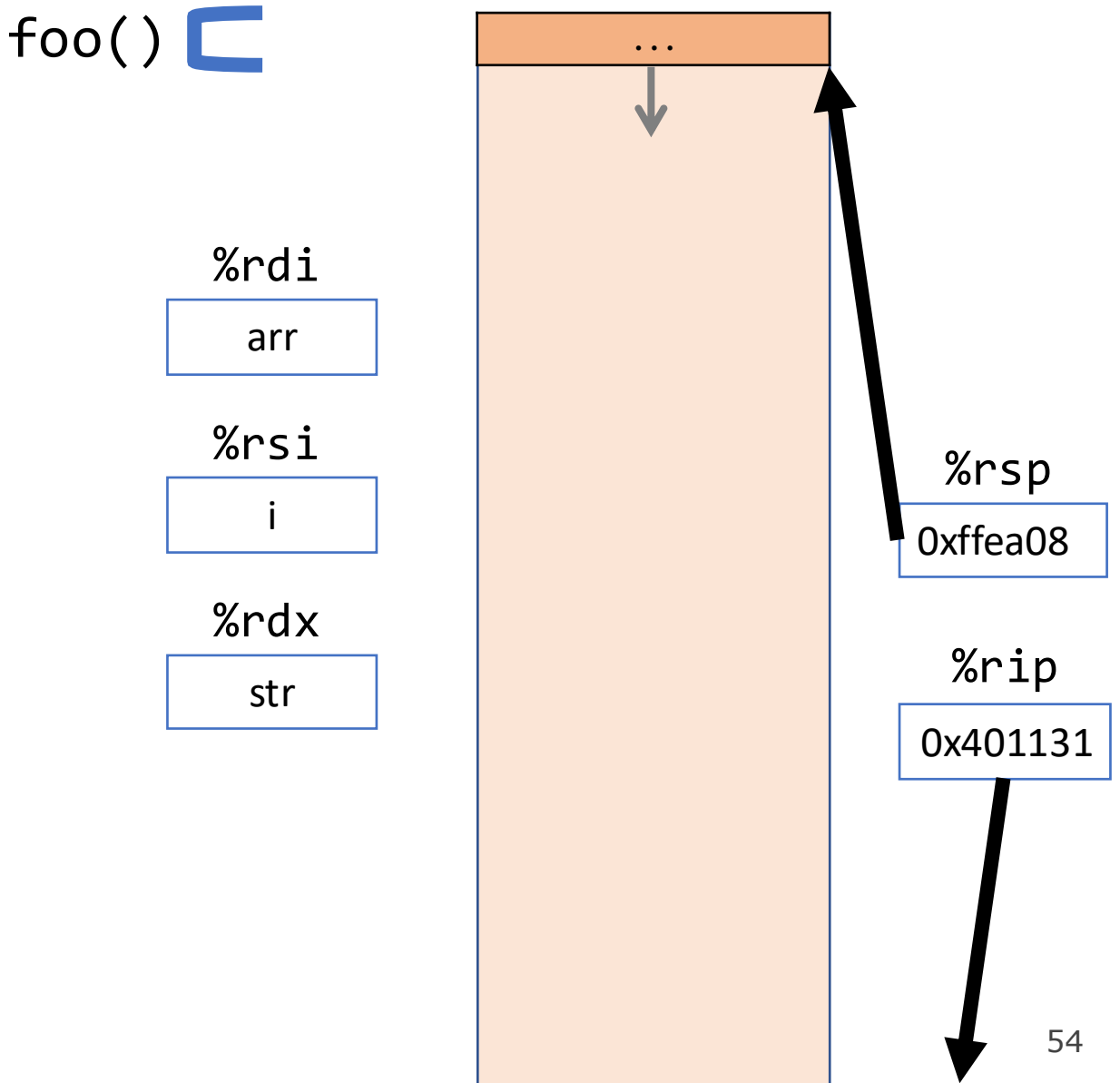
```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

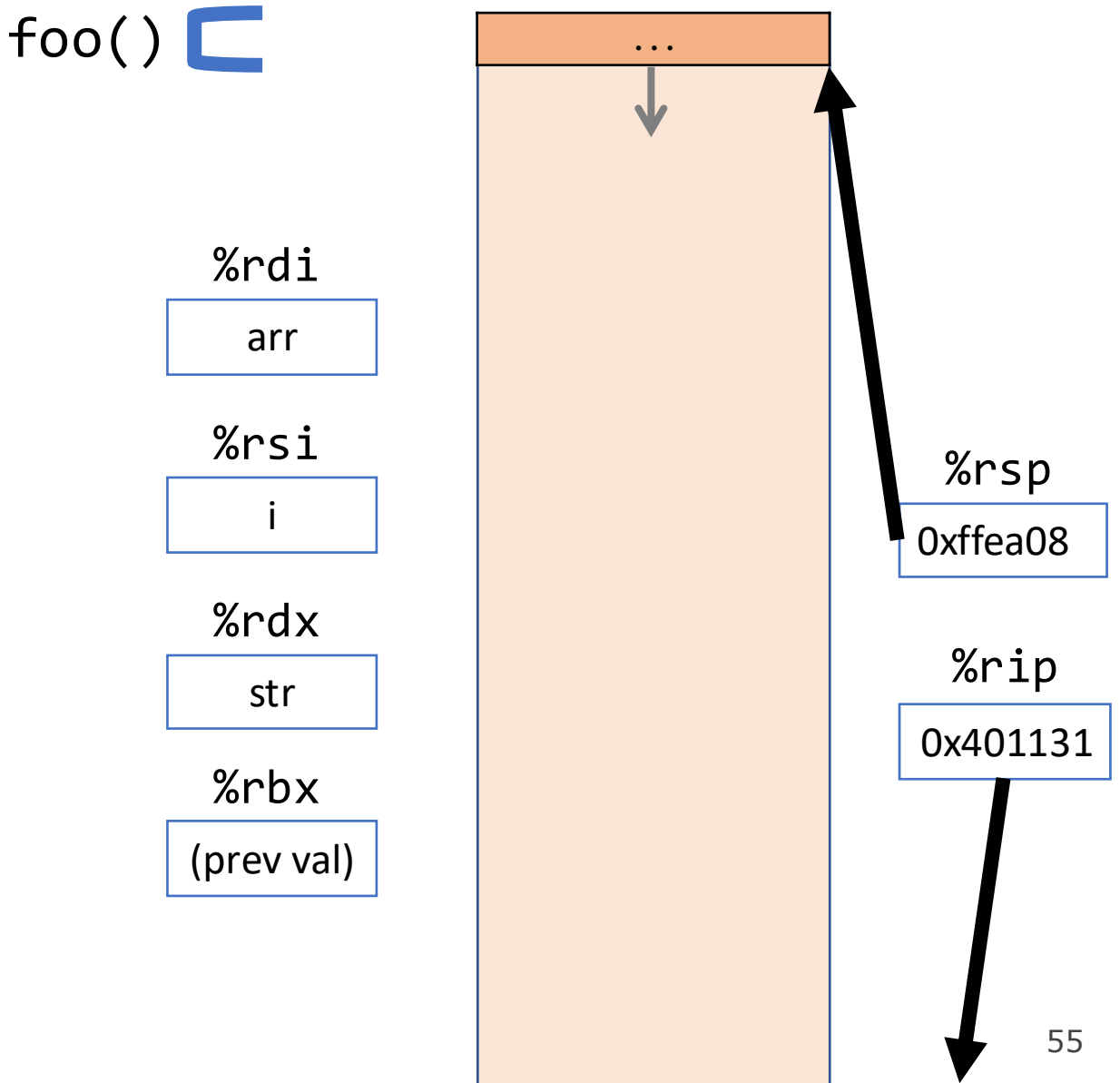


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

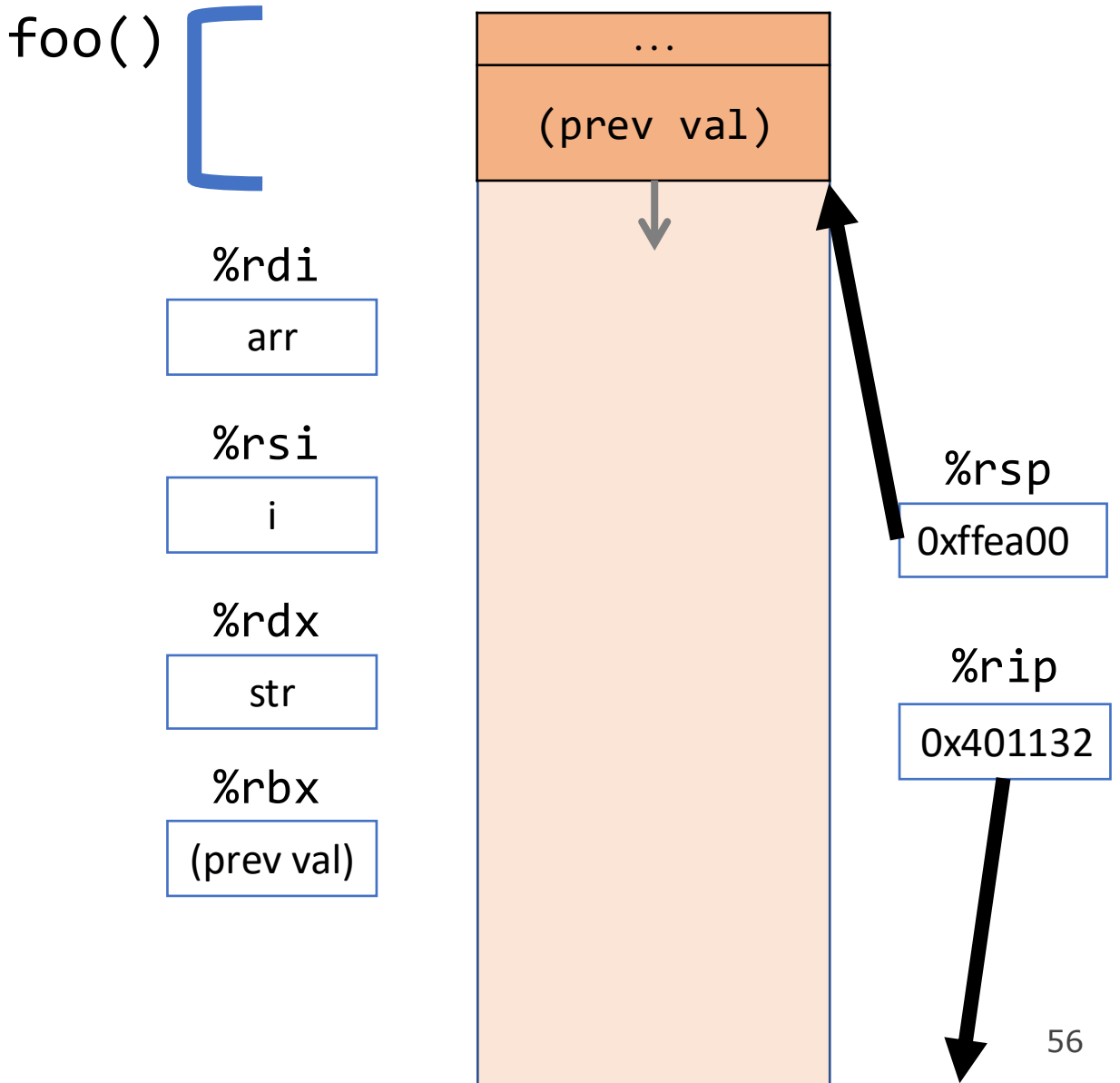


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

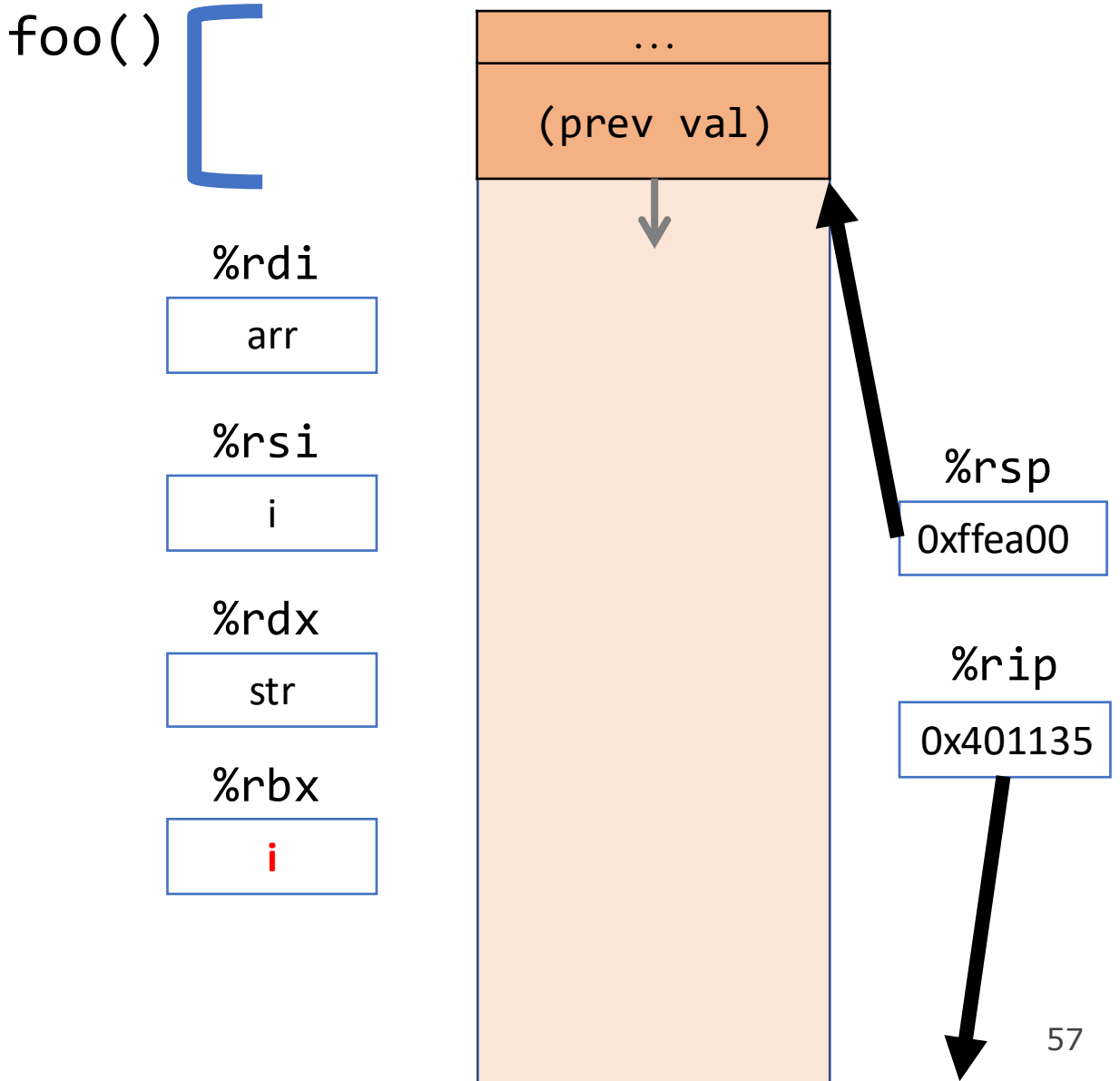


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

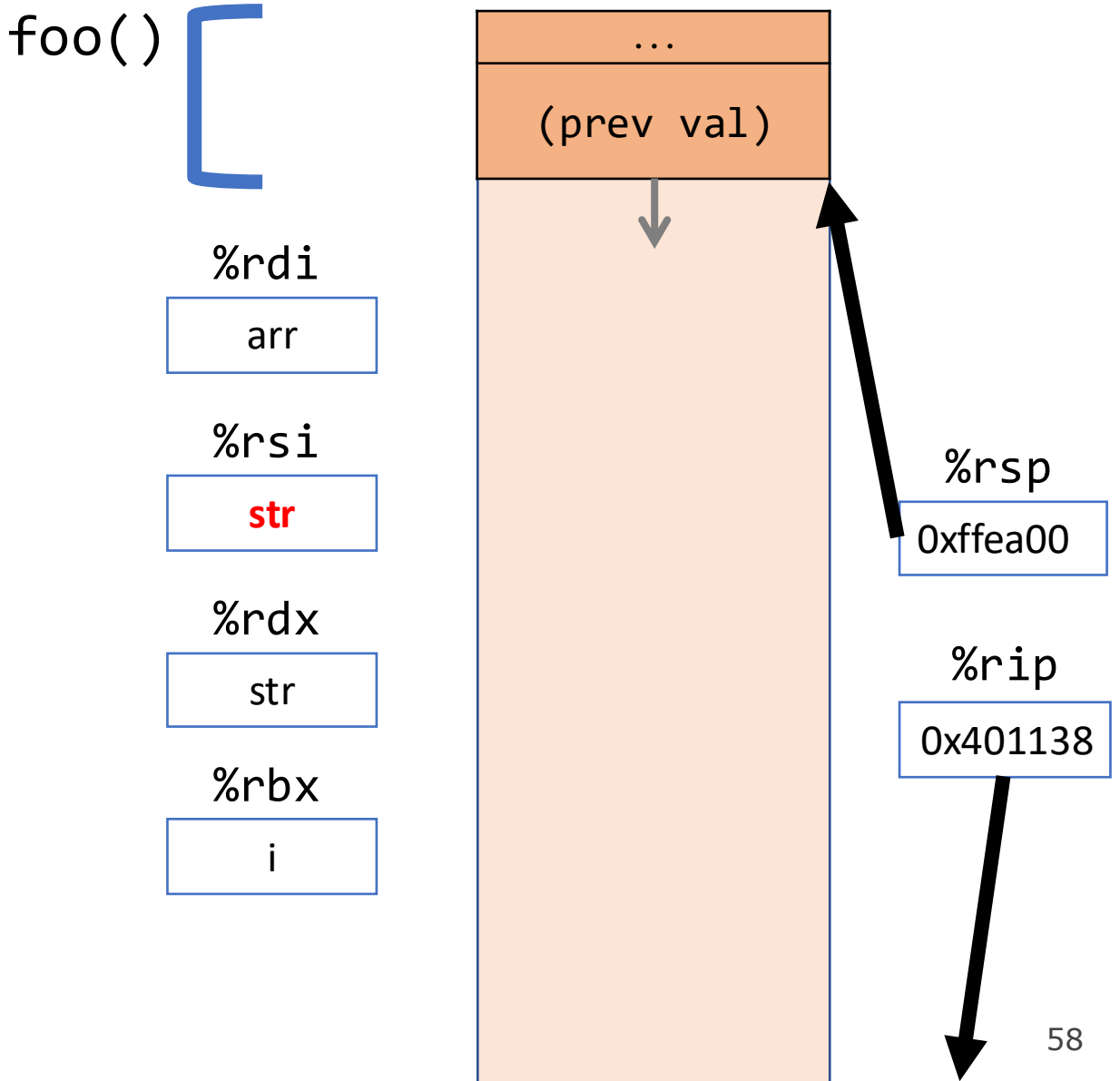


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

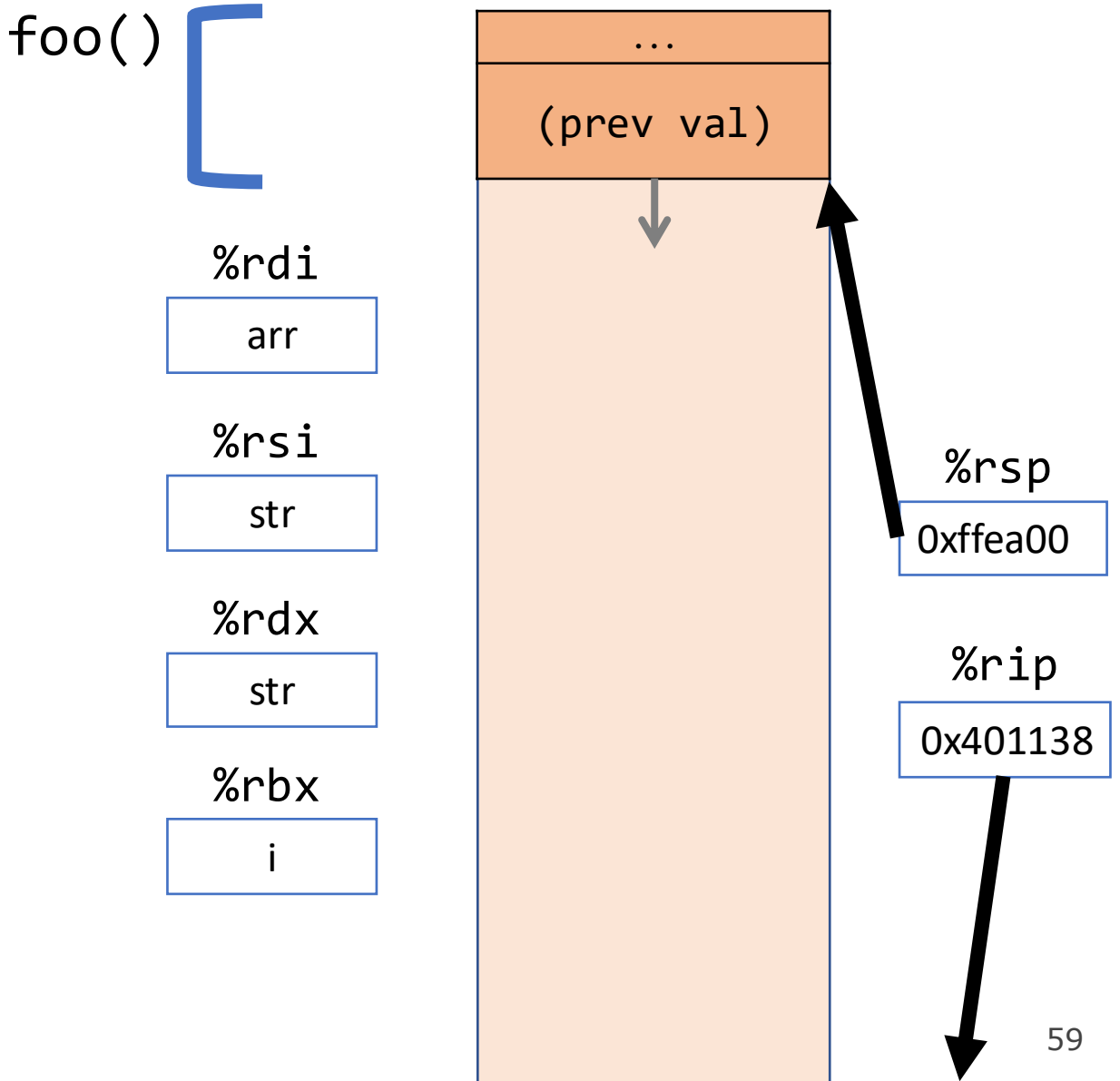


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

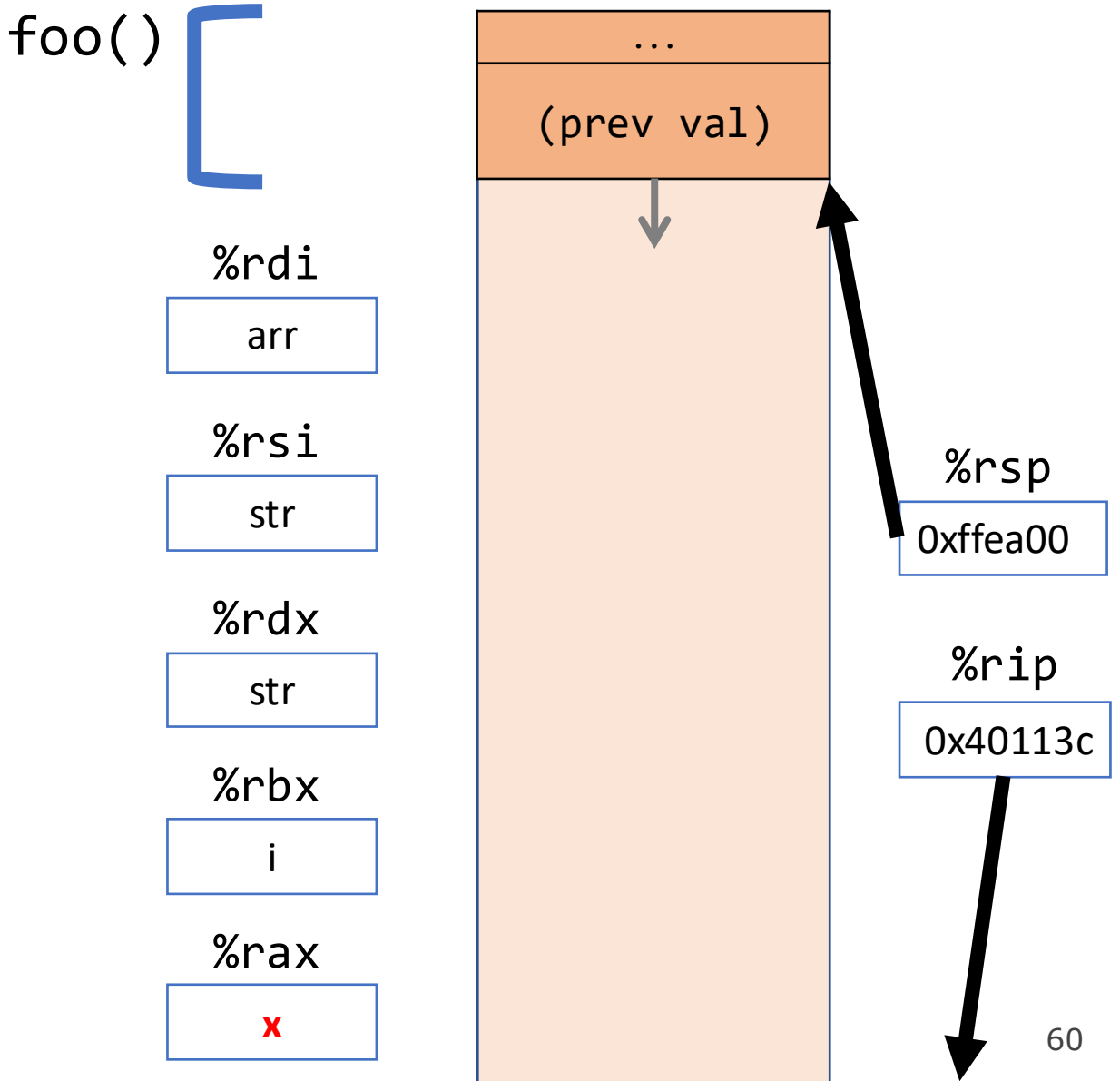


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

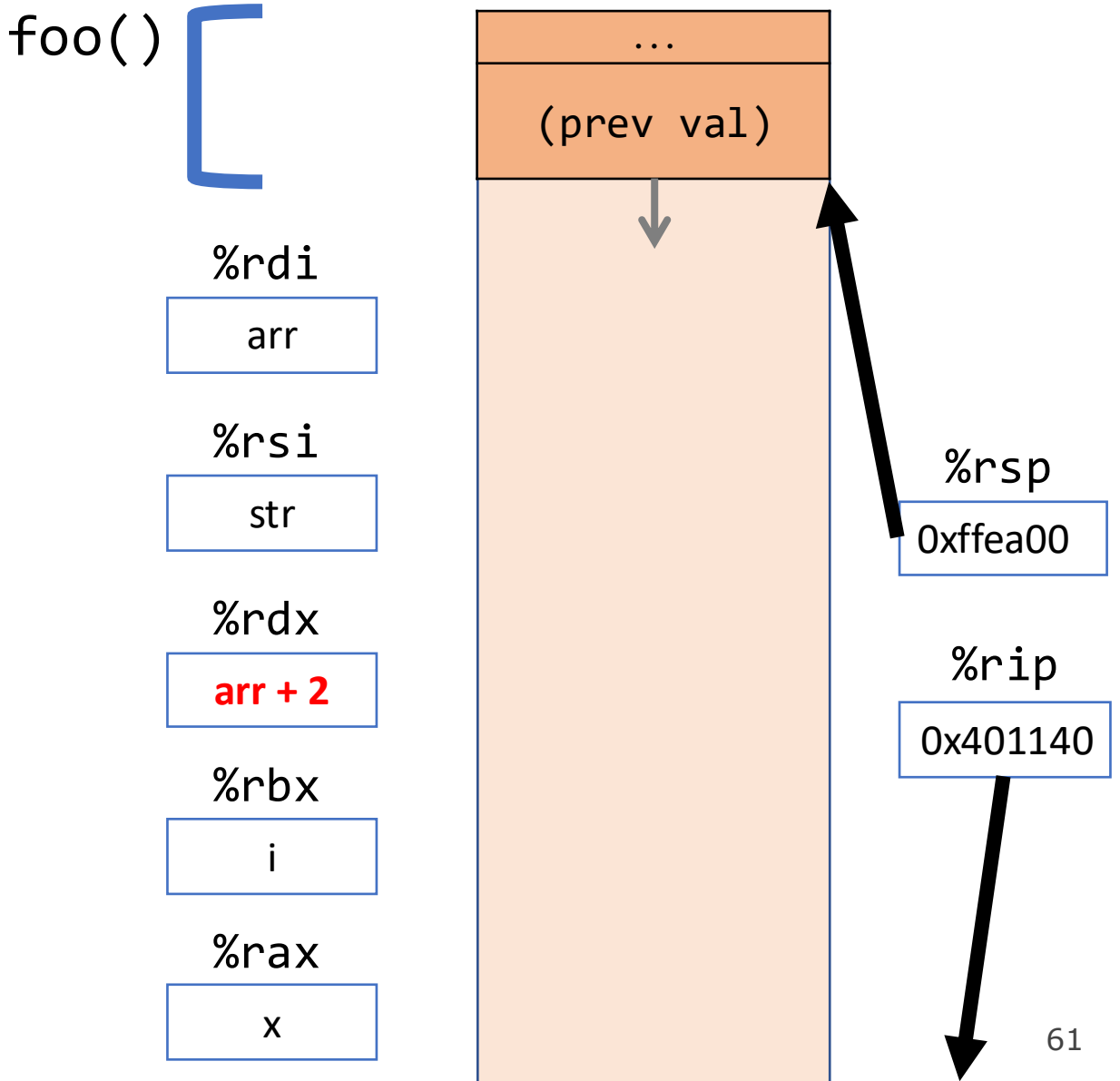


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

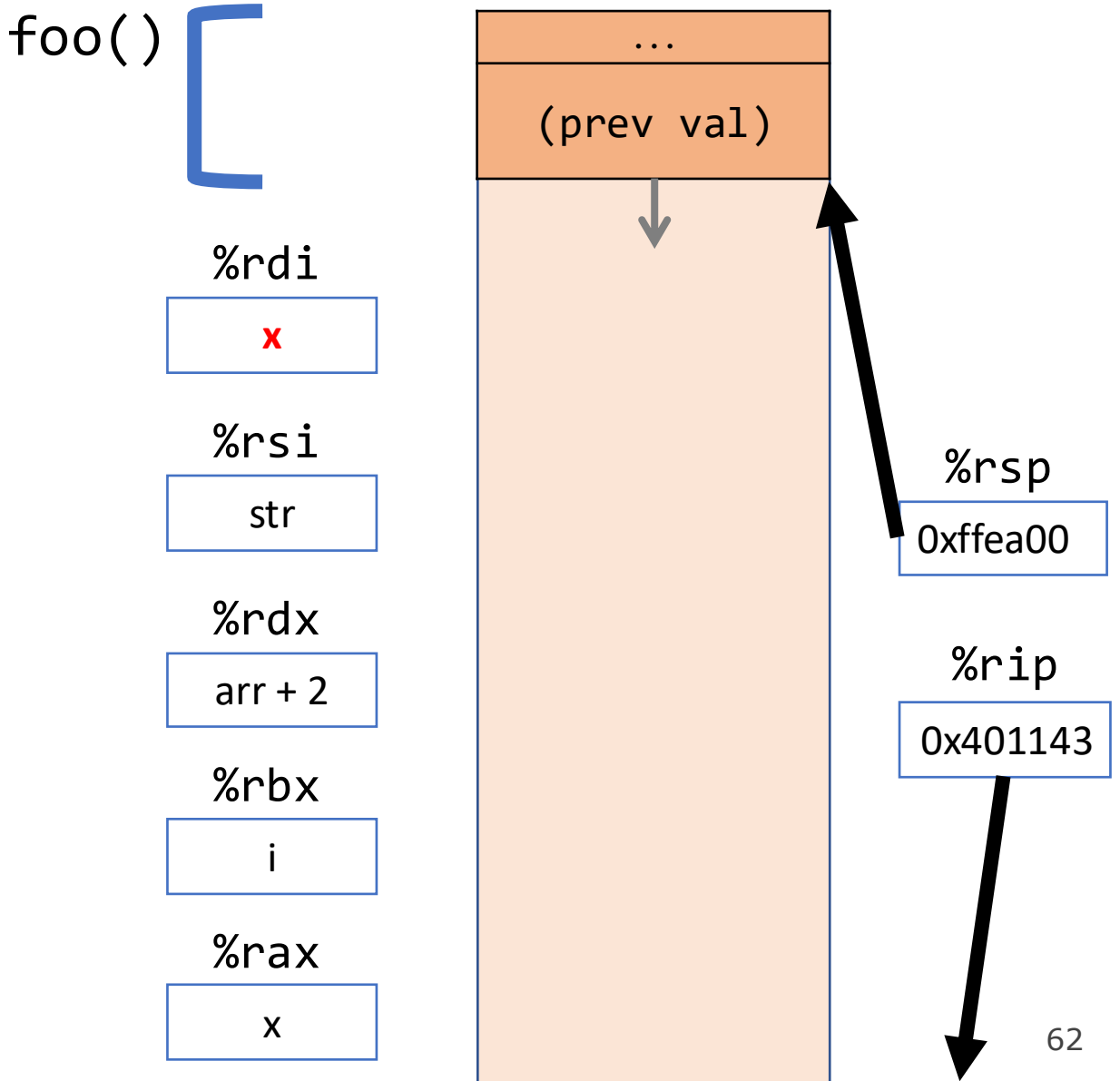


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

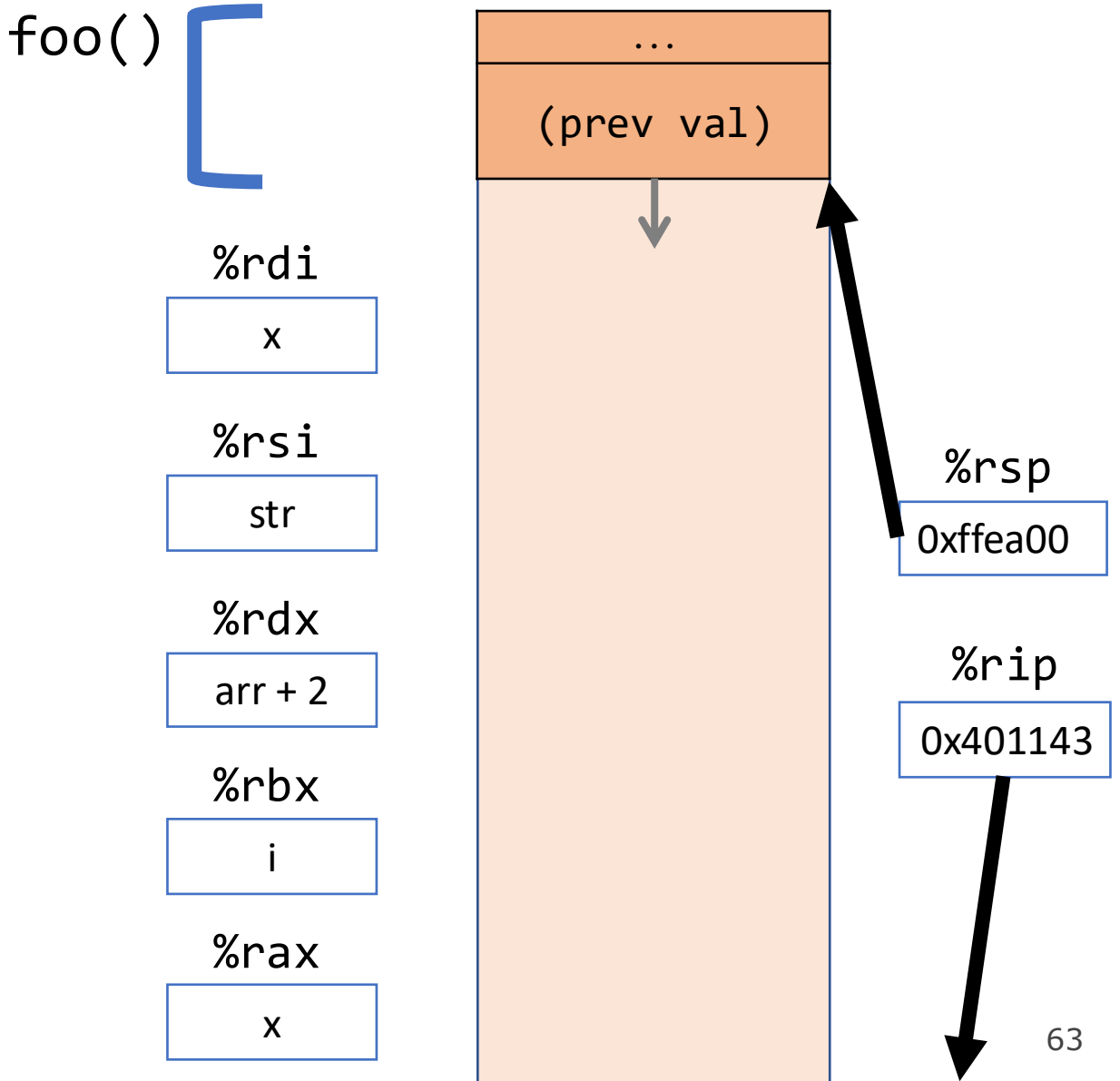


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

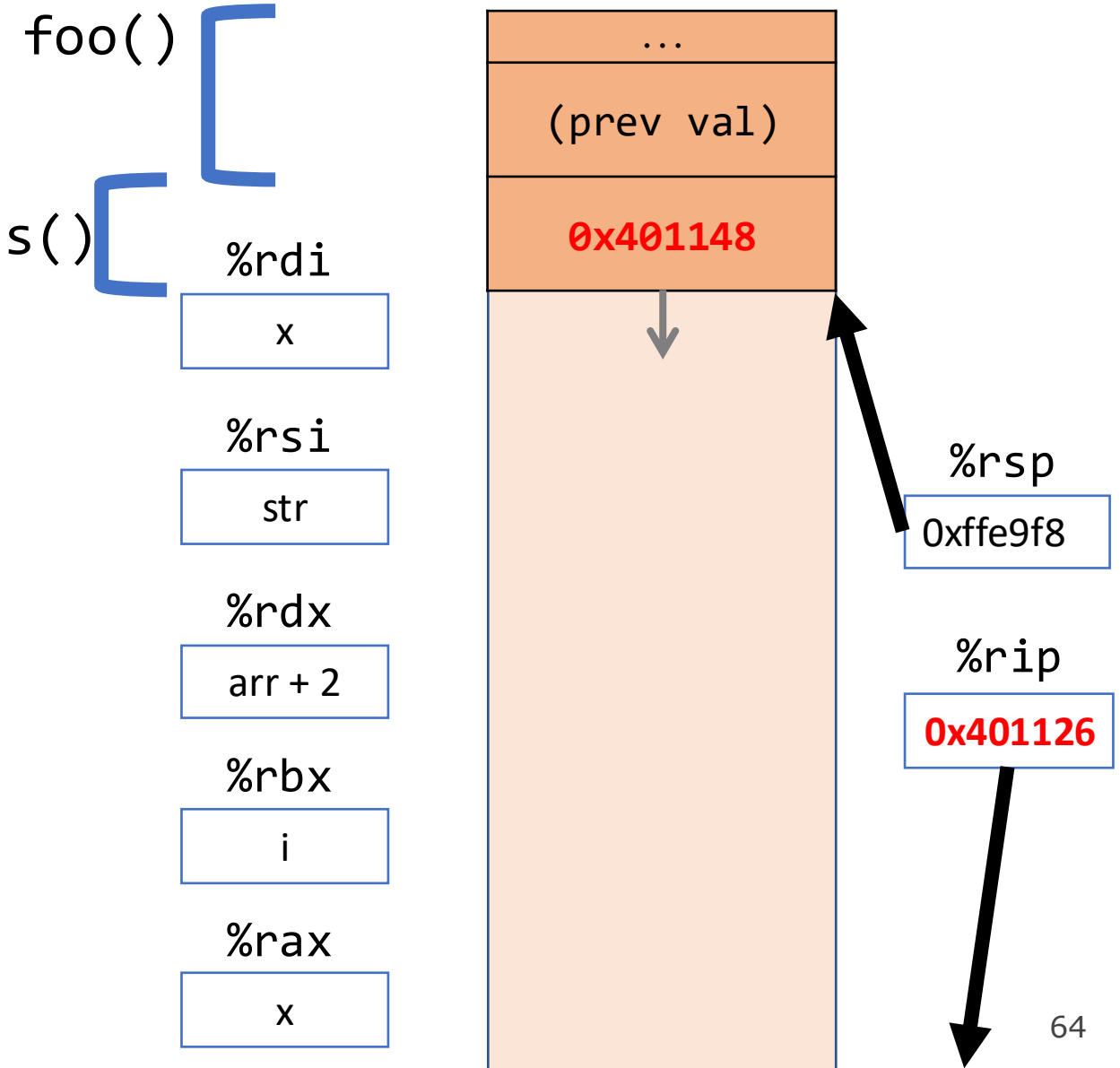


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

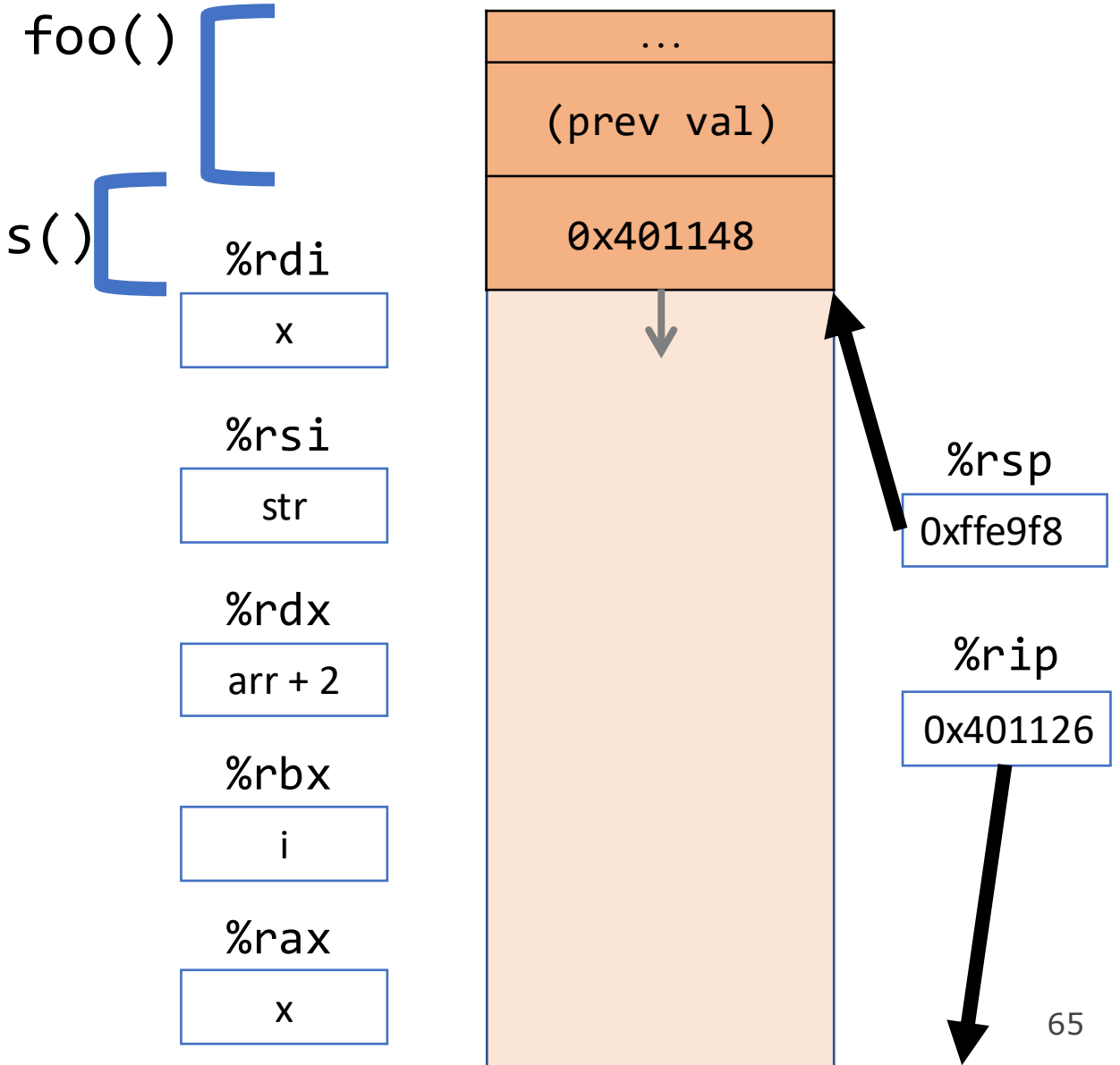


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```



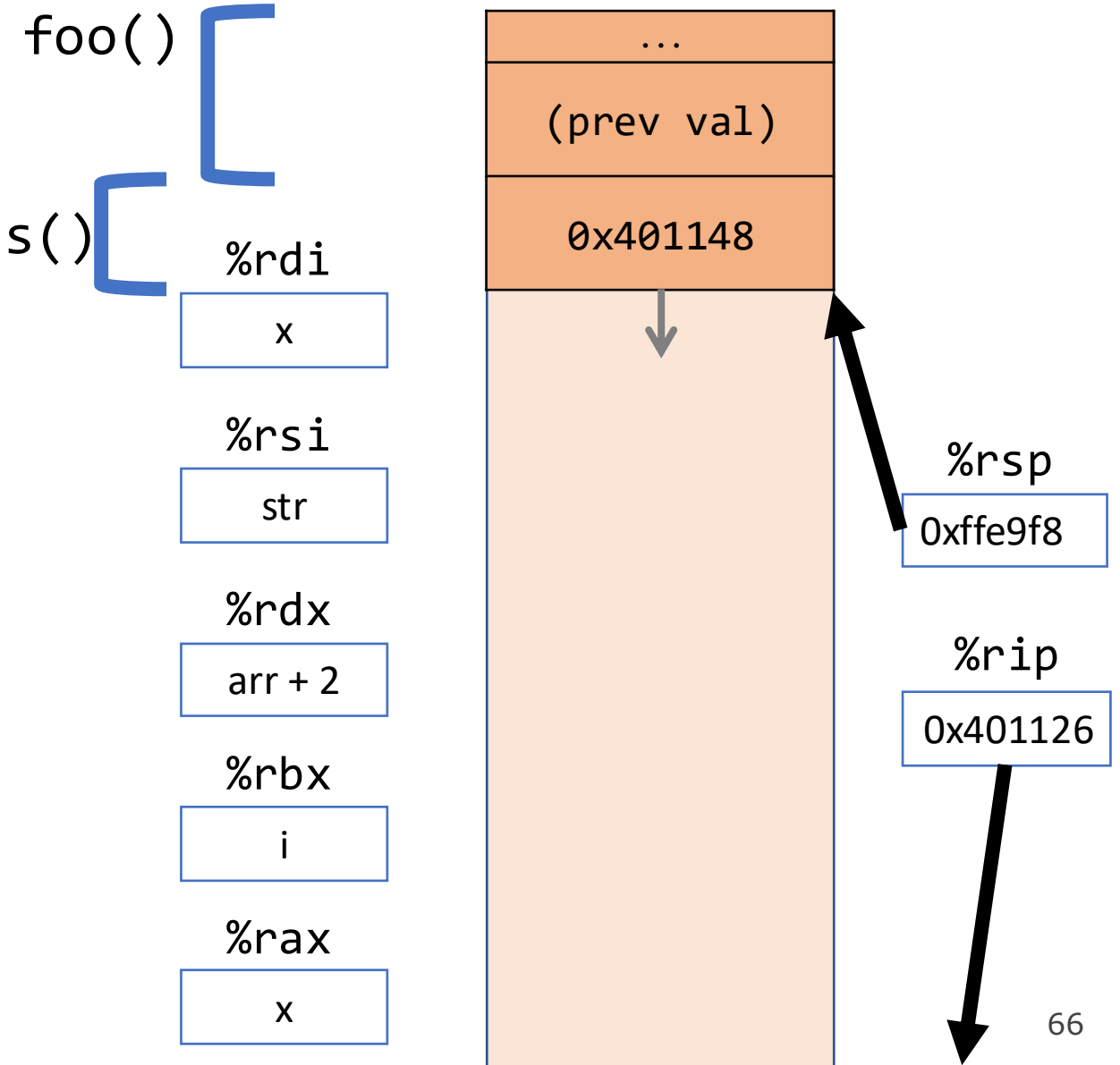
Parameters and Return

```
long s(long n, char *str, long *ptr) {
    return 2;
}

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    ...
```

```
0000000000401126 <s>:
  401126: mov    $0x2,%eax
  40112b: ret

0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  ...
```



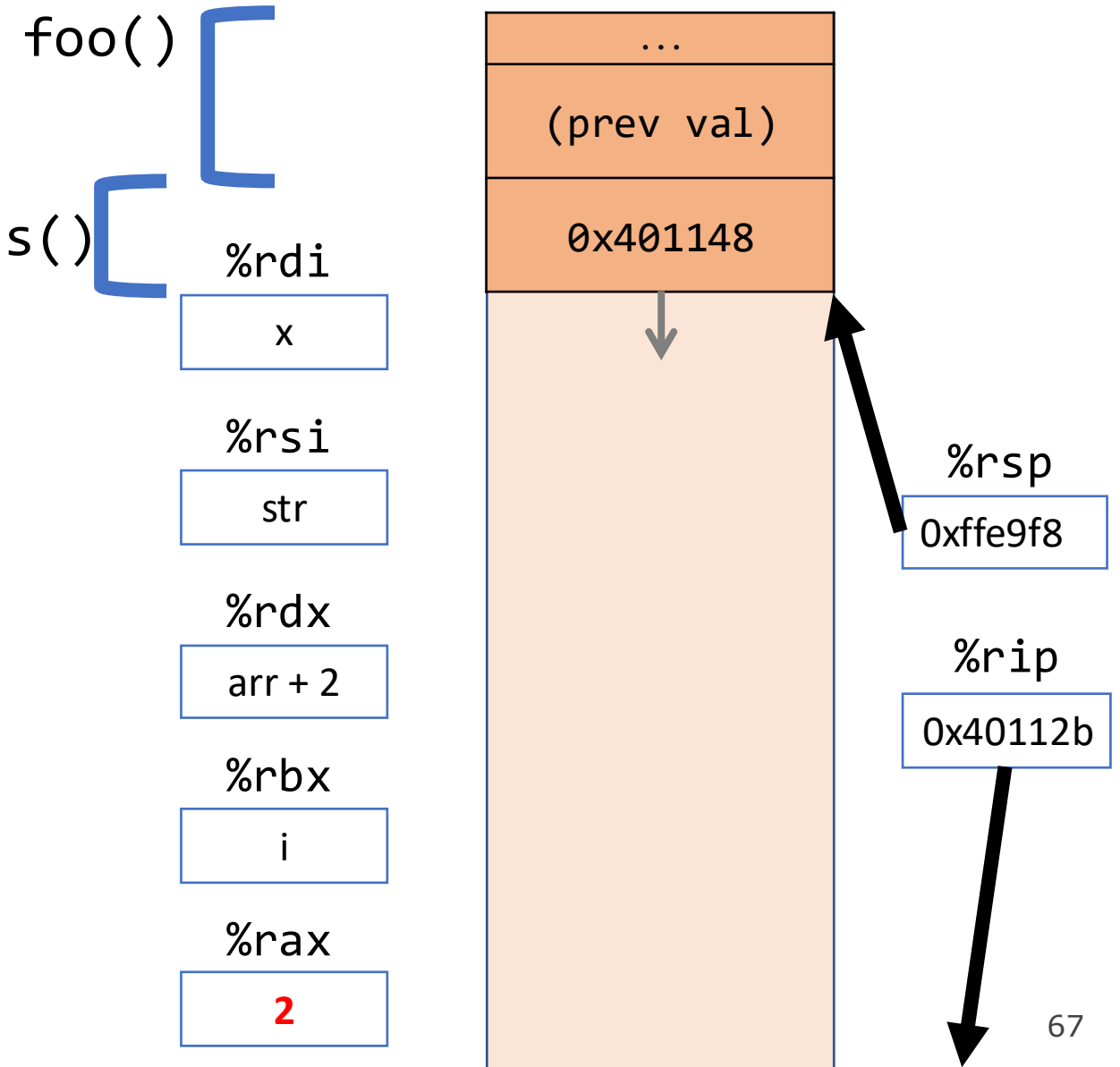
Parameters and Return

```
long s(long n, char *str, long *ptr) {
    return 2;
}

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    ...
```

```
0000000000401126 <s>:
  401126: mov    $0x2,%eax
  40112b: ret

0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  ...
```



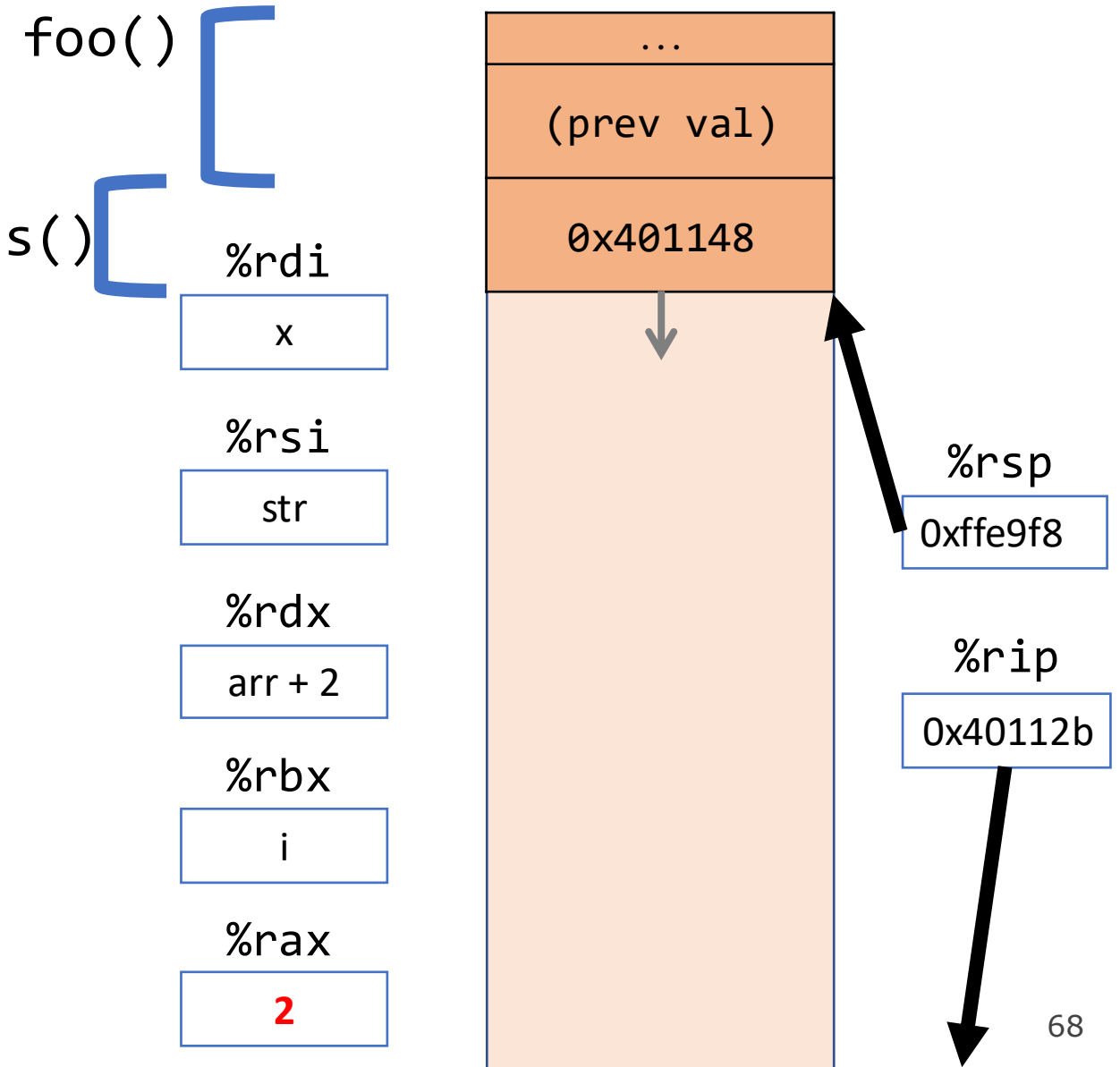
Parameters and Return

```
long s(long n, char *str, long *ptr) {
    return 2;
}

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    ...
```

```
0000000000401126 <s>:
  401126: mov    $0x2,%eax
  40112b: ret

0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  ...
```



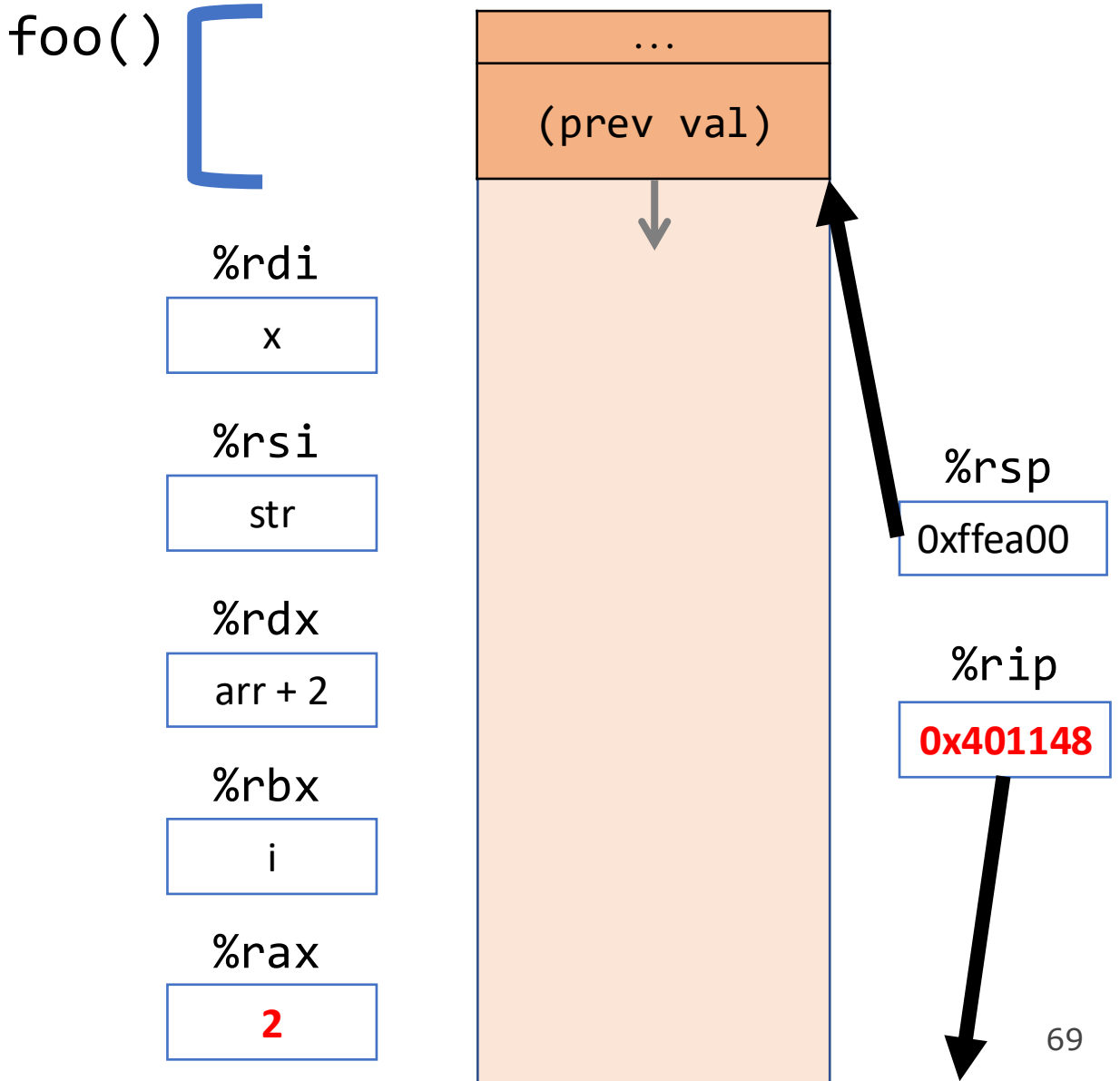
Parameters and Return

```
long s(long n, char *str, long *ptr) {
    return 2;
}

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    ...
```

```
0000000000401126 <s>:
  401126: mov    $0x2,%eax
  40112b: ret

0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  ...
```

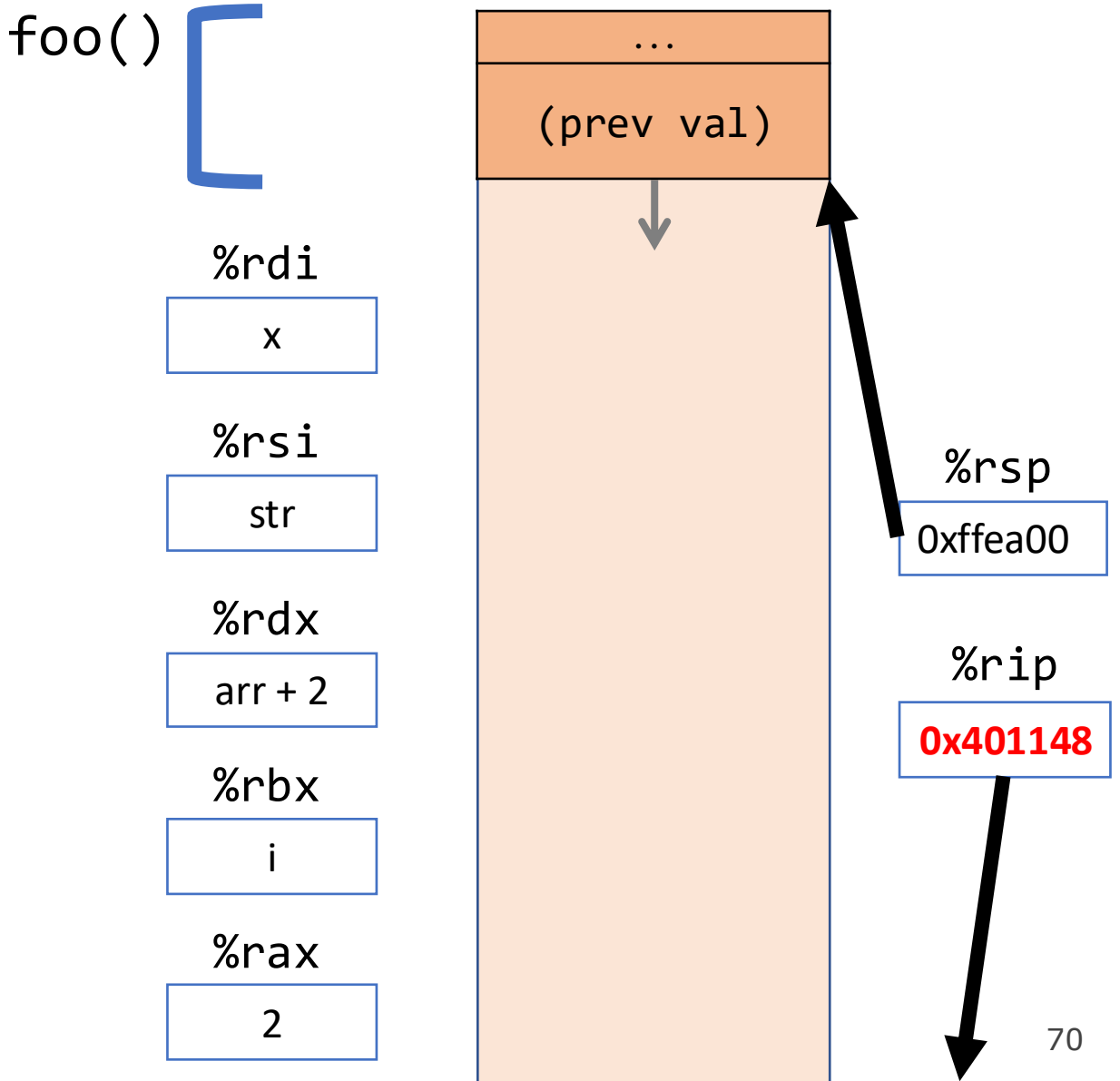


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

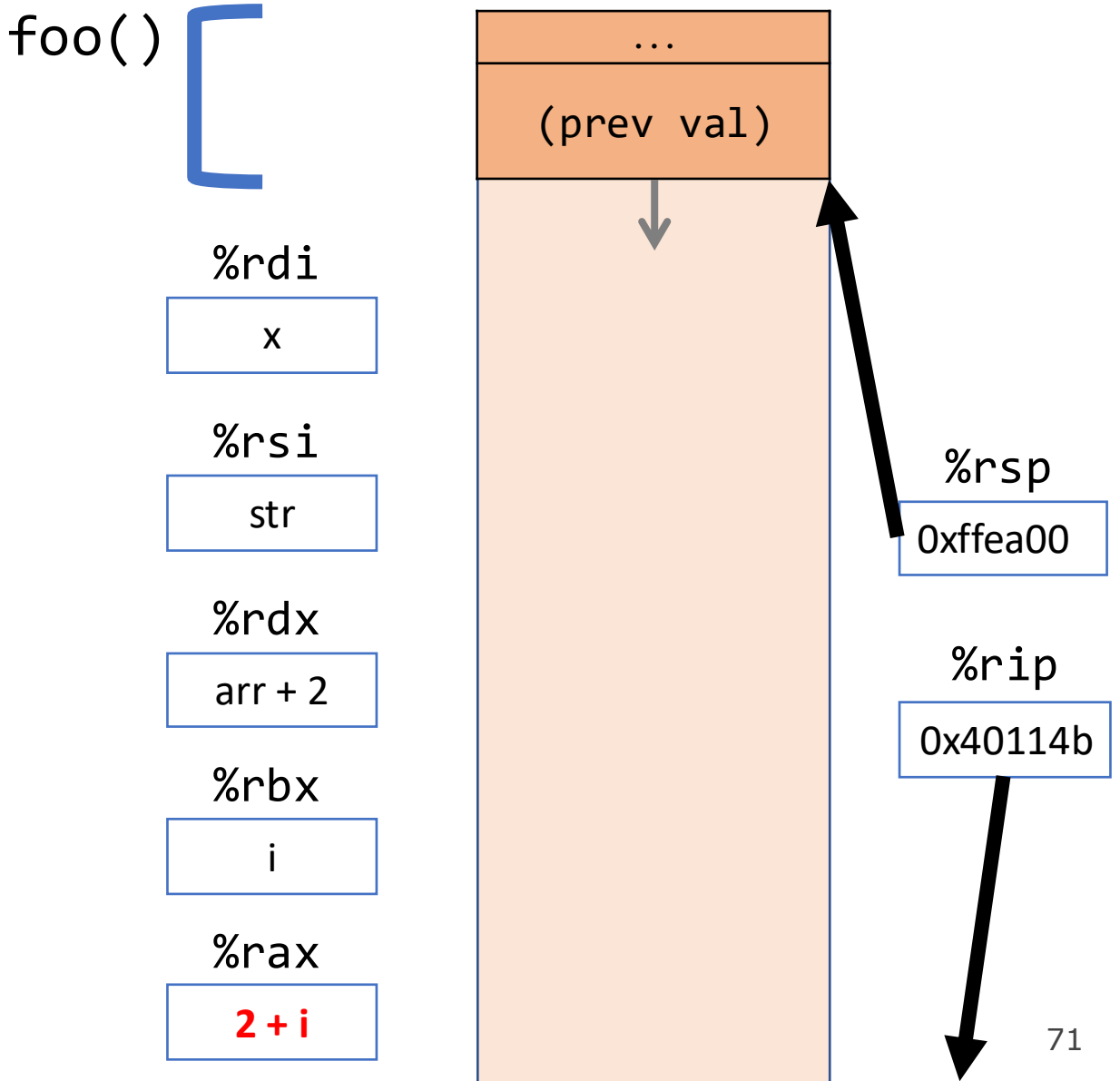


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

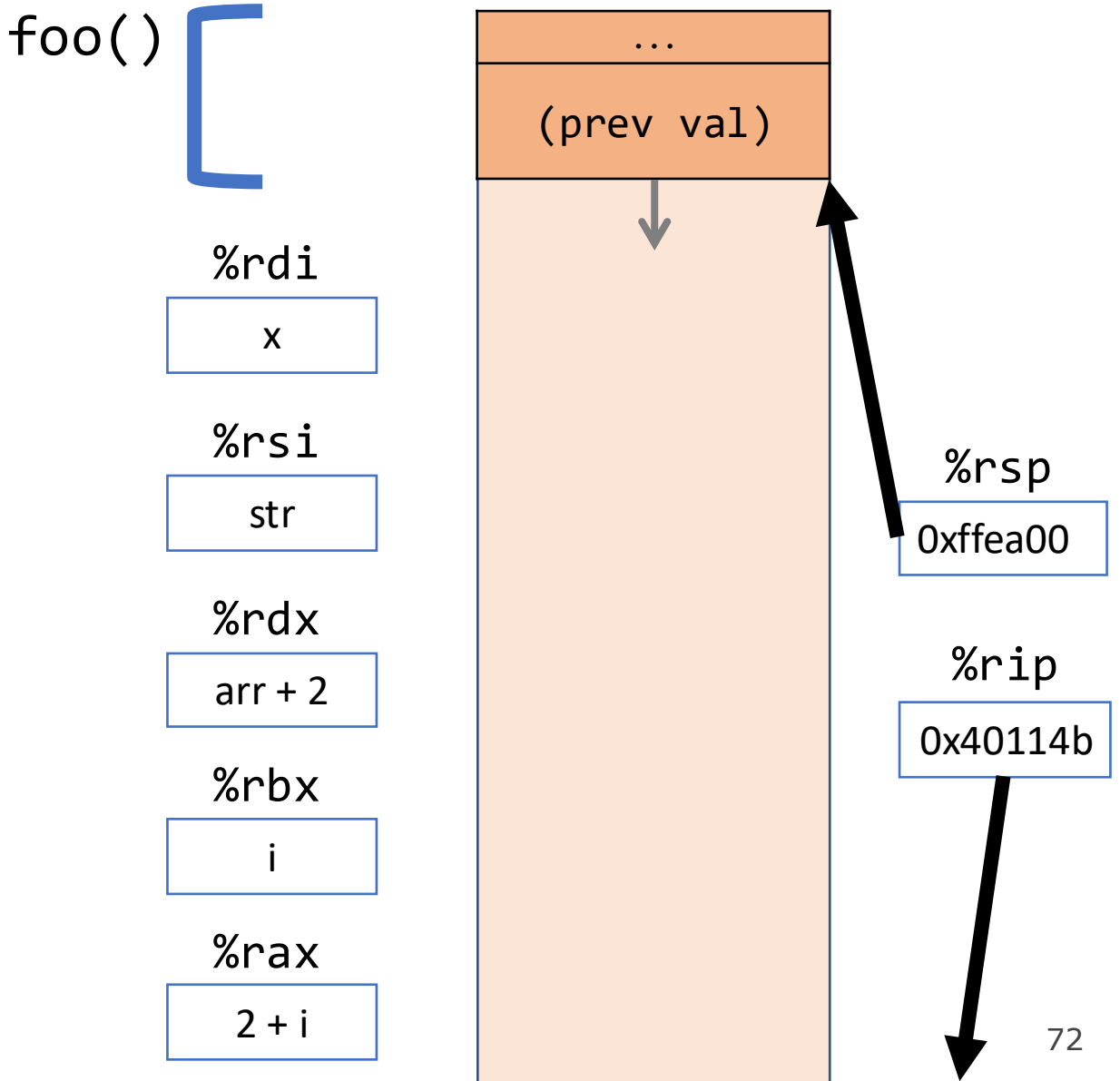


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

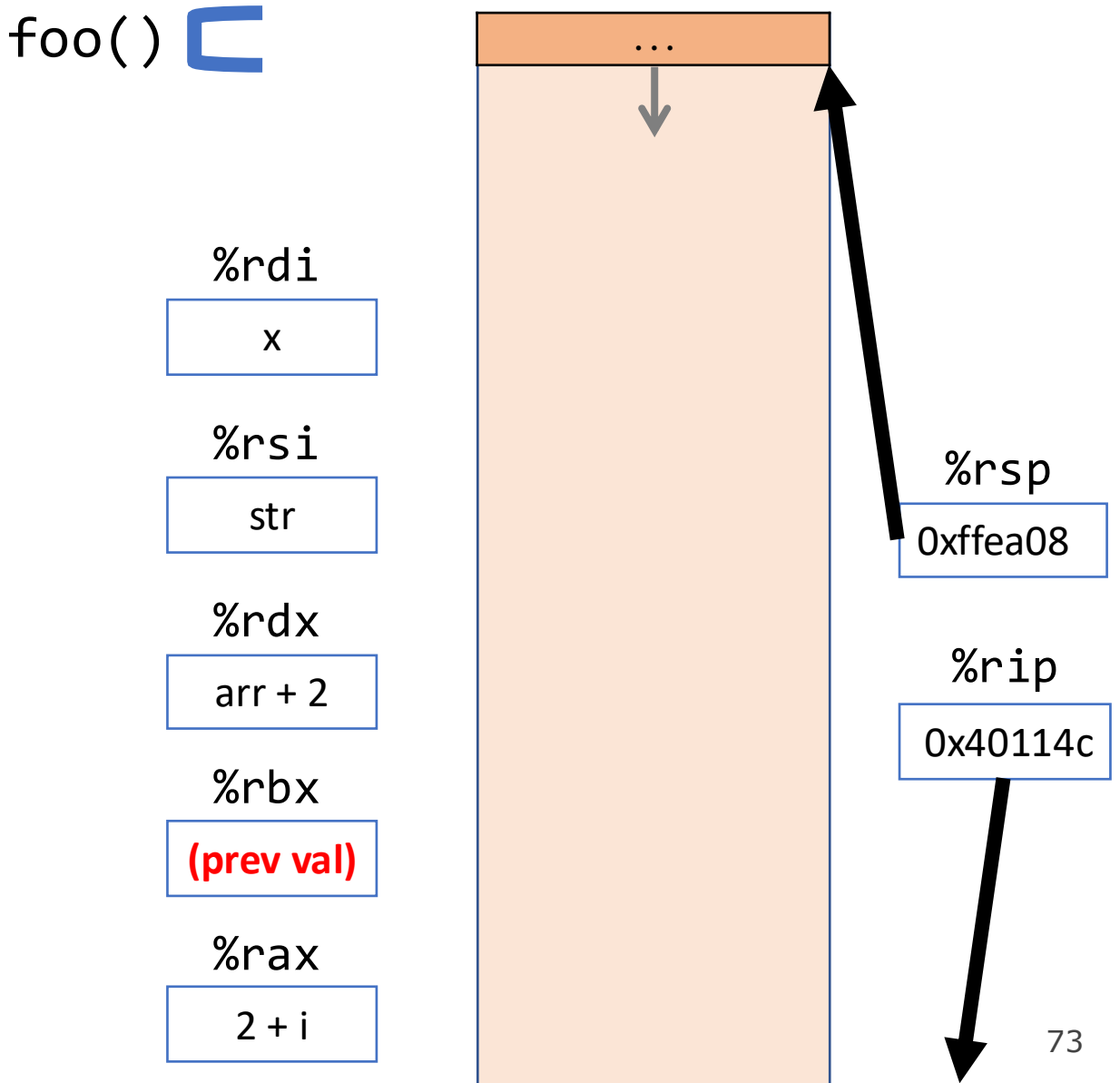


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

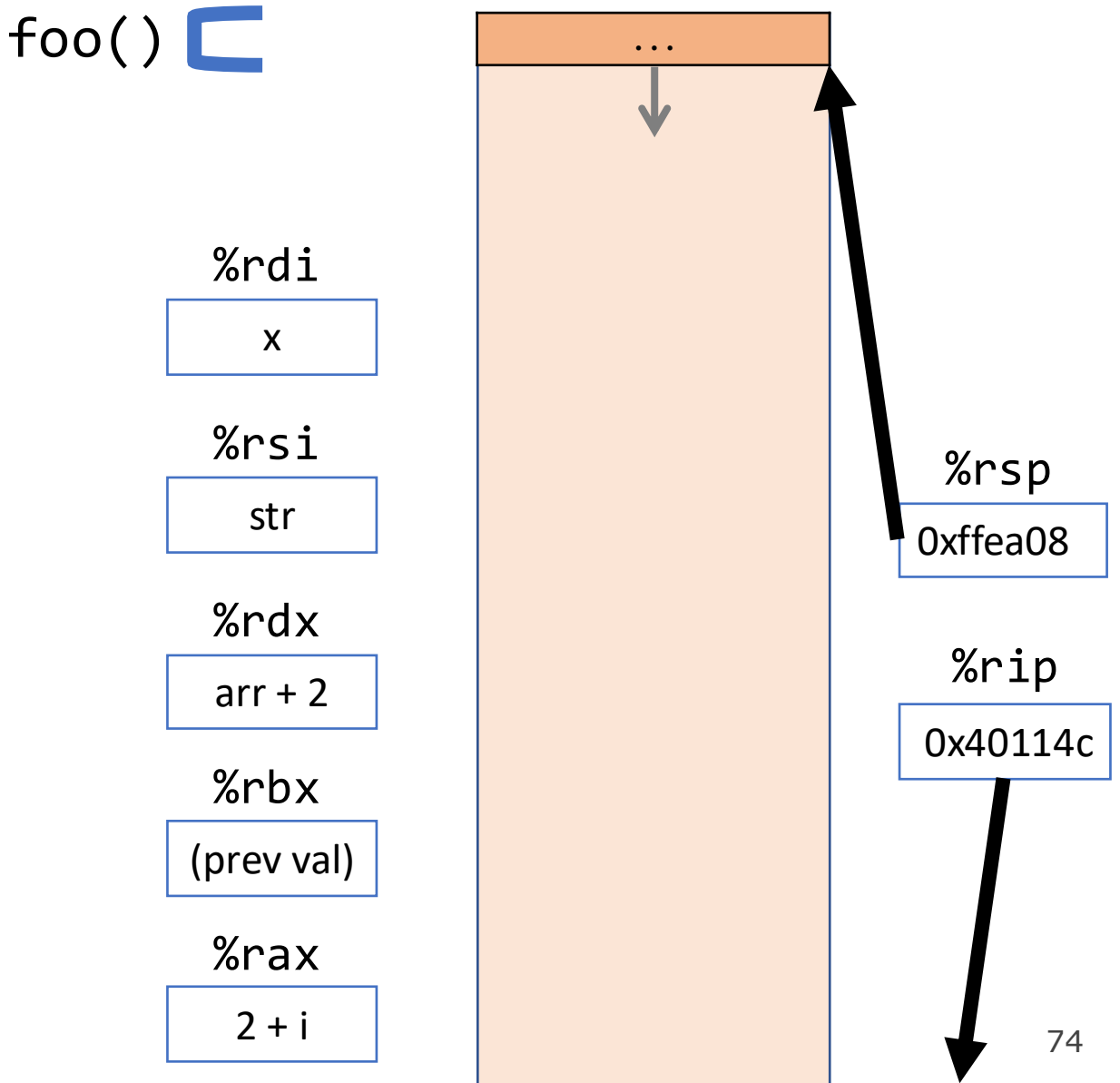


Parameters and Return

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long x = arr[i];
    long result = s(x, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```



Lecture Plan

- The Stack
- Calling Functions
  - Running another function's instructions
  - Parameters and return values
  - Trace: Calling a Function
- **Register Restrictions**

```
cp -r /afs/ir/class/cs107/lecture-code/lect20 .
```

Register Restrictions

There is only one copy of registers for all functions.

- **Problem:** what if ***funcA*** is building up a value in register %rbx, and calls ***funcB*** in the middle, which also has instructions that modify %rbx? ***funcA***'s value will be overwritten!

```
long func(long num) {
    return otherFunc(num) + num;
}

---

0000000000401131 <func>:
  401131: push   %rbx
  401132: mov    %rdi,%rbx
  401135: call   40112c <otherFunc>
  40113a: add    %rbx,%rax
  40113d: pop    %rbx
  40113e: ret
```

we use %rbx to store num here – but what if otherFunc also wants to use %rbx?

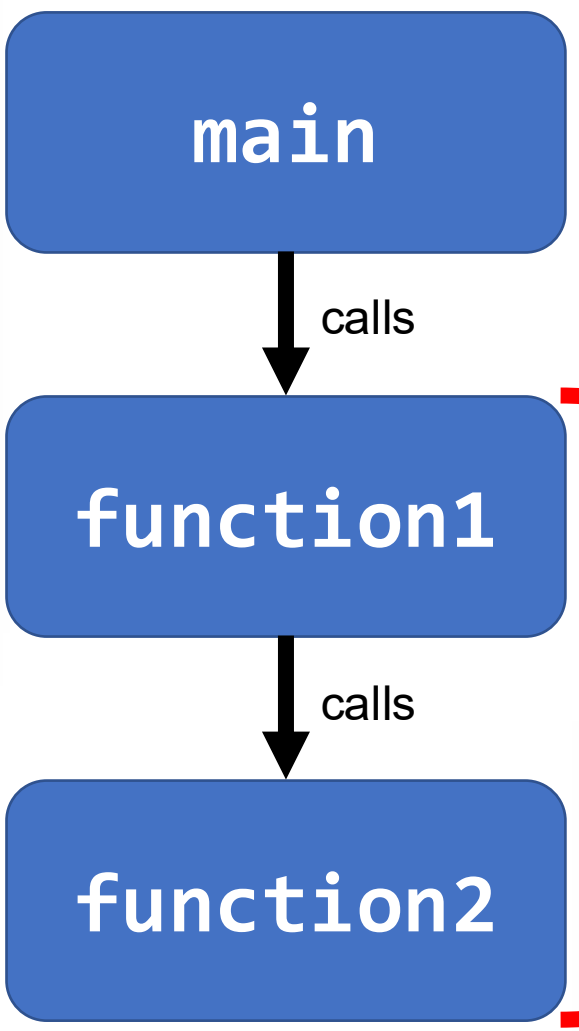
Register Restrictions

There is only one copy of registers for all functions.

- **Problem:** what if ***funcA*** is building up a value in register %rbx, and calls ***funcB*** in the middle, which also has instructions that modify %rbx? ***funcA***'s value will be overwritten!
- **Solution:** make some "rules of the road" that callers and callees must follow when using registers so they do not interfere with one another.
- These rules define two types of registers: **caller-owned** and **callee-owned**

Caller/Callee

Caller/callee is terminology that refers to a pair of functions. A single function may be both a caller and callee simultaneously (e.g. `function1` at right).



`main` is the caller, and `function1` is the callee.

`function1` is the caller, and `function2` is the callee.

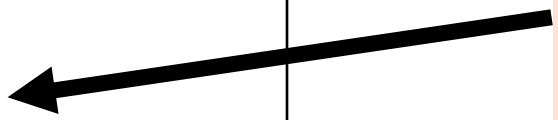
Caller-Owned Registers

A caller-owned register is a register where, if we call another function, the register is guaranteed to **still store the same value** after that function finishes.

```
long func(long num) {
    return otherFunc(num) + num;
}

---

0000000000401131 <func>:
  401131: push   %rbx
  401132: mov    %rdi,%rbx
  401135: call   40112c <otherFunc>
  40113a: add    %rbx,%rax
  40113d: pop    %rbx
  40113e: ret
```



%rbx is one of the caller-owned registers – it is guaranteed to still store our value even if we call otherFunc.

Caller-Owned Registers

A caller-owned register is a register where, if we call another function, the register is guaranteed to **still store the same value** after that function finishes.

However, this means that if *we* want to use a caller-owned register, *we* must restore it back to its original value when we are done.

```
long func(long num) {
    return otherFunc(num) + num;
}

---

0000000000401131 <func>:
  401131: push   %rbx
  401132: mov    %rdi,%rbx
  401135: call   40112c <otherFunc>
  40113a: add    %rbx,%rax
  40113d: pop    %rbx
  40113e: ret
```

%rbx is one of the caller-owned registers – therefore, if *we* want to use it, we must store its old value and restore it so that the function that called *us* has its value preserved too.

Callee-Owned Registers

A callee-owned register is a register where, if we call another function, the register's value **may be changed by that function**, and thus we may lose our value from before the function call.

Examples of callee-owned registers: all parameter registers (%rdi, %rsi), return register (%rax). All of these could be changed by calling another function!

However, this means that if *we* want to use a callee-owned register, there's nothing we need to do first – we do not need to restore it back to its original value, so we can just overwrite what's already there.

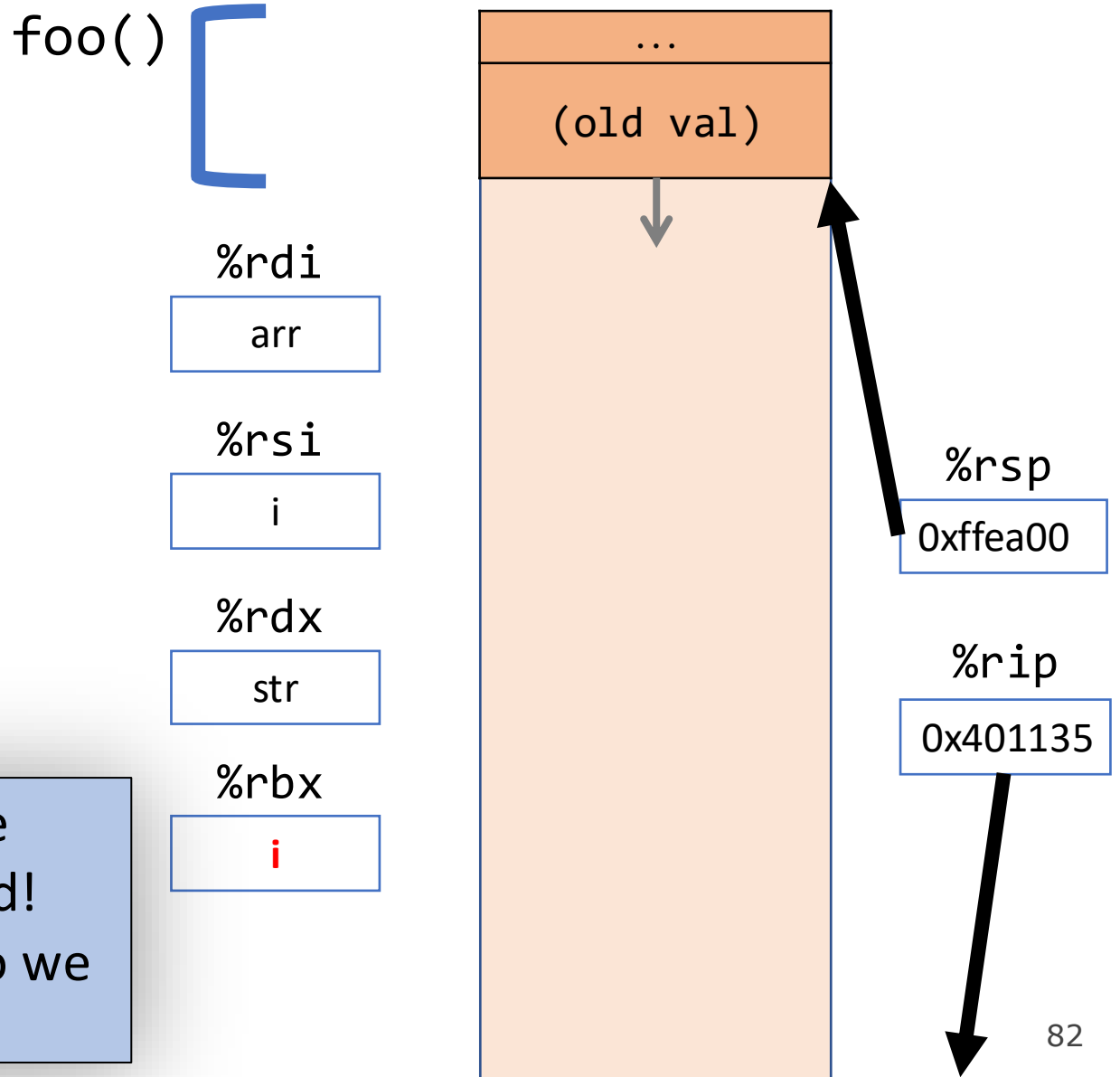
Callee-Owned and Caller-Owned

```
long s(long n, char *str, long *ptr);

long foo(long arr[], long i, char *str) {
    long val = arr[i];
    long result = s(val, str, arr + 2);
    return result + i;
}
```

```
0000000000401131 <foo>:
  401131: push   %rbx
  401132: mov    %rsi,%rbx
  401135: mov    %rdx,%rsi
  401138: mov    (%rdi,%rbx,8),%rax
  40113c: lea    0x10(%rdi),%rdx
  401140: mov    %rax,%rdi
  401143: call   401126 <s>
  401148: add    %rbx,%rax
  40114b: pop    %rbx
  40114c: ret
```

Calling s can cause %rsi to be changed! We need i later, so we save it elsewhere.



Register Restrictions

Caller-Owned

- Callee must *save* the existing value and *restore* it when done.
- Caller can store values and assume they will be preserved across function calls.

Callee-Owned

- Callee does not need to save the existing value.
- Caller's values could be overwritten by a callee! The caller may consider saving values elsewhere before calling functions.

Our First Assembly

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++) {
        sum += arr[i];
    }
    return sum;
}
```

We're done with all our assembly lectures! Now we can fully understand what's going on in the assembly below, including how someone would call sum_array in assembly and what the **ret** instruction does.

```
0000000000401136 <sum_array>:
  401136 <+0>:     mov    $0x0,%eax
  40113b <+5>:     mov    $0x0,%edx
  401140 <+10>:    jmp    0x40114b <sum_array+21>
  401142 <+12>:    movslq %eax,%rcx
  401145 <+15>:    add    (%rdi,%rcx,4),%edx
  401148 <+18>:    add    $0x1,%eax
  40114b <+21>:    cmp    %esi,%eax
  40114d <+23>:    jl     0x401142 <sum_array+12>
  40114f <+25>:    mov    %edx,%eax
  401151 <+27>:    ret
```

Recap

- The Stack
- Calling Functions
  - Running another function's instructions
  - Parameters and return values
  - Trace: Calling a Function
- Register Restrictions

Lecture 20 takeaway: Function calls rely on the special %rip and %rsp registers to execute another function's instructions and make stack space. We rely on special registers to pass parameters and the return value between functions. There are caller and callee owned registers to manage use across functions.

Extra Practice

Extra Practice – Escape Room

https://godbolt.org/z/P49cxfvWK

`escape_room`

Escape room assembly code

```
0000000000401148 <escape_room>:
  401148: 48 83 ec 08           sub    $0x8,%rsp
  40114c: ba 0a 00 00 00        mov    $0xa,%edx
  401151: be 00 00 00 00        mov    $0x0,%esi
  401156: e8 e5 fe ff ff        call   401040 <strtol@plt>
  40115b: 89 c7                 mov    %eax,%edi
  40115d: e8 d4 ff ff ff        call   401136 <transform>
  401162: a8 01                 test   $0x1,%al
  401164: 74 0a                 je     401170 <escape_room+0x28>
  401166: b8 00 00 00 00        mov    $0x0,%eax
  40116b: 48 83 c4 08           add    $0x8,%rsp
  40116f: c3                    ret
  401170: b8 01 00 00 00        mov    $0x1,%eax
  401175: eb f4                 jmp    40116b <escape_room+0x23>
```

Escape room assembly code

```
0000000000401136 <transform>:
  401136:    8d 04 bd 00 00 00 00     lea    0x0(,%rdi,4),%eax
  40113d:    8d 50 01                 lea    0x1(%rax),%edx
  401140:    83 fa 32                 cmp    $0x32,%edx
  401143:    7f 02                    jg     401147 <transform+0x11>
  401145:    89 d0                    mov    %edx,%eax
  401147:    c3                       ret
```