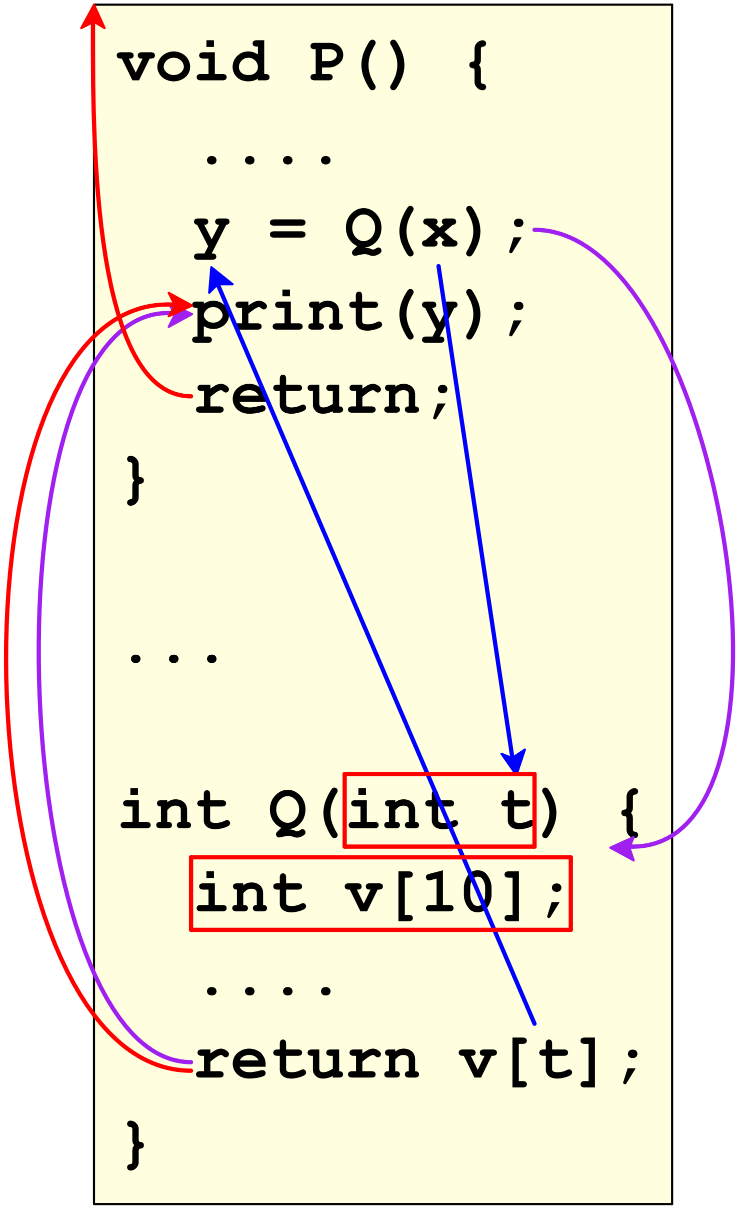
Procedures



```
void P() {
  ....
  y = Q(x);
  print(y);
  return;
}

...

int Q(int t) {
  int v[10];
  ....
  return v[t];
}
```

Passing control

- to called procedure
- back to caller

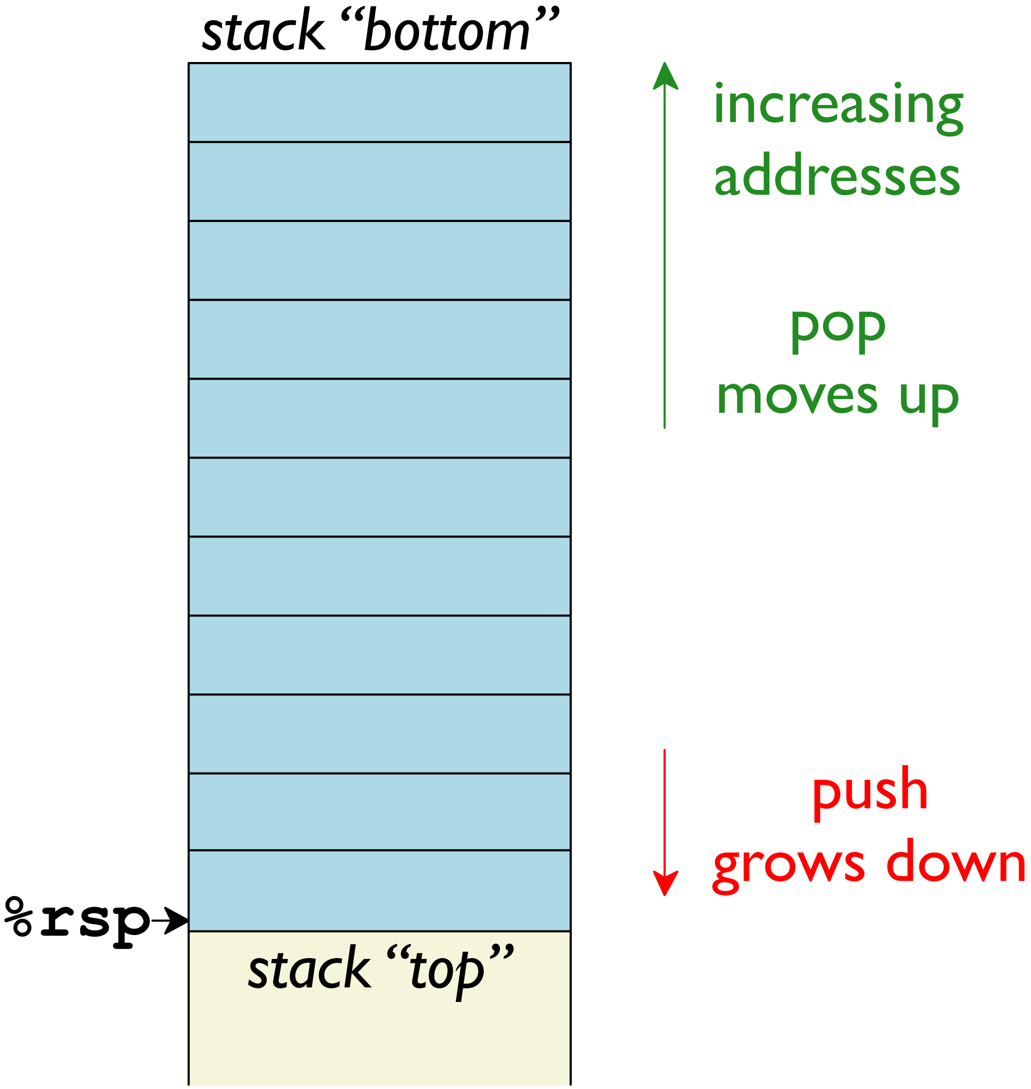
Passing data

- procedure arguments
- procedure result

Memory allocation

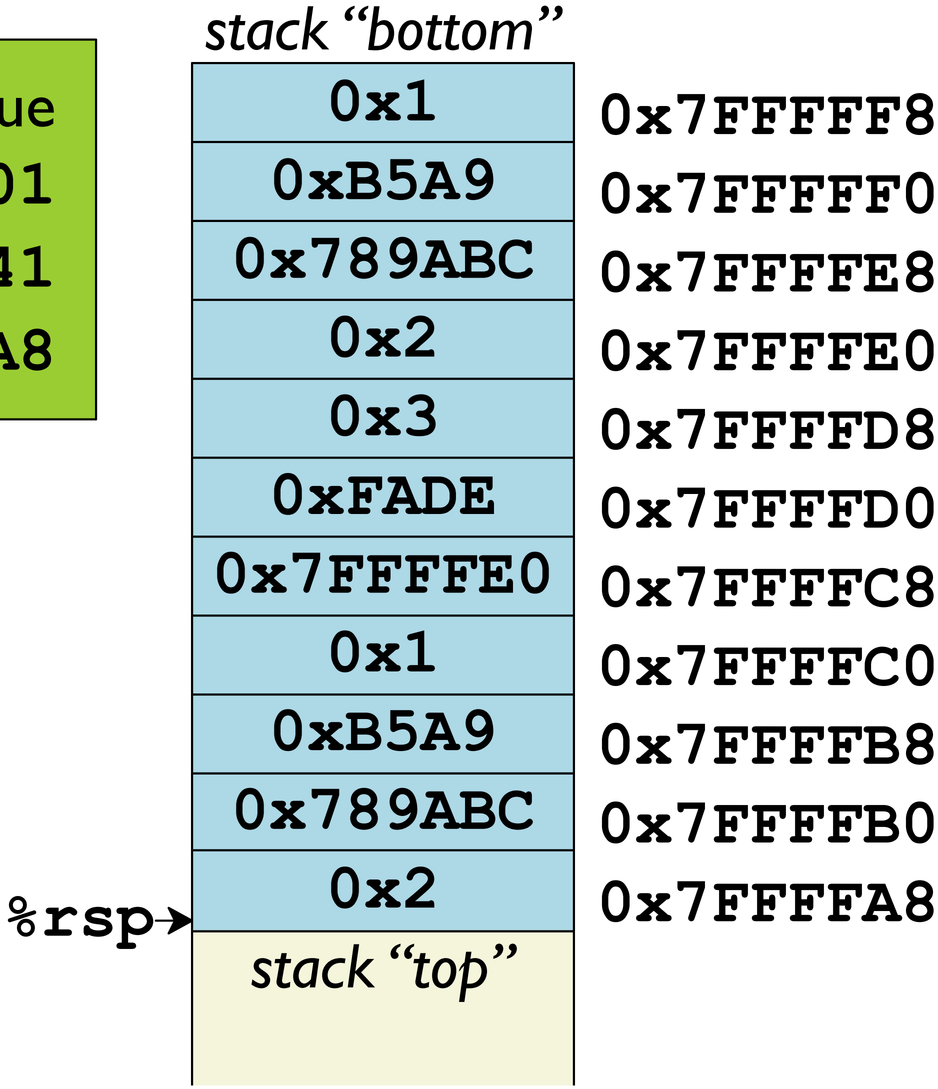
- local variables
- continuation

C Stack


stack "bottom"
increasing addresses
pop moves up
push grows down
%rsp
stack "top"

C Stack Operations

register	value
`%rax`	`0x101`
`%rbx`	`0x41`
`%rsp`	`0x7FFFFA8`



stack "bottom"

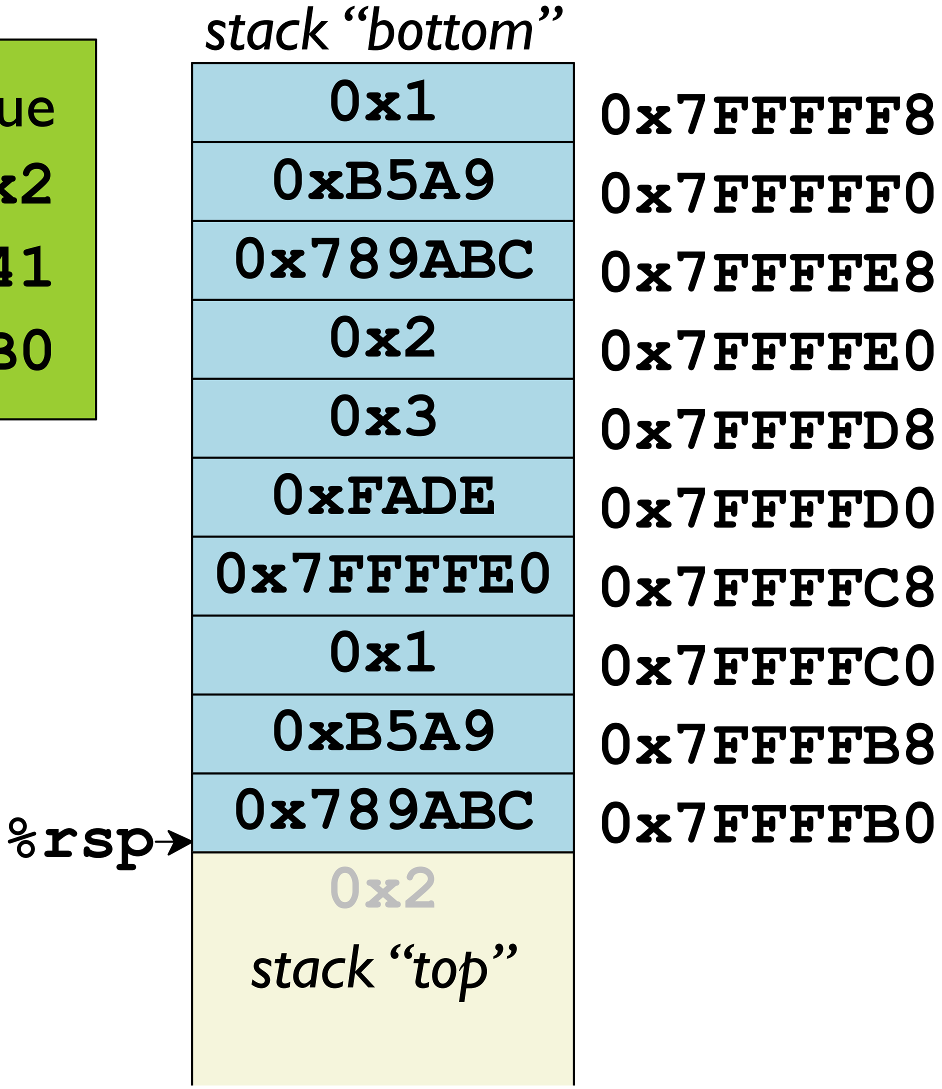
`0x1`	`0x7FFFFF8`
`0xB5A9`	`0x7FFFFF0`
`0x789ABC`	`0x7FFFFE8`
`0x2`	`0x7FFFFE0`
`0x3`	`0x7FFFFD8`
`0xFADE`	`0x7FFFFD0`
`0x7FFFFE0`	`0x7FFFFC8`
`0x1`	`0x7FFFFC0`
`0xB5A9`	`0x7FFFFB8`
`0x789ABC`	`0x7FFFFB0`
`0x2`	`0x7FFFFA8`

stack "top"

C Stack Operations

register	value
`%rax`	`0x2`
`%rbx`	`0x41`
`%rsp`	`0x7FFFFB0`

```
popq %rax
```



stack "bottom"

`0x1`	`0x7FFFFF8`
`0xB5A9`	`0x7FFFFF0`
`0x789ABC`	`0x7FFFFE8`
`0x2`	`0x7FFFFE0`
`0x3`	`0x7FFFFD8`
`0xFADE`	`0x7FFFFD0`
`0x7FFFFE0`	`0x7FFFFC8`
`0x1`	`0x7FFFFC0`
`0xB5A9`	`0x7FFFFB8`
`0x789ABC`	`0x7FFFFB0`
`0x2`	

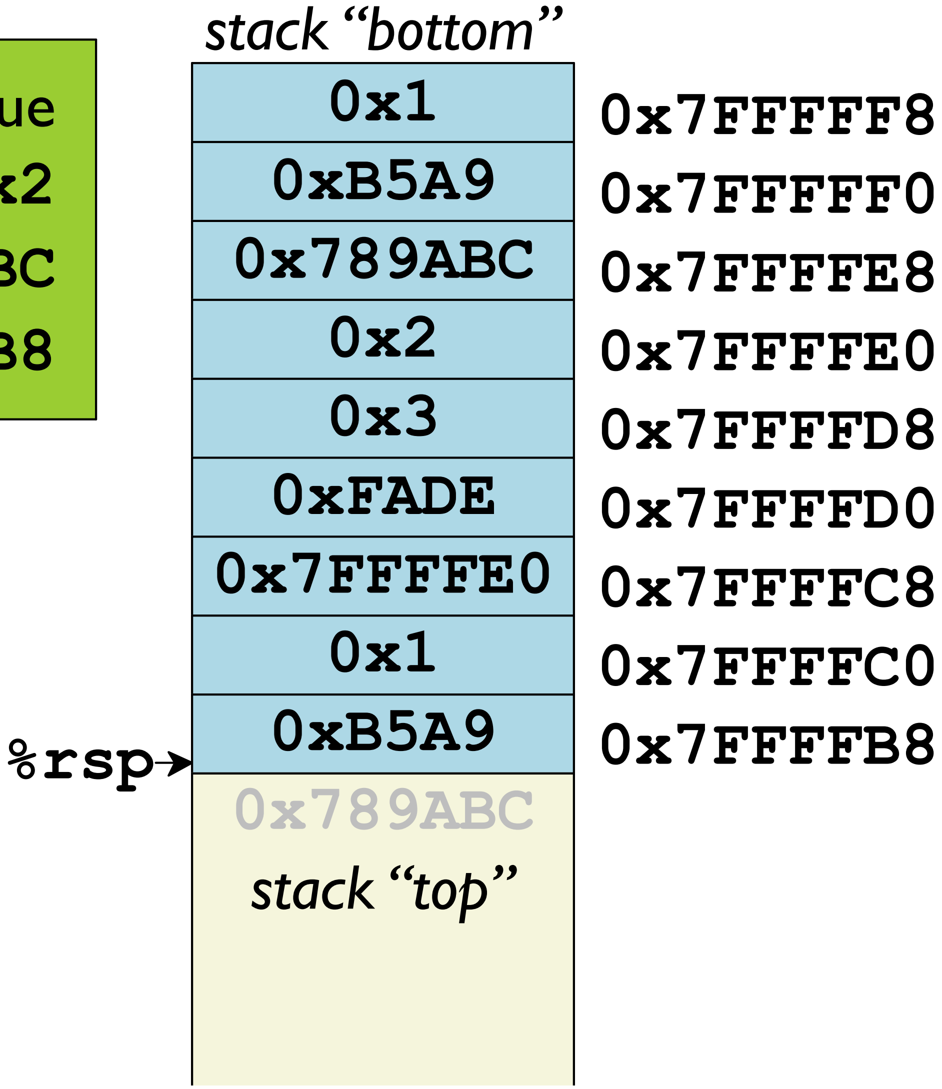
`%rsp` →

stack "top"

C Stack Operations

register	value
`%rax`	`0x2`
`%rbx`	`0x789ABC`
`%rsp`	`0x7FFFFB8`

```
popq %rax
popq %rbx
```



stack "bottom"

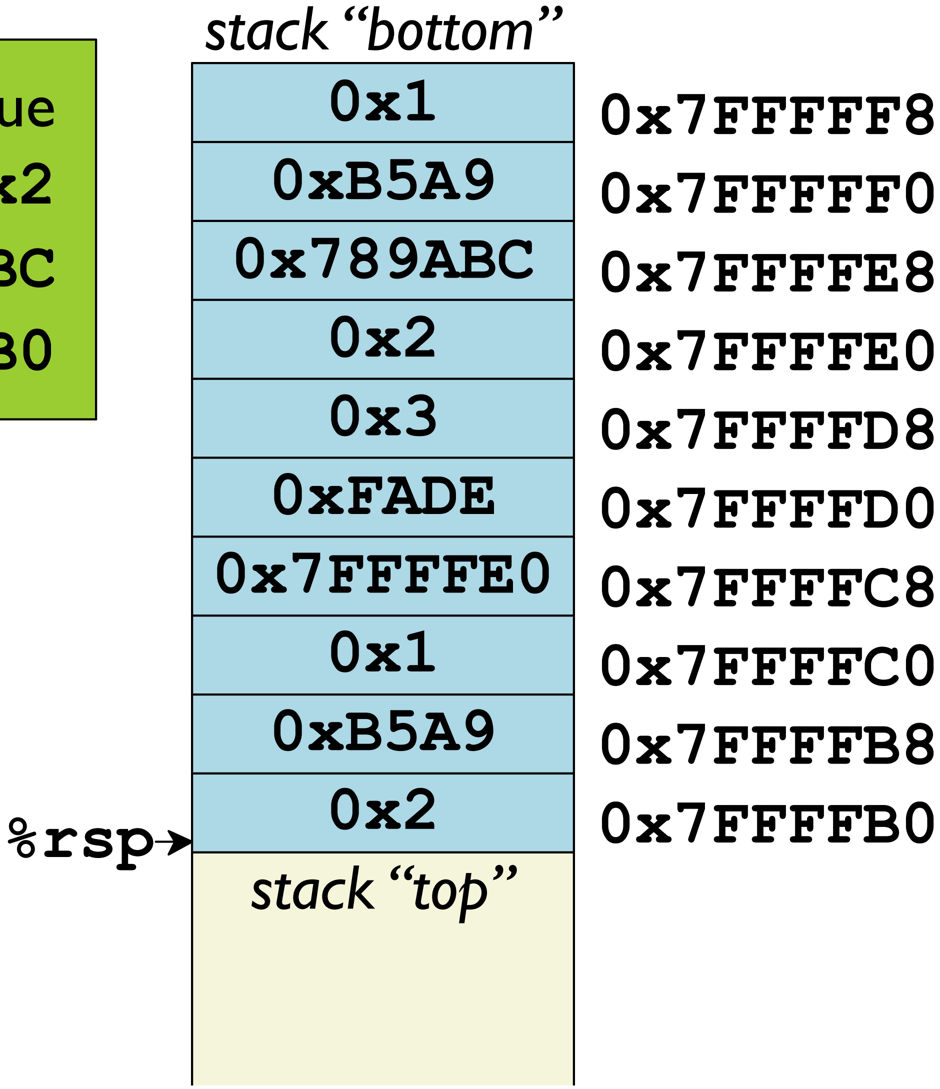
`0x1`	`0x7FFFFF8`
`0xB5A9`	`0x7FFFFF0`
`0x789ABC`	`0x7FFFFE8`
`0x2`	`0x7FFFFE0`
`0x3`	`0x7FFFFD8`
`0xFADE`	`0x7FFFFD0`
`0x7FFFFE0`	`0x7FFFFC8`
`0x1`	`0x7FFFFC0`
`0xB5A9`	`0x7FFFFB8`
`0x789ABC`	

stack "top"

C Stack Operations

register	value
%rax	0x2
%rbx	0x789ABC
%rsp	0x7FFFFB0

```
popq %rax
popq %rbx
pushq %rax
```



stack "bottom"

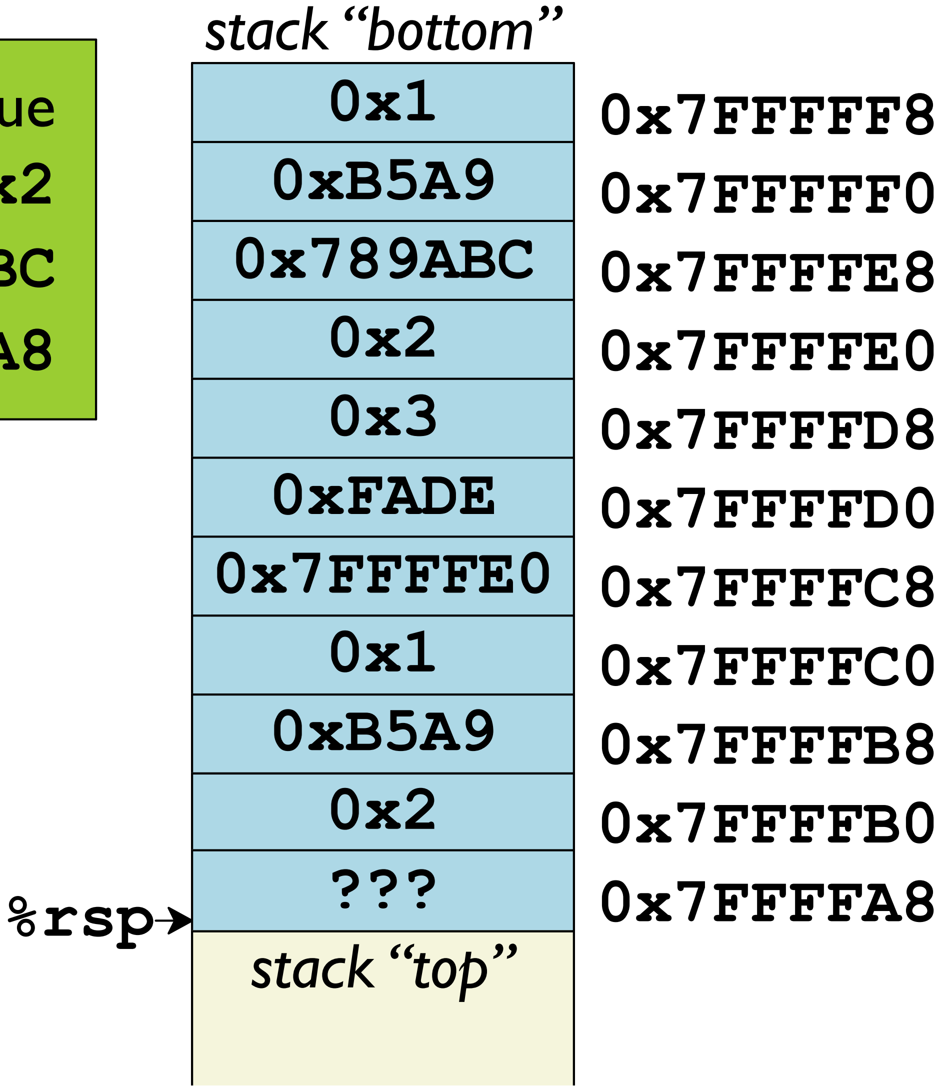
	0x1	0x7FFFFF8
	0xB5A9	0x7FFFFF0
	0x789ABC	0x7FFFFE8
	0x2	0x7FFFFE0
	0x3	0x7FFFFD8
	0xFADE	0x7FFFFD0
	0x7FFFFE0	0x7FFFFC8
	0x1	0x7FFFFC0
	0xB5A9	0x7FFFFB8
%rsp→	0x2	0x7FFFFB0

stack "top"

C Stack Operations

register	value
`%rax`	`0x2`
`%rbx`	`0x789ABC`
`%rsp`	`0x7FFFFA8`

```
popq %rax
popq %rbx
pushq %rax
subq $8,%rsp
```



stack "bottom"

`0x1`	`0x7FFFFF8`
`0xB5A9`	`0x7FFFFF0`
`0x789ABC`	`0x7FFFFE8`
`0x2`	`0x7FFFFE0`
`0x3`	`0x7FFFFD8`
`0xFADE`	`0x7FFFFD0`
`0x7FFFFE0`	`0x7FFFFC8`
`0x1`	`0x7FFFFC0`
`0xB5A9`	`0x7FFFFB8`
`0x2`	`0x7FFFFB0`
`???`	`0x7FFFFA8`

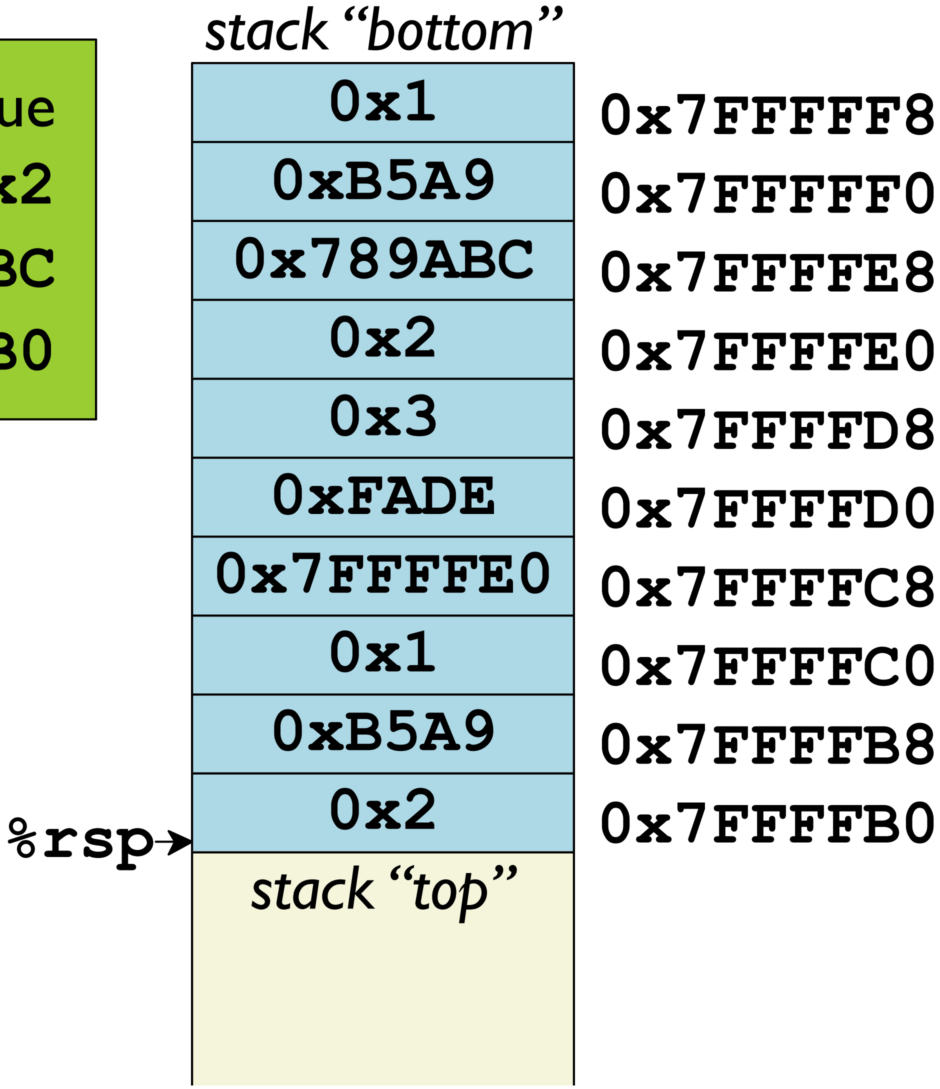
`%rsp`→

stack "top"

C Stack Operations

register	value
%rax	**0x2**
%rbx	**0x789ABC**
%rsp	**0x7FFFFB0**

```
popq %rax
popq %rbx
pushq %rax
subq $8,%rsp
addq $8,%rsp
```



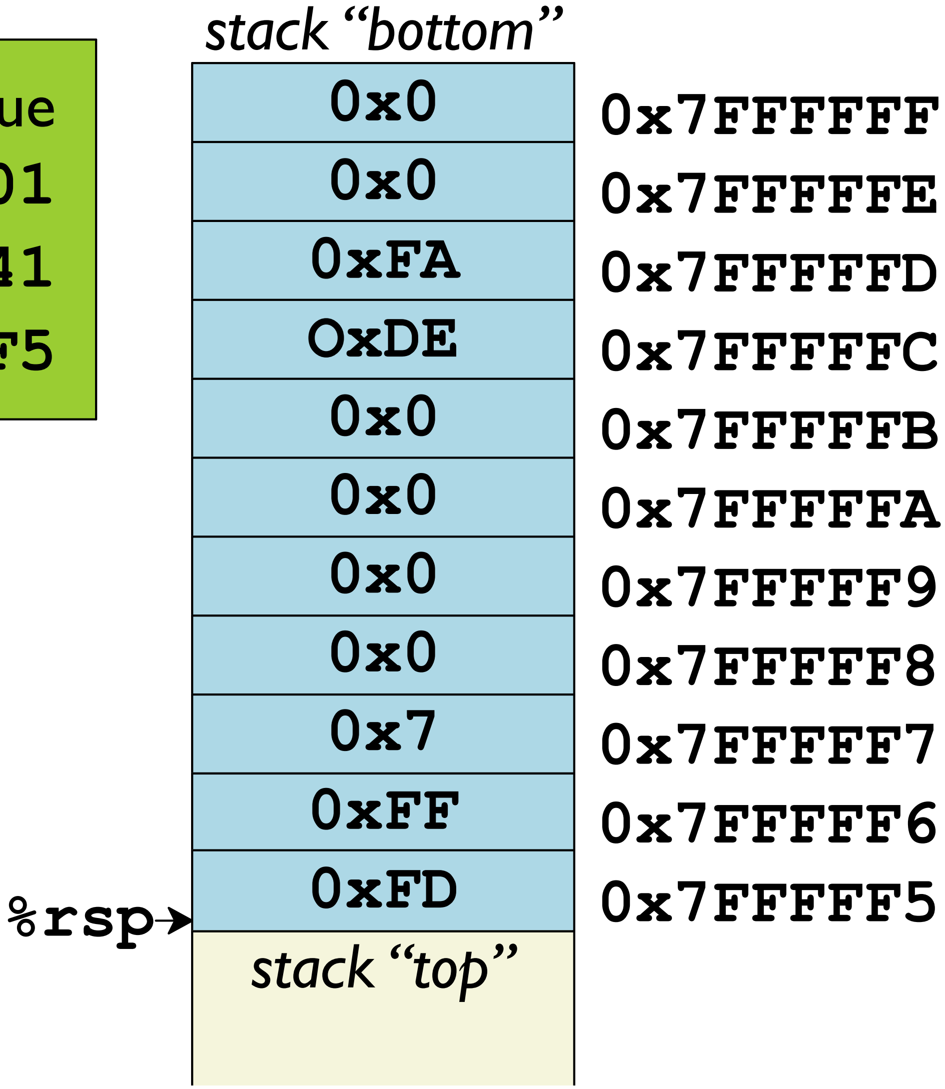
stack "bottom"

	0x1	0x7FFFFF8
	0xB5A9	0x7FFFFF0
	0x789ABC	0x7FFFFE8
	0x2	0x7FFFFE0
	0x3	0x7FFFFD8
	0xFADE	0x7FFFFD0
	0x7FFFFE0	0x7FFFFC8
	0x1	0x7FFFFC0
	0xB5A9	0x7FFFFB8
%rsp→	0x2	0x7FFFFB0

stack "top"

C Stack Operations

register	value
`%rax`	`0x101`
`%rbx`	`0x41`
`%rsp`	`0x7FFFFF5`



stack "bottom"

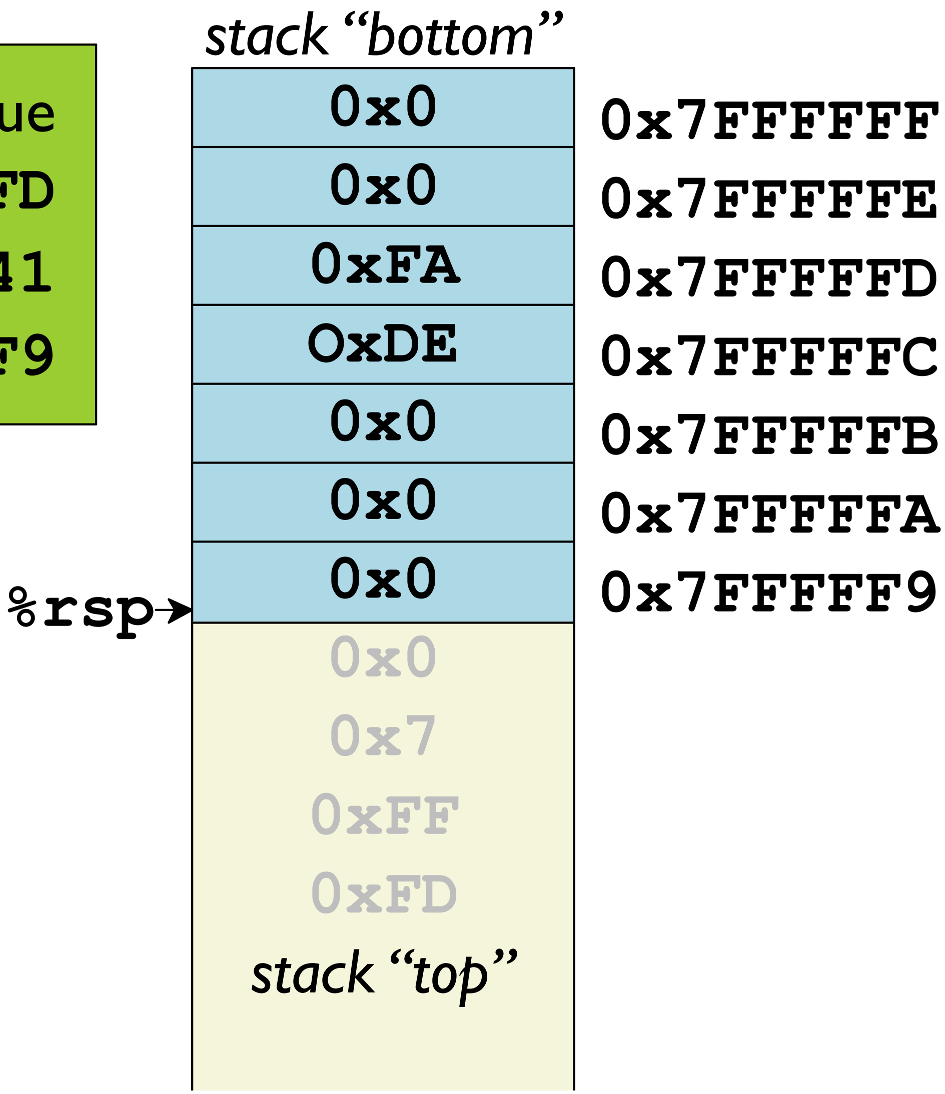
`0x0`	`0x7FFFFFF`
`0x0`	`0x7FFFFFE`
`0xFA`	`0x7FFFFFD`
`OxDE`	`0x7FFFFFC`
`0x0`	`0x7FFFFFB`
`0x0`	`0x7FFFFFA`
`0x0`	`0x7FFFFF9`
`0x0`	`0x7FFFFF8`
`0x7`	`0x7FFFFF7`
`0xFF`	`0x7FFFFF6`
`0xFD`	`0x7FFFFF5`

stack "top"

C Stack Operations

register	value
%rax	0x7FFFD
%rbx	0x41
%rsp	0x7FFFFF9

```
popl %eax
```



stack "bottom"

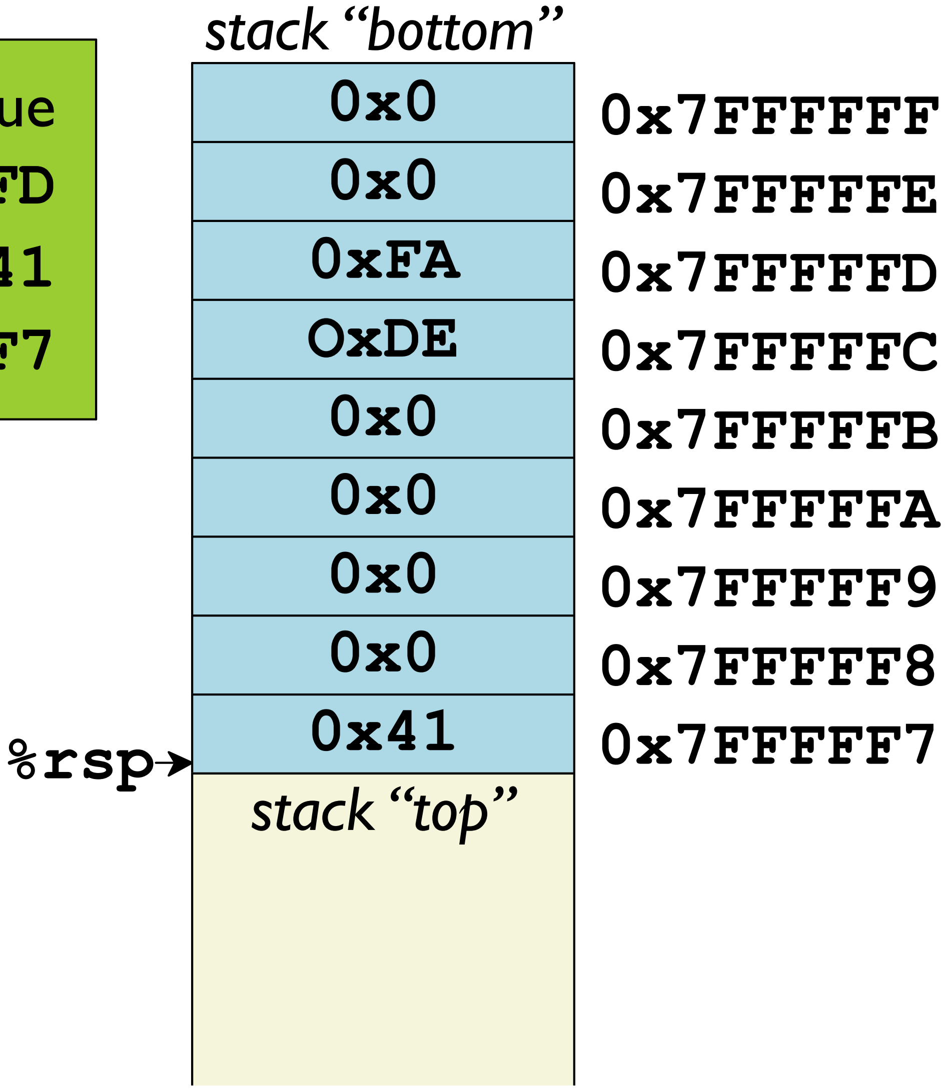
0x0	0x7FFFFFF
0x0	0x7FFFFFE
0xFA	0x7FFFFFD
OxDE	0x7FFFFFC
0x0	0x7FFFFFB
0x0	0x7FFFFFA
0x0	0x7FFFFF9
0x0	
0x7	
0xFF	
0xFD	

stack "top"

C Stack Operations

register	value
%rax	**0x7FFFD**
%rbx	**0x41**
%rsp	**0x7FFFFF7**

```
popl %eax
pushw %bx
```



0x0	0x7FFFFFF
0x0	0x7FFFFFE
0xFA	0x7FFFFFD
OxDE	0x7FFFFFC
0x0	0x7FFFFFB
0x0	0x7FFFFFA
0x0	0x7FFFFF9
0x0	0x7FFFFF8
0x41	0x7FFFFF7

Local Variables

```
#include <stdio.h>
void f();
void g();

int main() {
  int a;
  printf("&a in m: %p\n", &a);
  f();
  g();
  return 0;
}

void f() {
  double b;
  printf("&b in f: %p\n", &b);
}

void g() {
  char c;
  printf("&c in g: %p\n", &c);
}
```

Copy

Watching the Stack in gdb

```
$ gdb ./a.out
(gdb) break f
Breakpoint 1 at 0x400530: file main.c, line 13.
(gdb) run
Starting program: /home/mflatt/cs4400/./a.out
Breakpoint 1, f () at main.c:13
13   void f() {
(gdb) n
15   printf("&b in f: %p\n", &b);
(gdb) p &b
$1 = (double *) 0x7fffffffe0b8
(gdb) p $rsp
$2 = (void *) 0x7fffffffe0b0
(gdb) disassem f
Dump of assembler code for function f:
   0x0000000000400530 <+0>:   sub    $0x18,%rsp
=> 0x0000000000400534 <+4>:   lea    0x8(%rsp),%rsi
   0x0000000000400539 <+9>:   mov    $0x400630,%edi
   0x000000000040053e <+14>:  mov    $0x0,%eax
   0x0000000000400543 <+19>:  callq  0x400410 <printf@plt>
   0x0000000000400548 <+24>:  add    $0x18,%rsp
   0x000000000040054c <+28>:  retq
```

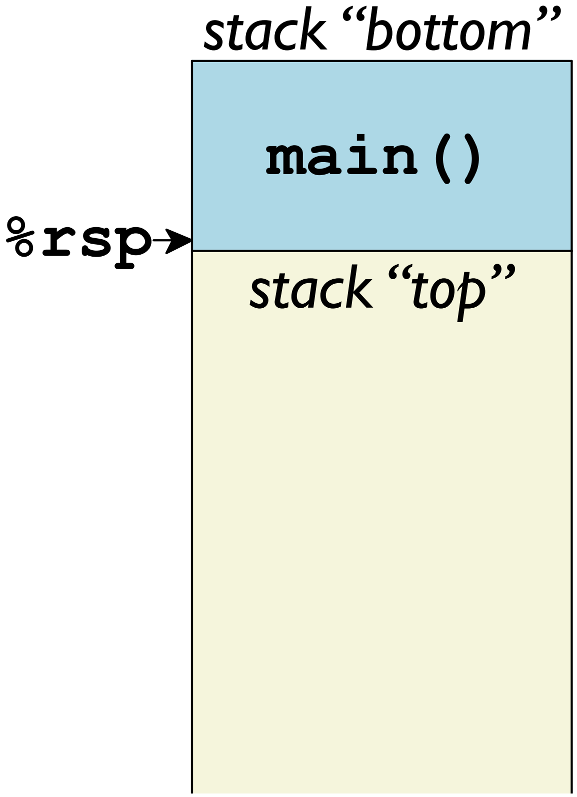
Recursive Functions Need Stack Frames

```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy



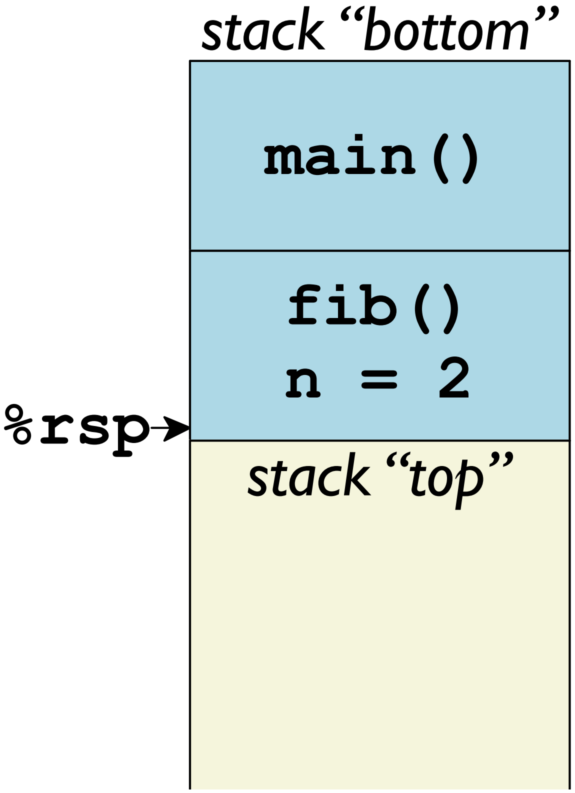
Recursive Functions Need Stack Frames

```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy



stack "bottom"

main()

fib()
n = 2

%rsp

stack "top"

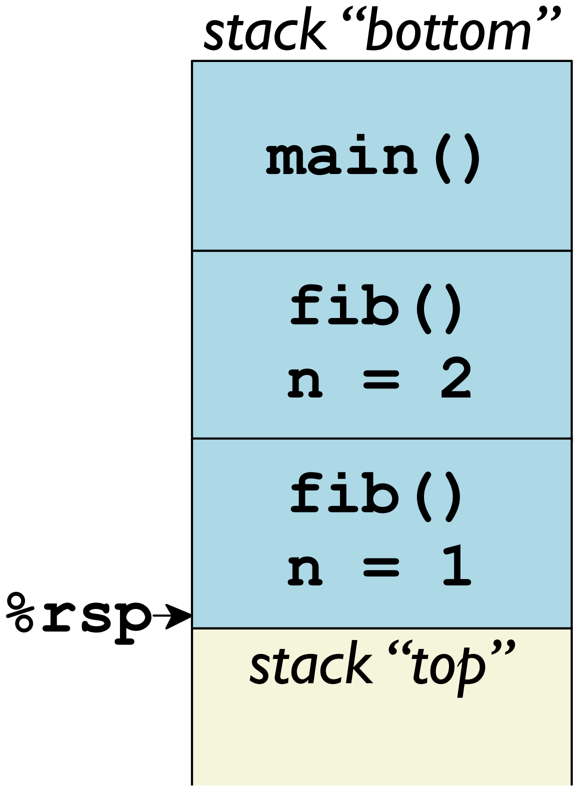
Recursive Functions Need Stack Frames

```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy



stack "bottom"

main()

fib()
n = 2

fib()
n = 1

%rsp

stack "top"

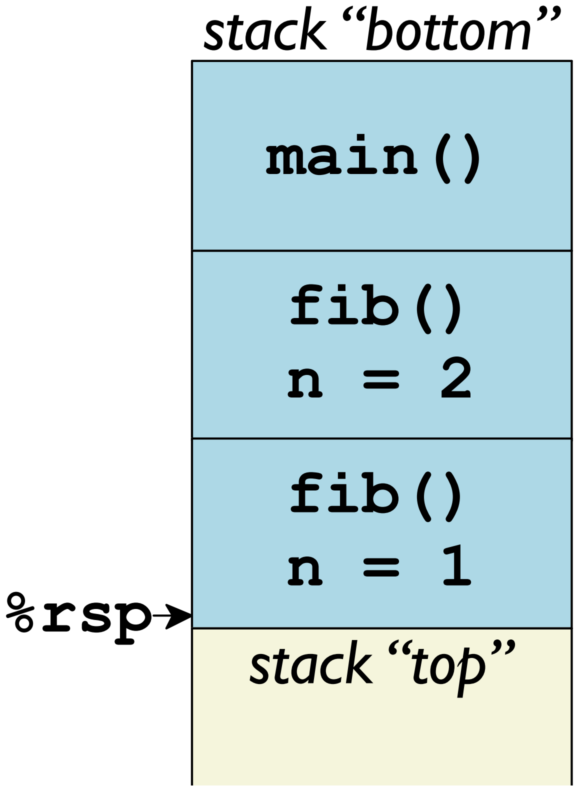
Recursive Functions Need Stack Frames

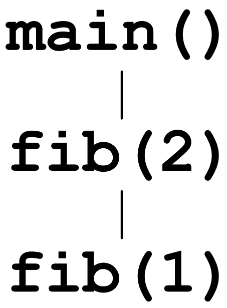
```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Copy





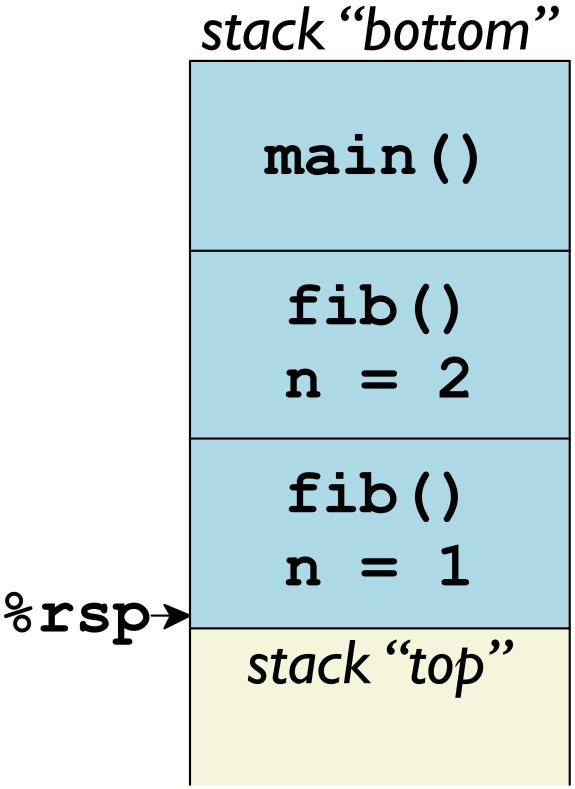
Recursive Functions Need Stack Frames

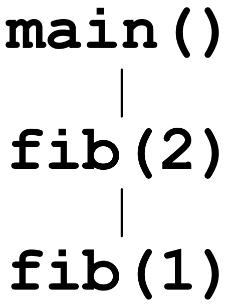
```c
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy





Each call of **`fib`** needs it own ***stack frame***

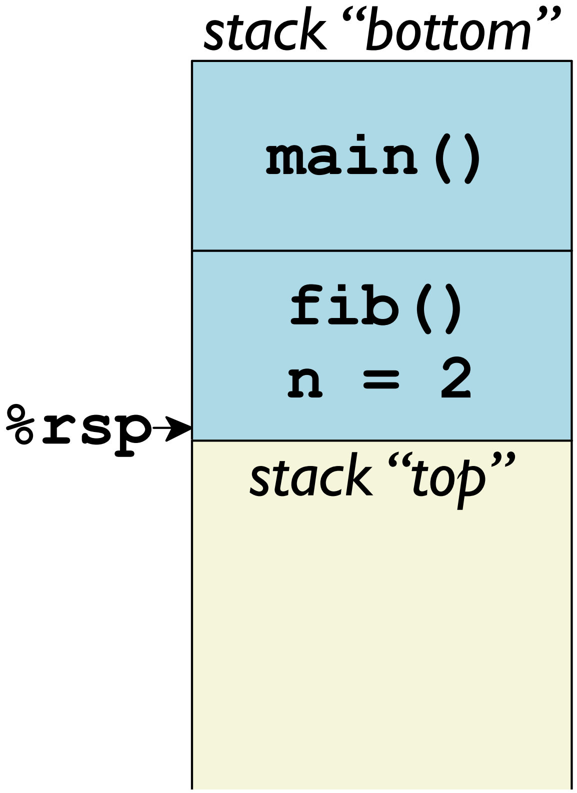
Recursive Functions Need Stack Frames

```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy



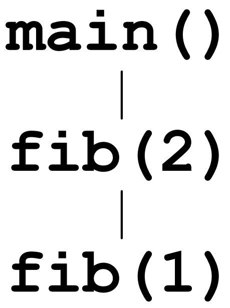
stack "bottom"

`main()`

`fib()`
`n = 2`

`%rsp`→

stack "top"



`main()`

`fib(2)`

`fib(1)`

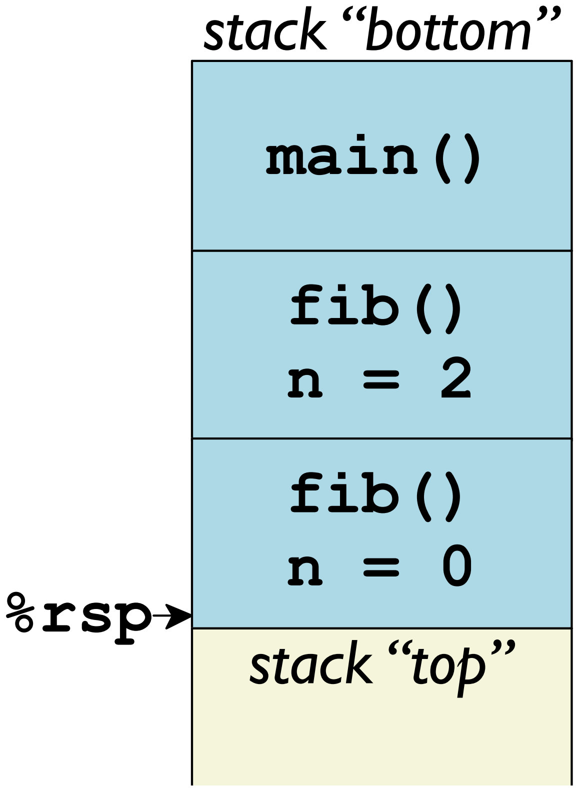
Recursive Functions Need Stack Frames

```
#include <stdio.h>

int main() {
  printf("%d\n", fib(2));
}

int fib(int n) {
  if ((n == 1) || (n == 0))
    return 1;
  else
   return fib(n-1) + fib(n-2);
}
```

Copy



stack “bottom”

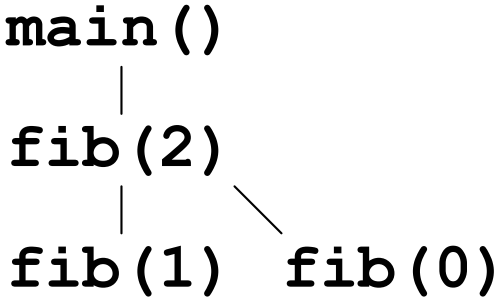
main()

fib()
n = 2

fib()
n = 0

%rsp

stack “top”

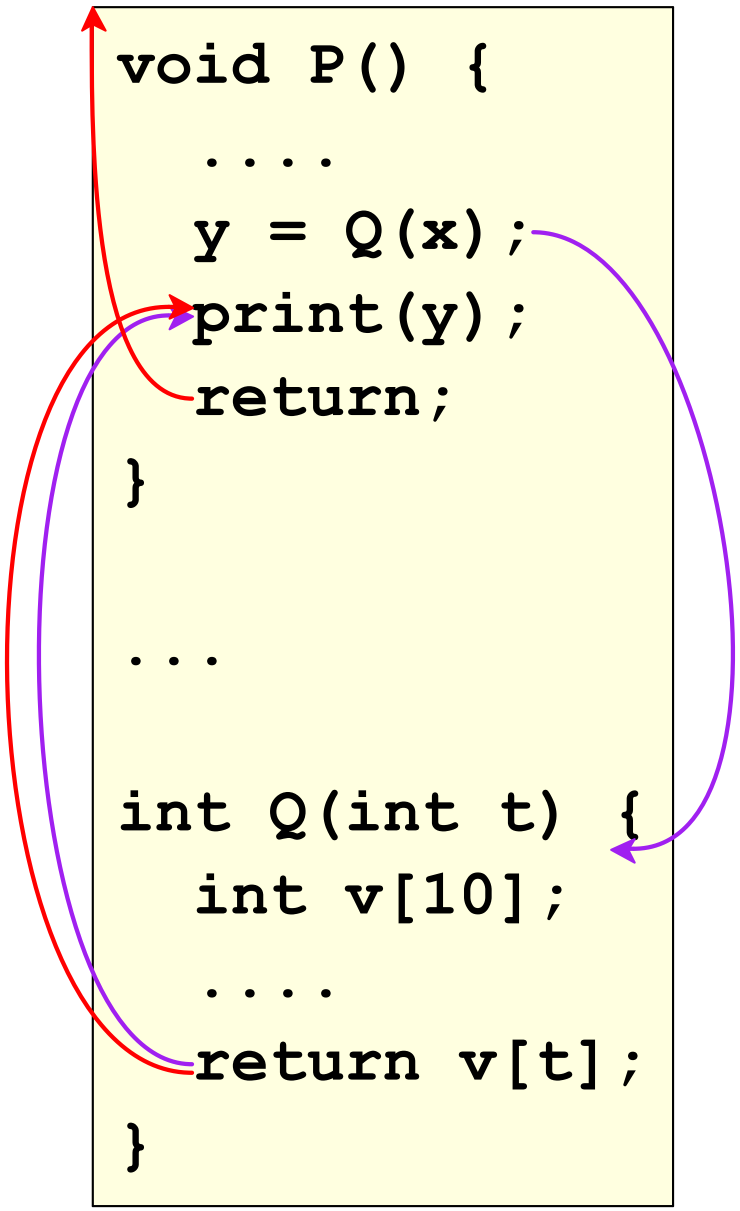


main()

fib(2)

fib(1) fib(0)

Calling Procedures



```
void P() {
   ....
   y = Q(x);
   print(y);
   return;
}

...

int Q(int t) {
   int v[10];
   ....
   return v[t];
}
```

Callee pops return address off the stack

Caller puts return address on the stack

Calling Procedures

`call`*x* *source*

Combines two actions:

- Pushes *next* value of `%rip`
- Jumps to *source* (i.e., sets `%rip` to *source*)

```
0x50300: callq 0x50640
0x50305: ....
```

Returning from Procedures

ret*x*

Pops value to **%rip**

```
0x50300: callq 0x50640
0x50305: ....
....
0x50640: ....
0x50650: retq
```

Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                    ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                    ....
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```

Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                 ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                 ....
```

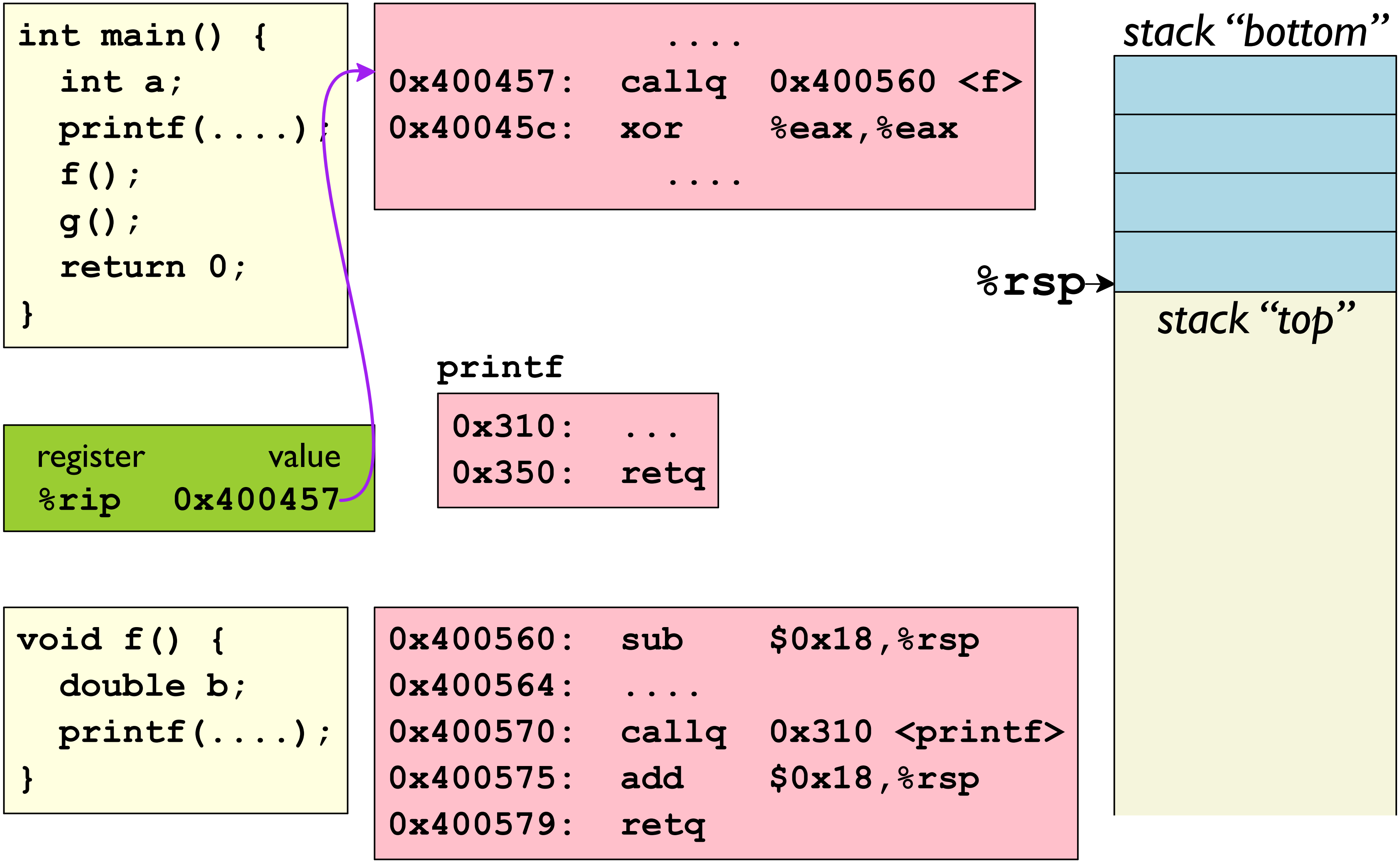
printf

```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```

Call Example



```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                ....
```

register	value
%rip	0x400457

printf

```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```

Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                ....
0x400457:   callq  0x400560 <f>
0x40045c:   xor    %eax,%eax
                ....
```

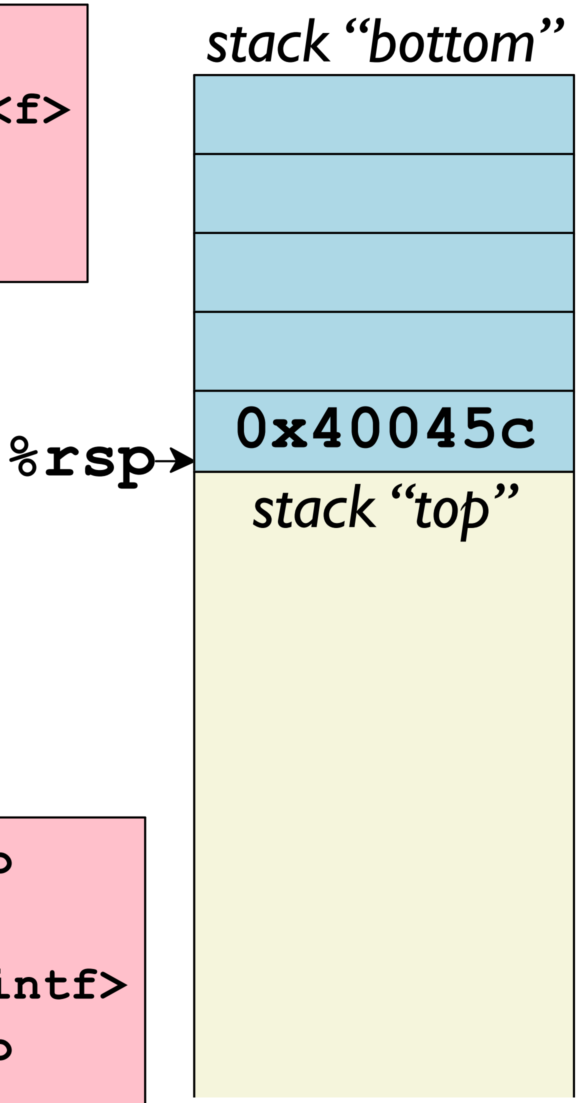
printf

```
0x310:   ...
0x350:   retq
```

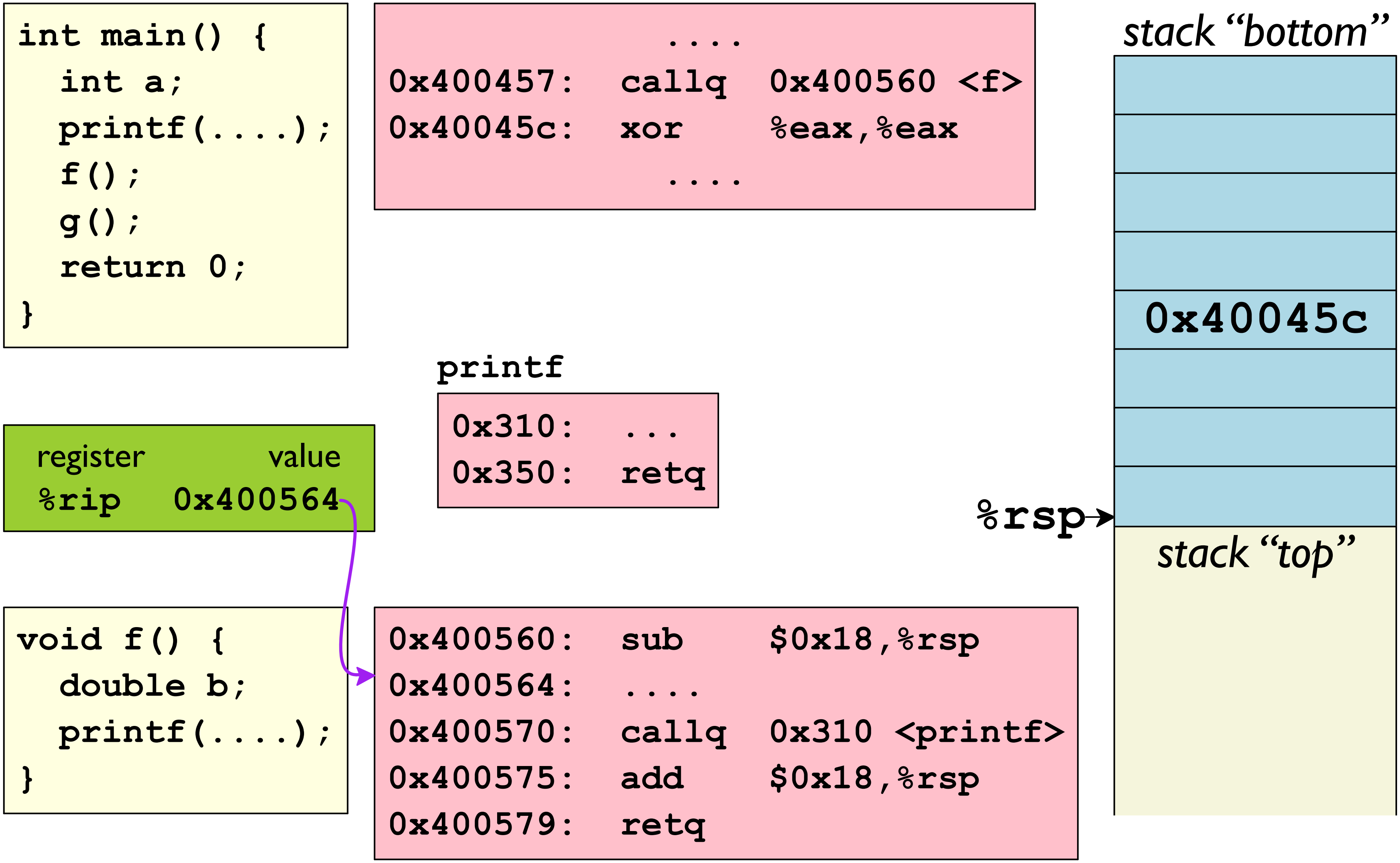
register	value
%rip	0x400560

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub    $0x18,%rsp
0x400564:   ....
0x400570:   callq  0x310 <printf>
0x400575:   add    $0x18,%rsp
0x400579:   retq
```



Call Example



```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                 ....
0x400457:   callq  0x400560 <f>
0x40045c:   xor    %eax,%eax
                 ....
```

register	value
%rip	0x400564

printf

```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub    $0x18,%rsp
0x400564:   ....
0x400570:   callq  0x310 <printf>
0x400575:   add    $0x18,%rsp
0x400579:   retq
```

Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                ....
```

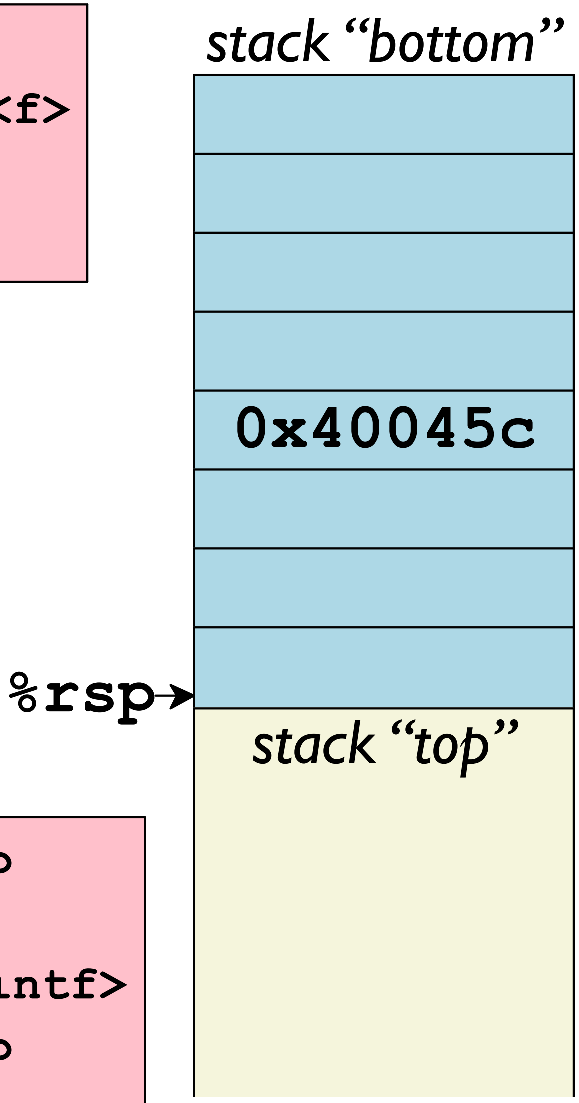
printf

```
0x310:   ...
0x350:   retq
```

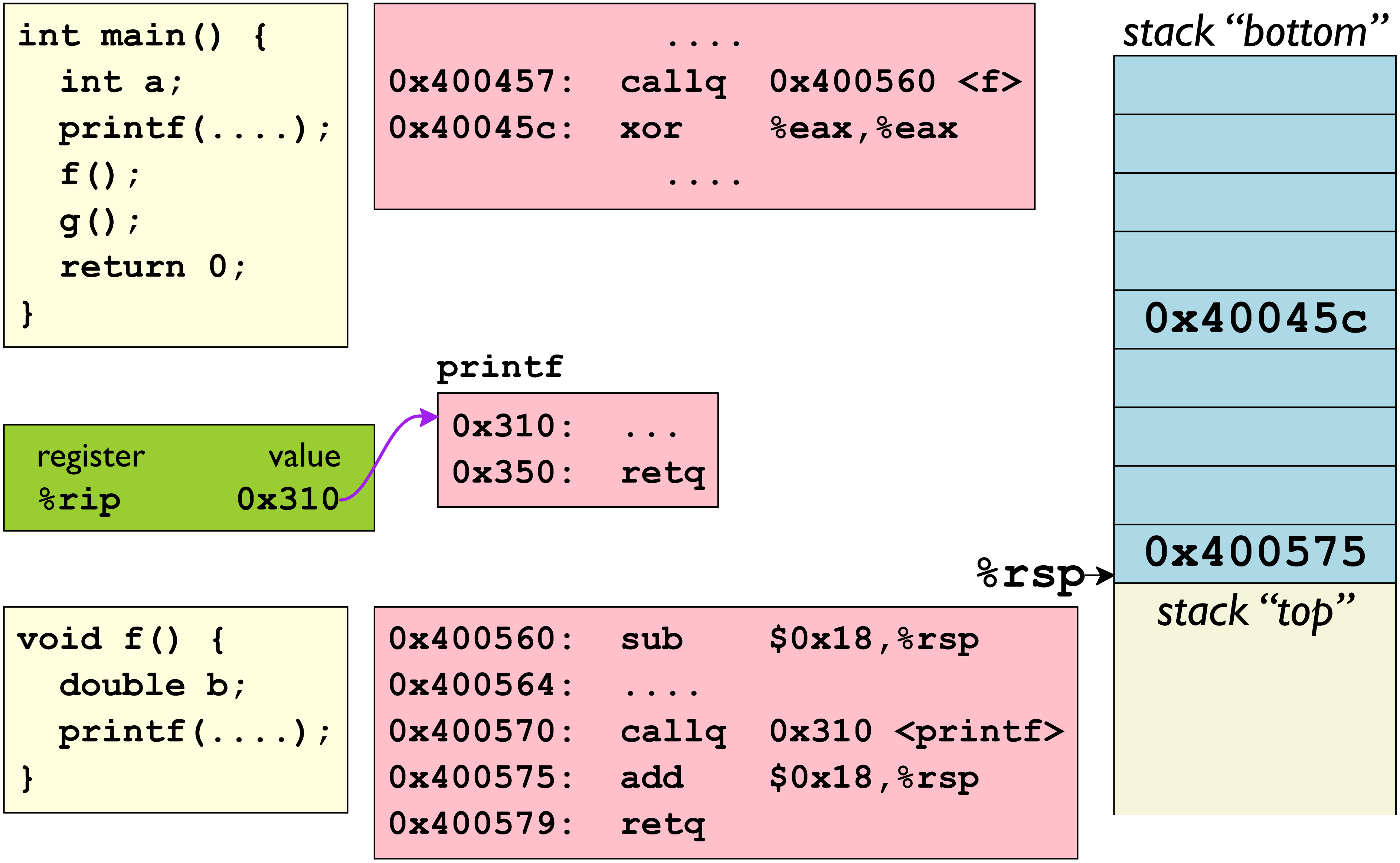
register	value
%rip	0x400570

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```



Call Example



```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                    ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                    ....
```

register	value
%rip	0x310

printf

```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```

Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                 ....
0x400457:   callq  0x400560 <f>
0x40045c:   xor    %eax,%eax
                 ....
```

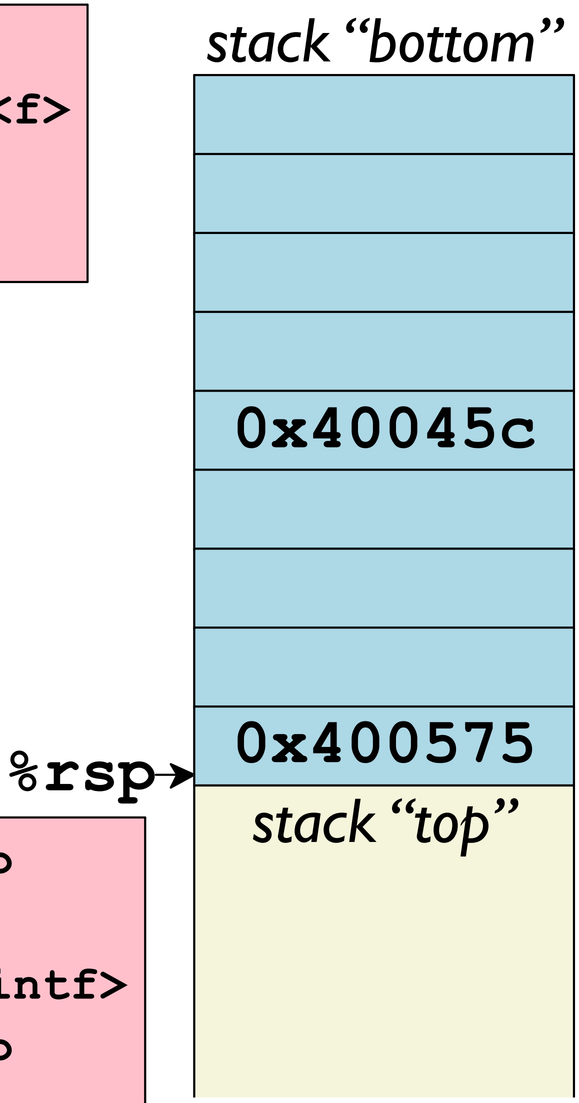
printf

```
0x310:   ...
0x350:   retq
```

register	value
%rip	0x350

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub    $0x18,%rsp
0x400564:   ....
0x400570:   callq  0x310 <printf>
0x400575:   add    $0x18,%rsp
0x400579:   retq
```



Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                  ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                  ....
```

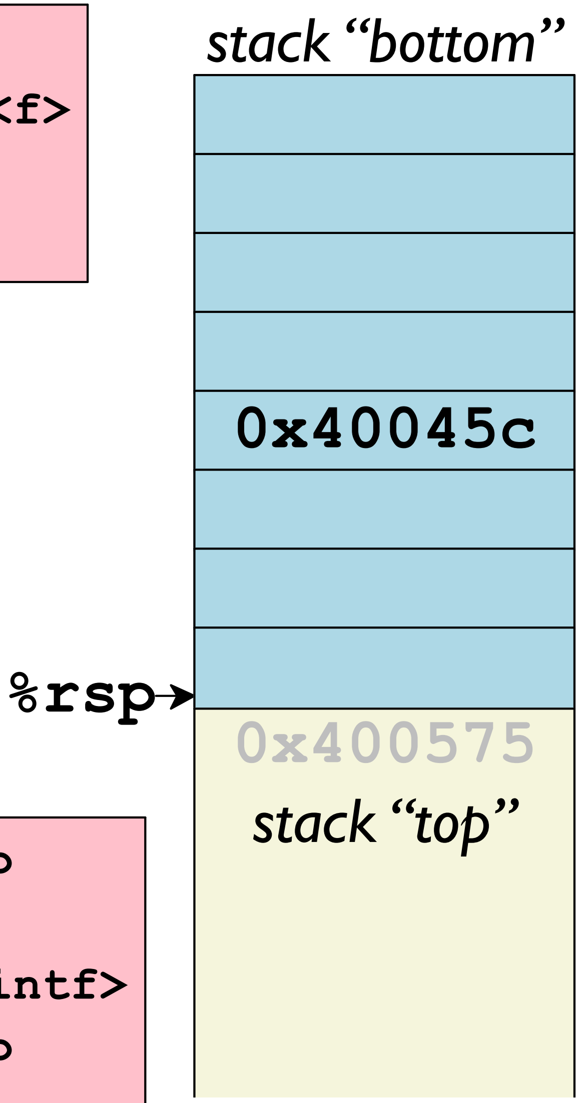
register	value
%rip	0x400575

printf

```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```



Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                   ....
0x400457:   callq  0x400560 <f>
0x40045c:   xor    %eax,%eax
                   ....
```

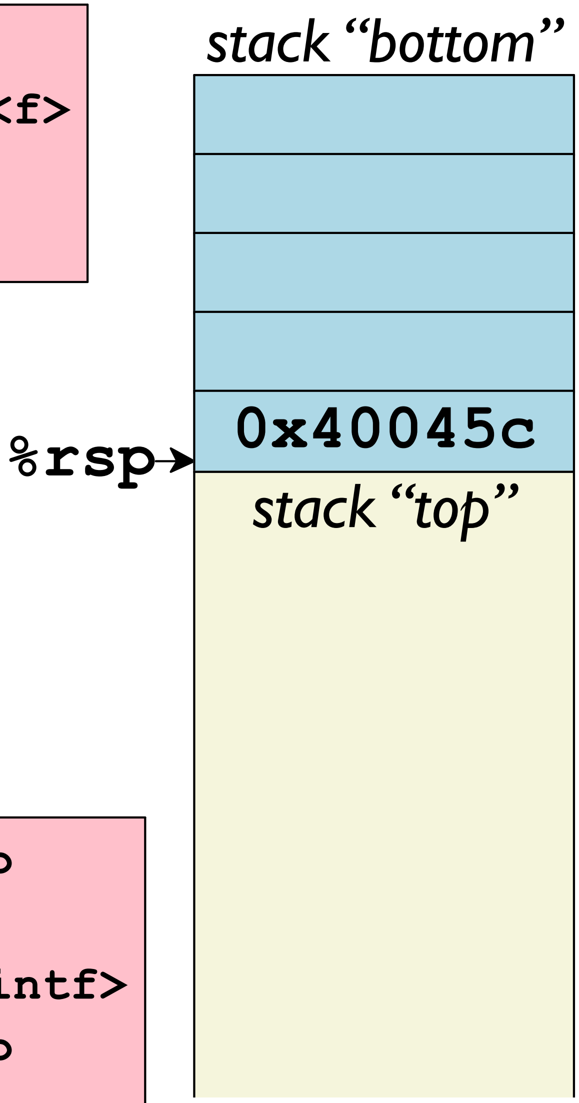
printf

```
0x310:   ...
0x350:   retq
```

register	value
%rip	0x400579

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub    $0x18,%rsp
0x400564:   ....
0x400570:   callq  0x310 <printf>
0x400575:   add    $0x18,%rsp
0x400579:   retq
```



Call Example

```
int main() {
  int a;
  printf(....);
  f();
  g();
  return 0;
}
```

```
                  ....
0x400457:   callq   0x400560 <f>
0x40045c:   xor     %eax,%eax
                  ....
```

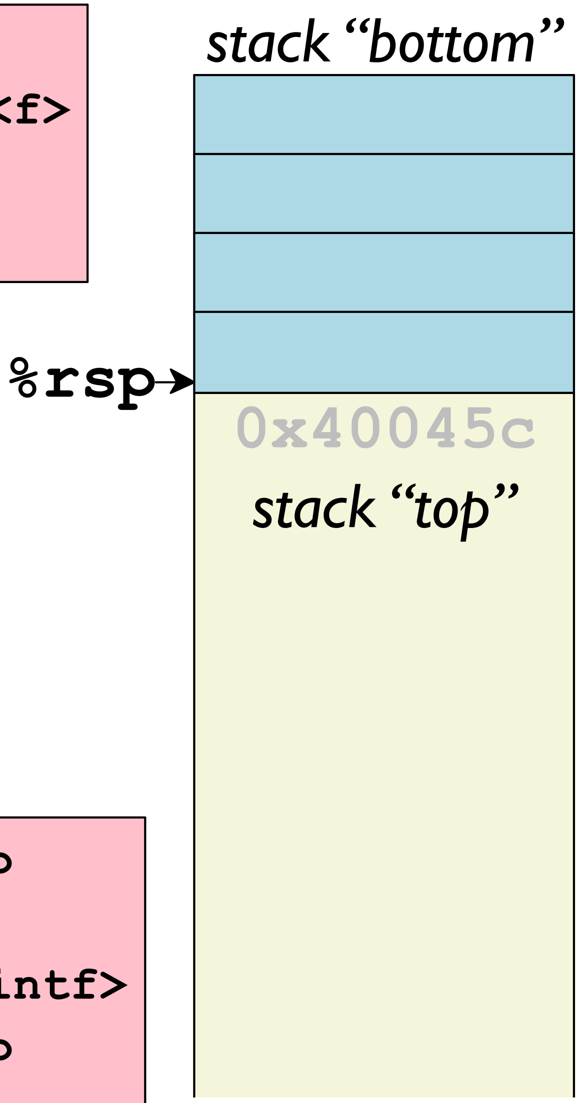
register	value
%rip	0x40045c

printf

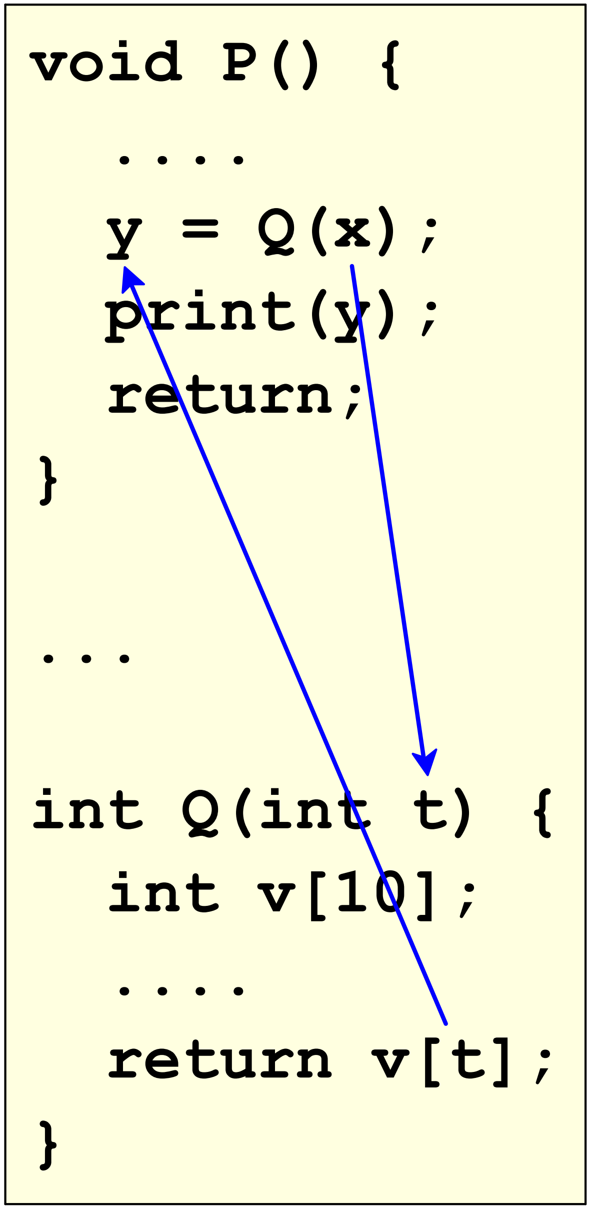
```
0x310:   ...
0x350:   retq
```

```
void f() {
  double b;
  printf(....);
}
```

```
0x400560:   sub     $0x18,%rsp
0x400564:   ....
0x400570:   callq   0x310 <printf>
0x400575:   add     $0x18,%rsp
0x400579:   retq
```



Procedure Arguments and Results



```
void P() {
   ....
   y = Q(x);
   print(y);
   return;
}

...

int Q(int t) {
   int v[10];
   ....
   return v[t];
}
```

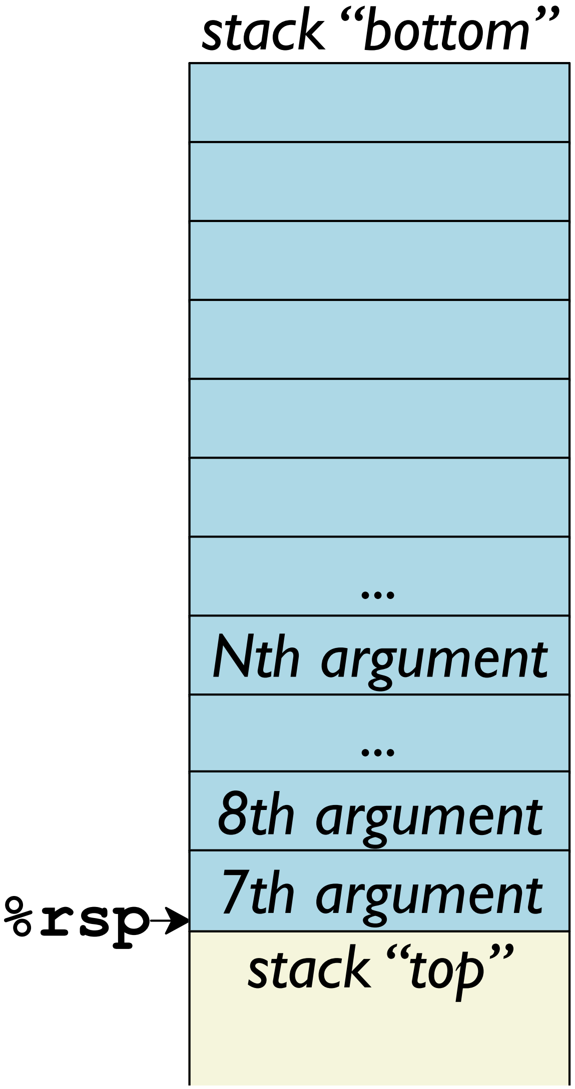
Procedure Arguments and Results

First six arguments:

register	value
`%rdi`	*1st argument*
`%rsi`	*2nd argument*
`%rdx`	*3rd argument*
`%rcx`	*4th argument*
`%r8`	*5th argument*
`%r9`	*6th argument*

Return value:

register	value
`%rax`	*result*



Example of Receiving Arguments

```
long mult2(long a, long b) {
  long s = a * b;
  return s;
}
```

register	value
%rdi	*1st argument*
%rsi	*2nd argument*
%rdx	*3rd argument*
%rcx	*4th argument*
%r8	*5th argument*
%r9	*6th argument*

```
mov    %rdi,%rax    # a
imul   %rsi,%rax    # a * b
retq                # Return
```

Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition

Example of Providing Arguments

```
long mult2(long a, long b);

int main() {
  return mult2(2, 3);
}
```

register	value
%rdi	*1st argument*
%rsi	*2nd argument*
%rdx	*3rd argument*
%rcx	*4th argument*
%r8	*5th argument*
%r9	*6th argument*

```
subq   $0x8,%rsp
movl   $0x3,%esi
movl   $0x2,%edi
callq   <mult2>
add    $0x8,%rsp
retq
```

Example of Providing Arguments

```
void rmultstore(long y, long x, long *dest) {
   long t = mult2(x, y);
   *dest = t;
}
```

register	value
%rdi	*1st argument*
%rsi	*2nd argument*
%rdx	*3rd argument*
%rcx	*4th argument*
%r8	*5th argument*
%r9	*6th argument*

```
....
movq   %rdi,%rax    # Save y
movq   %rsi,%rdi    # x as first argument
movq   %rax,%rsi    # y as second argument
callq  <mult2>      # mult2(x,y)
....
```

Example of Providing Arguments

```
void multstore(long x, long y, long *dest) {
   long t = mult2(x, y);
   *dest = t;
}
```

register	value
`%rdi`	*1st argument*
`%rsi`	*2nd argument*
`%rdx`	*3rd argument*
`%rcx`	*4th argument*
`%r8`	*5th argument*
`%r9`	*6th argument*

```
....
callq  <mult2>      # mult2(x,y)
....
```

What about **`dest`**?

Example of Providing Arguments

```
void multstore(long x, long y, long *dest) {
   long t = mult2(x, y);
   *dest = t;
}
```

register	value
`%rdi`	*1st argument*
`%rsi`	*2nd argument*
`%rdx`	*3rd argument*
`%rcx`	*4th argument*
`%r8`	*5th argument*
`%r9`	*6th argument*

```
pushq  %rbx          # Save %rbx
mov    %rdx,%rbx     # Save dest
callq  <mult2>       # mult2(x,y)
movq   %rax,(%rbx)   # Save at dest
popq   %rbx          # Restore %rbx
retq
```

`%rbx` is a ***preserved register***

Register Protocols

Some regsisters are ***temporaries***

- call a function ⇒ register value may change on return
- a.k.a. ***caller-saved*** e.g., `%r10`, `%rsi`

Some regsisters are ***preserved***

- call a function ⇒ register value the same on return
- a.k.a. ***callee-saved*** e.g., `%rbx`, `%rsp`

Classification of registers is part of an ***application binary interface*** (ABI)

x86-64 Linux Register Usage

	register	usage
Caller-saved	`%rax`	*return value*
	`%rdi`	*1st argument*
	`%rsi`	*2nd argument*
	`%rdx`	*3rd argument*
	`%rcx`	*4th argument*
	`%r8`	*5th argument*
	`%r9`	*6th argument*
	`%r10`	*temporary*
	`%r11`	*temporary*
Callee-saved	`%rbx`	*preserved*
	`%r12`	*preserved*
	`%r13`	*preserved*
	`%r14`	*preserved*
	`%rbp`	*stack frame*
	`%rsp`	*stack pointer*

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  retq
```

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  retq
```

save caller's `%rbx`

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  retq
```

use **%rbx** to save **x** across call

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  retq
```

after call, `%rbx` has `x`

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %
  popq     %rbx
  retq
```

restore caller’s `%rbx`

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  retq
```

make space for **`v1`**

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  retq
```

initialize **`v1`**

Another Callee-Saved Register Example

```
long incr(long *p, long val);

long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $152
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  retq
```

provide address of **v1**

Application Binary Interface

An OS-specific ABI defines

- How arguments are passed to functions

So far, only integer and address arguments

- How results are returned from functions

So far, only integer and address results

- Which registers are preserved (and not)

There are more registers...

- Other constraints, such as stack alignment

x86-64 Linux: stack aligned on call at 8 mod 16

- Optional debugging protocols

Debugging Information

```
gcc
```

vs.

```
gcc -g
```

vs.

```
gcc && strip -s
```

vs.

```
gcc -fno-asynchronous-unwind-tables
```

vs.

```
gcc -fno-asynchronous-unwind-tables
    -fno-omit-frame-pointer
```

Frame Pointer

Stack frames are optionally identified by a ***frame pointer***

- Frames form a linked list embedded in the stack
- Each function's ***prolog*** sets up the frame
- Each function's ***epilog*** destroys the frame
- **`%rbp`** points to the head of the list

  i.e., the current frame
- Local variables are accessed via **`%rbp`**

Using a Frame Pointer

```
       ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

Using a Frame Pointer

```
       ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

prolog

epilog

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

prolog

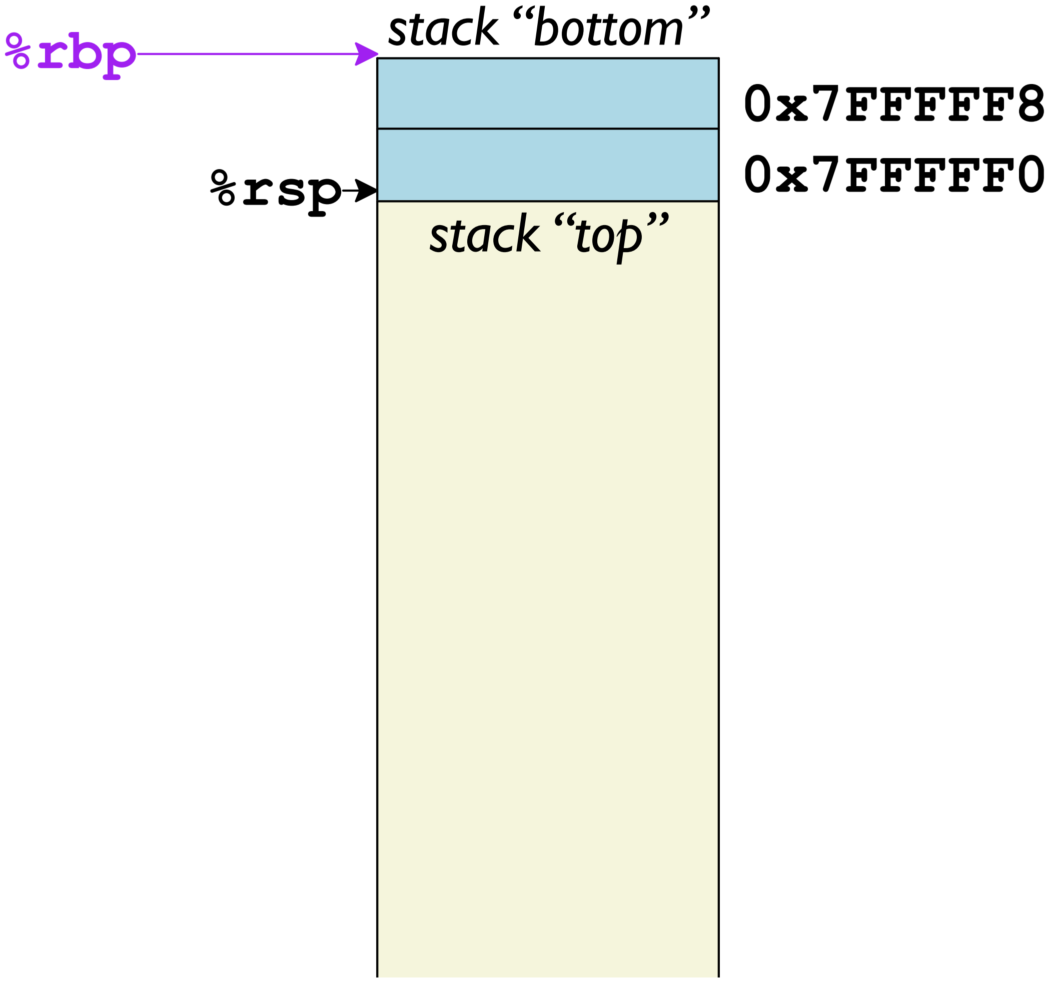
epilog

Using a Frame Pointer

```
      ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

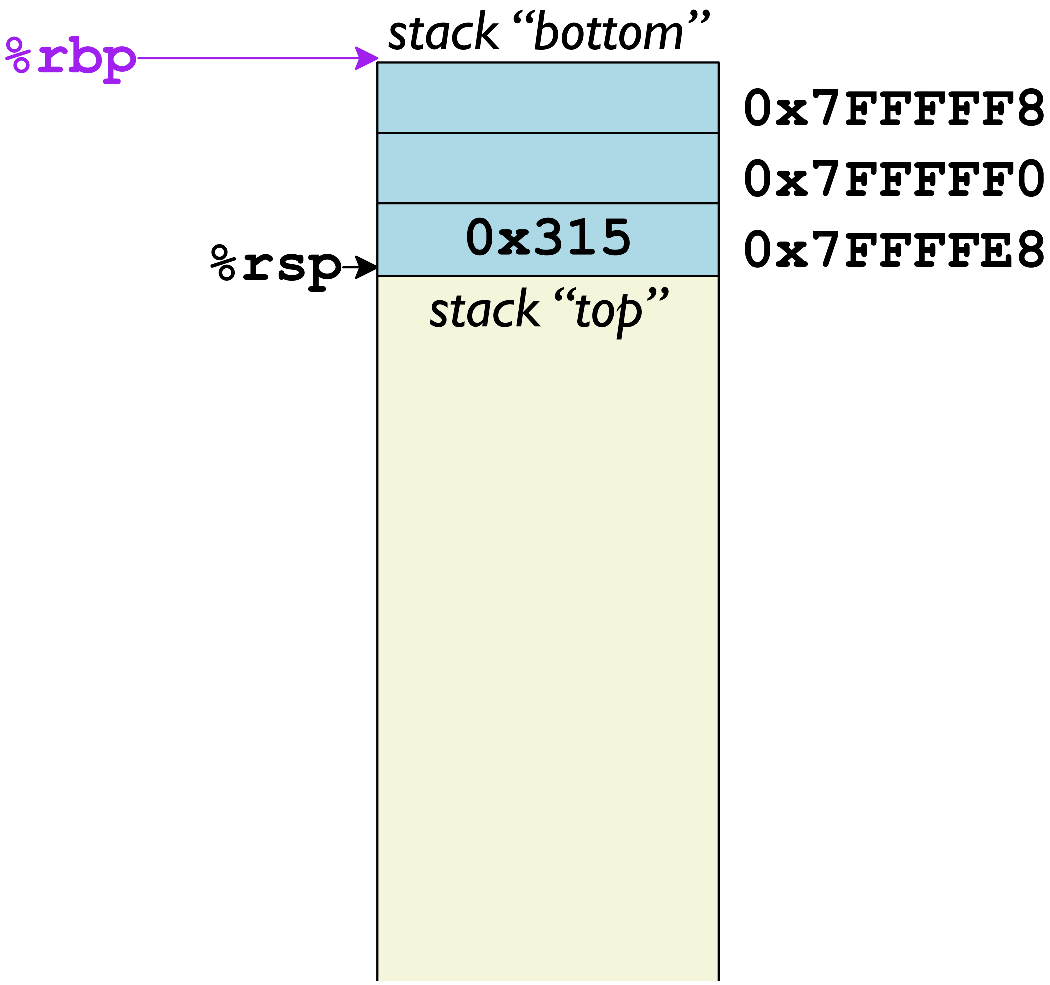


Using a Frame Pointer

```
        ....
0x310:  callq 0x400
0x315:  ....
```

```
0x400:  pushq %rbp
0x401:  movq  %rsp, %rbp
0x404:  ...
        ...-0x8(%rbp)...
        ...
0x420:  callq 0x500
0x425:  ....
0x430:  popq %rbp
0x431:  retq
```

```
0x500:  pushq %rbp
0x501:  movq  %rsp, %rbp
...
0x509:  popq %rbp
0x510:  retq
```

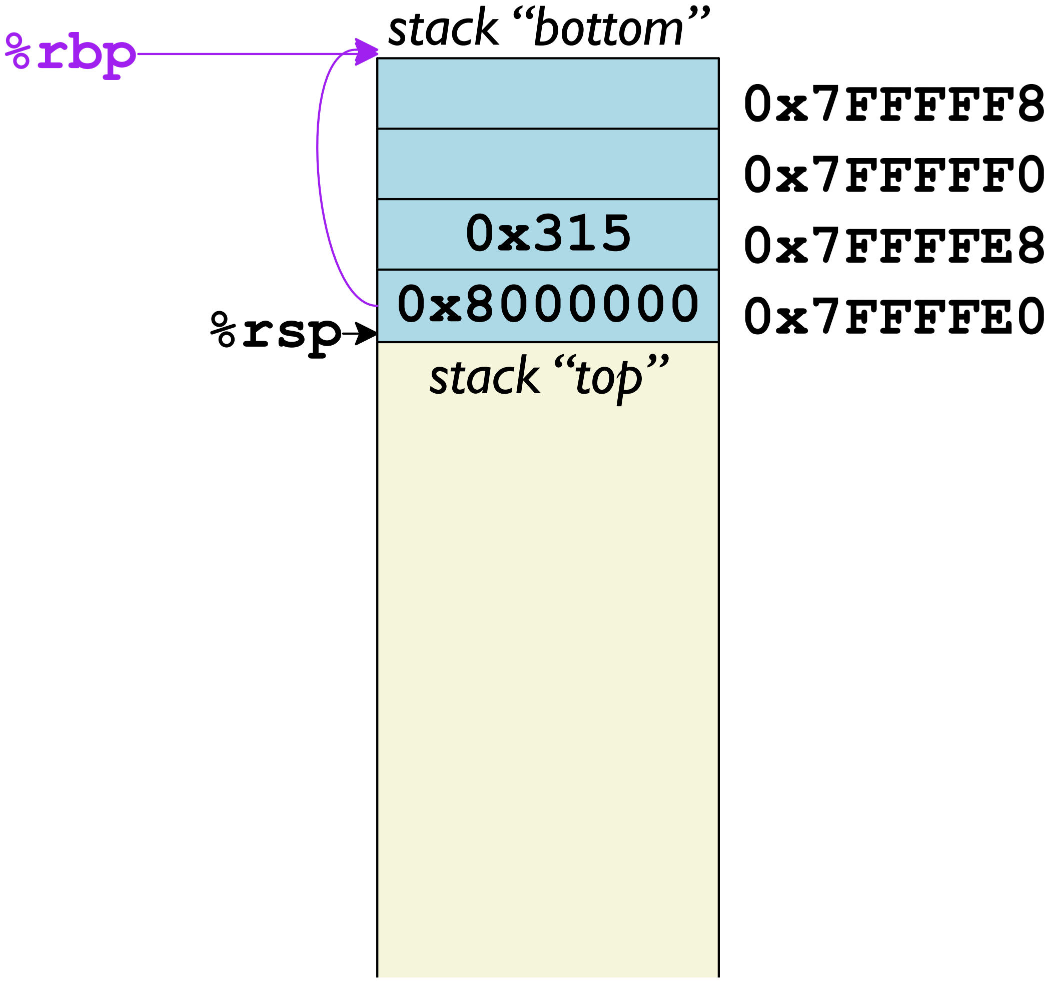


Using a Frame Pointer

```
      ....
0x310: callq 0x400
0x315: ....
```

```
  0x400: pushq %rbp
➡ 0x401: movq  %rsp, %rbp
  0x404: ...
         ...-0x8(%rbp)...
         ...
  0x420: callq 0x500
  0x425: ....
  0x430: popq %rbp
  0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

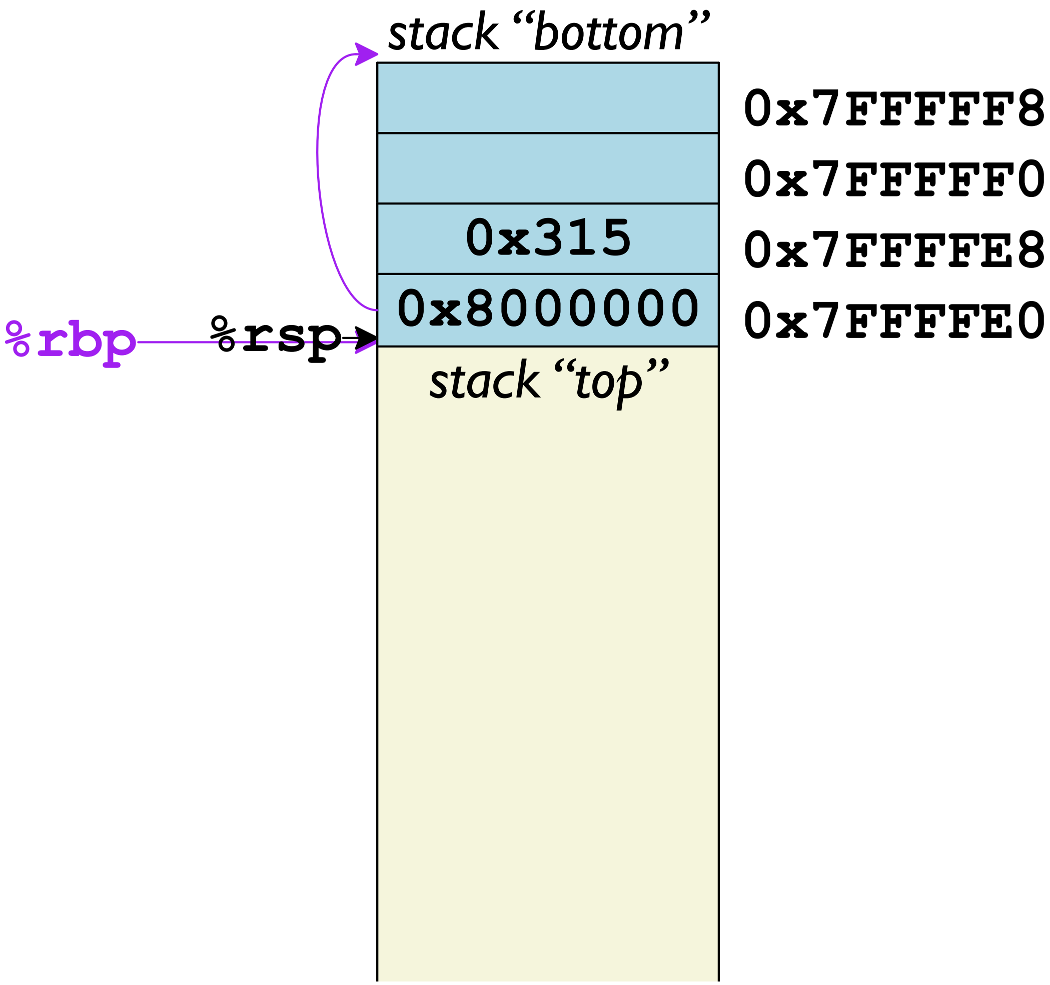


Using a Frame Pointer

```
        ....
0x310:  callq 0x400
0x315:  ....
```

```
0x400:  pushq %rbp
0x401:  movq  %rsp, %rbp
0x404:  ...
        ...-0x8(%rbp)...
        ...
0x420:  callq 0x500
0x425:  ....
0x430:  popq %rbp
0x431:  retq
```

```
0x500:  pushq %rbp
0x501:  movq  %rsp, %rbp
...
0x509:  popq %rbp
0x510:  retq
```

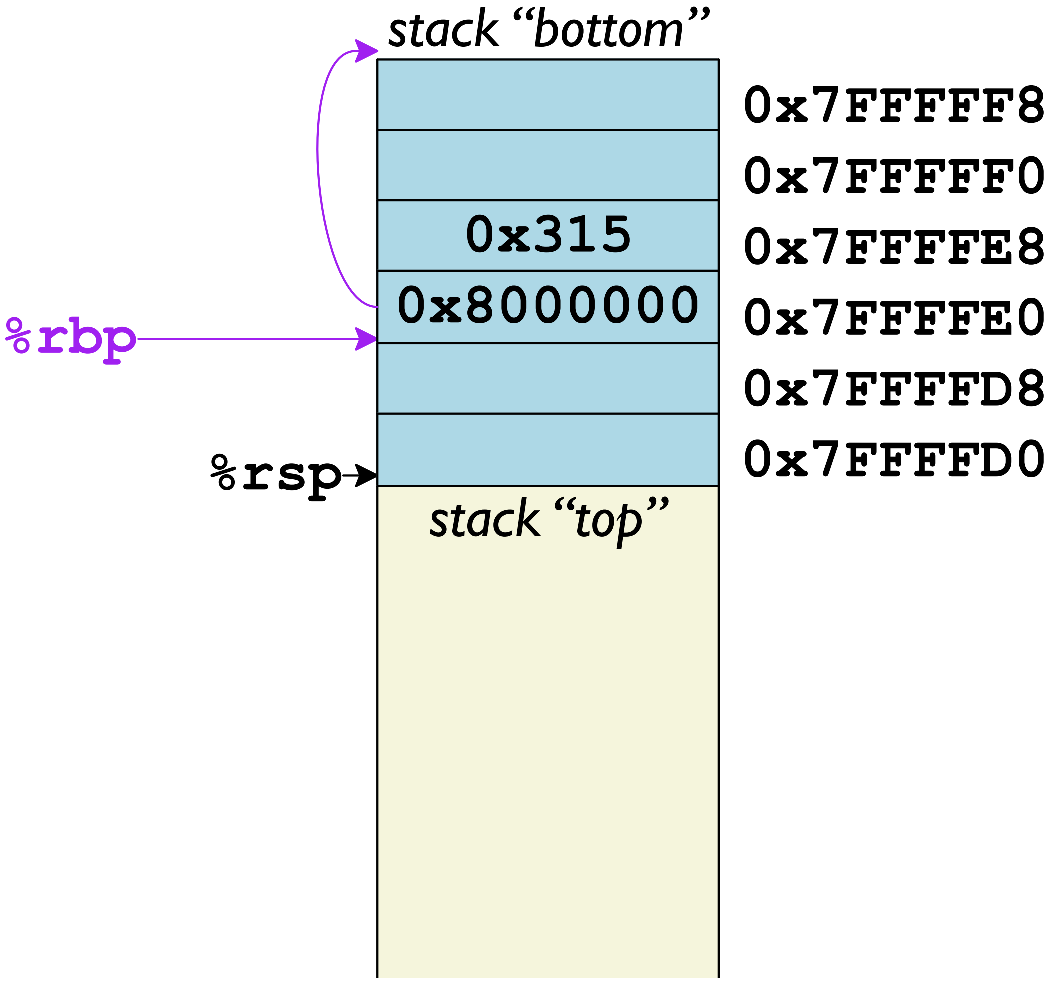


Using a Frame Pointer

```
        ....
0x310:  callq 0x400
0x315:  ....
```

```
0x400:  pushq %rbp
0x401:  movq  %rsp, %rbp
0x404:  ...
        ...-0x8(%rbp)...
        ...
0x420:  callq 0x500
0x425:  ....
0x430:  popq %rbp
0x431:  retq
```

```
0x500:  pushq %rbp
0x501:  movq  %rsp, %rbp
...
0x509:  popq %rbp
0x510:  retq
```



stack "bottom"

	0x7FFFFF8
	0x7FFFFF0
0x315	0x7FFFFE8
0x8000000	0x7FFFFE0
	0x7FFFFD8
	0x7FFFFD0

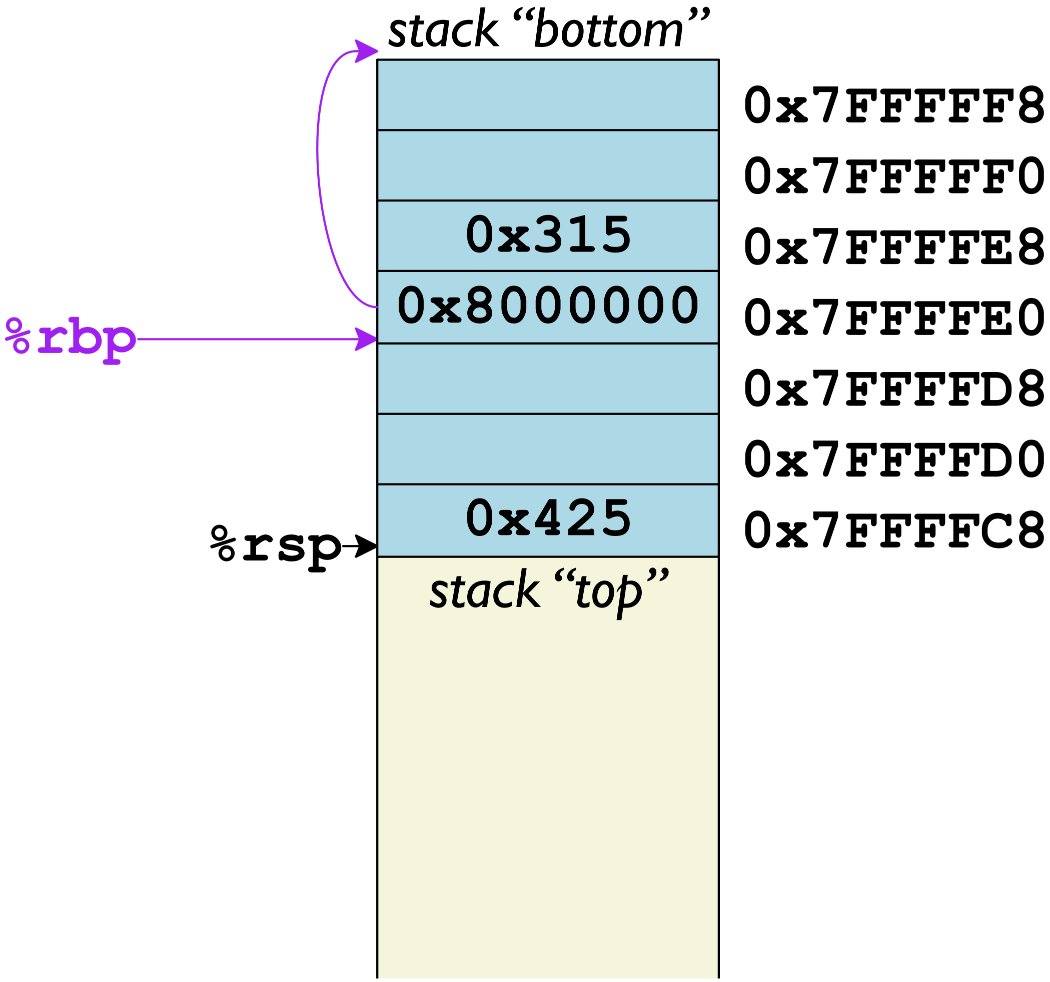
stack "top"

Using a Frame Pointer

```
        ....
0x310:  callq 0x400
0x315:  ....
```

```
0x400:  pushq %rbp
0x401:  movq  %rsp, %rbp
0x404:  ...
        ...-0x8(%rbp)...
        ...
0x420:  callq 0x500
0x425:  ....
0x430:  popq %rbp
0x431:  retq
```

```
0x500:  pushq %rbp
0x501:  movq  %rsp, %rbp
...
0x509:  popq %rbp
0x510:  retq
```

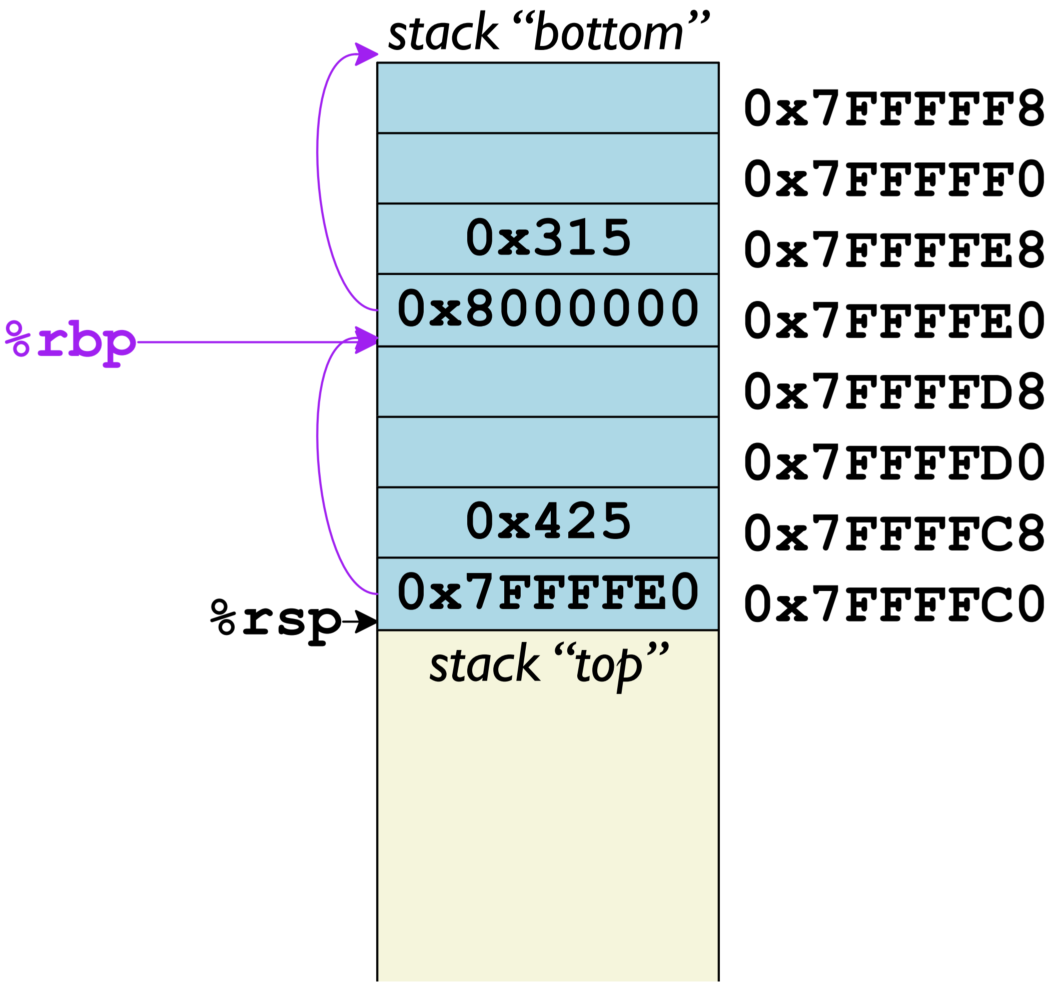


Using a Frame Pointer

```
      ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
  0x500: pushq %rbp
➡ 0x501: movq  %rsp, %rbp
  ...
  0x509: popq %rbp
  0x510: retq
```

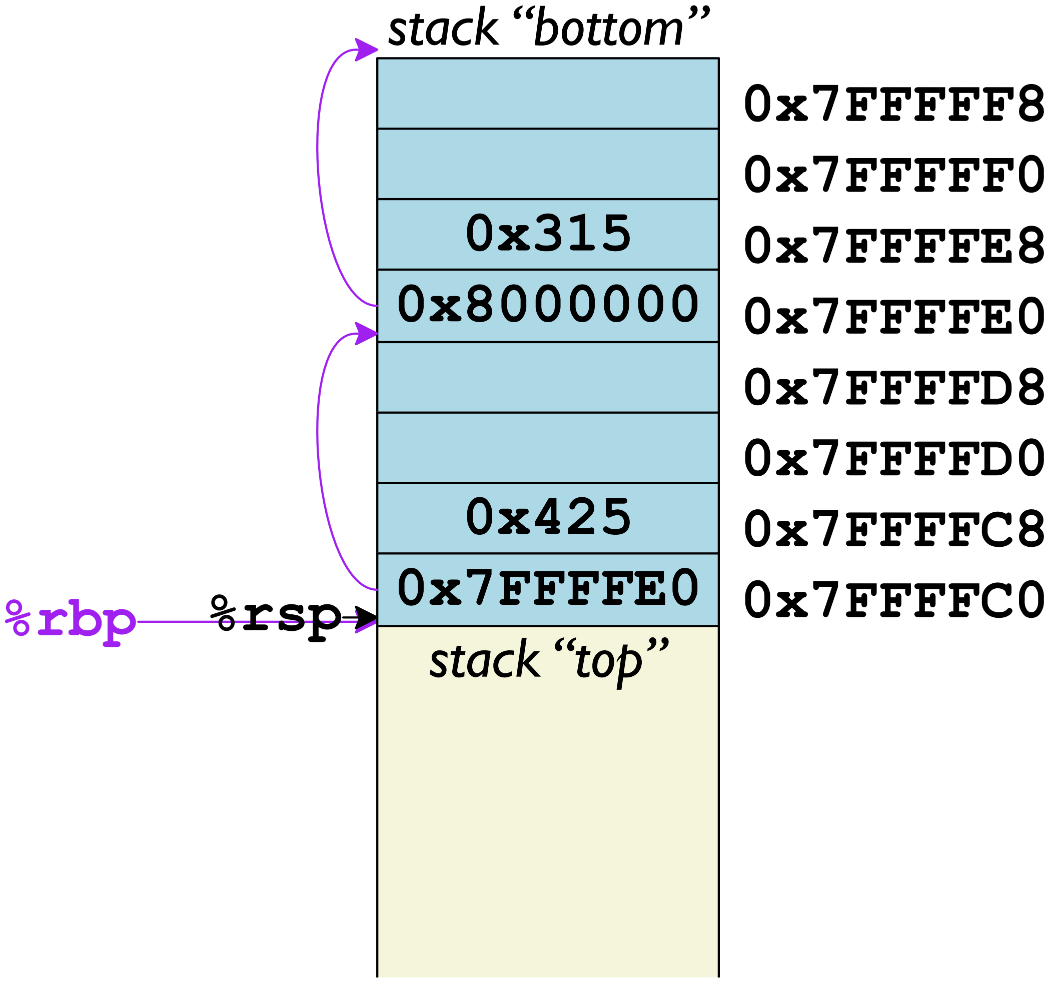


Using a Frame Pointer

```
        ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```



stack "bottom"

	0x7FFFFF8
	0x7FFFFF0
0x315	0x7FFFFE8
0x8000000	0x7FFFFE0
	0x7FFFFD8
	0x7FFFFD0
0x425	0x7FFFFC8
0x7FFFFE0	0x7FFFFC0

%rbp

%rsp

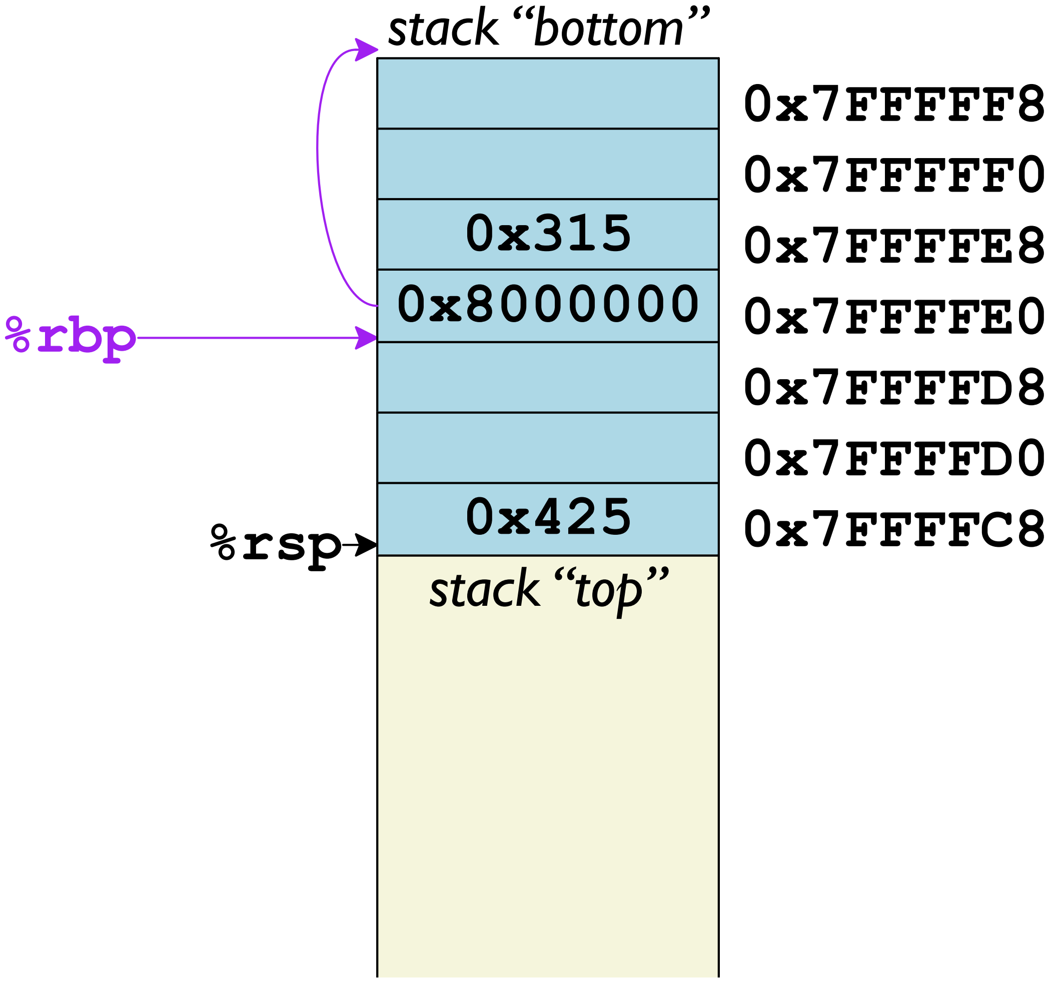
stack "top"

Using a Frame Pointer

```
       ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

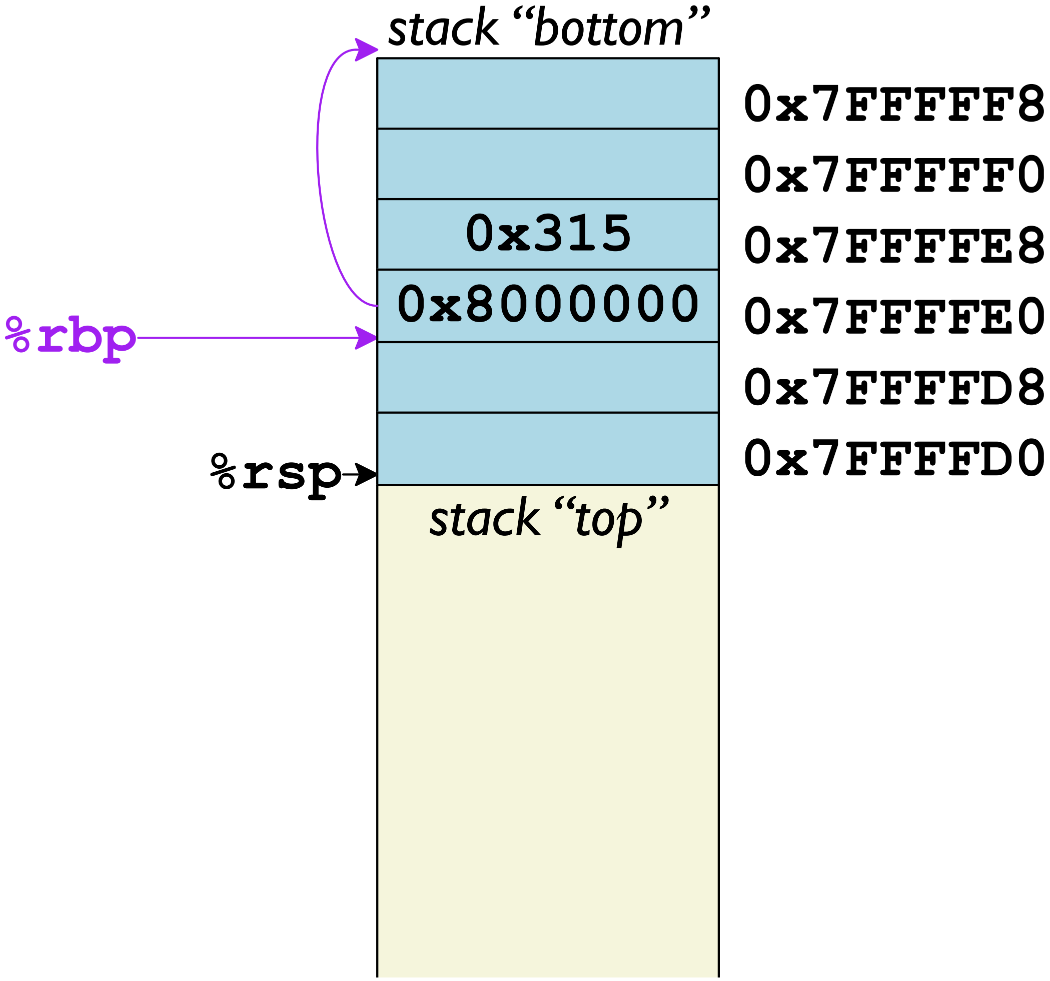


Using a Frame Pointer

```
        ....
0x310:  callq 0x400
0x315:  ....
```

```
0x400:  pushq %rbp
0x401:  movq  %rsp, %rbp
0x404:  ...
        ...-0x8(%rbp)...
        ...
0x420:  callq 0x500
0x425:  ....
0x430:  popq %rbp
0x431:  retq
```

```
0x500:  pushq %rbp
0x501:  movq  %rsp, %rbp
...
0x509:  popq %rbp
0x510:  retq
```

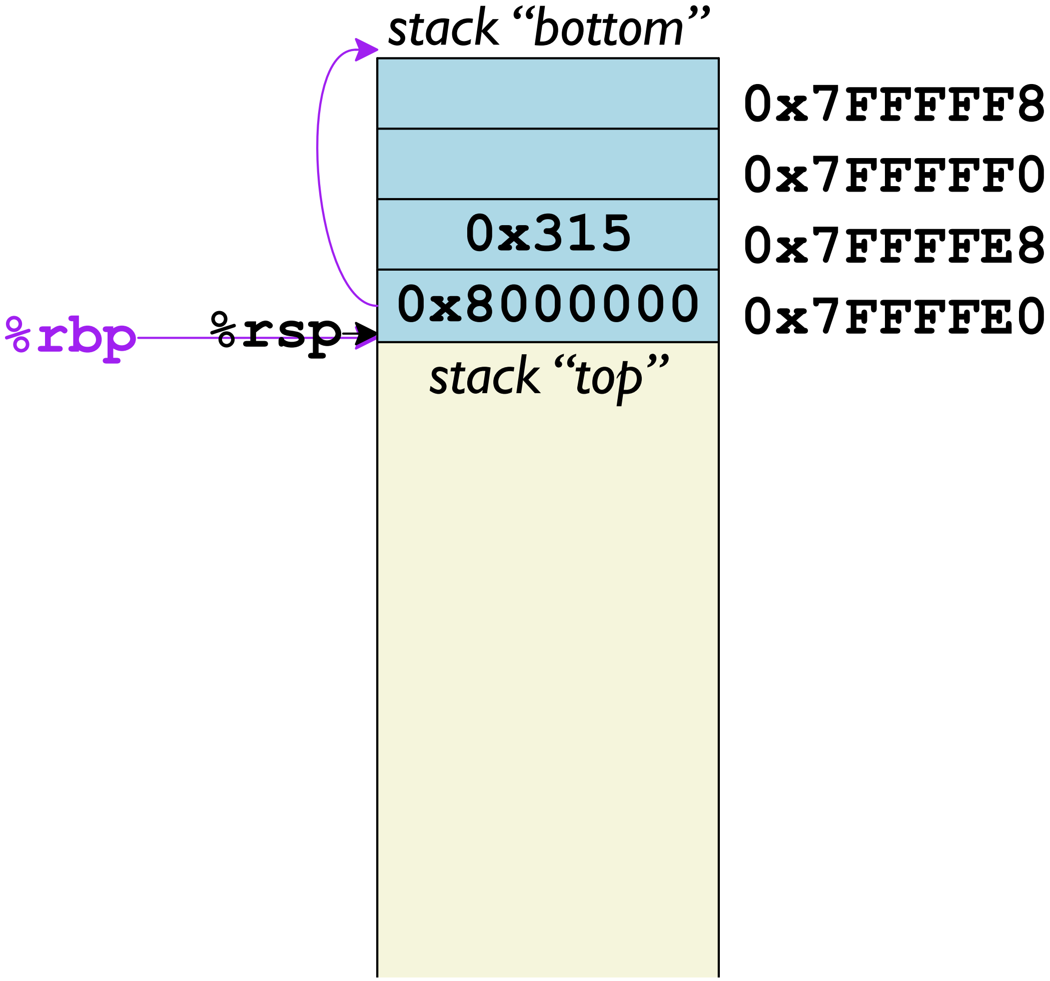


Using a Frame Pointer

```
      ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```

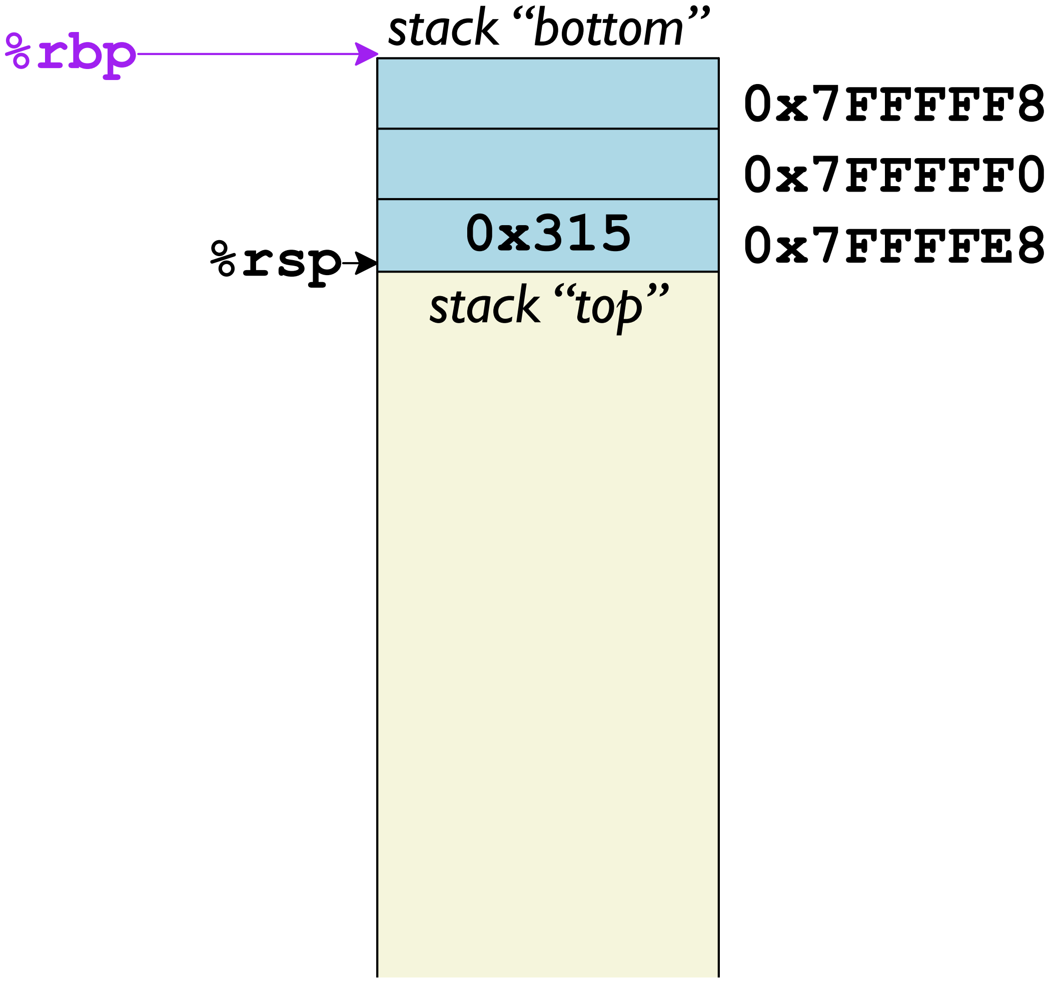


Using a Frame Pointer

```
       ....
0x310: callq 0x400
0x315: ....
```

```
0x400: pushq %rbp
0x401: movq  %rsp, %rbp
0x404: ...
       ...-0x8(%rbp)...
       ...
0x420: callq 0x500
0x425: ....
0x430: popq %rbp
0x431: retq
```

```
0x500: pushq %rbp
0x501: movq  %rsp, %rbp
...
0x509: popq %rbp
0x510: retq
```



Avoiding Stack Frames

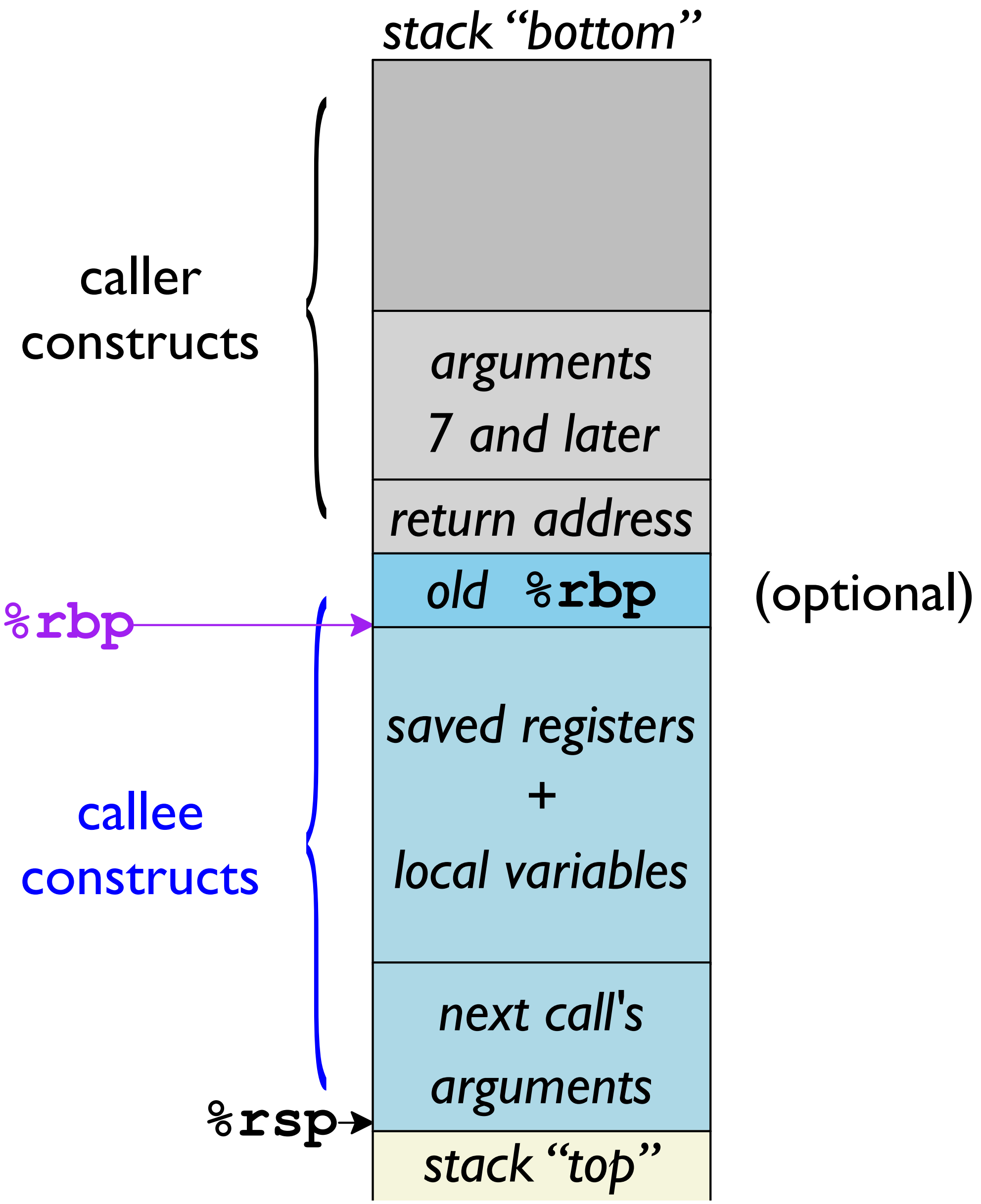
Modern compilers don’t need stack frames

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $15213, 8(%rsp)
  movl     $3000, %esi
  leaq     8(%rsp), %rdi
  call     incr
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  retq
```

“frame is `%rsp` plus 24”

DWARF format communicates from the compiler to the debugger

x86-64 Procedure Summary



Bryant and O'Hallaron, *Computer Systems: A Programmer's Perspective*, Third Edition