Computer Systems

CS107

Cynthia Lee

Today’s Topics

- Function call and return in x86-64
  - Registers
  - Call stack

NEXT TIME:

- NEW topic: the build process
  - Taking a look at each step of the process
  - Preprocessor, compiler, assembler, linker, loader

Function calls in assembly

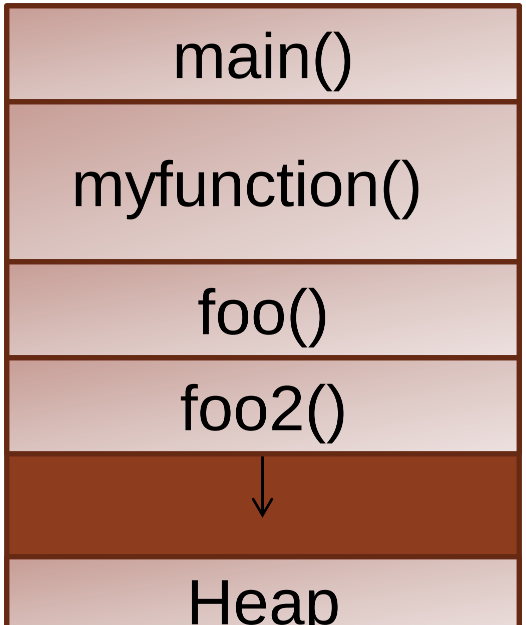
TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

Stanford University

Terminology: “caller” and “callee”

- When talking about function call and return:
  - the function that calls another is known as the “caller”
  - the function that is being called is known as the “callee”

- Of course, a function can simultaneously be a callee and a caller!
  - In using these terms, we just try to be clear for the context which particular caller-callee exchange we are speaking about.



Register state

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

Stanford University

Register state associated with function call and return

Registers (on CPU)

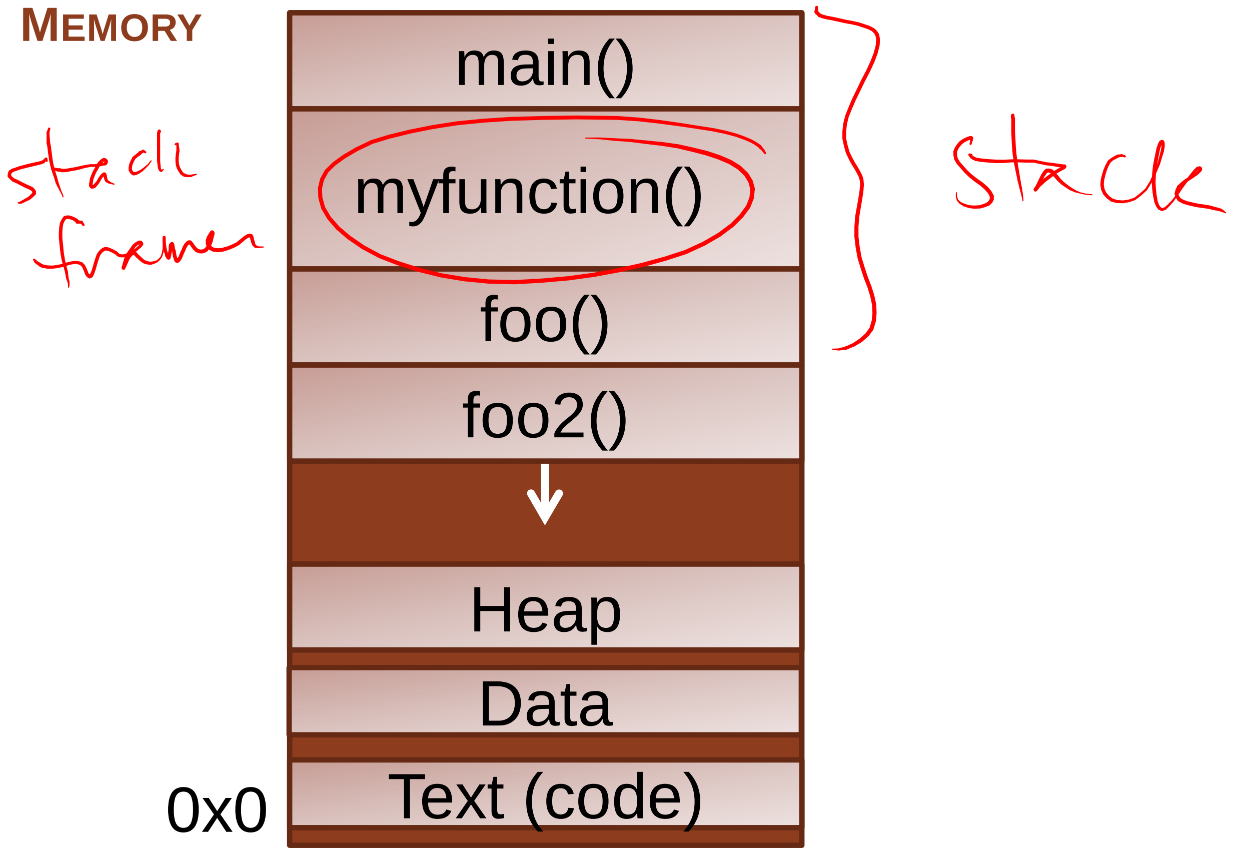
Return value	`%rax`
1st argument	`%rdi`
2nd argument	`%rsi`
3rd argument	`%rdx`
4th argument	`%rcx`
5th argument	`%r8`
6th argument	`%r9`
Stack pointer	`%rsp`
Instruction ptr	`%rip`

If the function takes more than 6 arguments, the extras are stored on the stack (in memory not registers)

Memory state

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

Reminder: what is a stack frame?

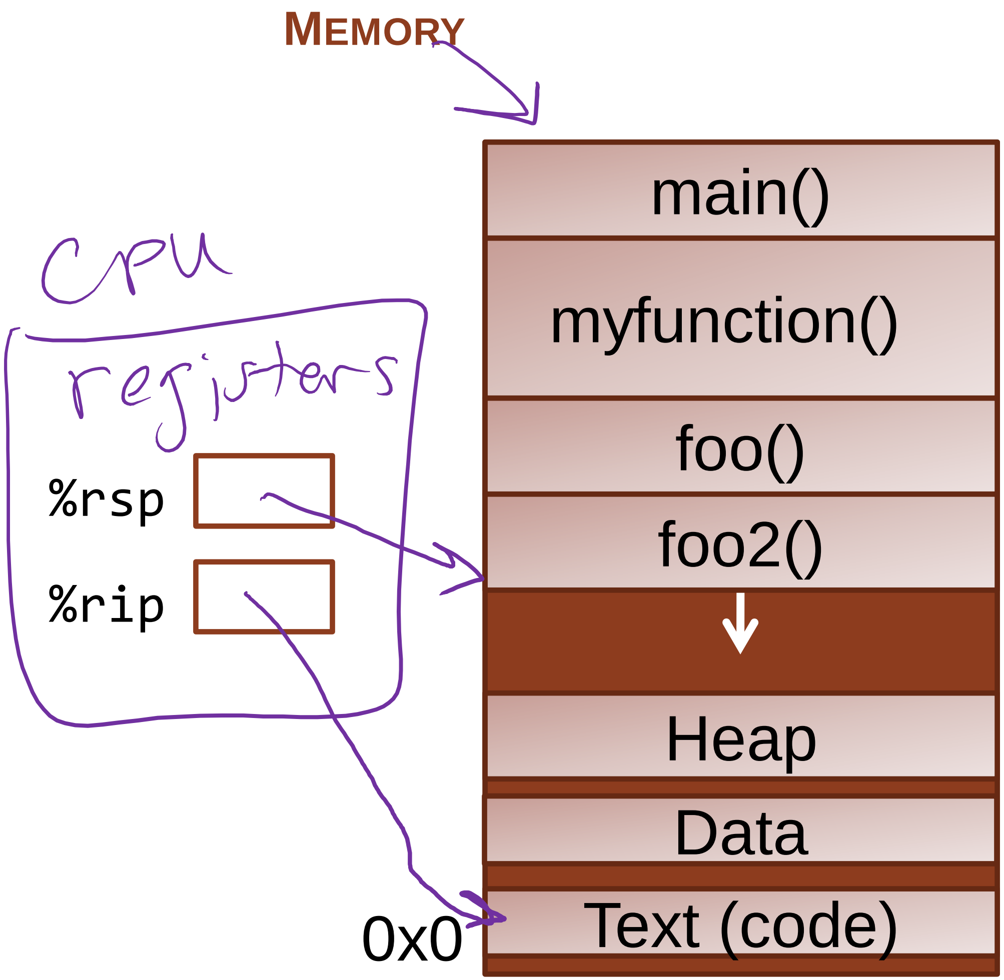


Putting it together: registers and memory

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

Stanford University

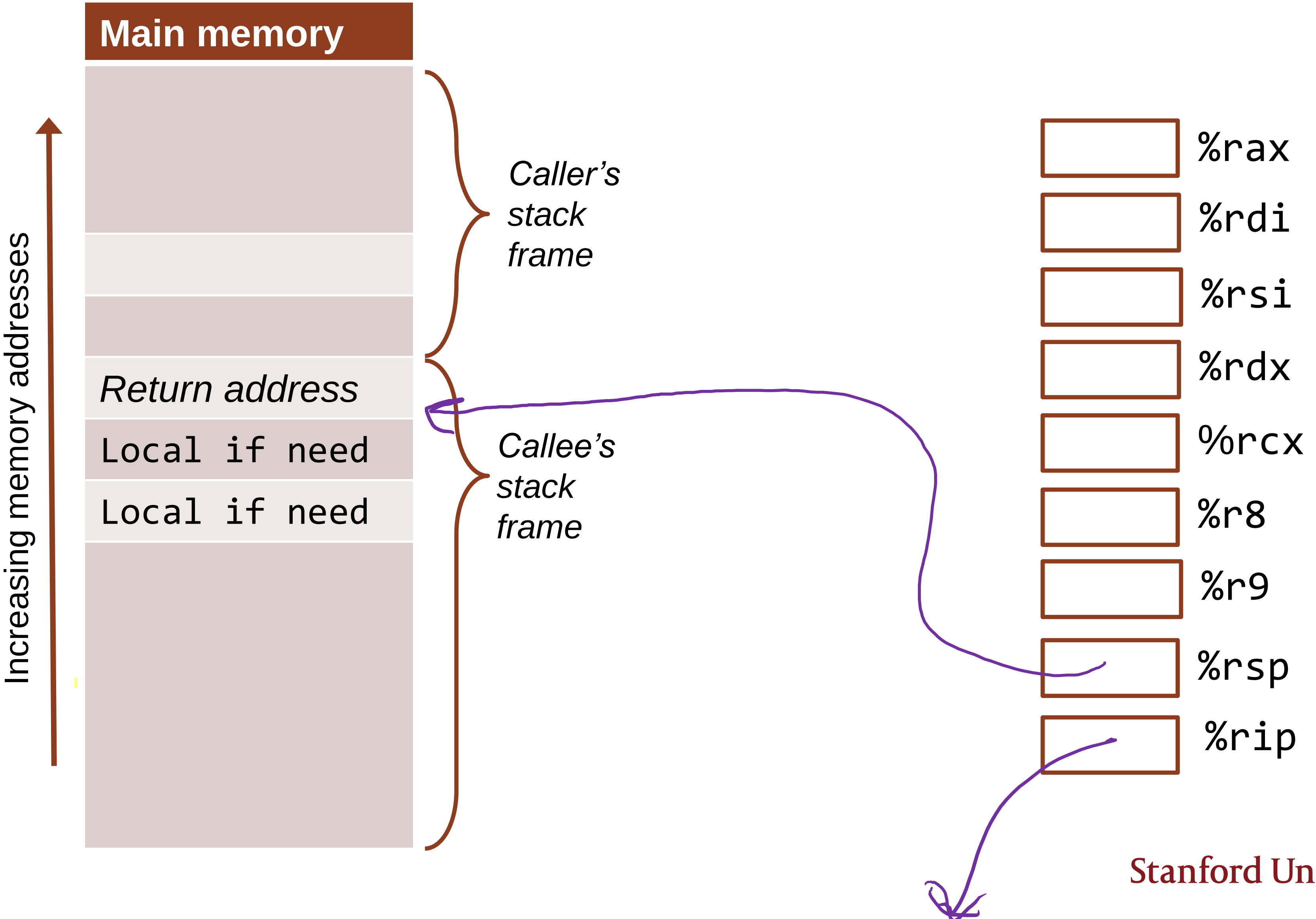
Your turn: RSP and RIP roles



Where, generally, do rsp and rip point?

A. rsp and rip both point to the stack
B. rsp points to the stack and rip points to the heap
[✓ C. Something else]

Typical stack frame layout and functions’ register use



Main memory

Increasing memory addresses

Return address

`Local if need`

`Local if need`

Caller’s stack frame

Callee’s stack frame

`%rax`

`%rdi`

`%rsi`

`%rdx`

`%rcx`

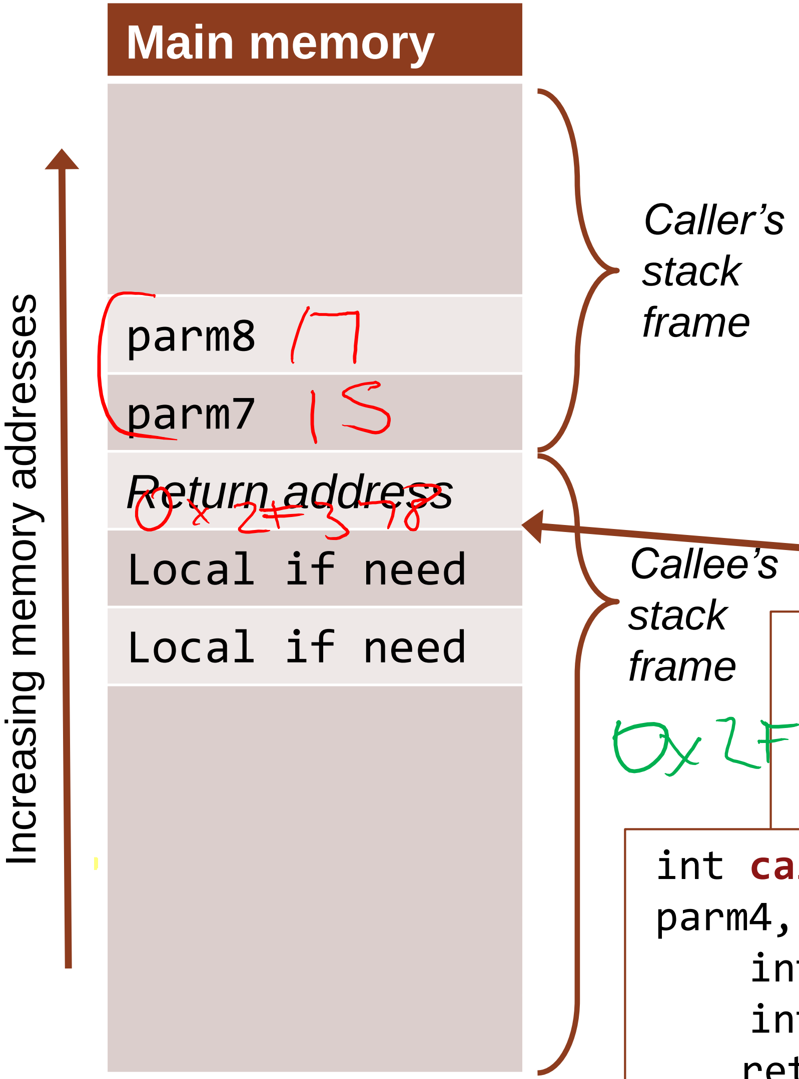
`%r8`

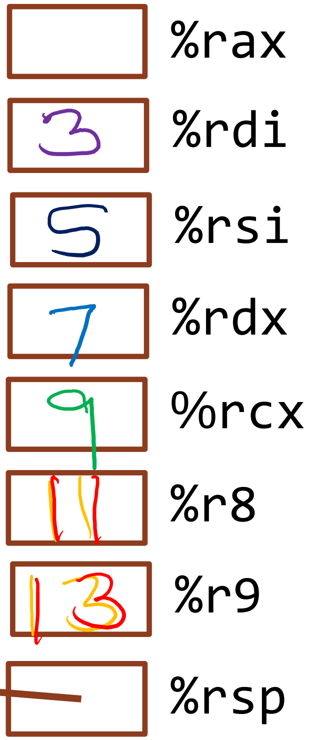
`%r9`

`%rsp`

`%rip`

Activity: fill in values





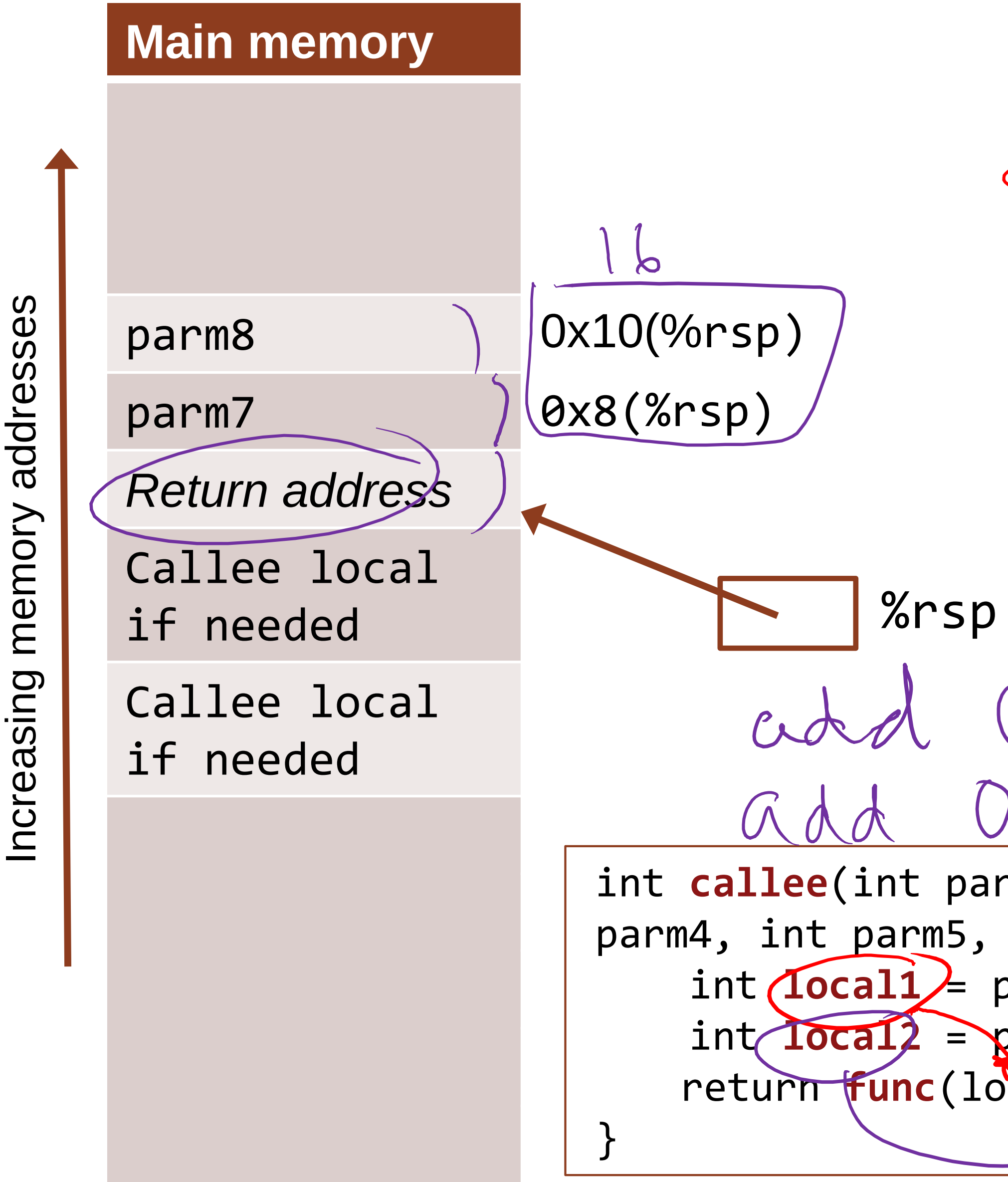
1 2 3 4 5 6

```
int caller() {
    int x = callee(3, 5, 7, 9, 11, 13, 15, 17);
    x++;
    return x;
```

addl $1, %eax

```
int callee(int parm1, int parm2, int parm3, int
parm4, int parm5, int parm6, int parm7, int parm8) {
     int local1 = parm1 + param2 + parm3 + parm4;
     int local2 = parm5 + parm6 + parm7 + parm8;
    return func(local1, local2);
}
```

How we address typical stack frame layout



Main memory

Increasing memory addresses

parm8

parm7

Return address

Callee local if needed

Callee local if needed

%rsp

16

0x10(%rsp)

0x8(%rsp)

why?

```
<callee>:
add     %esi,%edi
add     %edx,%edi
add     %ecx,%edi
add     %r9d,%r8d
mov     %r8d,%esi
(secret for now)
callq   4006d0 <func>
repz retq
```

p1+p2 +p3 +p4

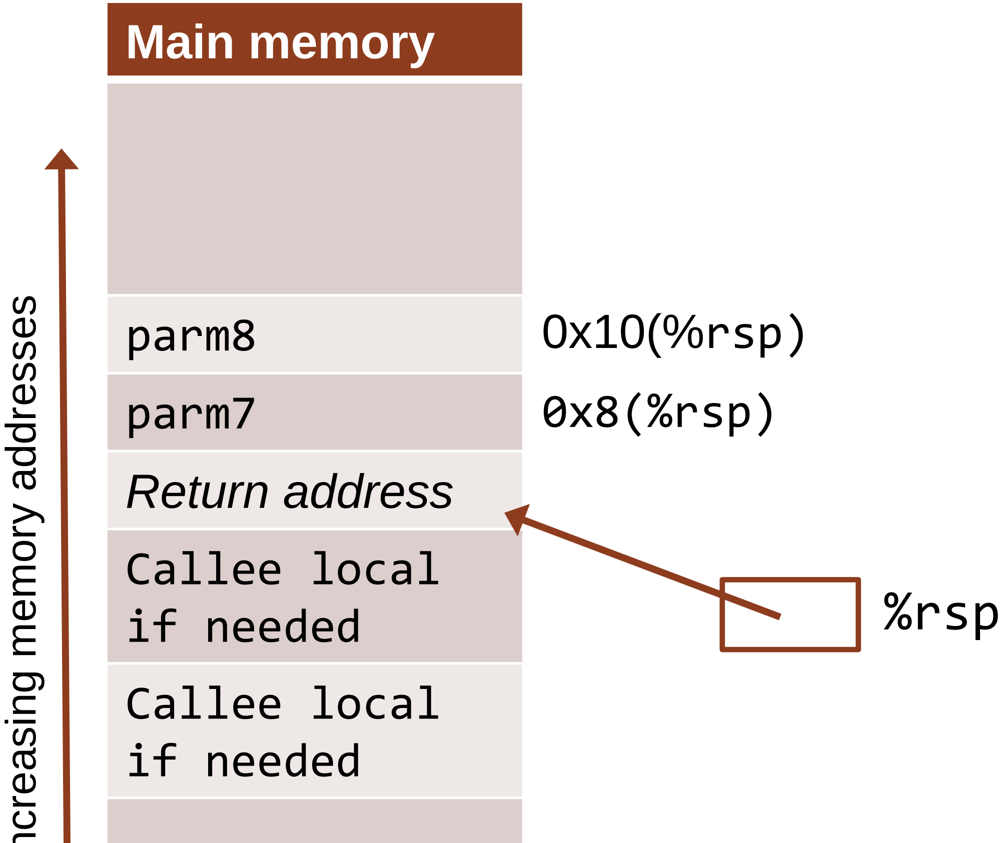
p5+p6 p

add 0x8(%rsp),%esi

add 0x10(%rsp),%esi

```
int callee(int parm1, int parm2, int parm3, int
parm4, int parm5, int parm6, int parm7, int parm8) {
    int local1 = parm1 + param2 + parm3 + parm4;
    int local2 = parm5 + parm6 + parm7 + parm8;
   return func(local1, local2);
}
```

How we address typical stack frame layout



Main memory

parm8 0x10(%rsp)

parm7 0x8(%rsp)

Return address

Callee local if needed

Callee local if needed

%rsp

Increasing memory addresses

```
<callee>:
add     %esi,%edi
add     %edx,%edi
add     %ecx,%edi
add     %r9d,%r8d
mov     %r8d,%esi
add     0x8(%rsp),%esi
add     0x10(%rsp),%esi
callq   4006d0 <func>
repz retq
```

```
int callee(int parm1, int parm2, int parm3, int
parm4, int parm5, int parm6, int parm7, int parm8) {
    int local1 = parm1 + param2 + parm3 + parm4;
    int local2 = parm5 + parm6 + parm7 + parm8;
   return func(local1, local2);
}
```

Caller-saved registers

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

Stanford University

Register usage: caller-saved and callee-saved

- There is only one copy of each register on the hardware
  - **Not** the case that each function call or stack frame has their own copy!

- So if you write something to %rax, you write to the %rax that EVEYRONE (in particular all other functions on the stack) sees

- If you write something to %rdi, you write to the %rdi that EVERYONE (in particular all other functions on the stack) sees

- To prevent functions from trashing each others' registers, we have **caller-saved and callee-saved register usage conventions**
  - A sort of etiquette for how to use registers in functions

Register usage: caller-saved and callee-saved

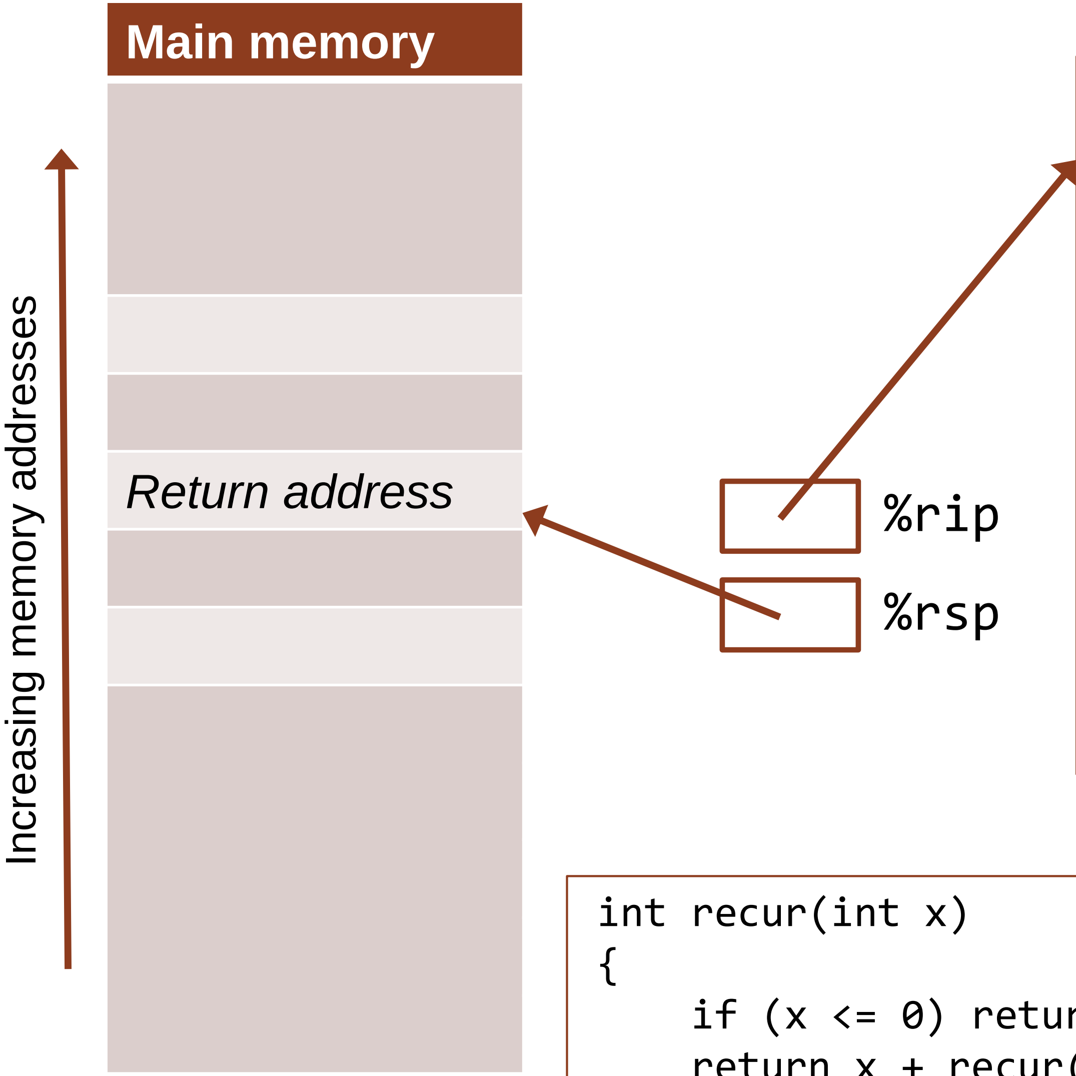
- **Caller-saved**: if you are the caller about to call another function, and you care about keeping the value of a register that is designated as "caller-saved" intact, you'd better copy that value elsewhere before making the function call.
  - It is not guaranteed that the value will be preserved by the callee!
  - Your caller-saved register could be ruined by the callee!
  - (If you are the callee, feel free to trash this register.)

- **Callee-saved**: if you are the callee about to change the value of a register that is designated as "callee-saved," you'd better copy that value elsewhere before changing the register value, and then restore the value from your saved copy before you return.
  - Callee must guarantee that the value is preserved (either unchanged, or at least restored to original state before returning).
  - (If you are the caller, feel free to not save a copy of the register before calling a function, it's guaranteed to be there for you safe and sound when the callee function returns!)

Saving backup copies of registers to the stack (memory) using push and pop

- To save caller-saved registers, we often use the stack (in memory, not registers)
- Two instructions help with this:
- `push op1`
  - Take the value op1 and store it to the next free slot on the stack (push onto the stack); adjust the `%rsp` to show that the stack now extends lower than before because it has one more item
- `pop op1`
  - Take the topmost (most recent) element on the stack and pop it off the stack, storing it into `op1`; adjust the `%rsp` to show that the stack now has one fewer item

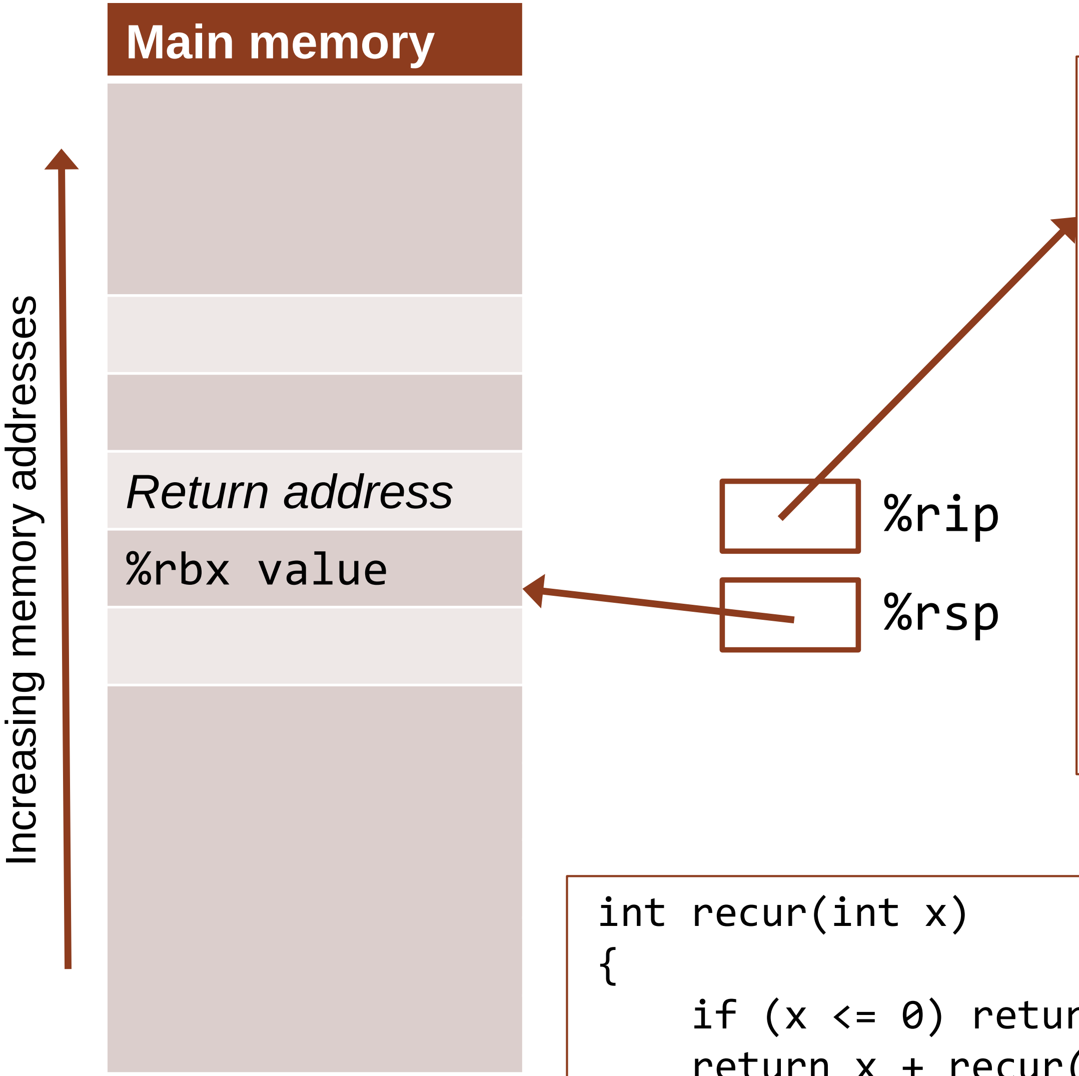
Saving caller-saved values using push/pop



```
<recur>:
push    %rbx
mov     %edi,%ebx
test    %edi,%edi
jle     <recur+0x13>
lea     -0x1(%rdi),%edi
callq   40060d <recur>
add     %ebx,%eax
jmp     <recur+0x18>
mov     $0x0,%eax
pop     %rbx
retq
```

```
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```

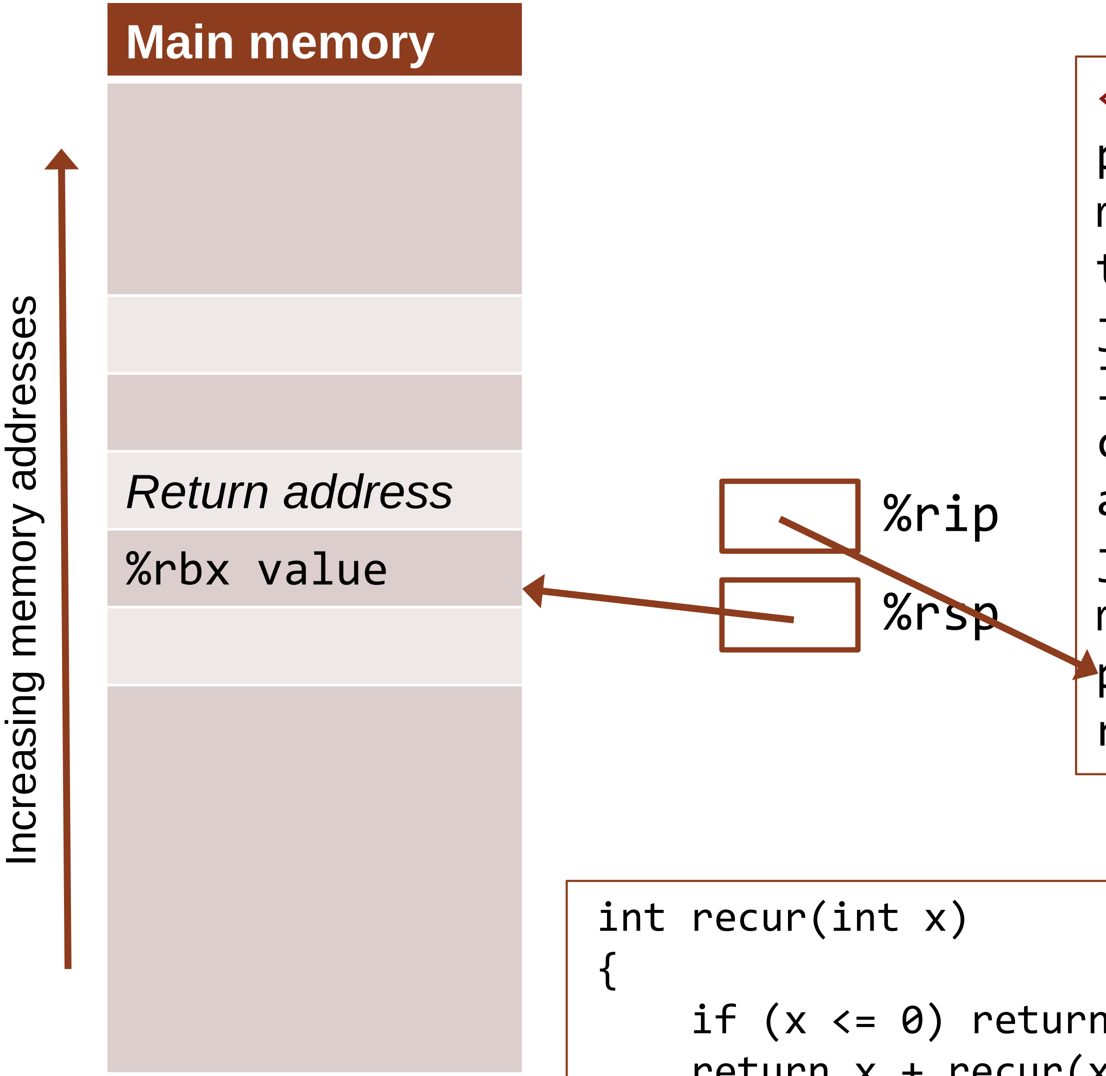
Saving caller-saved values using push/pop



```
<recur>:
push    %rbx
mov     %edi,%ebx
test    %edi,%edi
jle     <recur+0x13>
lea     -0x1(%rdi),%edi
callq   40060d <recur>
add     %ebx,%eax
jmp     <recur+0x18>
mov     $0x0,%eax
pop     %rbx
retq
```

```
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```

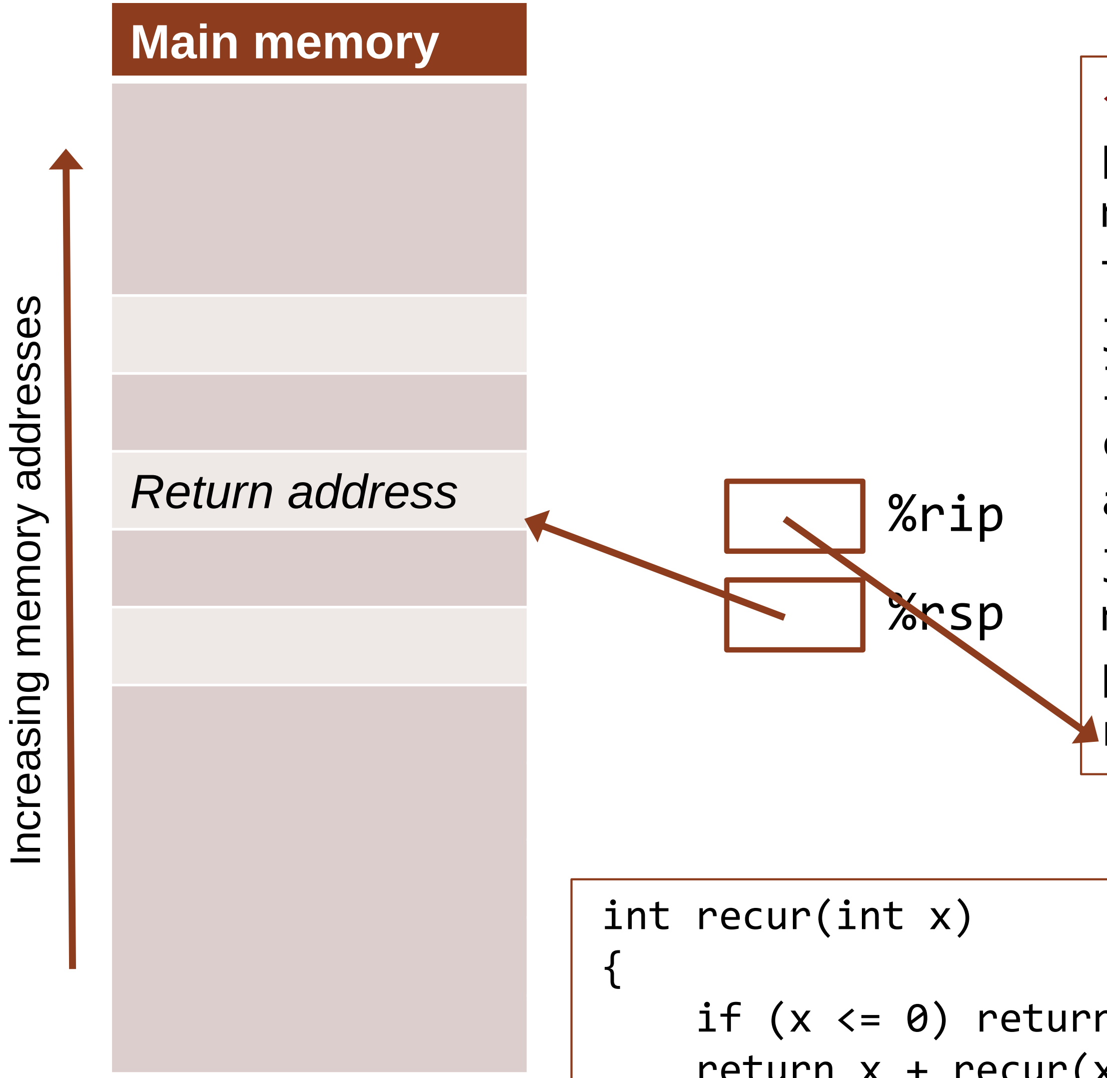
Saving caller-saved values using push/pop



```
<recur>:
push    %rbx
mov     %edi,%ebx
test    %edi,%edi
jle     <recur+0x13>
lea     -0x1(%rdi),%edi
callq   40060d <recur>
add     %ebx,%eax
jmp     <recur+0x18>
mov     $0x0,%eax
pop     %rbx
retq
```

```
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```

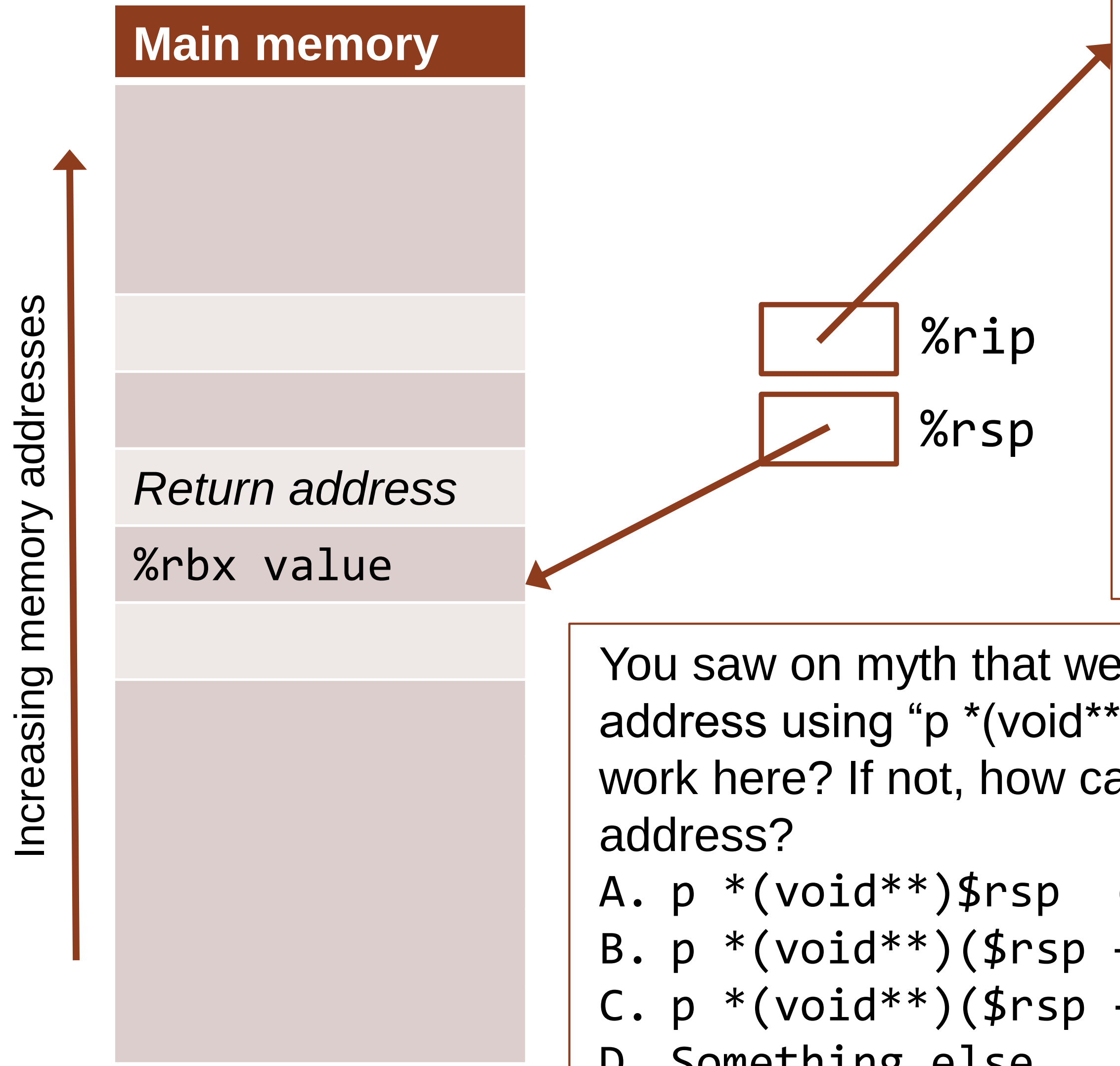
How we address typical stack frame layout



```
<recur>:
push    %rbx
mov     %edi,%ebx
test    %edi,%edi
jle     <recur+0x13>
lea     -0x1(%rdi),%edi
callq   40060d <recur>
add     %ebx,%eax
jmp     <recur+0x18>
mov     $0x0,%eax
pop     %rbx
retq
```

```
int recur(int x)
{
    if (x <= 0) return 0;
    return x + recur(x-1);
}
```

Your turn: the role of $rsp



```
<recur>:
push    %rbx
mov     %edi,%ebx
test    %edi,%edi
jle     <recur+0x13>
lea     -0x1(%rdi),%edi
callq   40060d <recur>
add     %ebx,%eax
jmp     <recur+0x18>
mov     $0x0,%eax
pop     %rbx
retq
```

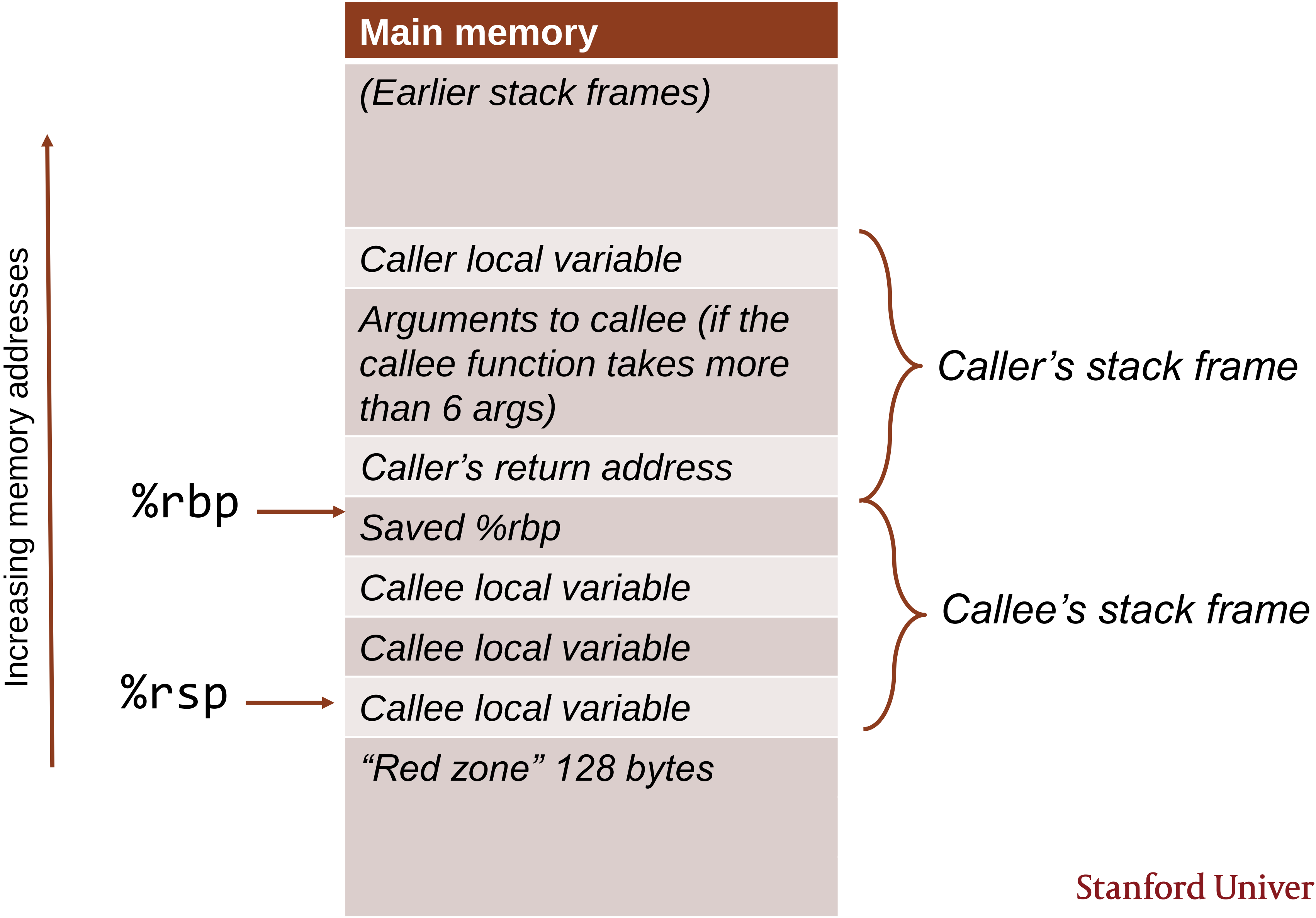
You saw on myth that we typically print the return address using “p *(void**)$rsp” in gdb. Would that work here? If not, how can we print the return address?

A. p *(void**)$rsp (same thing works here)
B. p *(void**)($rsp + 0x8)
C. p *(void**)($rsp – 0x8)
D. Something else

(optional study)
More complex stack frame management

THIS IS A LESS-COMMON WAY OF MANAGING THE STACK UNDER THE NEW X86-64, BUT YOU'LL SOMETIMES SEE IT IN GCC OUTPUT

More complex stack frame layout (with rbp)
For use with complex non-leaf functions



Main memory

(Earlier stack frames)

Caller local variable

Arguments to callee (if the callee function takes more than 6 args)

Caller’s return address

Saved %rbp

Callee local variable

Callee local variable

Callee local variable

“Red zone” 128 bytes

Caller’s stack frame

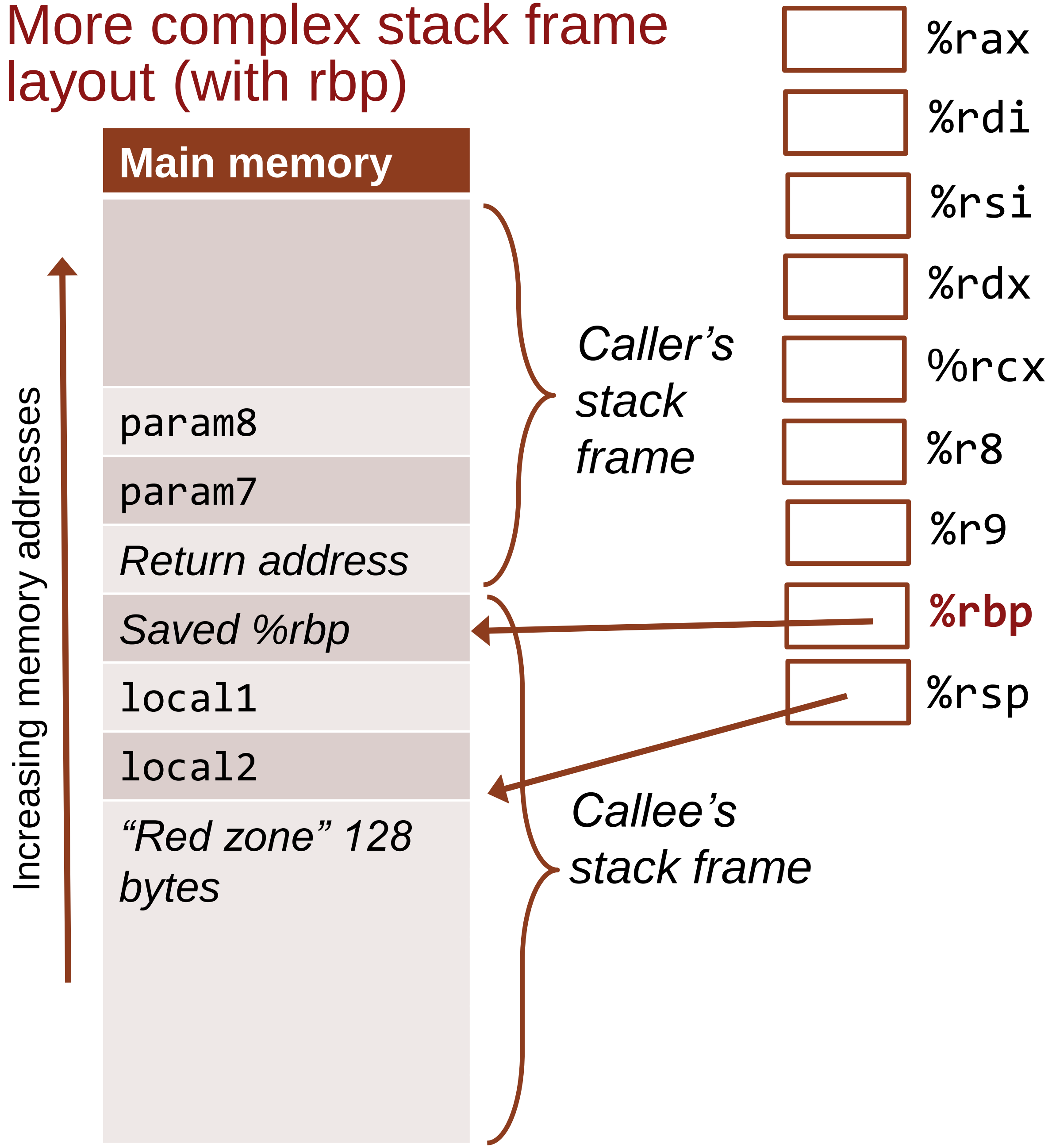
Callee’s stack frame

%rbp

%rsp

Increasing memory addresses

More complex stack frame layout (with rbp)



Main memory

`param8`

`param7`

Return address

Saved %rbp

`local1`

`local2`

“Red zone” 128 bytes

Increasing memory addresses

Caller’s stack frame

Callee’s stack frame

`%rax`

`%rdi`

`%rsi`

`%rdx`

`%rcx`

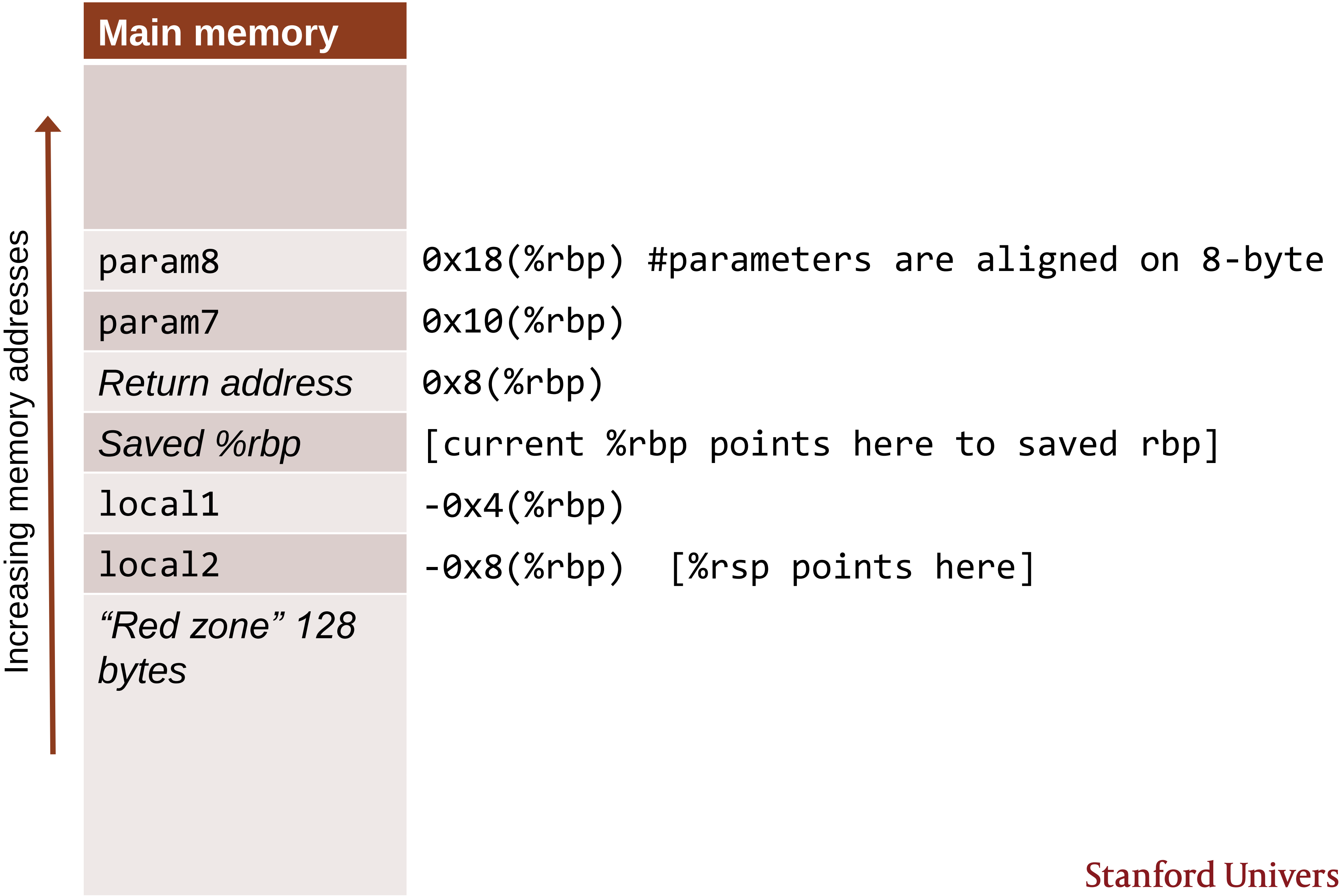
`%r8`

`%r9`

`%rbp`

`%rsp`

How we address the more complex stack frame layout (with rbp)



Main memory

Increasing memory addresses

```
param8          0x18(%rbp) #parameters are aligned on 8-byte
param7          0x10(%rbp)
Return address  0x8(%rbp)
Saved %rbp      [current %rbp points here to saved rbp]
local1          -0x4(%rbp)
local2          -0x8(%rbp)  [%rsp points here]
"Red zone" 128 bytes
```