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

Registers associated with function call and return

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

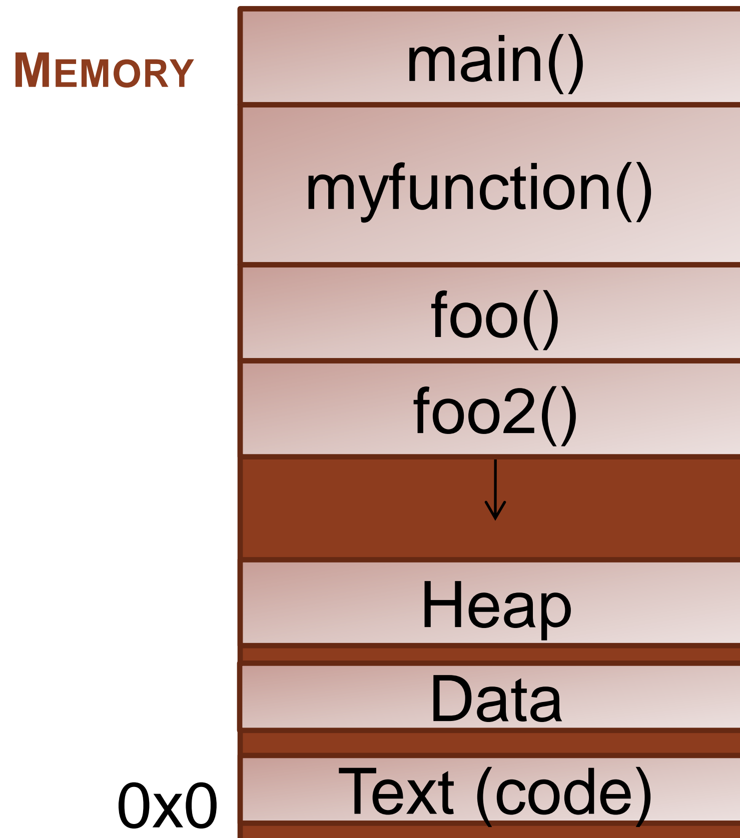
Register state associated with function call and return

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`	
…	…	

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

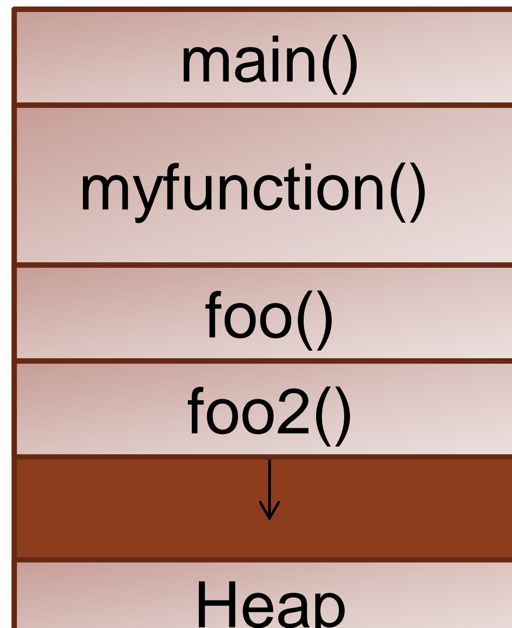
Reminder: what is a stack frame?



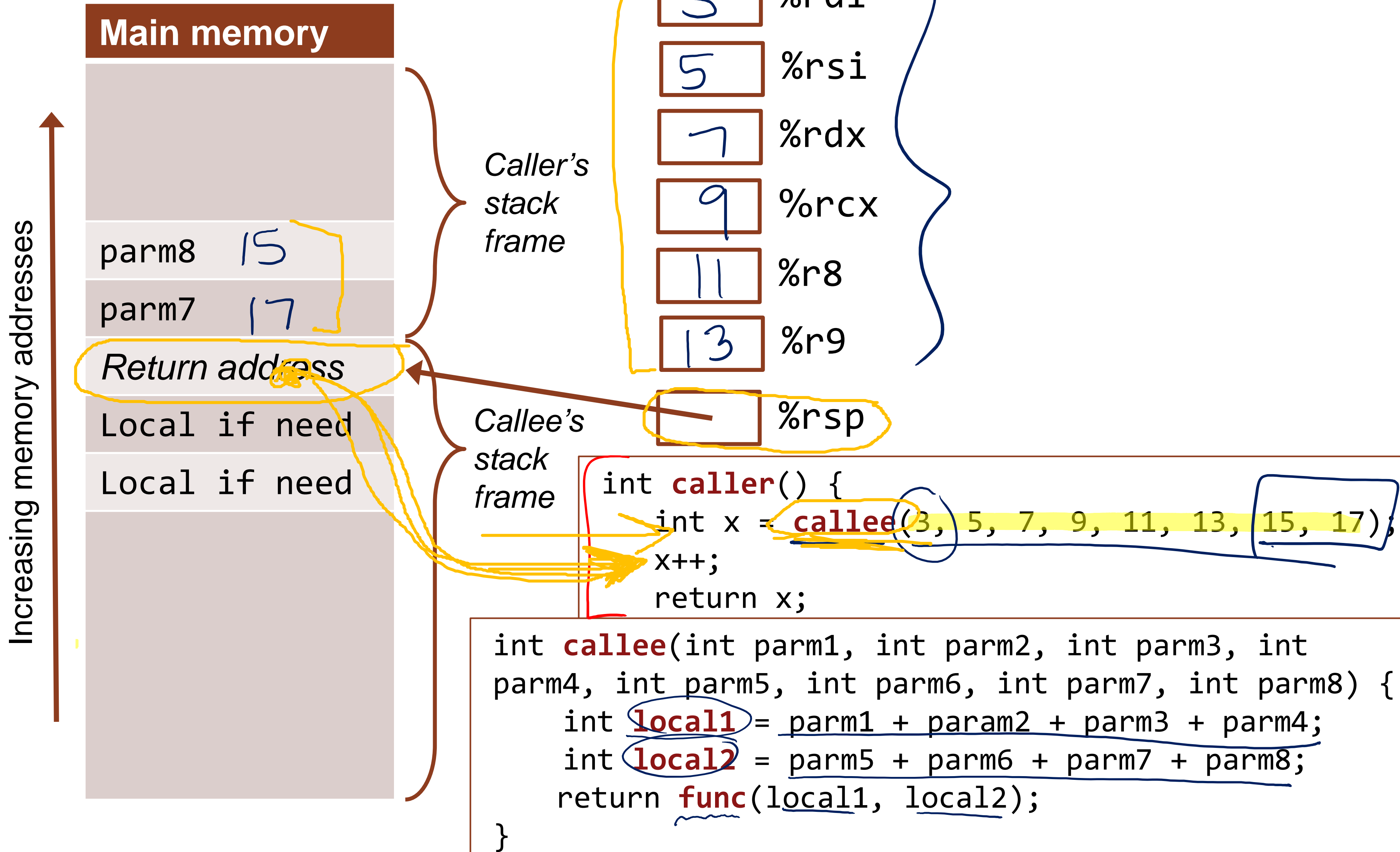
Terminology: “caller” and “callee”

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

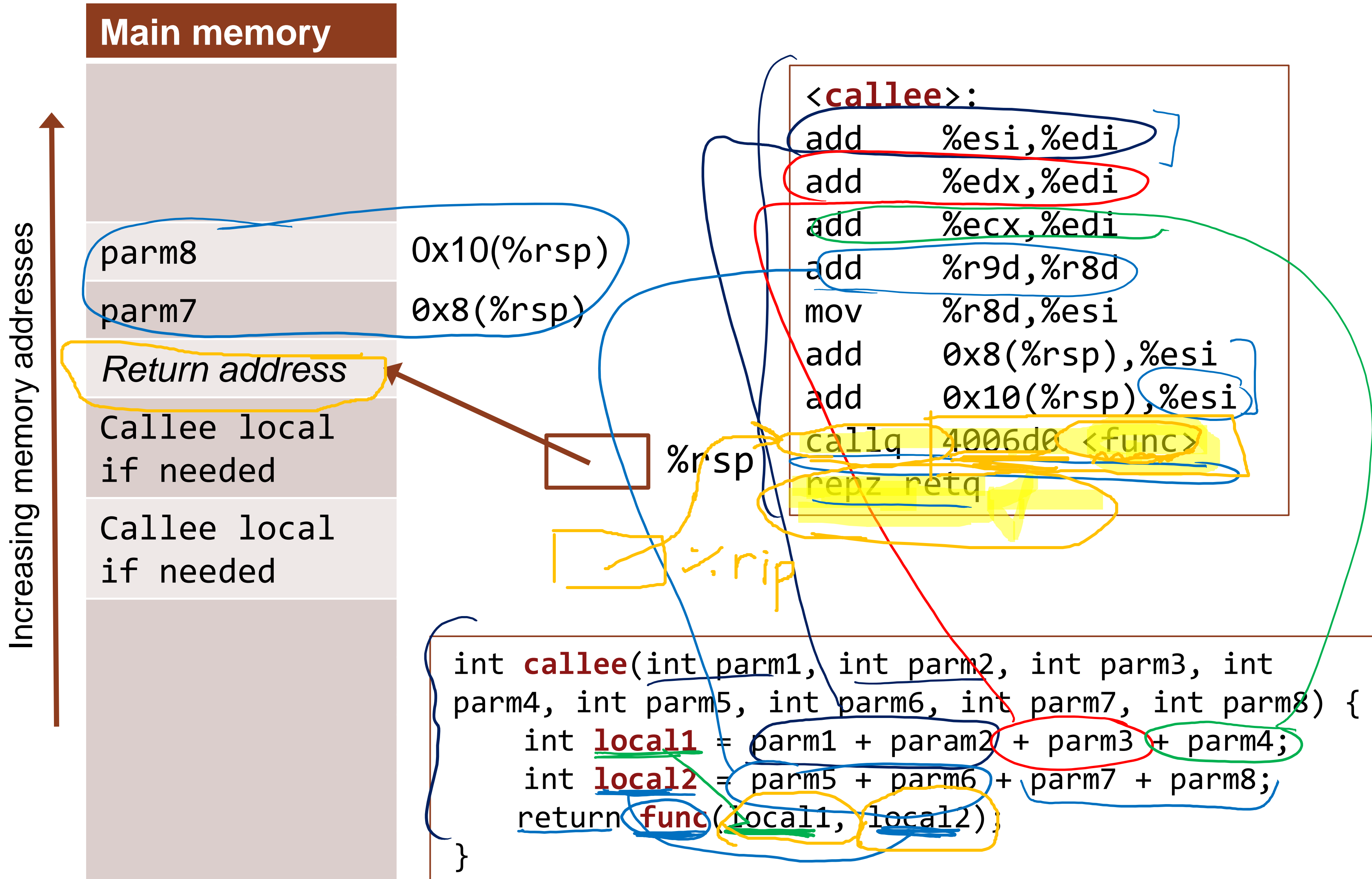
- Of course, a function can simultaneously be a callee and a caller!
  - In using these terms, we just try to be clear for the context which particular caller-callee exchange we are speaking about.



Typical stack frame layout



How we address typical stack frame layout



Caller-saved registers

TOOLS FOR IMPLEMENTING FUNCTION CALL AND RETURN

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

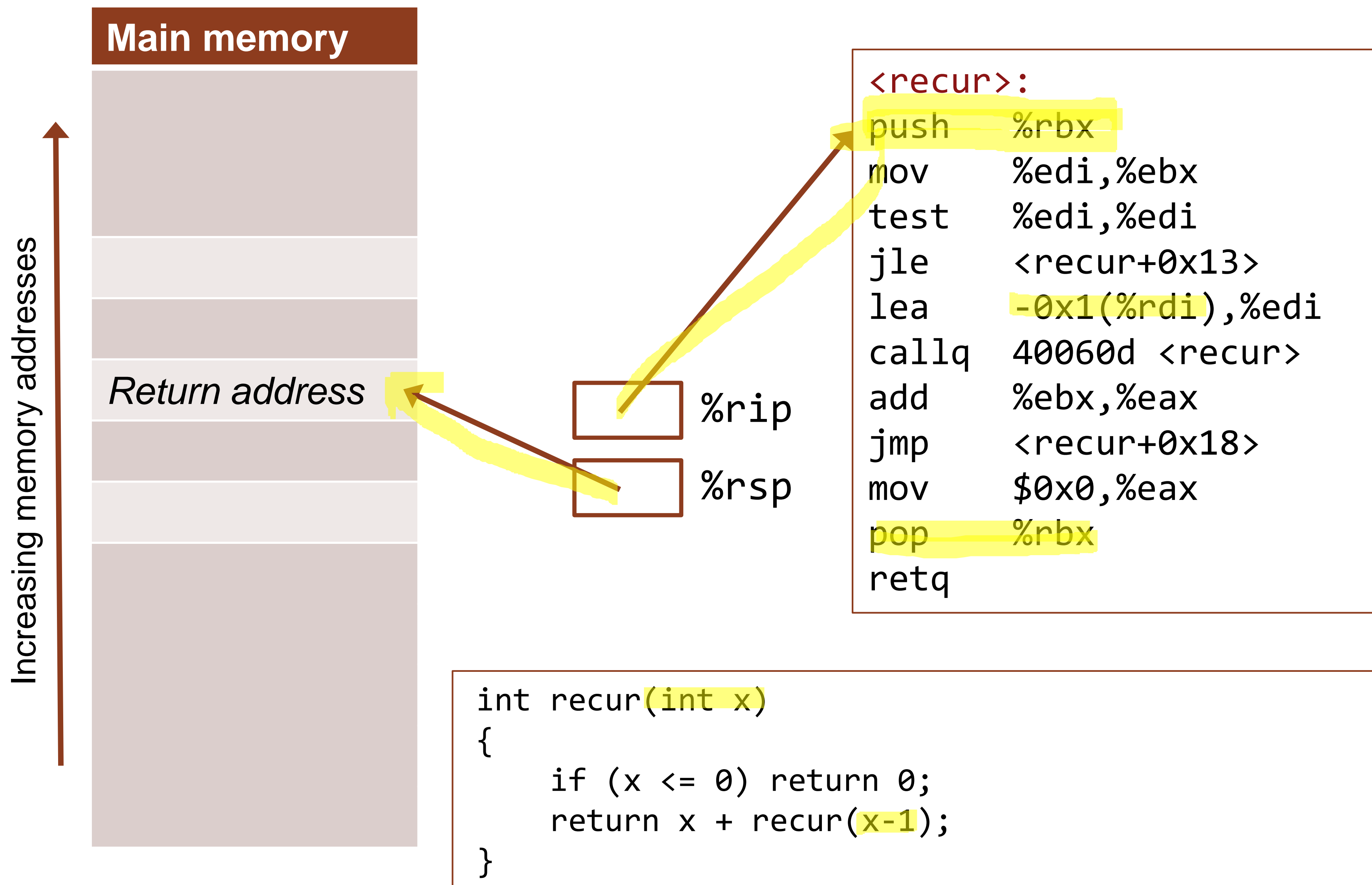
- **Caller-saved**: if you are the caller about to call another function, and you care about keeping the value of a register that is designated as "caller-saved" intact, you'd better copy that value elsewhere before making the function call.
  - It is not guaranteed that the value will be preserved by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

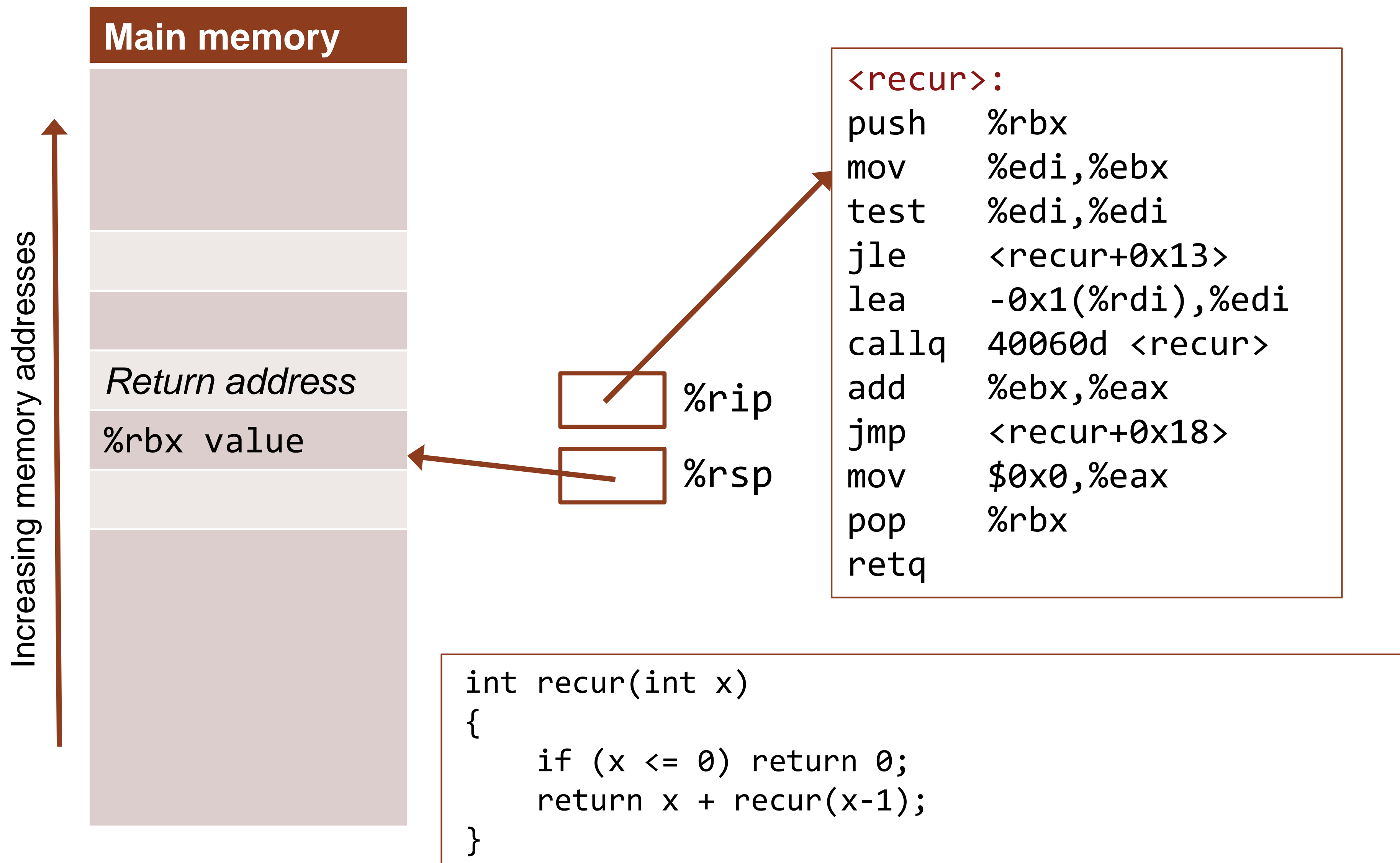
Saving backup copies of registers to the stack (memory) using push and pop

- 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

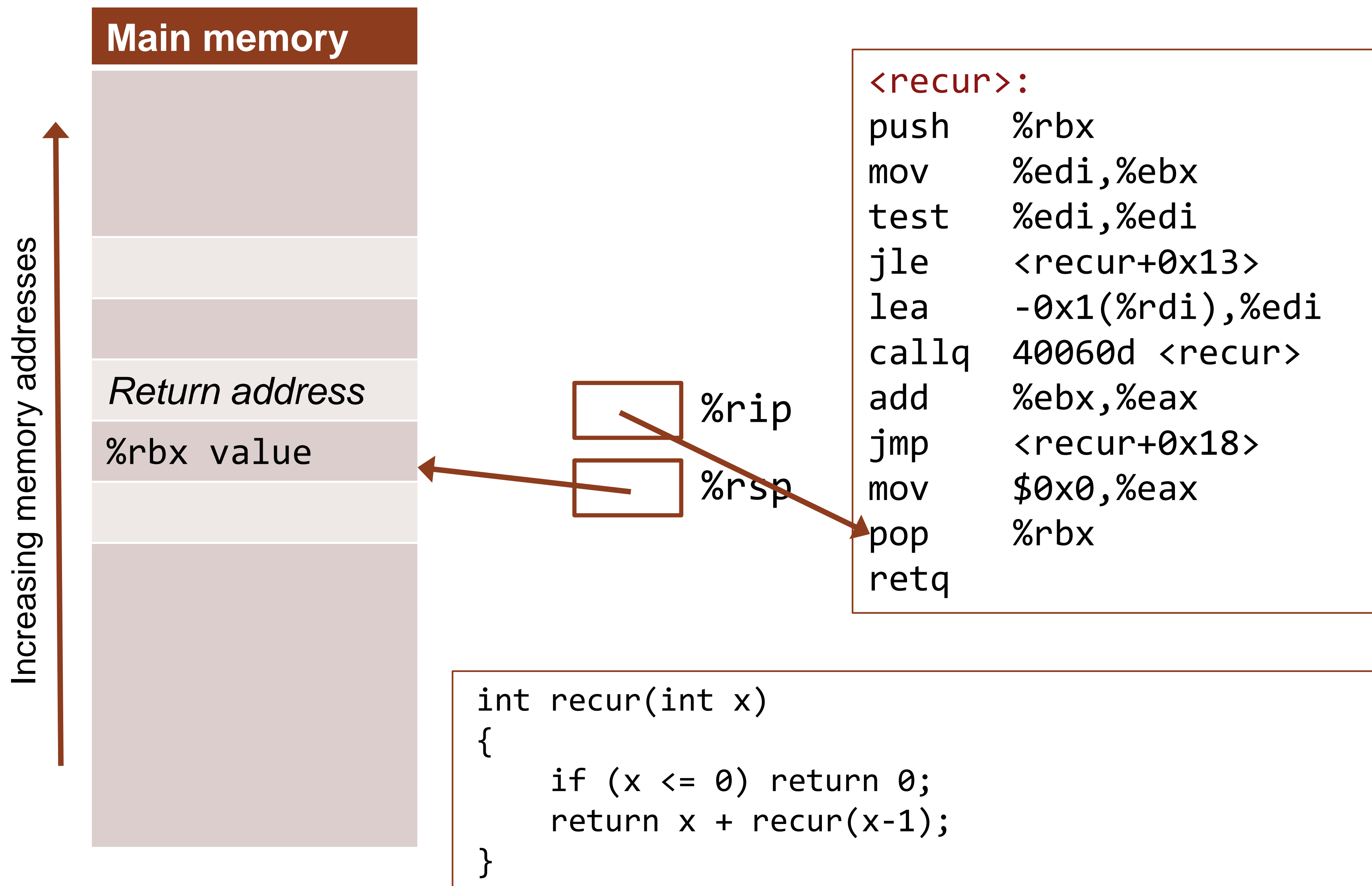
Saving caller-saved values using push/pop



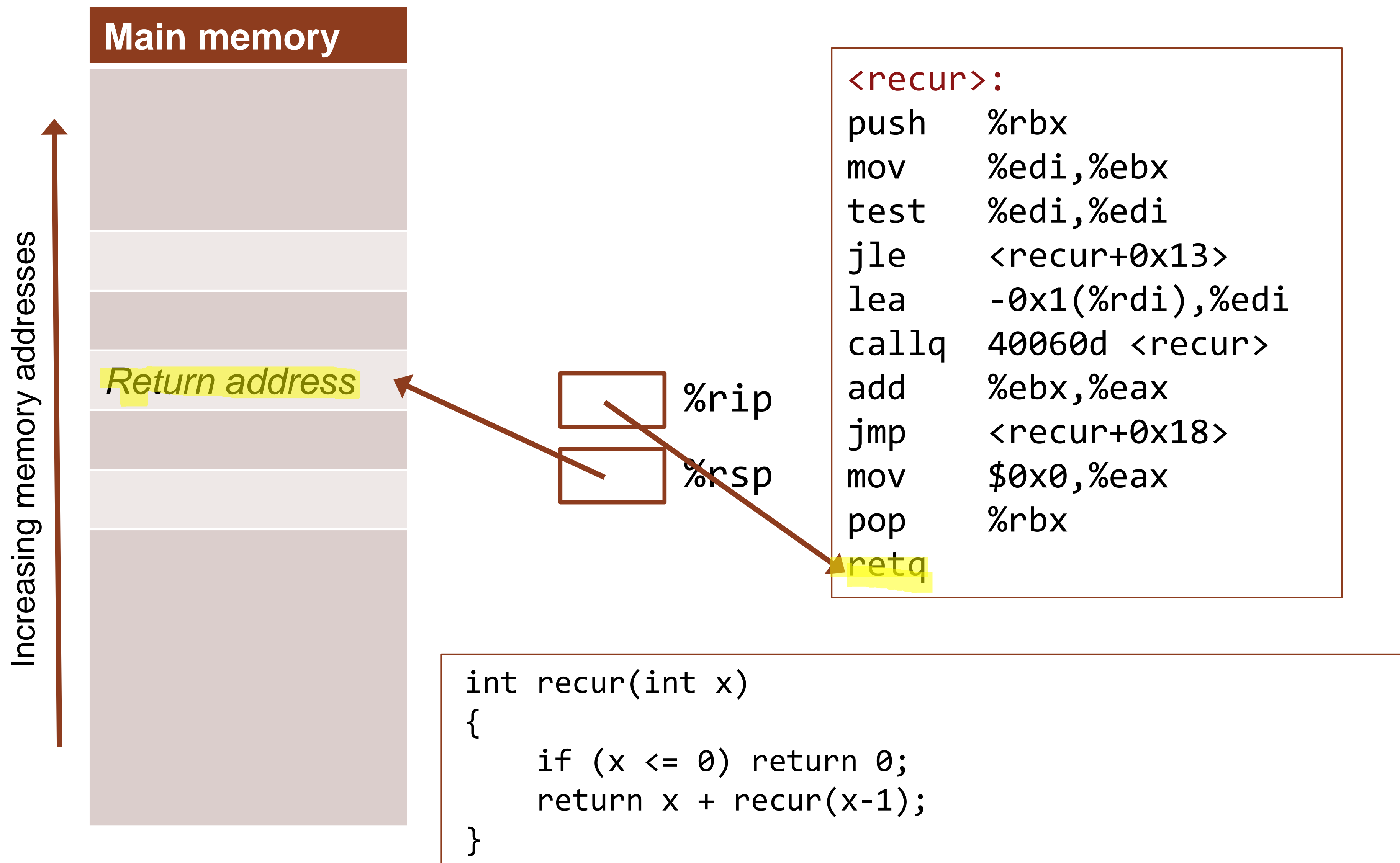
Saving caller-saved values using push/pop



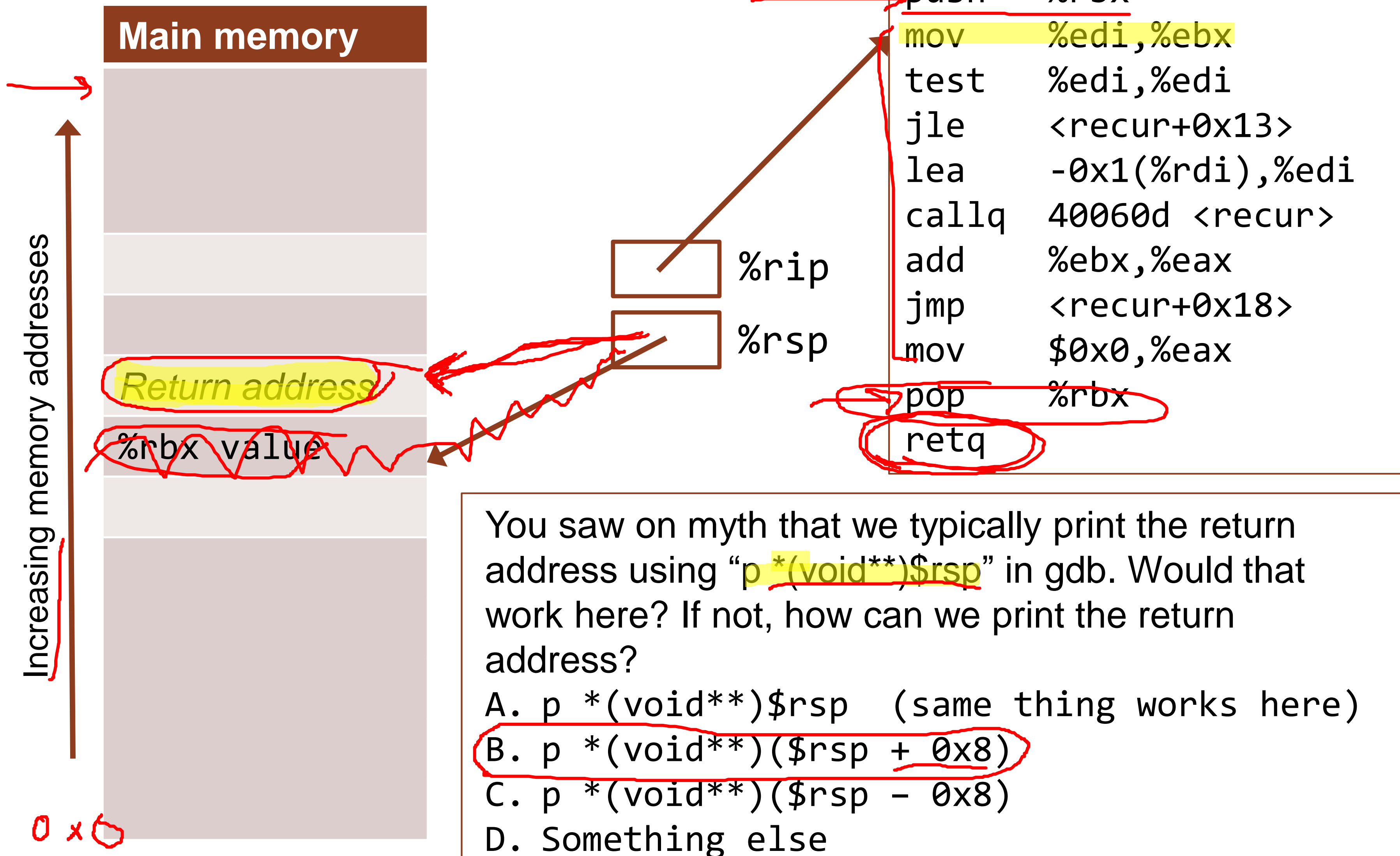
Saving caller-saved values using push/pop



How we address typical stack frame layout



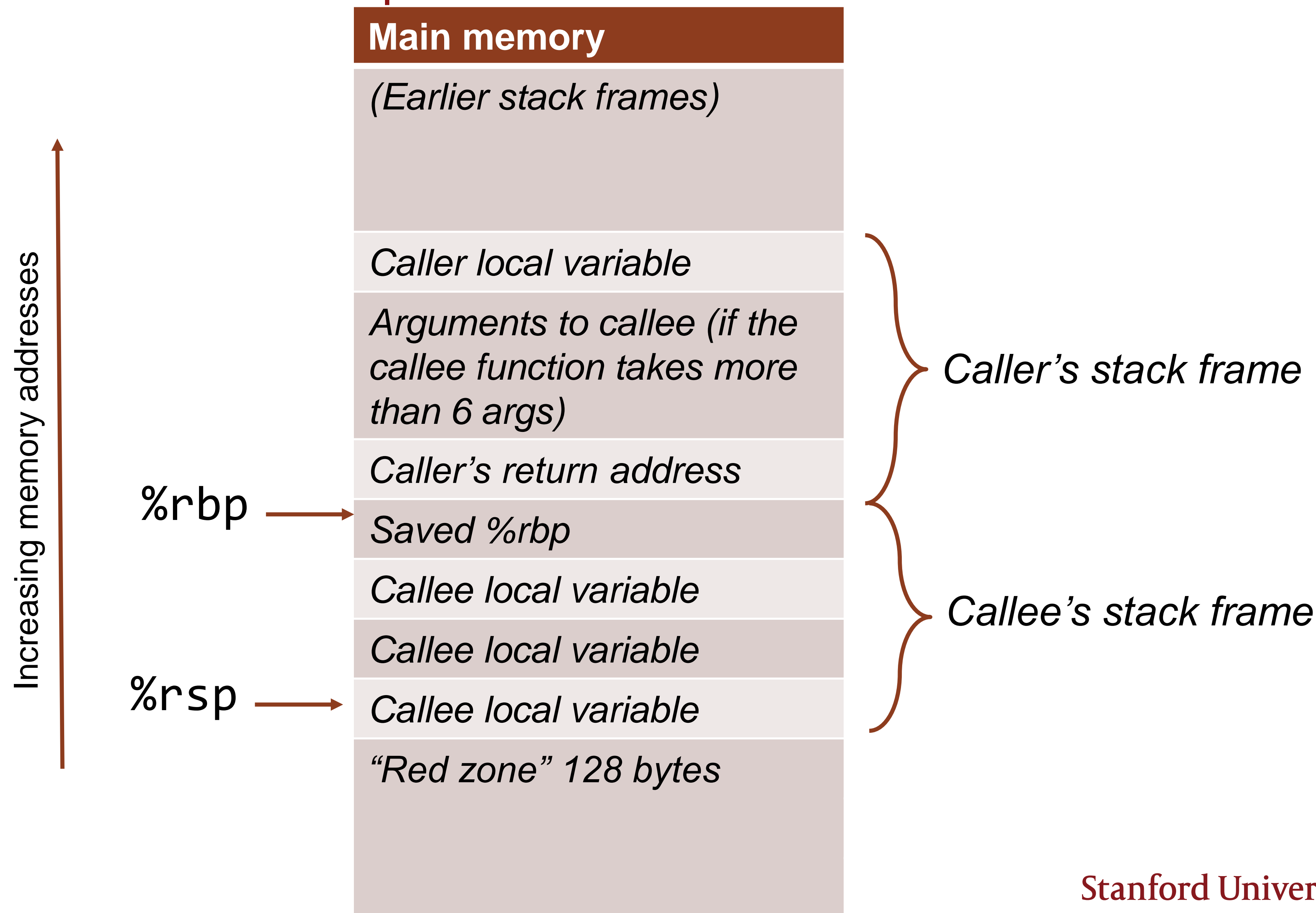
Your turn: the role of $rsp



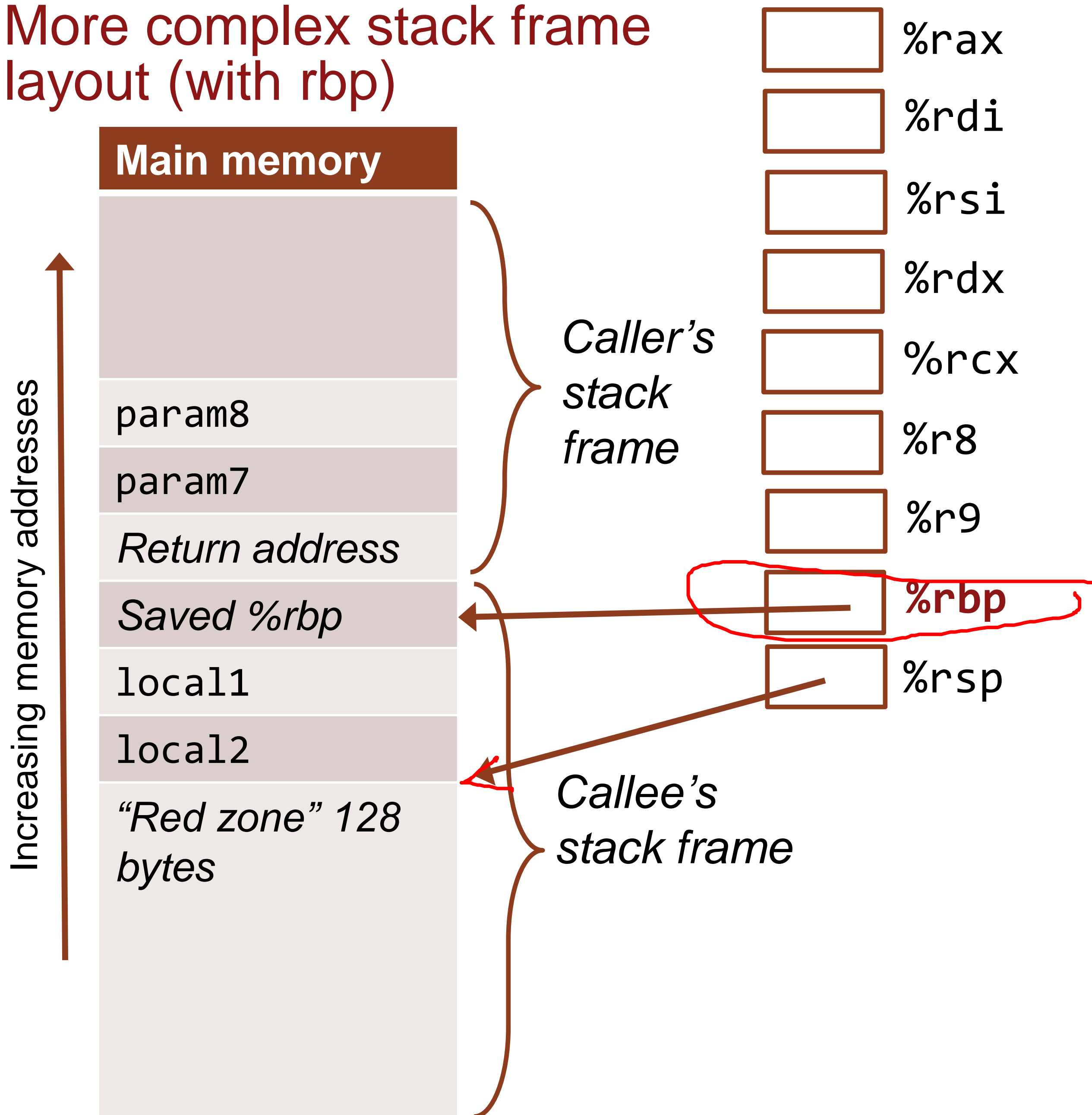
(optional study) More complex stack frame management

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



More complex stack frame layout (with rbp)



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