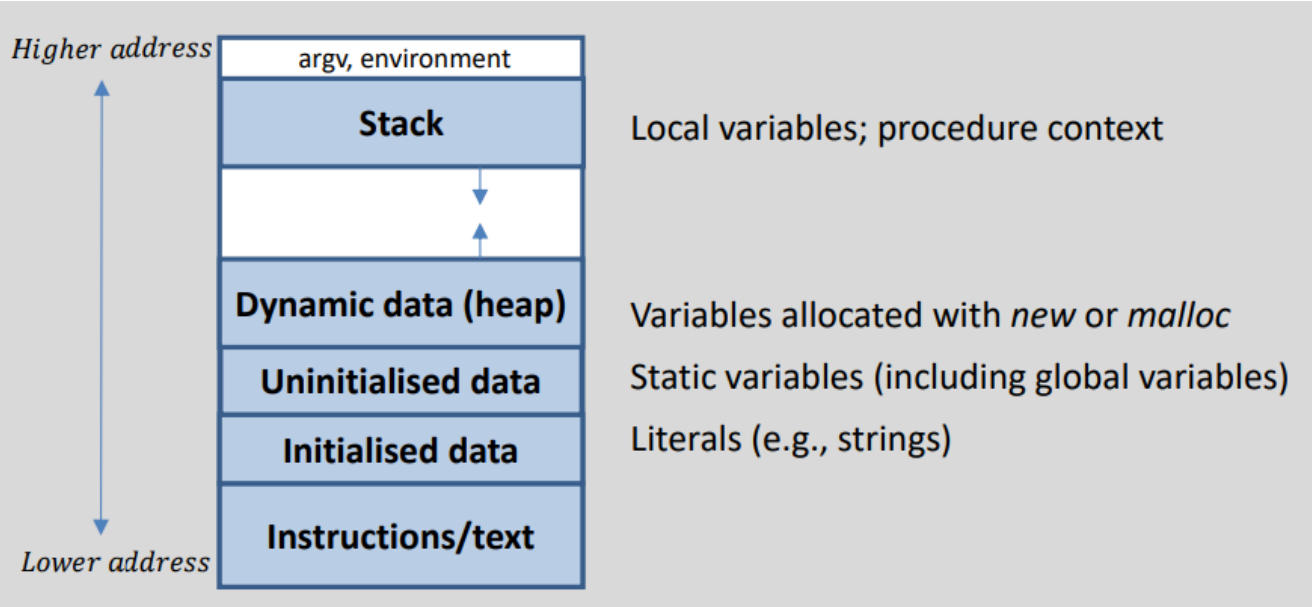
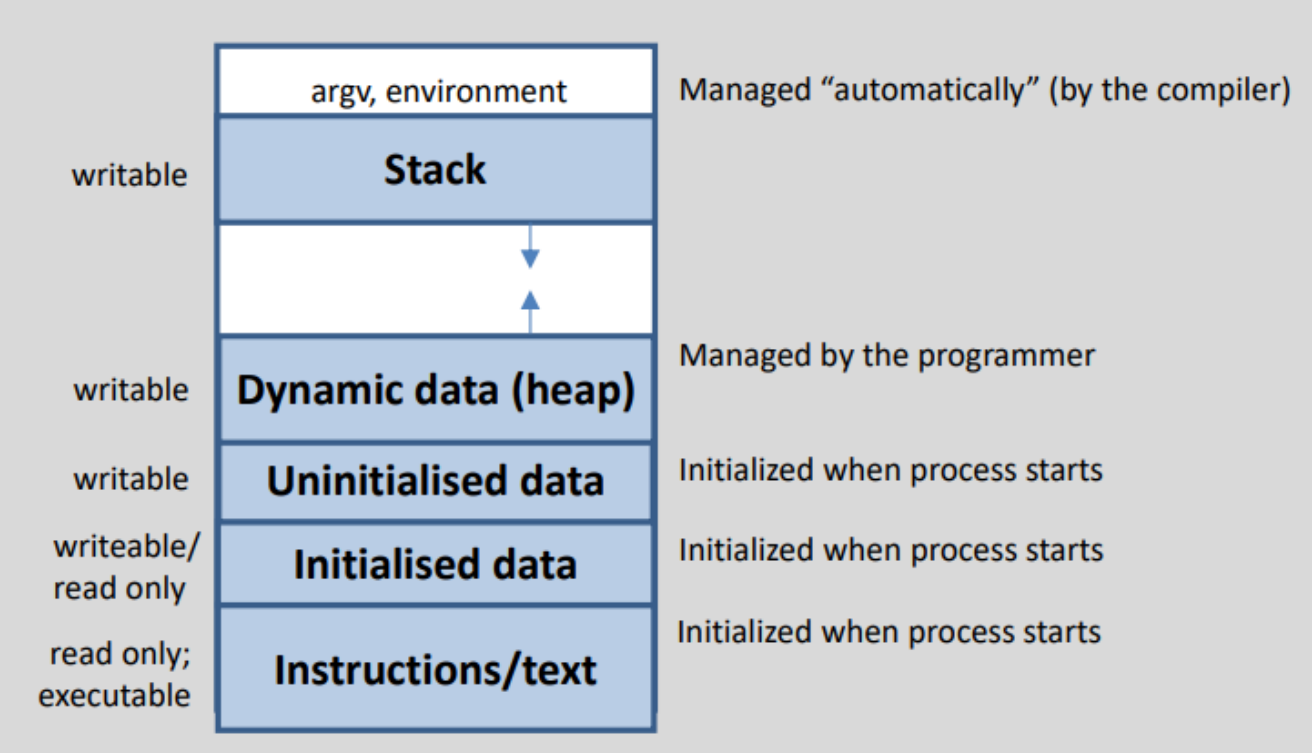
X86-64 Stack and Procedures

Memory layout

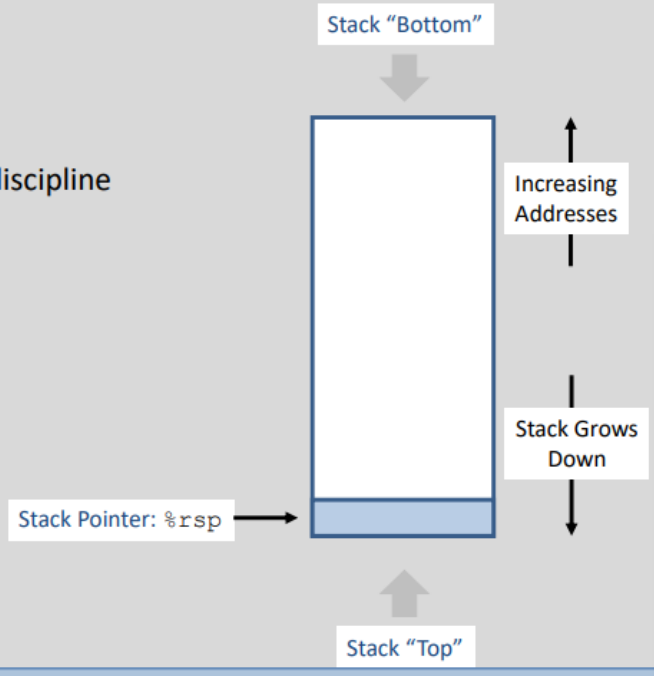




Stack discipline

Stack discipline

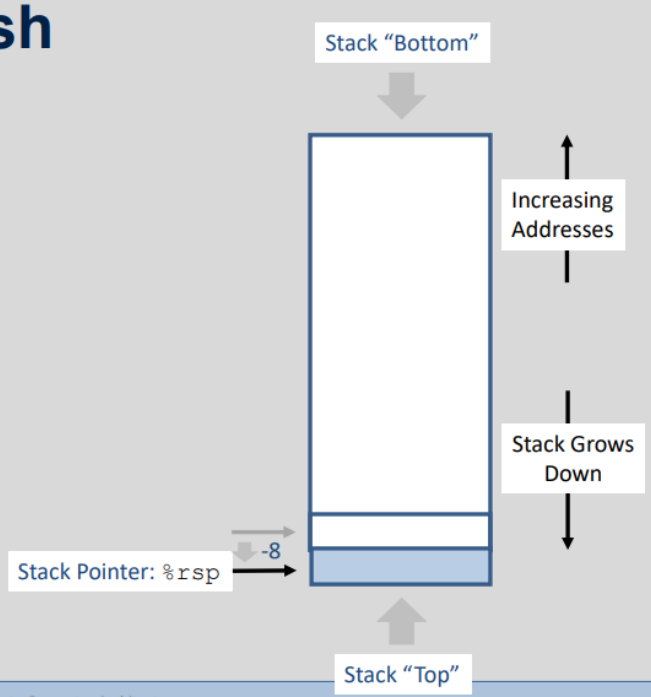
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address = address of "top" element



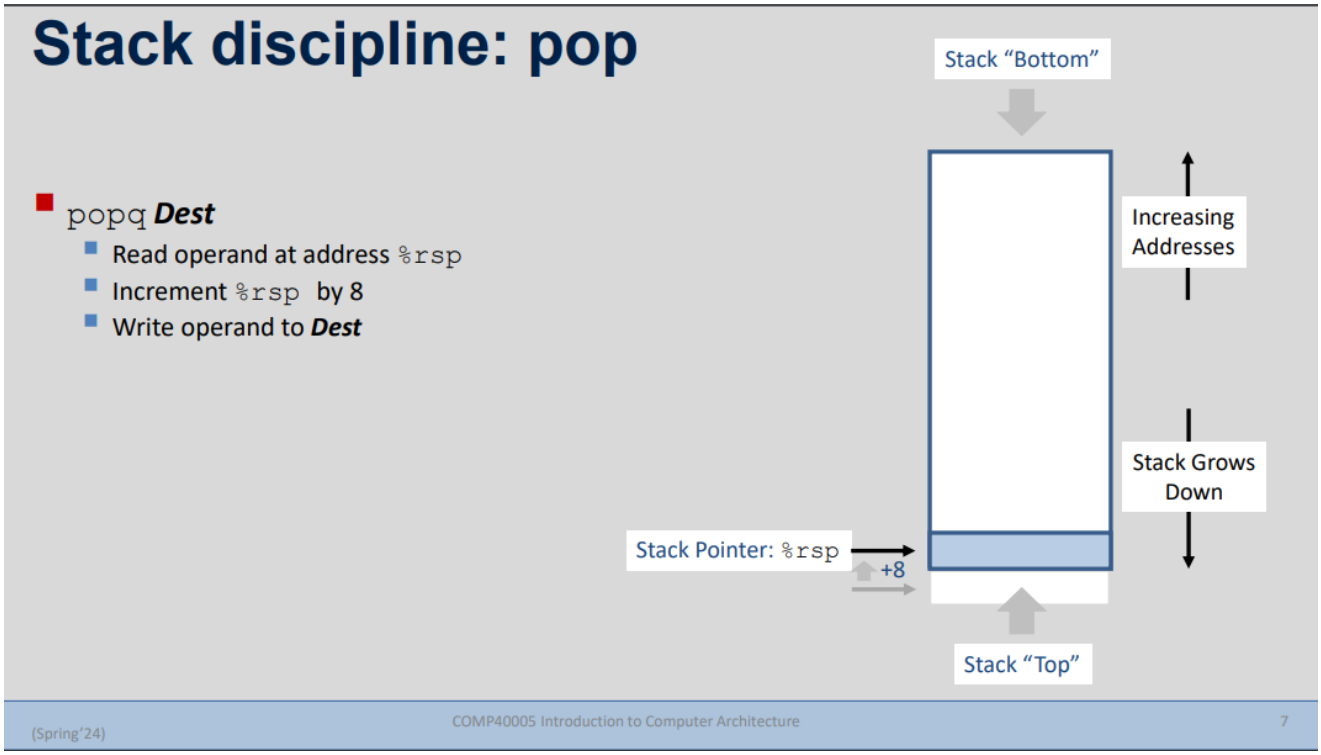
Stack discipline: `push`

Stack discipline: push

- `pushq` ***Src***
  - Fetch operand at ***Src***
  - Decrement `%rsp` by 8 (*why 8?*)
  - Write operand at address given by `%rsp`



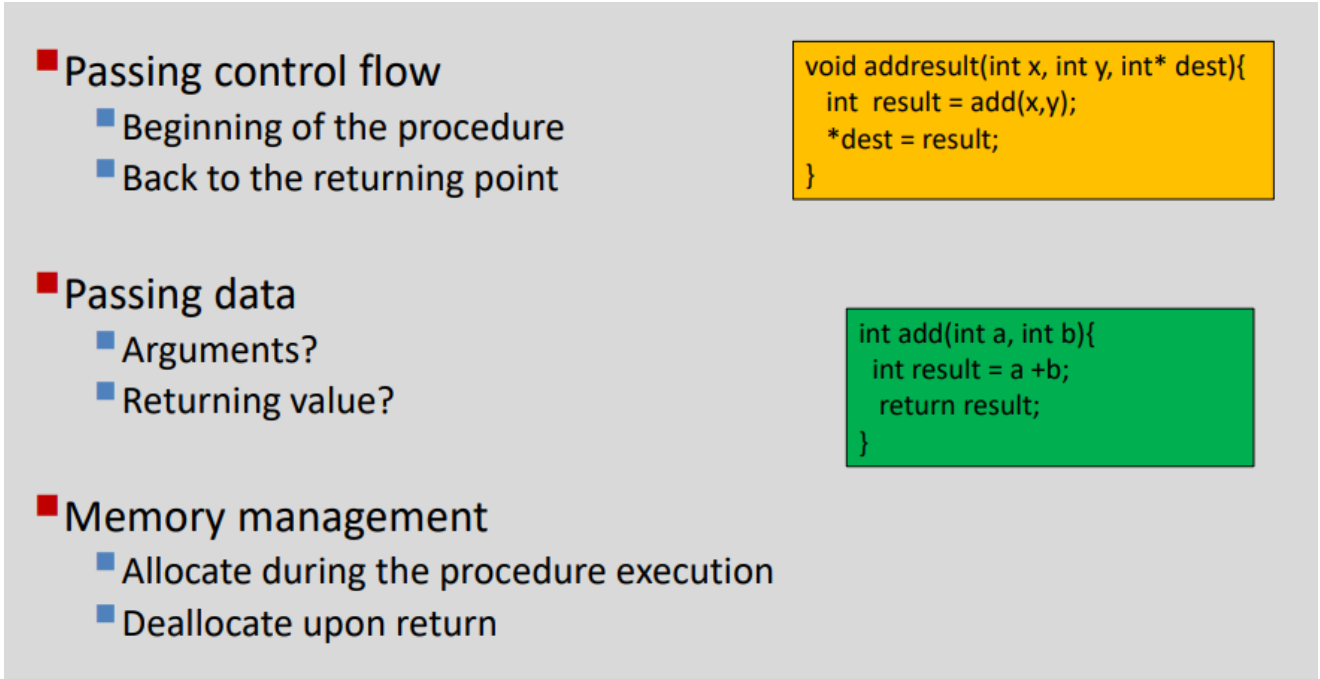
Stack discipline: `pop`



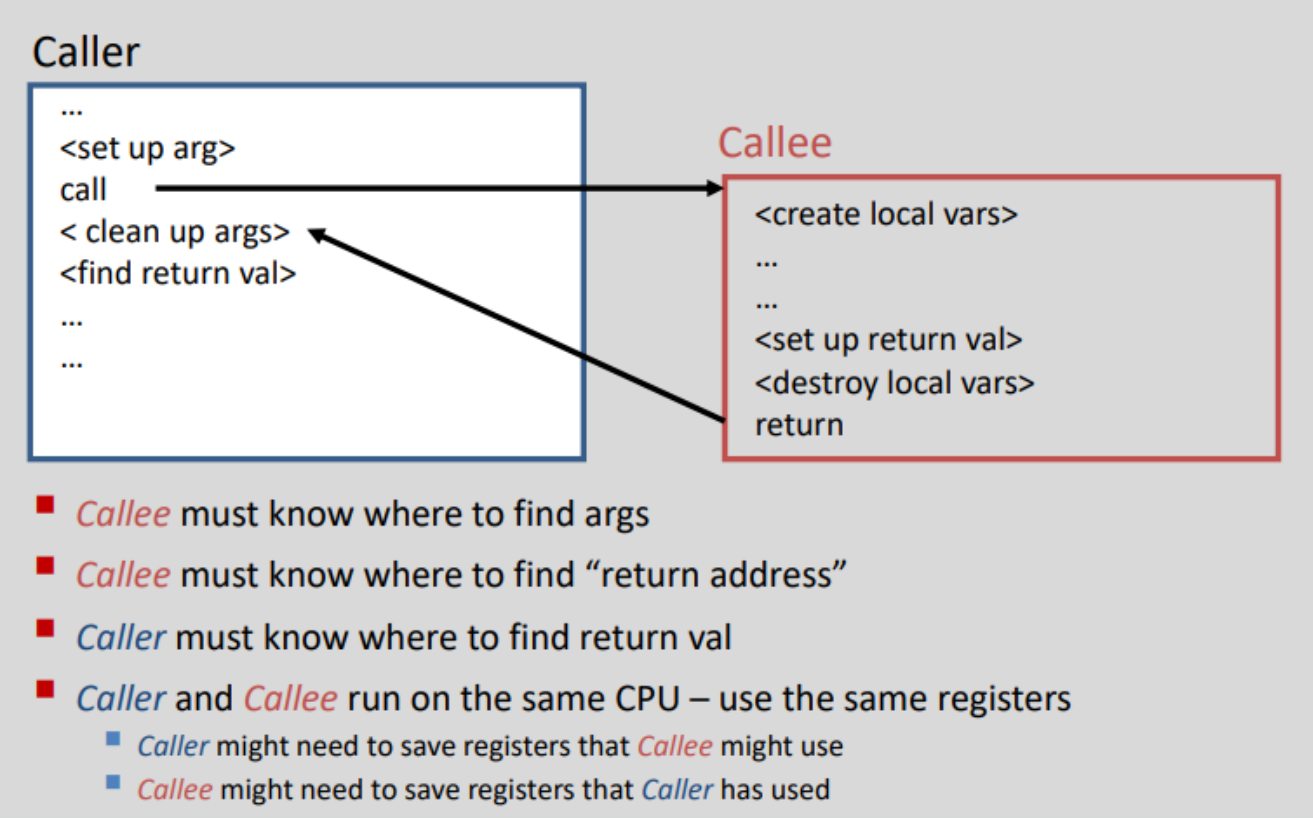
We don't delete the value at $\%rsp$, but it is now **overwritable**

Procedure call and return

Procedure call mechanisms

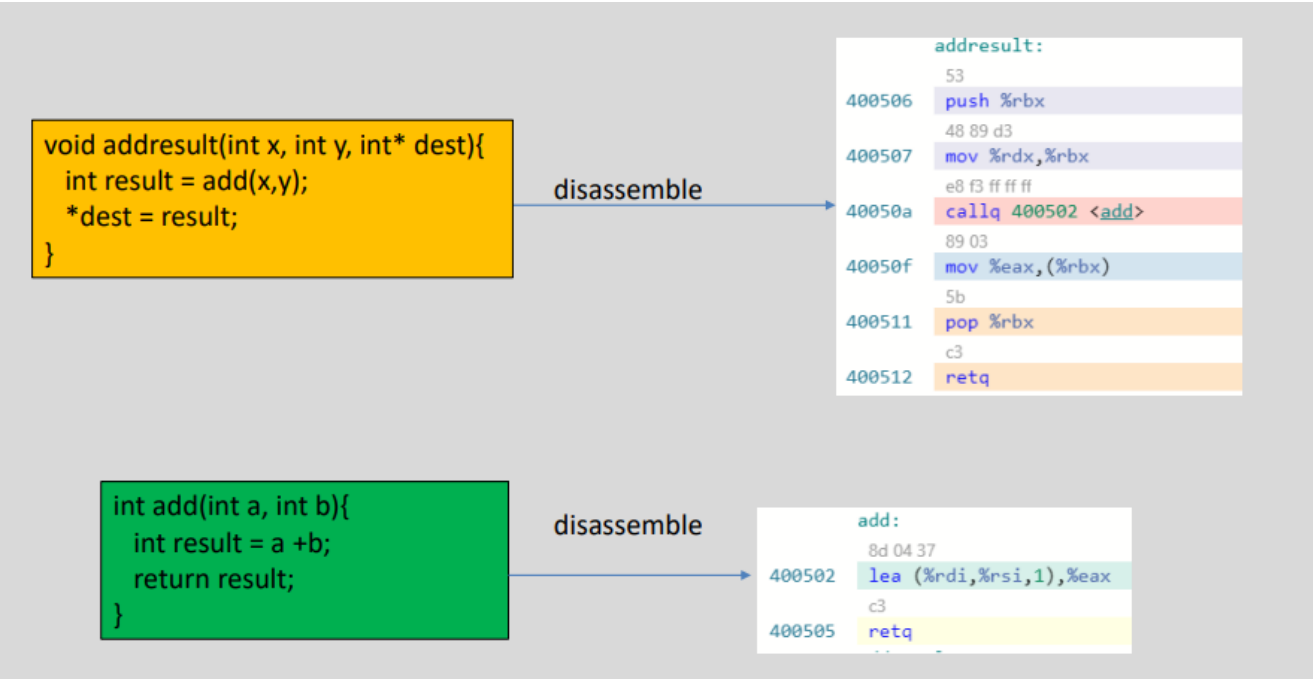


Procedure call mechanisms overview



- *Callee* must know where to find args
- *Callee* must know where to find "return address"
- *Caller* must know where to find return val
- *Caller* and *Callee* run on the same CPU – use the same registers
  - *Caller* might need to save registers that *Callee* might use
  - *Callee* might need to save registers that *Caller* has used

Procedure call mechanisms example



Procedure control flow

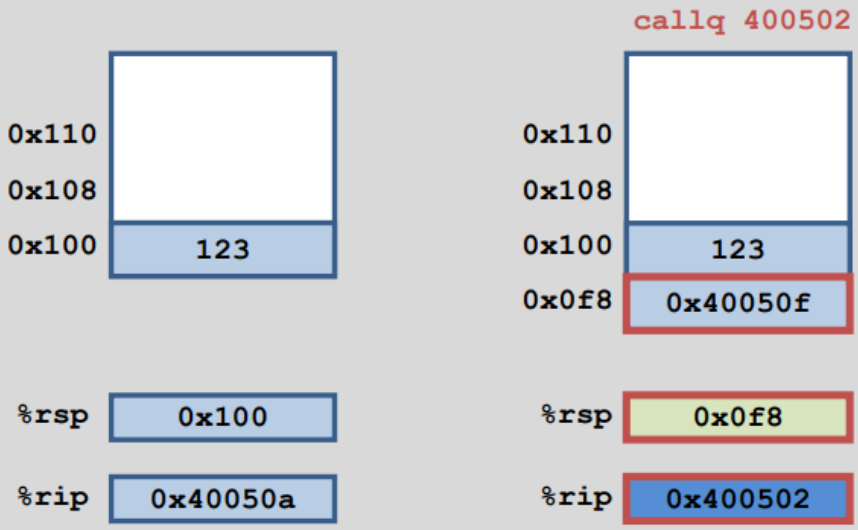
- **Use *stack* to support procedure call and return**
- **Procedure call**: `call` *label*
  - `push` return address on stack
  - `jmp` to *label*
- **Return address:**
  - Address of instruction beyond `call`
  - Example from disassembly

```
40050a:          e8 f3 ff ff ff          callq 400502 <add>
40050f:          89 03                   mov %eax,(%rbx)
```

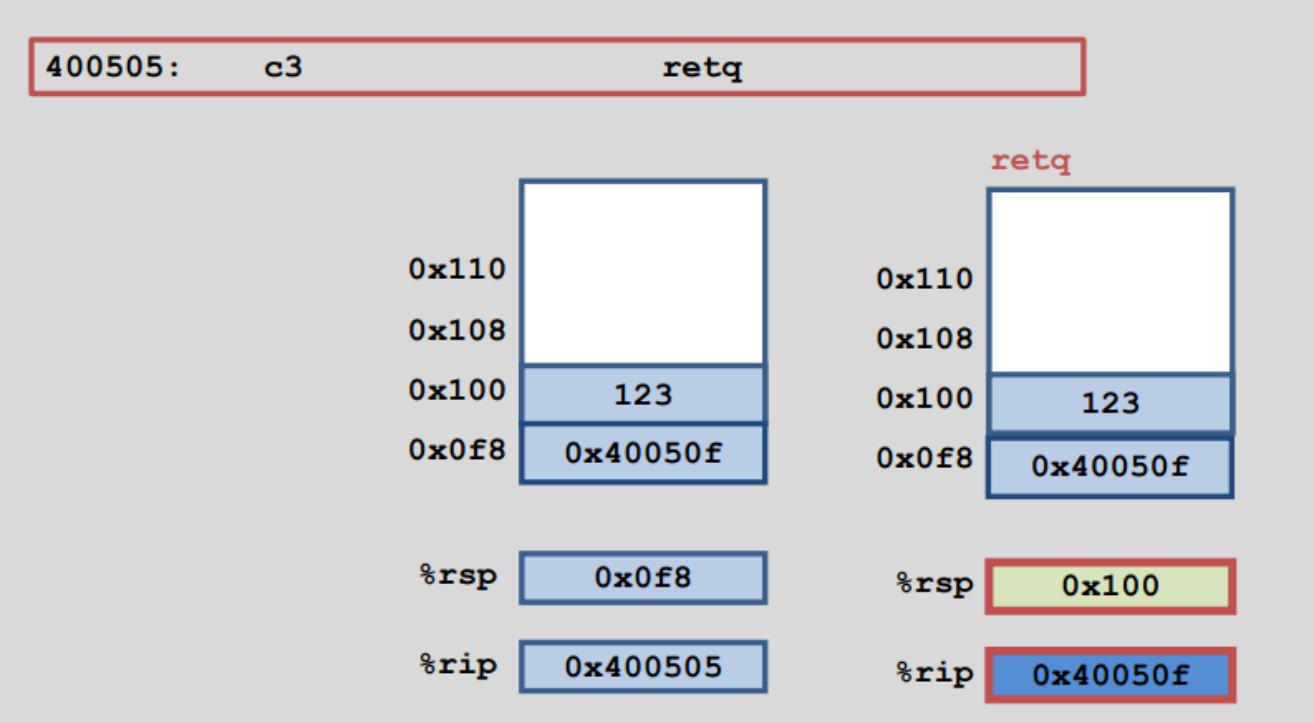
  - Return address = 0x40050f
- **Procedure return**: `ret`
  - `pop` address from stack
  - `jmp` to address

Control Flow - Procedure call example x86- 64

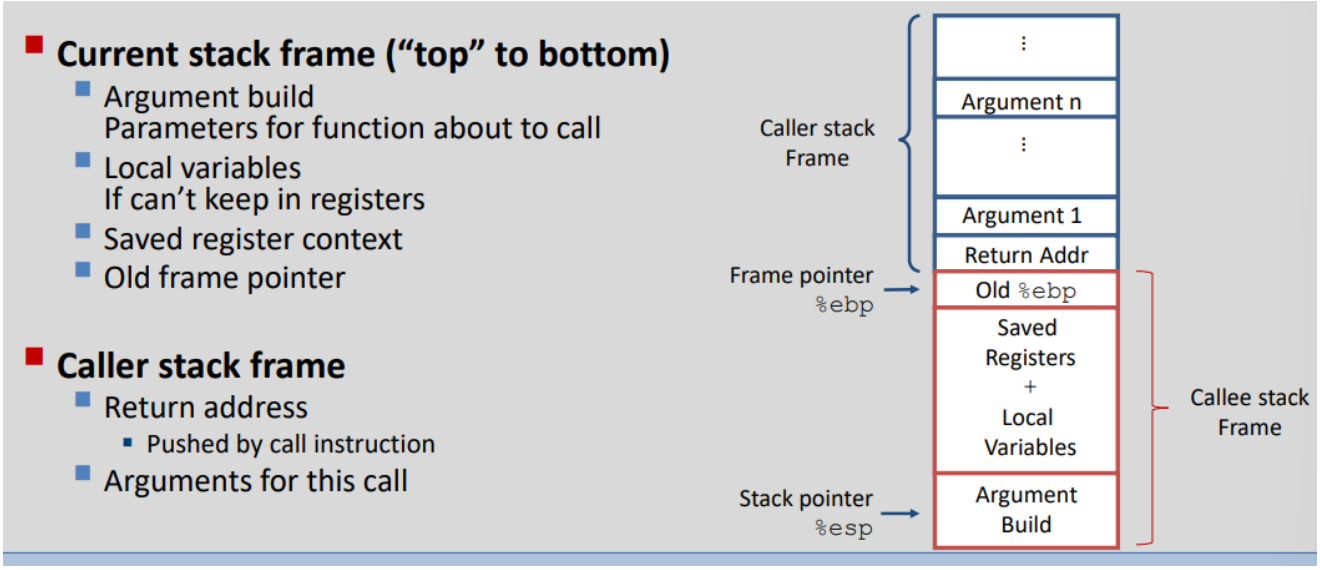
```
40050a:          e8 f3 ff ff ff          callq 400502 <add>
40050f:          89 03                   mov %eax,(%rbx)
```



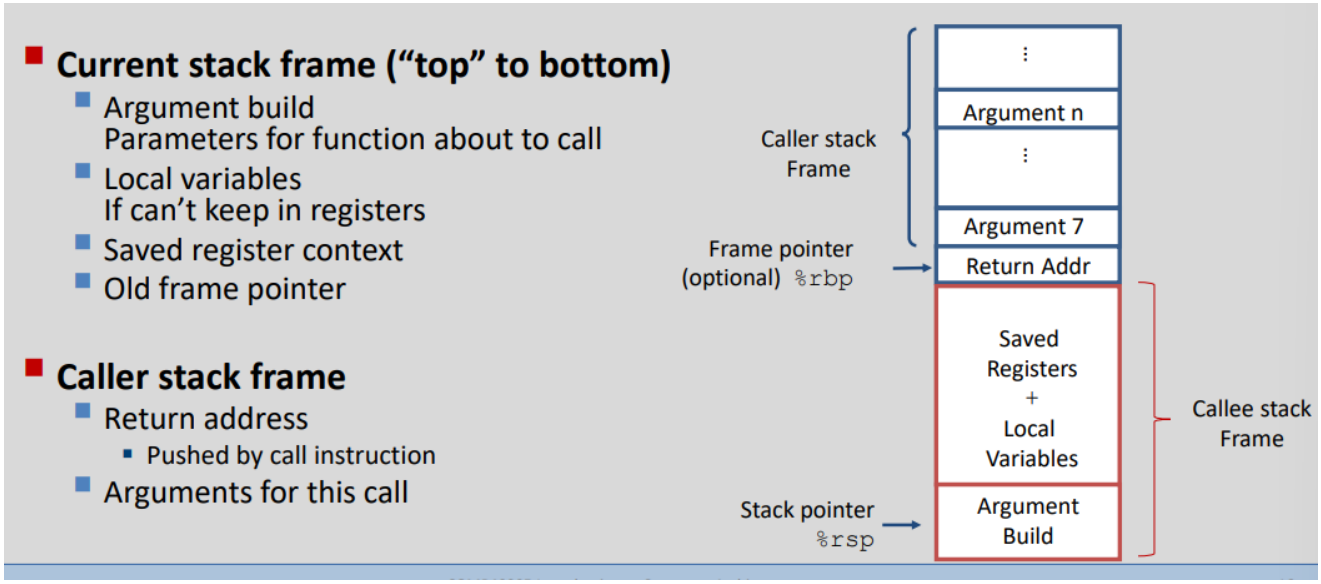
Control Flow - Procedure return example x86-64



Memory Management - IA32 / Linux stack frame



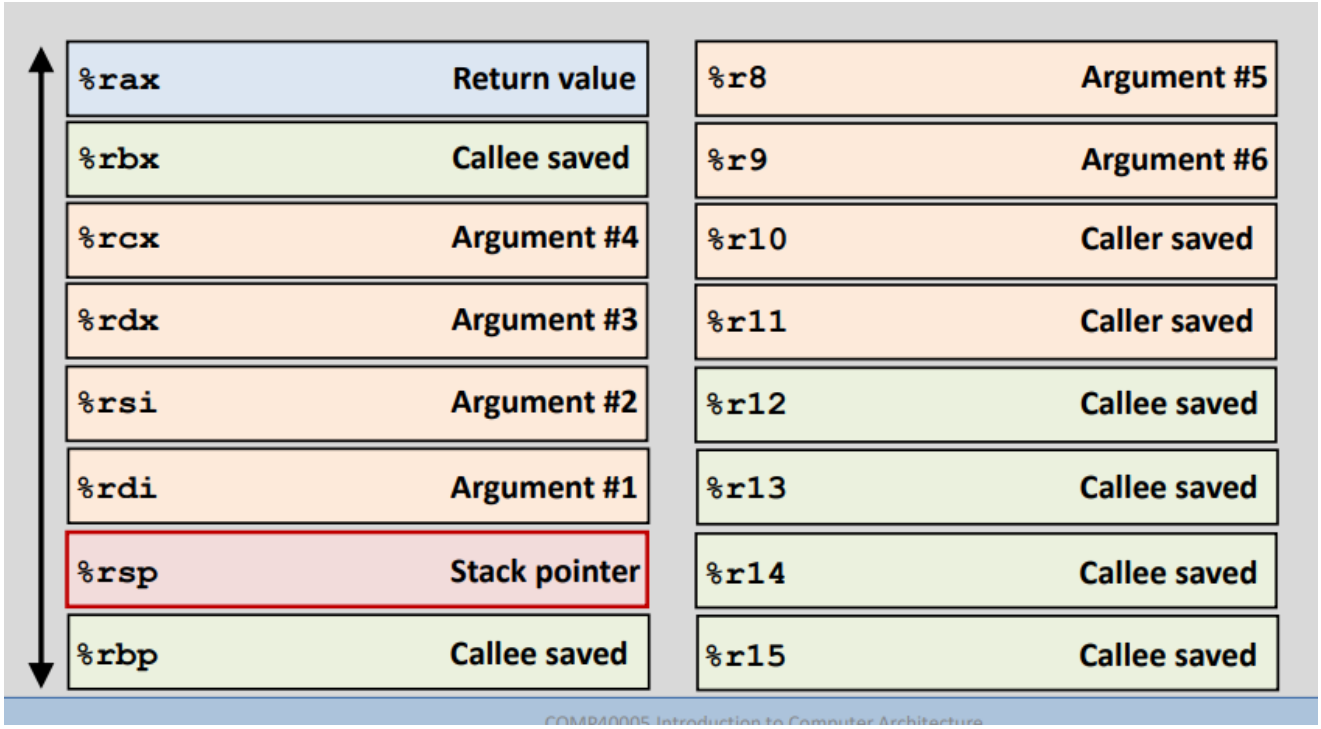
Memory Management – X86-64 / Linux stack frame



First 6 arguments are always stored in **registers**

Massive optimisation, less values on stack, Stack is on main memory = Slow

Passing data - x86-64 integer registers: conventions



Diana Silk Dress cost £89

Passing data – Local Storage on the stack

- There are not enough registers to hold all of the local data.
- The address operator '&' is used.
- Arrays or structures are used as local variables.
- **A procedure allocate space on the stack by decrementing the stack pointer**

Passing data – Local storage in registers

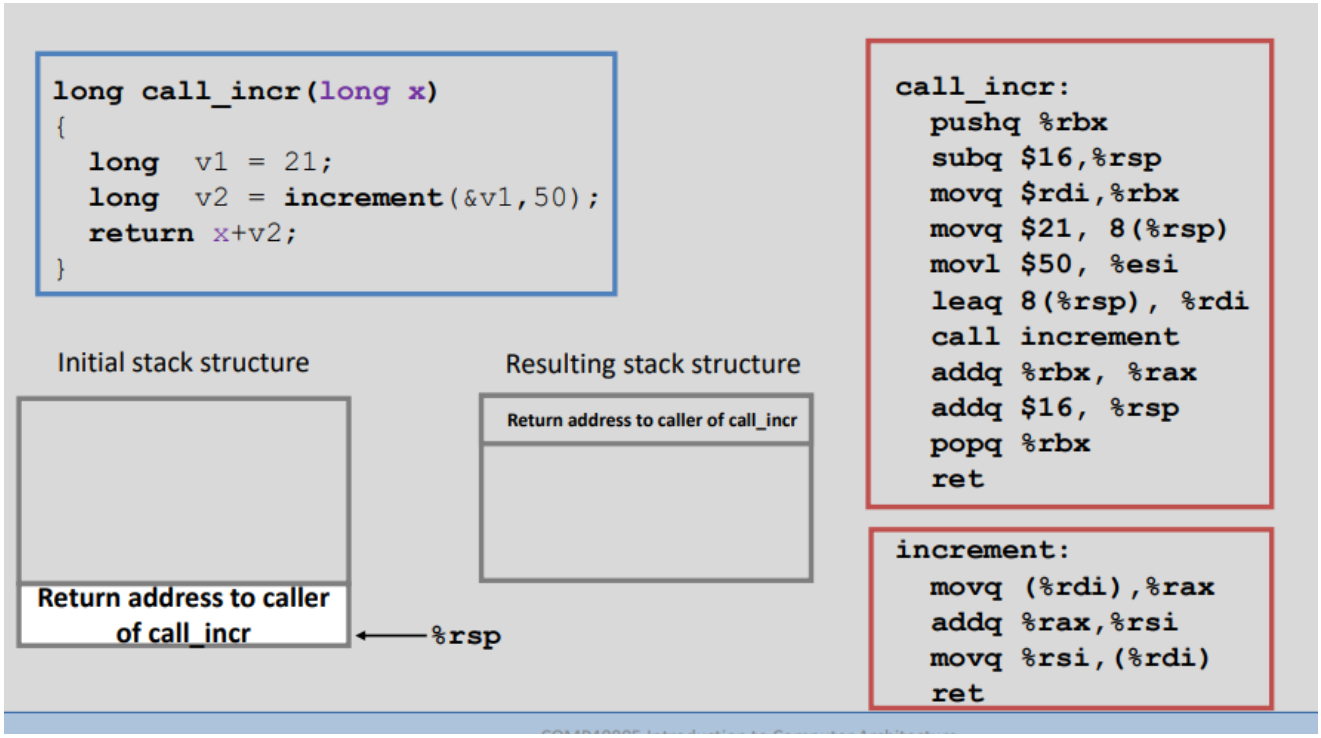
- The set of provided registers act as a single shared resource by all procedures.
- Callee must preserve the *calle-saved* register's values by: not changing the value at all or by pushing the original value to the stack.
- The *caller-saved* registers can be modified by any function. The caller needs to save the values in the stack to preserve the old value as the callee is free to alter the values.

Pointers

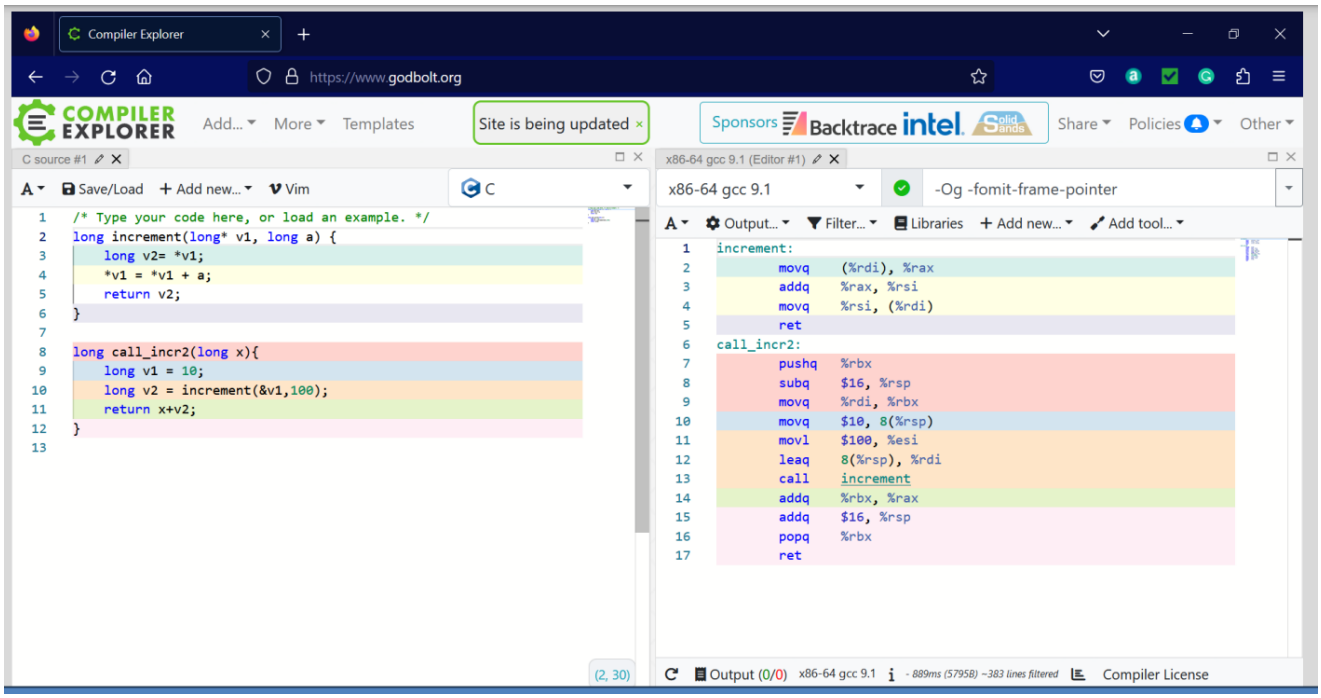
*

&

Callee-saved Example

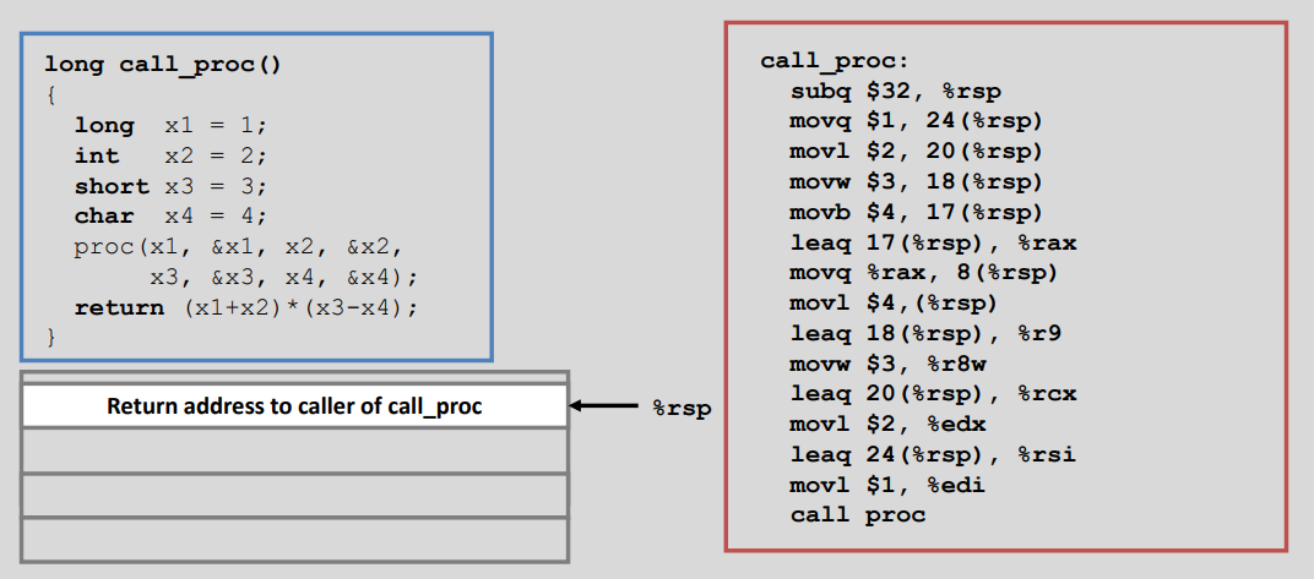


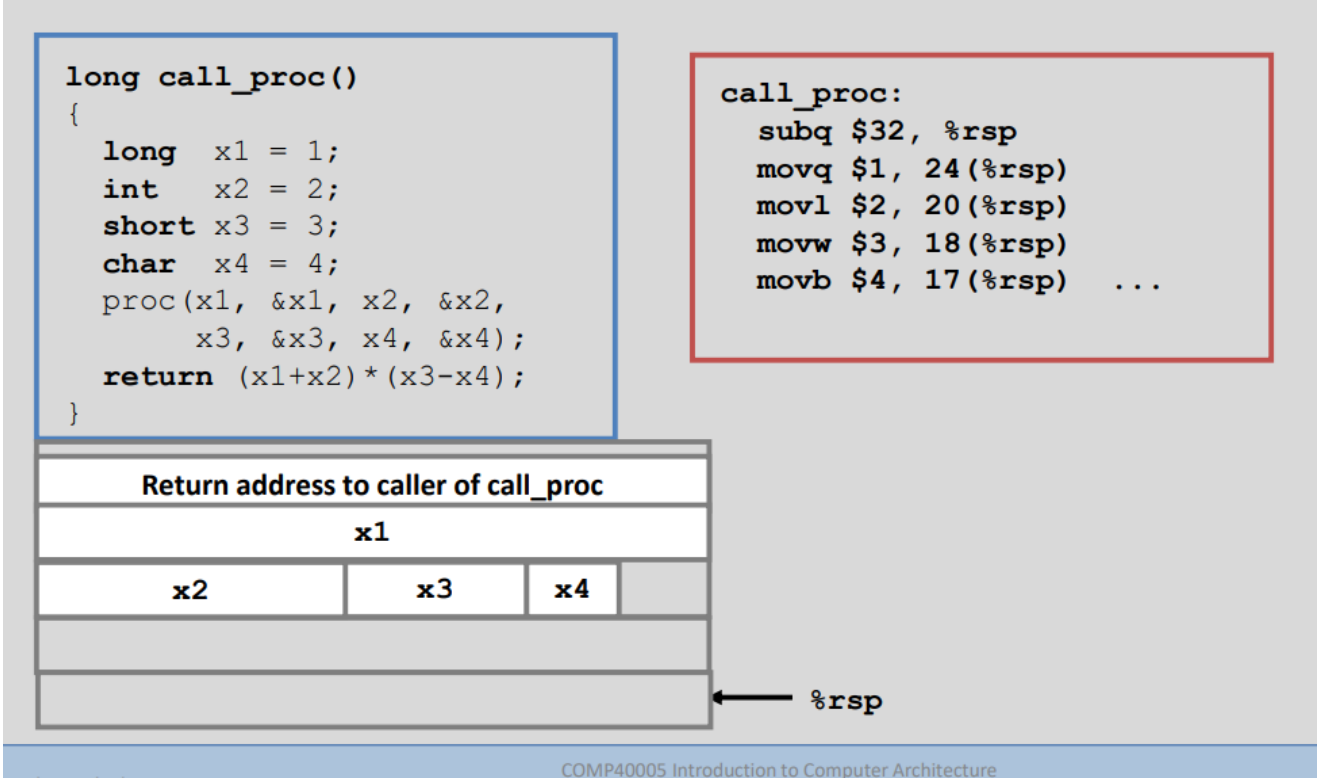
Callee-save example (part II)



See the flags to the right of the green tick!

An Example

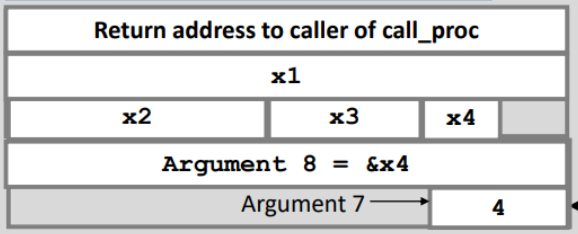

long call_proc()
{
long x1 = 1;
int x2 = 2;
short x3 = 3;
char x4 = 4;
proc(x1, &x1, x2, &x2,
x3, &x3, x4, &x4);
return (x1+x2)*(x3-x4);
}
call_proc:
subq $32, %rsp
movq $1, 24(%rsp)
movl $2, 20(%rsp)
movw $3, 18(%rsp)
movb $4, 17(%rsp)
leaq 17(%rsp), %rax
movq %rax, 8(%rsp)
movl $4,(%rsp)
leaq 18(%rsp), %r9
movw $3, %r8w
leaq 20(%rsp), %rcx
movl $2, %edx
leaq 24(%rsp), %rsi
movl $1, %edi
call proc
Return address to caller of call_proc
%rsp


long call_proc()
{
long x1 = 1;
int x2 = 2;
short x3 = 3;
char x4 = 4;
proc(x1, &x1, x2, &x2,
x3, &x3, x4, &x4);
return (x1+x2)*(x3-x4);
}
call_proc:
subq $32, %rsp
movq $1, 24(%rsp)
movl $2, 20(%rsp)
movw $3, 18(%rsp)
movb $4, 17(%rsp) ...
Return address to caller of call_proc
x1
x2
x3
x4
%rsp
COMP40005 Introduction to Computer Architecture

Example

```
long call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  ...
  subq $32, %rsp
  movq $1, 24(%rsp)
  movl $2, 20(%rsp)
  movw $3, 18(%rsp)
  movb $4, 17(%rsp)
  leaq 17(%rsp), %rax
  movq %rax, 8(%rsp)
  movl $4,(%rsp)
  leaq 18(%rsp), %r9
  movw $3, %r8w
  leaq 20(%rsp), %rcx
  movl $2, %edx
  leaq 24(%rsp), %rsi
  movl $1, %edi
  call proc
  ...
```

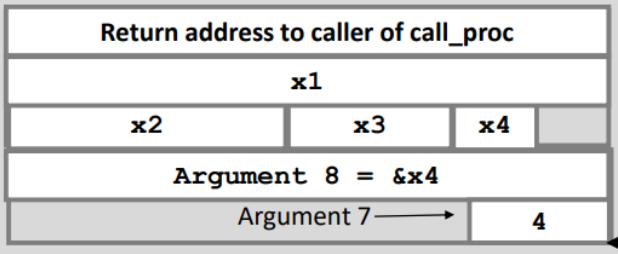


%rsp

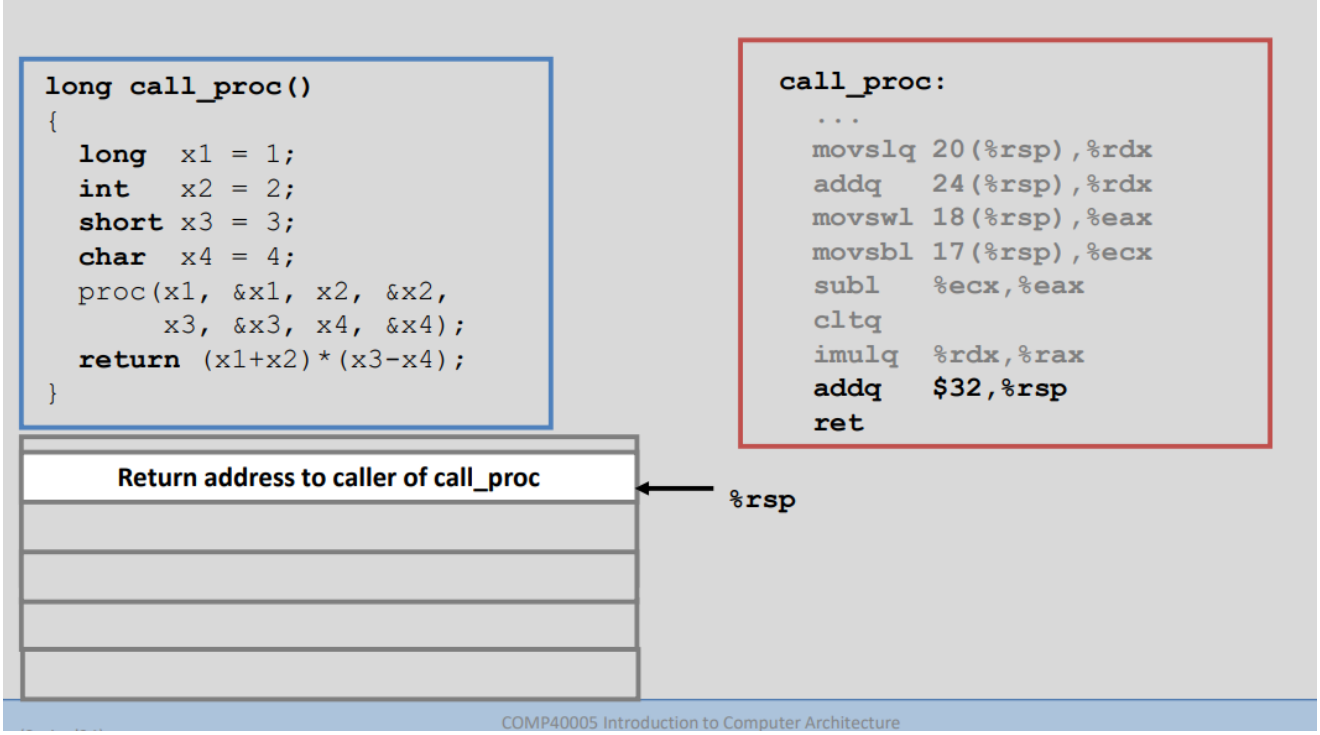
Arguments are passed (in order):
rdi, rsi, rdx, rcx, r8, r9, then stack


```
long call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  ...
  movslq 20(%rsp),%rdx
  addq   24(%rsp),%rdx
  movswl 18(%rsp),%eax
  movsbl 17(%rsp),%ecx
  subl   %ecx,%eax
  cltq
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```

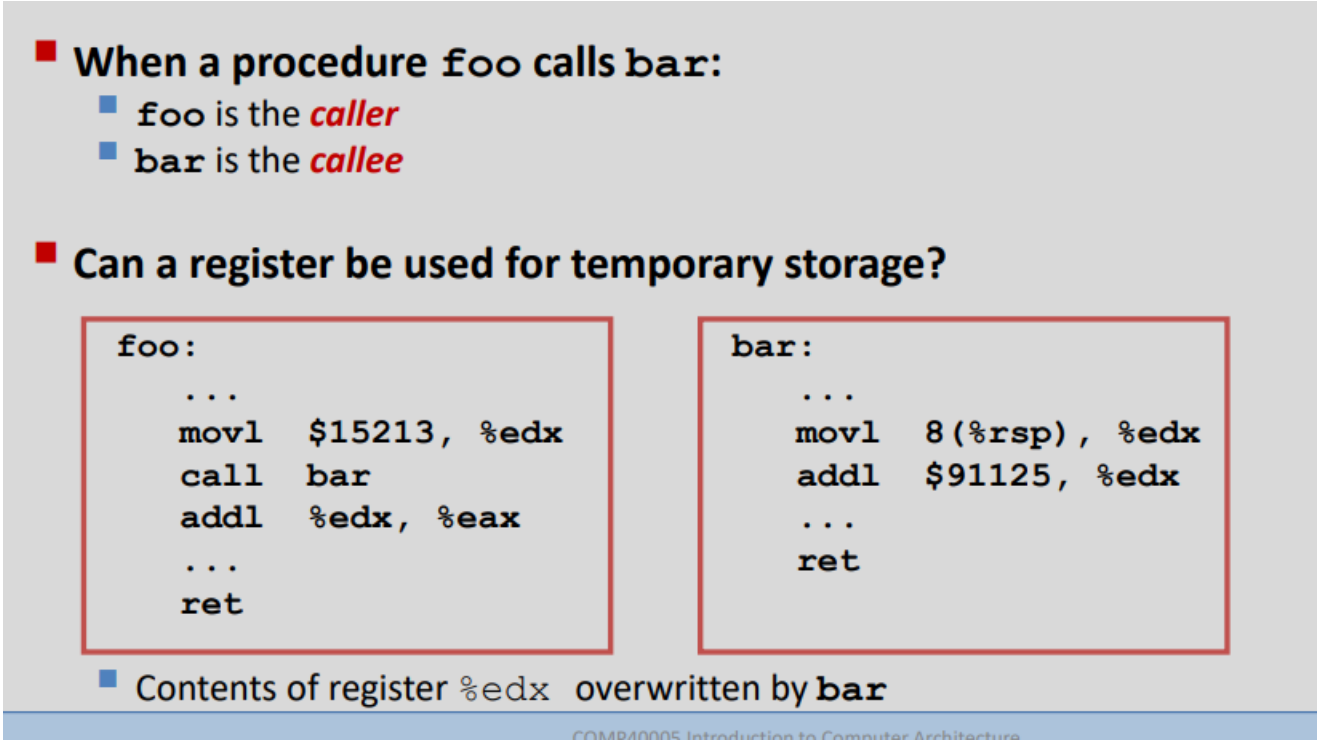


%rsp



x86-64 calling conventions

Register saving conventions



- **When a procedure `foo` calls `bar`:**
  - `foo` is the *caller*
  - `bar` is the *callee*
- **Can register be used for temporary storage?**
- **Conventions**
  - *"Caller Save"*
    Caller saves temporary in its frame before calling
  - *"Callee Save"*
    Callee saves temporary in its frame before using

Calling convention

Calling method (*Caller*)

- Set parameters in order (registers and stack)
- Call (jump to) method

- Remove parameters from the stack
- Use the returned result

Called method (*Callee*)

- Save registers on stack
- Execute body of method
- Copy result (if any) to eax/rax
- Restore registers from stack
- Return from method (jump back)

X86-64 procedure call highlights

- Arguments (up to first 6) in registers
  - Faster to get these values from registers than from stack in memory
- Local variables also in registers (if there is room)
- callq instruction stores 64-bit return address on stack
  - address pushed onto stack, decrementing %rsp by 8
- No frame pointer
  - All references to stack frame made relative to %rsp; eliminates the need to update %ebp/%rbp, which is now available for general-purpose use
- Registers still designated “caller-saved” or “callee-saved”

X86-64 stack frames

- Often (ideally), x86_64 functions need no stack frame at all
  - just a return address is pushed onto the stack when a function call is made
- A function does need a stack frame when it:
  - has too many local variables to hold in registers
  - has local variables that are arrays or structs
  - uses the address-of operator (&) to compute the address of a local variable
  - calls another function that takes more than six arguments
  - need to save the state of callee-save registers before modifying them

X86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers
- Minimal use of stack
  - Sometimes none
  - When needed, allocate/deallocate entire frame at once vs. multiple push/pop
  - No more need of frame pointer: address relative to stack pointer
- More room for compiler optimizations
  - Prefer to store data in registers rather than memory
  - Minimize modifications to stack pointer