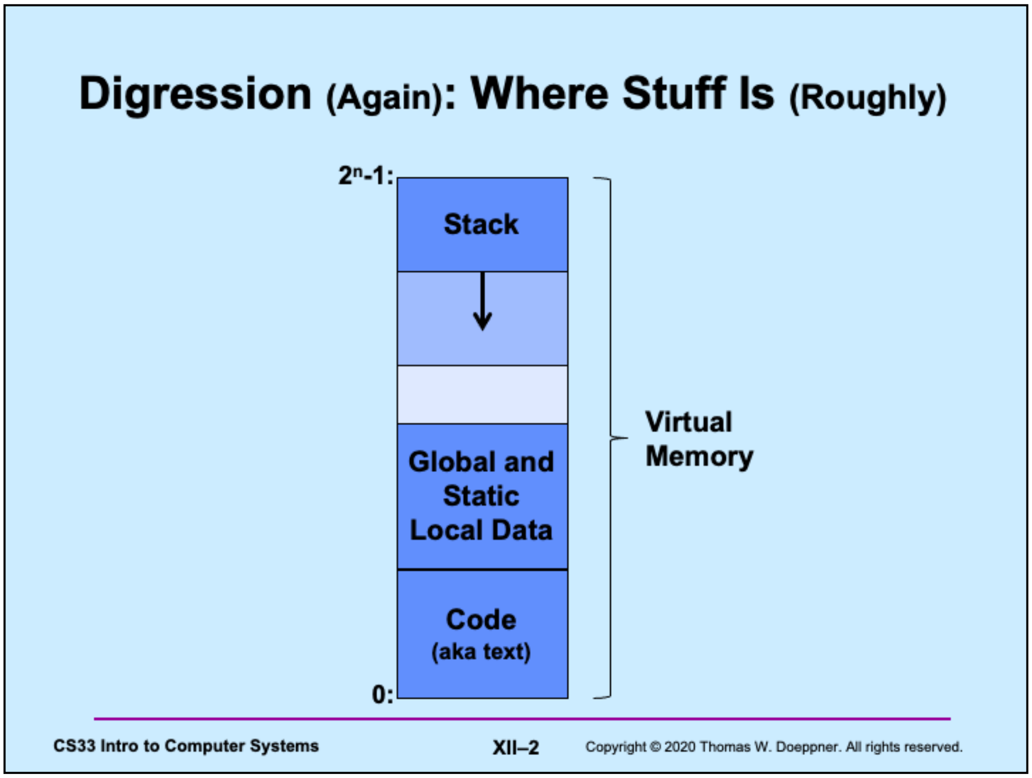
CS 33

Machine Programming (4)

Some of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook "Computer Systems: A Programmer's Perspective," 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O'Hallaron in Fall 2010. These slides are indicated "Supplied by CMU" in the notes section of the slides.



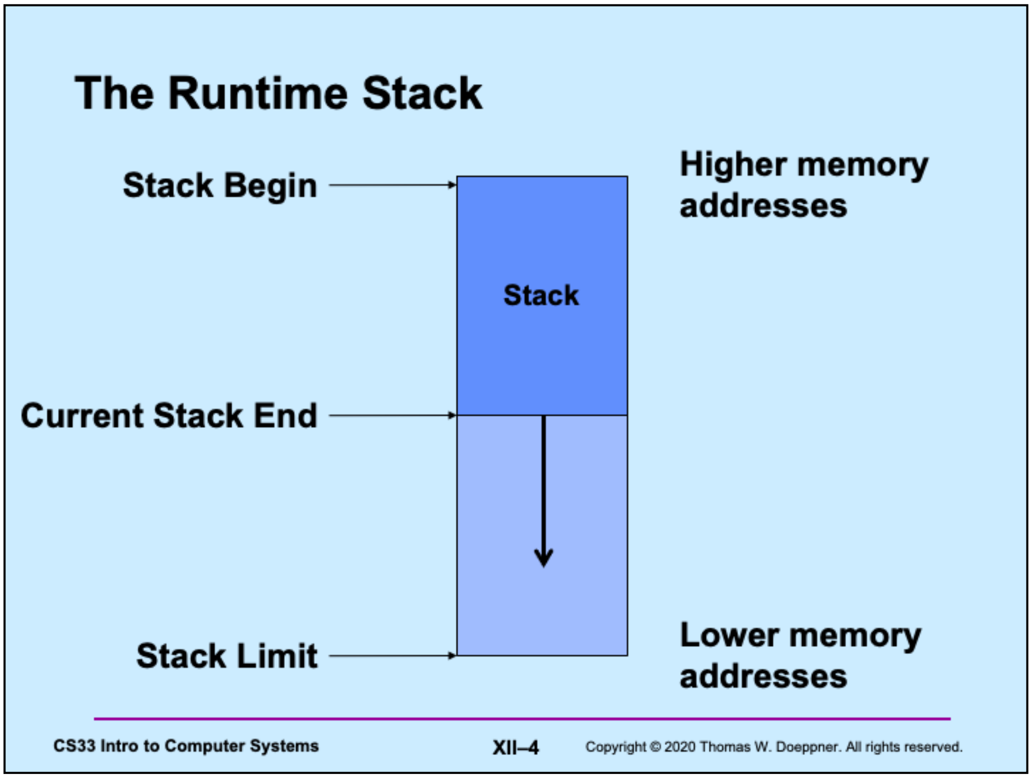
Here we revisit the slide we saw a few weeks ago, this time drawing it with high addresses at the top and low addresses at the bottom. The point is that a large amount of virtual memory is reserved for the stack. In most cases there's plenty of room for the stack and we don't have to worry about exceeding its bounds. However, if we do exceed its bounds (by accessing memory outside of what's been allocated), the program will get a seg fault.

Function Call and Return

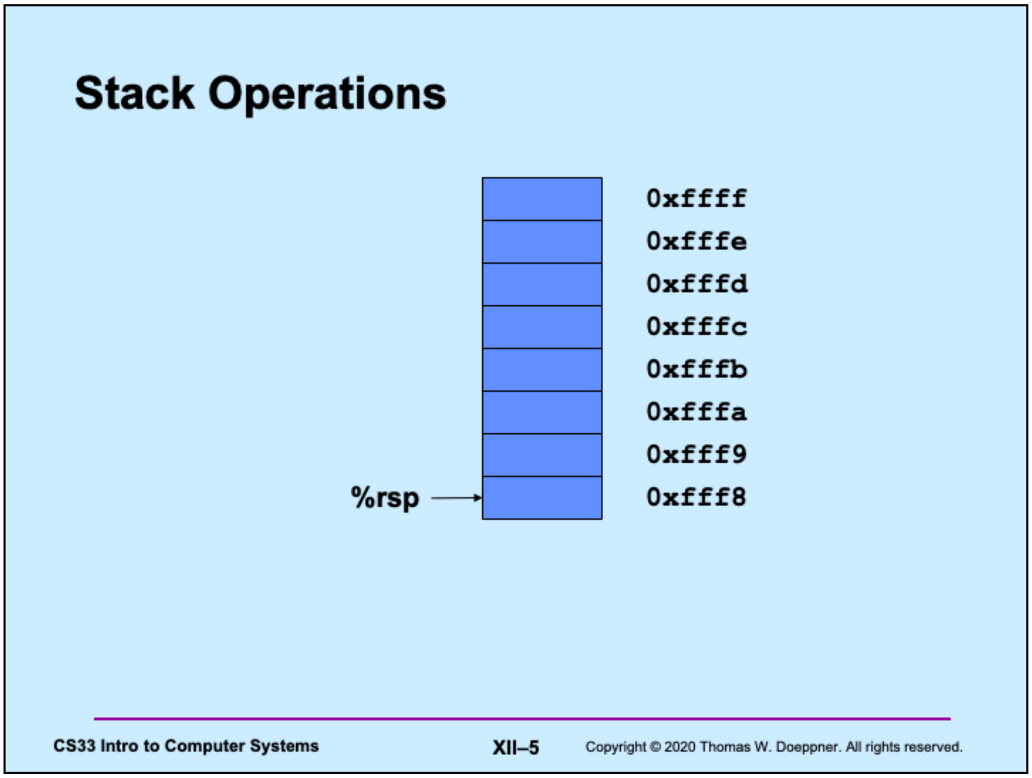
- Function A calls function B
- Function B calls function C

... several million instructions later

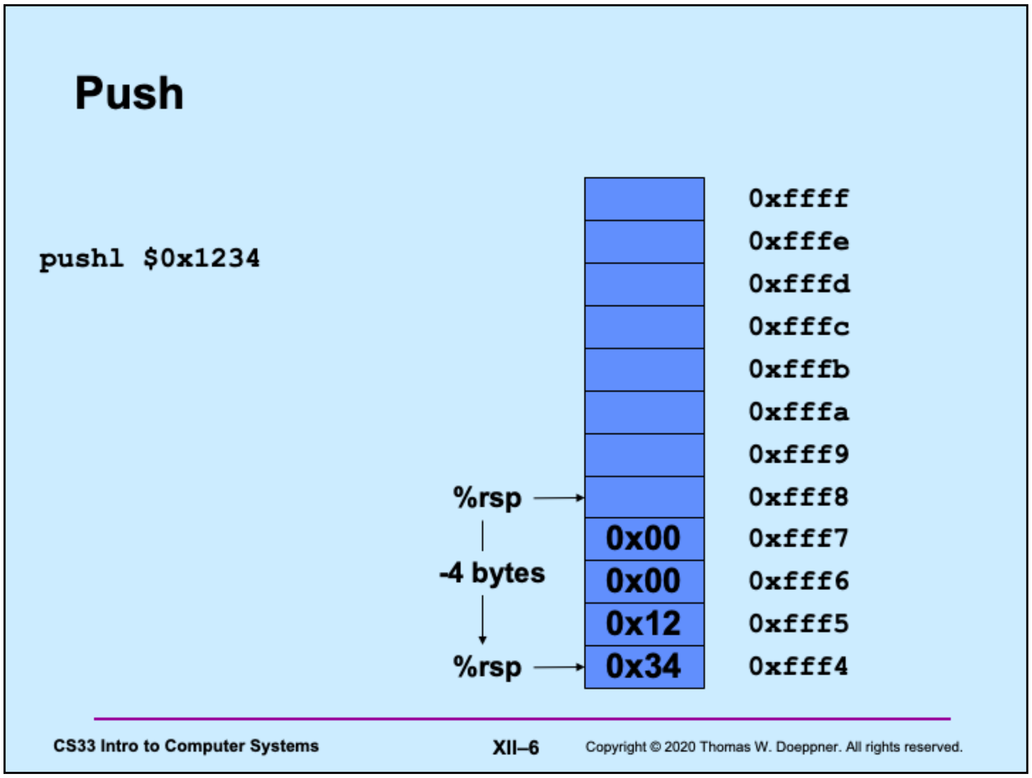
- C returns
  - how does it know to return to B?
- B returns
  - how does it know to return to A?



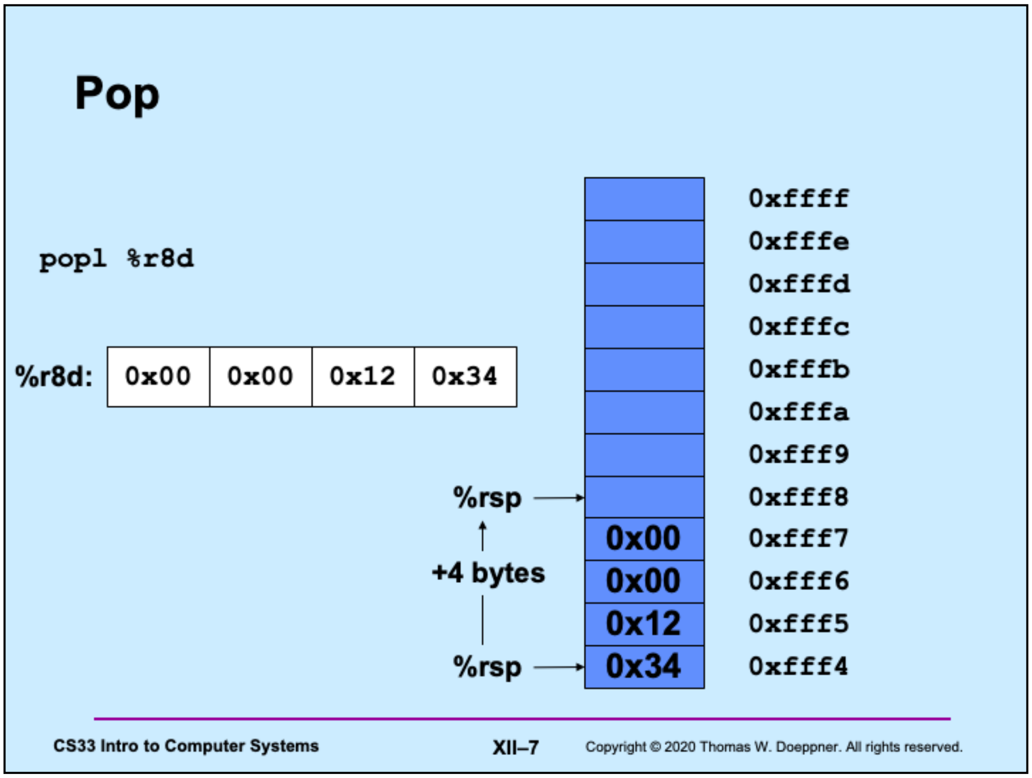
Stacks, as implemented on the X86 for most operating systems (and, in particular, Linux, OSX, and Windows) grow "downwards", from high memory addresses to low memory addresses. To avoid confusion, we will not use the works "top of stack" or "bottom of stack" but will instead use "stack begin" and "current stack end". The total amount of memory available for the stack is that between the beginning of the stack and the "stack limit". When the stack end reaches the stack limit, we're out of memory for the stack.



The stack-pointer register (%rsp) points to the last byte of the stack. Thus, with little-endian addressing, it points to the least-significant byte of the data item at the end of the stack. Thus, %rsp in the slide points to what's perhaps an 8-byte item at the end of the stack.



Here we call pushl to push a 4-byte item onto the end of the stack. First %rsp is decremented by 4 bytes, then the item is copied into the 4-byte location now pointed to by %rsp.



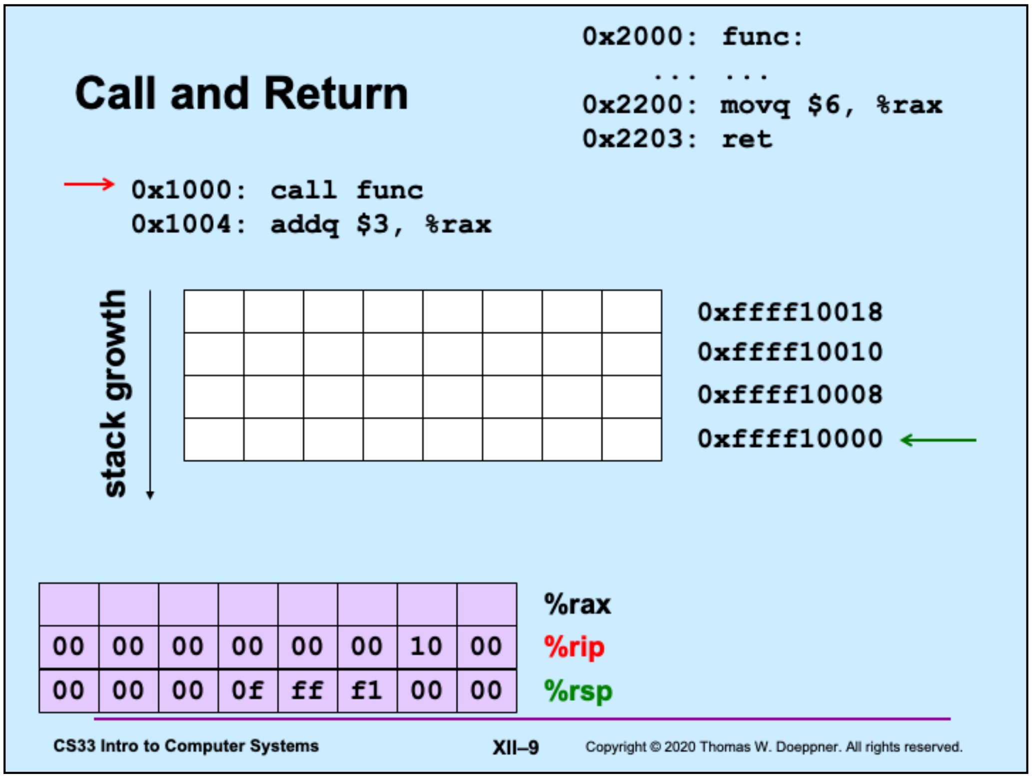
Here we pop an item off the stack. The popl instruction copies the 4-byte item pointed to by %rsp into its argument, then increments %rsp by 4.

Call and Return

```
0x1000: call func
0x1004: addq $3, %rax
```

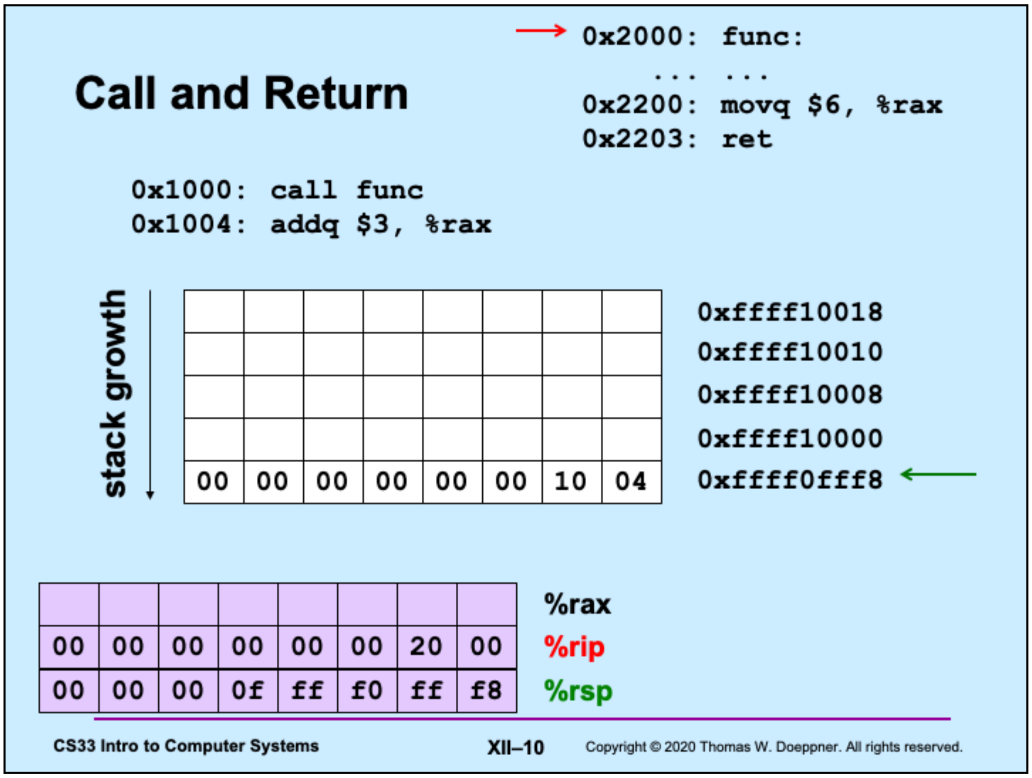
```
0x2000: func:
    ...  ...
0x2200: movq $6, %rax
0x2203: ret
```


When a function is called (using the *call* instruction), the (8-byte) address of the instruction just after the *call* (the "return address") is pushed onto the stack. Then when the called function returns (via the *ret* instruction), the 8-byte address at the end of the stack (pointed to by %rsp) is copied into the instruction pointer (%rip), thus causing control to resume at the instruction following the original call.

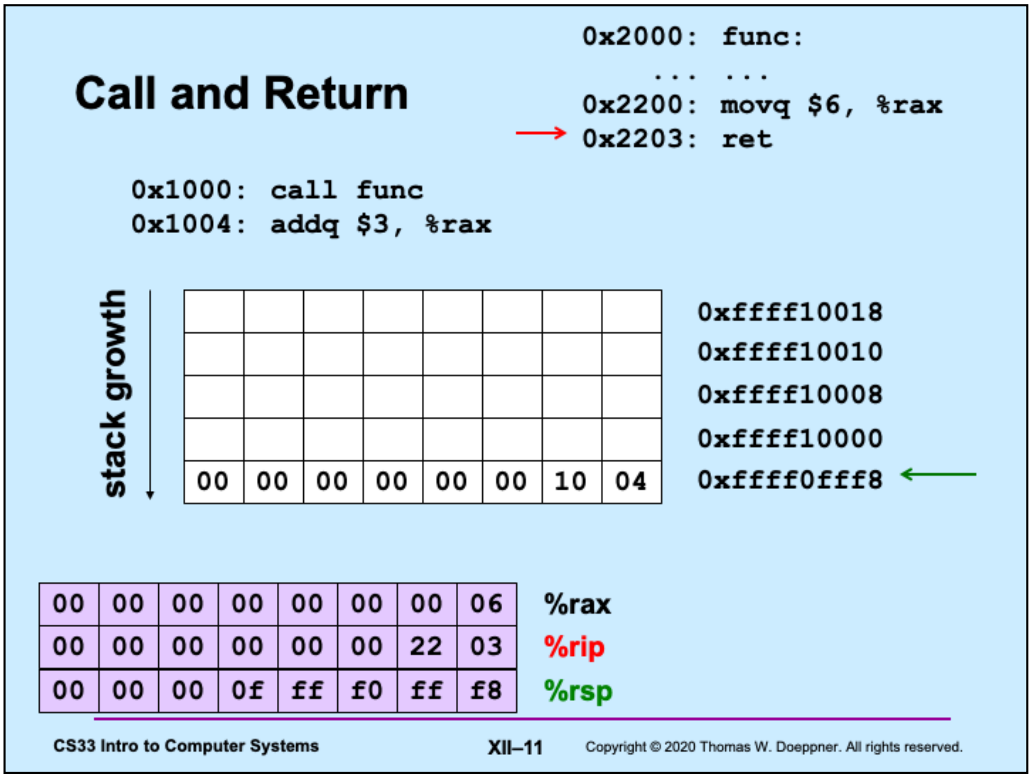


Here we begin walking through what happens during a call and return.

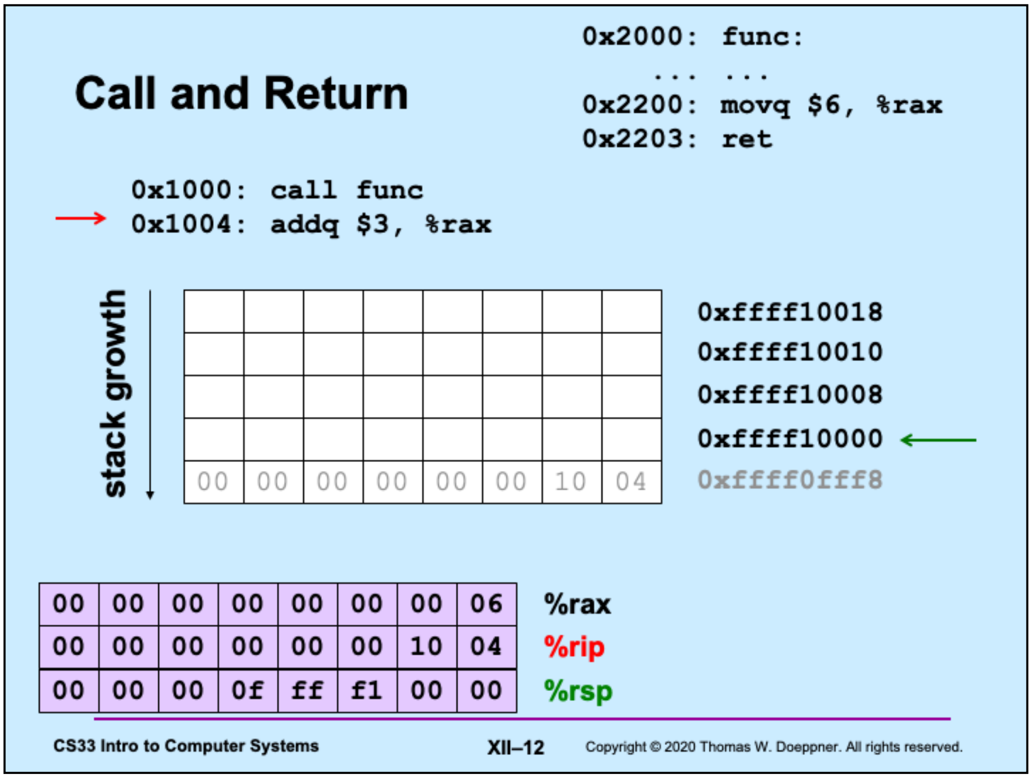
Initially, %rip (the instruction pointer, shown with a red arrow pointing to the right) points to the call instruction – thus it's the next instruction to be executed. %rsp (the stack pointer, shown with a green arrow pointing to the left) points to the current end of the stack. The actual values contained in the relevant registers are shown at the bottom of the slide (%rax isn't relevant yet, but will be soon!).



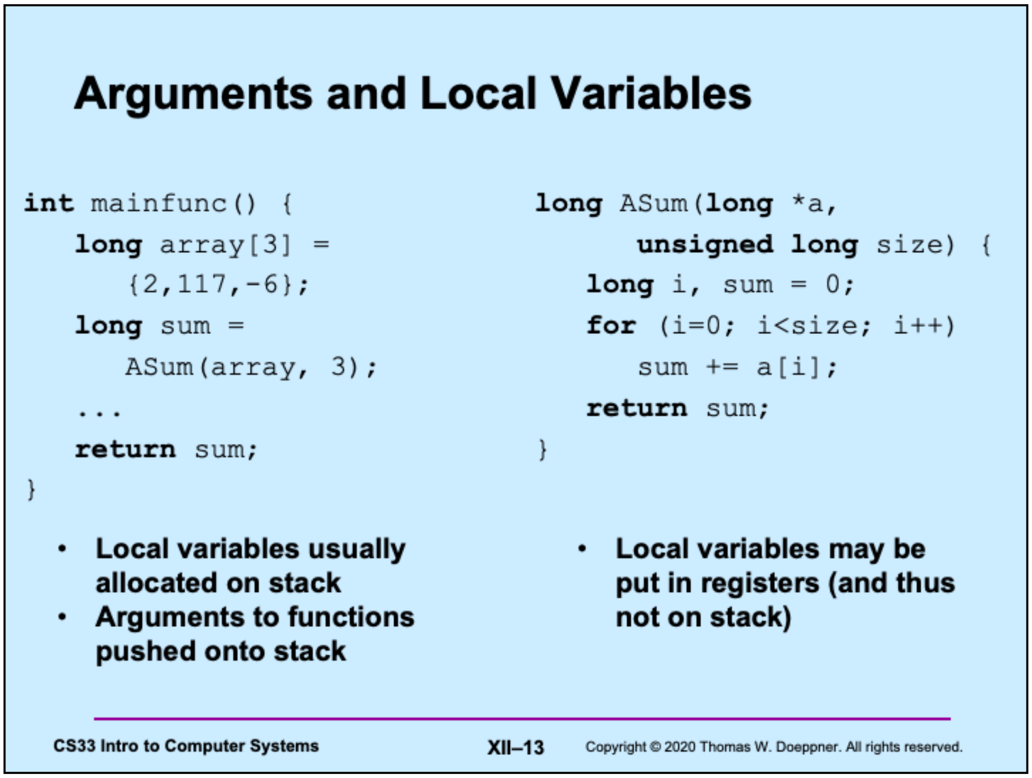
When the *call* instruction is executed, the address of the instruction after the *call* is pushed onto the stack. Thus %rsp is decremented by eight and 0x1004 is copied to the 8-byte location that is now at the end of the stack. The instruction pointer, %rip, now points to the first instruction of *func*.



Our function func puts its return value (6) into %rax, then executes the ret instruction. At this point, the address of the instruction following the call is at the end of the stack.



The address at the end of the stack (0x1004) is popped off the stack and into %rip. Thus execution resumes at the instruction following the call and %rsp is incremented by 8, The function's return value is in %rax, for access by its caller.



We explore these two functions in the next set of slides, looking at how arguments and local variables are stored on the stack.

Arguments and Local Variables

```
mainfunc:
   pushq %rbp                  # save old %rbp
   movq %rsp, %rbp             # set %rbp to point to stack frame
   subq $32, %rsp              # alloc. space for locals (array and sum)
   movq $2, -32(%rbp)          # initialize array[0]
   movq $117, -24(%rbp)        # initialize array[1]
   movq $-6, -16(%rbp)         # initialize array[2]
   pushq $3                    # push arg 2
   leaq -32(%rbp), %rax        # array address is put in %rax
   pushq %rax                  # push arg 1
   call ASum
   addq $16, %rsp              # pop args
   movq %rax, -8(%rbp)         # copy return value to sum
   addq $32, %rsp              # pop locals
   popq %rbp                   # pop and restore old %rbp
   ret
```


Here we have compiled code for *mainfunc*. We'll work through this in detail in upcoming slides.

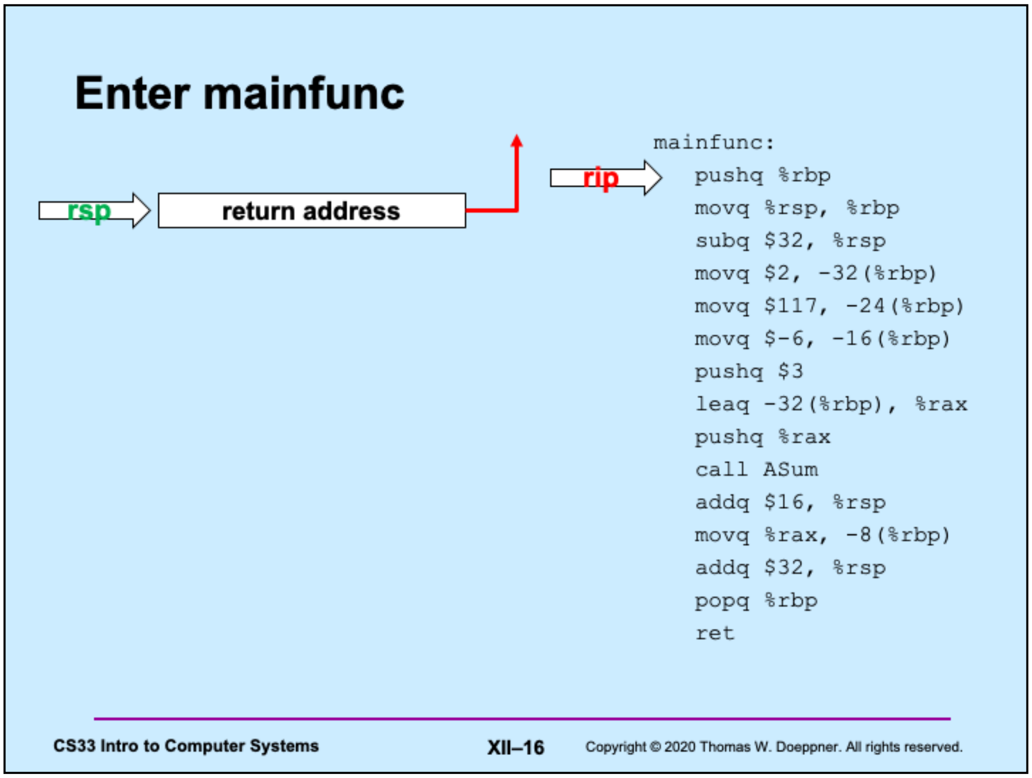
A function's stack frame is that part of the stack that holds its arguments, local variables, etc. In this example code, register %rbp points to a known location towards the beginning of the stack frame so that the arguments and local variables are located as offsets from what %rbp points to.

Note, as will be explained, this is not what one would see when compiling it for department computers, on which arguments are passed using registers.

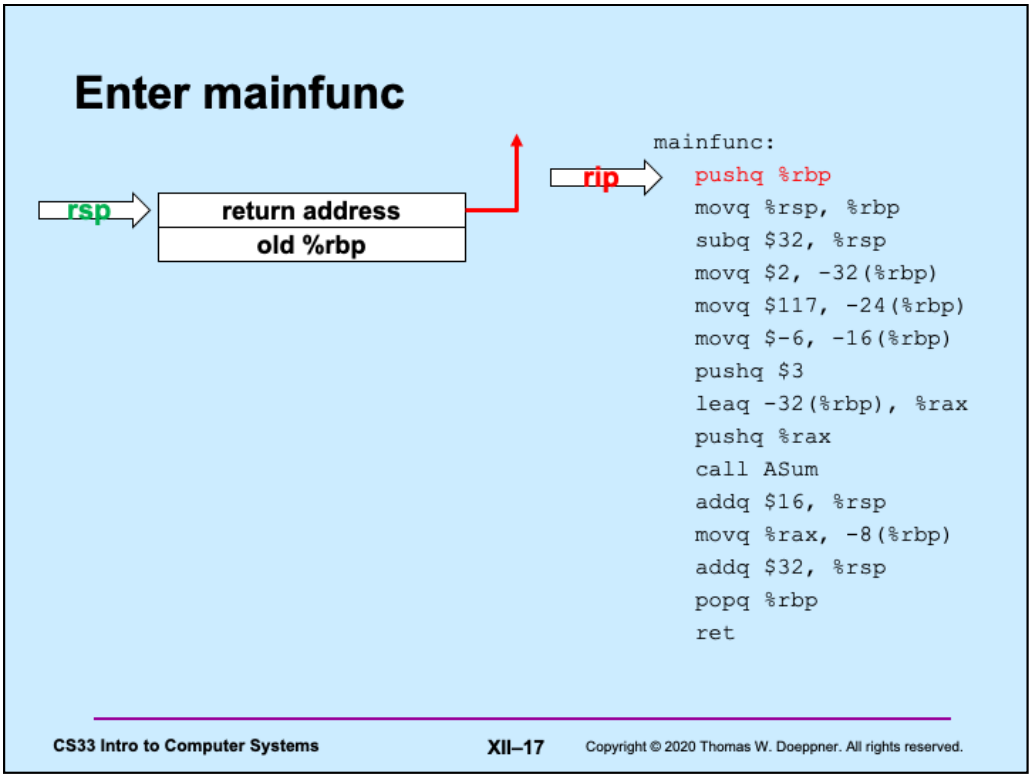
Arguments and Local Variables

```
ASum:
   pushq %rbp                  # save old %rbp
   movq %rsp, %rbp             # set %rbp to point to stack frame
   movq $0, %rcx               # i in %rcx
   movq $0, %rax               # sum in %rax
   movq 16(%rbp), %rdx         # copy arg 1 (array) into %rdx
loop:
   cmpq 24(%rbp), %rcx         # i < size?
   jge done
   addq (%rdx,%rcx,8), %rax    # sum += a[i]
   incq %rcx                   # i++
   ja loop
done:
   popq %rbp                   # pop and restore %rbp
   ret
```

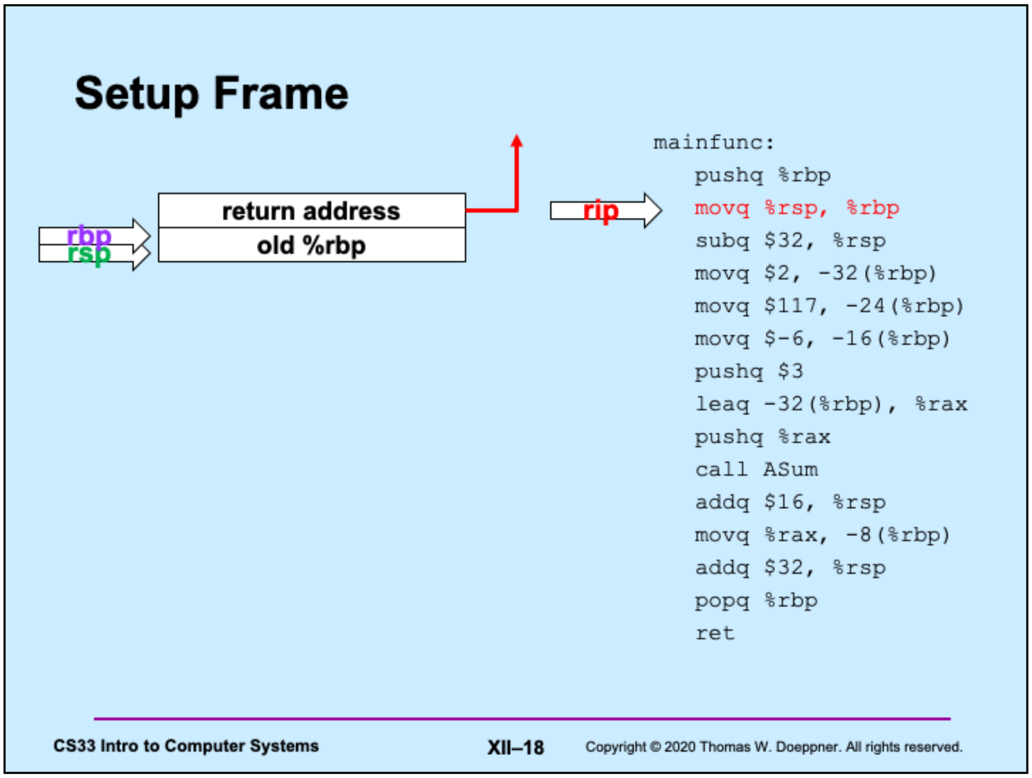

And here is the compiled code for *ASum*. The same caveats as given for the previous slide apply to this one as well.



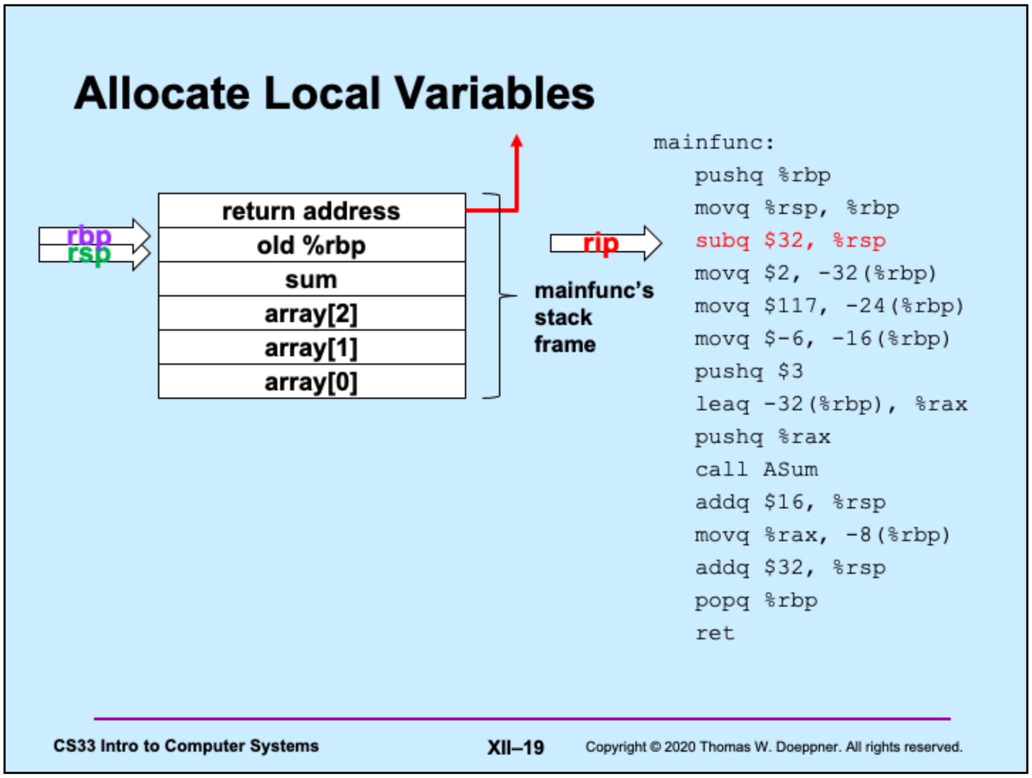
On entry to *mainfunc*, %rsp points to the caller's return address.



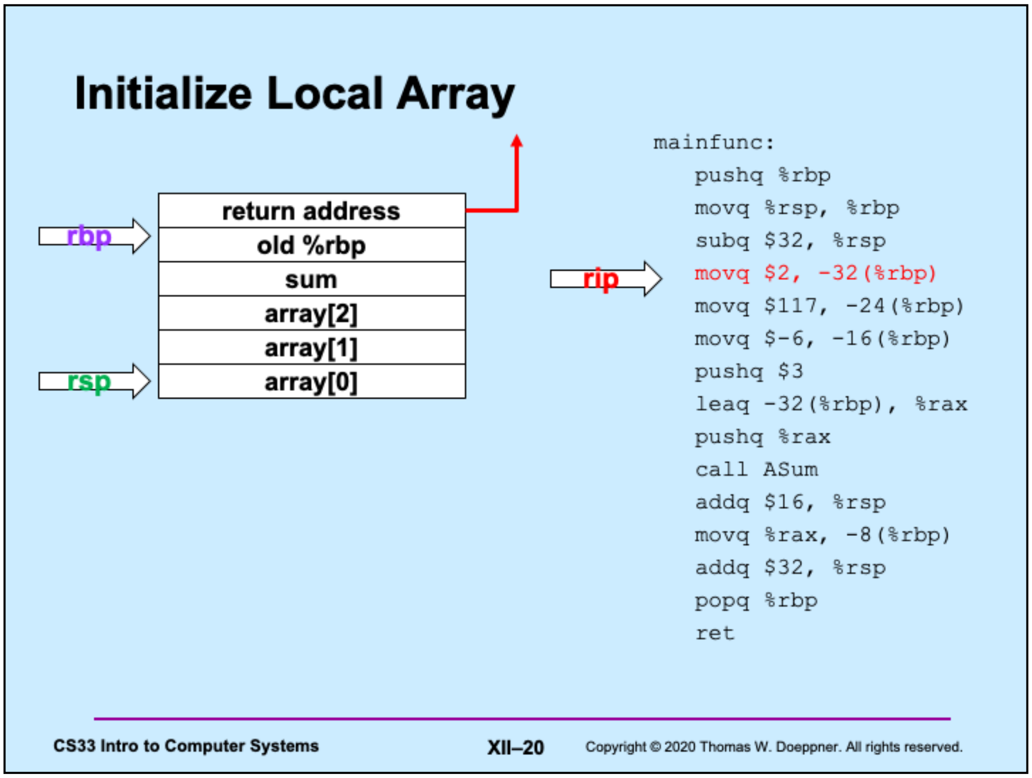
On entry to *mainfunc*, %rsp points to the caller's return address.



The first thing done by *mainfunc* is to save the caller's %rbp by pushing it onto the stack.

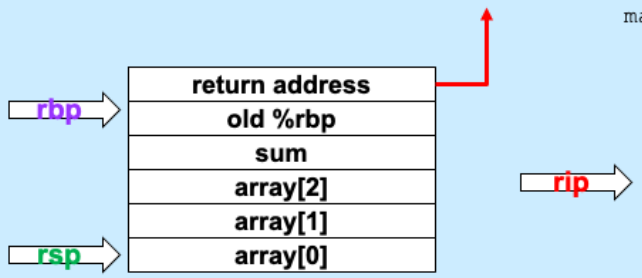


Next, space for *mainfunc*'s local variables is allocated on the stack by decrementing %rsp by their total size (32 bytes). At this point we have *mainfunc*'s stack frame in place.



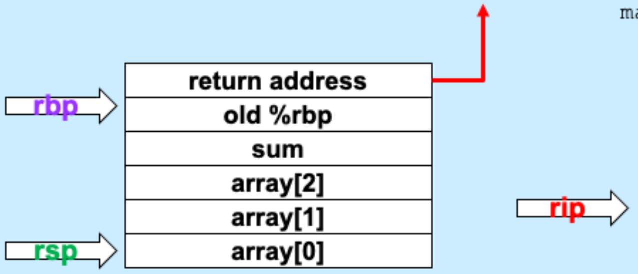
ASum now initializes the stack space containing its local variables.

Initialize Local Array

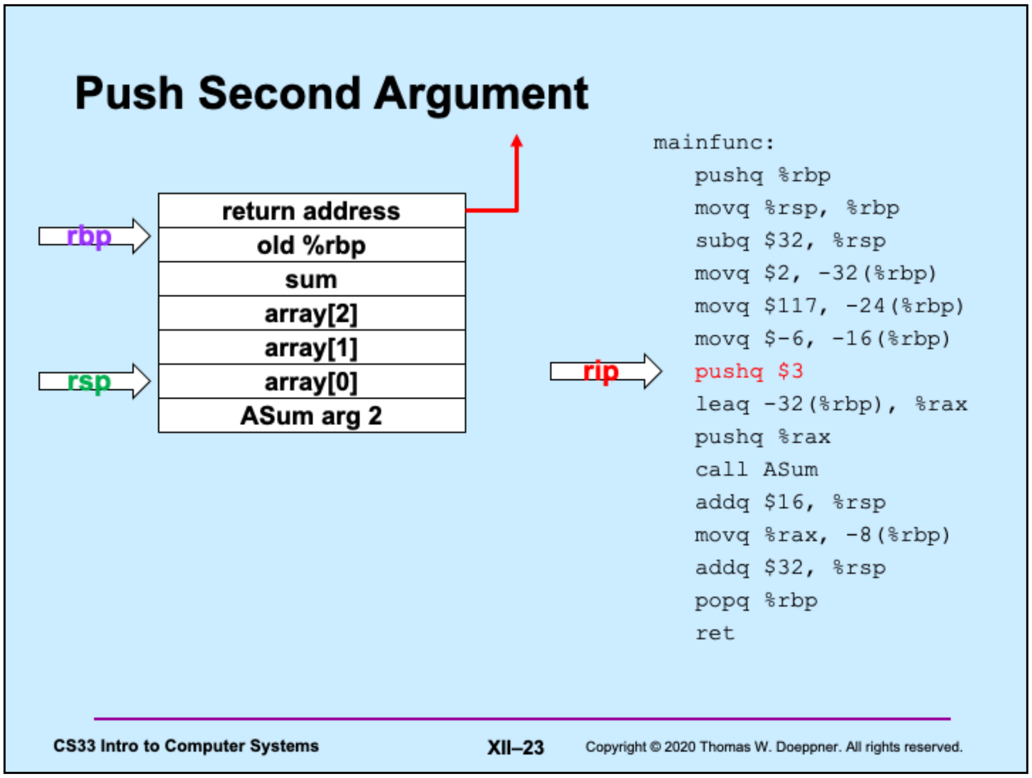



```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

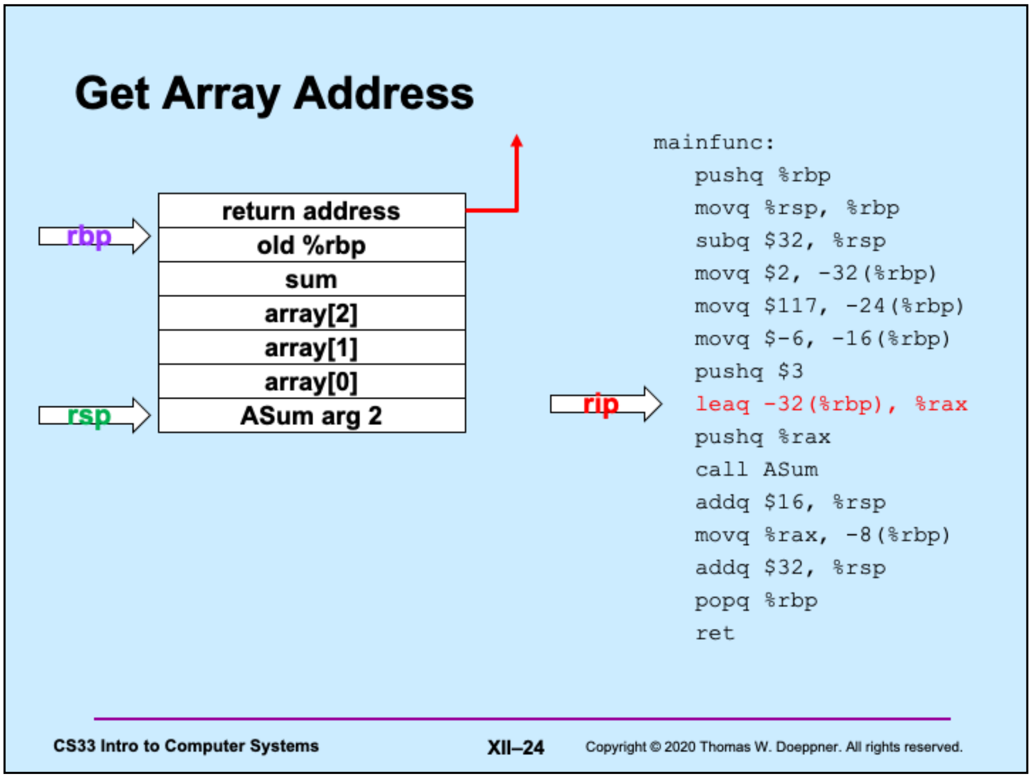
Initialize Local Array



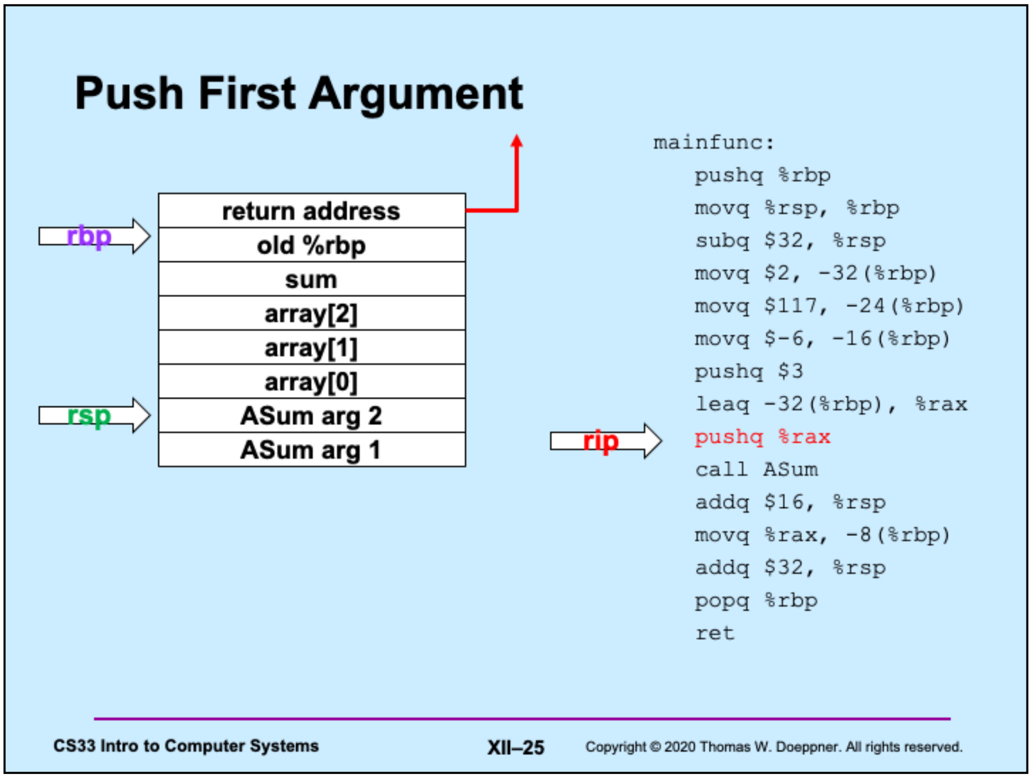

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```



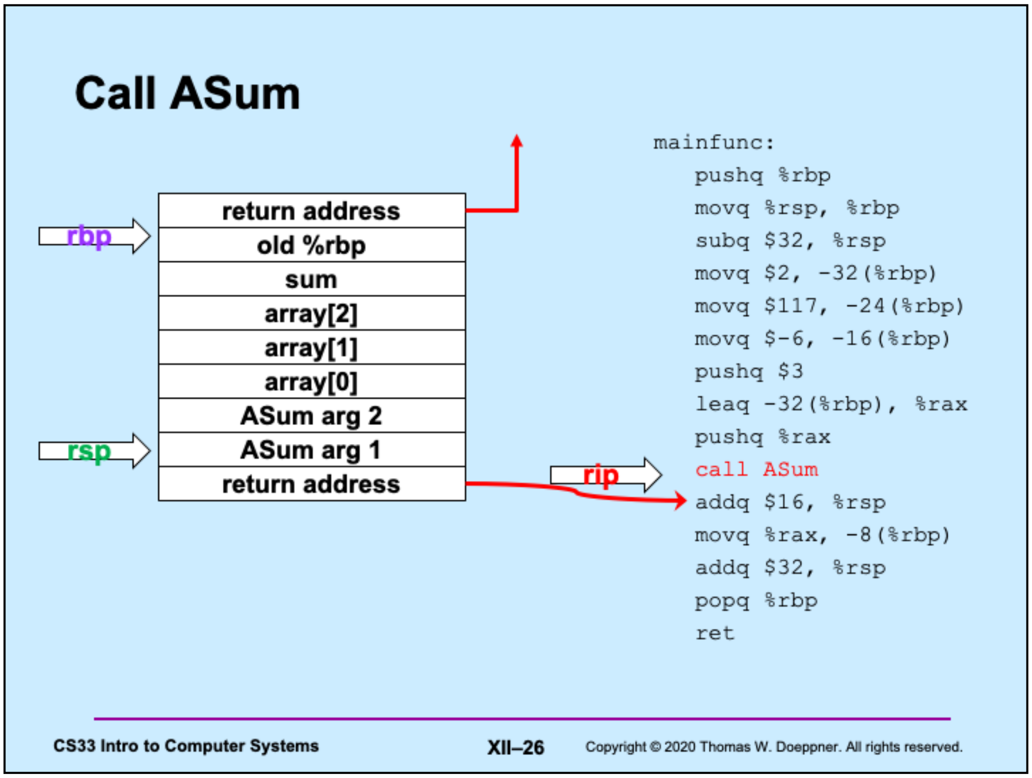
The second argument (3) to *ASum* is pushed onto the stack.



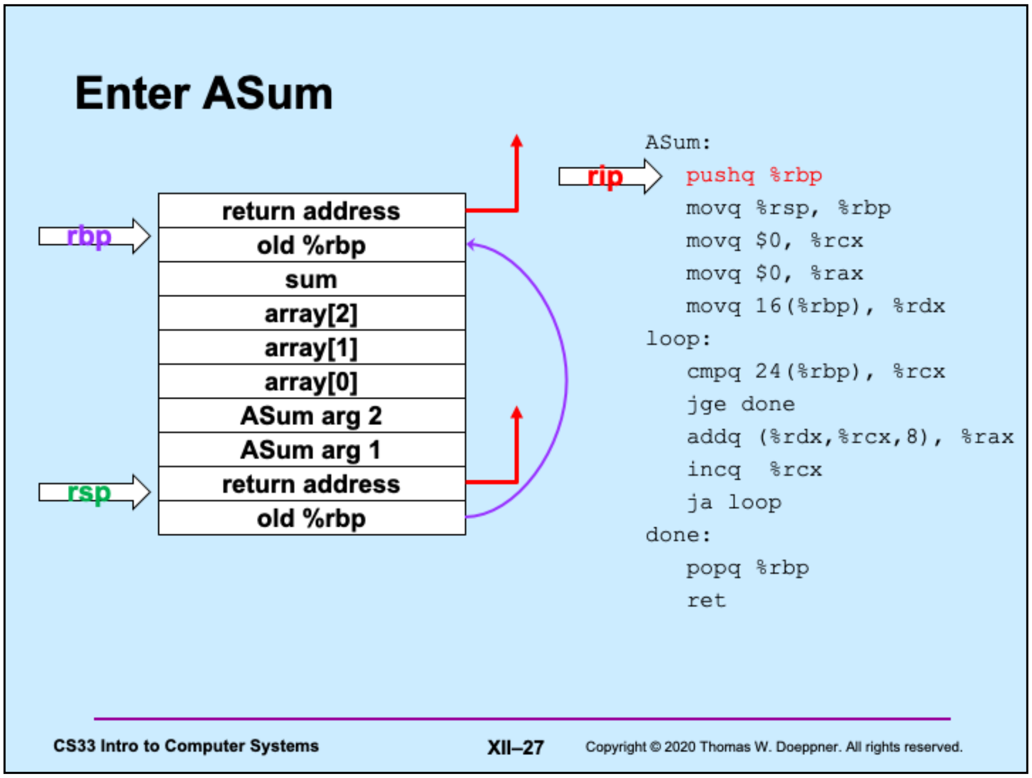
In preparation for pushing the first argument to *ASum* onto the stack, the address of the array is put into %rax.



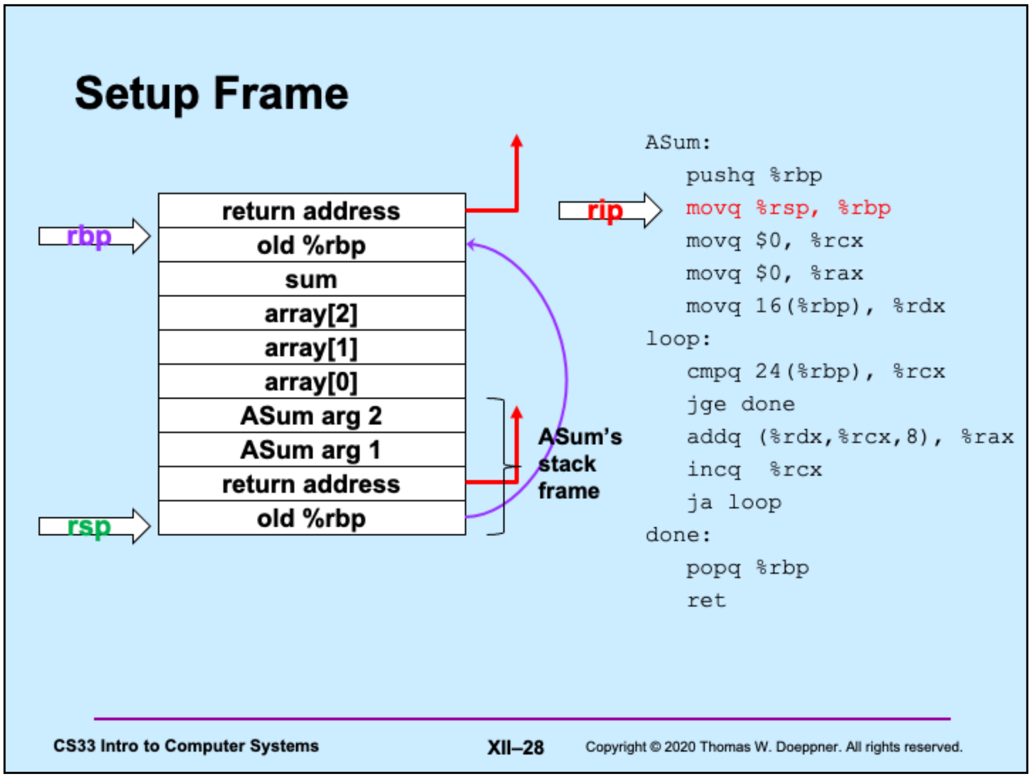
And finally, the address of the array is pushed onto the stack as *ASum*'s first argument.



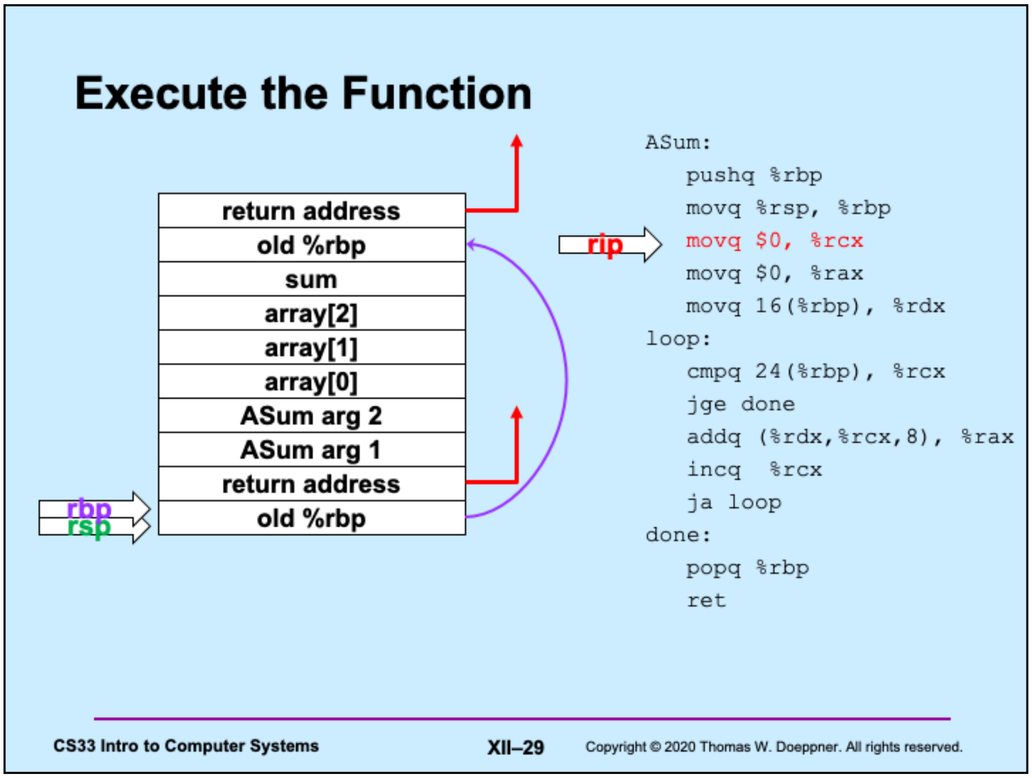
mainfunc now calls *ASum*, pushing its return address onto the stack.



As on entry to *mainfunc*, %rbp is saved by pushing it onto the stack.



%rbp is now modified to point into *ASum*'s stack frame.



ASum's code is now executed, summing the contents of its first argument and storing the result in %rax.

Quiz 1

What's at 24(%rbp)?

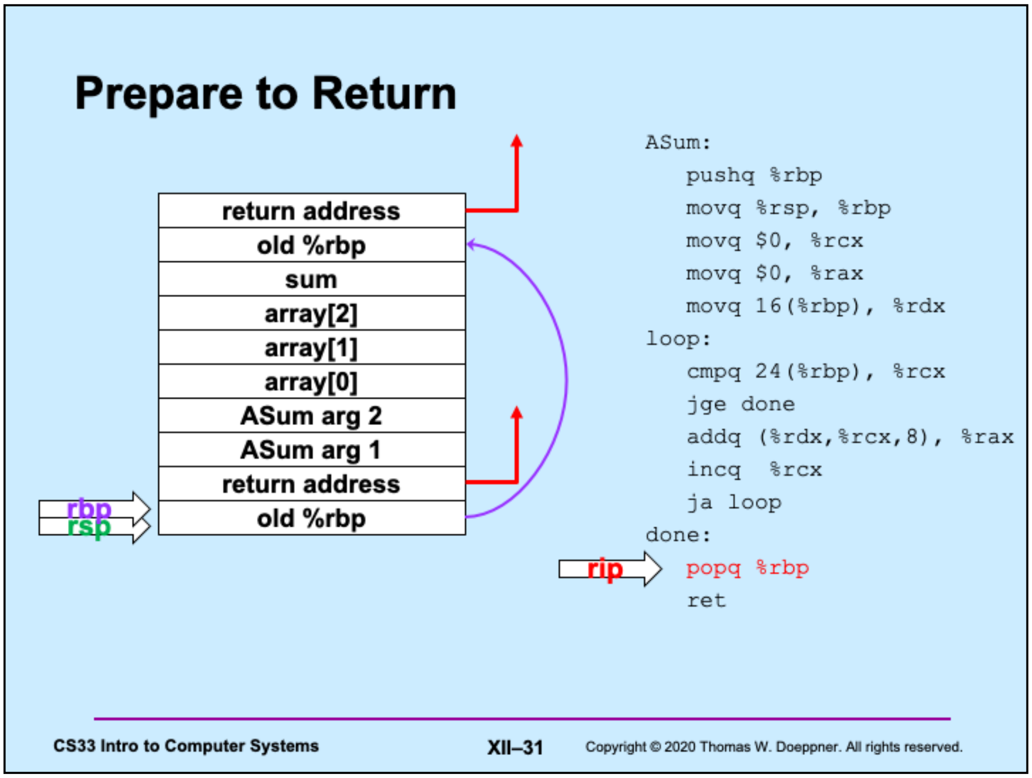
a) a local variable

b) the first argument to ASum

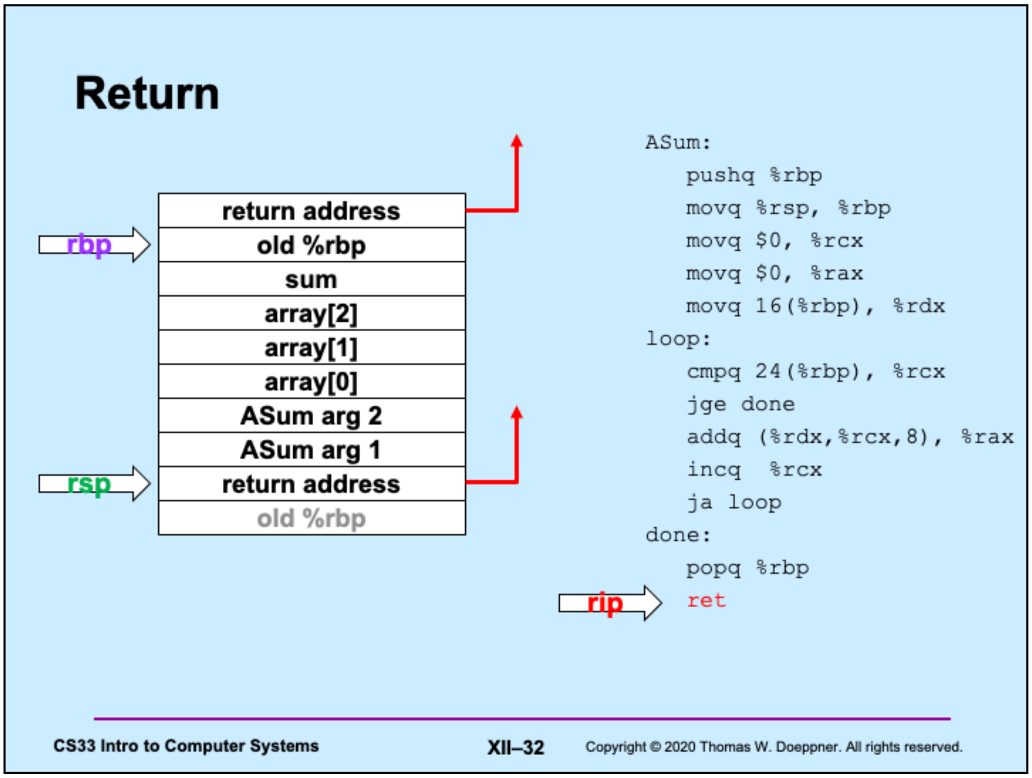
c) the second argument to ASum

d) something else

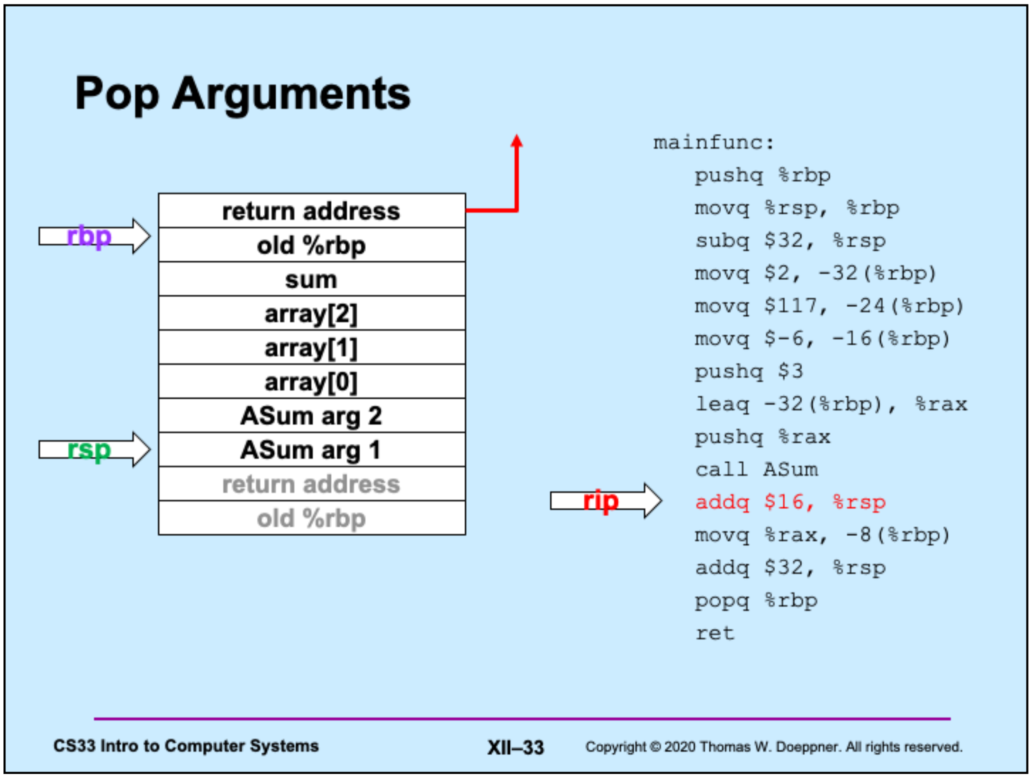
```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```



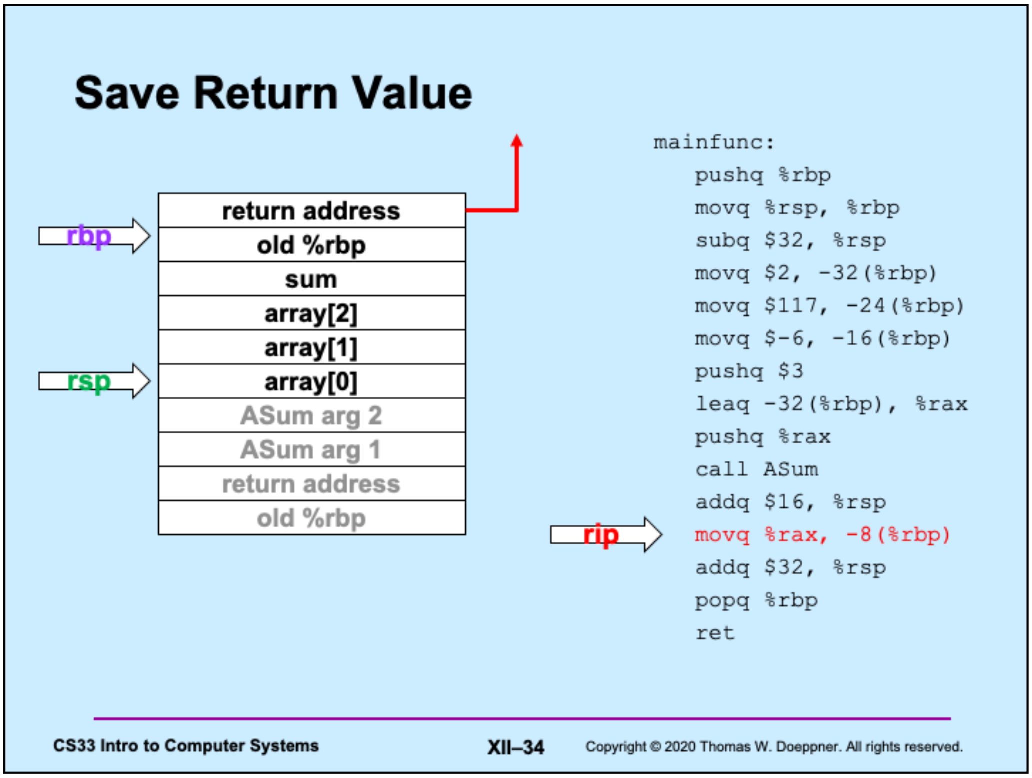
In preparation for returning to its caller, *ASum* restores the previous value of %rbp by popping it off the stack.



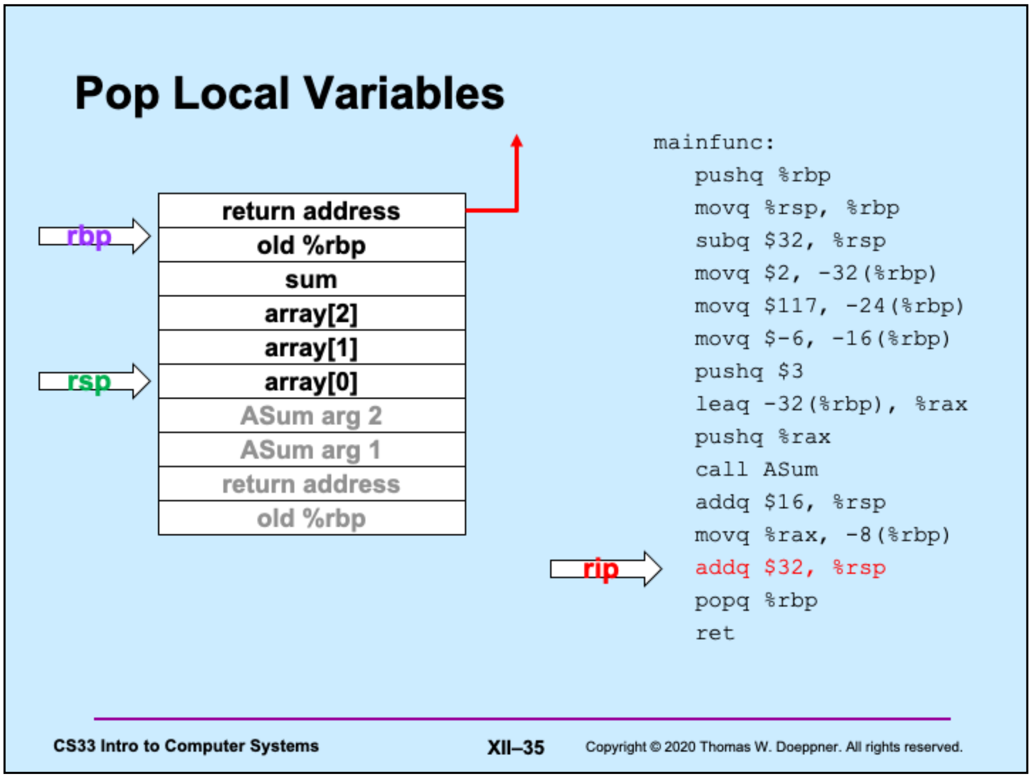
ASum returns by popping the return address off the stack and into %rip, so that execution resumes in its caller (*mainfunc*).



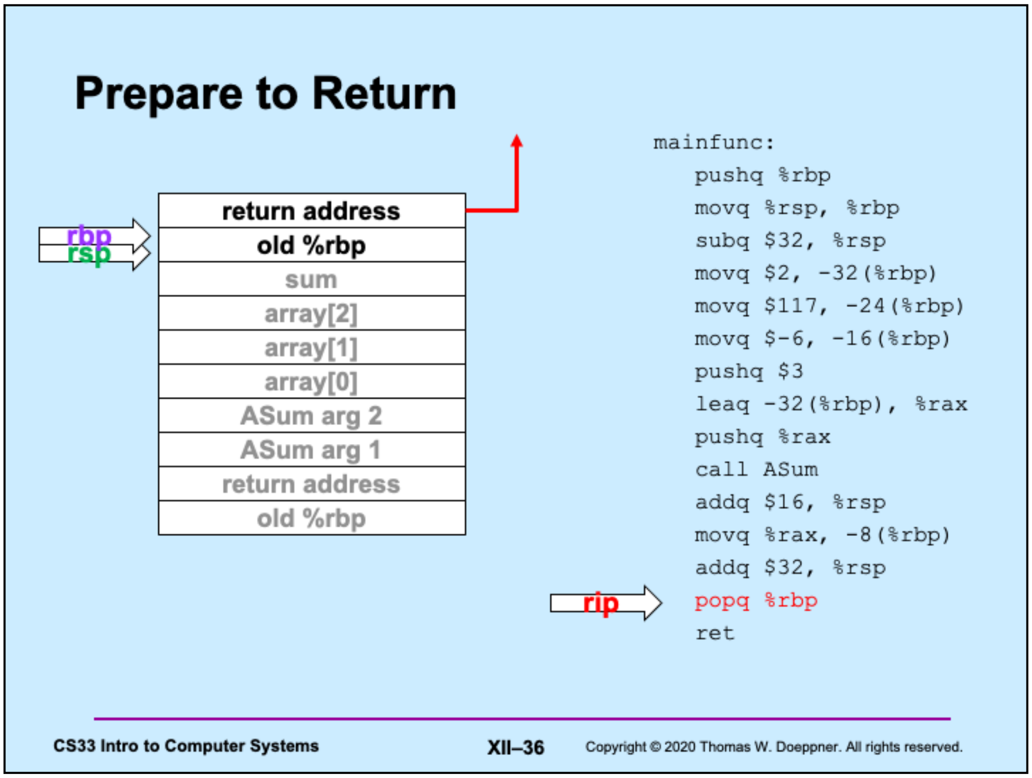
mainfunc no longer needs the arguments it had pushed onto the stack for *ASum*, so it pops them off the stack by adding their total size to %rsp.



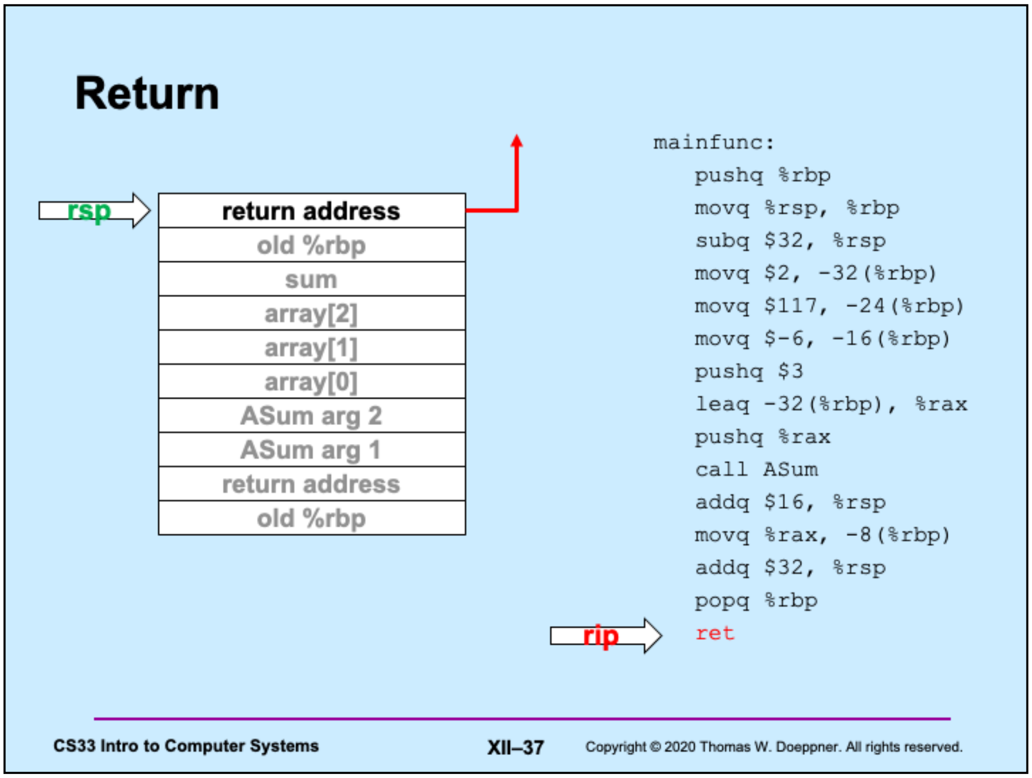
The value returned by *ASum* (in %rax) is copied into the local variable sum (which is in *mainfunc*'s stack frame).



mainfunc is about to return, so it pops its local variables off the stack (by adding their total size to %rsp).



In preparation for returning, *mainfunc* restores its caller's %rbp by popping it off the stack.



Finally, *mainfunc* returns by popping its caller's return address off the stack and into %rip.

Using Registers

- **ASum modifies registers:**
  - **%rsp**
  - **%rbp**
  - **%rcx**
  - **%rax**
  - **%rdx**
- **Suppose its caller uses these registers**

```
...
movq $33, %rcx
movq $167, %rdx
pushq $6
pushq array
call ASum
  # assumes unmodified %rcx and %rdx
addq $16, %rsp
addq %rax,%rcx    # %rcx was modified!
addq %rdx, %rcx   # %rdx was modified!
```

```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```


ASum modified a number of registers. But suppose its caller was using these registers and depended on their values' being unchanged?

Register Values Across Function Calls

- **ASum modifies registers:**
  - **%rsp**
  - **%rbp**
  - **%rcx**
  - **%rax**
  - **%rdx**
- **May the caller of ASum depend on its registers being the same on return?**
  - **ASum saves and restores %rbp and makes no net changes to %rsp**
    - » **their values are unmodified on return to its caller**
  - **%rax, %rcx, and %rdx are not saved and restored**
    - » **their values might be different on return**

```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```

Register-Saving Conventions

- **Caller-save registers**
  - if the caller wants their values to be the same on return from function calls, it must save and restore them

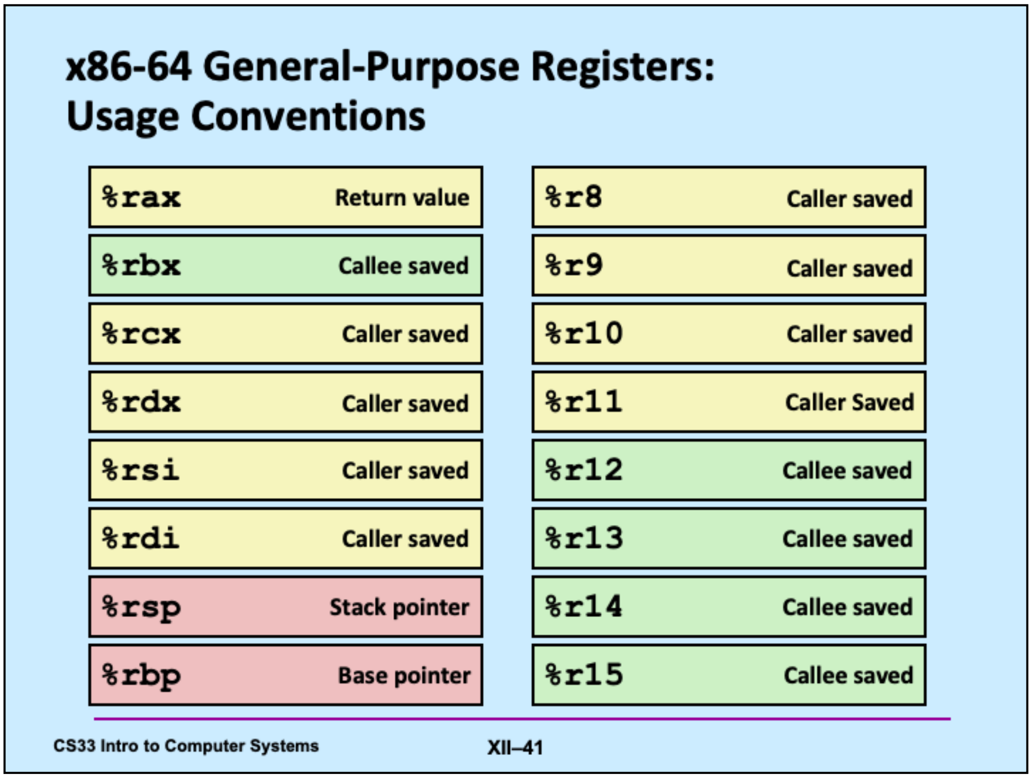
```
pushq %rcx
call func
popq %rcx
```

- **Callee-save registers**
  - if the callee wants to use these registers, it must first save them, then restore their values before returning

```
func:
    pushq %rbx
    movq $6, %rbx
    ...
    popq %rbx
```

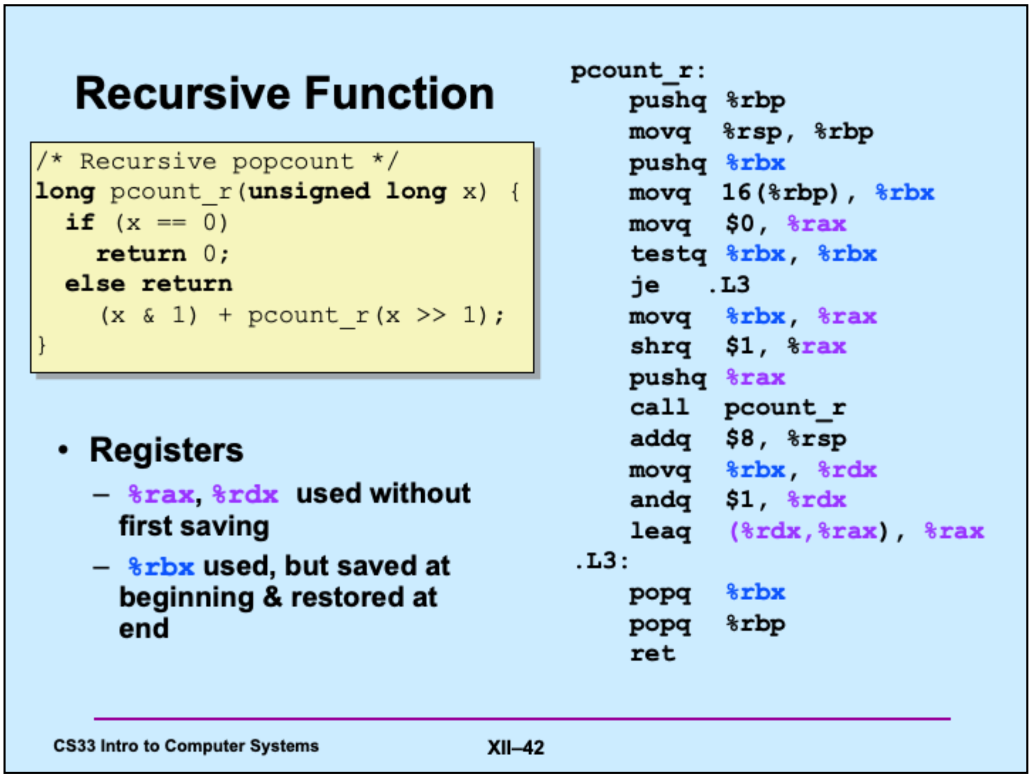

Certain registers are designated as *caller-save*: if the caller depends on their values being the same on return as they were before the function was called, it must save and restore their values. Thus the called function (the "callee"), is free to modify these registers.

Other registers are designated as *callee-save*: if the callee function modifies their values, it must restore them to their original values before returning. Thus the caller may depend upon their values being unmodified on return from the function call.



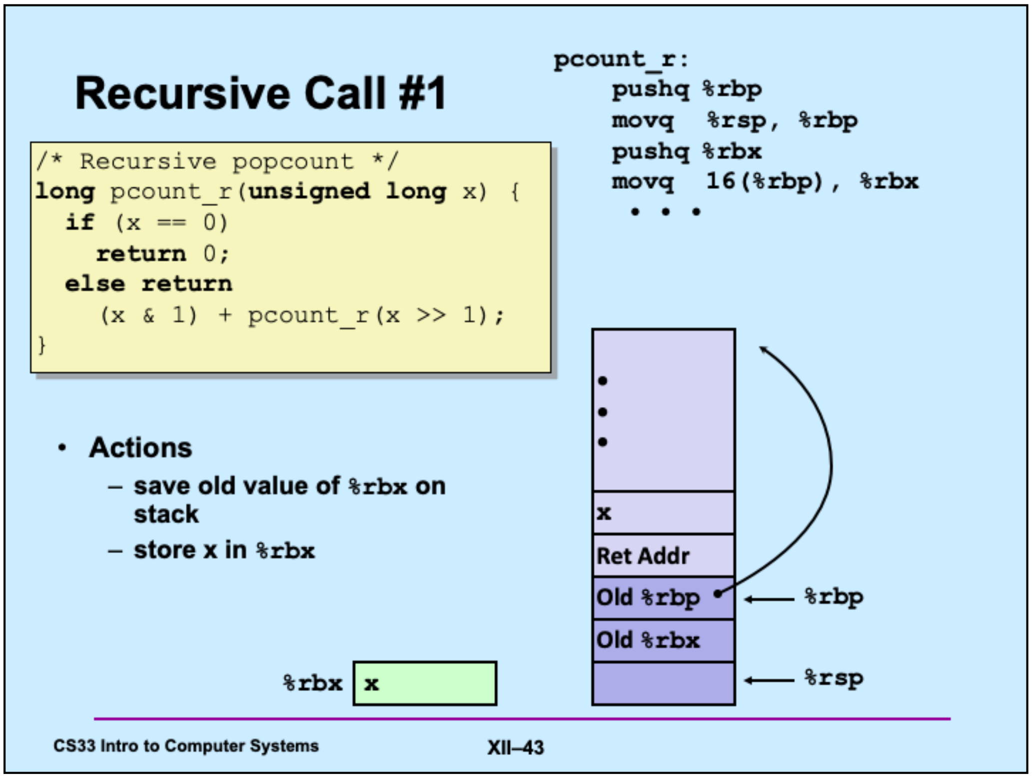
Based on a slide supplied by CMU.

Here is a list of which registers are callee-save, which are caller-save, and which have special purposes. Note that this is merely a convention and not an inherent aspect of the x86-64 architecture.

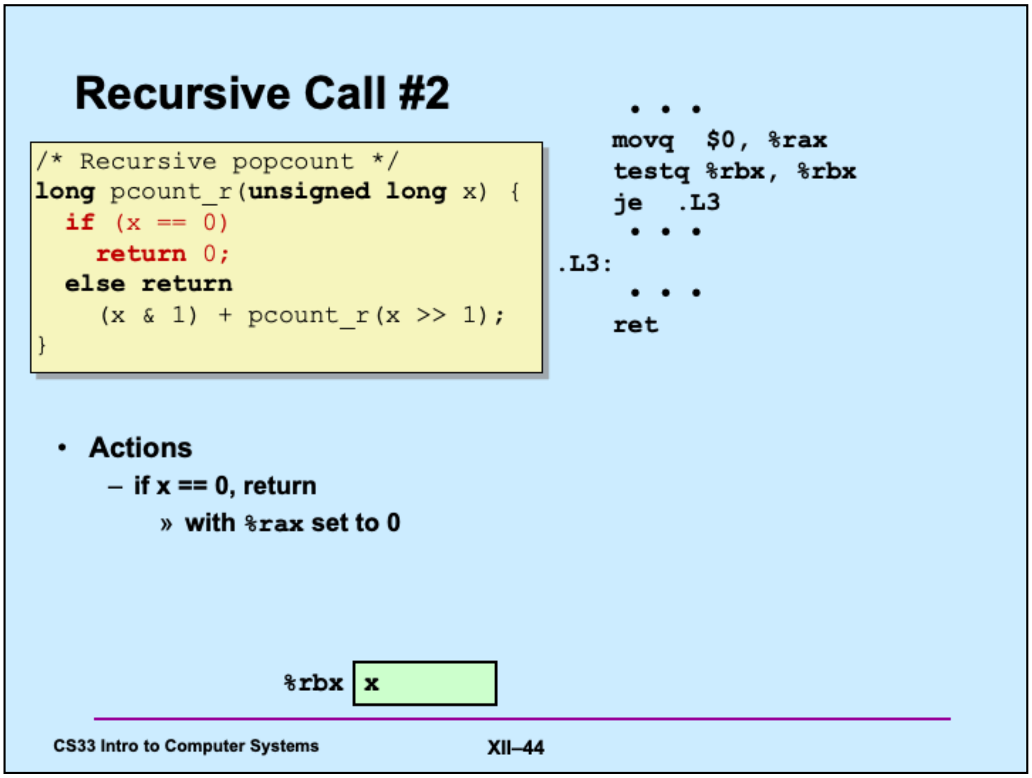


Adapted from a slide supplied by CMU.

We now walk through an example of the use of stacks for implementing recursive functions. Our example function, pcount_r, computes the number of one bits in its unsigned long argument.

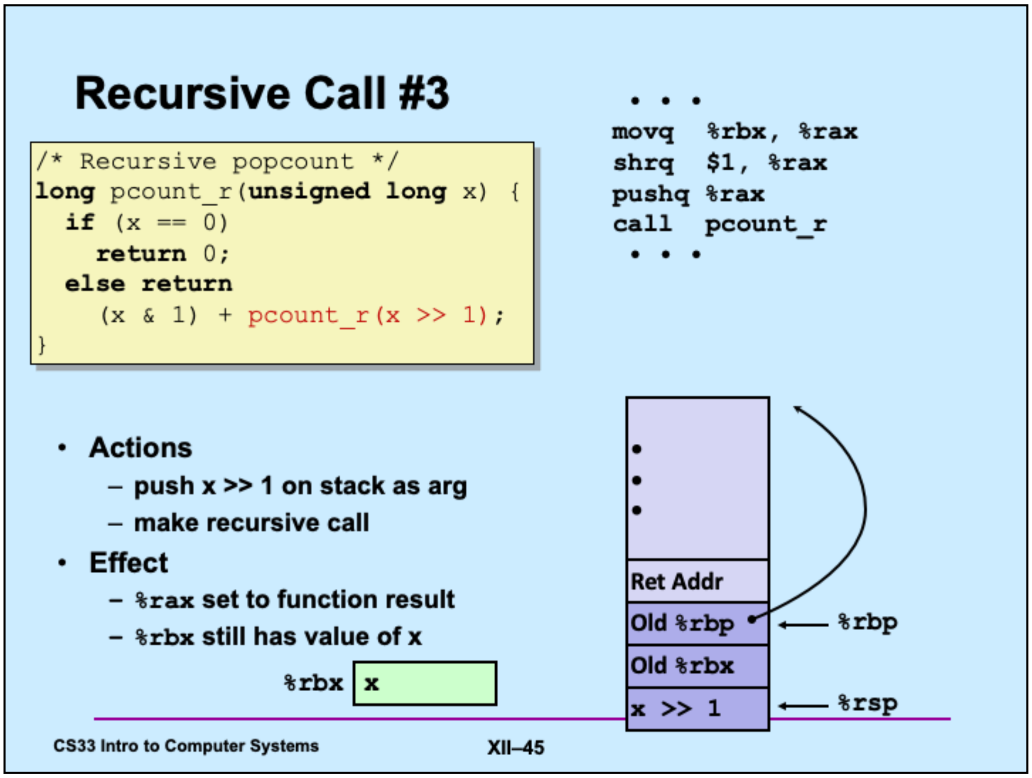


Adapted from a slide supplied by CMU.



Adapted from a slide supplied by CMU.

Recall that "testq b,a" sets the condition codes based on the value of "a&b". Thus "testq %rbx,%rbx" sets ZF to zero if and only if %rbx contains zero.



Adapted from a slide supplied by CMU.

Recursive Call #4

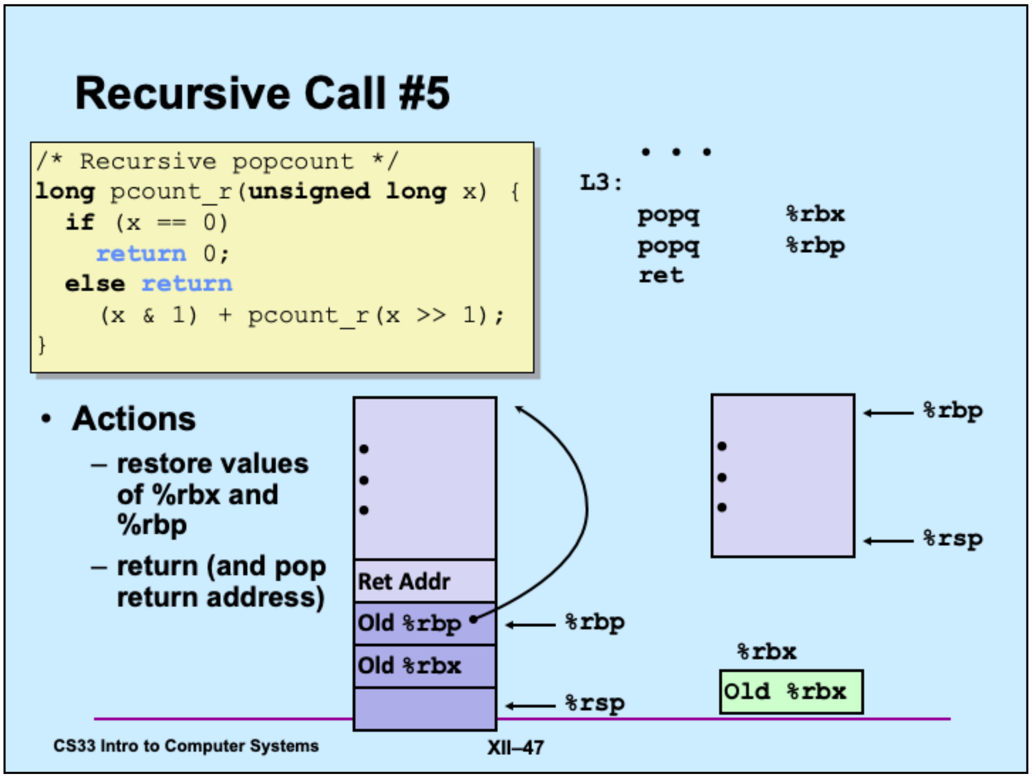
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
  • • •
movq    %rbx, %rdx
addq    $8, %rsp
andq    $1, %rdx
leaq    (%rdx,%rax), %rax
  • • •
```

- **Assume**
  - **`%rax` holds value from recursive call**
  - **`%rbx` holds x**
- **Actions**
  - **pop argument from stack**
  - **compute (x & 1) + computed value**
- **Effect**
  - **`%rax` set to function result**

%rbx	x

Adapted from a slide supplied by CMU.



Adapted from a slide supplied by CMU.

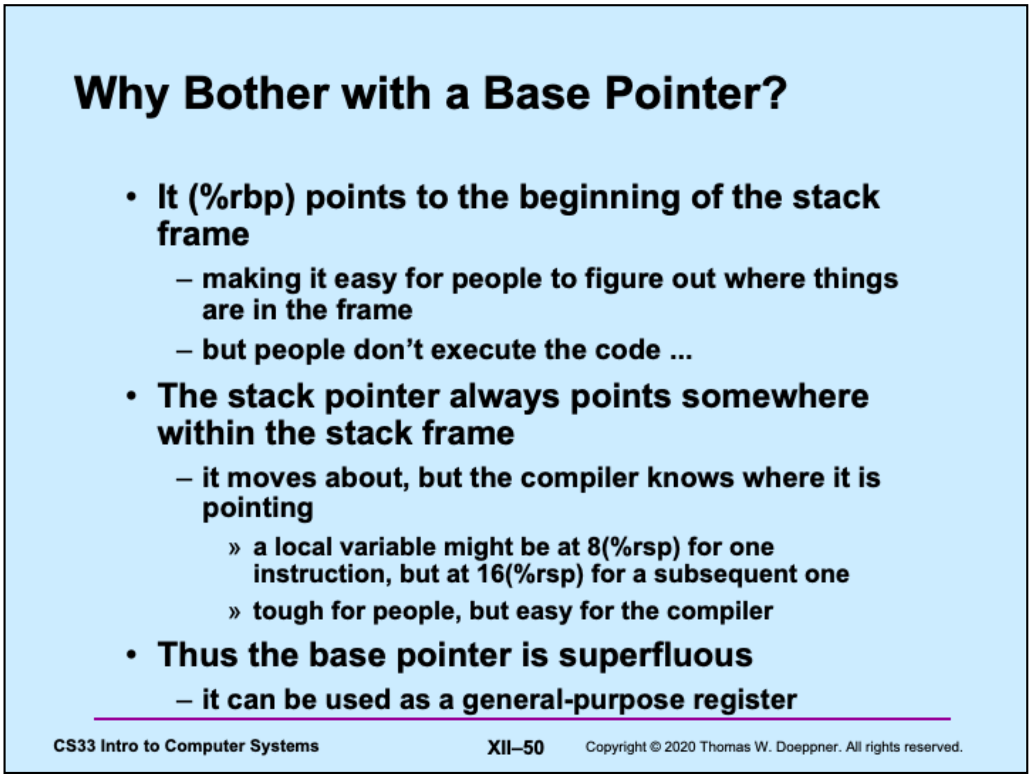
Observations About Recursion

- Handled without special consideration
  - stack frames mean that each function call has private storage
    - » saved registers & local variables
    - » saved return pointer
  - register-saving conventions prevent one function call from corrupting another's data
  - stack discipline follows call / return pattern
    - » if P calls Q, then Q returns before P
    - » last-in, first-out
- Also works for mutual recursion
  - P calls Q; Q calls P

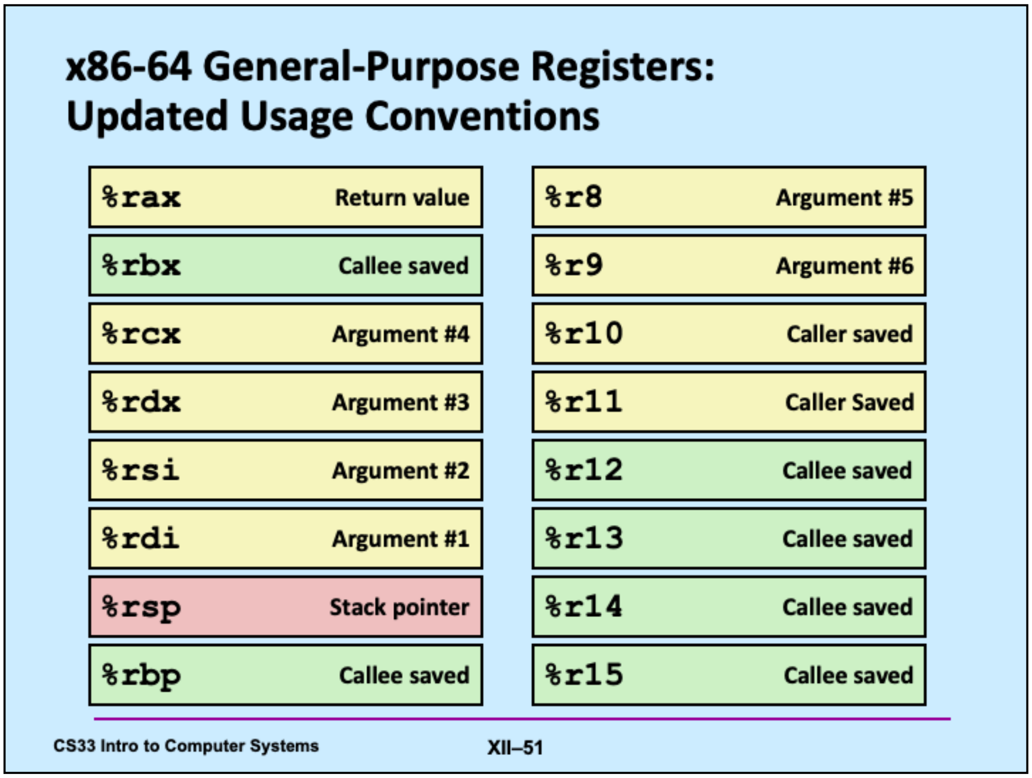
Supplied by CMU.

Passing Arguments in Registers

- Observations
  - accessing registers is much faster than accessing primary memory
    - » if arguments were in registers rather than on the stack, speed would increase
  - most functions have just a few arguments
- Actions
  - change calling conventions so that the first six arguments are passed in registers
    - » in caller-save registers
  - any additional arguments are pushed on the stack



If one gives gcc the –O0 flag (which turns off all optimization) when compiling, the base pointer (%rbp) will be used as in IA32: it is set to point to the stack frame and the arguments are copied from the registers into the stack frame. This clearly slows down the execution of the function, but makes the code easier for humans to read (and was done for the traps assignment).



Supplied by CMU.

Recursive Function (Improved)

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
   movq    $0, %rax
   testq   %rdi, %rdi
   je     .L8
   pushq   %rbx
   movq    %rdi, %rbx
   shrq    $1, %rdi
   call    pcount_r
   andq    $1, %rbx
   addq    %rbx, %rax
   popq    %rbx
.L8:
   ret
```

- **Registers**
  - the single argument (x) is passed in %rdi
  - %rbx is a callee-saved register and thus is saved and restored
  - %rax is caller-saved
  - %rbp isn't used

The assembler code in the slide is from gcc with the –O1 flag (turning on a small amount of optimization). The argument is passed in %rdi and there is no use of %rbp as a base pointer.

Summary

- What's pushed on the stack
  - return address
  - saved registers
    - » caller-saved by the caller
    - » callee-saved by the callee
  - local variables
  - function parameters
    - » those too large to be in registers (structs)
    - » those beyond the six that we have registers for
  - large return values (structs)
    - » caller allocates space on stack
    - » callee copies return value to that space

Quiz 2

Suppose function A is compiled using the convention that %rbp is used as the base pointer, pointing to the beginning of the stack frame. Function B is compiled using the convention that there's no need for a base pointer. Will there be any problems if A calls B or if B calls A?

a) Neither case will work
b) A calling B works, but B calling A doesn't
c) B calling A works, but A calling B doesn't
d) Both work

Tail Recursion

```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```


The slide shows two implementations of the factorial function. Both use recursion. In the version on the left, the result of each recursive call is used within the invocation that issued the call. In the second, the result of each recursive call is simply returned. This is known as *tail recursion*.

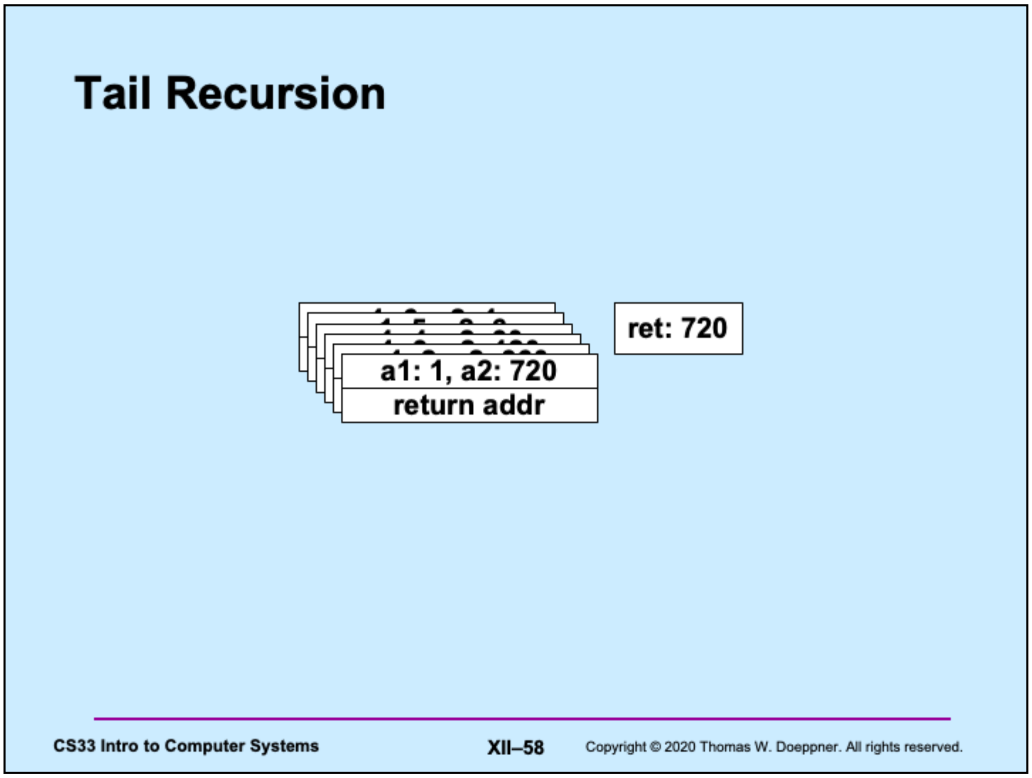
No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

Here we look at the stack usage for the version without tail recursion. Note that we have as many stack frames as the value of the argument; the results of the calls are combined after the stack reaches its maximum size.

No Tail Recursion (2)

x: 6	ret: 720
return addr	
x: 5	ret: 120
return addr	
x: 4	ret: 24
return addr	
x: 3	ret: 6
return addr	
x: 2	ret: 2
return addr	
x: 1	ret: 1
return addr	



With tail recursion, since the result of the recursive call is not used by the issuing stack frame, it's possible to reuse the issuing stack frame to handle the recursive invocation. Thus rather than push a new stack frame on the stack, the current one is written over. Thus the entire sequence of recursive calls can be handled within a single stack frame.

Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2       # recursive call!
        addq    $8, %rsp
.L5:
        ret
```


This is the result of compiling the tail-recursive version of factorial using gcc with the –O1 flag. This flags turns on a moderate level of code optimization, but not enough to cause the stack frame to be reused.

Not Using the Stack ...

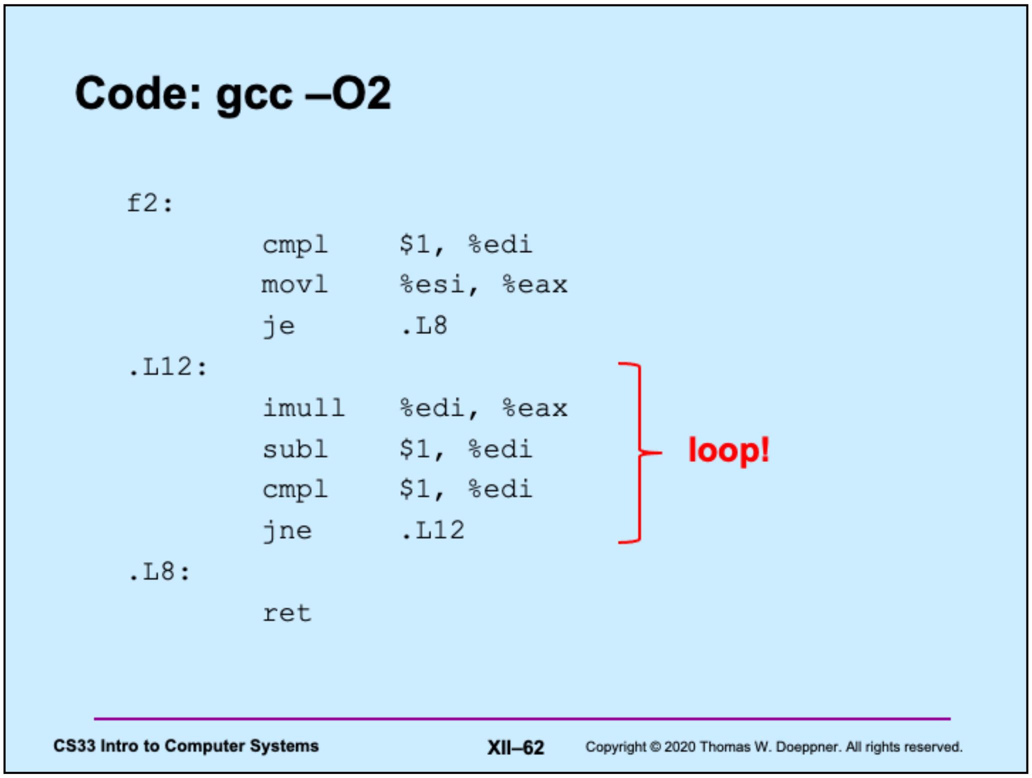
```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
.L5:
        ret
```


The code that moved the stack pointer down and then up again is clearly not needed, so we've removed it.

Not Recursive!

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        ja      f2          # goto!
.L5:
        ret
```


Since nothing is pushed onto the stack and nothing is popped from the stack – the code does everything in the registers %eax, %edi, and %esi -- there's really no need to call f2 recursively. Instead, it can simply jump to f2.



Here we've compiled the program using the –O2 flag, which turns on additional optimization (at the cost of increased compile time), with the result that the recursive calls are automatically optimized away — they are replaced with a loop.

Why not always compile with –O2? For "production code" that is bug-free (assuming this is possible), this is a good idea. But this and other aggressive optimizations make it difficult to relate the runtime code with the source code. Thus, a runtime error might occur at some point in the program's execution, but it is impossible to determine exactly which line of the source code was in play when the error occurred.

Note that gcc can do some very impressive optimization when given the –O2 flag. For example, in both the "normal" recursive implementation of factorial, as well as the pcount_r function discussed earlier, it produces assembler code that uses loops rather than recursive calls.