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.

Not a Quiz!

What C code would you compile to get the following assembler code?

```
        movq    $0, %rax
.L2:
        movq    %rax, a(,%rax,8)
        addq    $1, %rax
        cmpq    $10, %rax
        jne     .L2
        ret
```

```
long a[10];
void func() {
  long i=0;
  while (i<10)
    a[i]= i++;
}
```

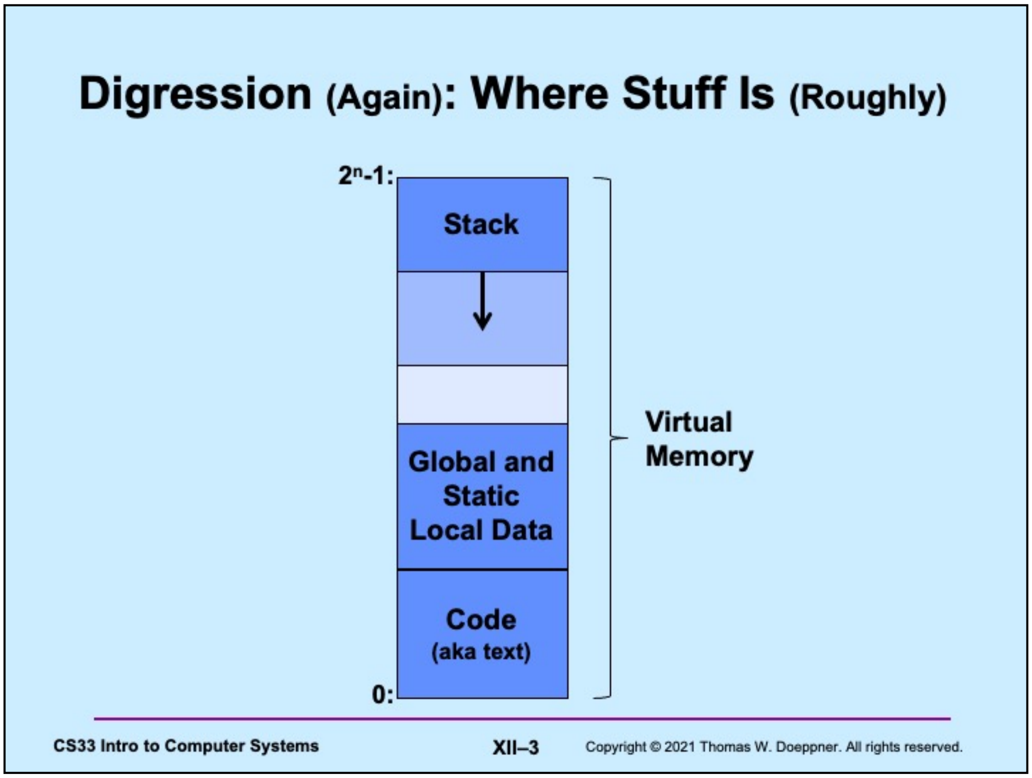
a

```
long a[10];
void func() {
  long i;
  for (i=0; i<10; i++)
    a[i]= 1;
}
```

b

```
long a[10];
void func() {
  long i=0;
  switch (i) {
case 0:
    a[i] = 0;
    break;
default:
    a[i] = 10
  }
}
```

c



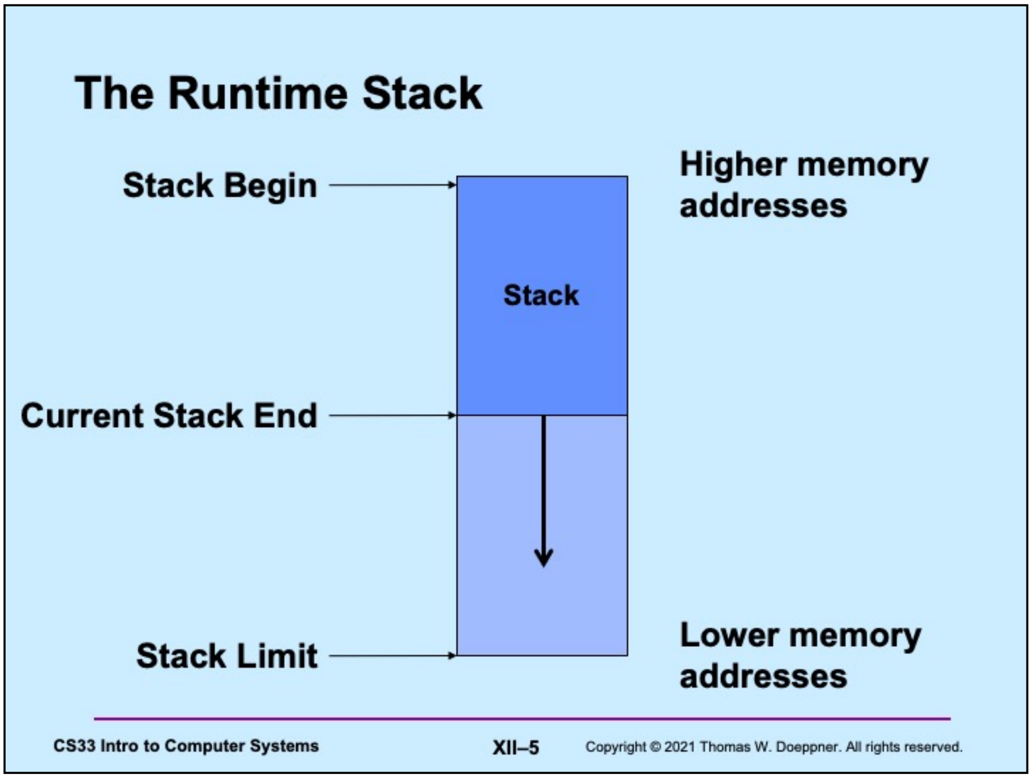
Here we revisit the slide we saw a few weeks ago, this time drawing it with high addresses at the top and low addresses at the bottom. The point is that a large amount of virtual memory is reserved for the stack. In most cases there's plenty of room for the stack and we don't have to worry about exceeding 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

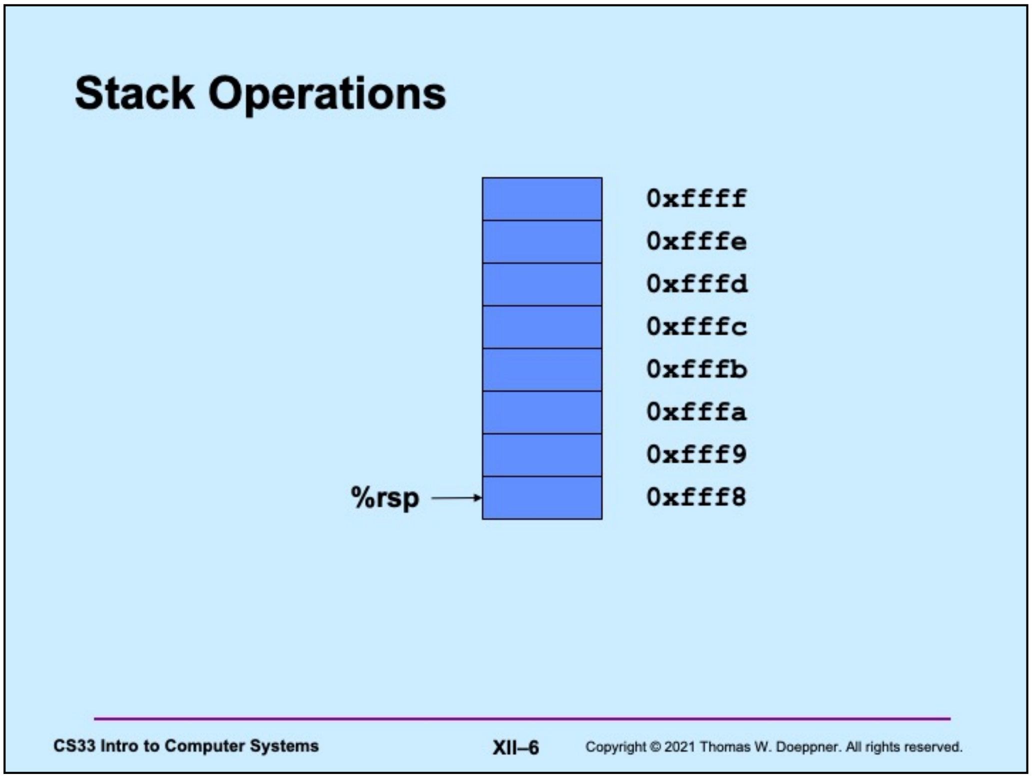
- Function A calls function B
- Function B calls function C

  ... several million instructions later

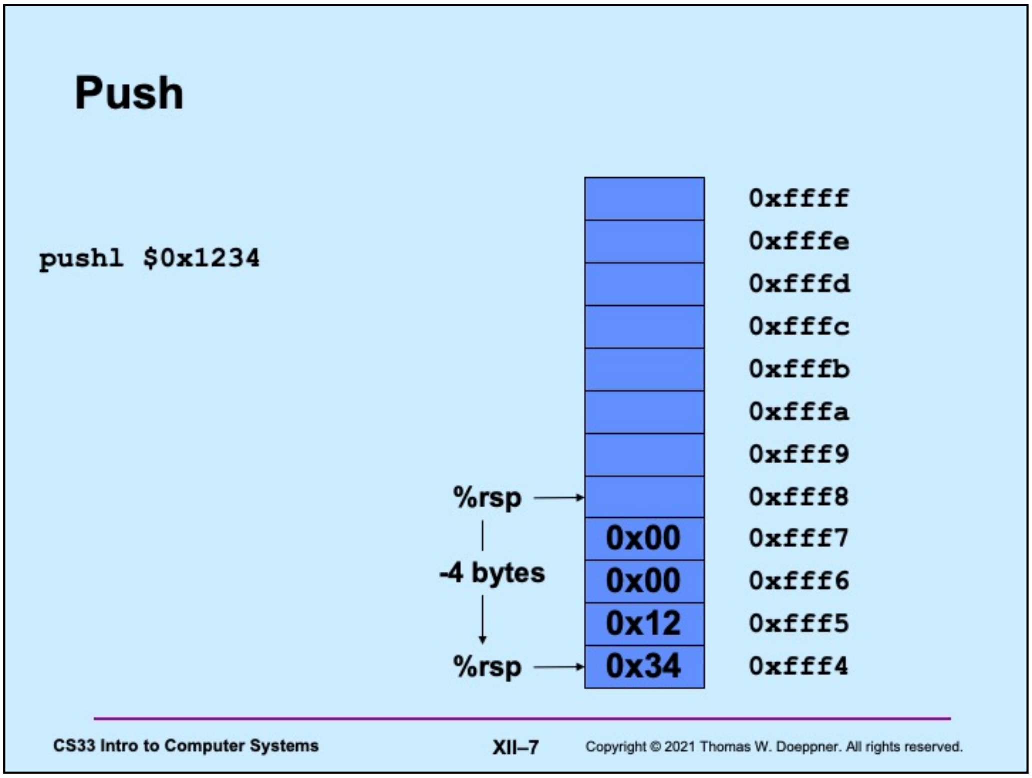
- C returns
  – how does it know to return to B?
- B returns
  – how does it know to return to A?



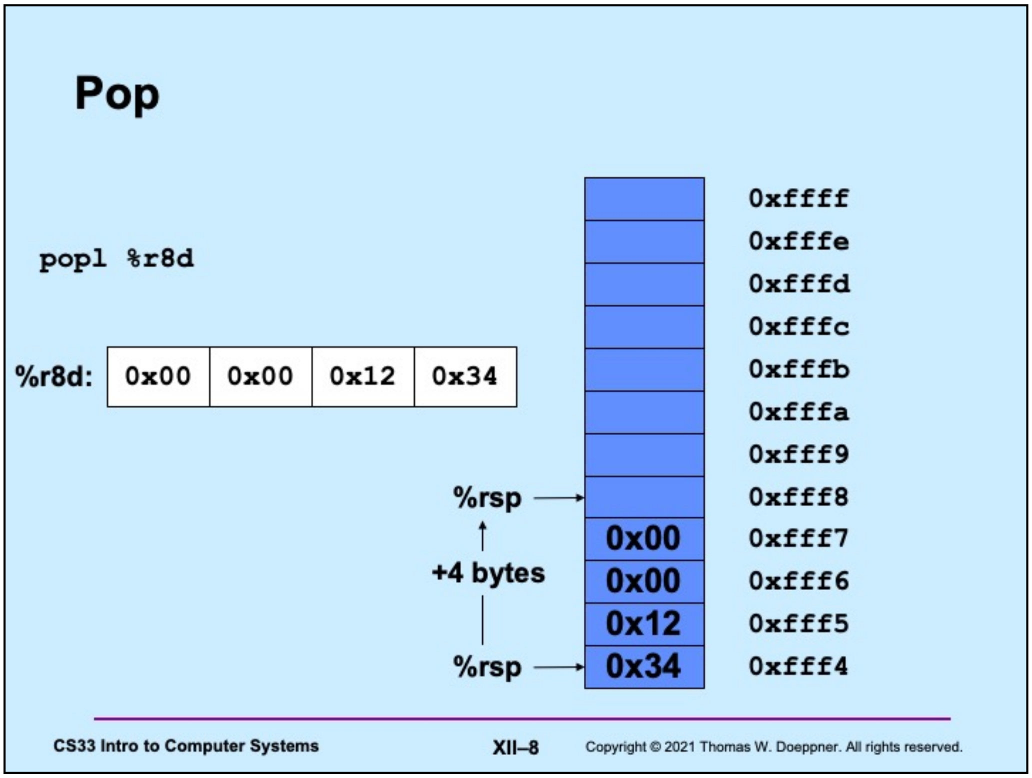
Stacks, as implemented on the X86 for most operating systems (and, in particular, Linux, OSX, and Windows) grow "downwards", from high memory addresses to low memory addresses. To avoid confusion, we will not use the works "top of stack" or "bottom of stack" but will instead use "stack begin" and "current 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 execute **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.



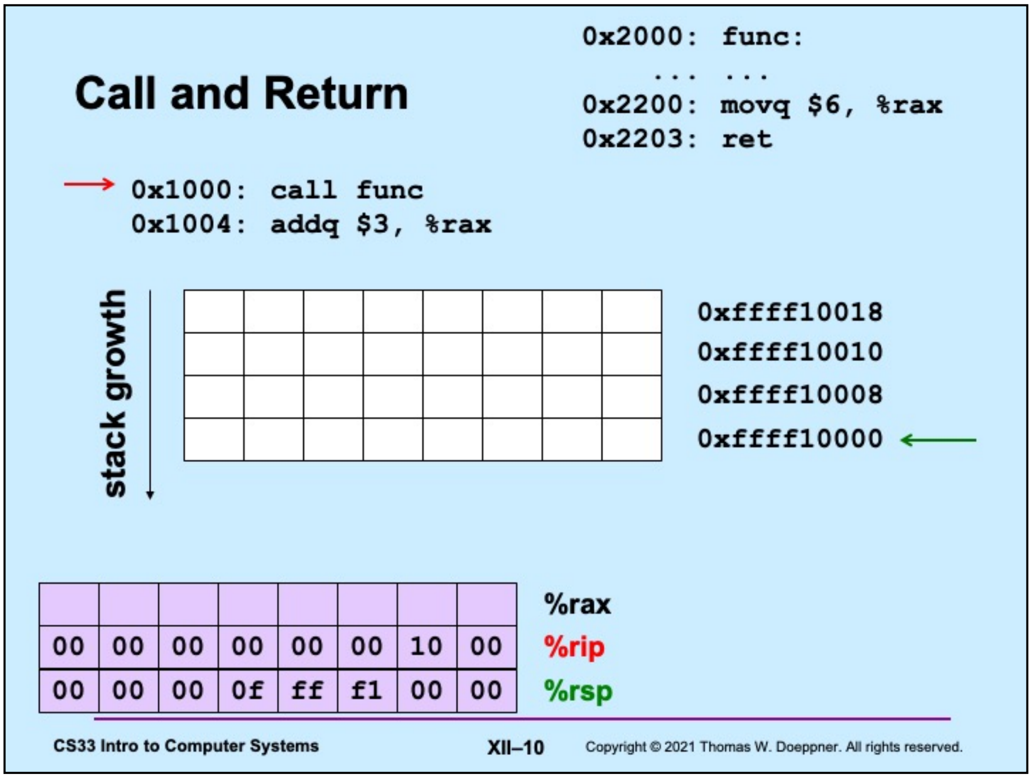
Here we pop an item off the stack. The **popl** instruction copies the 4-byte item pointed to by %rsp into its argument, then increments %rsp by 4.

Call and Return

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

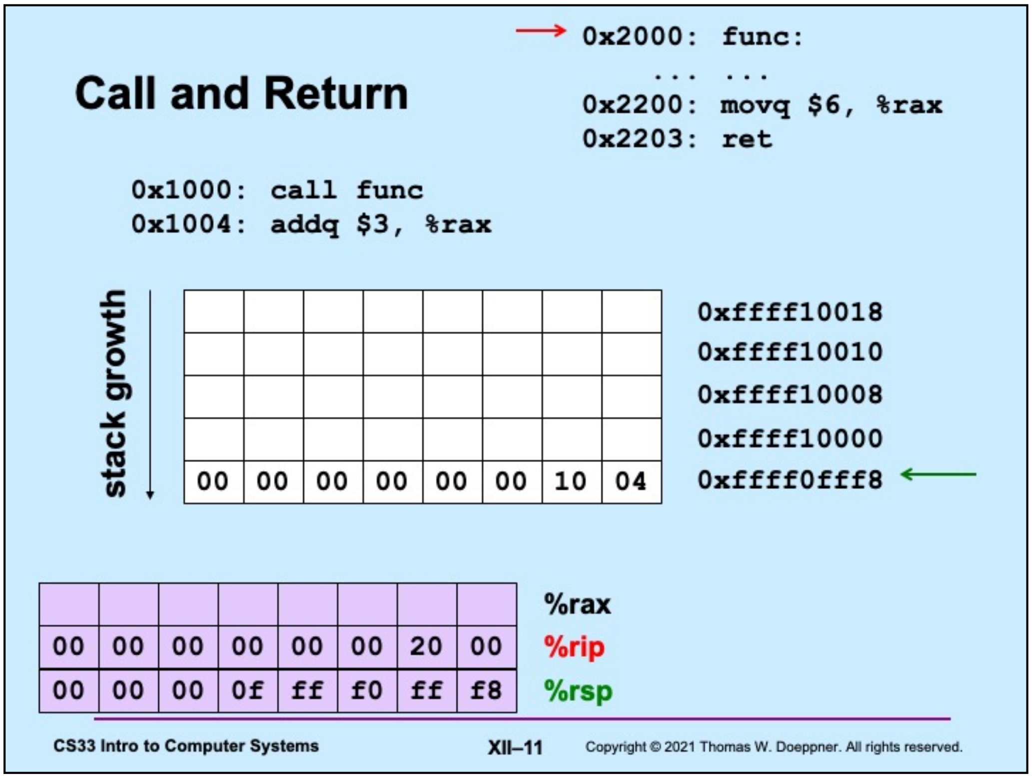
```
0x2000: func:
    ... ...
0x2200: movq $6, %rax
0x2203: ret
```


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.

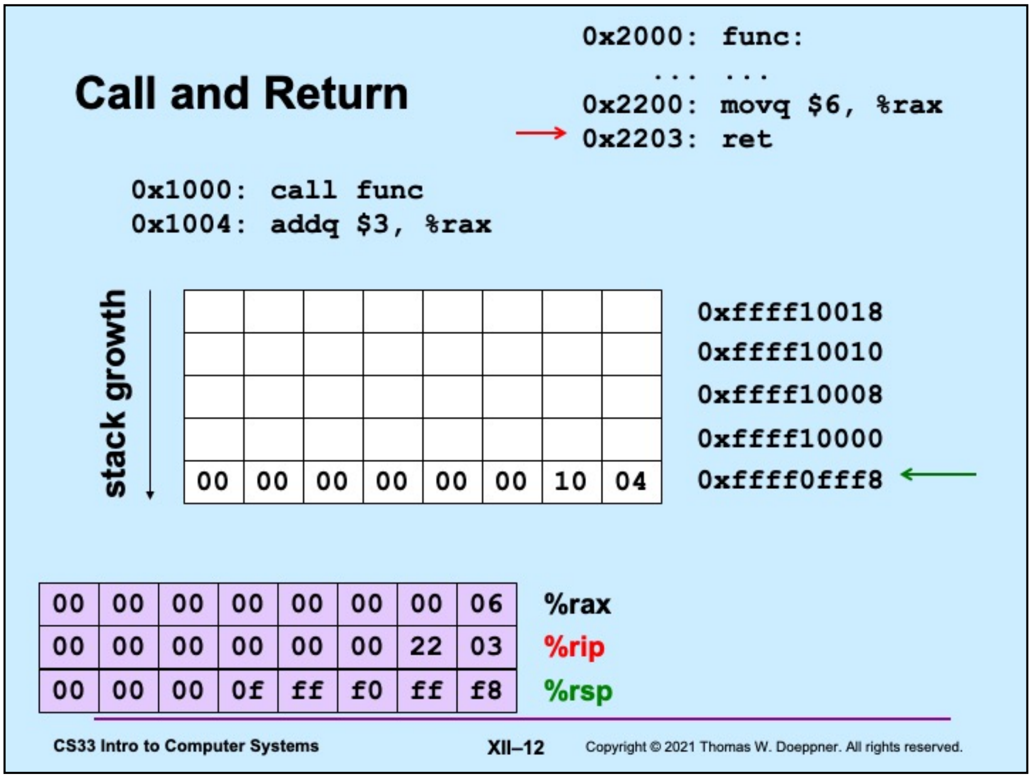


Here we begin walking through what happens during a call and return.

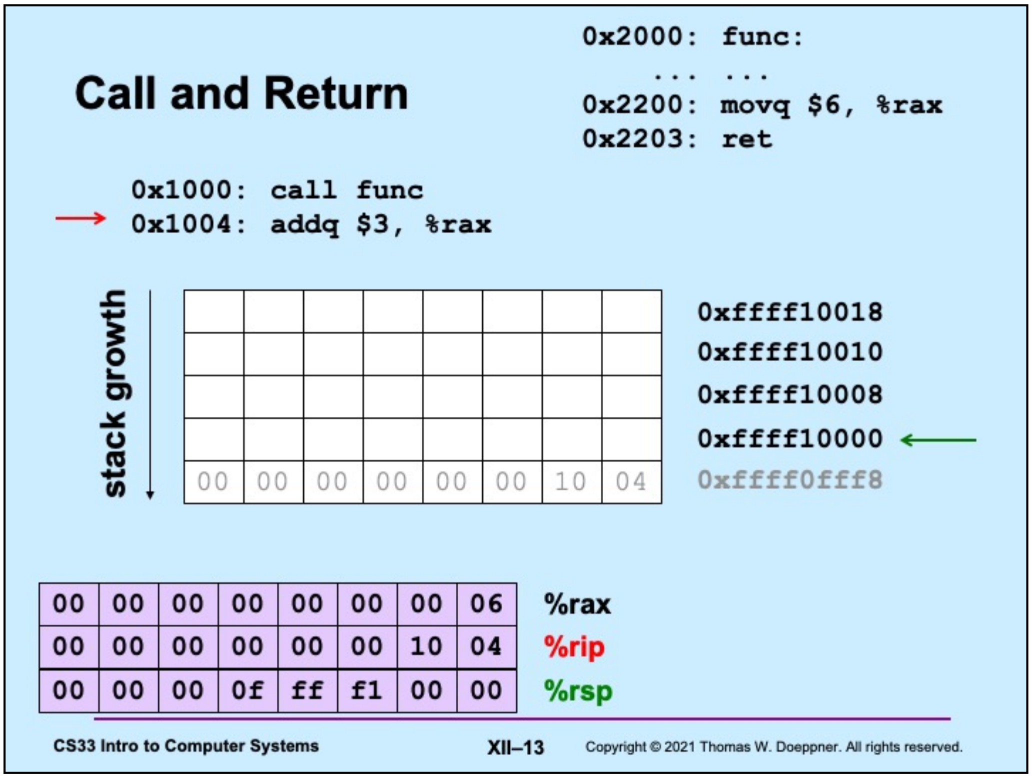
Initially, %rip (the instruction pointer – what it points to is shown with a red arrow pointing to the right) points to the call instruction – thus it's the next instruction to be executed. %rsp (the stack pointer, shown with a green arrow pointing to the left) points to the current end of the stack. 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.

Arguments and Local Variables

```
int mainfunc() {
   long array[3] =
      {2,117,-6};
   long sum =
      ASum(array, 3);
   ...
   return sum;
}
```

- Local variables usually allocated on stack
- Arguments to functions pushed onto stack

```
long ASum(long *a,
       unsigned long size) {
   long i, sum = 0;
   for (i=0; i<size; i++)
      sum += a[i];
   return sum;
}
```

- Local variables may be put in registers (and thus not on stack)

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

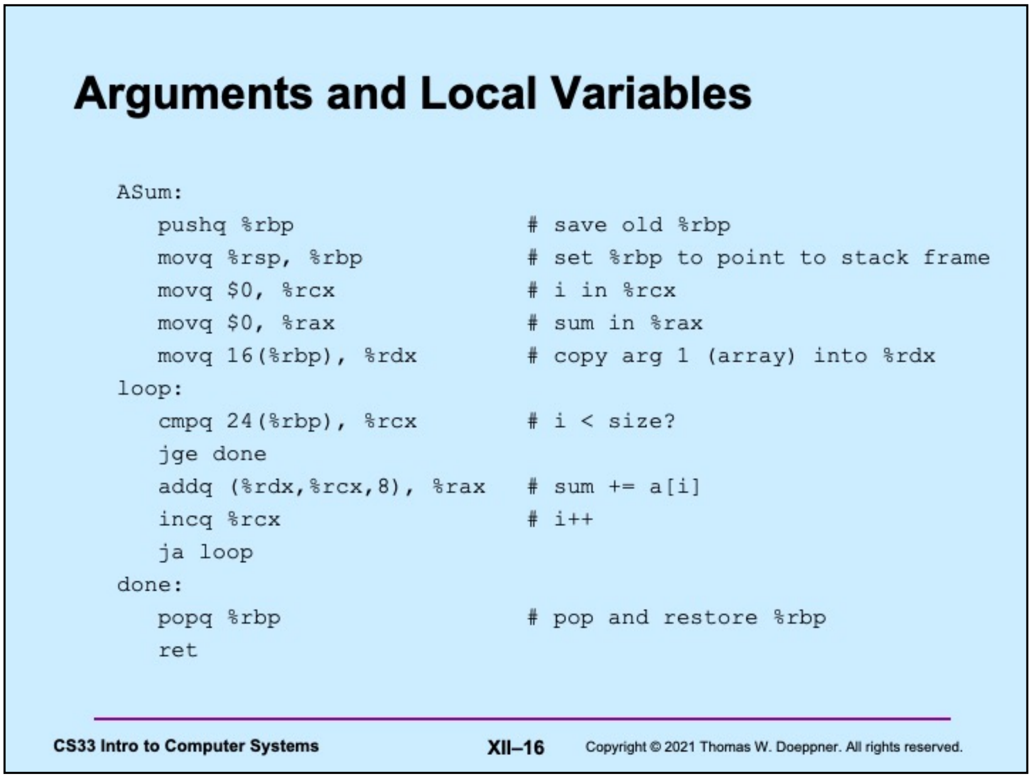
Arguments and Local Variables

```
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
```

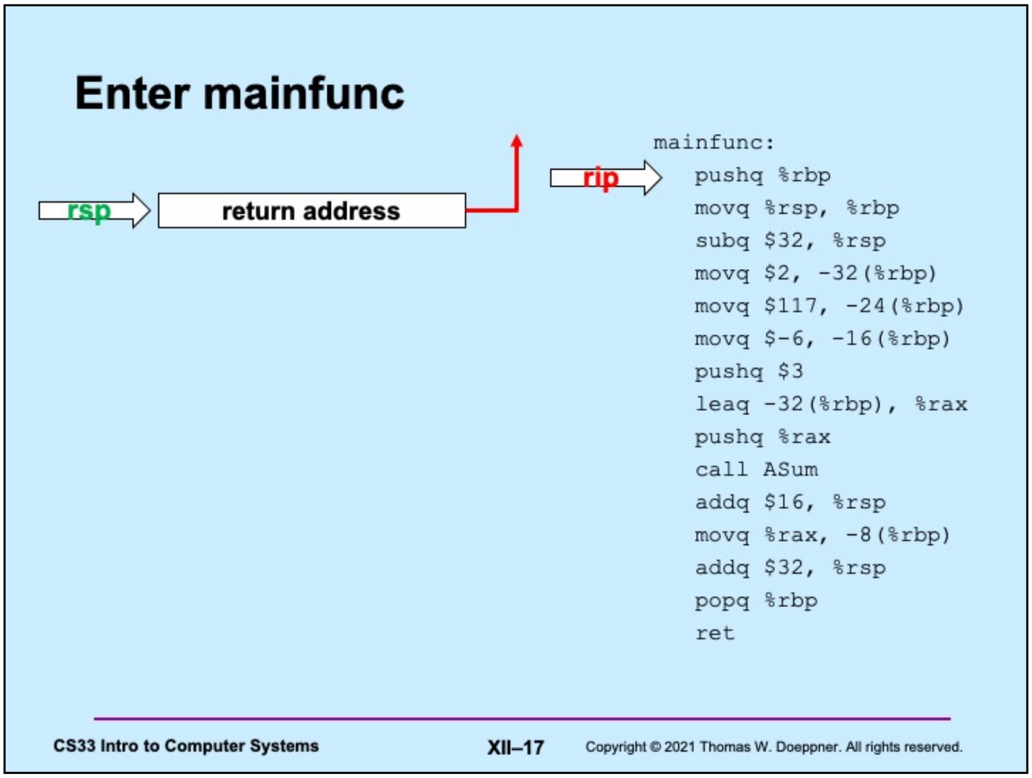

Here we have compiled code for **mainfunc**. We'll work through this in detail in upcoming slides.

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.

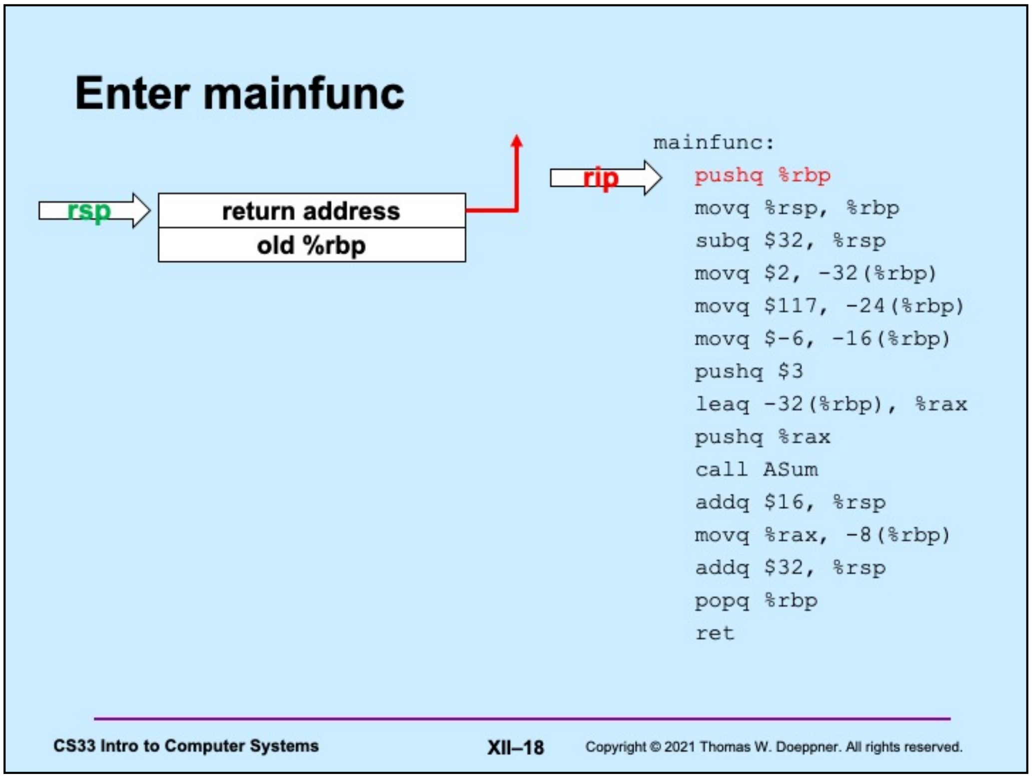
Note, as will be explained, this is not what one would see when compiling it for department computers, on which arguments are passed using registers.



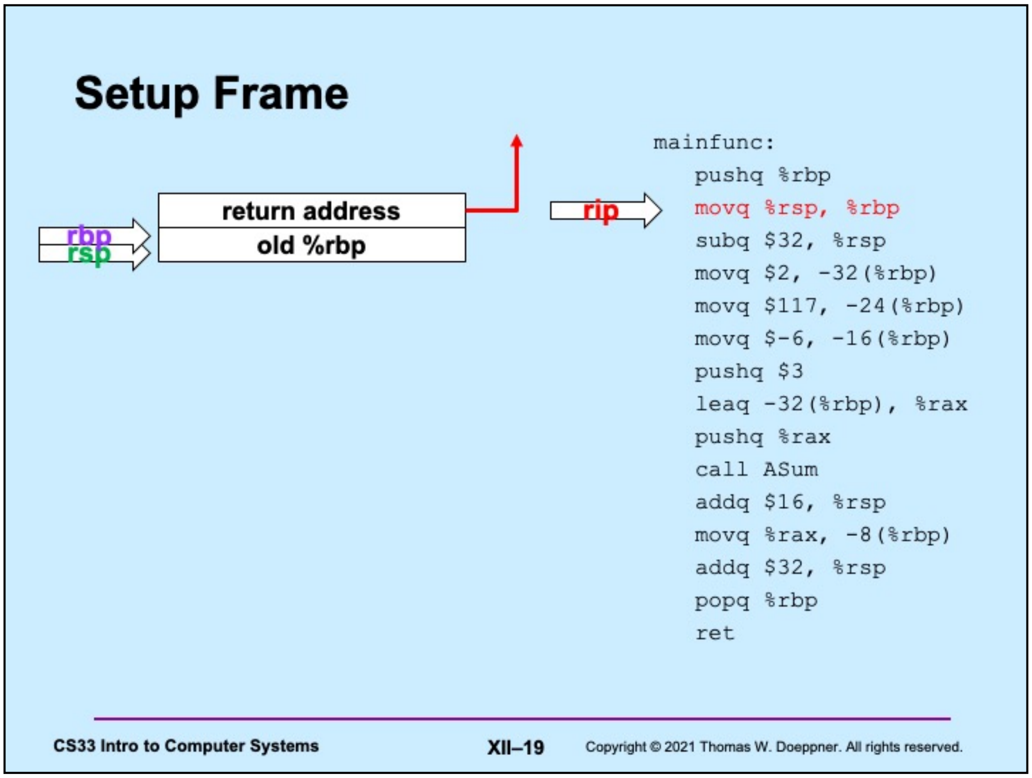
And here is the compiled code for **ASum**. The same caveats as given for the previous slide apply to this one as well.



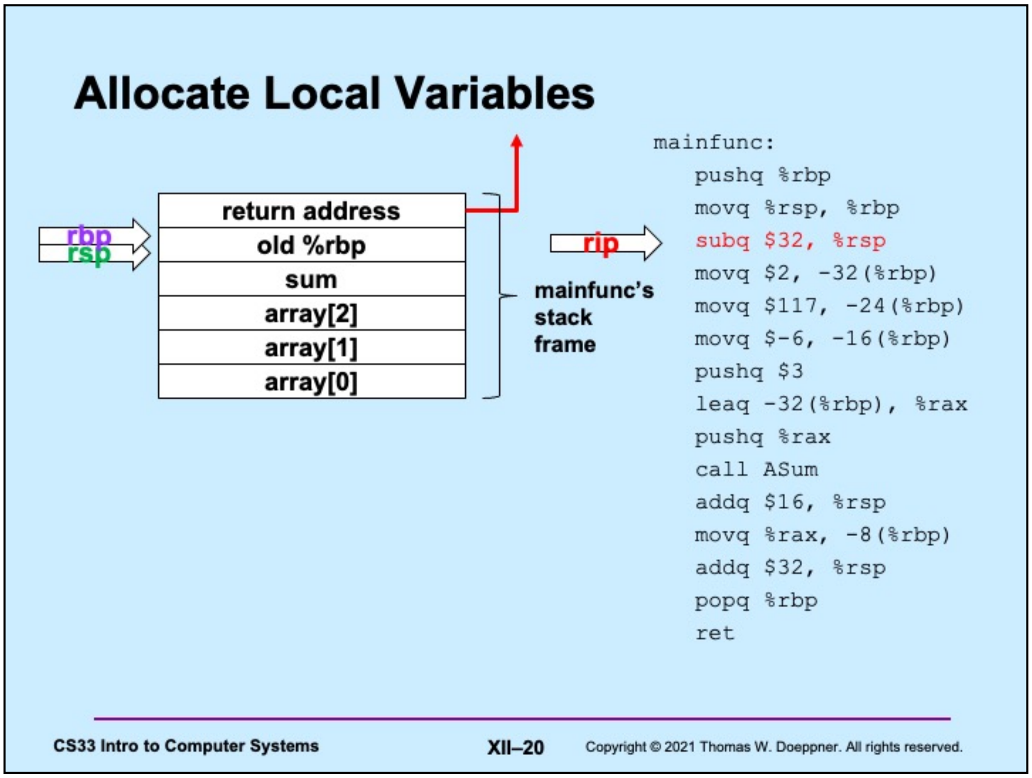
On entry to **mainfunc**, %rsp points to the caller's return address.



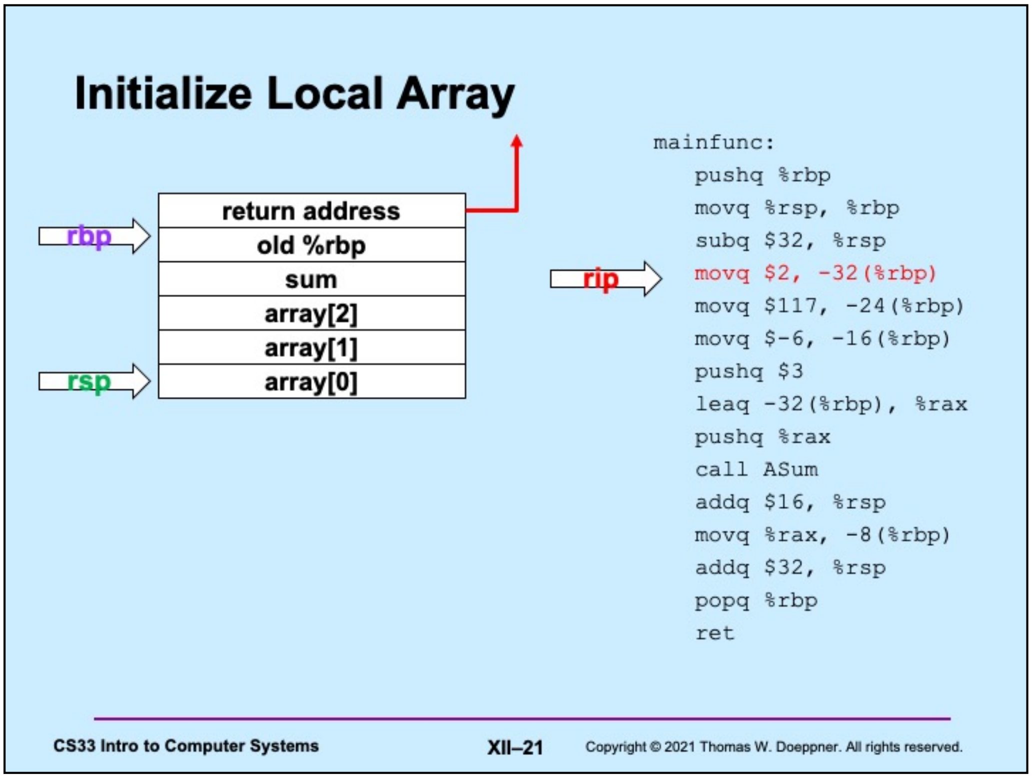
On entry to **mainfunc**, %rsp points to the caller's return address.



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

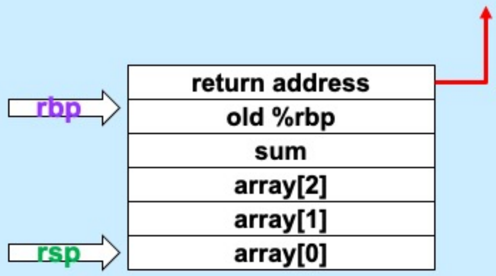


Next, space for **mainfunc**'s local variables is allocated on the stack by decrementing %rsp by their total size (32 bytes). At this point we have **mainfunc**'s stack frame in place.



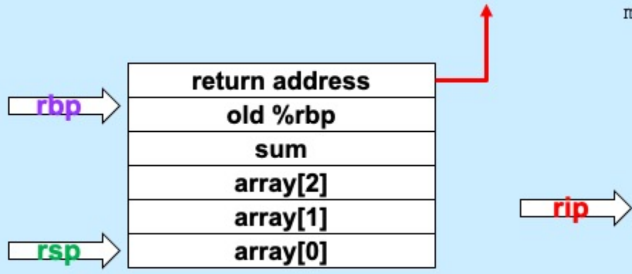
ASum now initializes the stack space containing its local variables.

Initialize Local Array



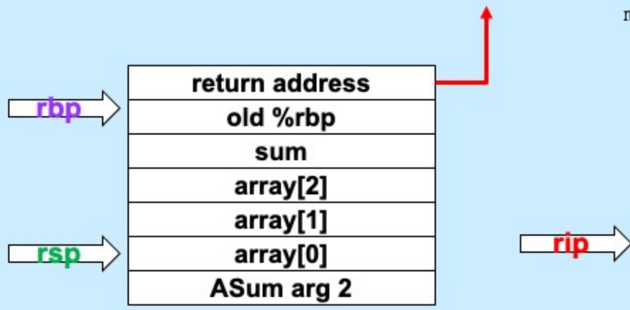

```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

Initialize Local Array



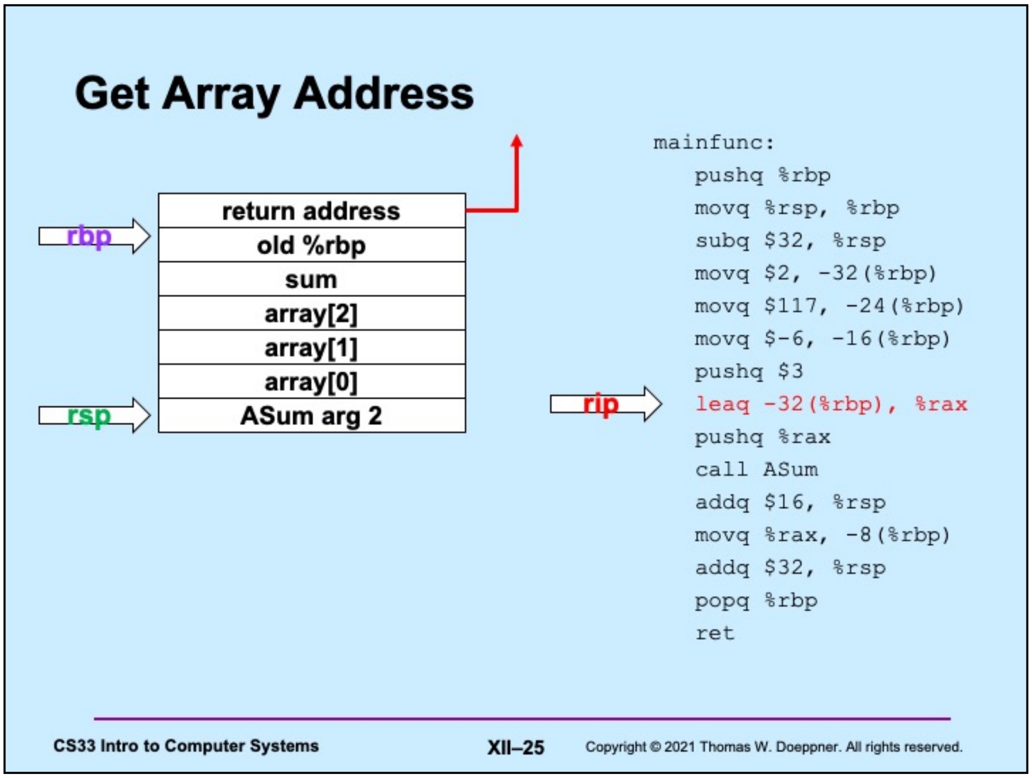

```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

Push Second Argument

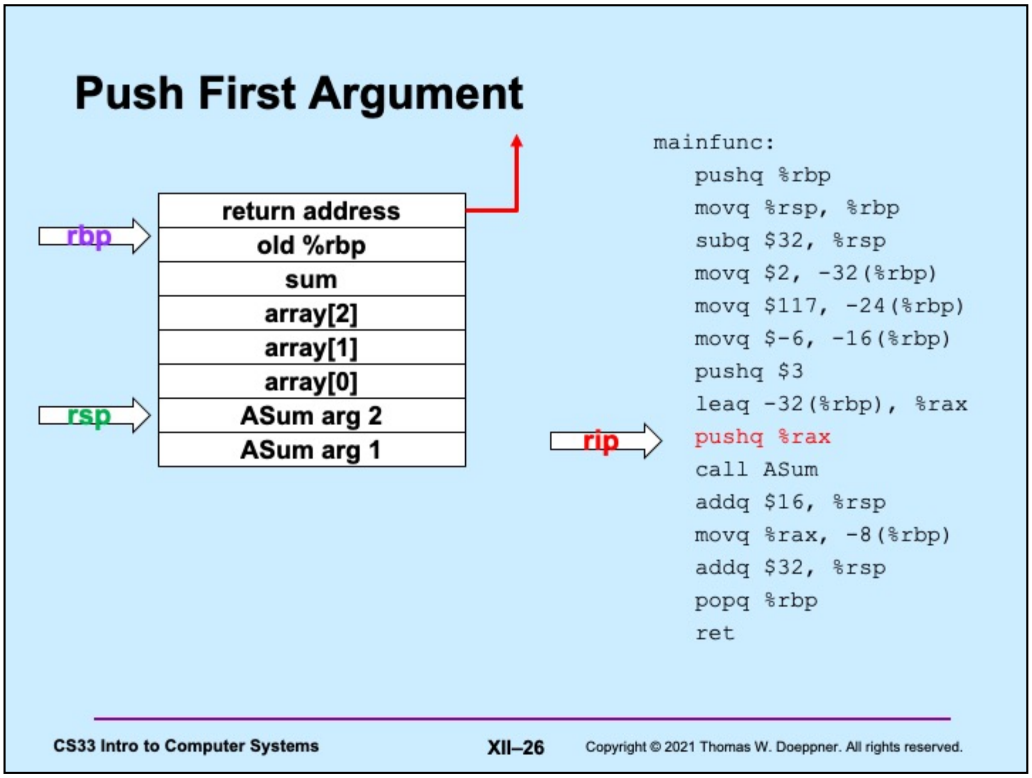



```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

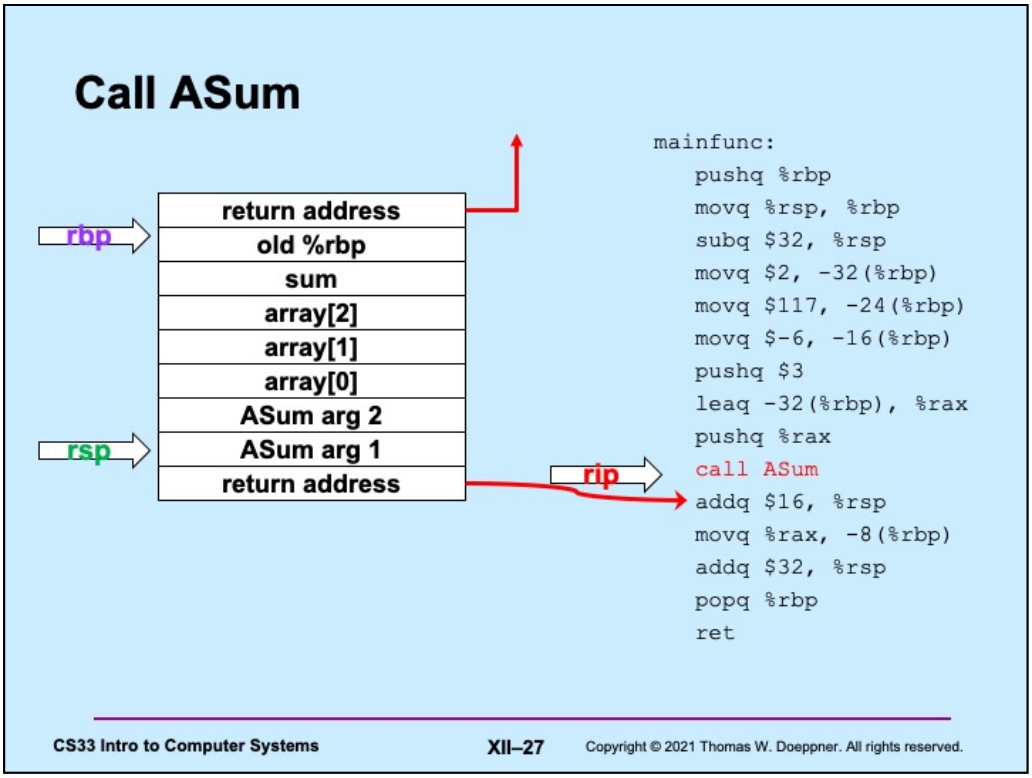

The second argument (3) to **ASum** is pushed onto the stack.



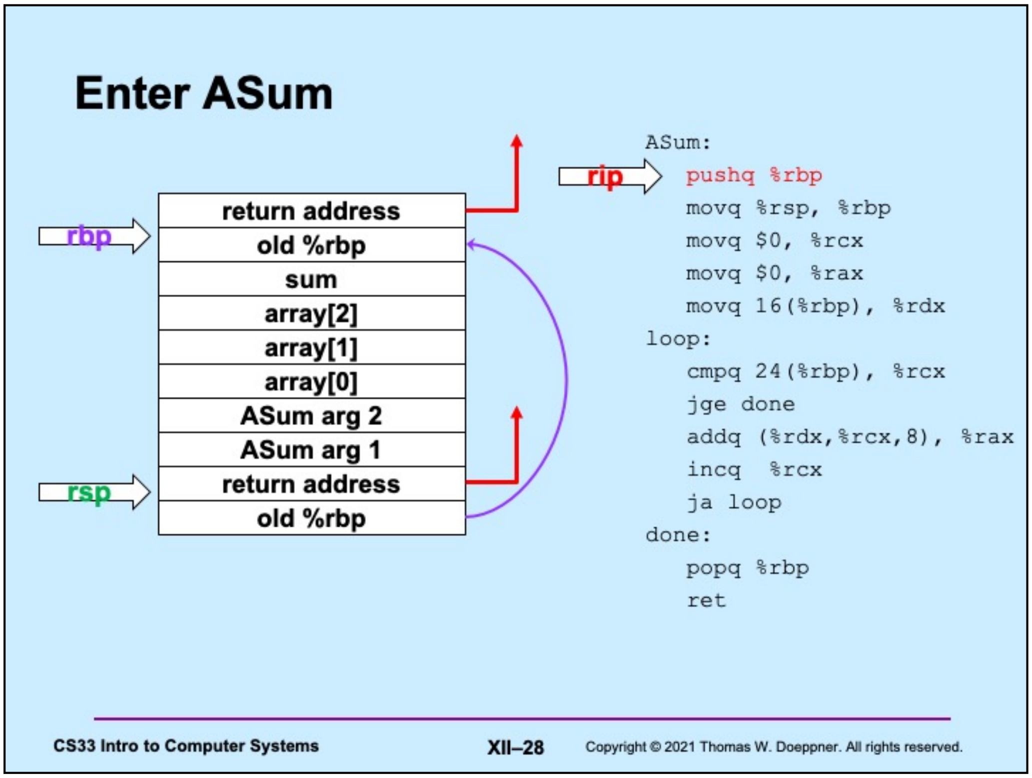
In preparation for pushing the first argument to **ASum** onto the stack, the address of the array is put into %rax.



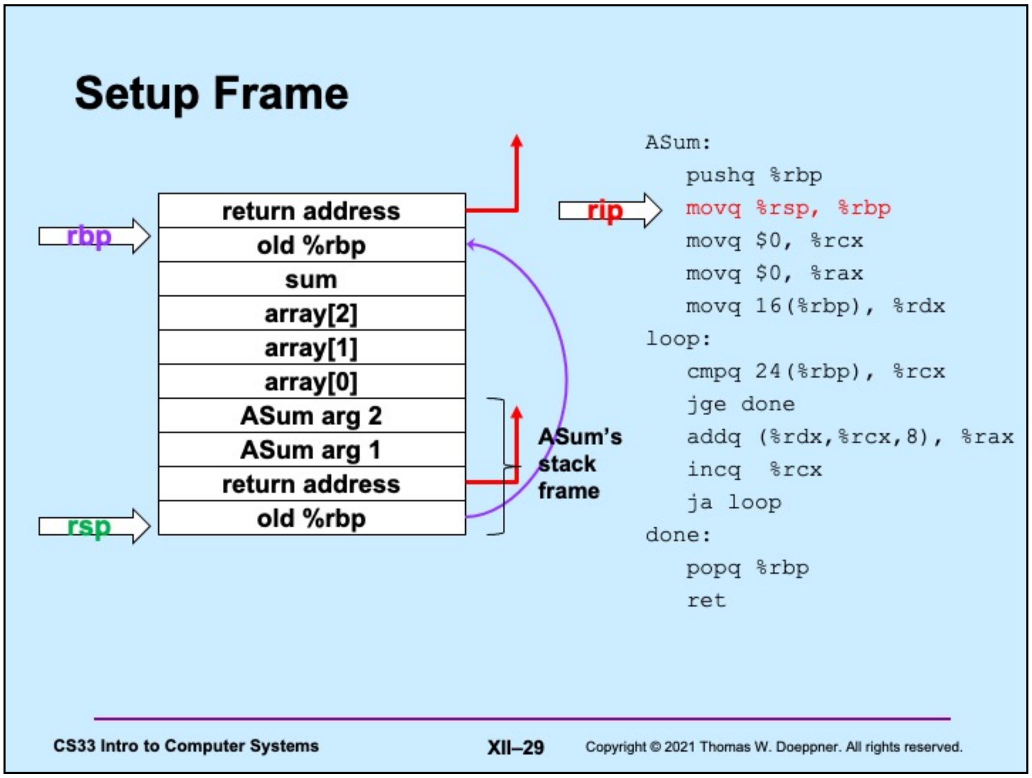
And finally, the address of the array is pushed onto the stack as **ASum**'s first argument.



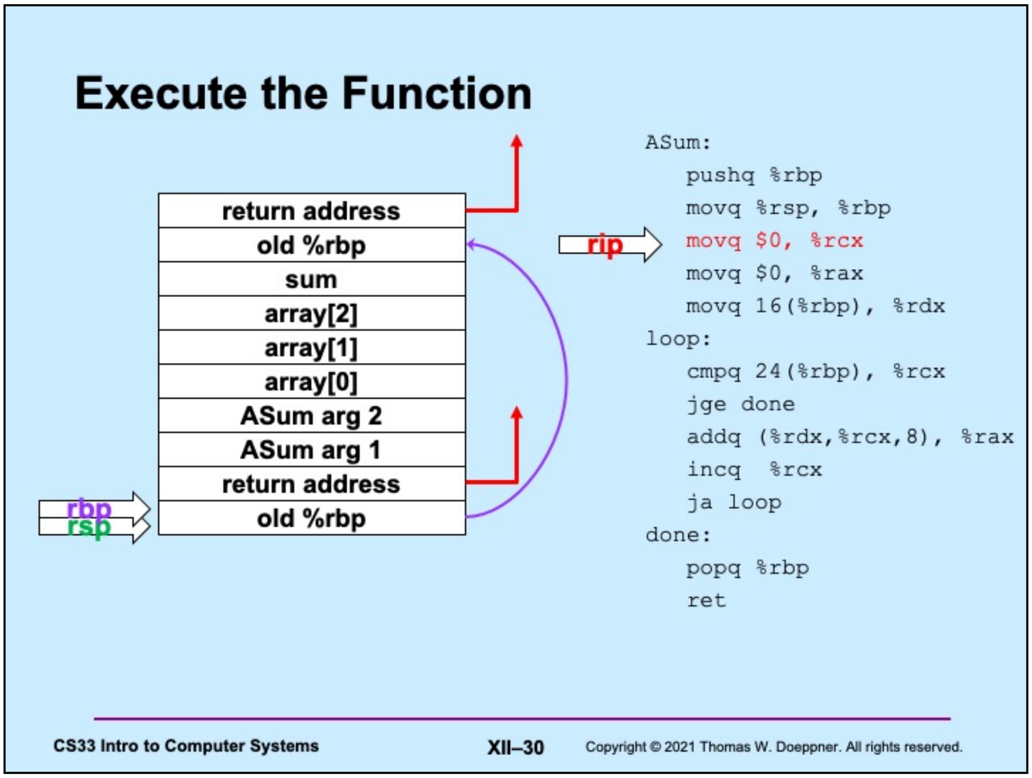
mainfunc now calls **ASum**, pushing its return address onto the stack.



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.



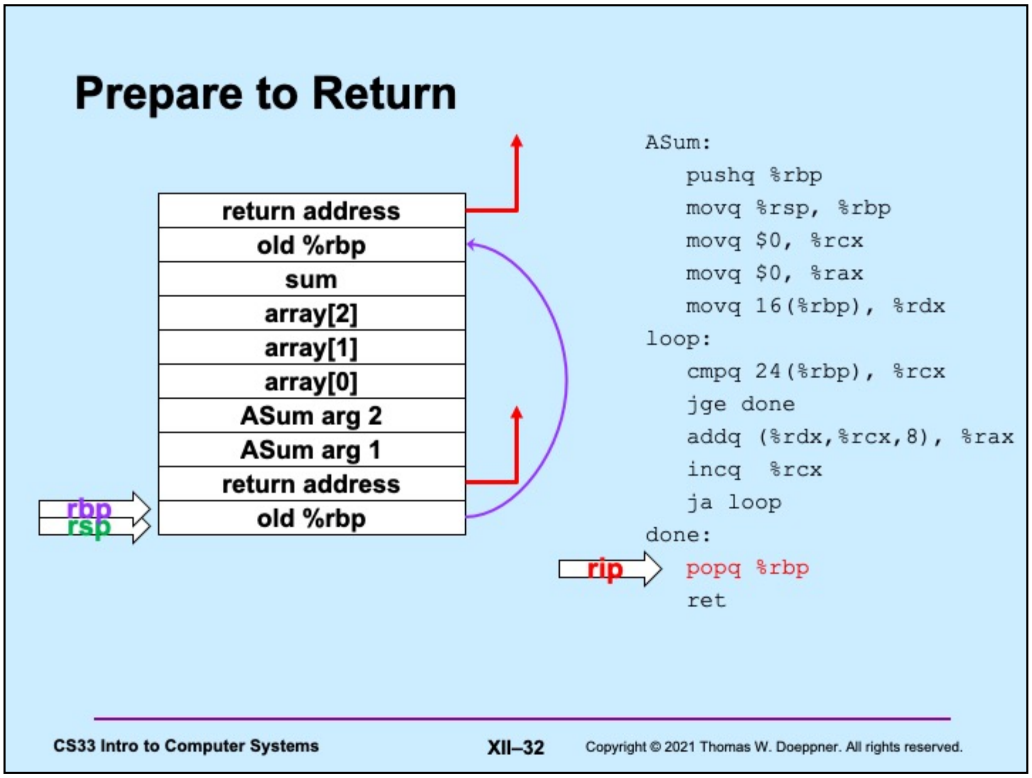
ASum's instructions are now executed, summing the contents of its first argument and storing the result in %rax.

Quiz 1

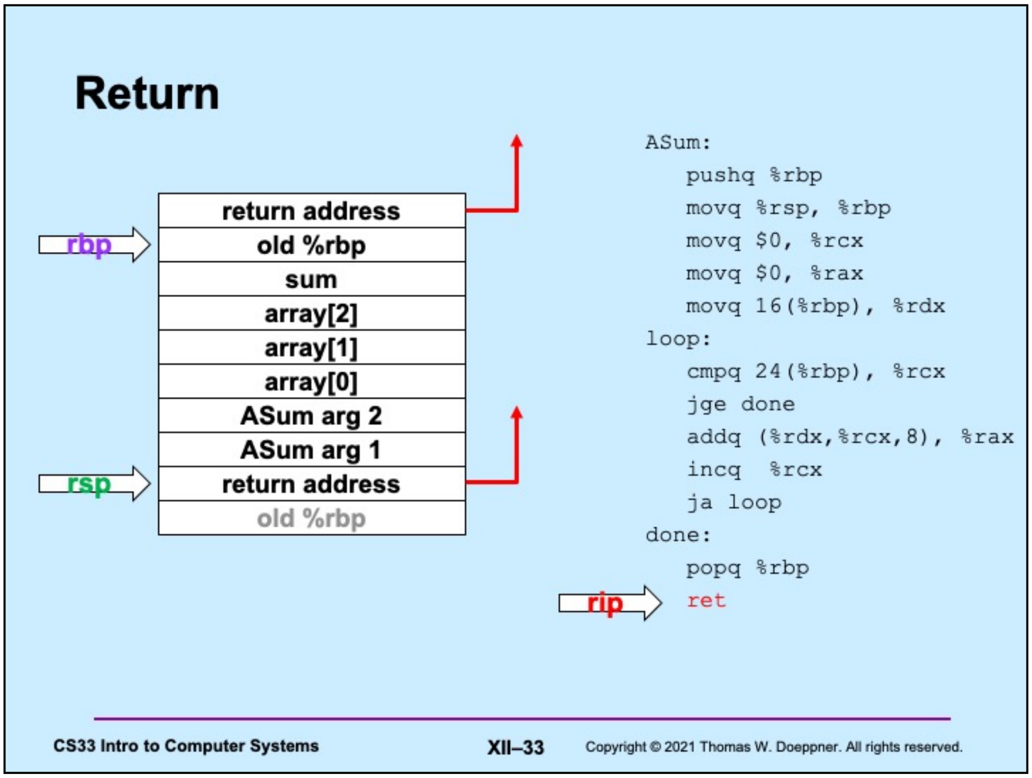
What's at 16(%rbp)?

a) a local variable
b) the first argument to ASum
c) the second argument to ASum
d) something else

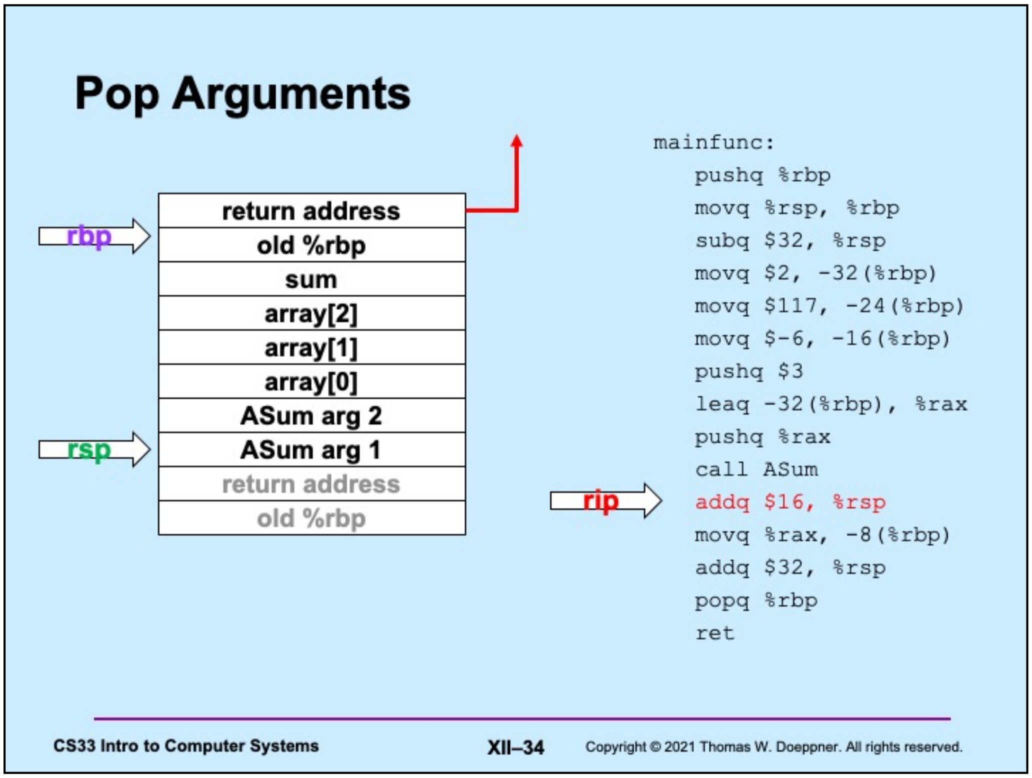
```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```

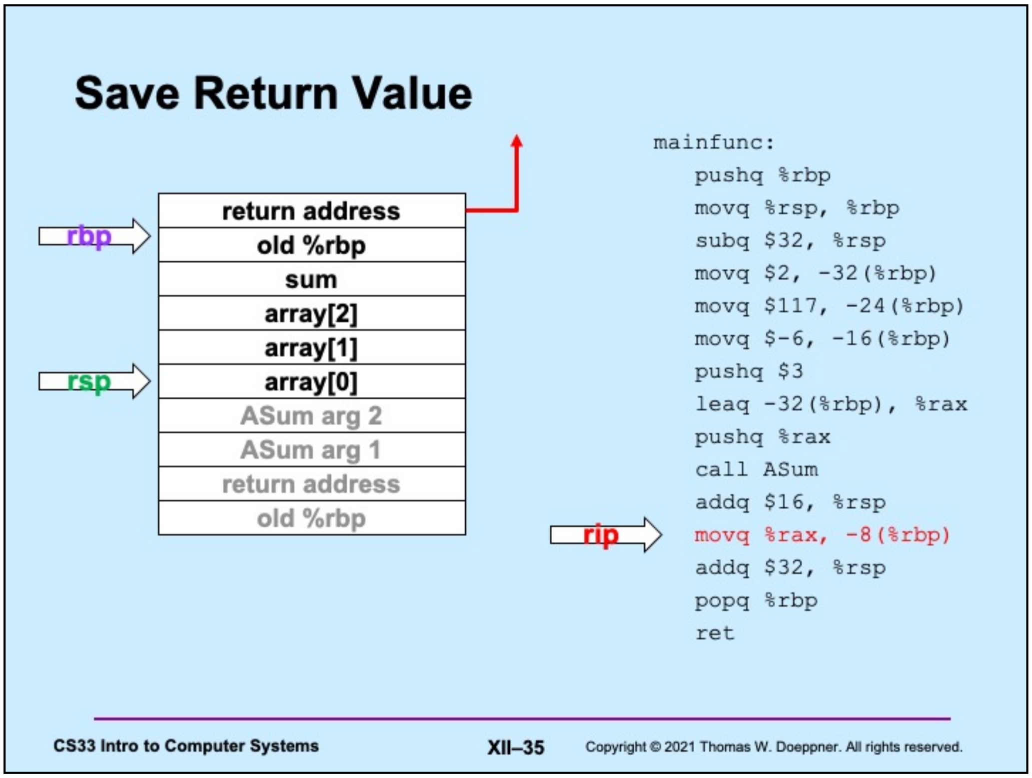
In preparation for returning to its caller, **ASum** restores the previous value of %rbp by popping it off the stack.



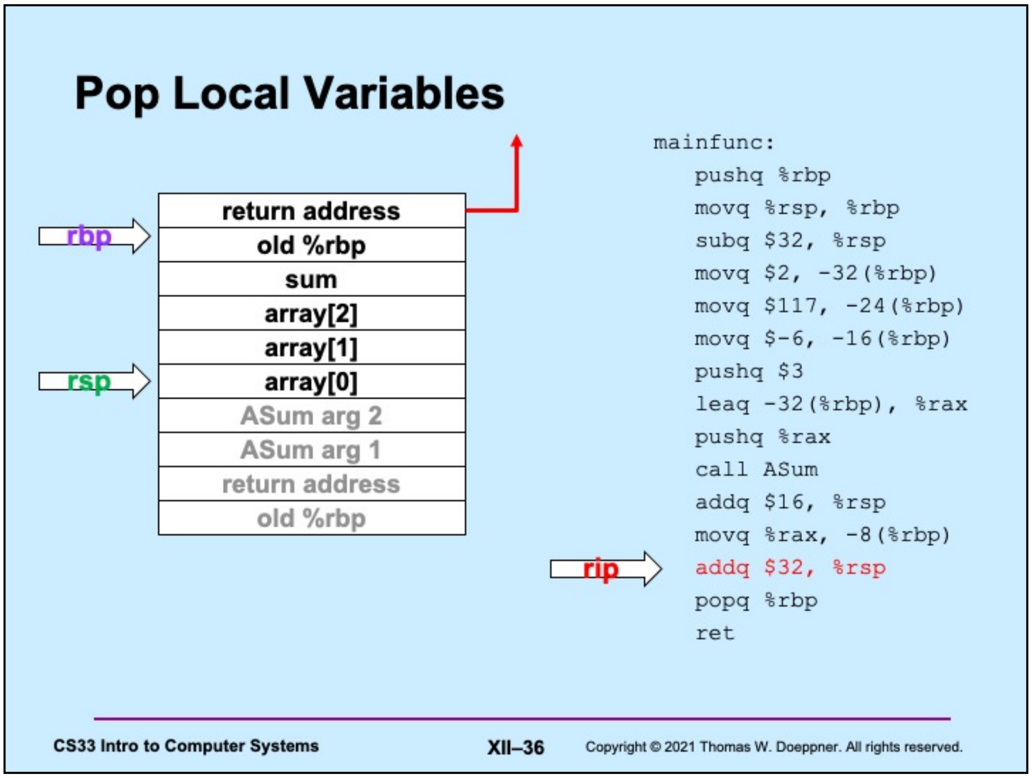
ASum returns by popping the return address off the stack and into %rip, so that execution resumes in its caller (**mainfunc**).



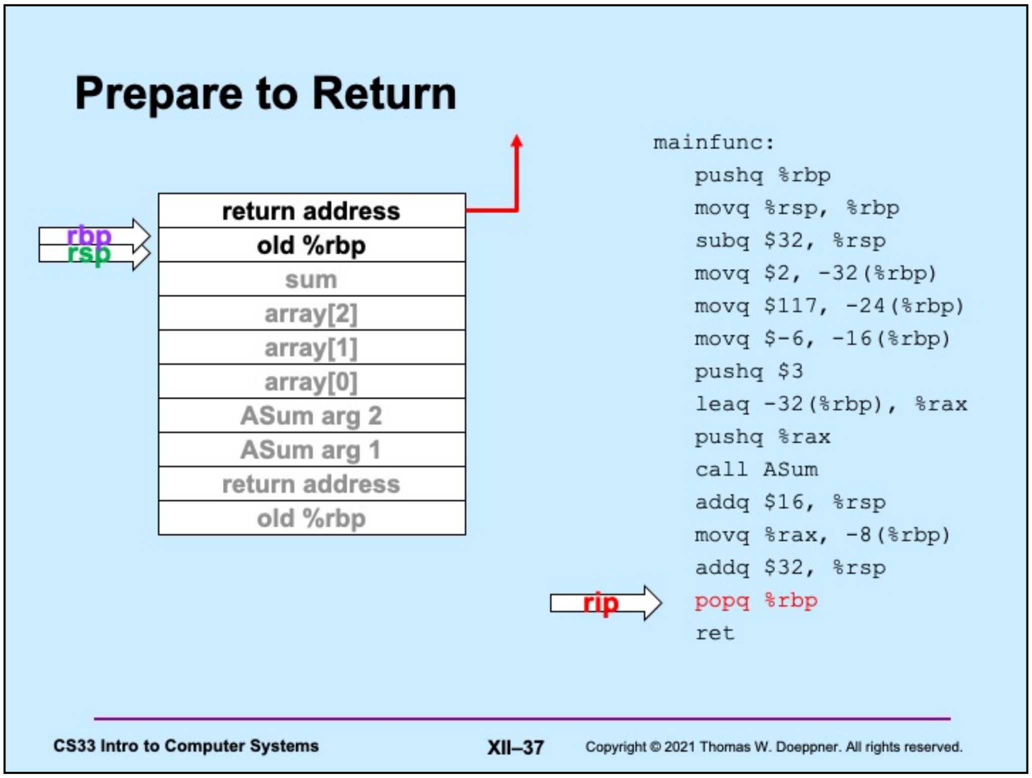
mainfunc no longer needs the arguments it had pushed onto the stack for **ASum**, so it pops them off the stack by adding their total size to %rsp.



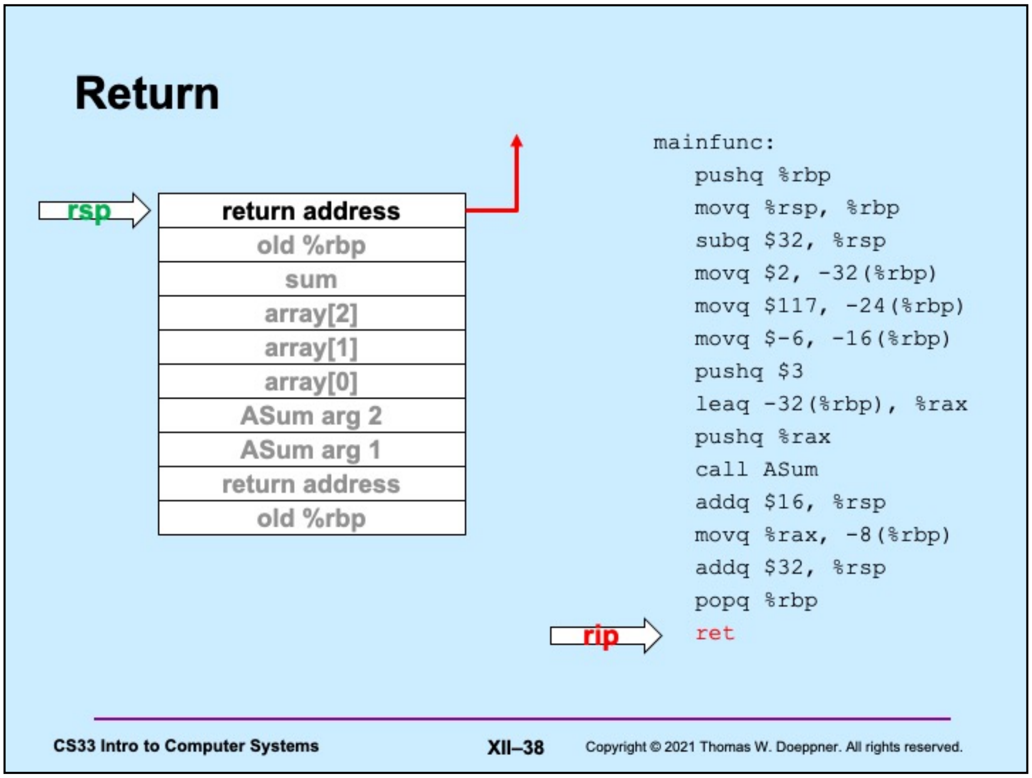
The value returned by **ASum** (in %rax) is copied into the local variable sum (which is in **mainfunc**'s stack frame).



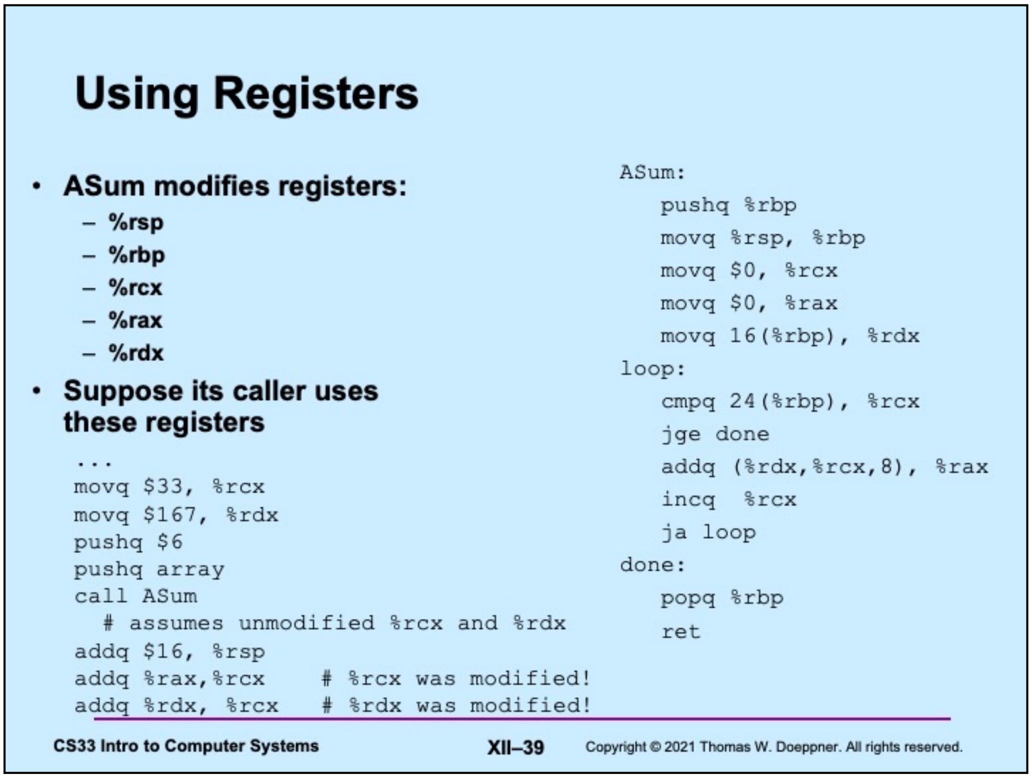
mainfunc is about to return, so it pops its local variables off the stack (by adding their total size to %rsp).



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.



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

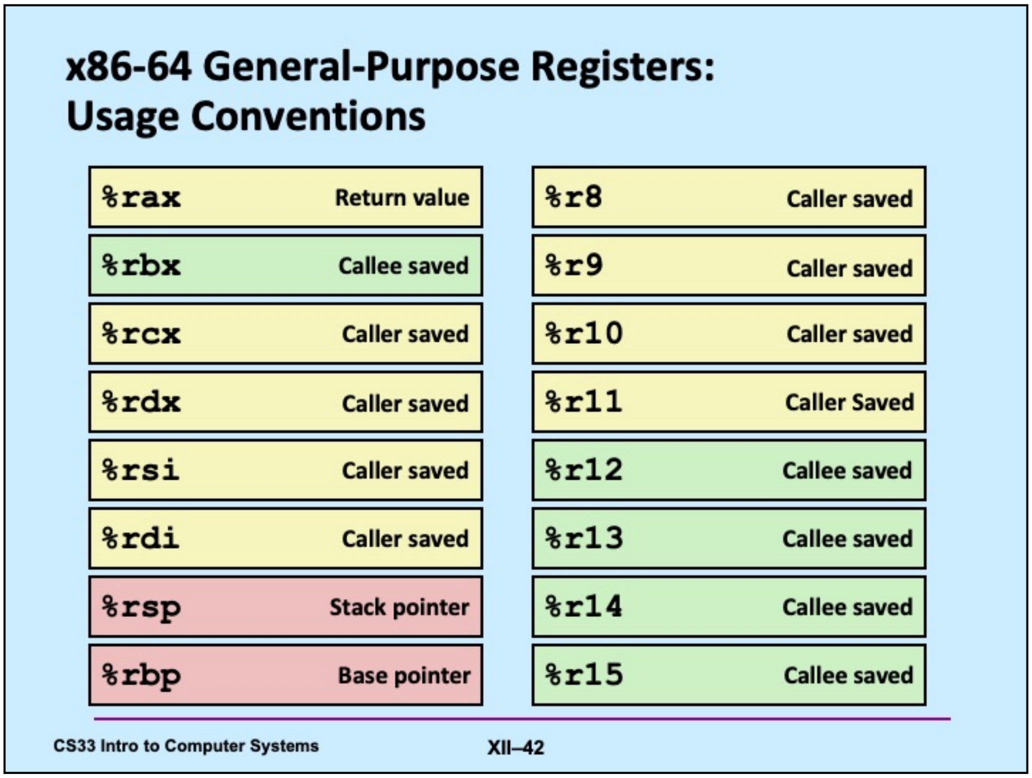
```
pushq %rcx
call func
popq %rcx
```

- **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.

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

Why Bother with a Base Pointer?

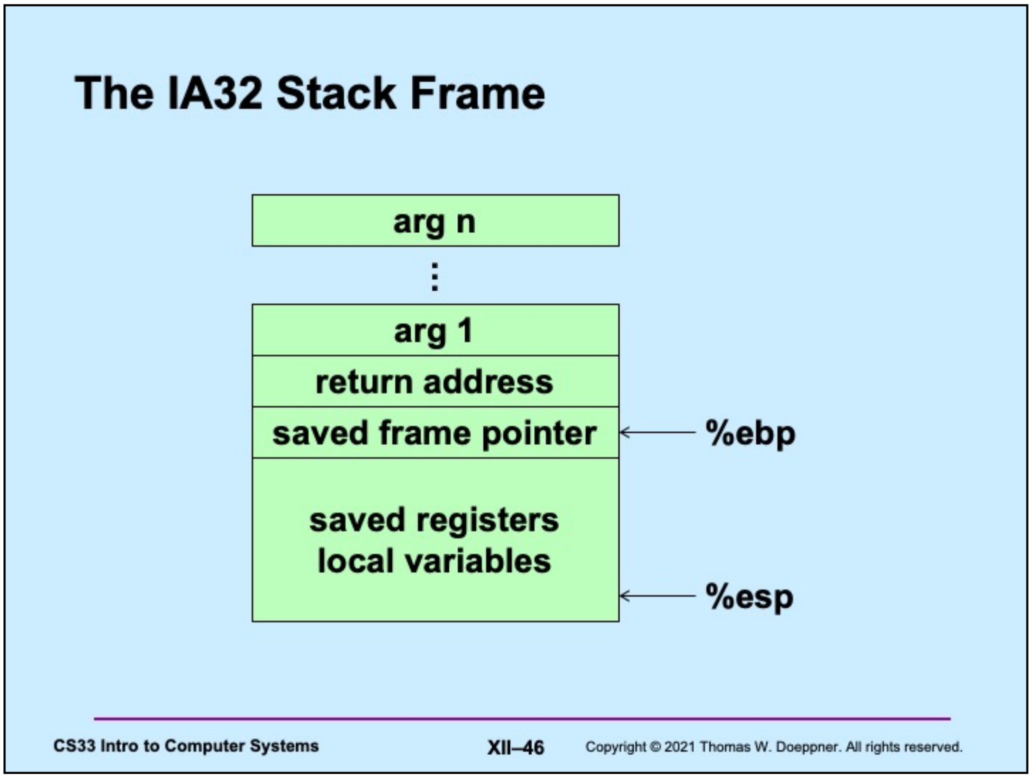
- It (%rbp) points to the beginning of the stack frame
  - making it easy for people to figure out where things are in the frame
  - but people don't execute the code ...
- The stack pointer always points somewhere within the stack frame
  - it moves about, but the compiler knows where it is pointing
    - » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    - » tough for people, but easy for the compiler
- Thus the base pointer is superfluous
  - it can be used as a general-purpose register

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).

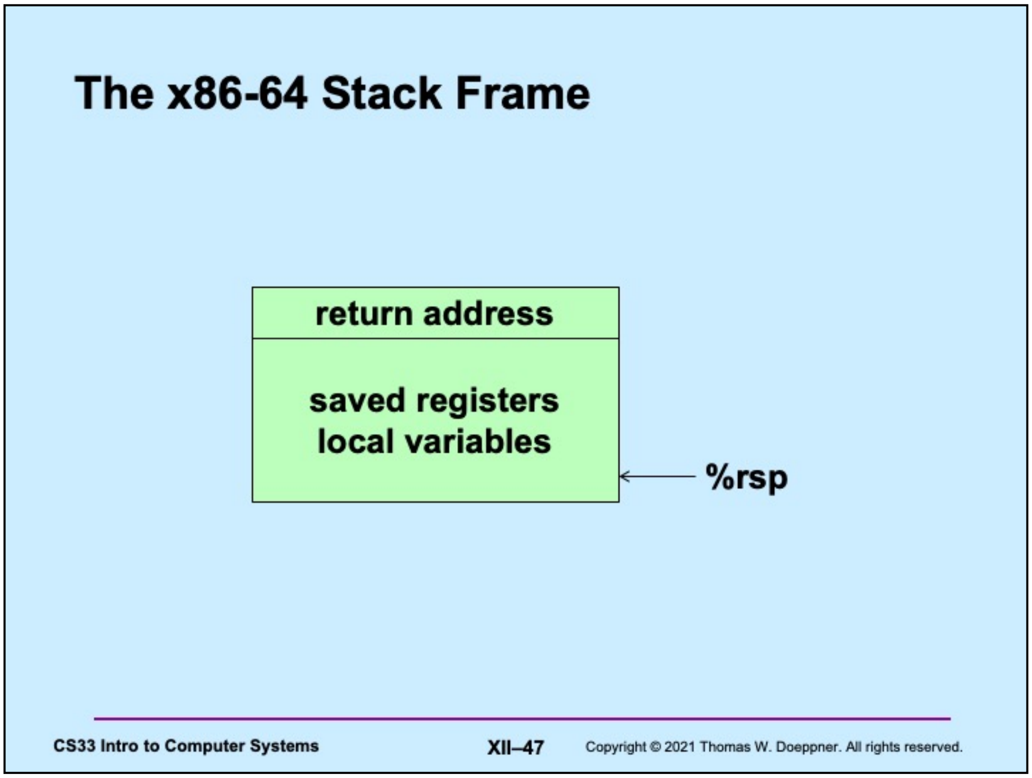
x86-64 General-Purpose Registers: Updated Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

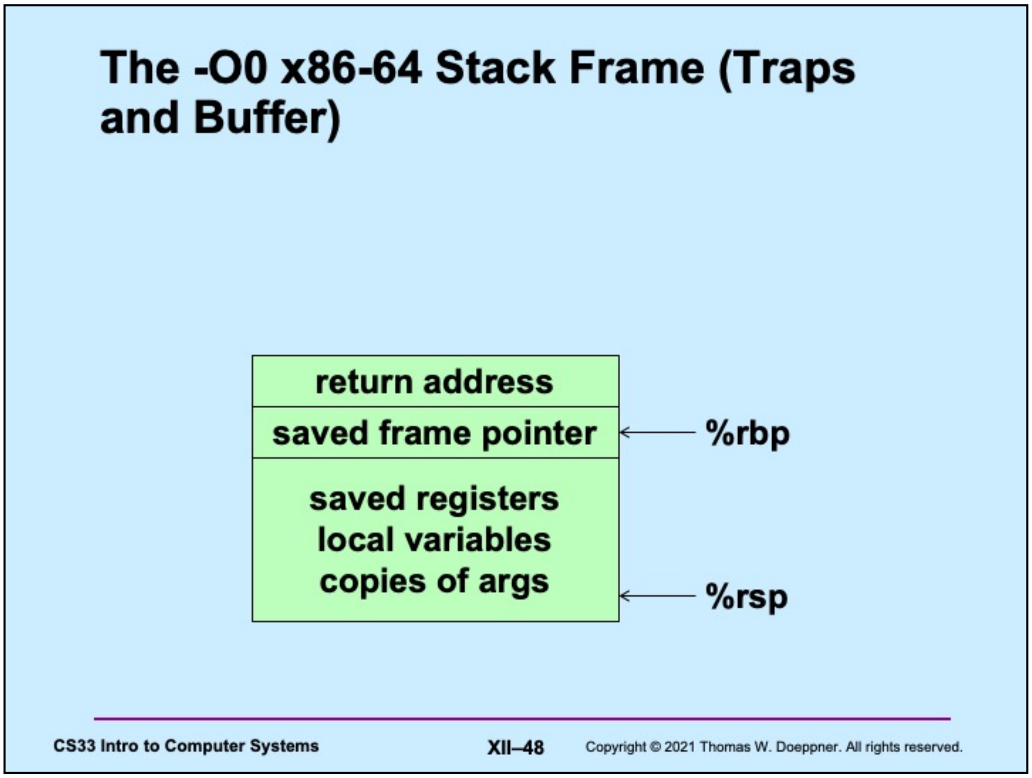
Supplied by CMU.



Here, again, is the IA32 stack frame. Recall that arguments are at positive offsets from %ebp, while local variables are at negative offsets.



The convention used for the x86-64 architecture is that the first 6 arguments to a function are passed in registers, there is no special frame-pointer register, and everything on the stack is referred to via offsets from %rsp.



When code is compiled with the –O0 flag on gdb, turning off all optimization, the compiler uses (unnecessarily) %rbp as a frame pointer so that the offsets to local variables are constant and thus easier for humans to read. It also copies the arguments from the registers to the stack frame (at a lower address than what %rbp contains).

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**