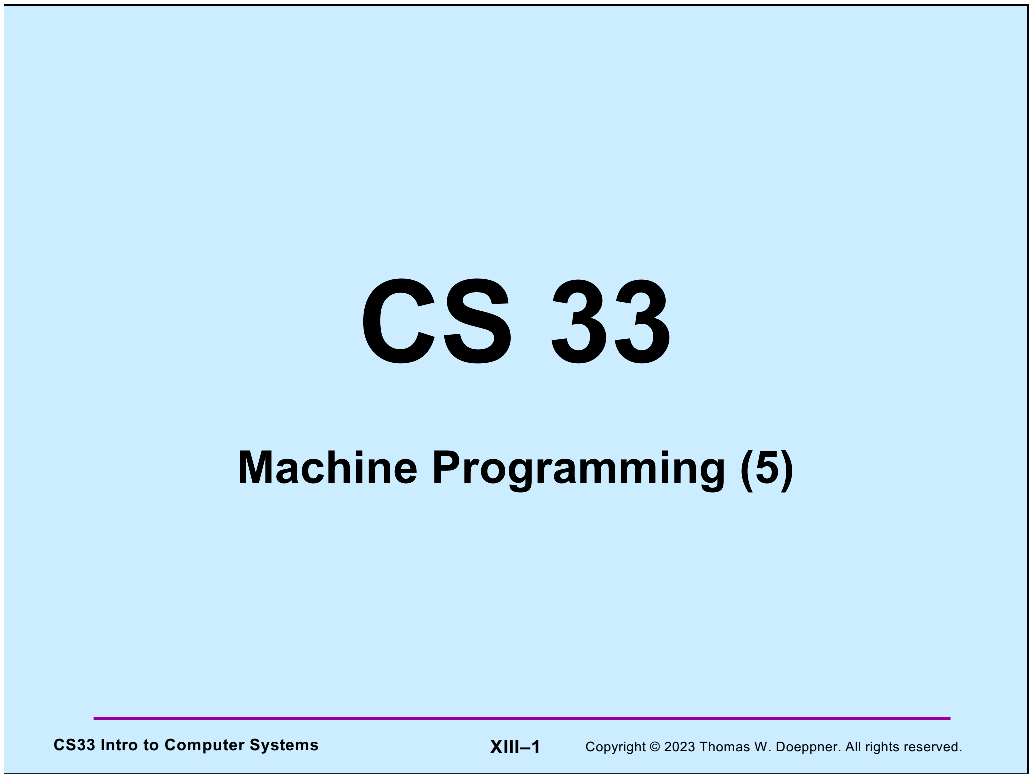
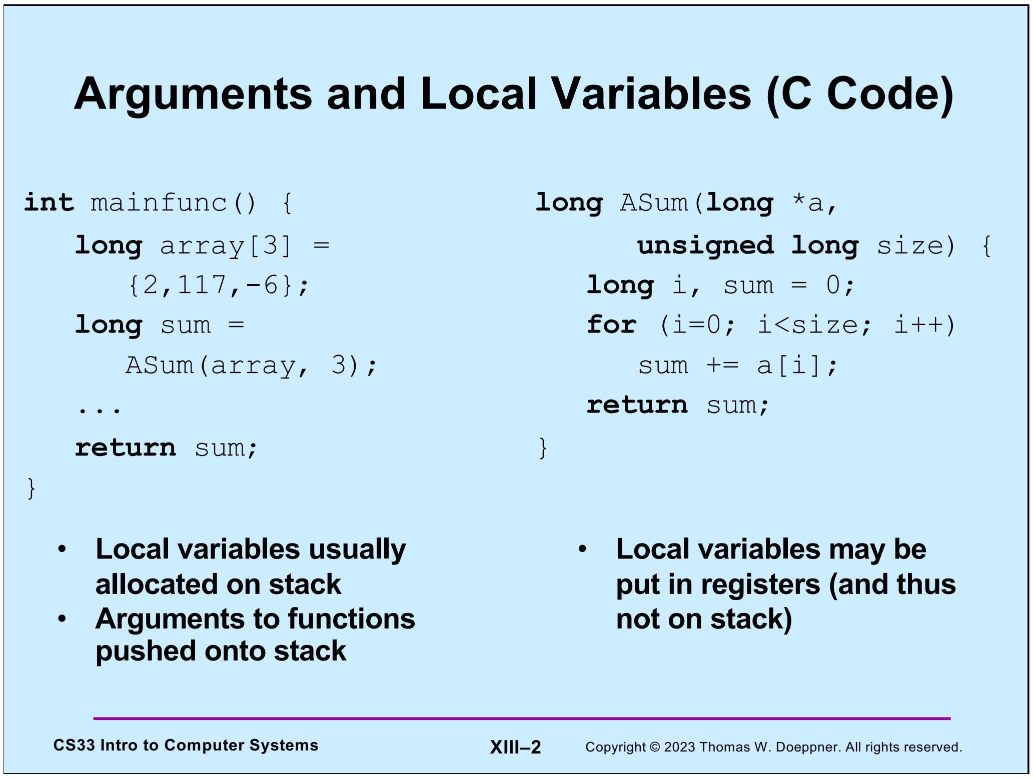


CS 33

Machine Programming (5)

Many 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.



Arguments and Local Variables (C Code)

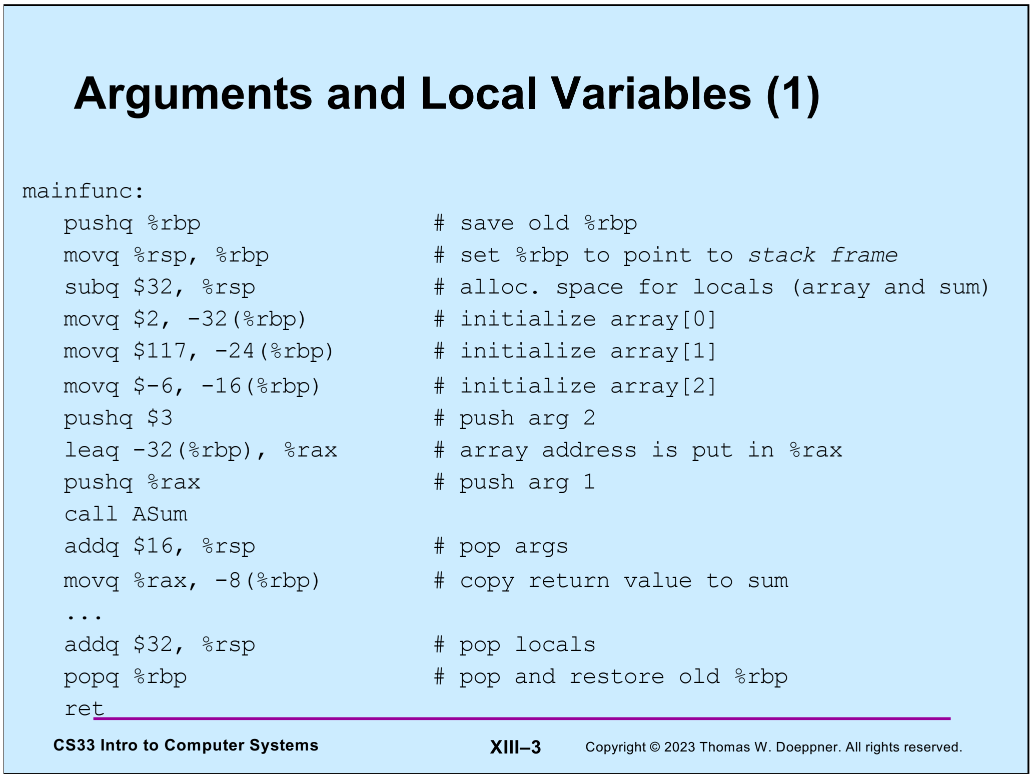
```
int mainfunc() {
   long array[3] =
      {2,117,-6};
   long sum =
      ASum(array, 3);
   ...
   return sum;
}
```

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

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

- **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. Note that the approach of storing arguments on the stack is used on the IA32 architecture, and on the x86-64 architecture when the –O0 optimization flag (meaning no optimization) is given to gcc.



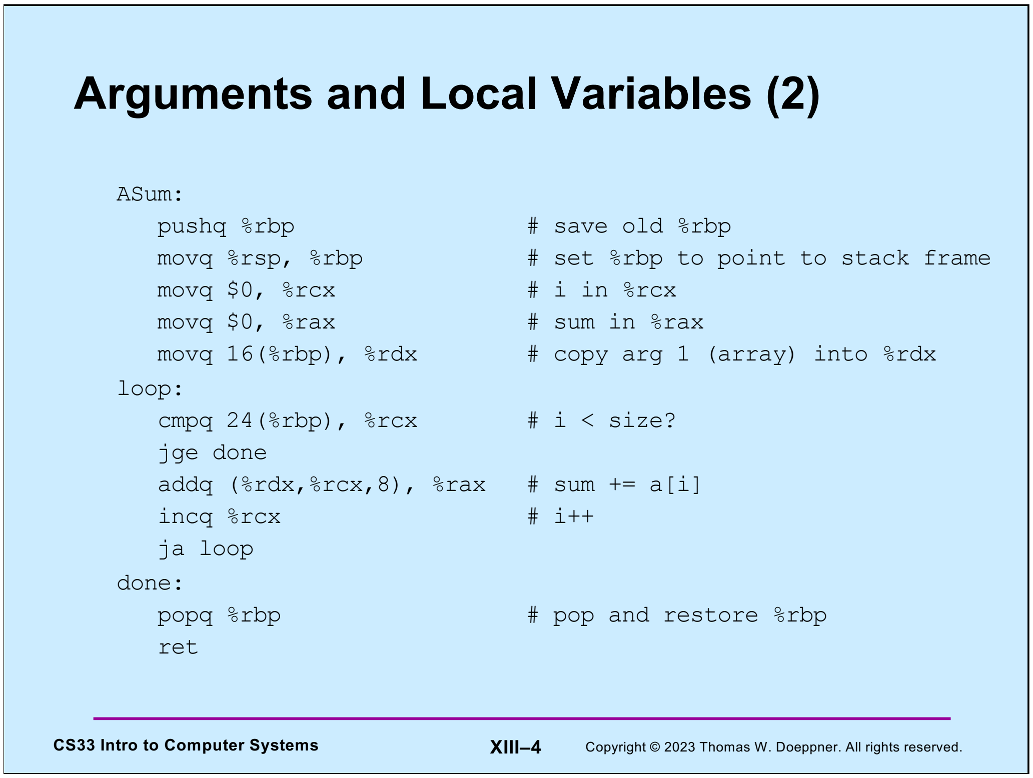
Arguments and Local Variables (1)

```
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.

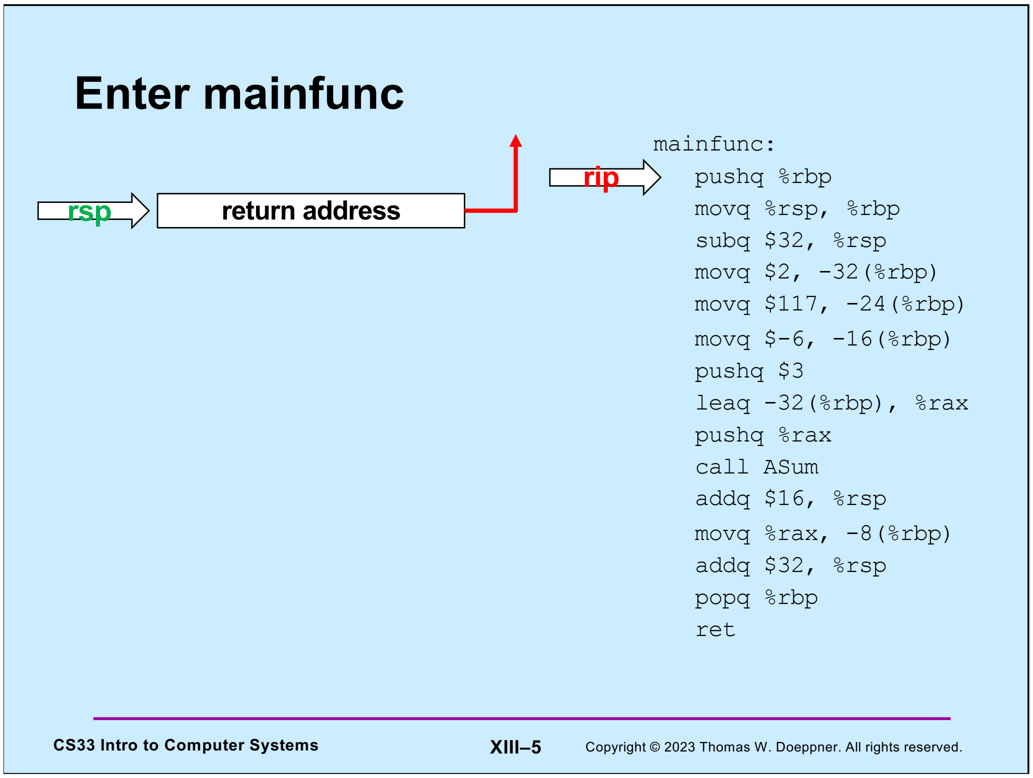
Note, as will be explained, this is not what one would see when compiling it with normal gcc options, which have arguments being passed via registers.



Arguments and Local Variables (2)

```
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
```

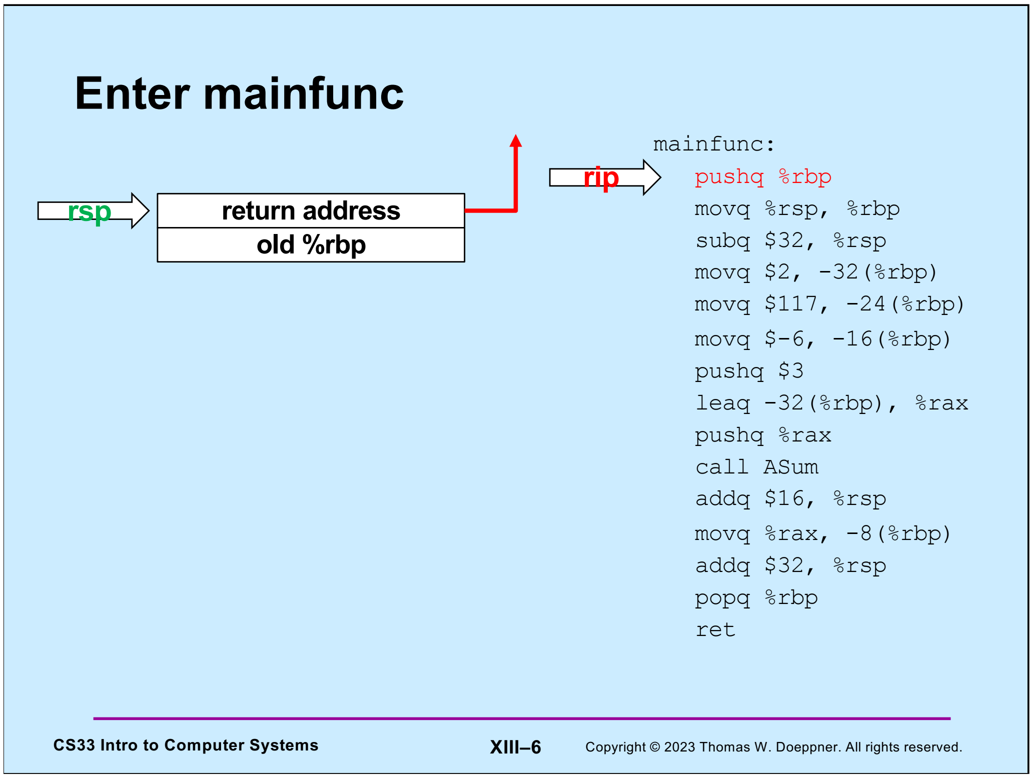
And here is the compiled code for **ASum**. The same caveats given for the previous slide apply to this one as well.



Enter mainfunc

```
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
```

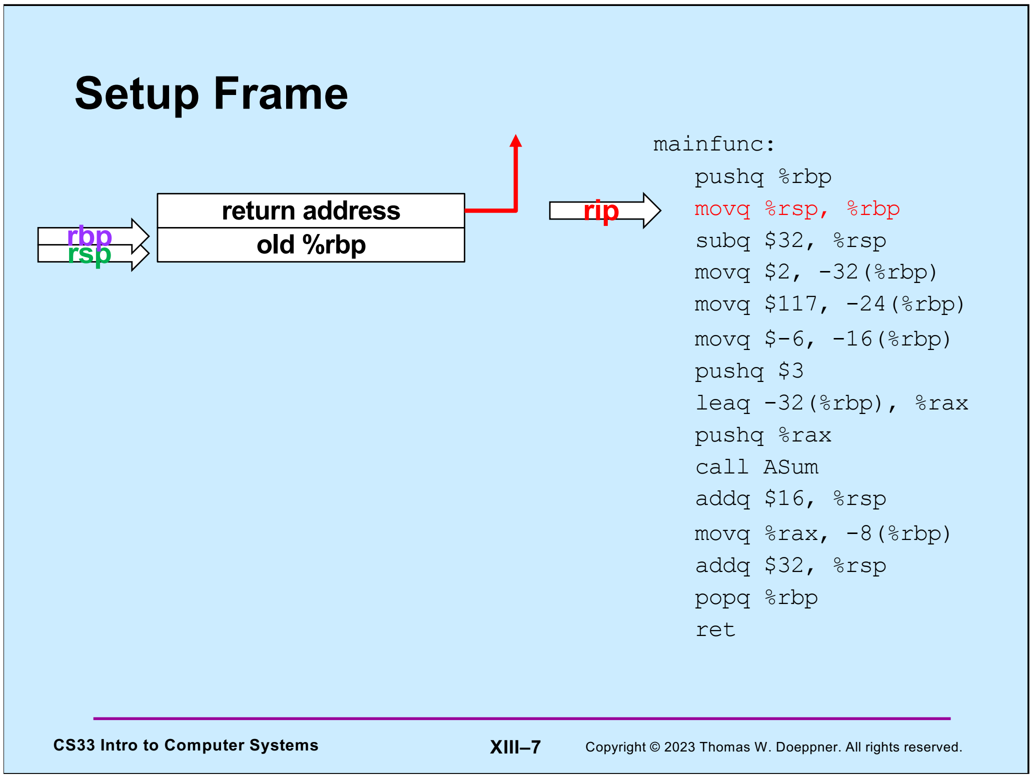
On entry to **mainfunc**, %rsp points to the caller's return address.



Enter mainfunc

```
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
```

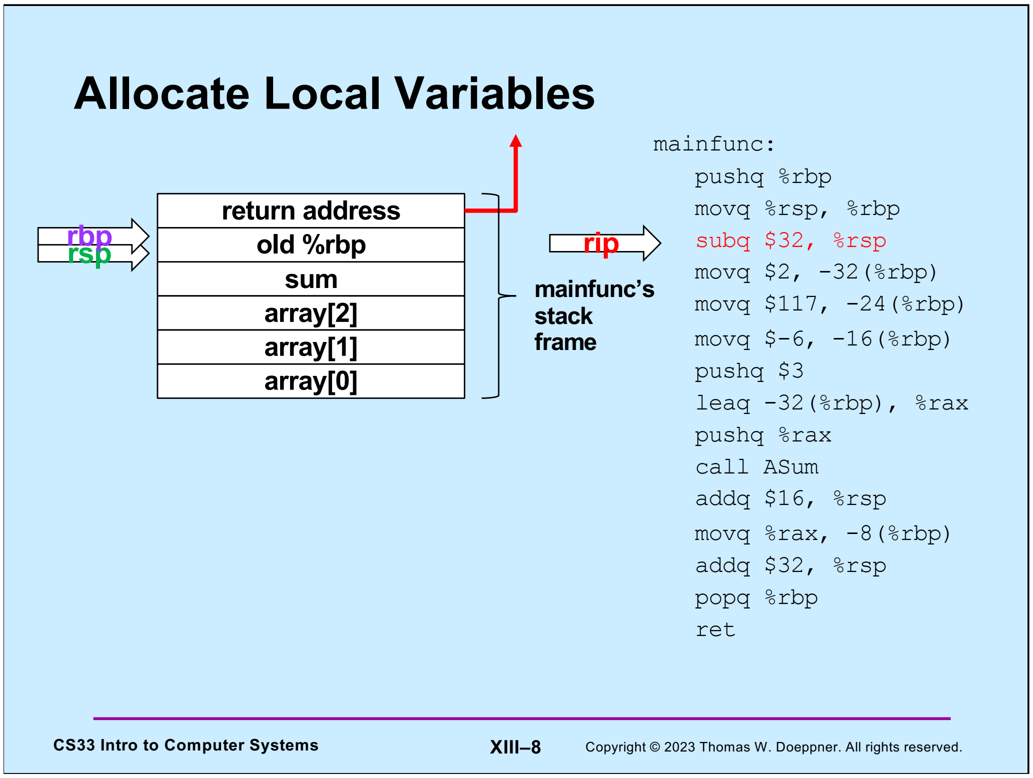
The first thing done by **mainfunc** is to save the caller's %rbp by pushing it onto the stack.



Setup Frame

```
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
```

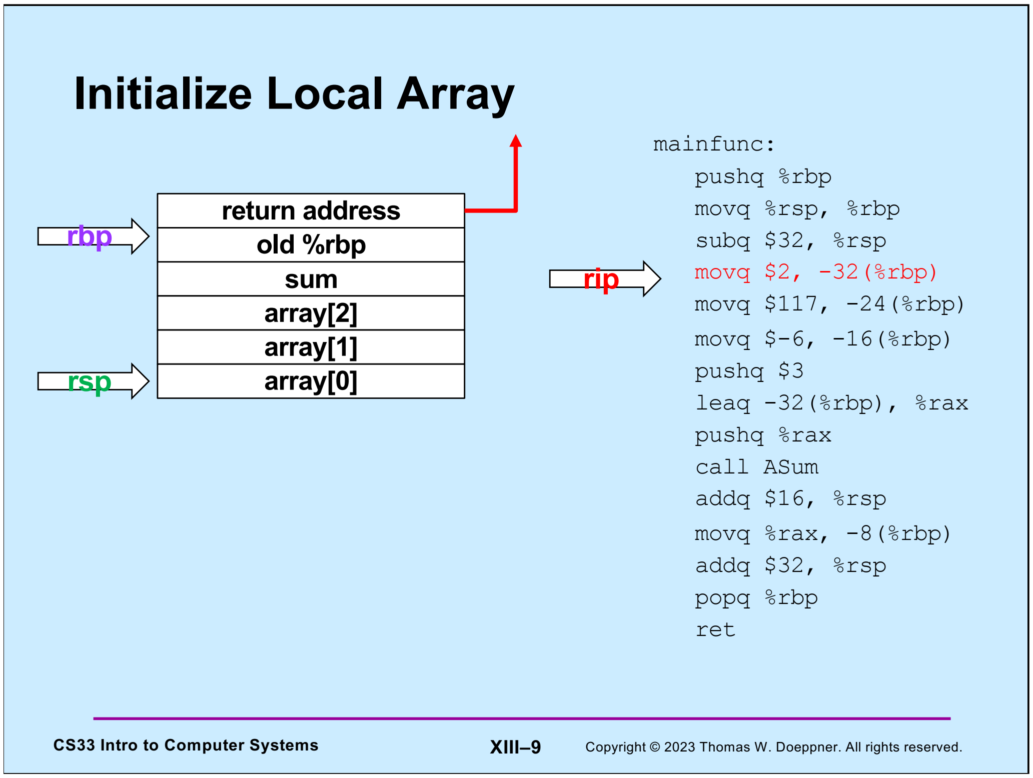
We then set up the new value of %rbp, so that it points to near the beginning of mainfunc’s stack frame.



Allocate Local Variables

```
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
```

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.



Initialize Local Array

rbp

rsp

return address

old %rbp

sum

array[2]

array[1]

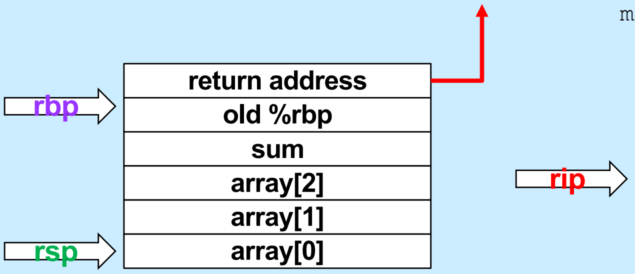
array[0]

rip

```
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
```

ASum now initializes the stack space containing its local variables.

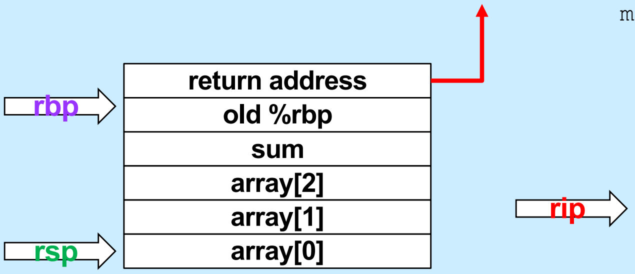
Initialize Local Array



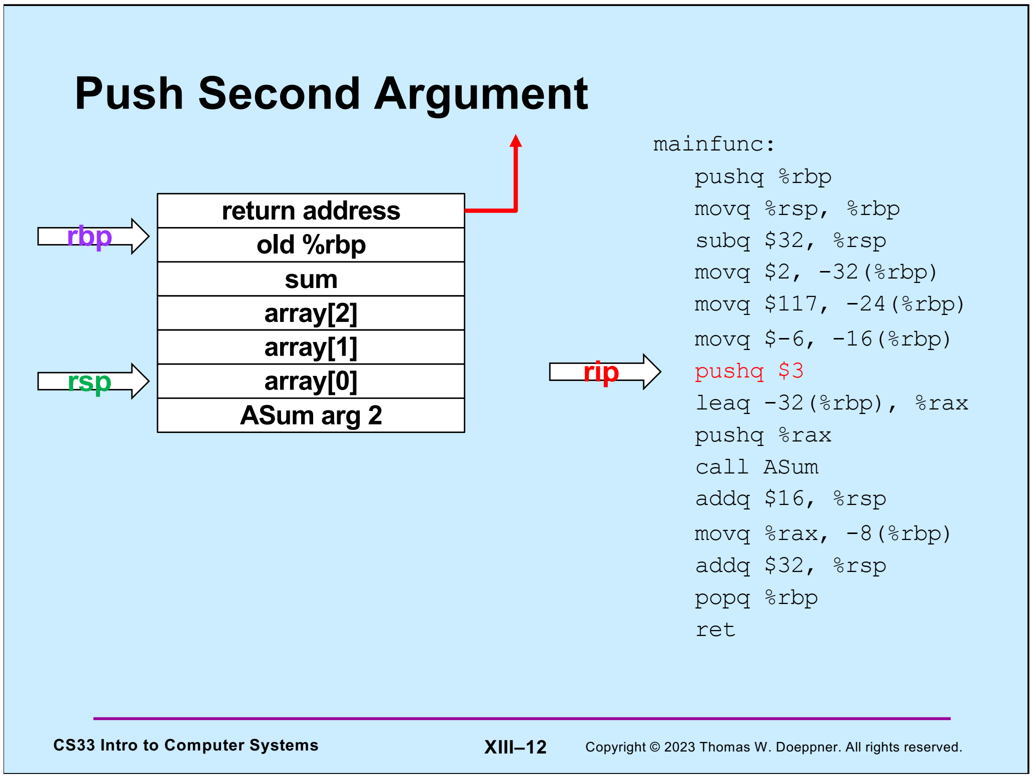
return address
old %rbp
sum
array[2]
array[1]
array[0]

```
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
```

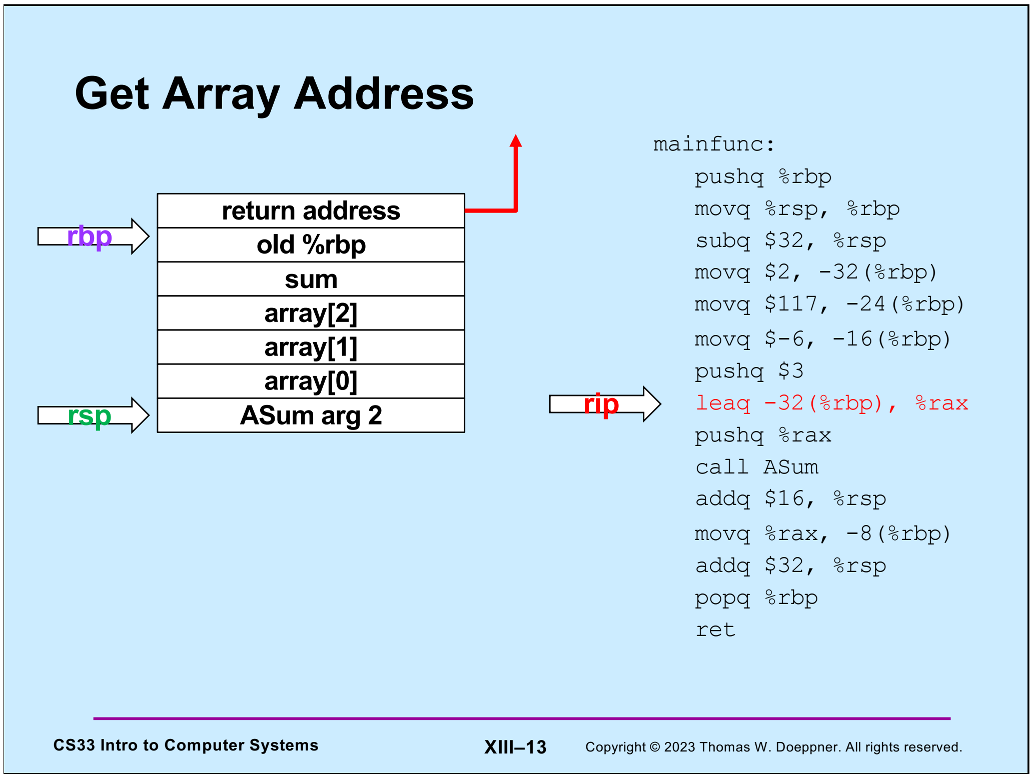


Push Second Argument

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2

```
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.



Get Array Address

rbp

rsp

return address

old %rbp

sum

array[2]

array[1]

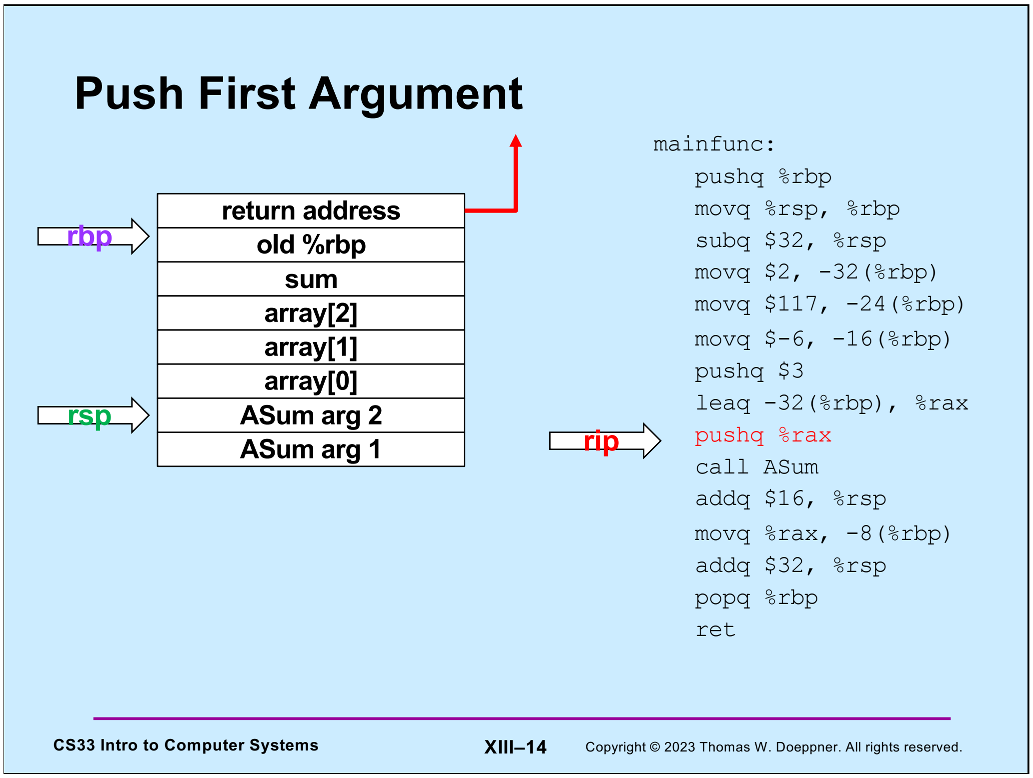
array[0]

ASum arg 2

rip

```
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
```

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

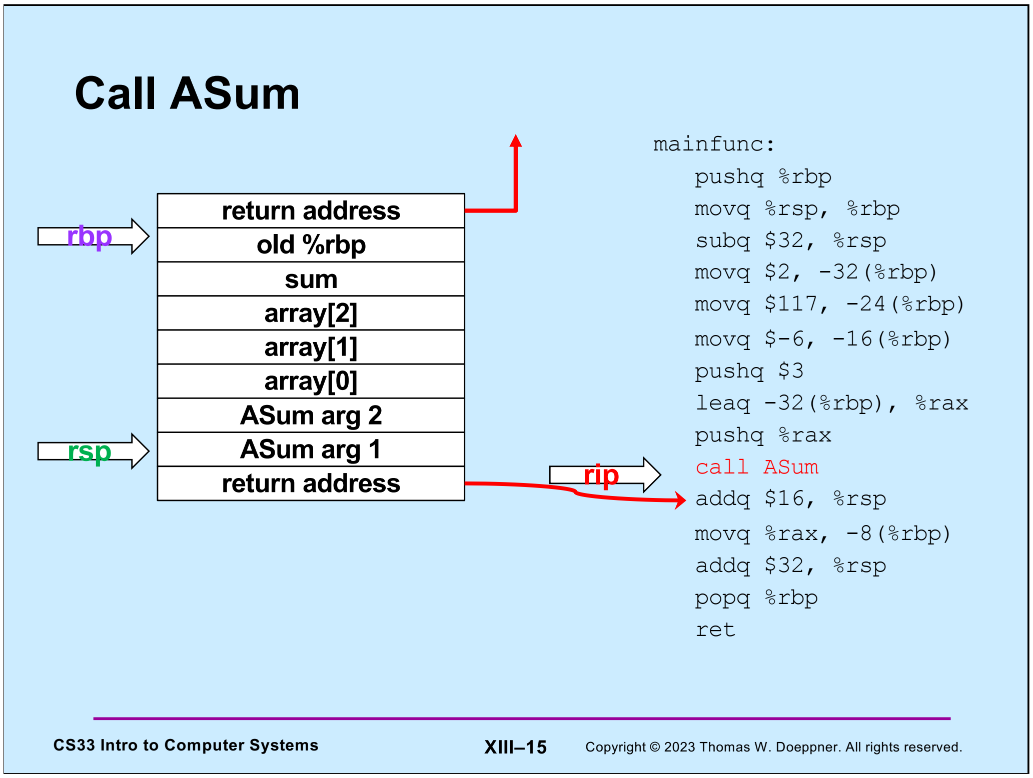


Push First Argument

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1

```
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
```

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

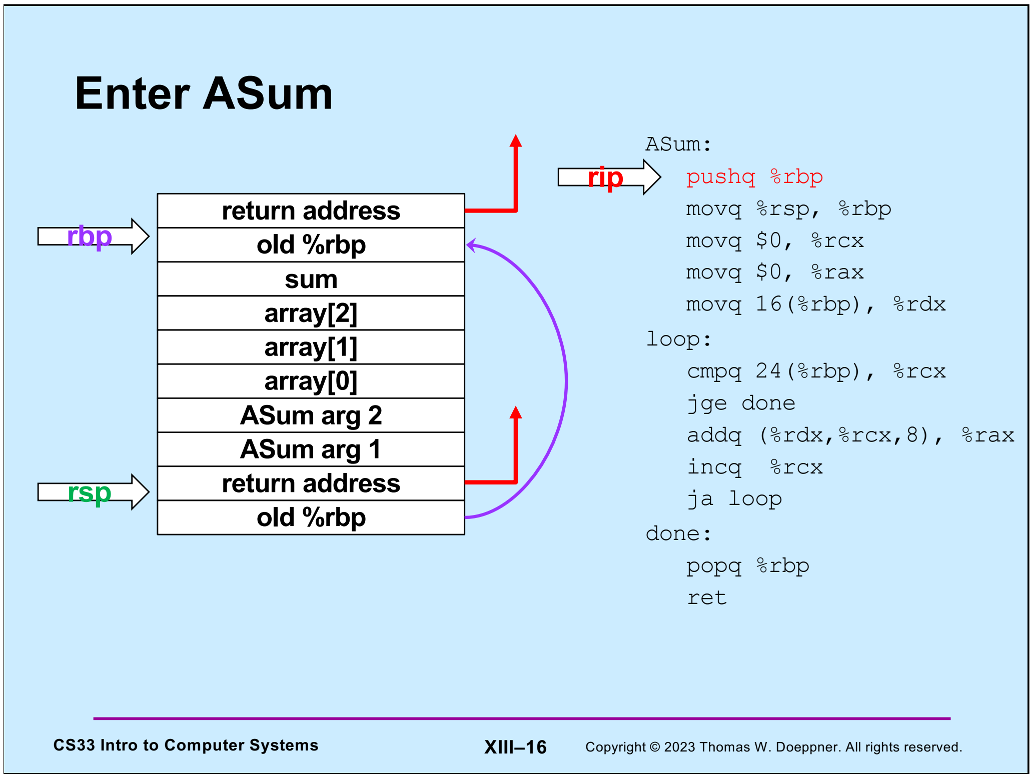


Call ASum

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address

```
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
```

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

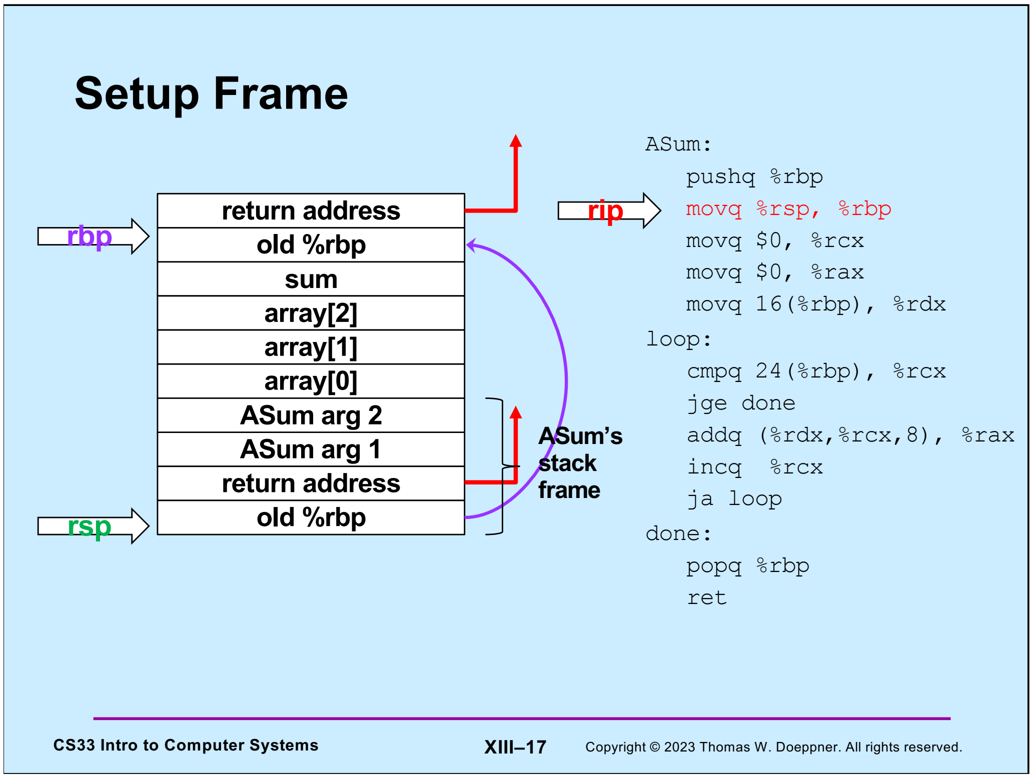


Enter ASum

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

```
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
```

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



Setup Frame

rbp

rsp

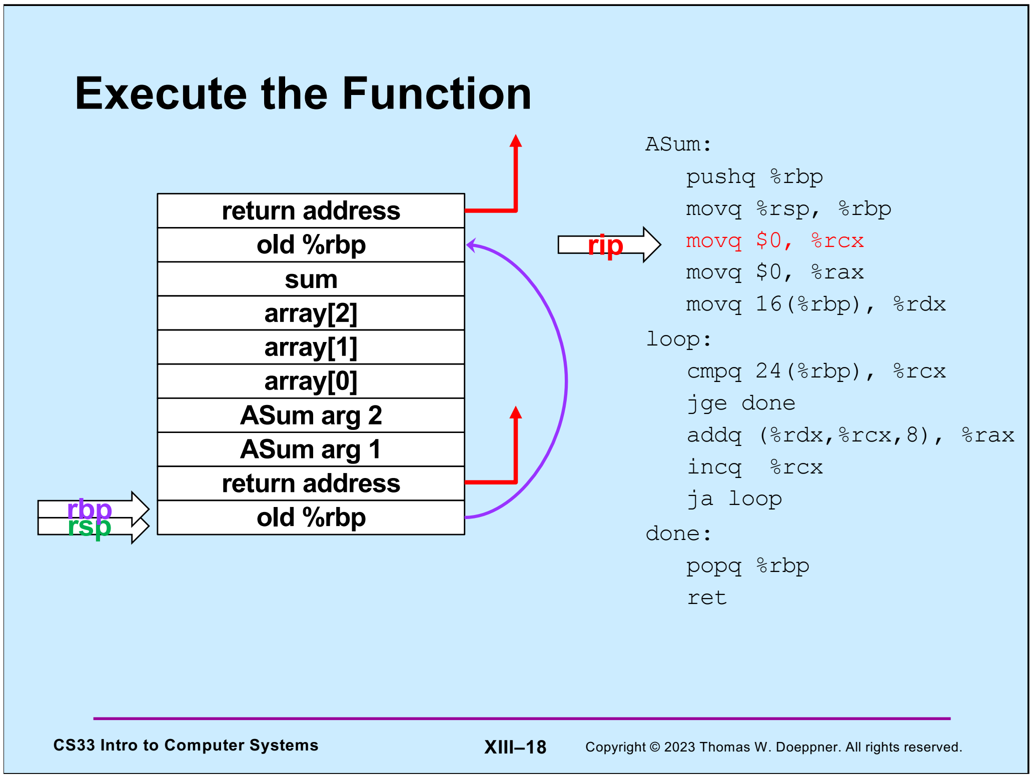
return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

ASum's stack frame

rip

```
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
```

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

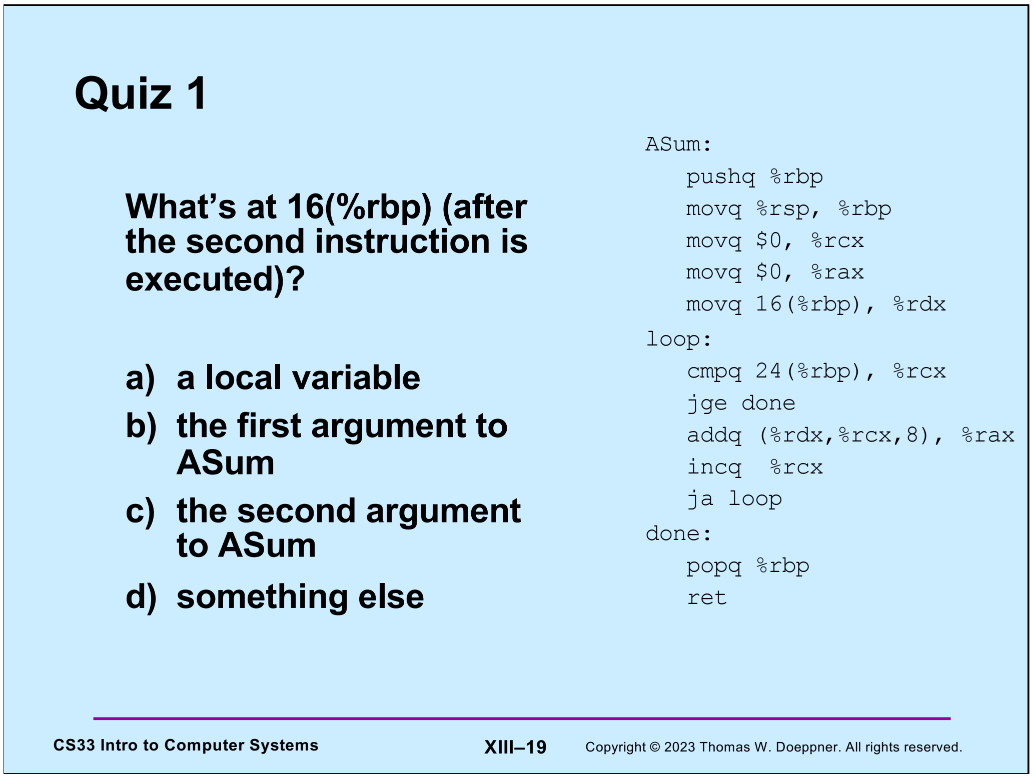


Execute the Function

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

```
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'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) (after the second instruction is executed)?

a) a local variable

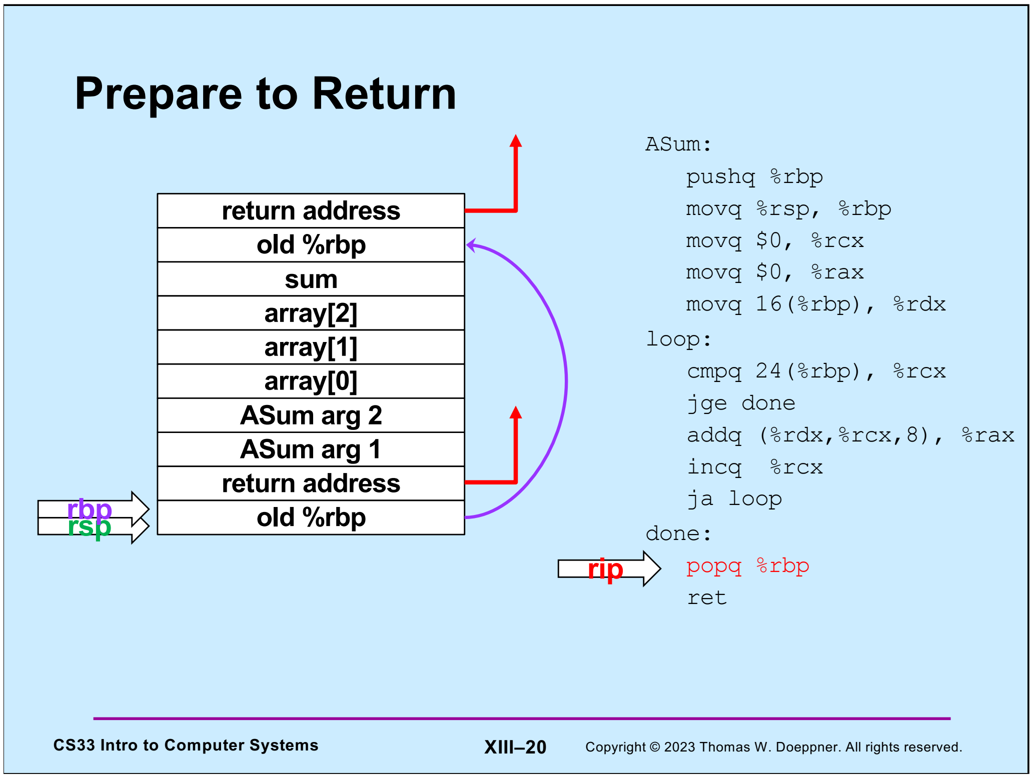
b) the first argument to ASum

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
```

Recall that when the function was entered, %rsp pointed to the return address (on the stack). It now points to something that’s 8 bytes below that. Also recall that arguments to a function are pushed onto the stack in reverse order.

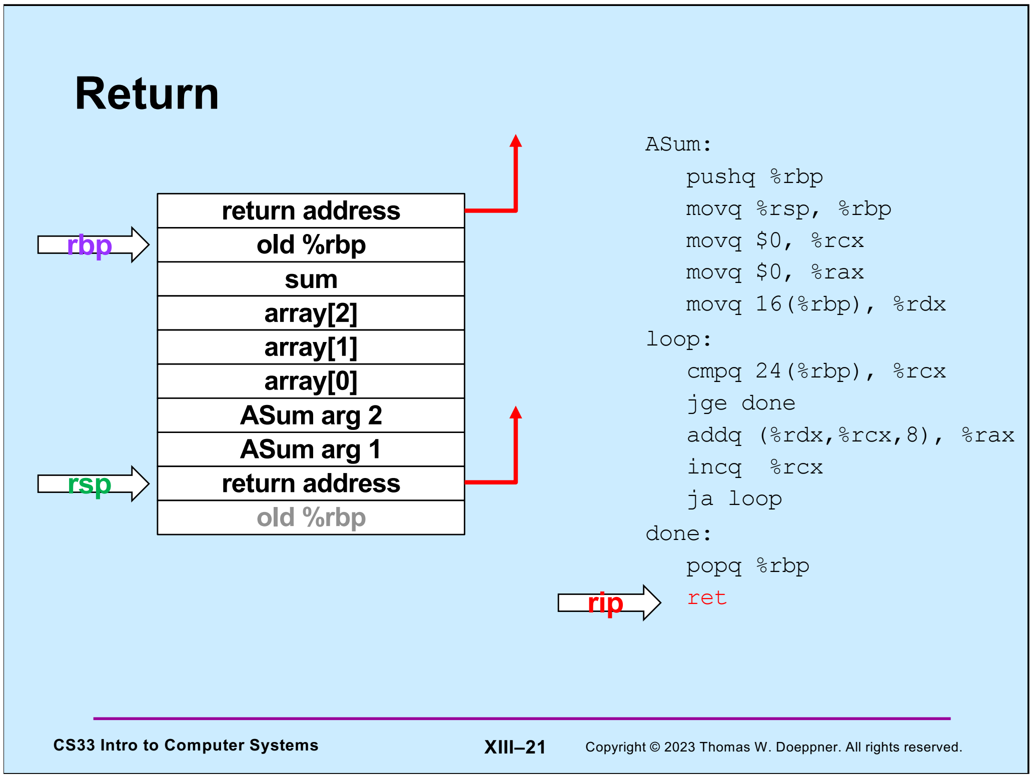


Prepare to Return

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

```
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.

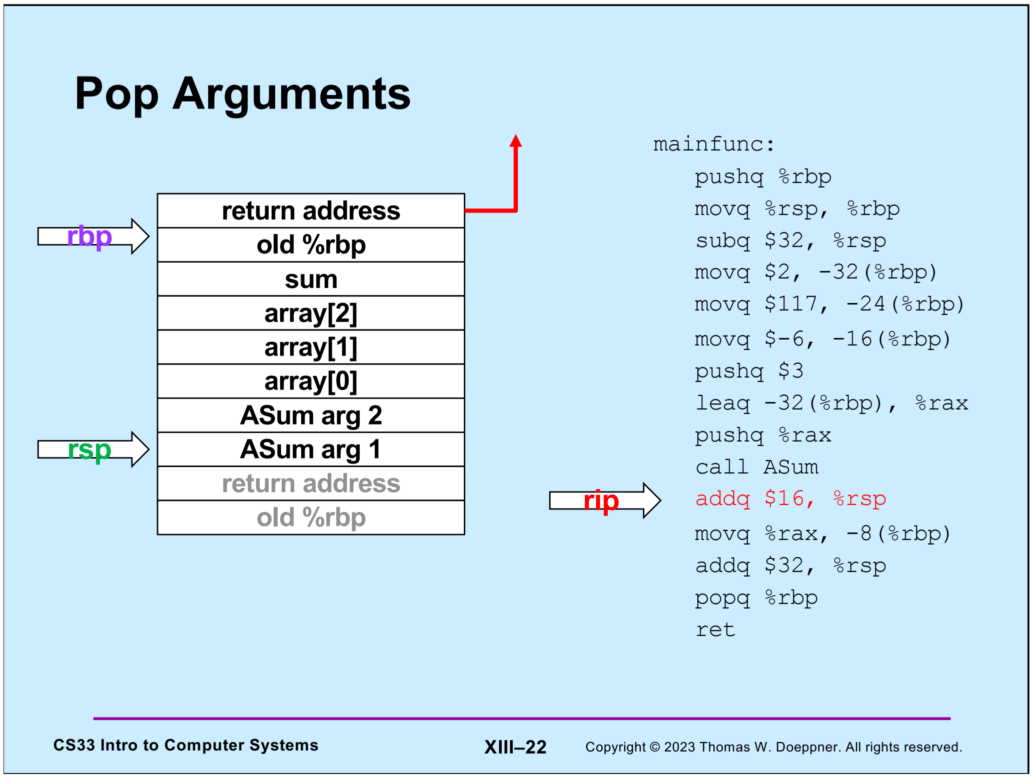


Return

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

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



Pop Arguments

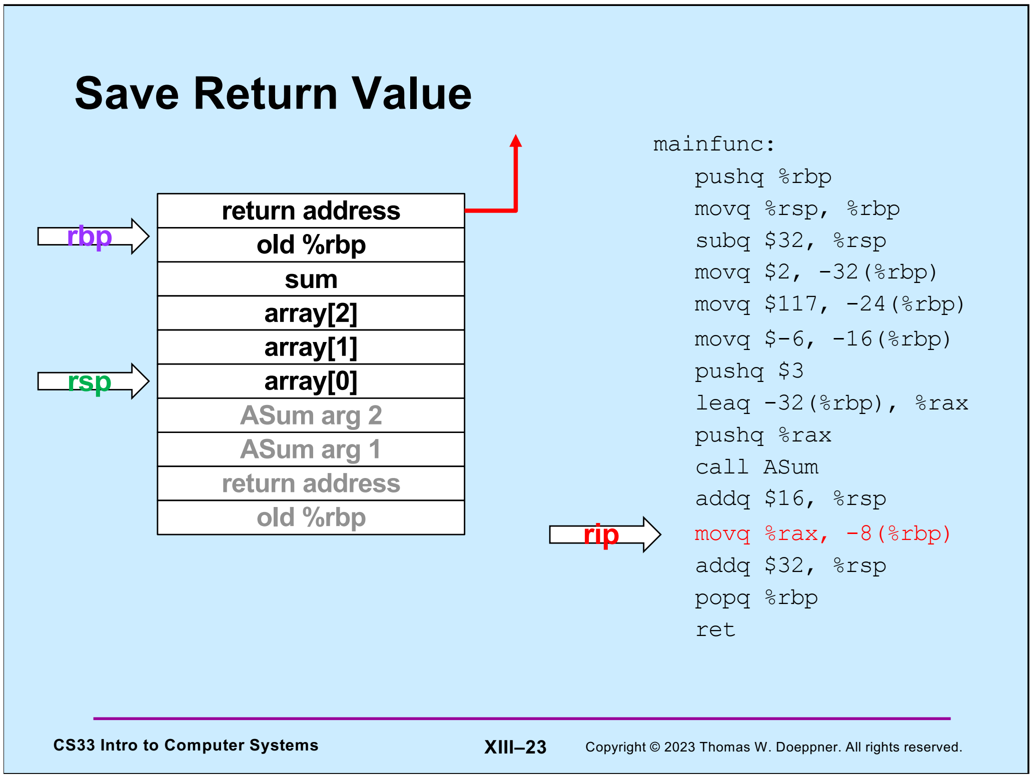
rbp
rsp

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

rip

```
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
```

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.



Save Return Value

rbp

rsp

return address

old %rbp

sum

array[2]

array[1]

array[0]

ASum arg 2

ASum arg 1

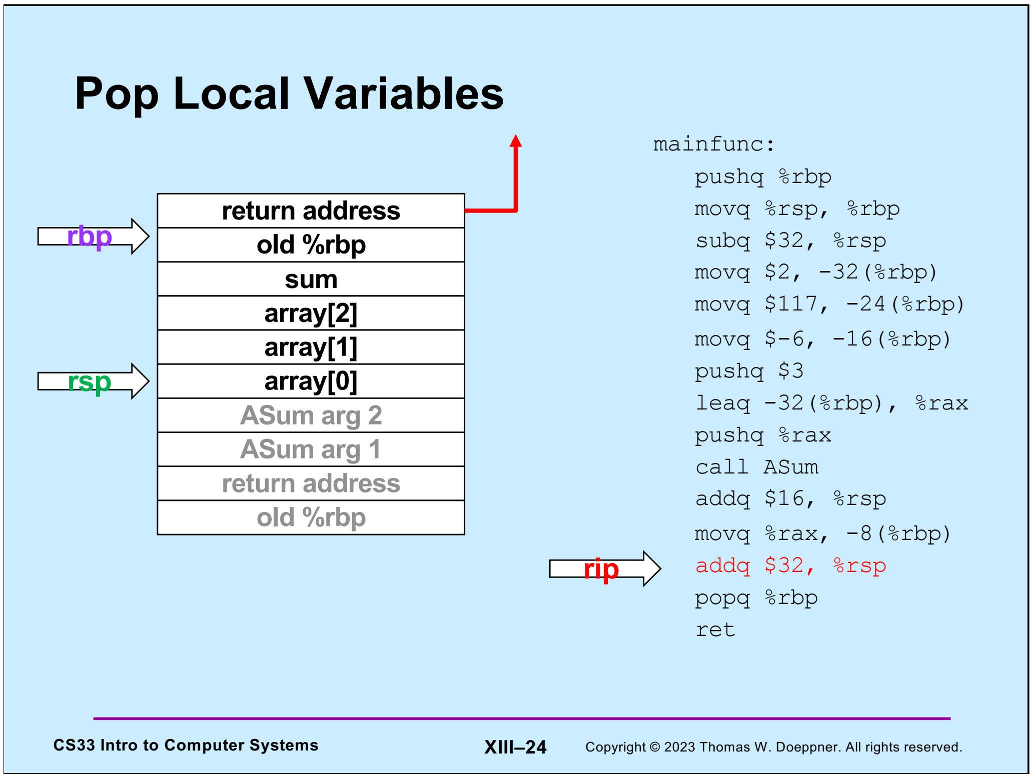
return address

old %rbp

rip

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



Pop Local Variables

rbp

rsp

return address

old %rbp

sum

array[2]

array[1]

array[0]

ASum arg 2

ASum arg 1

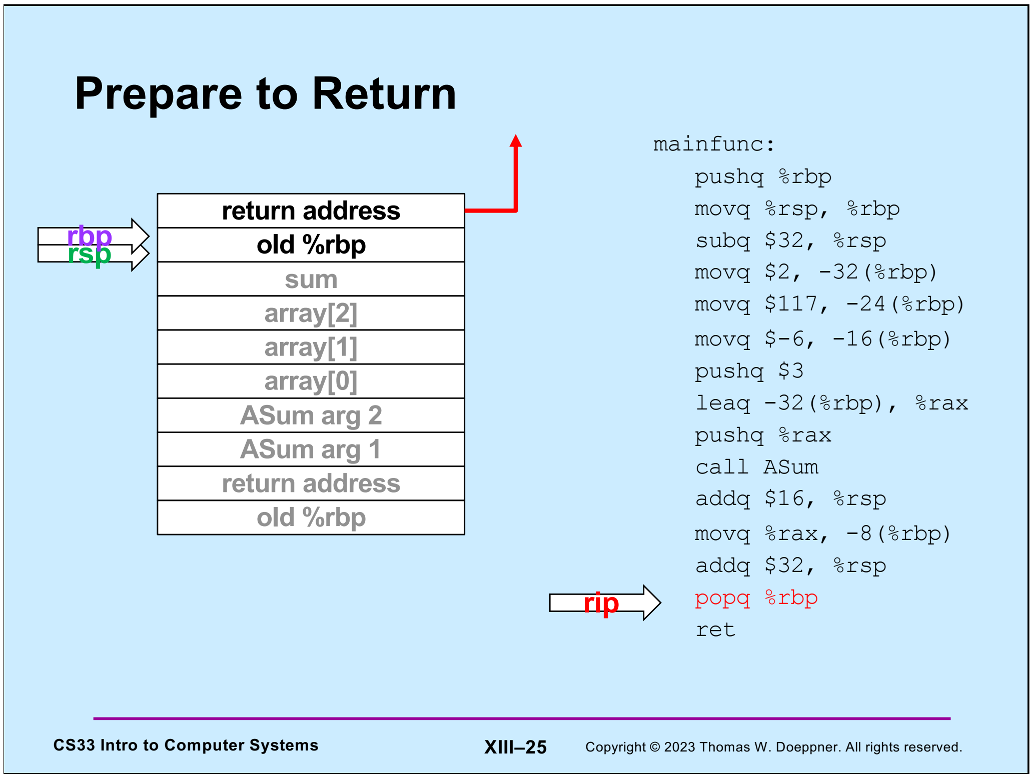
return address

old %rbp

rip

```
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
```

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

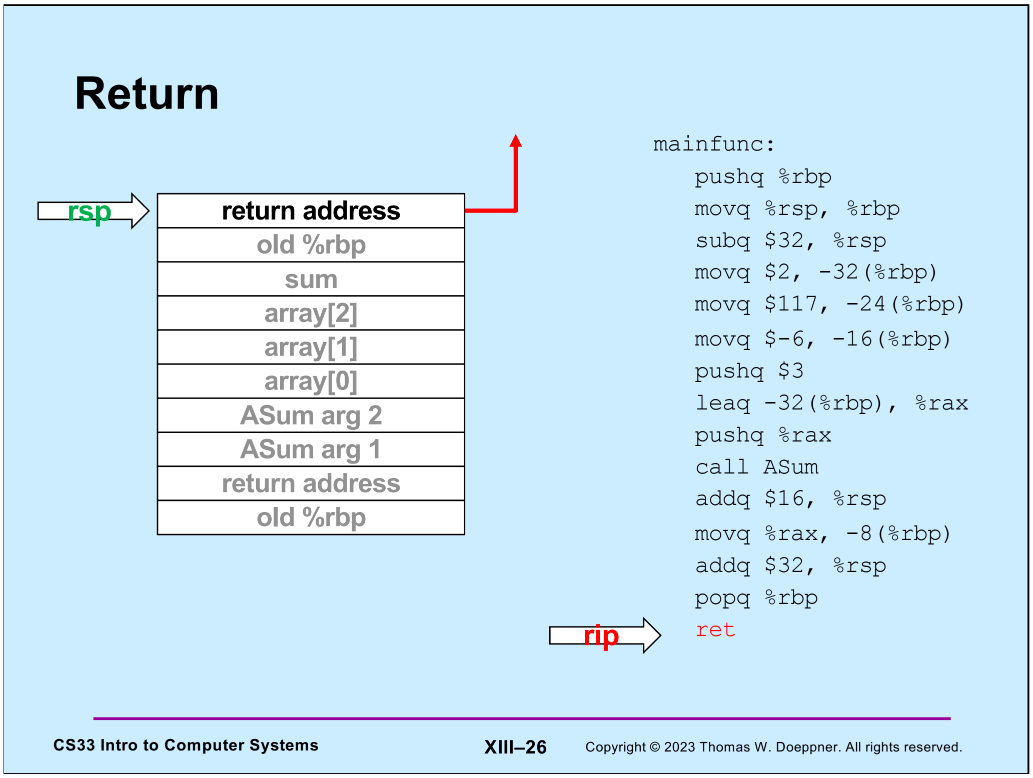


Prepare to Return

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

```
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
```

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

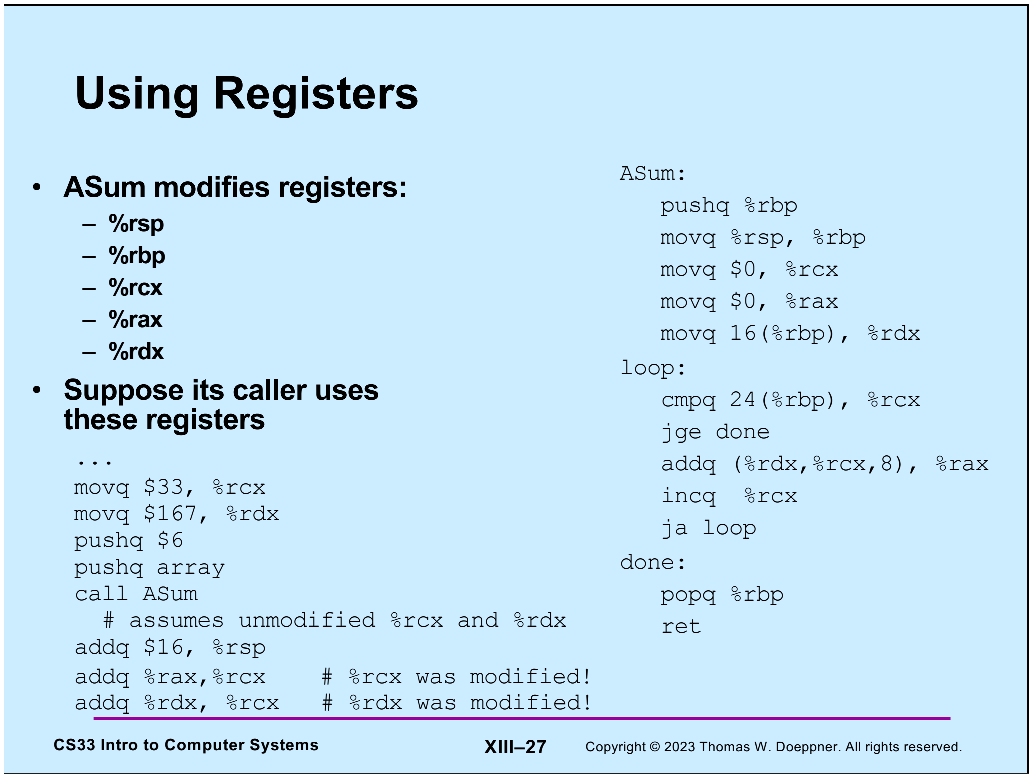


Return

return address
old %rbp
sum
array[2]
array[1]
array[0]
ASum arg 2
ASum arg 1
return address
old %rbp

```
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
```

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



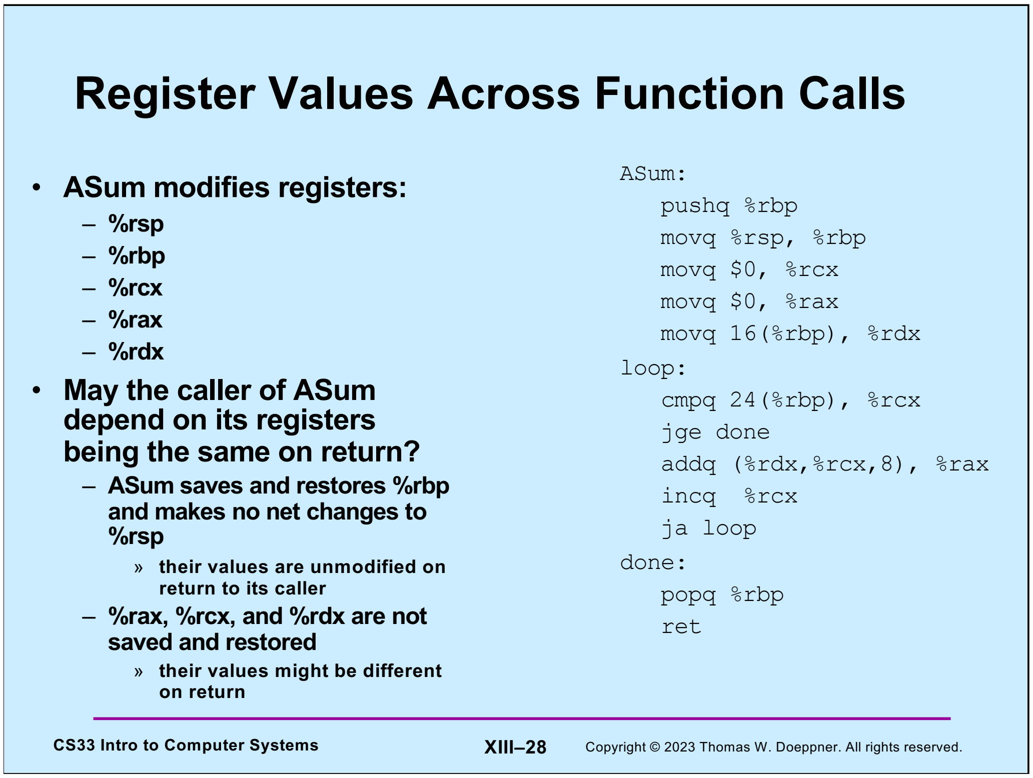
Using Registers

- **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
```

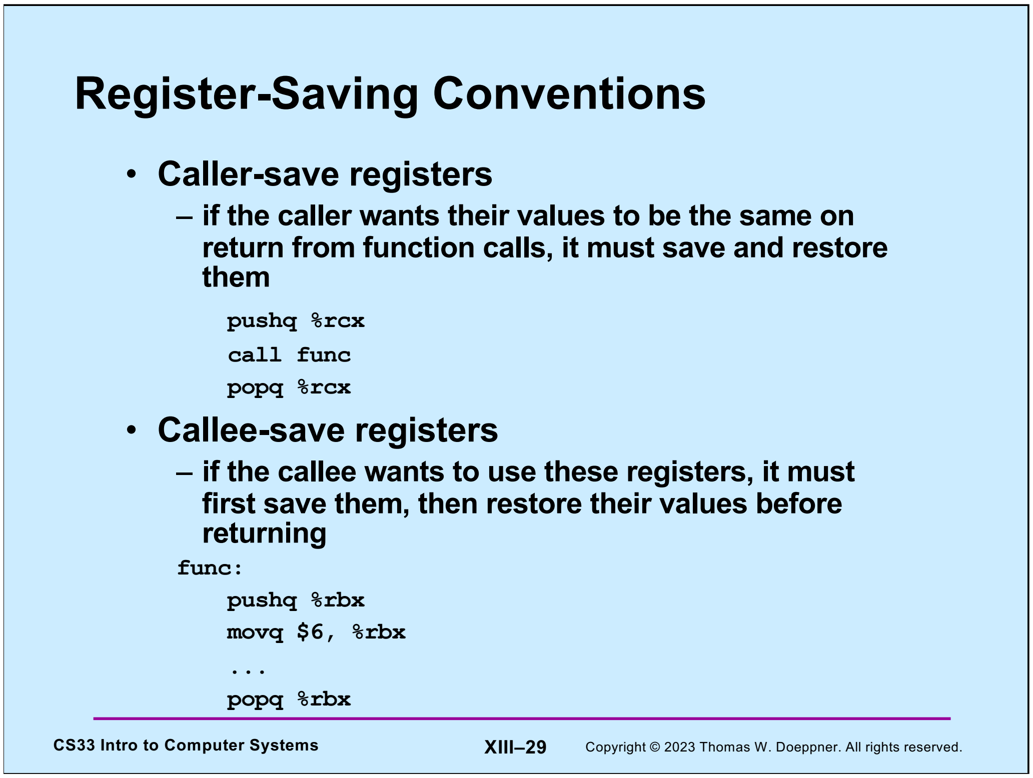
ASum modified a number of registers. But suppose its caller was using these registers and depended on their values' being unchanged?



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**

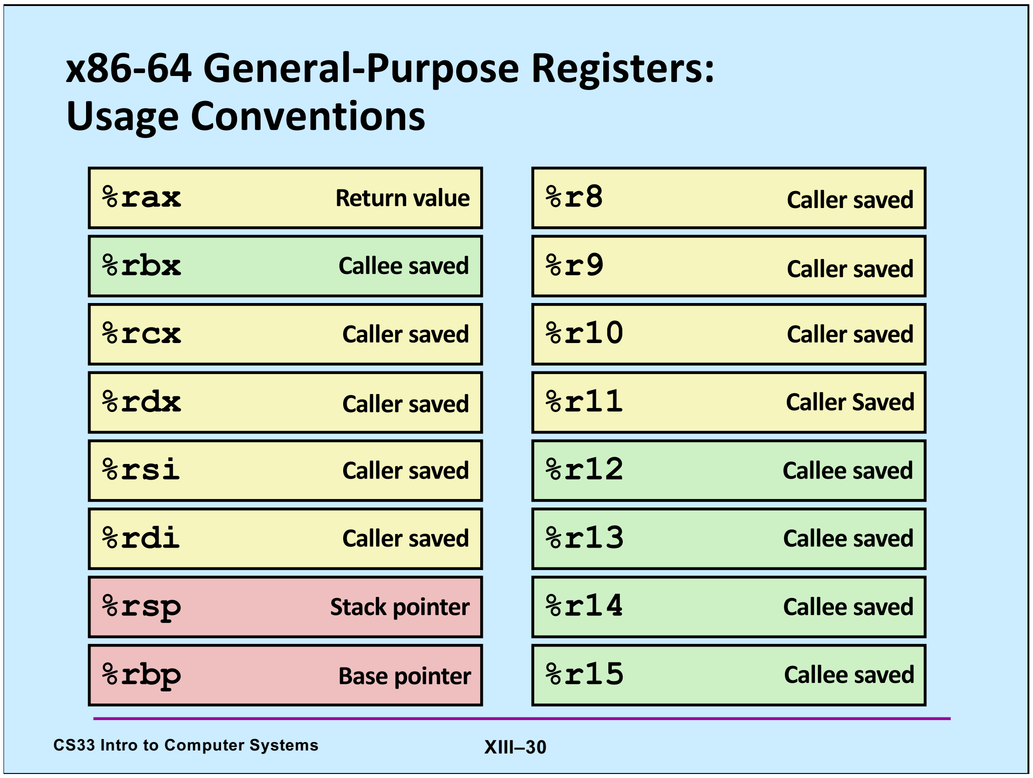
```
    pushq %rcx
    call func
    popq %rcx
```

- **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.

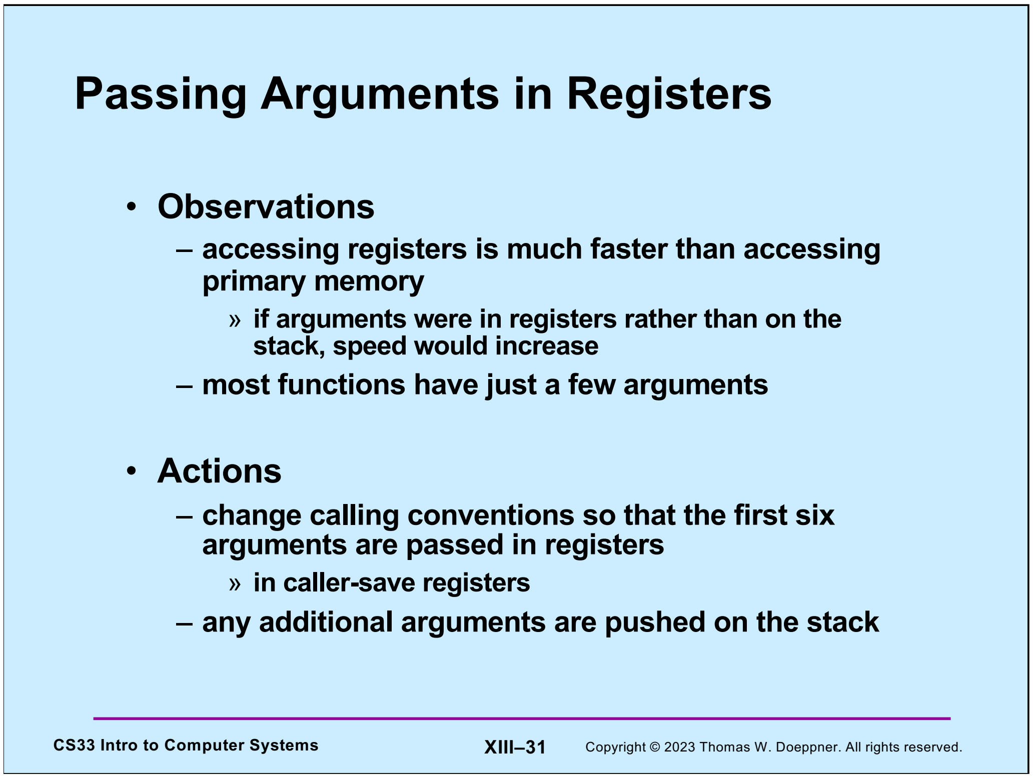


x86-64 General-Purpose Registers: Usage Conventions

%rax	Return value	%r8	Caller saved
%rbx	Callee saved	%r9	Caller saved
%rcx	Caller saved	%r10	Caller saved
%rdx	Caller saved	%r11	Caller Saved
%rsi	Caller saved	%r12	Callee saved
%rdi	Caller saved	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Base pointer	%r15	Callee saved

Based on a slide supplied by CMU.

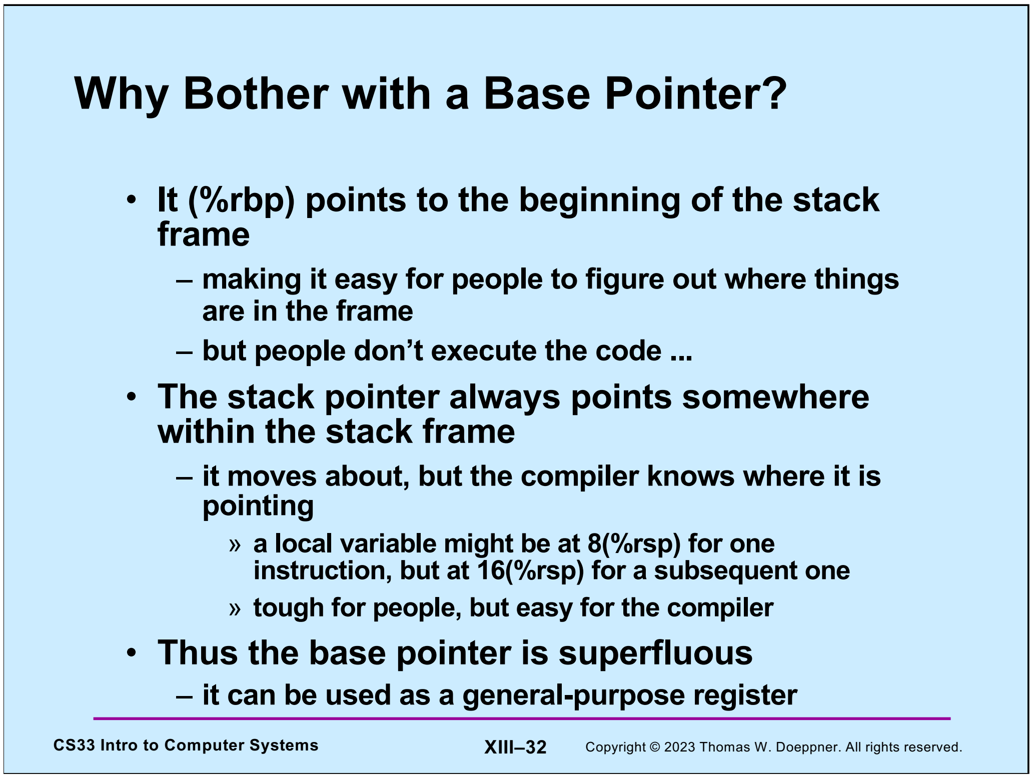
Here is a list of which registers are callee-save, which are caller-save, and which have special purposes. Note that this is merely a convention and not an inherent aspect of the x86-64 architecture.



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

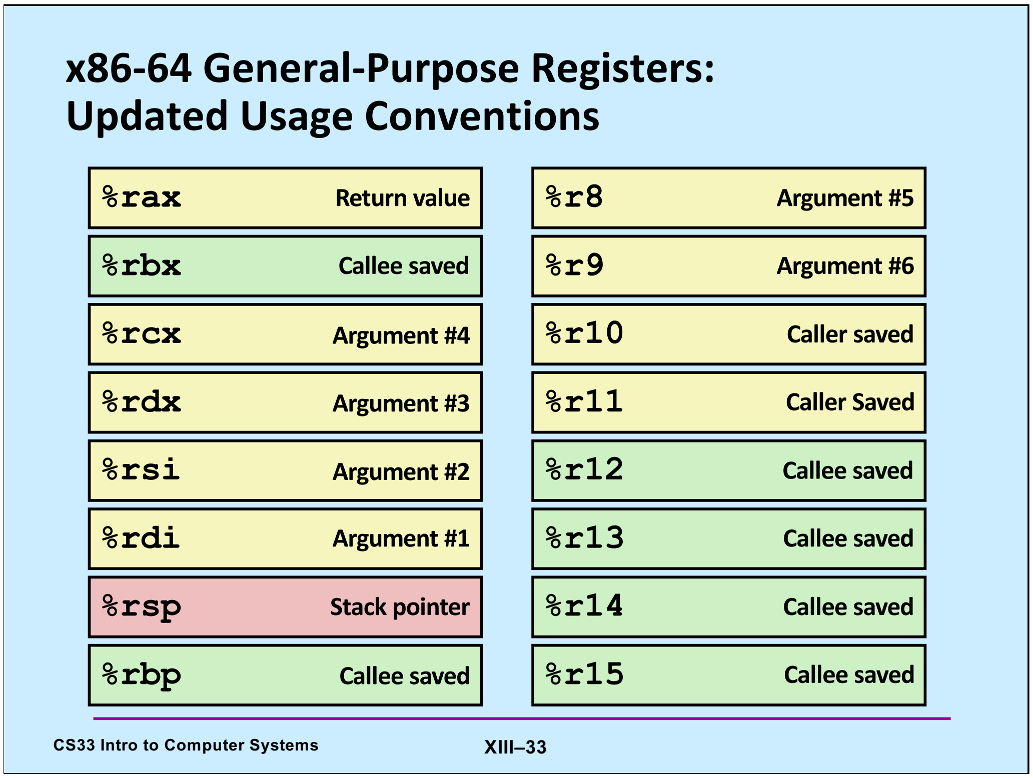
- Actions
  - – change calling conventions so that the first six arguments are passed in registers
    - » in caller-save registers
  - – any additional arguments are pushed on the stack



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

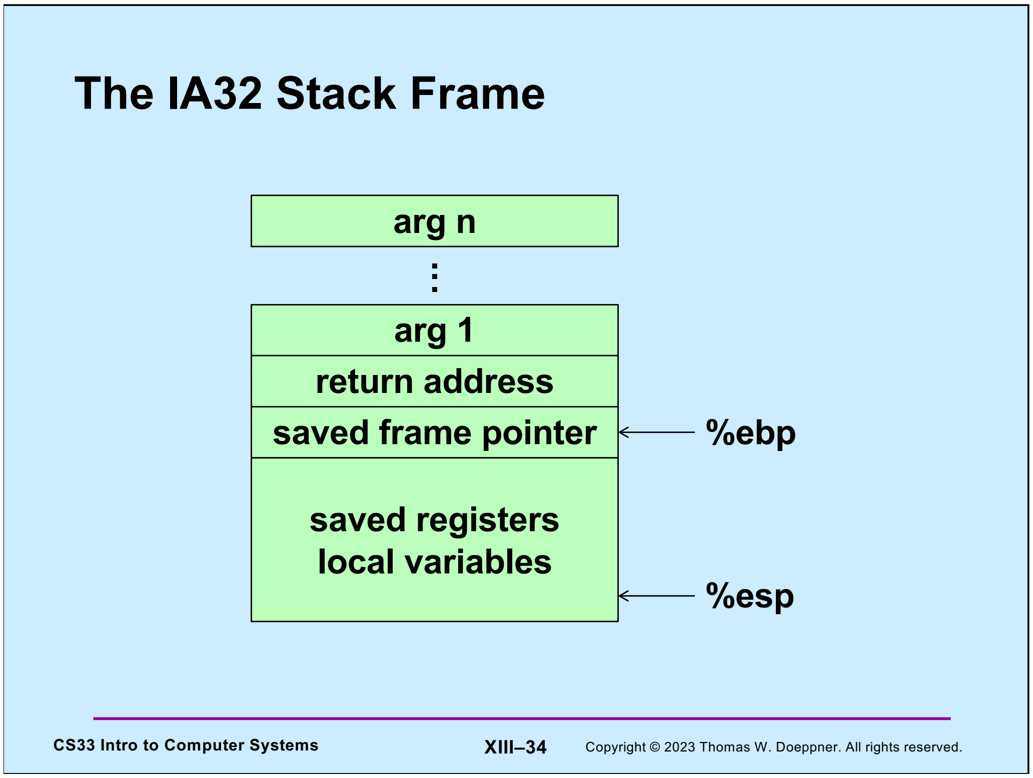
If one gives gcc the –O0 flag (which turns off all optimization) when compiling, the base pointer (%rbp) will be used as in IA32: it is set to point to the stack frame and the arguments are copied from the registers into the stack frame. This clearly slows down the execution of the function, but makes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



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.



The IA32 Stack Frame

arg n

⋮

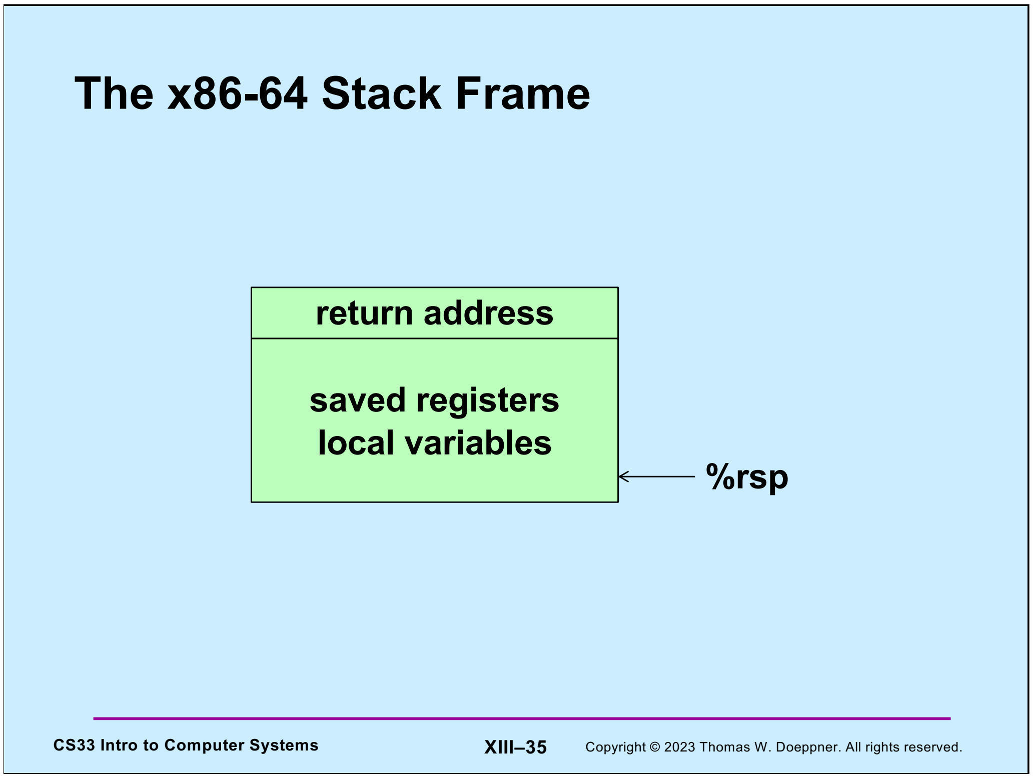
arg 1

return address

saved frame pointer

saved registers
local variables

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



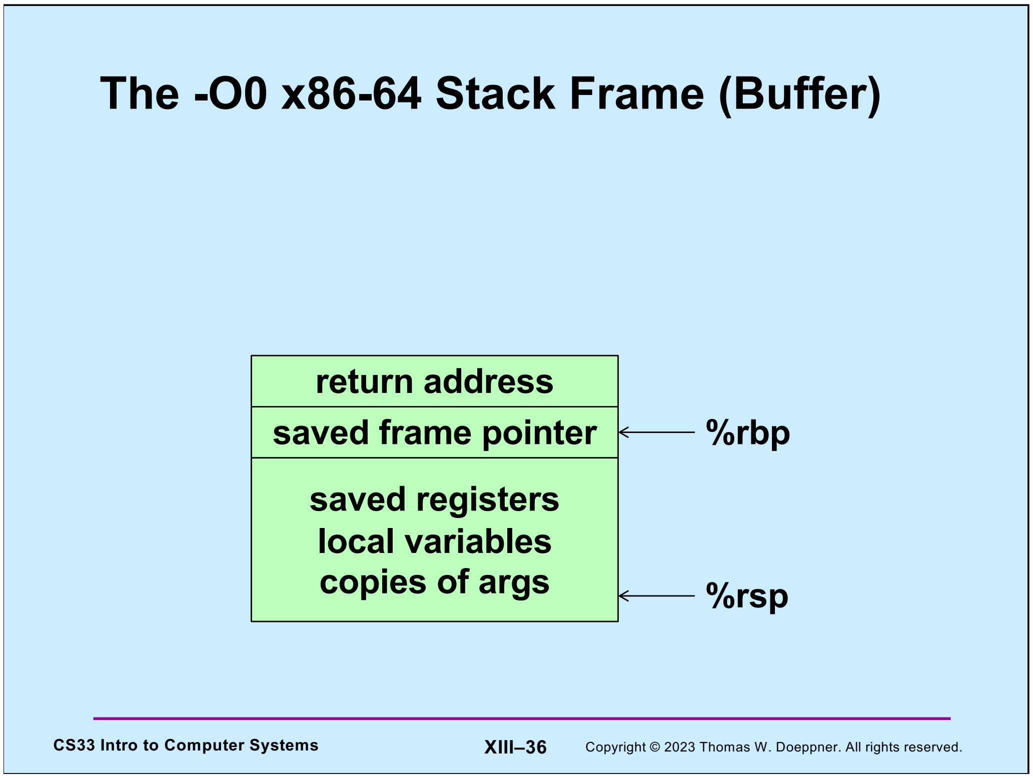
The x86-64 Stack Frame

return address

saved registers
local variables

%rsp

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.



The -O0 x86-64 Stack Frame (Buffer)

return address

saved frame pointer

saved registers
local variables
copies of args

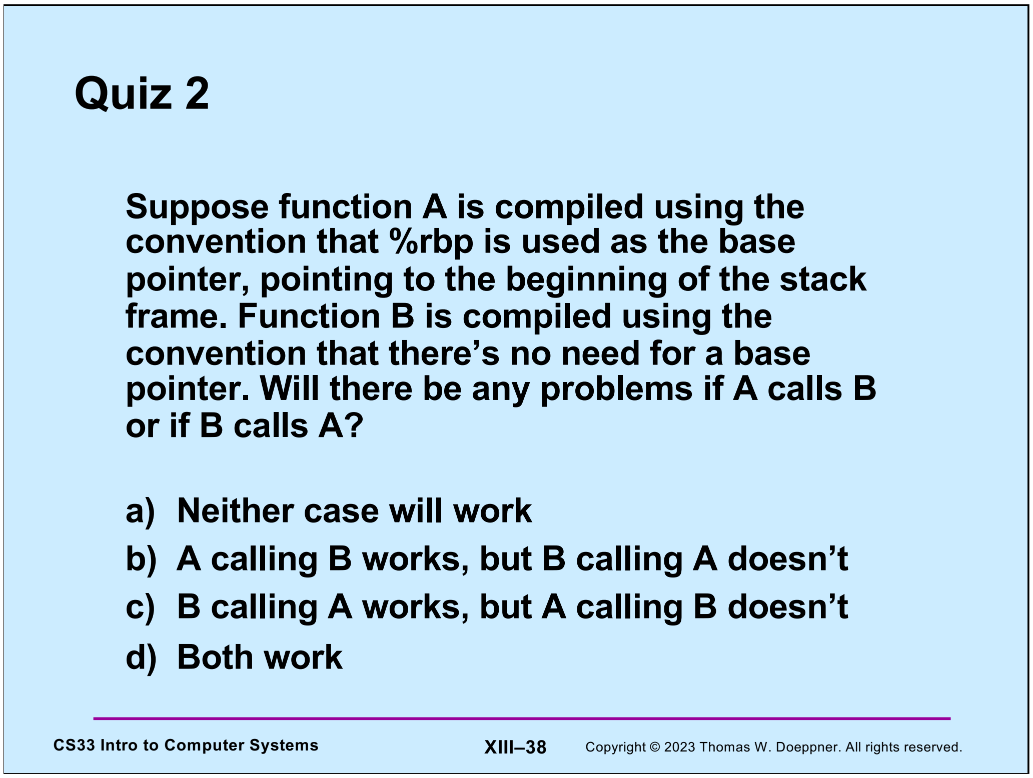
%rbp

%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). The code for the buffer project (to be released on Friday) is compiled with the –O0 flag.

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

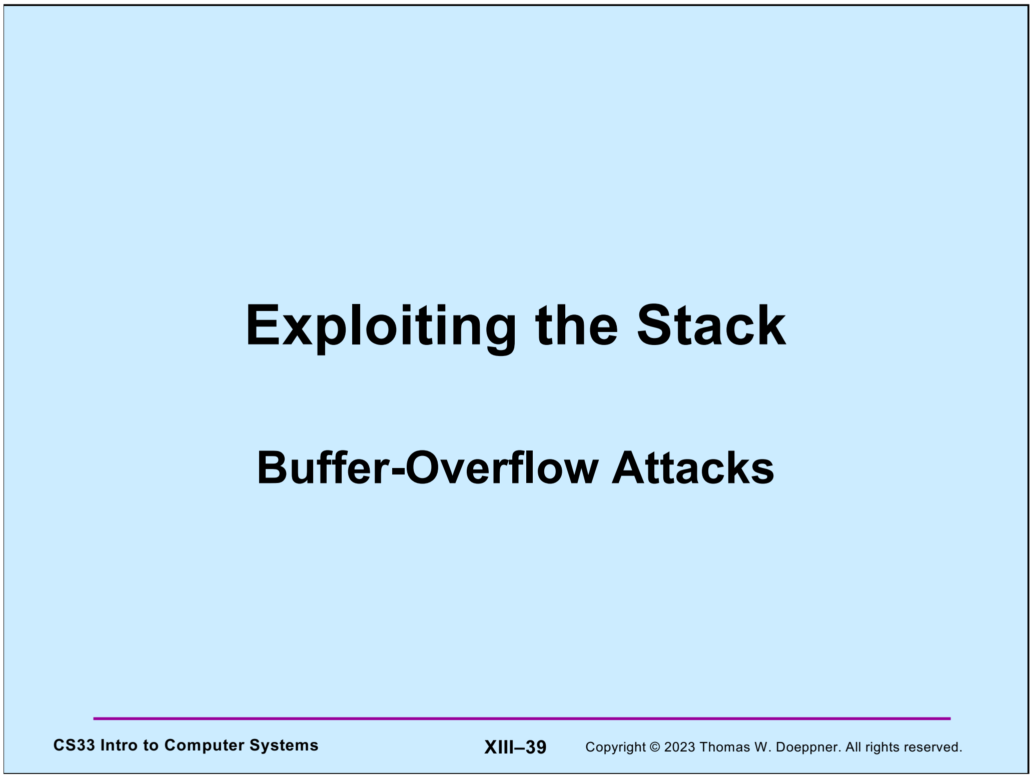


Quiz 2

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?

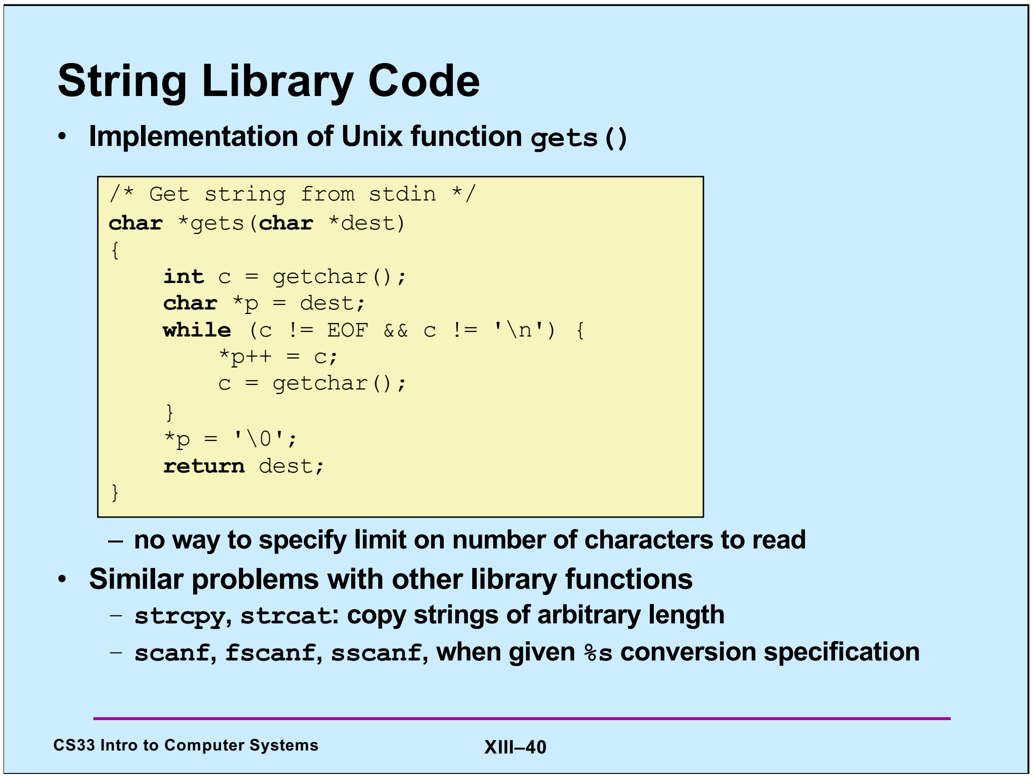
a) Neither case will work
b) A calling B works, but B calling A doesn’t
c) B calling A works, but A calling B doesn’t
d) Both work

Recall that %rbp is a callee-saved register.



Exploiting the Stack

Buffer-Overflow Attacks



String Library Code

- Implementation of Unix function `gets()`

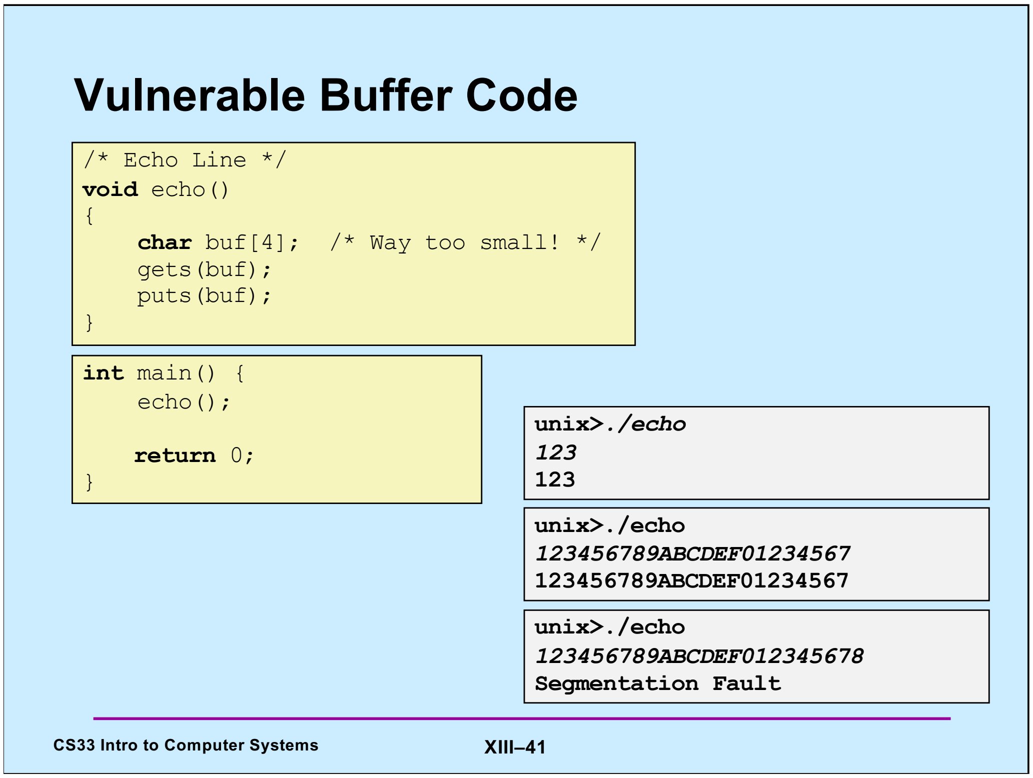
```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

  – no way to specify limit on number of characters to read

- Similar problems with other library functions
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification

Supplied by CMU.

The function getchar returns the next character to be typed in. If getchar returns EOF (which is coded as a byte containing all ones – not a coding of any valid ASCII character, but -1 if the byte is interpreted as a signed integer).



Vulnerable Buffer Code

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

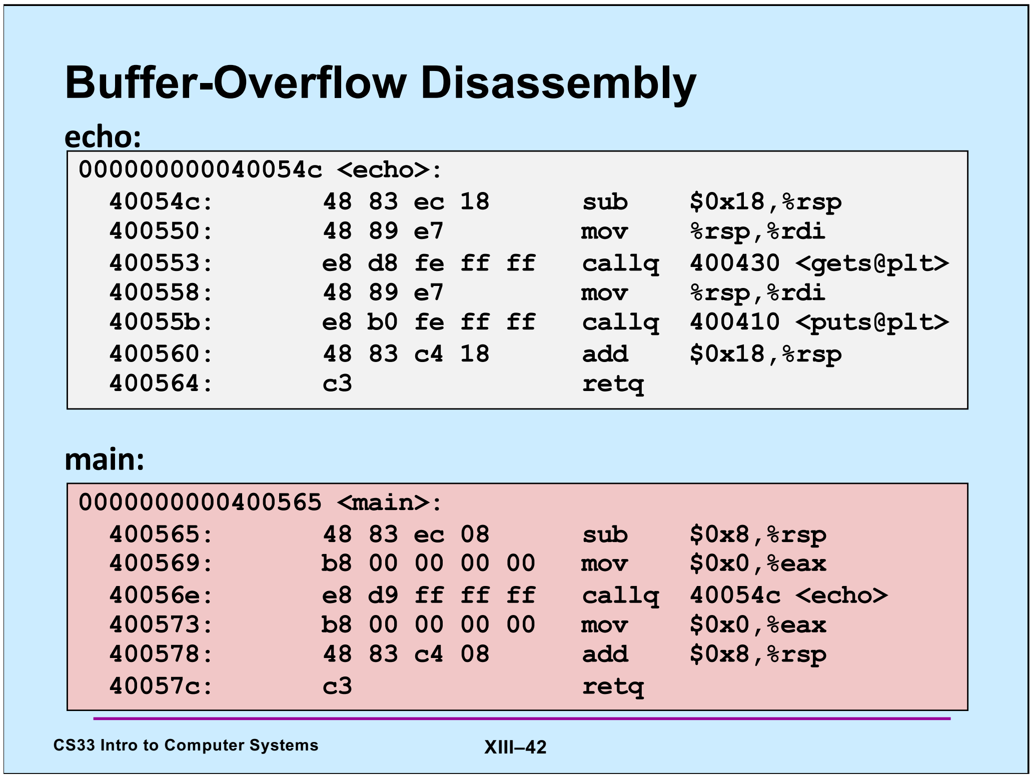
```
int main() {
    echo();

    return 0;
}
```

```
unix>./echo
123
123
```

```
unix>./echo
123456789ABCDEF01234567
123456789ABCDEF01234567
```

```
unix>./echo
123456789ABCDEF012345678
Segmentation Fault
```

Supplied by CMU, but adapted for x86-64.



Buffer-Overflow Disassembly

echo:

```
0000000000040054c <echo>:
  40054c:       48 83 ec 18      sub    $0x18,%rsp
  400550:       48 89 e7         mov    %rsp,%rdi
  400553:       e8 d8 fe ff ff   callq  400430 <gets@plt>
  400558:       48 89 e7         mov    %rsp,%rdi
  40055b:       e8 b0 fe ff ff   callq  400410 <puts@plt>
  400560:       48 83 c4 18      add    $0x18,%rsp
  400564:       c3               retq
```

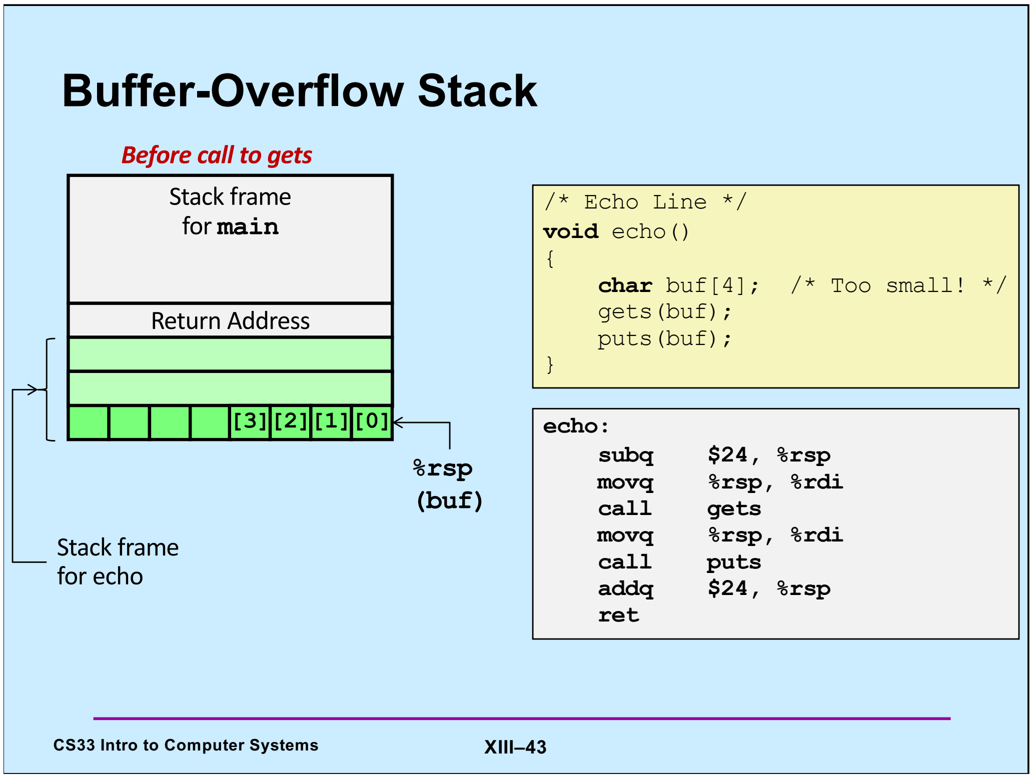
main:

```
0000000000400565 <main>:
  400565:       48 83 ec 08      sub    $0x8,%rsp
  400569:       b8 00 00 00 00   mov    $0x0,%eax
  40056e:       e8 d9 ff ff ff   callq  40054c <echo>
  400573:       b8 00 00 00 00   mov    $0x0,%eax
  400578:       48 83 c4 08      add    $0x8,%rsp
  40057c:       c3               retq
```

Supplied by CMU, but adapted for x86-64.

Note that 24 bytes are allocated on the stack for **buf**, rather than the 4 specified in the C code. This is an optimization having to do with the alignment of the stack pointer, a subject we will discuss in an upcoming lecture.

The text in the angle brackets after the calls to **gets** and **puts** mentions “plt”. This refers to the “procedure linkage table,” another topic we cover in an upcoming lecture. The calls are to the gets and puts functions, which are not statically linked to the program, but are dynamically linked. These concepts are not important now; we’ll cover them towards the end of the semester.



Buffer-Overflow Stack

Before call to gets

Stack frame
for **main**

Return Address

[3] [2] [1] [0]

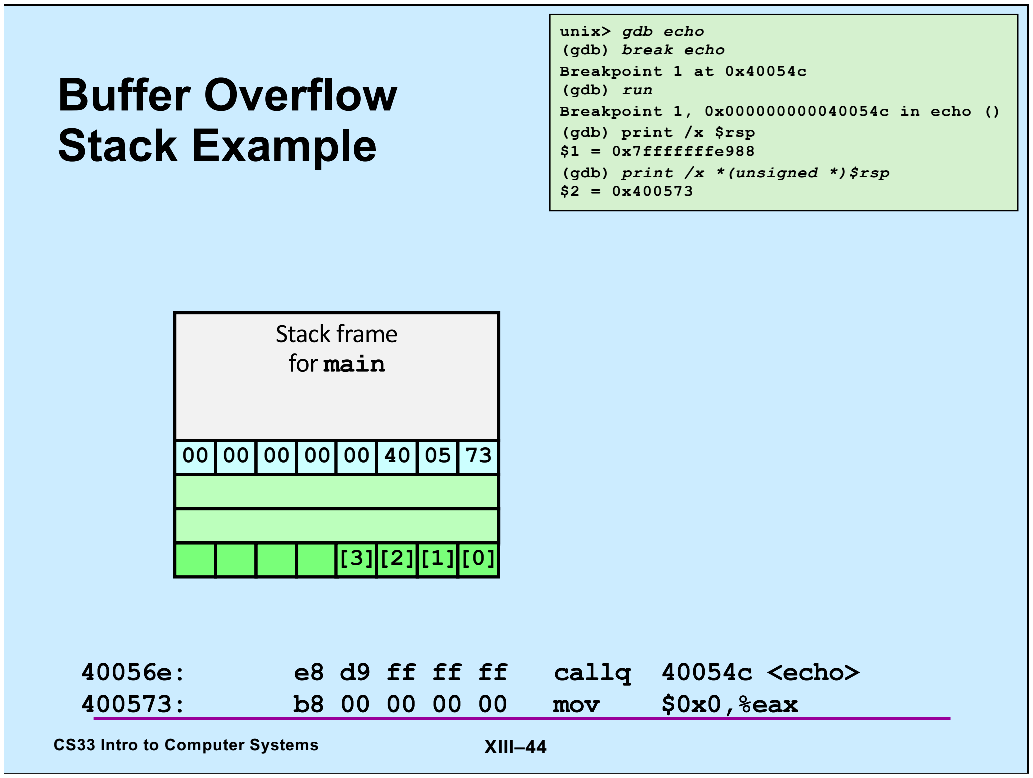
%rsp
(buf)

Stack frame
for echo

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Too small! */
    gets(buf);
    puts(buf);
}
```

```
echo:
    subq    $24, %rsp
    movq    %rsp, %rdi
    call    gets
    movq    %rsp, %rdi
    call    puts
    addq    $24, %rsp
    ret
```

Supplied by CMU, but adapted for x86-64.



Buffer Overflow Stack Example

```
unix> gdb echo
(gdb) break echo
Breakpoint 1 at 0x40054c
(gdb) run
Breakpoint 1, 0x000000000040054c in echo ()
(gdb) print /x $rsp
$1 = 0x7fffffffe988
(gdb) print /x *(unsigned *)$rsp
$2 = 0x400573
```

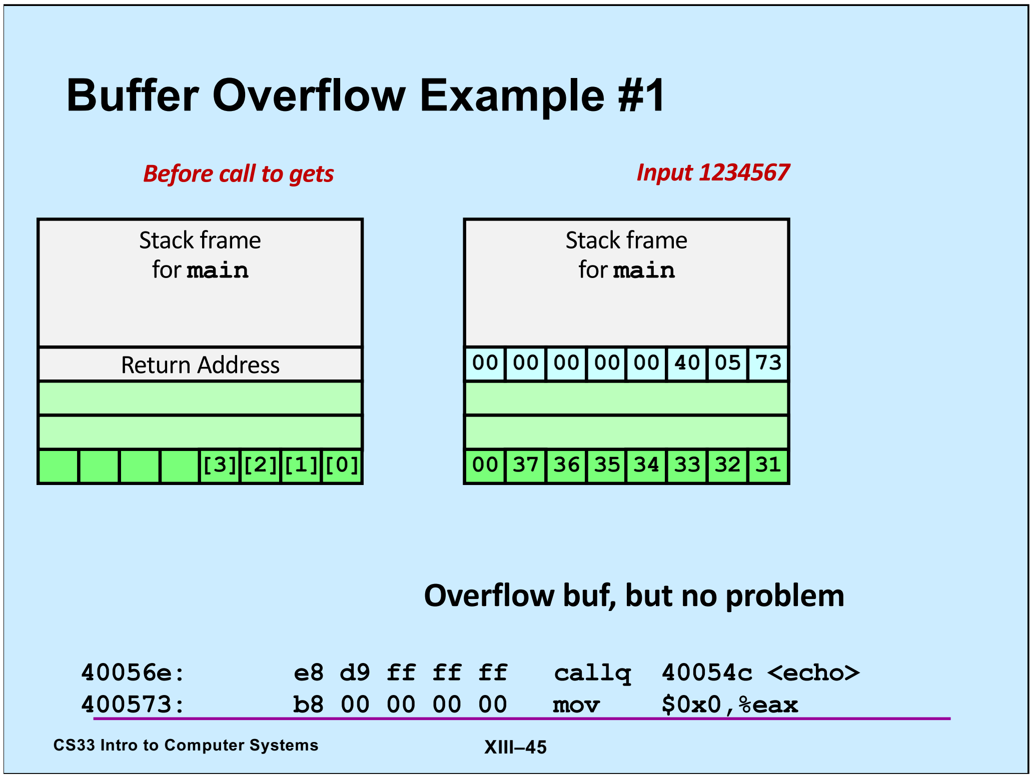
00 00 00 00 00 40 05 73

[3] [2] [1] [0]

```
40056e:        e8 d9 ff ff ff    callq  40054c <echo>
400573:        b8 00 00 00 00    mov    $0x0,%eax
```

Supplied by CMU, but adapted for x86-64.

Within gdb, the second print shows the 4-byte value at the end of the stack (i.e., pointed to by %rsp), interpreting it as an unsigned value. This is the return address, used by echo when it returns to main. What's in green will be the memory that will be allocated on the stack for buf.



Buffer Overflow Example #1

Before call to gets

Stack frame for **main**

Return Address

[3] [2] [1] [0]

Input 1234567

Stack frame for **main**

00 00 00 00 00 40 05 73

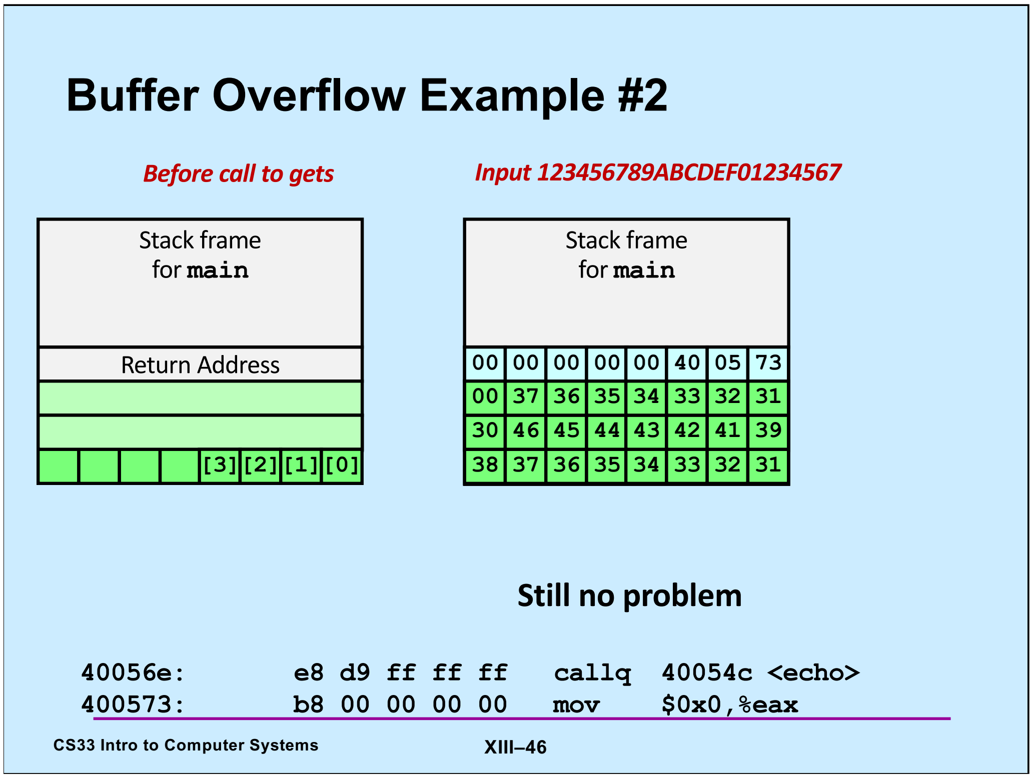
00 37 36 35 34 33 32 31

Overflow buf, but no problem

```
40056e:        e8 d9 ff ff ff    callq   40054c <echo>
400573:        b8 00 00 00 00    mov     $0x0,%eax
```

Supplied by CMU, but adapted for x86-64.

The ASCII-encoded input is shown in the green portion of the stack frame. Note that **gets** reads input until the first newline character, but then replaces it with the null character (0x0).



Buffer Overflow Example #2

Before call to gets

Stack frame for **main**							
Return Address							
				[3]	[2]	[1]	[0]

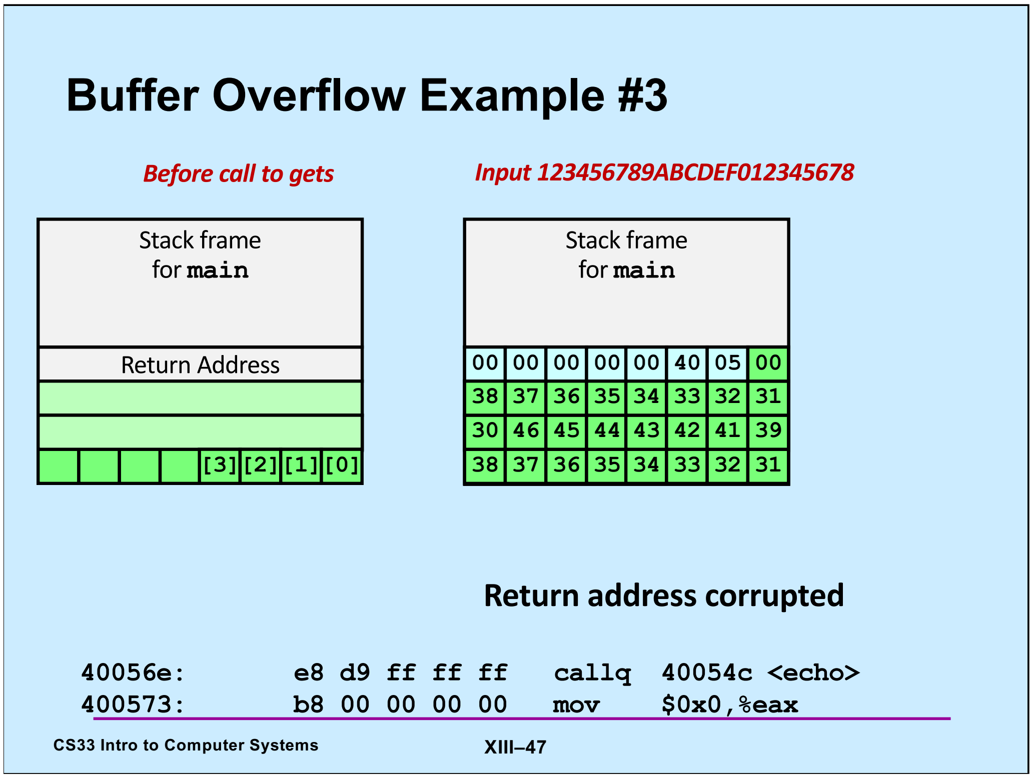
Input 123456789ABCDEF01234567

Stack frame for **main**							
00	00	00	00	00	40	05	73
00	37	36	35	34	33	32	31
30	46	45	44	43	42	41	39
38	37	36	35	34	33	32	31

Still no problem

```
40056e:        e8 d9 ff ff ff    callq  40054c <echo>
400573:        b8 00 00 00 00    mov    $0x0,%eax
```

Supplied by CMU, but adapted for x86-64.



Buffer Overflow Example #3

Before call to gets

Stack frame
for **main**

Return Address

[3] [2] [1] [0]

Input 123456789ABCDEF012345678

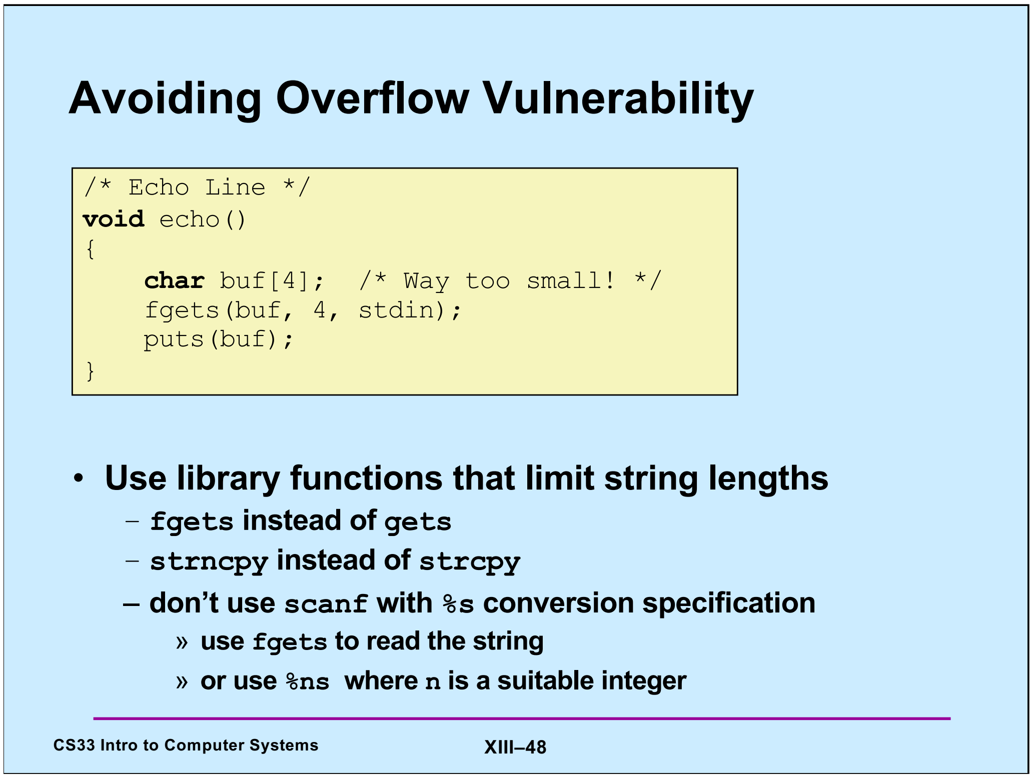
Stack frame
for **main**

00	00	00	00	00	40	05	00
38	37	36	35	34	33	32	31
30	46	45	44	43	42	41	39
38	37	36	35	34	33	32	31

Return address corrupted

```
40056e:       e8 d9 ff ff ff    callq  40054c <echo>
400573:       b8 00 00 00 00    mov    $0x0,%eax
```

Supplied by CMU, but adapted for x86-64.



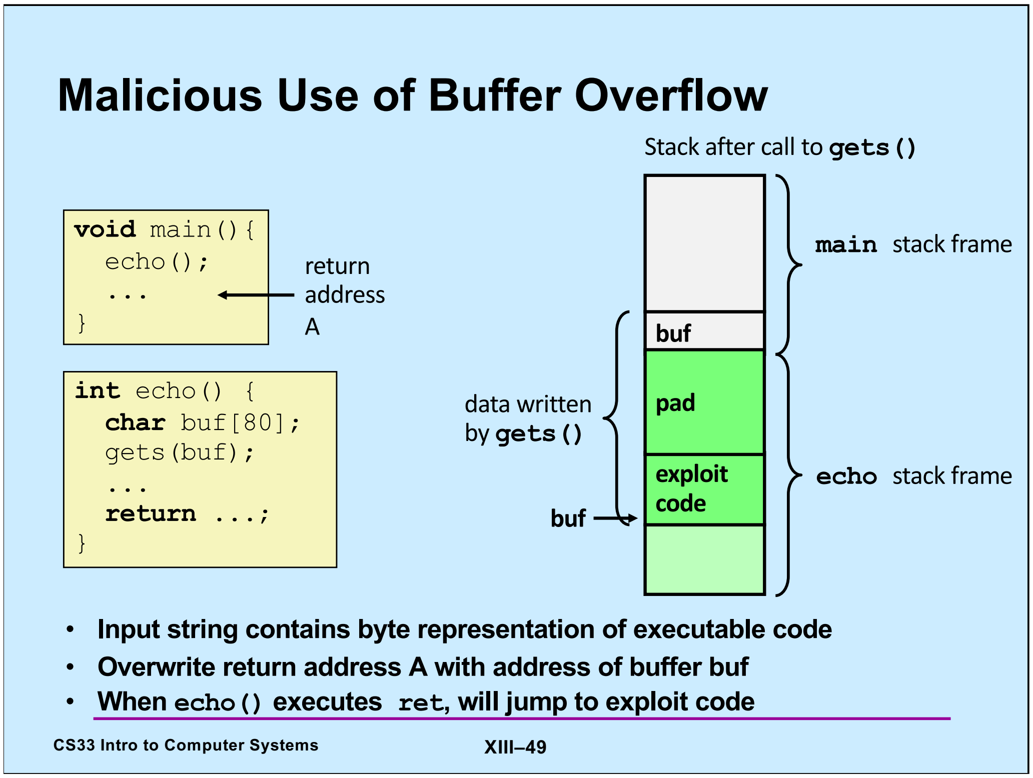
Avoiding Overflow Vulnerability

```
/* Echo Line */
void echo()
{
    char buf[4];  /* Way too small! */
    fgets(buf, 4, stdin);
    puts(buf);
}
```

- Use library functions that limit string lengths
  - `fgets` instead of `gets`
  - `strncpy` instead of `strcpy`
  - don’t use `scanf` with `%s` conversion specification
    - » use `fgets` to read the string
    - » or use `%ns` where `n` is a suitable integer

Supplied by CMU.

The man page for **gets** says (under Bugs): "Never use gets." One might wonder why it still exists – it's probably because too many programs would break if it were removed (but these programs probably should be allowed to break).



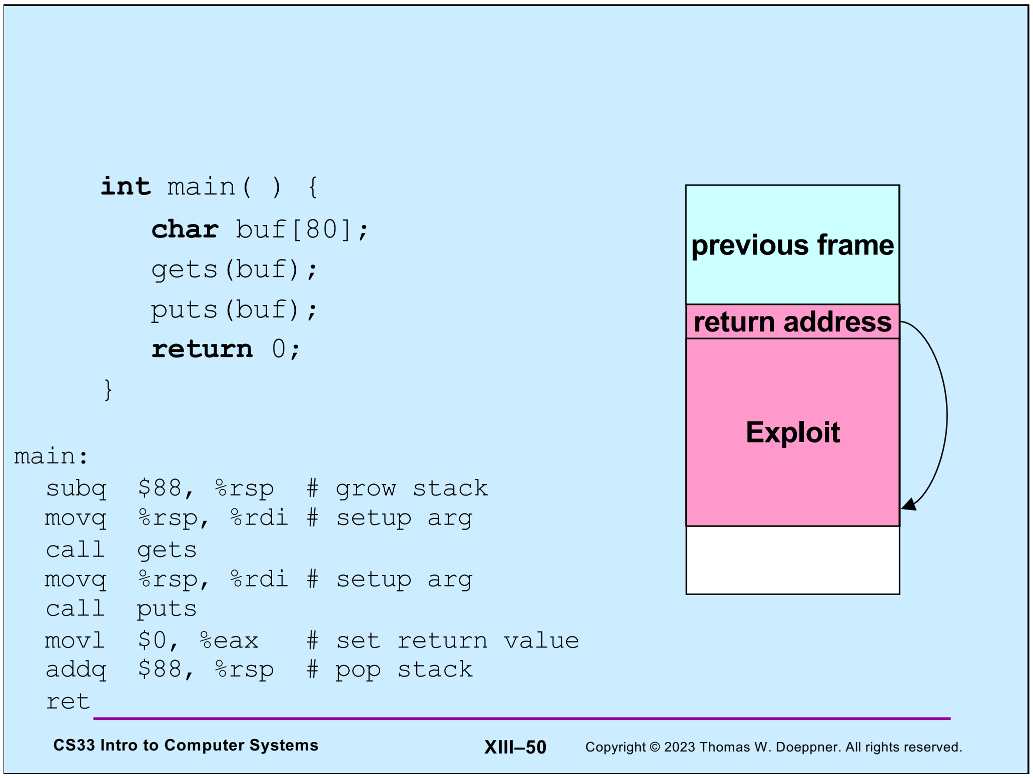
Malicious Use of Buffer Overflow

```
void main(){
   echo();
   ...
}
```

```
int echo() {
   char buf[80];
   gets(buf);
   ...
   return ...;
}
```

- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer buf
- When echo() executes ret, will jump to exploit code

Supplied by CMU, but adapted for x86-64.



```
int main( ) {
   char buf[80];
   gets(buf);
   puts(buf);
   return 0;
}
```

```
main:
  subq  $88, %rsp  # grow stack
  movq  %rsp, %rdi # setup arg
  call  gets
  movq  %rsp, %rdi # setup arg
  call  puts
  movl  $0, %eax   # set return value
  addq  $88, %rsp  # pop stack
  ret
```

previous frame

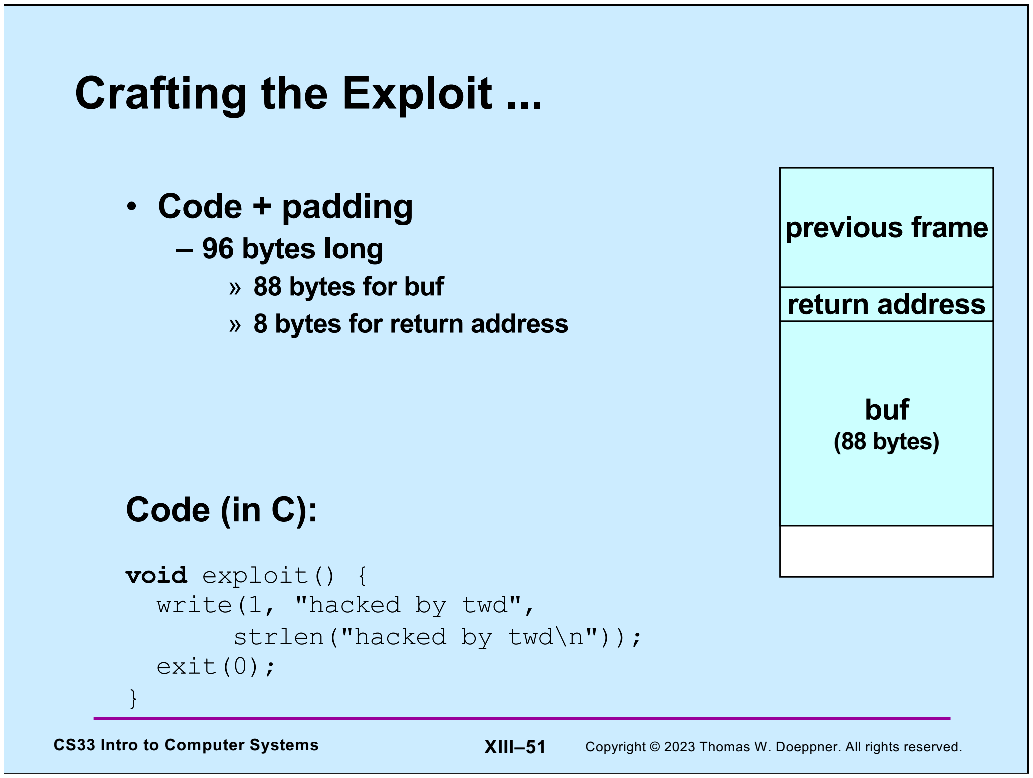
return address

Exploit

Programs susceptible to buffer-overflow attacks are amazingly common and thus such attacks are probably the most numerous of the bug-exploitation techniques. Even drivers for network interface devices might have such problems, making machines vulnerable to attacks by maliciously created packets.

Here we have a too-simple implementation of an echo program, for which we will design and implement an exploit. Note that, strangely, gcc has allocated 88 bytes for buf. We'll discuss reasons for this later — it has to do with cache alignment.

Note that in this version of our example, there is no function called "echo" – everything is done starting from **main**.



Crafting the Exploit ...

- Code + padding
  - 96 bytes long
    - » 88 bytes for buf
    - » 8 bytes for return address

previous frame

return address

buf
(88 bytes)

Code (in C):

```
void exploit() {
  write(1, "hacked by twd",
       strlen("hacked by twd\n"));
  exit(0);
}
```

The “write” function is the lowest-level output function (which we discuss in a later lecture). The first argument indicates we are writing to “standard output” (normally the display). The second argument is what we’re writing, and the third argument is the length of what we’re writing.

The “exit” function instructs the OS to terminate the program.

Quiz 3

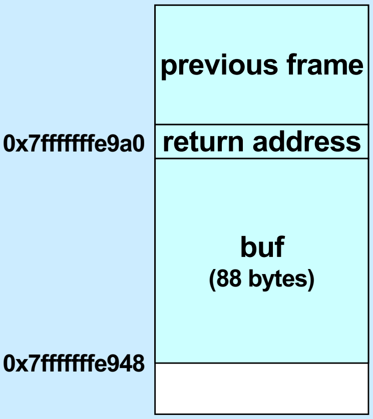
The exploit code will be read into memory starting at location 0x7fffffffe948. What value should be put into the return-address portion of the stack frame?

a) 0

b) 0x7fffffffe9a0

c) 0x7fffffffe948

d) it doesn’t matter what value goes there



previous frame

0x7fffffffe9a0 return address

buf
(88 bytes)

0x7fffffffe948