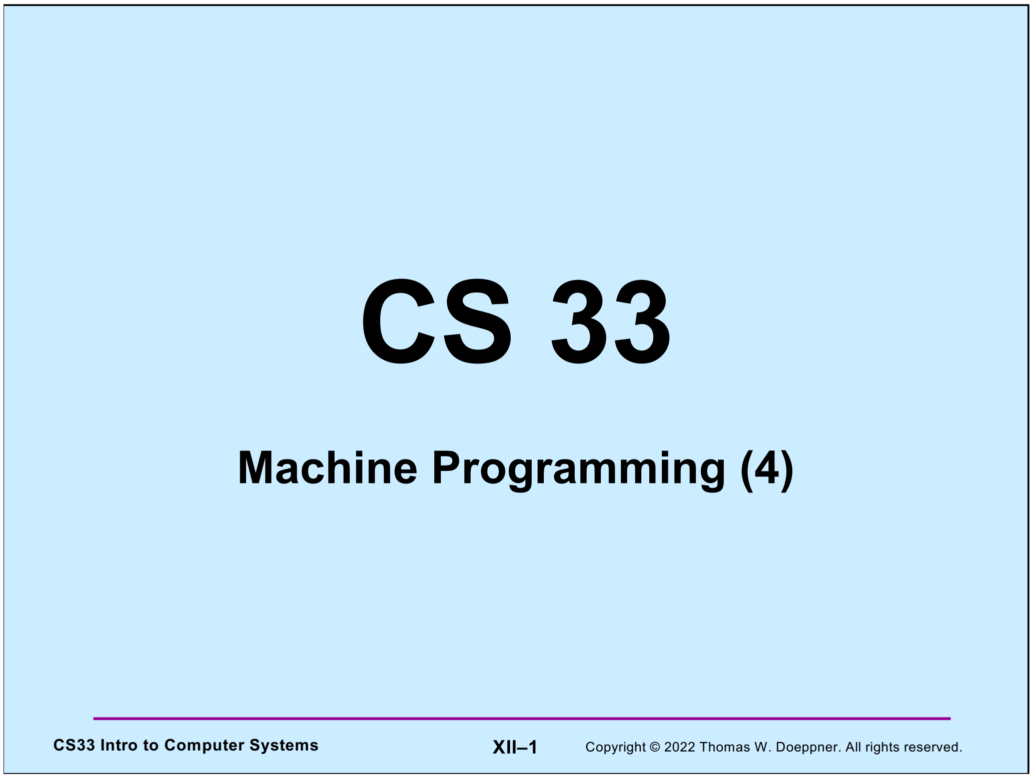
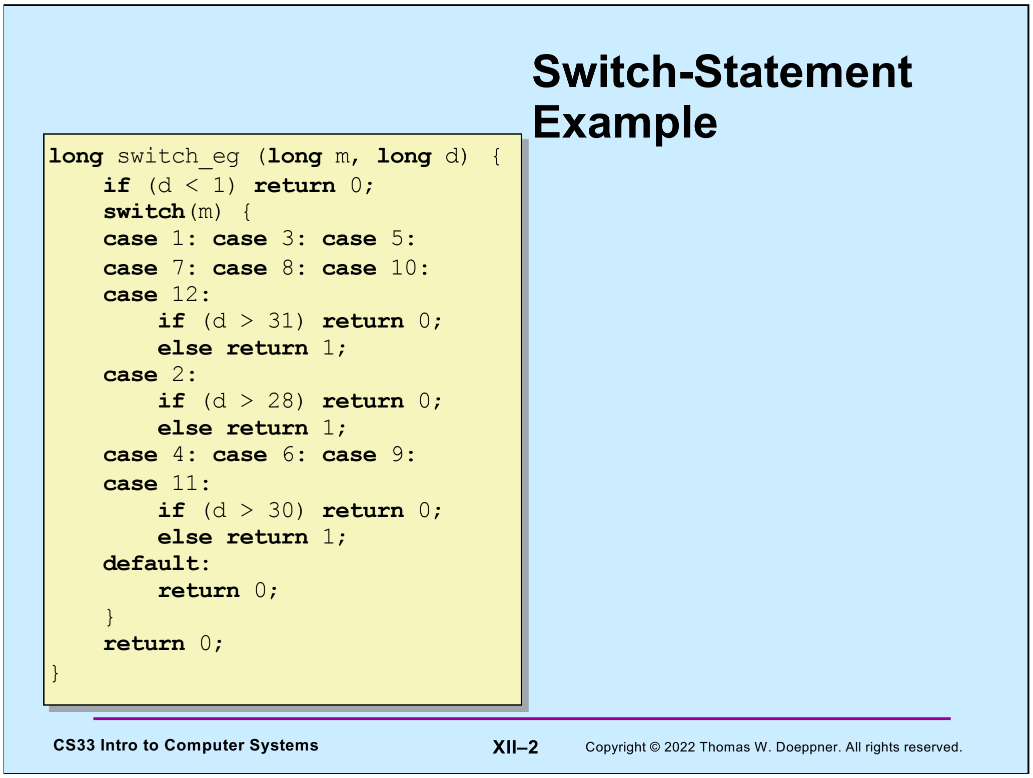


CS 33

Machine Programming (4)

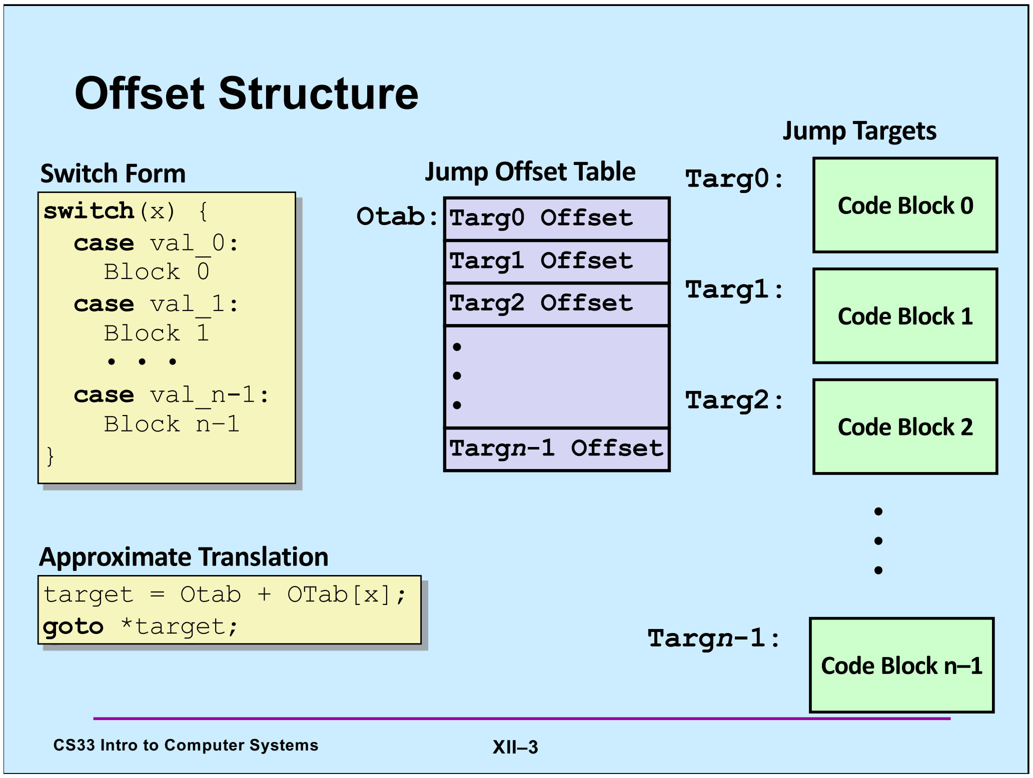
Some of the slides in this lecture are either from or adapted from slides provided by the authors of the textbook “Computer Systems: A Programmer’s Perspective,” 2nd Edition and are provided from the website of Carnegie-Mellon University, course 15-213, taught by Randy Bryant and David O’Hallaron in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



Switch-Statement Example

```
long switch_eg (long m, long d) {
    if (d < 1) return 0;
    switch(m) {
    case 1: case 3: case 5:
    case 7: case 8: case 10:
    case 12:
        if (d > 31) return 0;
        else return 1;
    case 2:
        if (d > 28) return 0;
        else return 1;
    case 4: case 6: case 9:
    case 11:
        if (d > 30) return 0;
        else return 1;
    default:
        return 0;
    }
    return 0;
}
```

Code very much like this appears in level three of the traps project.



Offset Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Offset Table

```
Otab: Targ0 Offset
      Targ1 Offset
      Targ2 Offset
      •
      •
      •
      Targn-1 Offset
```

Jump Targets

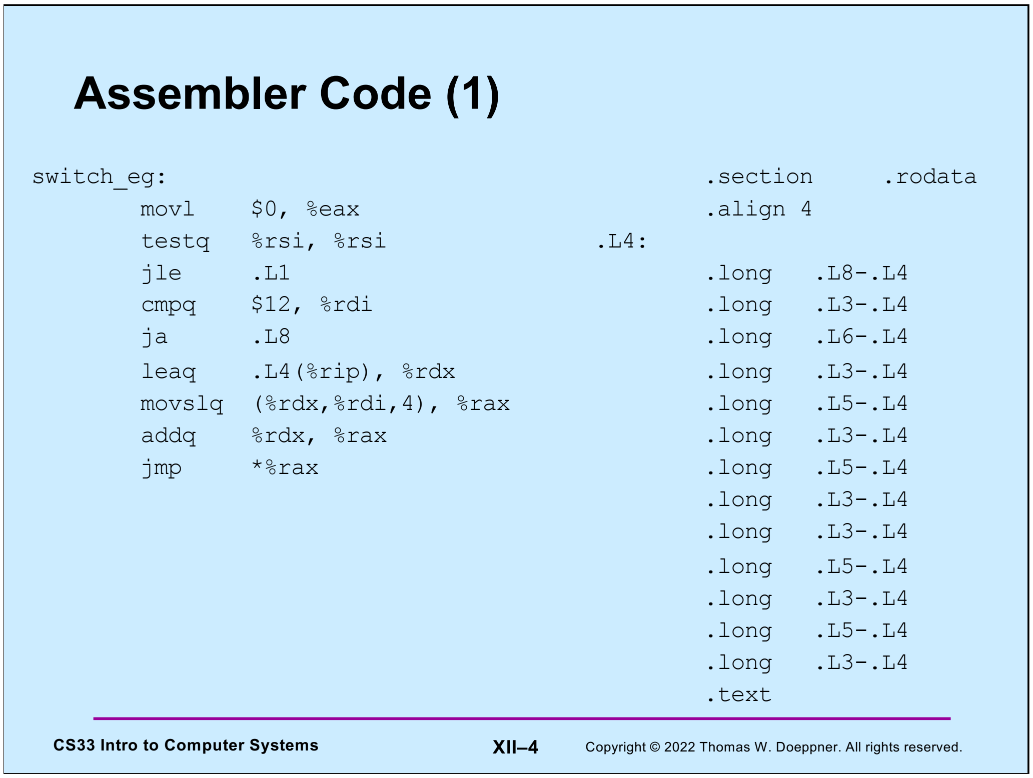
Approximate Translation

```
target = Otab + OTab[x];
goto *target;
```

Adapted from slide supplied by CMU to account for changes in gcc.

The translation is “approximate” because C doesn’t have the notion of the target of a goto being a variable. But, if it did, then the translation is what we’d want!

Otab (for "offset table") is a table of relative address of the jump targets. The idea is, given a value of x, **Otab[x]** contains a reference to the code block that should be handled for that case in the switch statement (this code block is known as the **jump target**). These references are offsets from the address **Otab**. In other words, **Otab** is an address, if we add to it the offset of a particular jump target, we get the absolute address of that jump target.



Assembler Code (1)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

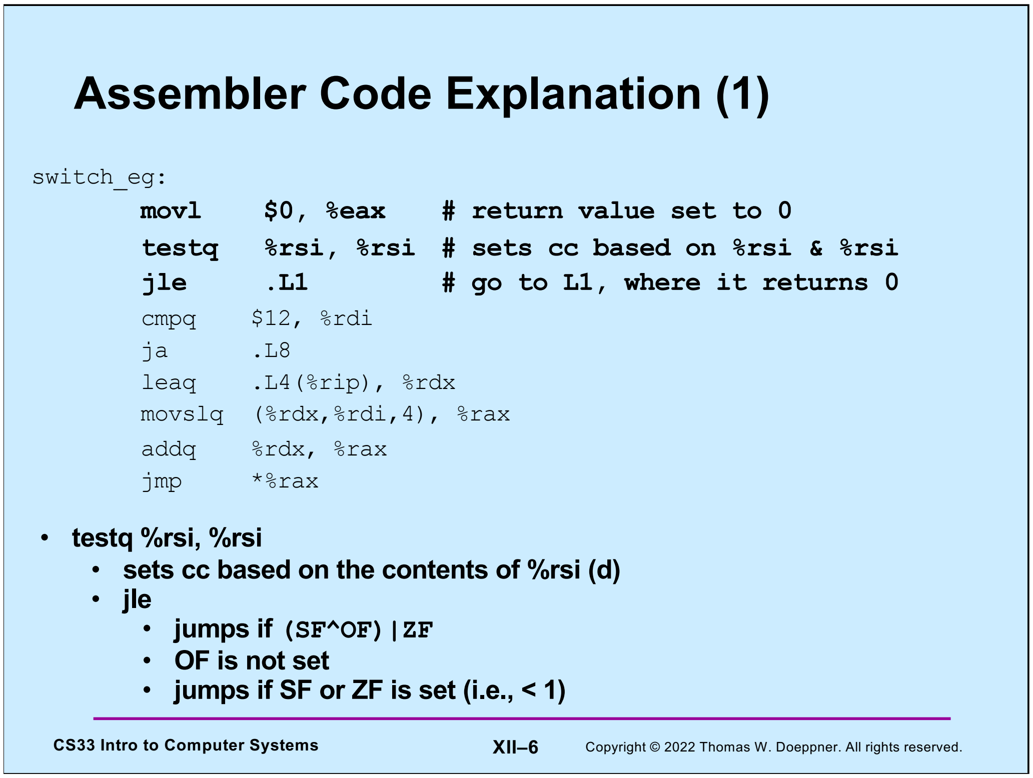
```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4
        .long   .L3-.L4
        .long   .L6-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .long   .L5-.L4
        .long   .L3-.L4
        .text
```

Here's the assembler code obtained by compiling our C code in gcc with the –O1 optimization flag (specifying that some, but not lots of optimization should be done). We explain this code in subsequent slides. The jump offset table starts at label .L4.

Assembler Code (2)

```
.L3:
        cmpq    $31, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L6:
        cmpq    $28, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
```

```
.L5:
        cmpq    $30, %rsi
        setle   %al
        movzbl  %al, %eax
        ret
.L8:
        movl    $0, %eax
.L1:
        ret
```

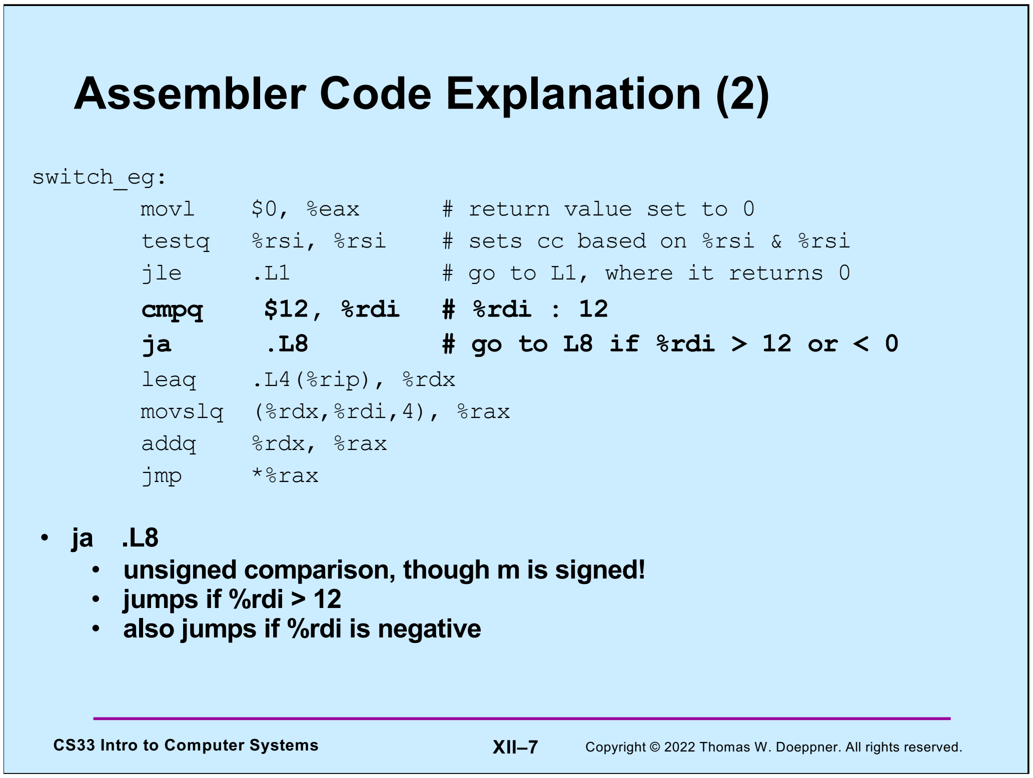


Assembler Code Explanation (1)

```
switch_eg:
        movl    $0, %eax    # return value set to 0
        testq   %rsi, %rsi  # sets cc based on %rsi & %rsi
        jle     .L1         # go to L1, where it returns 0
        cmpq    $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **testq %rsi, %rsi**
  - **sets cc based on the contents of %rsi (d)**
  - **jle**
    - **jumps if `(SF^OF)|ZF`**
    - **OF is not set**
    - **jumps if SF or ZF is set (i.e., < 1)**

The first three instructions cause control to go to .L1 if the second argument (d) is less than 1. At .L1 is code that simply returns (with a return value of 0).



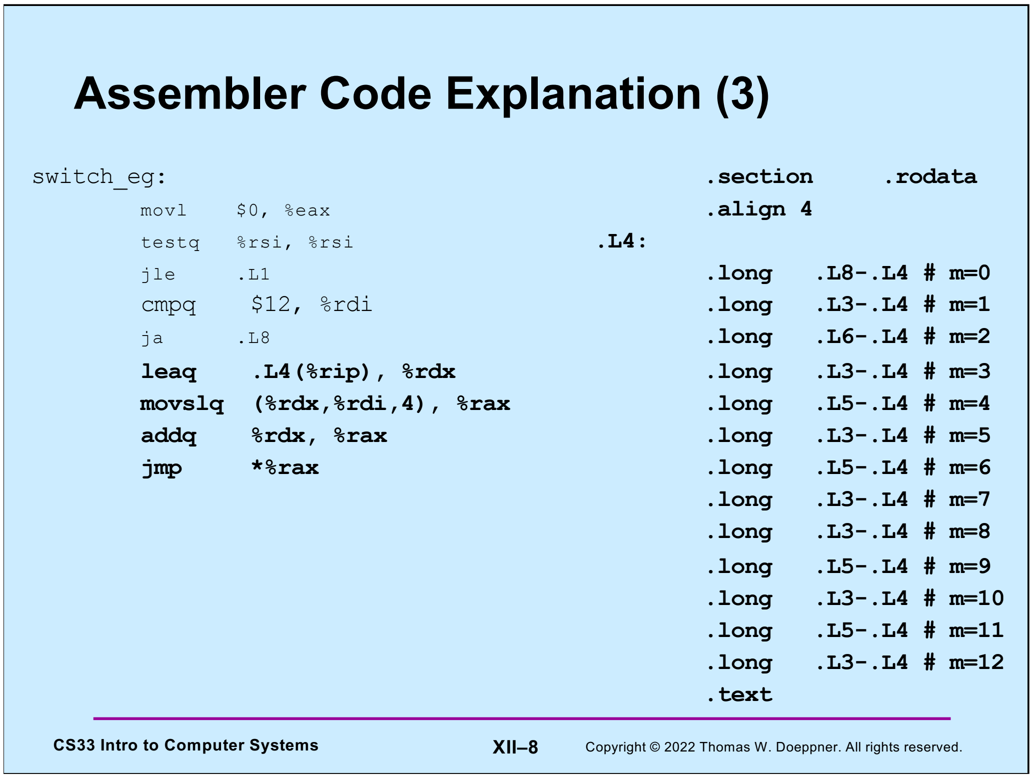
Assembler Code Explanation (2)

```
switch_eg:
        movl    $0, %eax      # return value set to 0
        testq   %rsi, %rsi    # sets cc based on %rsi & %rsi
        jle     .L1           # go to L1, where it returns 0
        cmpq    $12, %rdi     # %rdi : 12
        ja      .L8           # go to L8 if %rdi > 12 or < 0
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

- **ja .L8**
  - **unsigned comparison, though m is signed!**
  - **jumps if %rdi > 12**
  - **also jumps if %rdi is negative**

The next two instructions simply check to make sure that %rdi (the first argument, m) is less than or equal to 12. If not, control goes to .L8, which sets the return value to 0 and returns. Of course, the return value (in %rax/%eax) is already zero, so setting it to zero again is unnecessary.

Note that we're using **ja** (jump if above), which is normally used after comparing unsigned values. The first argument, m, is a (signed) **long**. But if it is interpreted as an unsigned value, then if the leftmost bit (the sign bit) is set, it appears to be a very large unsigned value, and thus the jump is taken.



Assembler Code Explanation (3)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

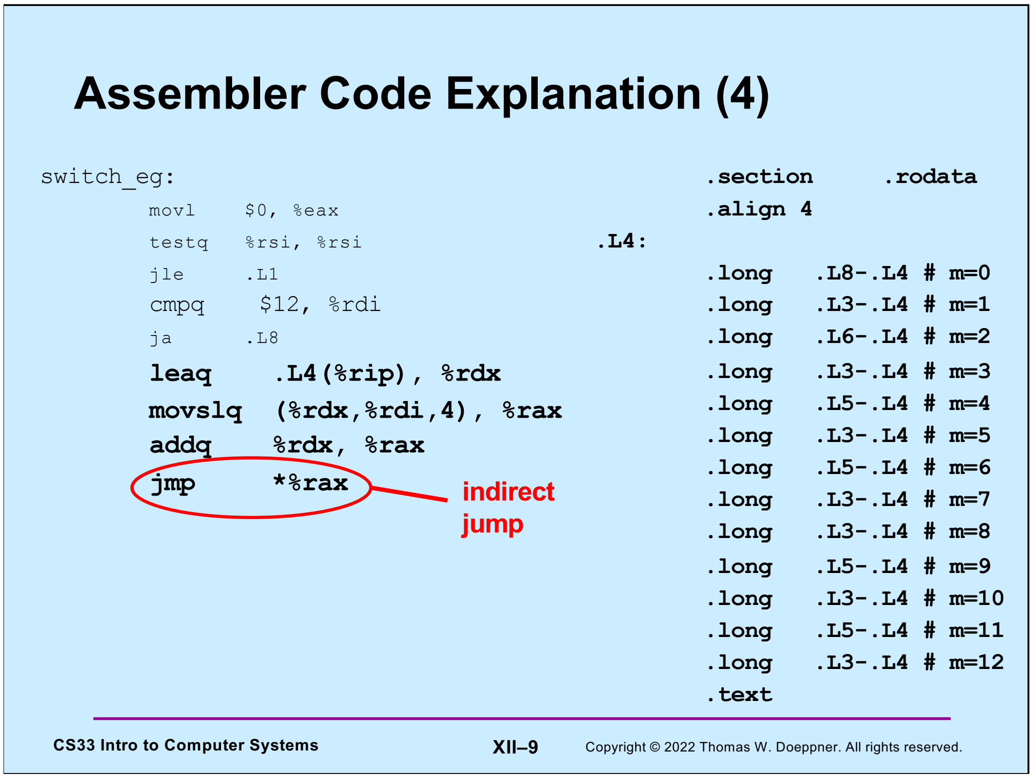
```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

The table on the right is known as an **offset table**. Each line refers to the code to be executed for the corresponding value of m. Each entry in the table is a long (recall that in x86-64 assembler, long means 32 bits). The value of each entry is the difference between the address of the table (.L4) and the address of the code to be executed for a particular value of m (the other .L labels). Thus each entry is the distance (or offset) from the beginning of the table to the code for each case. Note that this offset will be negative, as explained below. It's assumed that the offset fits in a 32-bit signed quantity (which the system guarantees to be true.)

One might ask why we put 32-bit offsets in the table rather than 64-bit addresses. The reason is to reduce the size of these tables – if we used addresses, they'd be twice the size.

This table is not executable (it just contains offsets), but it should be treated as read-only – its contents will never change. The directive ".section .rodata" tells the assembler that we want this table to be located in memory that is read-only, but not executable. The directive at the end of the table (".text") tells the assembler that what follows is (again) executable code. This read-only, non-executable memory is located at a higher address than the executable code is (accept this as a fact for now, we'll see later why it is so). Thus the offsets in the table are negative.

The highlighted code on the left is what interprets the table, We examine it next.



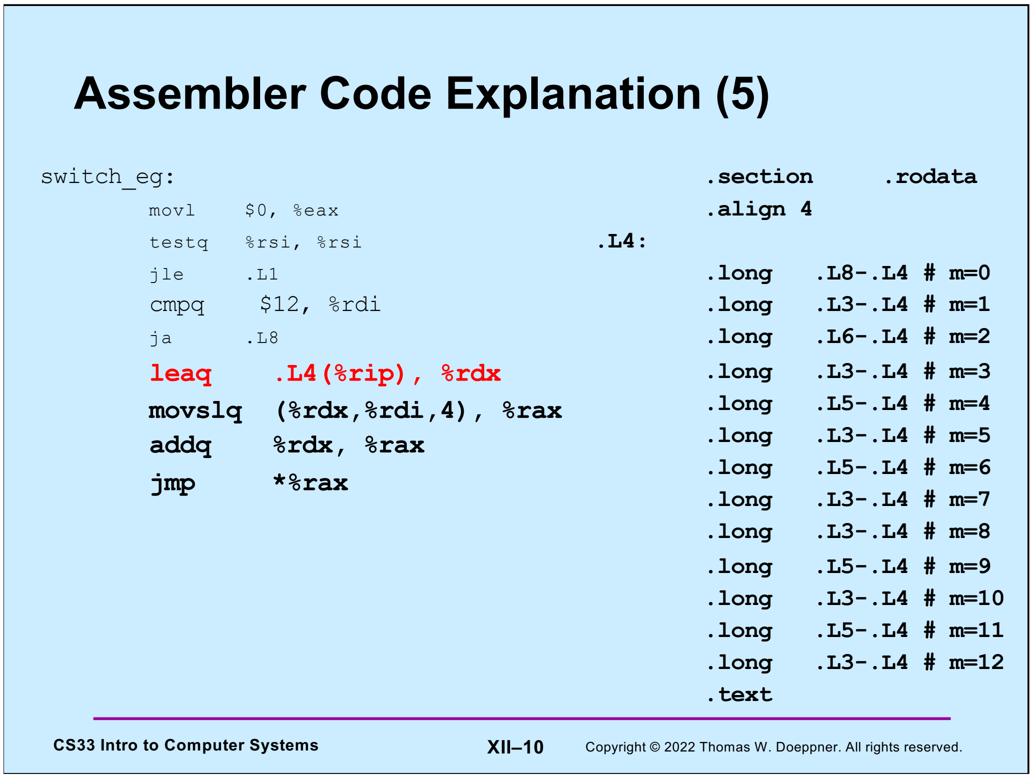
Assembler Code Explanation (4)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

indirect jump

```
         .section     .rodata
         .align 4
.L4:
         .long   .L8-.L4 # m=0
         .long   .L3-.L4 # m=1
         .long   .L6-.L4 # m=2
         .long   .L3-.L4 # m=3
         .long   .L5-.L4 # m=4
         .long   .L3-.L4 # m=5
         .long   .L5-.L4 # m=6
         .long   .L3-.L4 # m=7
         .long   .L3-.L4 # m=8
         .long   .L5-.L4 # m=9
         .long   .L3-.L4 # m=10
         .long   .L5-.L4 # m=11
         .long   .L3-.L4 # m=12
         .text
```

The highlighted code makes use of an indirect jump instruction, indicated by having an asterisk before its register operand. The register contains an address, and the jump is made to the code at that address. Note that jump instructions that are not indirect have constants as their operands. We'll see later on that, because of this, indirect jumps are often much slower than non-indirect jumps.



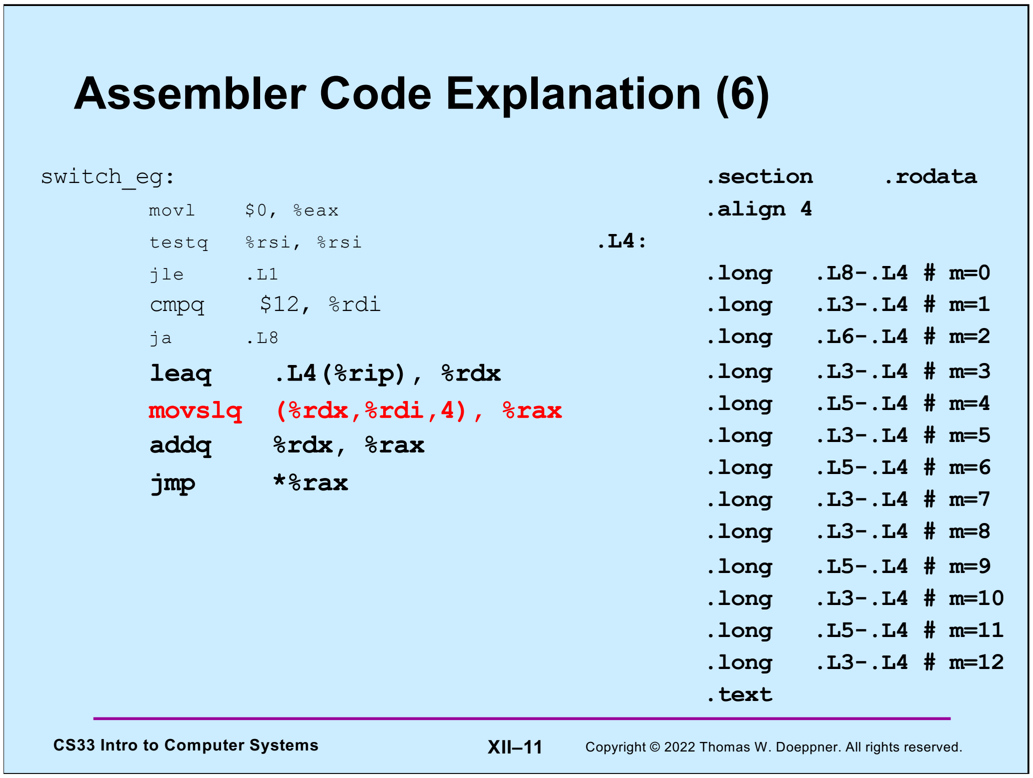
Assembler Code Explanation (5)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq    .L4(%rip), %rdx
        movslq  (%rdx,%rdi,4), %rax
        addq    %rdx, %rax
        jmp     *%rax
```

```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

The **leaq** instruction (load effective address, quad), performs an address computation, but rather than fetching the data at the address, it stores the address itself in %rdx.

What's unusual about the instruction is that it uses %rip (the instruction pointer) as the base register, and has a displacement that is a label. This is a special case for the assembler, which can compute the offset between the leaq instruction and the label, and use that value for the displacement field. Thus the instruction puts the address of the offset table (.L4) into %rdx.



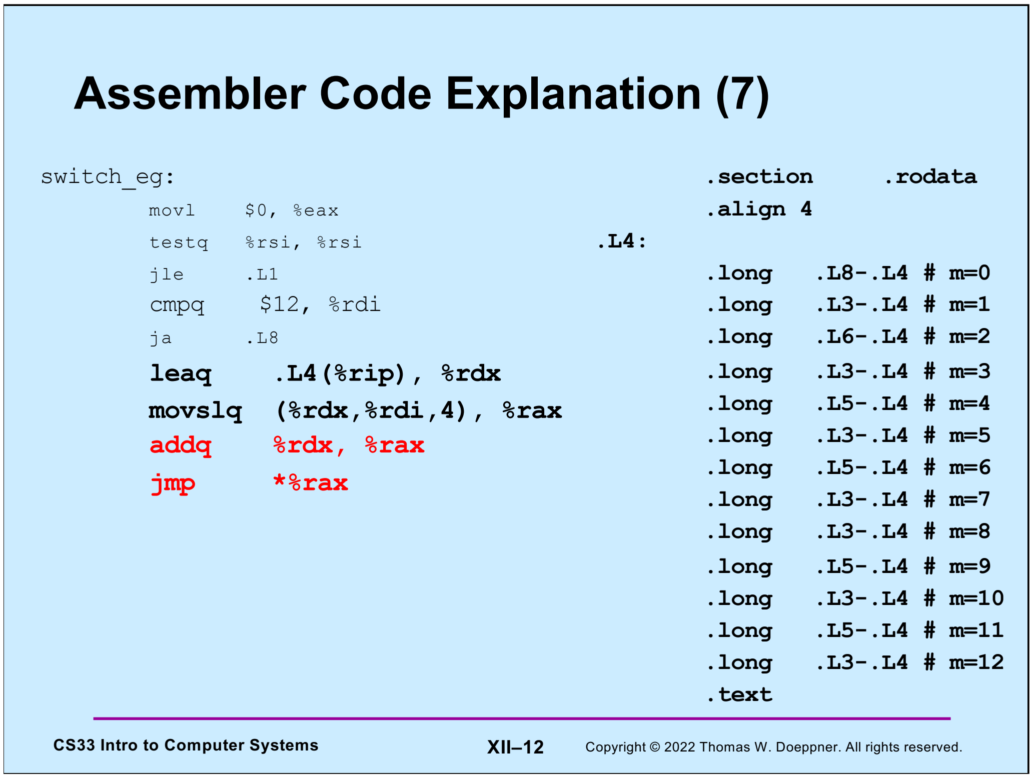
Assembler Code Explanation (6)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq   (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

```
        .section     .rodata
        .align 4
.L4:
        .long   .L8-.L4 # m=0
        .long   .L3-.L4 # m=1
        .long   .L6-.L4 # m=2
        .long   .L3-.L4 # m=3
        .long   .L5-.L4 # m=4
        .long   .L3-.L4 # m=5
        .long   .L5-.L4 # m=6
        .long   .L3-.L4 # m=7
        .long   .L3-.L4 # m=8
        .long   .L5-.L4 # m=9
        .long   .L3-.L4 # m=10
        .long   .L5-.L4 # m=11
        .long   .L3-.L4 # m=12
        .text
```

The **movslq** instruction copies a long (32 bits) into a quad (64 bits), and does sign extension so as to preserve the sign of the value being copied.

%rdi contains m, the first argument, which is also the argument of the switch statement. We use it to index into the offset table: As we saw in the previous slide, %rdx contains the address of the table, whose entries are each 4 bytes long. Thus we use %rdi as an index register, with a scale factor of 4. The contents of that entry (which is the distance from the table to the code that should be executed to handle this case) is copied into %rax, using sign extension to fill the register.



Assembler Code Explanation (7)

```
switch_eg:
        movl    $0, %eax
        testq   %rsi, %rsi
        jle     .L1
        cmpq     $12, %rdi
        ja      .L8
        leaq     .L4(%rip), %rdx
        movslq   (%rdx,%rdi,4), %rax
        addq     %rdx, %rax
        jmp      *%rax
```

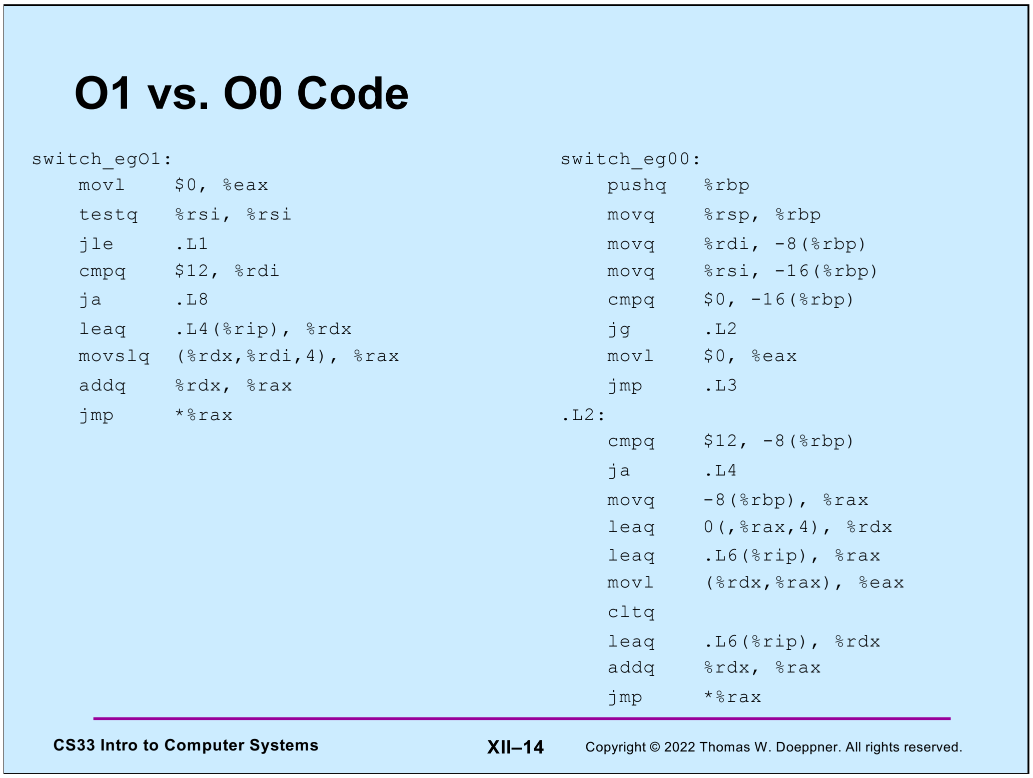
```
        .section      .rodata
        .align 4
.L4:
        .long    .L8-.L4 # m=0
        .long    .L3-.L4 # m=1
        .long    .L6-.L4 # m=2
        .long    .L3-.L4 # m=3
        .long    .L5-.L4 # m=4
        .long    .L3-.L4 # m=5
        .long    .L5-.L4 # m=6
        .long    .L3-.L4 # m=7
        .long    .L3-.L4 # m=8
        .long    .L5-.L4 # m=9
        .long    .L3-.L4 # m=10
        .long    .L5-.L4 # m=11
        .long    .L3-.L4 # m=12
        .text
```

The offset of the code we want to jump to is in %rax. To convert this offset into an absolute address, we need to add to it the address of the table. That’s what the **addq** instruction does.

We can now do the indirect jump, to the address contained in %rax.

Switch Statements and Traps

- The code we just looked at was compiled with gcc’s O1 flag
  - a moderate amount of “optimization”
- Traps originally was compiled with the O0 flag
  - no optimization
- O0 often produces easier-to-read (but less efficient) code
  - not so for switch



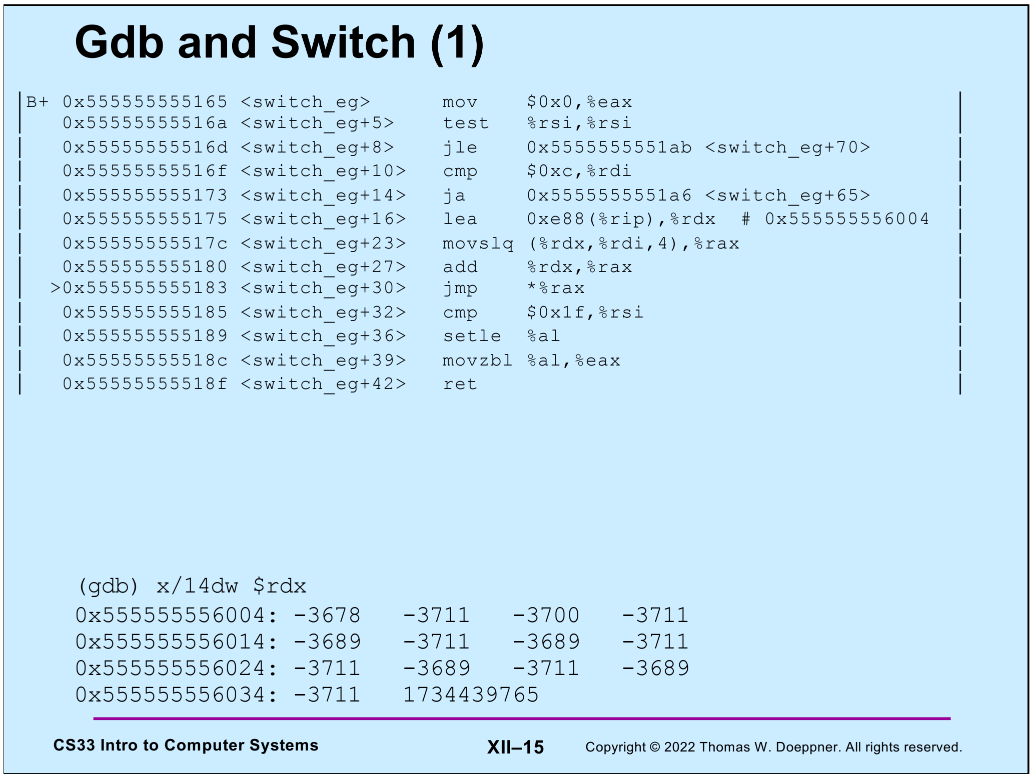
O1 vs. O0 Code

```
switch_egO1:
    movl    $0, %eax
    testq   %rsi, %rsi
    jle     .L1
    cmpq    $12, %rdi
    ja      .L8
    leaq    .L4(%rip), %rdx
    movslq  (%rdx,%rdi,4), %rax
    addq    %rdx, %rax
    jmp     *%rax
```

```
switch_eg00:
    pushq   %rbp
    movq    %rsp, %rbp
    movq    %rdi, -8(%rbp)
    movq    %rsi, -16(%rbp)
    cmpq    $0, -16(%rbp)
    jg      .L2
    movl    $0, %eax
    jmp     .L3
.L2:
    cmpq    $12, -8(%rbp)
    ja      .L4
    movq    -8(%rbp), %rax
    leaq    0(,%rax,4), %rdx
    leaq    .L6(%rip), %rax
    movl    (%rdx,%rax), %eax
    cltq
    leaq    .L6(%rip), %rdx
    addq    %rdx, %rax
    jmp     *%rax
```

On the left we have the O1 version of the code, on the right we have the O0.

This is why we released a new version of traps that was compiled with –O1 (there were no other changes).



Gdb and Switch (1)

```
|B+ 0x555555555165 <switch_eg>      mov    $0x0,%eax                               |
|   0x55555555516a <switch_eg+5>    test   %rsi,%rsi                               |
|   0x55555555516d <switch_eg+8>    jle    0x5555555551ab <switch_eg+70>           |
|   0x55555555516f <switch_eg+10>   cmp    $0xc,%rdi                               |
|   0x555555555173 <switch_eg+14>   ja     0x5555555551a6 <switch_eg+65>           |
|   0x555555555175 <switch_eg+16>   lea    0xe88(%rip),%rdx  # 0x555555556004      |
|   0x55555555517c <switch_eg+23>   movslq (%rdx,%rdi,4),%rax                      |
|   0x555555555180 <switch_eg+27>   add    %rdx,%rax                               |
|  >0x555555555183 <switch_eg+30>   jmp    *%rax                                   |
|   0x555555555185 <switch_eg+32>   cmp    $0x1f,%rsi                              |
|   0x555555555189 <switch_eg+36>   setle  %al                                     |
|   0x55555555518c <switch_eg+39>   movzbl %al,%eax                                |
|   0x55555555518f <switch_eg+42>   ret                                            |
```

```
(gdb) x/14dw $rdx
0x555555556004: -3678   -3711   -3700   -3711
0x555555556014: -3689   -3711   -3689   -3711
0x555555556024: -3711   -3689   -3711   -3689
0x555555556034: -3711   1734439765
```

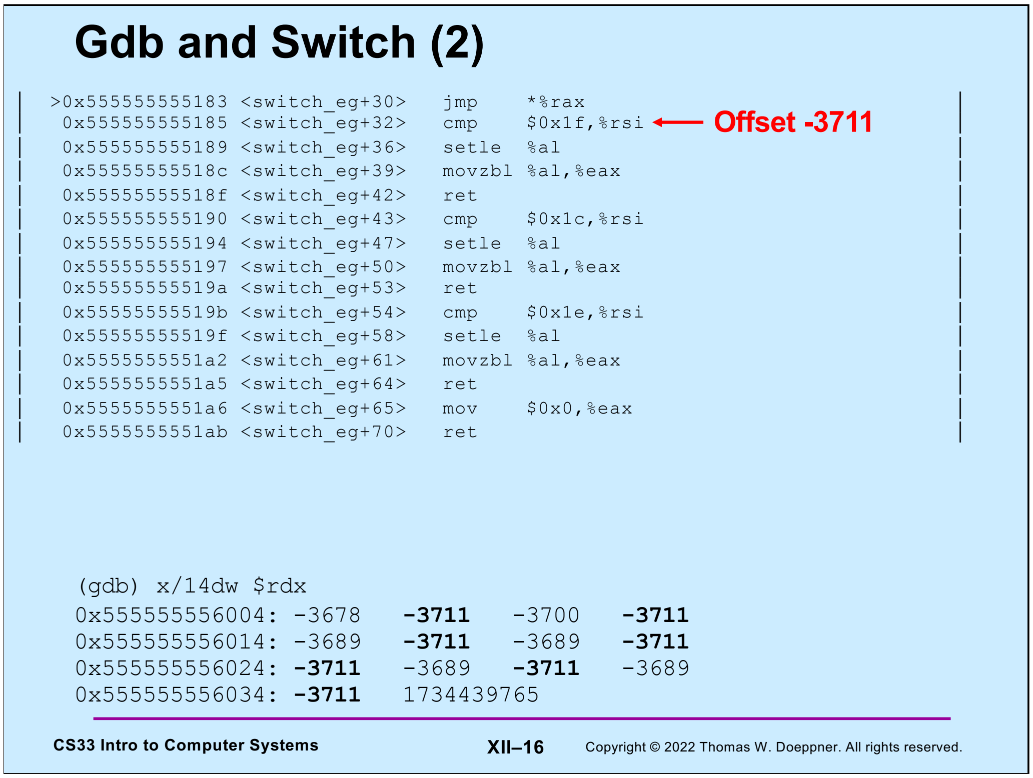
So, now that we know how switch statements are implemented, how might we "reverse engineer" object code to figure out the switch statement it implements?

Here we're running gdb on a program that contains a call to **switch_eg**. We gave the command "layout asm" so that we can see the assembly listing at the top of the slide. We set a breakpoint at **switch_eg**.

Assuming no knowledge of the original source code, we look at the code for **switch_eg** and see an indirect jump instruction at switch_eg+30, which is a definite indication that the C code contained a switch statement. We can see that %rdx contains the address of the offset table, and that %rax will be set to the entry in the table at the index given in %rdi. The contents of %rdx are added to %rax, thus causing %rax to point to the instruction the indirect jump will go to.

Note also that for **leaq** instructions in which the base register is %rip, gdb indicates (as a comment) what the computed address is (`0x555555556004 in this case, which is the address of the offset table).`

So, with all this in mind, after the breakpoint was reached, we issued the **stepi** (si) command 8 times so that we could see the values of all registers just before the indirect jmp. We then used the **x/14dw** gdb command to print 14 entries of a jump offset table starting at the address contained in %rdx. We had to guess how many entries there are – 14 seems reasonable in that it seems unlikely that a switch statement has more than 14 cases, though it might. We know that the table comes after the executable code, so the entries are negative. We see seven entries with values reasonably close to one another, while the remaining entry is very different, so we conclude that the jump table contains13 entries.



Gdb and Switch (2)

```
>0x555555555183 <switch_eg+30>   jmp    *%rax
 0x555555555185 <switch_eg+32>   cmp    $0x1f,%rsi
 0x555555555189 <switch_eg+36>   setle  %al
 0x55555555518c <switch_eg+39>   movzbl %al,%eax
 0x55555555518f <switch_eg+42>   ret
 0x555555555190 <switch_eg+43>   cmp    $0x1c,%rsi
 0x555555555194 <switch_eg+47>   setle  %al
 0x555555555197 <switch_eg+50>   movzbl %al,%eax
 0x55555555519a <switch_eg+53>   ret
 0x55555555519b <switch_eg+54>   cmp    $0x1e,%rsi
 0x55555555519f <switch_eg+58>   setle  %al
 0x5555555551a2 <switch_eg+61>   movzbl %al,%eax
 0x5555555551a5 <switch_eg+64>   ret
 0x5555555551a6 <switch_eg+65>   mov    $0x0,%eax
 0x5555555551ab <switch_eg+70>   ret
```

Offset -3711

```
(gdb) x/14dw $rdx
0x555555556004: -3678   -3711   -3700   -3711
0x555555556014: -3689   -3711   -3689   -3711
0x555555556024: -3711   -3689   -3711   -3689
0x555555556034: -3711   1734439765
```

The code for some case of the switch should come immediately after the **jmp** (what else would go there?!). So the smallest (most negative) offset in the jump offset table must be the offset for this first code segment. Thus offset -3711 corresponds to switch_eg+32 in the assembly listing. It's at indices 1, 3, 5, 7, 8, 10, and 12 of the table, so it's this code that's executed when the first argument of switch_eg is 1, 3, 5, 7, 8, 10, or 12.

Knowing this, we can figure out the rest. The slide contains all the code of switch_eg from the indirect jump to the end of the function (and thus the code for all the cases of the switch statement).

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
        jl     .L2
        ret
```

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

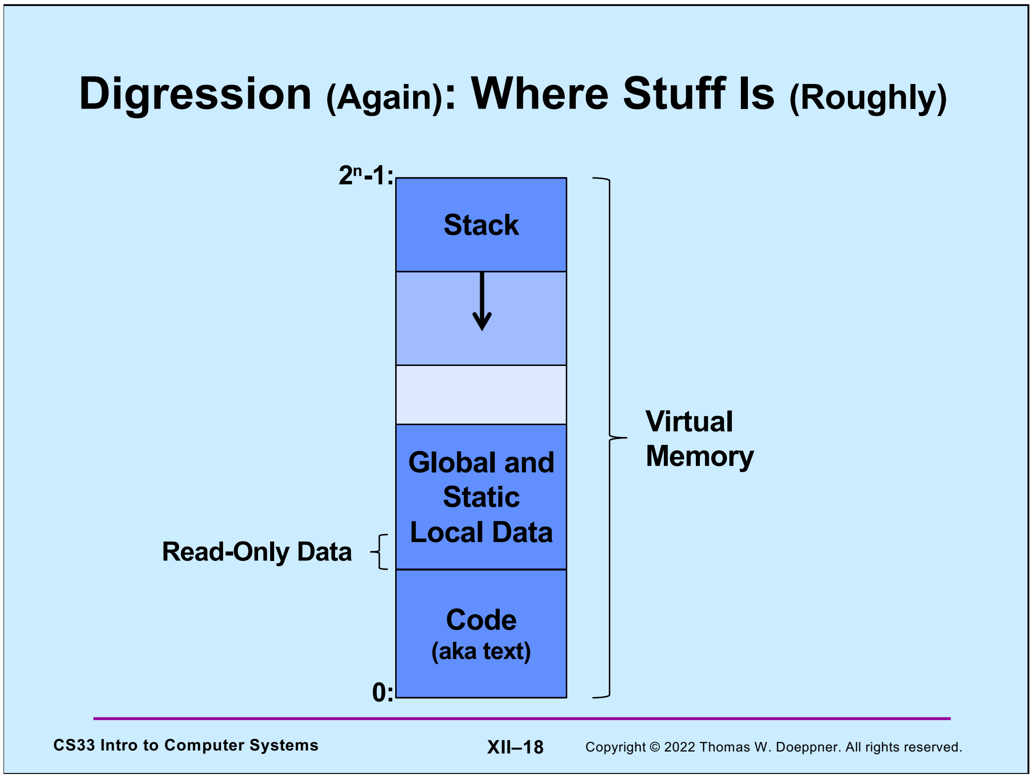
a

```
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



Digression (Again): Where Stuff Is (Roughly)

2^n-1:

Stack

Global and Static Local Data

Read-Only Data

Code (aka text)

0:

Virtual Memory

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.

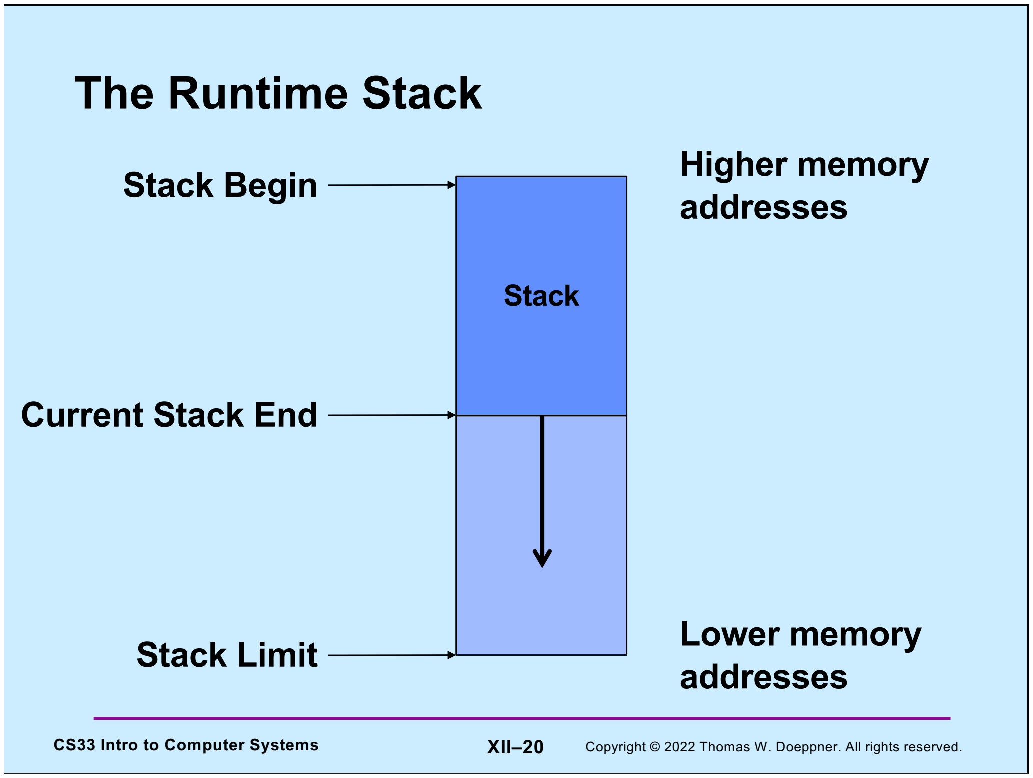
Note that read-only data (such as the offset tables used for switch statements) is placed just above the executable code.

Function Call and Return

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

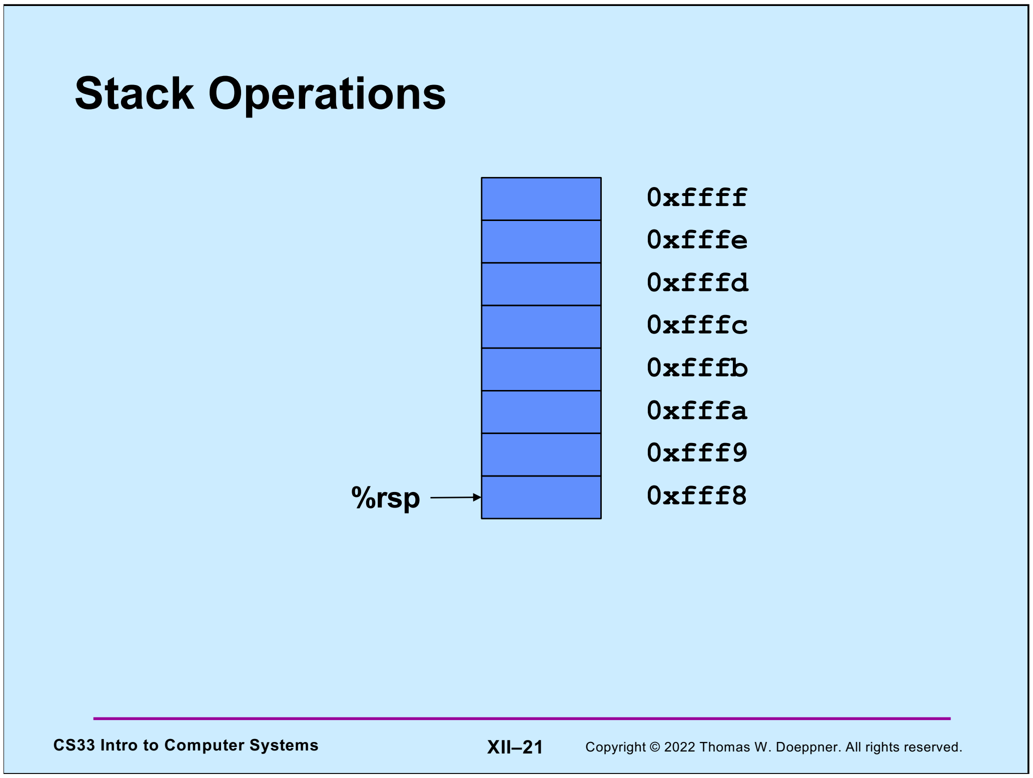
... several million instructions later

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



The Runtime Stack

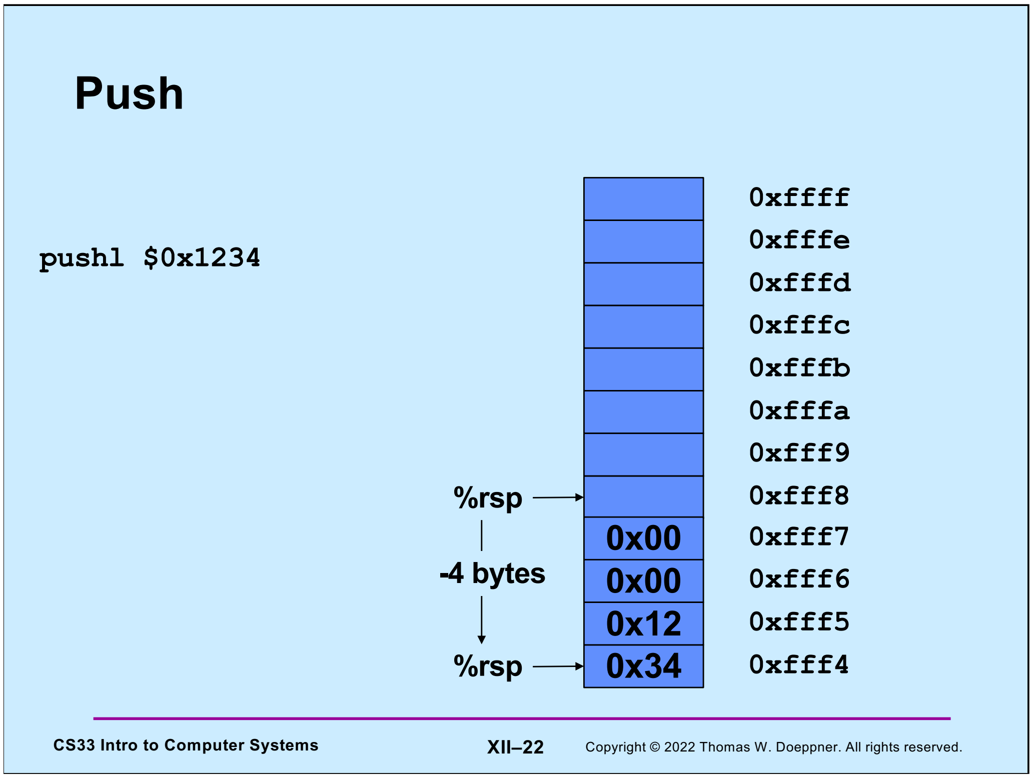
Stacks, as implemented on the X86 for most operating systems (and, in particular, Linux, OSX, and Windows) grow "downwards", from high memory addresses to low memory addresses. To avoid confusion, we will not use the works "top of stack" or "bottom of stack" but will instead use "stack begin" and "current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



Stack Operations

0xffff
0xfffe
0xfffd
0xfffc
0xfffb
0xfffa
0xfff9
%rsp → 0xfff8

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.

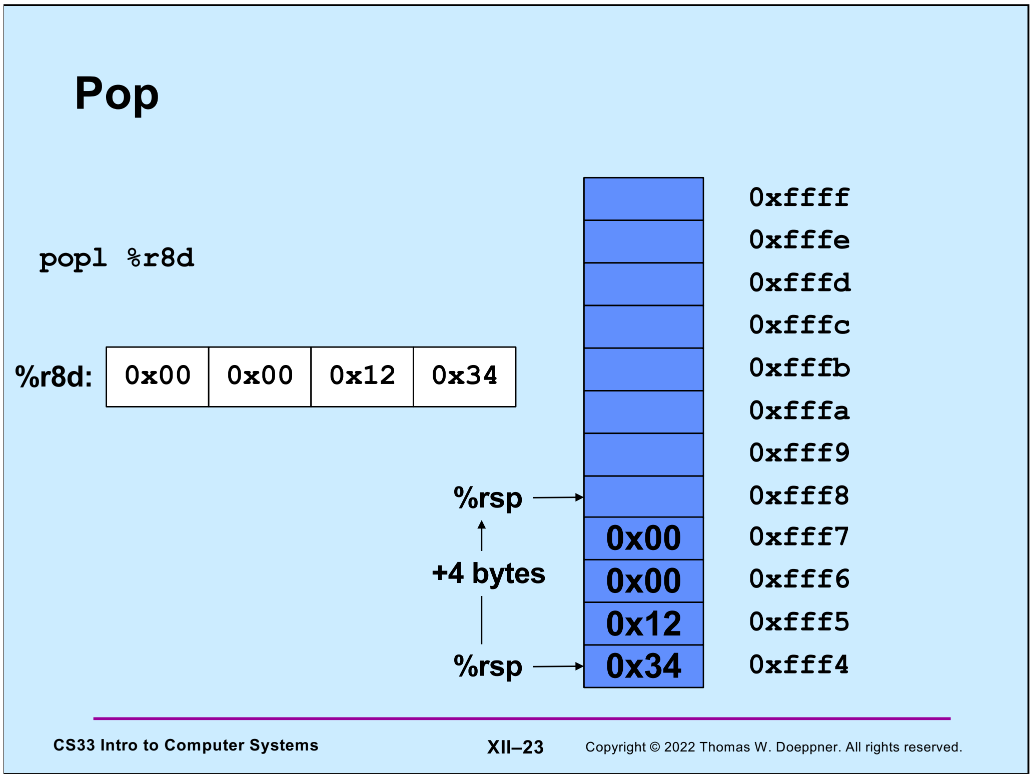


Push

```
pushl $0x1234
```

		0xffff
		0xfffe
		0xfffd
		0xfffc
		0xfffb
		0xfffa
		0xfff9
%rsp		0xfff8
	0x00	0xfff7
-4 bytes	0x00	0xfff6
	0x12	0xfff5
%rsp	0x34	0xfff4

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.



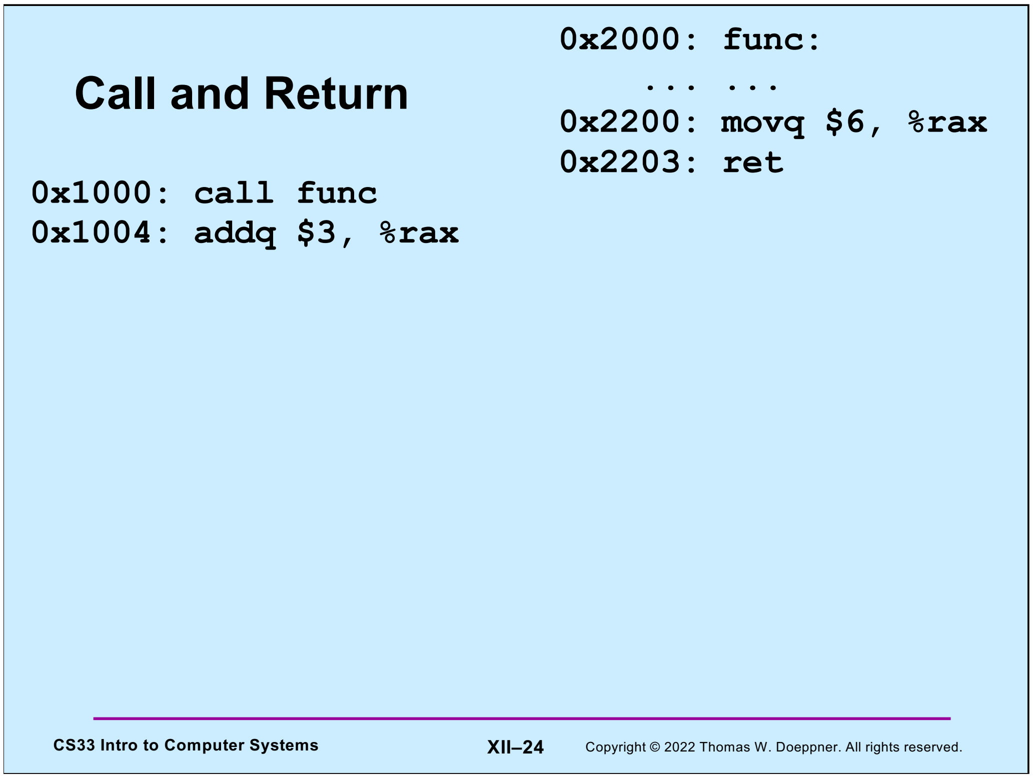
Pop

```
popl %r8d
```

%r8d:	0x00	0x00	0x12	0x34

	0xffff
	0xfffe
	0xfffd
	0xfffc
	0xfffb
	0xfffa
	0xfff9
	0xfff8
0x00	0xfff7
0x00	0xfff6
0x12	0xfff5
0x34	0xfff4

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.

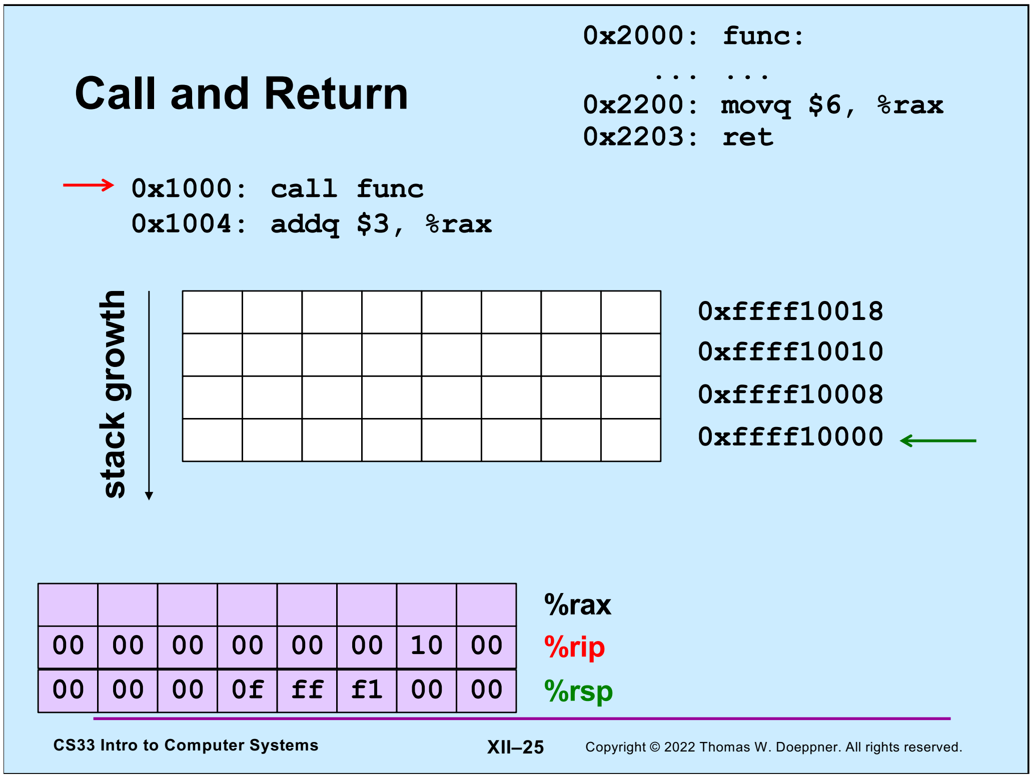


Call and Return

```
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.



Call and Return

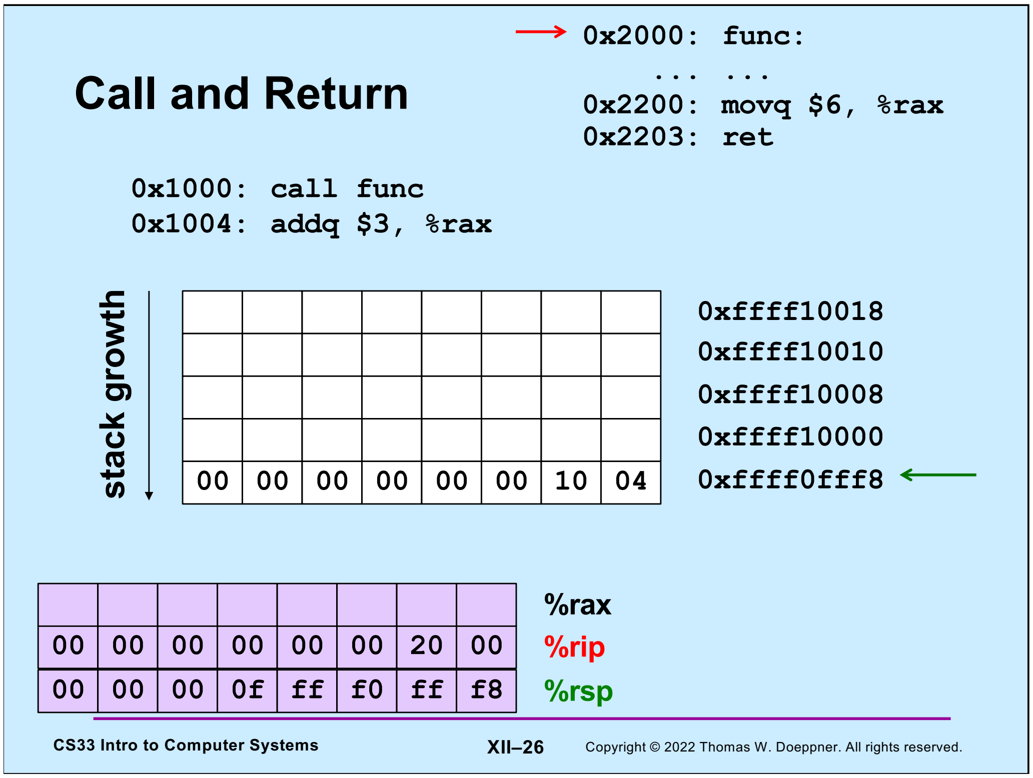
```
0x2000: func:
   ...  ...
0x2200: movq $6, %rax
0x2203: ret
```

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

								%rax
00	00	00	00	00	00	10	00	%rip
00	00	00	0f	ff	f1	00	00	%rsp

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!).



Call and Return

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

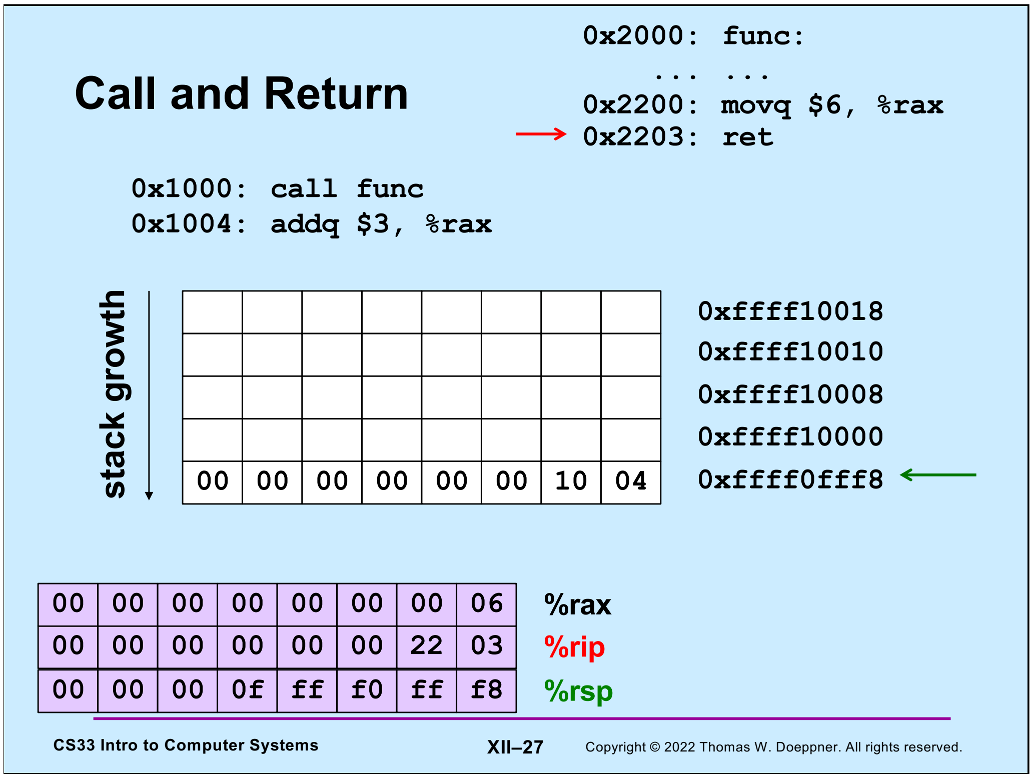
```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8 ←

								%rax
00	00	00	00	00	00	20	00	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

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**.



Call and Return

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

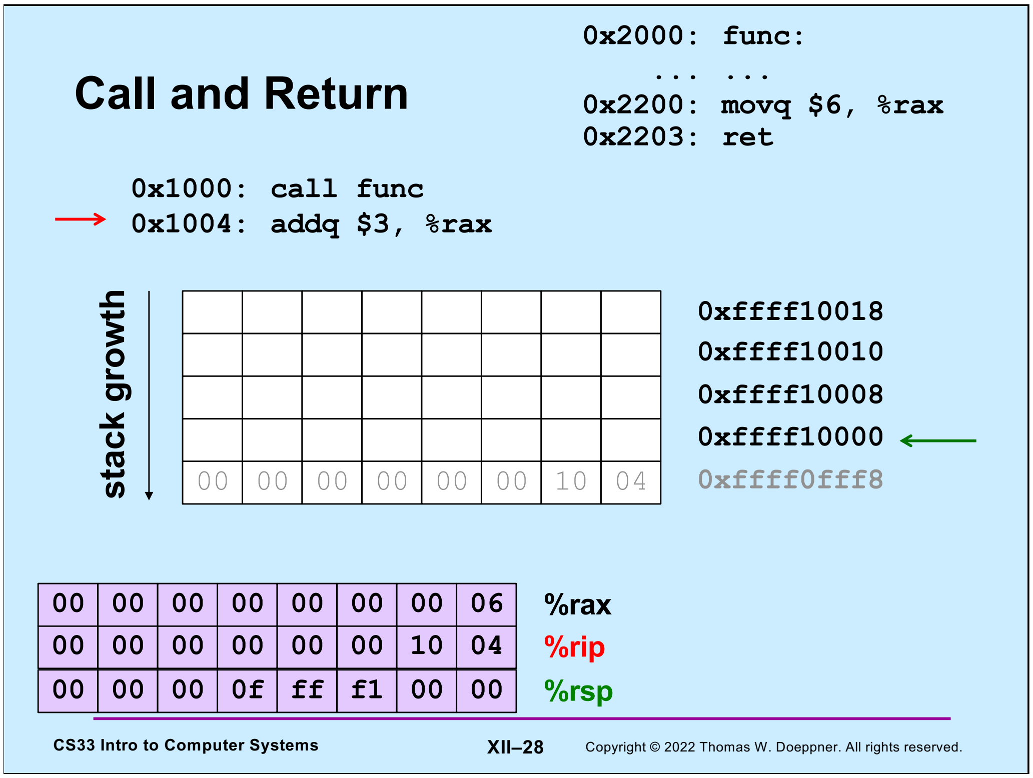
```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8 ←

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	22	03	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

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.



Call and Return

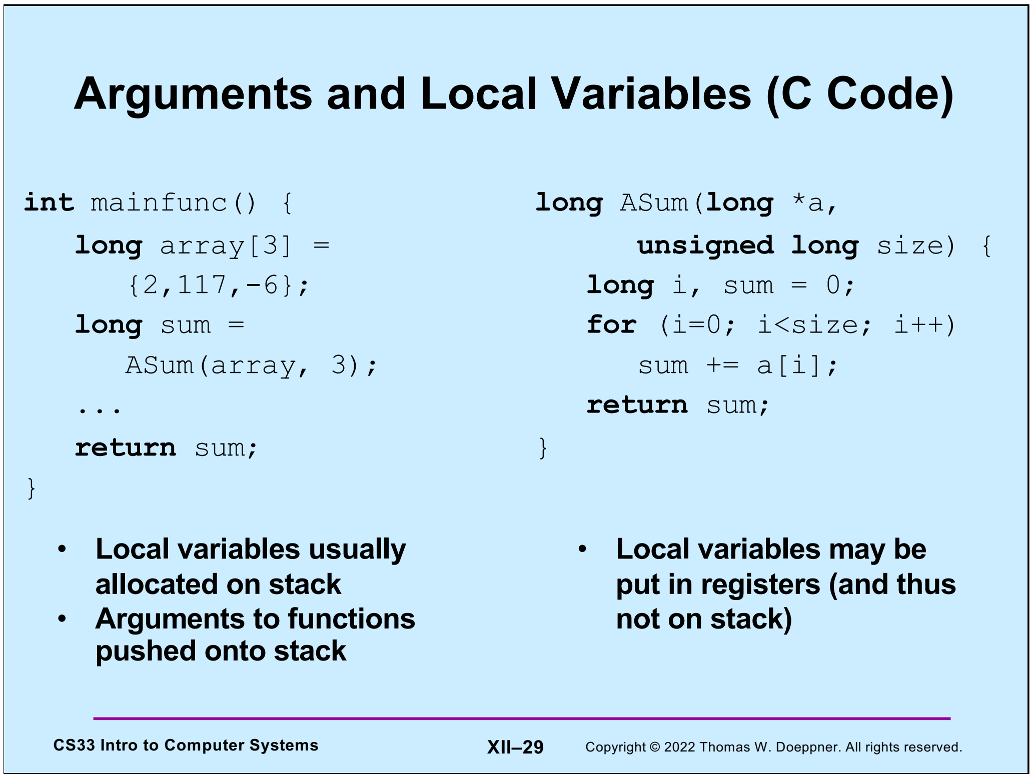
```
0x2000: func:
   ...  ...
0x2200: movq $6, %rax
0x2203: ret
```

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

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	10	04	%rip
00	00	00	0f	ff	f1	00	00	%rsp

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 (C Code)

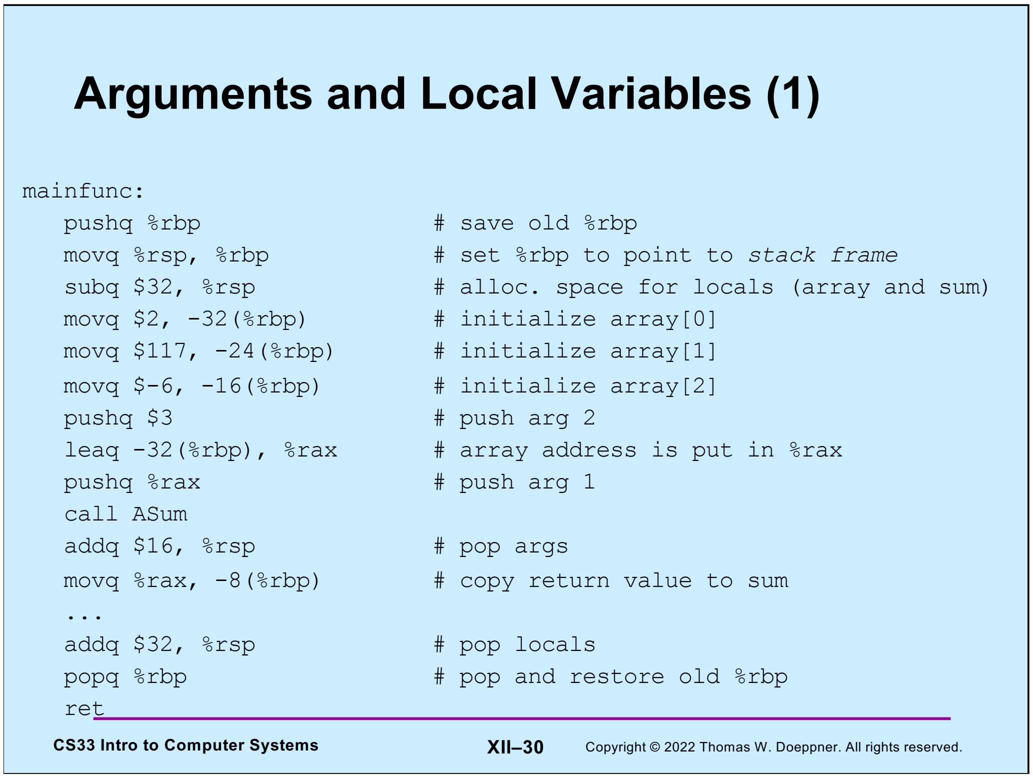
```
int mainfunc() {
   long array[3] =
      {2,117,-6};
   long sum =
      ASum(array, 3);
   ...
   return sum;
}
```

- 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.



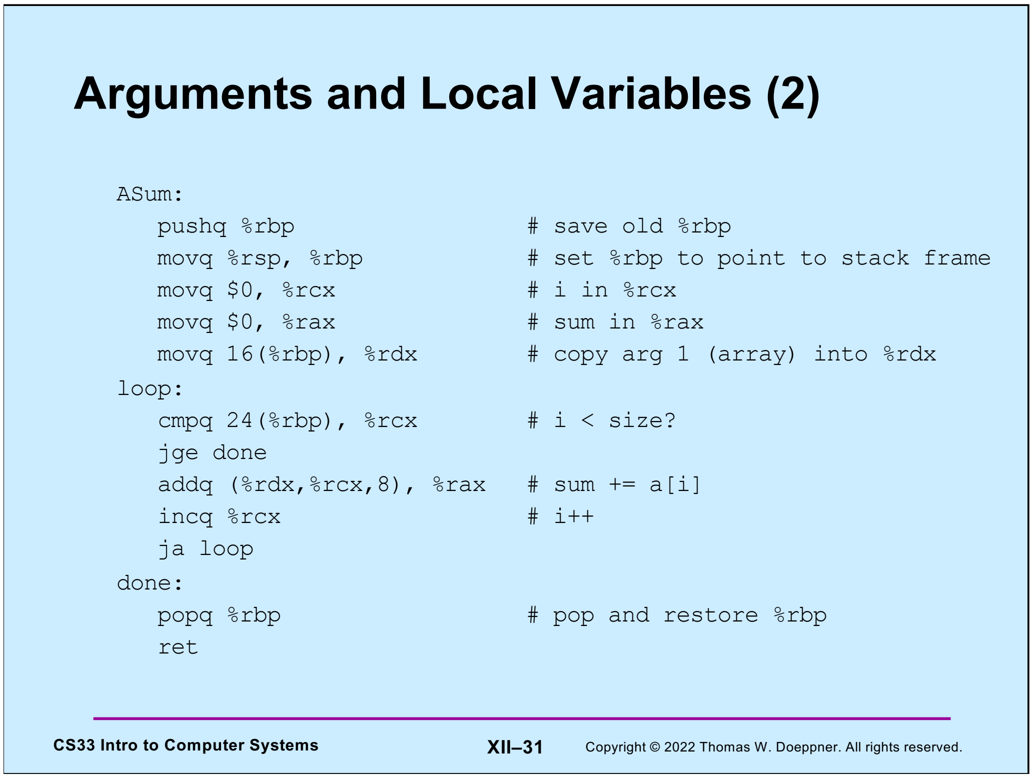
Arguments and Local Variables (1)

```
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.

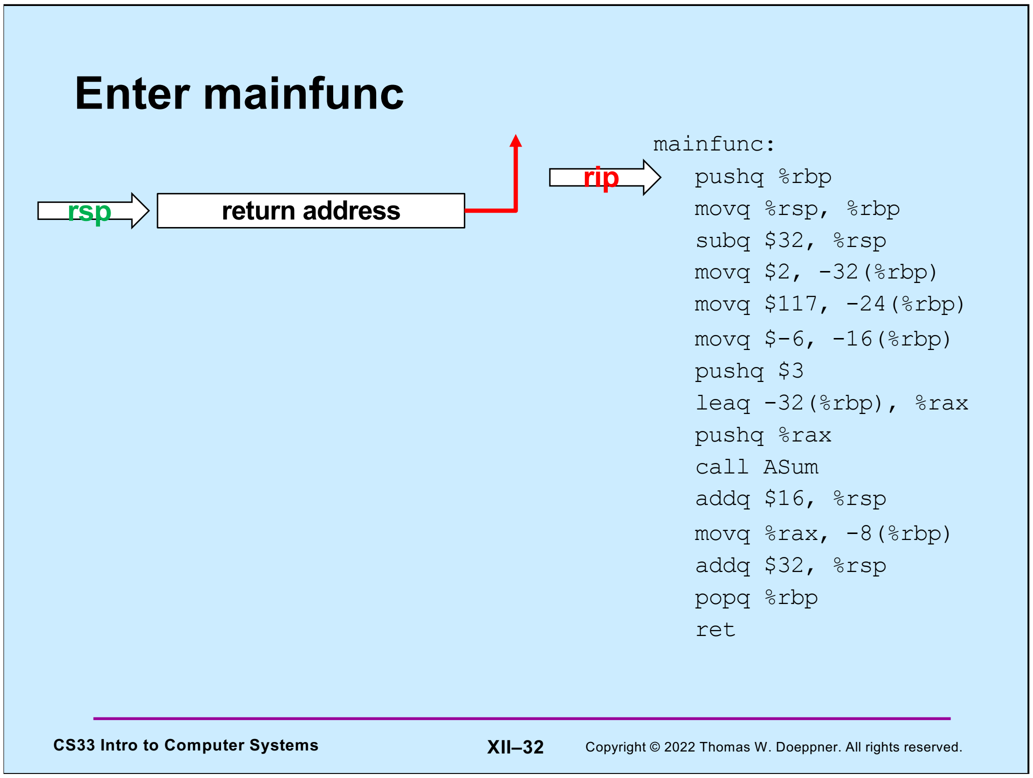
Note, as will be explained, this is not what one would see when compiling it for department computers, on which arguments are passed using registers.



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
```

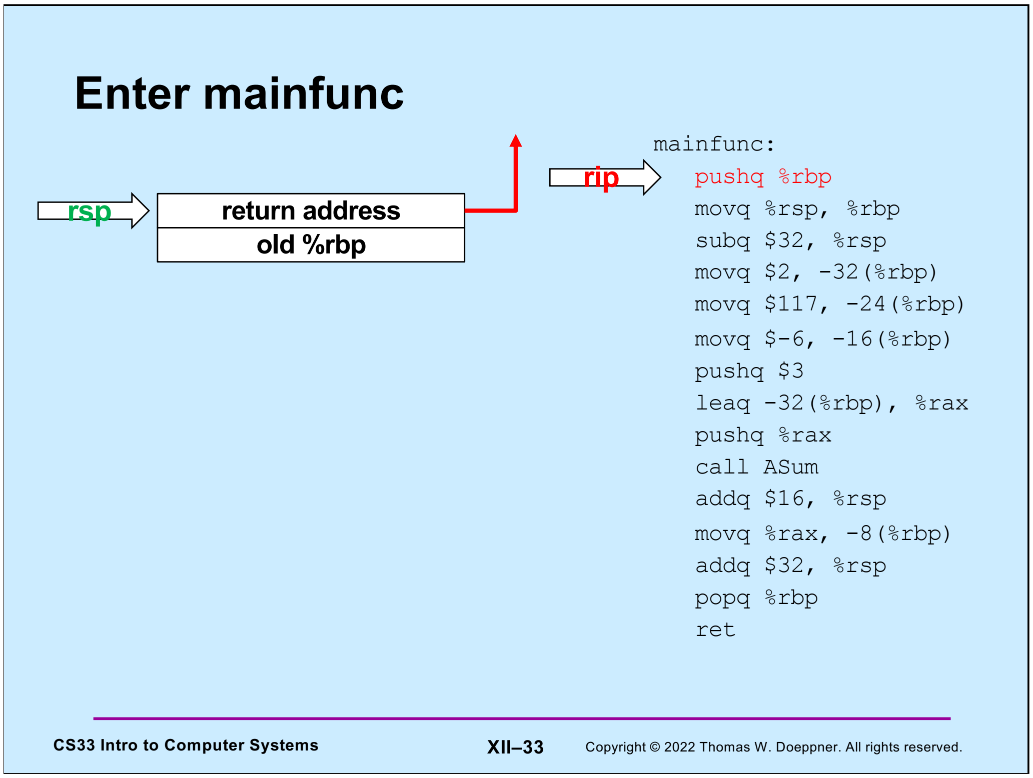
And here is the compiled code for **ASum**. The same caveats as given for the previous slide apply to this one as well.



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

rsp

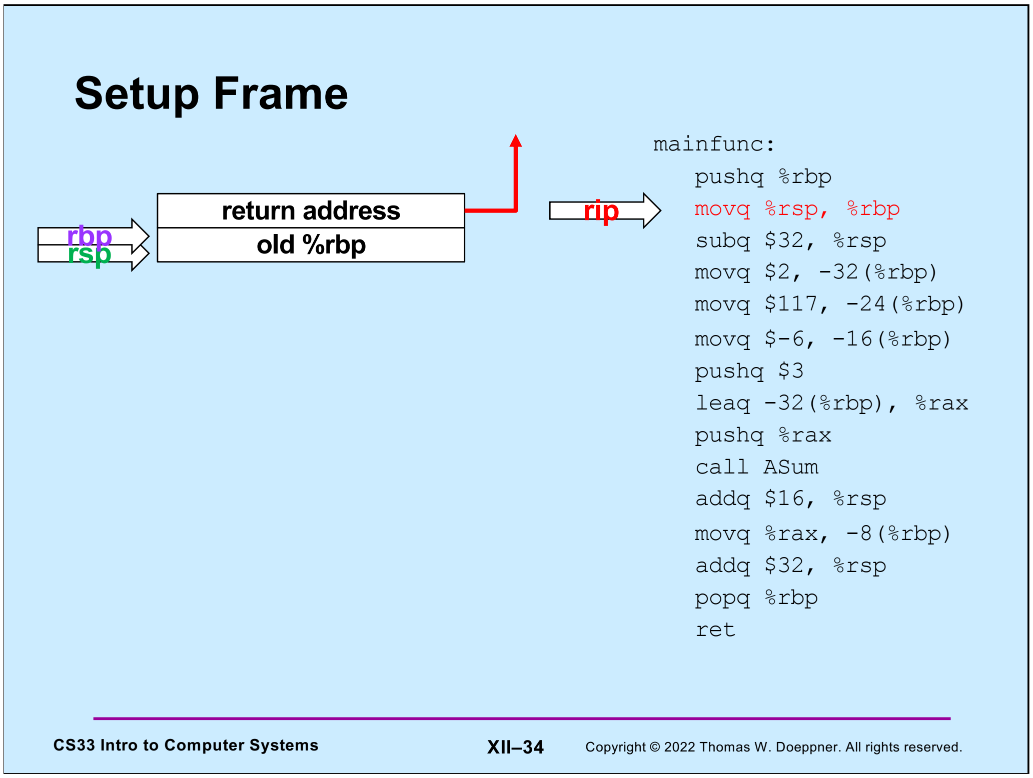
return address

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
```

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



Setup Frame

return address

old %rbp

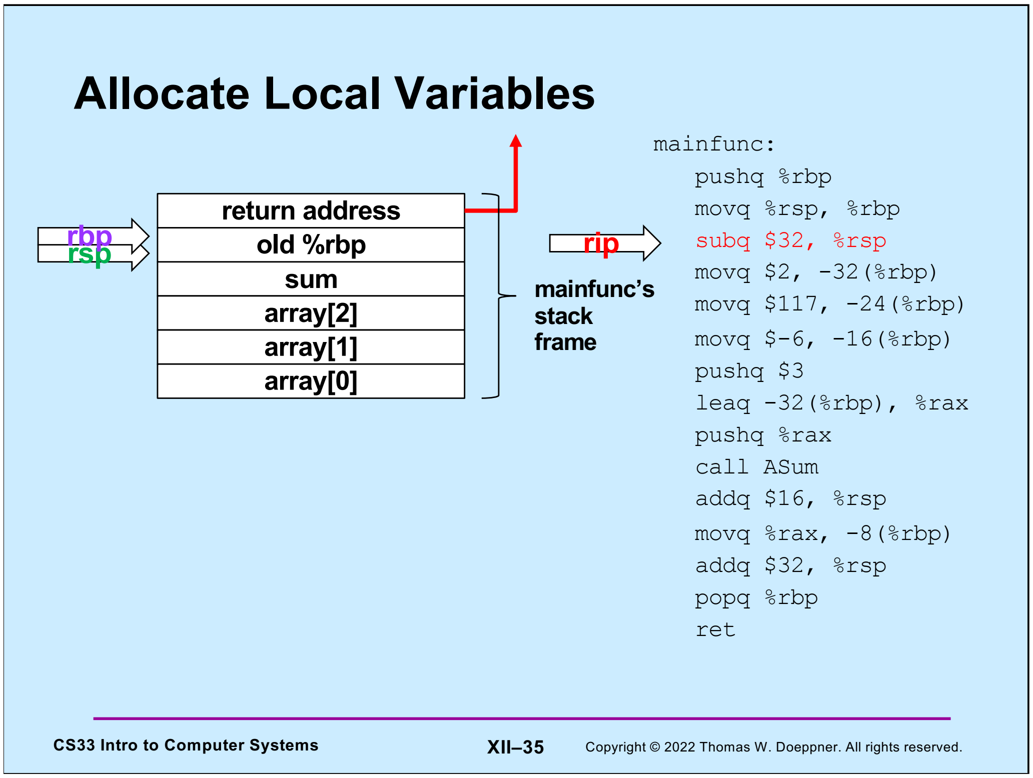
rbp

rsp

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
```

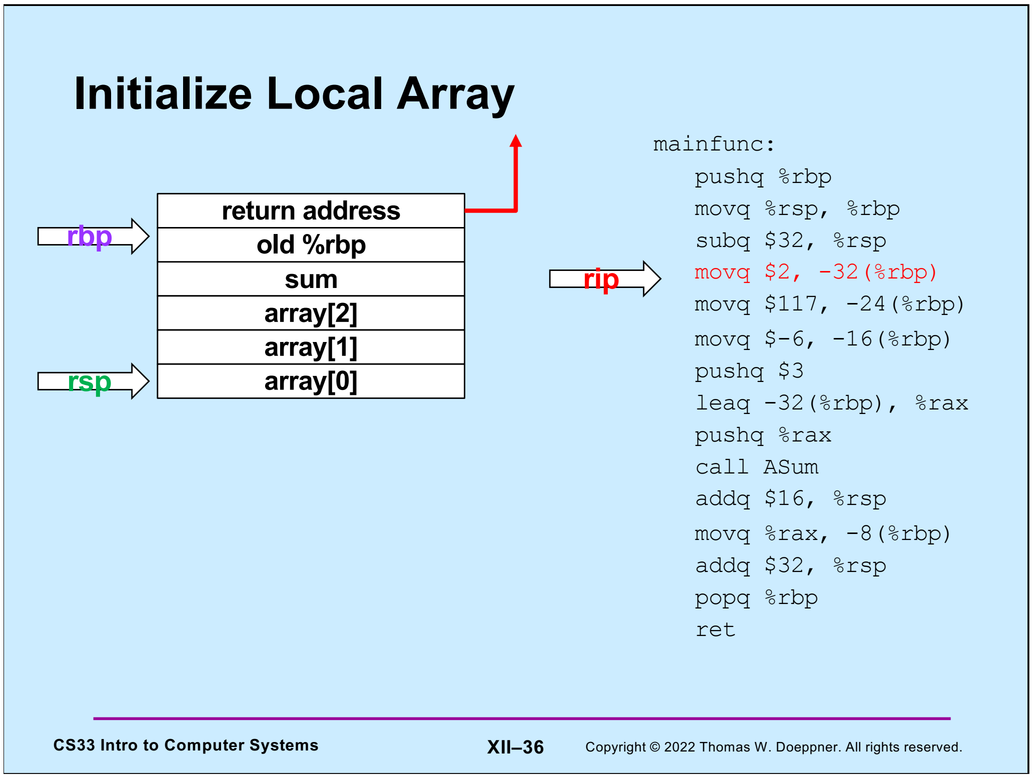
The first thing done by **mainfunc** is to save the caller's %rbp by pushing it onto the stack.



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]

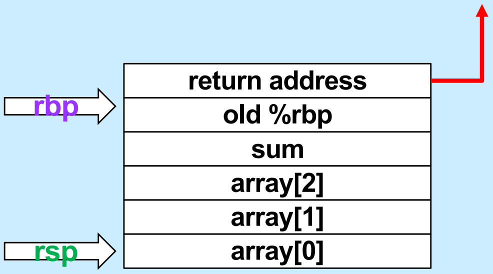
array[0]

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



rbp

rsp

return address

old %rbp

sum

array[2]

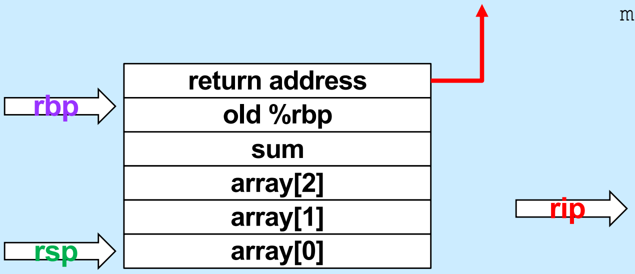
array[1]

array[0]

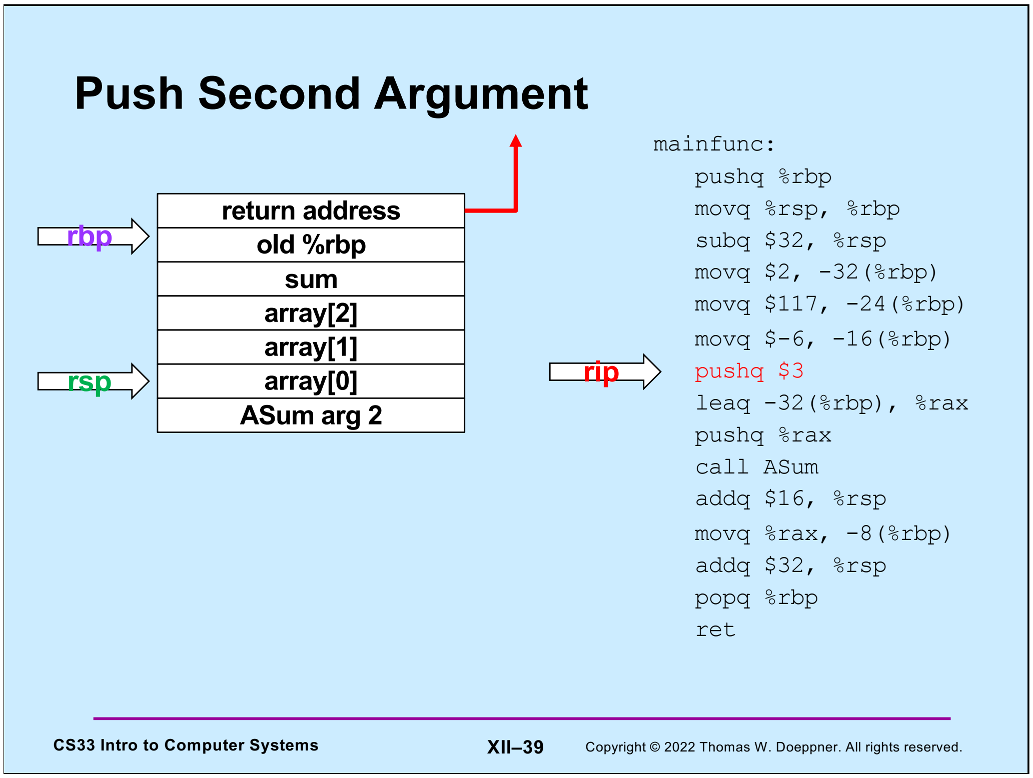
rip

```
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
```

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
```

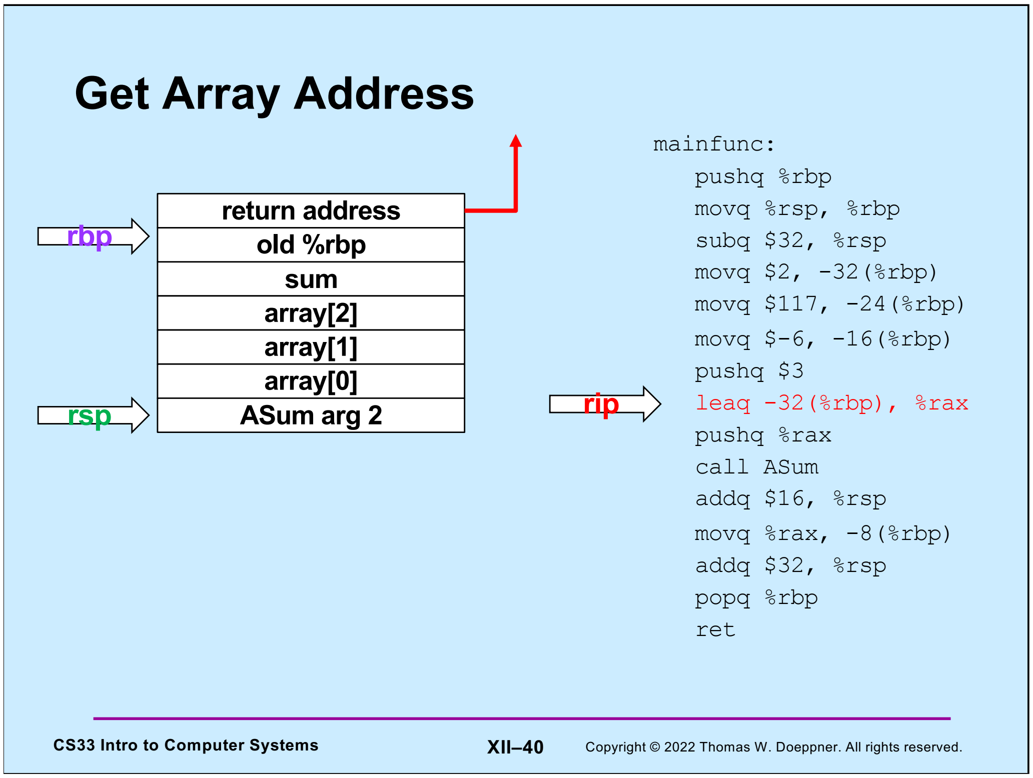


Push Second Argument

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
```

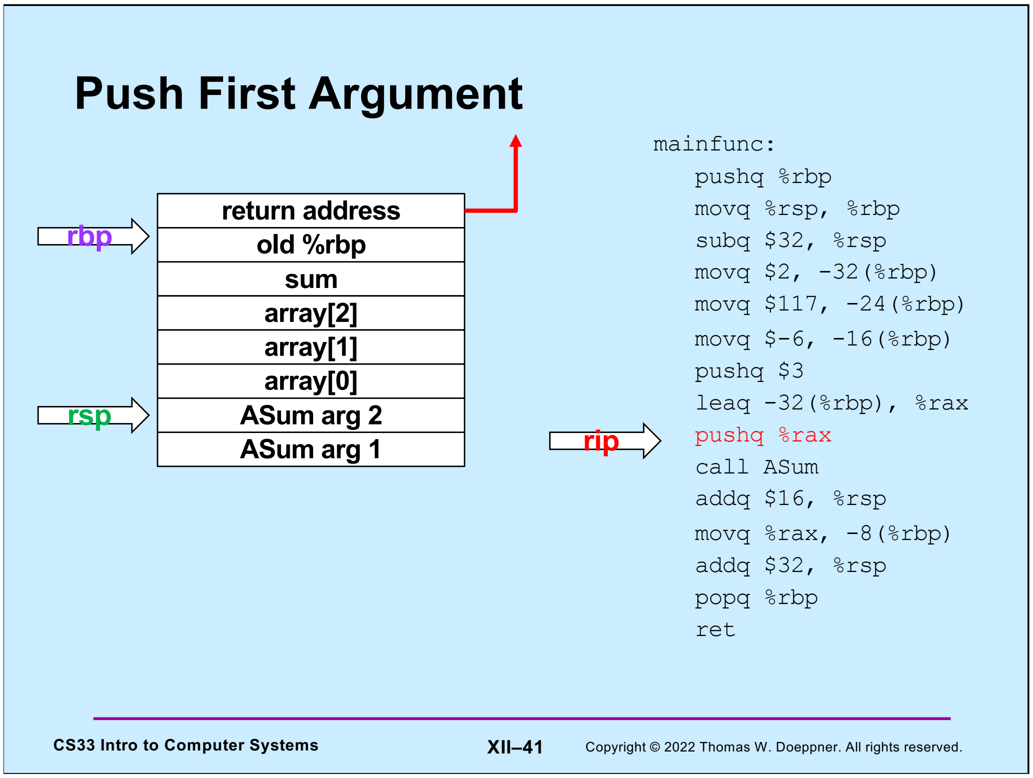
The second argument (3) to **ASum** is pushed onto the stack.



Get Array Address

```
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
```

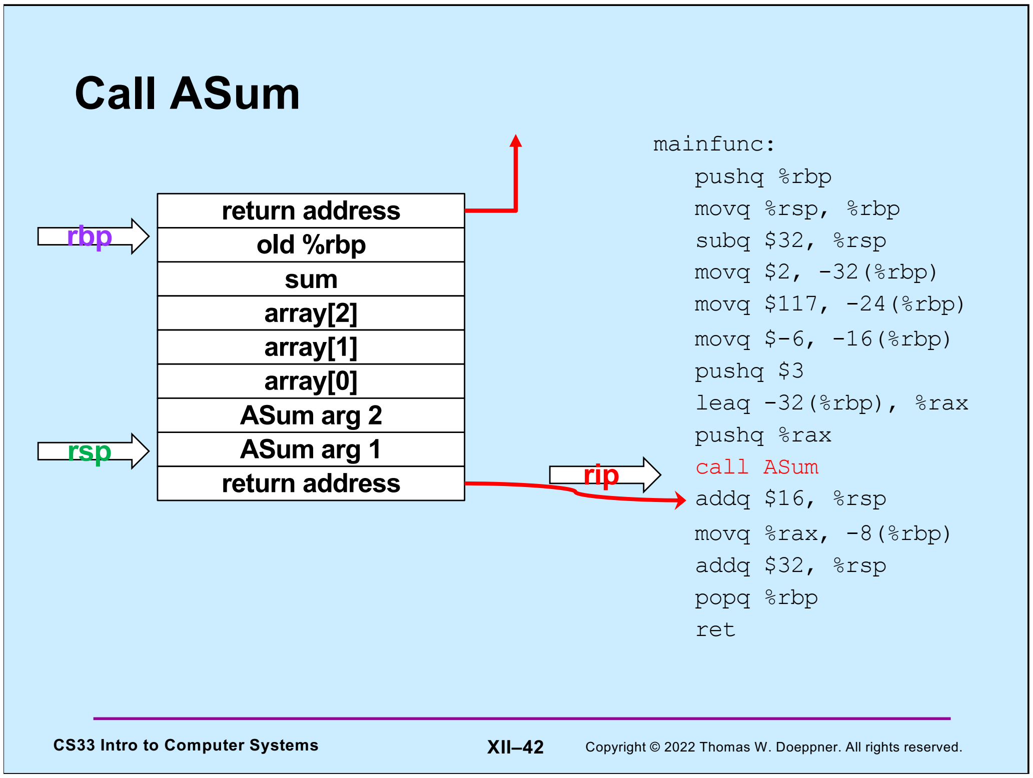
In preparation for pushing the first argument to **ASum** onto the stack, the address of the array is put into %rax.



Push First Argument

```
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
```

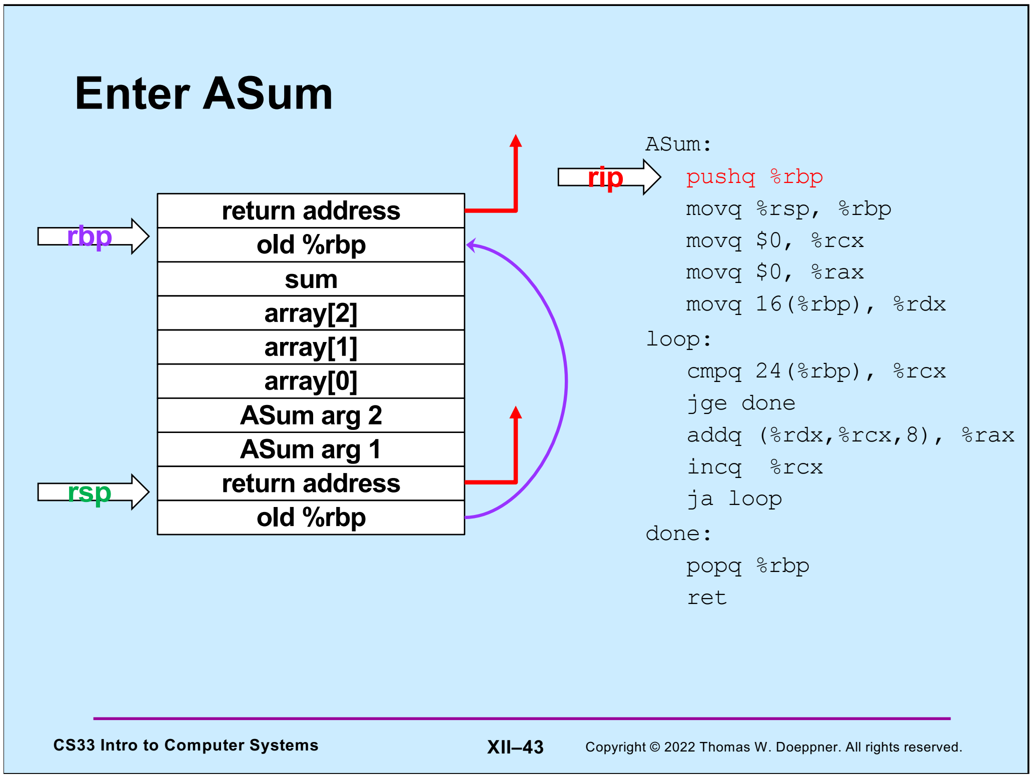
And finally, the address of the array is pushed onto the stack as **ASum**'s first argument.



Call ASum

```
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.

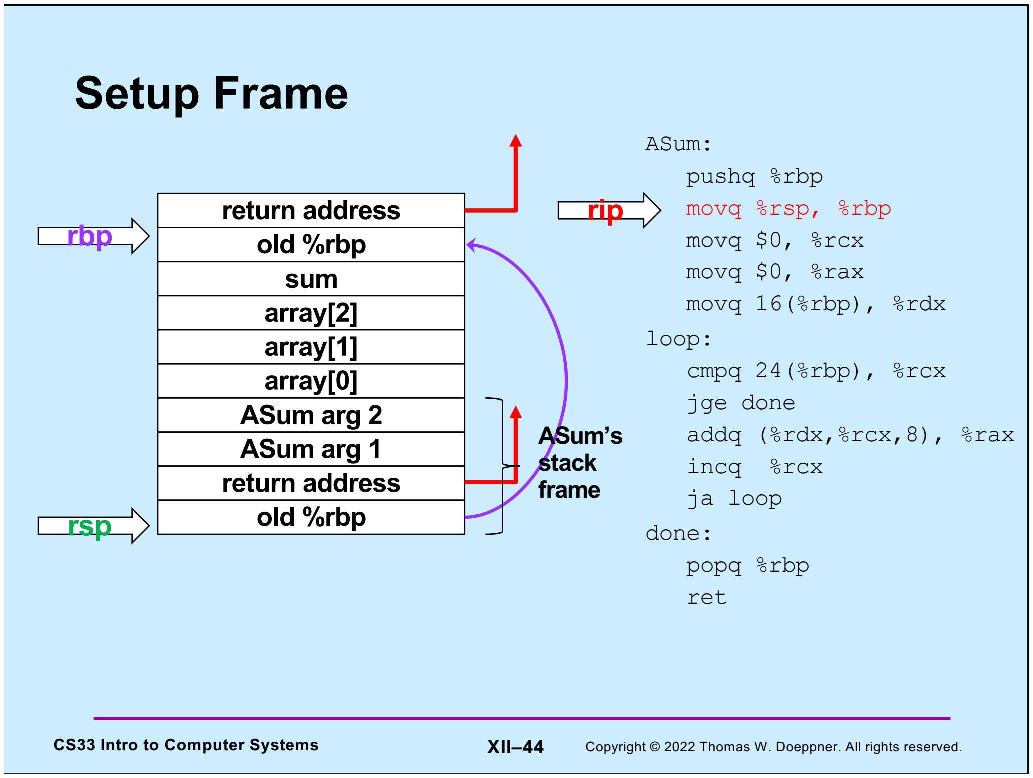


Enter ASum

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

return address

old %rbp

sum

array[2]

array[1]

array[0]

ASum arg 2

ASum arg 1

return address

old %rbp

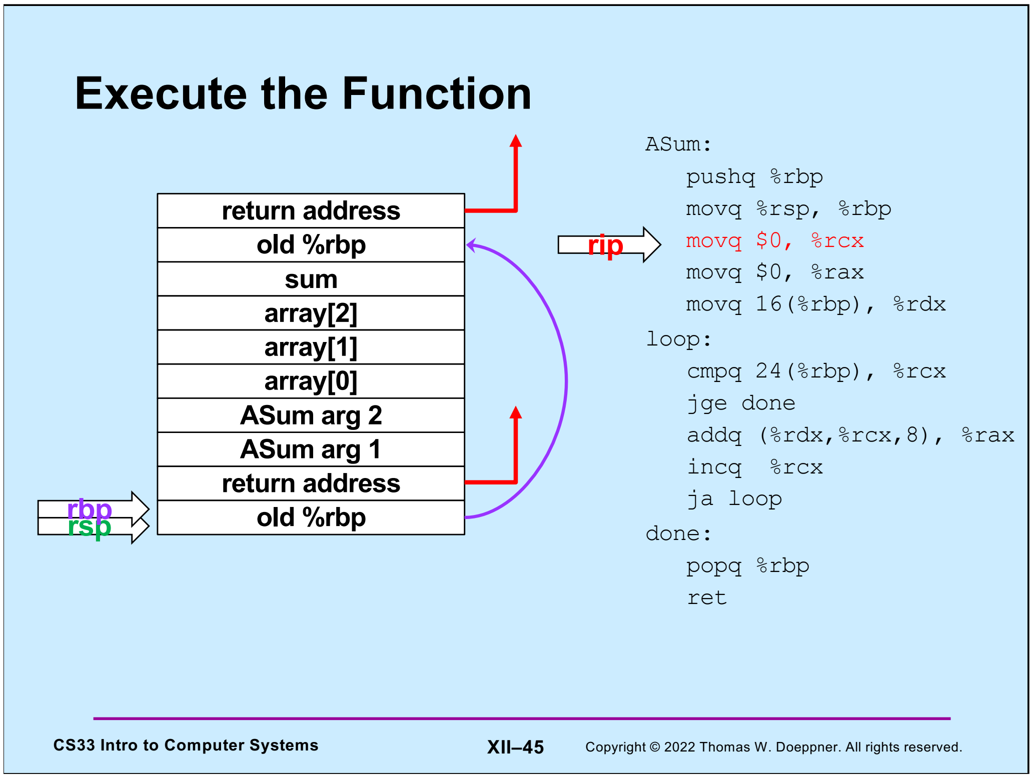
rsp

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.

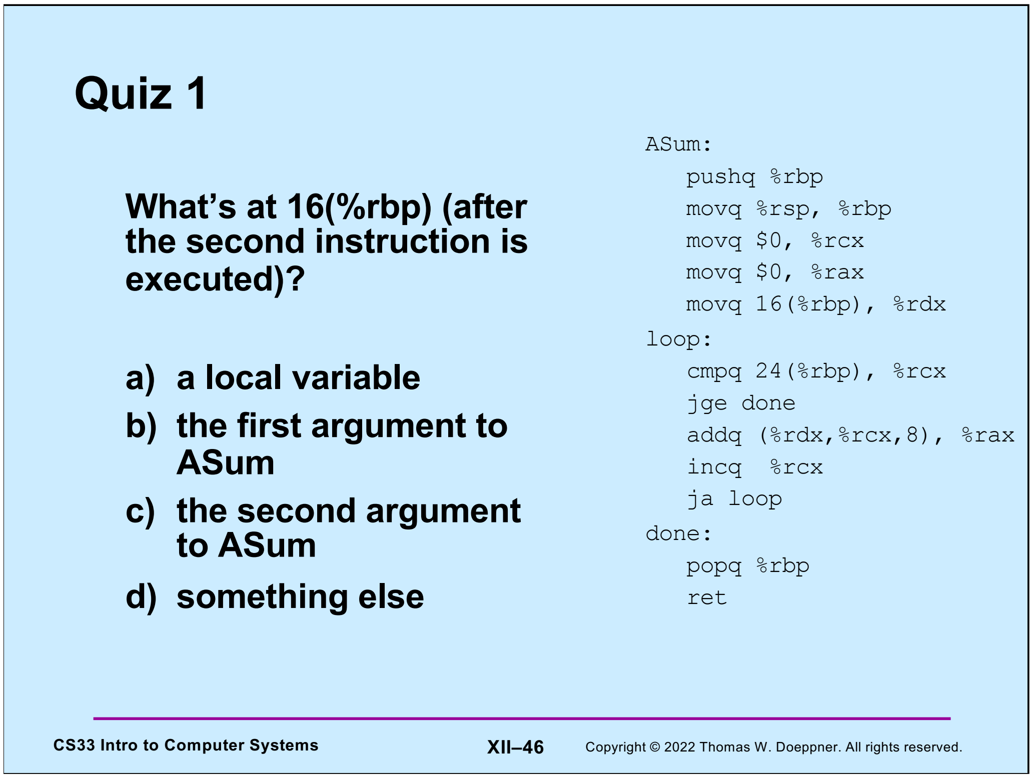


Execute the Function

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

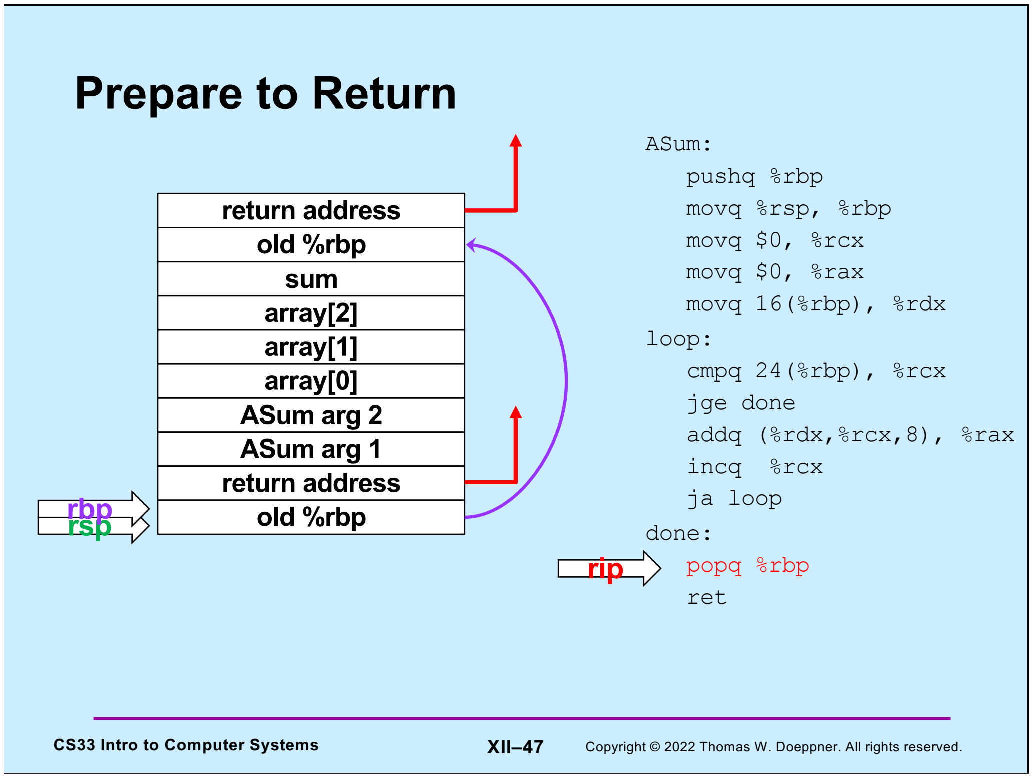
b) the first argument to ASum

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.

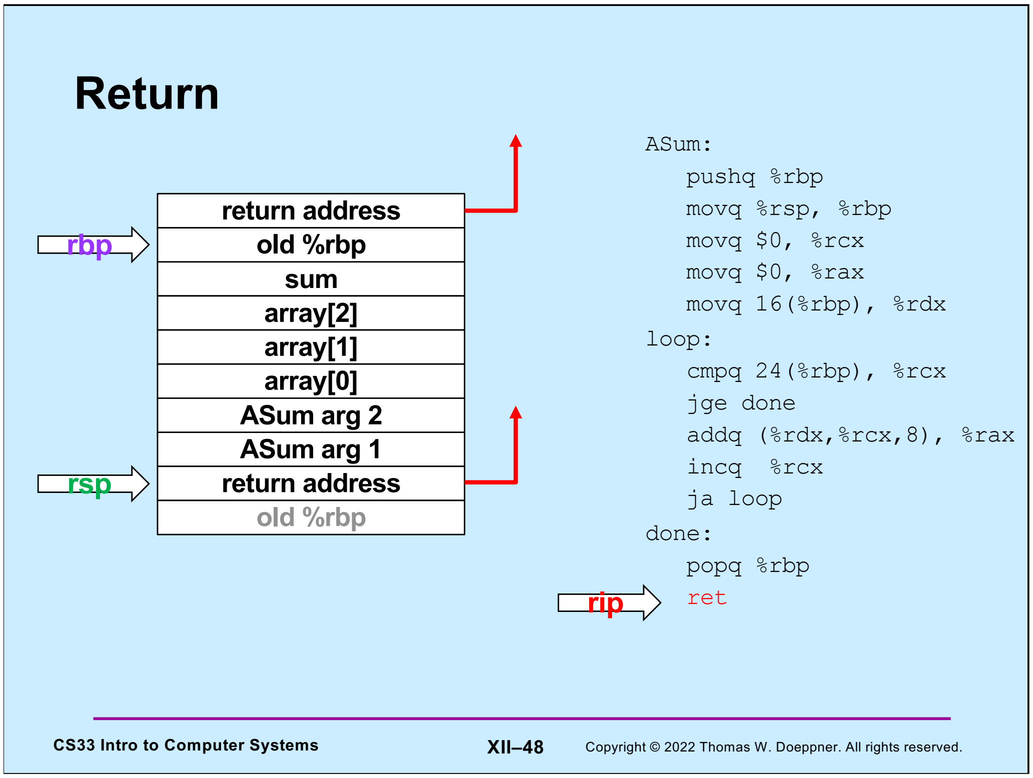


Prepare to Return

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

rbp

rsp

return address

old %rbp

sum

array[2]

array[1]

array[0]

ASum arg 2

ASum arg 1

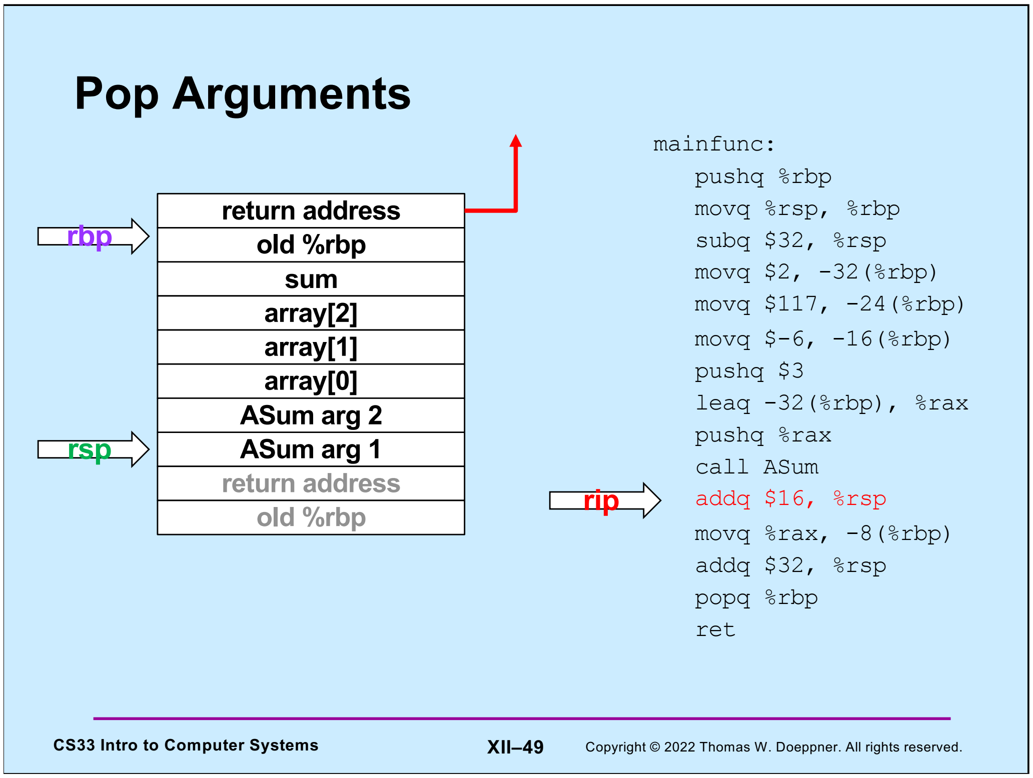
return address

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
```

rip

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

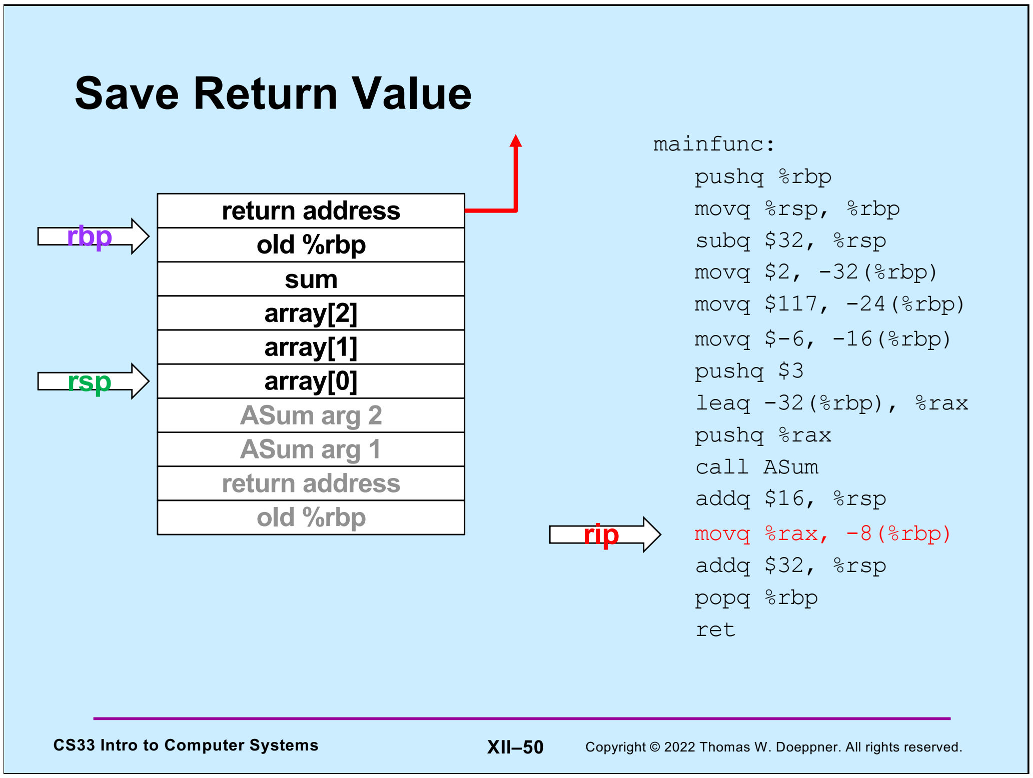


Pop Arguments

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
```

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.

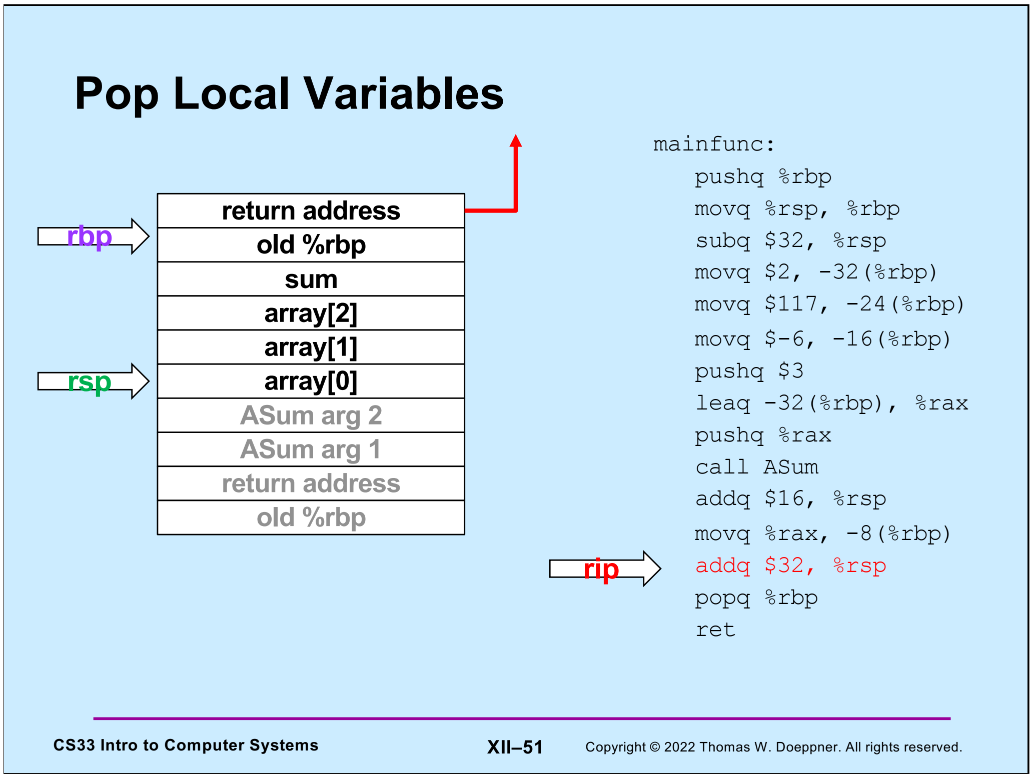


Save Return Value

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
```

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

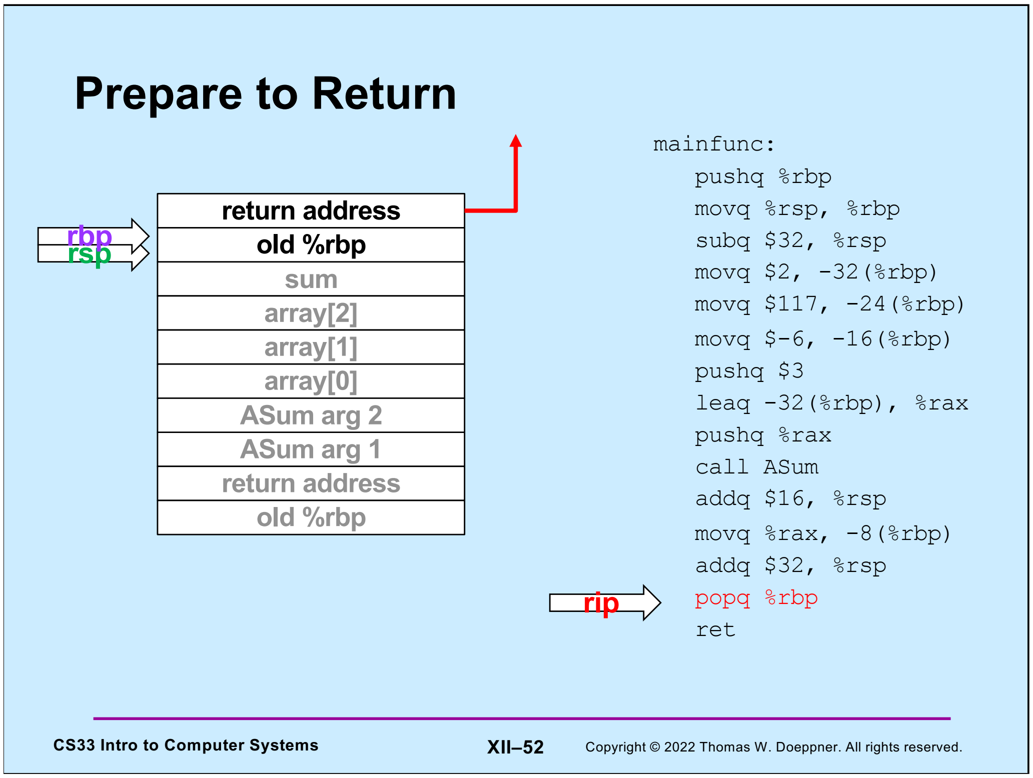
rsp

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

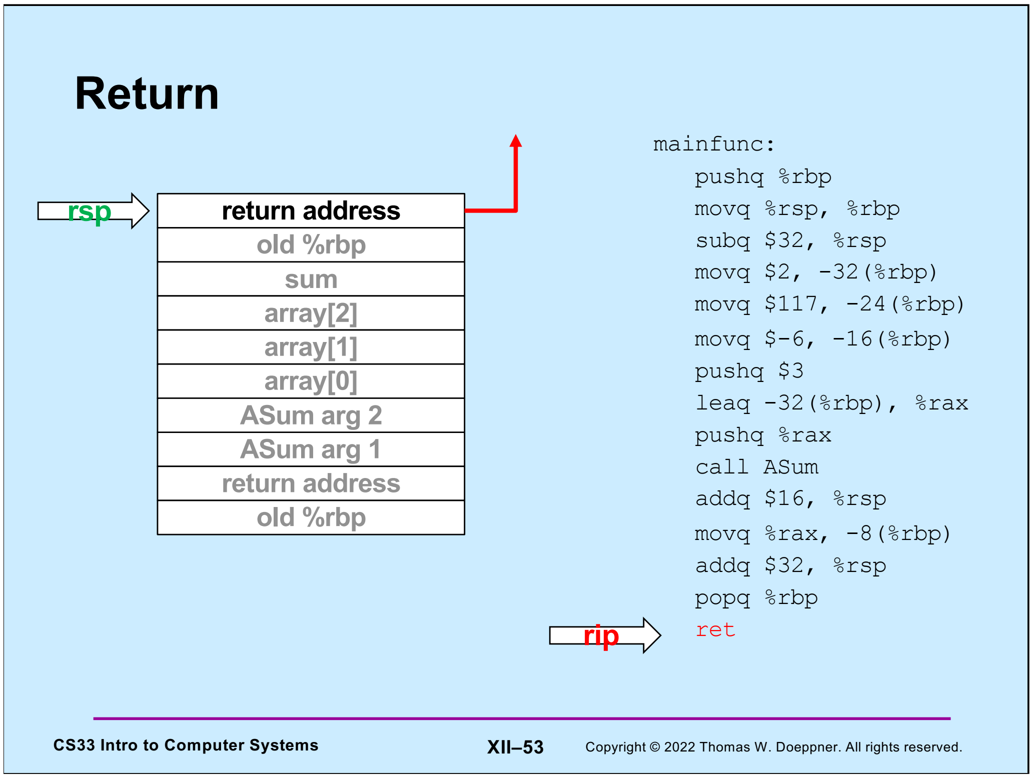
rbp
rsp

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
```

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

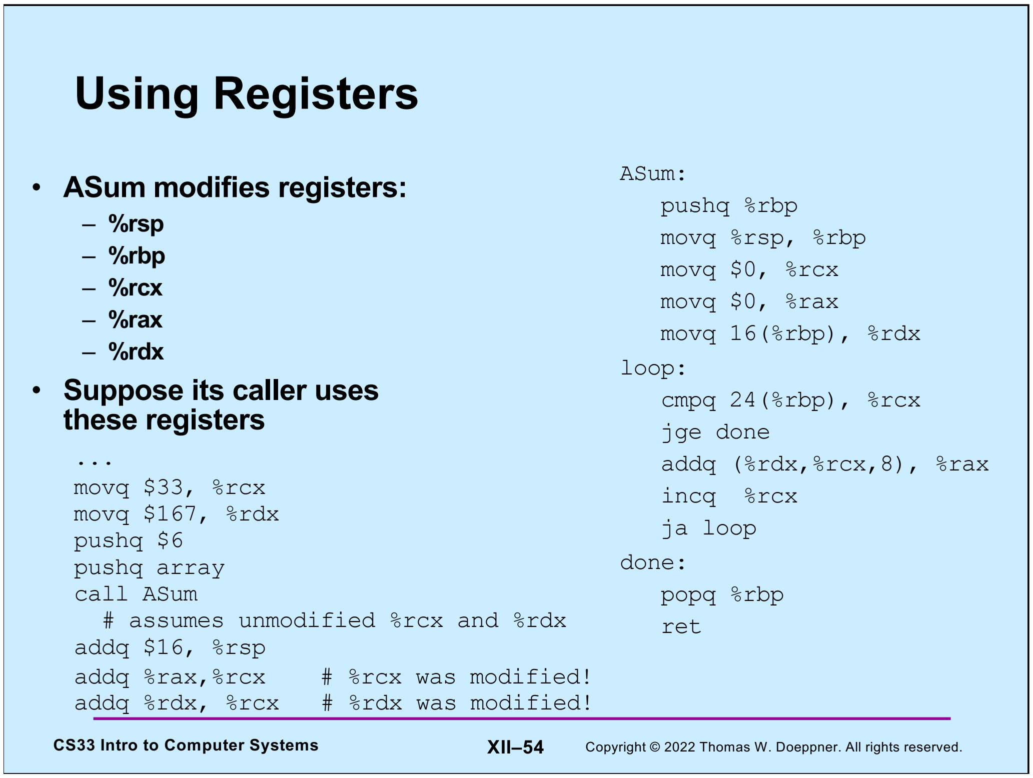


Return

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!
```

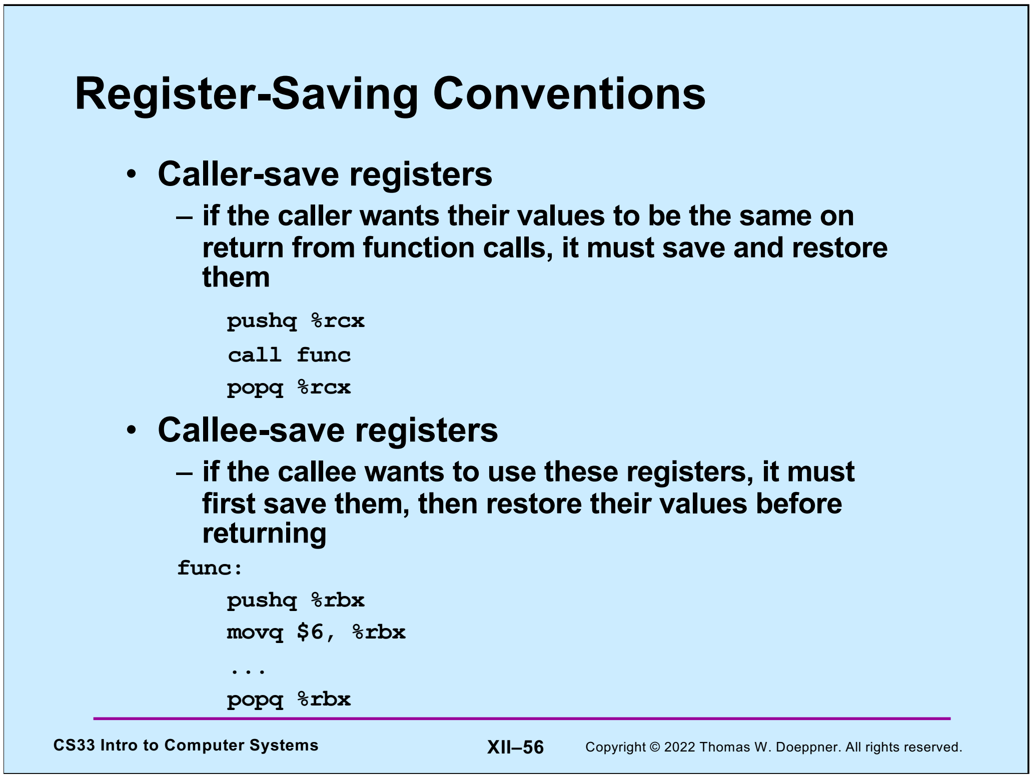
```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```

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**

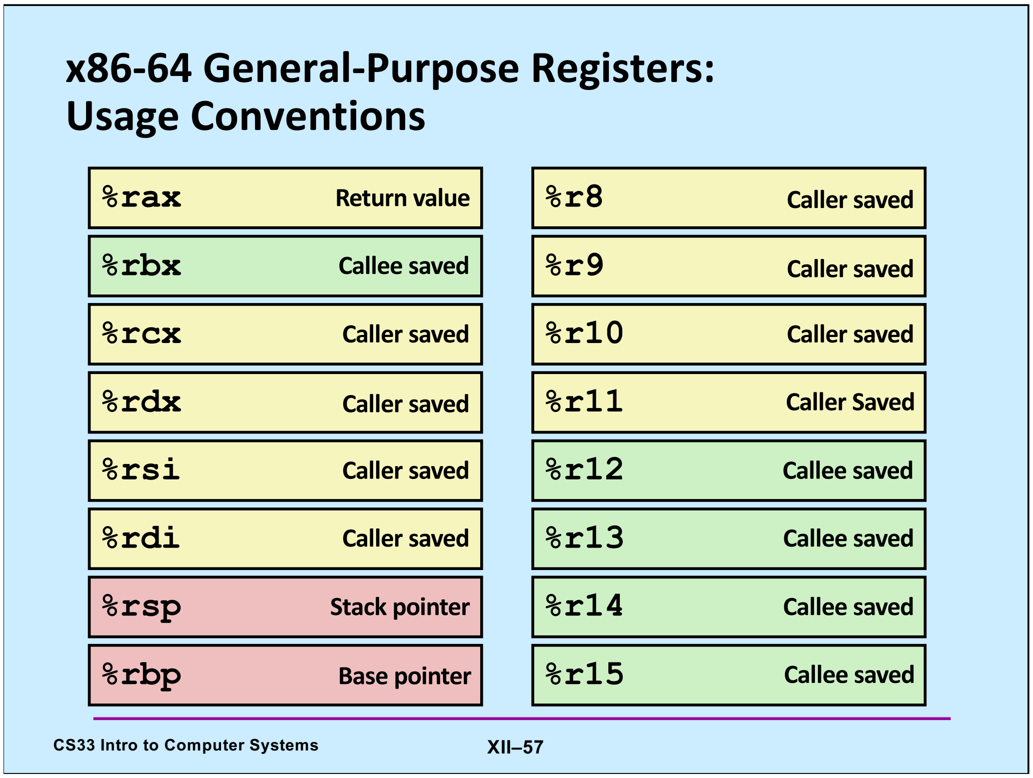
```
    pushq %rcx
    call func
    popq %rcx
```

- **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.

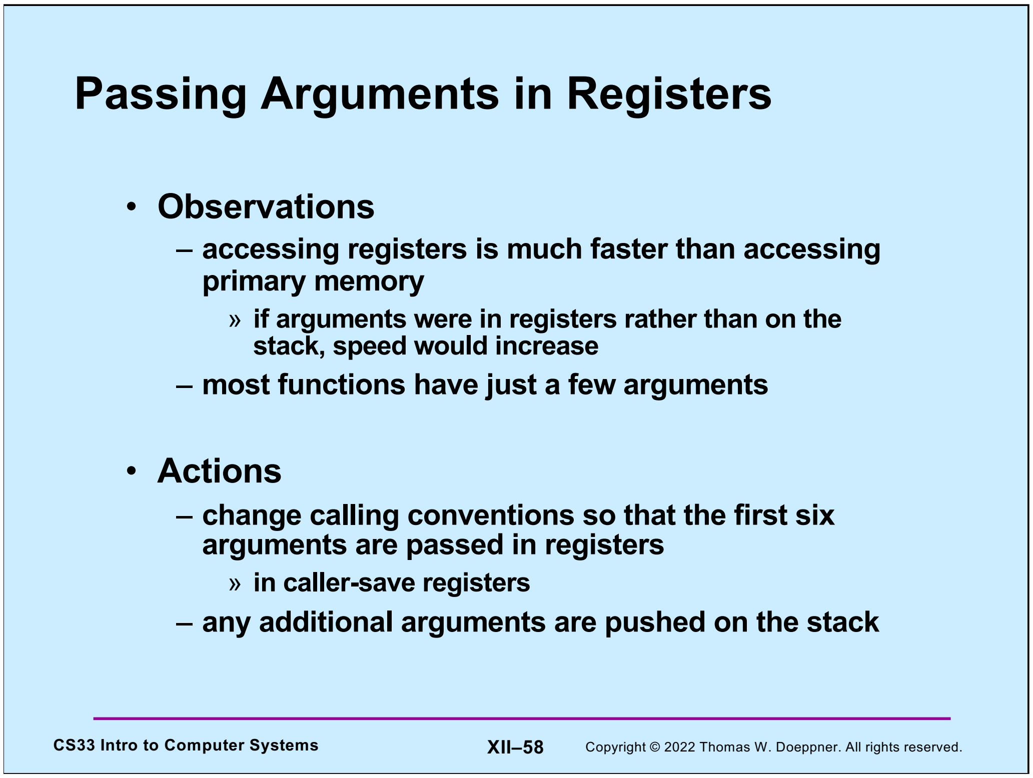


x86-64 General-Purpose Registers: Usage Conventions

`%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.

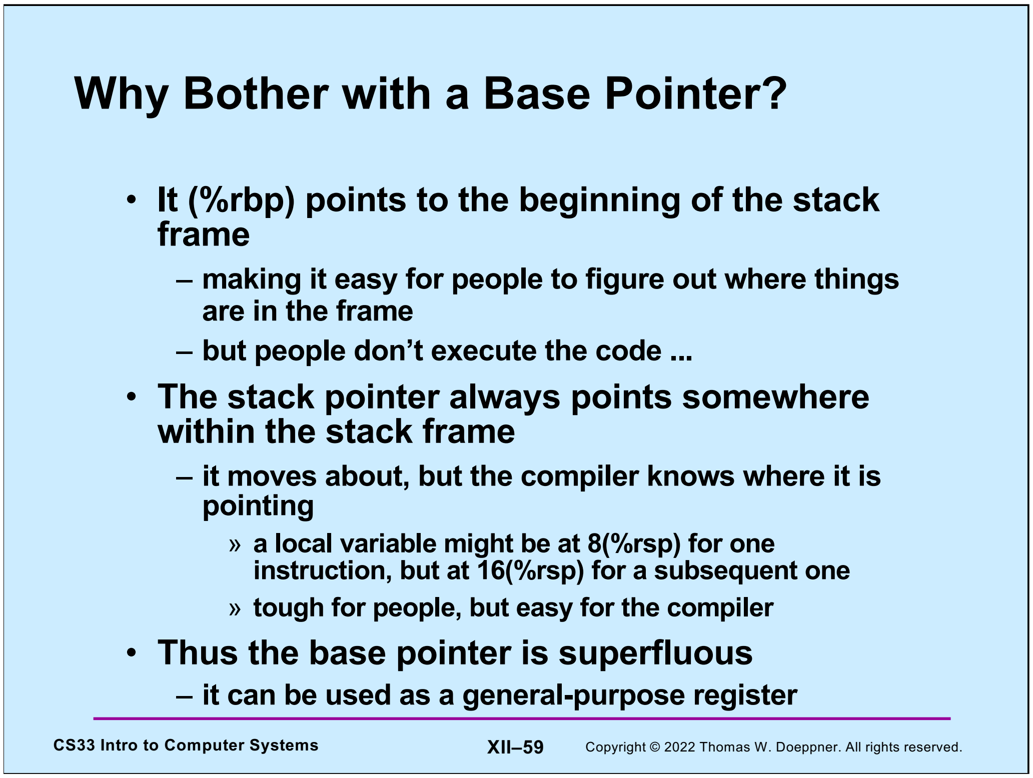
Here is a list of which registers are callee-save, which are caller-save, and which have special purposes. Note that this is merely a convention and not an inherent aspect of the x86-64 architecture.



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

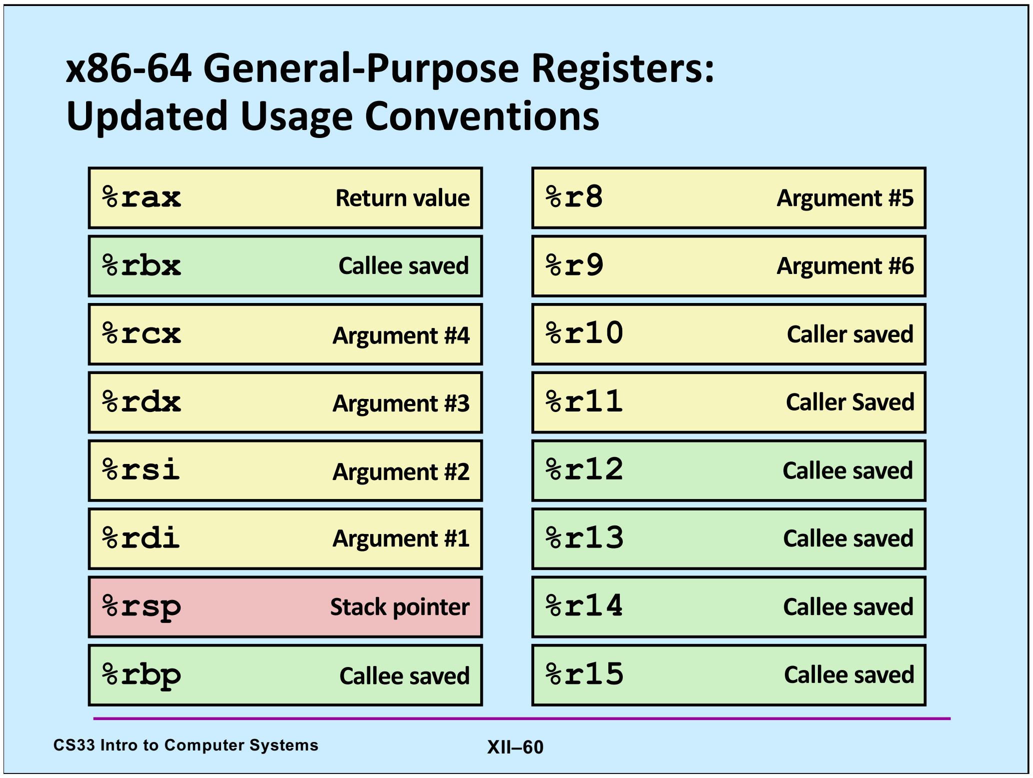
- Actions
  - – change calling conventions so that the first six arguments are passed in registers
    - » in caller-save registers
  - – any additional arguments are pushed on the stack



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

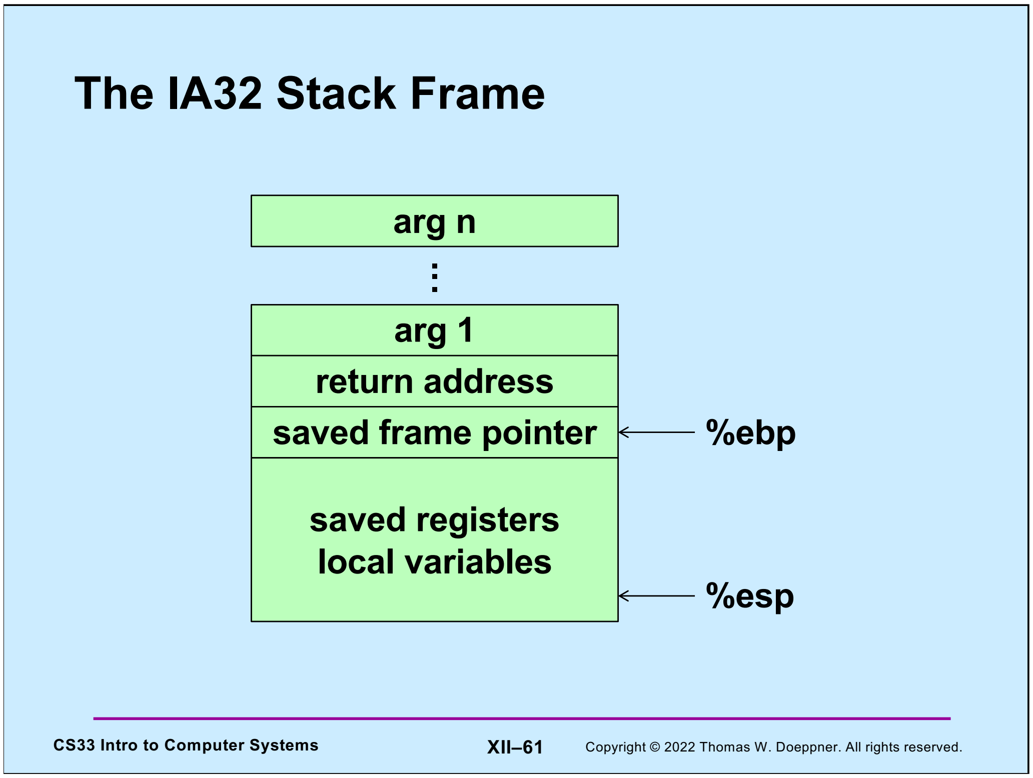
If one gives gcc the –O0 flag (which turns off all optimization) when compiling, the base pointer (%rbp) will be used as in IA32: it is set to point to the stack frame and the arguments are copied from the registers into the stack frame. This clearly slows down the execution of the function, but makes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



The IA32 Stack Frame

arg n

⋮

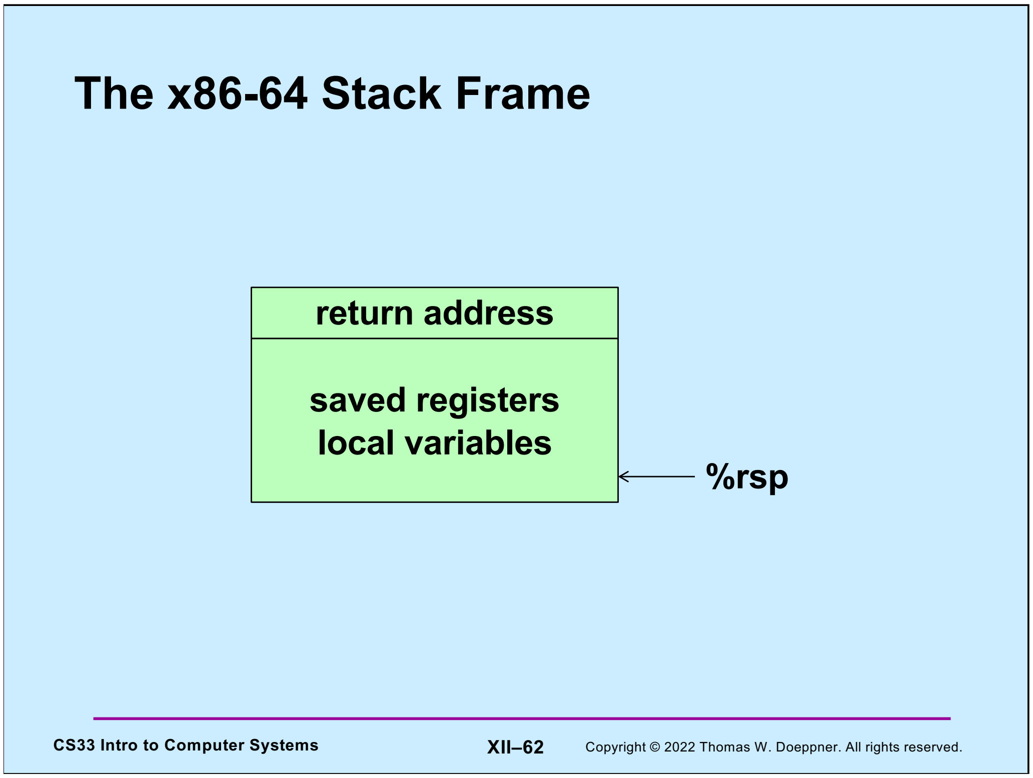
arg 1

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.



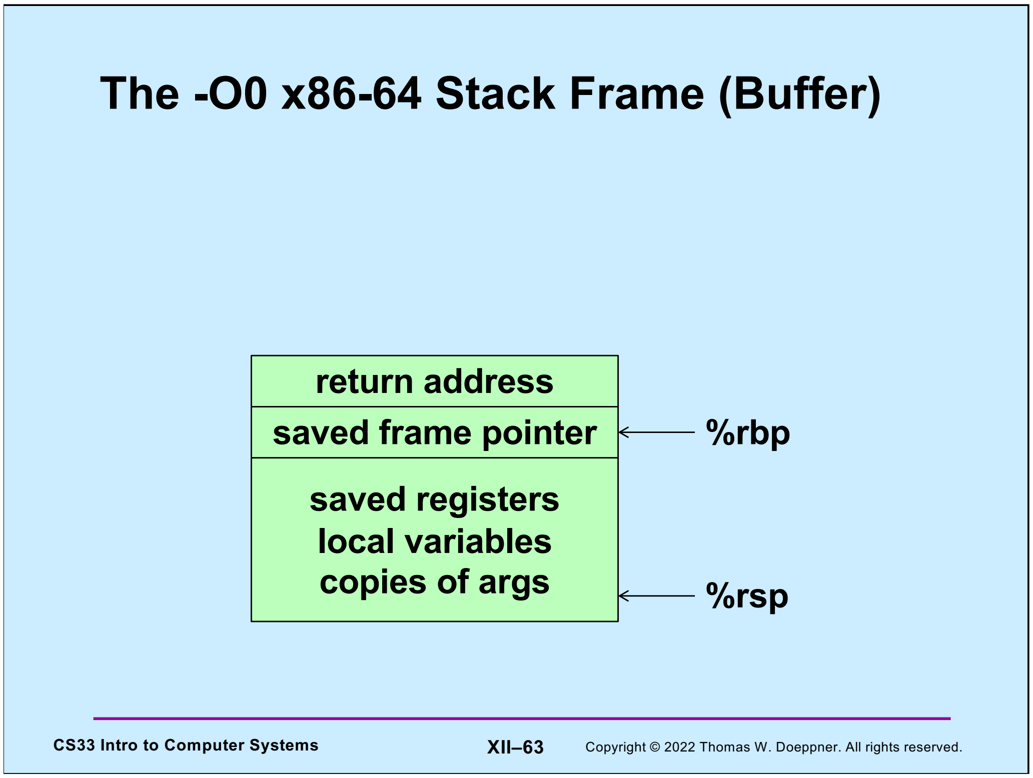
The x86-64 Stack Frame

return address

saved registers
local variables

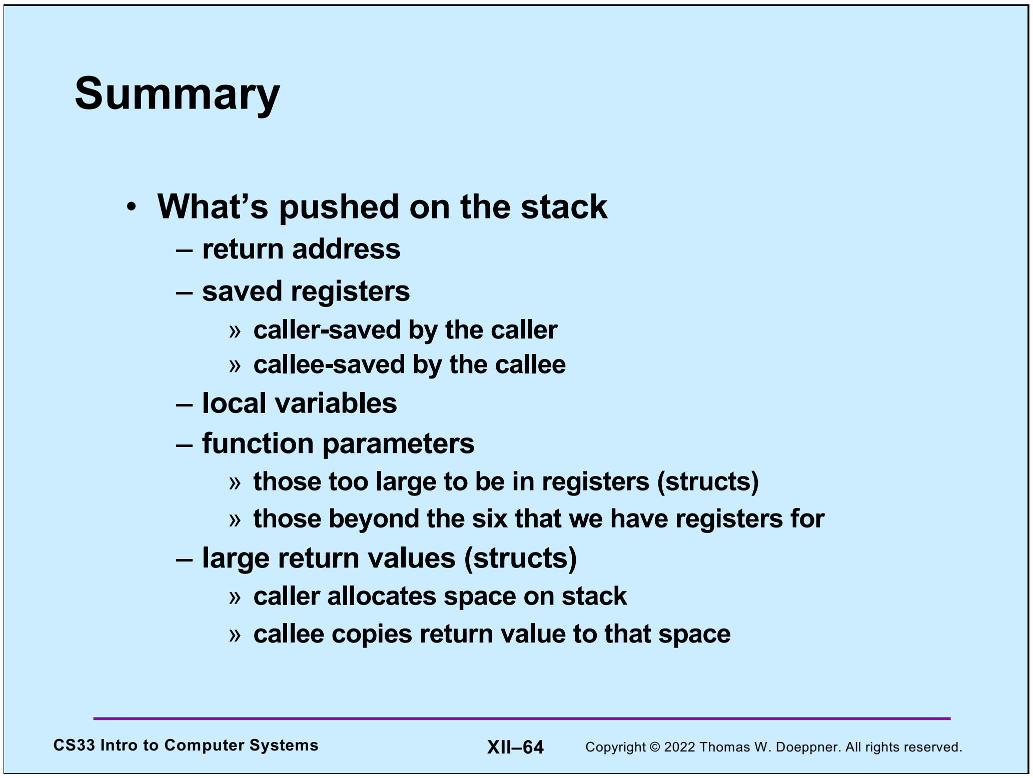
%rsp

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.



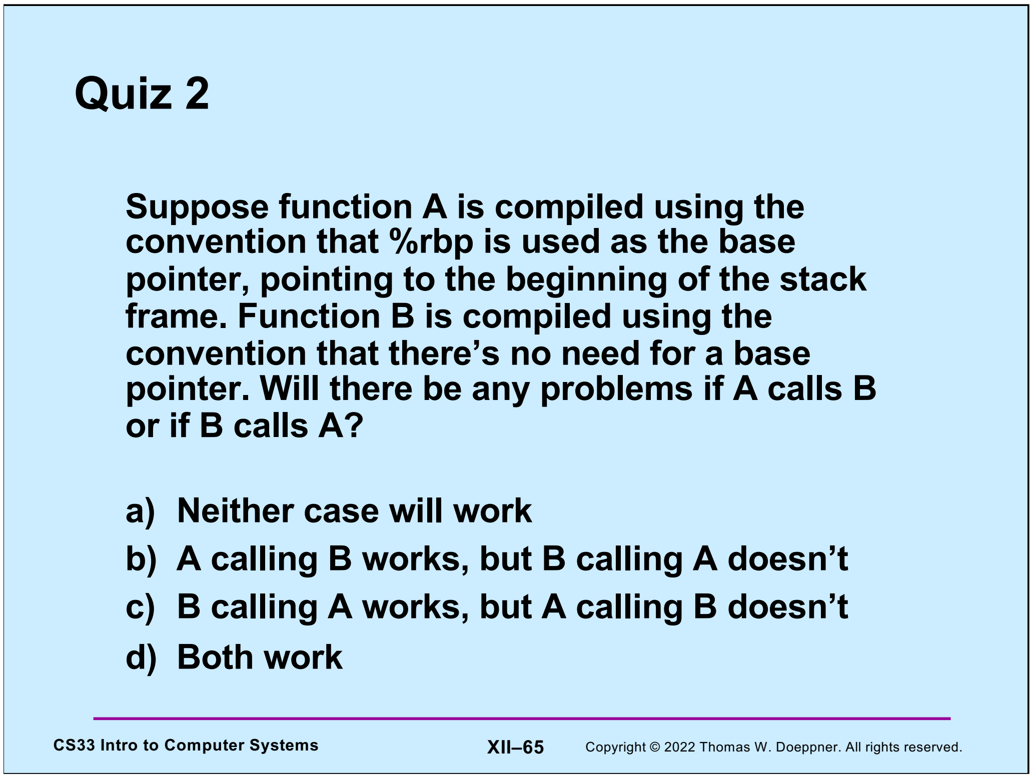
The -O0 x86-64 Stack Frame (Buffer)

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 –O 0 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.