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.

Jumping

- **jX instructions**
  - **Jump to different part of program depending on condition codes**

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Supplied by CMU.

See the notes for slide 28.

Conditional-Branch Example

```c
int absdiff(int x, int y)
{
    int result;
    if (x > y) {
      result = x-y;
    } else {
      result = y-x;
    }
    return result;
}
```

```
absdiff:
    movl    %esi, %eax     } Body1
    cmpl    %esi, %edi     }
    jle     .L6            }
    subl    %eax, %edi     } Body2a
    movl    %edi, %eax     }
    jmp     .L7            }
.L6:
    subl %edi, %eax        } Body2b
.L7:
    ret
```

x in %edi

y in %esi

Supplied by CMU, but converted to x86-64.

The function computes the absolute value of the difference between its two arguments.

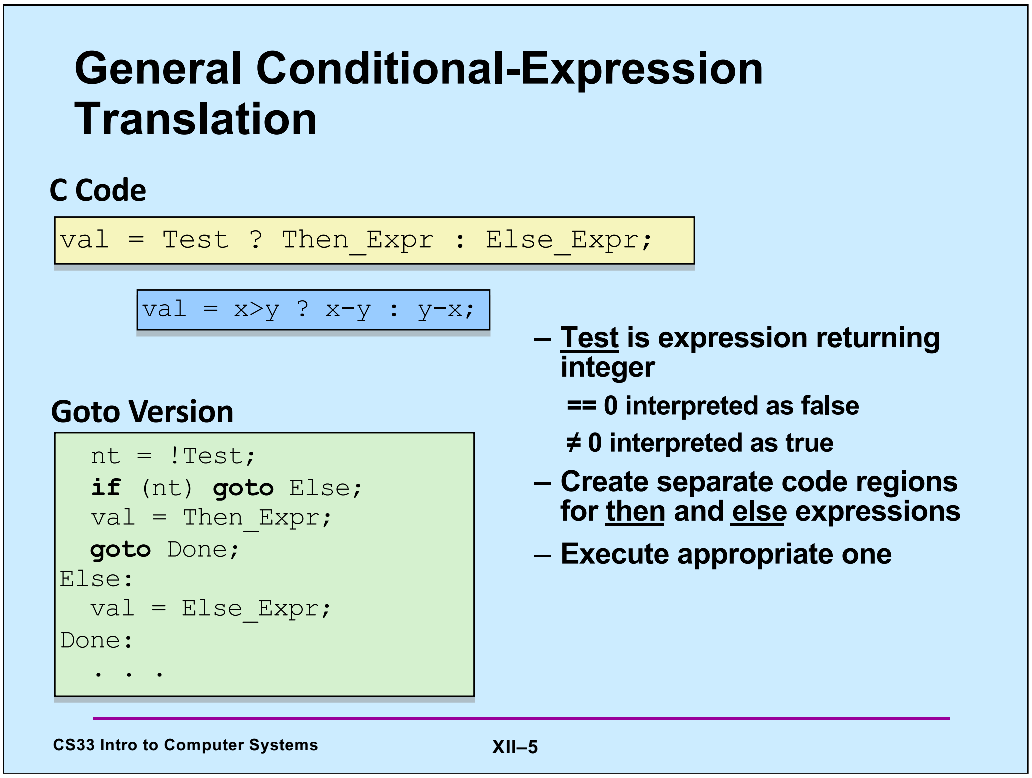
Conditional-Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
  goto Exit;
Else:
  result = y-x;
Exit:
  return result;
}
```

```
absdiff:
    movl   %esi, %eax     } Body1
    cmpl   %esi, %edi
    jle    .L6
    subl   %eax, %edi     } Body2a
    movl   %edi, %eax
    jmp .L7
.L6:
    subl %edi, %eax       } Body2b
.L7:
    ret
```

- C allows “goto” as means of transferring control
  - closer to machine-level programming style
- Generally considered bad coding style

Supplied by CMU, but converted to x86-64.



Supplied by CMU.

C's conditional expression, as shown in the slide, is sometimes useful, but often results in really difficult-to-read code.

(There's an "International Obfuscated C Code Contest" (IOCCC) that awards prizes to those who use valid syntax to write the most difficult-to-understand implementations of simple functions. The conditional expression features prominently in winners' code. See https://www.ioccc.org/.)

“Do-While” Loop Example

C Code

```
int pcount_do(unsigned x)
{
  int result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)
{
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

- **Count number of 1’s in argument x (“popcount”)**
- **Use conditional branch either to continue looping or to exit loop**

Supplied by CMU.

"Do-While" Loop Compilation

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
  return result;
}
```

Registers:

%edi	x
%eax	result

```
    movl   $0, %eax     #   result = 0
.L2:       # loop:
    movl   %edi, %ecx
    andl   $1, %ecx     #   t = x & 1
    addl   %ecx, %eax   #   result += t
    shrl   %edi         #   x >>= 1
    jne    .L2          #   if !0, goto loop
```


Supplied by CMU.

Note that the condition codes are set as part of the execution of the **shrl** instruction.

General “Do-While” Translation

C Code

```
do
  Body
  while (Test);
```

- **Body:**

```
{
  Statement1;
  Statement2;
    …
  Statementn;
}
```

- **Test returns integer**
  - **= 0 interpreted as false**
  - **≠ 0 interpreted as true**

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```


Supplied by CMU.

"While" Loop Example

C Code

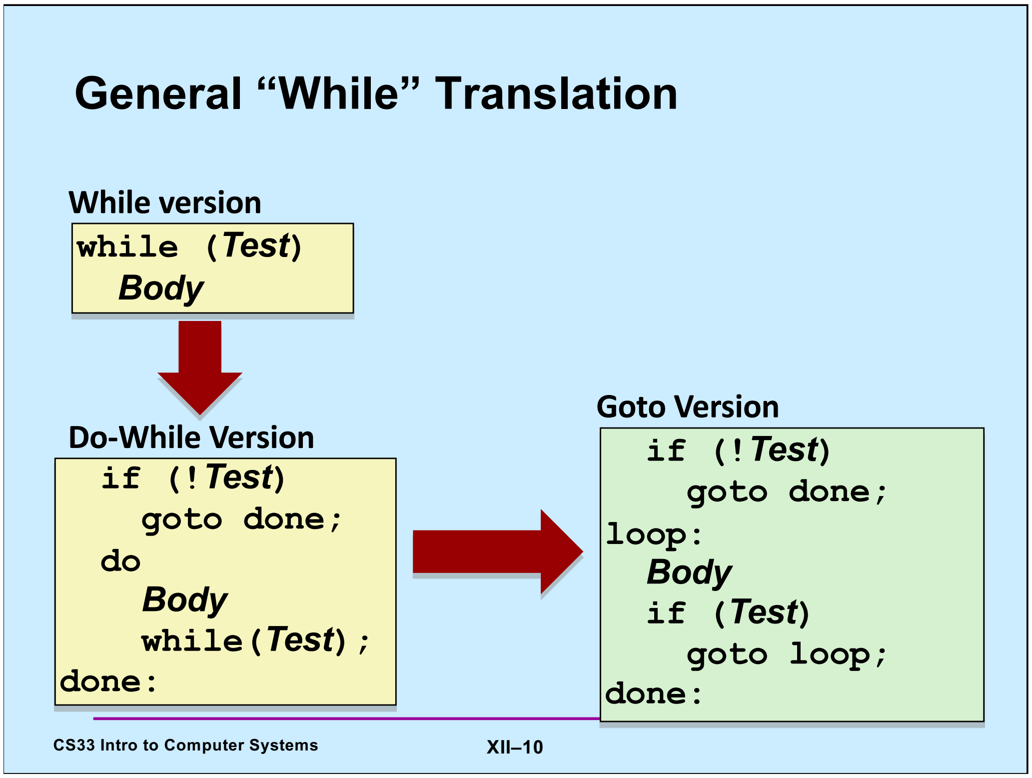
```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Goto Version

```
int pcount_do(unsigned x)  {
  int result = 0;
  if (!x) goto done;
loop:
  result += x & 0x1;
  x >>= 1;
  if (x)
    goto loop;
done:
  return result;
}
```

- **Is this code equivalent to the do-while version?**
  - **must jump out of loop if test fails**

Supplied by CMU.



Supplied by CMU.

“For” Loop Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

- **Is this code equivalent to other versions?**

Supplied by CMU.

"For" Loop Form

General Form

```
for (Init; Test; Update )
    Body
```

```
for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned mask = 1 << i;
  result += (x & mask) != 0;
}
```


Supplied by CMU.

"For" Loop → While Loop

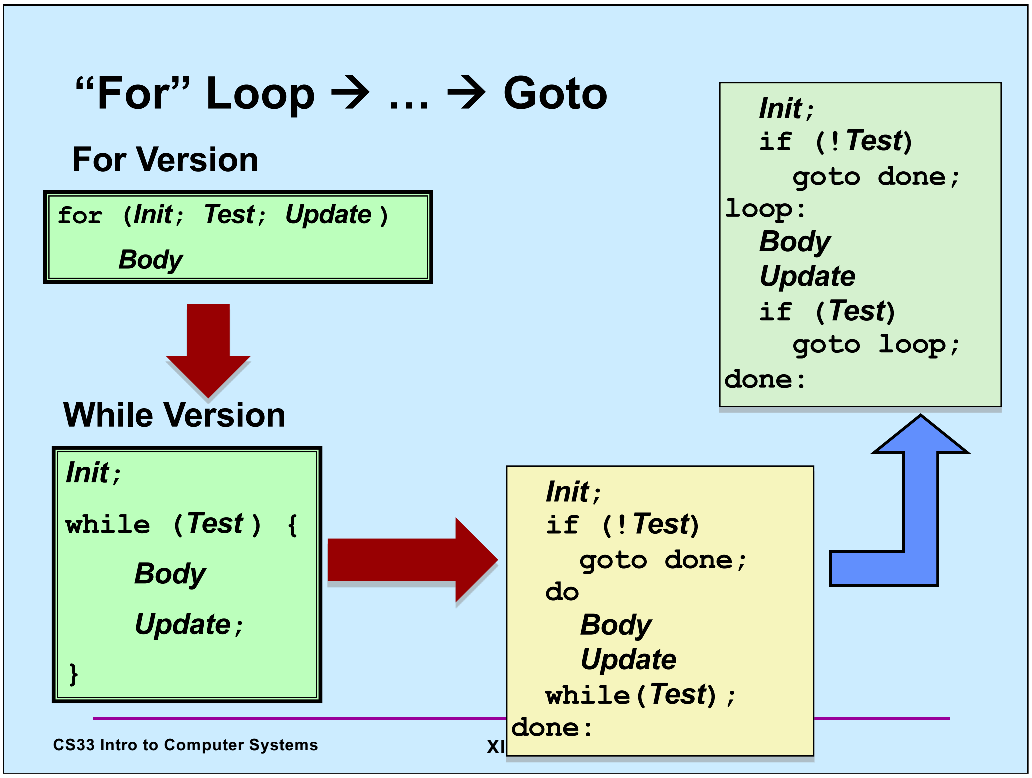
For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```


Supplied by CMU.



Supplied by CMU.

"For" Loop Conversion Example

C Code

```
#define WSIZE 8*sizeof(int)
int pcount_for(unsigned x) {
  int i;
  int result = 0;
  for (i = 0; i < WSIZE; i++) {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  return result;
}
```

Initial test can be optimized away

Goto Version

```
int pcount_for_gt(unsigned x) {
  int i;
  int result = 0;
  i = 0;                       Init
  if (!(i < WSIZE))            !Test
    goto done;
 loop:                         Body
  {
    unsigned mask = 1 << i;
    result += (x & mask) != 0;
  }
  i++;  Update
  if (i < WSIZE)  Test
    goto loop;
 done:
  return result;
}
```

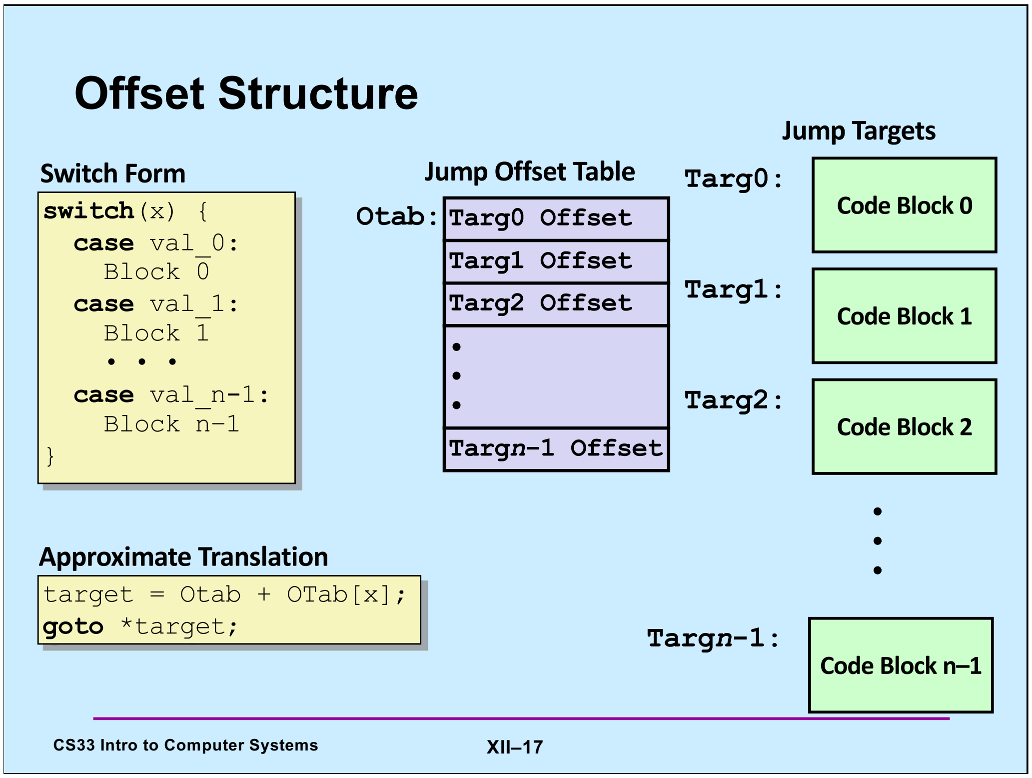
(In the slide, the lines `if (!(i < WSIZE))` and `goto done;` are crossed out.)

Supplied by CMU.

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.



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:                                          .section     .rodata
        movl    $0, %eax                            .align 4
        testq   %rsi, %rsi                  .L4:
        jle     .L1                                 .long   .L8-.L4
        cmpq    $12, %rdi                           .long   .L3-.L4
        ja      .L8                                 .long   .L6-.L4
        leaq    .L4(%rip), %rdx                     .long   .L3-.L4
        movslq  (%rdx,%rdi,4), %rax                 .long   .L5-.L4
        addq    %rdx, %rax                          .long   .L3-.L4
        jmp     *%rax                               .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
```

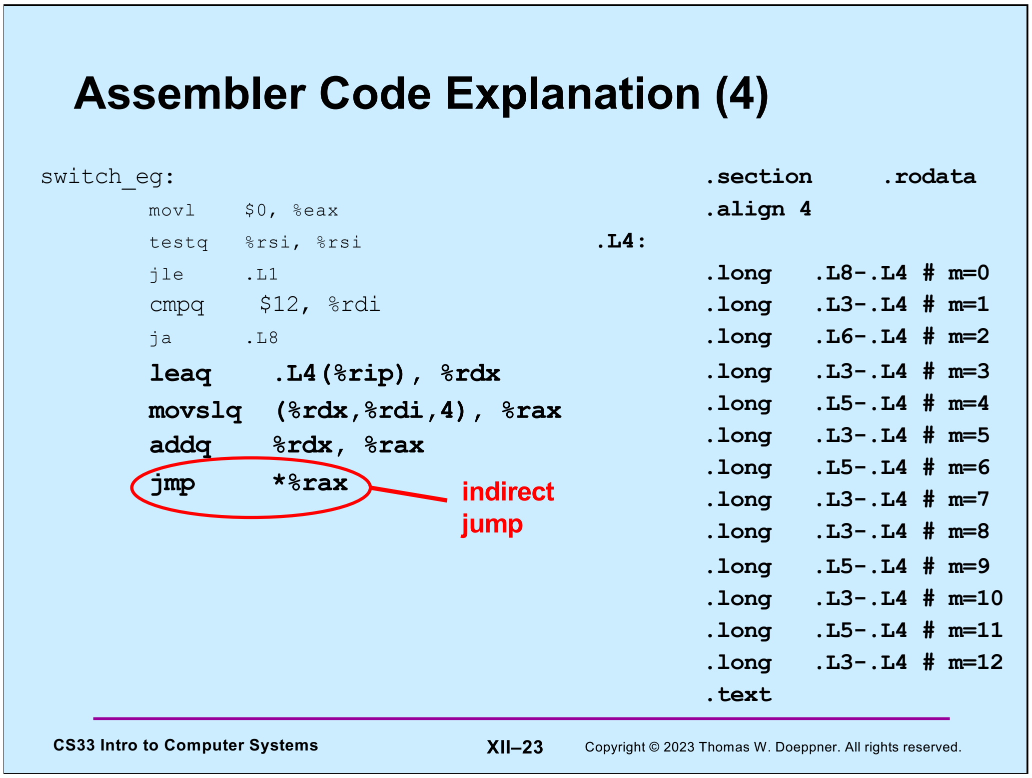
```
         .section     .rodata
         .align 4
.L4:
         .long   .L8-.L4 # m=0
         .long   .L3-.L4 # m=1
         .long   .L6-.L4 # m=2
         .long   .L3-.L4 # m=3
         .long   .L5-.L4 # m=4
         .long   .L3-.L4 # m=5
         .long   .L5-.L4 # m=6
         .long   .L3-.L4 # 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.



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 was compiled with the O1 flag
  - no optimization
- O0 often produces easier-to-read (but less efficient) code
  - not so for switch

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
```

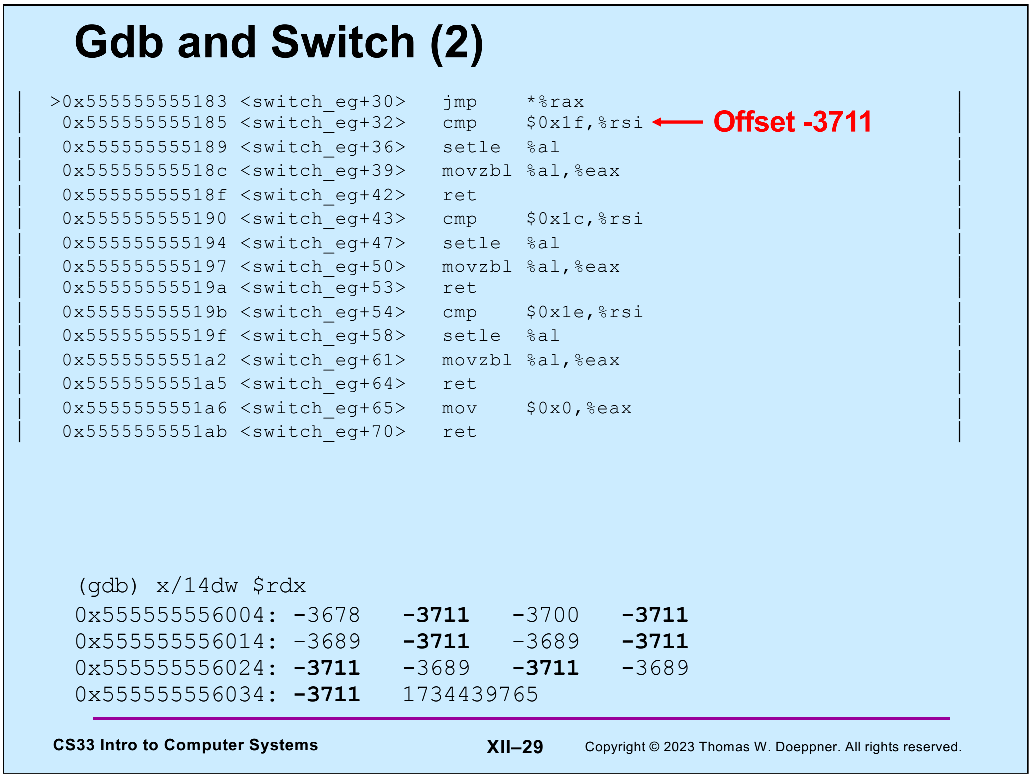

So, now that we know how switch statements are implemented, how might we "reverse engineer" object code to figure out the switch statement it implements?

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 contains 13 entries.



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).

Quiz 1

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
```

a

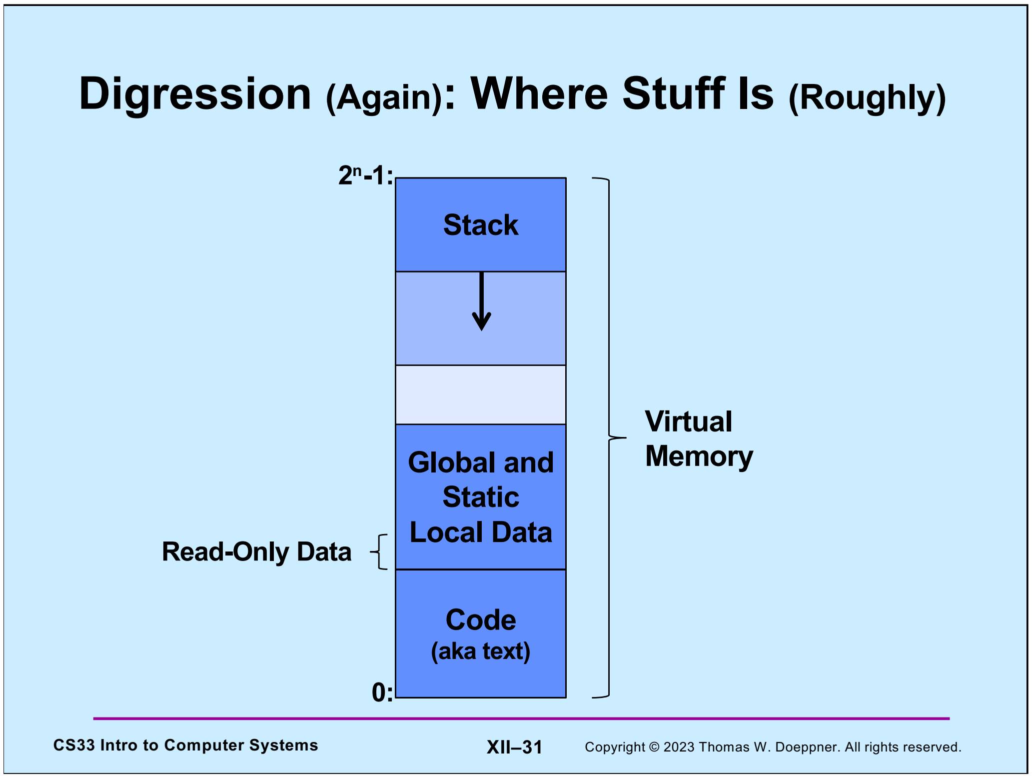
```
long a[10];
void func() {
  long i=0;
  while (i<10)
    a[i]= i++;
}
```

b

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

c

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



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.

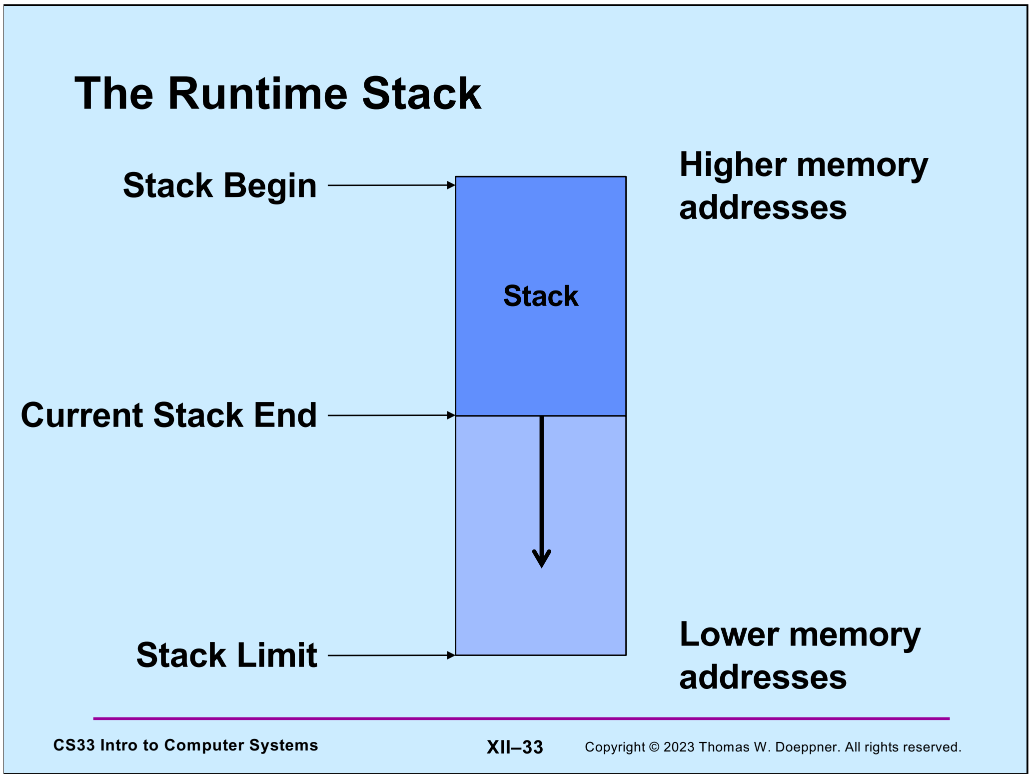
Note that read-only data (such as the offset tables used for switch statements) is placed just above the executable code.

Function Call and Return

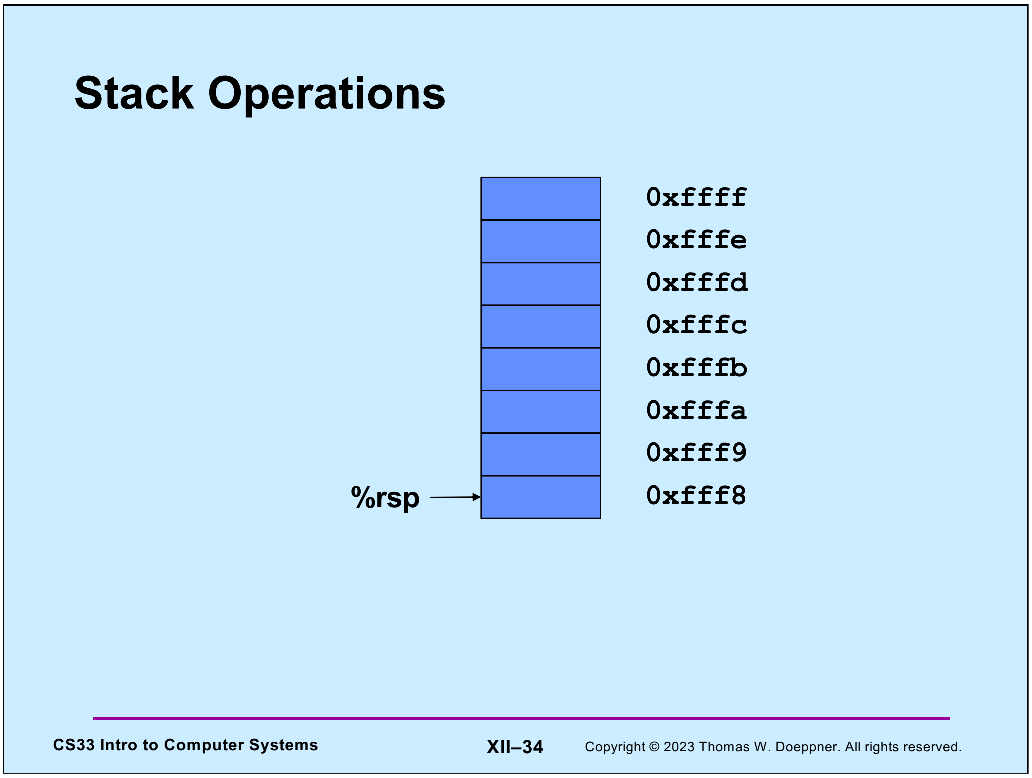
- Function A calls function B
- Function B calls function C

... several million instructions later

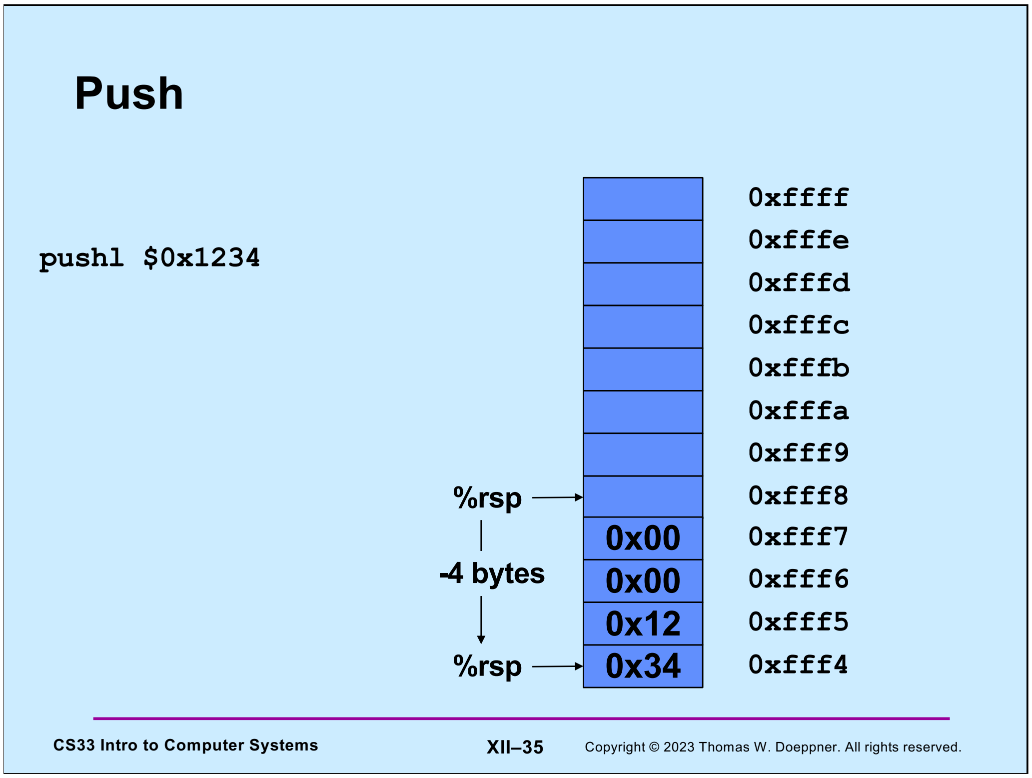
- C returns
  - how does it know to return to B?
- B returns
  - how does it know to return to A?



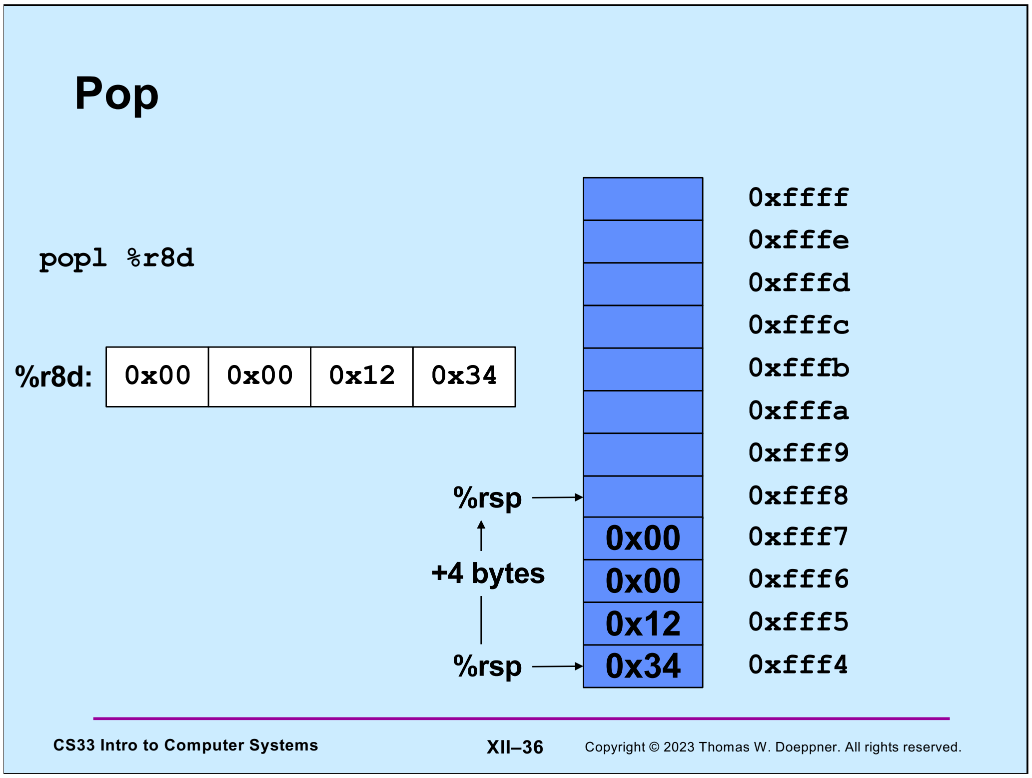
Stacks, as implemented on the X86 for most operating systems (and, in particular, Linux, OSX, and Windows) grow "downwards", from high memory addresses to low memory addresses. To avoid confusion, we will not use the works "top of stack" or "bottom of stack" but will instead use "stack begin" and "current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



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



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



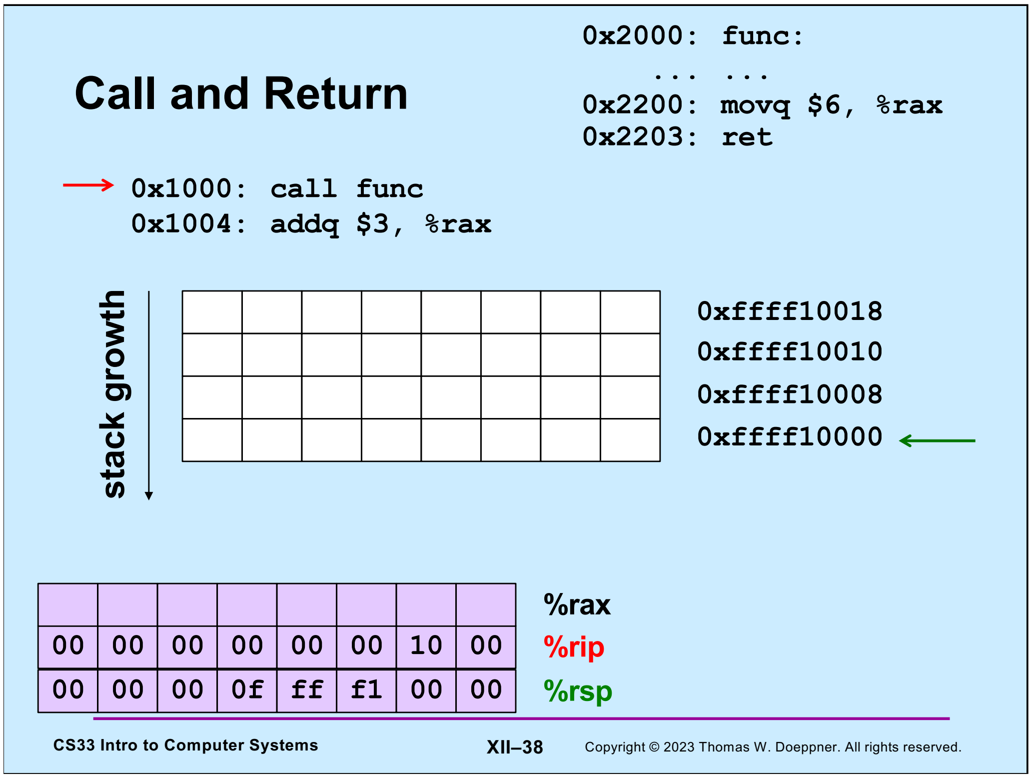
Here we pop an item off the stack. The **popl** instruction copies the 4-byte item pointed to by %rsp into its argument, then increments %rsp by 4.

Call and Return

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

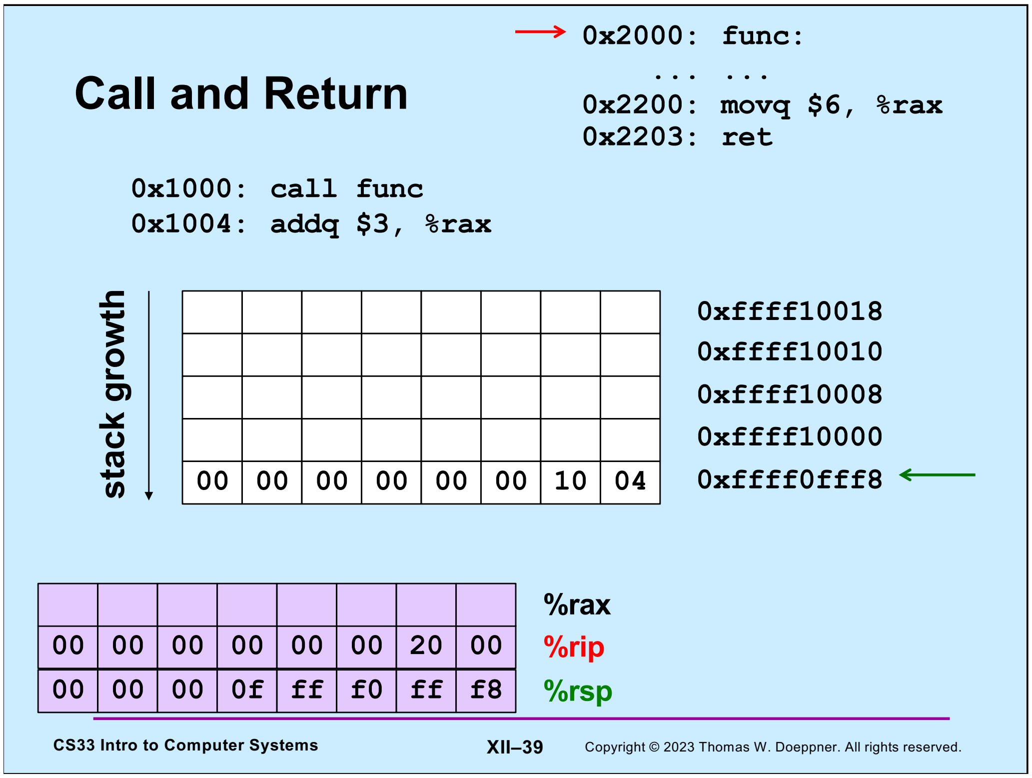
```
0x2000: func:
    ... ...
0x2200: movq $6, %rax
0x2203: ret
```


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

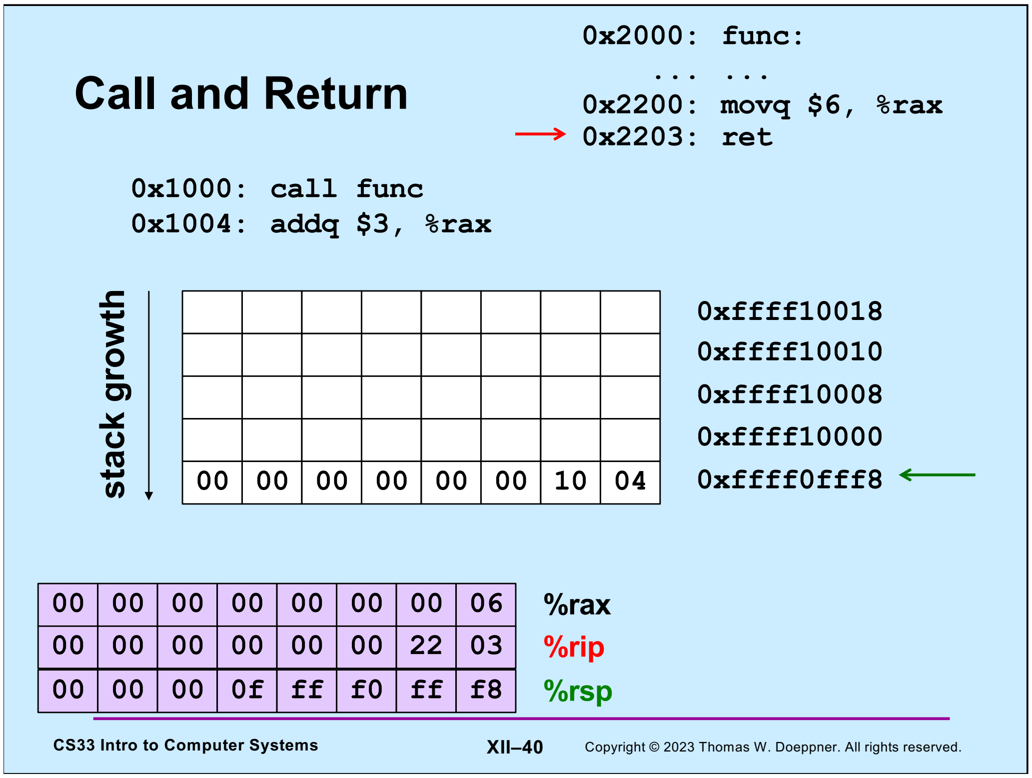


Here we begin walking through what happens during a call and return.

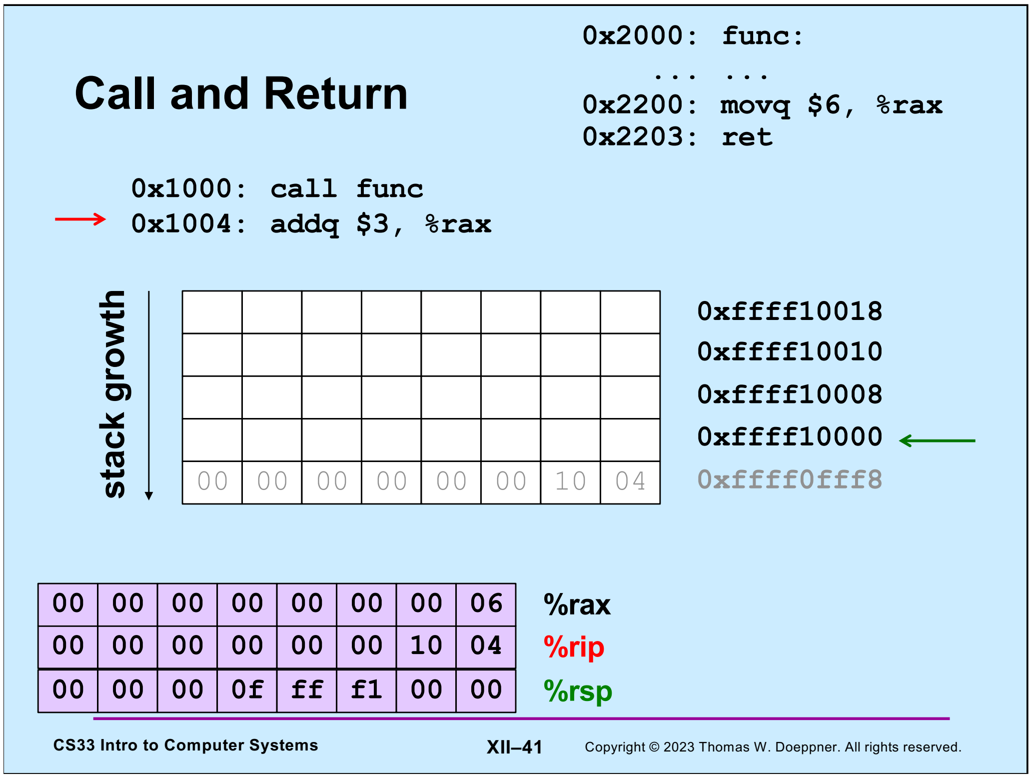
Initially, %rip (the instruction pointer – what it points to is shown with a red arrow pointing to the right) points to the call instruction – thus it's the next instruction to be executed. %rsp (the stack pointer, shown with a green arrow pointing to the left) points to the current end of the stack.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



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



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



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

Arguments and Local Variables (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. 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.

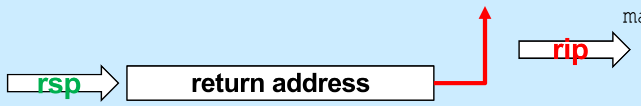
Note, as will be explained, this is not what one would see when compiling it for department computers, on which arguments are passed using registers.

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
```

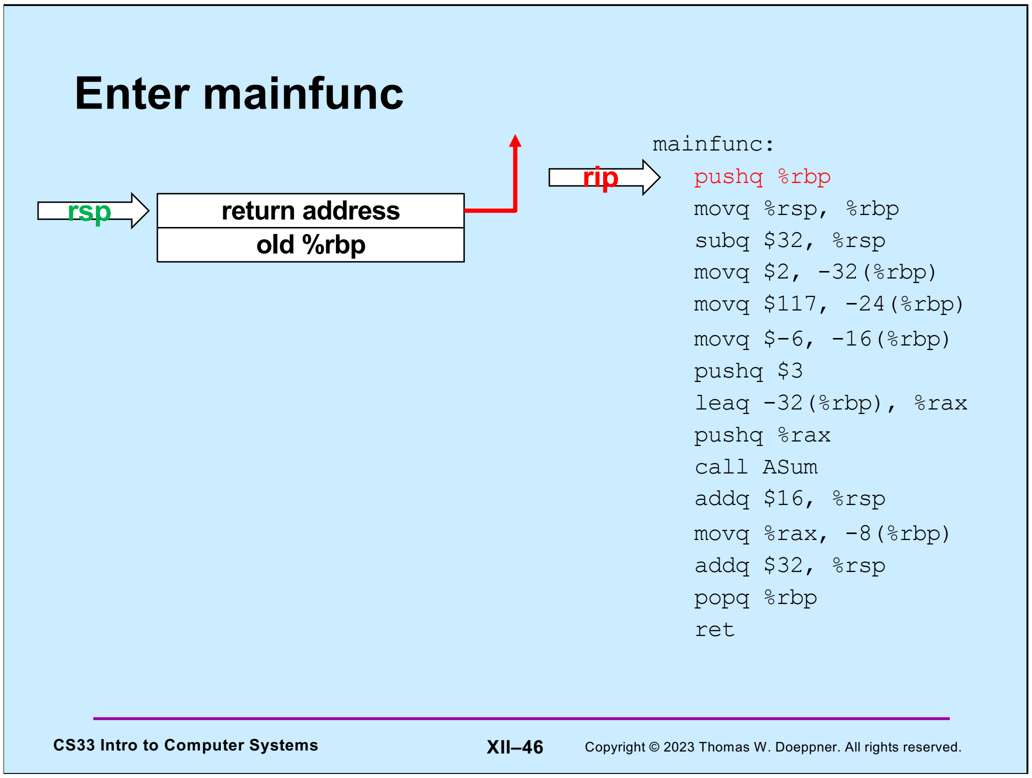

And here is the compiled code for **ASum**. The same caveats as given for the previous slide apply to this one as well.

Enter mainfunc

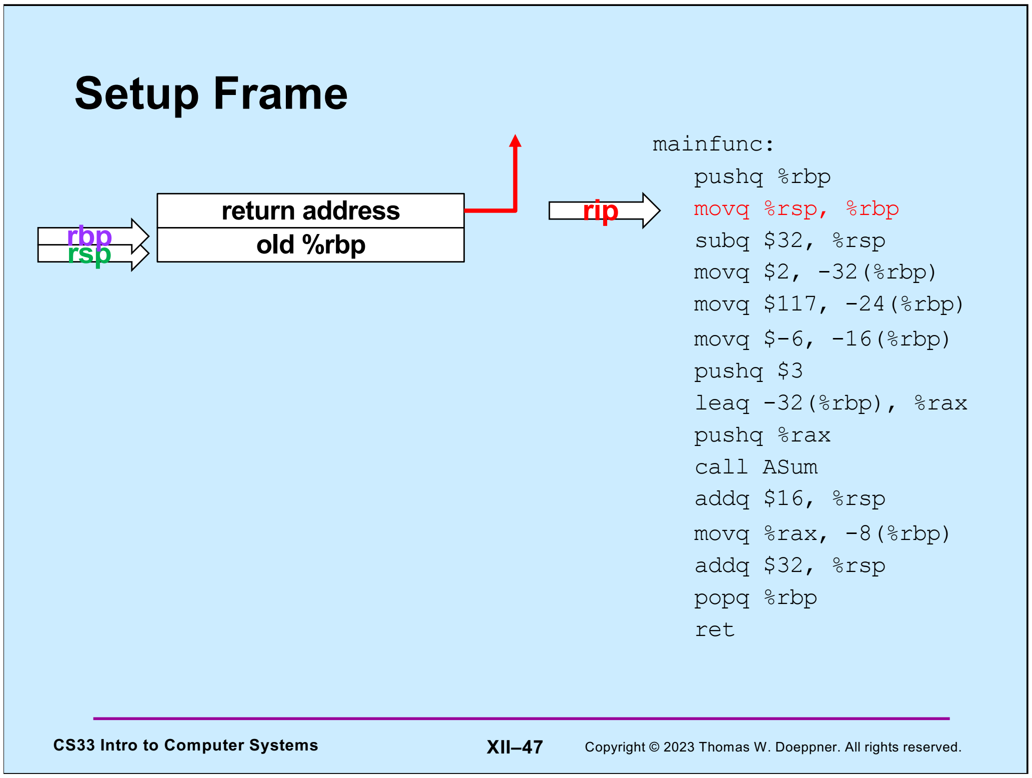



```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

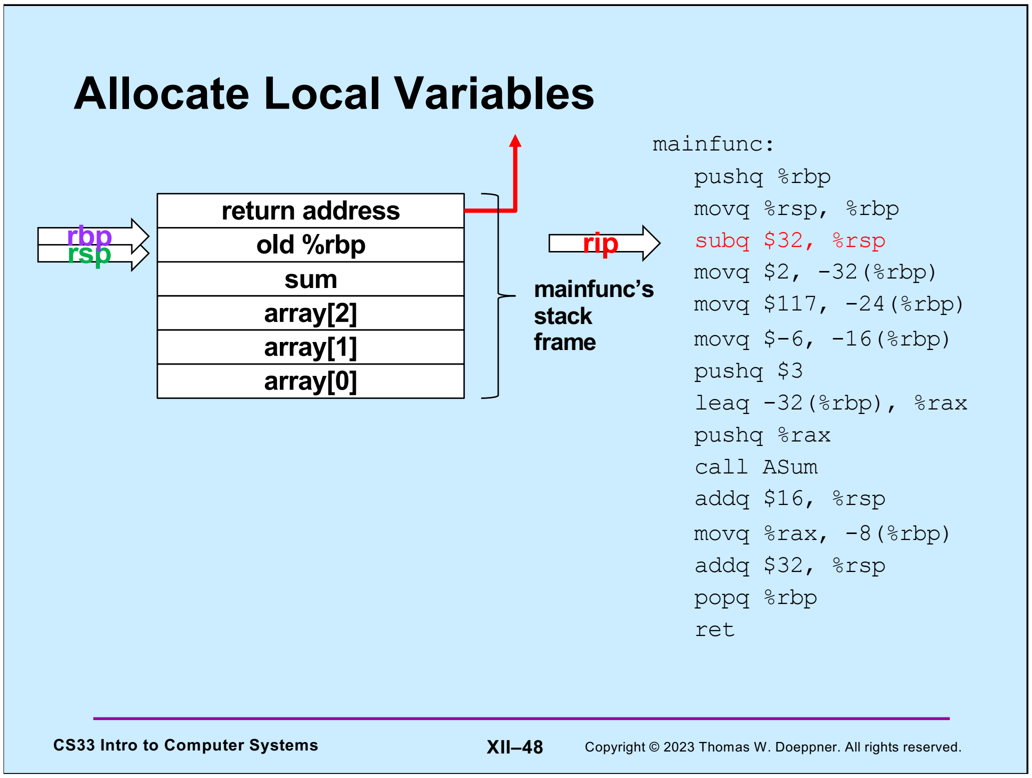

On entry to **mainfunc**, %rsp points to the caller's return address.



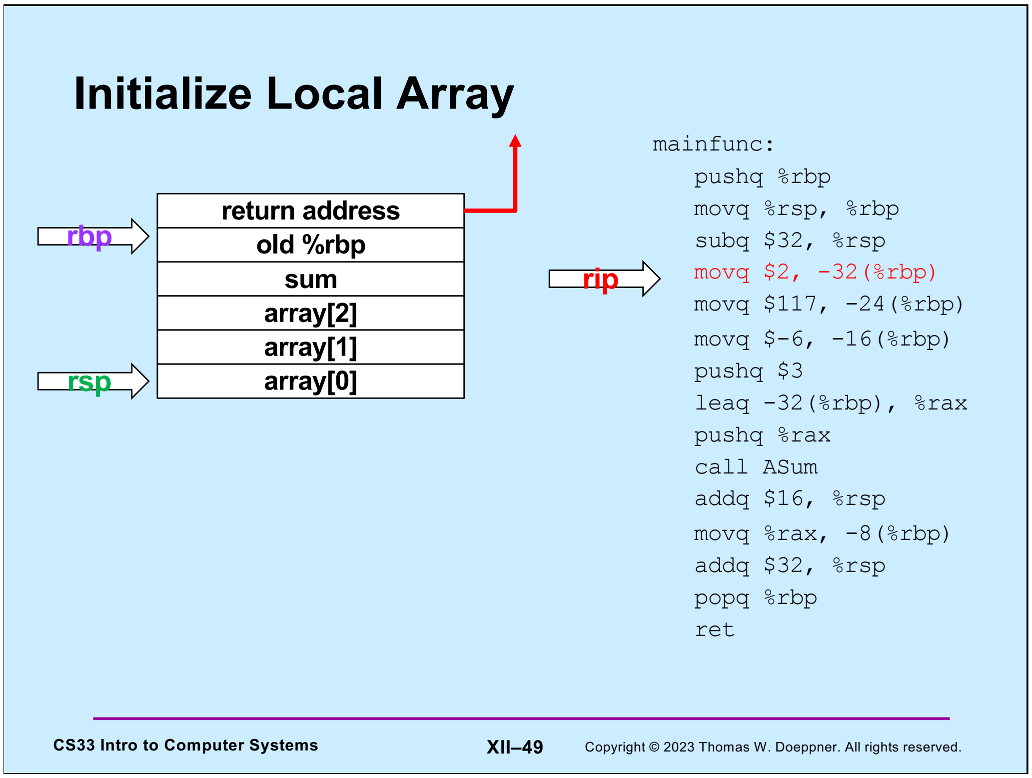
On entry to **mainfunc**, %rsp points to the caller's return address.



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

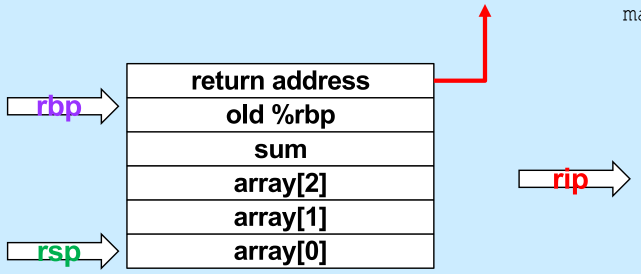


Next, space for **mainfunc**'s local variables is allocated on the stack by decrementing %rsp by their total size (32 bytes). At this point we have **mainfunc**'s stack frame in place.



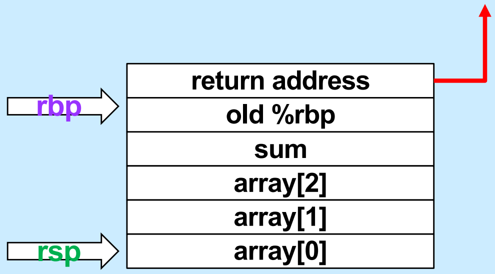
ASum now initializes the stack space containing its local variables.

Initialize Local Array



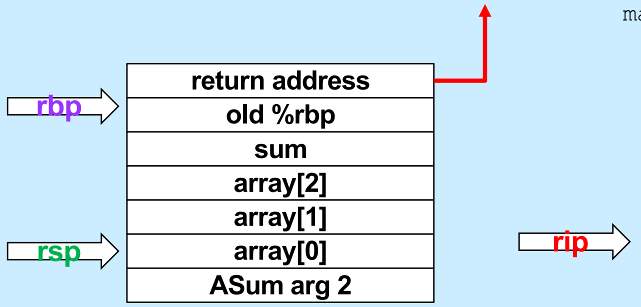

```
mainfunc:
   pushq %rbp
   movq %rsp, %rbp
   subq $32, %rsp
   movq $2, -32(%rbp)
   movq $117, -24(%rbp)
   movq $-6, -16(%rbp)
   pushq $3
   leaq -32(%rbp), %rax
   pushq %rax
   call ASum
   addq $16, %rsp
   movq %rax, -8(%rbp)
   addq $32, %rsp
   popq %rbp
   ret
```

Initialize Local Array




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

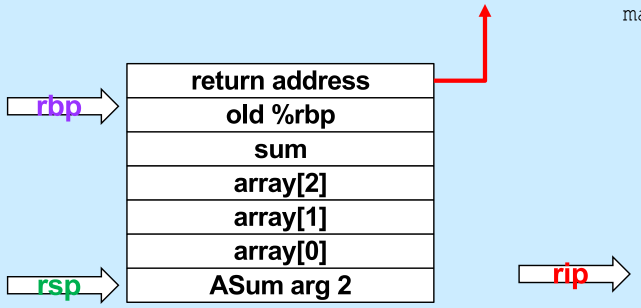
Push Second Argument




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

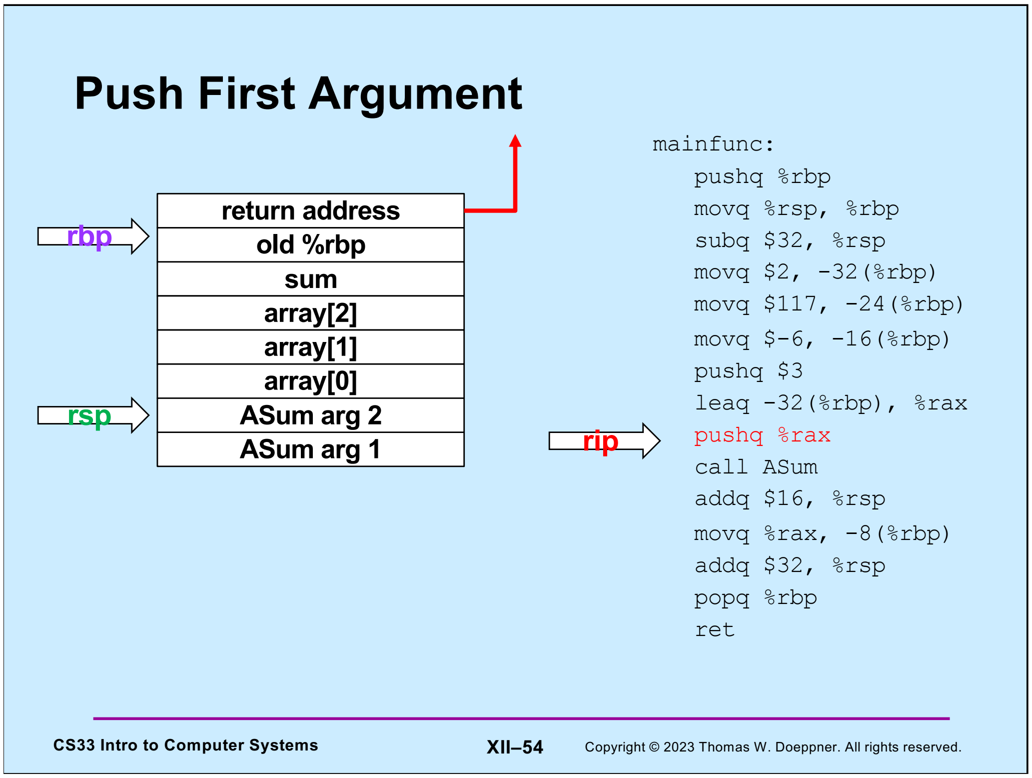

The second argument (3) to **ASum** is pushed onto the stack.

Get Array Address

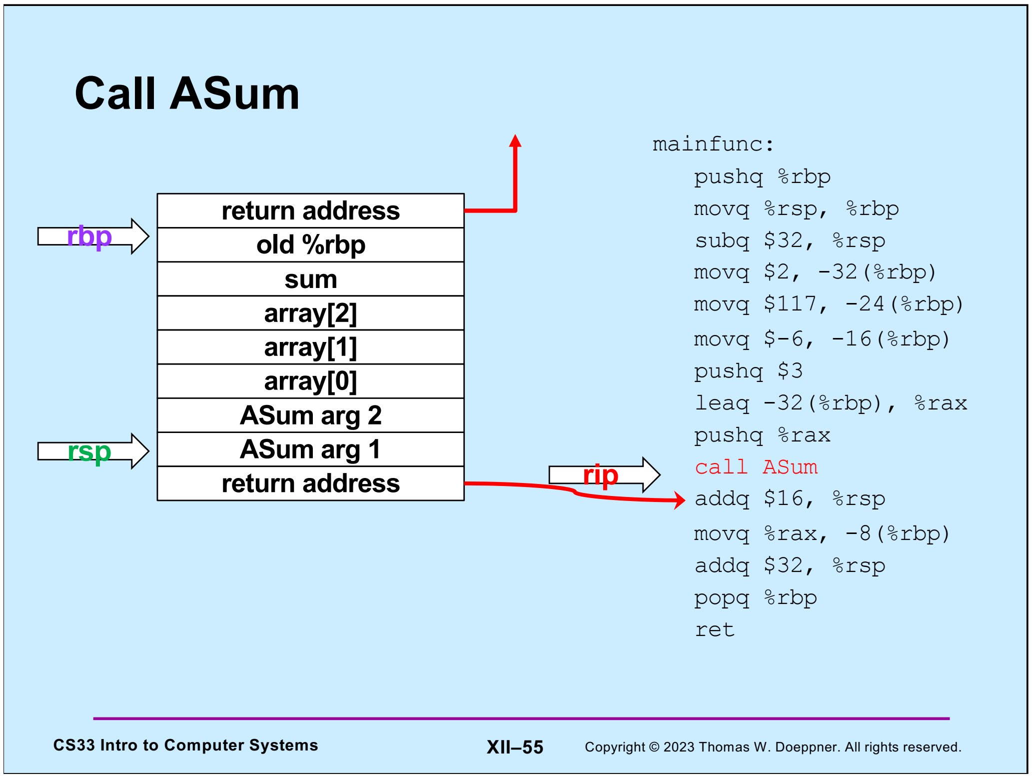



```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```


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

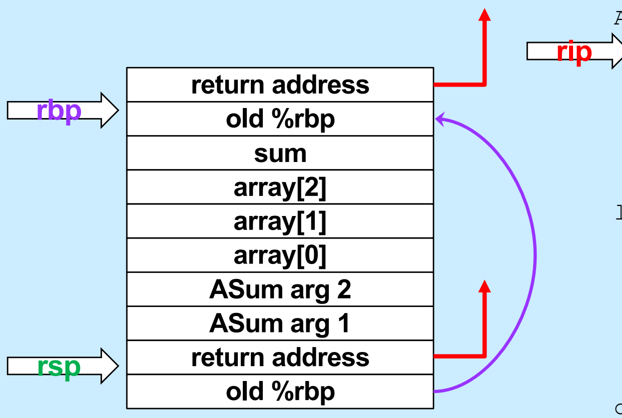


And finally, the address of the array is pushed onto the stack as **ASum**'s first argument.



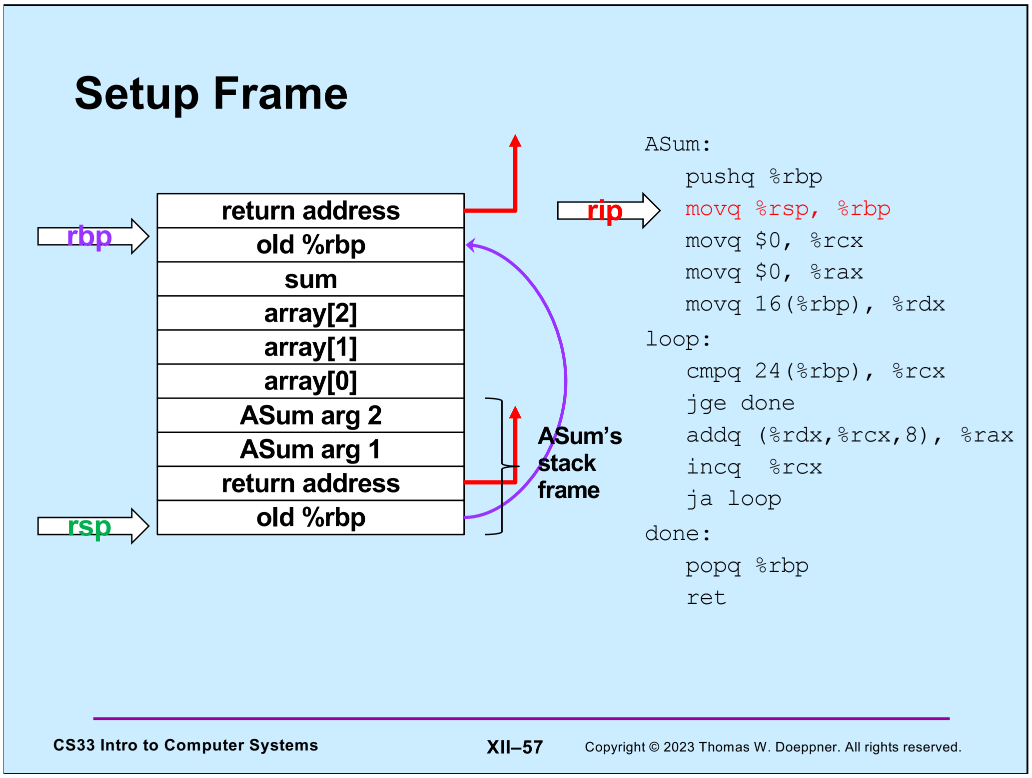
mainfunc now calls **ASum**, pushing its return address onto the stack.

Enter ASum



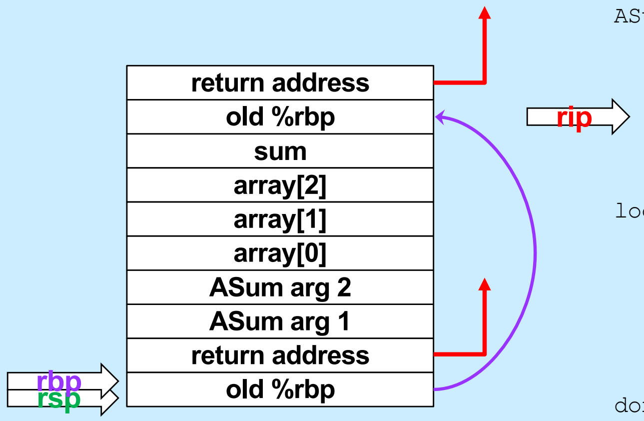

```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```


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



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

Execute the Function




```
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 2

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.