CS 33

Machine Programming (4)

 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

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

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

x in %edi

y in %esi

Supplied by CMU, but converted to x86-64.

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
    cmpl   %esi, %edi        Body1
    jle    .L6
    subl   %eax, %edi
    movl   %edi, %eax        Body2a
    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.

General Conditional-Expression Translation

C Code

```
val = Test ? Then_Expr : Else_Expr;
```

```
val = x>y ? x-y : y-x;
```

- Test is expression returning integer
  - == 0 interpreted as false
  - ≠ 0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```


Supplied by CMU.

“Do-While” Loop Example

C Code

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

Goto Version

```c
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);
```

Goto Version

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

- **Body:**

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

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

Supplied by CMU.

"While" Loop Example

C Code

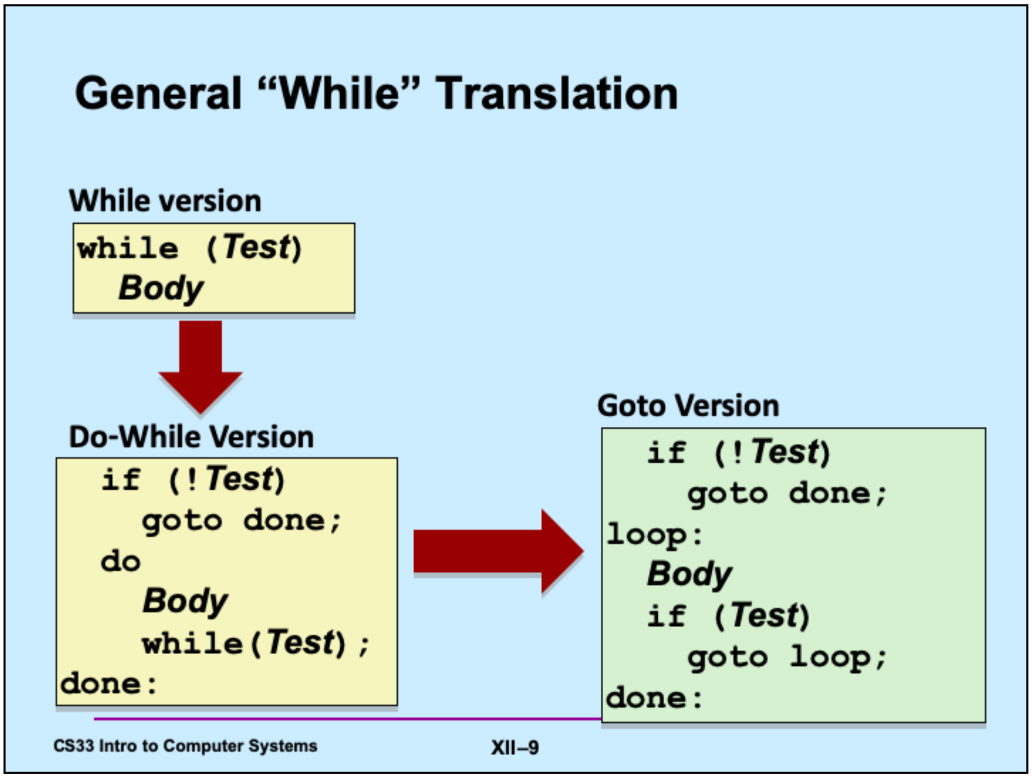
```
int pcount_while(unsigned x) {
  int result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

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.

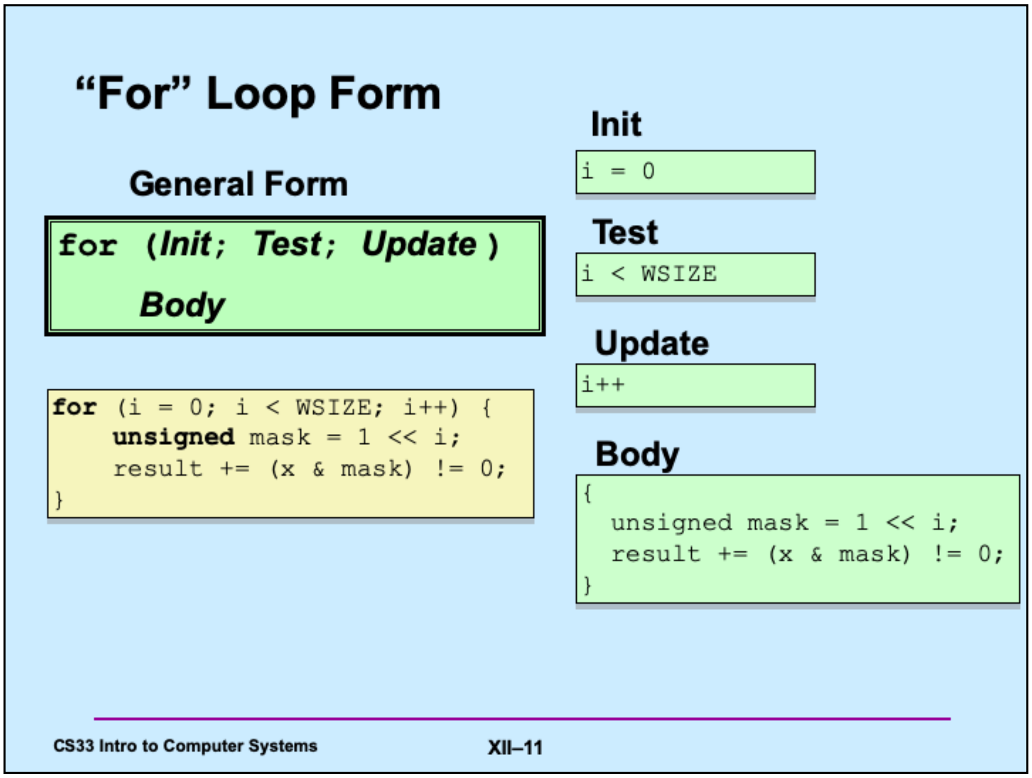
“For” Loop Example

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.



Supplied by CMU.

“For” Loop → While Loop

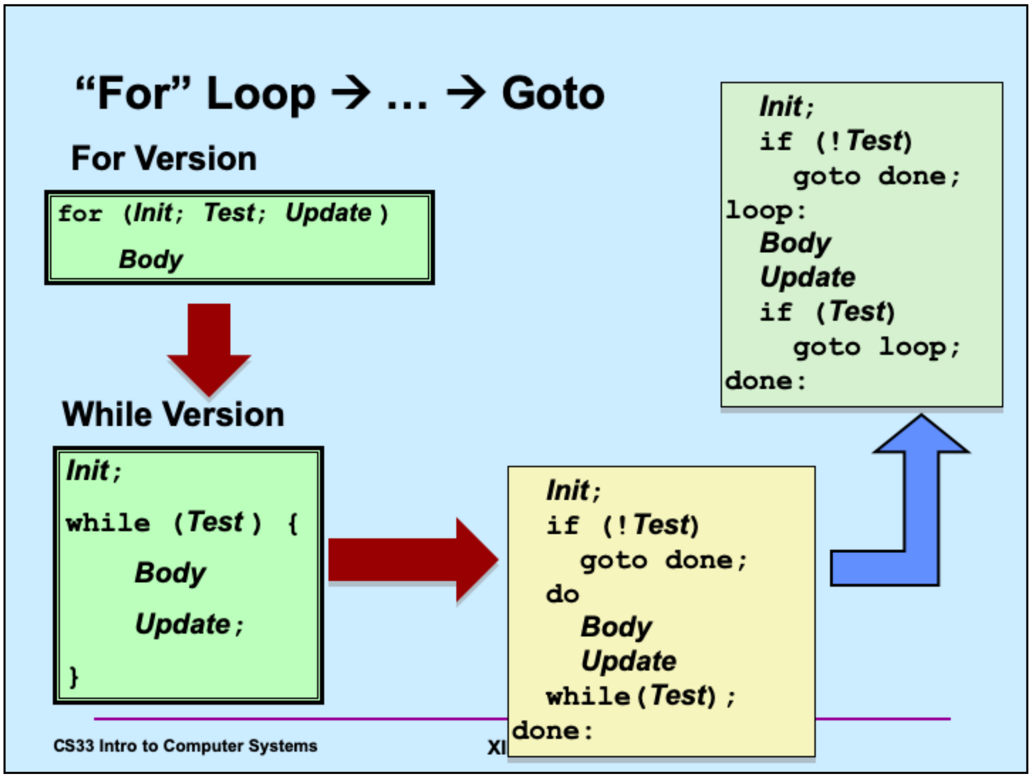
For Version

```
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;
}
```

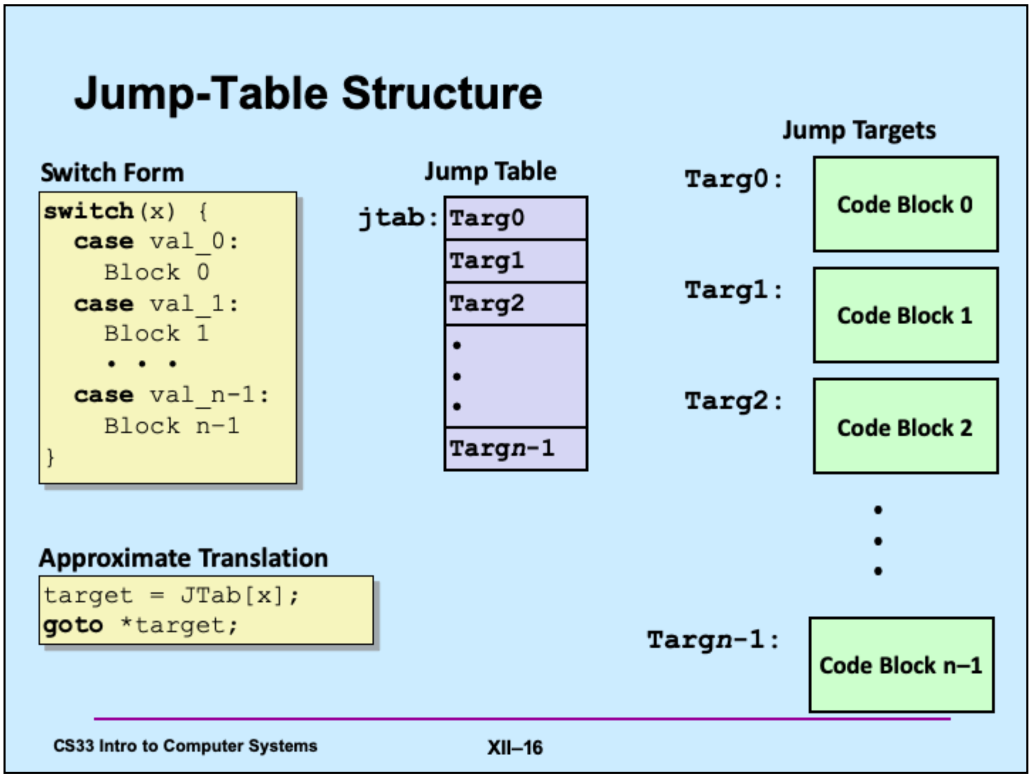

Supplied by CMU.

Switch-Statement Example

```
long switch_eg
   (long x, long y, long z) {
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

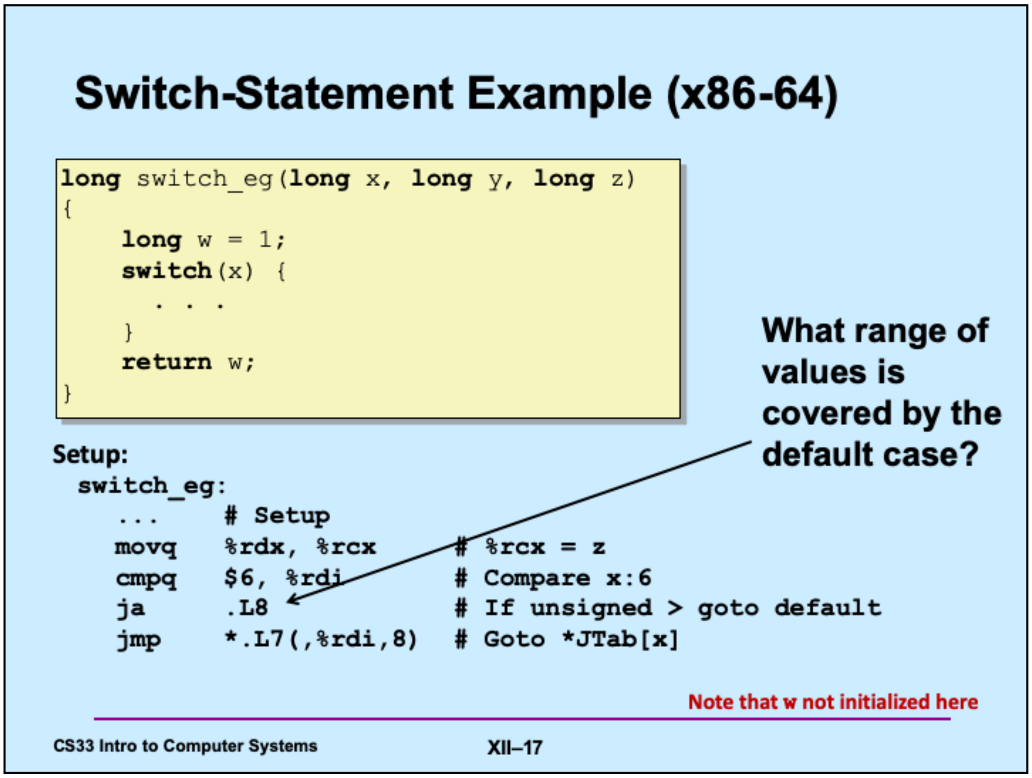
- **Multiple case labels**
  - here: 5 & 6
- **Fall-through cases**
  - here: 2
- **Missing cases**
  - here: 4

Supplied by CMU.



Supplied by CMU.

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!



Supplied by CMU, but converted to x86-64.

Note that the *ja* in the slide causes a jump to occur if the previous comparison is interpreted as being performed on unsigned values, and the result is that x is greater than (above) 6. Given that x is declared to be a *signed* value, for what range of values of x will *ja* cause a jump to take place?

Note that the assembler code shown in the examples was produced by compiling the C code using gcc with the “-O1” flag.

Switch-Statement Example

```c
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section    .rodata
  .align 4
.L7:
  .quad     .L8  # x = 0
  .quad     .L3  # x = 1
  .quad     .L4  # x = 2
  .quad     .L9  # x = 3
  .quad     .L8  # x = 4
  .quad     .L6  # x = 5
  .quad     .L6  # x = 6
```

Setup:

```
switch_eg:
  . . .     # Setup
  movq      %rdx, %rcx      # %rcx = z
  cmpq      $6, %rdi        # Compare x:6
  ja        .L8             # If unsigned > goto default
  jmp       *.L7(,%rdi,8)   # Goto *JTab[x]
```

Indirect jump

Supplied by CMU, but converted to x86-64.

Assembly-Setup Explanation

- **Table structure**
  - – each target requires 8 bytes
  - – base address at `.L7`

Jump table

```
.section   .rodata
  .align 4
.L7:
  .quad    .L8 # x = 0
  .quad    .L3 # x = 1
  .quad    .L4 # x = 2
  .quad    .L9 # x = 3
  .quad    .L8 # x = 4
  .quad    .L6 # x = 5
  .quad    .L6 # x = 6
```

- **Jumping**

  direct: `jmp .L8`
  - – jump target is denoted by label `.L8`

  indirect: `jmp *.L7(,%rdi,8)`
  - – start of jump table: `.L7`
  - – must scale by factor of 8 (labels have 8 bytes on x86-64)
  - – fetch target from effective address `.L7 + rdi*8`
    - » only for $0 \le x \le 6$

CS33 Intro to Computer Systems XII–19

Supplied by CMU, but converted to x86-64.

The *jmp* instruction is doing a couple things that require explanation: The asterisk means it's an *indirect jump* (such indirection is allowed only in jumps). The address specified after the asterisk is the address of an entry in the *jump table*. The asterisk means, rather than jumping directly to that entry, jump to the address that's in that table entry. ".L7" is a label that's being used as a displacement in the address computation. The value of .L7 is the address of the area of memory it labels. In this case, it's the address of the jump table. Thus, an unconditional jump is to take place to the address contained in the 8-byte entry of the jump table indexed by the contents of %rdi. Thus, if %rdi is, say, 2, then a jump will take place to address in the location starting 16 bytes beyond the beginning of the table. This will be a jump to .L4. .L4 itself is a label of code specified elsewhere, the reference to the label is replaced by the assembler with the address of the code labelled with .L4.

The jump table is separate from the code (it's not executable). This is specified by the ".section" directive, which also specifies that it should be placed in memory that's made read-only (".rodata" indicates this). The ".align 4" says that the address of the start of the table should be divisible by four (why this is important is something we'll get to in a week or two).

Jump Table

Jump table

```
.section   .rodata
  .align 4
.L7:
  .quad     .L8 # x = 0
  .quad     .L3 # x = 1
  .quad     .L4 # x = 2
  .quad     .L9 # x = 3
  .quad     .L8 # x = 4
  .quad     .L6 # x = 5
  .quad     .L6 # x = 6
```

```
switch(x) {
case 1:        // .L3
    w = y*z;
    break;
case 2:        // .L4
    w = y/z;
    /* Fall Through */
case 3:        // .L9
    w += z;
    break;
case 5:
case 6:        // .L6
    w -= z;
    break;
default:       // .L8
    w = 2;
}
```

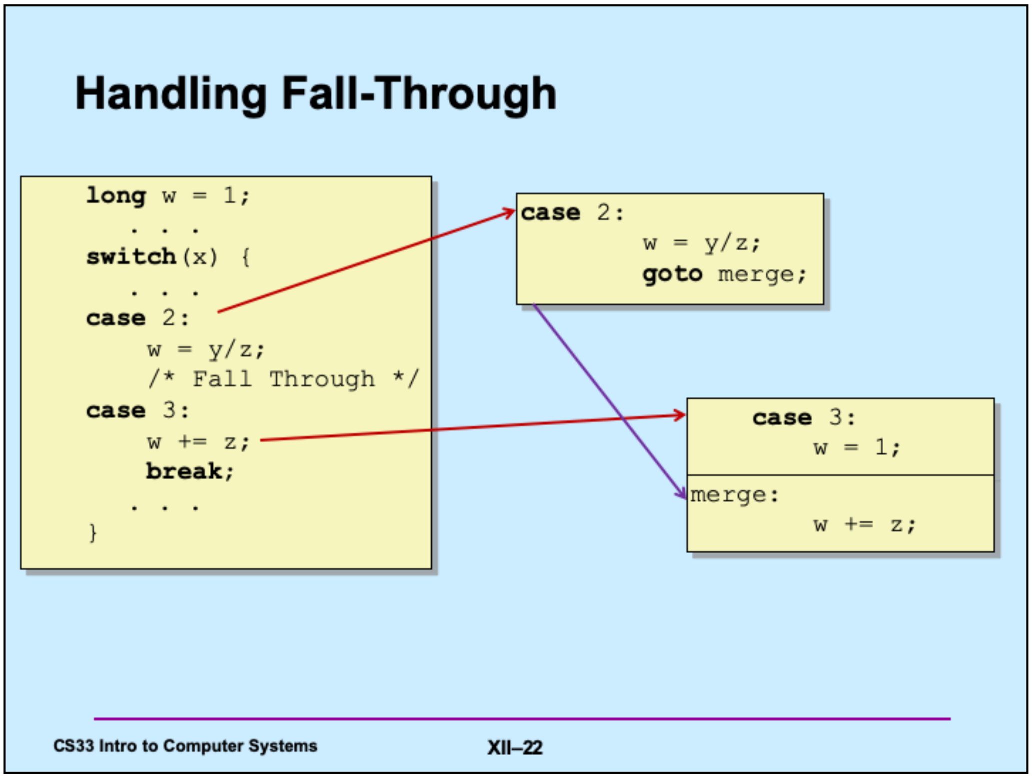

Supplied by CMU, but converted to x86-64.

Code Blocks (Partial)

```
switch(x) {
case 1:       // .L3
      w = y*z;
      break;
 . . .
case 5:       // .L6
case 6:       // .L6
    w -= z;
    break;
default:      // .L8
    w = 2;
}
```

```
.L3:    # x == 1
  movl  %rsi, %rax   # y
  imulq %rdx, %rax   # w = y*z
  ret
.L6:               # x == 5, x == 6
  movl  $1, %eax # w = 1
  subq  %rdx, %rax   # w -= z
  ret
.L8:               # Default
  movl  $2, %eax # w = 2
  ret
```


Supplied by CMU, but converted to x86-64.



Supplied by CMU, but converted to x86-64.

Code Blocks (Rest)

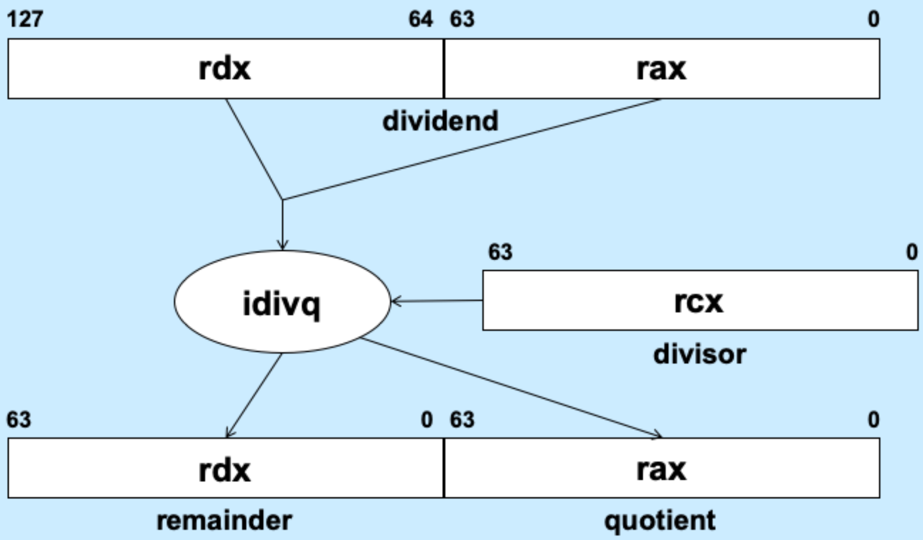
```
switch(x) {
   . . .
   case 2:  // .L4
       w = y/z;
       /* Fall Through */
   case 3:  // .L9
       w += z;
       break;
   . . .
}
```

```
.L4:    # x == 2
  movq  %rsi, %rax
  movq  %rsi, %rdx
  sarq  $63, %rdx
  idivq %rcx      # w = y+z
  jmp   .L5
.L9:    # x == 3
  movl  $1, %eax # w = 1
.L5:    # merge:
  addq  %rcx, %rax # w += z
  ret
```


Supplied by CMU, but converted to x86-64.

The code following the .L4 label requires some explanation. The *idivq* instruction is special in that it takes a 128-bit dividend that is implicitly assumed to reside in registers *rdx* and *rax*. Its single operand specifies the divisor. The quotient is always placed in the *rax* register, and the remainder in the *rdx* register. In our example, *y*, which we want to be the dividend, is copied into both the *rax* and *rdx* registers. The *sarq* (shift arithmetic right quadword) instruction propagates the sign bit of *rdx* across the entire register, replacing its original contents. Thus, if one considers *rdx* to contain the most-significant bits of the dividend and *rax* to contain the least-significant bits, the pair of registers now contains the 128-bit version of *y*. The *idivq* instruction computes the quotient from dividing this 128-bit value by the 64-bit value contained in register *rcx* (containing *z*). The quotient is stored register *rax* (implicitly) and the remainder is stored in register *rdx* (and is ignored in our example). This illustrated in the next slide.

idivq



 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

x86-64 Object Code

- **Setup**
  - label `.L8` becomes address `0x4004e5`
  - label `.L7` becomes address `0x4005c0`

Assembly code

```
switch_eg:
  . . .
  ja     .L8           # If unsigned > goto default
  jmp    *.L7(,%rdi,8) # Goto *JTab[x]
```

Disassembled object code

```
00000000004004ac <switch_eg>:
 . . .
 4004b3: 77 30          ja     4004e5 <switch_eg+0x39>
 4004b5: ff 24 fd c0 05 40 00 jmpq    *0x4005c0(,%rdi,8)
```


Supplied by CMU, but converted to x86-64.

Disassembly was accomplished using “objdump –d”. Note that the text enclosed in angle brackets (“<“, “>”) is essentially a comment, relating the address (4004e5) to a symbolic location (0x39 bytes after the beginning of *switch_eg*).

x86-64 Object Code (cont.)

- **Jump table**
  - doesn't show up in disassembled code
  - can inspect using gdb

```
gdb switch
(gdb) x/7xg 0x4005c0
```

  - » examine 7 hexadecimal format "giant" words (8-bytes each)
  - » use command "`help x`" to get format documentation

```
0x4005c0:    0x00000000004004e5    0x00000000004004bc
0x4005d0:    0x00000000004004c4    0x00000000004004d3
0x4005e0:    0x00000000004004e5    0x00000000004004dc
0x4005f0:    0x00000000004004dc
```


Supplied by CMU, but converted to x86-64. We assume that the switch_eg function was included in a program whose name is *switch*. Hence, gdb is invoked from the shell with the argument "switch".

x86-64 Object Code (cont.)

- **Deciphering jump table**

```
0x4005c0:        0x00000000004004e5        0x00000000004004bc
0x4005d0:        0x00000000004004c4        0x00000000004004d3
0x4005e0:        0x00000000004004e5        0x00000000004004dc
0x4005f0:        0x00000000004004dc
```

Address	Value	x
0x4005c0	0x4004e5	0
0x4005c8	0x4004bc	1
0x4005d0	0x4004c4	2
0x4005d8	0x4004d3	3
0x4005e0	0x4004e5	4
0x4005e8	0x4004dc	5
0x4005f0	0x4004dc	6

Supplied by CMU, but converted to x86-64.

Disassembled Targets

```
(gdb) disassemble 0x4004bc,0x4004eb
 Dump of assembler code from 0x4004bc to 0x4004eb
    0x00000000004004bc <switch_eg+16>:   mov    %rsi,%rax
    0x00000000004004bf <switch_eg+19>:   imul   %rdx,%rax
    0x00000000004004c3 <switch_eg+23>:   retq
    0x00000000004004c4 <switch_eg+24>:   mov    %rsi,%rax
    0x00000000004004c7 <switch_eg+27>:   mov    %rsi,%rdx
    0x00000000004004ca <switch_eg+30>:   sar    $0x3f,%rdx
    0x00000000004004ce <switch_eg+34>:   idiv   %rcx
    0x00000000004004d1 <switch_eg+37>:   jmp    0x4004d8 <switch_eg+44>
    0x00000000004004d3 <switch_eg+39>:   mov    $0x1,%eax
    0x00000000004004d8 <switch_eg+44>:   add    %rcx,%rax
    0x00000000004004db <switch_eg+47>:   retq
    0x00000000004004dc <switch_eg+48>:   mov    $0x1,%eax
    0x00000000004004e1 <switch_eg+53>:   sub    %rdx,%rax
    0x00000000004004e4 <switch_eg+56>:   retq
    0x00000000004004e5 <switch_eg+57>:   mov    $0x2,%eax
    0x00000000004004ea <switch_eg+62>:   retq
```

 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Matching Disassembled Targets

Value	x
0x4004e5	0
0x4004bc	1
0x4004c4	2
0x4004d3	3
0x4004e5	4
0x4004dc	5
0x4004dc	6

```
0x00000000004004bc:   mov    %rsi,%rax
0x00000000004004bf:   imul   %rdx,%rax
0x00000000004004c3:   retq
0x00000000004004c4:   mov    %rsi,%rax
0x00000000004004c7:   mov    %rsi,%rdx
0x00000000004004ca:   sar    $0x3f,%rdx
0x00000000004004ce:   idiv   %rcx
0x00000000004004d1:   jmp    0x4004d8
0x00000000004004d3:   mov    $0x1,%eax
0x00000000004004d8:   add    %rcx,%rax
0x00000000004004db:   retq
0x00000000004004dc:   mov    $0x1,%eax
0x00000000004004e1:   sub    %rdx,%rax
0x00000000004004e4:   retq
0x00000000004004e5:   mov    $0x2,%eax
0x00000000004004ea:   retq
```


Copyright © 2019 Thomas W. Doeppner. All rights reserved.

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
        jne     .L2
        ret
```

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

a

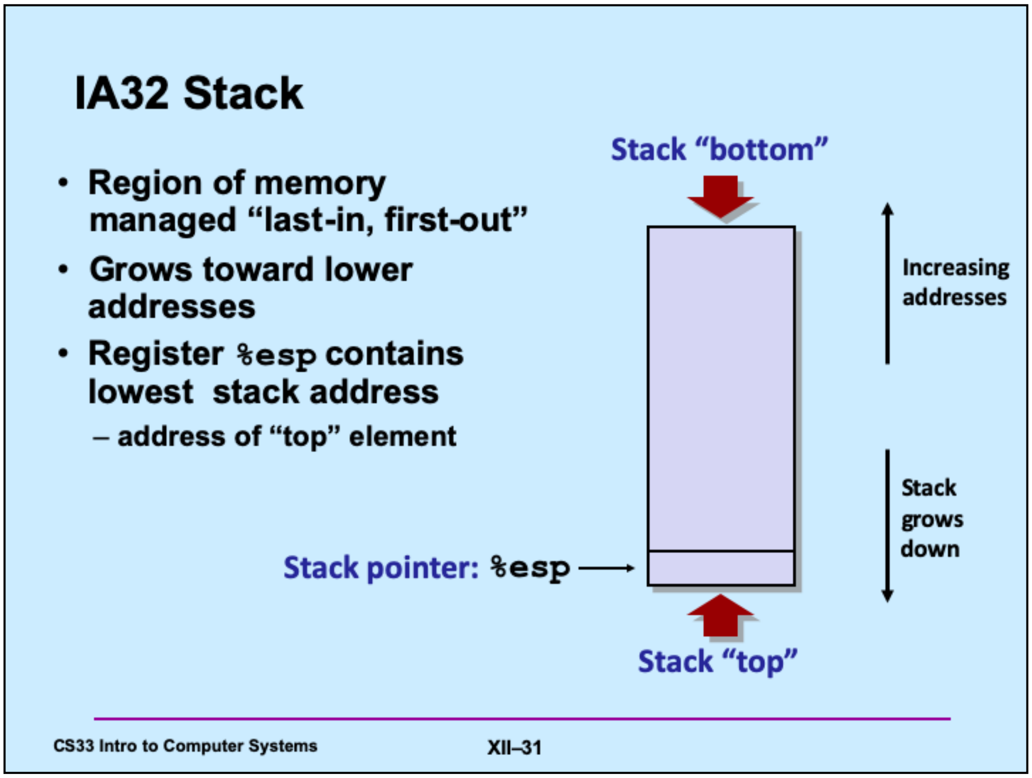
```
long a[10];
void func() {
  long i=0;
  while (i<10)
    a[i]= i++;
}
```

b

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

c

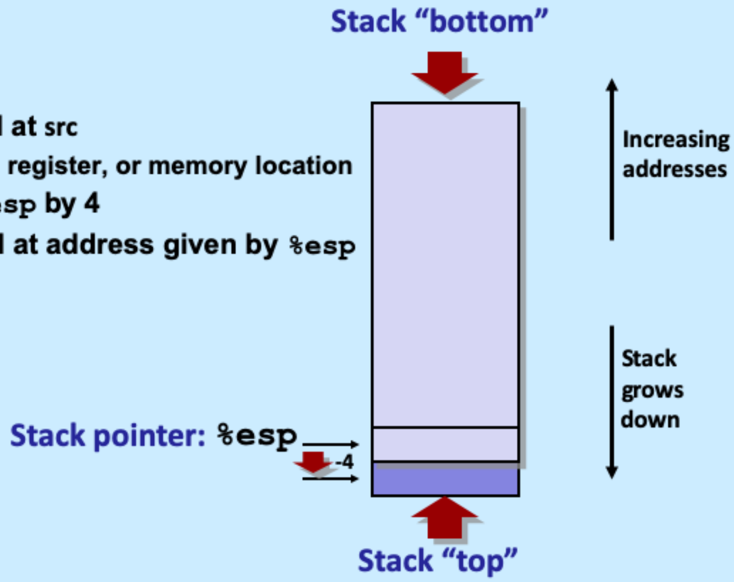
 Copyright © 2019 Thomas W. Doeppner. All rights reserved.



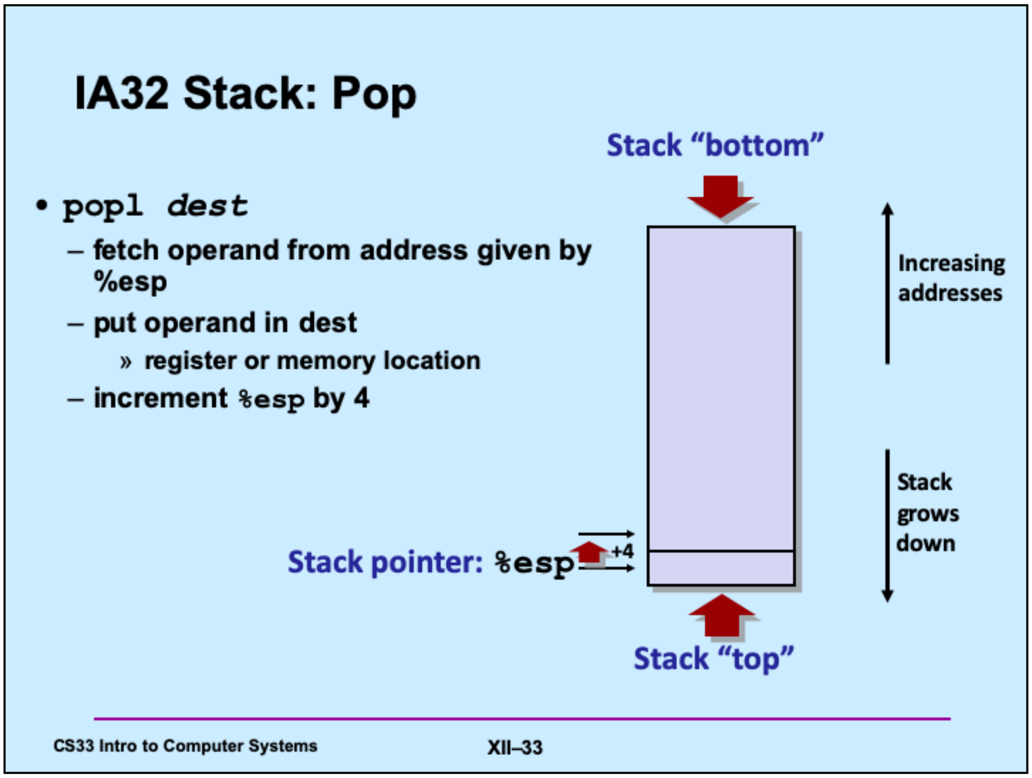
Supplied by CMU.

IA32 Stack: Push

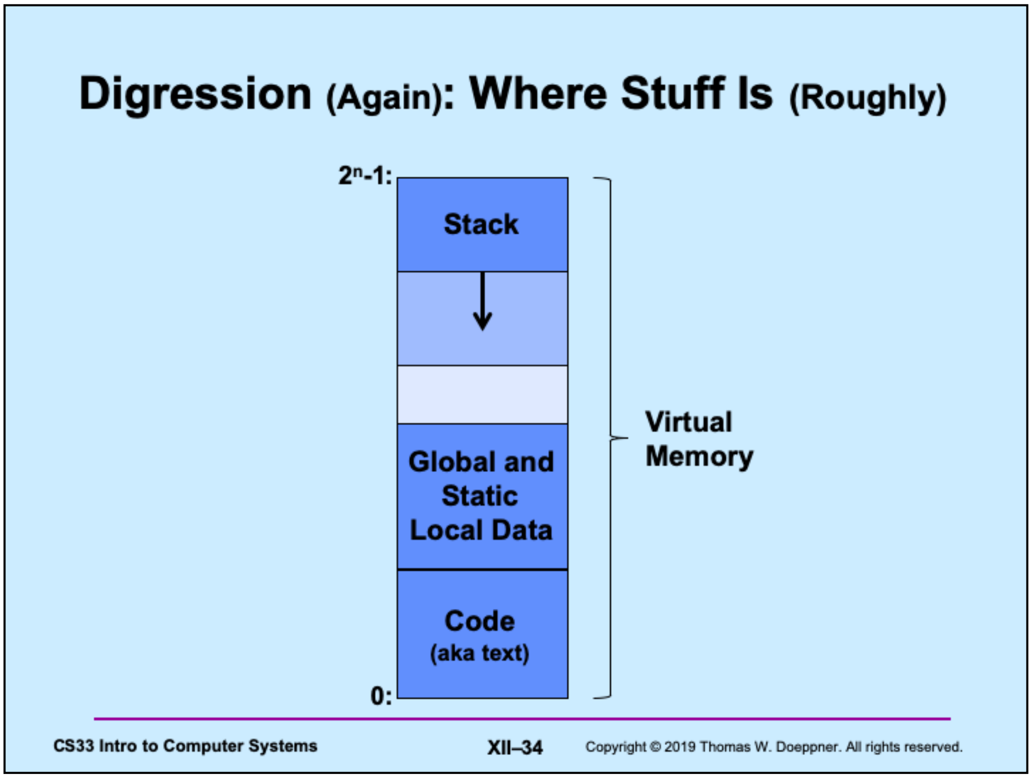
- `pushl` *`src`*
  - fetch operand at src
    - » immediate, register, or memory location
  - decrement `%esp` by 4
  - store operand at address given by `%esp`



Supplied by CMU.



Supplied by CMU.



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

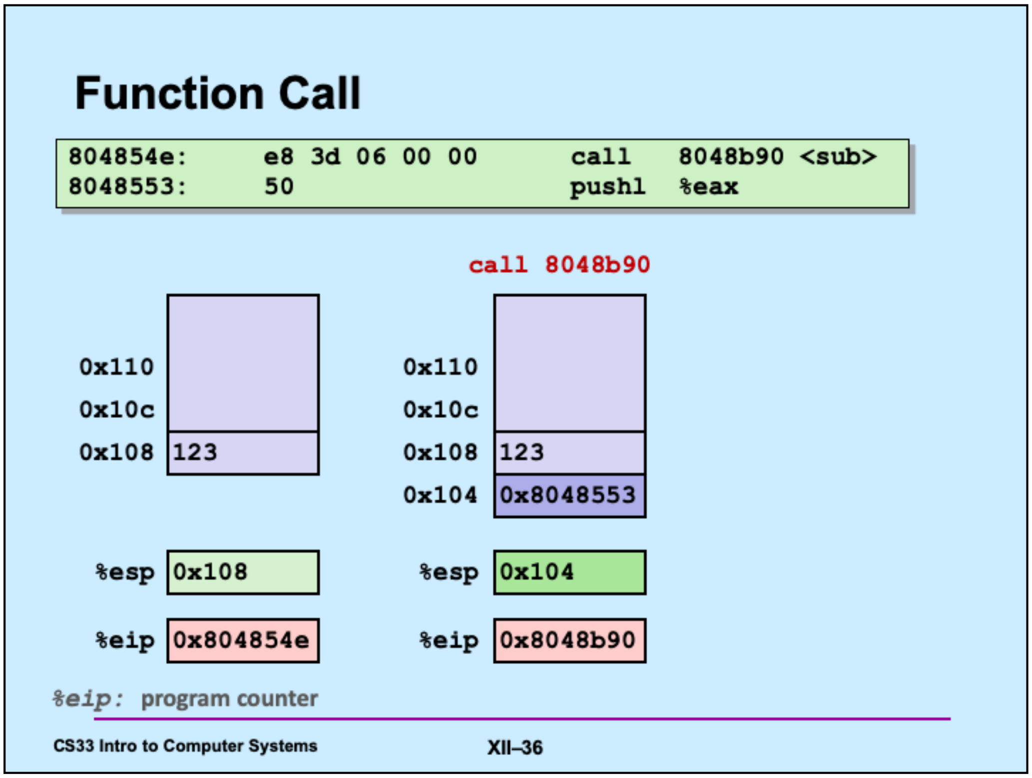
Function Control Flow

- Use stack to support function call and return
- **Function call:** `call sub`
  - push return address on stack
  - jump to sub
- Return address:
  - address of the next instruction after call
  - example from disassembly

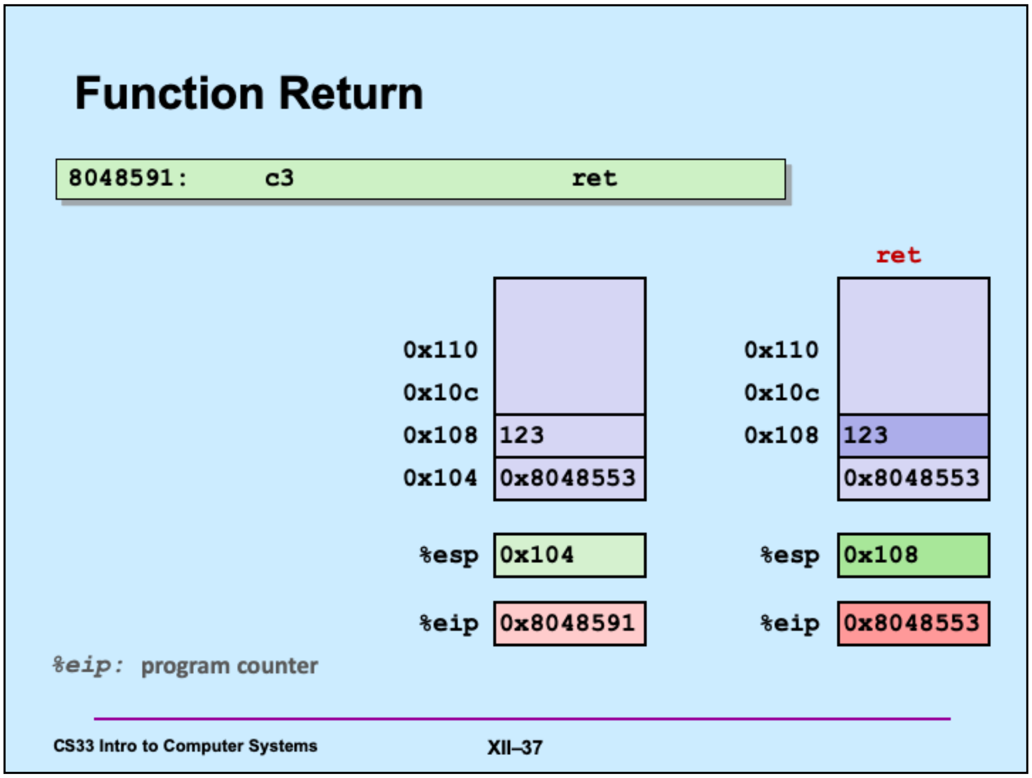
```
804854e:  e8 3d 06 00 00     call   8048b90 <sub>
8048553:  50                 pushl  %eax
```

  - return address = `0x8048553`
- **Function return:** `ret`
  - pop address from stack
  - jump to address

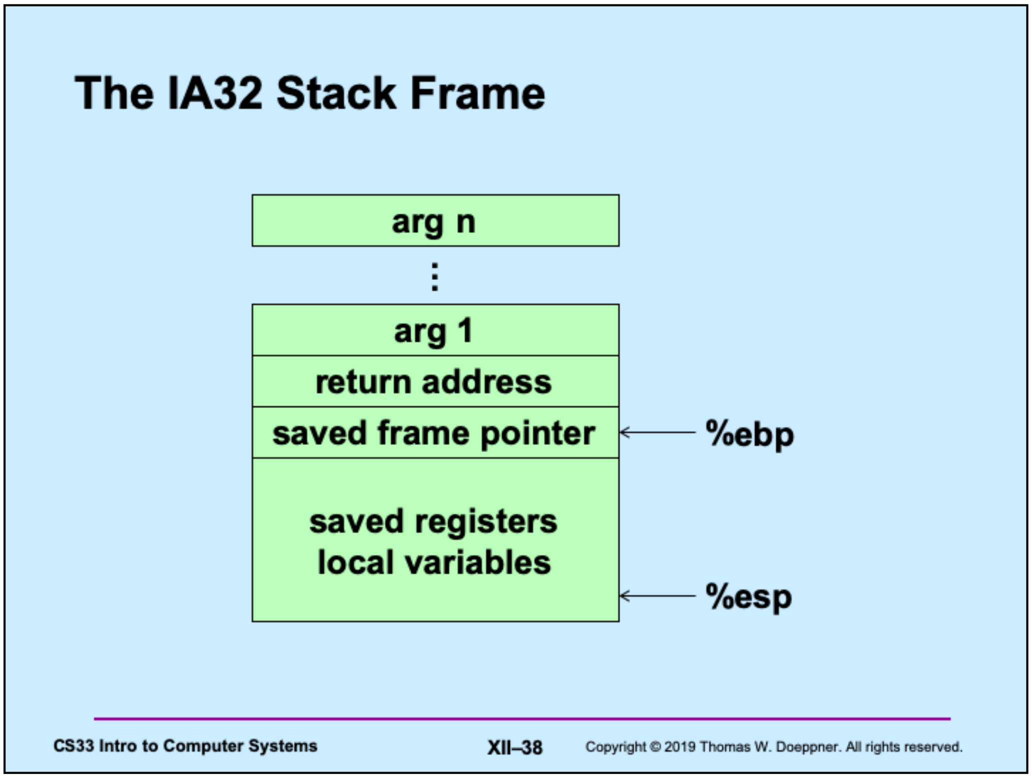
Supplied by CMU.



Supplied by CMU.



Supplied by CMU.

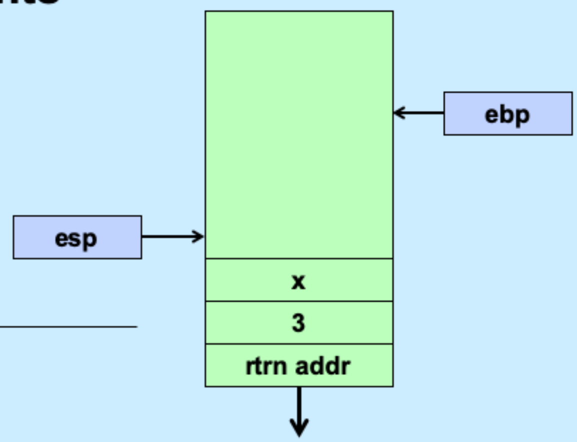


For the IA32 architecture, each function's stack frame is organized as in the slide. %ebp, sometimes called the base pointer, but more generically the frame pointer, points to a standard offset within stack frame. It's used to refer to the arguments pushed into the caller's stack frame as well as to local variables, etc., pushed into the function's stack frame.

Passing Arguments

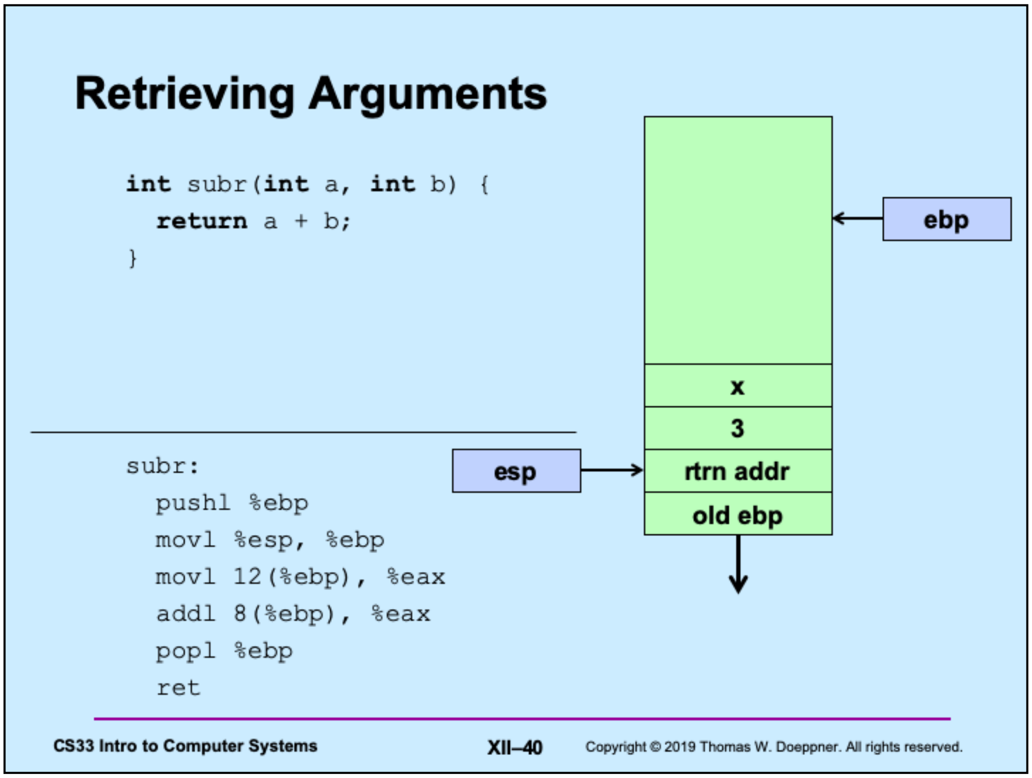
```
int x;
int res;
int main() {
  ...
  res = subr(3, x);
  ...
}
```

```
main:
  ...
  pushl x
  pushl $3
  call subr
  movl %eax,res
  ...
```



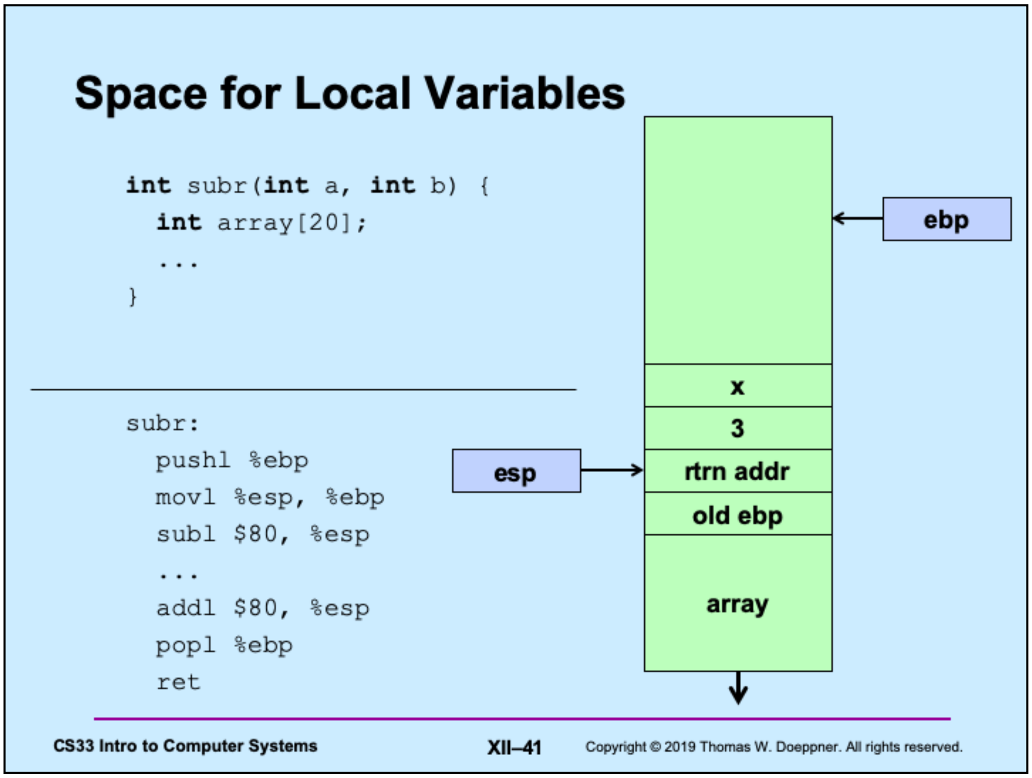
 Copyright © 2019 Thomas W. Doeppner. All rights reserved.

The convention for the IA32 architecture is for the caller of a function to push its arguments on the stack in reverse order. It then calls the function, which has the effect of pushing the return address (the address of the instruction following the call) onto the stack.



Again, following the IA32 convention, the first think a function does is to push the contents of %ebp onto the stack, thus saving the pointer to the caller's stack frame. It then copies the current stack pointer (%esp) into %ebp, so that %ebp now refers to the current stack frame. Having done this, the function can now refer to its arguments via offsets from %ebp.

When the function is ready to return to its caller, it first pops off the stack the copy of the caller's %ebp that was pushed onto the stack, replacing the current contents of %ebp with this saved value. This has the effect of making the caller's stack frame the current frame. Next the function calls *ret*, which pops the return address off the stack and sets %eip (the instruction pointer) to that value, causing control to return to the caller at the instruction following the call instruction.



If the function has local variables, these are allocated on the stack by decrementing the stack pointer to account for the space needed, and then popped of the stack when the function returns by adding the space occupied back to the stack pointer.

Register-Saving Conventions

- When function `yoo` calls `who`:
  - `yoo` is the **caller**
  - `who` is the **callee**
- Can registers be used for temporary storage?

```
yoo:
   • • •
   movl $33, %edx
   call who
   addl %edx, %eax
   • • •
   ret
```

```
who:
   • • •
   movl 8(%ebp), %edx
   addl $32, %edx
   • • •
   ret
```

  - contents of register `%edx` overwritten by `who`
  - this could be trouble: something should be done!
    » need some coordination

Supplied by CMU.

Register-Saving Conventions

- **When function `yoo` calls `who`:**
  - `yoo` is the caller
  - `who` is the callee
- **Can registers be used for temporary storage?**
- **Conventions**
  - "caller save"
    - » caller saves registers containing temporary values on stack before the call
    - » restores them after call
  - "callee save"
    - » callee saves registers on stack before using
    - » restores them before returning

Supplied by CMU.

IA32/Linux+Windows Register Usage

- `%eax`, `%edx`, `%ecx`
  - caller saves prior to call if values are used later
- `%eax`
  - also used to return integer value
- `%ebx`, `%esi`, `%edi`
  - callee saves if wants to use them
- `%esp`, `%ebp`
  - special form of callee-save
  - restored to original values upon exit from function

Supplied by CMU.

Register-Saving Example

```
yoo:
   ...
   movl $33, %edx
   pushl %edx
   call who
   popl %edx
   addl %edx, %eax
   ...
   ret
```

```
who:
   ...
   pushl %ebx
   ...
   movl 4(%ebp), %ebx
   addl %53, %ebx
   movl 8(%ebp), %edx
   addl $32, %edx
   ...
   popl %ebx
   ...
   ret
```


Supplied by CMU.

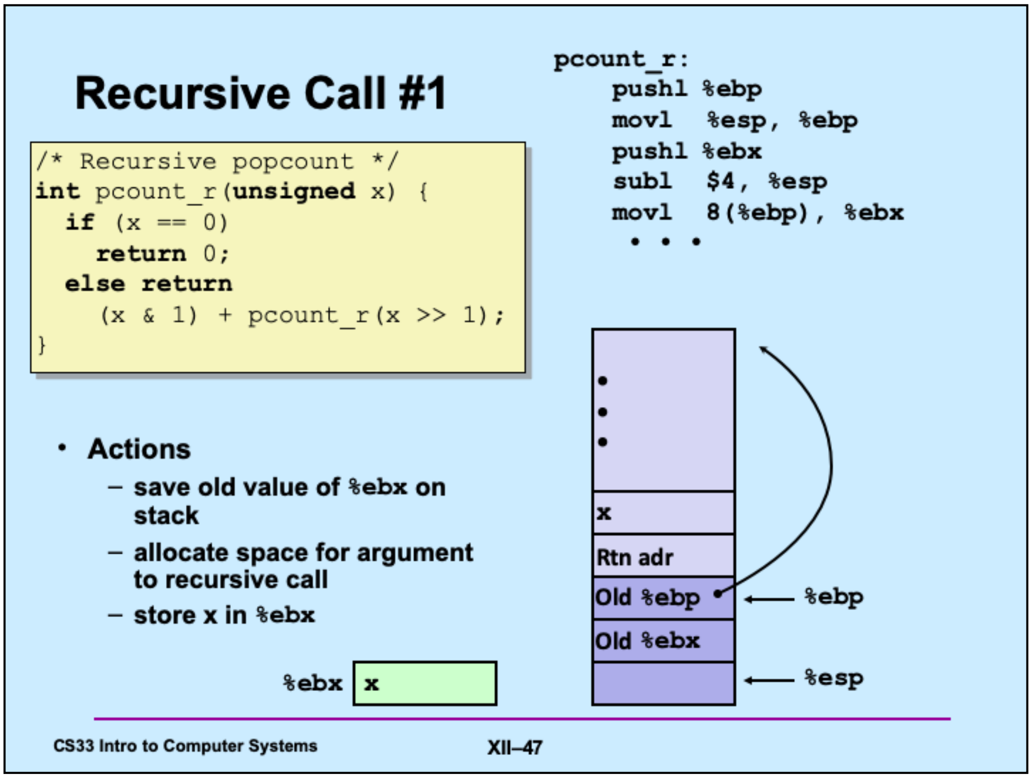
Recursive Function

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

- **Registers**
  - `%eax`, `%edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

```
pcount_r:
    pushl  %ebp
    movl   %esp, %ebp
    pushl  %ebx
    subl   $4, %esp
    movl   8(%ebp), %ebx
    movl   $0, %eax
    testl  %ebx, %ebx
    je     .L3
    movl   %ebx, %eax
    shrl   $1, %eax
    movl   %eax, (%esp)
    call   pcount_r
    movl   %ebx, %edx
    andl   $1, %edx
    leal   (%edx,%eax), %eax
.L3:
    addl   $4, %esp
    popl   %ebx
    popl   %ebp
    ret
```


Supplied by CMU.



Supplied by CMU.

Note that space for the argument to recursive call to *pcount_r* is allocated on the stack (by subtracting 4 from %esp) even if turns out that *pcount_r* won't be called.

Recursive Call #2

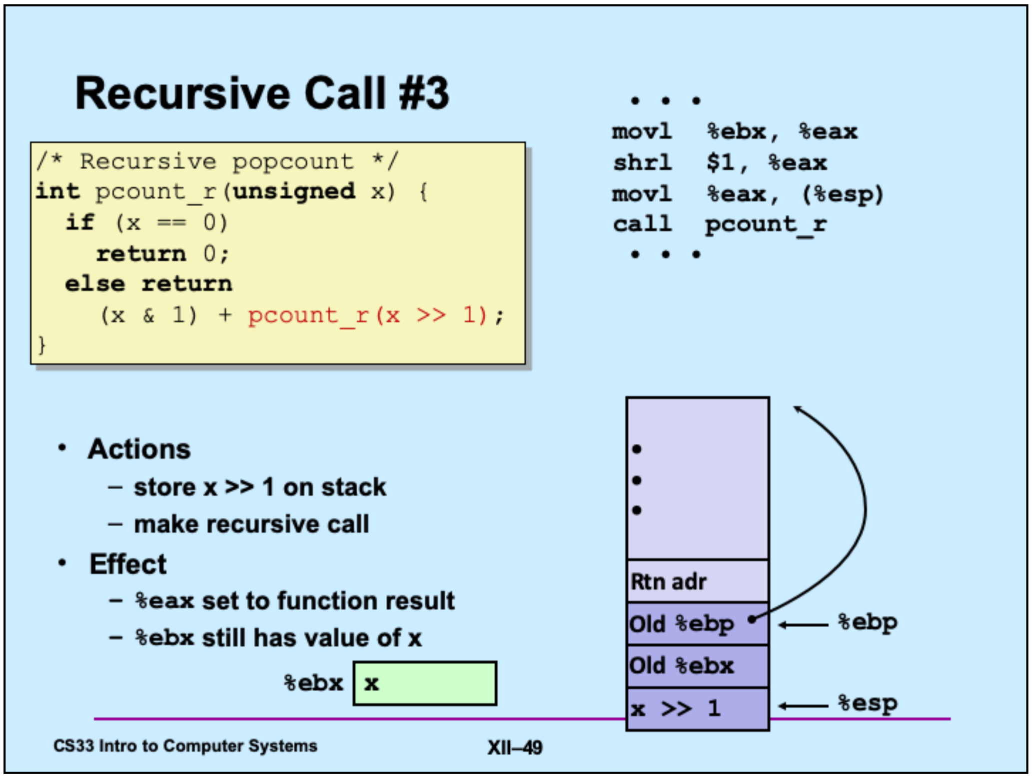
```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
    • • •
    movl  $0, %eax
    testl %ebx, %ebx
    je   .L3
    • • •
.L3:
    • • •
    ret
```

- **Actions**
  - if x == 0, return
    - » with `%eax` set to 0

`%ebx` | x

Supplied by CMU.



Supplied by CMU.

Recall that space for *pcount_r*'s argument has already been allocated on the stack, thus %esp already points to where the argument should go.

Recursive Call #4

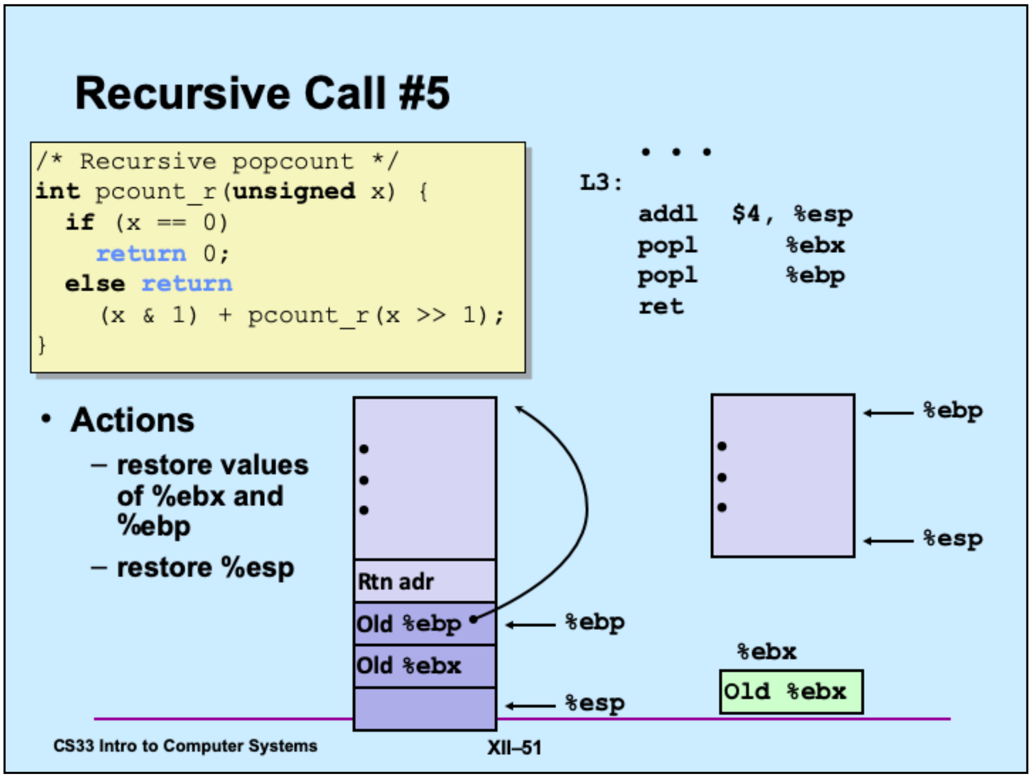
```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
  • • •
movl    %ebx, %edx
andl    $1, %edx
leal    (%edx,%eax), %eax
  • • •
```

- Assume
  - %eax holds value from recursive call
  - %ebx holds x
- Actions
  - compute (x & 1) + computed value
- Effect
  - %eax set to function result

%ebx	x

Supplied by CMU.



Supplied by CMU.

At this point, general cleanup is done: the space allocated for the argument to the recursive call to *pcount_r* is restored (by adding 4 to %esp), and the values of %ebx (used as temporary storage) and %ebp (the base pointer) are restored.

Observations About Recursion

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

Supplied by CMU.

Why Bother with a Frame 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 frame pointer is superfluous**
  - it can be used as a general-purpose register

Copyright © 2019 Thomas W. Doeppner. All rights reserved.

Note that "frame pointer" is synonymous with "base pointer".

If one gives gcc the –O0 flag (which turns off all optimization) when compiling, the frame 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: 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.