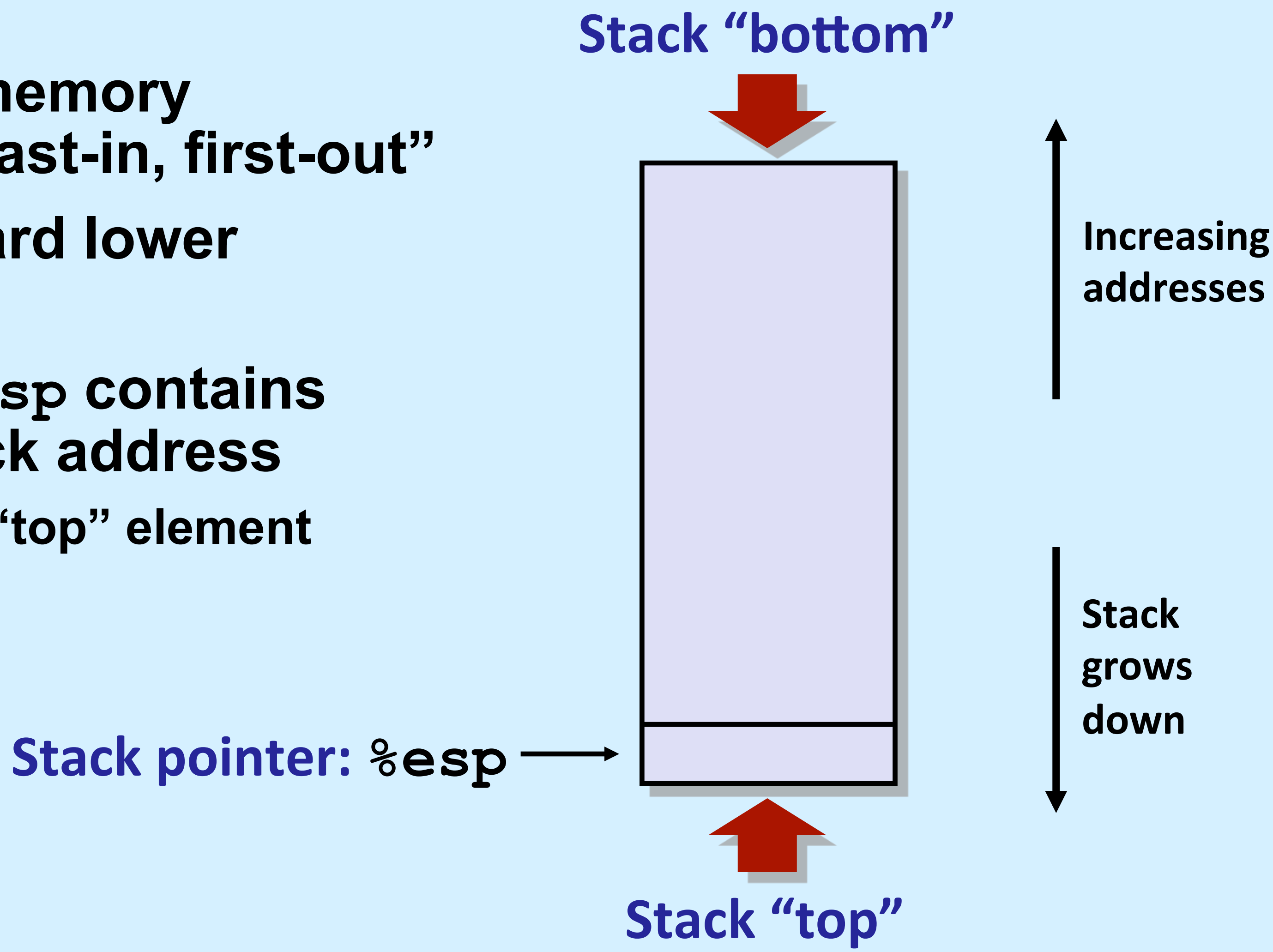
CS 33

Machine Programming (3)

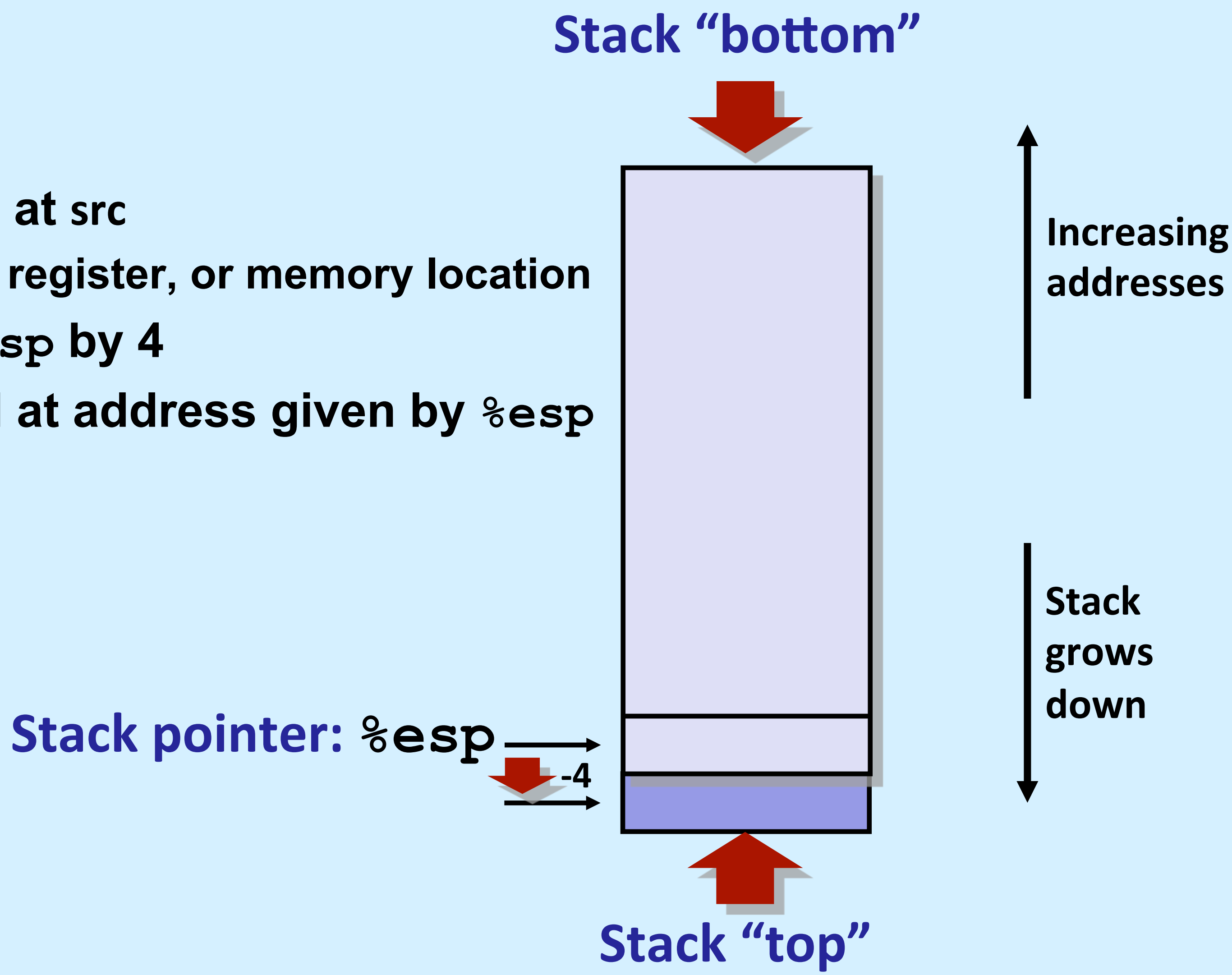
IA32 Stack

- Region of memory managed "last-in, first-out"
- Grows toward lower addresses
- Register `%esp` contains lowest stack address
  – address of "top" element



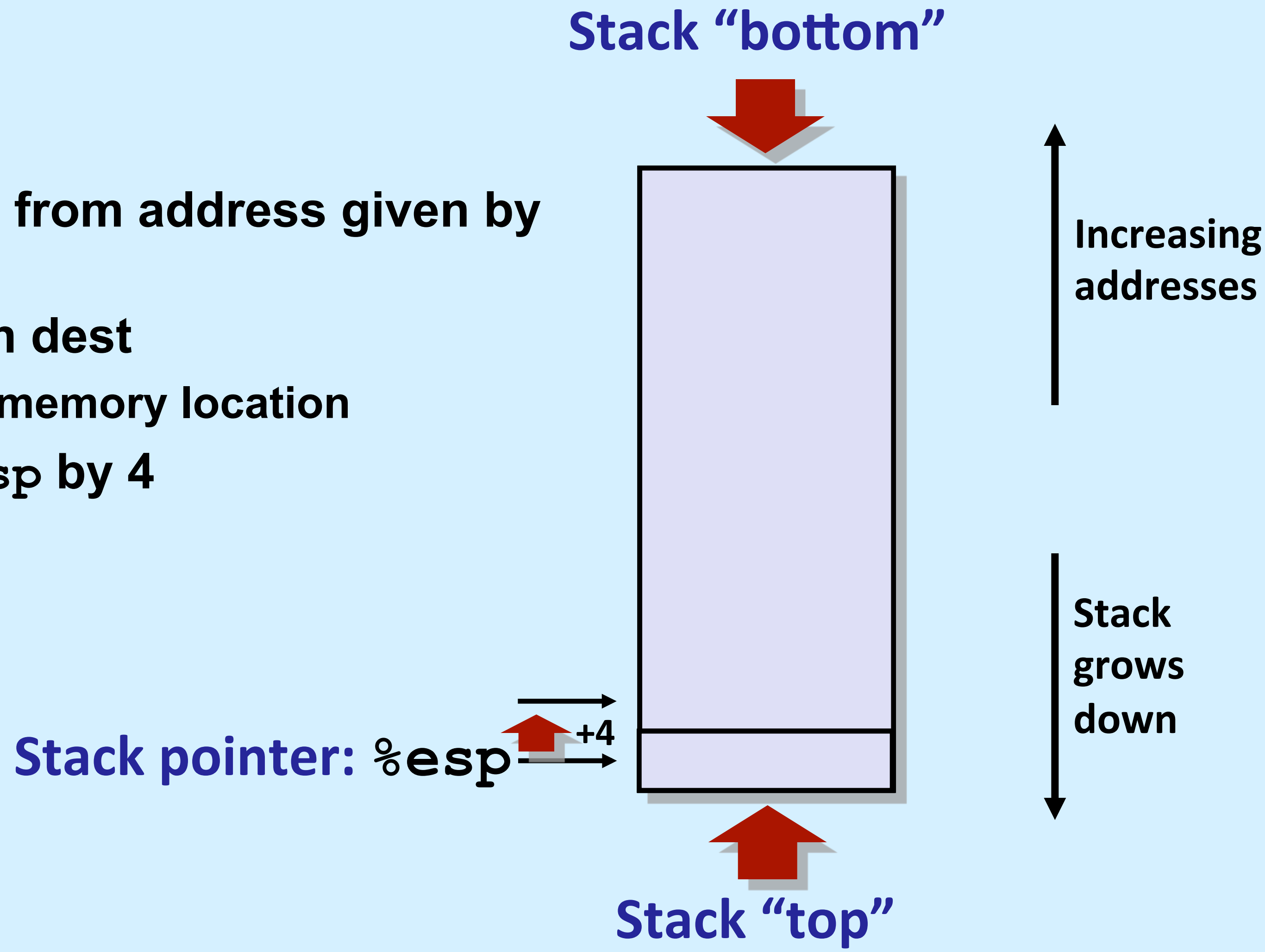
IA32 Stack: Push

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



IA32 Stack: Pop

- `popl` ***`dest`***
  - fetch operand from address given by %esp
  - put operand in dest
    - » register or memory location
  - increment `%esp` by 4



Procedure Control Flow

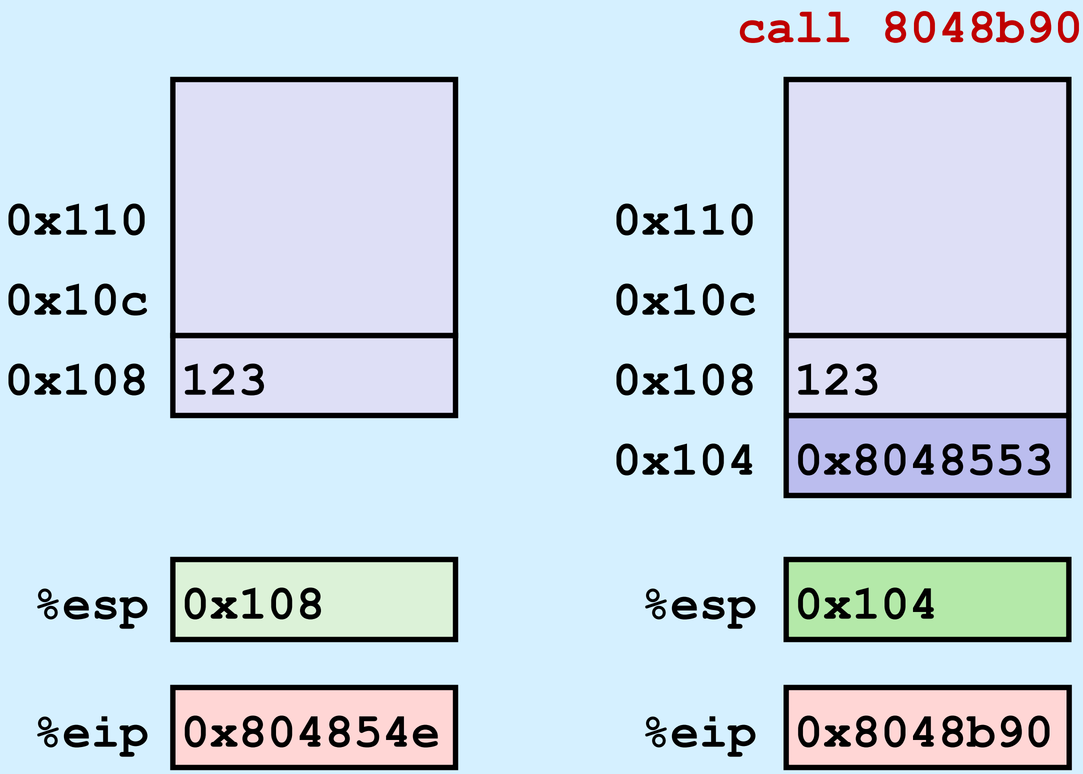
- Use stack to support procedure call and return
- Procedure call: `call sub`
  - push return address on stack
  - jump to sub
- Return address:
  - address of the next instruction after call
  - example from disassembly

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

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

Procedure Call

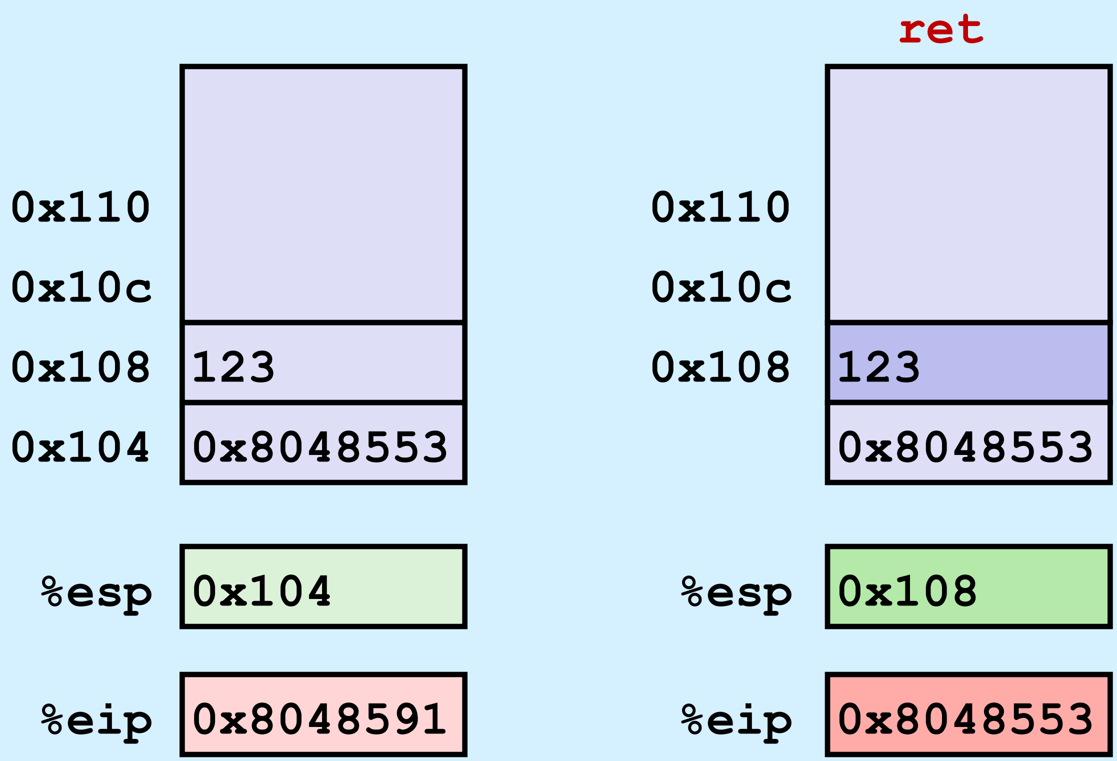
```
804854e:      e8 3d 06 00 00        call    8048b90 <sub>
8048553:      50                    pushl   %eax
```



`%eip`: program counter

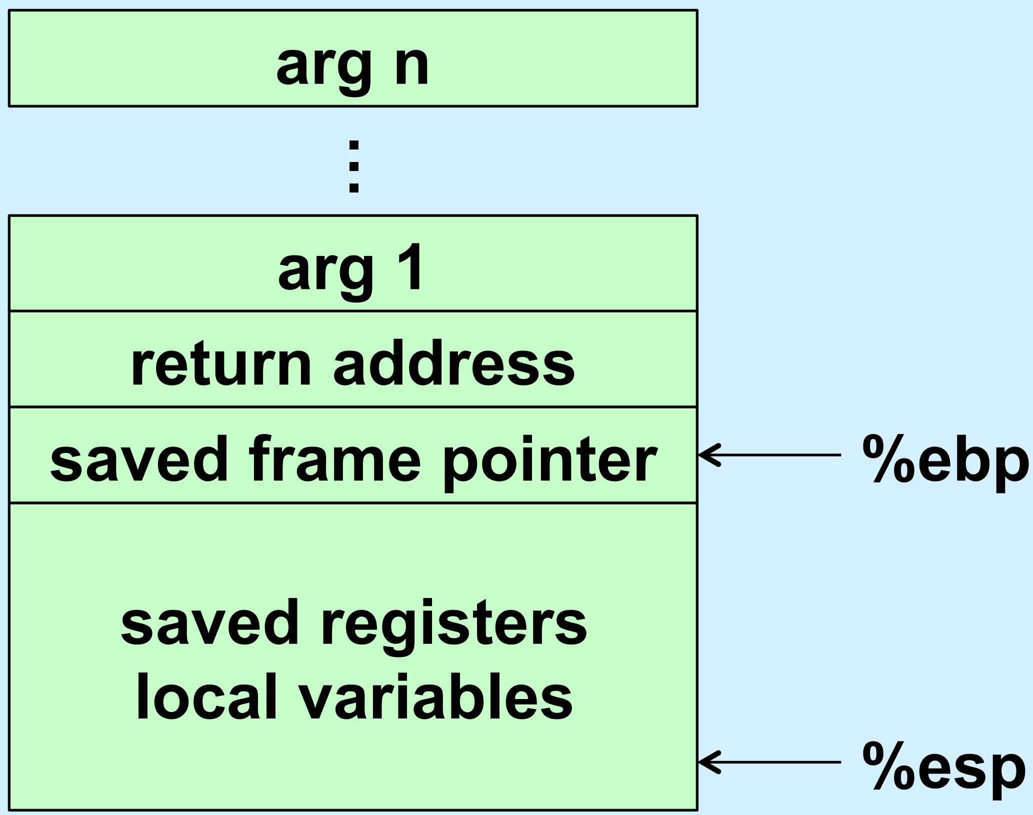
Procedure Return

```
8048591:     c3                    ret
```



`%eip:` program counter

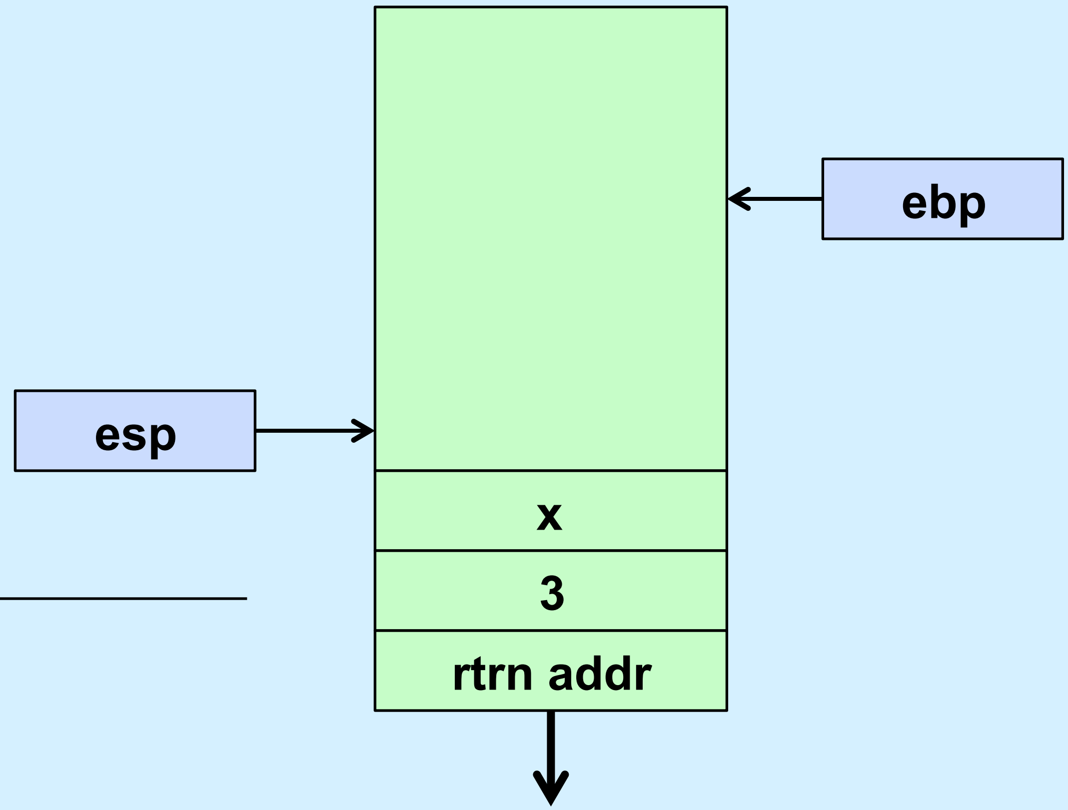
The IA32 Stack Frame



Passing Arguments

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

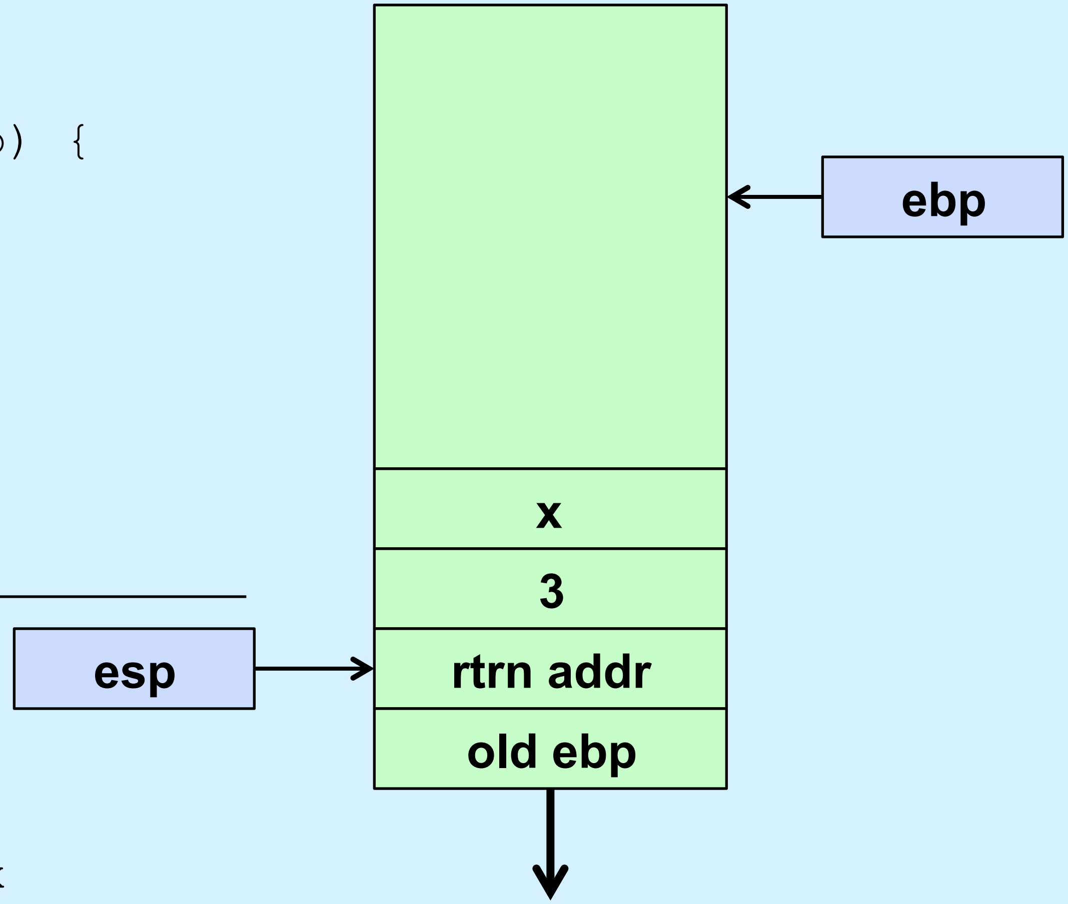
```
main:
   ...
   pushl x
   pushl $3
   call subr
   movl %eax,res
   ...
```



Retrieving Arguments

```
int subr(int a, int b) {
  return a + b;
}
```

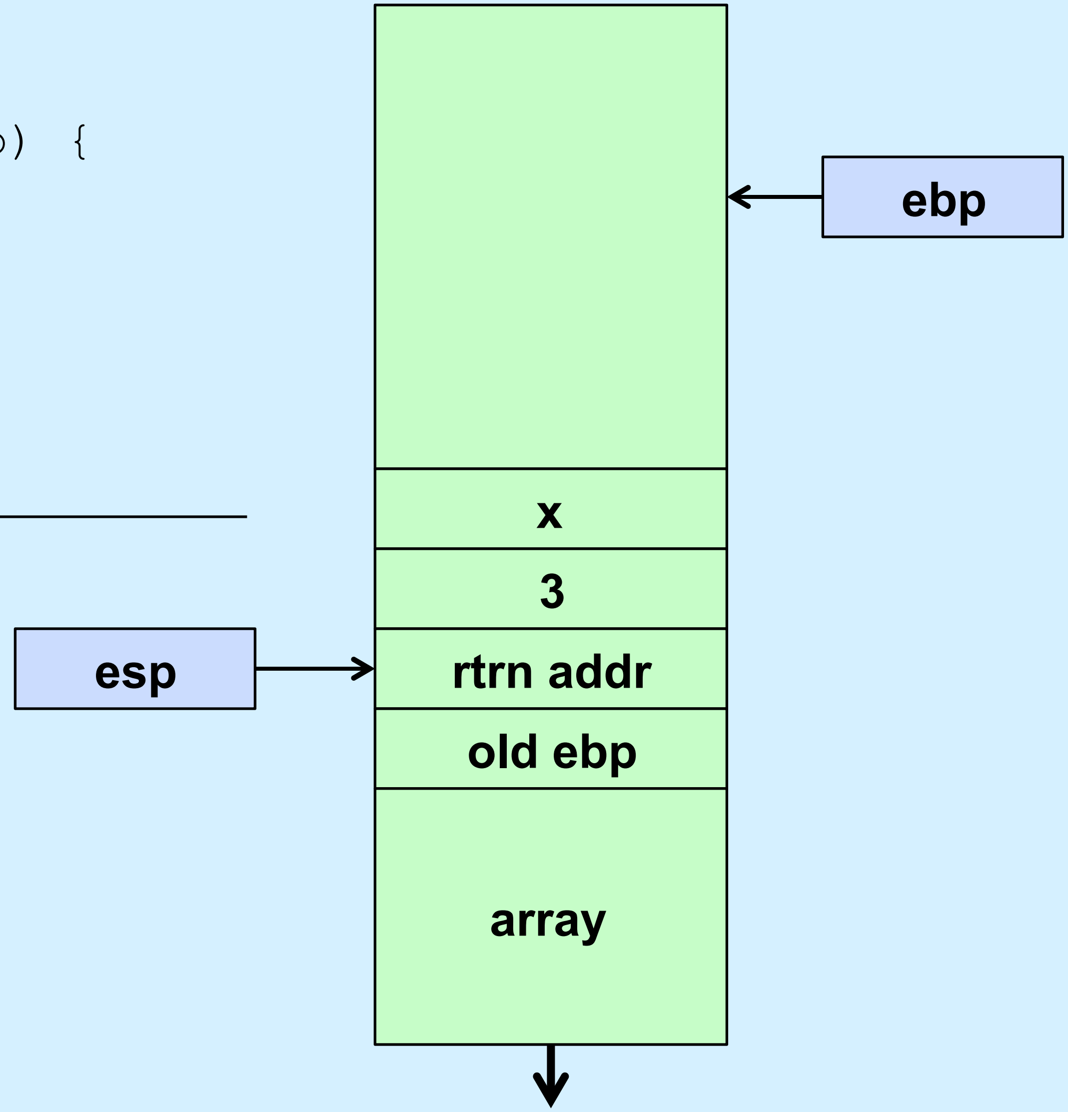
```
subr:
  pushl %ebp
  movl %esp, %ebp
  movl 12(%ebp), %eax
  addl 8(%ebp), %eax
  popl %ebp
  ret
```



Space for Local Variables

```
int subr(int a, int b) {
    int array[20];
    ...
}
```

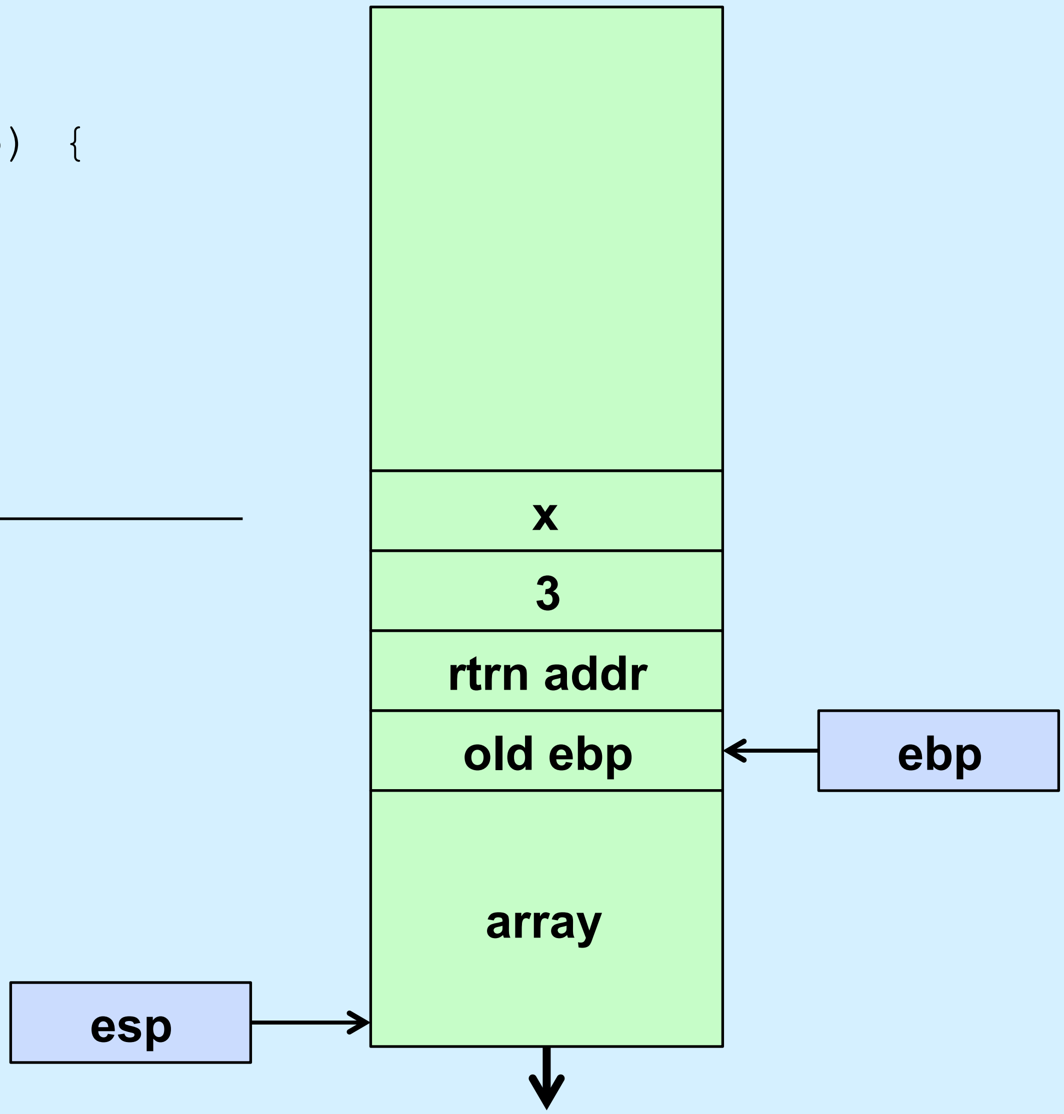
```
subr:
    pushl %ebp
    movl %esp, %ebp
    subl $80, %esp
    ...
    addl $80, %esp
    popl %ebp
    ret
```



Quick Exit ...

```
int subr(int a, int b) {
  int array[20];
  ...
}
```

```
subr:
  pushl %ebp
  movl %esp, %ebp
  subl $80, %esp
  ...
  leave
  ret
```



Register-Saving Conventions

- When procedure `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
```

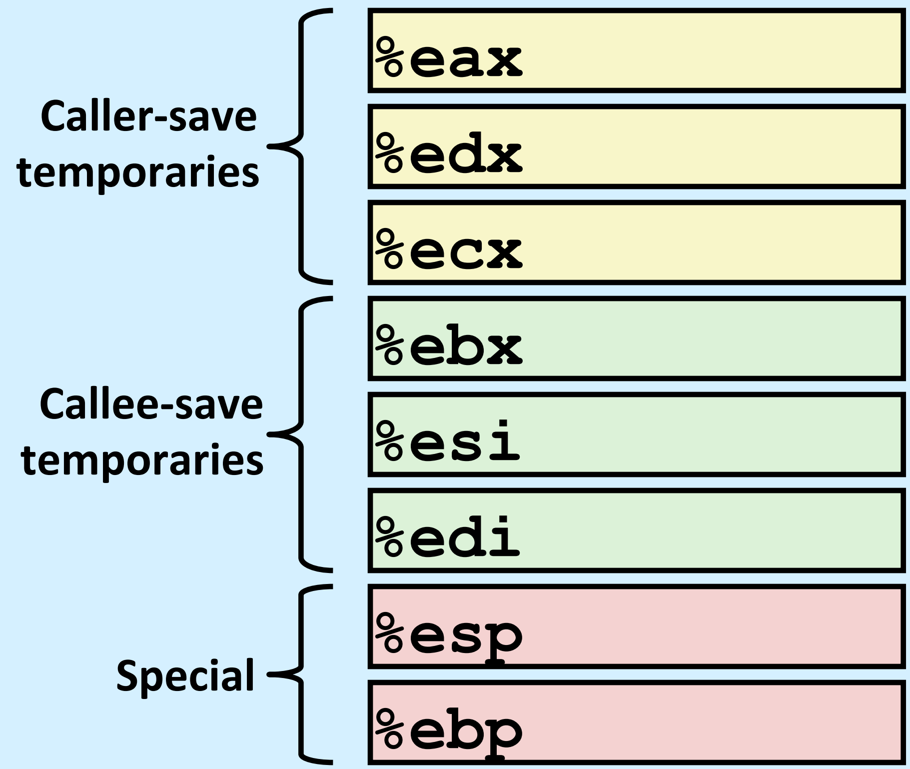
  - contents of register `%edx` overwritten by `who`
  - this could be trouble: something should be done!
    - » need some coordination

Register-Saving Conventions

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

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 procedure



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
```

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

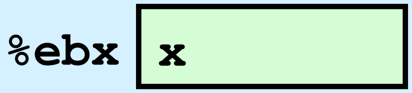
```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    movl  $0, %eax
    testl %ebx, %ebx
    je    .L3
    movl  %ebx, %eax
    shrl  $1, %eax
    movl  %eax, (%esp)
    call  pcount_r
    movl  %ebx, %edx
    andl  $1, %edx
    leal  (%edx,%eax),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

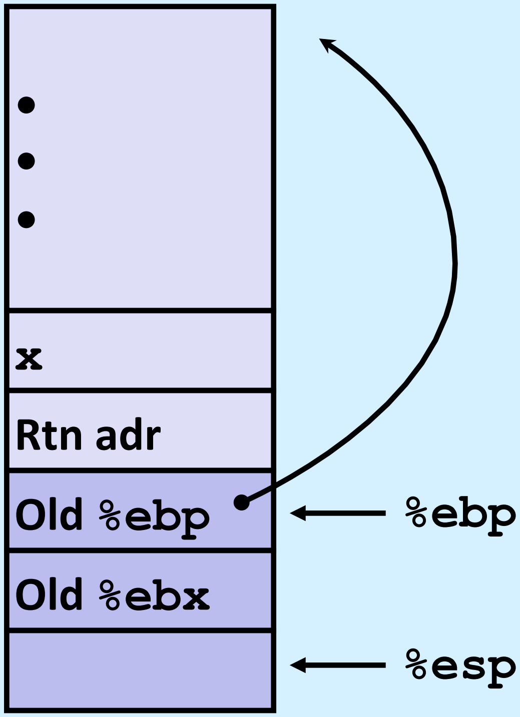
Recursive Call #1

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

```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
     • • •
```

- **Actions**
  - save old value of `%ebx` on stack
  - allocate space for argument to recursive call
  - store x in `%ebx`





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

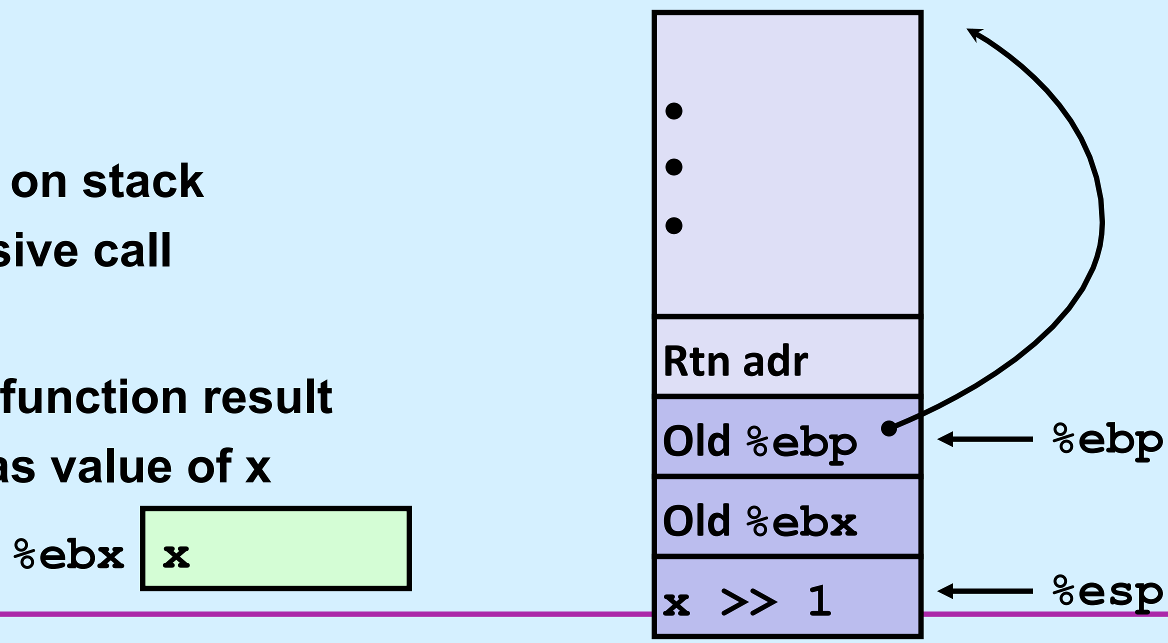
`%ebx` | `x`

Recursive Call #3

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

```
  . . .
movl   %ebx, %eax
shrl   $1, %eax
movl   %eax, (%esp)
call   pcount_r
  . . .
```

- **Actions**
  - store x >> 1 on stack
  - make recursive call
- **Effect**
  - `%eax` set to function result
  - `%ebx` still has value of x



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

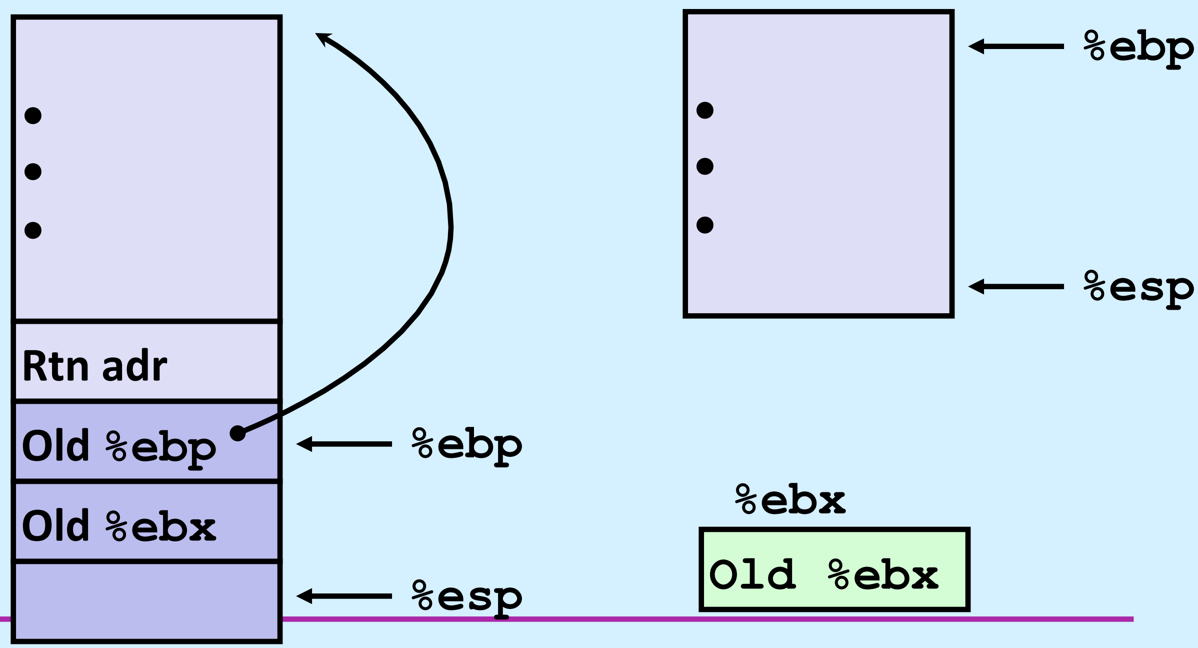
%ebx	x

Recursive Call #5

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

```
  • • •
L3:
    addl $4, %esp
    popl %ebx
    popl %ebp
    ret
```

- **Actions**
  - restore values of %ebx and %ebp
  - restore %esp

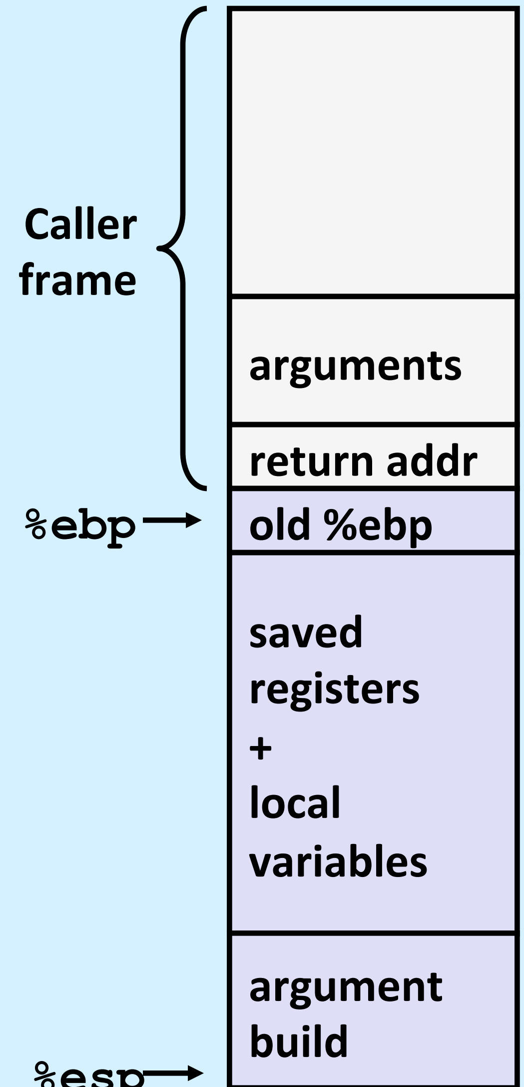


Observations About Recursion

- **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

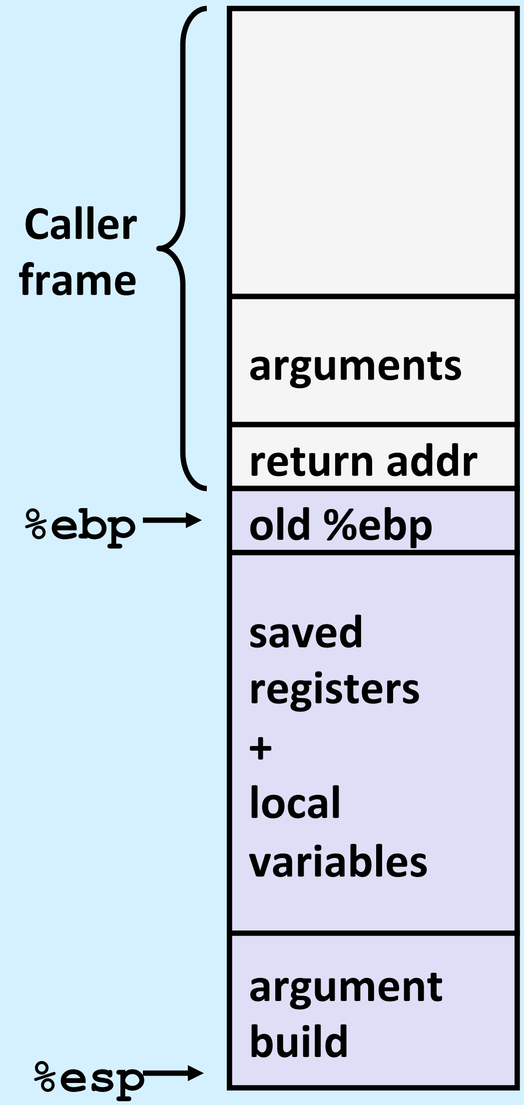
IA 32 Procedure Summary

- Important Points
  - stack is the right data structure for procedure call / return
    - » if P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - can safely store values in local stack frame and in callee-saved registers
  - put function arguments at top of stack
  - result return in `%eax`
- Pointers are addresses of values
  - on stack or global



Quiz 1

- The leave instruction copies the current value of %ebp into %esp. It's followed by a ret instruction. Does this approach for returning from a procedure work if there are saved registers in the stack frame?
  a) always
  b) usually
  c) never



Why Bother with a Frame Pointer?

- It 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

x86-64 General-Purpose Registers: Usage Conventions

`%rax`	Return value
`%rbx`	Callee saved
`%rcx`	Argument #4
`%rdx`	Argument #3
`%rsi`	Argument #2
`%rdi`	Argument #1
`%rsp`	Stack pointer
`%rbp`	Callee saved

`%r8`	Argument #5
`%r9`	Argument #6
`%r10`	Caller saved
`%r11`	Caller Saved
`%r12`	Callee saved
`%r13`	Callee saved
`%r14`	Callee saved
`%r15`	Callee saved

x86-64 Registers

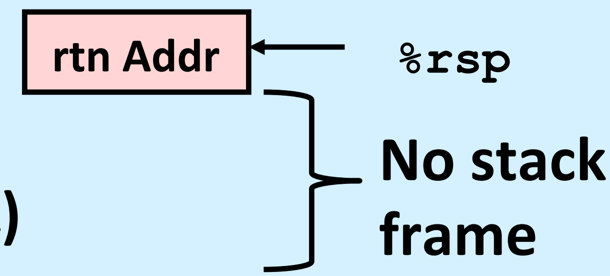
- Arguments passed to functions via registers
  - if more than 6 integral parameters, then pass rest on stack
  - these registers can be used as caller-saved as well
- All references to stack frame via stack pointer
  - eliminates need to update `%ebp/%rbp`
- Other registers
  - 6 callee-saved
  - 2 caller-saved
  - 1 return value (also usable as caller-saved)
  - 1 special (stack pointer)

x86-64 Long Swap

```
void swap_l(long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
   movq   (%rdi), %rdx
   movq   (%rsi), %rax
   movq   %rax, (%rdi)
   movq   %rdx, (%rsi)
   ret
```

- **Operands passed in registers**
  - **first (`xp`) in `%rdi`, second (`yp`) in `%rsi`**
  - **64-bit pointers**
- **No stack operations required (except `ret`)**
- **Avoiding stack**
  - **can hold all local information in registers**

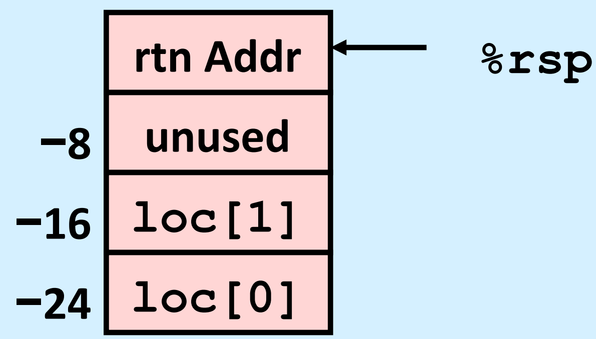


x86-64 Locals in the Red Zone

```
/* Swap, using local array */
void swap_a(long *xp, long *yp)
{
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
  movq  (%rdi), %rax
  movq  %rax, -24(%rsp)
  movq  (%rsi), %rax
  movq  %rax, -16(%rsp)
  movq  -16(%rsp), %rax
  movq  %rax, (%rdi)
  movq  -24(%rsp), %rax
  movq  %rax, (%rsi)
  ret
```

- **Avoiding stack-pointer change**
  - can hold all information within small window beyond stack pointer
    - » 128 bytes



x86-64 NonLeaf without Stack Frame

```
/* Swap a[i] & a[i+1] */
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee-save registers needed
- `rep` instruction inserted as no-op
  – based on recommendation from AMD
    » can't handle transfer of control to ret

```
swap_ele:
   movslq %esi,%rsi              # Sign extend i
   leaq   8(%rdi,%rsi,8), %rax   # &a[i+1]
   leaq   (%rdi,%rsi,8), %rdi    # &a[i] (1st arg)
   movq   %rax, %rsi             # (2nd arg)
   call   swap
   rep                           # No-op
   ret
```

x86-64 Stack Frame Example

```
long sum = 0;
/* Swap a[i] & a[i+1] */
void swap_ele_su
   (long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

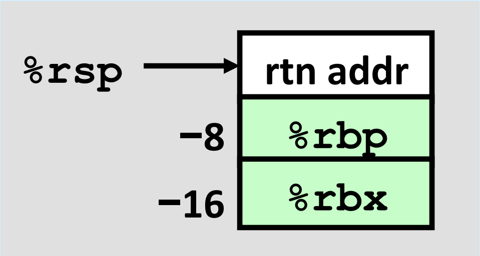
- Keeps values of `&a[i]` and `&a[i+1]` in callee-save registers
  - rbx and rbp
- Must set up stack frame to save these registers
  - else clobbered in swap

Understanding x86-64 Stack Frame

```
swap_ele_su:
   movq    %rbx, -16(%rsp)         # Save %rbx
   movq    %rbp, -8(%rsp)          # Save %rbp
   subq    $16, %rsp               # Allocate stack frame
   movslq  %esi,%rax               # Extend i into quad word
   leaq    8(%rdi,%rax,8), %rbx    # &a[i+1] (callee save)
   leaq    (%rdi,%rax,8), %rbp     # &a[i]   (callee save)
   movq    %rbx, %rsi              # 2nd argument
   movq    %rbp, %rdi              # 1st argument
   call    swap
   movq    (%rbx), %rax            # Get a[i+1]
   imulq   (%rbp), %rax            # Multiply by a[i]
   addq    %rax, sum(%rip)         # Add to sum
   movq    (%rsp), %rbx            # Restore %rbx
   movq    8(%rsp), %rbp           # Restore %rbp
   addq    $16, %rsp               # Deallocate frame
   ret
```

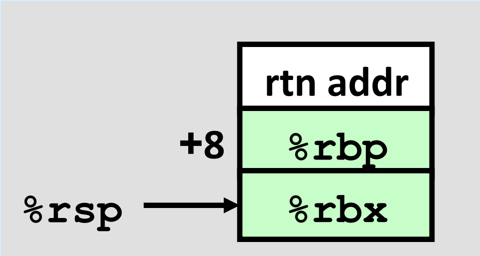
Understanding x86-64 Stack Frame

```
movq    %rbx, -16(%rsp)          # Save %rbx
movq    %rbp, -8(%rsp)           # Save %rbp
```




```
subq    $16, %rsp                # Allocate stack frame
```

• • •




```
movq    (%rsp), %rbx             # Restore %rbx
movq    8(%rsp), %rbp            # Restore %rbp

addq    $16, %rsp                # Deallocate frame
```

Quiz 2

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)       # Add to sum
    movq    (%rsp), %rbx          # Restore %rbx
    movq    8(%rsp), %rbp         # Restore %rbp
    addq    $16, %rsp             # Deallocate frame
    ret
```

Since a 128-byte red zone is allowed, is it necessary to allocate the stack frame by subtracting 16 from %rsp?

a) yes
b) no

Interesting Features of Stack Frame

- **Allocate entire frame at once**
  - all stack accesses can be relative to `%rsp`
  - do by decrementing stack pointer
  - can delay allocation, since safe to temporarily use red zone
- **Simple deallocation**
  - increment stack pointer
  - no base/frame pointer needed

x86-64 Procedure Summary

- Heavy use of registers
  - parameter passing
  - more temporaries since more registers
- Minimal use of stack
  - sometimes none
  - allocate/deallocate entire block
- Many tricky optimizations
  - what kind of stack frame to use
  - various allocation techniques

Tail Recursion

```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

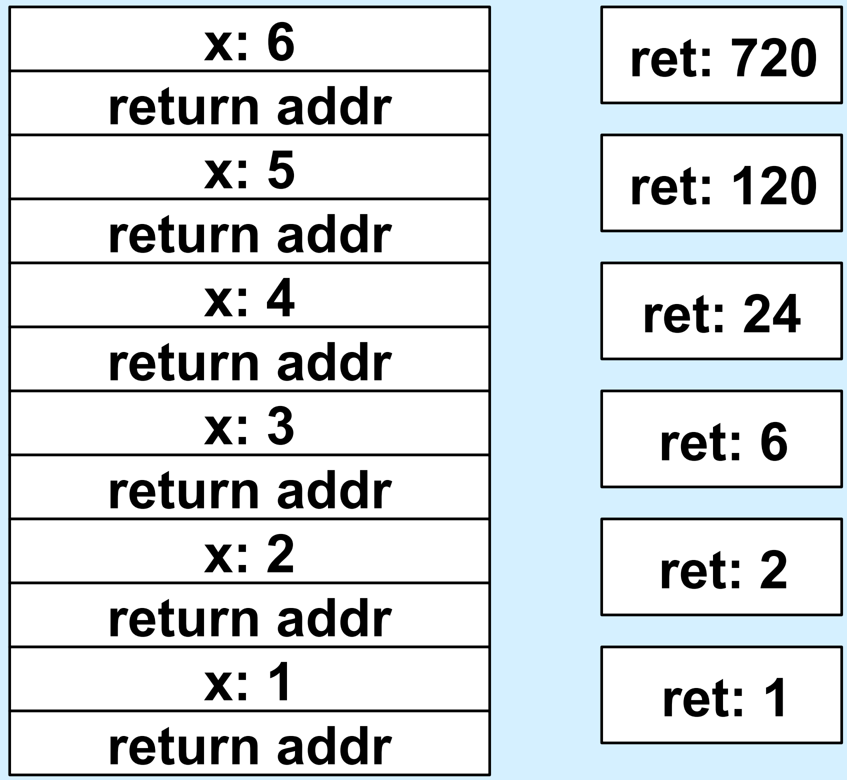
```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```

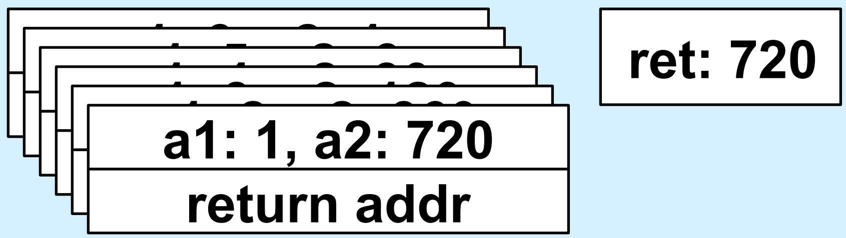

No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

No Tail Recursion (2)



Tail Recursion



Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
        addq    $8, %rsp
.L5:
        rep
        ret
```


Code: gcc –O2

```
f2:
        cmpl    $1, %edi
        movl    %esi, %eax
        je      .L8
.L12:
        imull   %edi, %eax
        subl    $1, %edi
        cmpl    $1, %edi
        jne     .L12
.L8:
        rep
        ret
```

loop!