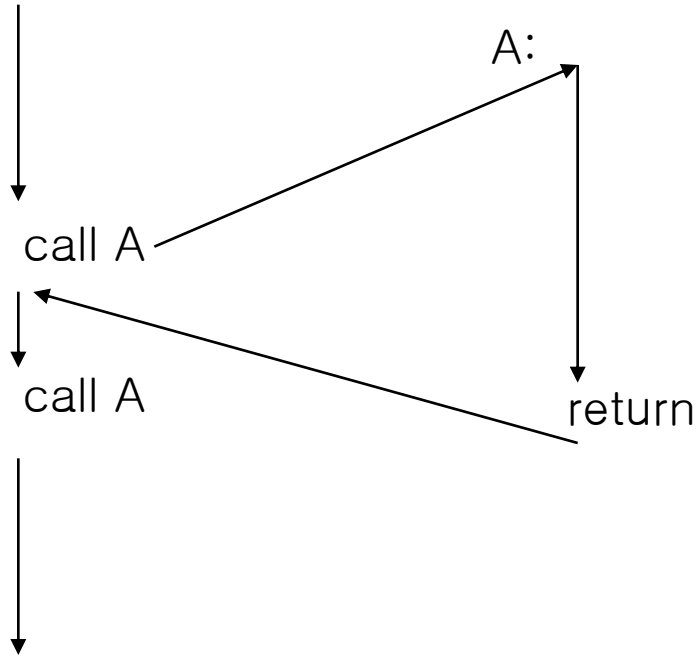
Machine-level Programming (3)

Procedures



- Two issues
  - How to return to the correct position?
  - How to pass arguments and return values between callee to caller?

Procedure Control Flow

- Use stack to support procedure call and return

- Procedure call:

  `call` *`label`* Push return address on stack; Jump to *`label`*

- Return address value
  - Address of instruction beyond `call`
  - Example from disassembly

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

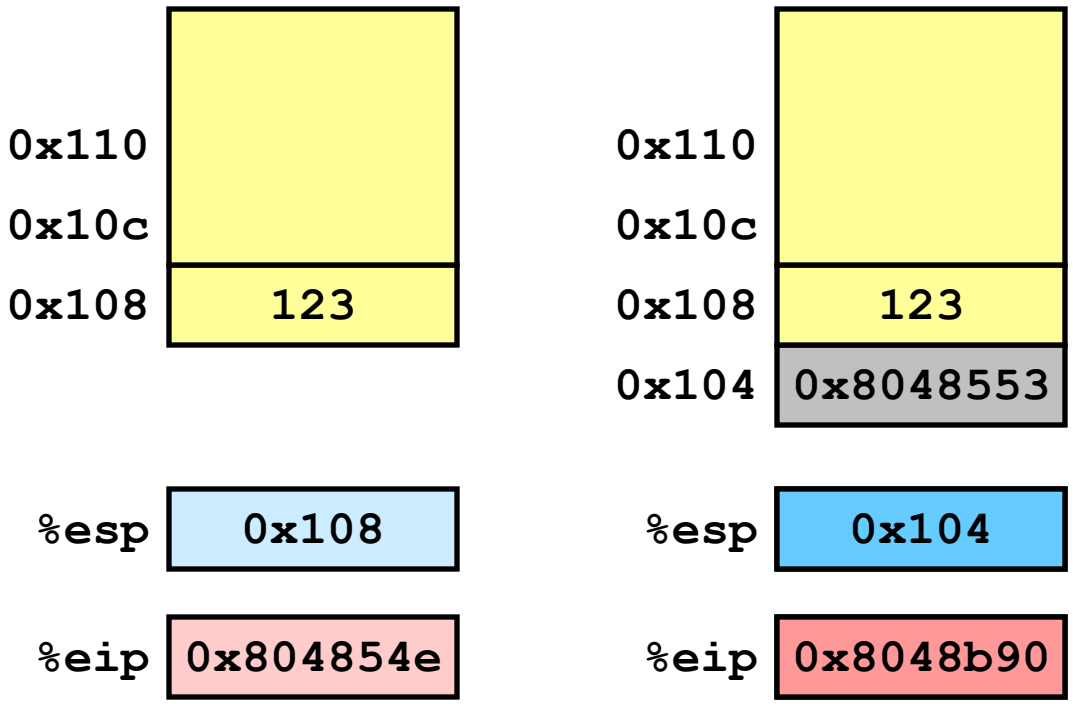
    - Return address = `0x8048553`

- Procedure return:
  - `ret` Pop address from stack; Jump to address

Procedure Call Example

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

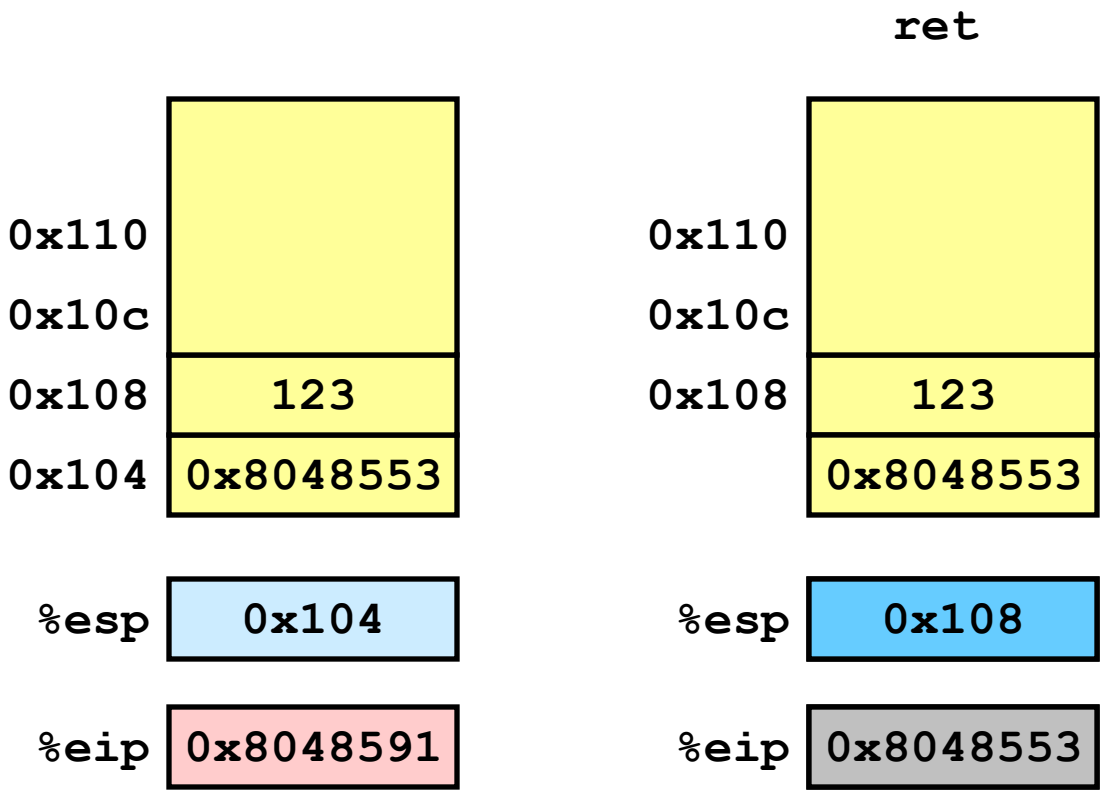
`call 8048b90`



`%eip` is program counter

Procedure Return Example

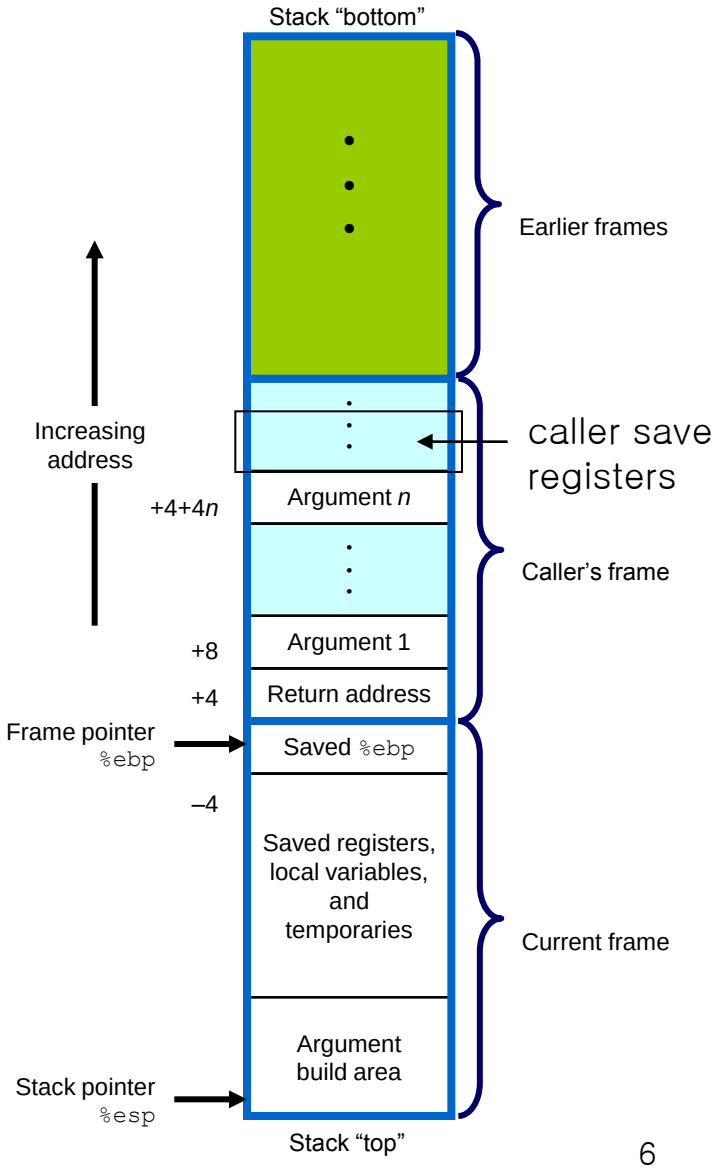
```
8048591:   c3                    ret
```



`%eip` **is program counter**

IA32 Procedure Call Conventions

- Argument and return value passing
  - Argument: Stack (and Registers)
  - Return value: eax
- Caller and Callee's role
  - Caller save registers: eax, edx, ecx.
    - Caller saves (pushes) these registers if it wants to use them later. So, Callee is free to overwrite them.
  - Callee save registers: ebx, esi, edi.
    - If Callee want use these registers, it MUST save (push) them before overwrite. It MUST restore (pop) them before returning.
- Stack-frame (ebp, esp) per each instantiation of procedures



Call Chain Example

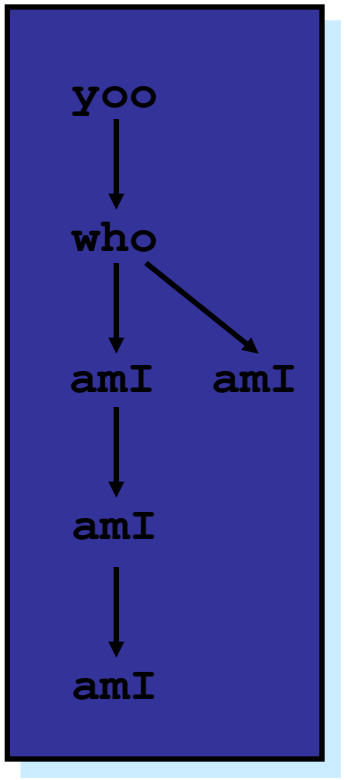
- Code Structure

```
yoo(…)
{
    •
    •
    who();
    •
    •
}
```

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

```
amI(…)
{
    •
    •
    amI();
    •
    •
}
```

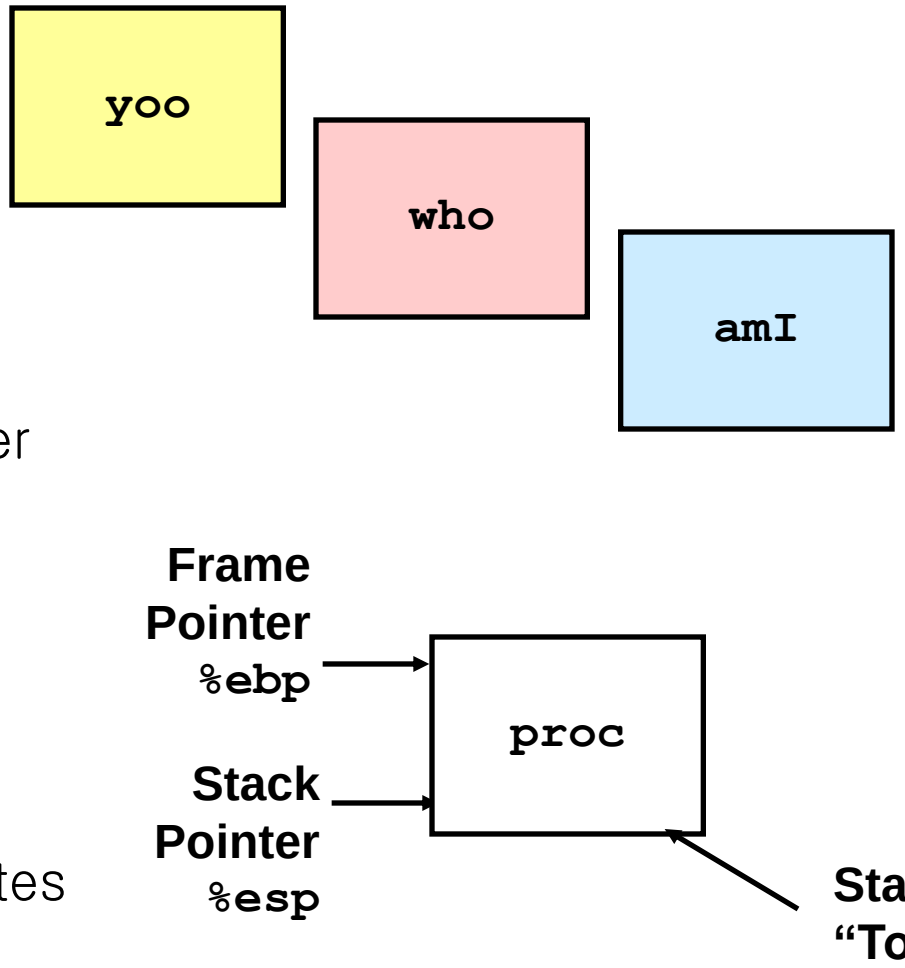
Call Chain



  - Procedure `amI` recursive

Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when enter procedure
    - "Set-up" code
  - Deallocated when return
    - "Finish" code
- Pointers
  - Stack pointer `%esp` indicates stack top
  - Frame pointer `%ebp` indicates start of current frame

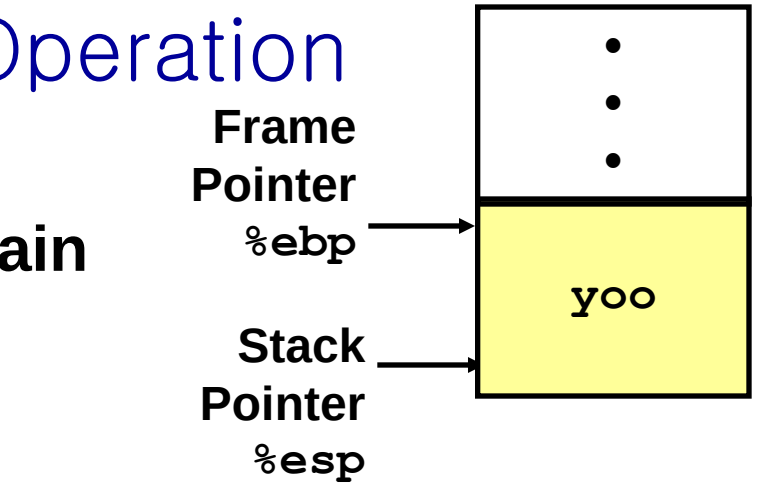


Stack Operation

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

Call Chain

yoo



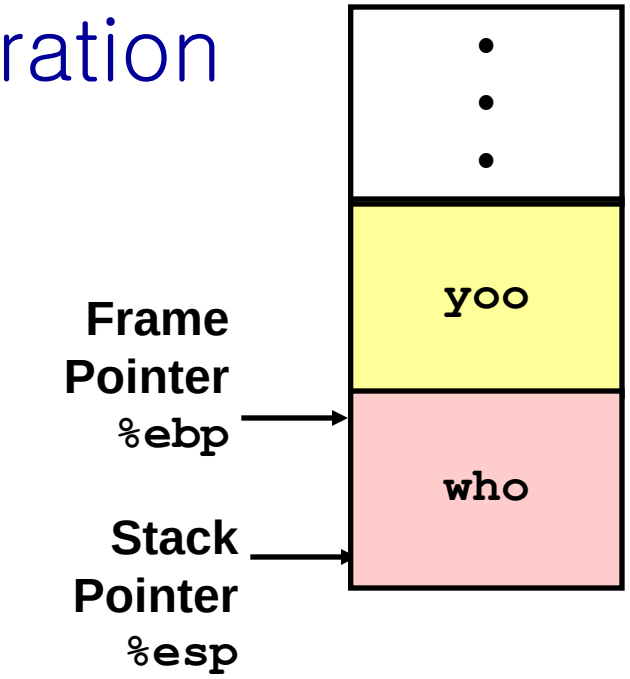
Stack Operation

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

Call Chain

yoo

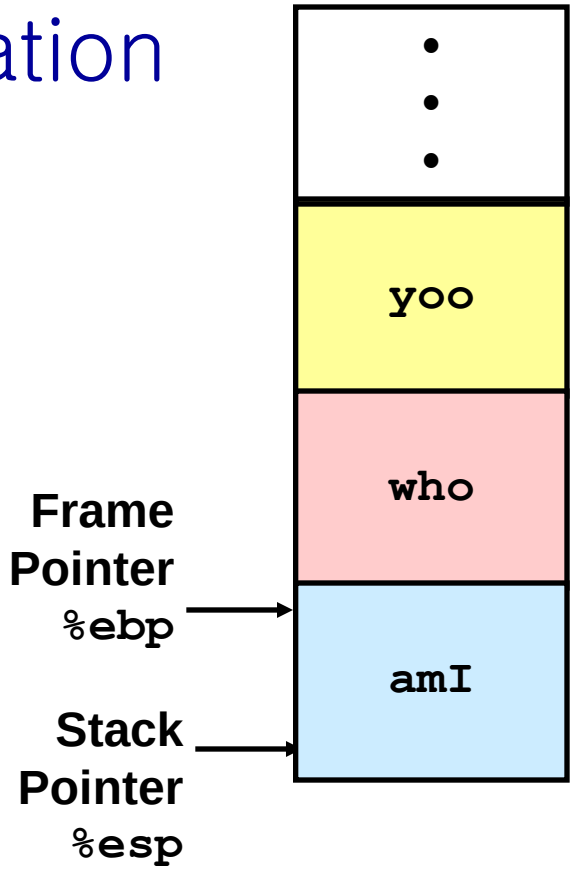
who



Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

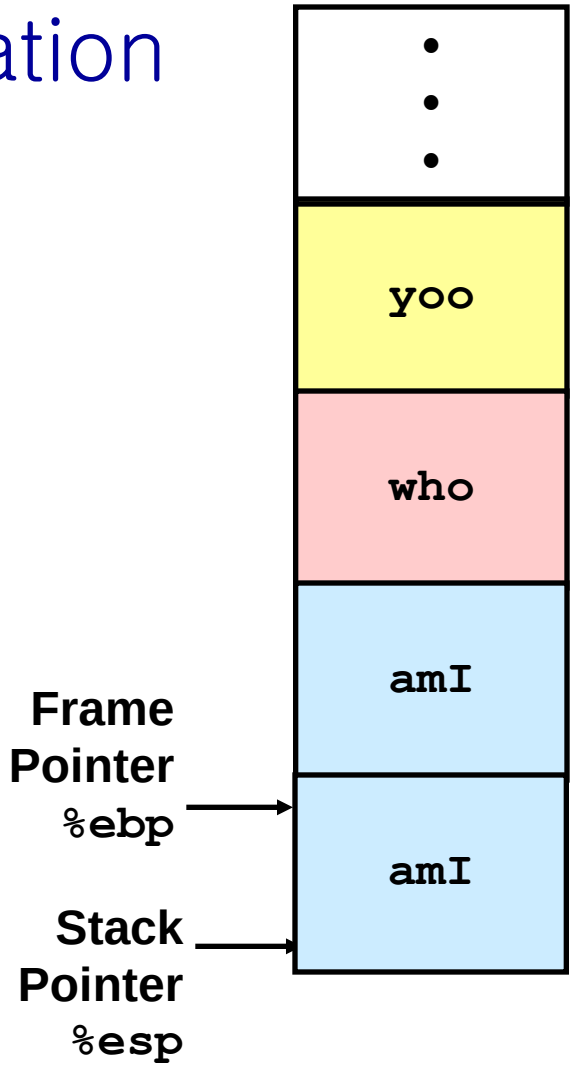


Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

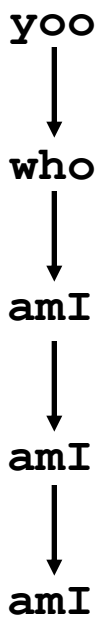
yoo
↓
who
↓
amI
↓
amI

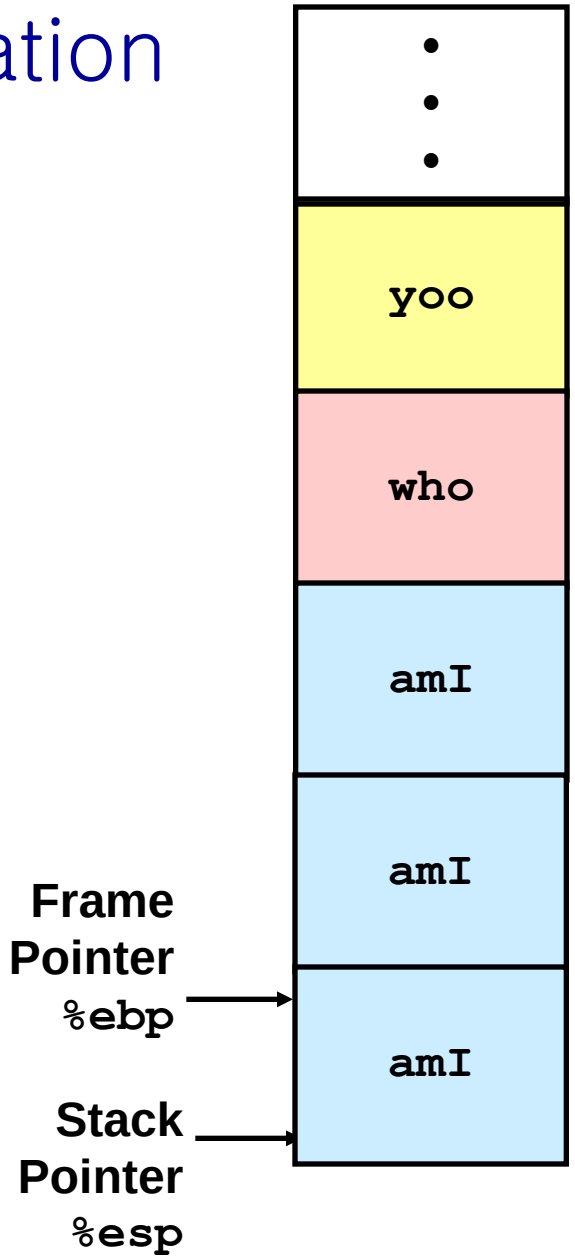


Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

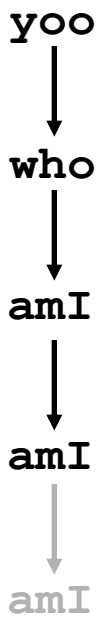


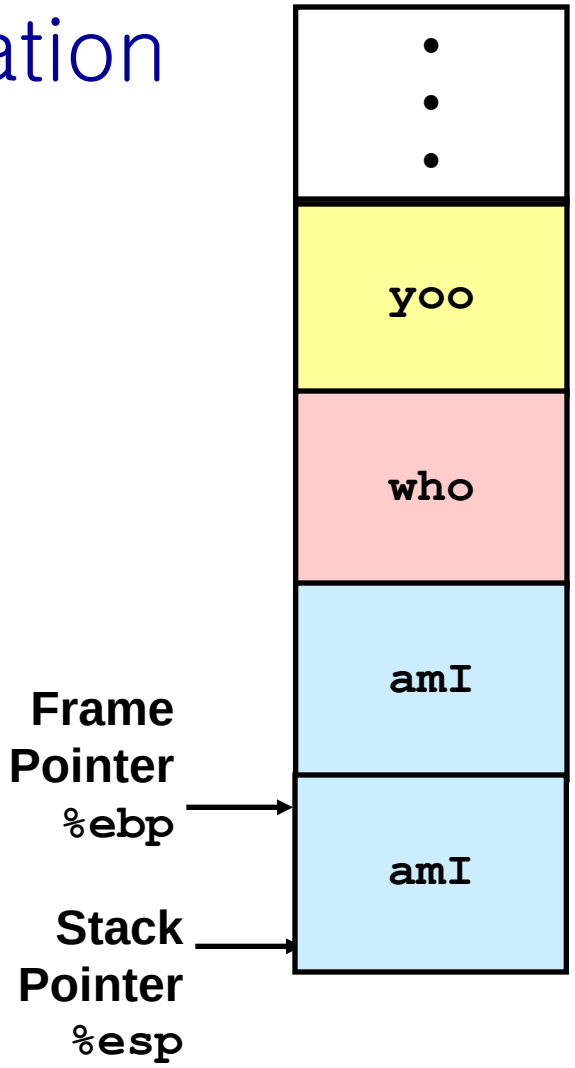


Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

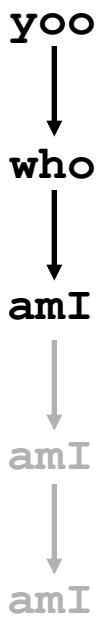


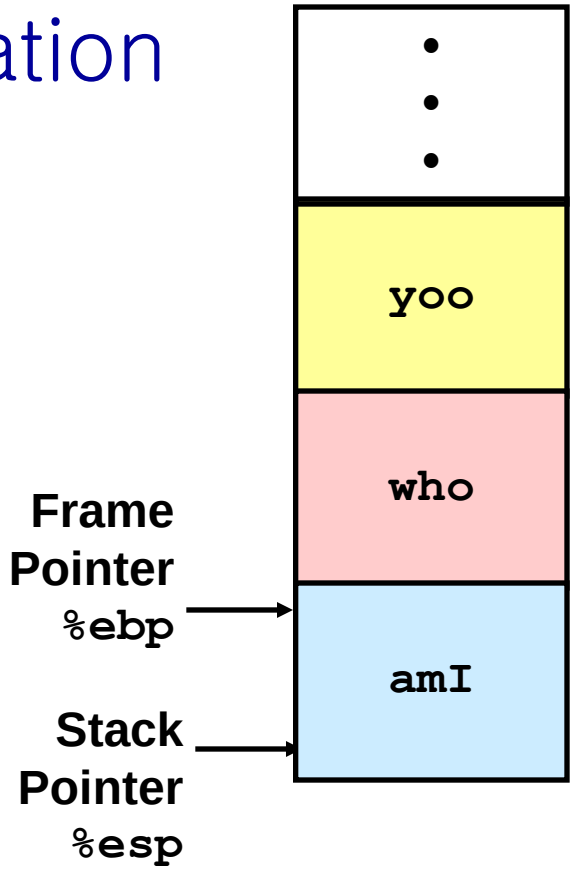


Stack Operation

```
amI(...)
{
    •
    •
    amI();
    •
    •
}
```

Call Chain

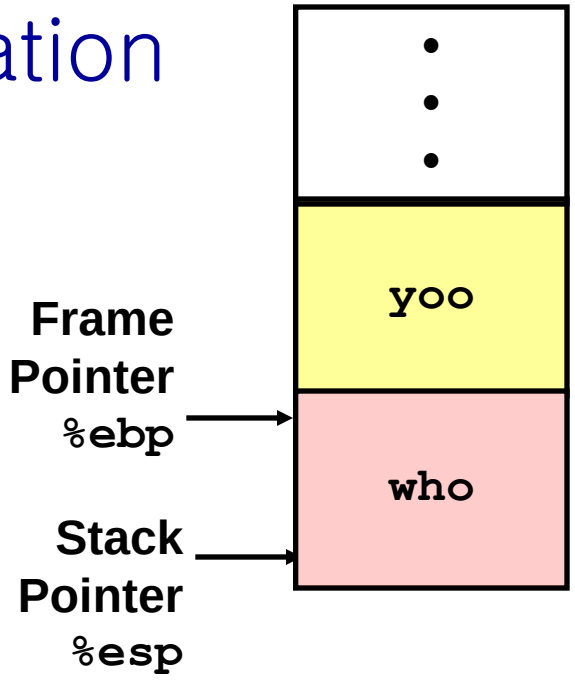




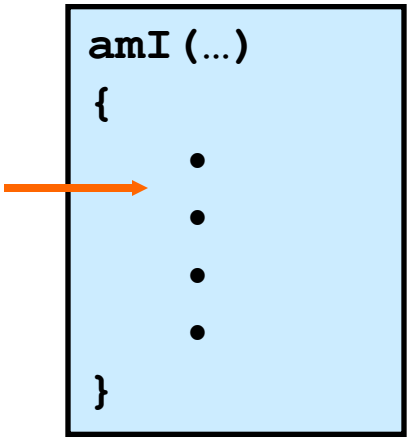
Stack Operation

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

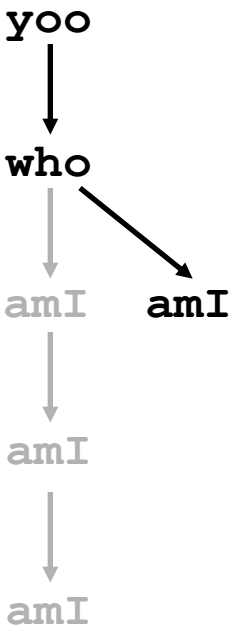
Call Chain

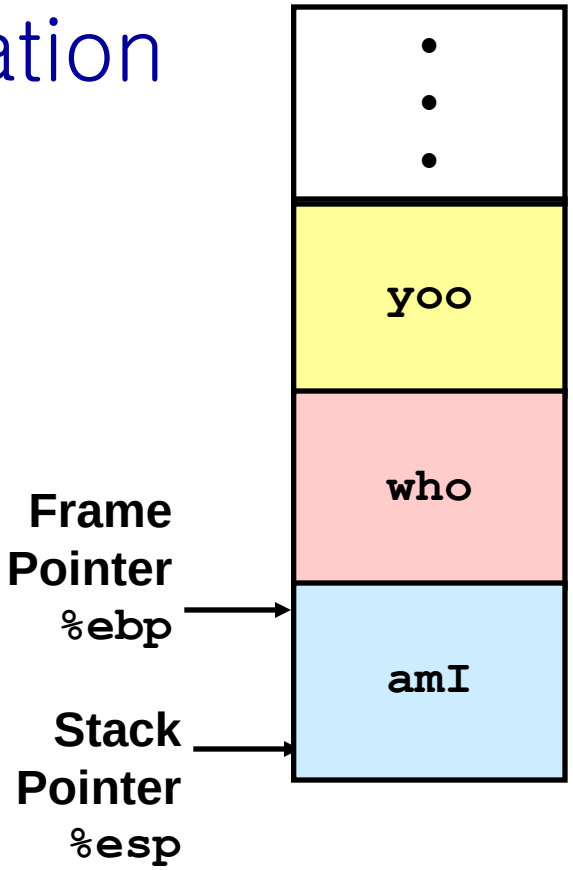


Stack Operation



Call Chain

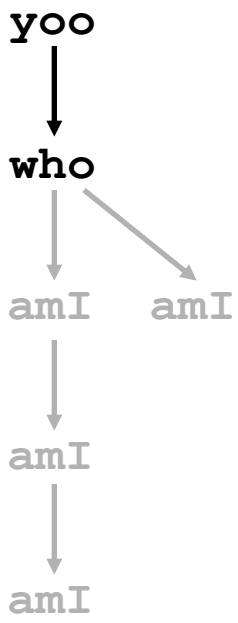


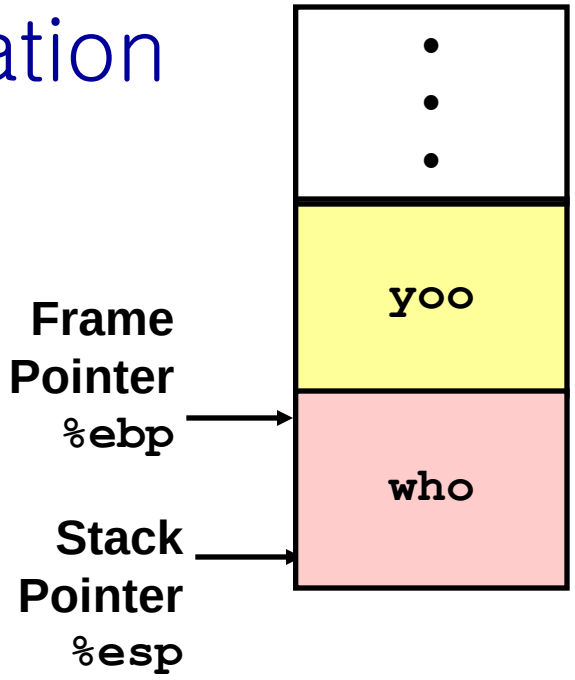


Stack Operation

```
who(…)
{
    • • •
    amI();
    • • •
    amI();
    • • •
}
```

Call Chain

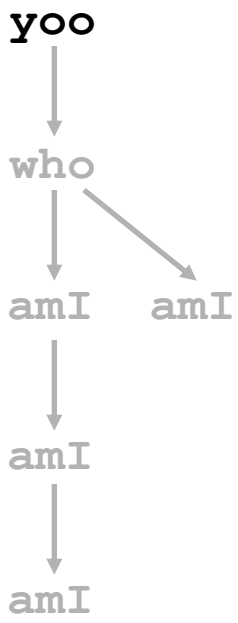


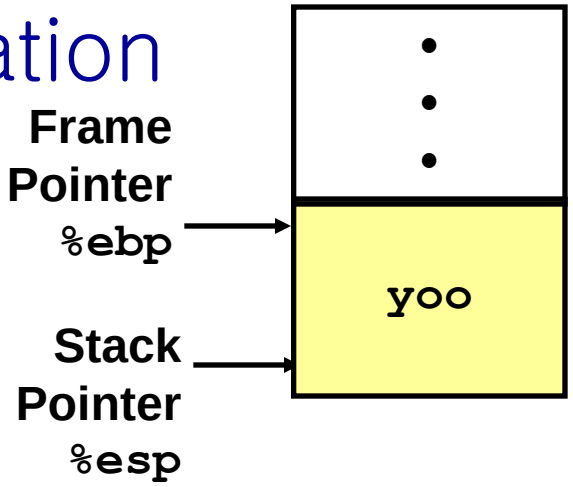


Stack Operation

```
yoo(...)
{
    •
    •
    who();
    •
    •
}
```

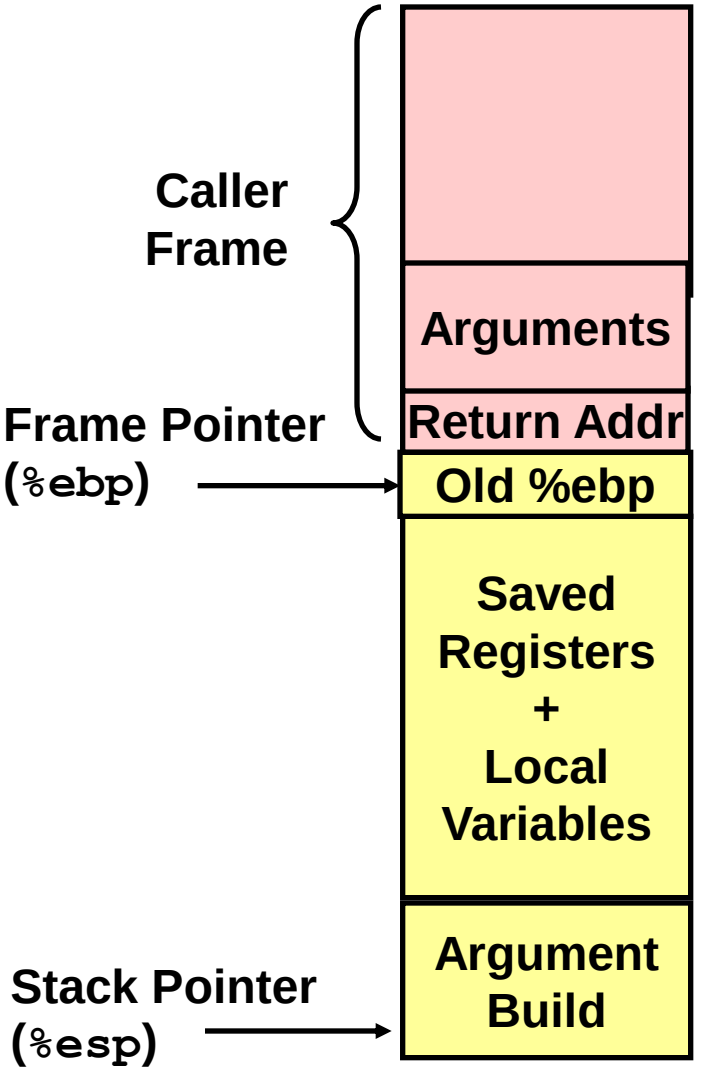
Call Chain





IA32/Linux Stack Frame

- Current Stack Frame ("Top" to Bottom)
  - Parameters for function about to call
    - "Argument build"
  - Local variables
    - If can't keep in registers
  - Saved register context
  - Old frame pointer
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call



Revisiting swap

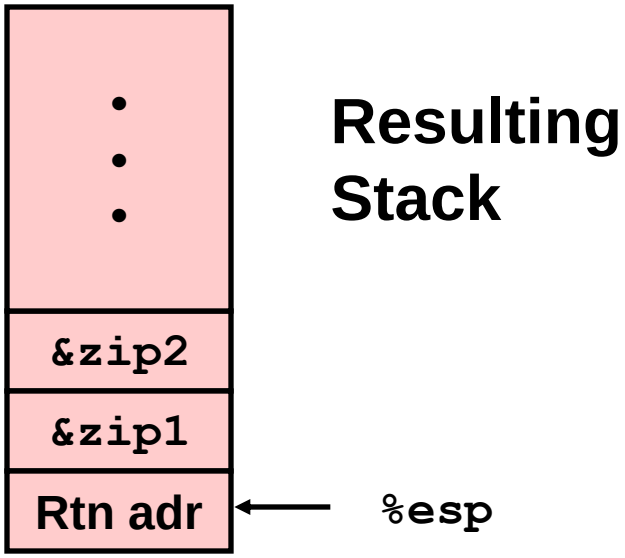
```
int zip1 = 15213;
int zip2 = 91125;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

Calling swap from call_swap

```
call_swap:
    • • •
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    • • •
```



Revisiting swap

```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx

    movl 12(%ebp),%ecx
    movl 8(%ebp),%edx
    movl (%ecx),%eax
    movl (%edx),%ebx
    movl %eax,(%edx)
    movl %ebx,(%ecx)

    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

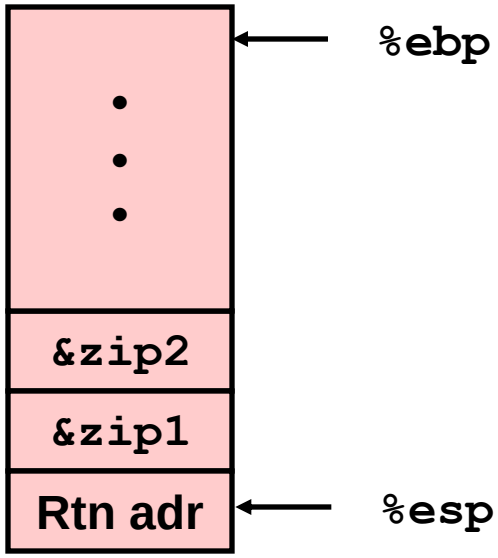
Set Up

Body

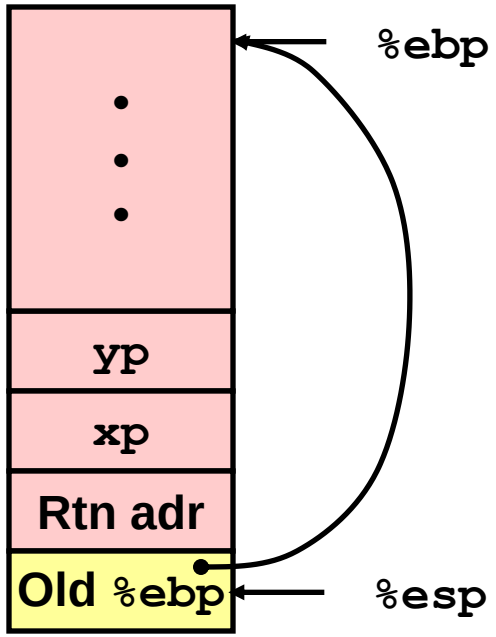
Finish

swap Setup #1

Entering Stack



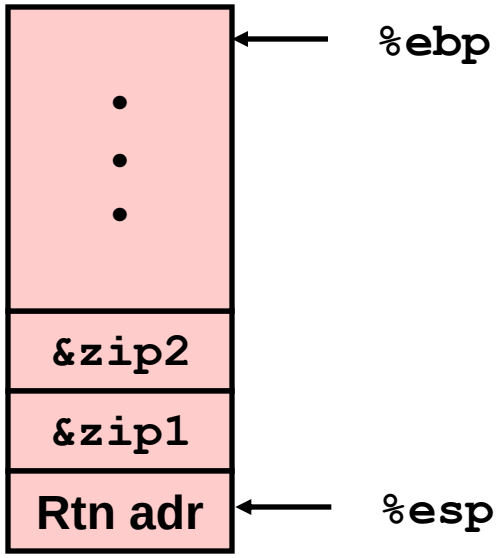
Resulting Stack



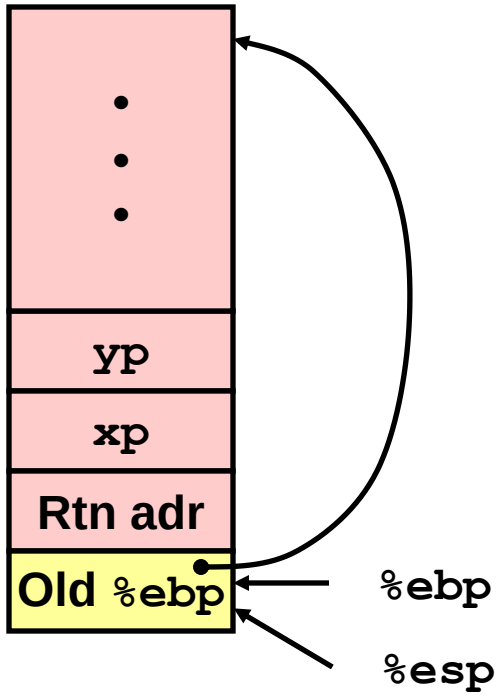

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



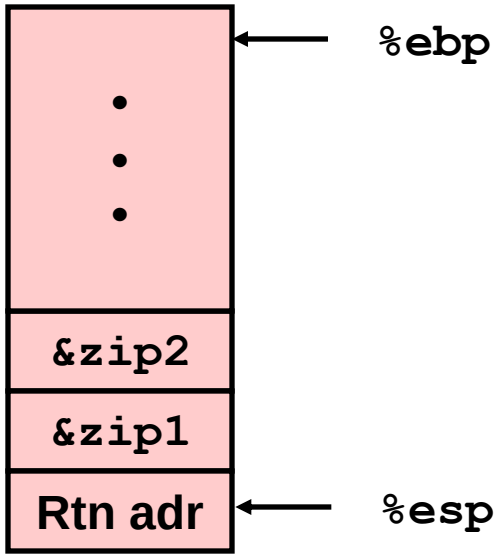
Resulting Stack



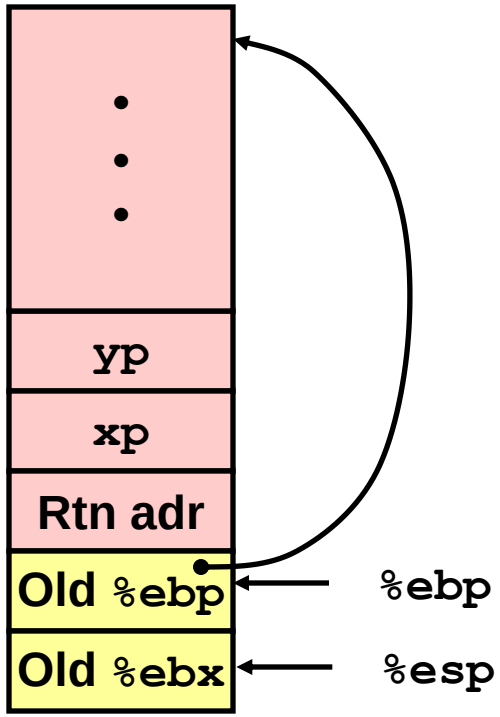

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack

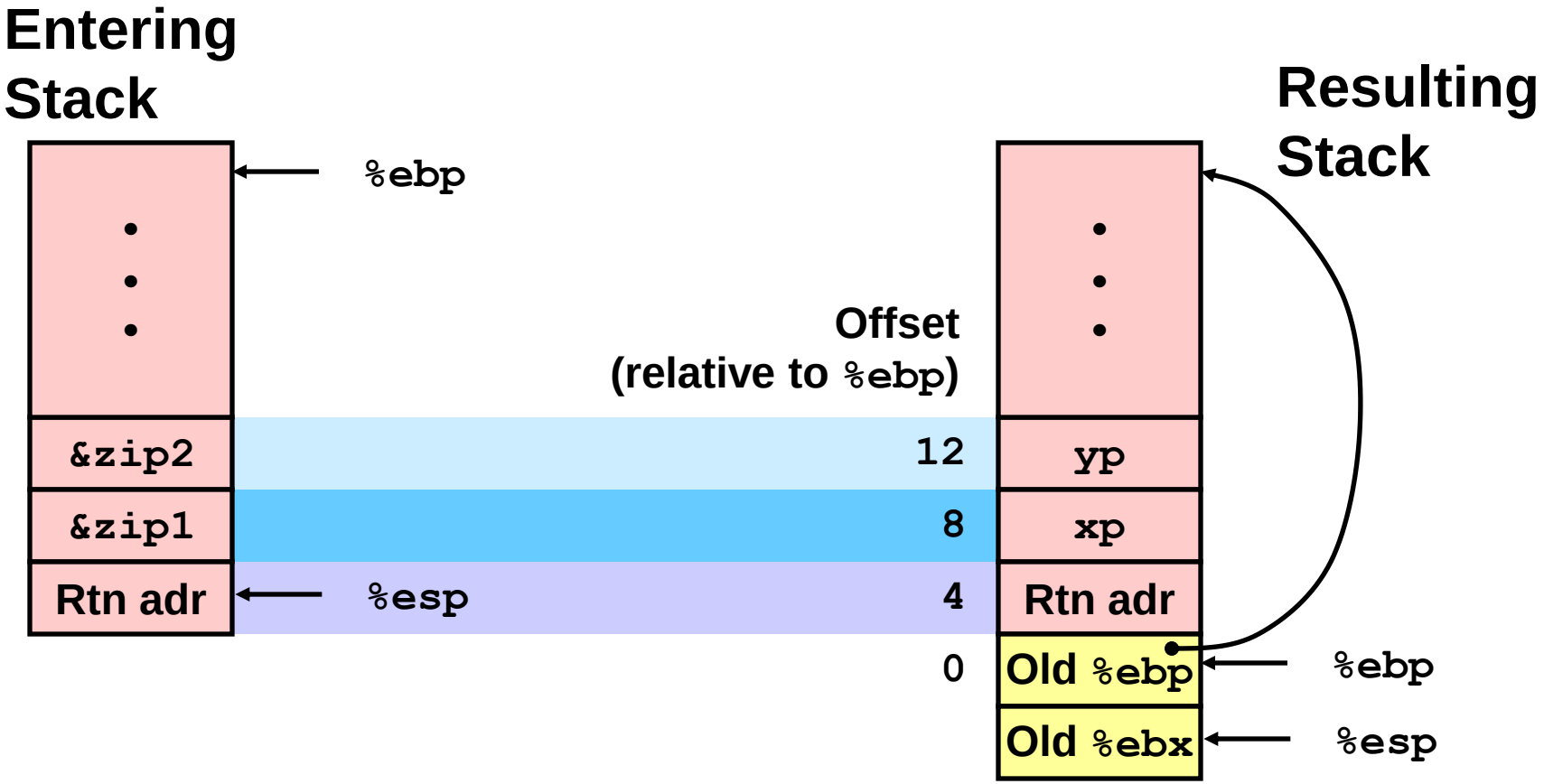


Resulting Stack




```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

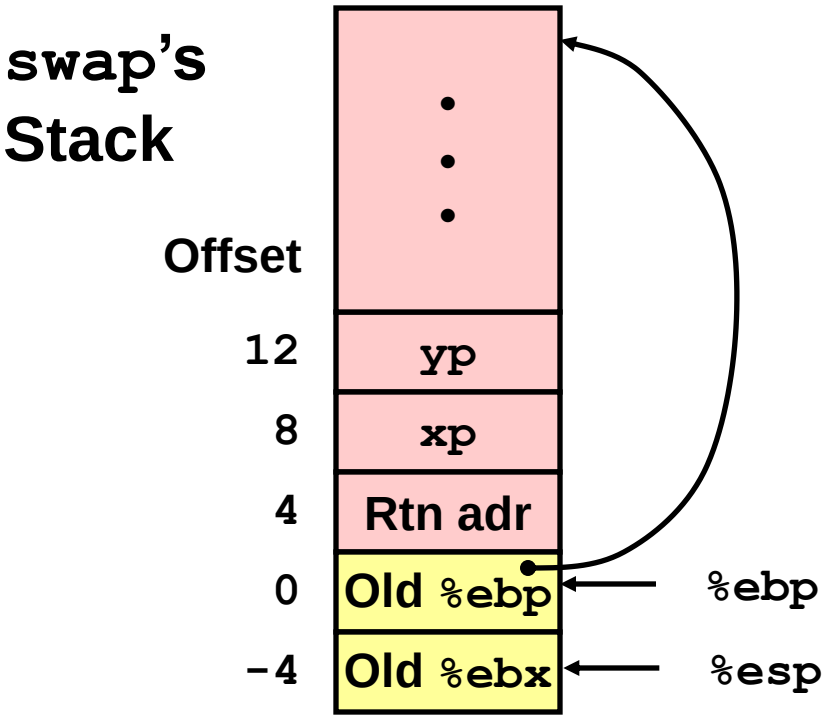
Effect of swap Setup

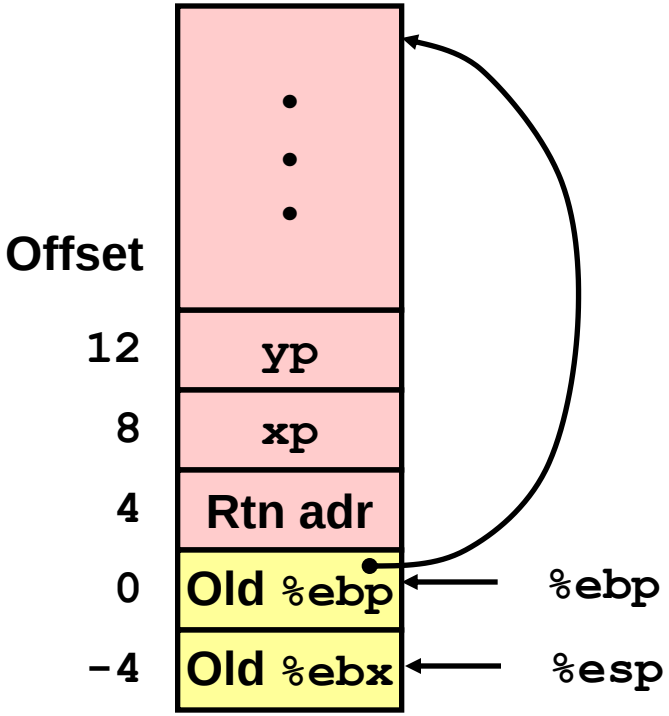



```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

} Body

swap Finish #1

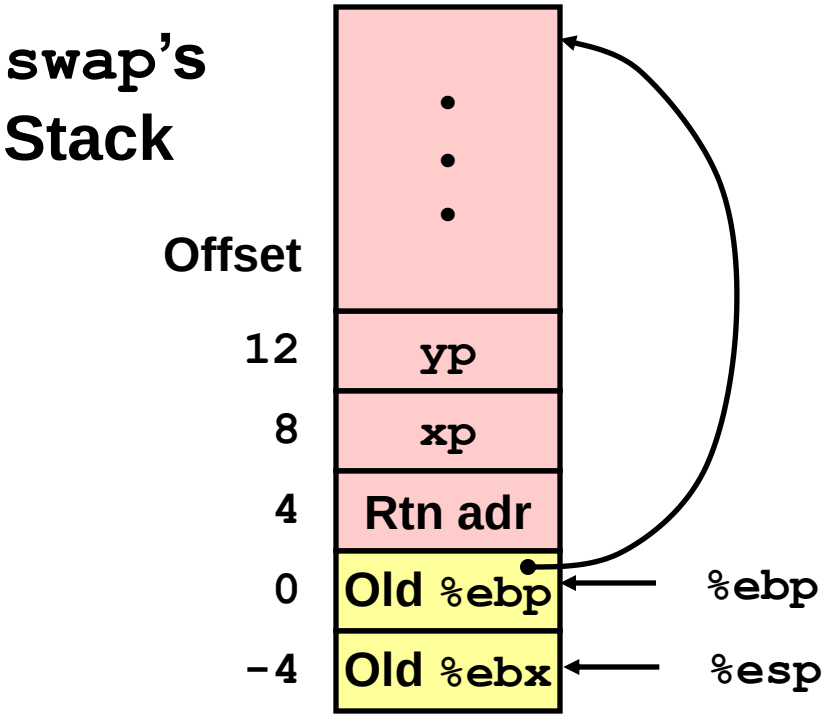


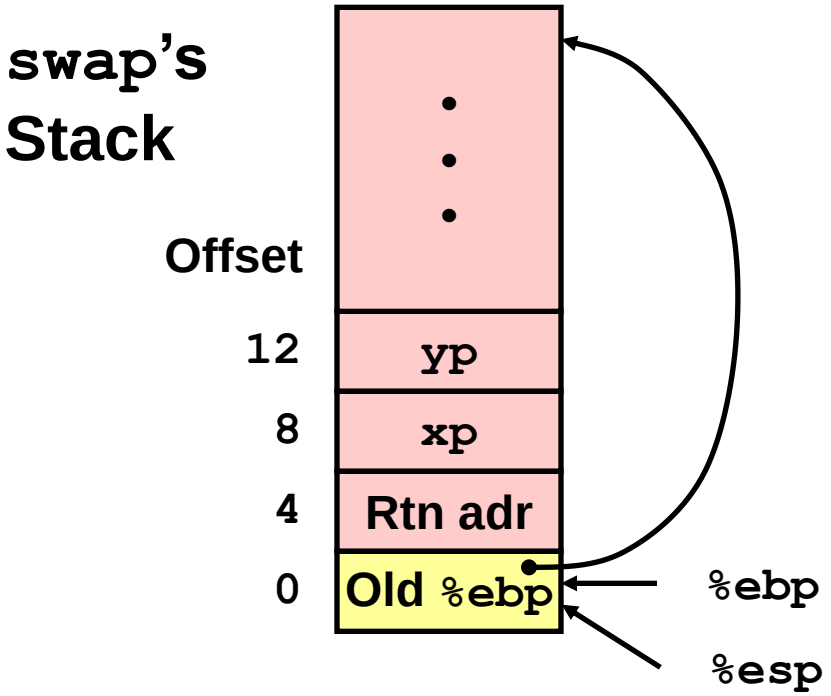



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register `%ebx`

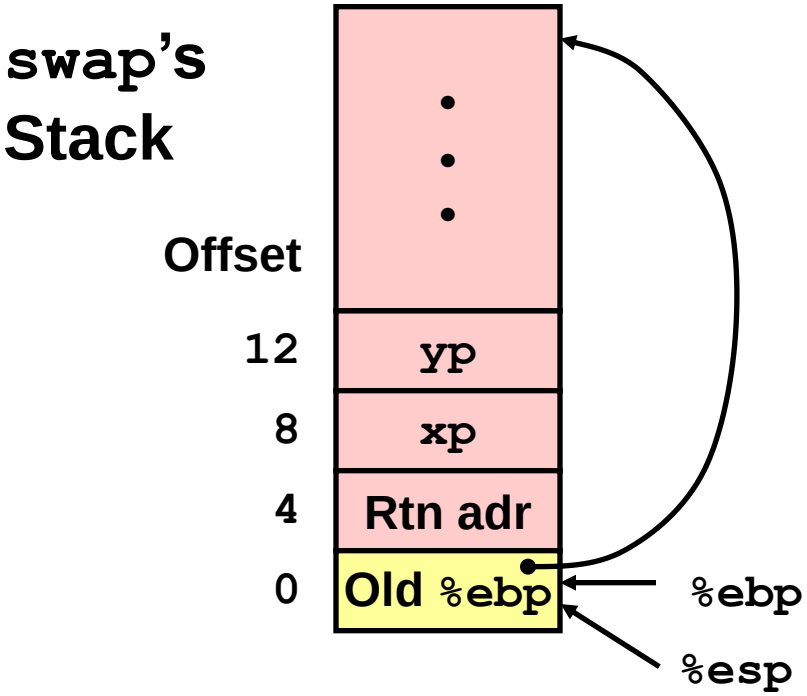
swap Finish #2

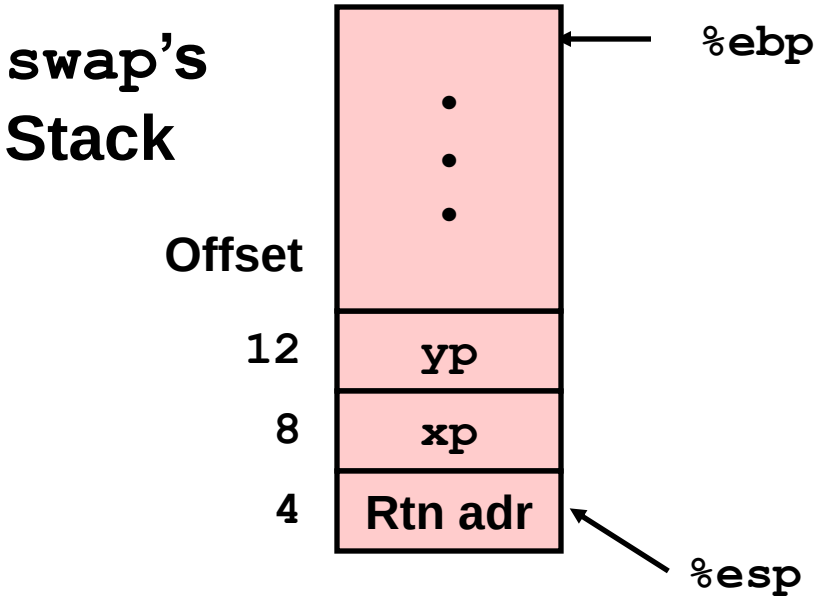





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

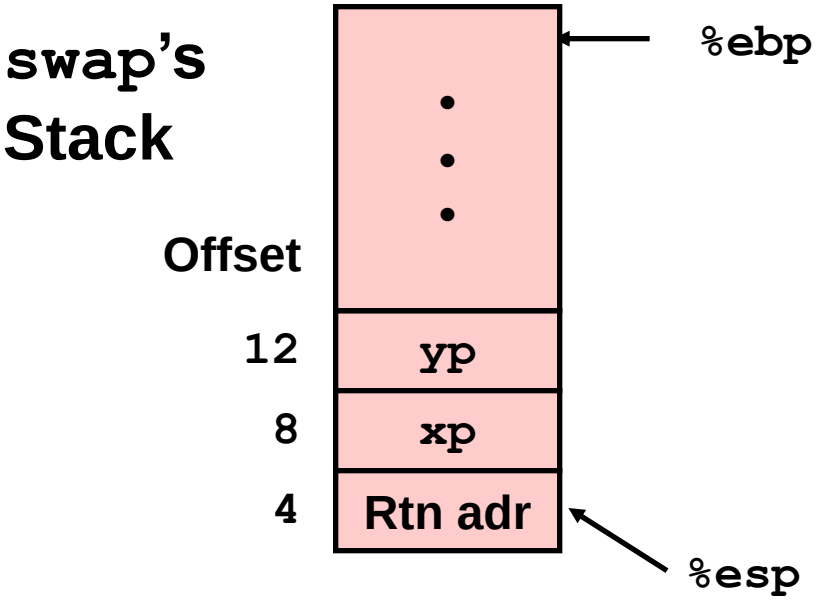
swap Finish #3

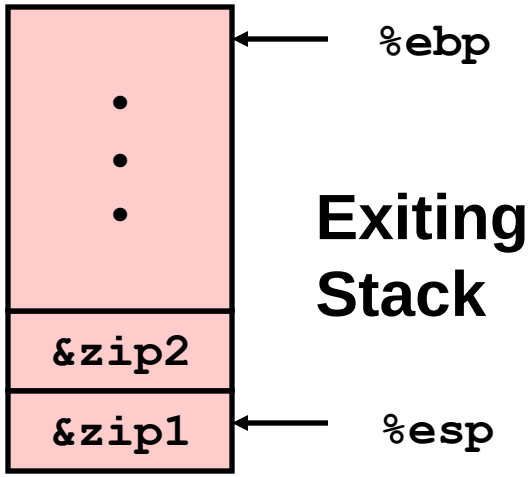





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #4






```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register `%ebx`
  - Didn't do so for `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can Register be Used for Temporary Storage?

```
yoo:
    • • •
    movl $15213, %edx
    call who
    addl %edx, %eax
    • • •
    ret
```

```
who:
    • • •
    movl 8(%ebp), %edx
    addl $91125, %edx
    • • •
    ret
```

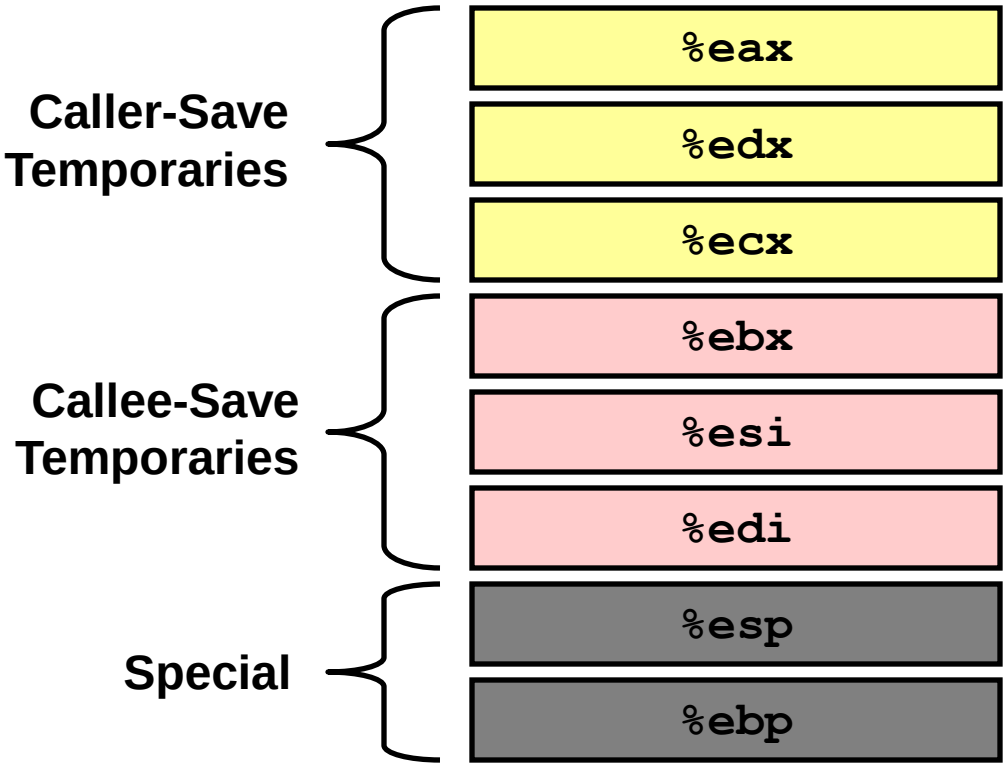
  - Contents of register `%edx` overwritten by `who`

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the *caller*, `who` is the *callee*
- Can Register be Used for Temporary Storage?
- Conventions
  - "Caller Save"
    - Caller saves temporary in its frame before calling
  - "Callee Save"
    - Callee saves temporary in its frame before using

IA32/Linux Register Usage

- Integer Registers
  - Two have special uses
    `%ebp`, `%esp`
  - Three managed as callee-save
    `%ebx`, `%esi`, `%edi`
    - Old values saved on stack prior to using
  - Three managed as caller-save
    `%eax`, `%edx`, `%ecx`
    - Do what you please, but expect any callee to do so, as well
  - Register `%eax` also stores returned value



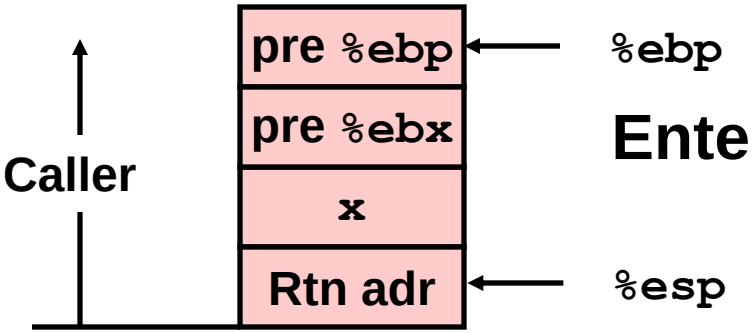
Recursive Factorial

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1);
  return rval * x;
}
```

- Registers
  - `%eax` used without first saving
  - `%ebx` used, but save at beginning & restore at end

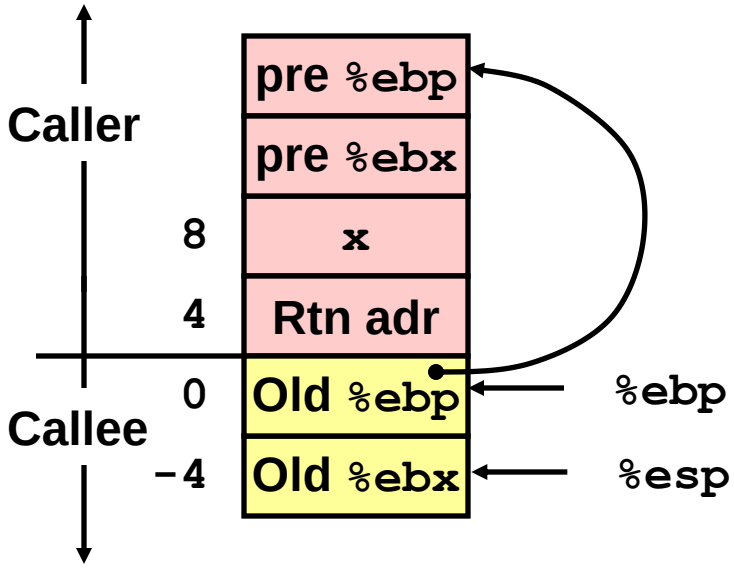
```
.globl rfact
    .type
rfact,@function
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ebx
    cmpl $1,%ebx
    jle .L78
    leal -1(%ebx),%eax
    pushl %eax
    call rfact
    imull %ebx,%eax
    jmp .L79
    .align 4
.L78:
    movl $1,%eax
.L79:
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Rfact Stack Setup



Entering Stack

```
rfact:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```



Rfact Body

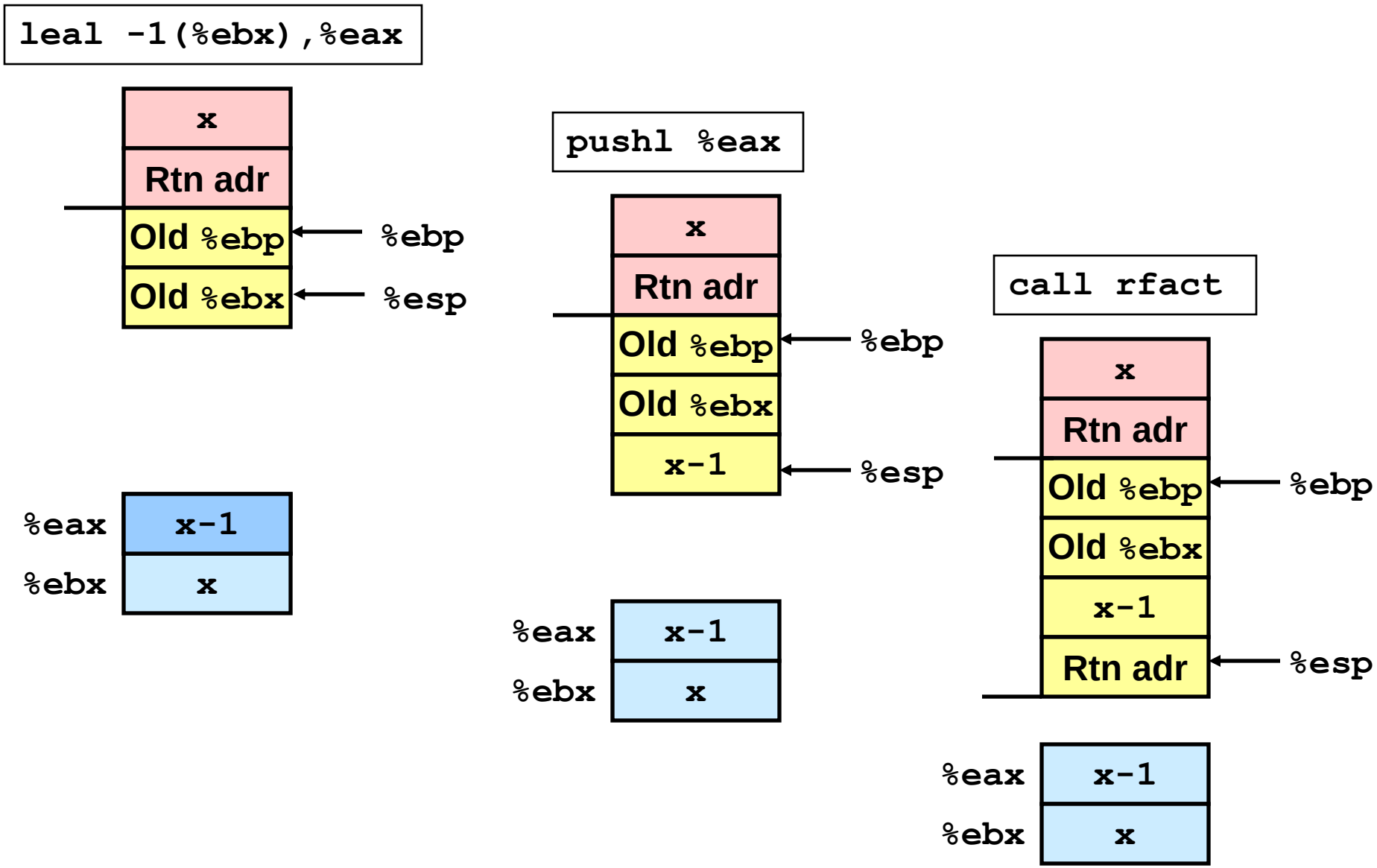
Recursion

```
  movl 8(%ebp),%ebx    # ebx = x
  cmpl $1,%ebx         # Compare x : 1
  jle .L78             # If <= goto Term
  leal -1(%ebx),%eax   # eax = x-1
  pushl %eax           # Push x-1
  call rfact           # rfact(x-1)
  imull %ebx,%eax      # rval * x
  jmp .L79             # Goto done
.L78:                # Term:
  movl $1,%eax         # return val = 1
.L79:                # Done:
```

```
int rfact(int x)
{
  int rval;
  if (x <= 1)
    return 1;
  rval = rfact(x-1) ;
  return rval * x;
}
```

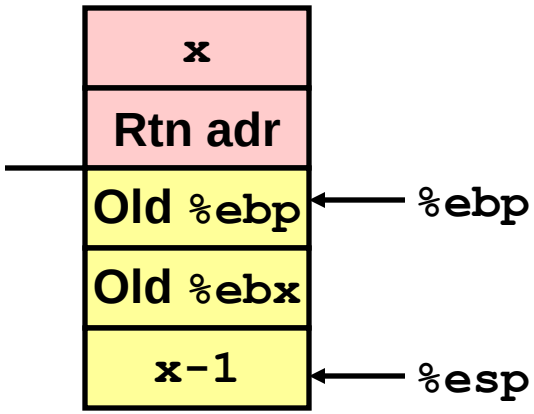
- Registers
  - `%ebx` Stored value of x
  - `%eax`
    - Temporary value of `x-1`
    - Returned value from `rfact(x-1)`
    - Returned value from this call

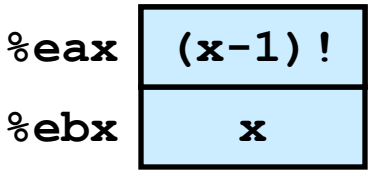
Rfact Recursion



Rfact Result

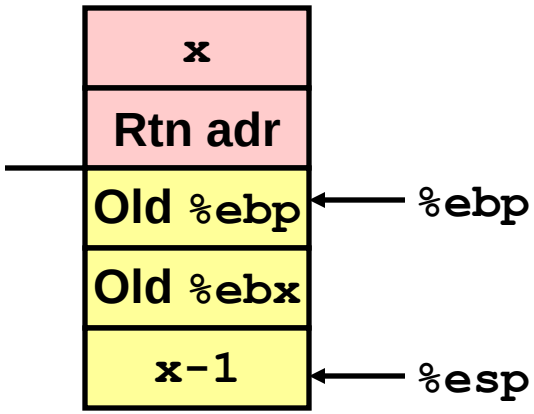
Return from Call

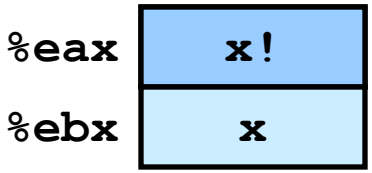




Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

```
imull %ebx,%eax
```

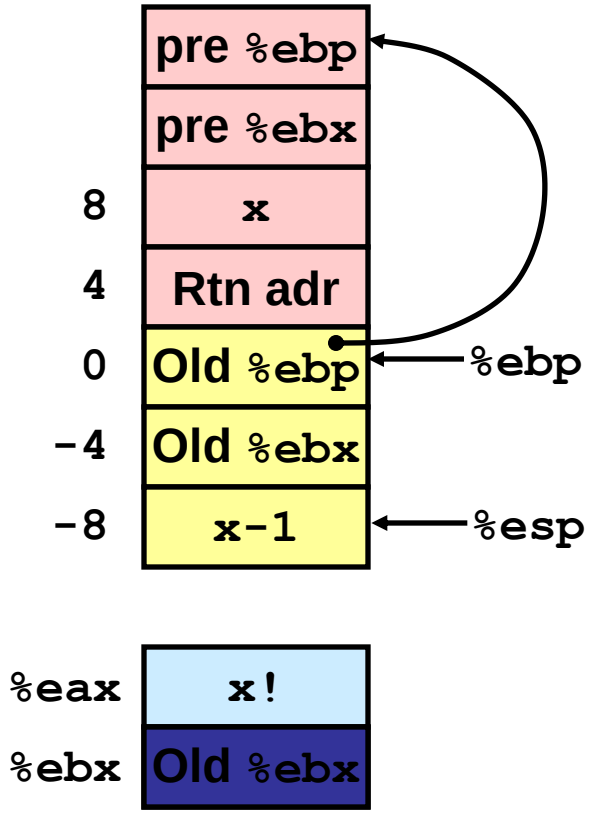


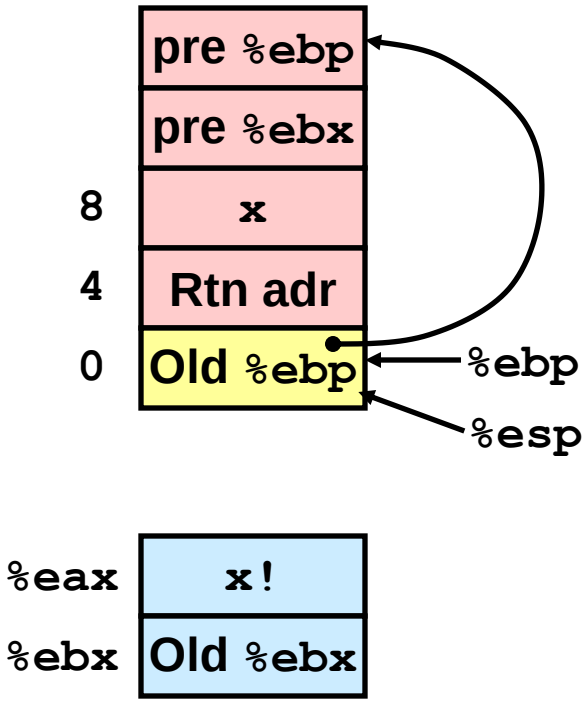


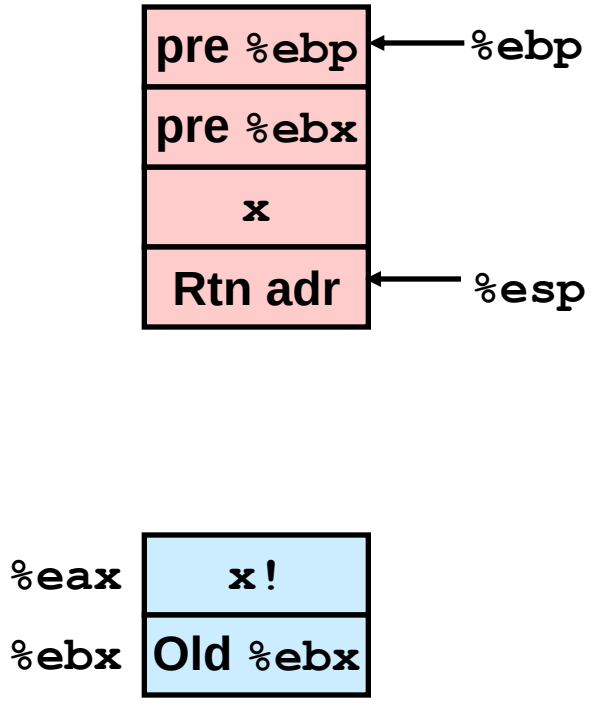
Rfact Completion

leave

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```







Pointer Code

Recursive Procedure

```
void s_helper
  (int x, int *accum)
{
  if (x <= 1)
    return;
  else {
    int z = *accum * x;
    *accum = z;
    s_helper (x-1,accum);
  }
}
```

Top-Level Call

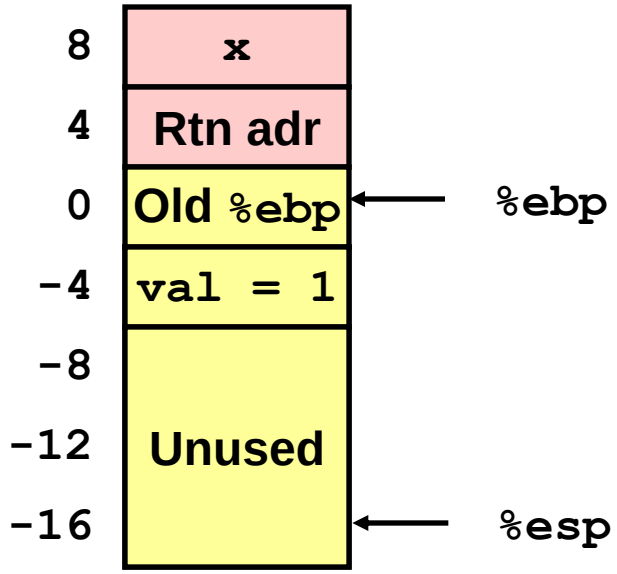
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

- Pass pointer to update location

Creating & Initializing Pointer

Initial part of `sfact`

```
_sfact:
   pushl %ebp              # Save %ebp
   movl %esp,%ebp          # Set %ebp
   subl $16,%esp           # Add 16 bytes
   movl 8(%ebp),%edx       # edx = x
   movl $1,-4(%ebp)        # val = 1
```



- Using Stack for Local Variable
  - Variable `val` must be stored on stack
    - Need to create pointer to it
  - Compute pointer as
    - `-4(%ebp)`
  - Push on stack as second argument

```
int sfact(int x)
{
   int val = 1;
   s_helper(x, &val);
   return val;
}
```

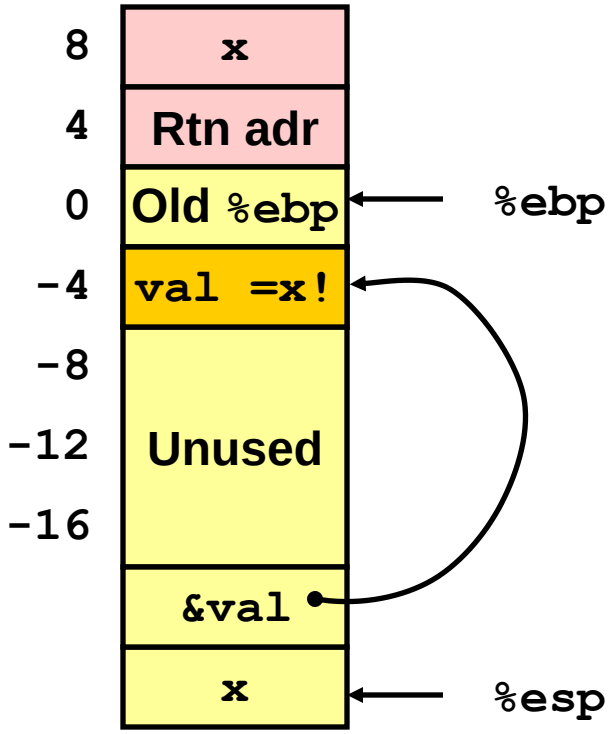
Passing Pointer

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
call s_helper       # call
movl -4(%ebp),%eax  # Return val
• • •               # Finish
```

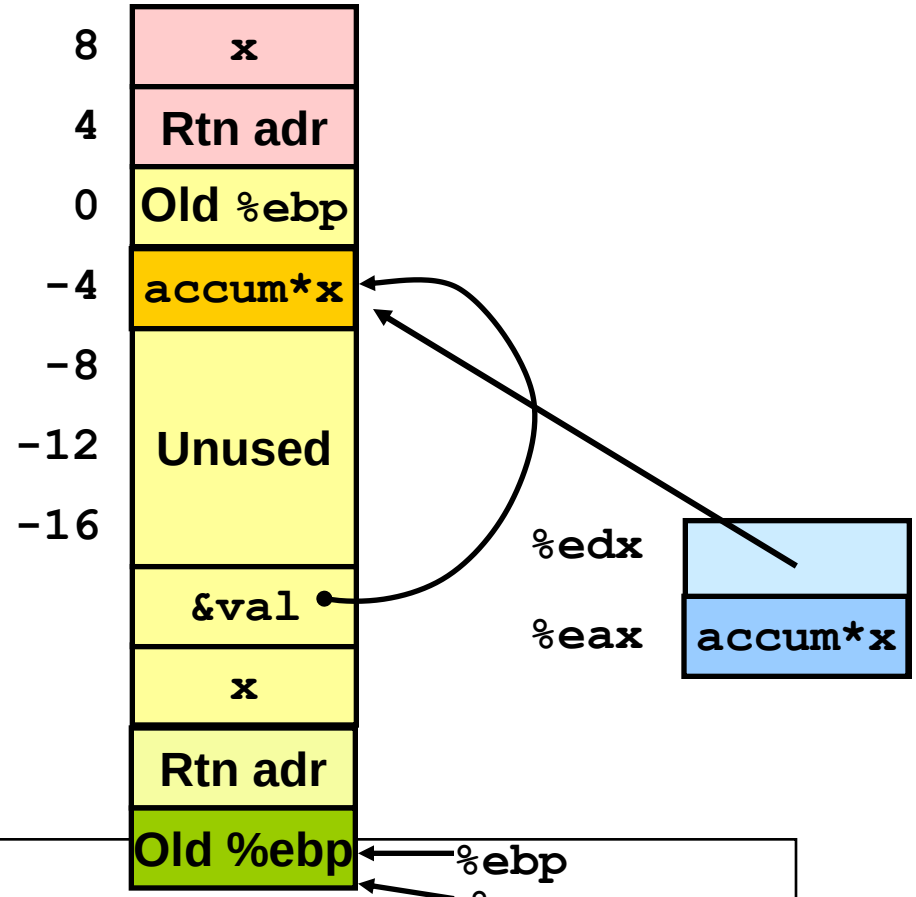
```
int sfact(int x)
{
  int val = 1;
  s_helper(x, &val);
  return val;
}
```

Stack at time of call



Using Pointer

```
void s_helper
  (int x, int *accum)
{
  • • •
    int z = *accum * x;
    *accum = z;
  • • •
}
```



movl 8(%ebp), %ecx
movl 12(%ebp), %edx

```
• • •
movl %ecx,%eax     # z = x
imull (%edx),%eax  # z *= *accum
movl %eax,(%edx)   # *accum = z
• • •
```

- Register `%ecx` holds `x`
- Register `%edx` holds pointer to `accum`
  - Use access `(%edx)` to reference memory

Summary

- The Stack Makes Recursion Work
  - Private storage for each *instance* of procedure call
    - Instantiations don't clobber each other
    - Addressing of locals + arguments can be relative to stack positions
  - Can be managed by stack discipline
    - Procedures return in inverse order of calls
- IA32 Procedures: Combination of Instructions + Conventions
  - Call / Ret instructions
  - Register usage conventions
    - Caller / Callee save
    - `%ebp` and `%esp`
  - Stack frame organization conventions