Machine-Level Programming III - Procedures

Today

- IA32 stack discipline
- Register saving conventions
- Creating pointers to local variables

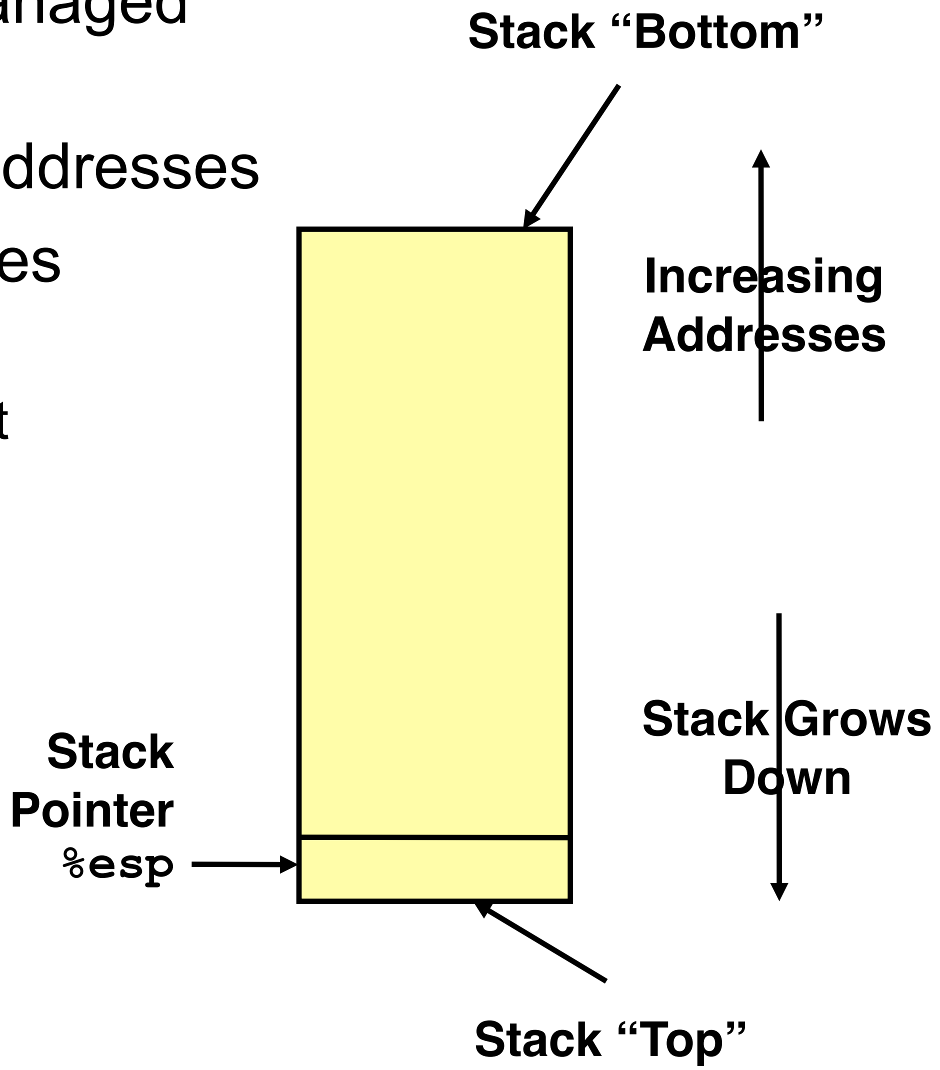
Next time

- Structured data

Chris Riesbeck, Fall 2011

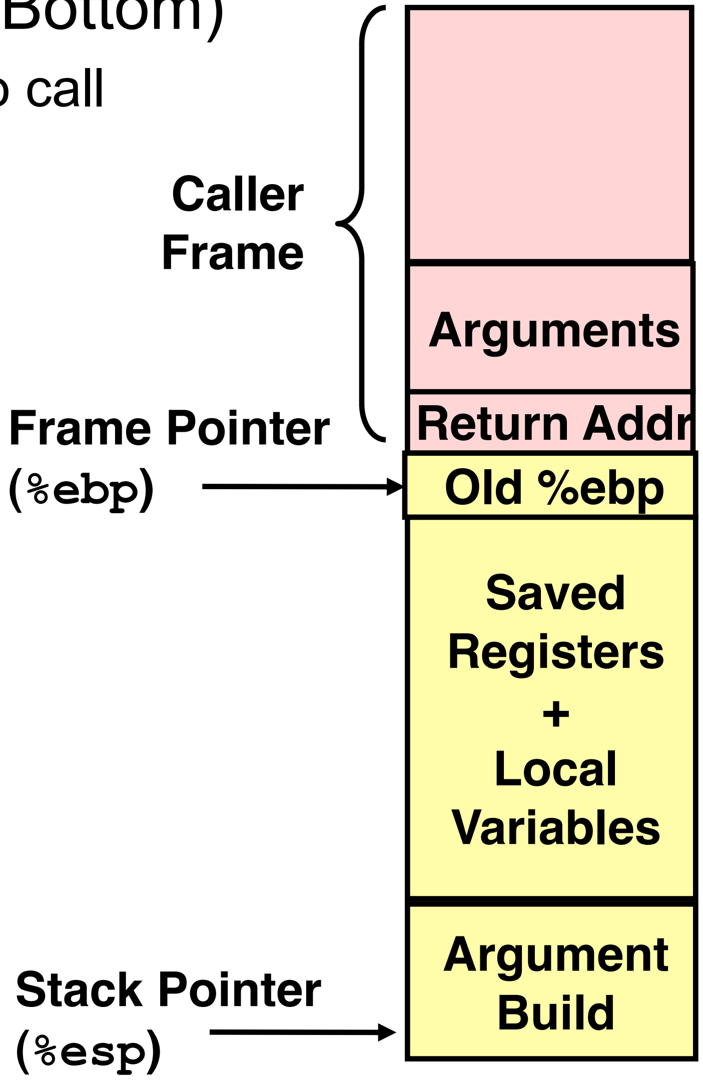
IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
  - address of top element



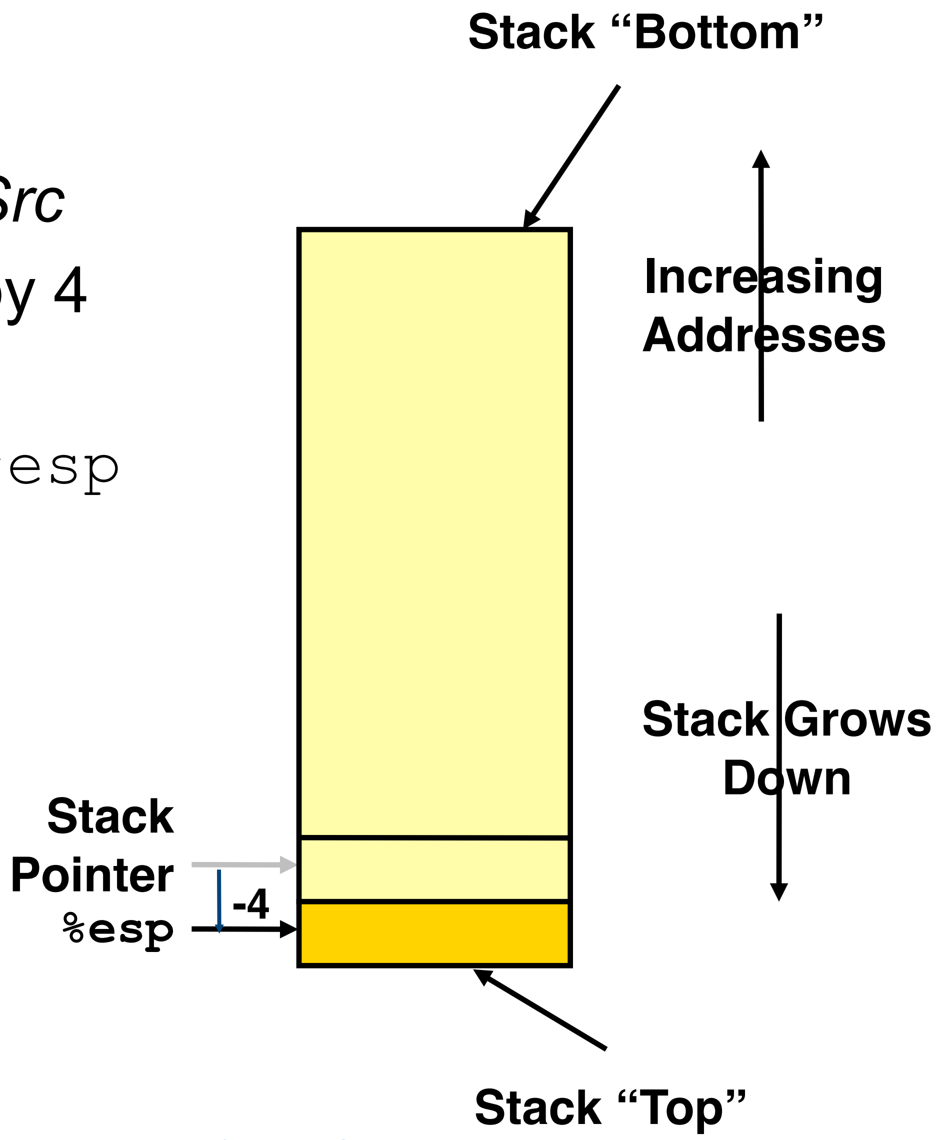
IA32/Linux stack frame

- 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



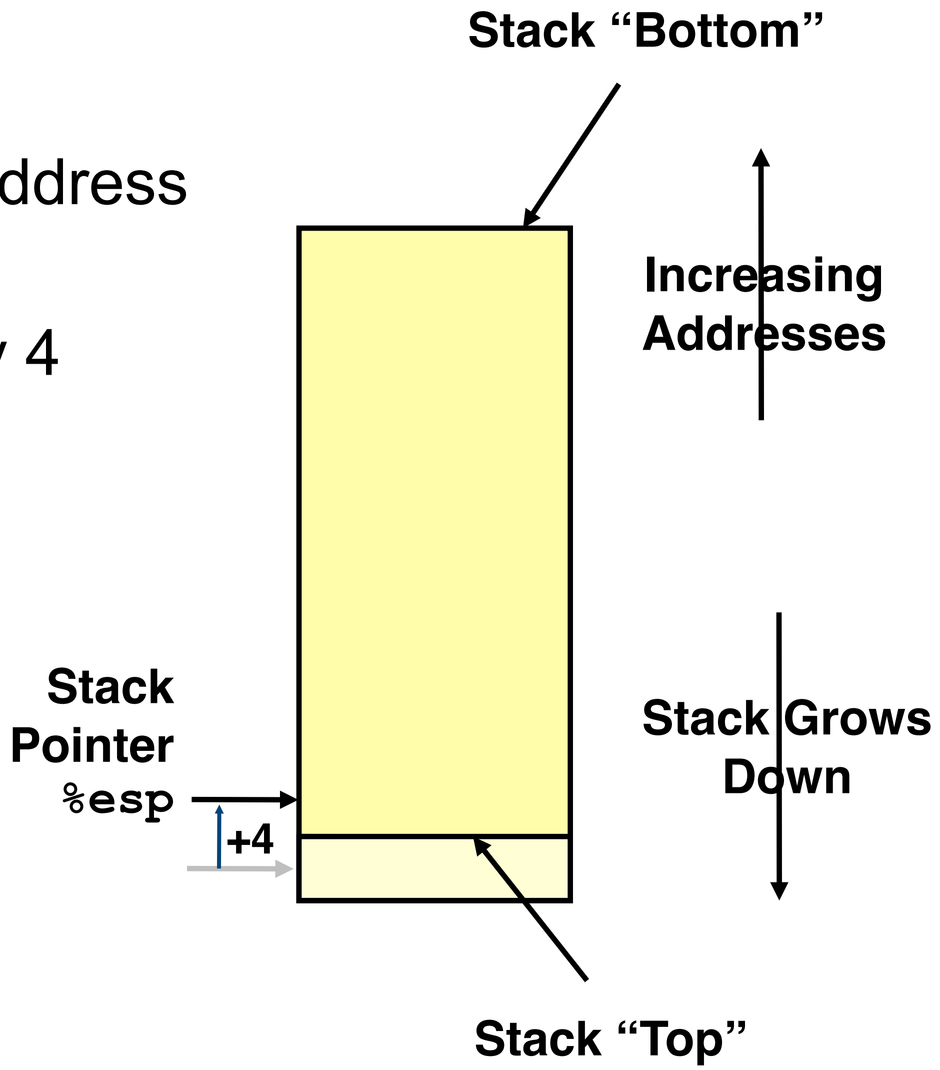
IA32 Stack pushing

- Pushing
  - `pushl` *Src*
  - Fetch operand at *Src*
  - Decrement `%esp` by 4
  - Write operand at address given by `%esp`

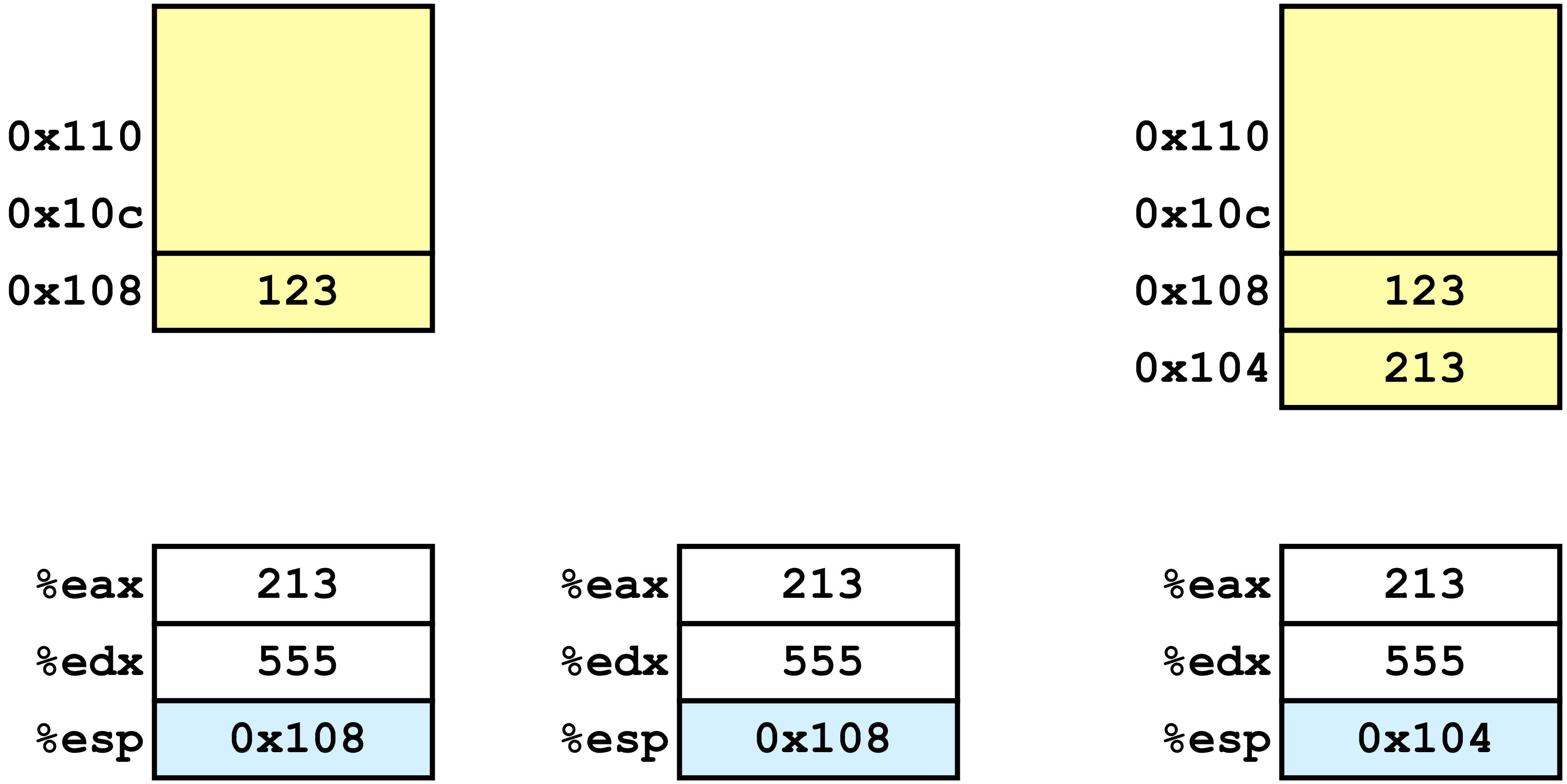


IA32 Stack popping

- Popping
  - `popl` *Dest*
  - Read operand at address given by `%esp`
  - Increment `%esp` by 4
  - Write to *Dest*

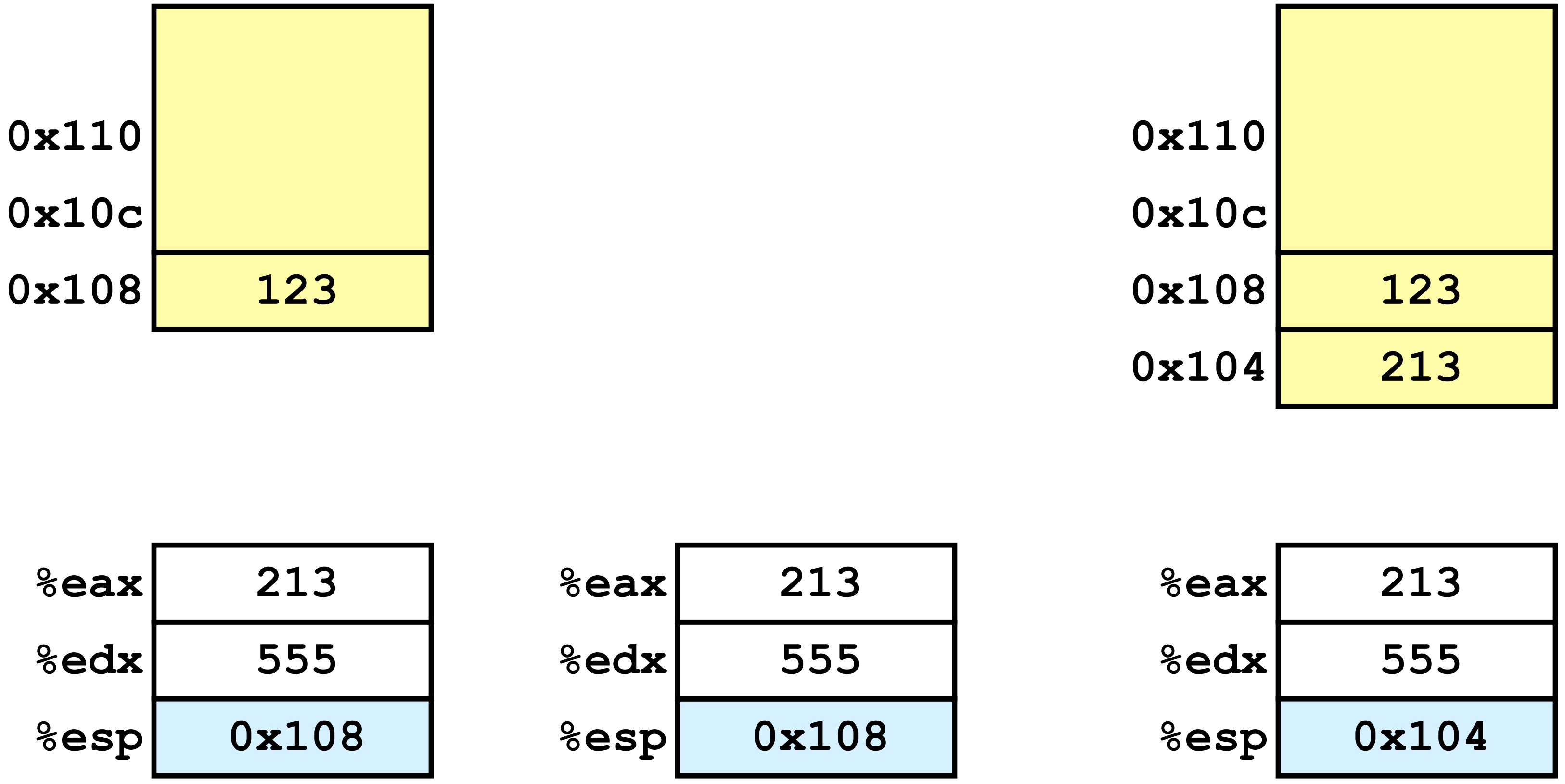


Stack operation examples

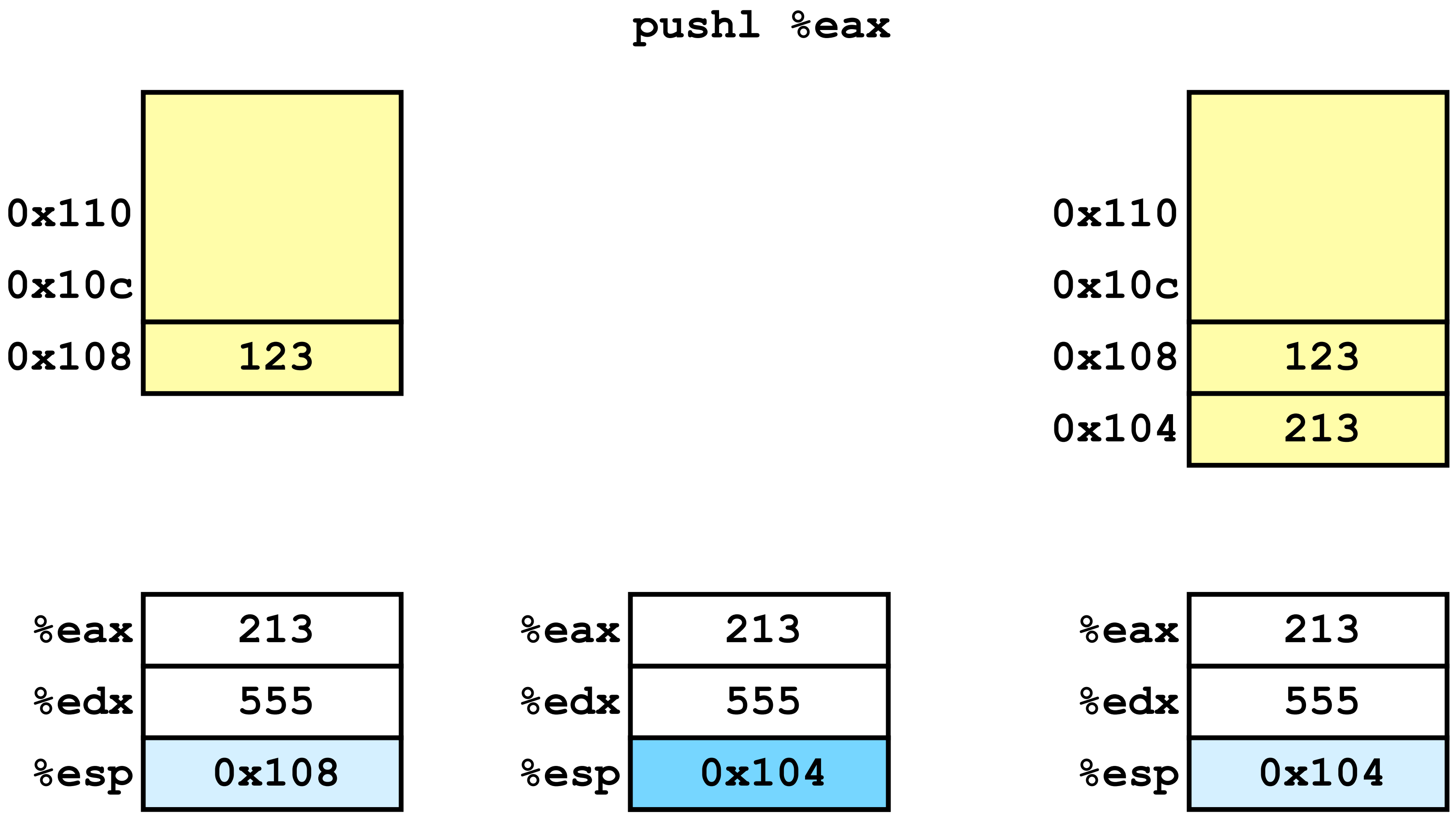


Stack operation examples

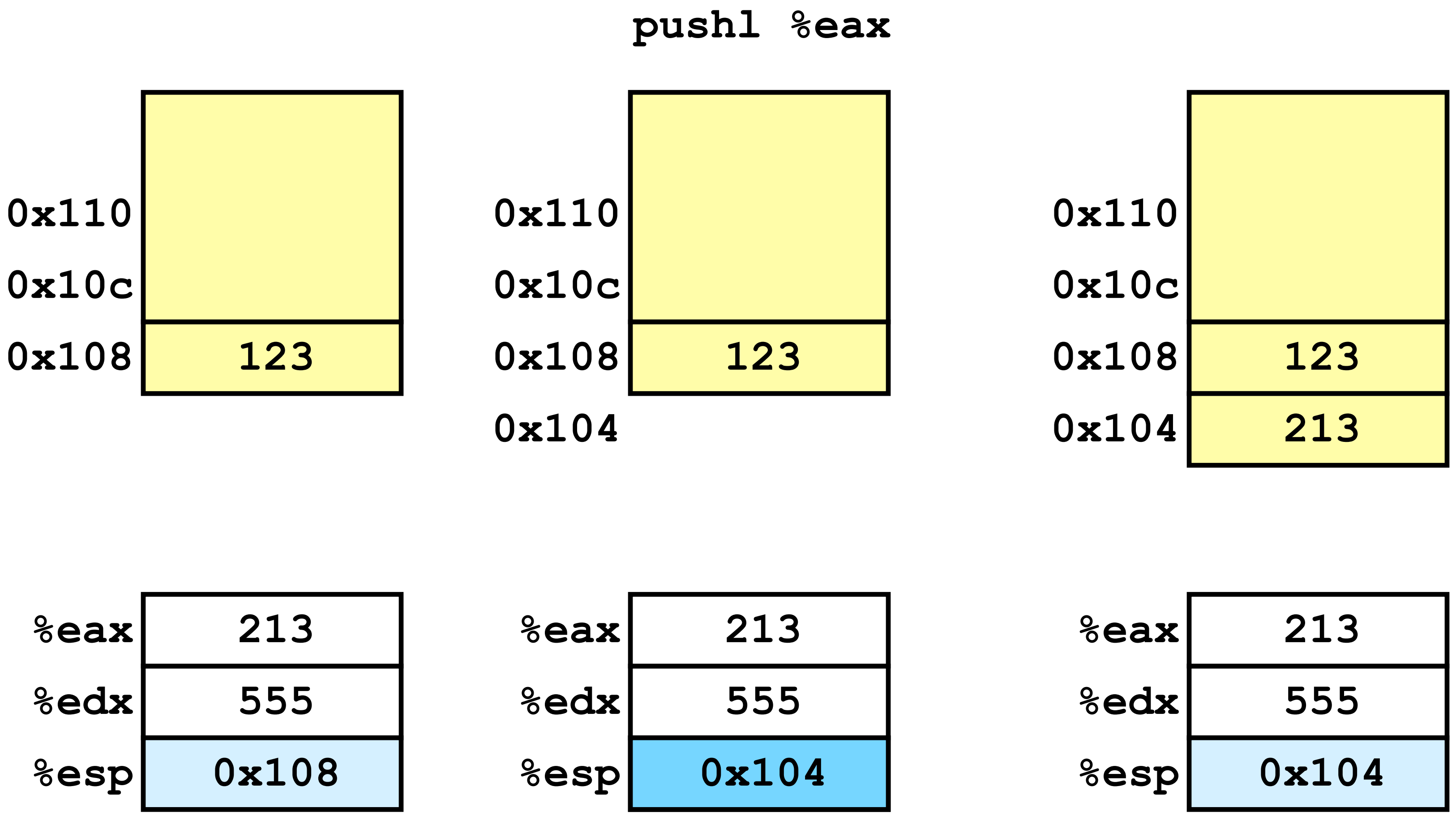
`pushl %eax`



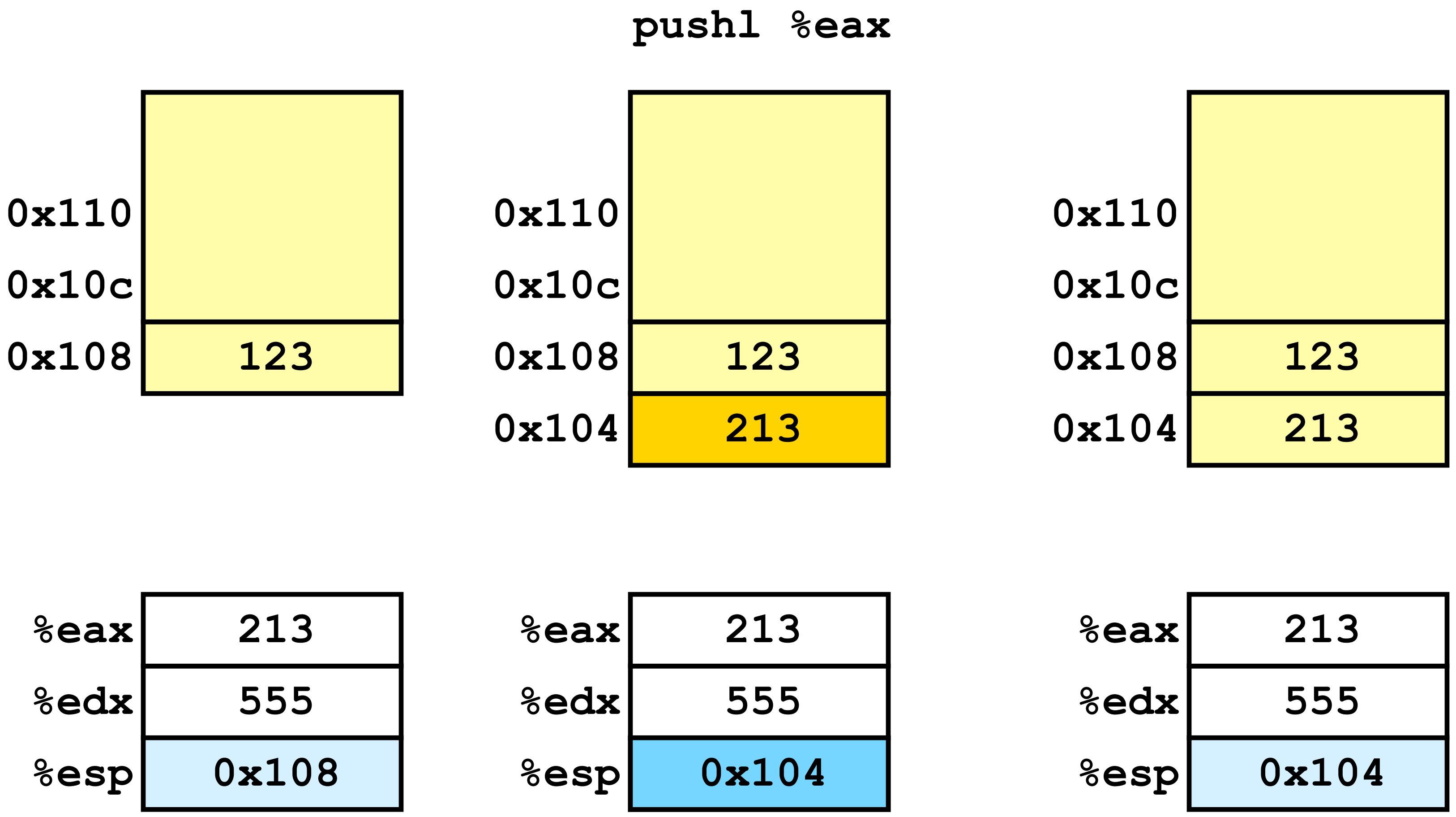
Stack operation examples



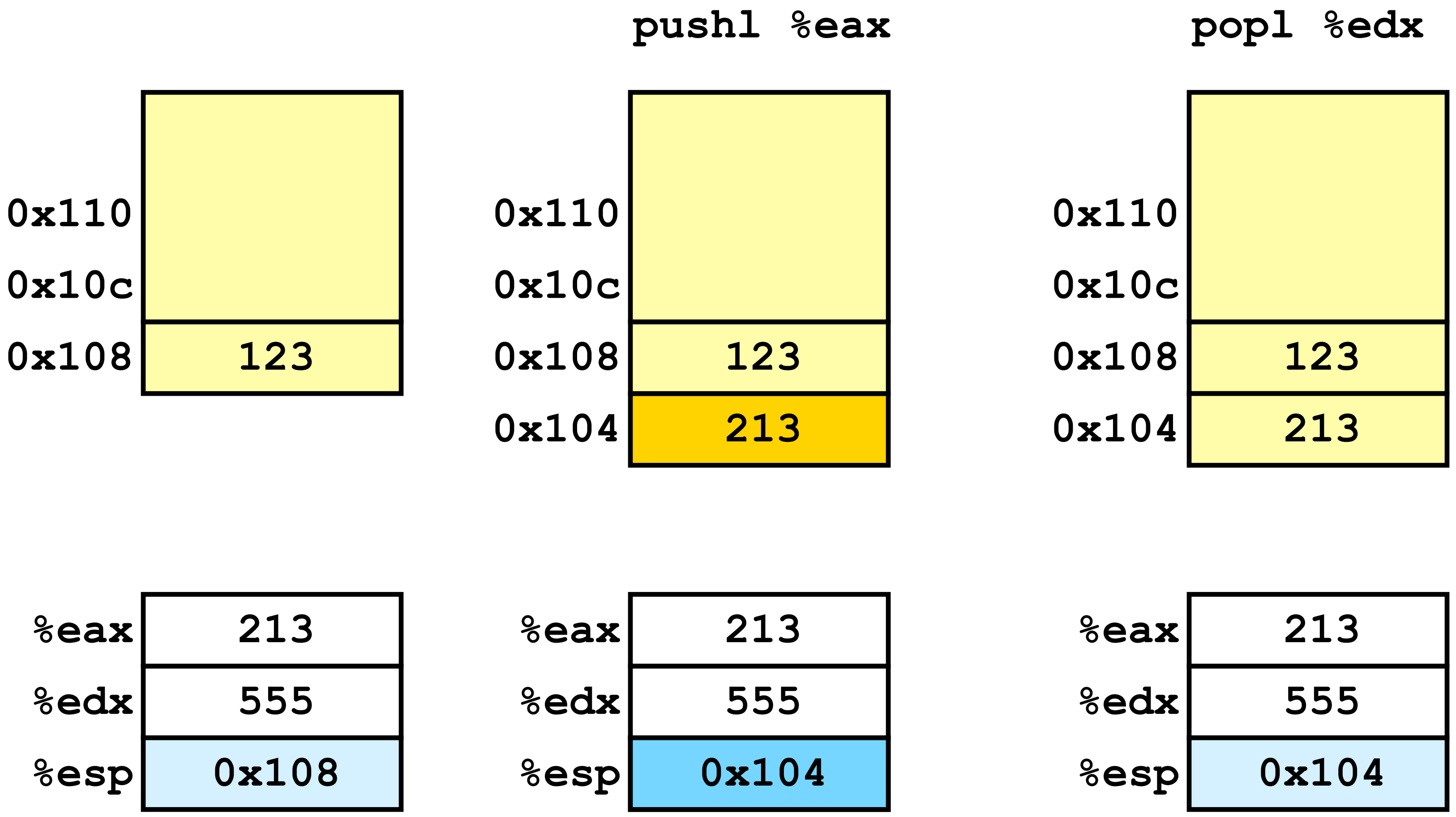
Stack operation examples



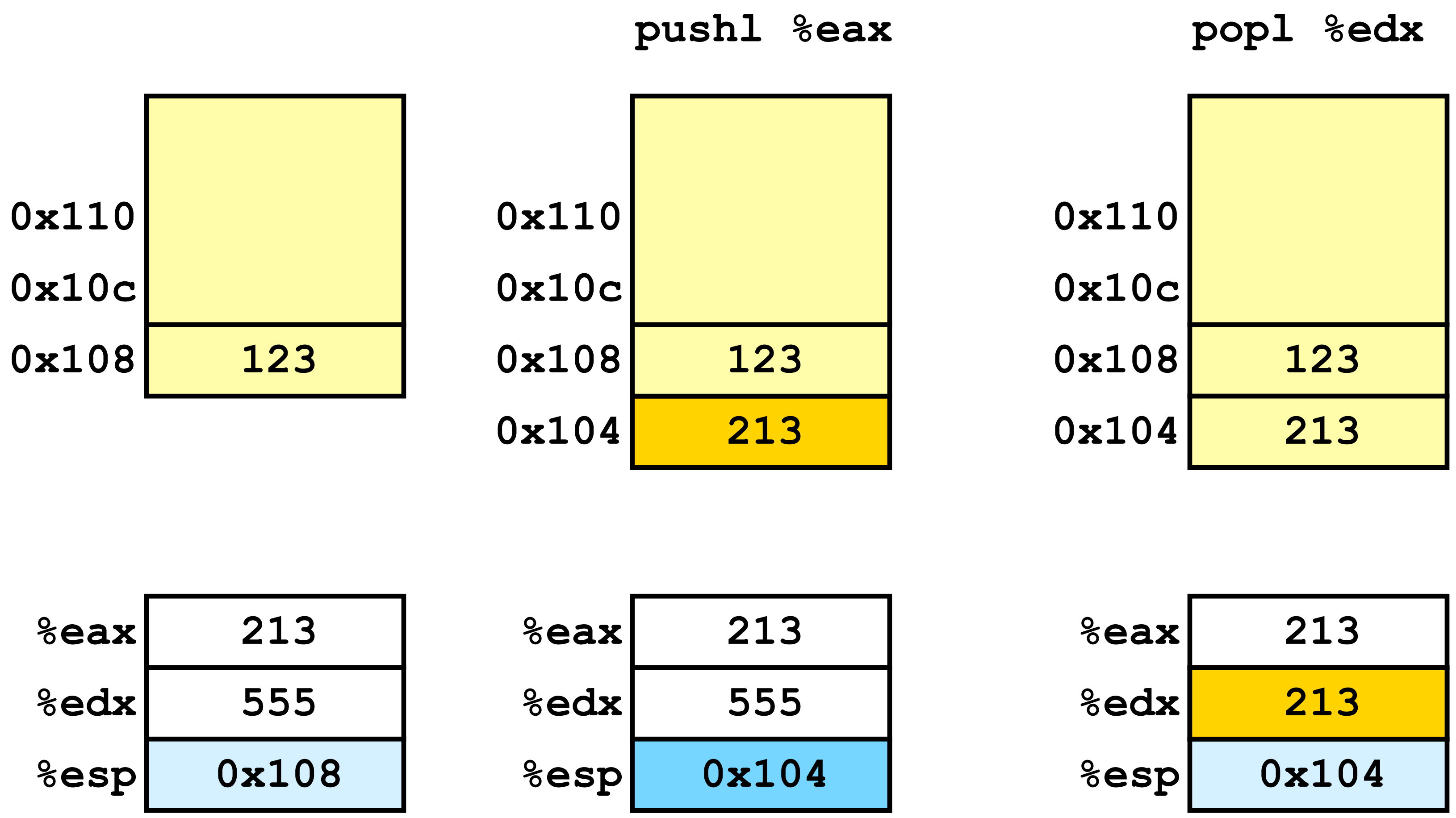
Stack operation examples



Stack operation examples

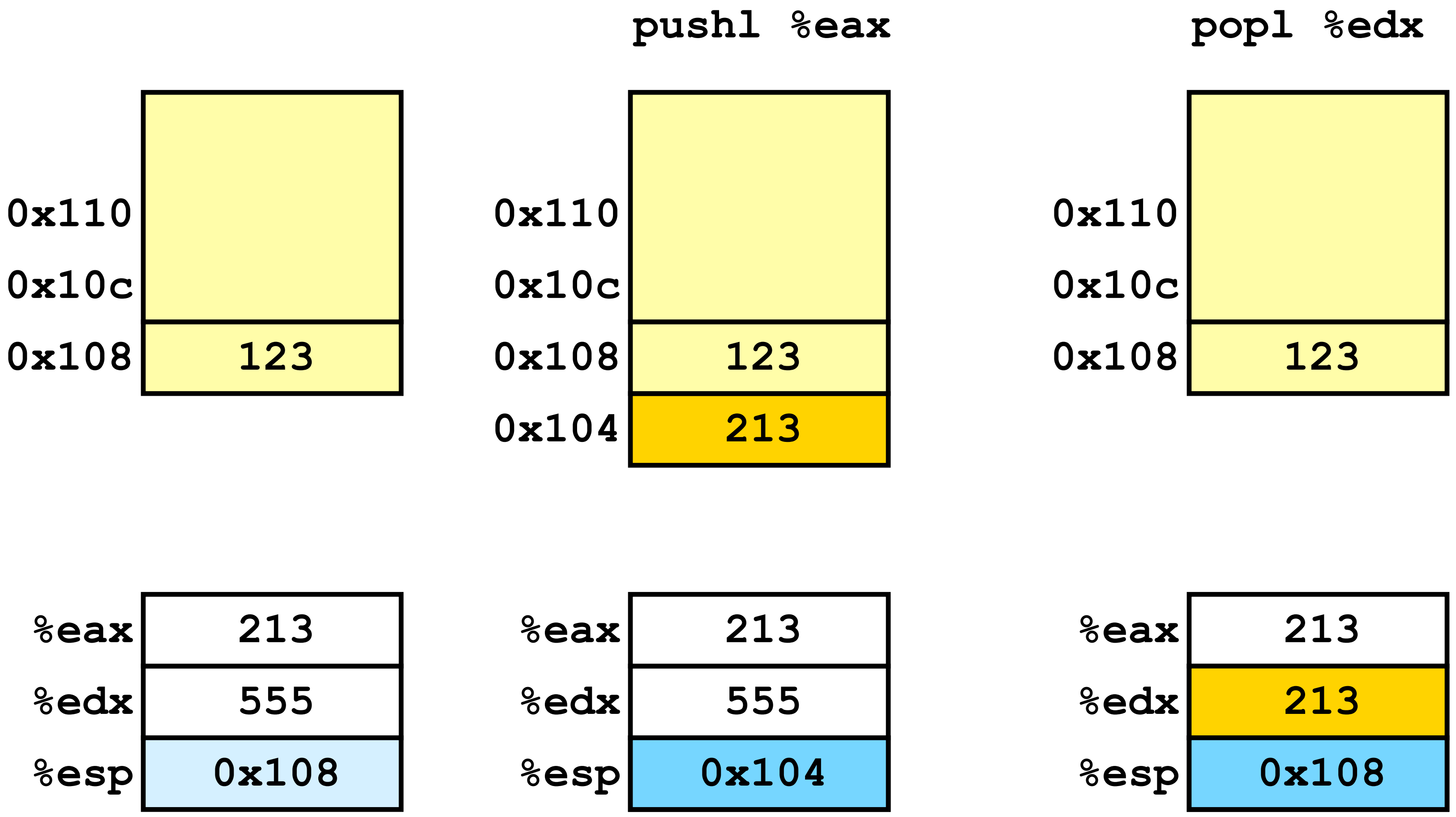


Stack operation examples



Sunday, October 16, 2011

Stack operation examples



Procedure control flow: call

- Use stack to support procedure call and return
- Procedure call

`call label`	Push return address on stack; Jump to `label`
`call *Operand`	Indirect call/jump

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

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

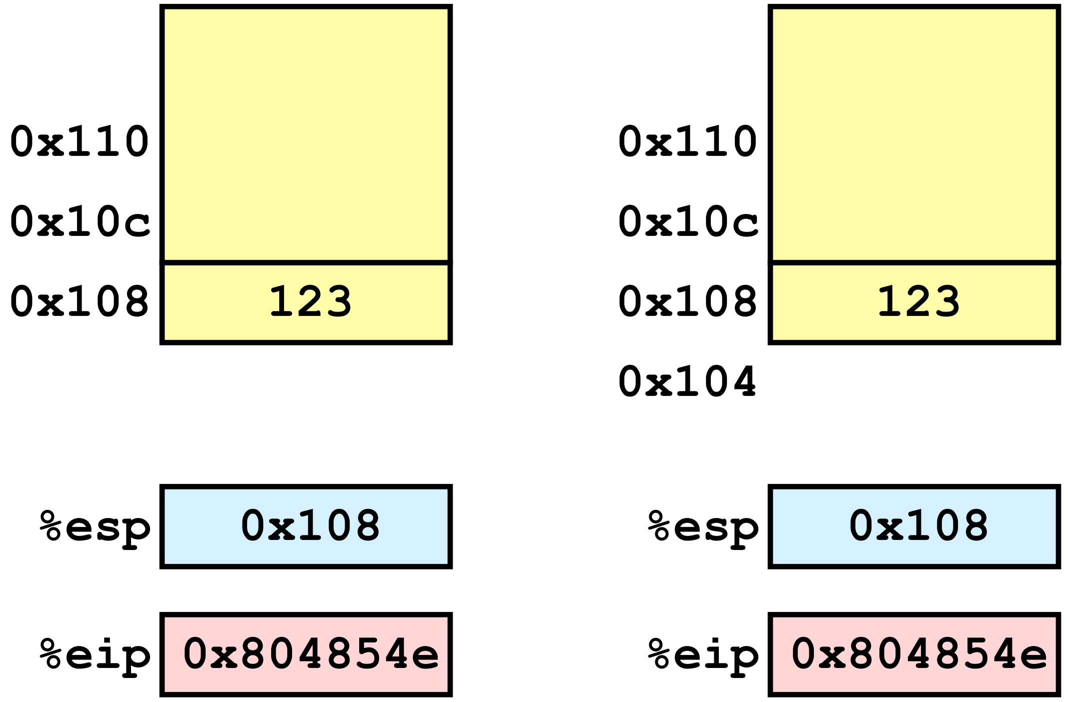
    - Return address = `0x8048553`

Procedure control flow: return

- Procedure return
  - `leave` Prepare stack for return; equivalent to
    - `movl %ebp, %esp`
    - `popl %ebp`
  - `ret` Pop address from stack; Jump to address (after stack is ready)

Procedure call example

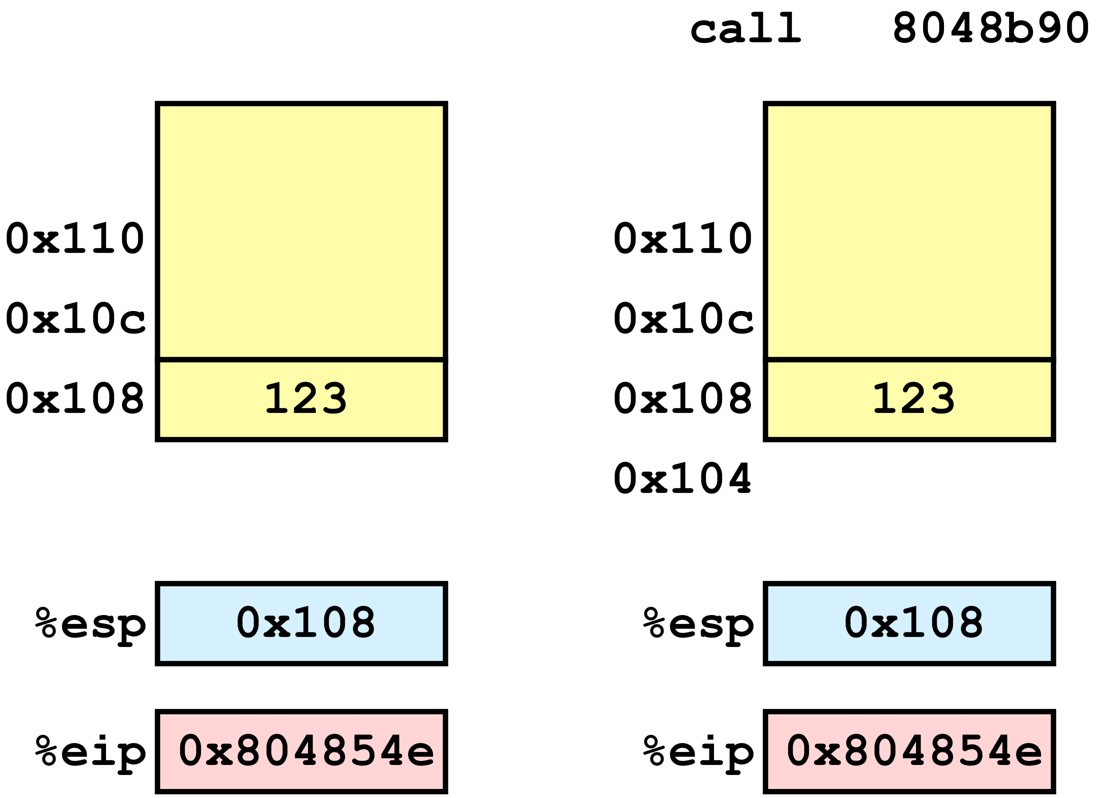
```
804854e:   e8 3d 06 00 00        call    8048b90 <main>
8048553:   50                    pushl   %eax
```



`%eip` is program counter

Procedure call example

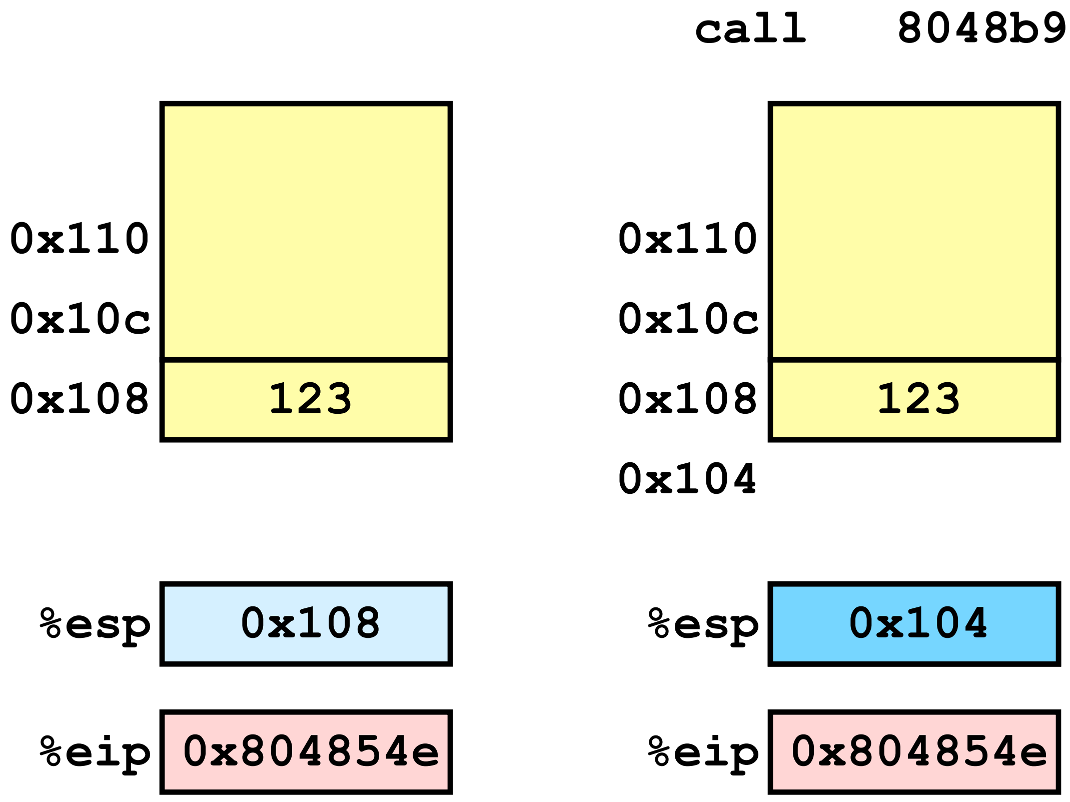
```
804854e:  e8 3d 06 00 00       call    8048b90 <main>
8048553:  50                   pushl   %eax
```



`%eip` **is program counter**

Procedure call example

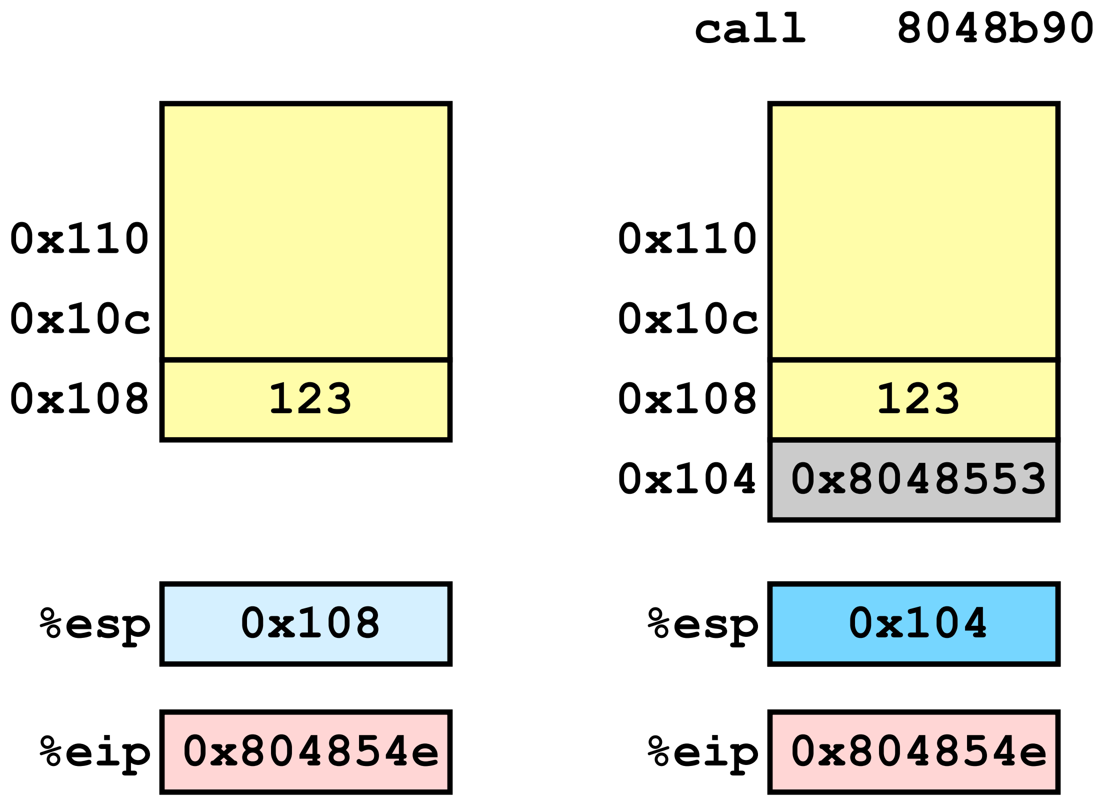
```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```



`%eip` **is program counter**

Procedure call example

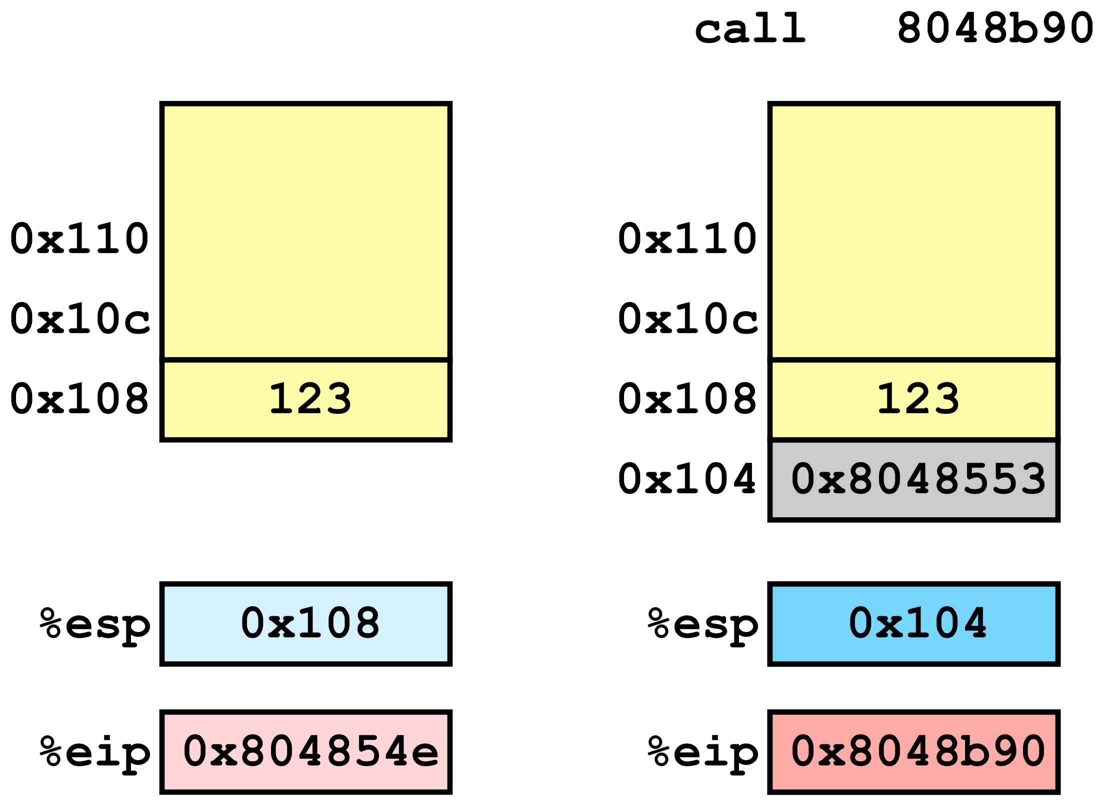
```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```



`%eip` **is program counter**

Procedure call example

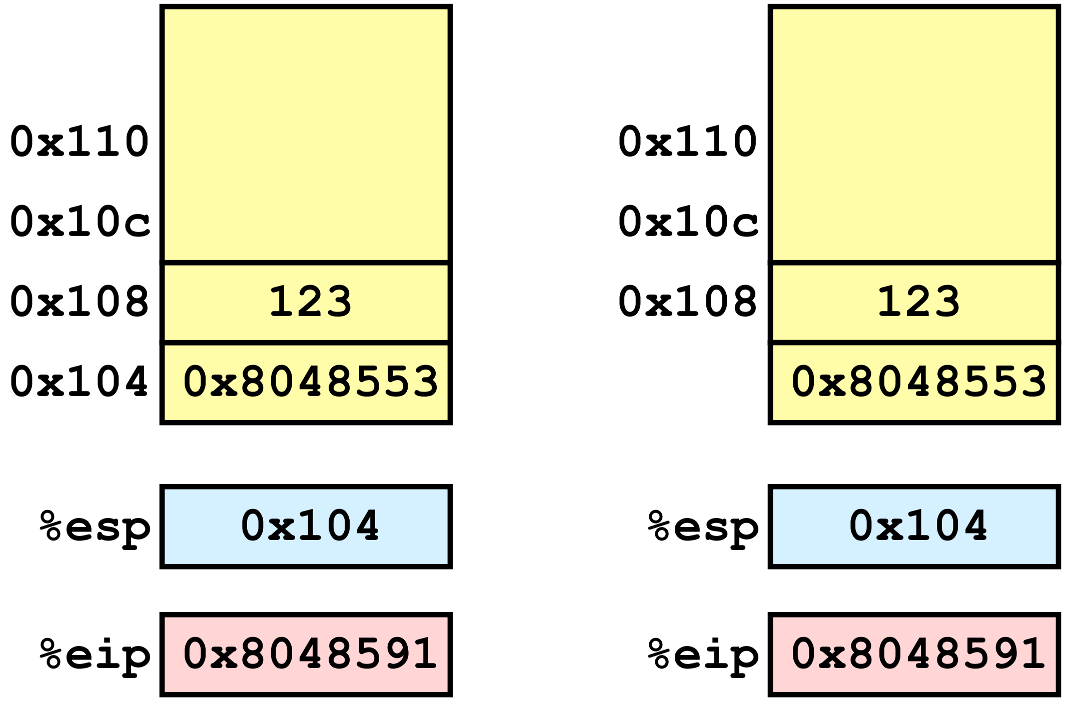
```
804854e:  e8 3d 06 00 00        call   8048b90 <main>
8048553:  50                    pushl  %eax
```



`%eip` **is program counter**

Procedure return example

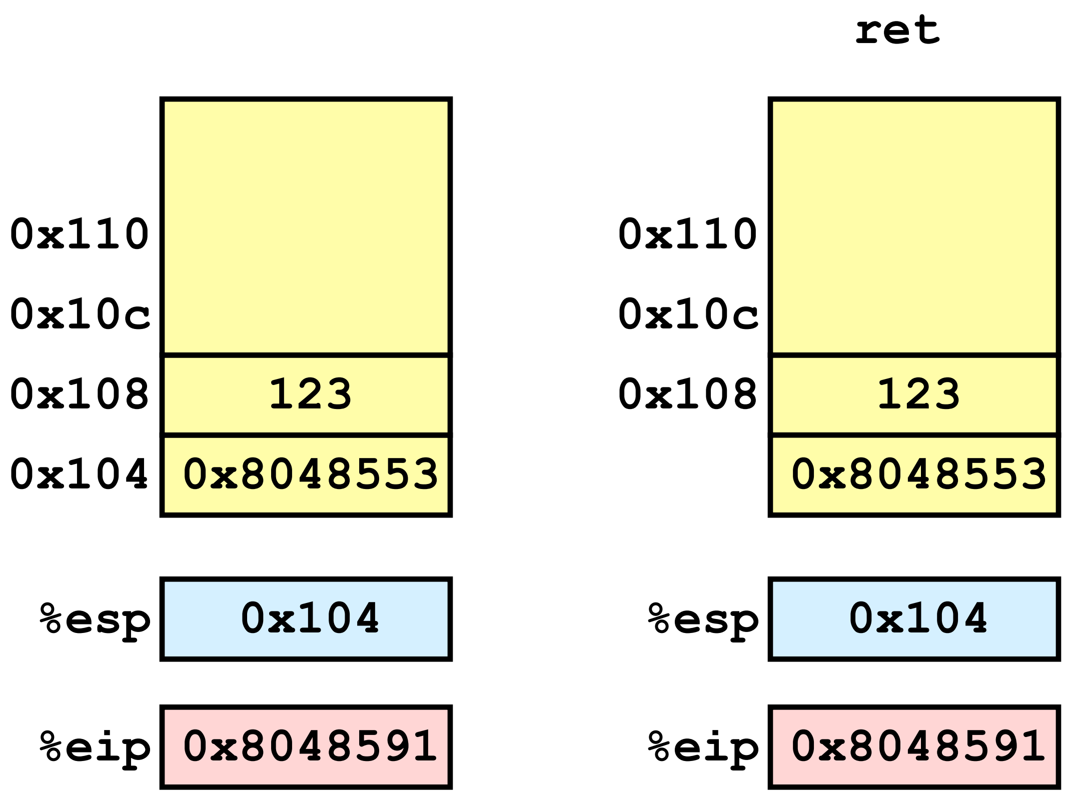
```
8048591:  c3                    ret
```



`%eip` is program counter

Procedure return example

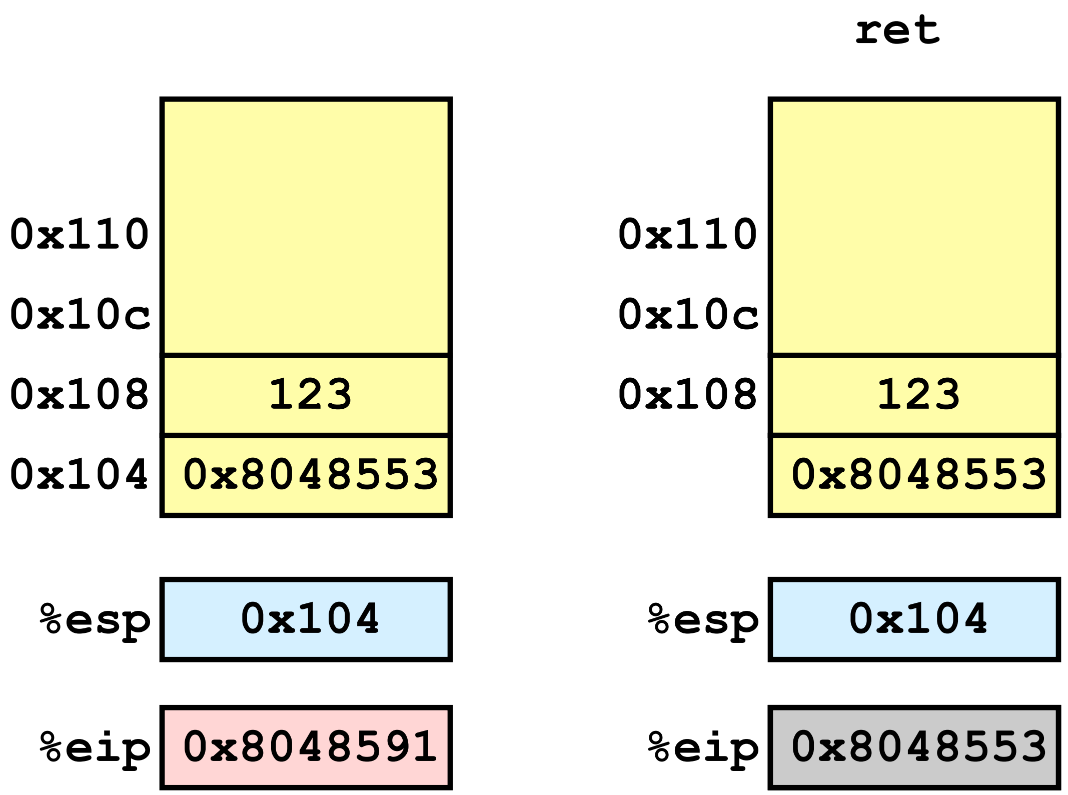
```
8048591:   c3                    ret
```



`%eip` **is program counter**

Procedure return example

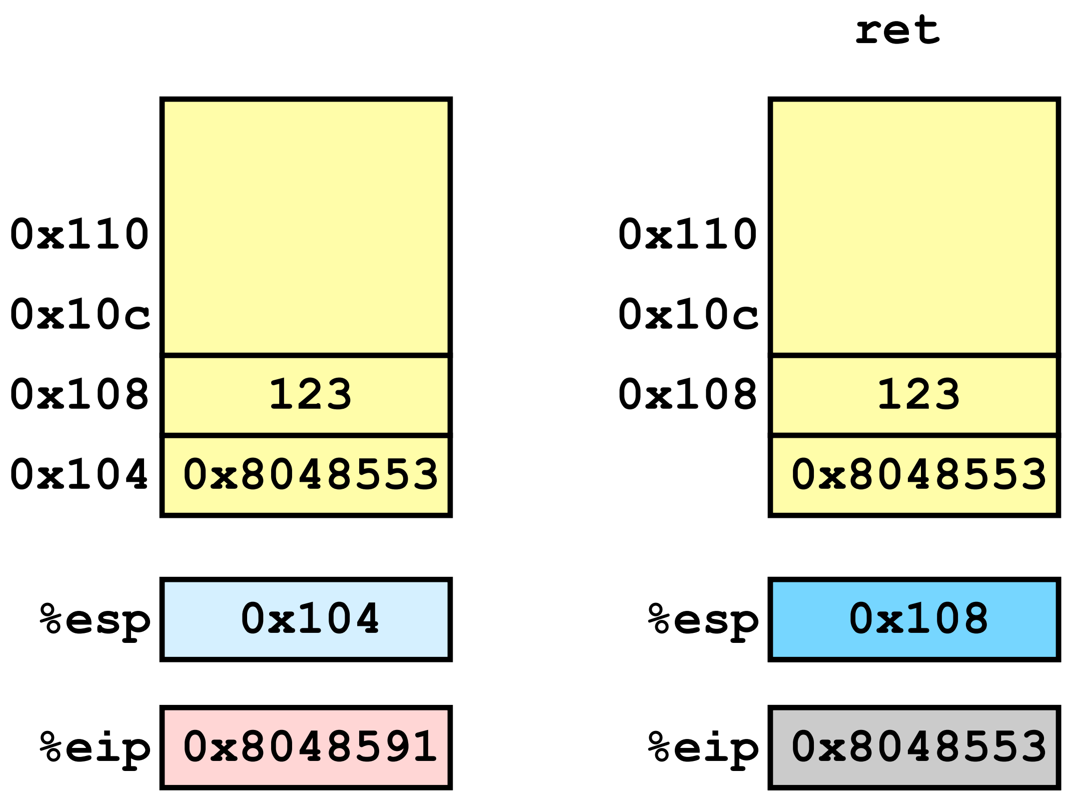
```
8048591:  c3                    ret
```



`%eip` **is program counter**

Procedure return example

```
8048591:  c3                  ret
```



`%eip` **is program counter**

Stack-based languages

- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be "*reentrant*"
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer
- Stack discipline
  - State for given procedure needed for limited time
    - From when called to when return
  - Callee returns before caller does
- Stack allocated in *frames*
  - state for single procedure instantiation

Call chain example

Code structure

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

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

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

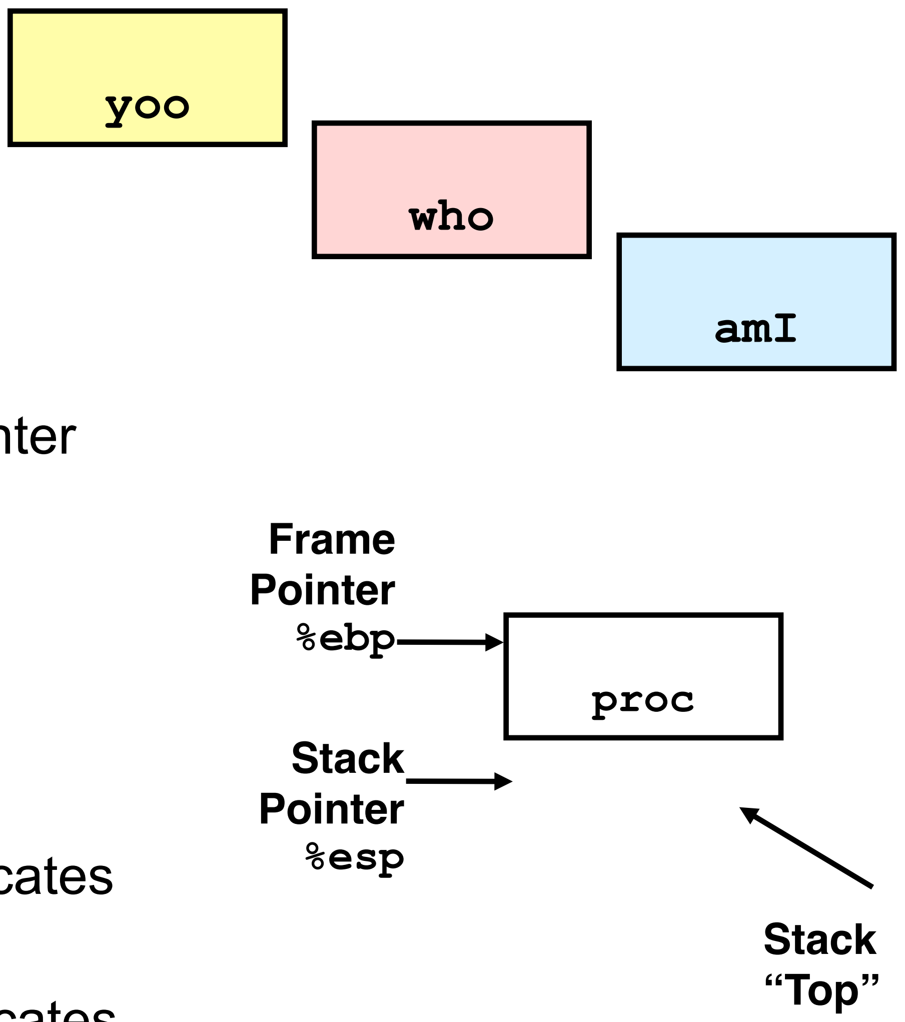
Procedure `amI` recursive

Call Chain

$$\begin{tikzcd}
\texttt{yoo} \arrow[d] & \\
\texttt{who} \arrow[d] \arrow[dr] & \\
\texttt{amI} \arrow[d] & \texttt{amI} \\
\texttt{amI} \arrow[d] & \\
\texttt{amI} &
\end{tikzcd}$$

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

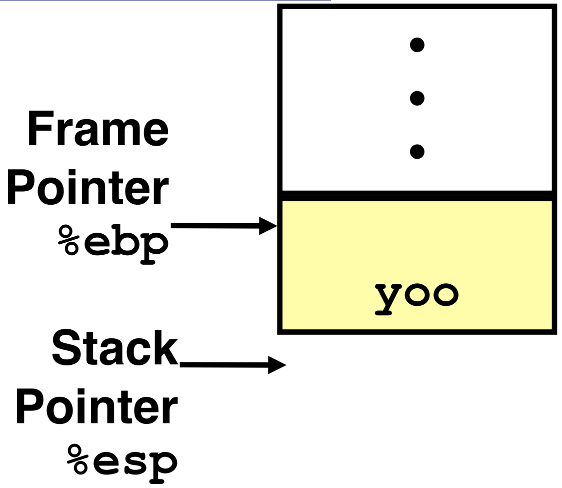


Stack operation

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

Call Chain

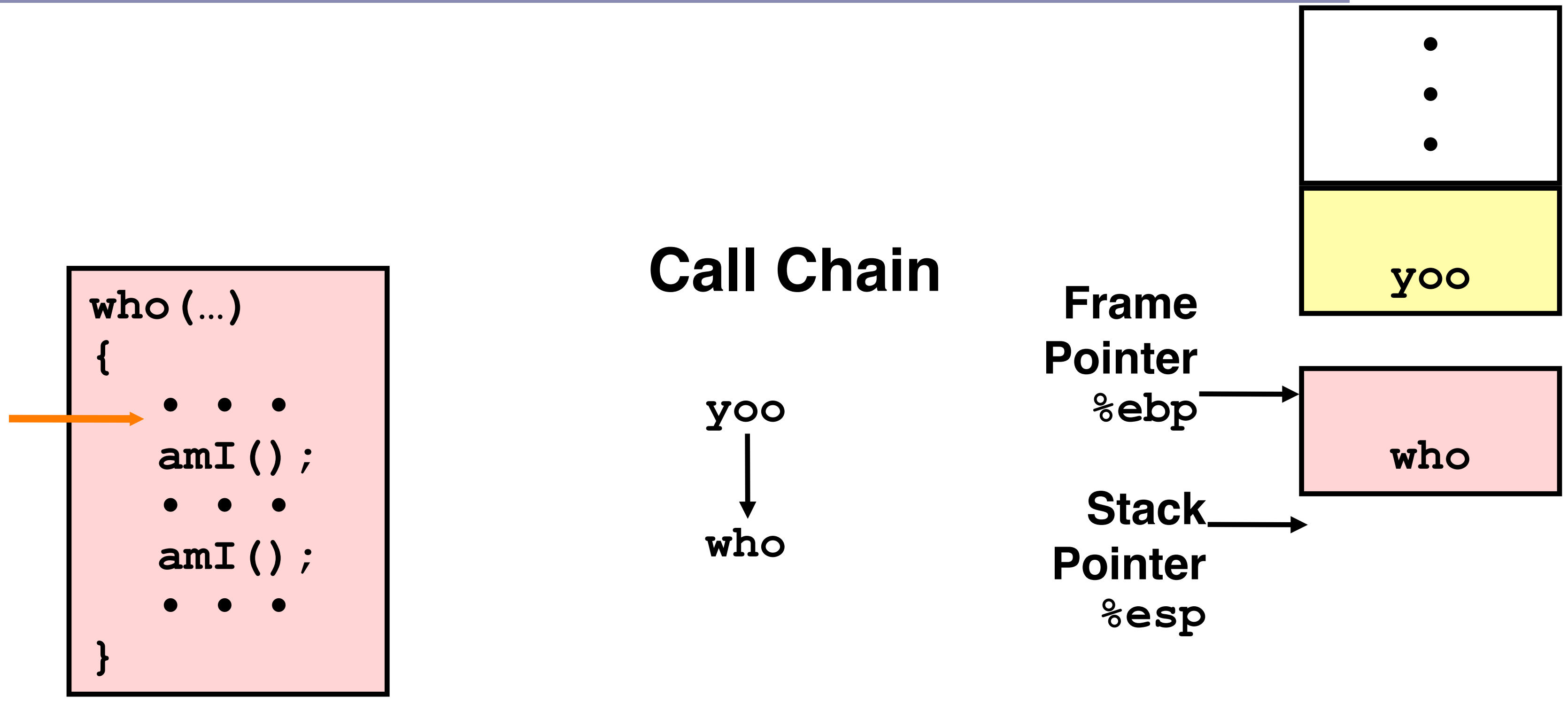
`yoo`



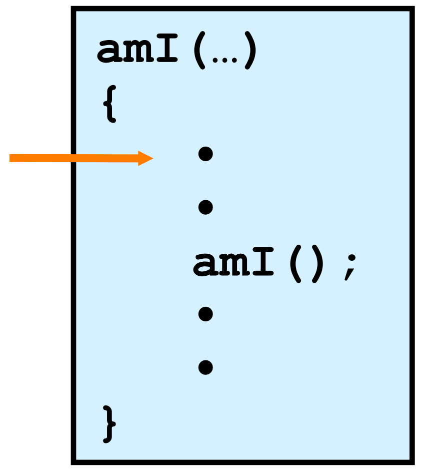
Stack operation

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

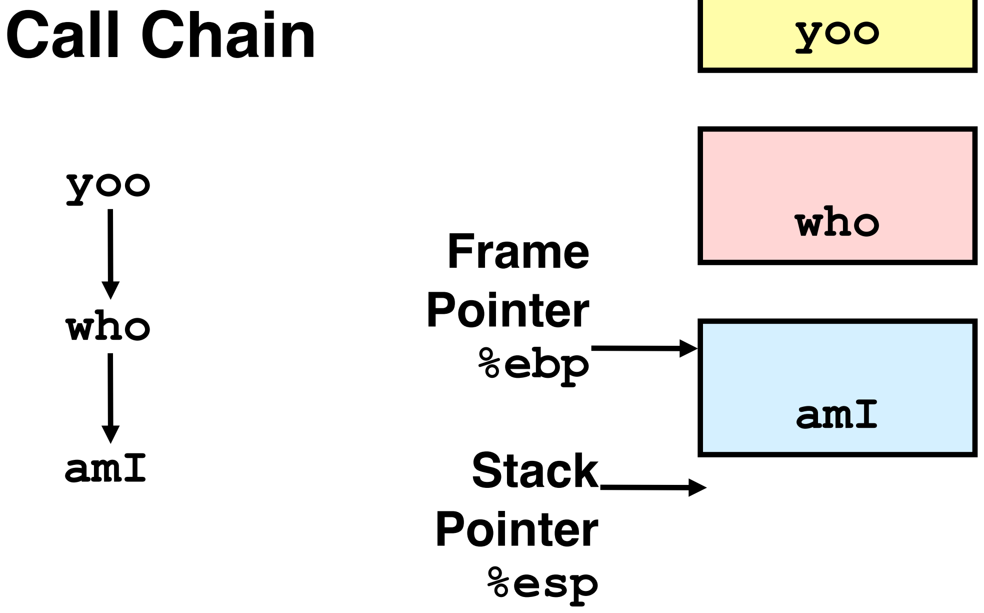
Call Chain



Stack operation



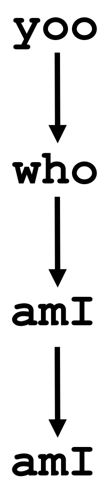
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

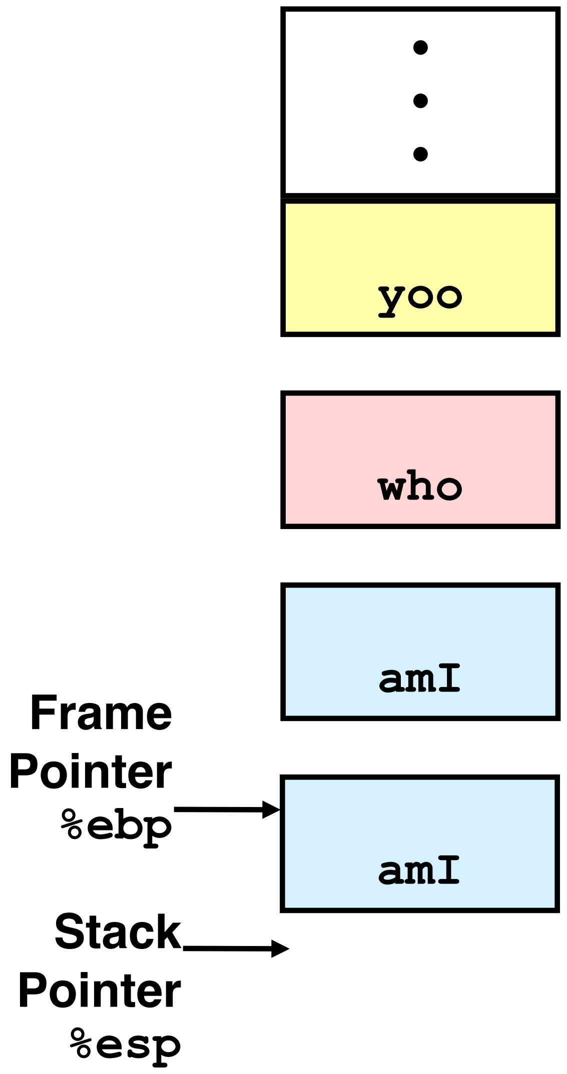


Stack operation

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

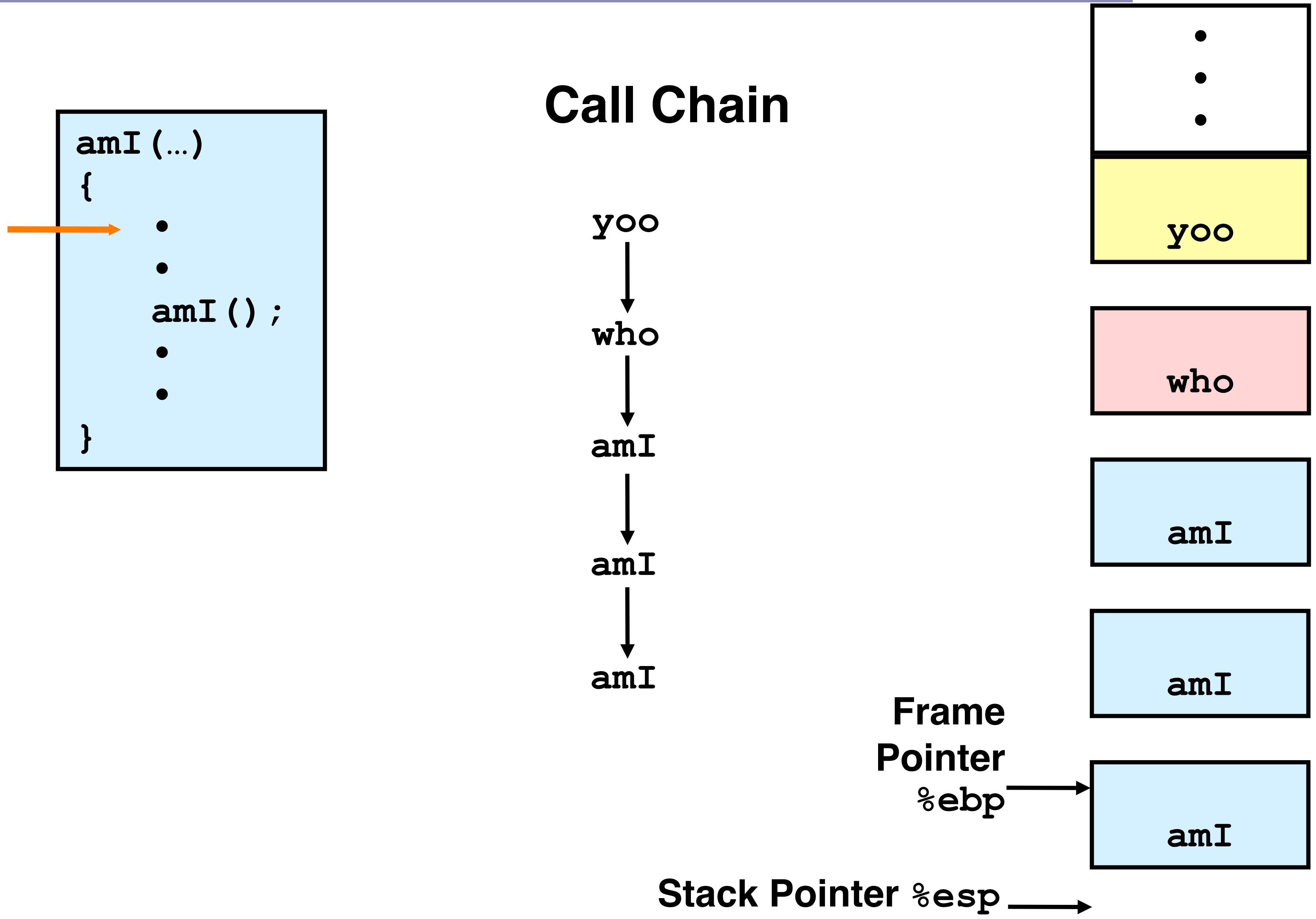
Call Chain





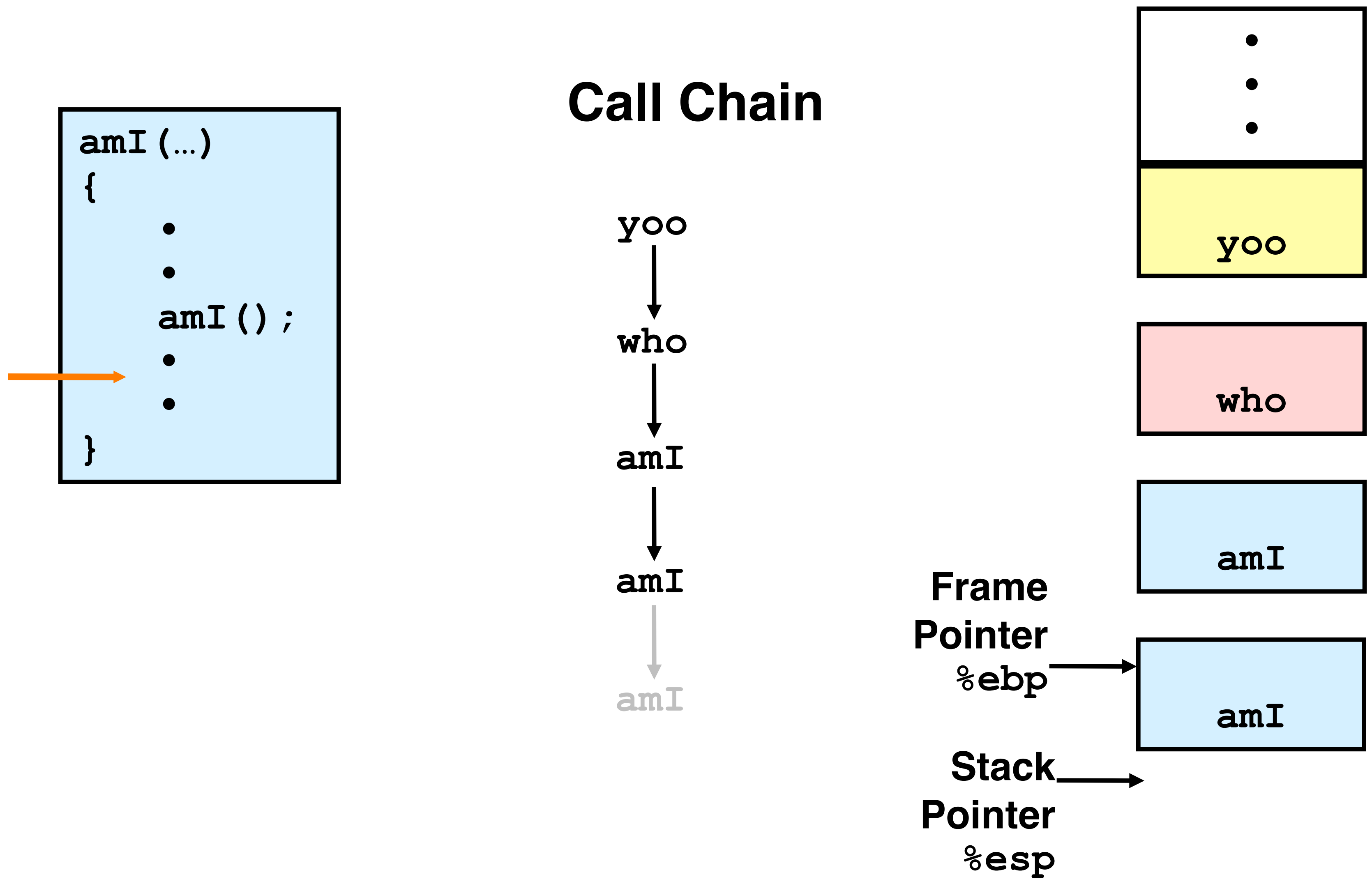
Stack operation

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



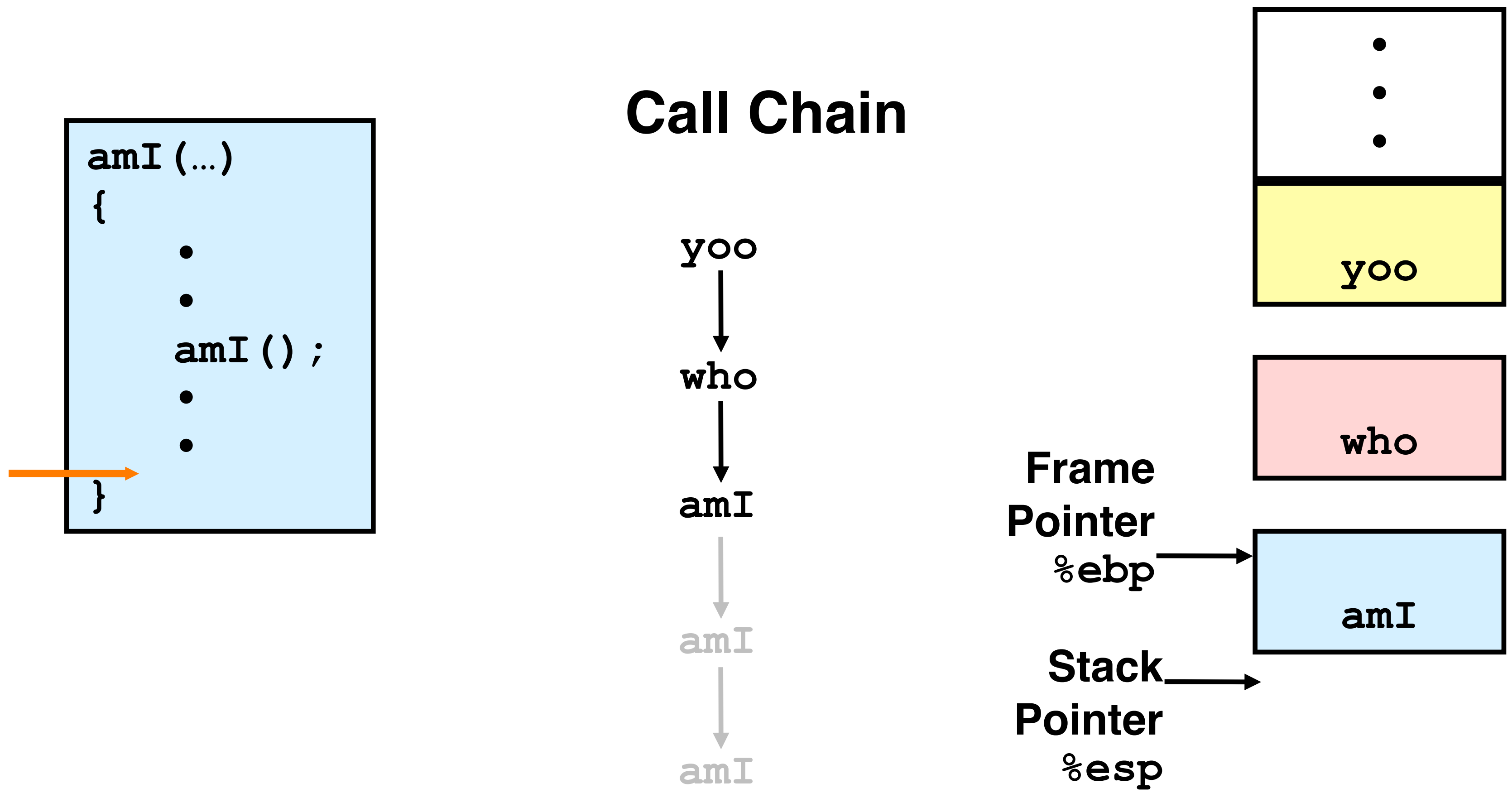
Stack operation

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



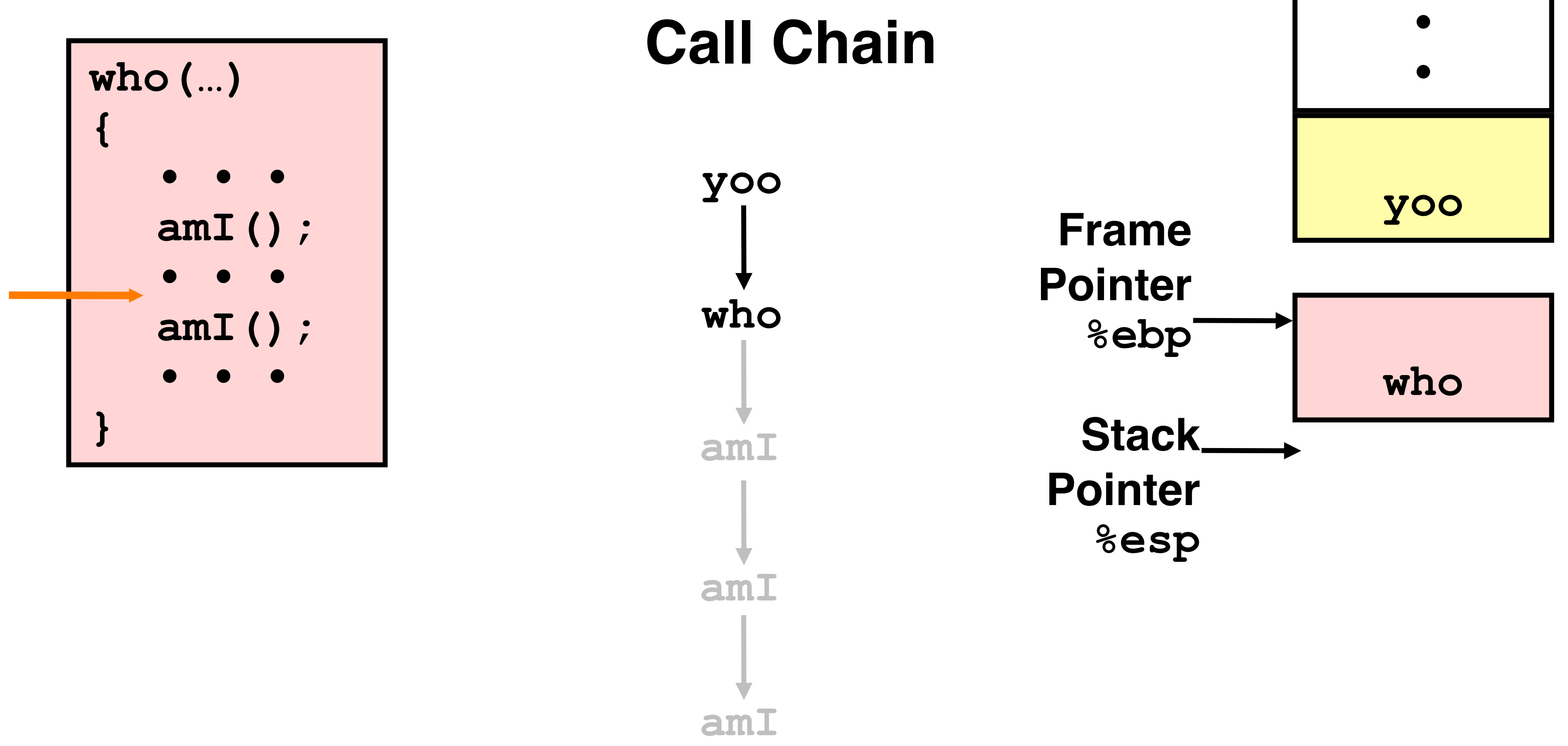
Stack operation

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

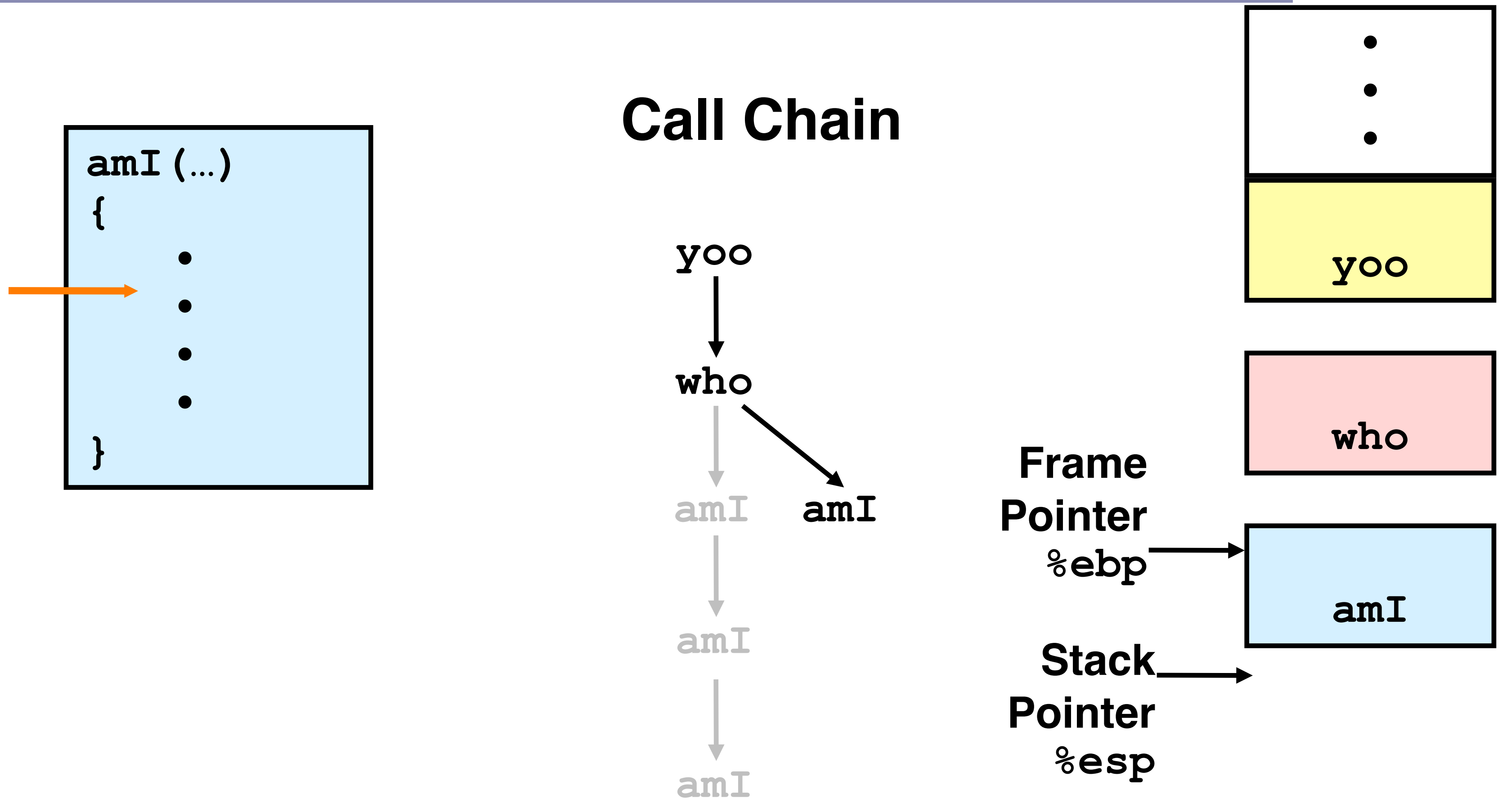


Stack operation

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

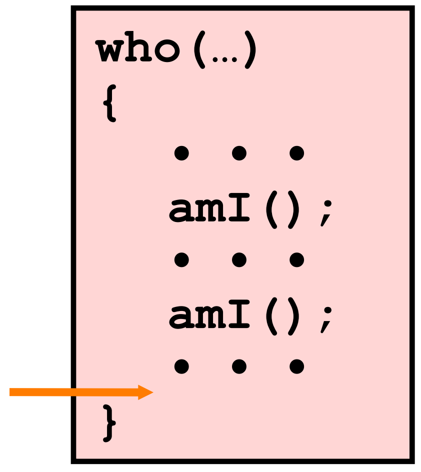


Stack operation



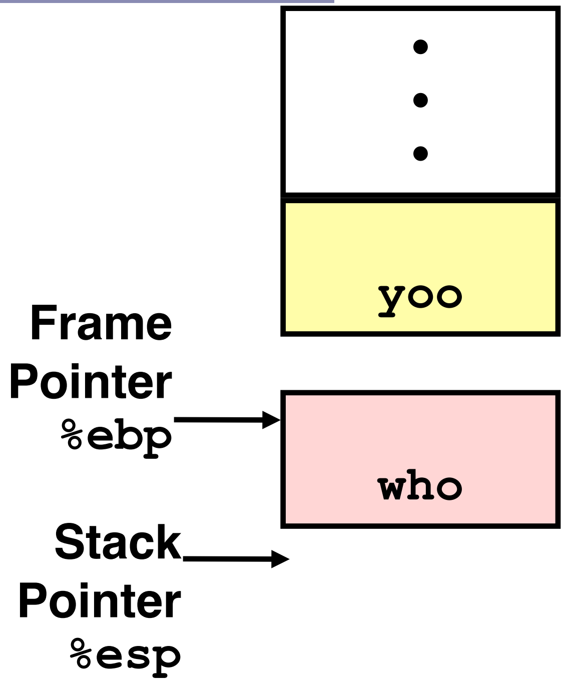
Stack operation

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

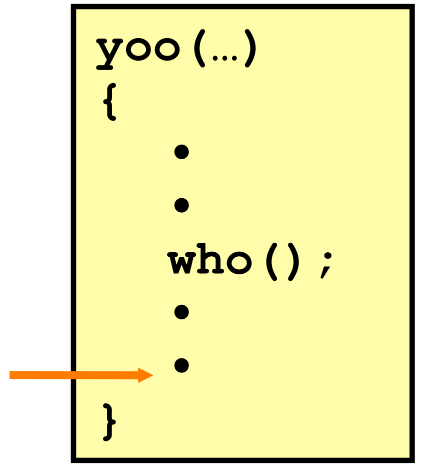


Call Chain

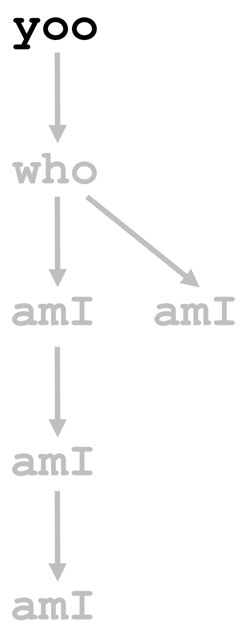
$$\begin{tikzcd}[column sep=small] & \texttt{yoo} \arrow[d] & \\ & \texttt{who} \arrow[d] \arrow[dr] & \\ & \texttt{amI} \arrow[d] & \texttt{amI} \\ & \texttt{amI} \arrow[d] & \\ & \texttt{amI} & \end{tikzcd}$$

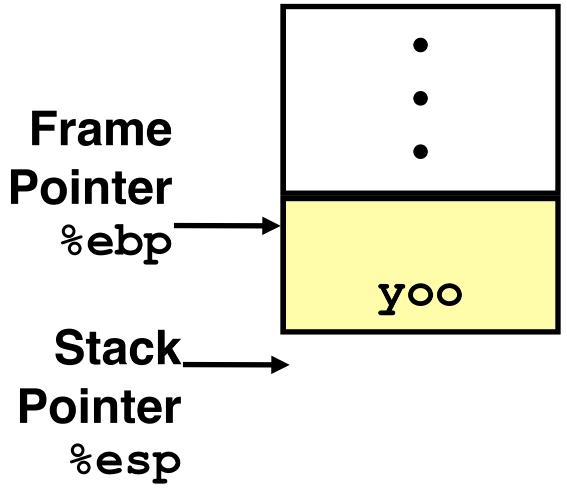


Stack operation


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

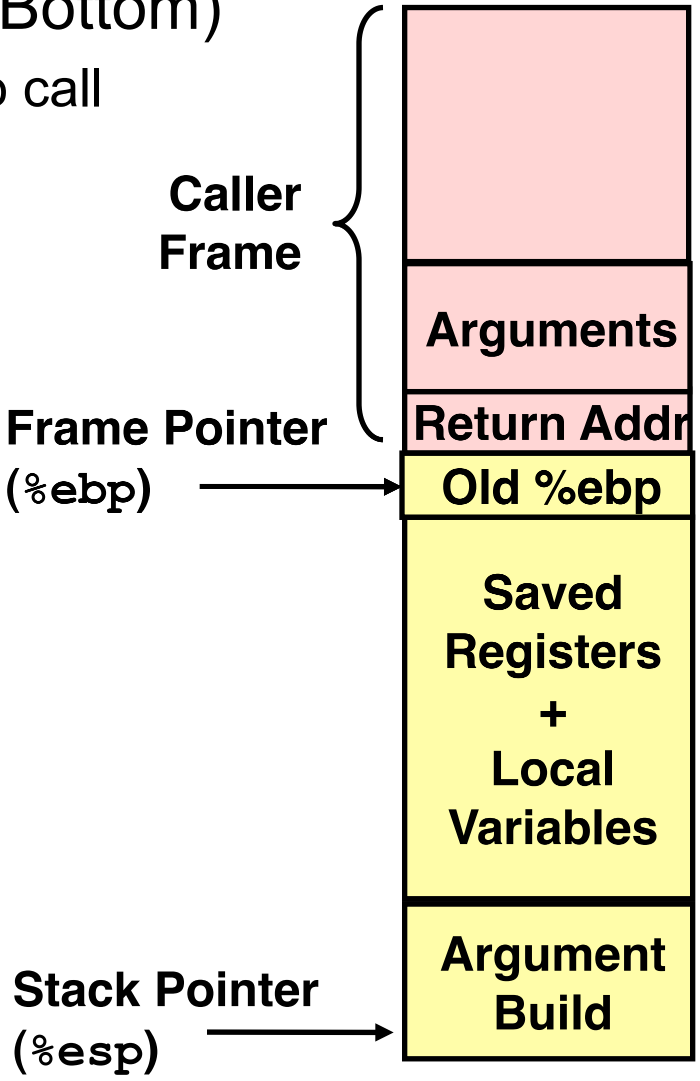
Call Chain


$$\begin{tikzcd}
\texttt{yoo} \arrow[d] & \\
\texttt{who} \arrow[d] \arrow[dr] & \\
\texttt{amI} \arrow[d] & \texttt{amI} \\
\texttt{amI} \arrow[d] & \\
\texttt{amI} &
\end{tikzcd}$$



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

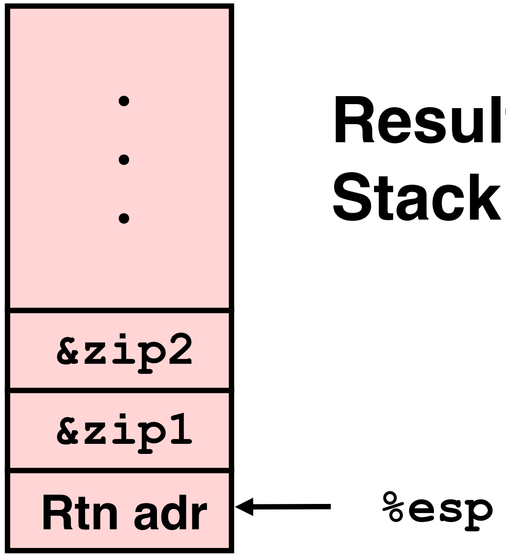
```
int zip1 = 15213;
int zip2 = 91125;

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
    • • •
```



Resulting Stack

Revisiting swap

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

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

Set Up

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

Body

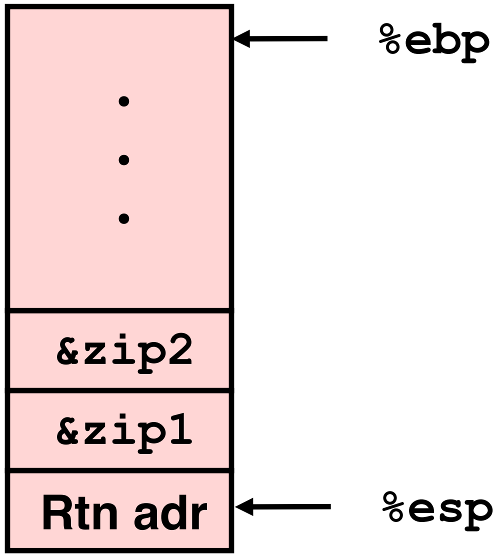
```
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

Finish

swap Setup #1

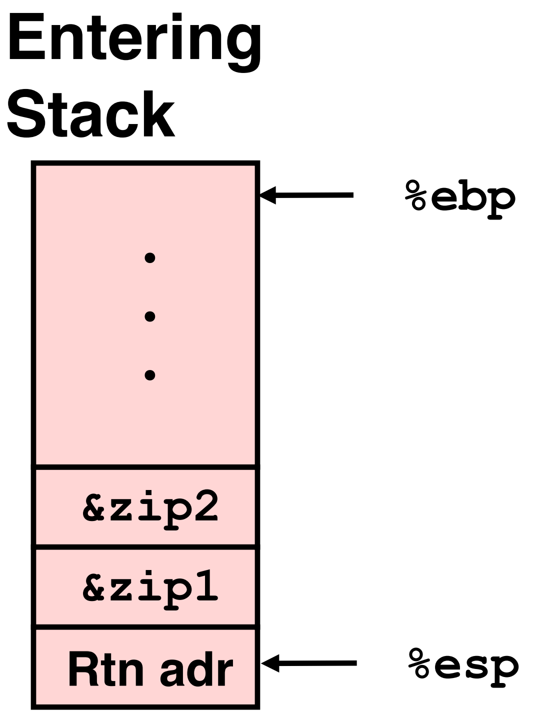
Entering Stack

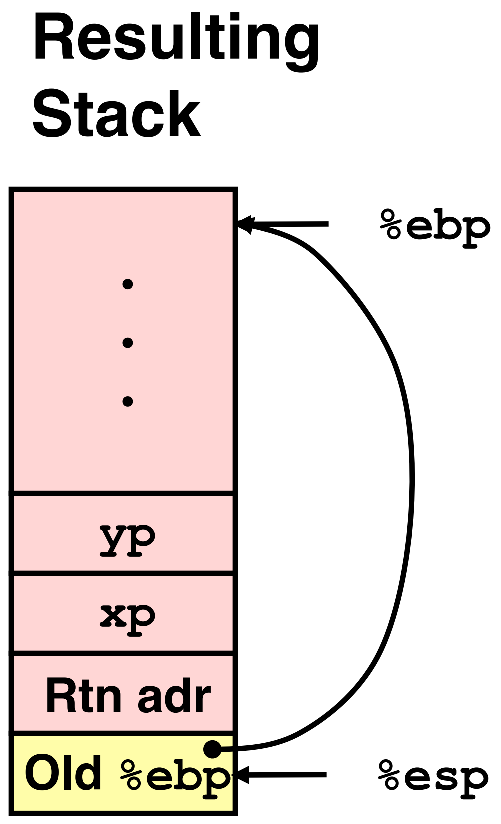
Resulting Stack



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

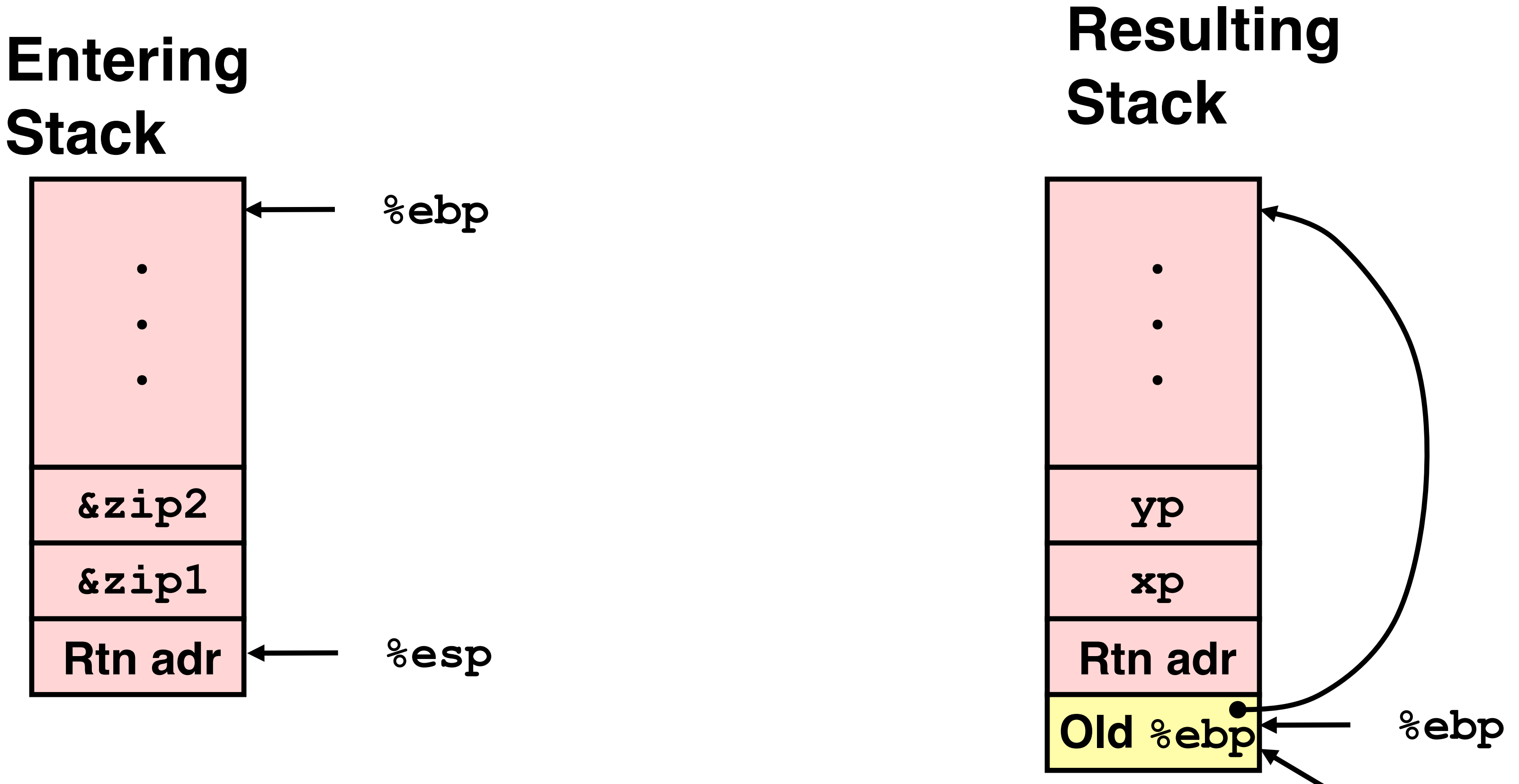
swap Setup #1





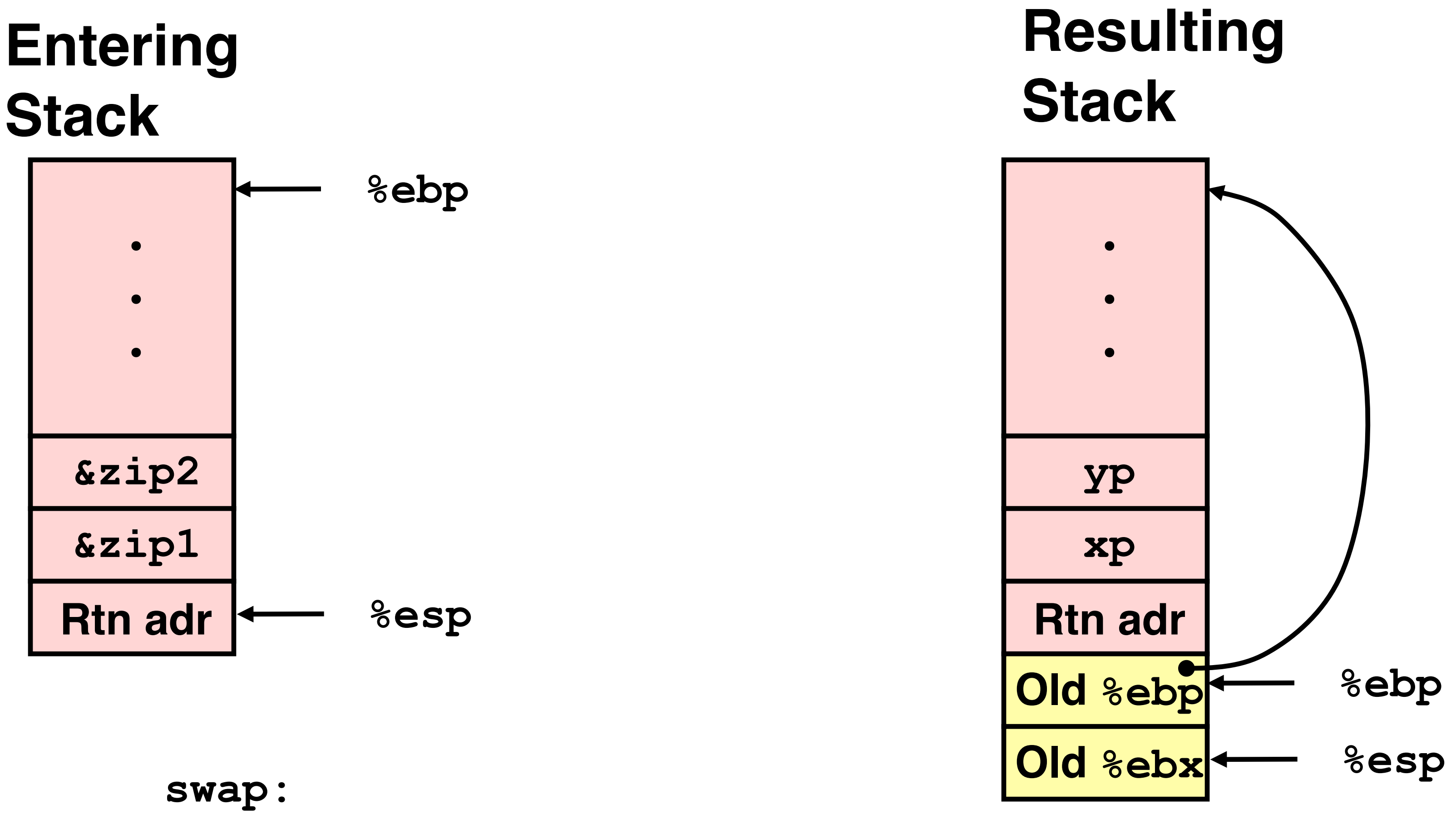
```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

swap Setup #2



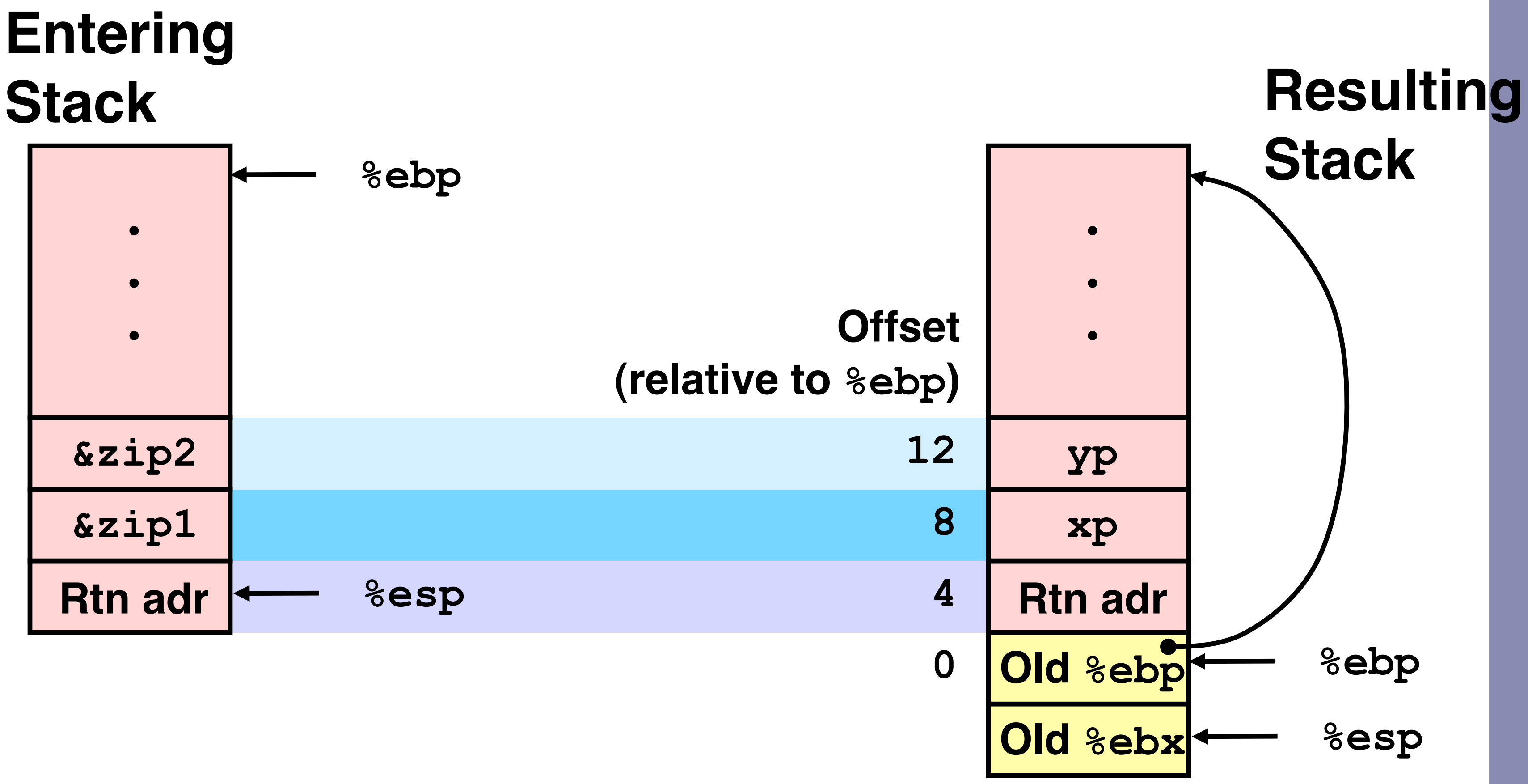
```
swap:
    pushl  %ebp
    movl  %esp,%ebp
    pushl  %ebx
```

swap Setup #3



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

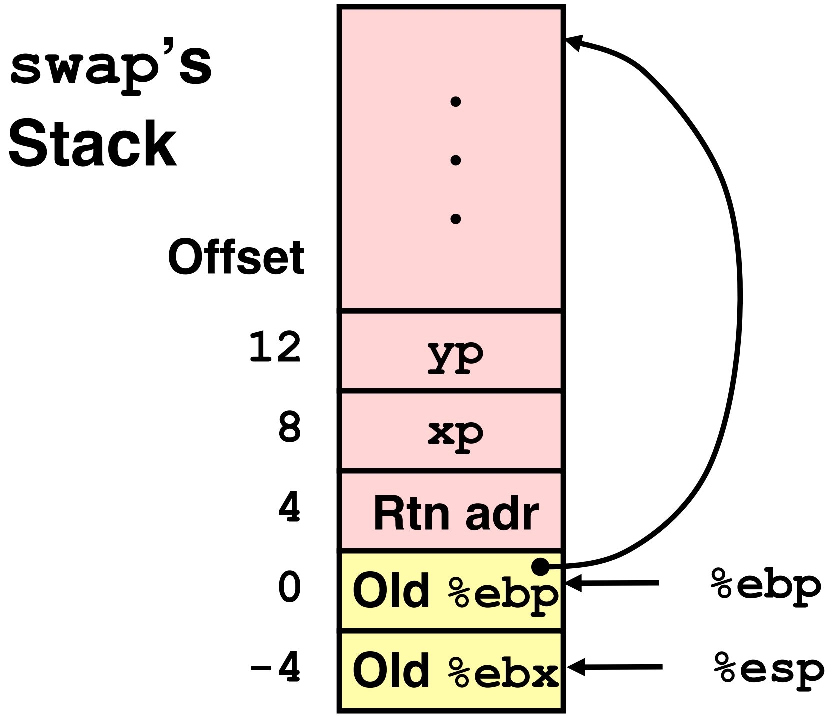
Effect of swap setup

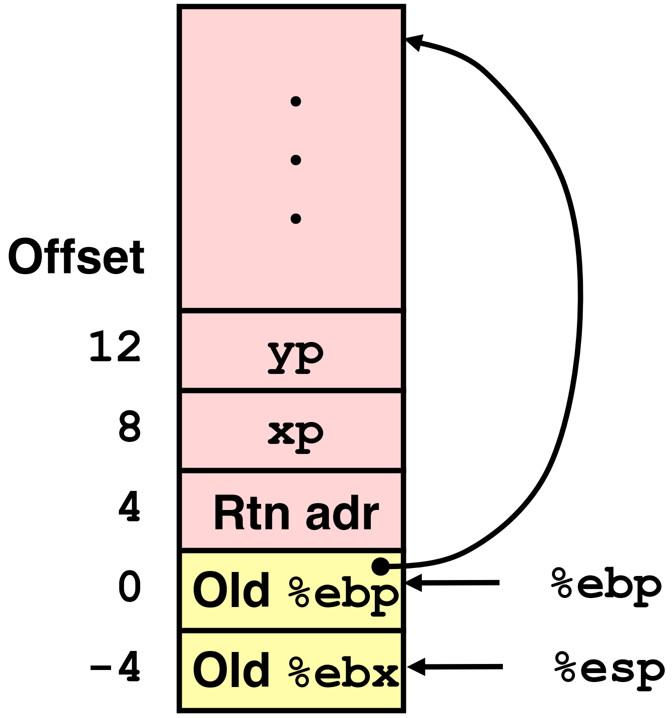


```
movl 12(%ebp),%ecx # get yp
movl 8(%ebp),%edx  # get xp
. . .
```

} Body

swap Finish #1

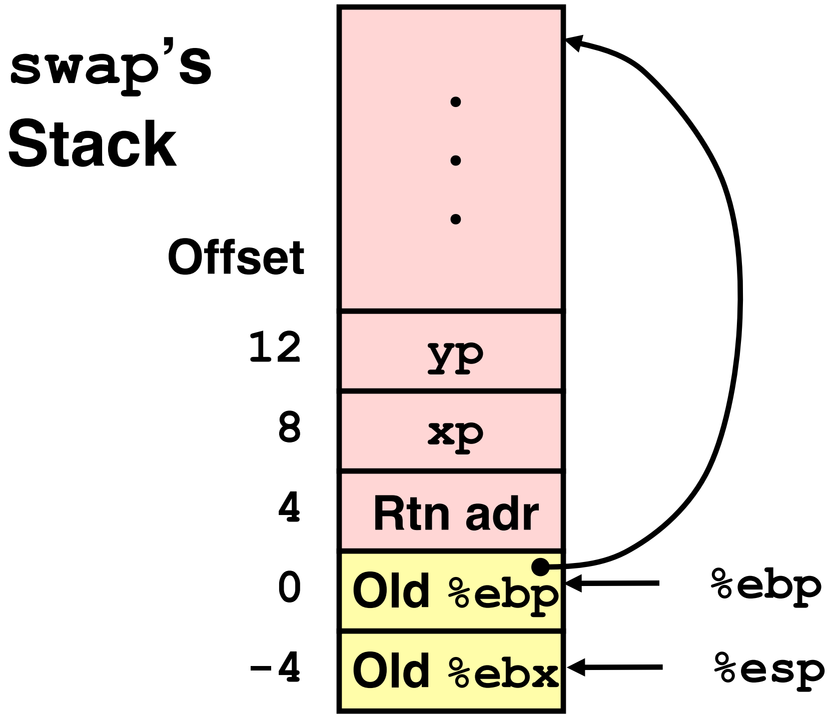


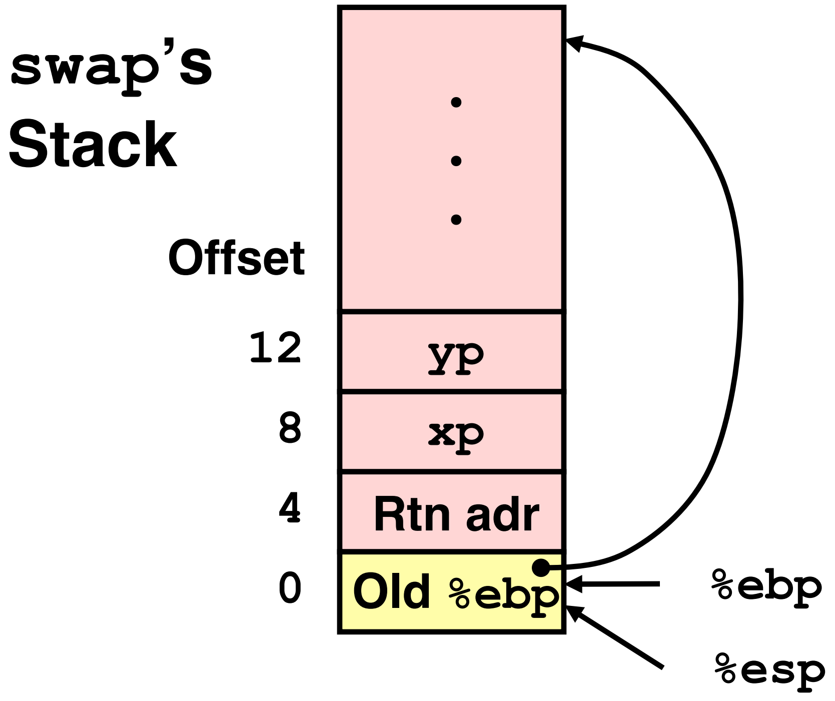


```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

- Observation
  - Saved & restored register `%ebx`

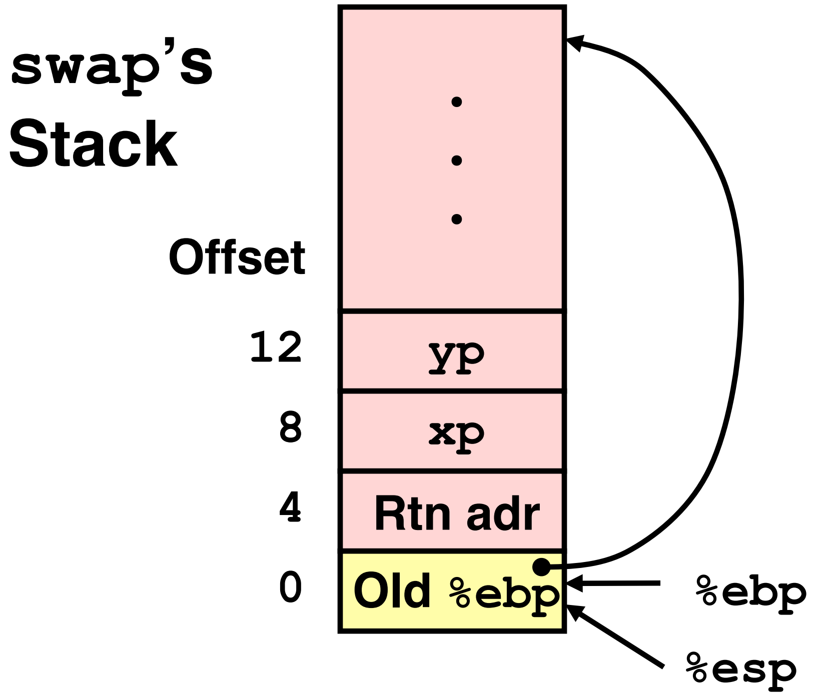
swap Finish #2

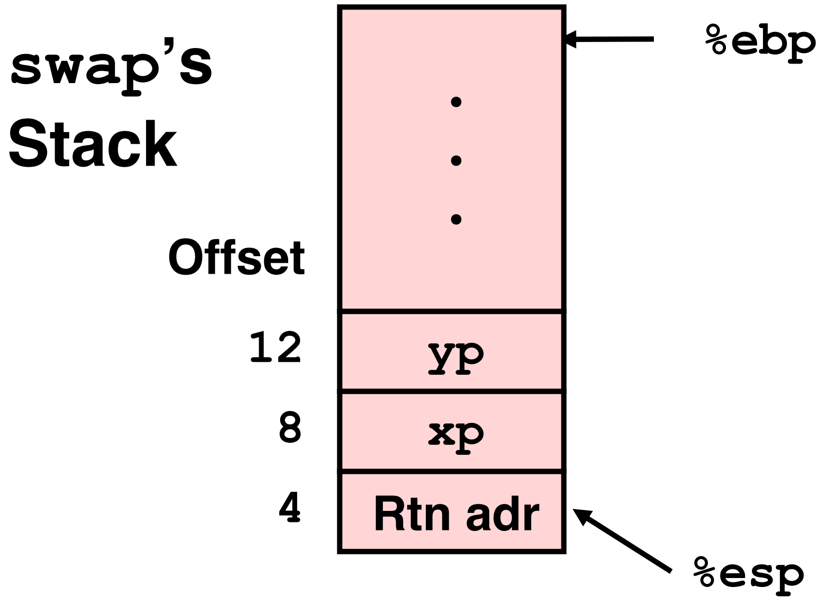




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

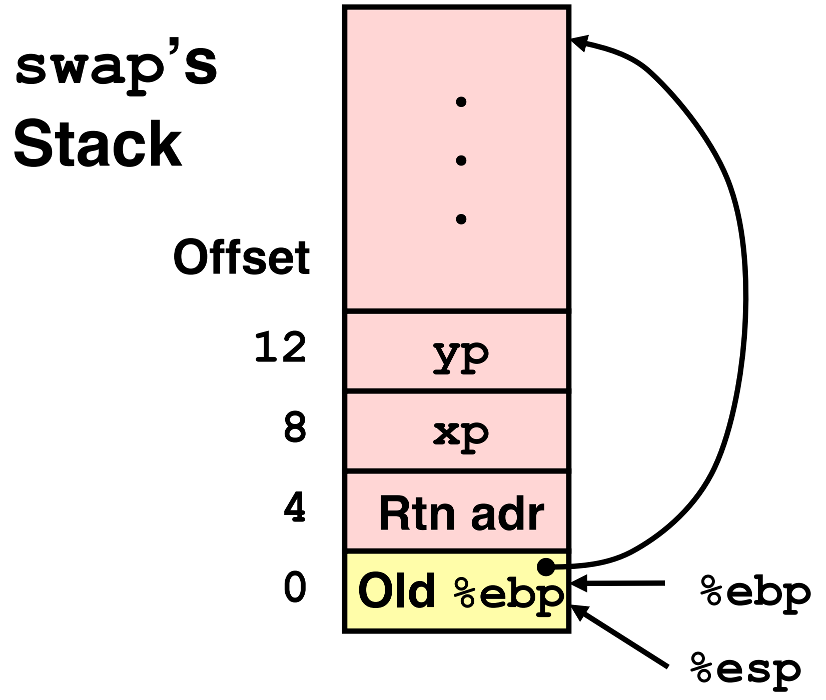
swap Finish #3

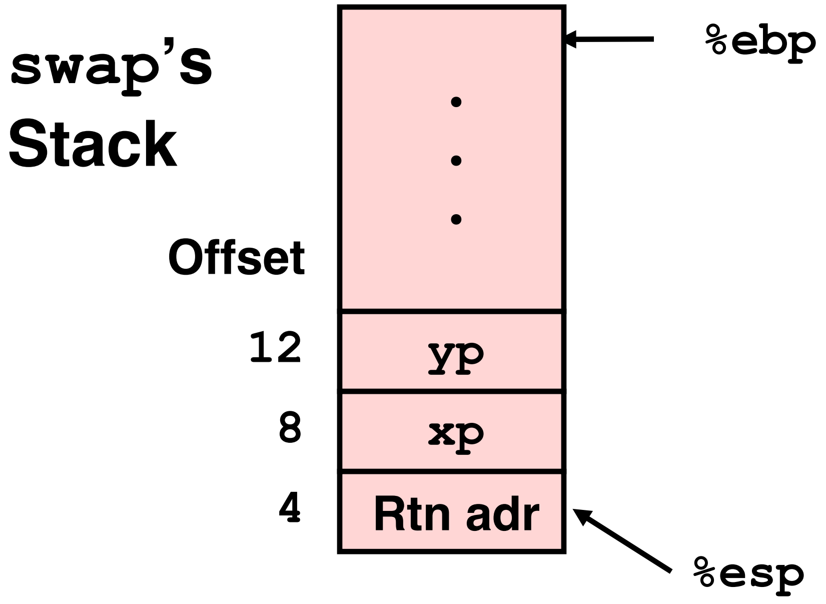




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

swap Finish #3

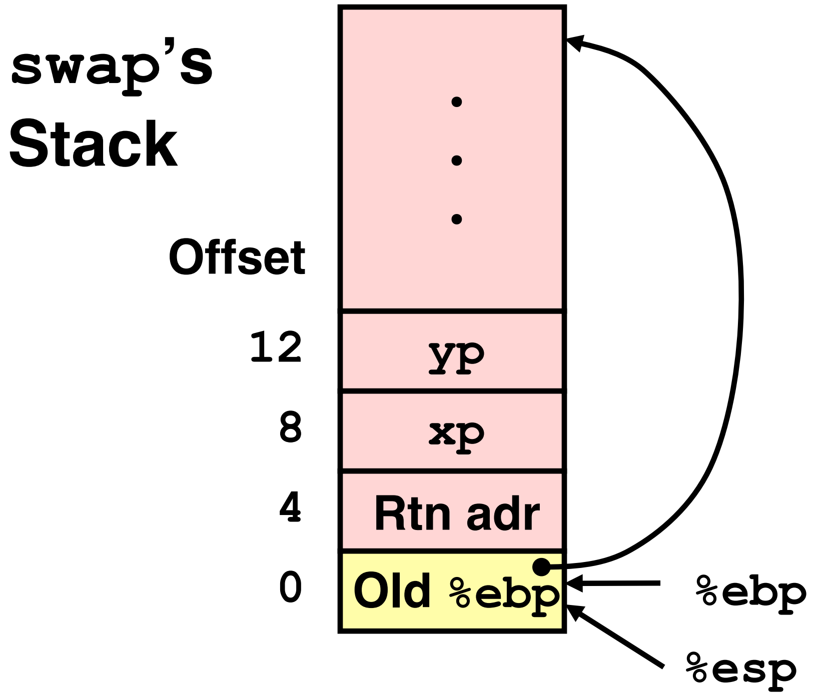


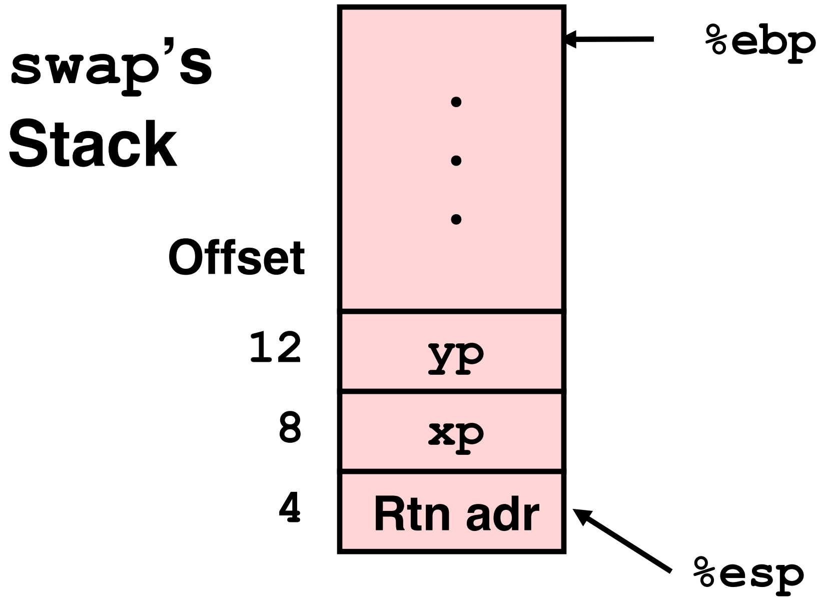


```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Pop address from stack & jump there

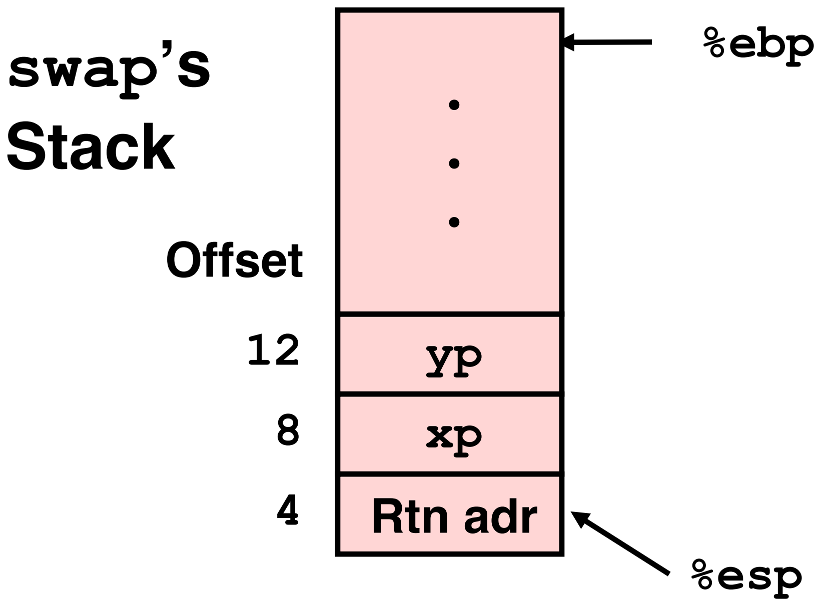
swap Finish #3

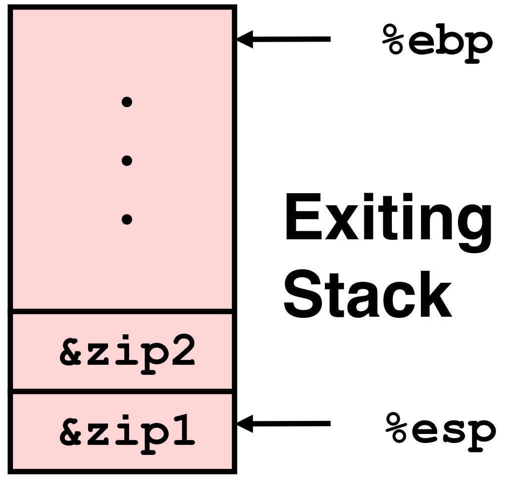




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

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

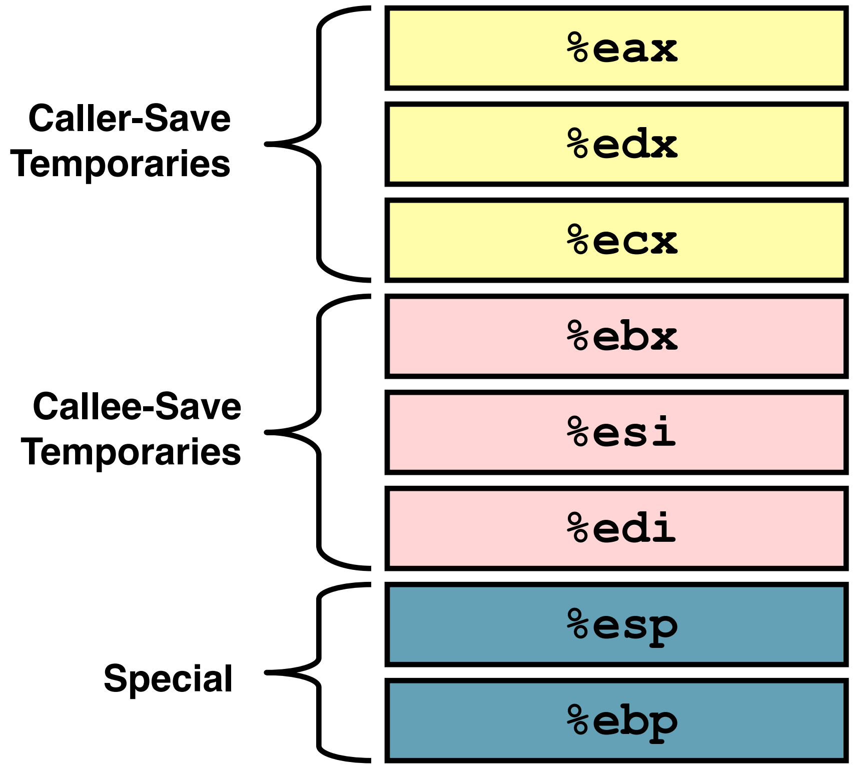
IA32/Linux register usage

- 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
```

Recursive factorial

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

```
.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
```

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

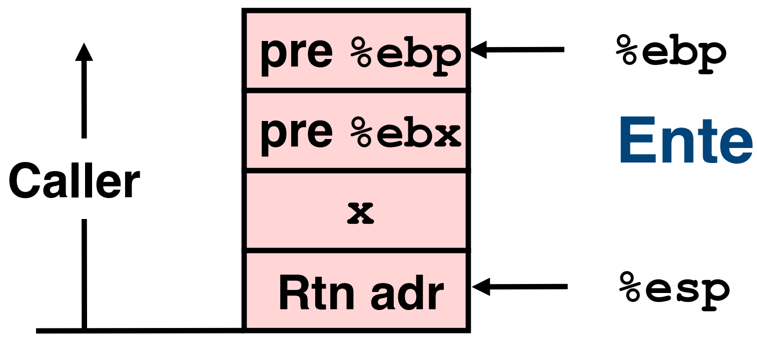
Recursive factorial

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

```
.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
```

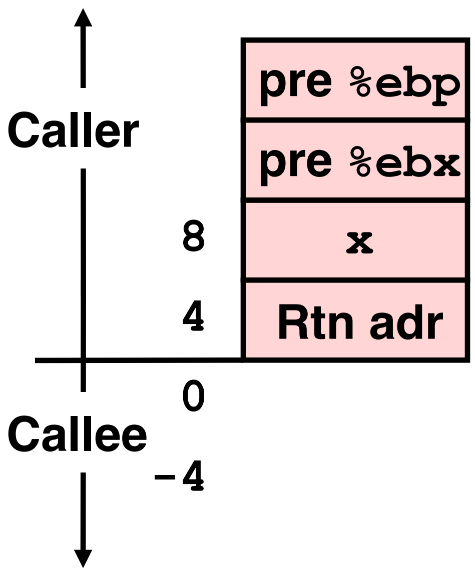
- Registers
  - `%eax` used without first saving
  - `%ebx` used, but save at beginning & restore at end

Rfact stack setup

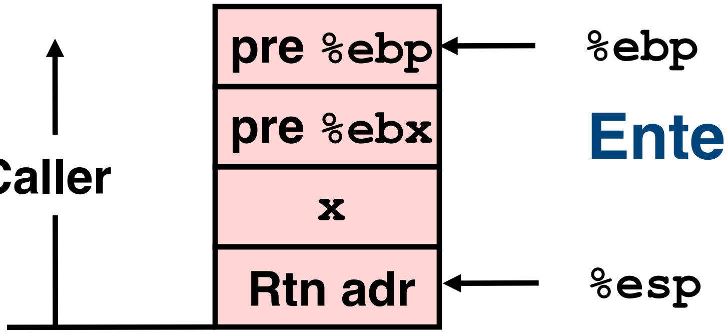


Entering Stack

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

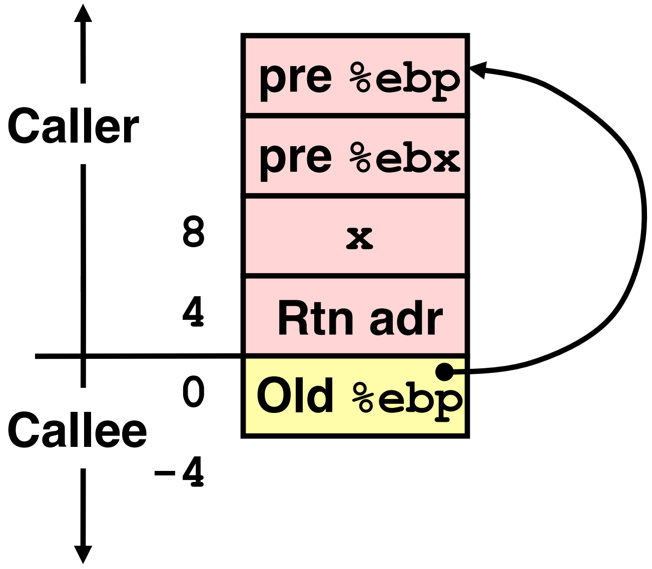


Rfact stack setup

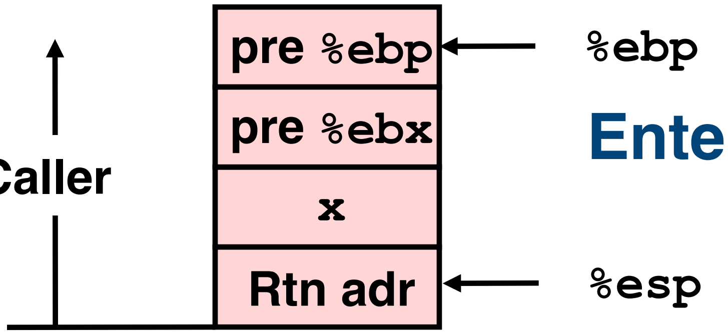


Entering Stack

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

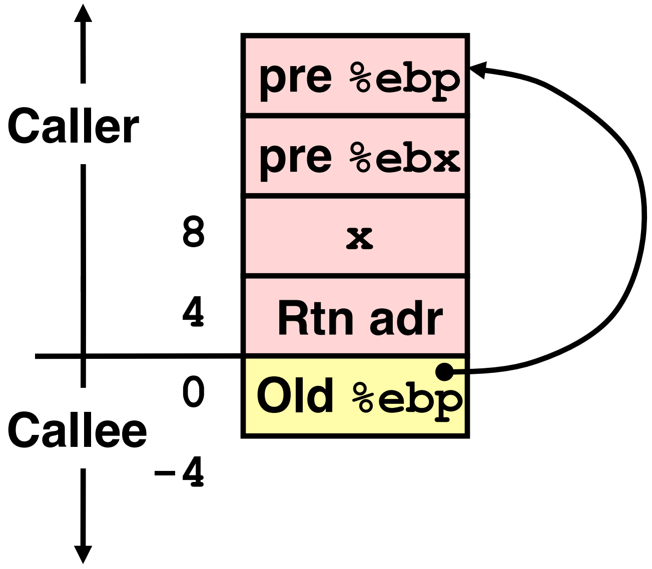


Rfact stack setup

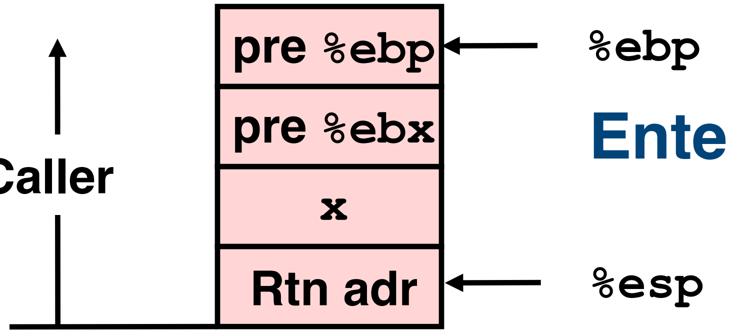


Entering Stack

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

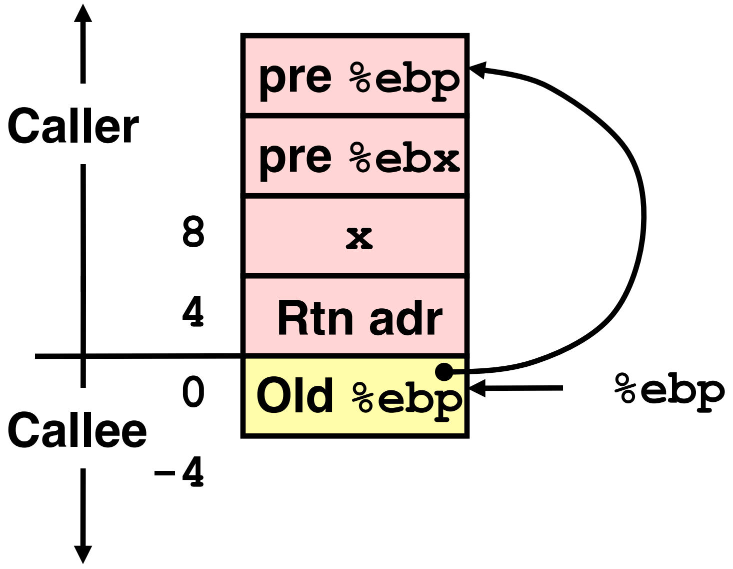


Rfact stack setup

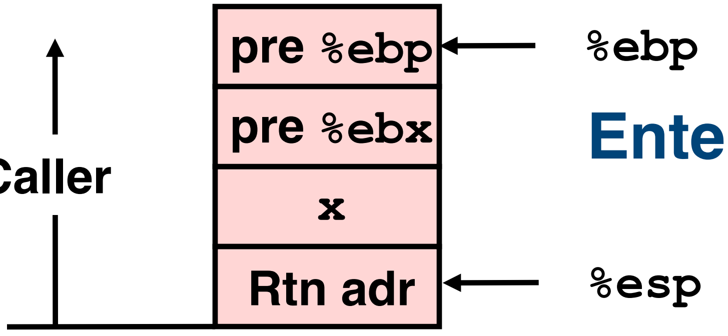


Entering Stack

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

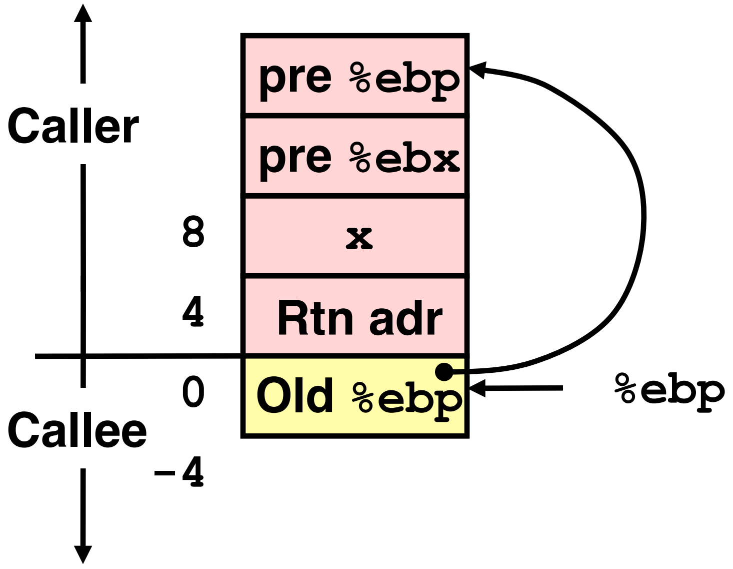


Rfact stack setup

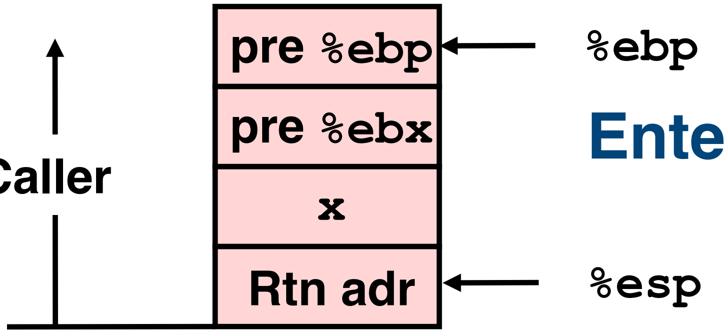


Entering Stack

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

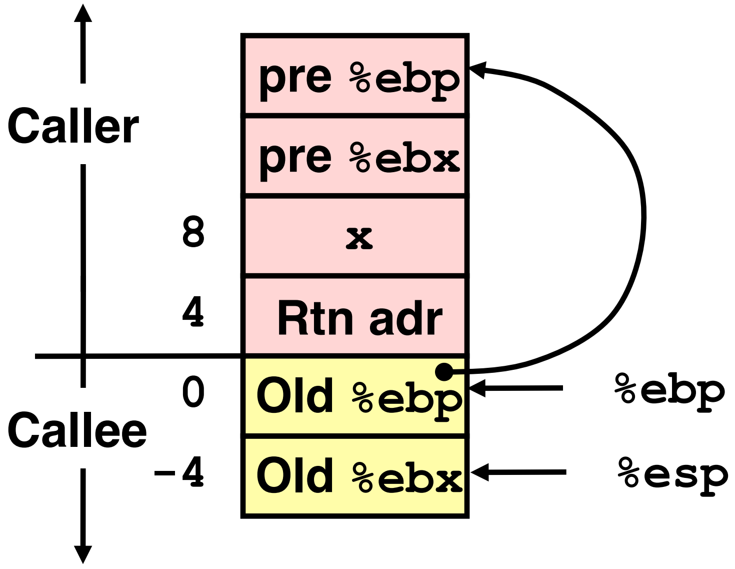


Rfact stack setup



Entering Stack

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



Rfact body

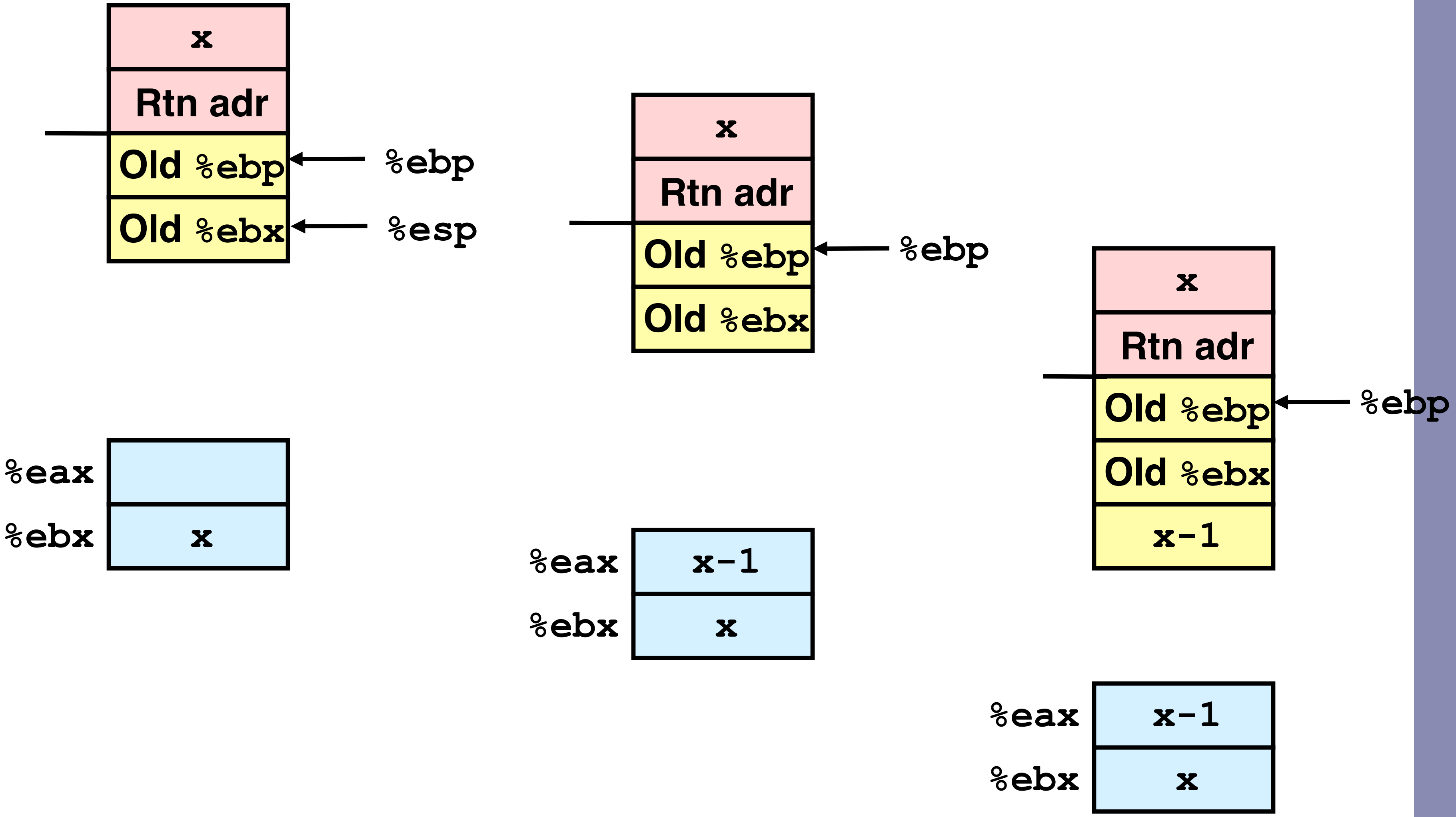
Recursion

```
   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;
}
```

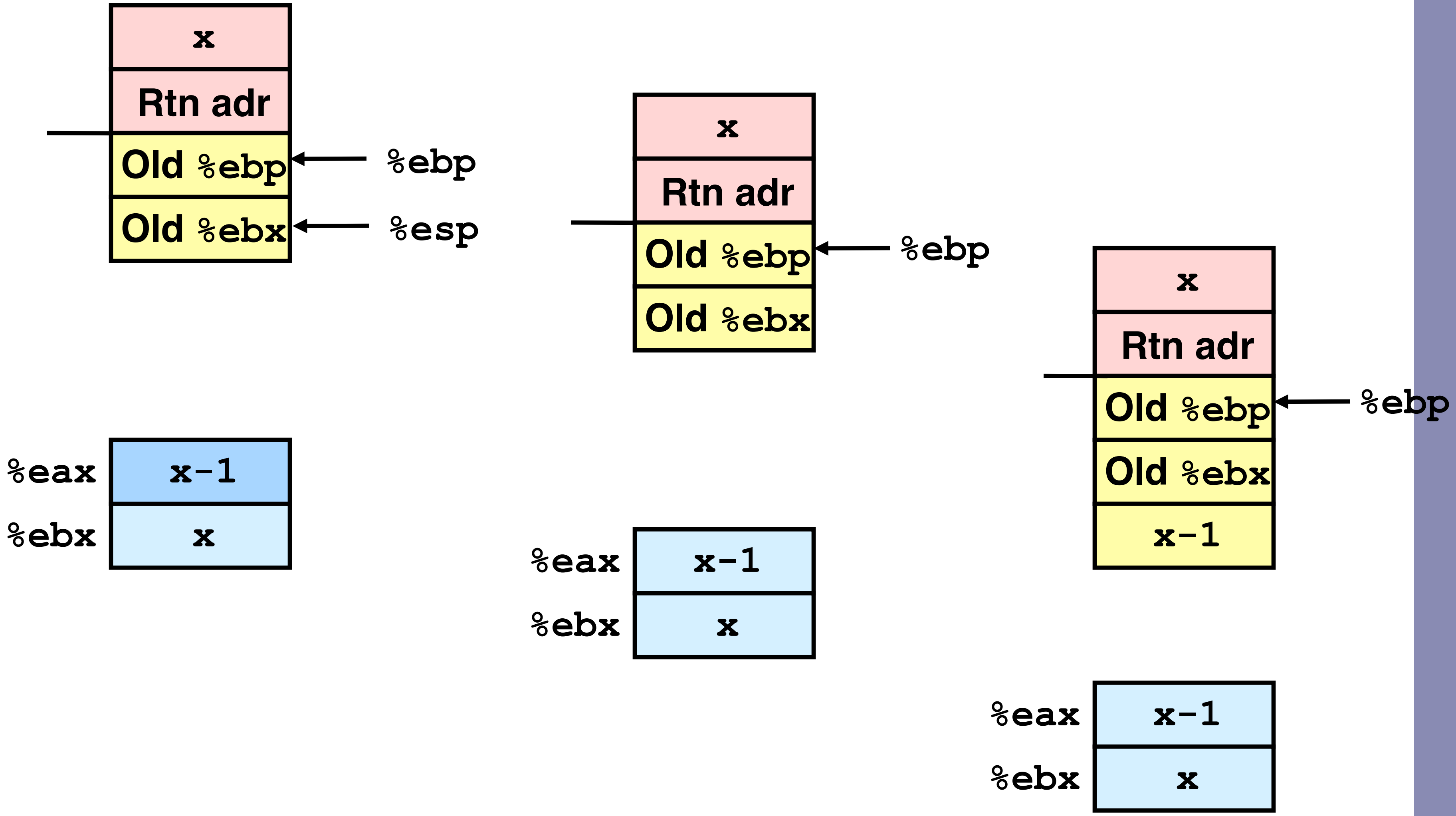
- Registers
  - `%ebx` Stored value of x
  - `%eax`
    - Temporary value of `x-1`
    - Returned value from `rfact(x-1)`
    - Returned value from this call

Rfact recursion



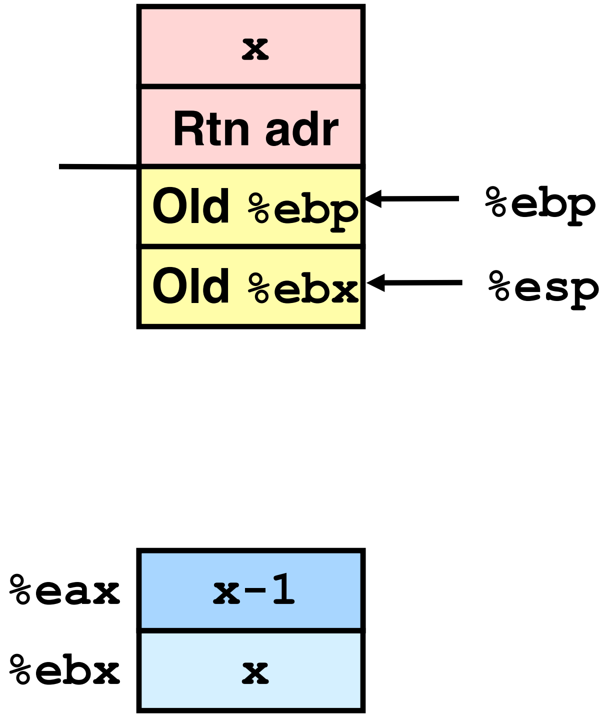
Rfact recursion

```
leal -1(%ebx),%eax
```

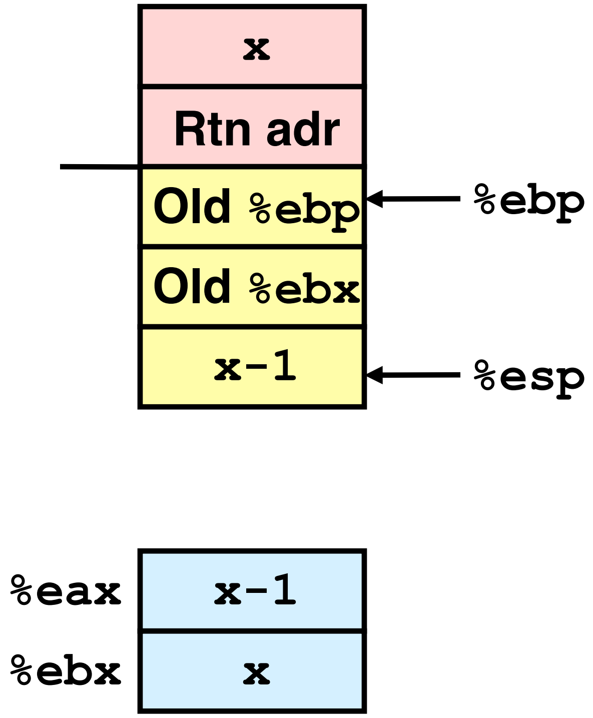


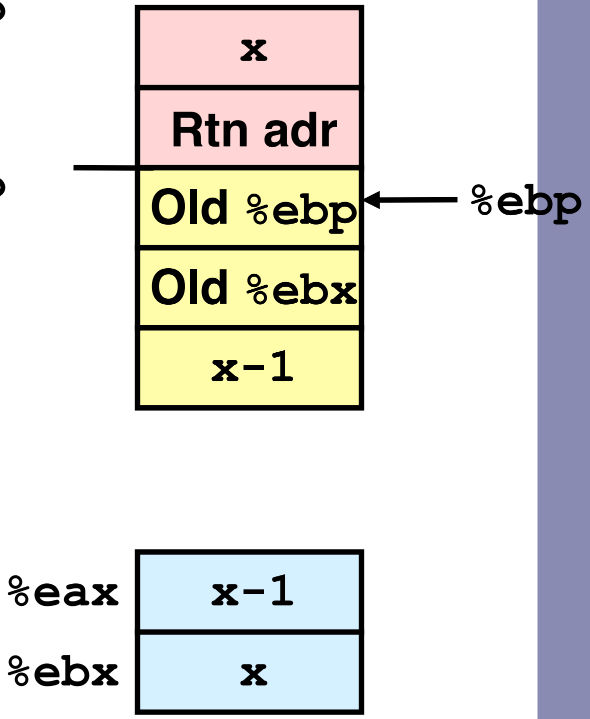
Rfact recursion

```
leal -1(%ebx),%eax
```



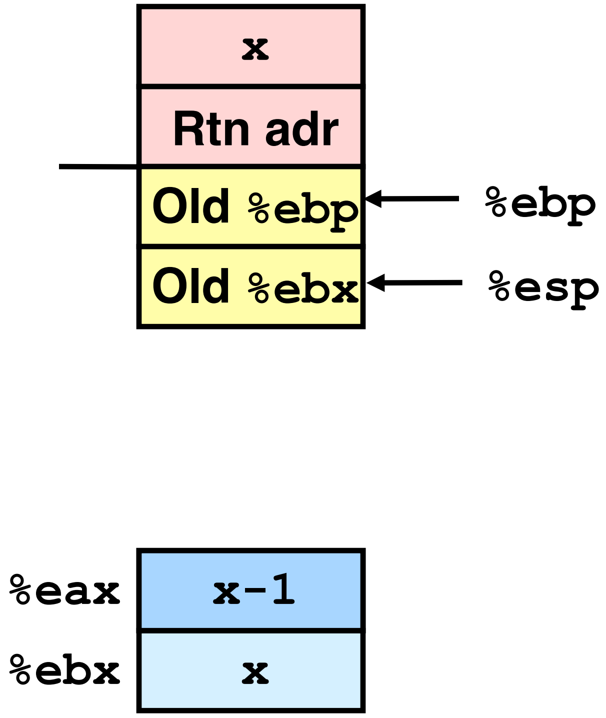
```
pushl %eax
```



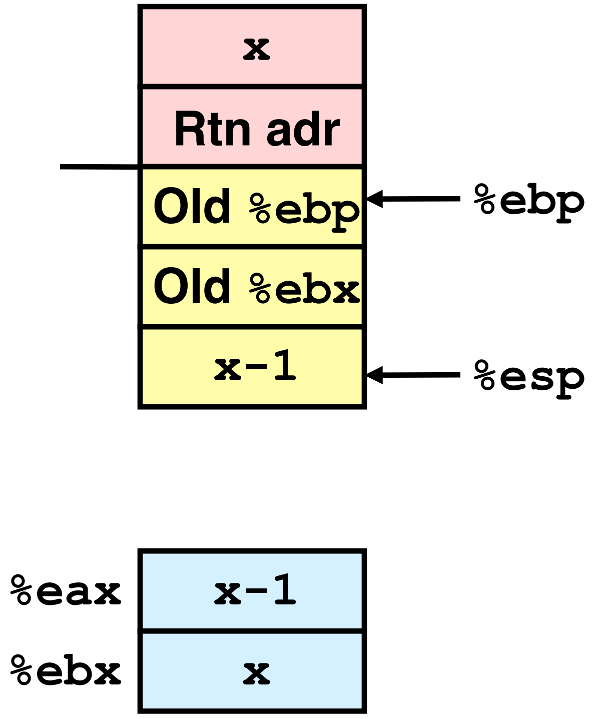


Rfact recursion

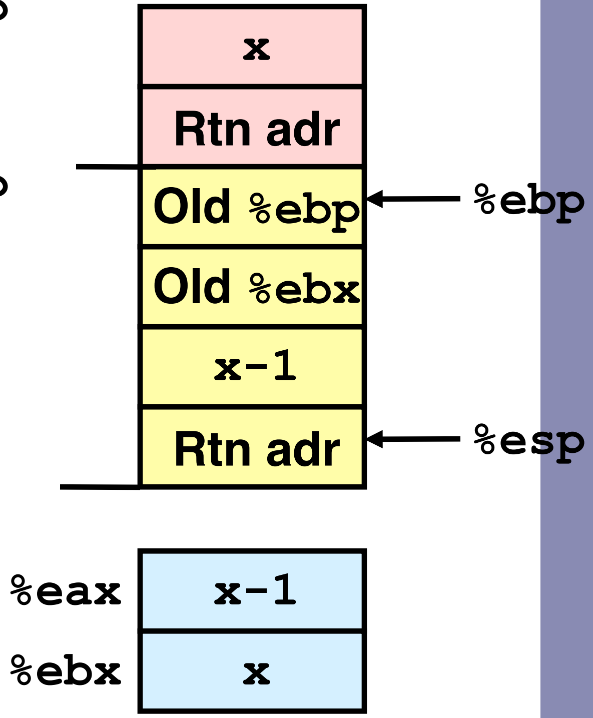
```
leal -1(%ebx),%eax
```



```
pushl %eax
```

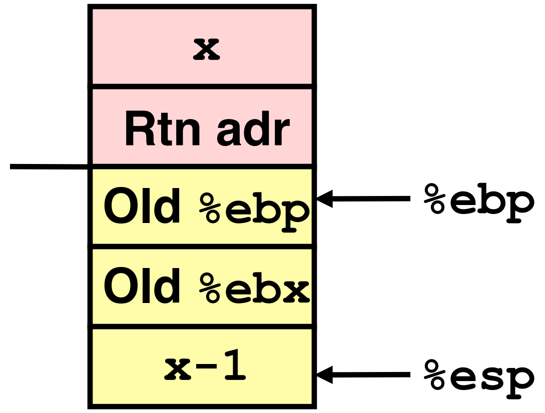


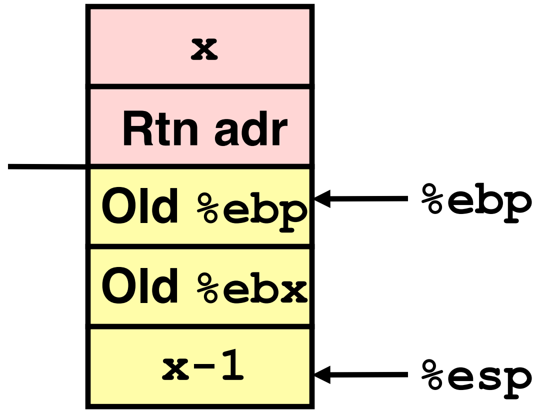
```
call rfact
```

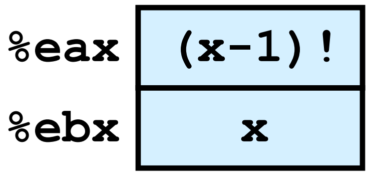


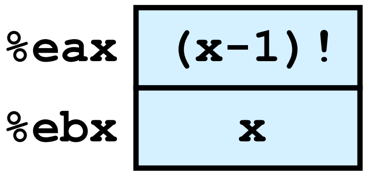
Rfact result

Return from Call





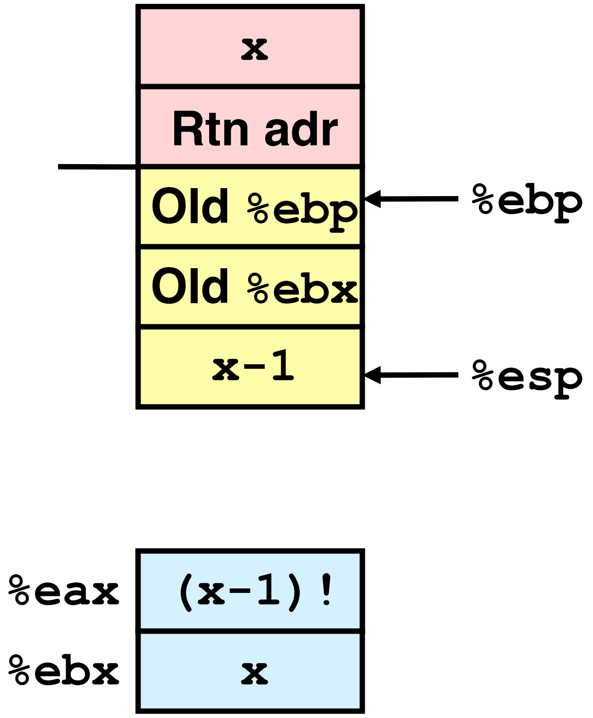




Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

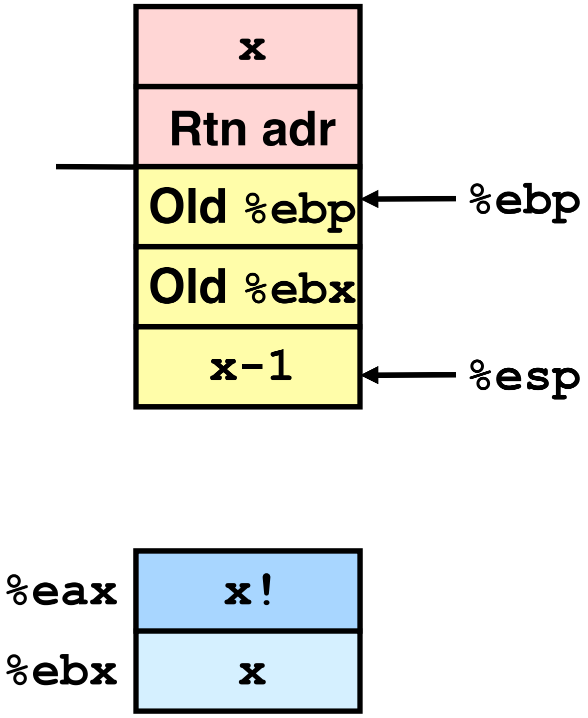
Rfact result

Return from Call

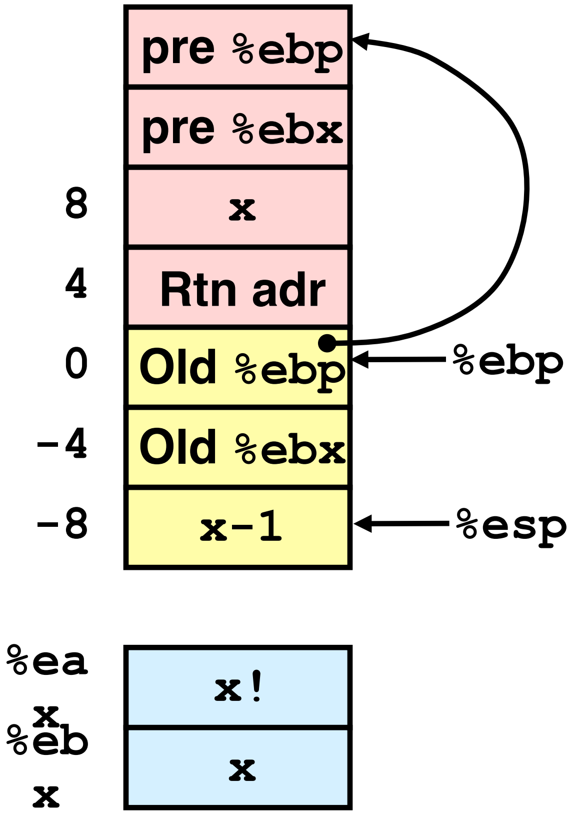


Assume that `rfact(x-1)` returns `(x-1)!` in register `%eax`

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

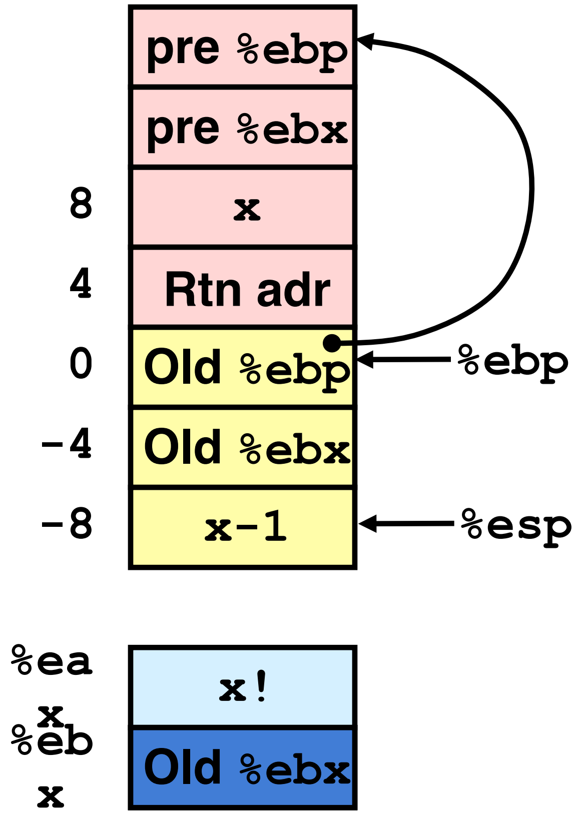


Rfact completion



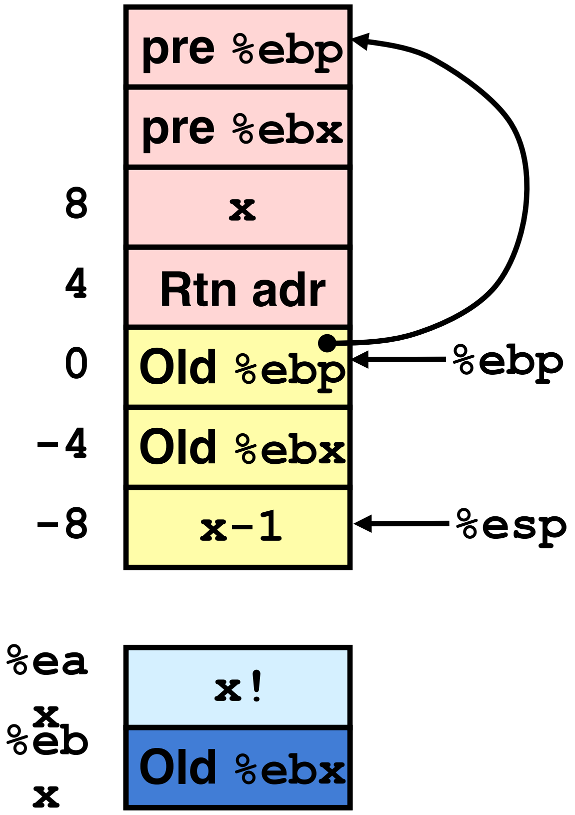
```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Rfact completion



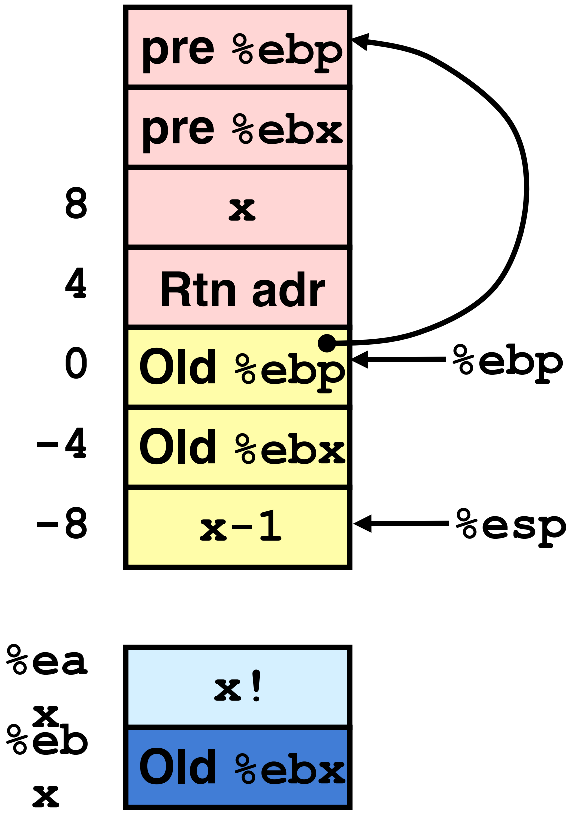
```
movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret
```

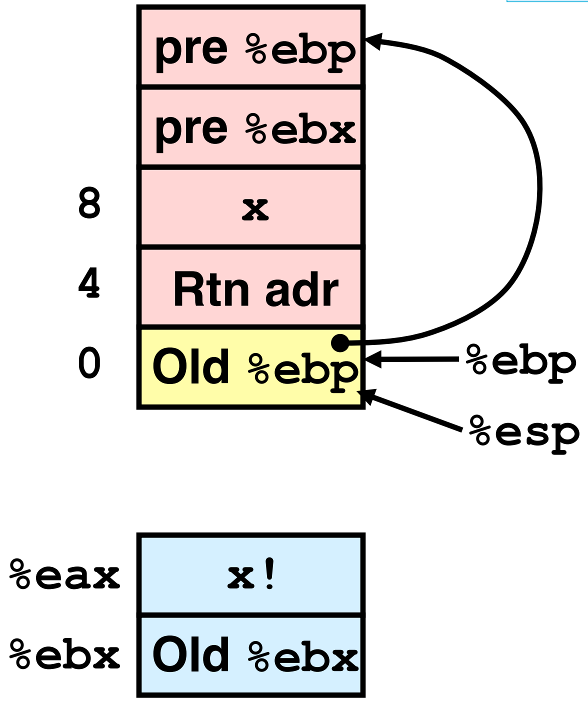
Rfact completion



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

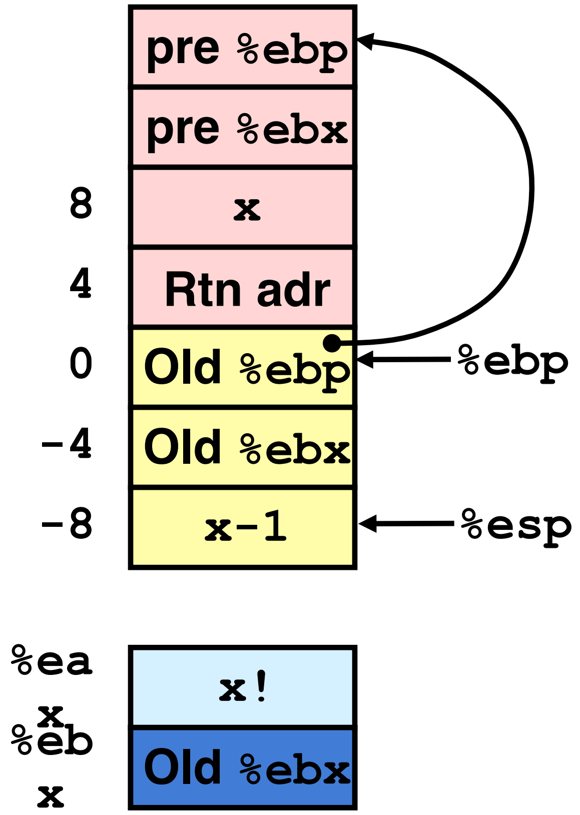
Rfact completion

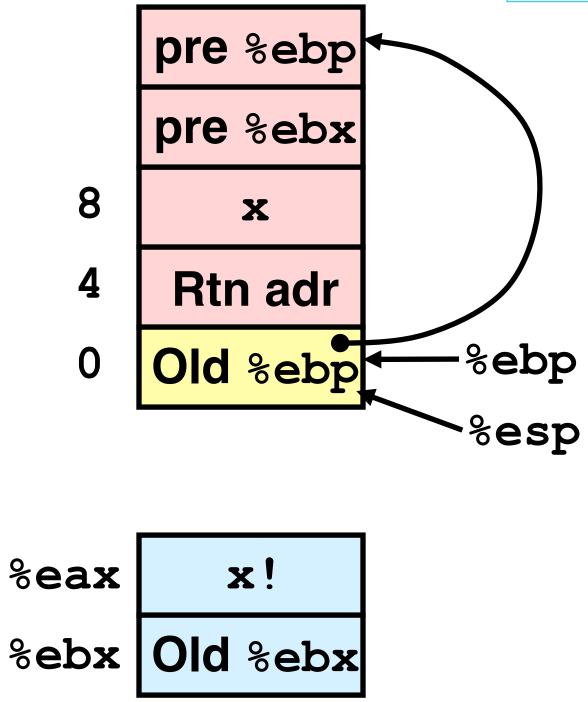




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Rfact completion

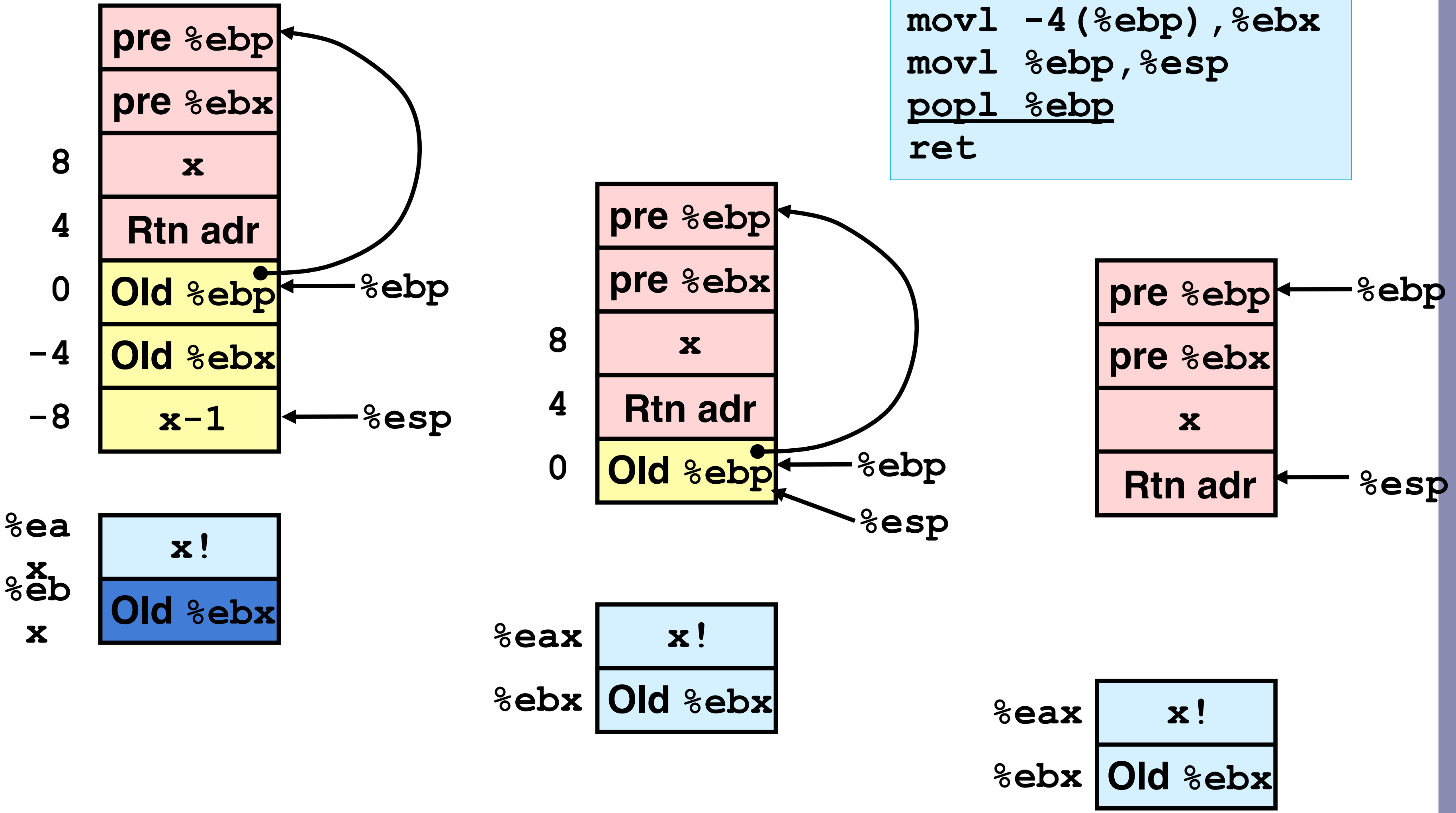




```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Rfact completion

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



Checkpoint

Checkpoint

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