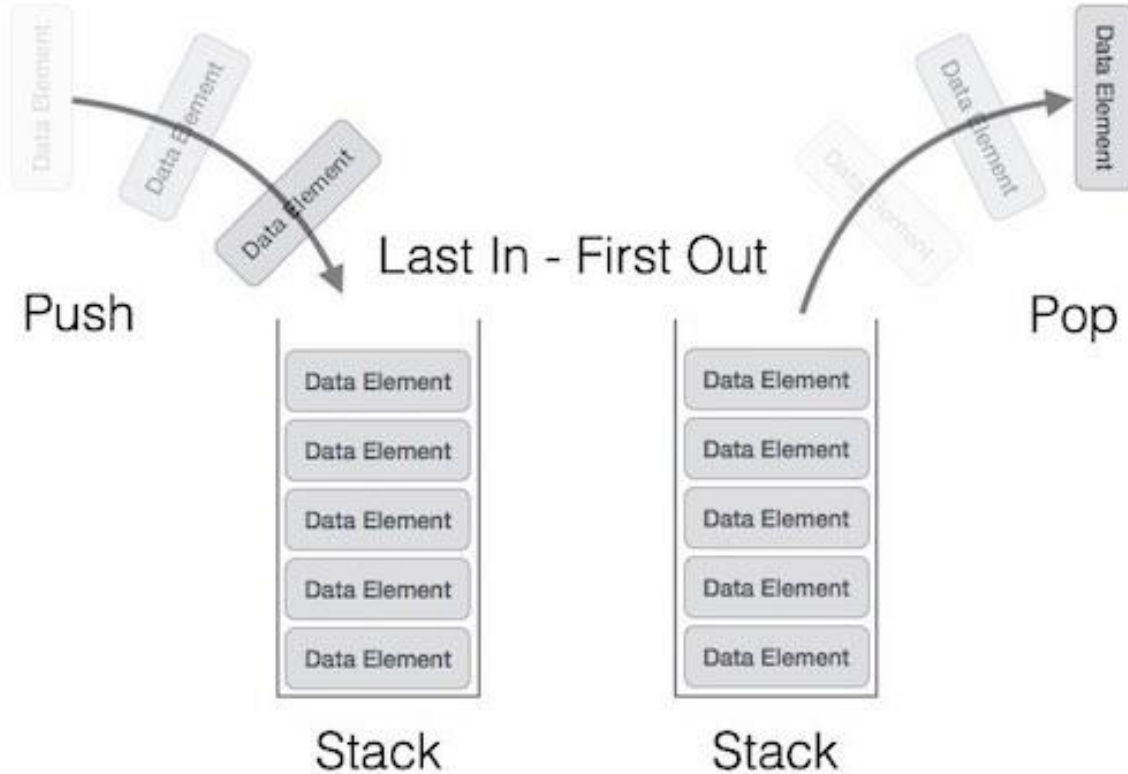


ASSEMBLY III: PROCEDURES

Jo, Heeseung

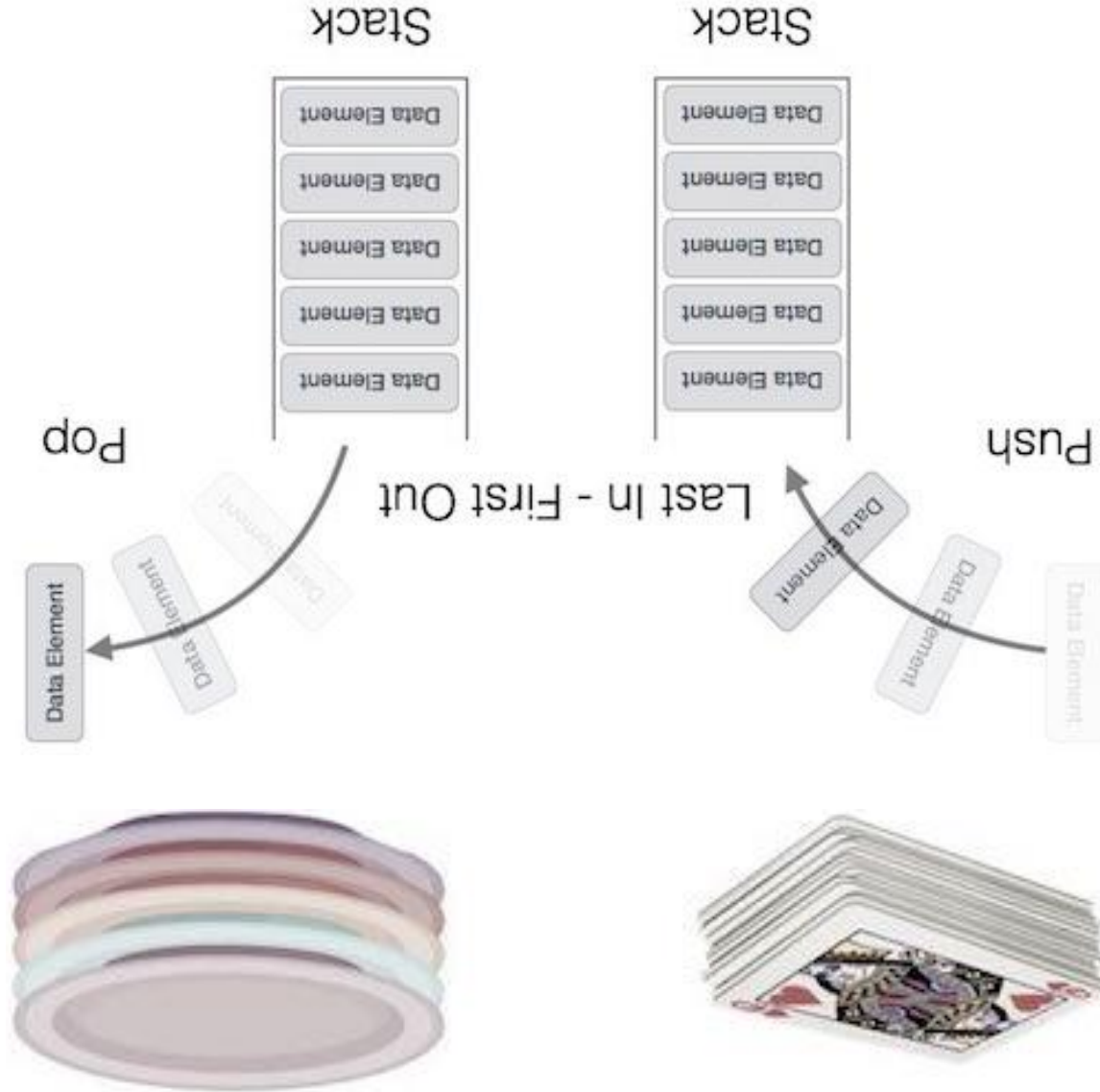
Stack structure

Stack



Stack structure

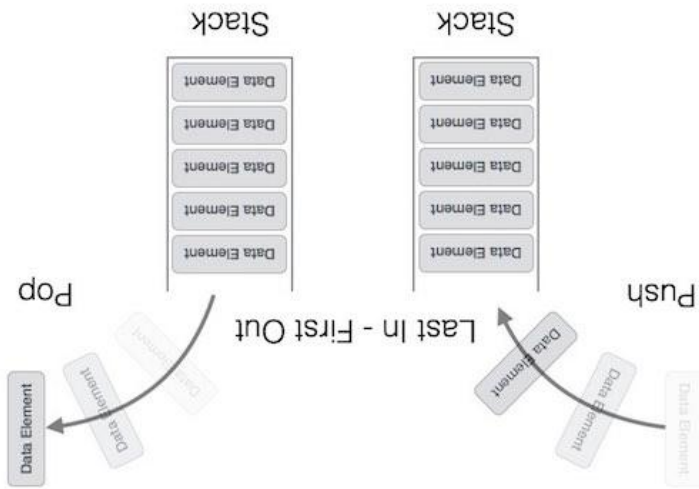
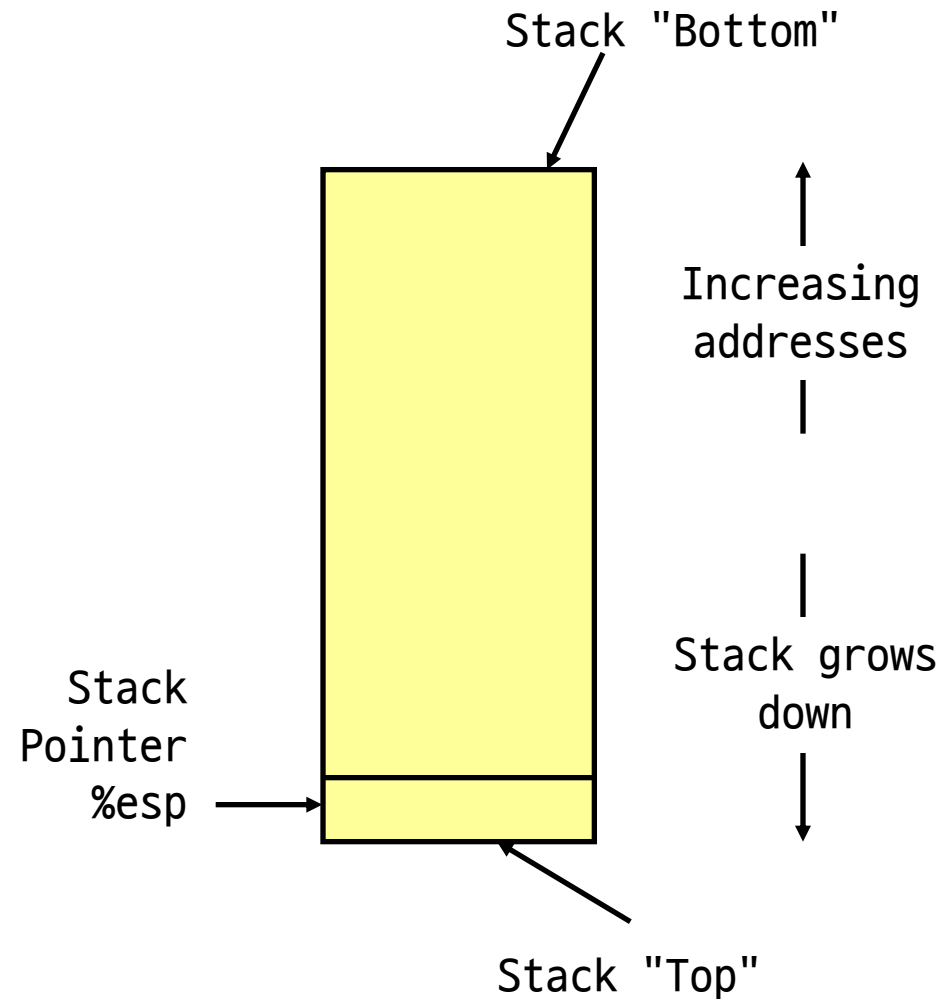
Stack



IA-32 Stack (1)

Characteristics

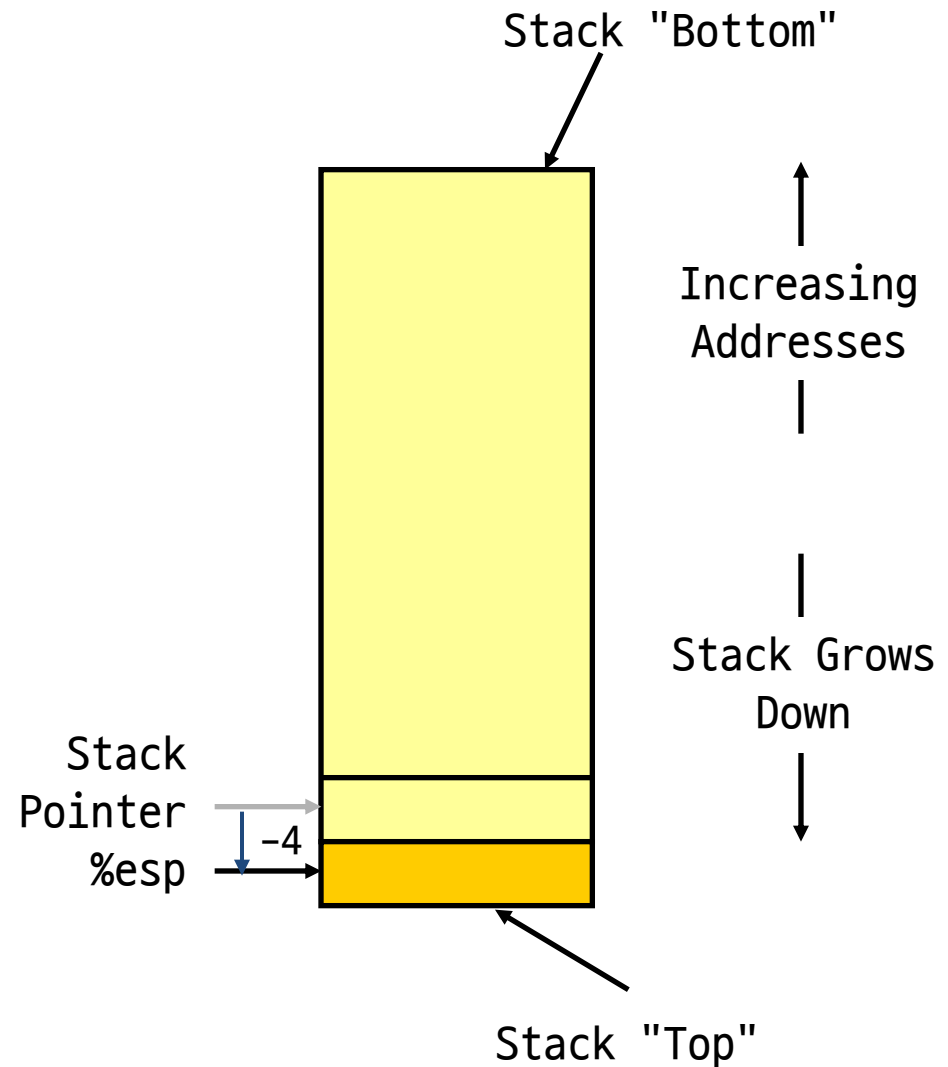
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
 - address of top element



IA-32 Stack (2)

Pushing

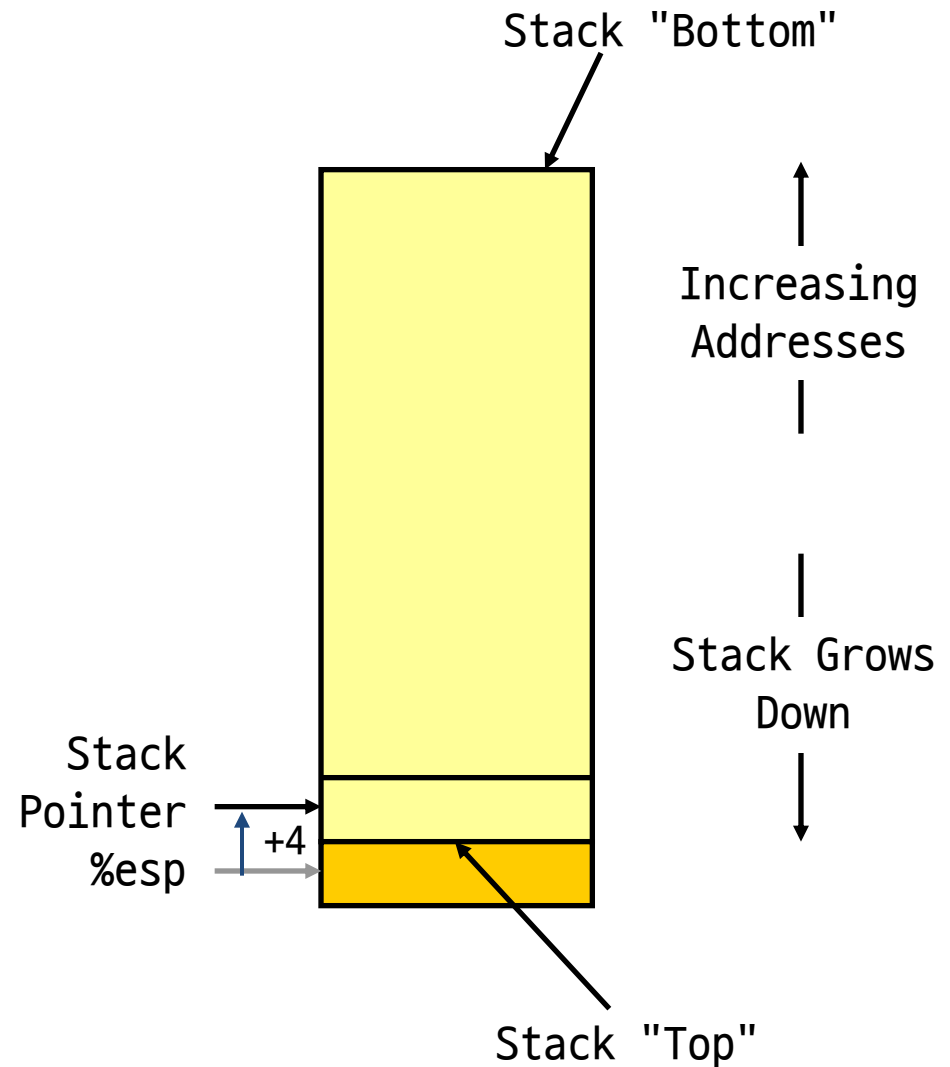
- `pushl Src`
- Fetch operand at Src
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



IA-32 Stack (3)

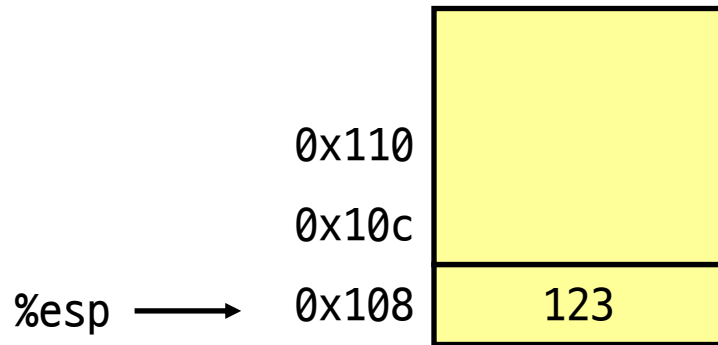
Popping

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



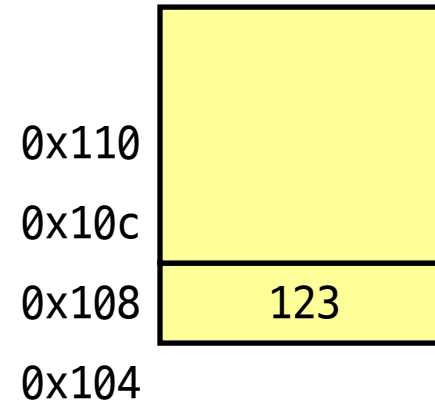
IA-32 Stack (4)

Stack operation examples



%eax	213
%edx	555
%esp	0x108

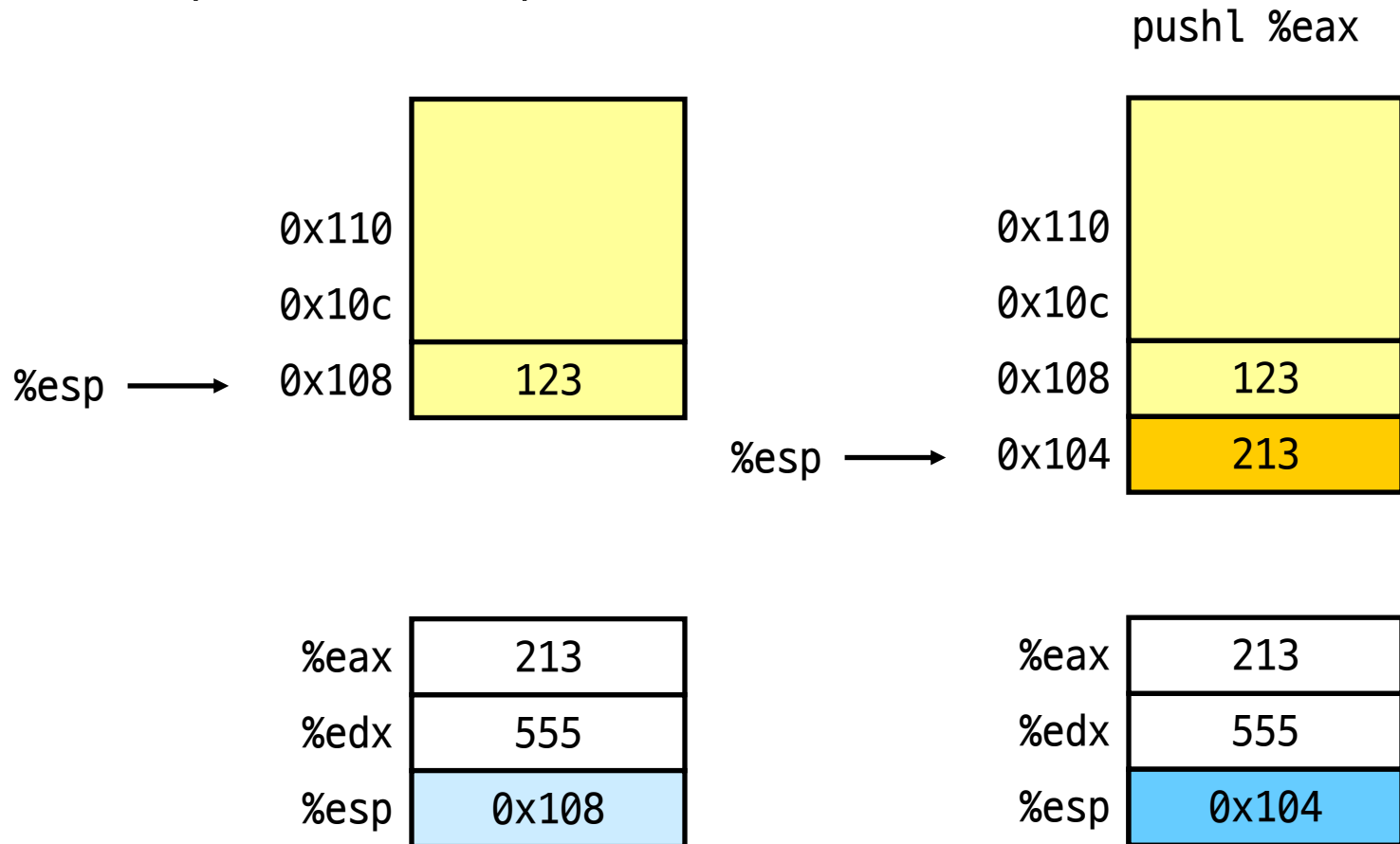
`pushl %eax`



%eax	213
%edx	555
%esp	0x104

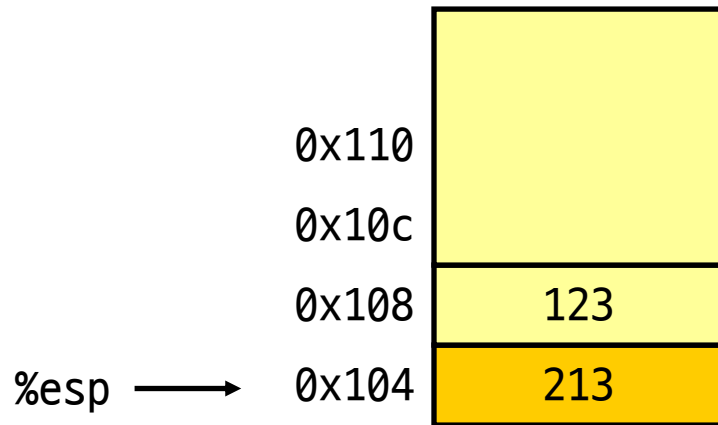
IA-32 Stack (4)

Stack operation examples



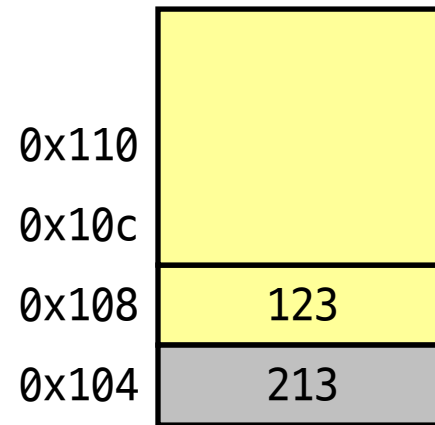
IA-32 Stack (4)

Stack operation examples



%eax	213
%edx	555
%esp	0x104

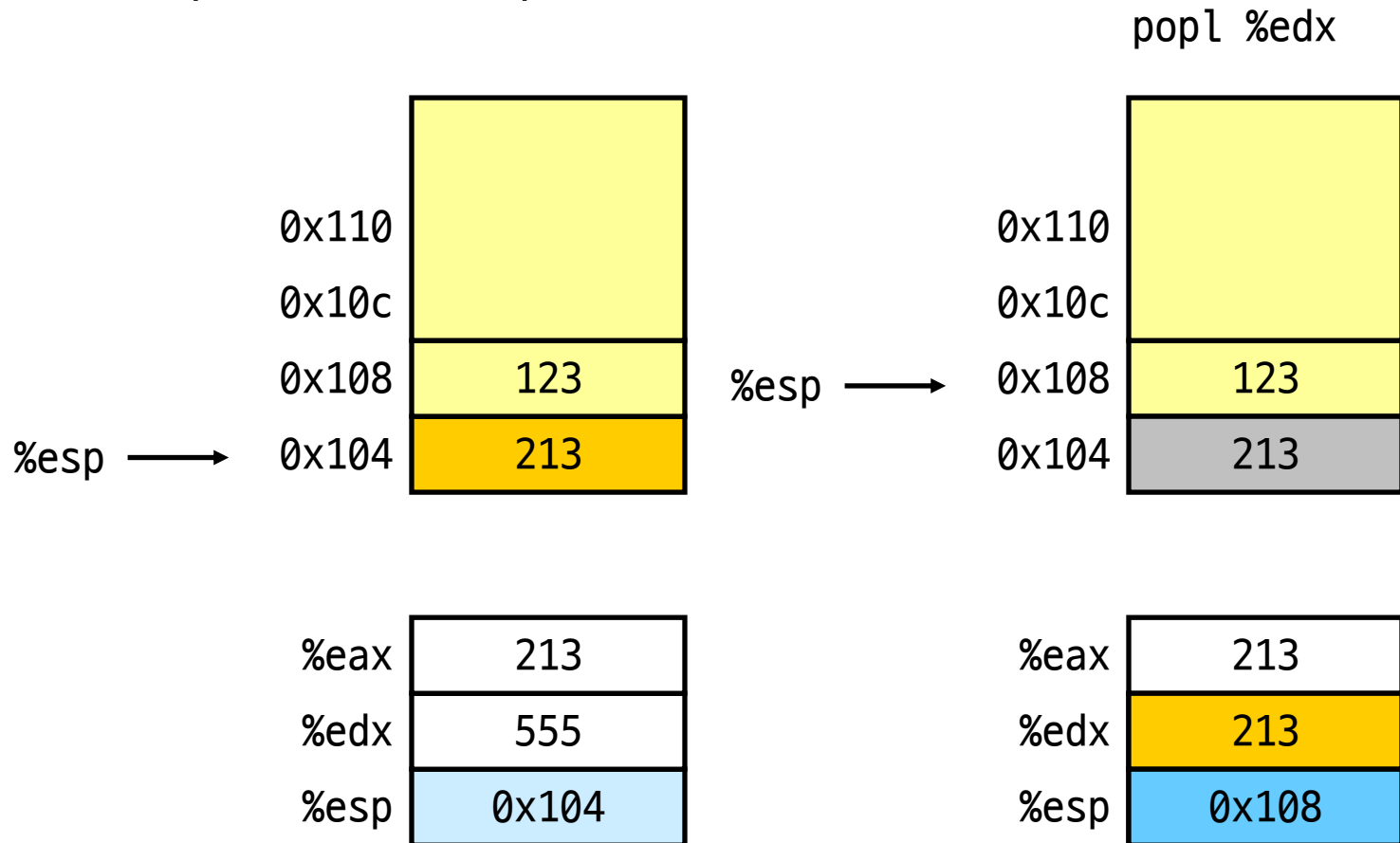
`popl %edx`



%eax	213
%edx	555
%esp	0x104

IA-32 Stack (4)

Stack operation examples



IA-32 Stack (5)

Exercise

0x110	
0x10c	456
0x108	123
0x104	213

%eax	213
%edx	555
%esp	0x108

pushl %edx

0x110	
0x10c	
0x108	
0x104	

%eax	
%edx	
%esp	

pushl %eax

0x110	
0x10c	
0x108	
0x104	

%eax	
%edx	
%esp	

IA-32 Stack (6)

Exercise

`movl %edx, %esp`

`popl %eax`

0x110	
0x10c	456
0x108	123
0x104	213

0x110	
0x10c	
0x108	
0x104	

0x110	
0x10c	
0x108	
0x104	

%eax	213
%edx	0x10c
%esp	0x108

%eax	
%edx	
%esp	

%eax	
%edx	
%esp	

Procedure Control Flow

Use stack to support procedure call and return

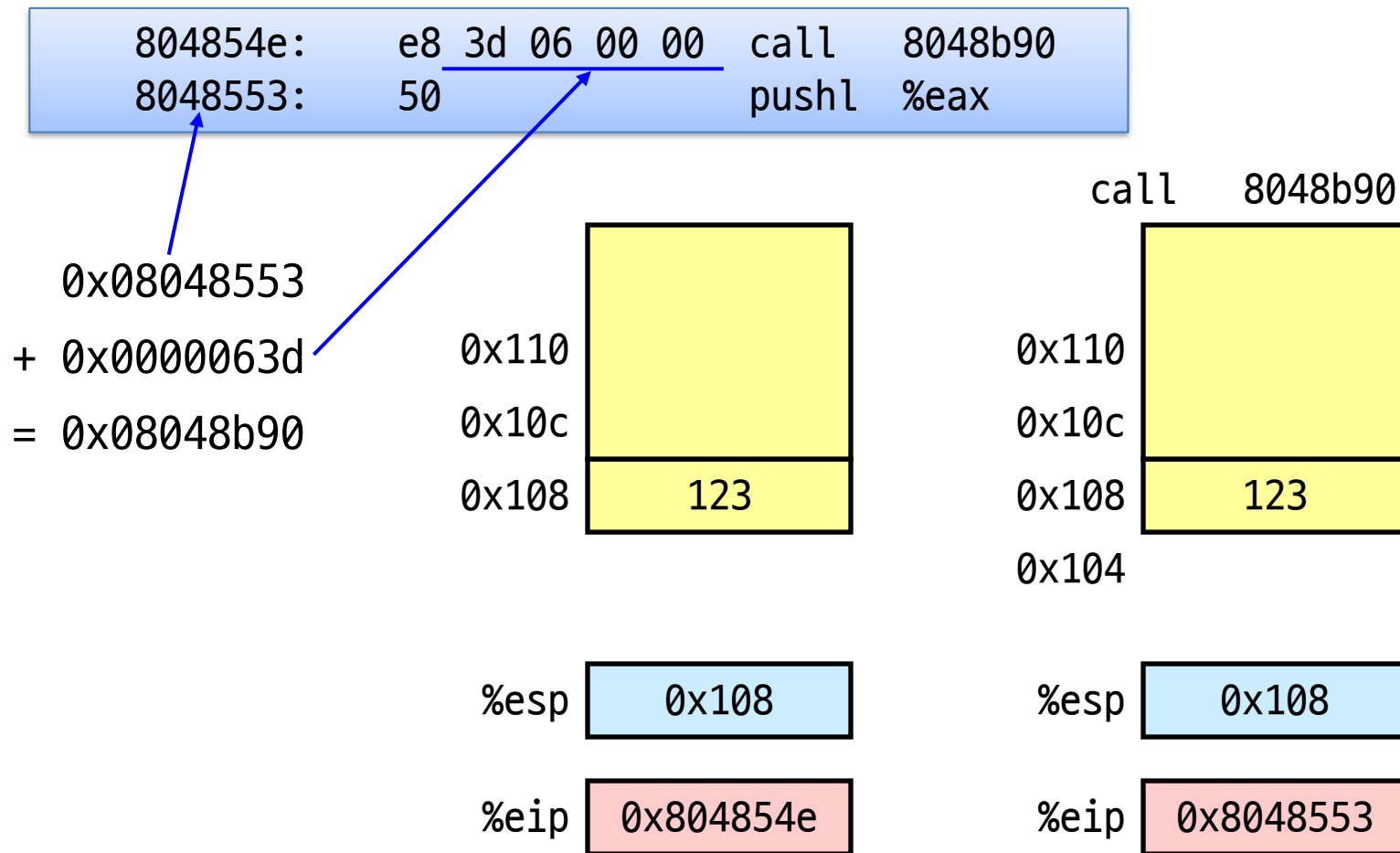
Procedure call

- `call label`
 1. Push return address (current EIP register value) on stack
 2. Jump to `label`
- Return address value
 - Address of instruction beyond call

Procedure return

- `ret`
 1. Pop return address from stack
 2. Jump to address

Procedure Call Example

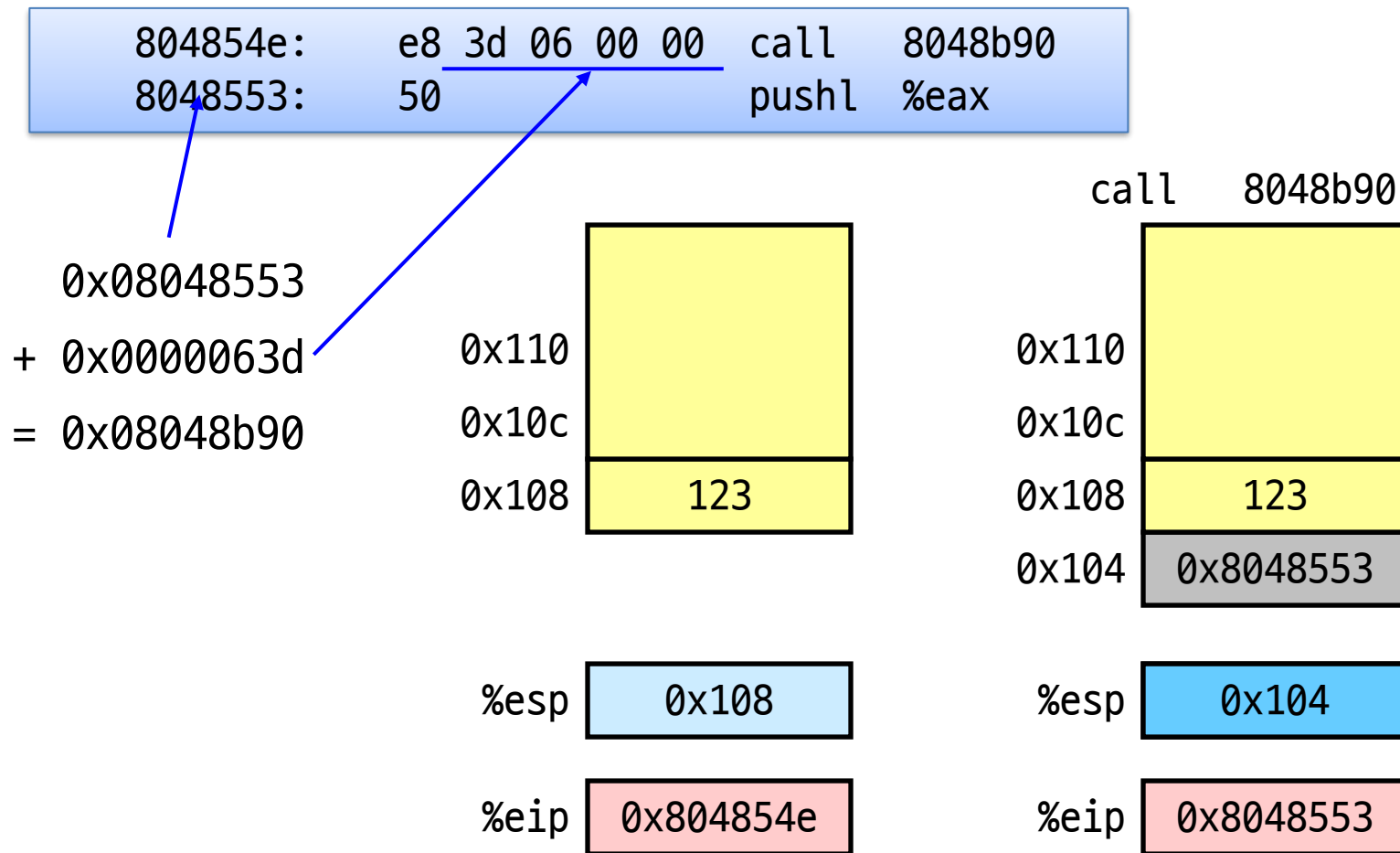


`%eip` is program counter

`call label`

1. Push return address (current EIP register value) on stack
2. Jump to *label*

Procedure Call Example

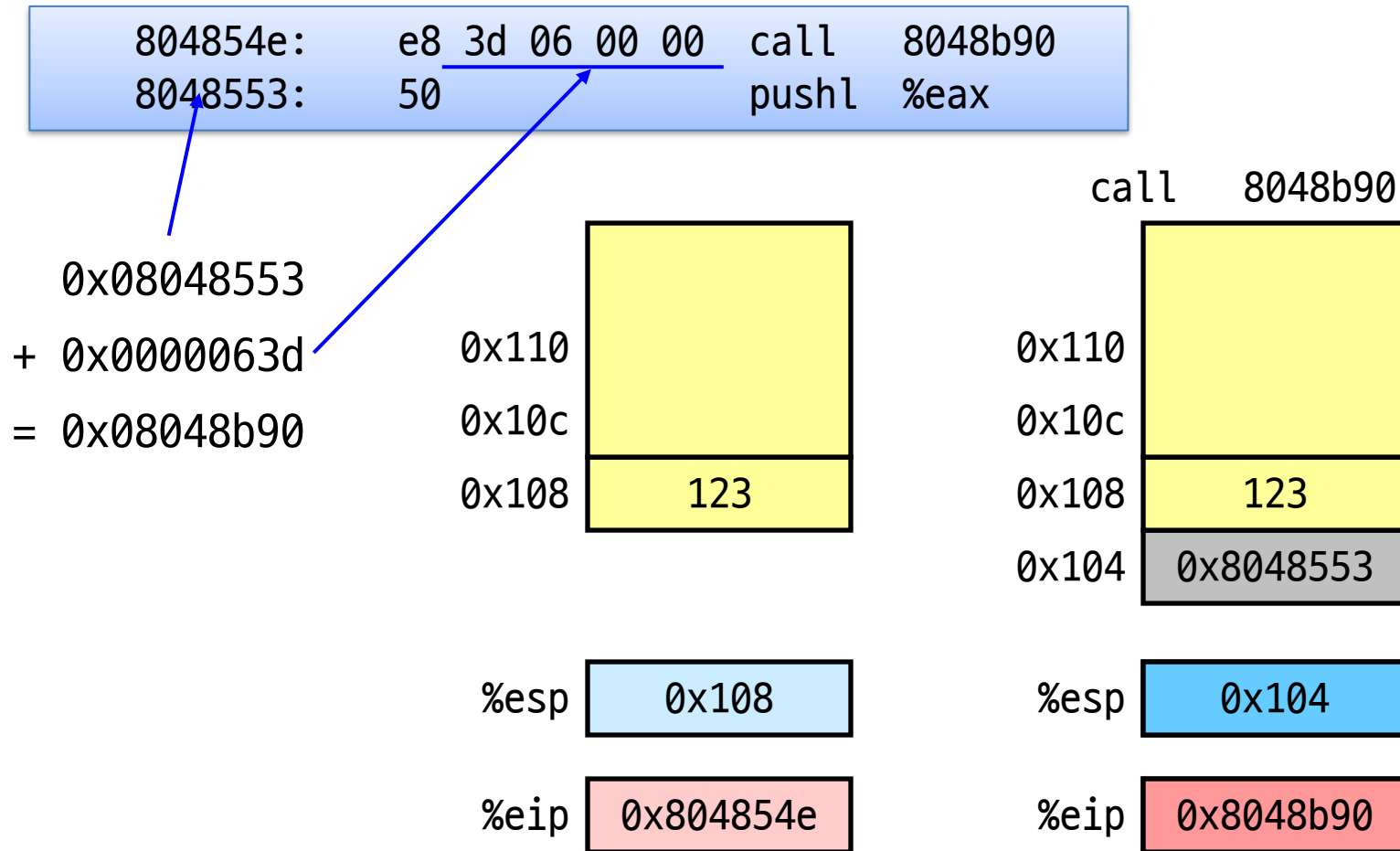


`%eip` is program counter

`call label`

1. Push return address (current EIP register value) on stack
2. Jump to *label*

Procedure Call Example

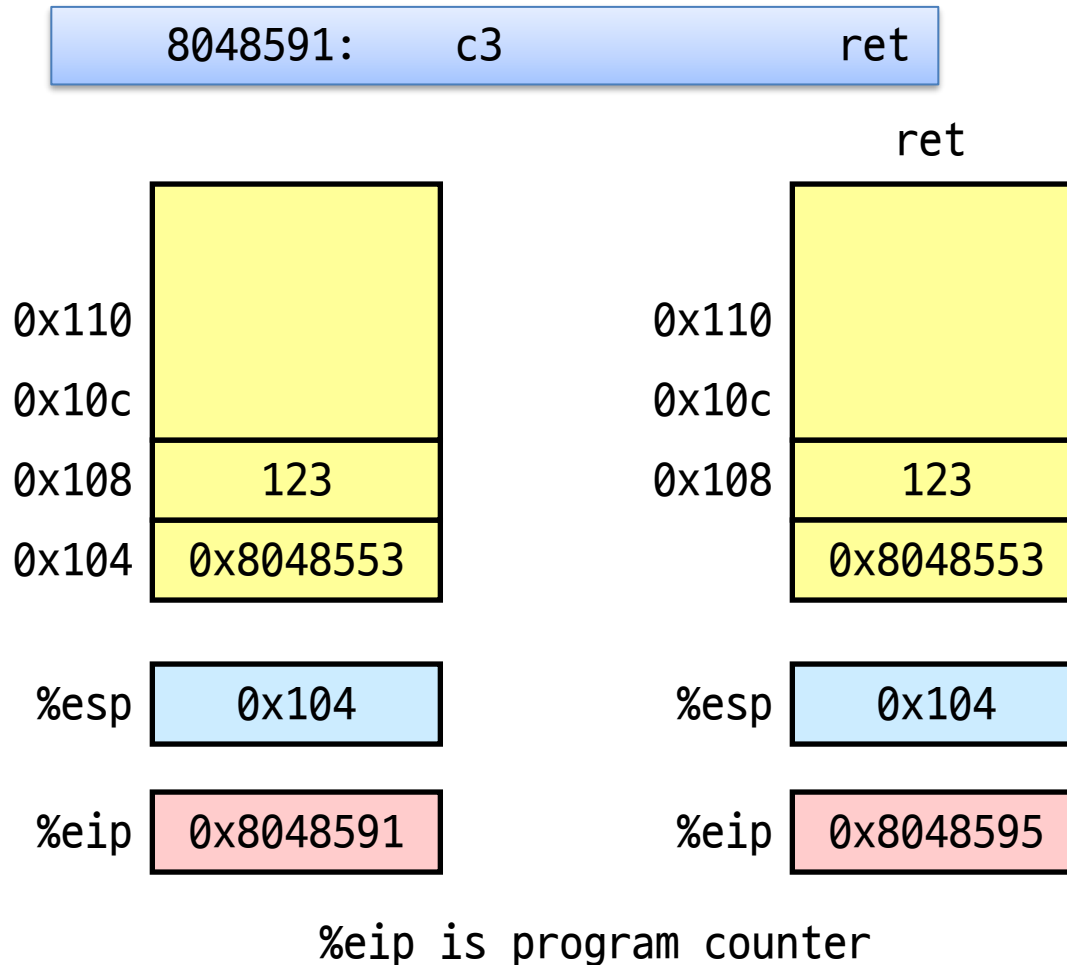


`%eip` is program counter

`call label`

1. Push return address (current EIP register value) on stack
2. Jump to *label*

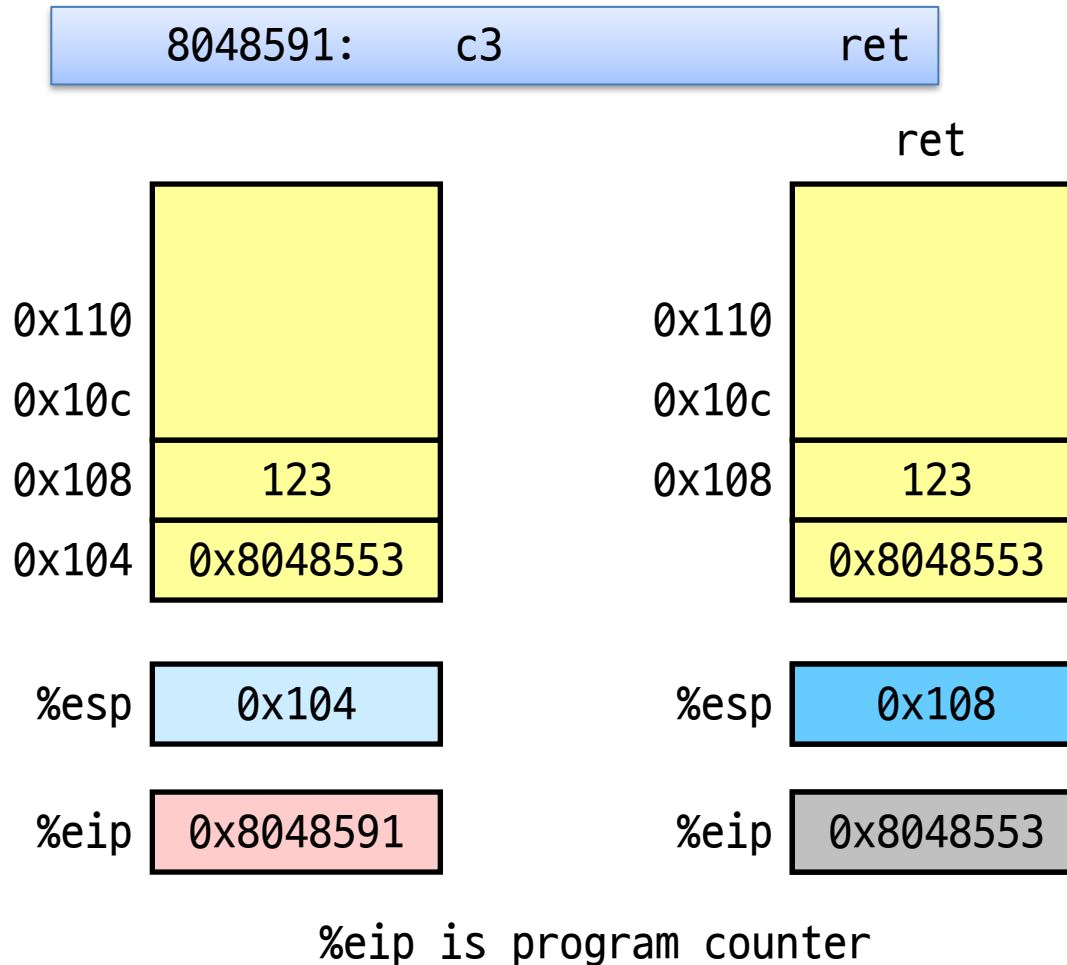
Procedure Return Example



ret

1. Pop return address from stack
2. Jump to address

Procedure Return Example



ret

1. Pop return address from stack
2. Jump to address

Return Value (1)

How does callee function send return value back to caller function?

In principle

- Store return value in stack frame of caller

Or, for efficiency

- Known small size => store return value in register
- Other => store return value in stack

Return Value (2)

IA-32 Convention

Integral type or pointer

- Store return value in **EAX register**
- char, short, int, long, pointer

Floating-point type

- Store return value in **floating point register**
- Beyond scope of course

Structure

- Store return value on **stack**
- Beyond scope of course

Register Saving Conventions (1)

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 (2)

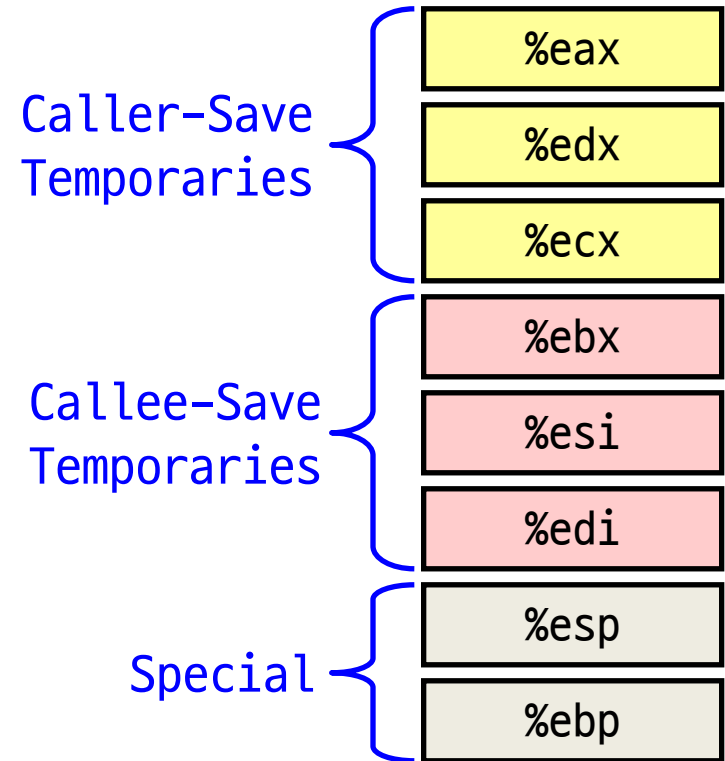
Conventions

- Caller save
 - Caller saves temporary in its frame before calling
- Callee save
 - Callee saves temporary in its frame before using
- IA-32 ?

IA-32/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 vs. Iteration

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

```
int whilefact(int x)
{
    int rval = 1;
    while (x > 1) {
        rval = rval * x;
        x = x - 1;
    }
    return rval;
}
```


Stack-based Languages (1)

Languages that support recursion

- e.g., C, Pascal, Java, etc.
- Code must be "Reentrant"
 - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantiation
 - Arguments, local variables, return pointer

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

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

Stack-based Languages (2)

Stack discipline

- State for given procedure needed for **limited time**
 - From when called to when return
- Callee returns before caller does

Stack frame

- **State for single procedure instantiation**

Stack Frames (1)

Code Structure

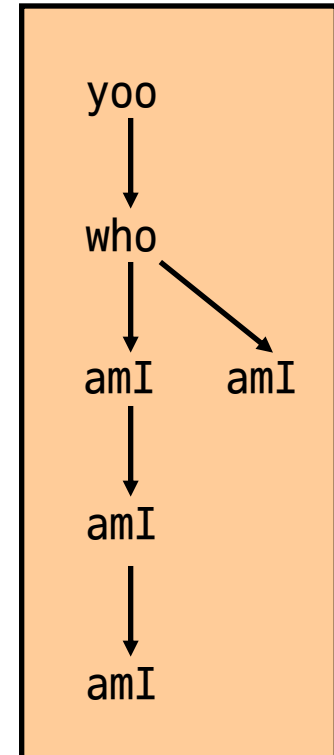
```
yoo(...)  
{  
  .  
  .  
  who();  
  .  
  .  
}
```

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

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

Procedure amI recursive

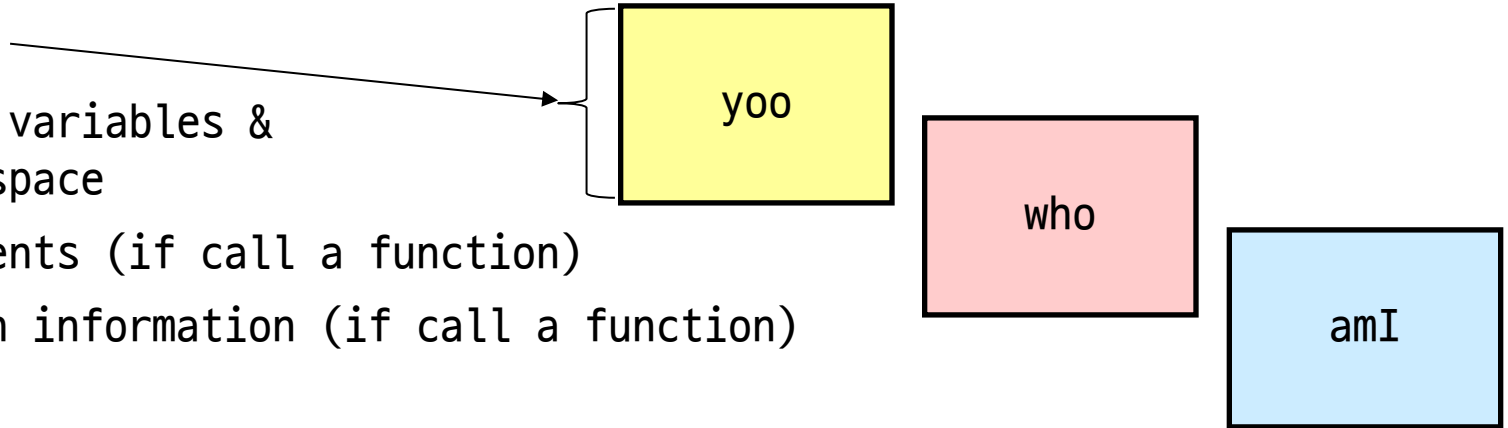
Call Chain



Stack Frames (2)

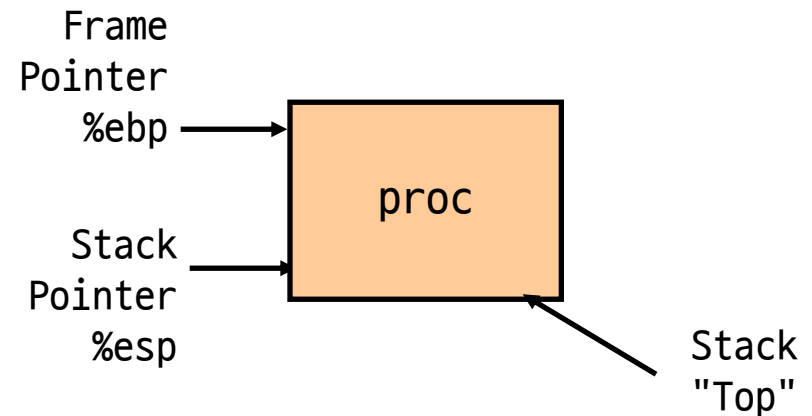
Contents

- Local variables & temp space
- Arguments (if call a function)
- Return information (if call a function)



Management

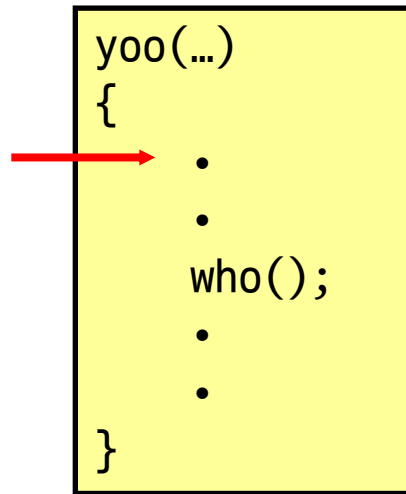
- Space allocated when enter procedure
 - "set-up" code
- Deallocated when return
 - "finish" code



Pointers

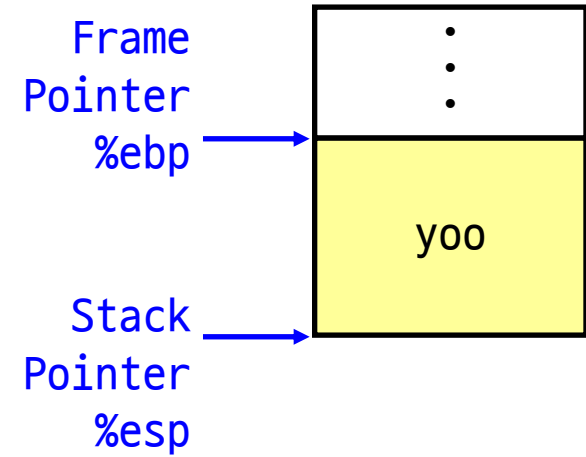
- Stack pointer `%esp` indicates stack top
- Frame pointer `%ebp` indicates start of current frame

Stack Frames (3)

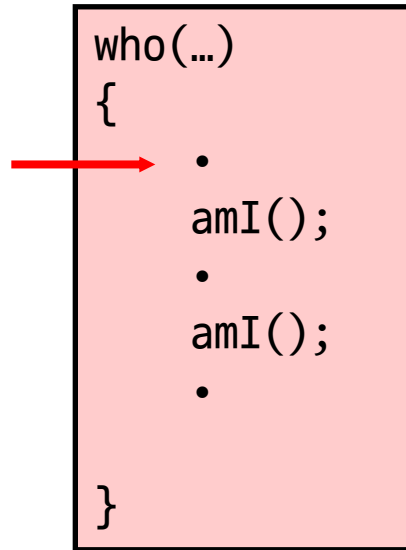


Call Chain

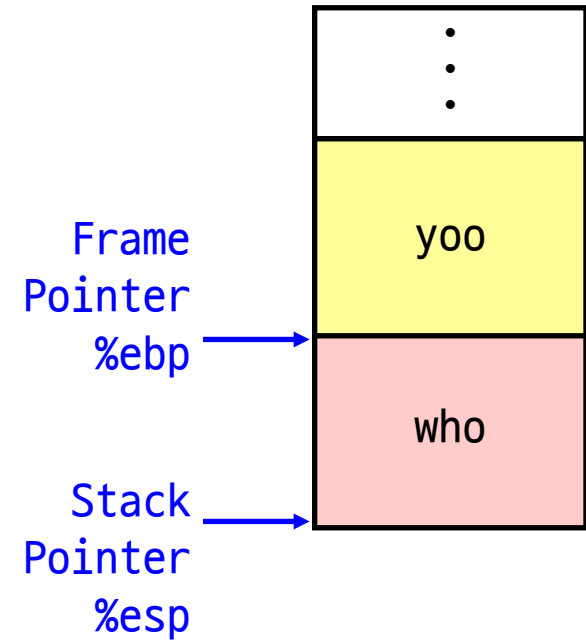
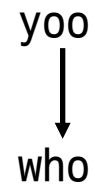
yoo



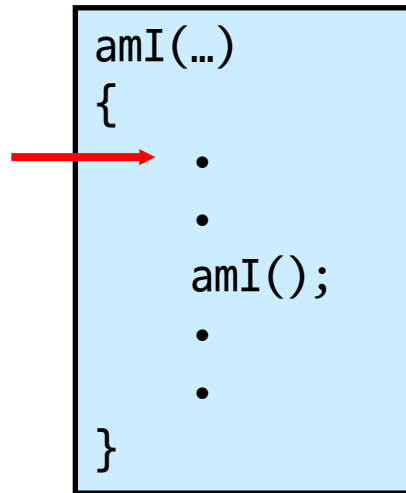
Stack Frames (4)



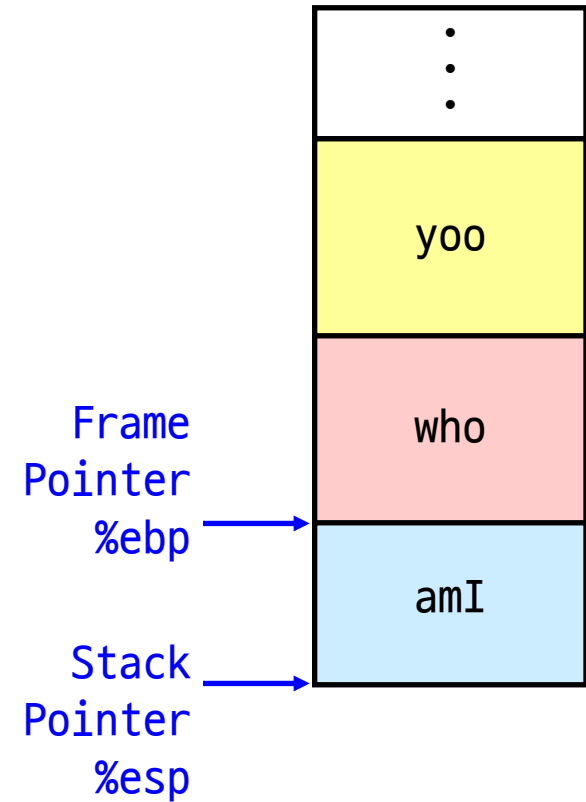
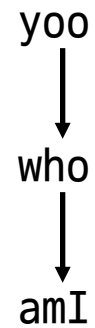
Call Chain



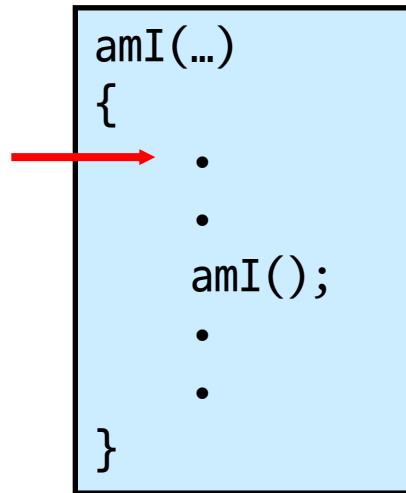
Stack Frames (5)



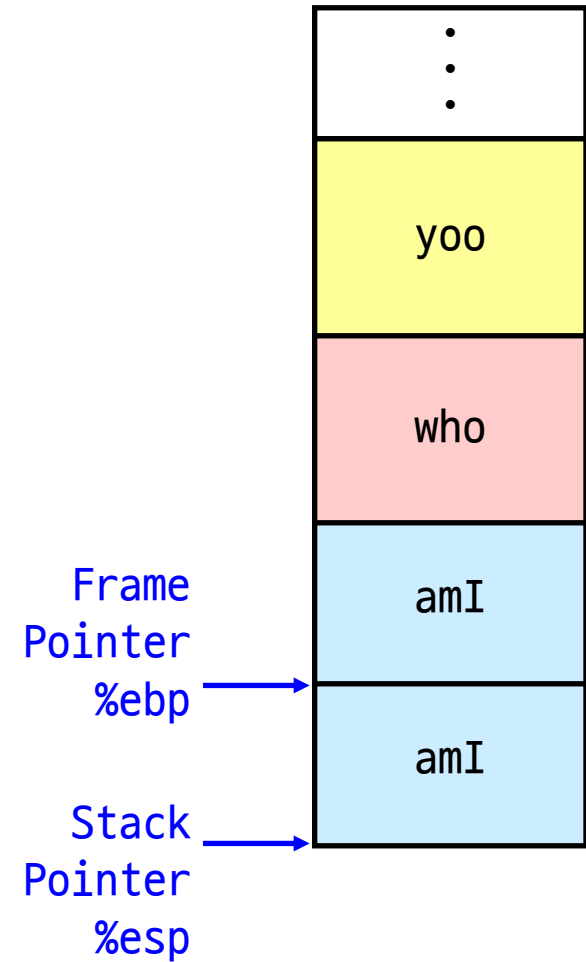
Call Chain



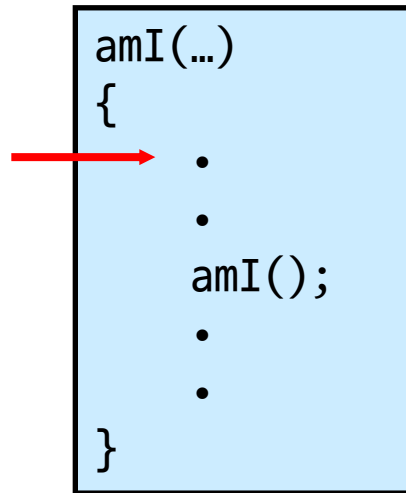
Stack Frames (6)



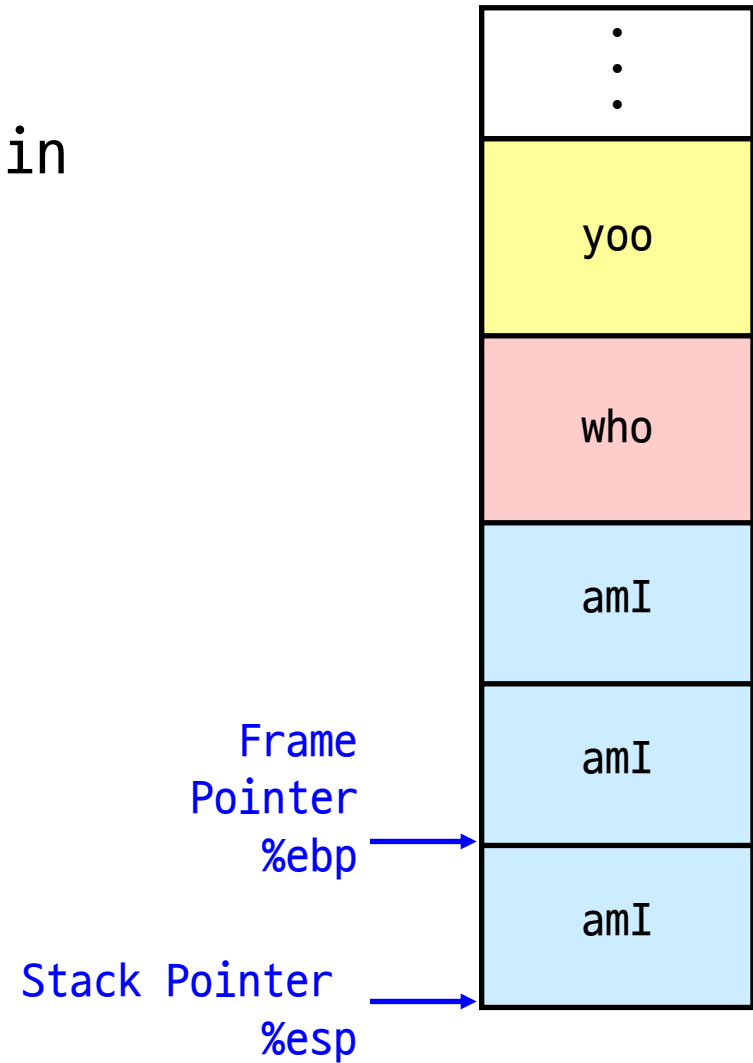
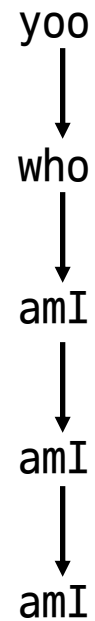
Call Chain



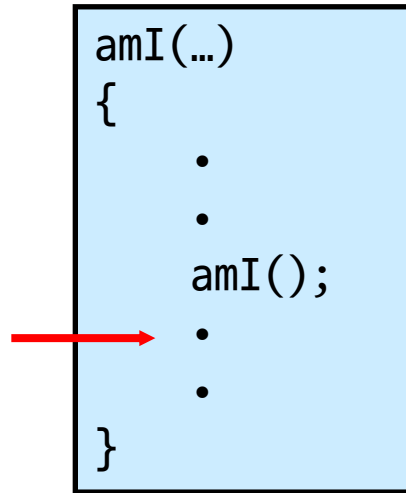
Stack Frames (7)



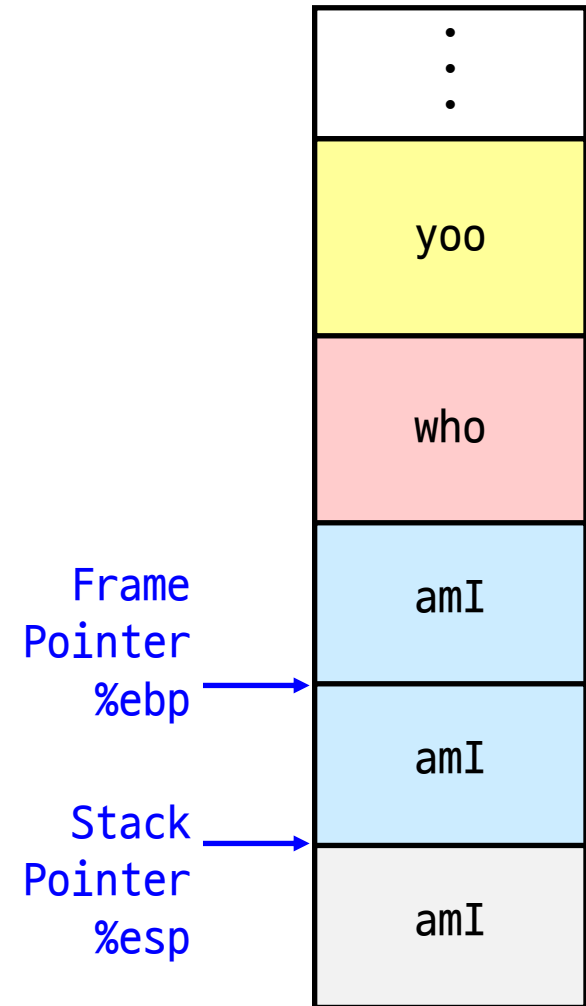
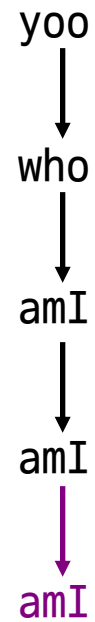
Call Chain



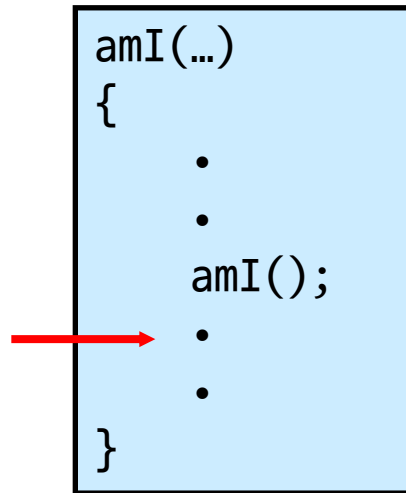
Stack Frames (8)



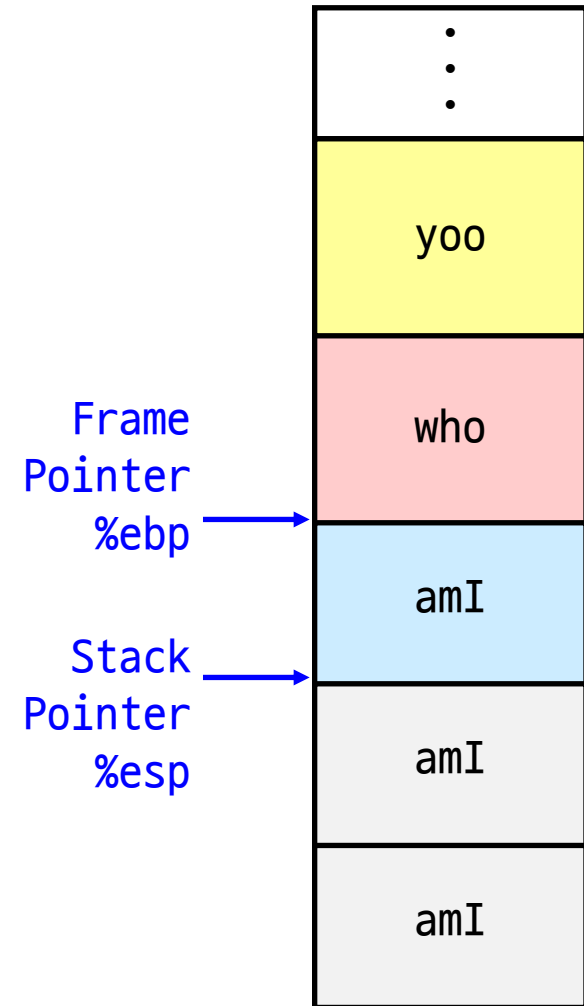
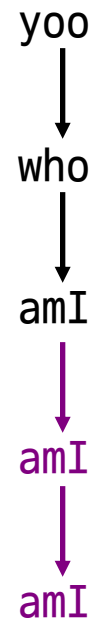
Call Chain



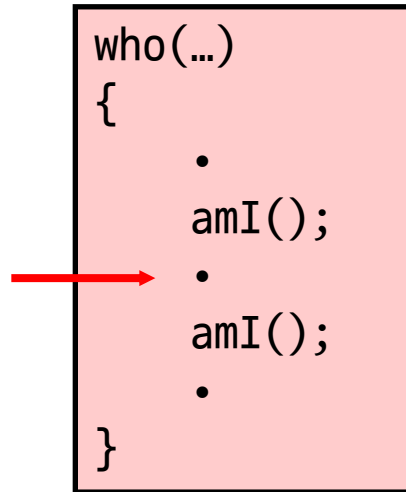
Stack Frames (9)



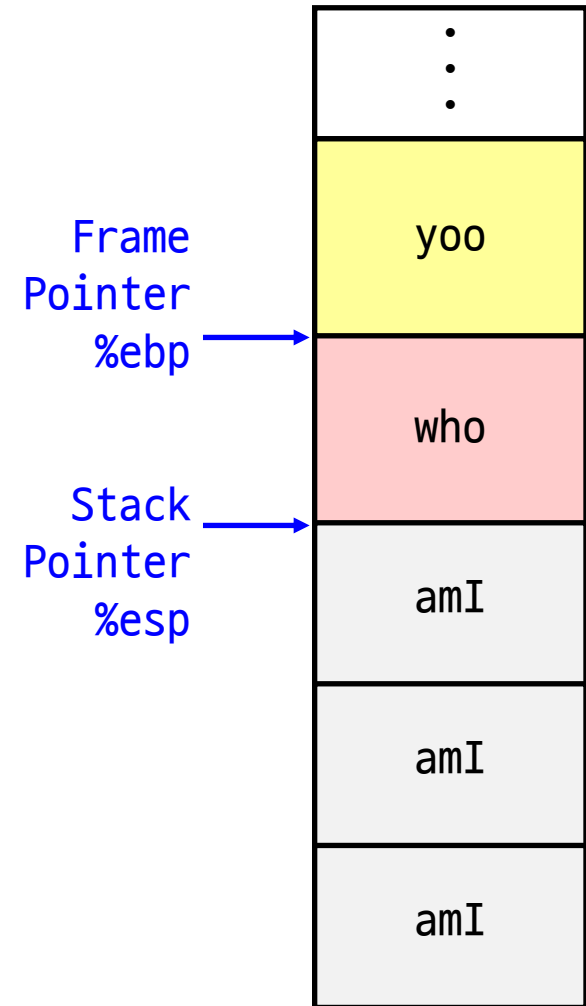
Call Chain



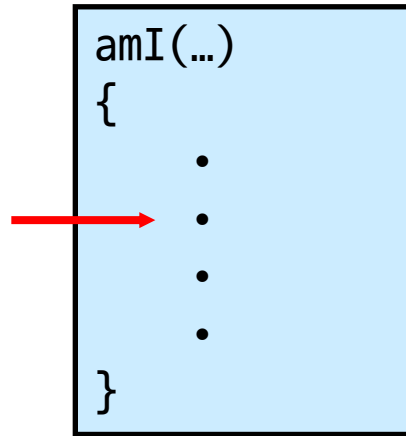
Stack Frames (10)



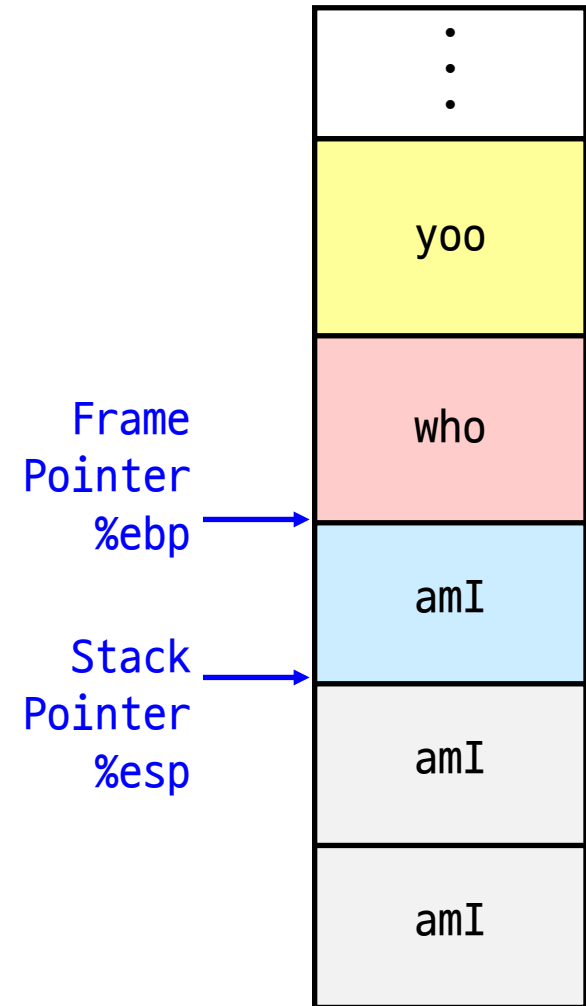
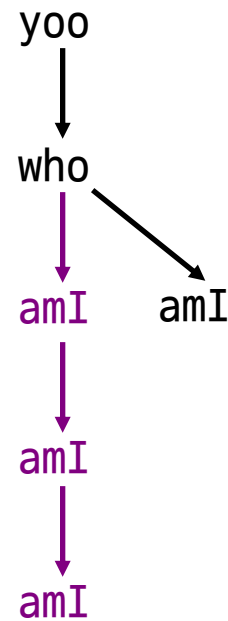
Call Chain



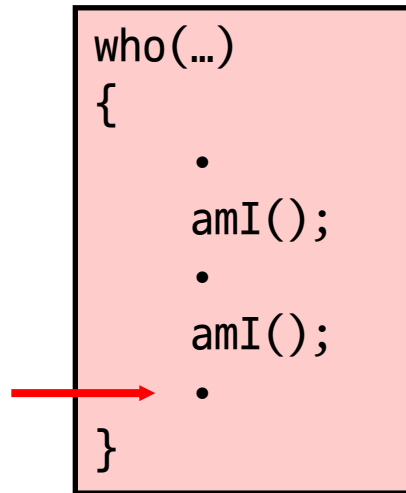
Stack Frames (11)



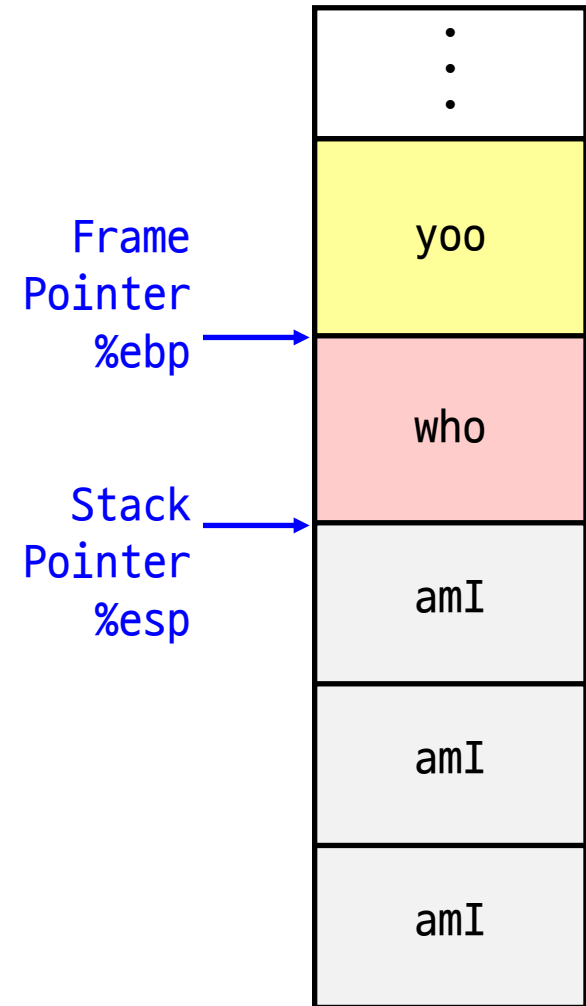
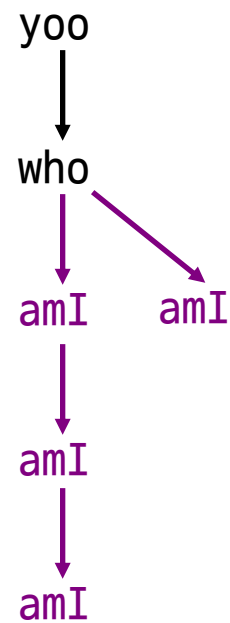
Call Chain



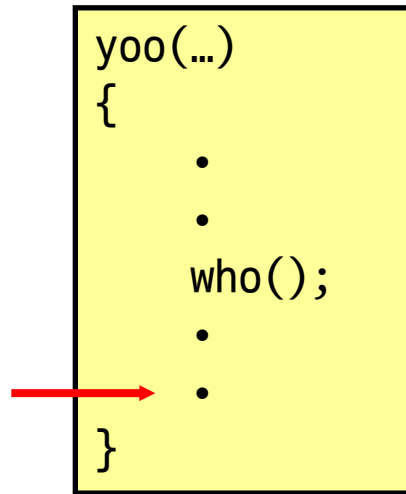
Stack Frames (12)



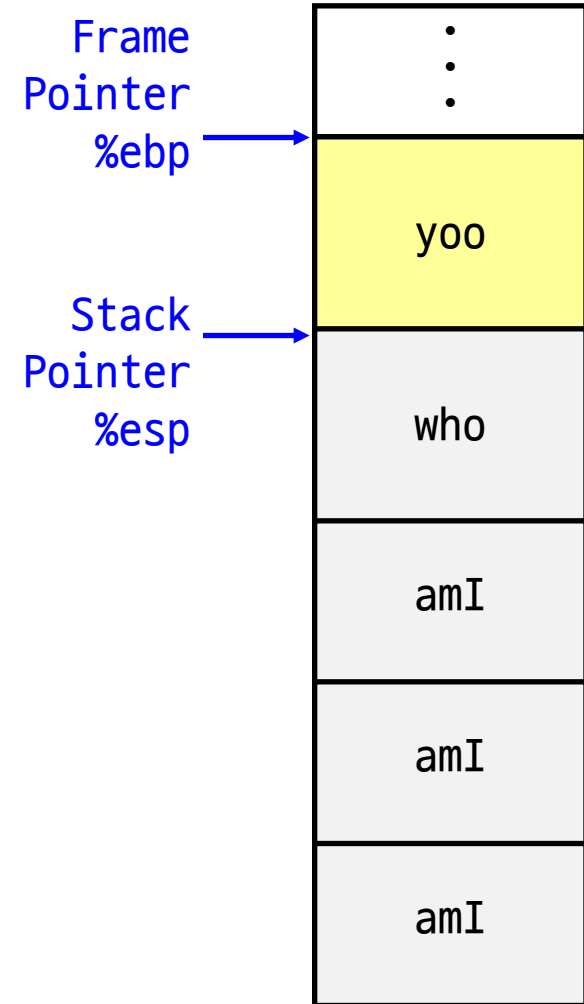
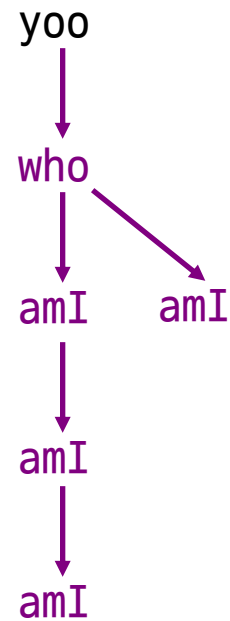
Call Chain



Stack Frames (13)



Call Chain



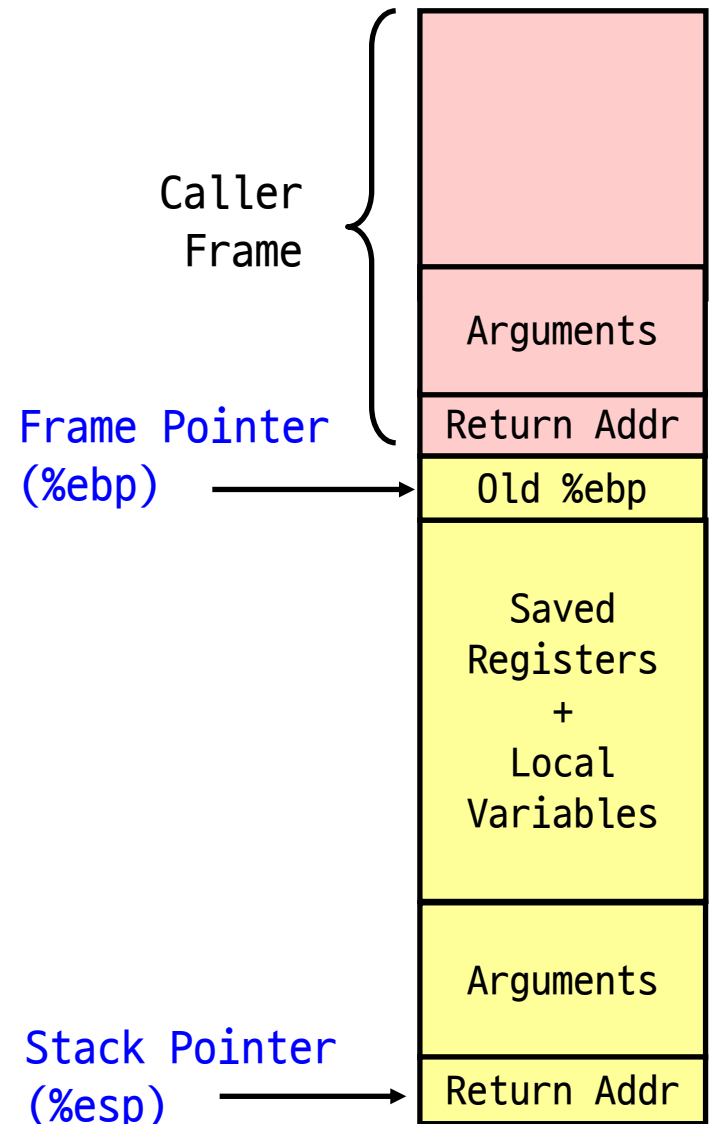
IA-32/Linux Stack Frame

Caller stack frame

- Arguments to call
- Return address
 - Pushed by call instruction

Current (Callee) stack frame

- Old frame pointer
- Saved register context
 - `%ebx, %esi, %edi` (callee-save)
- Local variables
 - If can't keep in registers
- If call another function,
 - Parameters for function about to call (Argument)
 - Return address



Revisiting swap (1)

```
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:

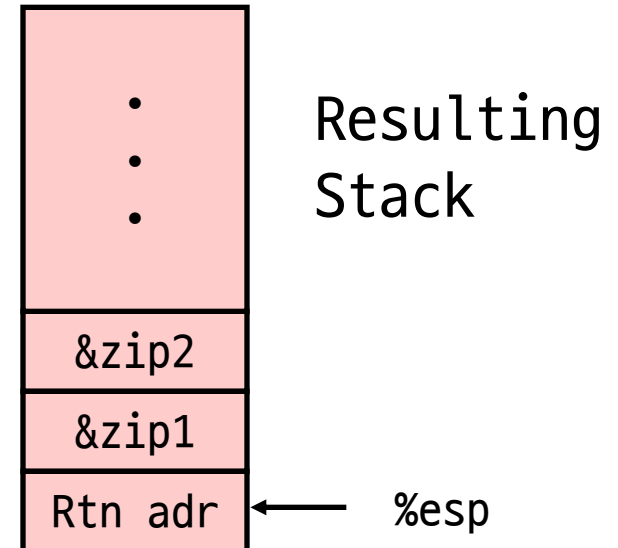
• • •

pushl \$zip2 # Global Var

pushl \$zip1 # Global Var

call swap

• • •



Revisiting swap (2)

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

swap:

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

} Setup

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

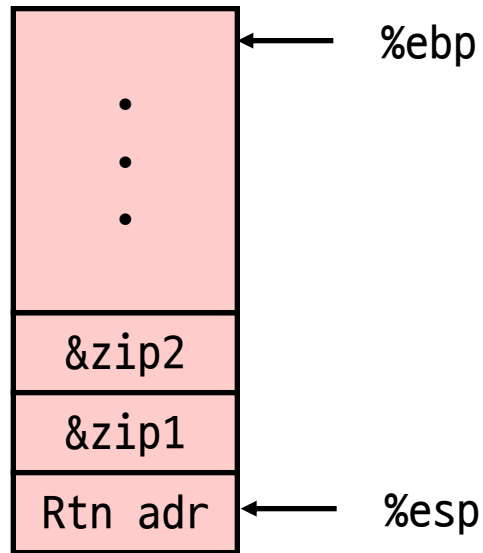
} Body

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

} Finish

Swap Setup (1)

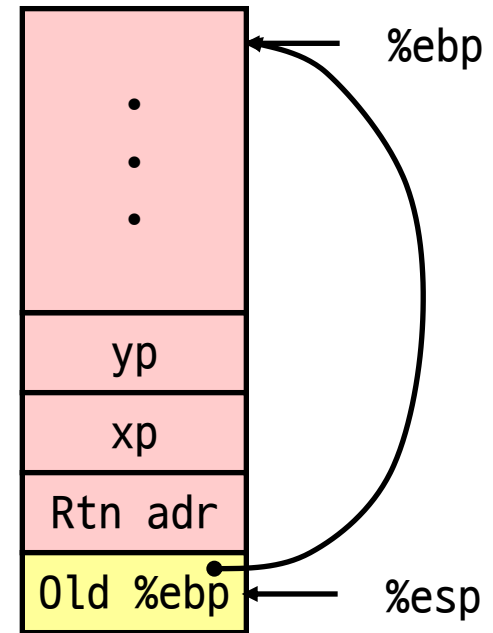
Entering
Stack



swap:

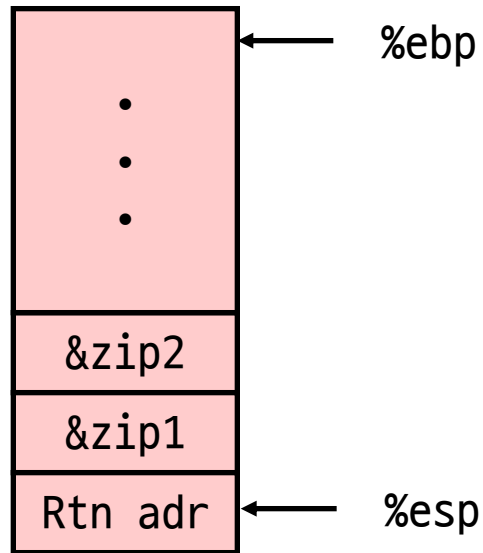
```
pushl %ebp  
movl %esp,%ebp  
pushl %ebx
```

Resulting
Stack



Swap Setup (2)

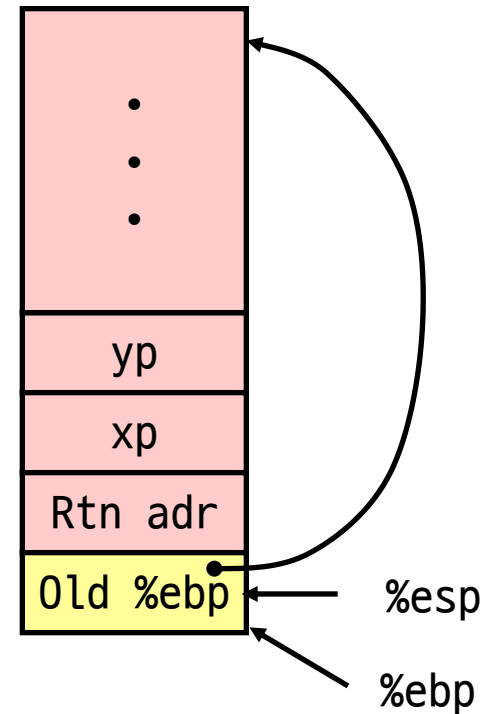
Entering
Stack



swap:

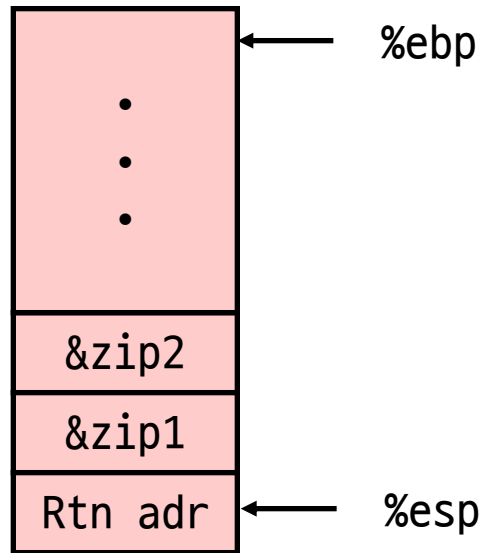
```
pushl %ebp
movl %esp,%ebp
pushl %ebx
```

Resulting
Stack



Swap Setup (3)

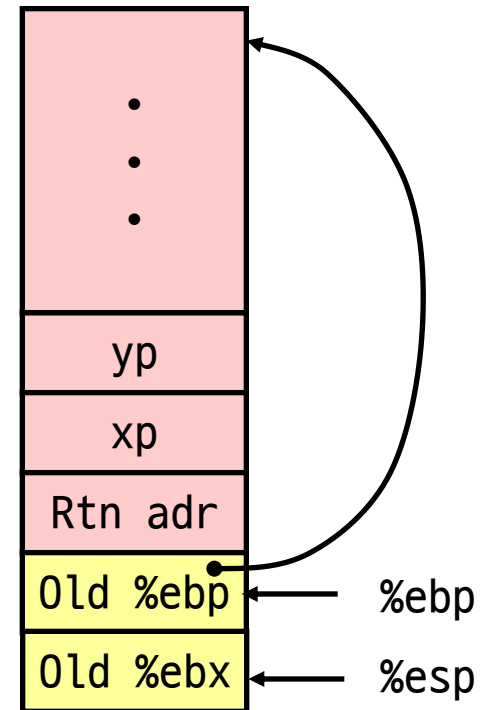
Entering
Stack



swap:

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

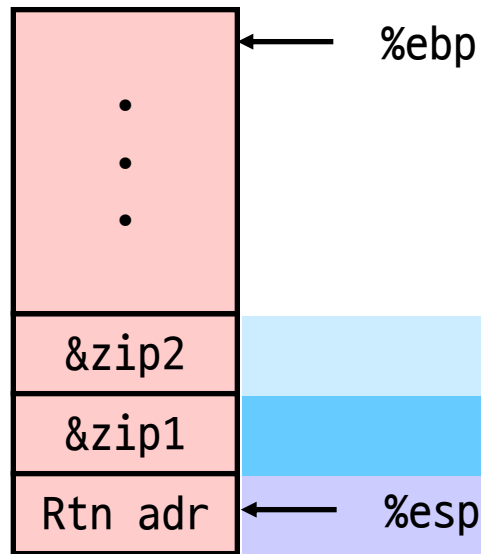
Resulting
Stack



→ Why ?

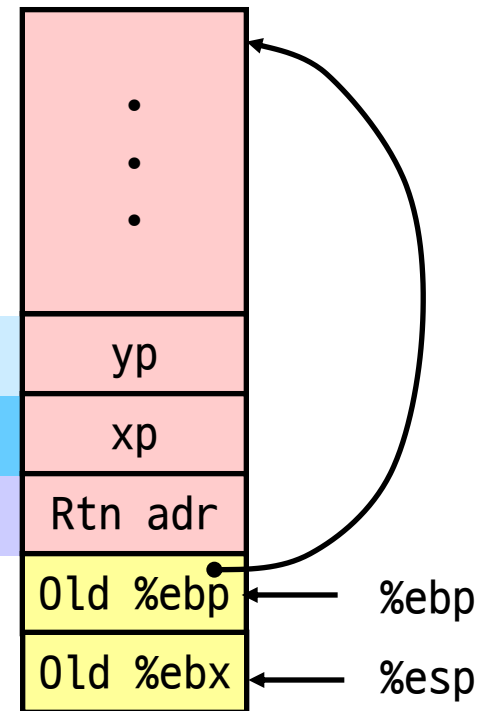
Effect of swap Setup

Entering
Stack



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

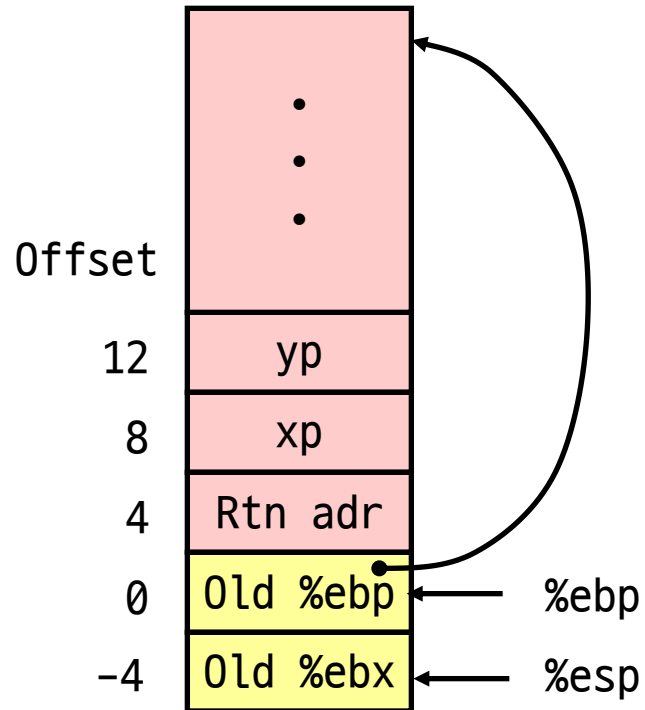
Resulting
Stack



} Body

swap Finish (1)

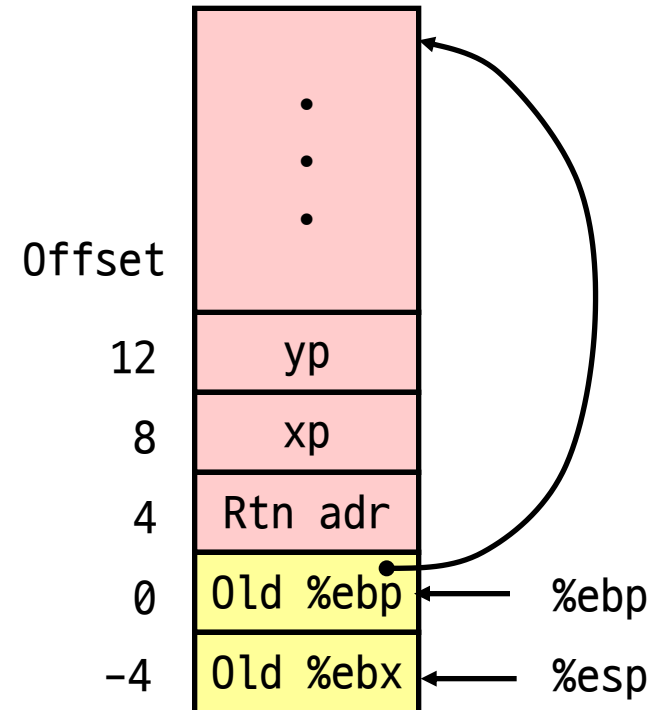
swap's Stack



Observation

- Saved & restored register %ebx

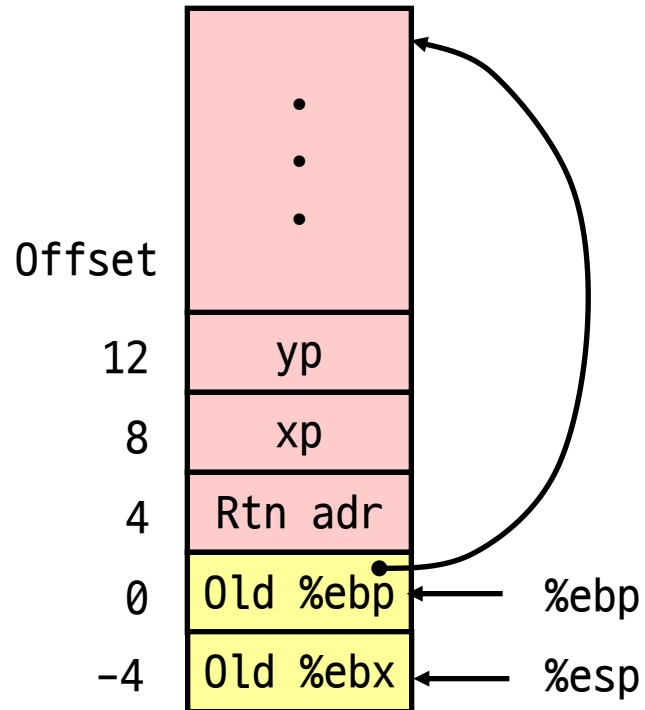
Exiting Stack



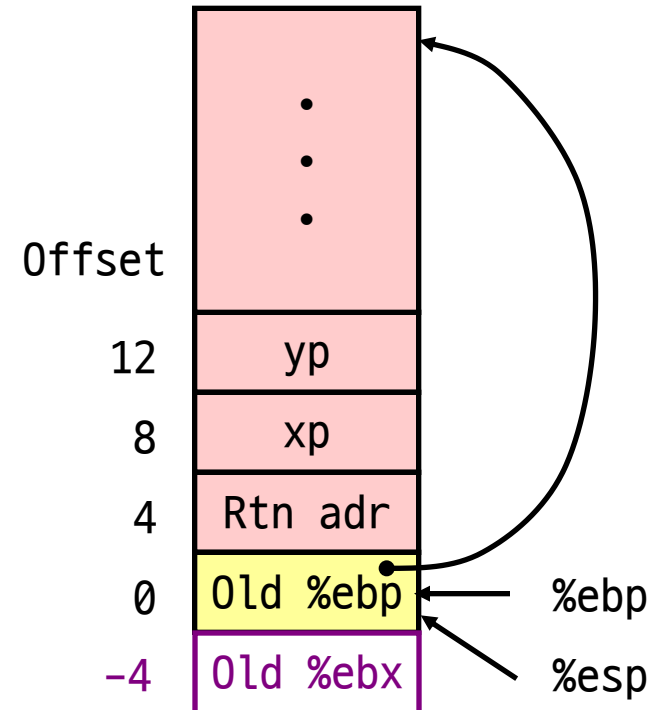
```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

swap Finish (2)

swap's Stack

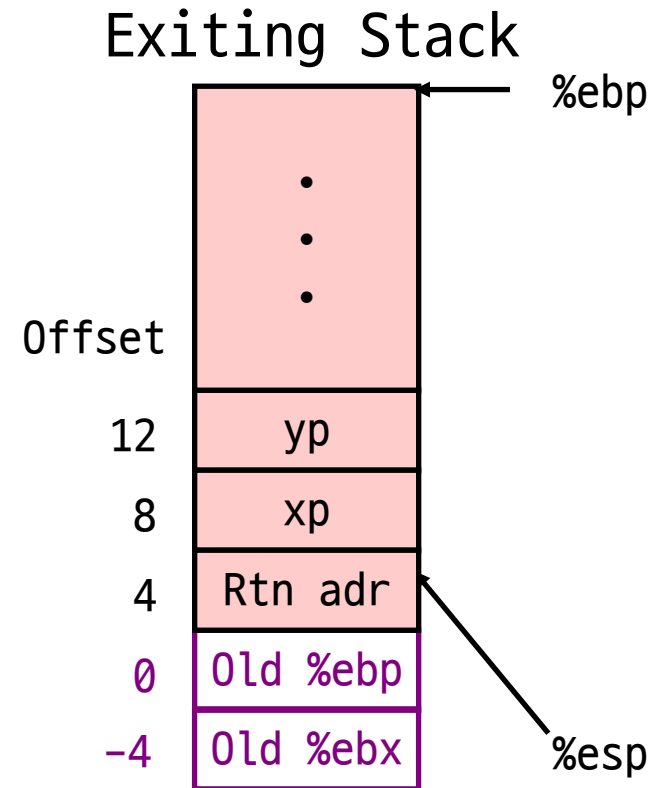
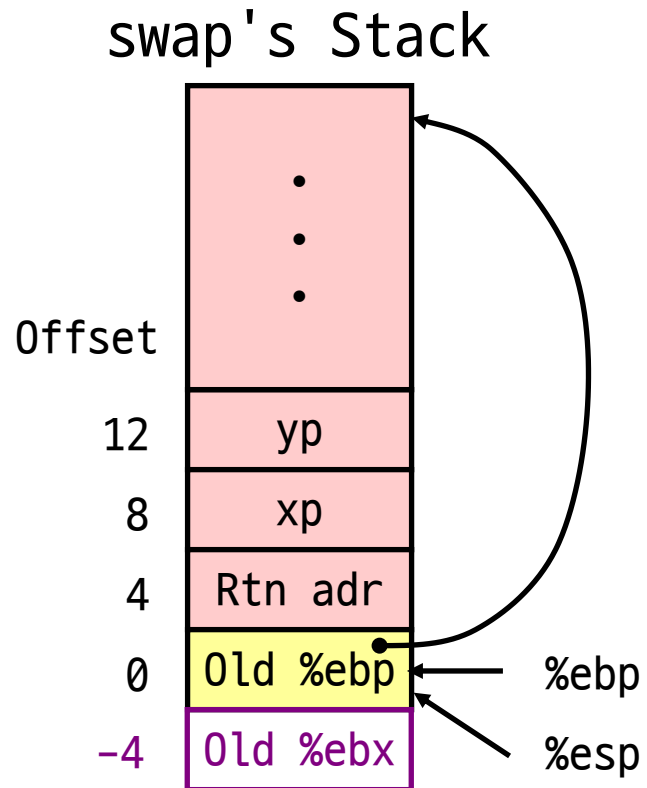


Exiting Stack



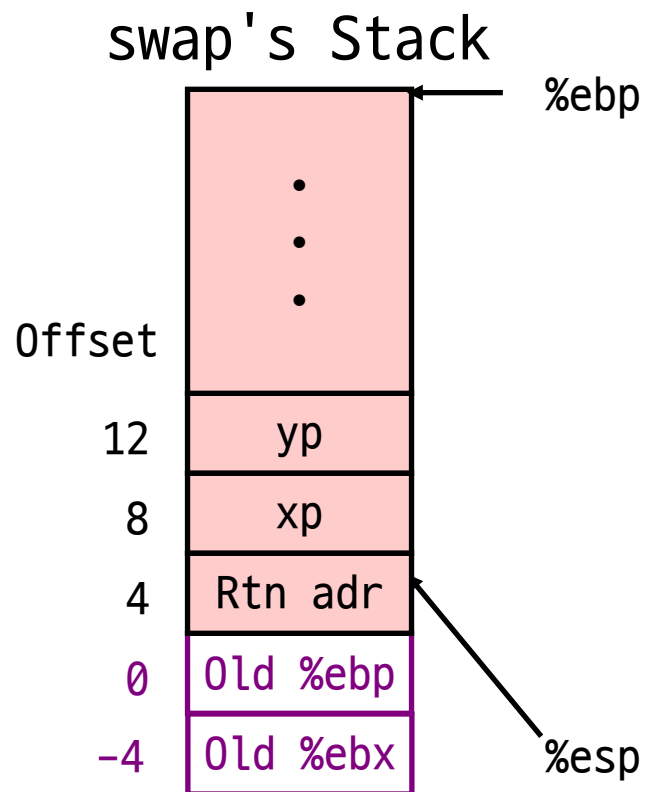
```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```


swap Finish (3)



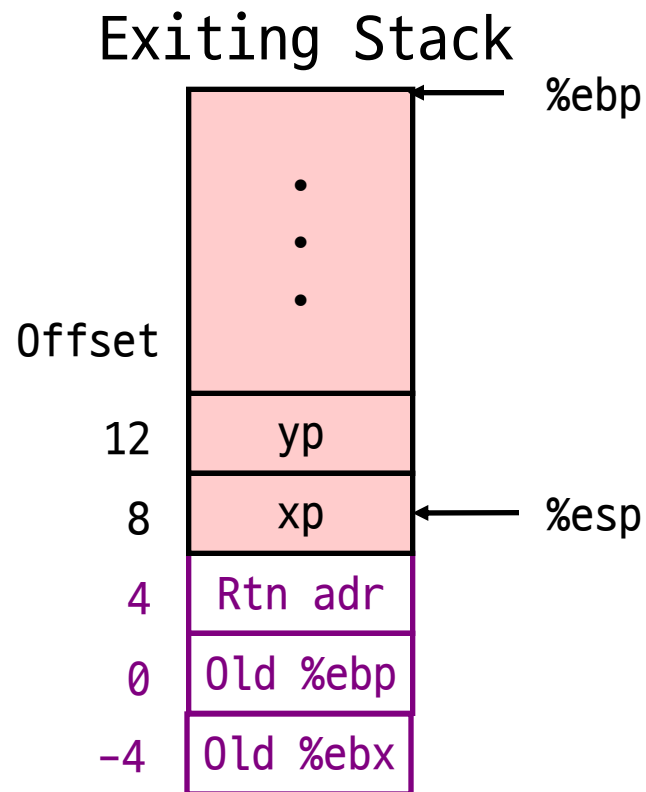
```
movl -4(%ebp),%ebx  
movl %ebp,%esp  
popl %ebp  
ret
```

swap Finish (4)



Observation

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



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

Revisiting arith

```
int arith (int x, int y, int z)
{
    int t1 = x + y;
    int t2 = z + t1;
    int t3 = x + 4;
    int t4 = y * 48;
    int t5 = t3 + t4;
    int rval = t2 * t5;

    return rval;
}
```

arith:

```
    pushl %ebp
    movl %esp,%ebp
```

} Set Up

```
    movl 8(%ebp),%eax
    movl 12(%ebp),%edx
    leal (%edx,%eax),%ecx
    leal (%edx,%edx,2),%edx
    sall $4,%edx
    addl 16(%ebp),%ecx
    leal 4(%edx,%eax),%eax
    imull %ecx,%eax
```

} Body

```
    movl %ebp,%esp
    popl %ebp
    ret
```

} Finish

No use of %ebx, %esi, %edi
(callee-save)

Revisiting max

```
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

_max:

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

} Set Up

```
movl 8(%ebp),%ebx
movl 12(%ebp),%eax
cmpl %eax,%ebx
jle L9
movl %ebx,%eax
```

} Body

L9:

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

} Finish

Revisiting Do-While

Goto Version

```
int fact_goto (int x)
{
    int result = 1;
loop:
    result *= x;
    x = x - 1;
    if (x > 1)
        goto loop;
    return result;
}
```

Assembly

```
_fact_goto:
    pushl %ebp                # Setup
    movl %esp,%ebp           # Setup

    movl $1,%eax              # eax = 1
    movl 8(%ebp),%edx          # edx = x
L11:
    imull %edx,%eax            # result *= x
    decl %edx                  # x--
    cmpl $1,%edx               # Compare x : 1
    jg L11                     # if > goto loop

    movl %ebp,%esp            # Finish
    popl %ebp                  # Finish
    ret                        # Finish
```

No use of %ebx, %esi, %edi
(callee-save)

Recursive Factorial: rfact

Registers

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

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

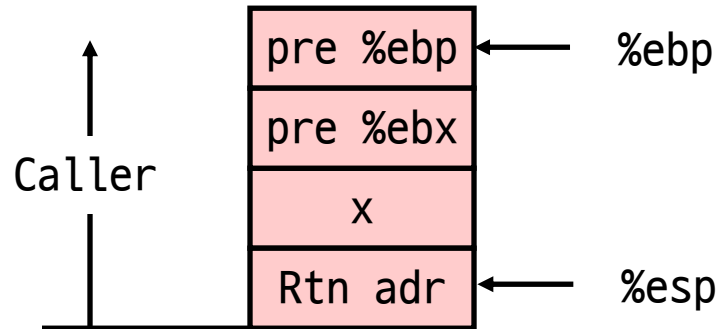
```
rfact:
    pushl %ebp
    movl %esp, %ebp
    pushl %ebx

    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
```

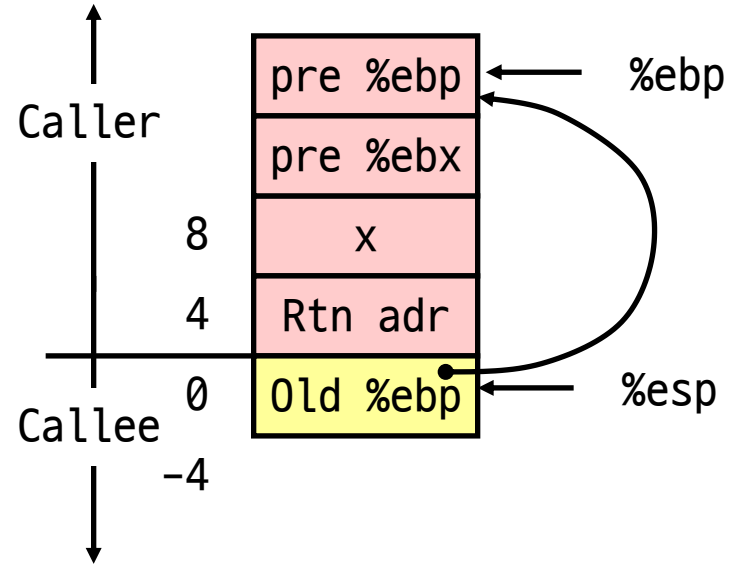
rfact Stack Setup

Entering Stack



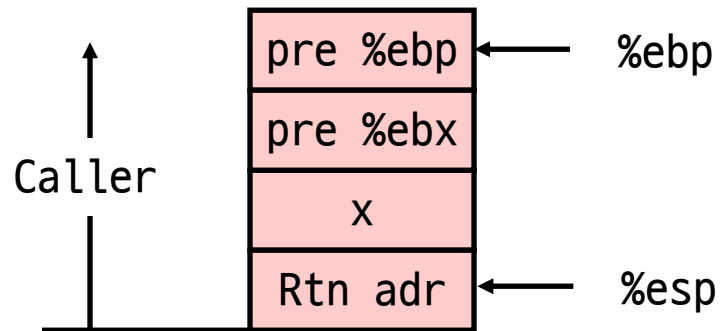
rfact:

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



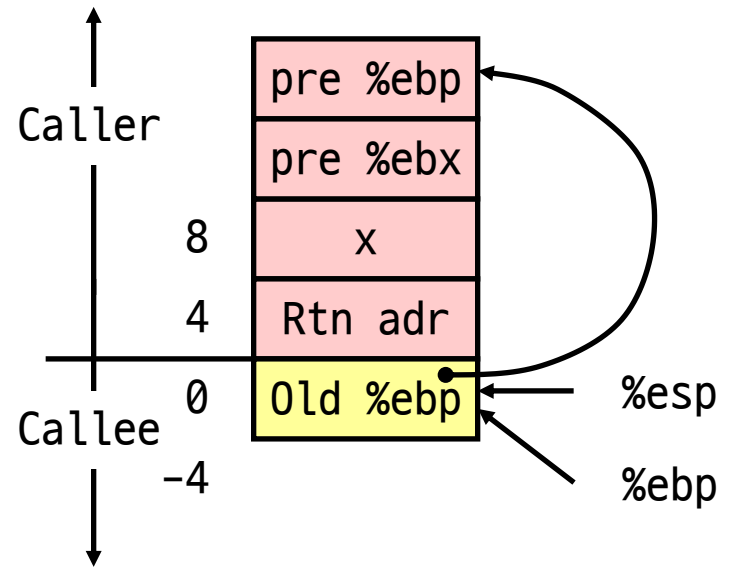
rfact Stack Setup

Entering Stack



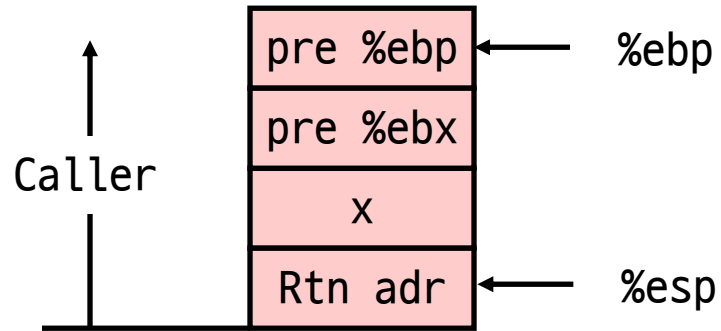
rfact:

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



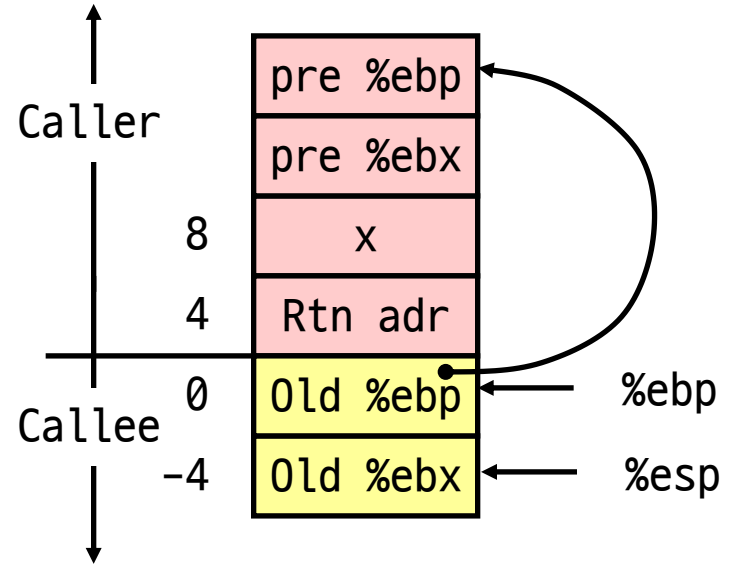
rfact Stack Setup

Entering Stack



rfact:

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



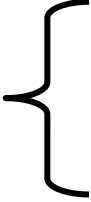
rfact Body

Registers

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

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

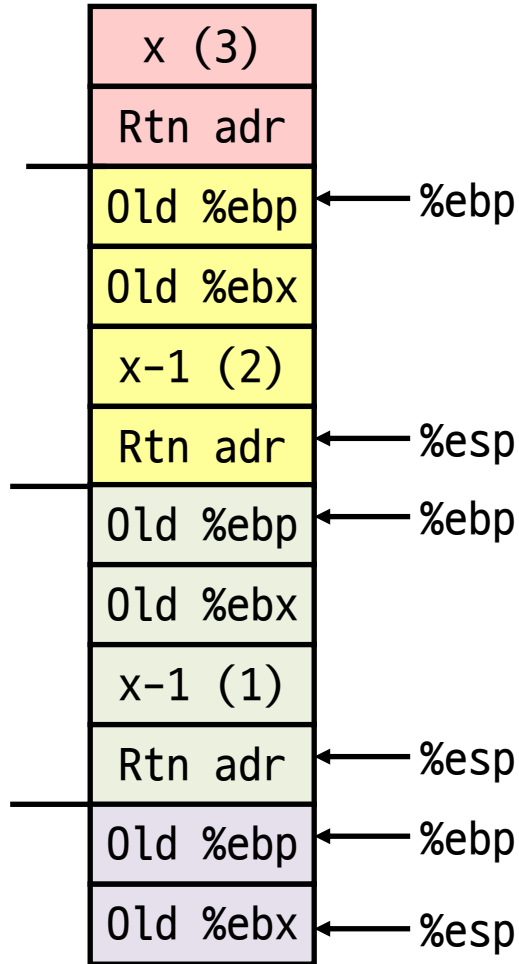
Recursion



```
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:
```

rfact Body

rfact(3)



```
movl 8(%ebp), %ebx      # ebx = x (3)
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:
```

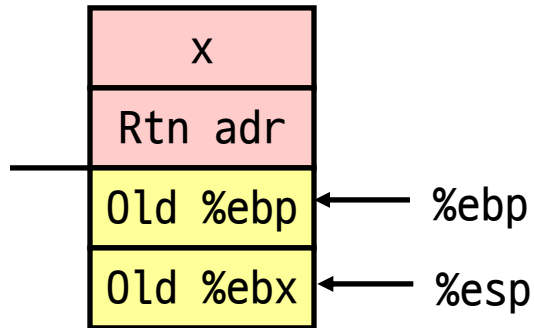
```
movl 8(%ebp), %ebx      # ebx = x (2)
cmpl $1,%ebx           # Compare x : 1
jle .L78               # If <= goto Term
leal -1(%ebx), %eax     # eax = x-1
pushl %eax             # Push x-1
call rfact              # rfact(x-1)
imull %ebx, %eax        # rval * x
jmp .L79               # Goto done
.L78:                  # Term:
movl $1, %eax          # return val = 1
.L79:                  # Done:
```

```
movl 8(%ebp), %ebx      # ebx = x (1)
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:
```

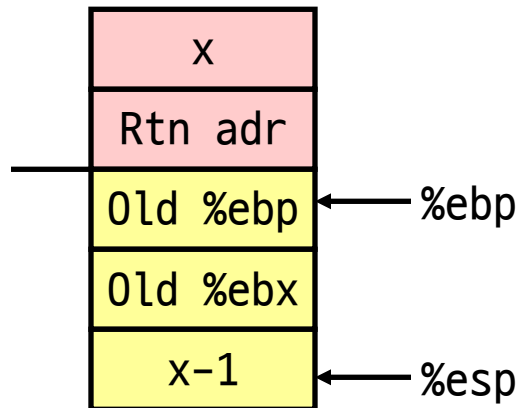
rfact Recursion

```
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)
... ..
```

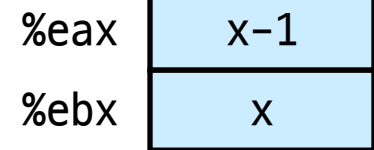
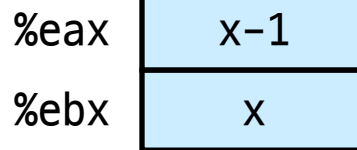
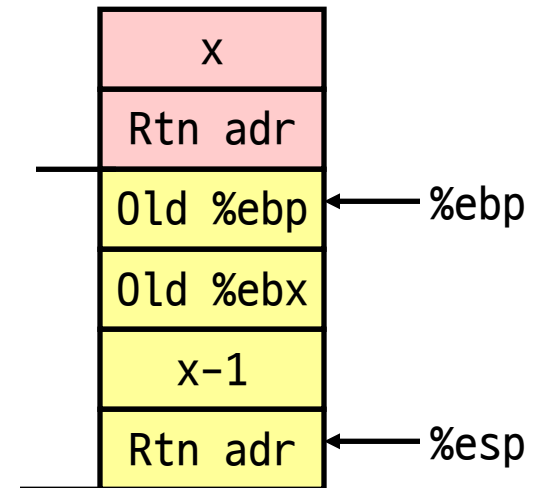
leal -1(%ebx), %eax



pushl %eax



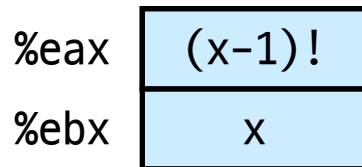
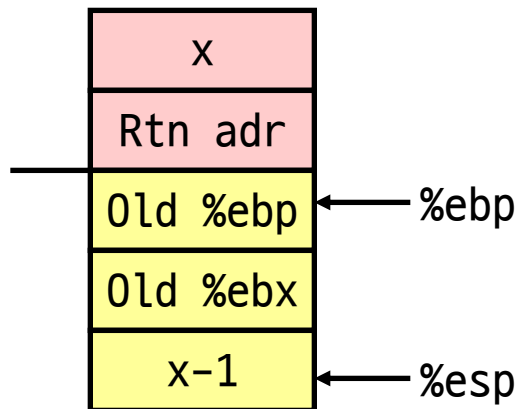
call rfact



rfact Result

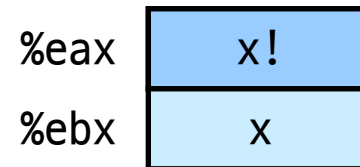
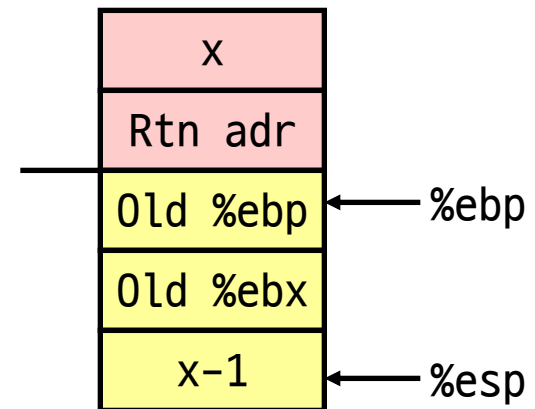
```
... ..  
call rfact                # rfact(x-1)  
imull %ebx,%eax           # rval * x  
jmp .L79                  # Goto done  
.L78:                     # Term:  
    movl $1,%eax          # return val = 1  
.L79:                     # Done:
```

Return from Call



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

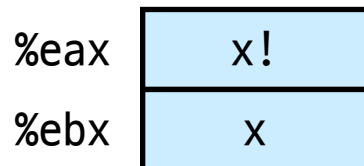
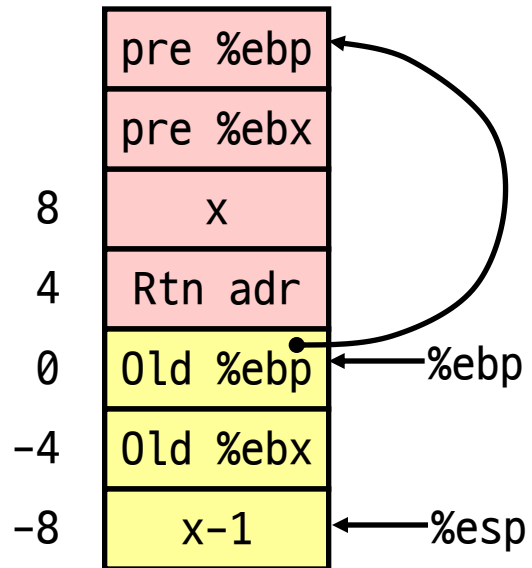
imull %ebx, %eax



rfact Completion

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

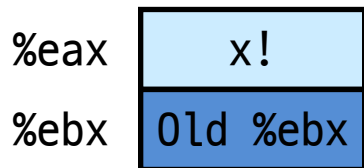
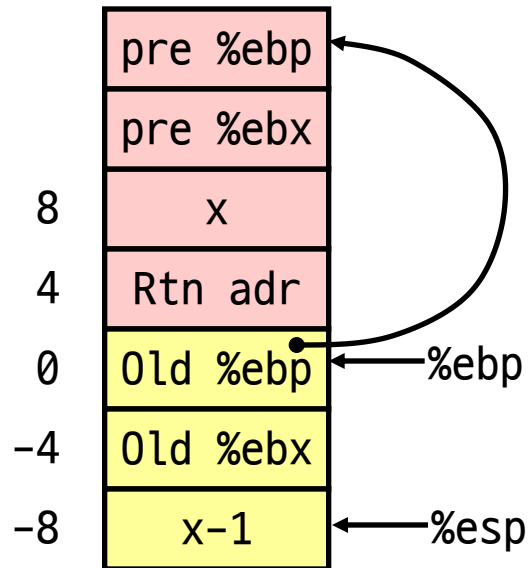
```
movl -4(%ebp), %ebx
```



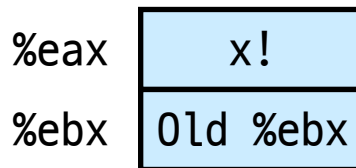
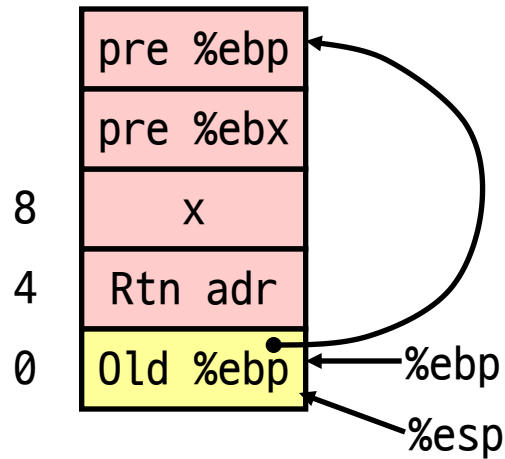
rfact Completion

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

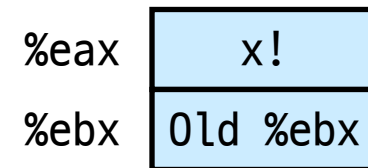
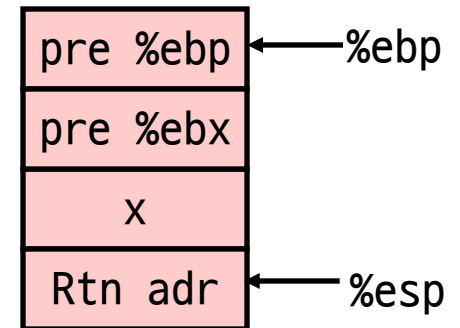
movl -4(%ebp), %ebx



movl %ebp, %esp



popl %ebp



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

Procedures = Instructions + Conventions

- Call / Ret instructions
- Register usage conventions
 - Caller save (%eax, %ecx, %edx) / Callee save (%ebx, %esi, %edi)
 - %ebp and %esp
- Stack frame organization conventions