

EECS08

Instructor: Aleksandar Kuzmanovic

TA: Ionut Trestian

Recitation 4

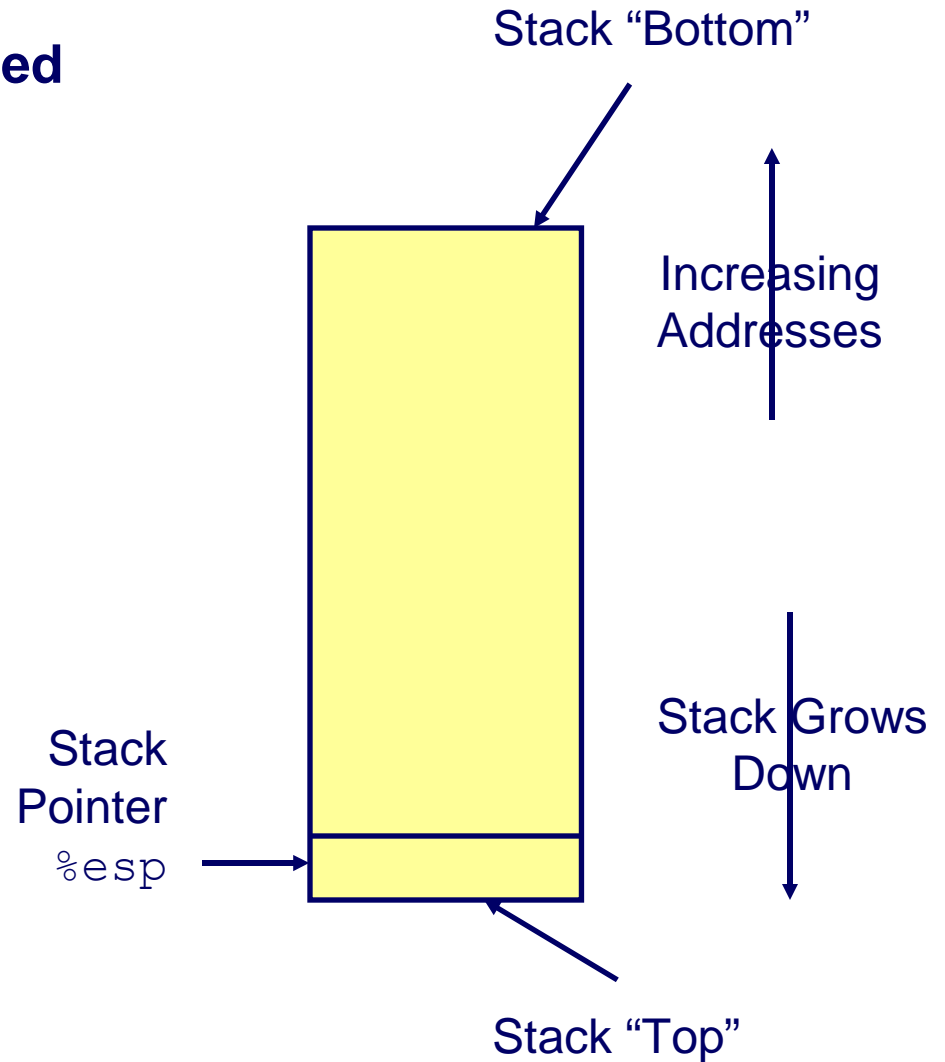
Machine-Level Programming III: Procedures

Topics

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

IA32 Stack

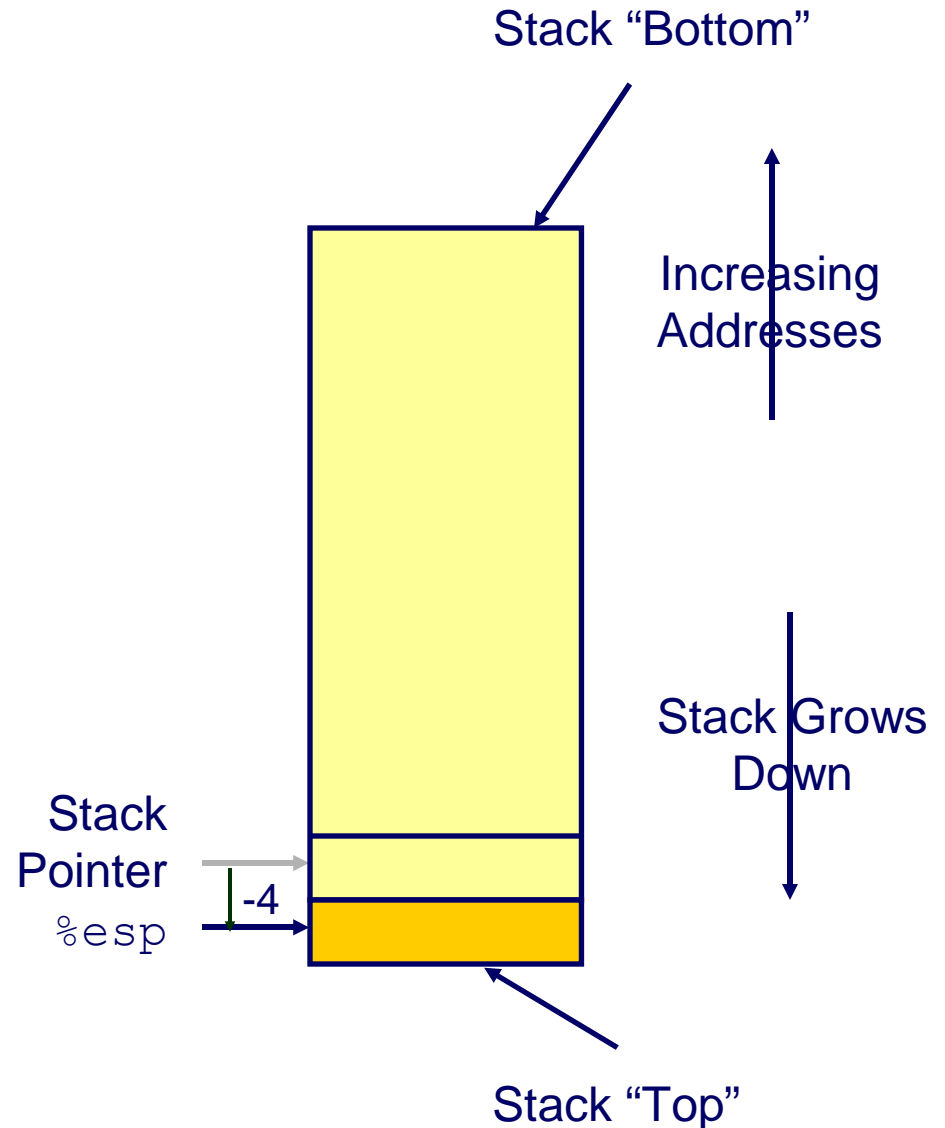
- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
 - address of top element



IA32 Stack Pushing

Pushing

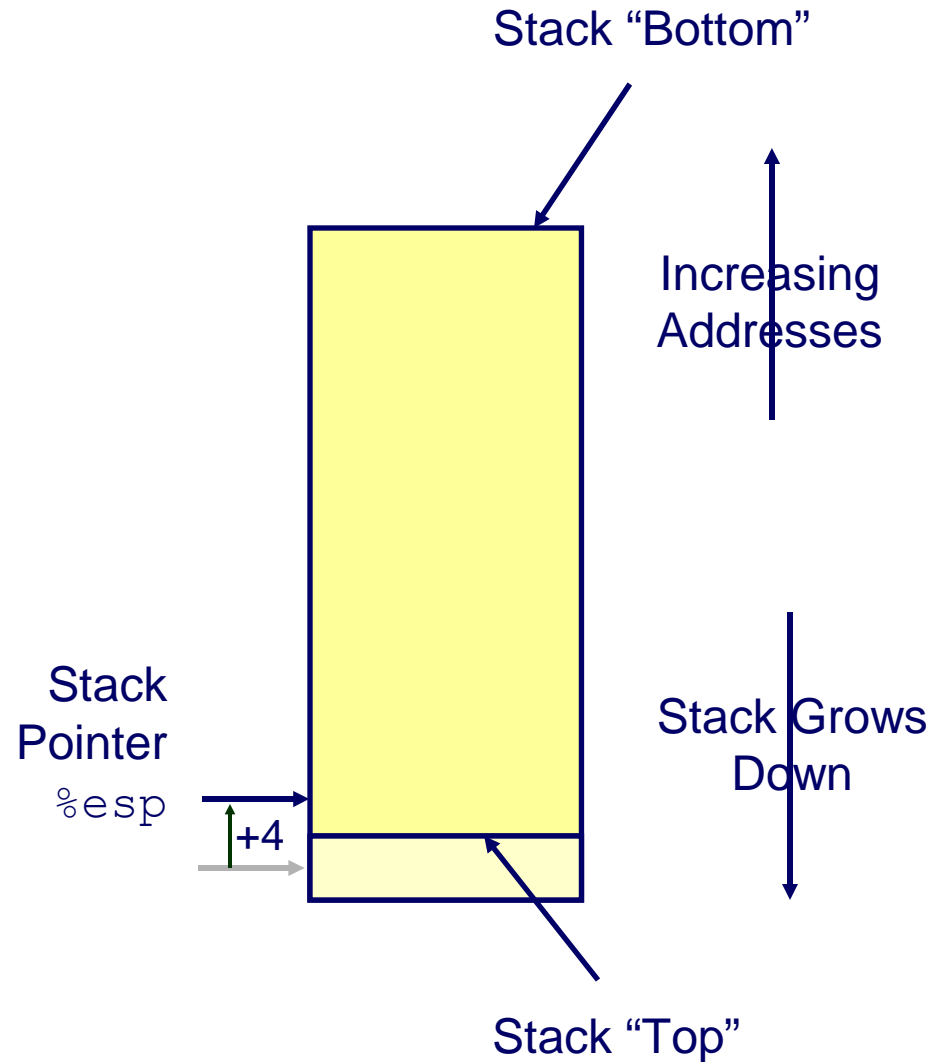
- `pushl Src`
- Fetch operand at `Src`
- Decrement `%esp` by 4
- Write operand at address given by `%esp`



IA32 Stack Popping

Popping

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



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

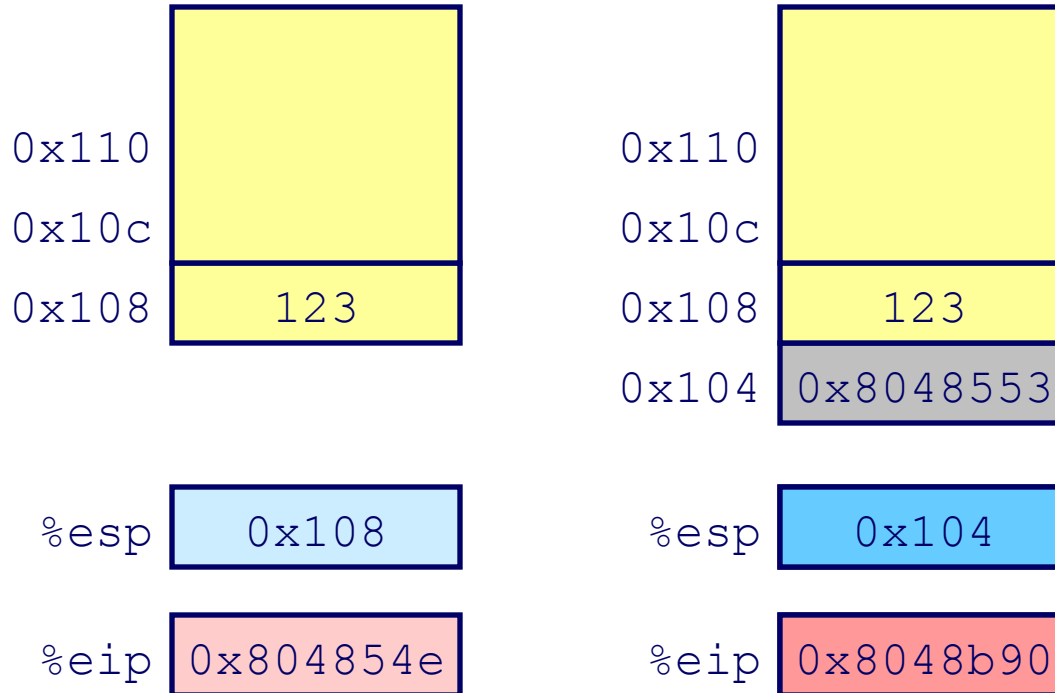
Procedure return:

- `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

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

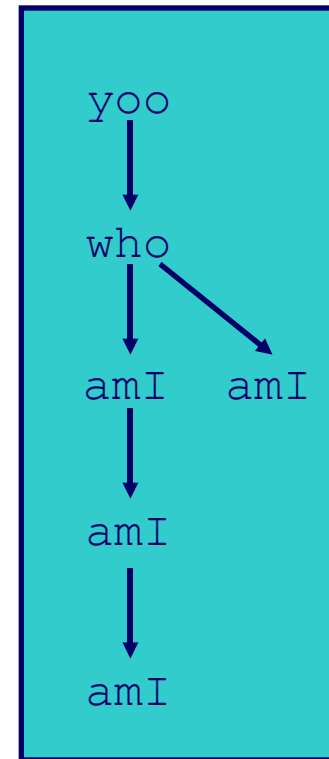
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

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

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

- Procedure `amI` recursive

Call Chain



Stack Frames

Contents

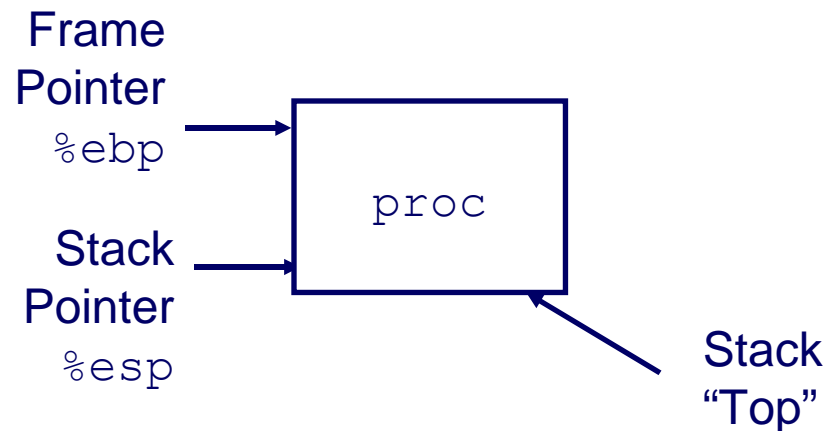
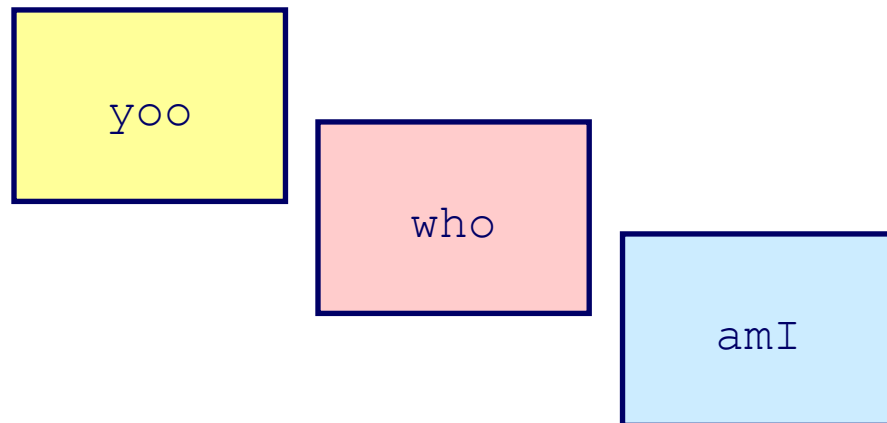
- Local variables
- Return information
- Temporary space

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

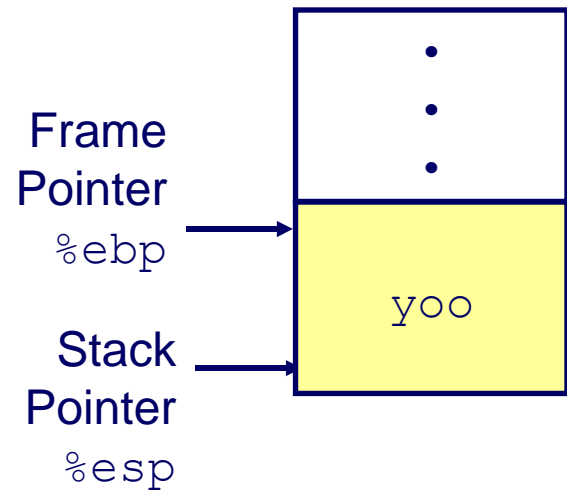


Stack Operation

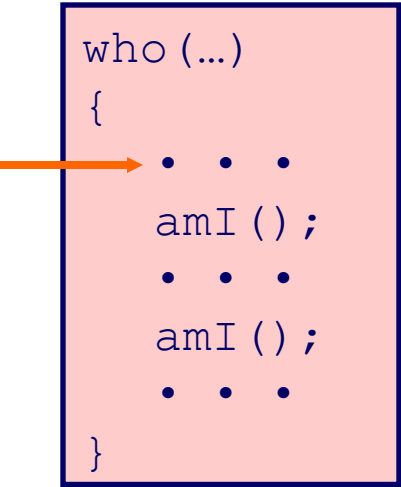
```
yoo (...)  
{  
  .  
  .  
  who ();  
  .  
  .  
}
```

Call Chain

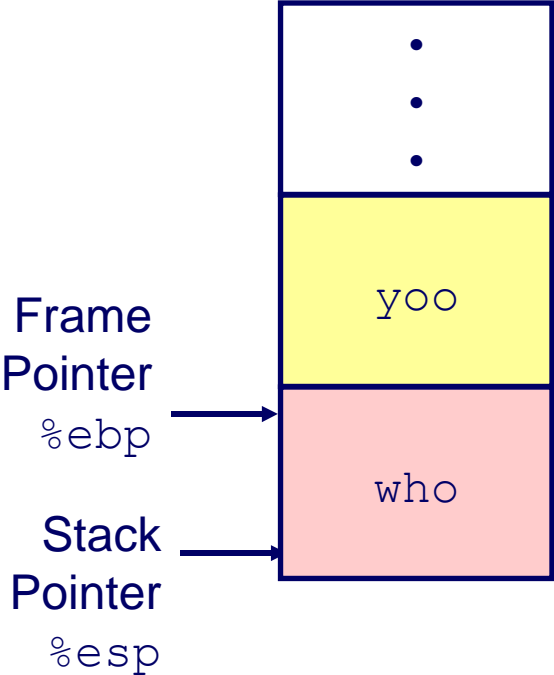
yoo



Stack Operation



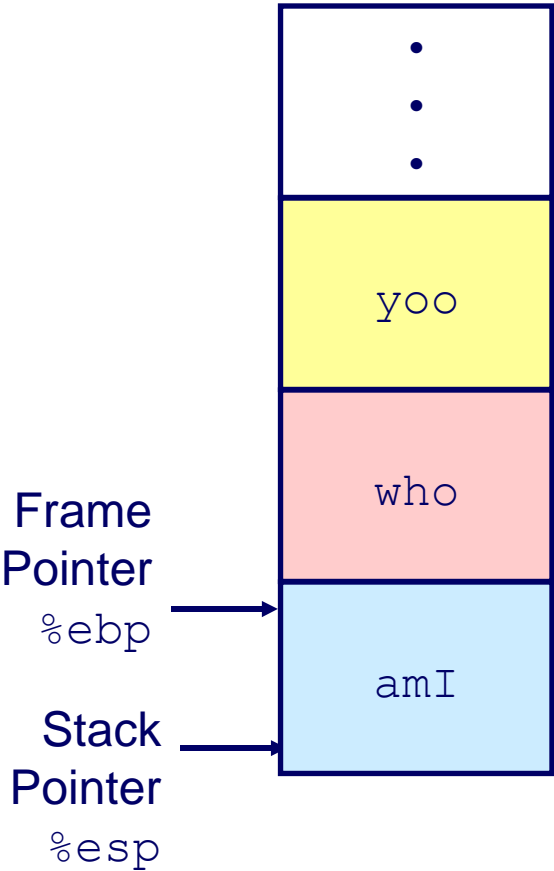
Call Chain



Stack Operation

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

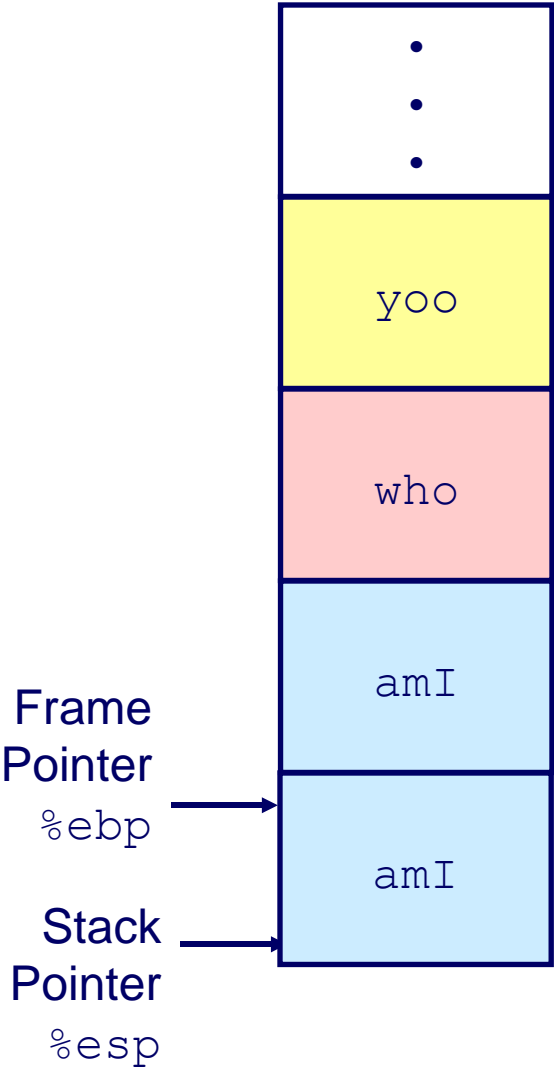
Call Chain



Stack Operation

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

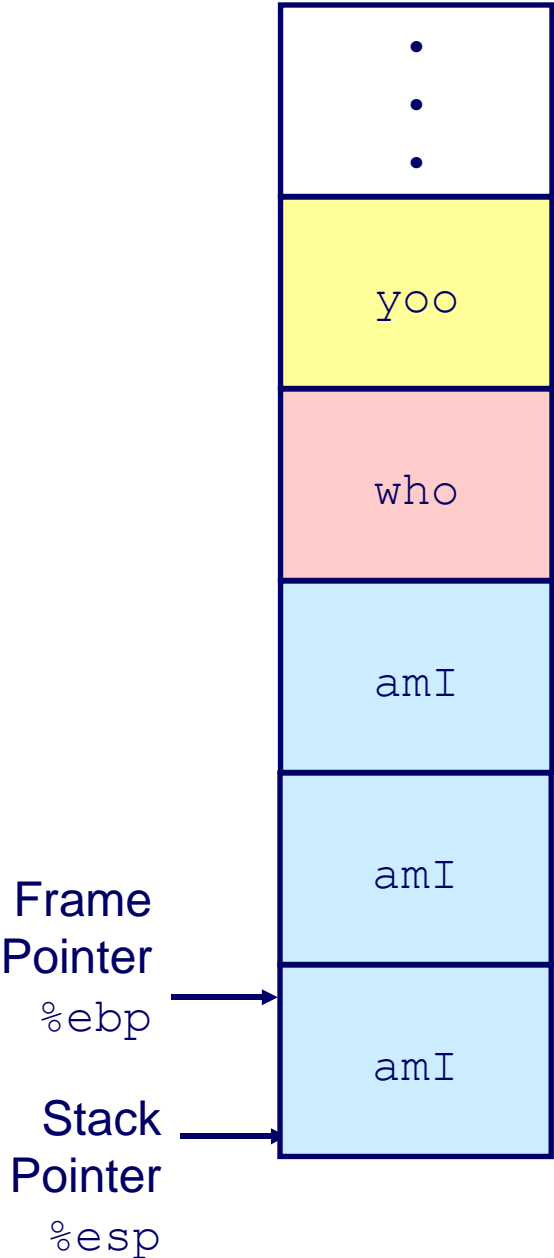
Call Chain



Stack Operation

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

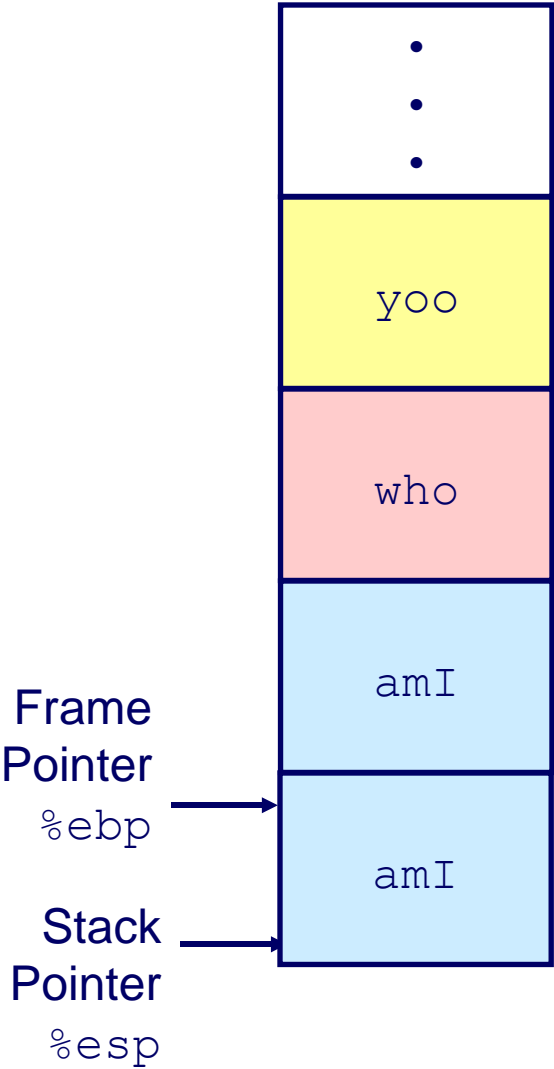
Call Chain



Stack Operation

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

Call Chain

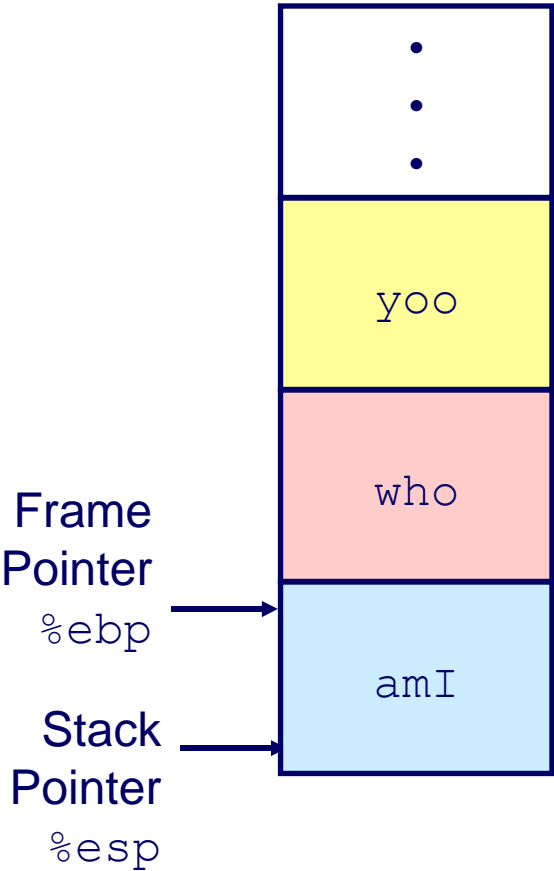


Stack Operation

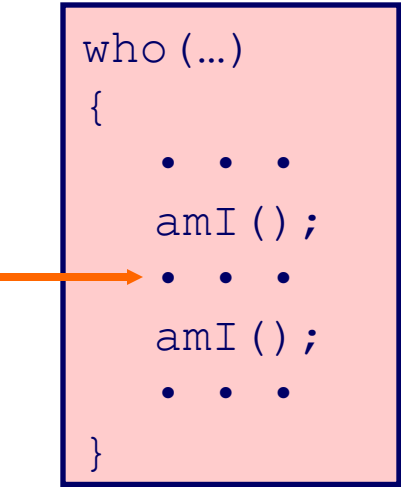
```
amI (...)  
{  
  .  
  .  
  amI ();  
  .  
  .  
}
```



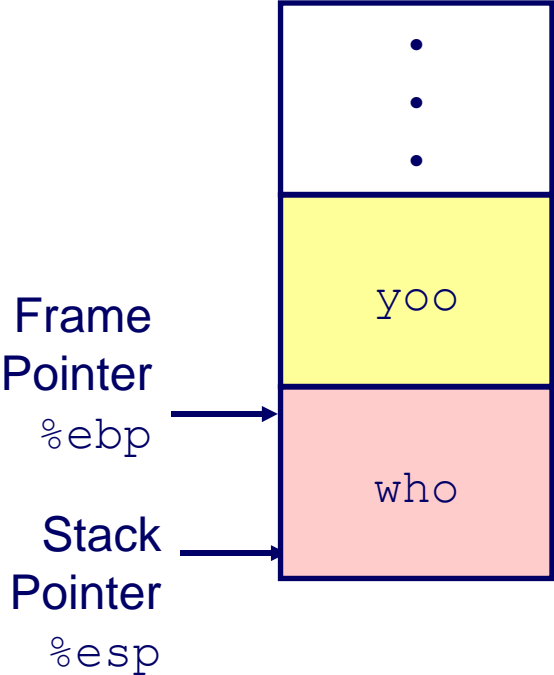
Call Chain



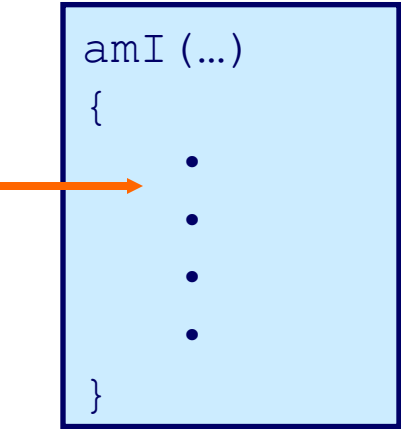
Stack Operation



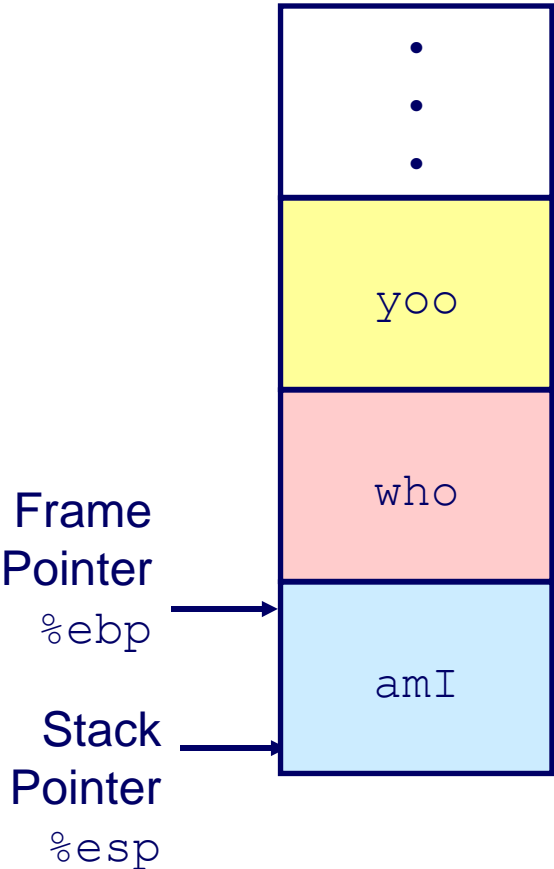
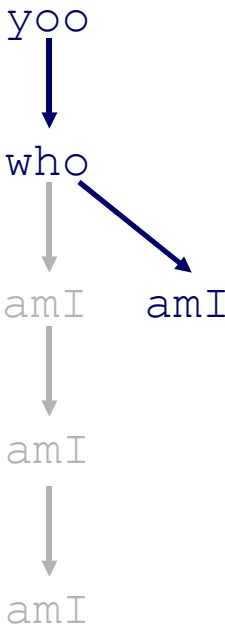
Call Chain



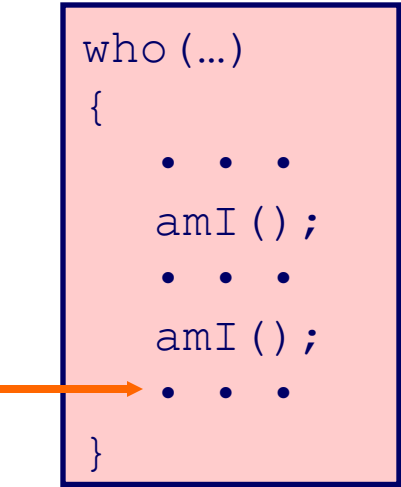
Stack Operation



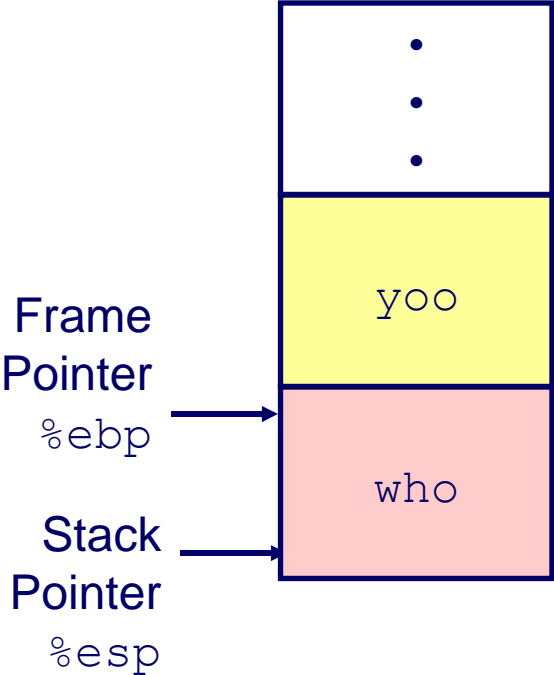
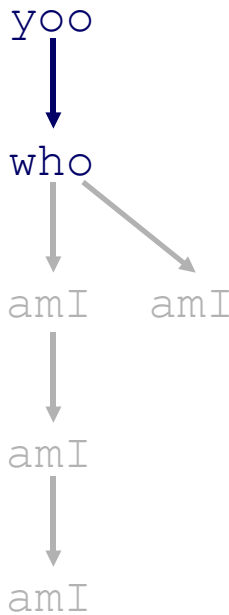
Call Chain



Stack Operation



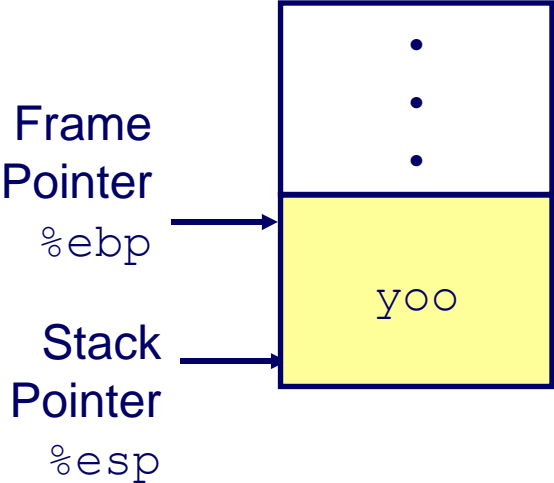
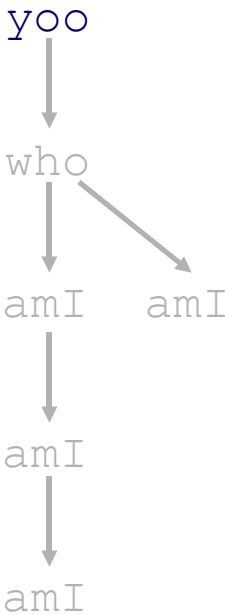
Call Chain



Stack Operation

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

Call Chain



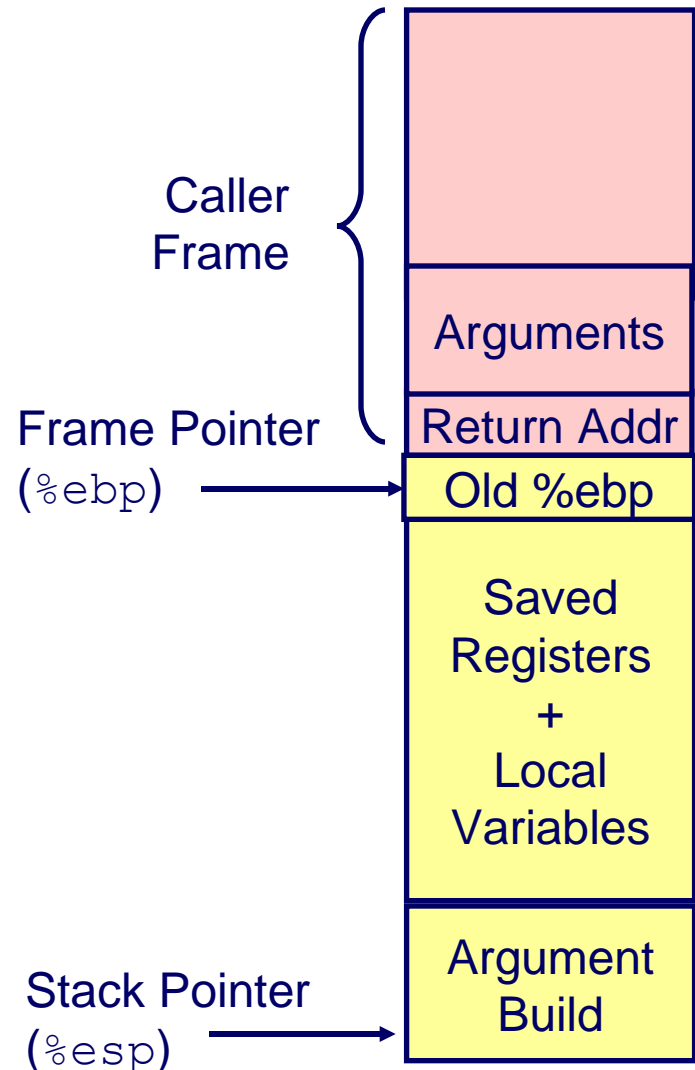
IA32/Linux Stack Frame

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

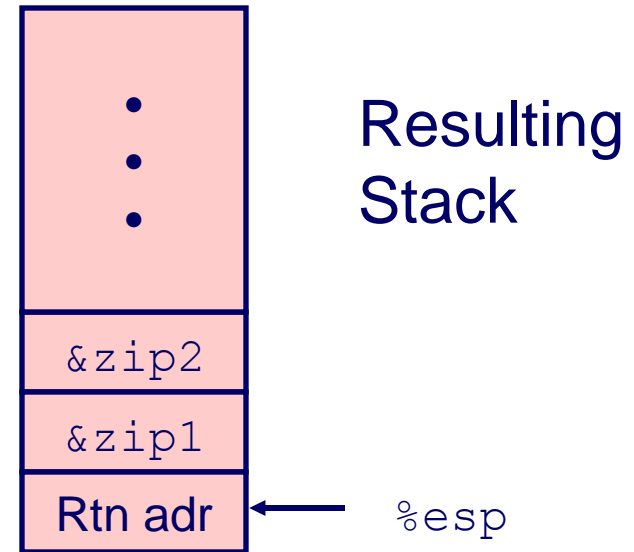
```
int zip1 = 15213;
int zip2 = 91125;

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:

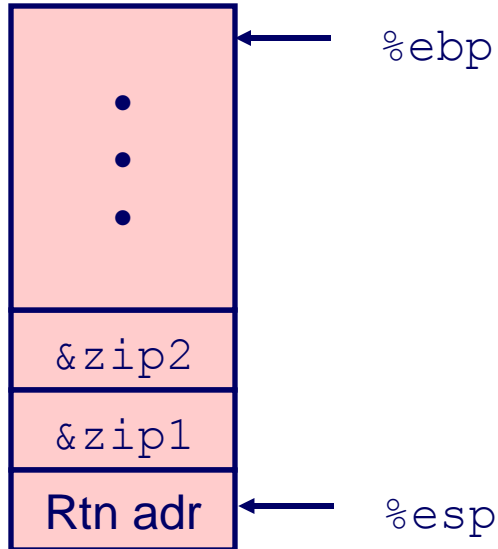
```
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
} Set Up

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

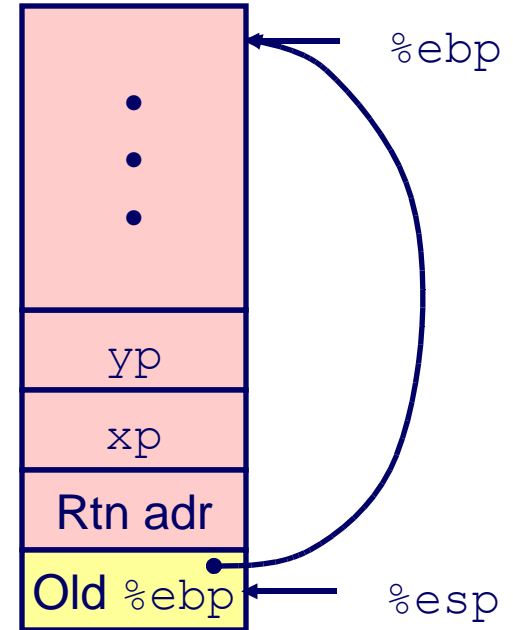
    movl -4(%ebp),%ebx
    movl %ebp,%esp
    popl %ebp
    ret
} Finish
```

swap Setup #1

Entering
Stack



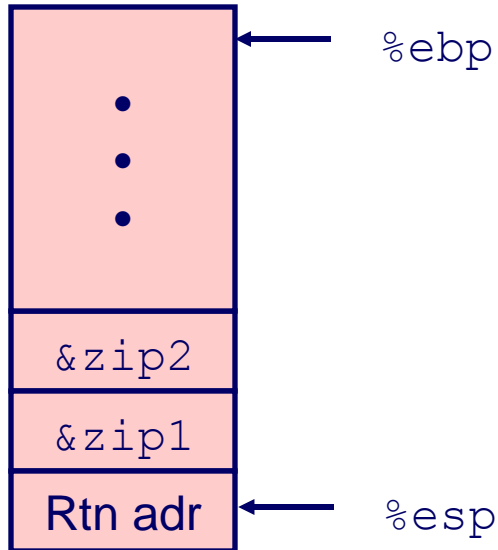
Resulting
Stack



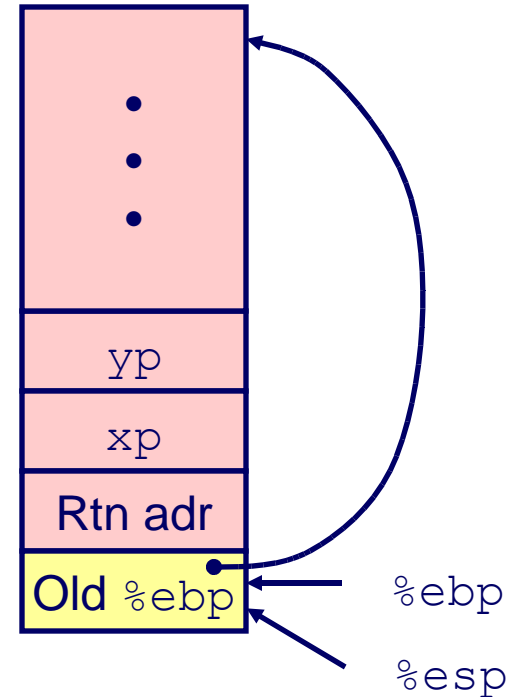
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

swap Setup #2

Entering Stack



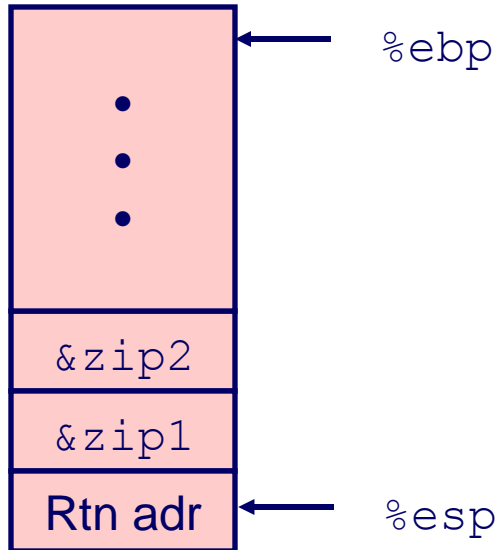
Resulting Stack



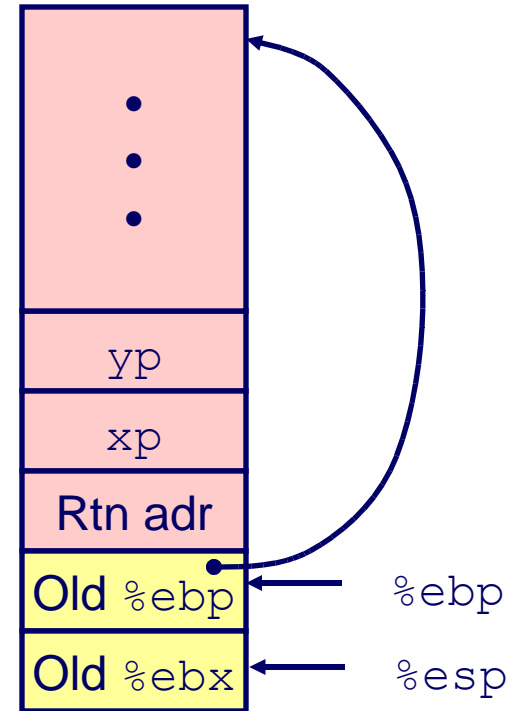
```
swap:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx
```

swap Setup #3

Entering Stack



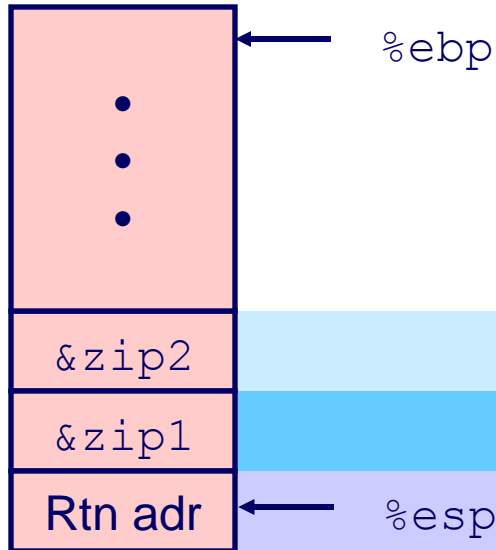
Resulting Stack



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

Effect of swap Setup

Entering Stack



Offset
(relative to %ebp)

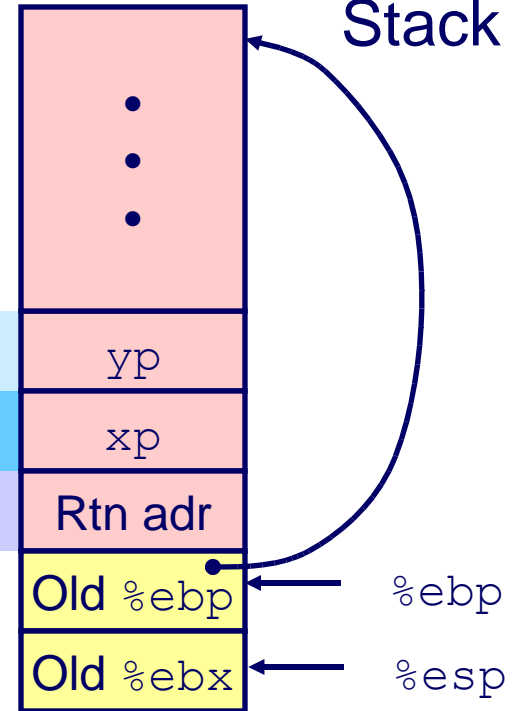
12

8

4

0

Resulting Stack



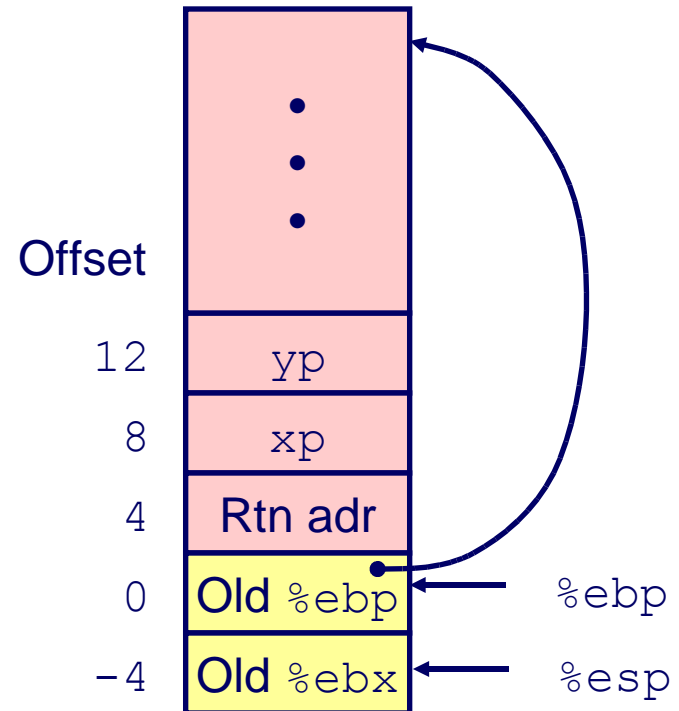
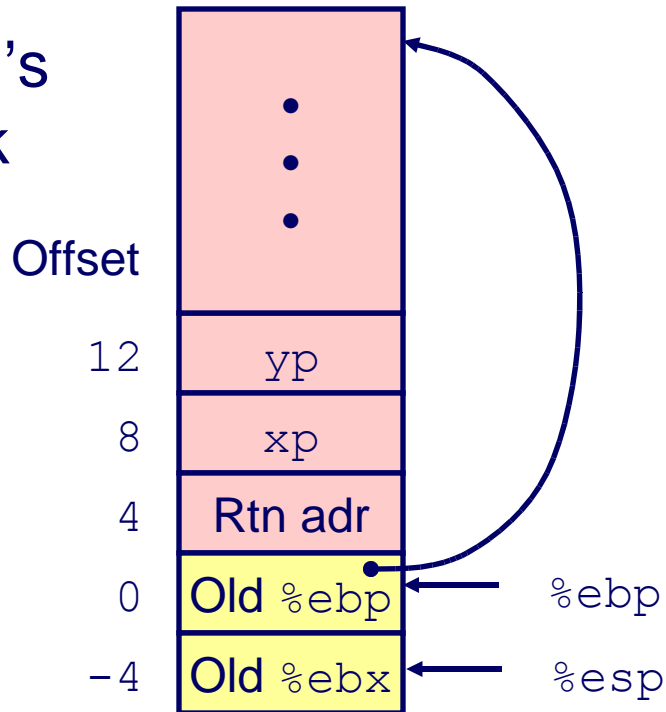
```

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

} Body

swap Finish #1

swap's
Stack



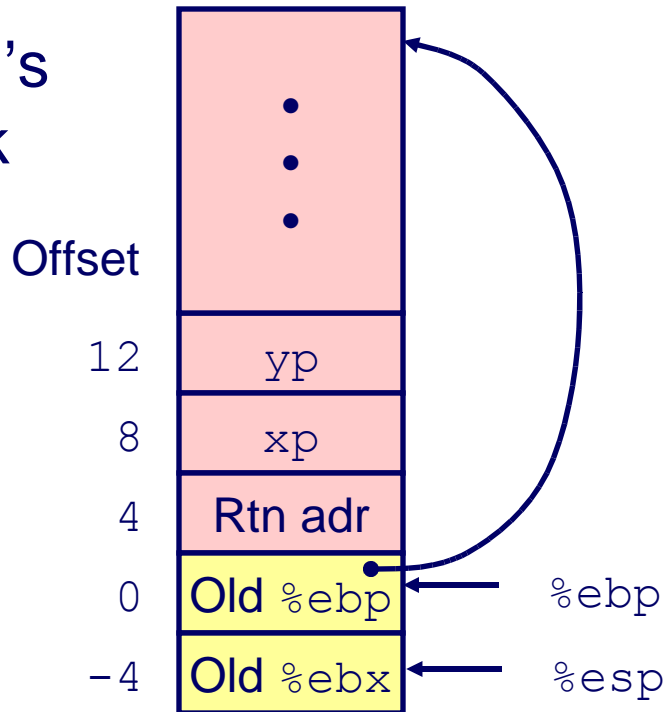
```
movl -4(%ebp), %ebx  
movl %ebp, %esp  
popl %ebp  
ret
```

Observation

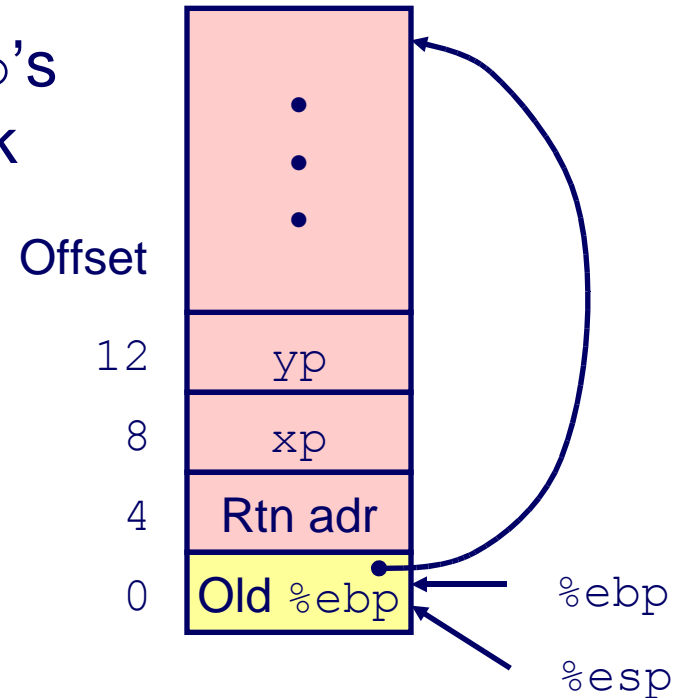
- Saved & restored register %ebx

swap Finish #2

swap's
Stack



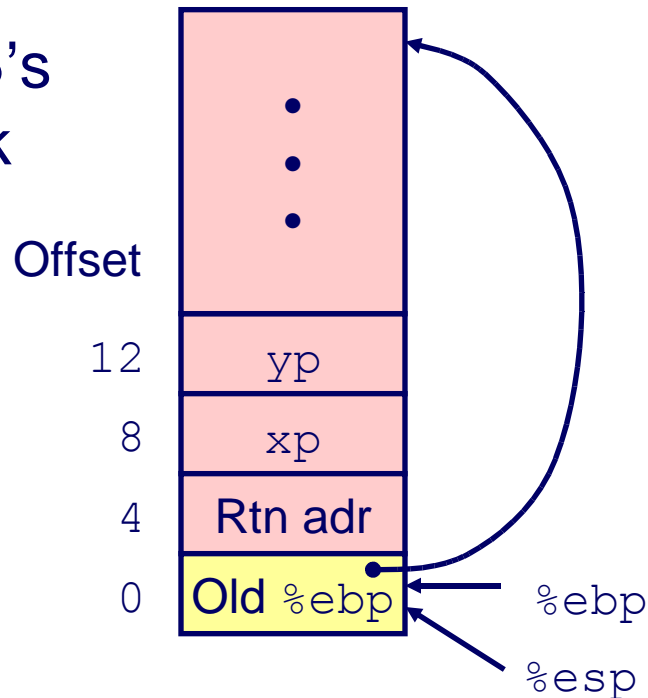
swap's
Stack



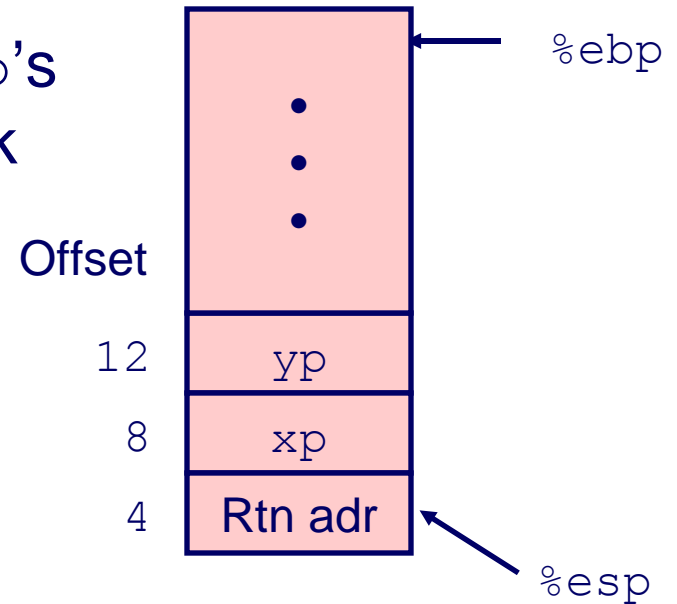
```
movl -4(%ebp), %ebx
movl %ebp, %esp
popl %ebp
ret
```

swap Finish #3

swap's
Stack

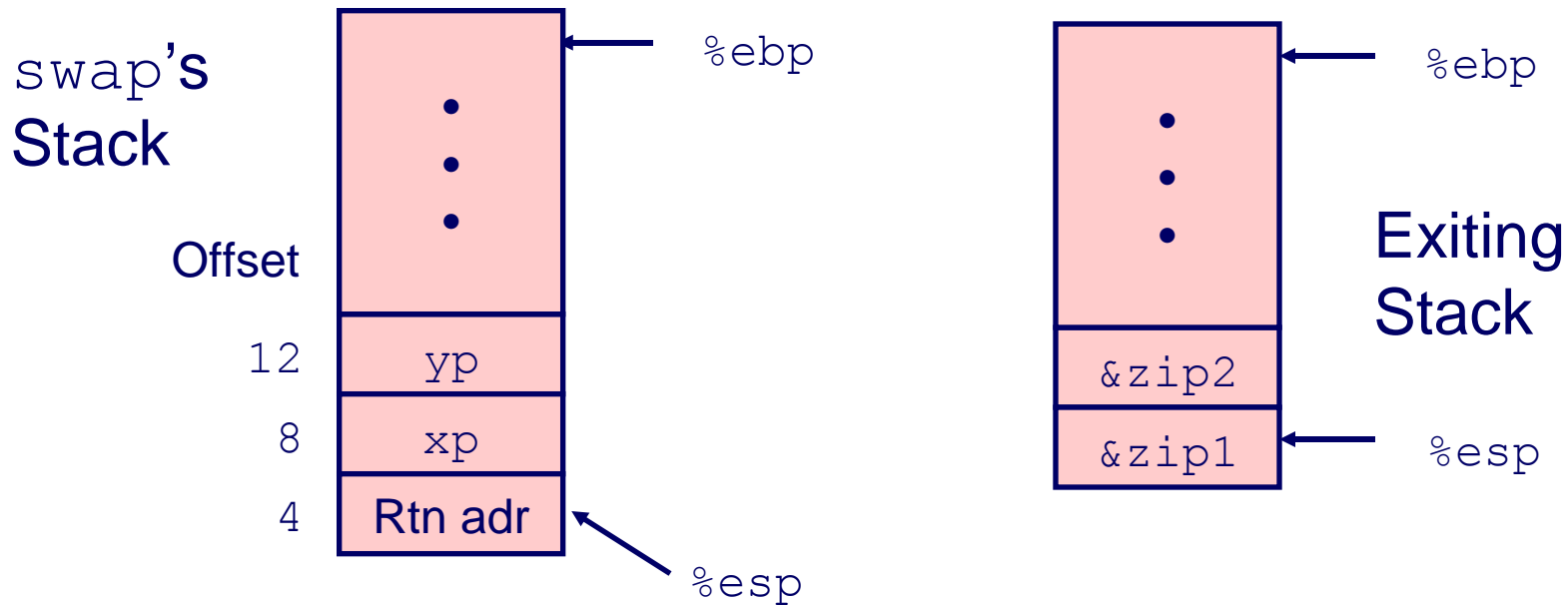


swap's
Stack



```
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, or %edx

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

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?

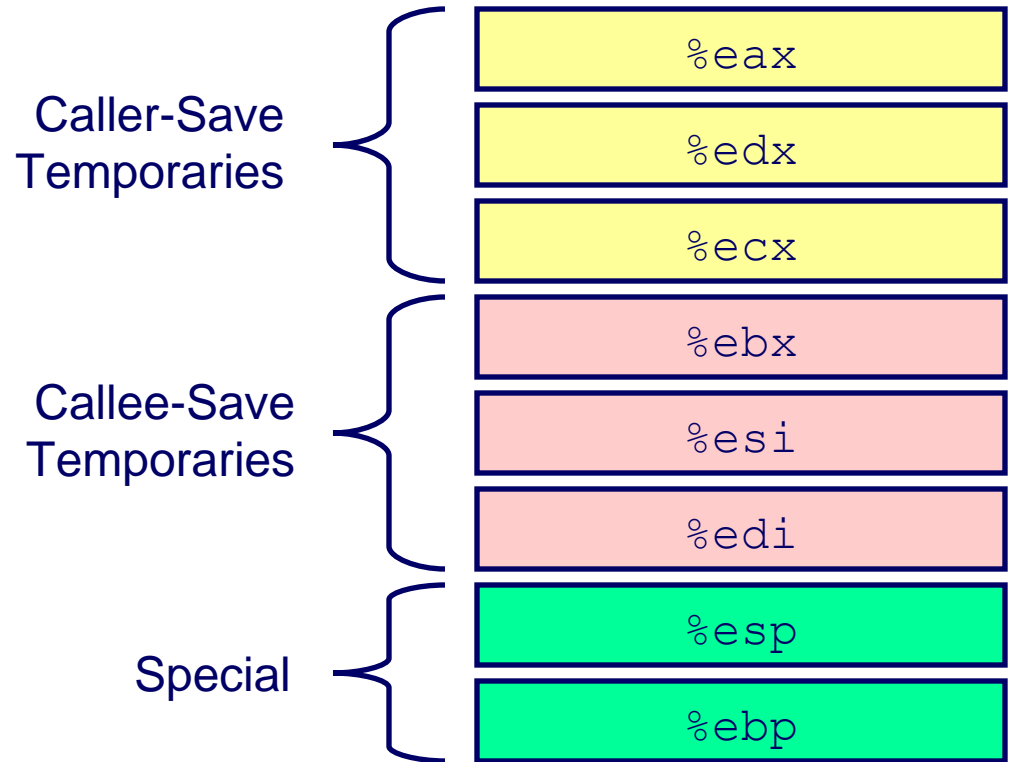
Conventions

- “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
 - Old values saved on stack prior to using
- Three managed as caller-save
 - Do what you please, but expect any callee to do so, as well
- Register `%eax` also stores returned value



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