The Hardware/Software Interface

CSE351 Winter 2011

Module 6: Memory Layout & Procedure Call

Memory Layout

- **Memory holds instructions and data**
- **There are four kinds of data, distinguished by their lifetime and mutability**
  - lifetime: when does it come into existence? when does it leave?
  - mutability: can it change value as the program runs?
- ***Note 1: we're talking about what is enforced at runtime, not whatever additional restrictions the compiler might enforce***
  - How are these different?
- ***Note 2: we're not talking about scope at all***
  - That's a purely language/compiler concept
    - (Plus the linker, which we'll see later in the course)

Lifetime

- **When is it <u>created</u> and <u>destroyed</u>?**
  - Before any code runs / after all code completes
    - i.e., program load time / program termination
    - Example: s = “literal string”;
  - During execution, according to rules set by the language
    - Example: local variables
      - { int myInt; myInt = getCount(); … }
  - During execution, because of specific requests by the programmer
    - Example: myFoo = new foo; // Note: this is NOT C (but close)

      …

      delete myFoo; // Not C either!

- ***Note: Java does automatic garbage collection. We'll think of that for now as 'delete,' even though the programmer doesn't write a delete statement.***

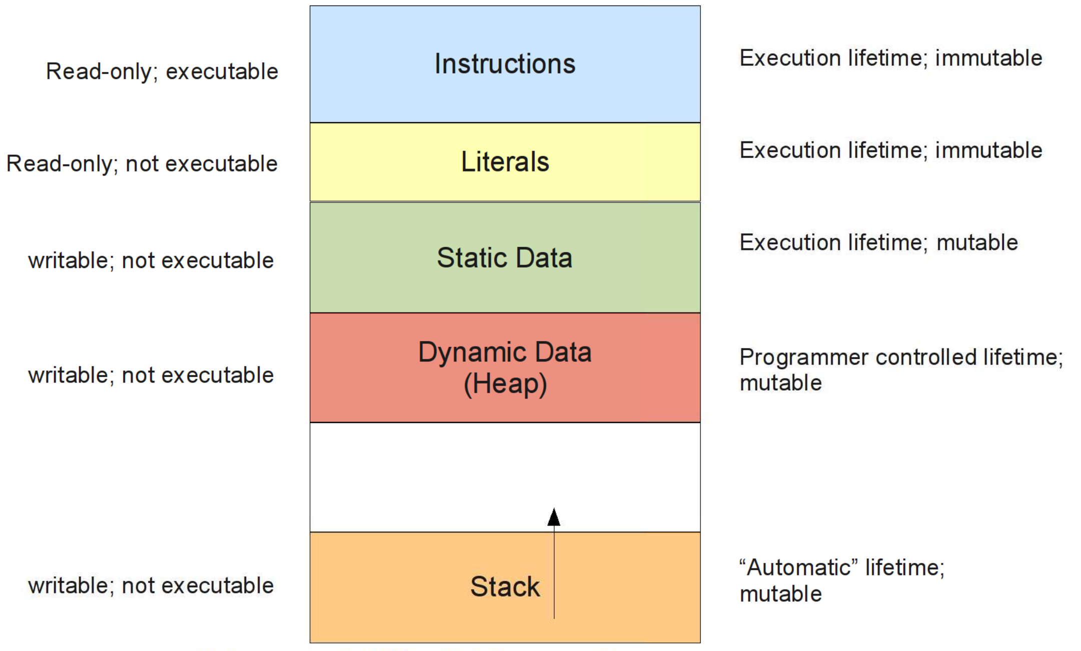
Mutability

- **<u>Mutable</u>: value can change during program execution**
  - Example: myInt = 4;

- **<u>Immutable</u>: value is not allowed to change during execution**
  - Example: char* s = “literal string”; // s initialized to address of literal

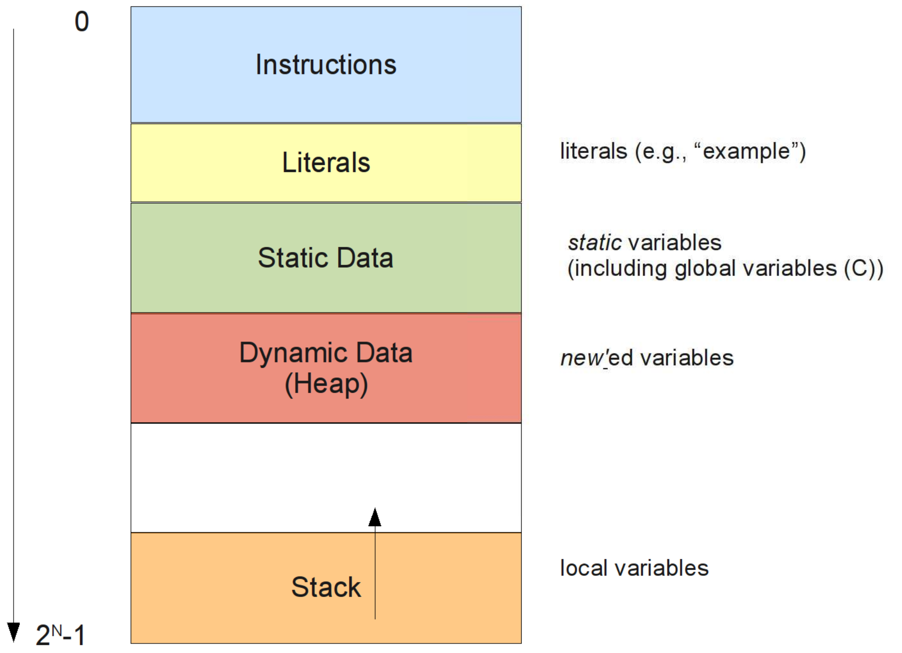
    strcpy(s, “new string”); // try to copy “new string” to *s

Memory Layout


Read-only; executable
Instructions
Execution lifetime; immutable
Read-only; not executable
Literals
Execution lifetime; immutable
writable; not executable
Static Data
Execution lifetime; mutable
writable; not executable
Dynamic Data (Heap)
Programmer controlled lifetime; mutable
writable; not executable
Stack
"Automatic" lifetime; mutable

Note: executability of data areas is system dependent...

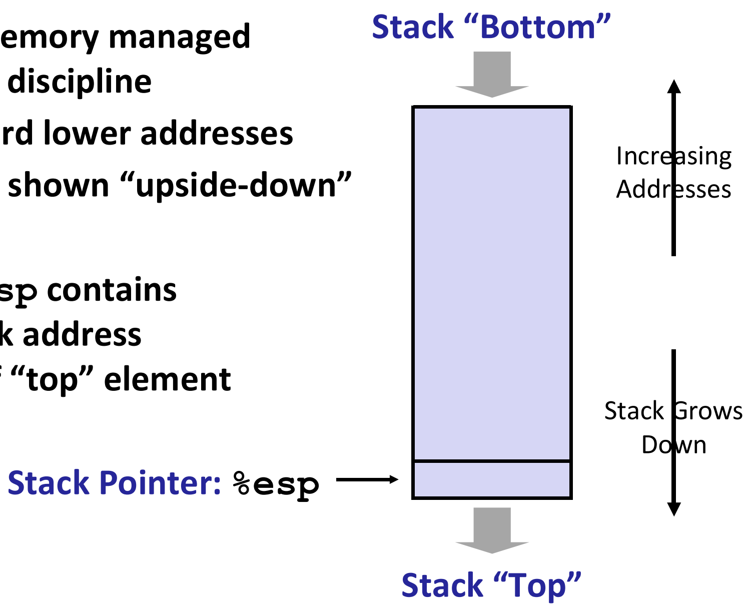
Memory Layout



IA32 Stack

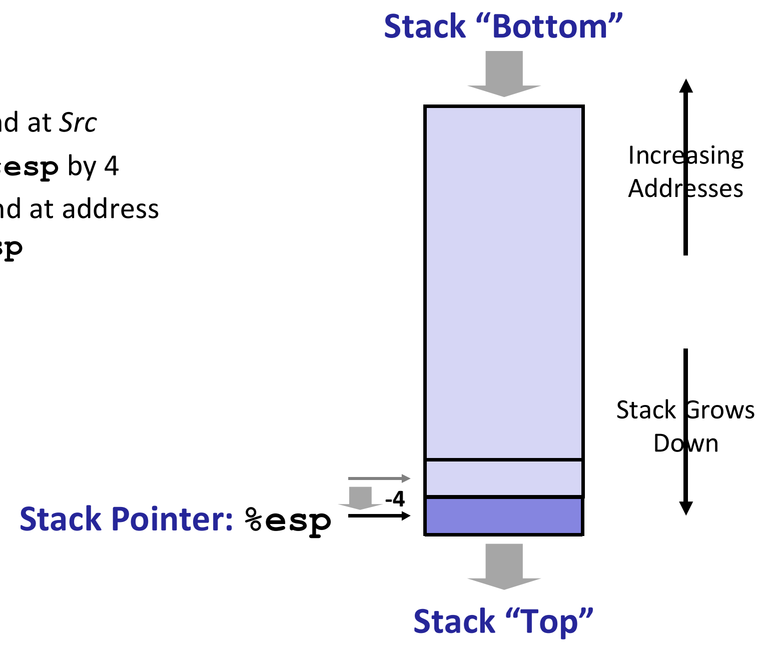
- Region of memory managed with a stack discipline
- Grows toward lower addresses
- Customarily shown "upside-down"

- Register `%esp` contains lowest stack address = address of "top" element



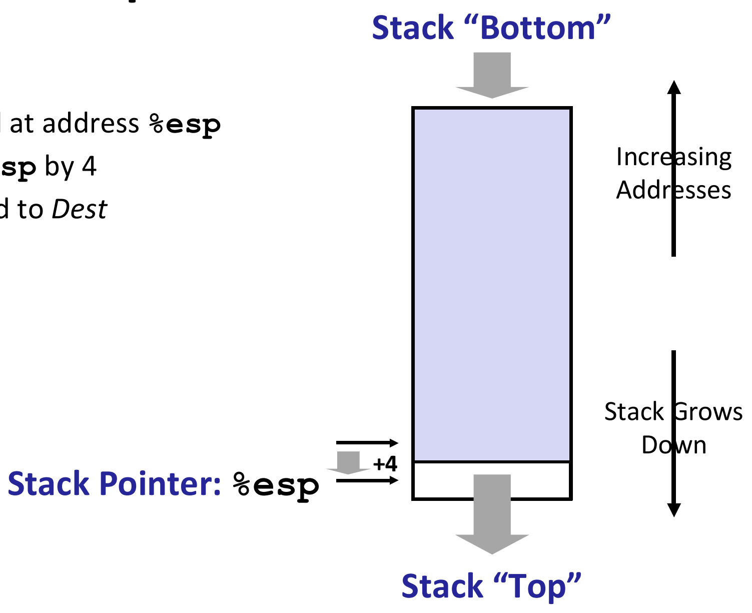
IA32 Stack: Push

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

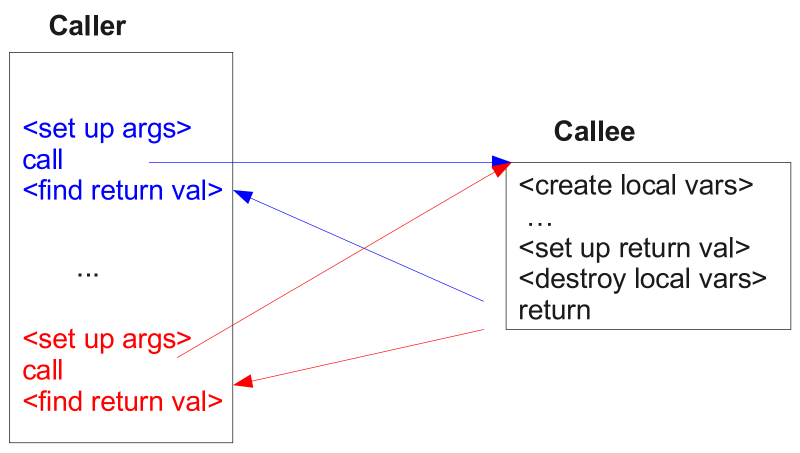


IA32 Stack: Pop

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

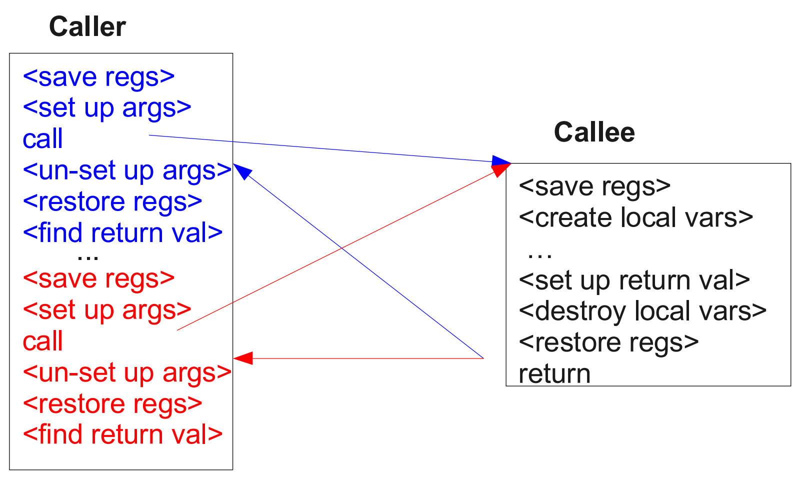


Procedure Call Overview



- Caller must leave args in a place callee knows to look for them
- Caller must leave "return address" in a place callee knows to look for it
- Caller and callee run on the same CPU → use the same registers
  - What can the caller expect the register state to be when callee returns?

Procedure Call Overview (cont.)



- The convention for where to leave/find things is called the procedure call linkage
  - It is implemented by the compiler
    - The hardware provides some basic functionality
    - Linkage convention details differ from one system type to another

Procedure Control Flow

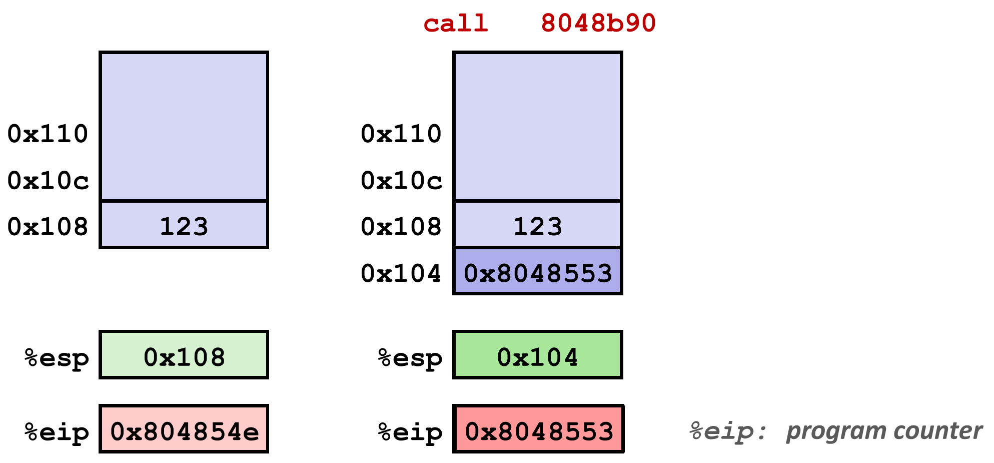
- **Use stack to support procedure call and return**
- **Procedure call: `call label`**
  - Push return address on stack
  - Jump to ***`label`***
- **Return address:**
  - Address of instruction immediately following **`call`**
  - Example from disassembly

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

  - Return address = **`0x8048553`**
- **Procedure return: `ret`**
  - Pop address from stack
  - Jump to address

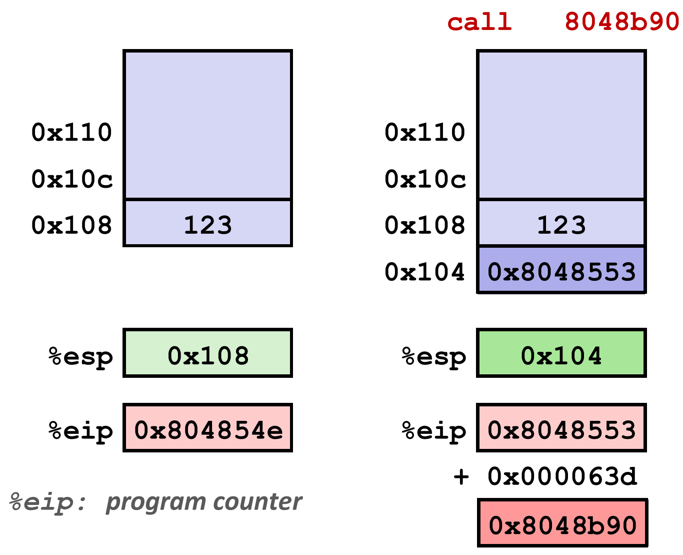
Procedure Call Example

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



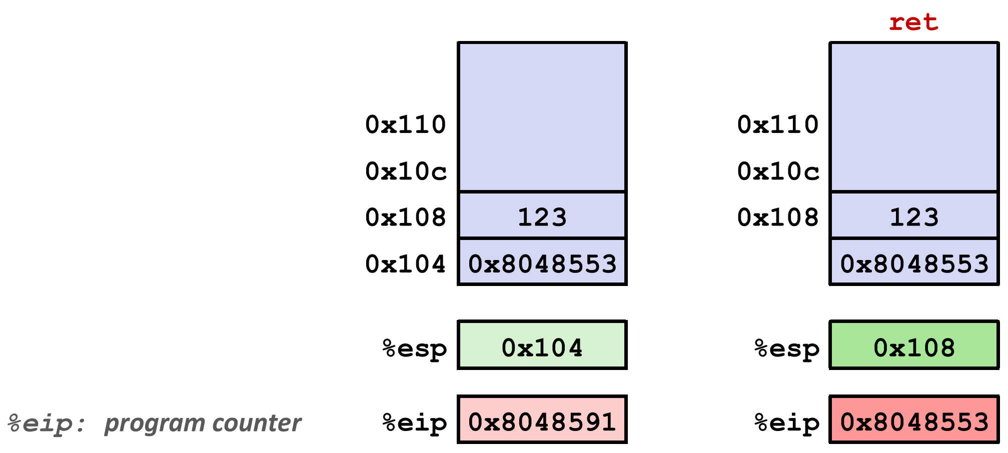
Procedure Call Example

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



Procedure Return Example

```
8048591:  c3                     ret
```



Stack-Based Languages

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

Call Chain Example

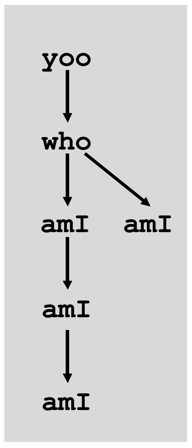
```
yoo(…)
{
    •
    •
    who();
    •
    •
    }
```

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

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

Procedure `amI` is recursive (calls itself)

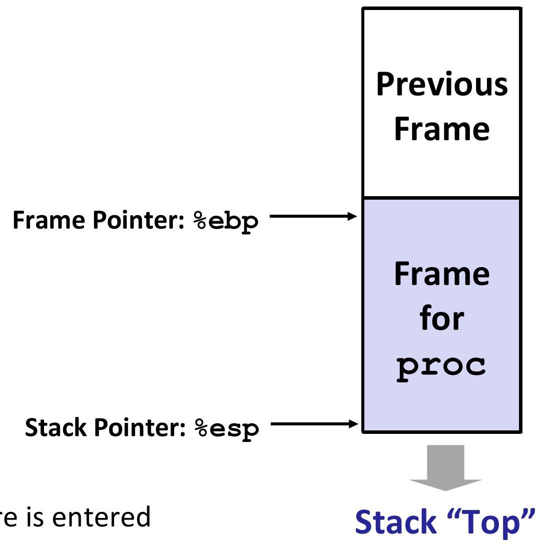
Example Call Chain



Stack Frames

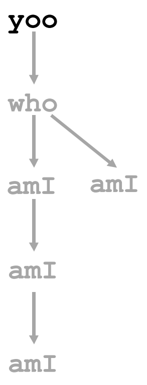
- **Contents**
  - Local variables
  - Return information
  - Temporary space

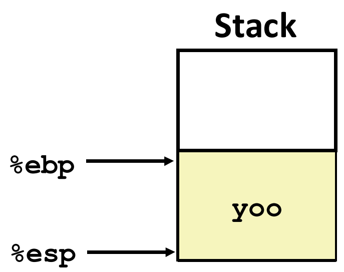
- **Management**
  - Space allocated when procedure is entered
    - "Set-up" code
  - Space deallocated upon return
    - "Finish" code



Example

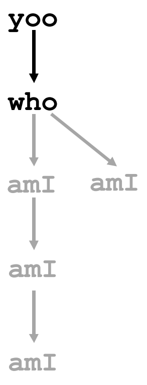
```
yoo(…)
{
   •
   •
   who();
   •
   •
   }
```

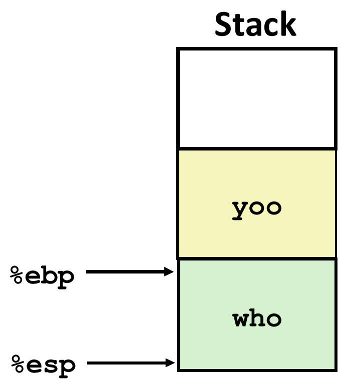




Example

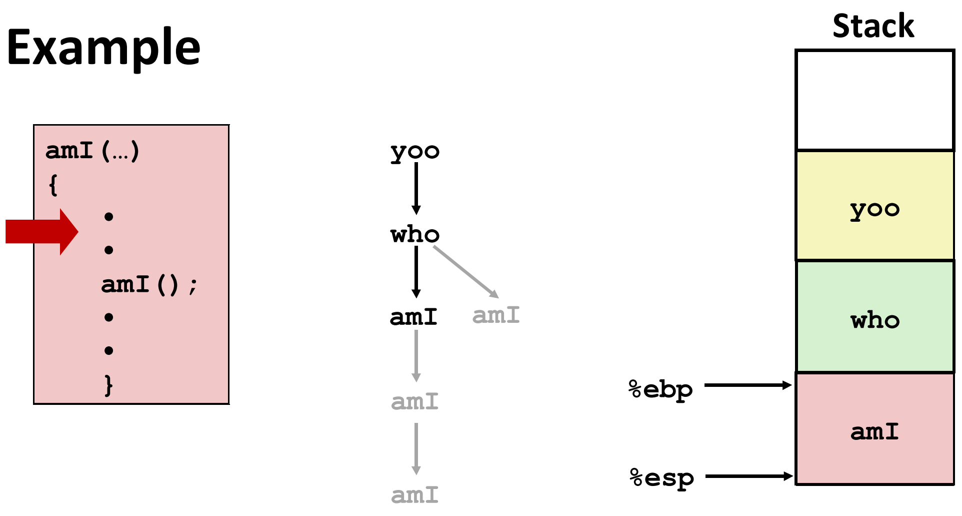
```
who(…)
{
   •  •  •
   amI();
   •  •  •
   amI();
   •  •  •
   }
```





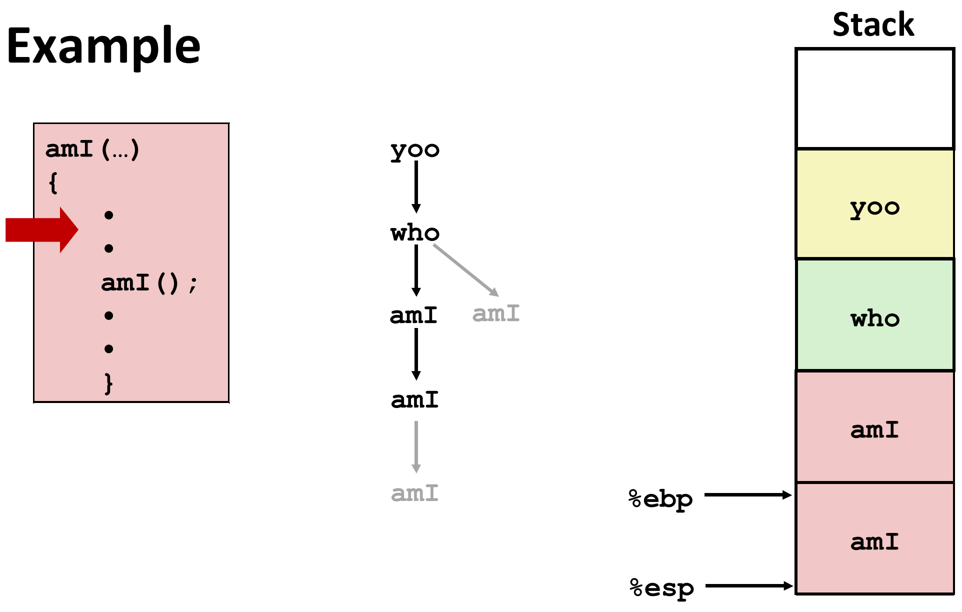
Example

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



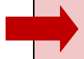
Example

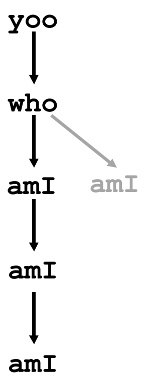
```
amI (…)
{
    •
    •
    amI();
    •
    •
    }
```

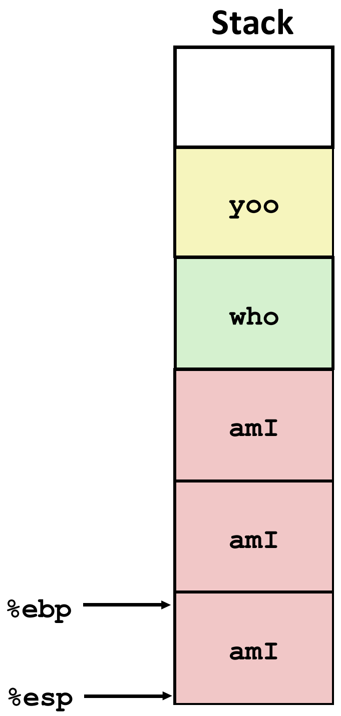


Example

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

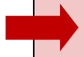


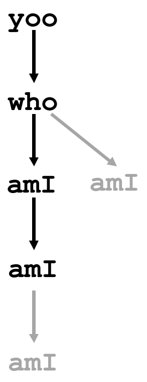


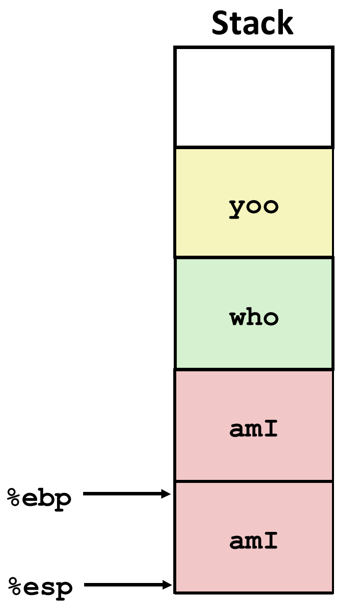


Example

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

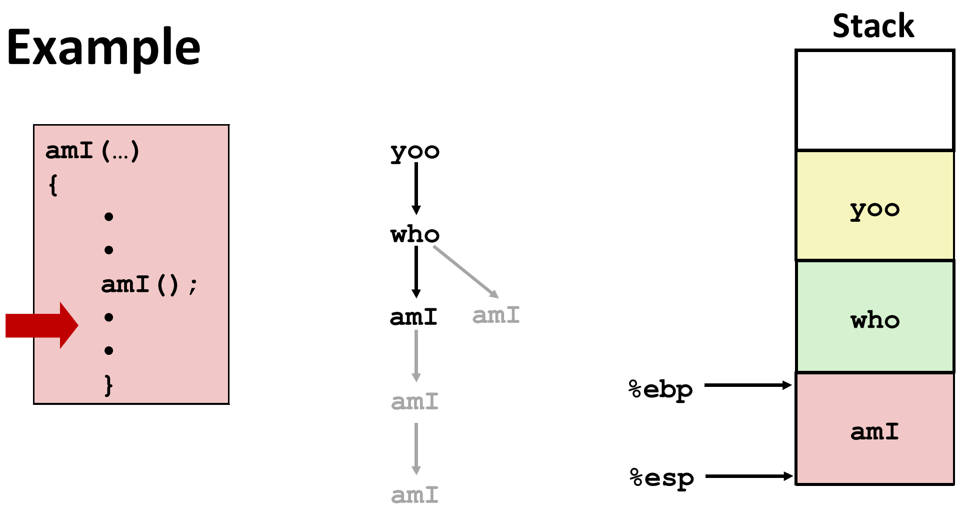






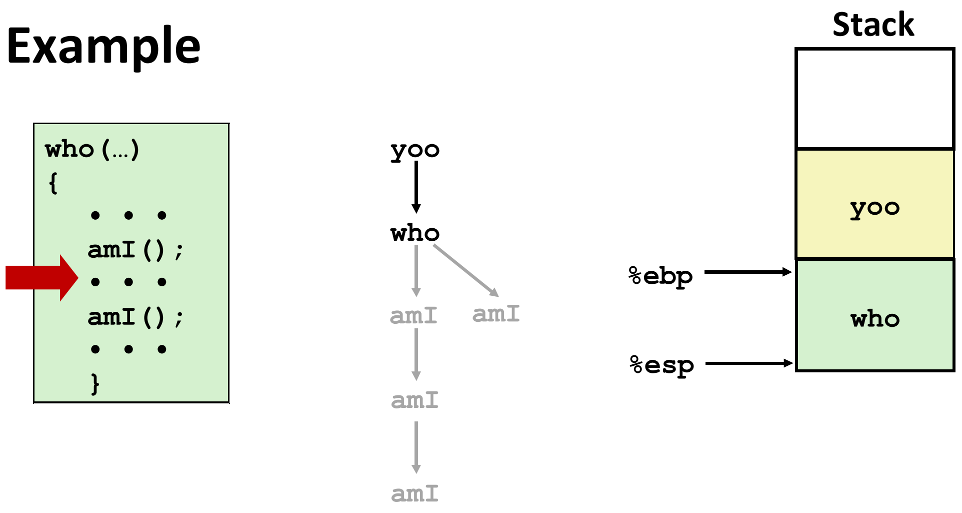
Example

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



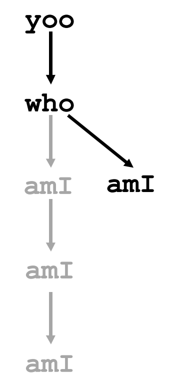
Example

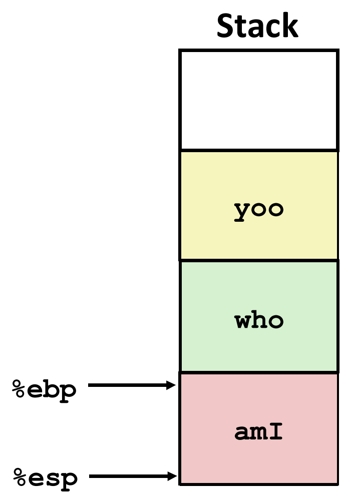
```
who (…)
{
      • • •
      amI ();
      • • •
      amI ();
      • • •
      }
```



Example

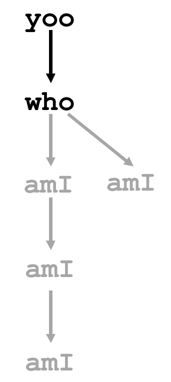
```
amI (…)
{
     •
     •
     •
     •
     •
     }
```

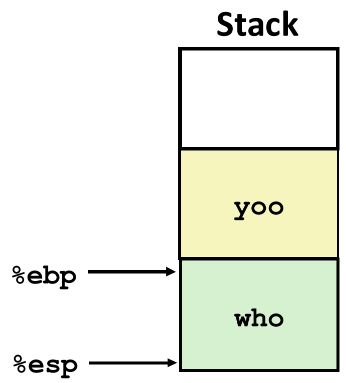




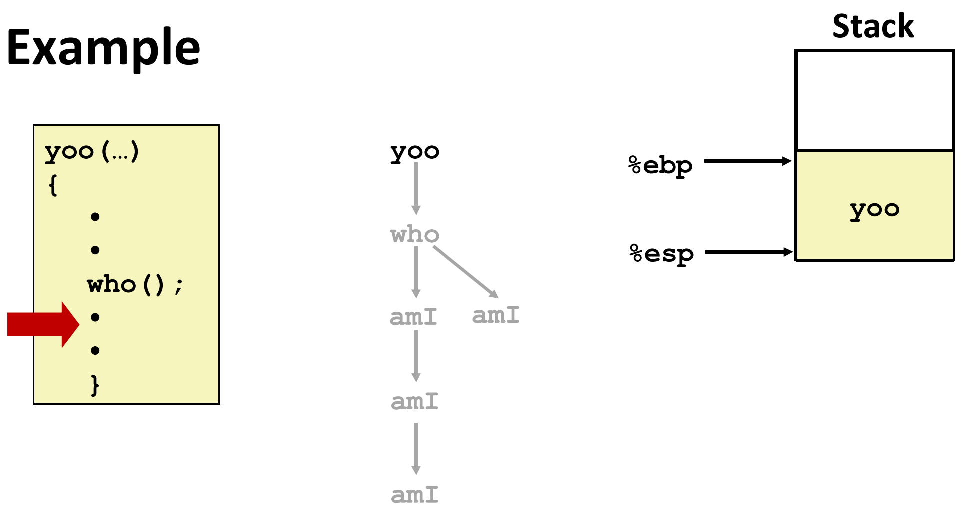
Example

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





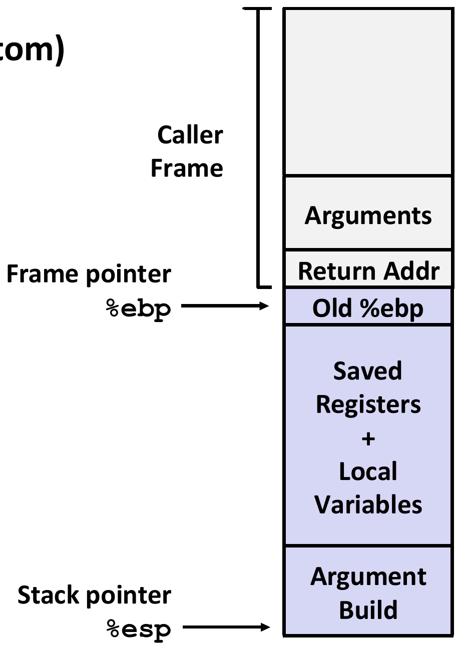
Example



IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - Old frame pointer
  - Local variables
    If can't be just kept in registers
  - Saved register context
    When reusing registers
  - "Argument build area"
    Parameters for function
    about to be called

- **Caller Stack Frame**
  - Return address
    Pushed by `call` instruction
  - Arguments for this call



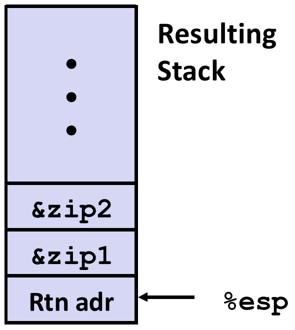
Revisiting swap

```
int zip1 = 15213;
int zip2 = 98195;

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
   • • •
```

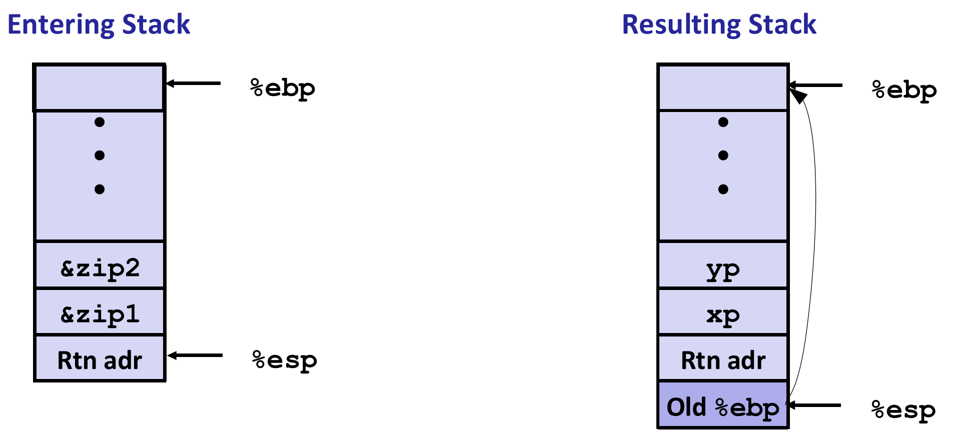


Revisiting swap

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

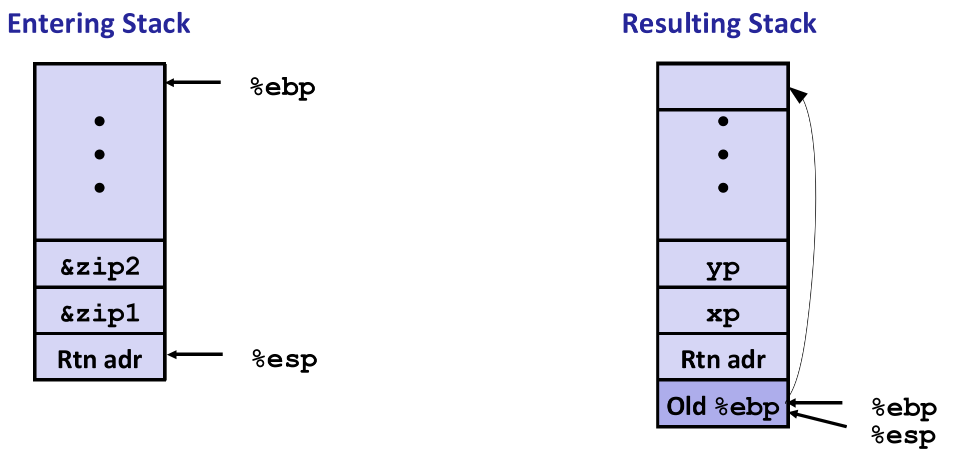
```
swap:
   pushl %ebp             ]
   movl %esp,%ebp         ]  Set Up
   pushl %ebx             ]

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

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

swap Setup #1



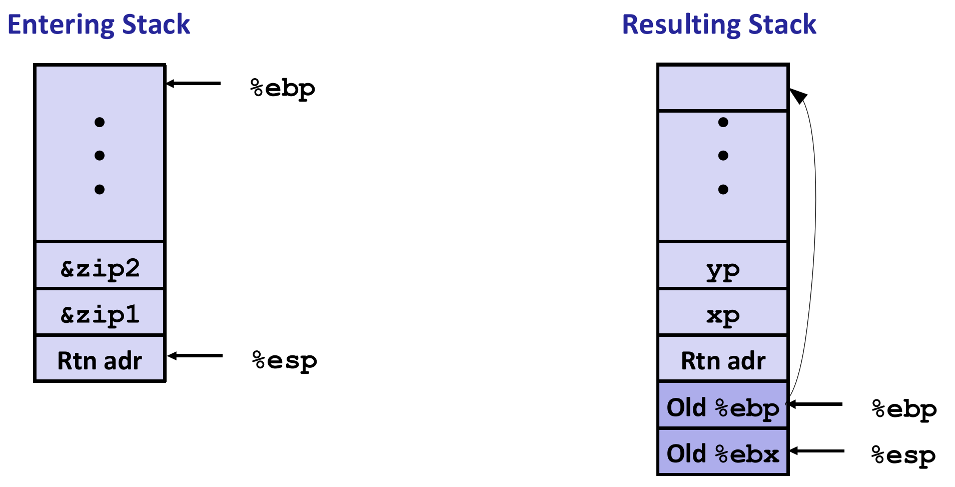
```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

swap Setup #1



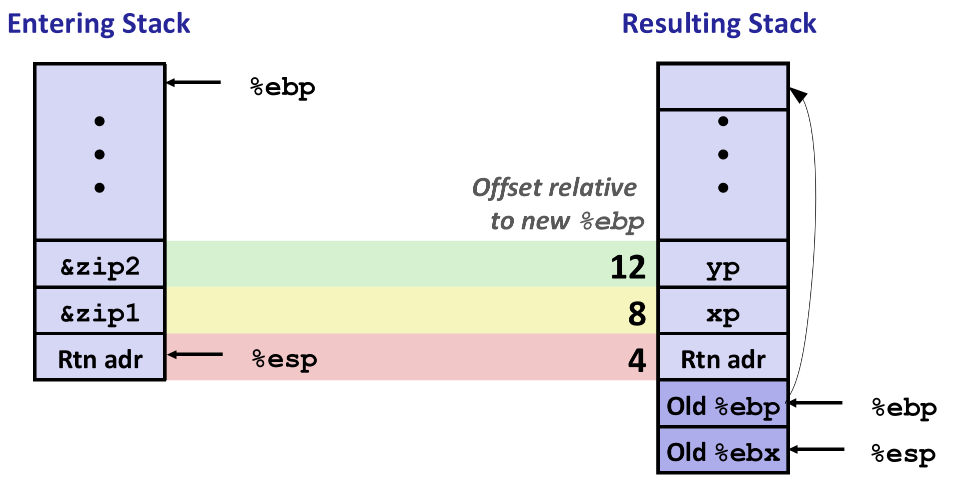
```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
```

swap Setup #1



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

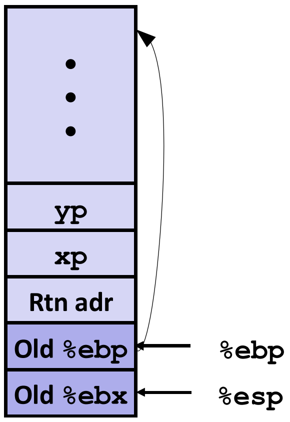
swap Setup #1



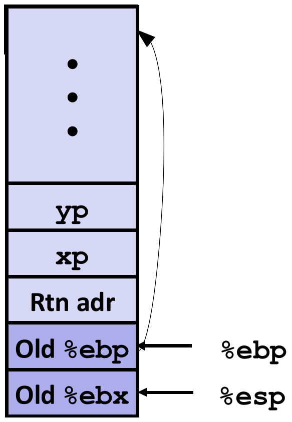
```
movl 12(%ebp),%ecx # initialize yp
movl 8(%ebp),%edx  # initialize xp
. . .
```

swap Finish #1

swap’s Stack



Resulting Stack

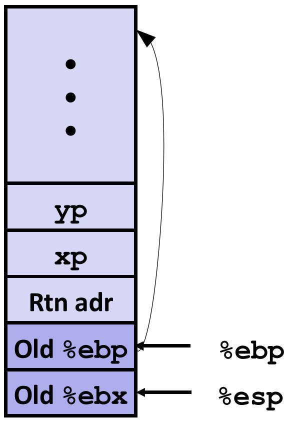


```
movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret
```

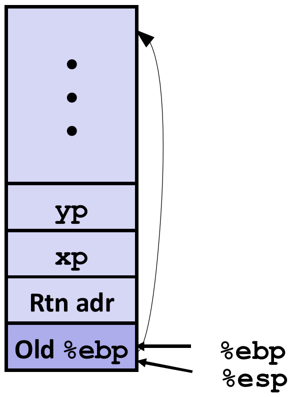
Observation: Saved and restored register %ebx

swap Finish #2

swap’s Stack

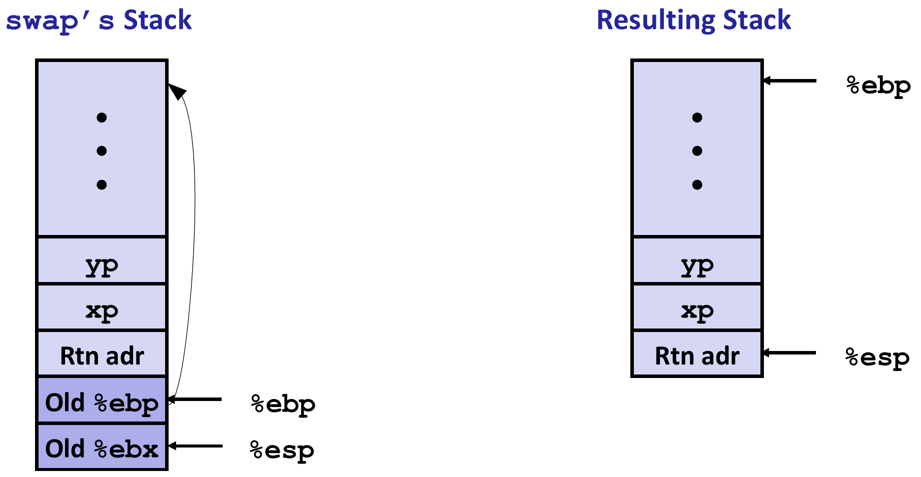


Resulting Stack



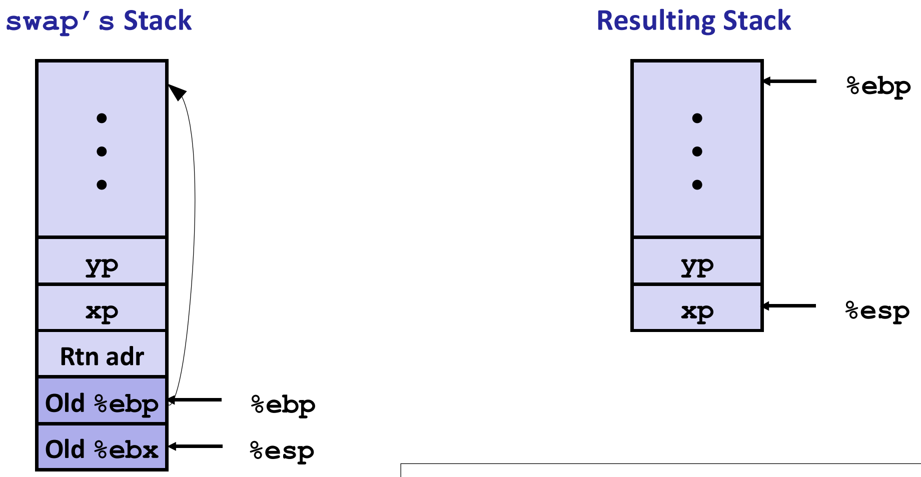
```
movl  -4(%ebp),%ebx
movl  %ebp,%esp
popl  %ebp
ret
```

swap Finish #3



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

swap Finish #4



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

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

Disassembled swap

```
080483a4 <swap>:
 80483a4:    55              push   %ebp
 80483a5:    89 e5           mov    %esp,%ebp
 80483a7:    53              push   %ebx
 80483a8:    8b 55 08        mov    0x8(%ebp),%edx
 80483ab:    8b 4d 0c        mov    0xc(%ebp),%ecx
 80483ae:    8b 1a           mov    (%edx),%ebx
 80483b0:    8b 01           mov    (%ecx),%eax
 80483b2:    89 02           mov    %eax,(%edx)
 80483b4:    89 19           mov    %ebx,(%ecx)
 80483b6:    5b              pop    %ebx
 80483b7:    c9              leave
 80483b8:    c3              ret
```

```
mov      %ebp,%esp
pop      %ebp
```

Calling Code

```
 8048409:    e8 96 ff ff ff    call  80483a4 <swap>
 804840e:    8b 45 f8          mov   0xfffffff8(%ebp),%eax
```

```
0x0804840e + 0xffffff96 = 0x080483a4
```

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  $98195, %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

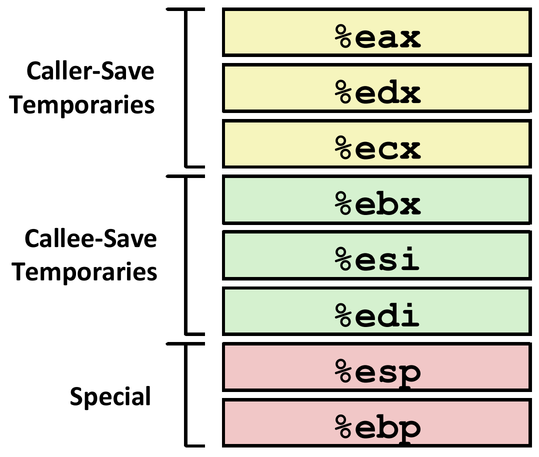
IA32/Linux Register Usage

- **`%eax, %edx, %ecx`**
  - Caller saves prior to call if values are used later

- **`%eax`**
  - also used to return integer value

- **`%ebx, %esi, %edi`**
  - Callee saves if wants to use them

- **`%esp, %ebp`**
  - special



Recursive Factorial

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

•**Registers**

- `%ebx` used, but saved at beginning & restored at end
- `%eax` used without first saving
  - expect caller to save
  - pushed onto stack as parameter for next call
  - used for return value

```
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
```

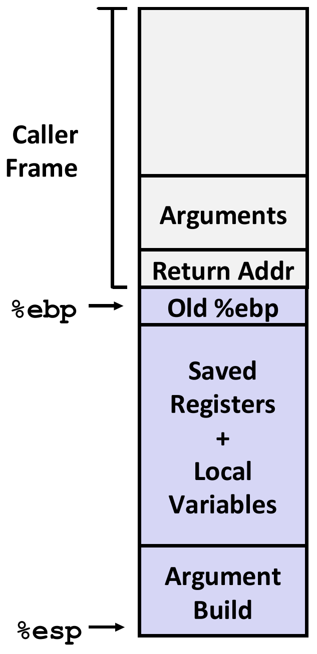
IA 32 Procedure Summary

- **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
  - 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


Caller Frame
Arguments
Return Addr
%ebp
Old %ebp
Saved Registers + Local Variables
Argument Build
%esp