CS429: Computer Organization and Architecture

Instruction Set Architecture IV

Warren Hunt, Jr. and Bill Young
Department of Computer Sciences
University of Texas at Austin

Last updated: October 13, 2014 at 16:17

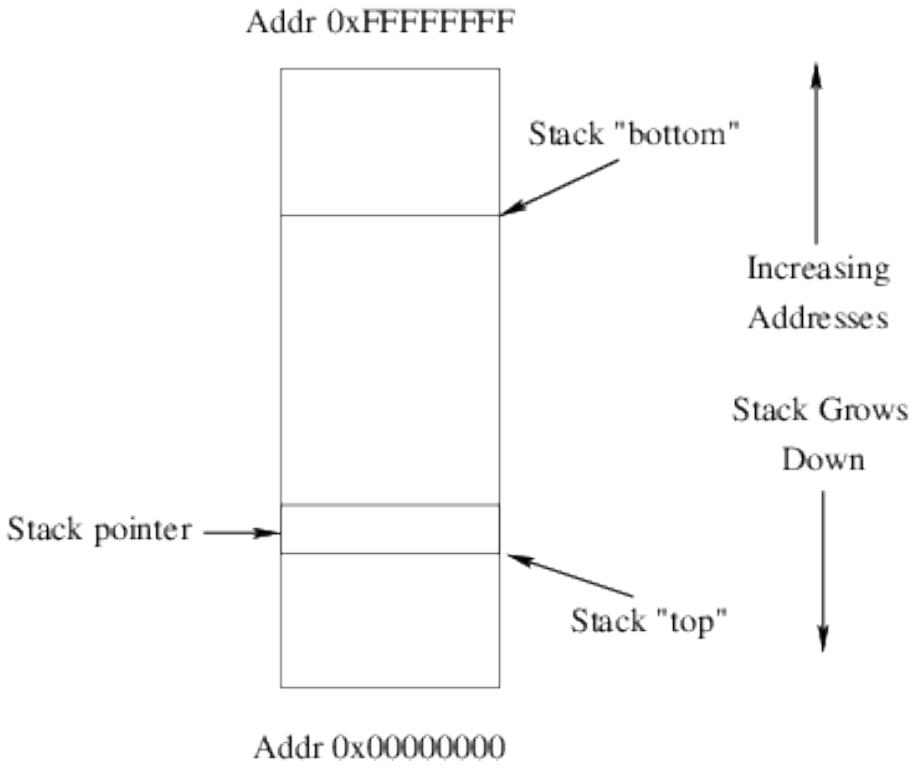
Procedural Memory Usage

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

- Where is the memory that holds t0 and t1 (or local variables in general)?
- What happens if we run out of registers (x86 only has 8)?
- Where are parameters passed from callers to callee? Registers? Memory? What memory?

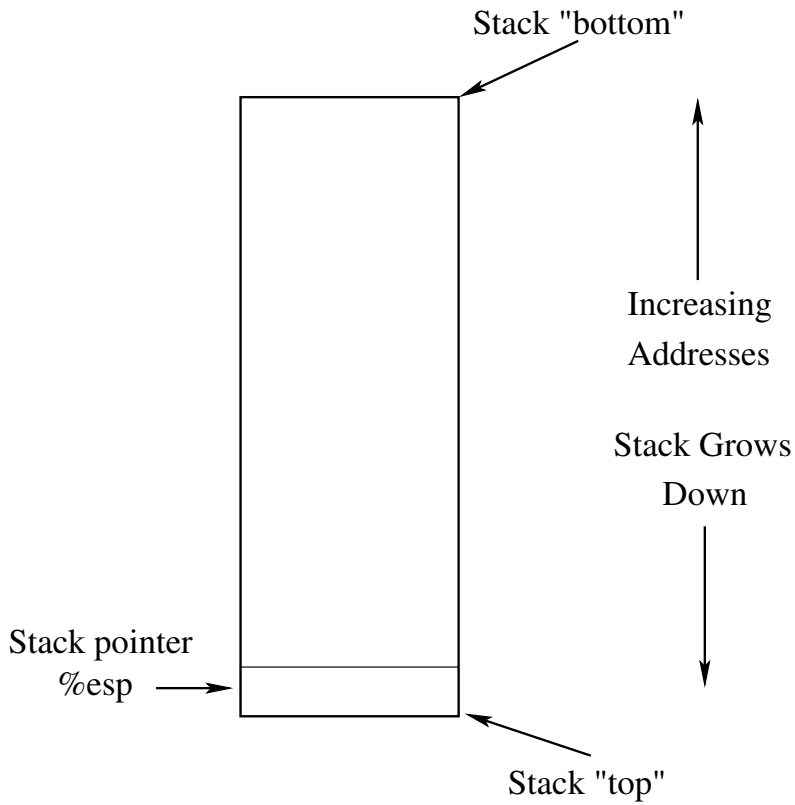
Stack Data Structure

- Stack is a LIFO (Last In, First Out) structure.
- It's allocated somewhere in memory; where doesn't really matter as long as we store the stack pointer.
- By convention, the stack grows toward smaller addresses, but it could be the other way.
- Values within the stack are referenced relative to the stack pointer.



IA32 Stack

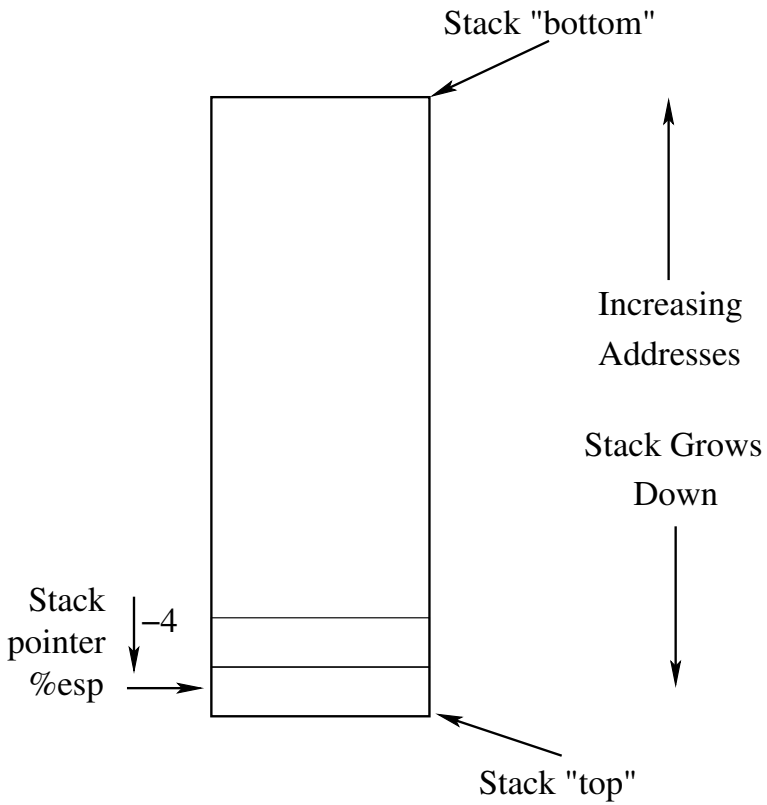
- Region of memory managed with stack discipline.
- Grows toward lower addresses.
- Register %esp is the stack pointer, and always points to lowest stack address, which is the top element on the stack.



IA32 Stack Pushing

Pushing

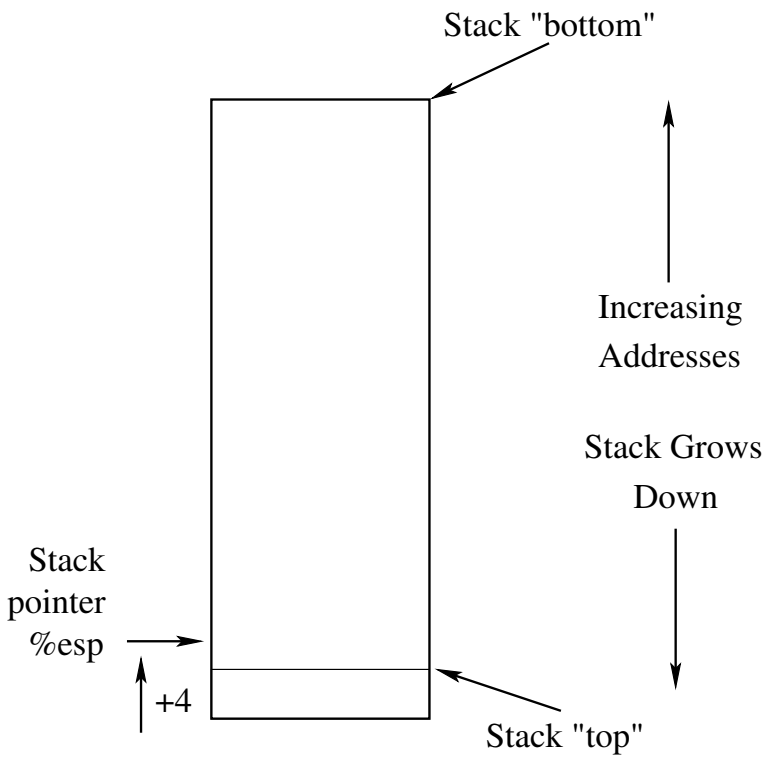
- `pushl` Src
- Decrement %esp by 4
- Write operand at address given by %esp



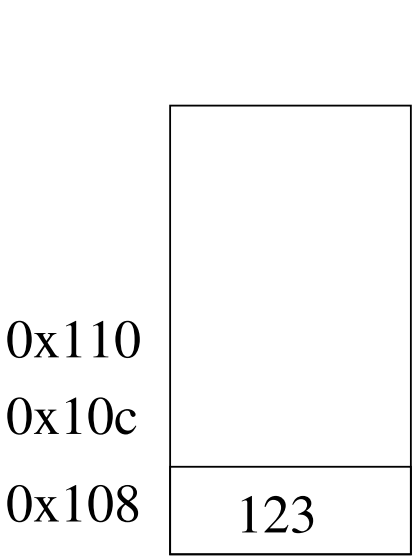
IA32 Stack Popping

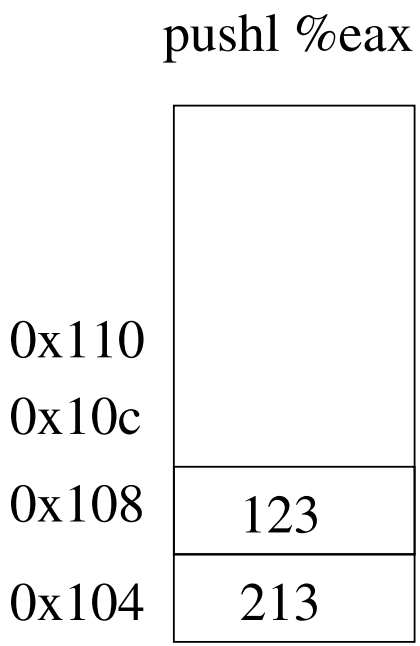
Popping

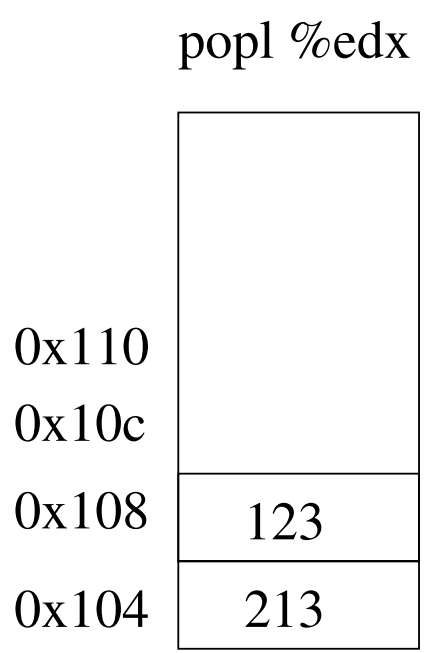
- `popl` Dest
- Read operand at address given by %esp
- Increment %esp by 4
- Write to Dest



Stack Operation Examples







%eax	213
%edx	555
%esp	0x108

%eax	213
%edx	555
%esp	0x104

%eax	213
%edx	213
%esp	0x108

Elements for Procedures

- Need to compute address of first instruction of called procedure.
- Place to store passed parameters.
  - Call by value
  - Call by reference
- Need to compute and store return address (first instruction after point of procedure call).
- Need to pass returned value(s) back to the caller.

Procedure Control Flow

We use the stack to support procedure call and return.

Procedure call:
Push return address on stack and jump to label

```
call label
```

Return address value:

- Address of instruction beyond call site.
- Example from disassembly:

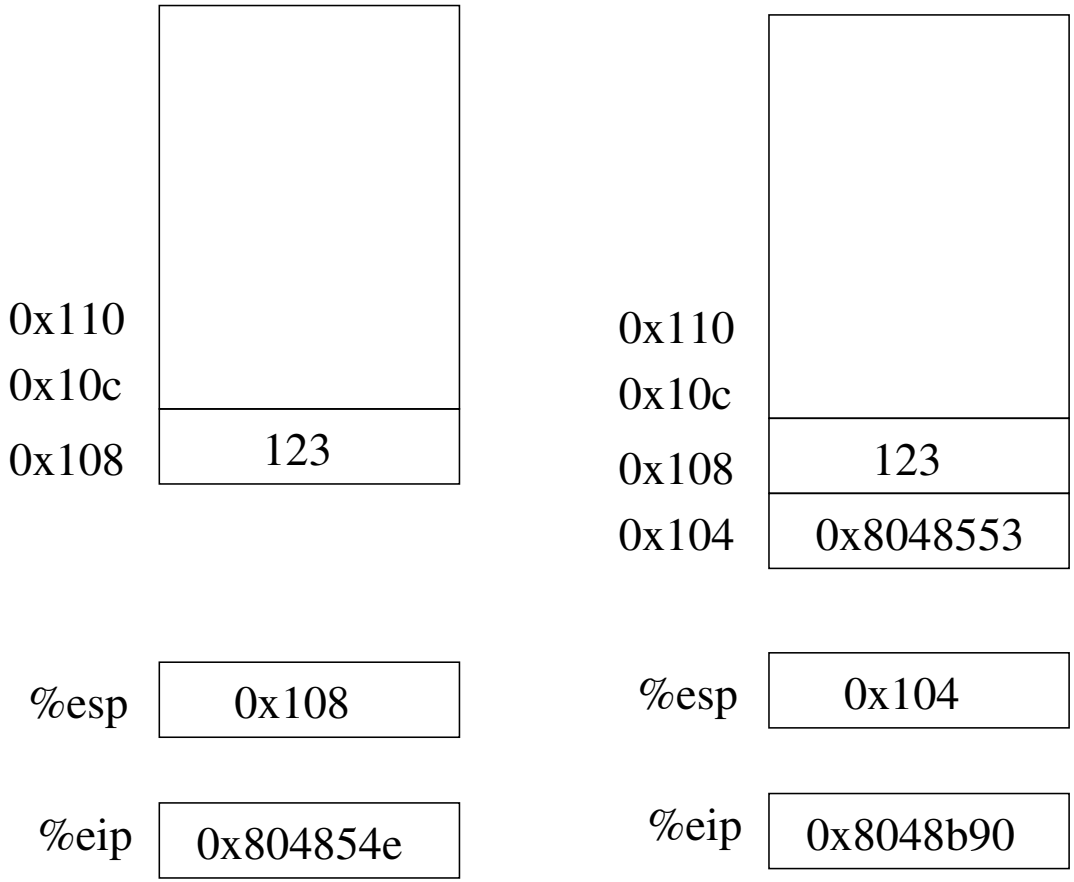
```
804854e:   e8 3d 06 00 00     call  8048b90 <main>
8048553:   50                 pushl %eax
```

Procedure return:
Pop address from stack and jump to address.

```
ret
```

Procedure Call Example

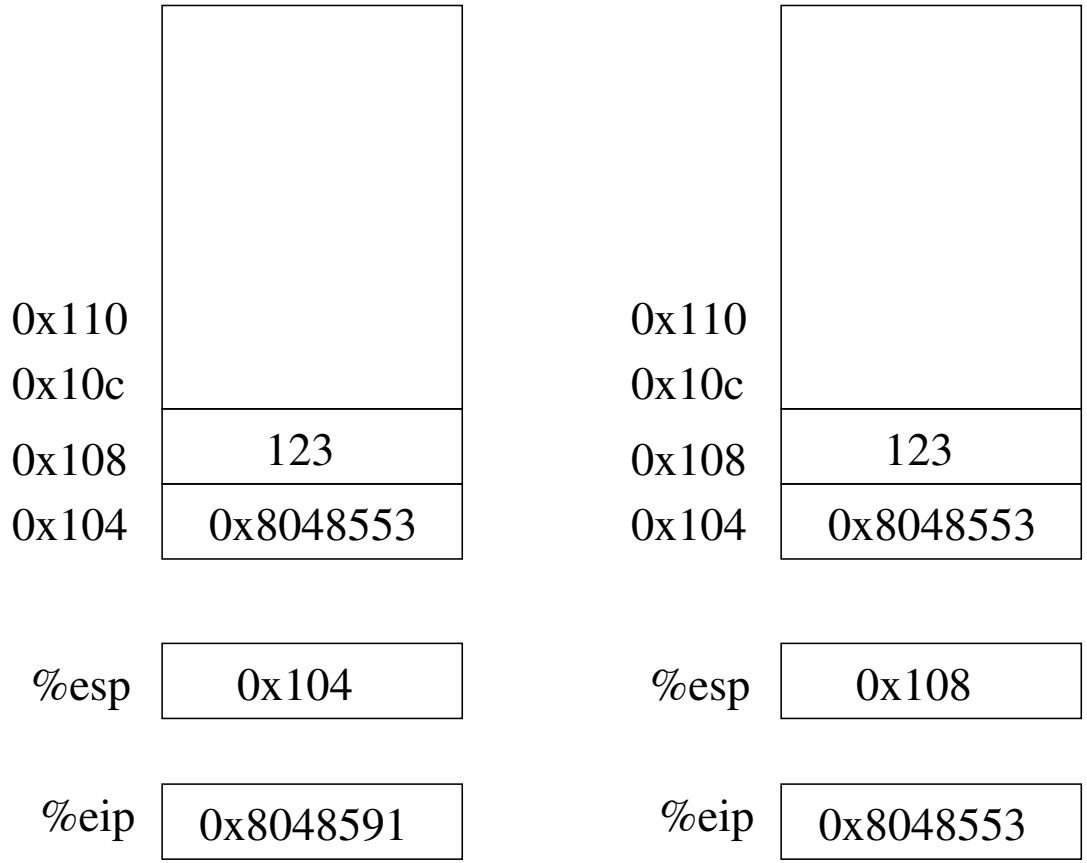
```
804854e:   e8 3d 06 00 00      call 8048b90 <main>
8048553:   50                  pushl %eax
```



%eip is the program counter.

Procedure Return Example

```
8048591:   c3                    ret
```



`%eip` is the 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 address

Stack Discipline

- State for given procedure needed for a limited time (from call to return)
- Callee always returns before caller does.

Stack is allocated in Frames: state for a single procedure invocation.

Call Chain Example

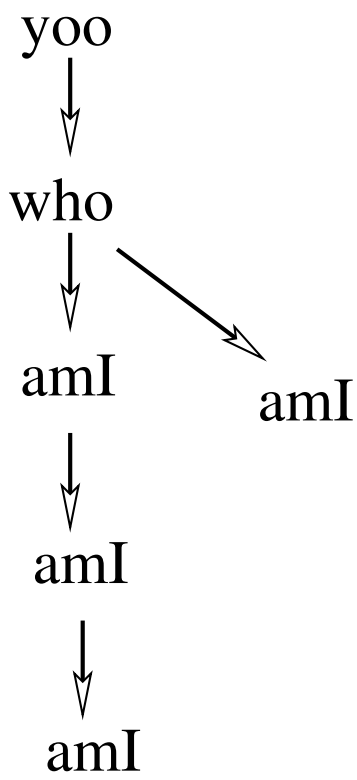
Code Structure

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

who(...) {
    ...
    aml();
    ...
    aml();
    ...
}

aml(...) {
    ...
    aml();
    ...
}
```

Procedure aml is recursive.



Stack Frames

Contents

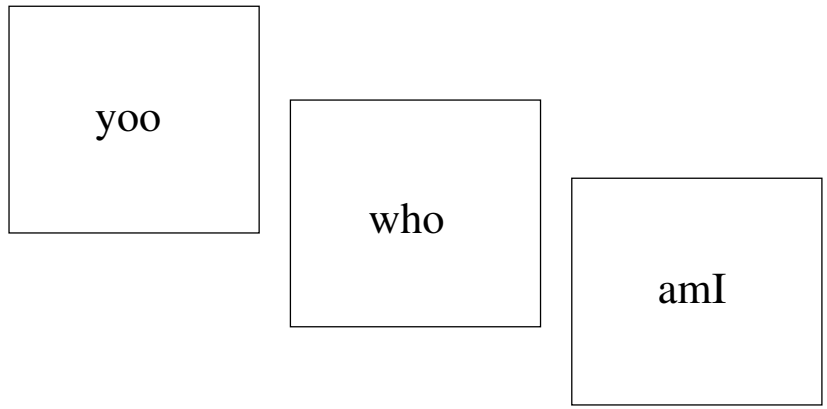
- Local variables
- Return information
- Temporary space

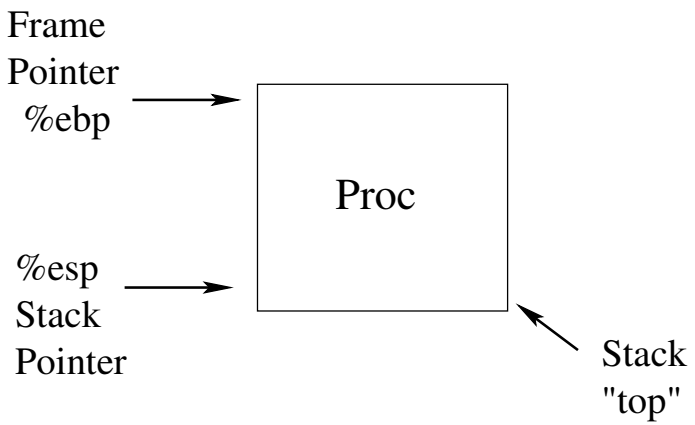
Management

- Space is allocated when you enter the procedure ("set-up" code).
- Space is deallocated at return ("finish" code).

Pointers

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



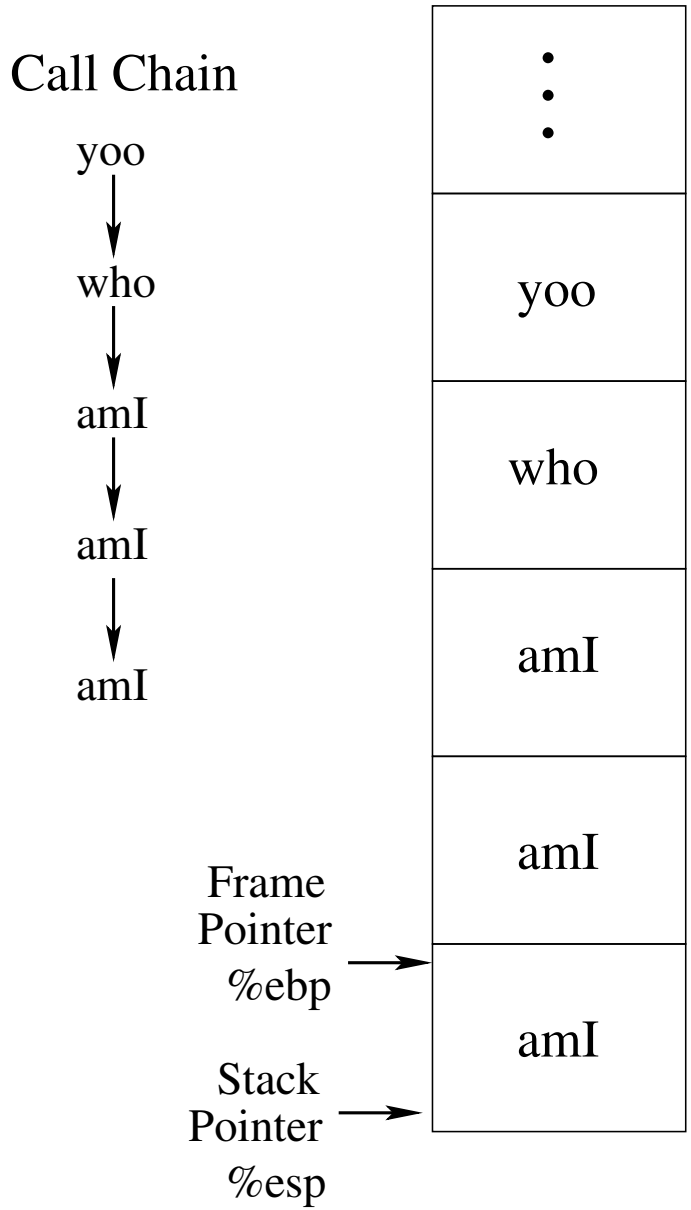


Stack Snapshot

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

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

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



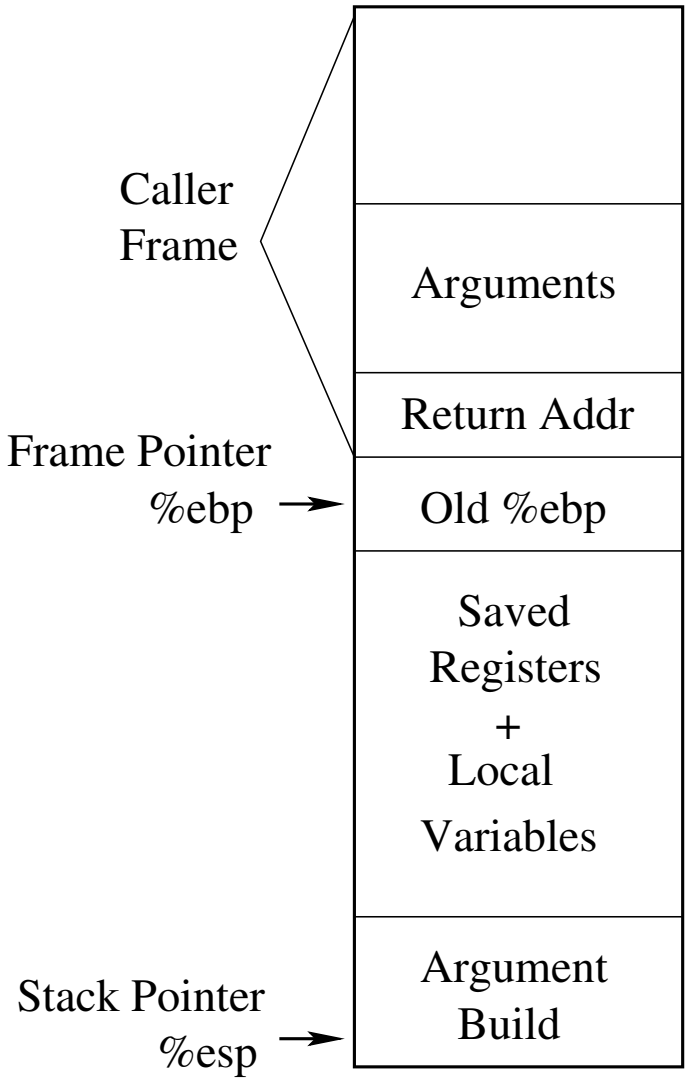
IA32/Linux Stack Frame

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

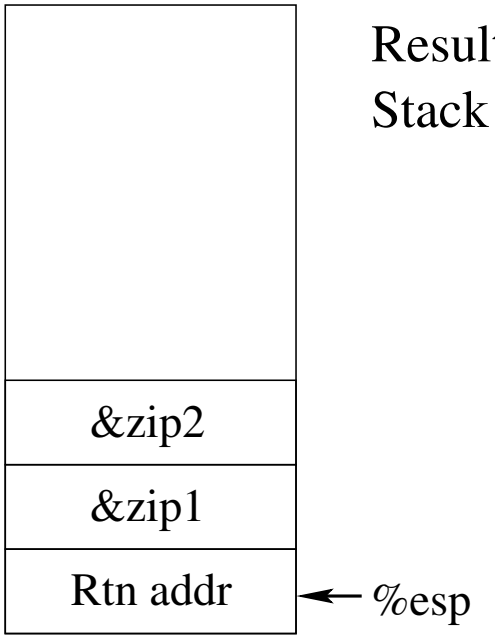
```
int zip1 = 15213;
int zip2 = 91125;

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

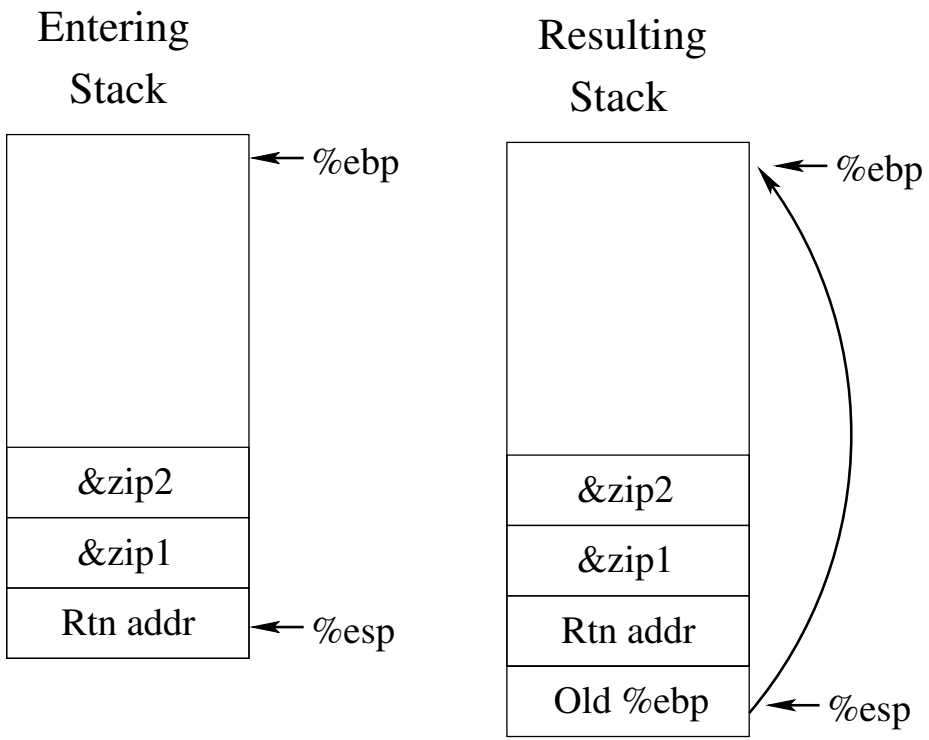
Revisiting Swap

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

```
swap:
    # Set up
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx
    # Body
    movl  12(%ebp),%ecx
    movl  8(%ebp),%edx
    movl  (%ecx),%eax
    movl  (%edx),%ebx
    movl  %eax,(%edx)
    movl  %ebx,(%ecx)
    # Finish
    movl  -4(%ebp),%ebx
    movl  %ebp,%esp
    popl  %ebp
    ret
```

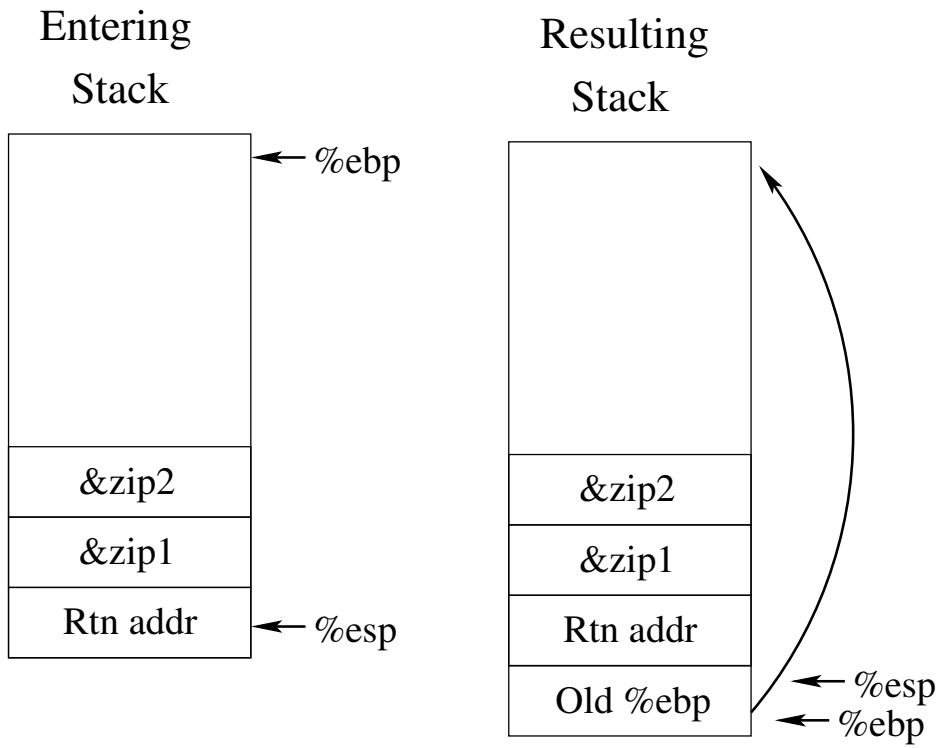
Swap Setup 1

```
swap:
    pushl   %ebp            # <-- save old %ebp
    movl    %esp, %ebp
    pushl   %ebx
```



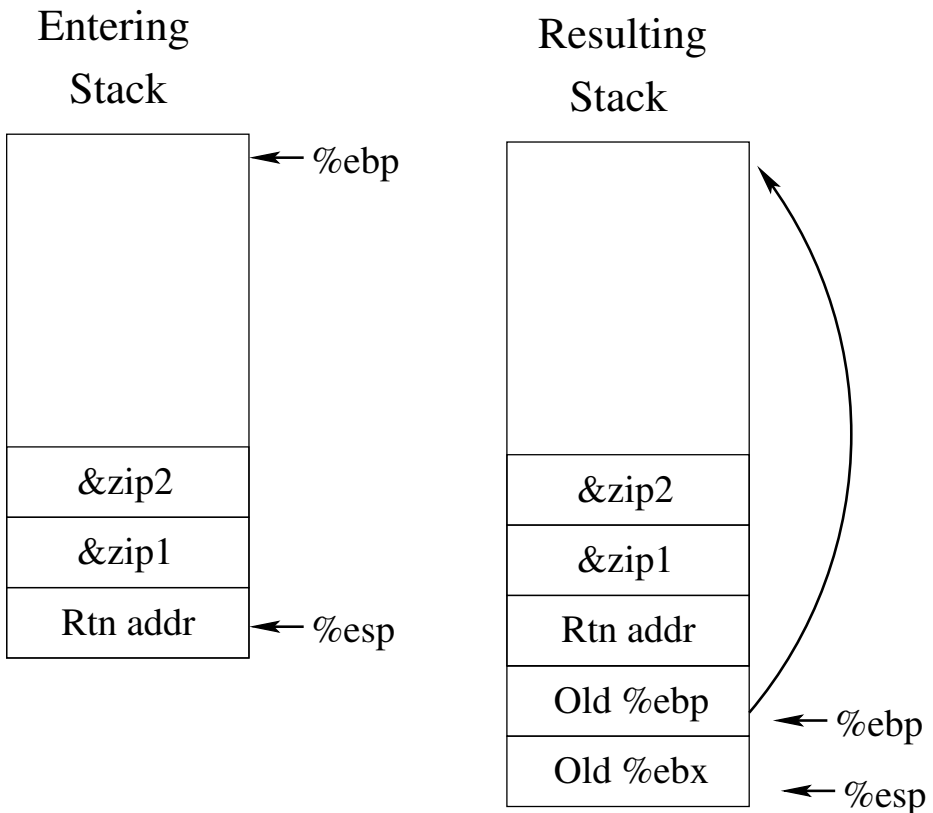
Swap Setup 2

```
swap:
    pushl   %ebp            # save old %ebp
    movl    %esp, %ebp      # <-- establish new frame
    pushl   %ebx
```



Swap Setup 3

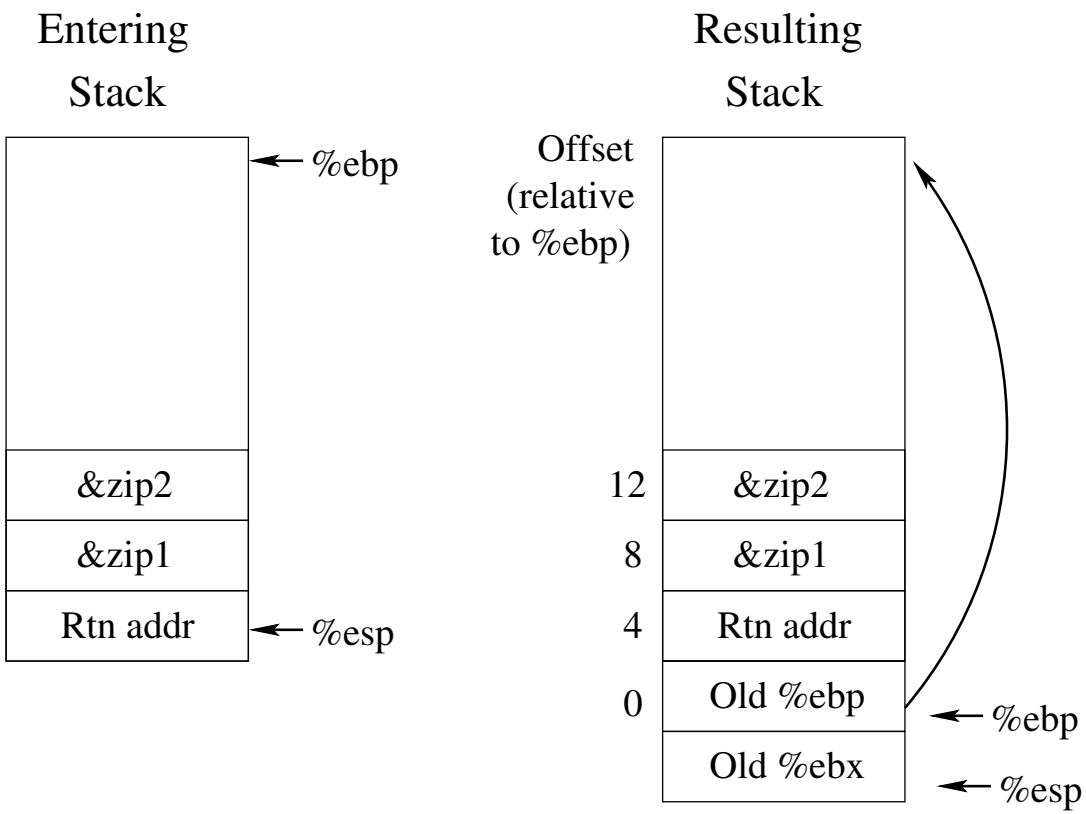
```
swap:
    pushl   %ebp            # save old %ebp
    movl    %esp, %ebp      # establish new frame
    pushl   %ebx            # <-- save reg. %ebx
```



Effect of Swap Setup

Executing the body

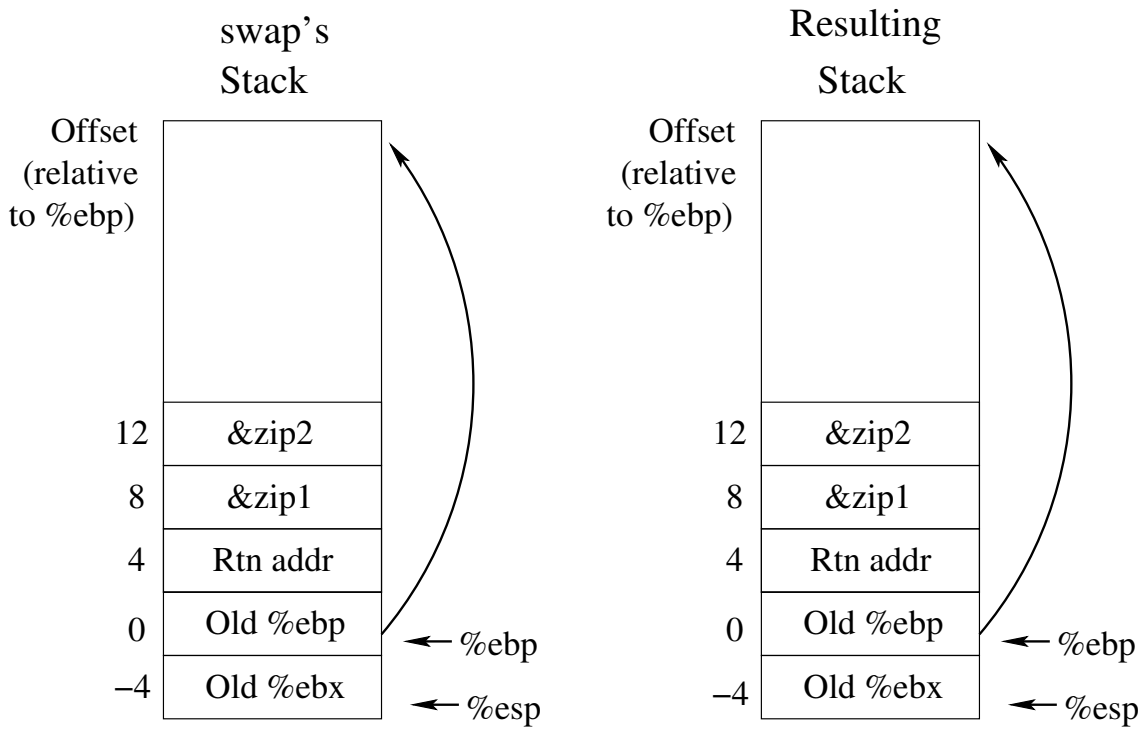
```
movl   12(%ebp),%ecx    # get yp
movl   8(%ebp),%edx     # get xp
...
```



Swap Finish 1

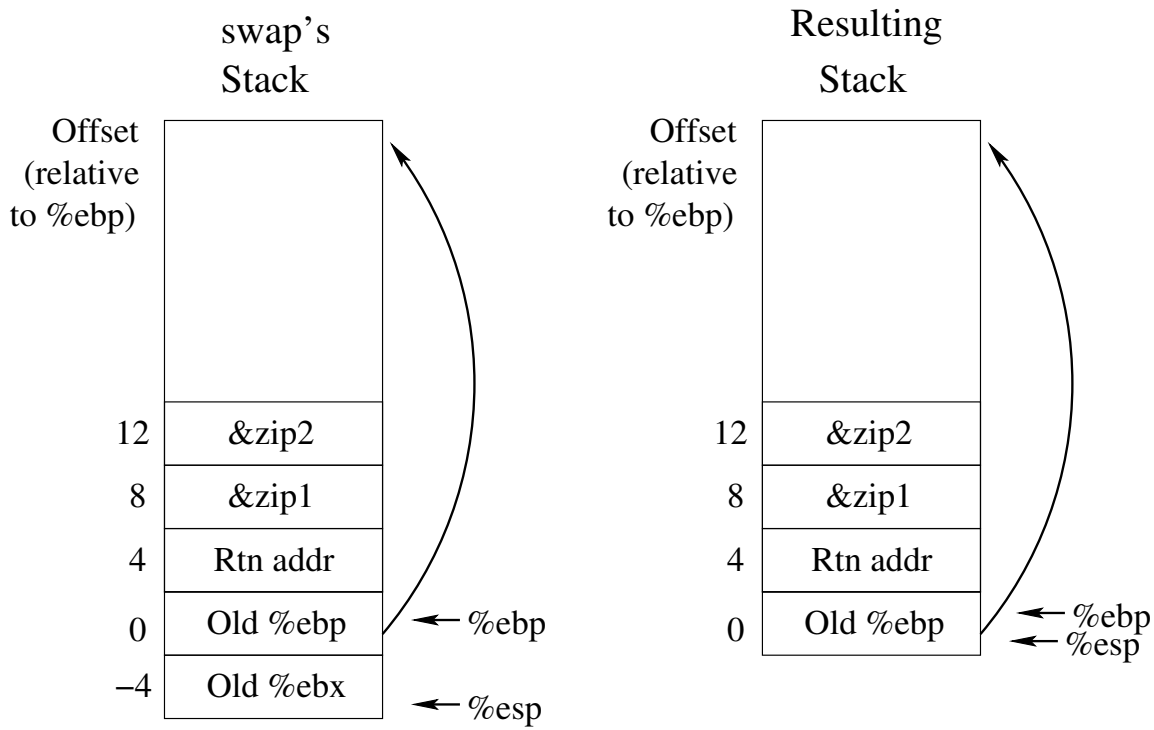
Observation: We've saved and restored register %ebx.

```
movl -4(%ebp),%ebx  # <-- restore saved %ebx
movl %ebp,%esp      #
popl %ebp
ret
```



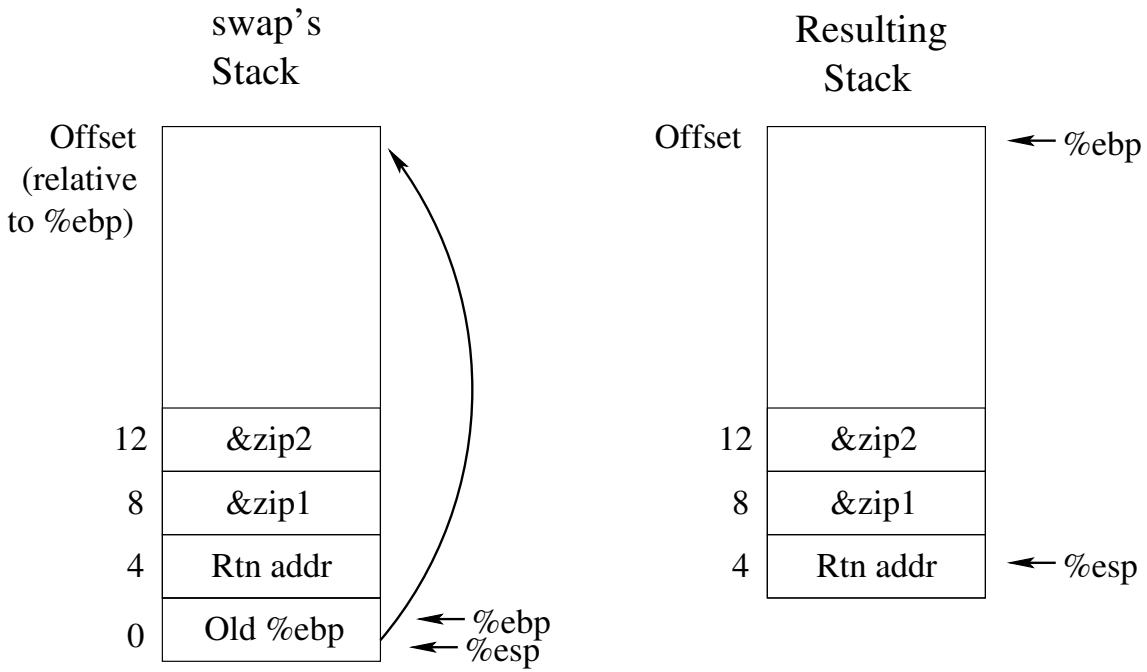
Swap Finish 2

```
movl -4(%ebp),%ebx   # restore saved %ebx
movl %ebp,%esp       # <-- discard callee's frame
popl %ebp
ret
```



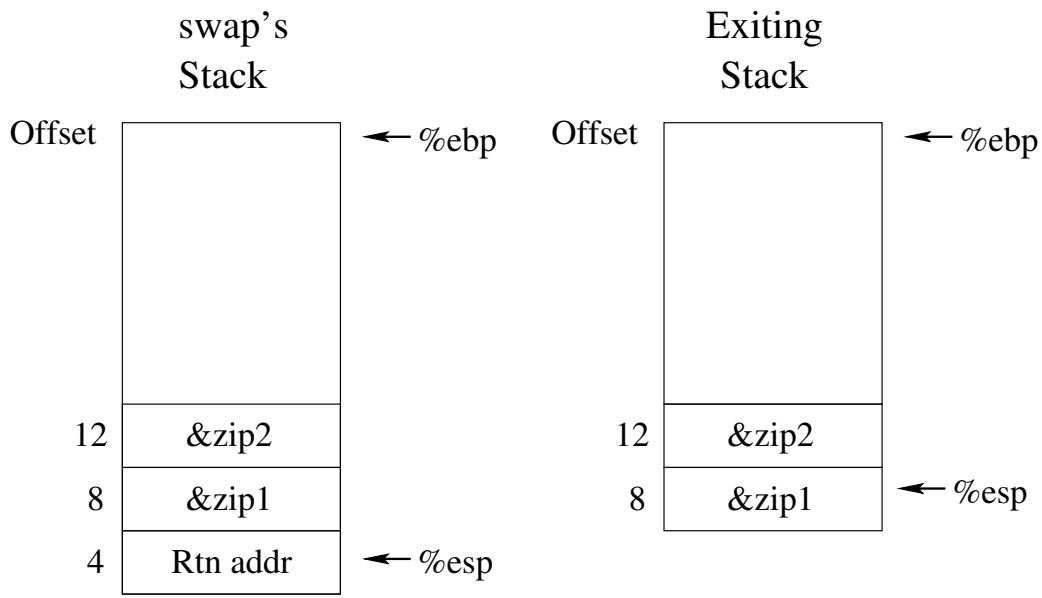
Swap Finish 3

```
movl -4(%ebp),%ebx   # restore saved %ebx
movl %ebp,%esp       # discard callee's frame
popl %ebp            # <-- restore old %ebp
ret
```



Swap Finish 4

```
movl -4(%ebp),%ebx   # restore saved %ebx
movl %ebp,%esp       # discard callee's frame
popl %ebp            # restore old %ebp
ret                  # <-- return to the caller
```



Note: We saved and restored %ebx, but not %eax, %ecx, or %edx

Register Saving Conventions

When procedure `yoo` calls `who`: `yoo` is the caller, `who` is the callee.

Can some 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 are overwritten by `who`.

Register Saving Conventions

- When procedure `yoo` calls `who`: `yoo` is the caller, `who` is the callee.
- Can some register be used for temporary storage?
- Conventions:
  - "Caller Save" means caller saves temporary in its frame before calling.
  - "Callee Save" means callee saves temporary in its frame before using.

IA32/Linux Register Usage

Integer Registers

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

Reg.	saved by
%eax	caller
%edx	caller
%ecx	caller
%ebx	callee
%esi	callee
%edi	callee
%esp	special
%ebp	special

Recursive Factorial

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

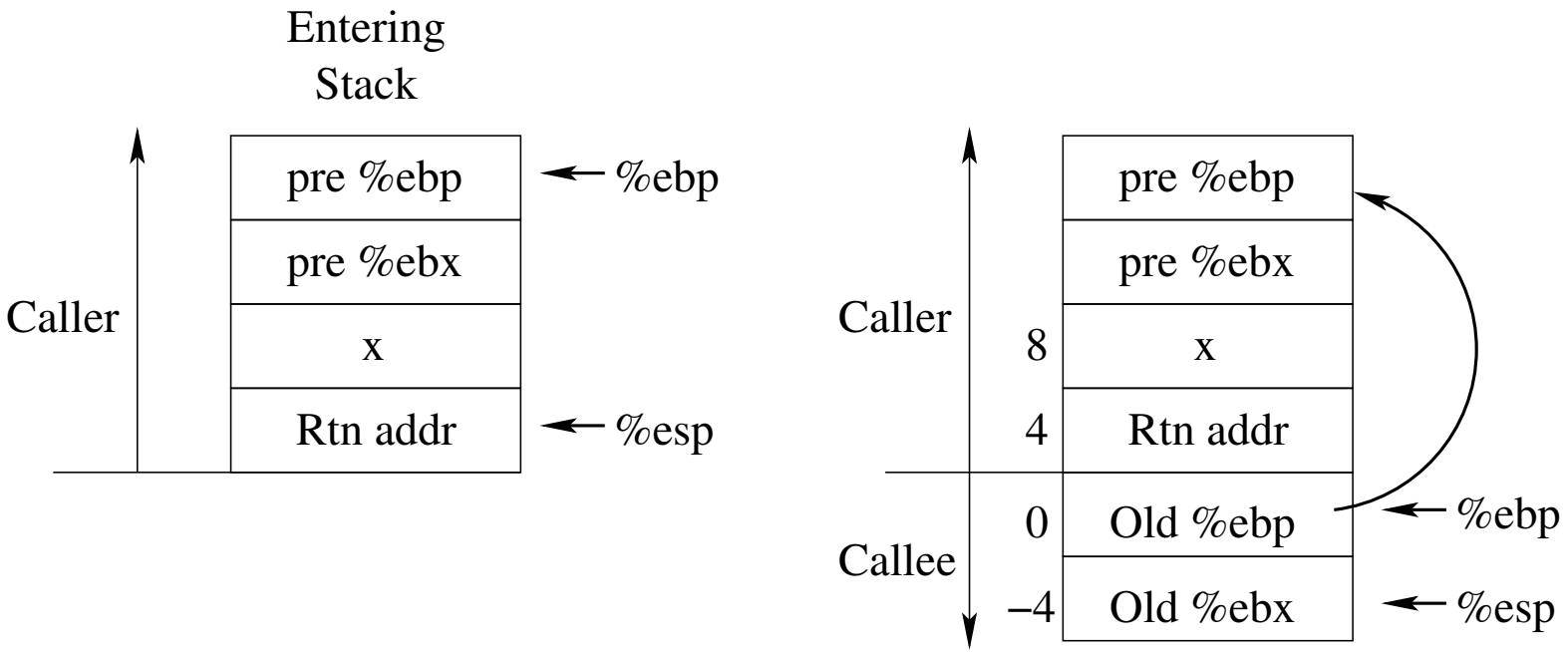
Registers

- %eax is used without first saving.
- %ebx used, but save at beginning and restore at end.

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

Rfact Stack Setup

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



Rfact Body

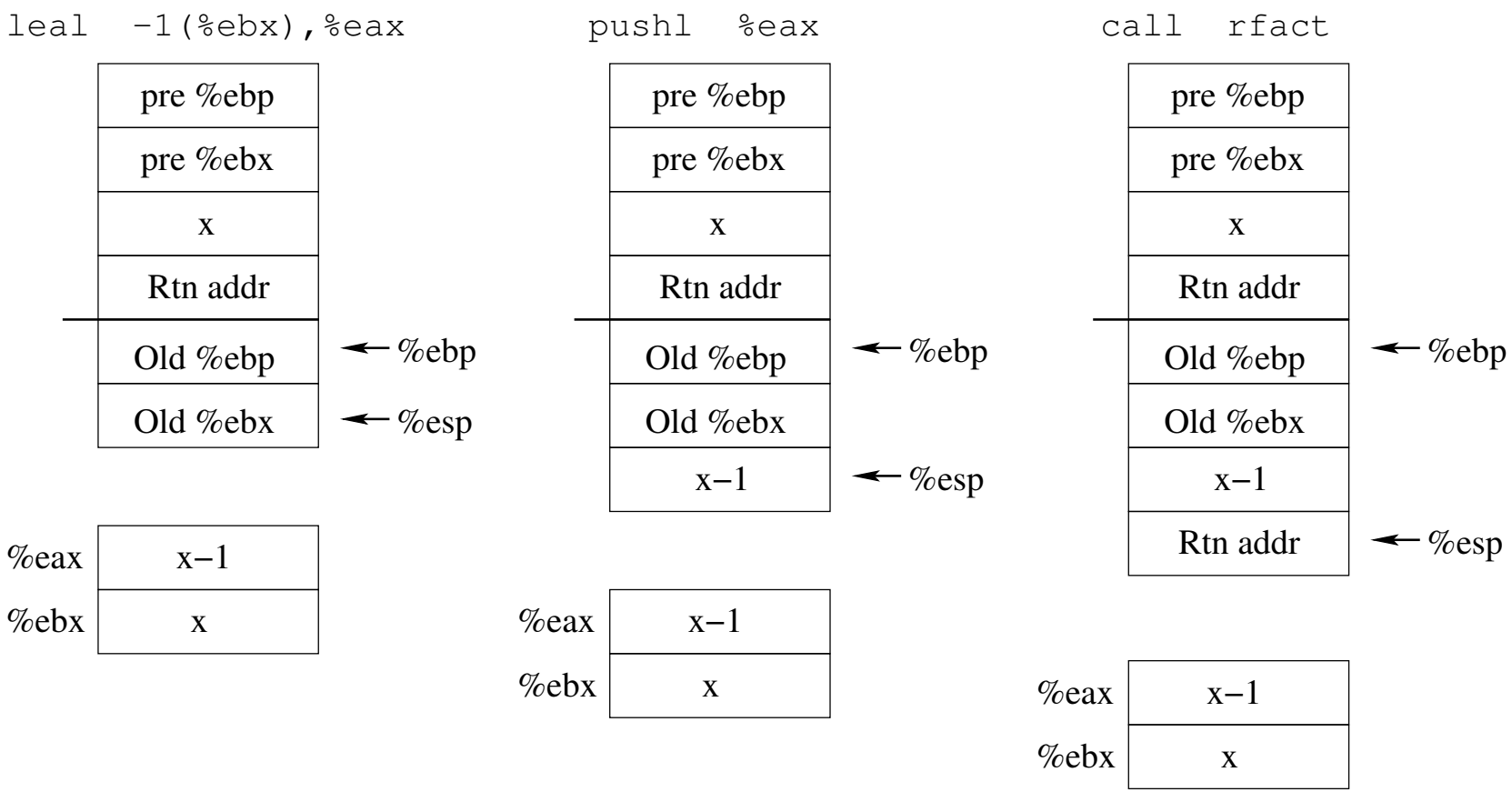
```
    movl    8(%ebp),%ebx      # ebx = x
    cmpl    $1,%ebx           # compare x:1
    jle     .L78              # if <= goto Term
    leal    -1(%ebx),%eax     # eax = x-1
    pushl   %eax              # push x-1
    call    rfact             # rfact(x-1)
    imull   %ebx,%eax         # rval * 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:

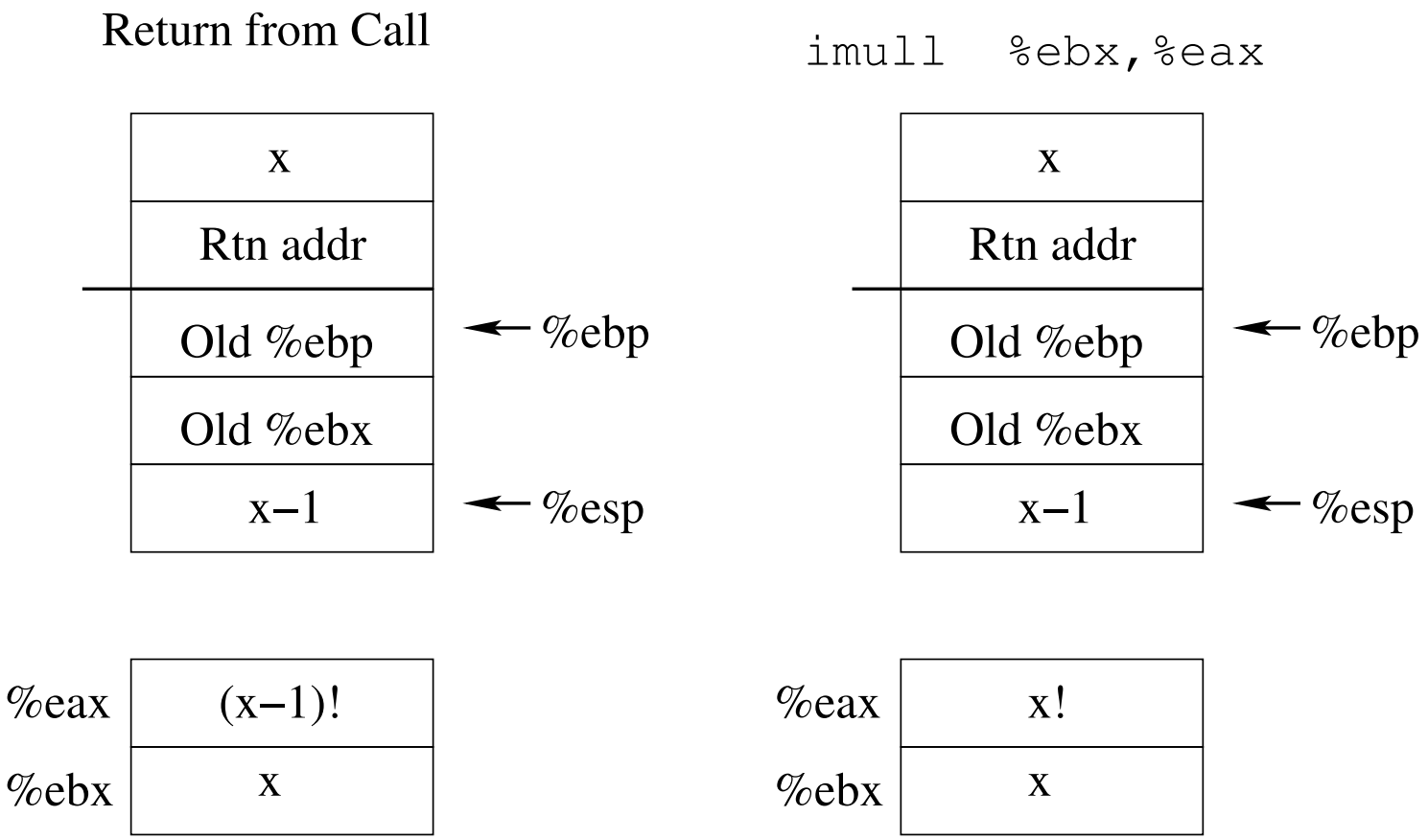
- %ebx: Stored value of x
- %eax:
  - Temp value of x-1
  - Returned value from `rfact( x-1 )`
  - Returned value from this call.

Rfact Recursion



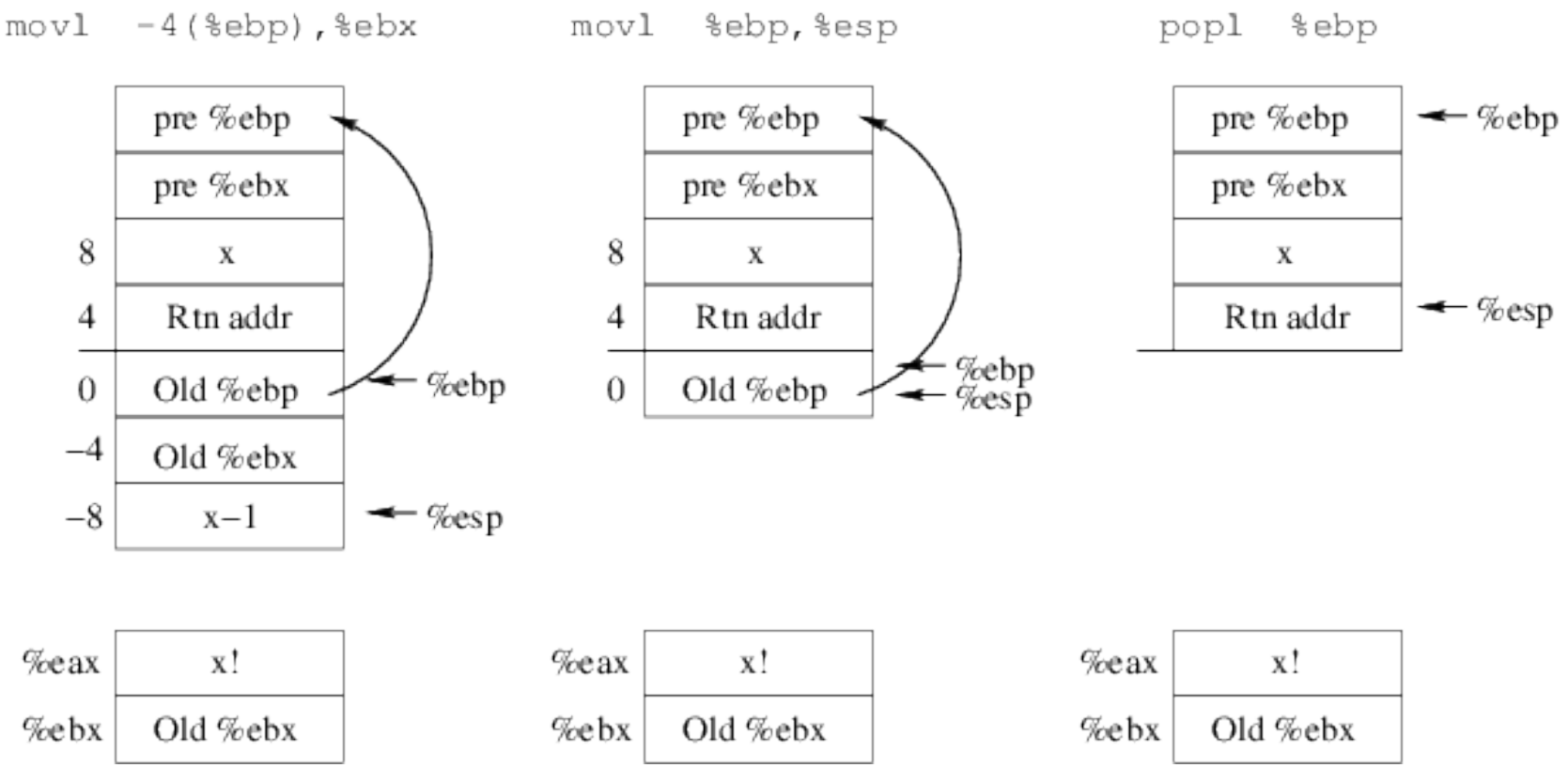
Rfact Result

We assume that `rfact( x-1 )` returns `(x-1)!` in register `%eax`.



Rfact Result

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



Pointer Code

Recursive Procedure

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

We pass a pointer to the update location.

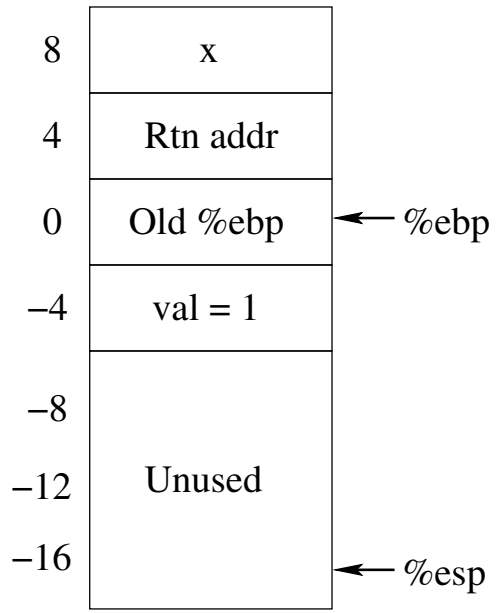
Top Level Call

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

Creating and Initializing Pointers

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

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



Use the stack for local variables.

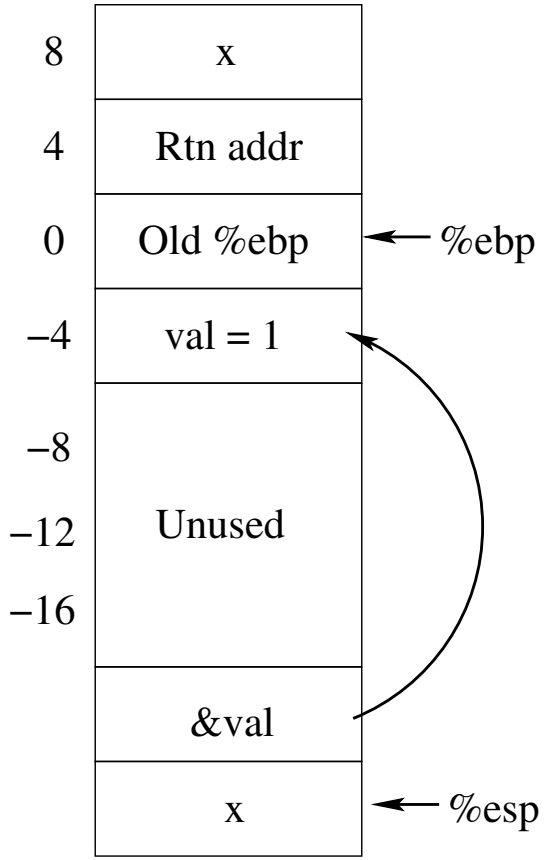
- Variable `val` must be stored on the stack.
- Need to create a pointer for it.
- Compute the pointer as `-4(%ebp)`.
- Push on the stack as the second argument.

Passing Pointer

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

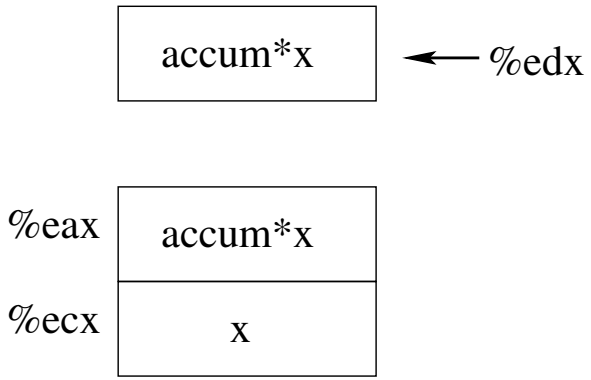
Calling `s_helper` from `sfact`.

```
   leal   -4(%ebp),%eax   # compute &val
   pushl  %eax            # push on stack
   pushl  %edx            # push x
   call   s_helper        # call
   movl   -4(%ebp),%eax   # return val
   ...                    # finish
```



Using Pointer

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




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

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

Summary

The stack makes recursion work.

- Private storage for each *instance* of procedure call.
- Instantiations don't clobber one another.
- Addressing of locals and arguments can be relative to stack positions.
- This all can be managed by stack discipline.
- Procedures always return in reverse order of the calls.

IA32 procedures are combinations of instructions and conventions.

- Call / Ret instructions.
- Register usage conventions.
- Caller / Callee save conventions.
- Stack frame organization conventions.
- Special registers %esp and %ebp.