IA32 Stack

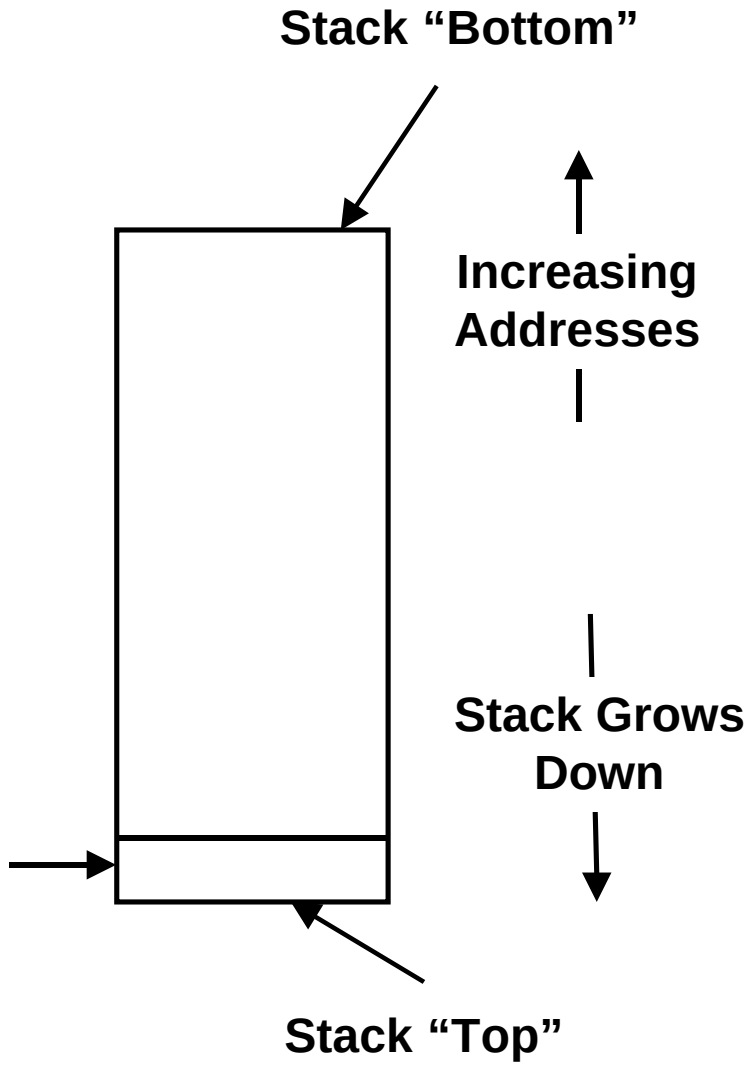
- **Region of memory managed with stack discipline**
- **Register `%esp` indicates lowest allocated position in stack**
  - i.e., address of top element

Pushing

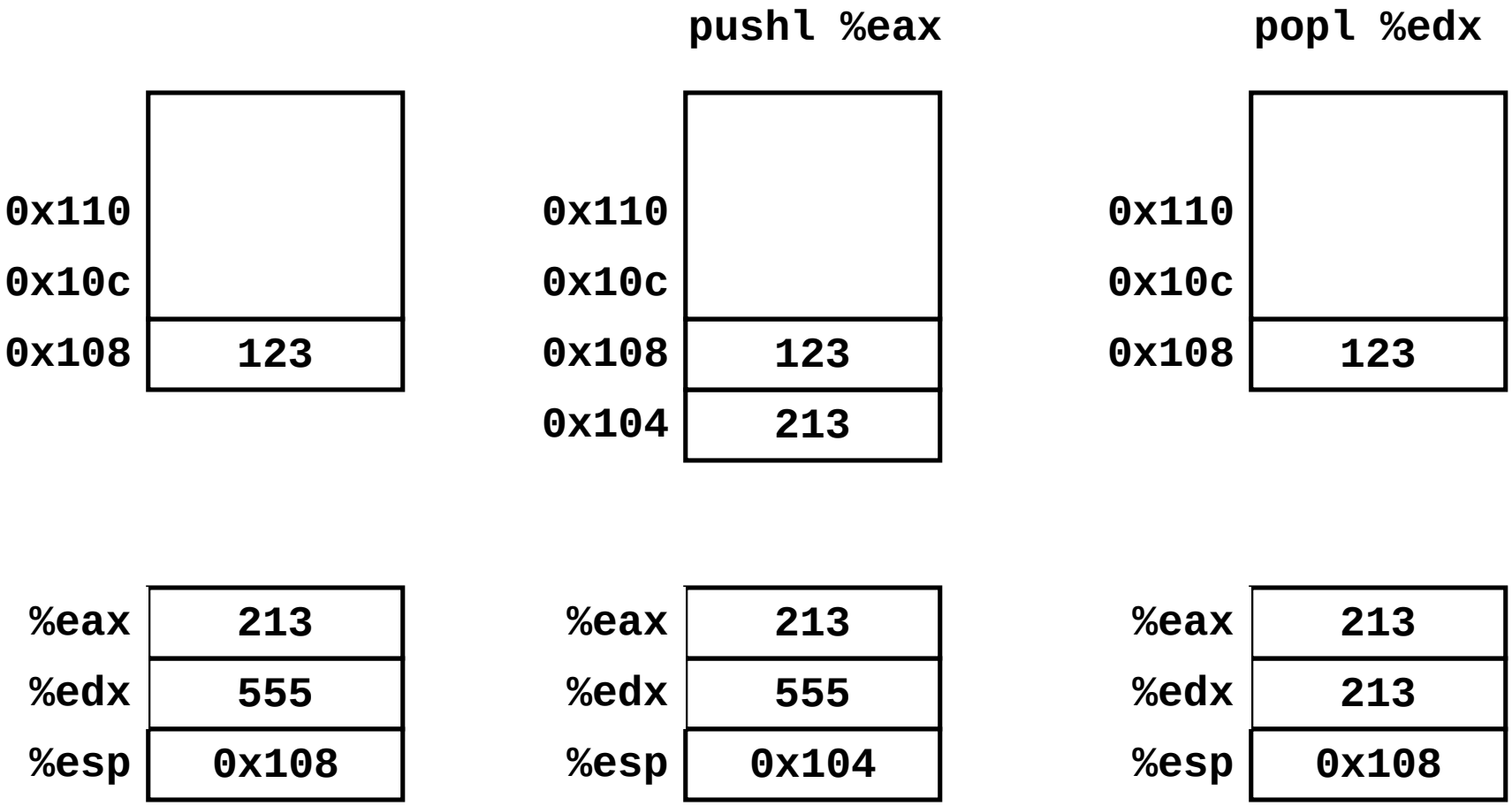
- **`pushl` *Src***
- **Fetch operand at *Src***
- **Decrement `%esp` by 4**
- **Write operand at address given by `%esp`**

Popping

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



Stack Operation Examples



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
```

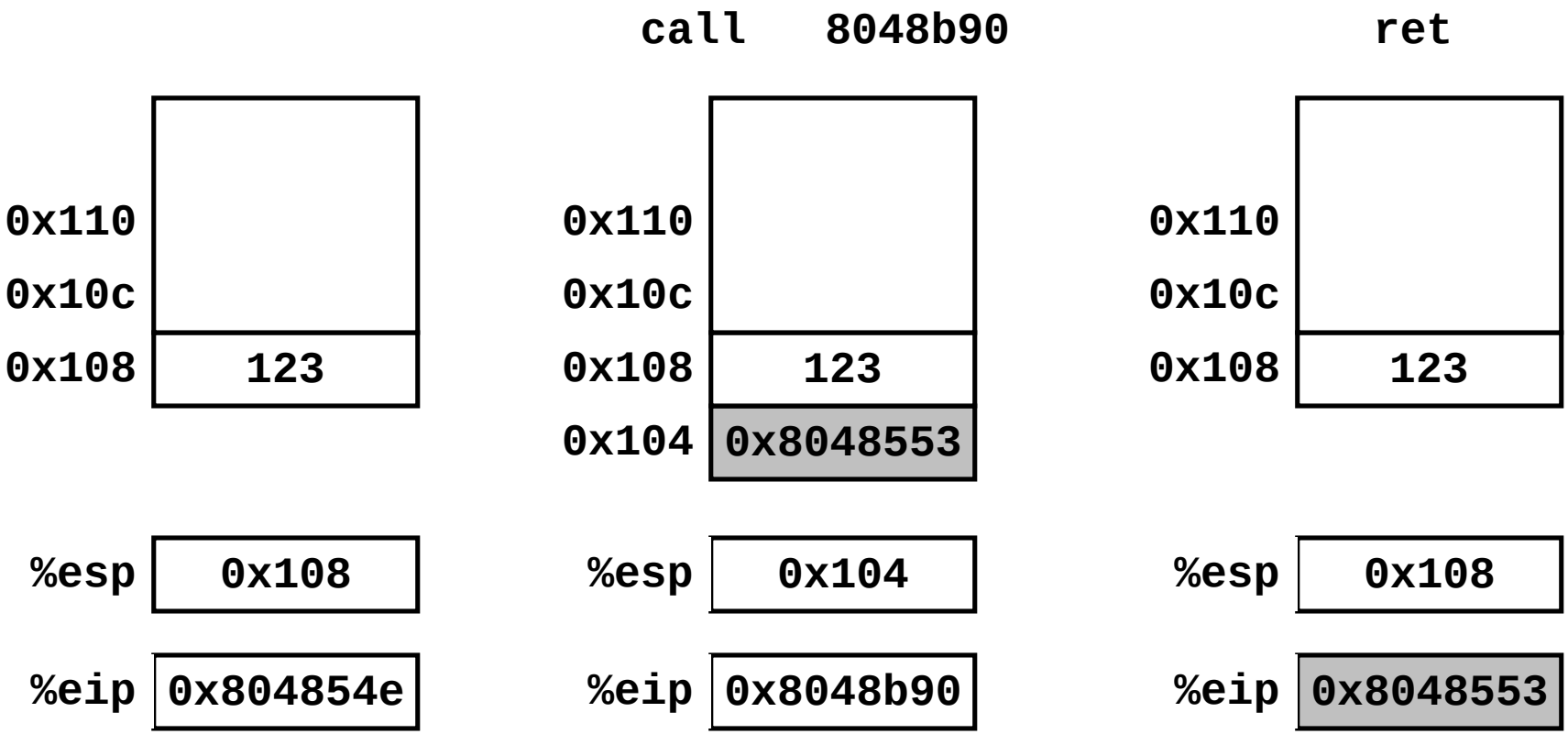
  - Return address = **`0x8048553`**

Procedure return:

- **`ret`** Pop address from stack; Jump to address

Procedure Call / Return Example

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



%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

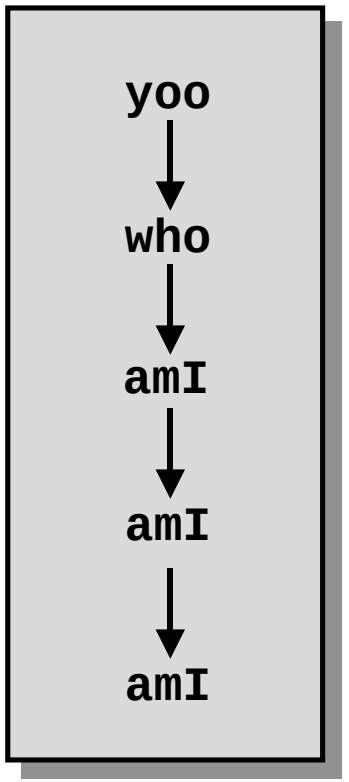
Code Structure

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

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

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

Call Chain



- Procedure `amI` recursive

IA32 Stack Structure

Stack Growth

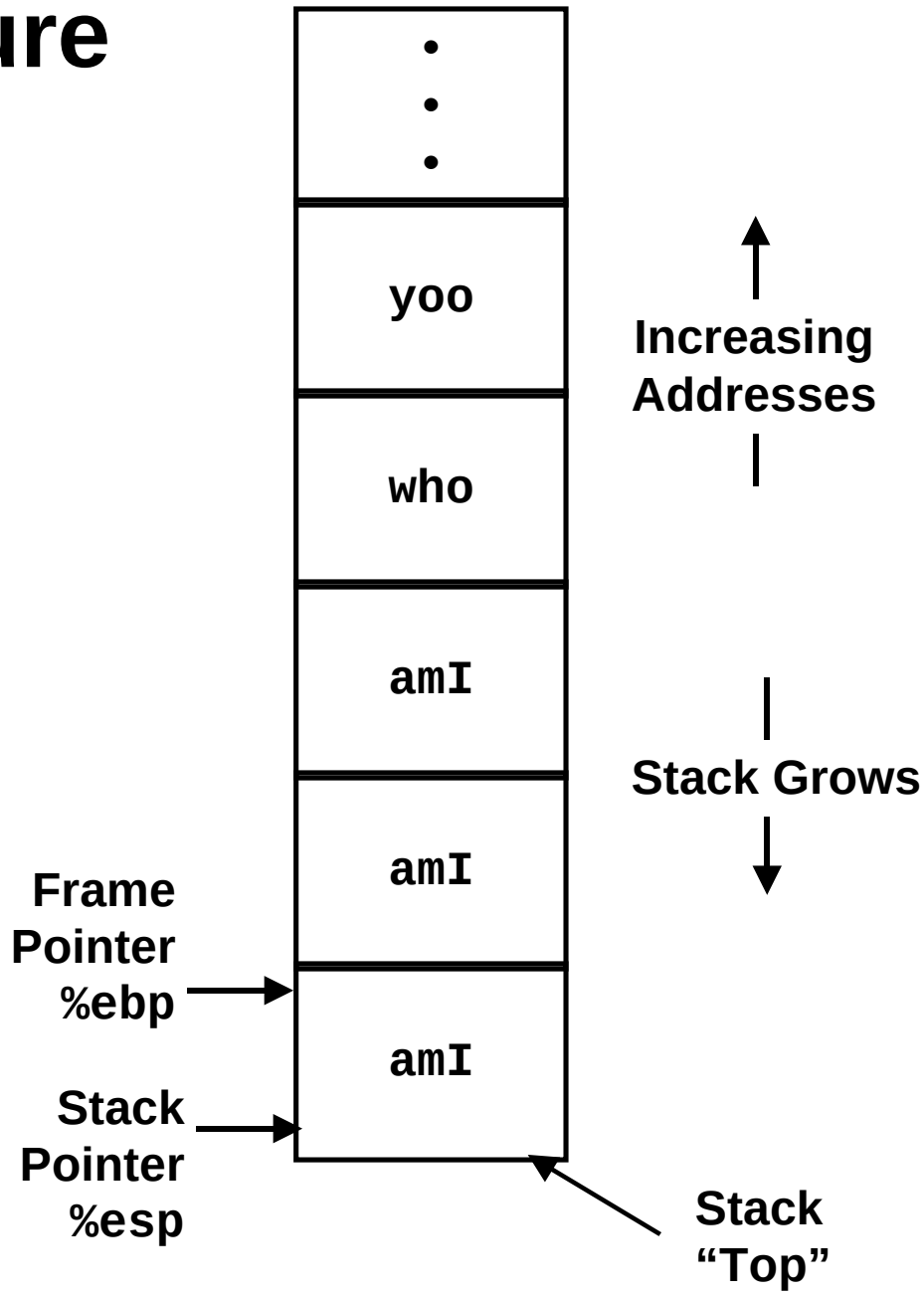
- Toward lower addresses

Stack Pointer

- Address of next available location in stack
- Use register %esp

Frame Pointer

- Start of current stack frame
- Use register %ebp



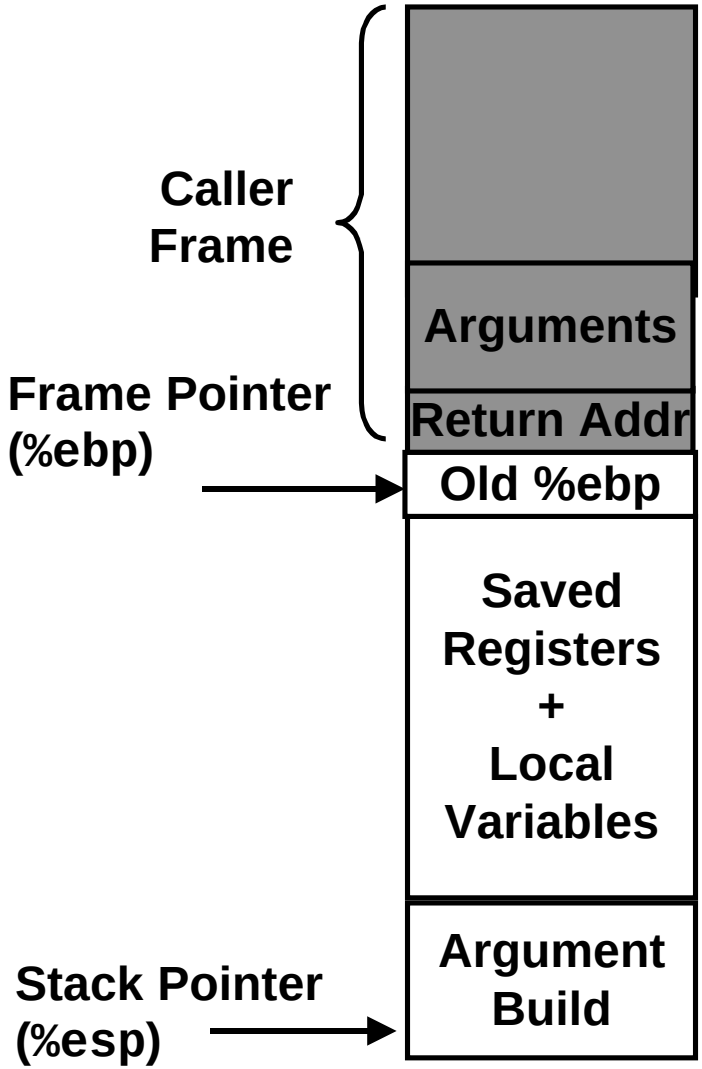
IA32/Linux Stack Frame

Callee Stack Frame ("Top" to Bottom)

- **Parameters for called functions**
- **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**



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**

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

- **Surmised by looking at code examples**

Integer Registers

- **Two have special uses**

  `%ebp`, `%esp`

- **Three managed as callee-save**

  `%ebx`, `%esi`, `%edi`

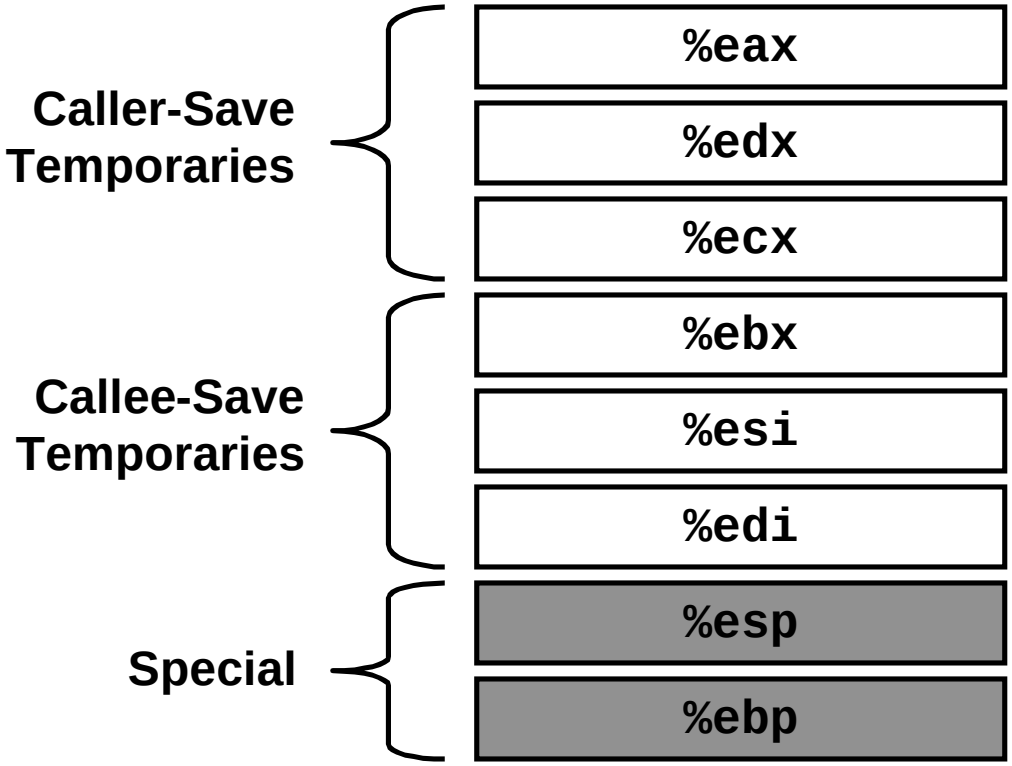
  – Old values saved on stack prior to using

- **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**



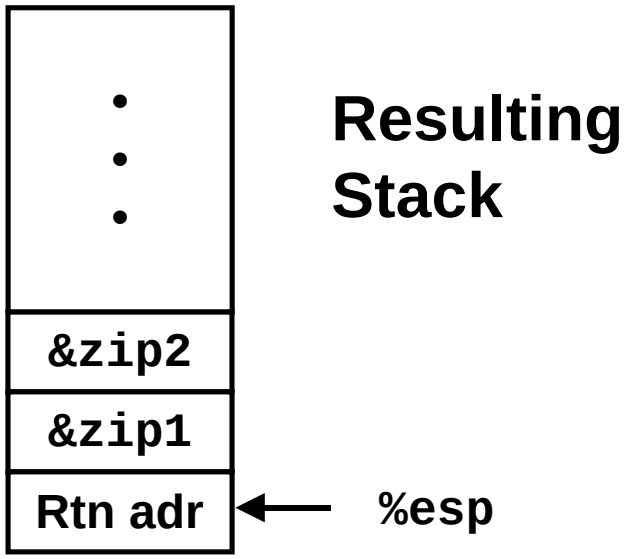
swap

```
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;
}
```

```
call_swap:
    . . .
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    . . .
```



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

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

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

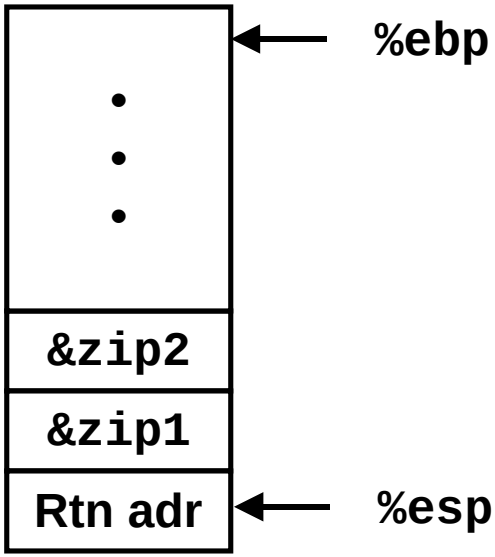
Set Up

Body

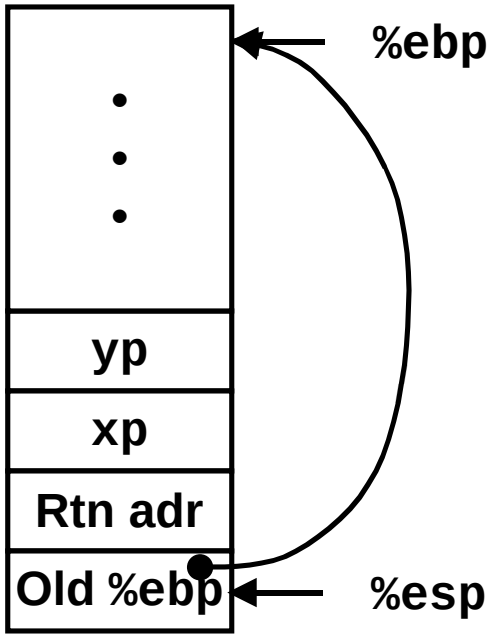
Finish

swap Setup #1

Entering Stack



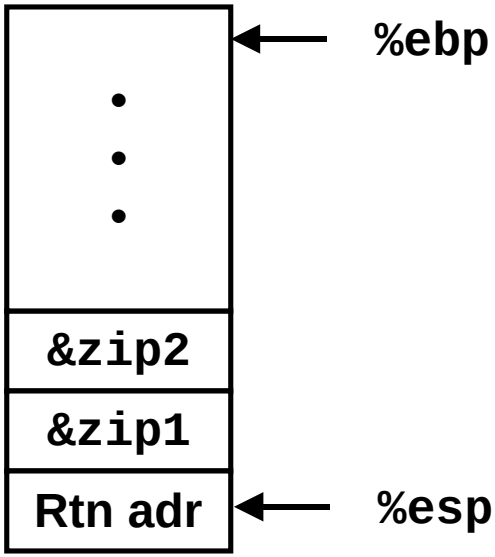
Resulting Stack



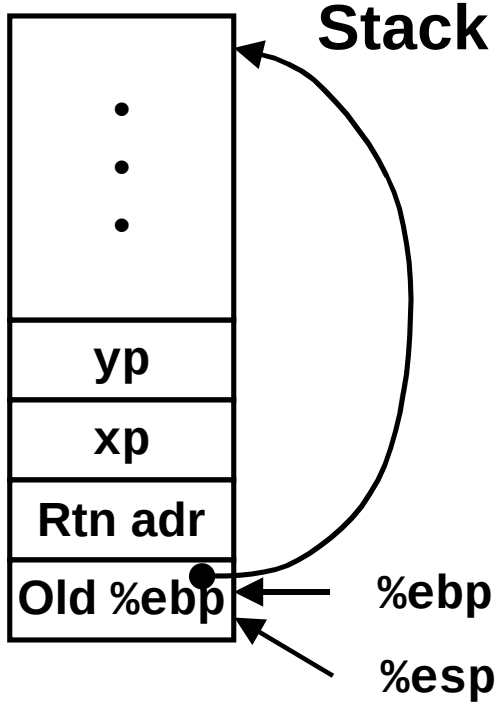

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #2

Entering Stack



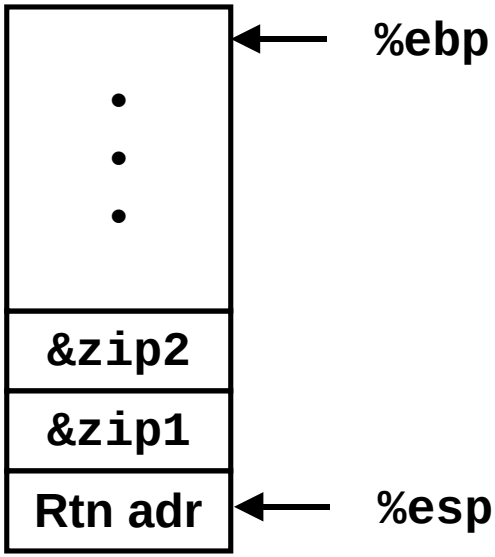
Resulting Stack



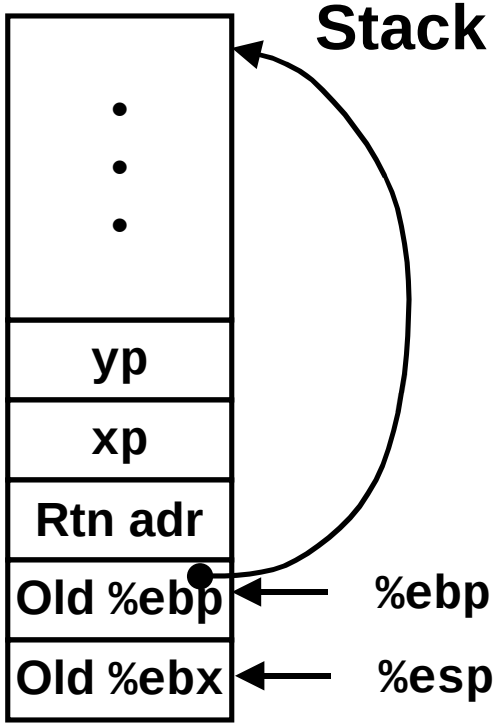

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

swap Setup #3

Entering Stack



Resulting Stack

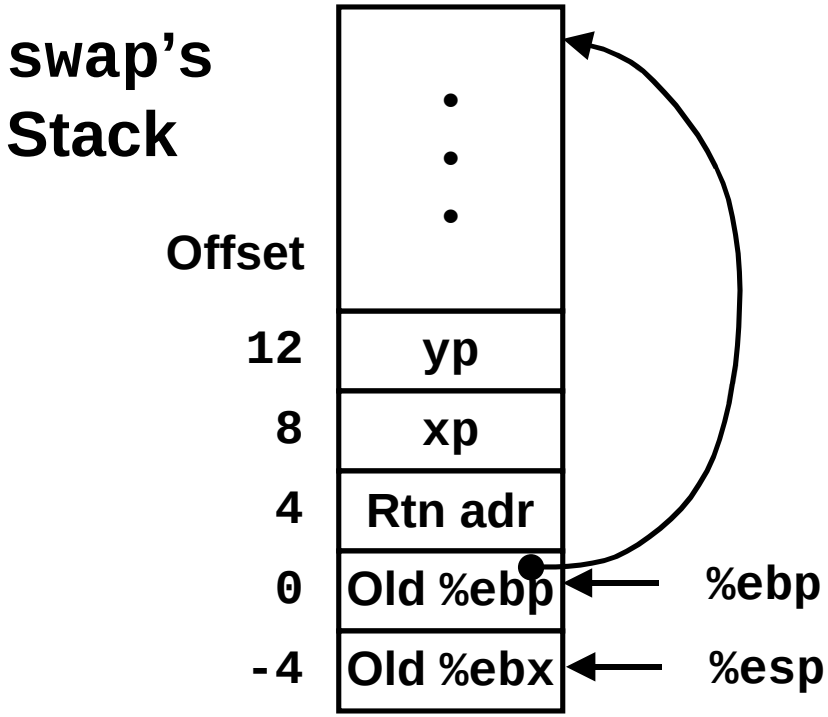


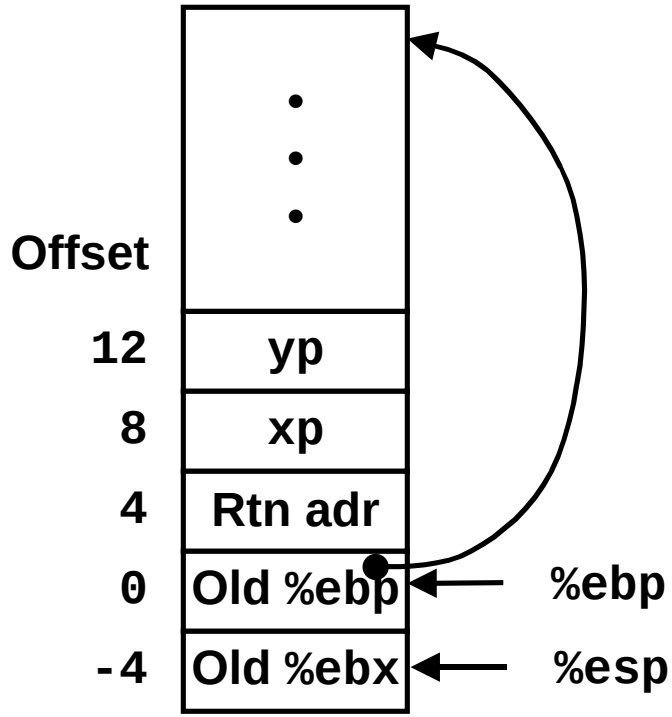

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Effect of swap Setup


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

swap Finish #1



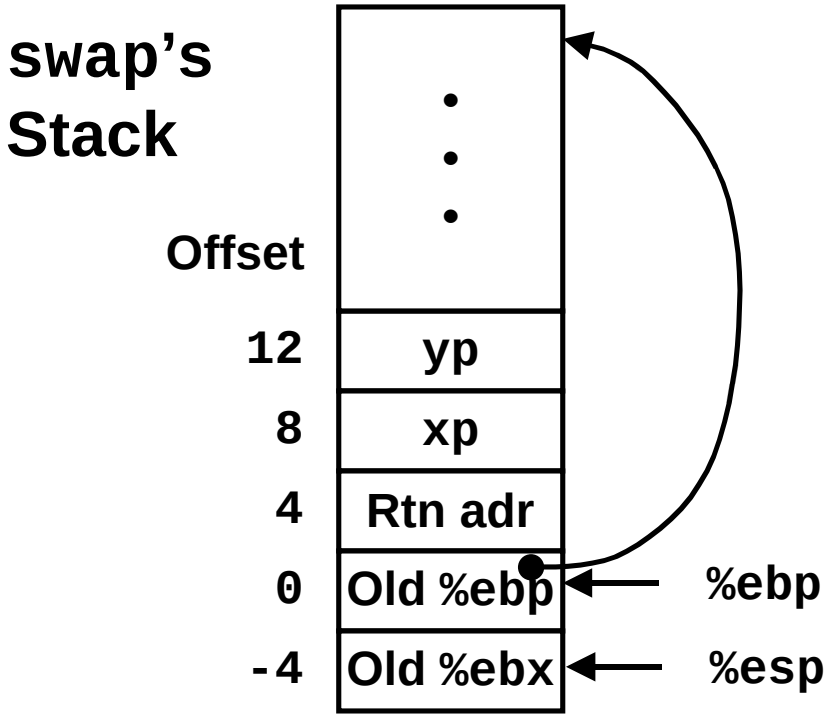


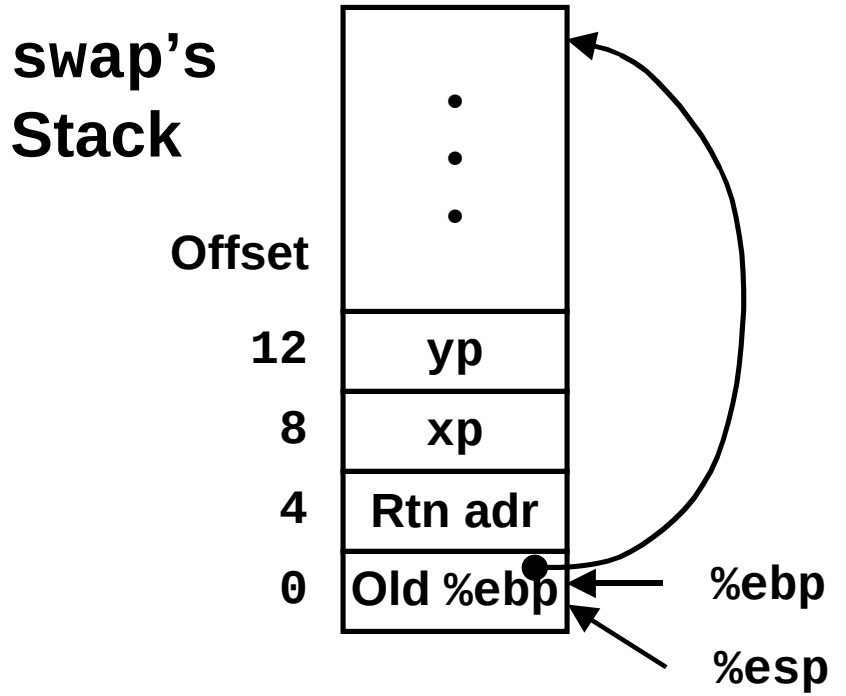

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Observation

- Saved & restored register %ebx

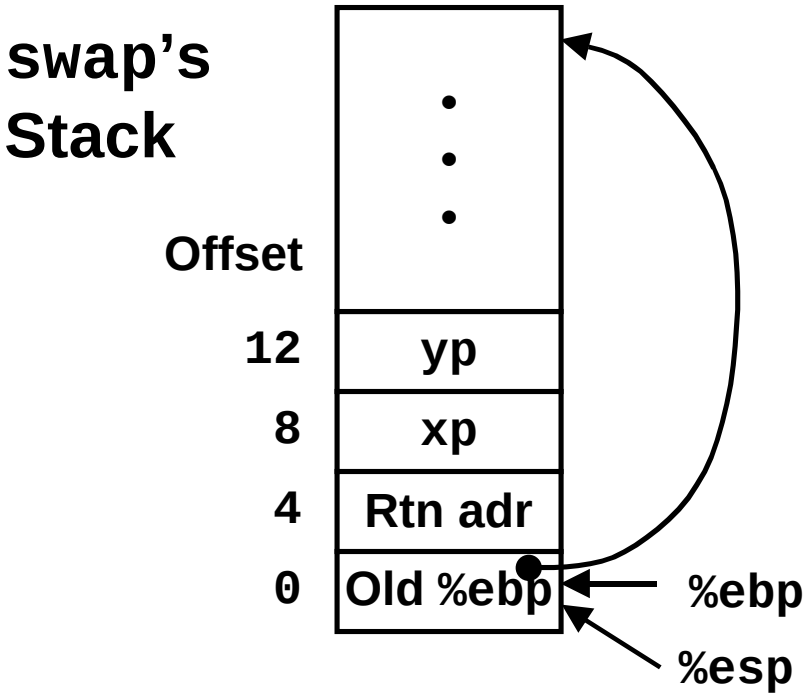
swap Finish #2

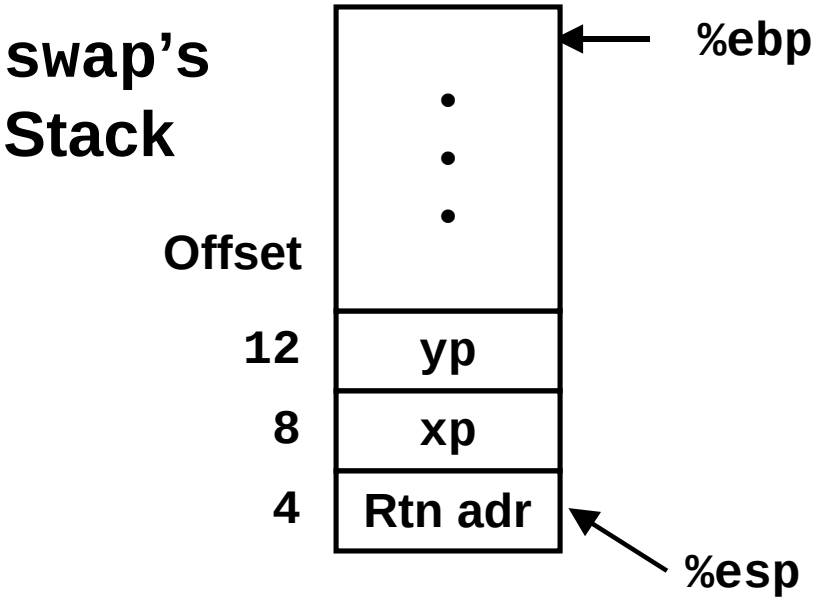





```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

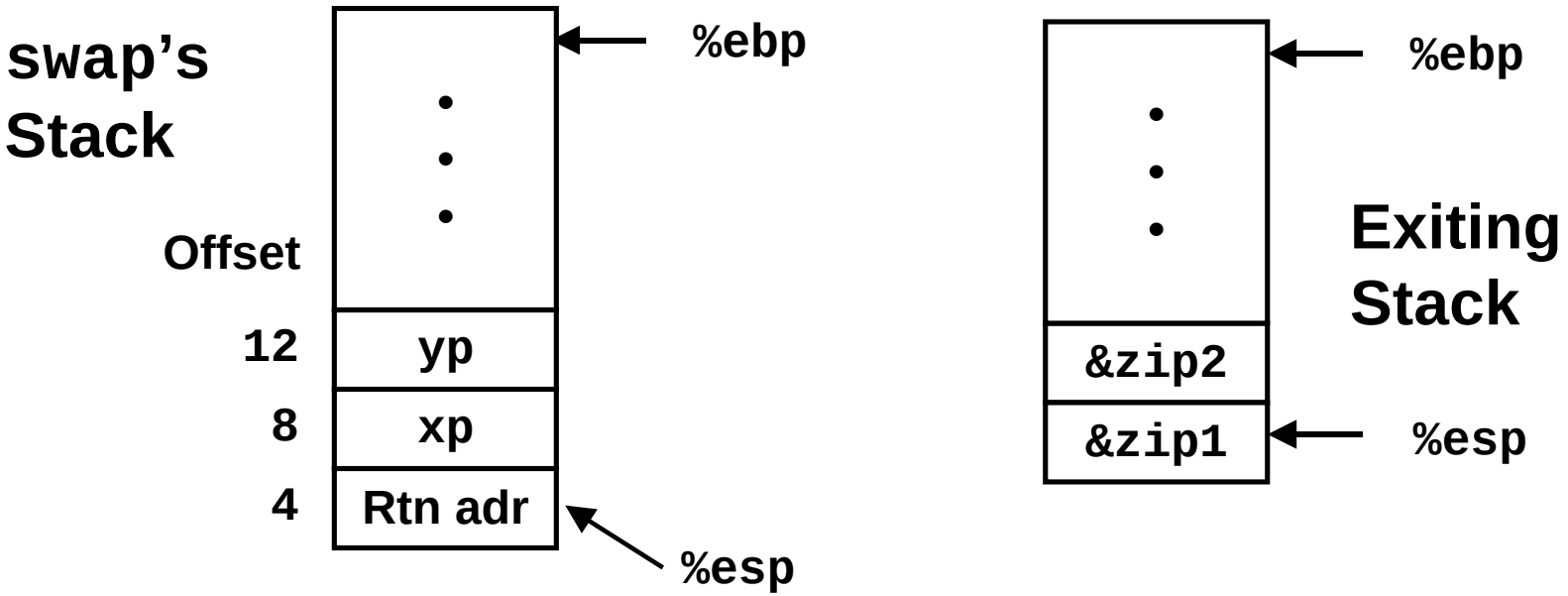
swap Finish #3






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

swap Finish #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