Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
    c.getMPG();
```


Assembly language:

```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

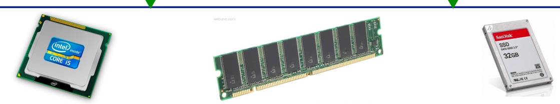
Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

OS:

Windows 8 Mac

Computer system:

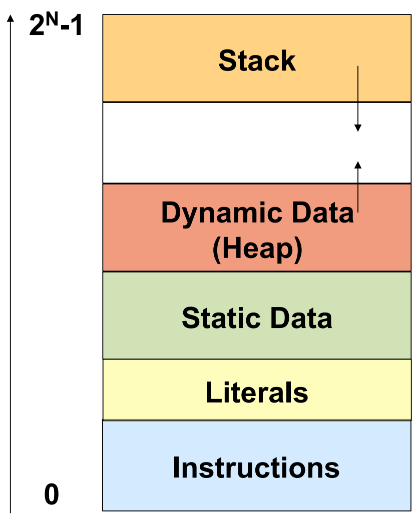


Procedures and Call Stacks

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer these questions, we need a *call stack* …

Memory Layout



local variables; procedure context

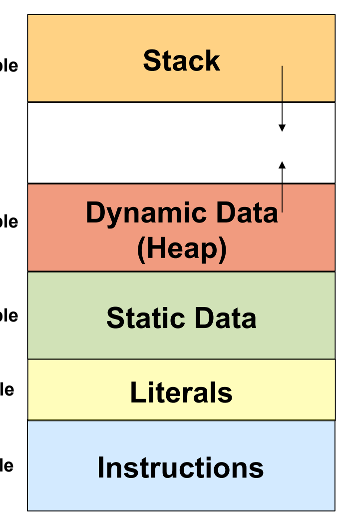
variables allocated with *new* or *malloc*

static variables (including global variables (C))

literals (e.g., "example")

segmentation faults?

Memory Layout

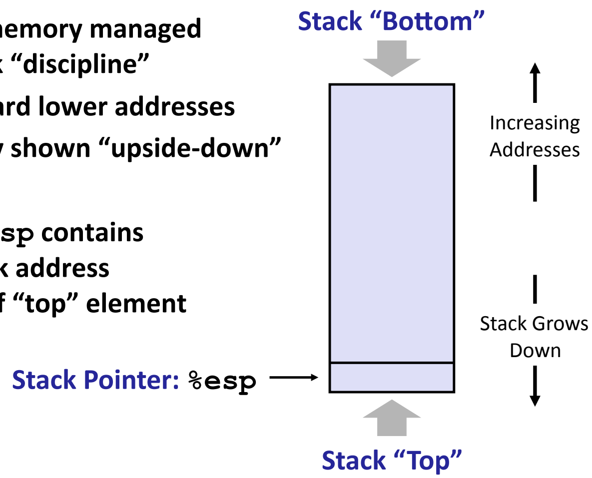


writable; not executable	Stack	Managed "automatically" (by compiler)
writable; not executable	Dynamic Data (Heap)	Managed by programmer
writable; not executable	Static Data	Initialized when process starts
read-only; not executable	Literals	Initialized when process starts
read-only; executable	Instructions	Initialized when process starts

IA32 Call Stack

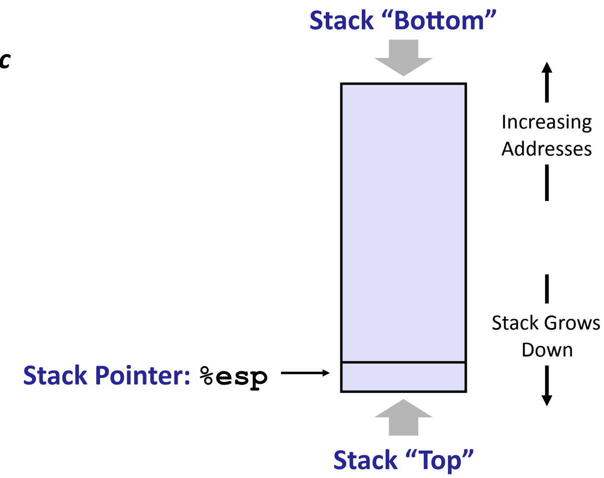
- Region of memory managed with a stack "discipline"
- Grows toward lower addresses
- Customarily shown "upside-down"

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



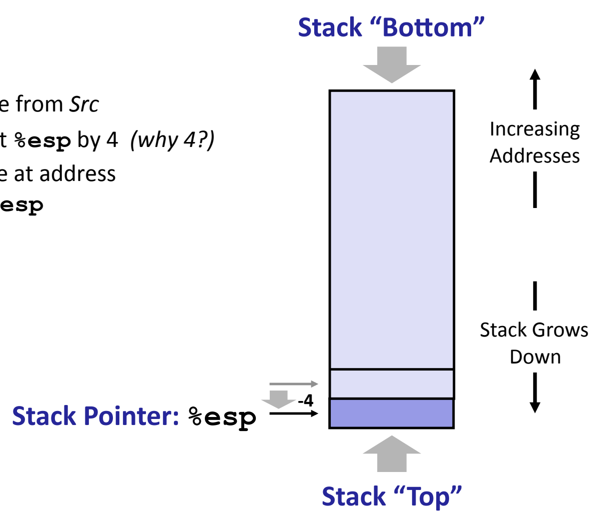
IA32 Call Stack: Push

- `pushl` *Src*



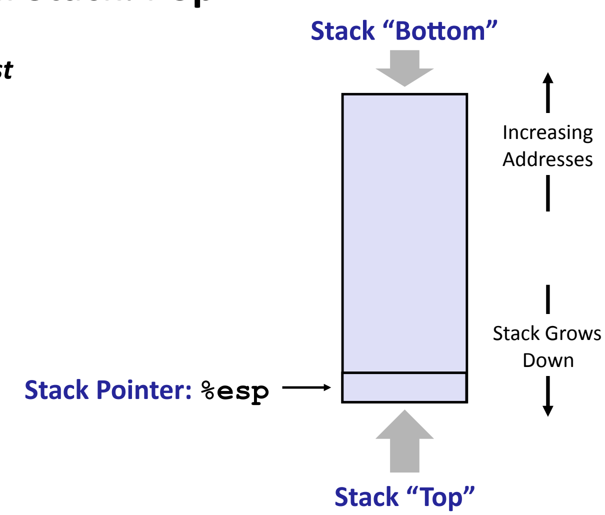
IA32 Call Stack: Push

- `pushl` *Src*
  - Fetch value from *Src*
  - Decrement `%esp` by 4 *(why 4?)*
  - Store value at address given by `%esp`



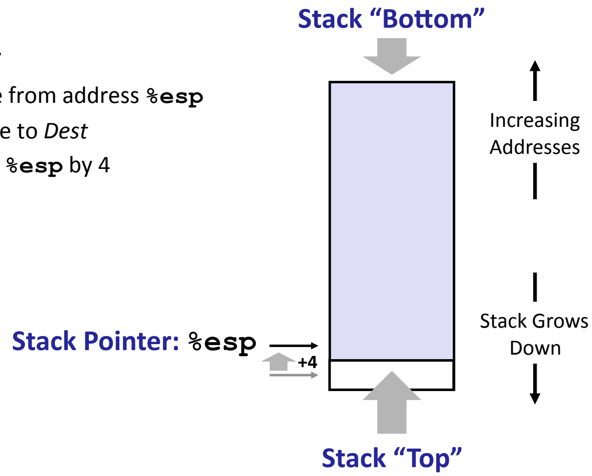
IA32 Call Stack: Pop

- `popl` *Dest*



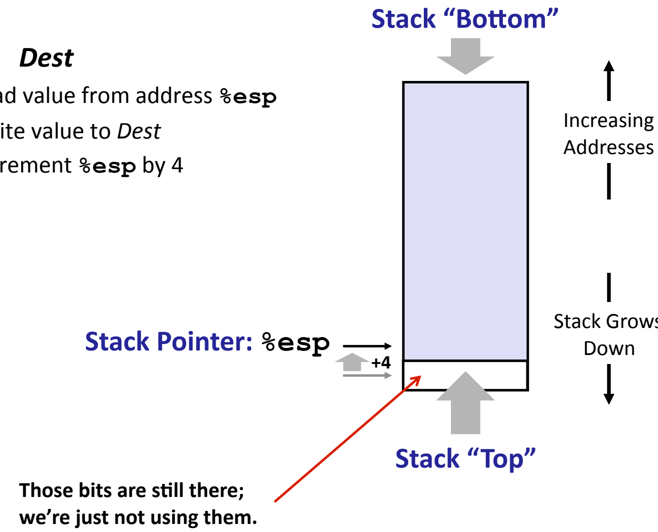
IA32 Call Stack: Pop

- **`popl` *Dest***
  - Load value from address **`%esp`**
  - Write value to *Dest*
  - Increment **`%esp`** by 4

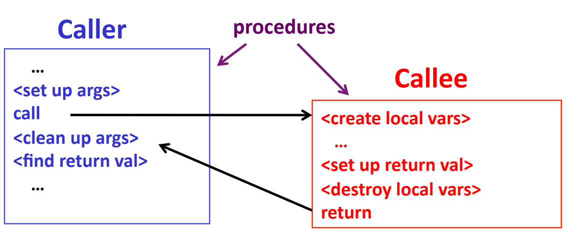


IA32 Call Stack: Pop

- **`popl` *Dest***
  - Load value from address **`%esp`**
  - Write value to *Dest*
  - Increment **`%esp`** by 4

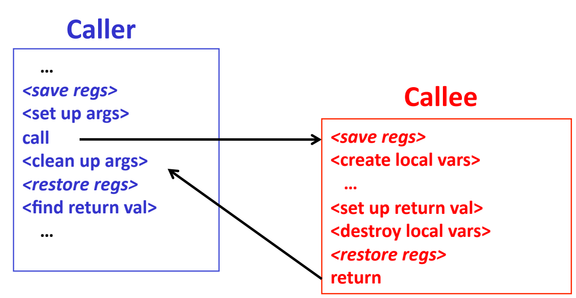


Procedure Call Overview



- **Callee must know where to find args**
- **Callee must know where to find "return address"**
- **Caller must know where to find return val**
- **Caller and Callee run on same CPU → use the same registers**
  - So how do we deal with register reuse?

Procedure Call Overview



- **The convention of where to leave/find things is called the calling convention (or procedure call linkage).**
  - Details vary between systems
  - We will see the convention for IA32/Linux in detail
  - What could happen if our program didn't follow these conventions?

Procedure Control Flow

- **Use stack to support procedure call and return**
- **Procedure call:** `call label`
  - Push return address on stack (*why?, and which exact address?*)
  - Jump to `label`

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 after `call`
  - Example from disassembly:

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

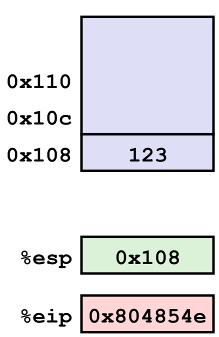
  - Return address = `0x8048553`
- **Procedure return:** `ret`
  - Pop return address from stack
  - Jump to address

next instruction just happens to be a push could be anything

Procedure Call Example

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

`call 8048b90`

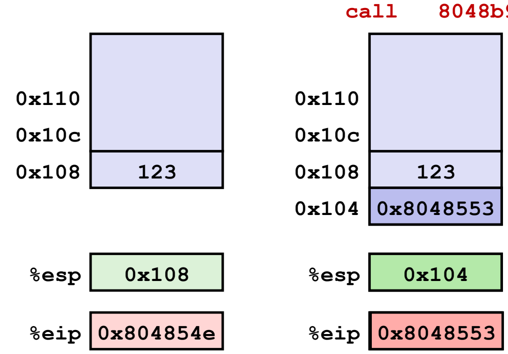


`%eip`: *program counter*

Procedure Call Example

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

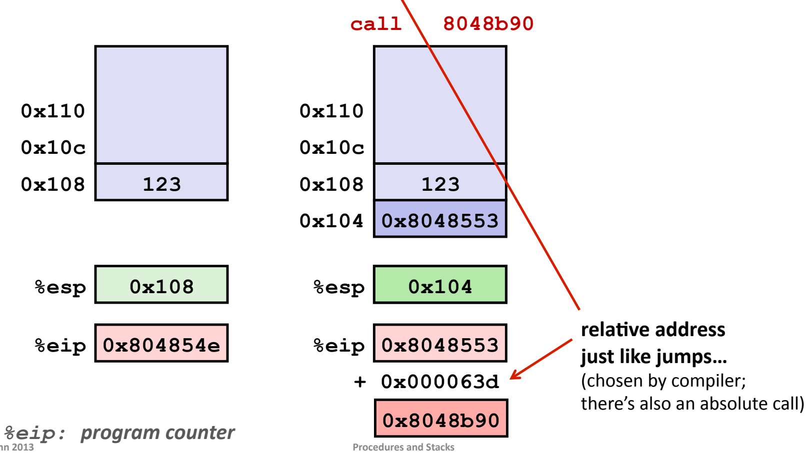
`call 8048b90`



`%eip`: *program counter*

Procedure Call Example

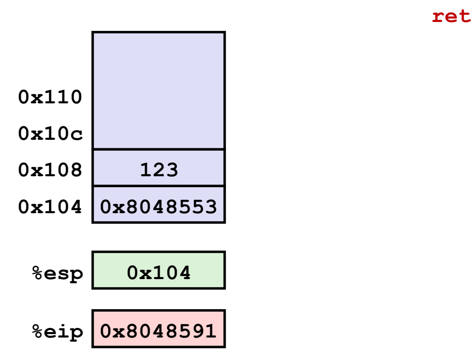
```
804854e:     e8 3d 06 00 00        call   8048b90 <main>
8048553:     50                    pushl  %eax
```



%eip: program counter

Procedure Return Example

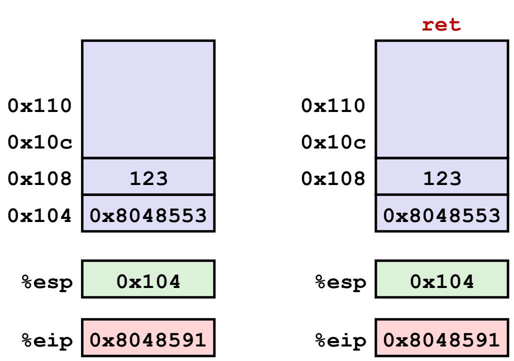
```
8048591:     c3                    ret
```



%eip: program counter

Procedure Return Example

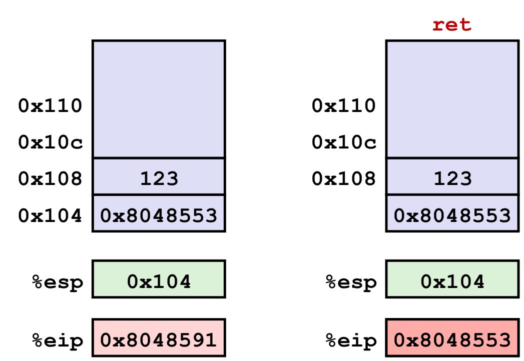
```
8048591:     c3                    ret
```



%eip: program counter

Procedure Return Example

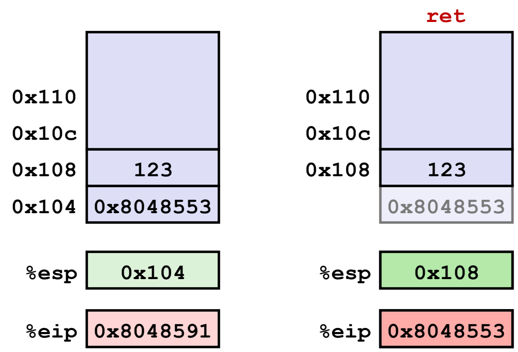
```
8048591:     c3                    ret
```



%eip: program counter

Procedure Return Example

```
8048591:     c3                    ret
```



%eip: program counter

Stack-Based Languages

- **Languages that support recursion**
  - e.g., C, Java, most modern languages
  - 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

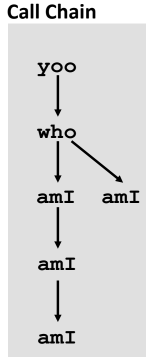
Call Chain Example

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

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

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

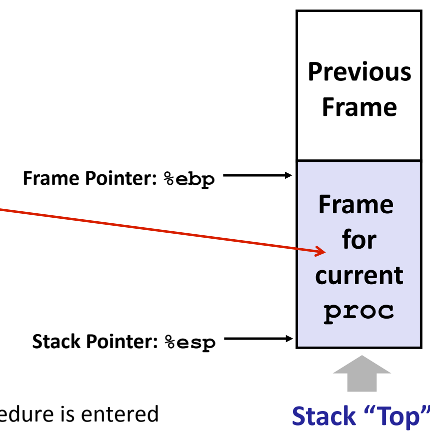
Example Call Chain



Procedure amI is recursive (calls itself)

Stack Frames

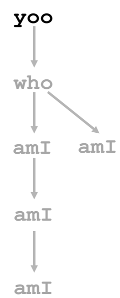
- **Contents**
  - Local variables
  - Function arguments
  - Return information
  - Temporary space

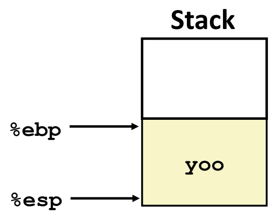


- **Management**
  - Space allocated when procedure is entered
    - "Set-up" code
  - Space deallocated upon return
    - "Finish" code

Example

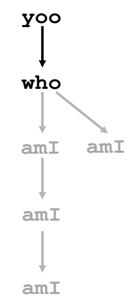
```
yoo (…)
{
    •
    •
    who () ;
    •
    •
}
```

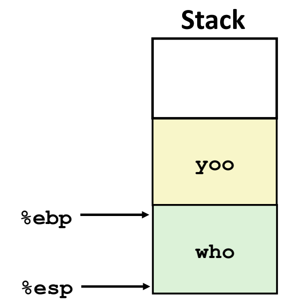




Example

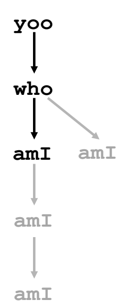
```
who (…)
{
    • • •
    amI () ;
    • • •
    amI () ;
    • • •
}
```

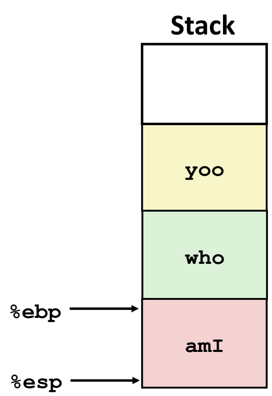




Example

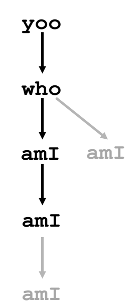
```
amI (…)
{
    •
    •
    amI () ;
    •
    •
}
```

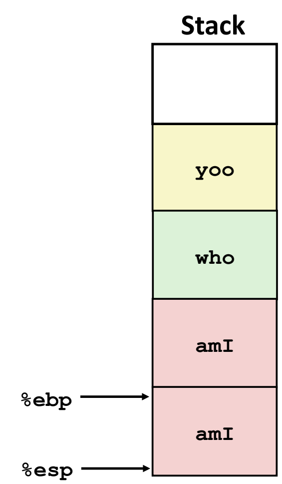




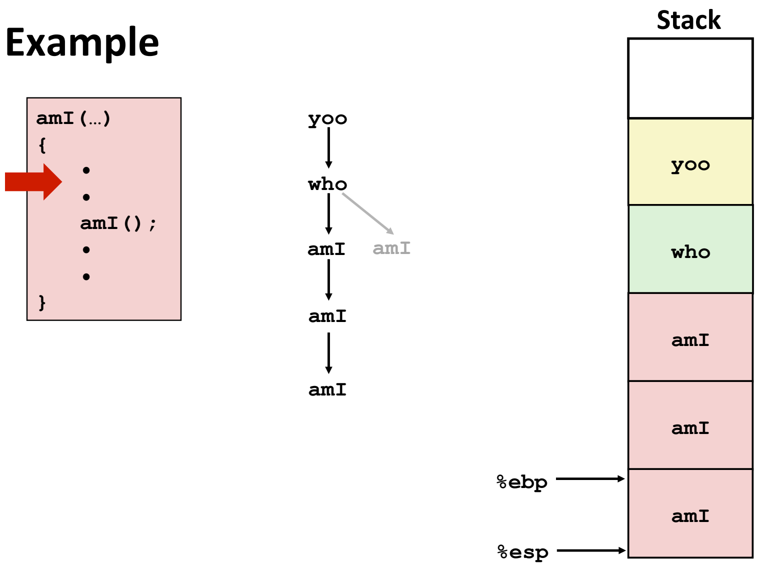
Example

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

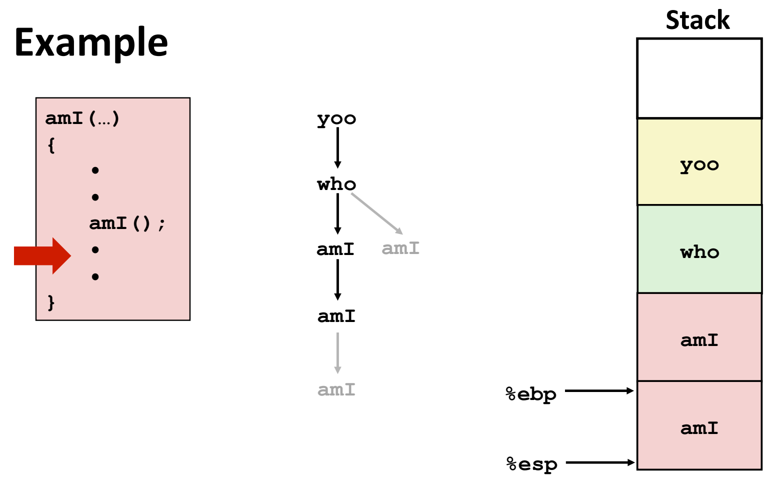




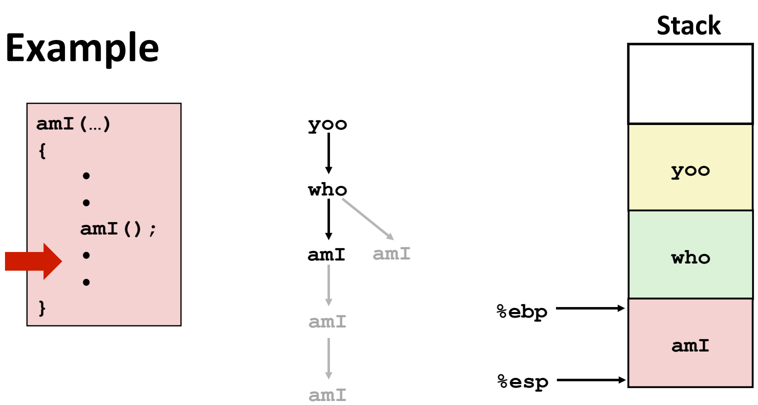
Example



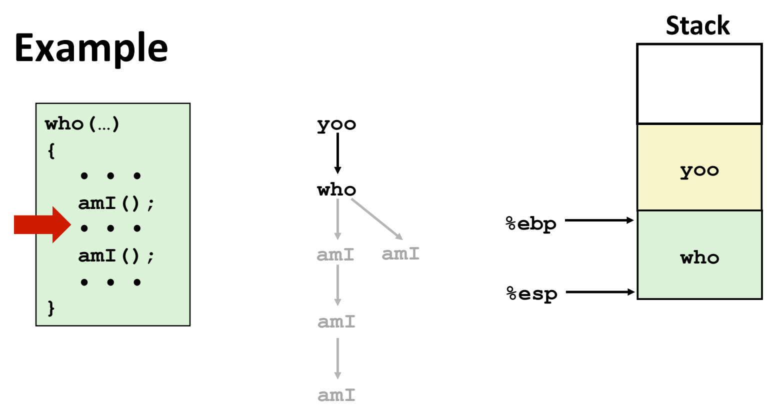
Example



Example

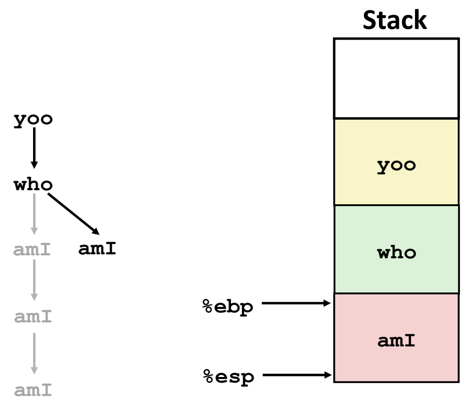


Example



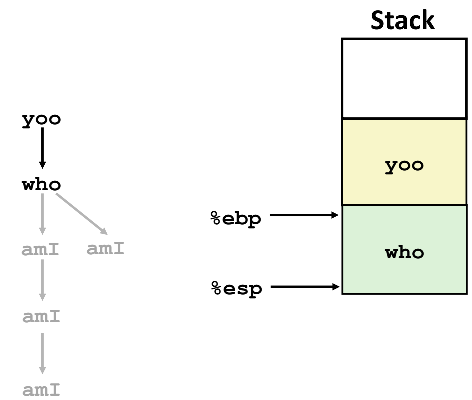
Example

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



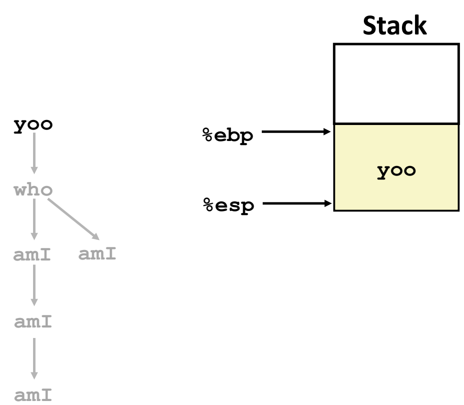
Example

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



Example

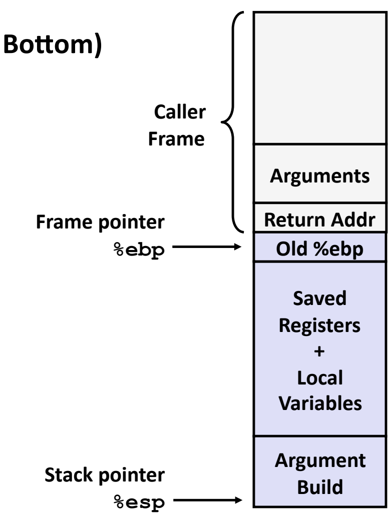
```
yoo (…)
{
    •
    •
    who () ;
    •
    •
}
```



How did we remember where to point %ebp when returning?

IA32/Linux Stack Frame

- **Current Stack Frame ("Top" to Bottom)**
  - "Argument build" area (parameters for function about to be called)
  - Local variables (if can't be kept in registers)
  - Saved register context (when reusing registers)
  - Old frame pointer (for caller)
- **Caller's 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;
}
```


Revisiting swap

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

void call_swap()
{
  swap(&zip1, &zip2);
}
```

Calling swap from `call_swap`

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

we know the address

not Global War!

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


Revisiting swap

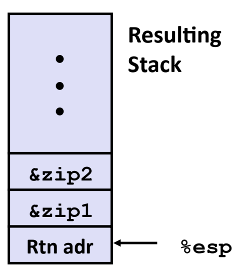
```
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
  swap(&zip1, &zip2);
}
```

Calling swap from `call_swap`

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

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



Revisiting swap

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

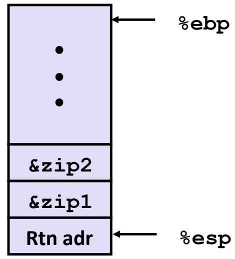
```
swap:
    pushl %ebp              } Set
    movl %esp,%ebp          } Up
    pushl %ebx              }

    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

Entering Stack



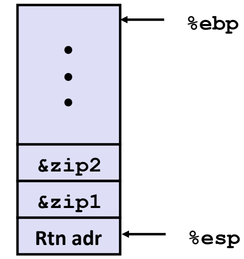
Resulting Stack?

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

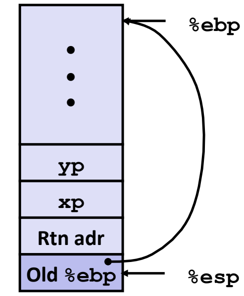
Set Up

swap Setup #1

Entering Stack



Resulting Stack

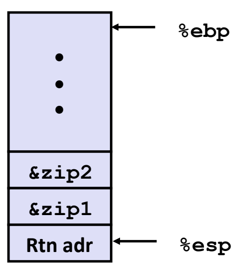



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

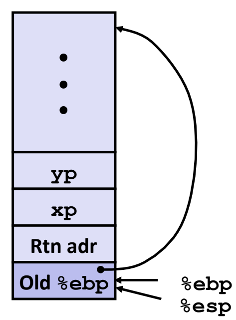
Set Up

swap Setup #2

Entering Stack



Resulting Stack

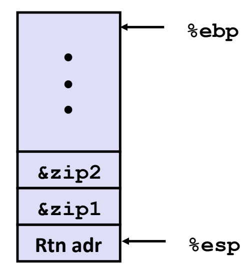



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

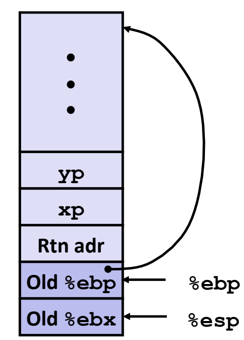
Set Up

swap Setup #3

Entering Stack



Resulting Stack

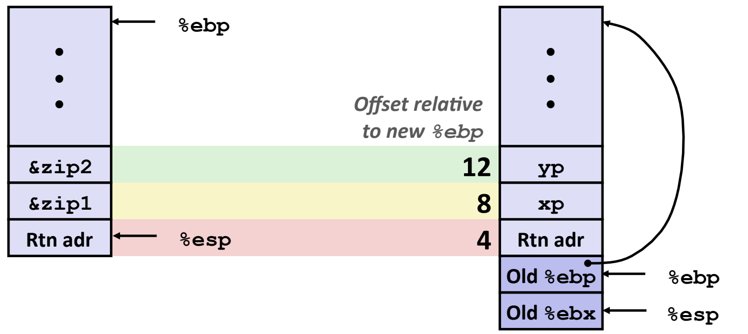



```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

Set Up

swap Body

Entering Stack **Resulting Stack**

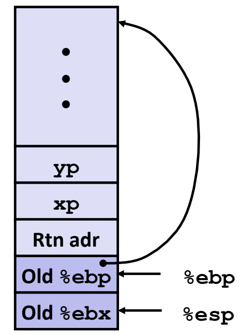



```
movl 12(%ebp),%ecx  # get yp
movl  8(%ebp),%edx  # get xp
. . .
```

Body

swap Finish #1

Finishing Stack **Resulting Stack?**

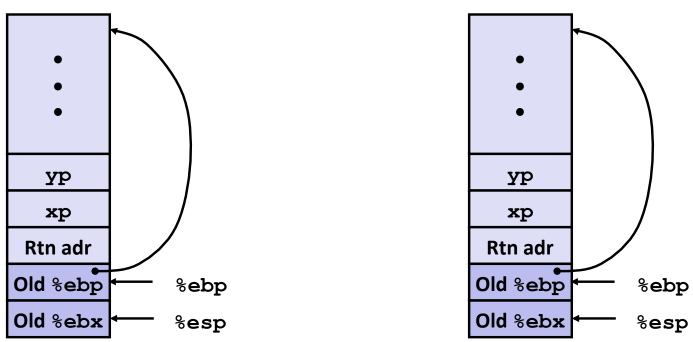



```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish

swap Finish #1

Finishing Stack **Resulting Stack**



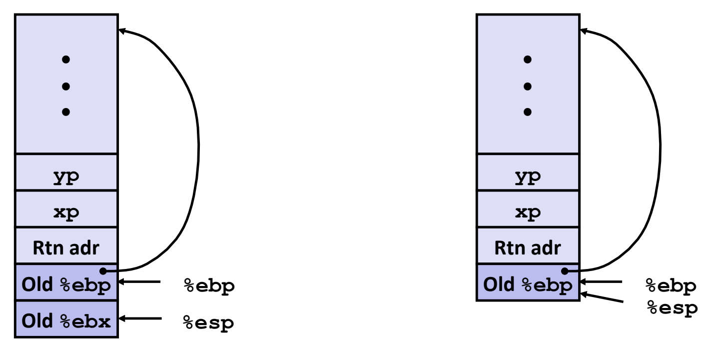

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish

Observation: Saved and restored register %ebx

swap Finish #2

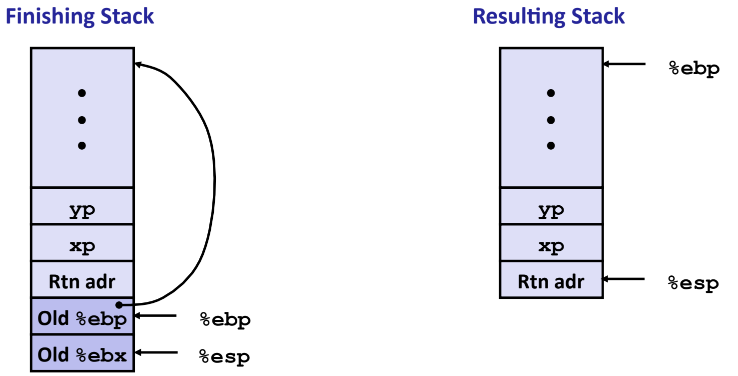
Finishing Stack **Resulting Stack**




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

Finish

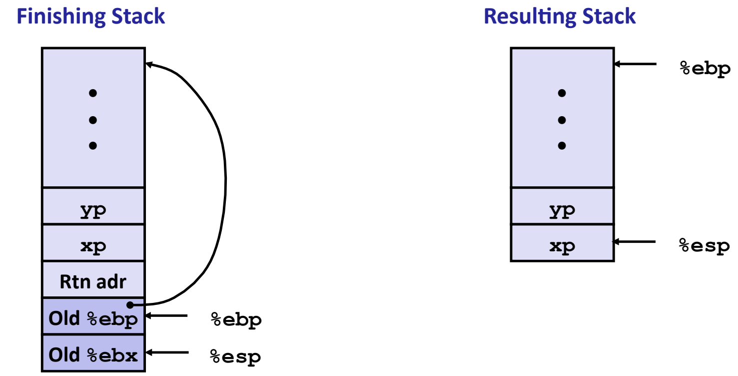
swap Finish #3




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

Finish

swap Finish #4




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

Finish

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
```

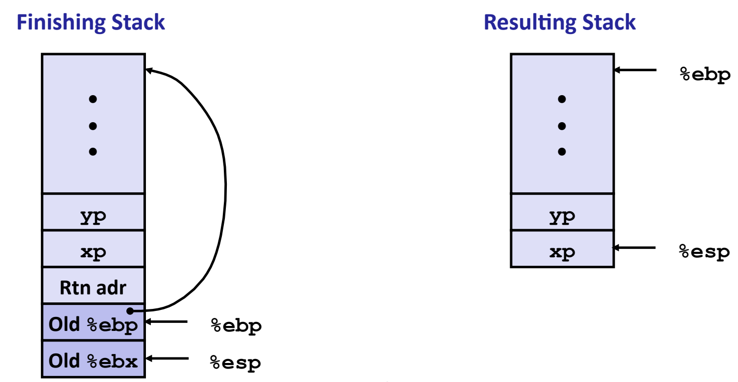
```
mov    %ebp,%esp
pop    %ebp
```

Calling Code

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

relative address (little endian)

swap Finish #4




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

Finish

- **Observation**
  - Saved & restored register `%ebx`
  - but not `%eax`, `%ecx`, or `%edx`

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the ***caller***
  - `who` is the ***callee***

- Can a register be used for temporary storage?

```
yoo:
    • • •
    movl $12345, %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 a register be used for temporary storage?
- Conventions
  - ***"Caller Save"***
    - Caller saves temporary values in its frame before calling
  - ***"Callee Save"***
    - Callee saves temporary values 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 form of callee save – restored to original values upon exit from procedure

Caller-Save Temporaries	`%eax`
	`%edx`
	`%ecx`
Callee-Save Temporaries	`%ebx`
	`%esi`
	`%edi`
Special	`%esp`
	`%ebp`

Example: Pointers to Local Variables

Top-Level Call

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

sfact(3)

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);
  }
}
```

Pass pointer to update location

Example: Pointers to Local Variables

Top-Level Call

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

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);
  }
}
```

sfact(3) val = 1
s_helper(3, &val)

Pass pointer to update location

Example: Pointers to Local Variables

Top-Level Call

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

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);
  }
}
```

sfact(3) val = 1
s_helper(3, &val) val = 3
s_helper(2, &val)

Pass pointer to update location

Example: Pointers to Local Variables

Top-Level Call

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

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);
  }
}
```

sfact(3) val = 1
s_helper(3, &val) val = 3
s_helper(2, &val) val = 6
s_helper(1, &val) .

Pass pointer to update location

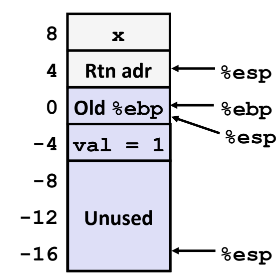
Creating & Initializing Pointer

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

- **Variable `val` must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as `-4(%ebp)`**
- **Push on stack as second argument**

Initial part of `sfact`

```
_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
```

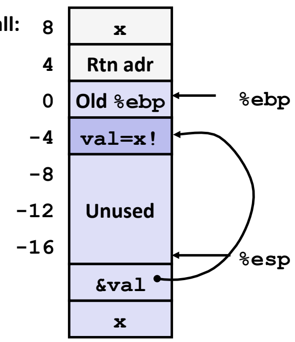


Passing Pointer

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

- **Variable `val` must be stored on stack**
  - Because: Need to create pointer to it
- **Compute pointer as `-4(%ebp)`**
- **Push on stack as second argument**

Stack at time of call:

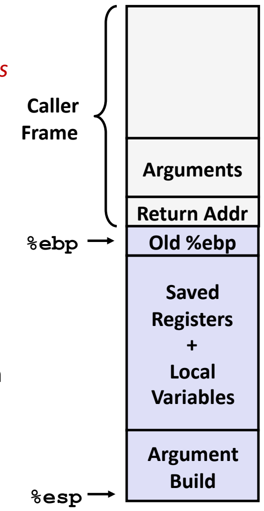


Calling `s_helper` from `sfact`

```
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
```


IA 32 Procedure Summary

- **Important points:**
  - IA32 procedures are a combination of *instructions* and *conventions*
    - Conventions prevent functions from disrupting each other
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- **Recursion handled by normal calling conventions**
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result returned in **`%eax`**



x86-64 Procedure Calling Convention

- **Doubling of registers makes us less dependent on stack**
  - Store argument in registers
  - Store temporary variables in registers
- **What do we do if we have too many arguments or too many temporary variables?**

x86-64 64-bit Registers: Usage Conventions

`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

Revisiting swap, IA32 vs. x86-64 versions

```
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                        }
```

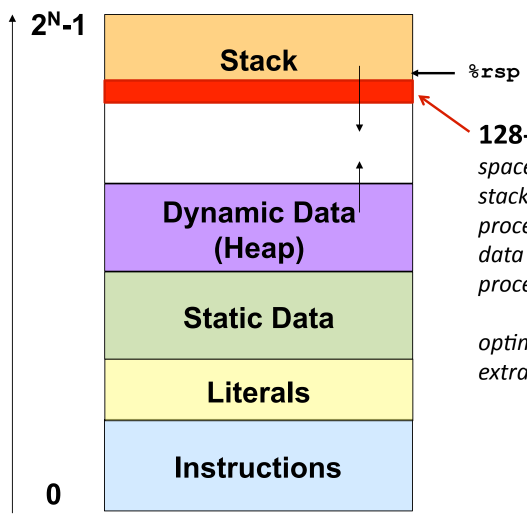
```
swap (64-bit long ints):
   movq    (%rdi), %rdx
   movq    (%rsi), %rax
   movq    %rax, (%rdi)
   movq    %rdx, (%rsi)
   ret
```

- **Arguments passed in registers**
  - First (`xp`) in `%rdi`, second (`yp`) in `%rsi`
  - 64-bit pointers
- **No stack operations required (except `ret`)**
- **Avoiding stack**
  - Can hold all local information in registers

X86-64 procedure call highlights

- **Arguments (up to first 6) in registers**
  - Faster to get these values from registers than from stack in memory
- **Local variables also in registers (if there is room)**
- **Registers still designated "caller-saved" or "callee-saved"**
- **`callq` instruction stores 64-bit return address on stack**
  - Address pushed onto stack, decrementing %rsp by 8
- **No frame pointer**
  - All references to stack frame made relative to %rsp; eliminates need to update %ebp/%rbp, which is now available for general-purpose use
- **Functions can access memory up to 128 bytes beyond %rsp: the "red zone"**
  - Can store some temps on stack without altering %rsp

x86-64 Memory Layout



128-byte red zone

space lower than the stack pointer that procedures can use for data not needed across procedure calls

optimization to avoid extra %rsp updates

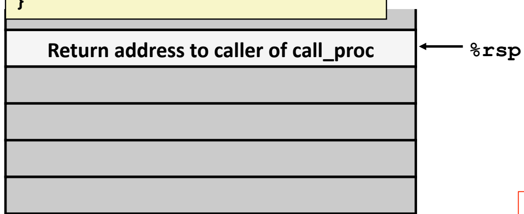
x86-64 Stack Frames

- **Often (ideally), x86-64 functions need no stack frame at all**
  - Just a return address is pushed onto the stack when a function call is made
- **A function *does* need a stack frame when it:**
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of calle**r**-save registers before calling a procedure
  - Needs to save the state of calle**e**-save registers before modifying them

Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)
  movl  $2,24(%rsp)
  movw  $3,28(%rsp)
  movb  $4,31(%rsp)
  • • •
```

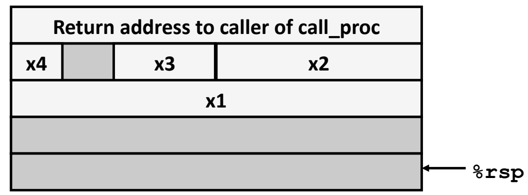


NB: Details may vary depending on compiler.

Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

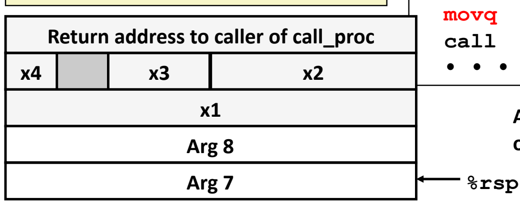
```
call_proc:
  subq  $32,%rsp
  movq  $1,16(%rsp)
  movl  $2,24(%rsp)
  movw  $3,28(%rsp)
  movb  $4,31(%rsp)
  • • •
```



Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  leaq  24(%rsp),%rcx
  leaq  16(%rsp),%rsi
  leaq  31(%rsp),%rax
  movq  %rax,8(%rsp)
  movl  $4,(%rsp)
  leaq  28(%rsp),%r9
  movl  $3,%r8d
  movl  $2,%edx
  movq  $1,%rdi
  call  proc
  • • •
```



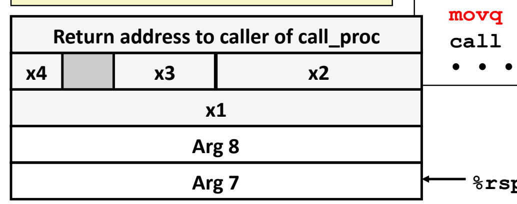
Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  leaq  24(%rsp),%rcx
  leaq  16(%rsp),%rsi
  leaq  31(%rsp),%rax
  movq  %rax,8(%rsp)
  movl  $4,(%rsp)
  leaq  28(%rsp),%r9
  movl  $3,%r8d
  movl  $2,%edx
  movq  $1,%rdi
  call  proc
  • • •
```

note sizes

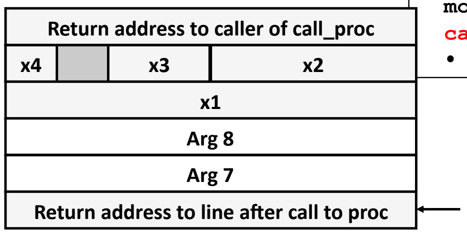


Arguments passed in (in order): rdi, rsi, rdx, rcx, r8, r9

Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  leaq  24(%rsp),%rcx
  leaq  16(%rsp),%rsi
  leaq  31(%rsp),%rax
  movq  %rax,8(%rsp)
  movl  $4,(%rsp)
  leaq  28(%rsp),%r9
  movl  $3,%r8d
  movl  $2,%edx
  movq  $1,%rdi
  call  proc
  • • •
```

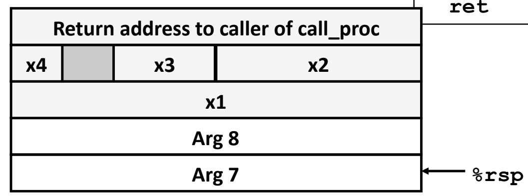


Example

sign extension!

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

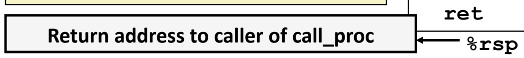
```
call_proc:
  • • •
  movswl 28(%rsp),%eax
  movsbl 31(%rsp),%edx
  subl   %edx,%eax
  cltq
  movslq 24(%rsp),%rdx
  addq   16(%rsp),%rdx
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```



Example

```
long int call_proc()
{
  long  x1 = 1;
  int   x2 = 2;
  short x3 = 3;
  char  x4 = 4;
  proc(x1, &x1, x2, &x2,
       x3, &x3, x4, &x4);
  return (x1+x2)*(x3-x4);
}
```

```
call_proc:
  • • •
  movswl 28(%rsp),%eax
  movsbl 31(%rsp),%edx
  subl   %edx,%eax
  cltq
  movslq 24(%rsp),%rdx
  addq   16(%rsp),%rdx
  imulq  %rdx,%rax
  addq   $32,%rsp
  ret
```



x86-64 Procedure Summary

- **Heavy use of registers (faster than using stack in memory)**
  - Parameter passing
  - More temporaries since more registers
- **Minimal use of stack**
  - Sometimes none
  - When needed, allocate/deallocate entire frame at once
  - No more frame pointer: address relative to stack pointer
- **More room for compiler optimizations**
  - Prefer to store data in registers rather than memory
  - Minimize modifications to stack pointer