Imperial College London

113: Architecture

Spring 2018

Lecture: X86 Stack and Procedures

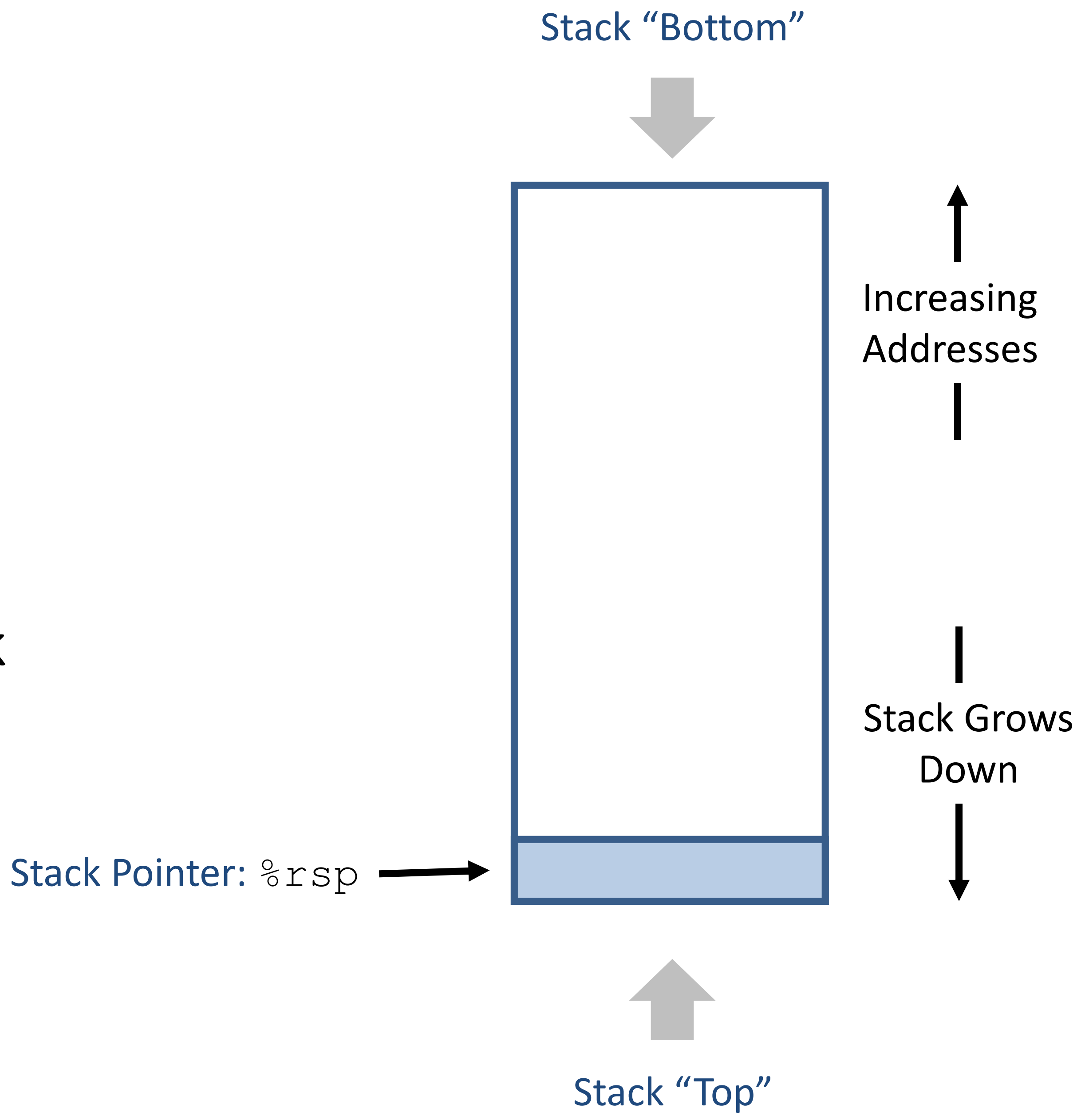
Instructor: Dr. Jana Giceva

Today

- **Procedure call and return**
- x86-64 calling conventions

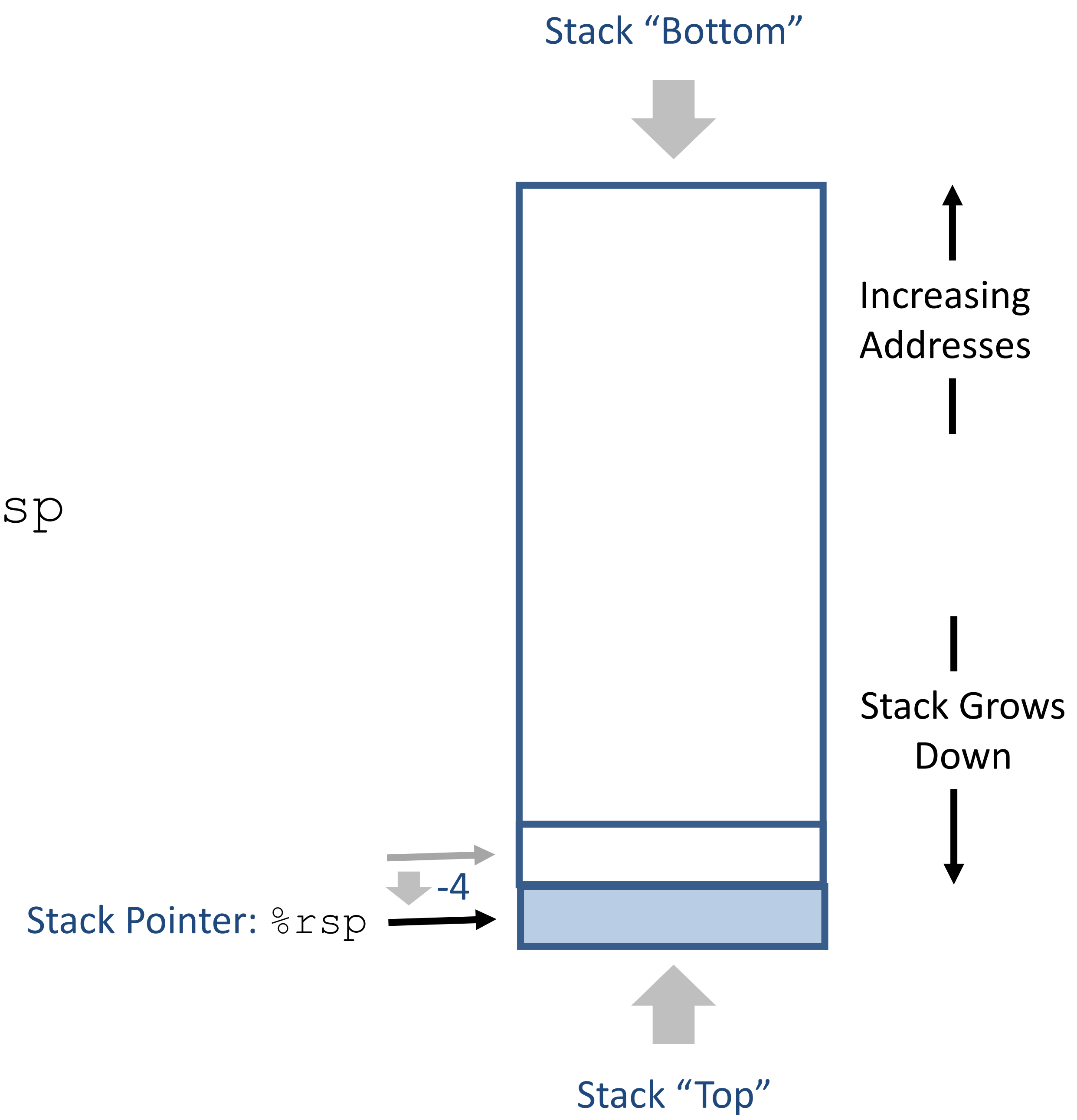
x86_64 stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%rsp` contains lowest stack address = address of “top” element



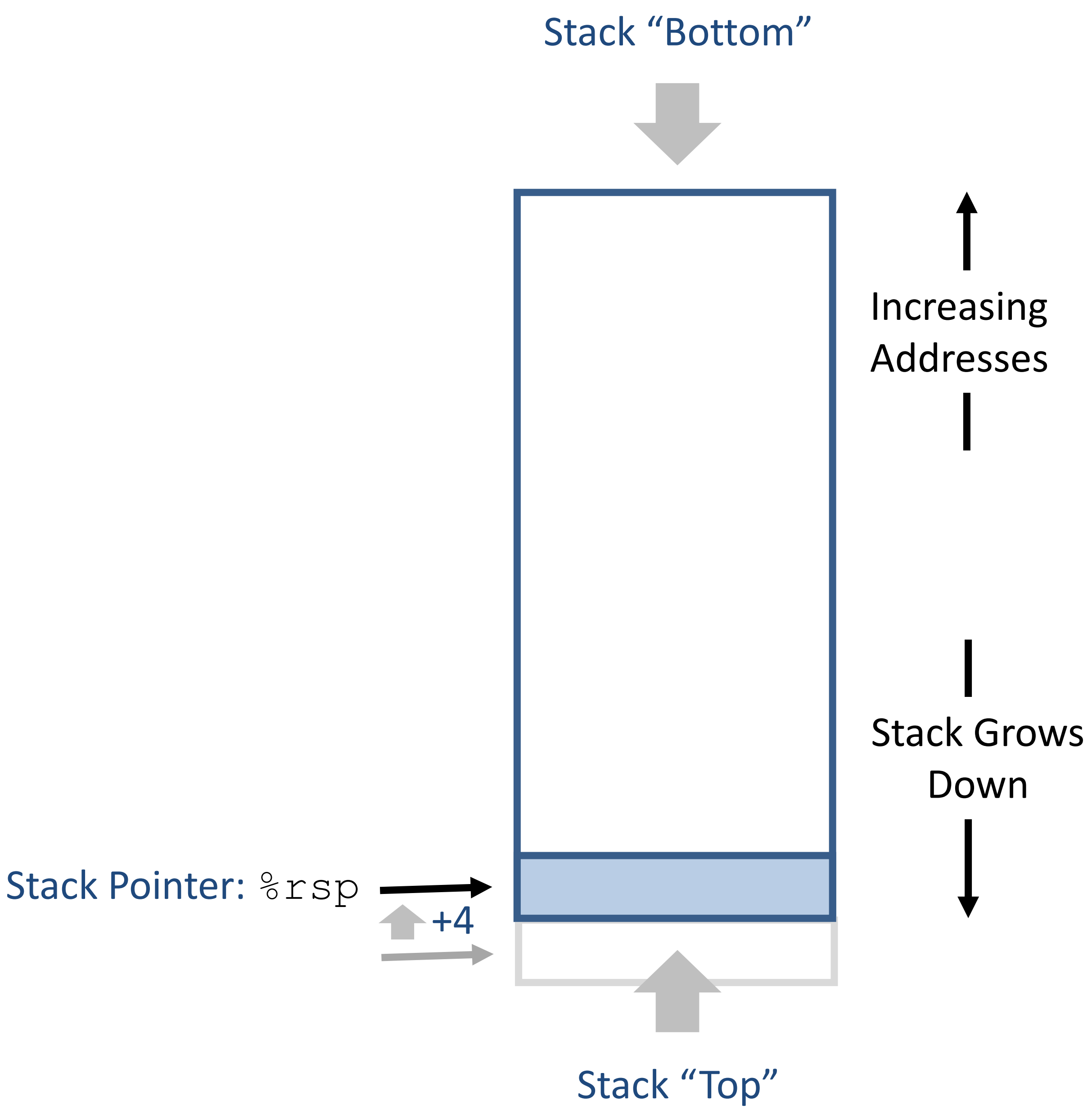
x86_64 stack: push

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



x86_64 stack: pop

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



Procedure control flow

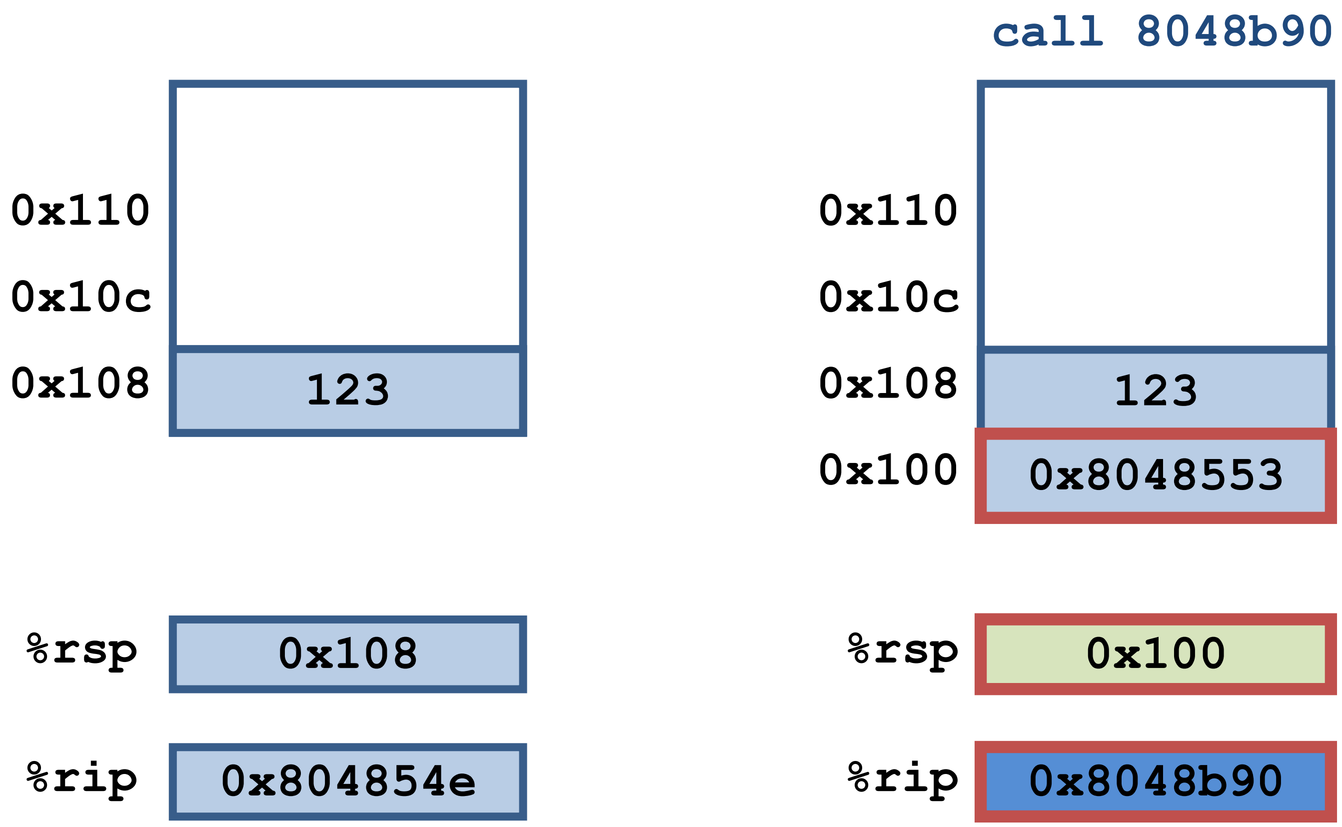
- **Use stack to support procedure call and return**
- **Procedure call:** `call` *Label*
  - Push return address on stack
  - Jump to *Label*
- **Return address:**
  - Address of instruction beyond `call`
  - Example from disassembly

```
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

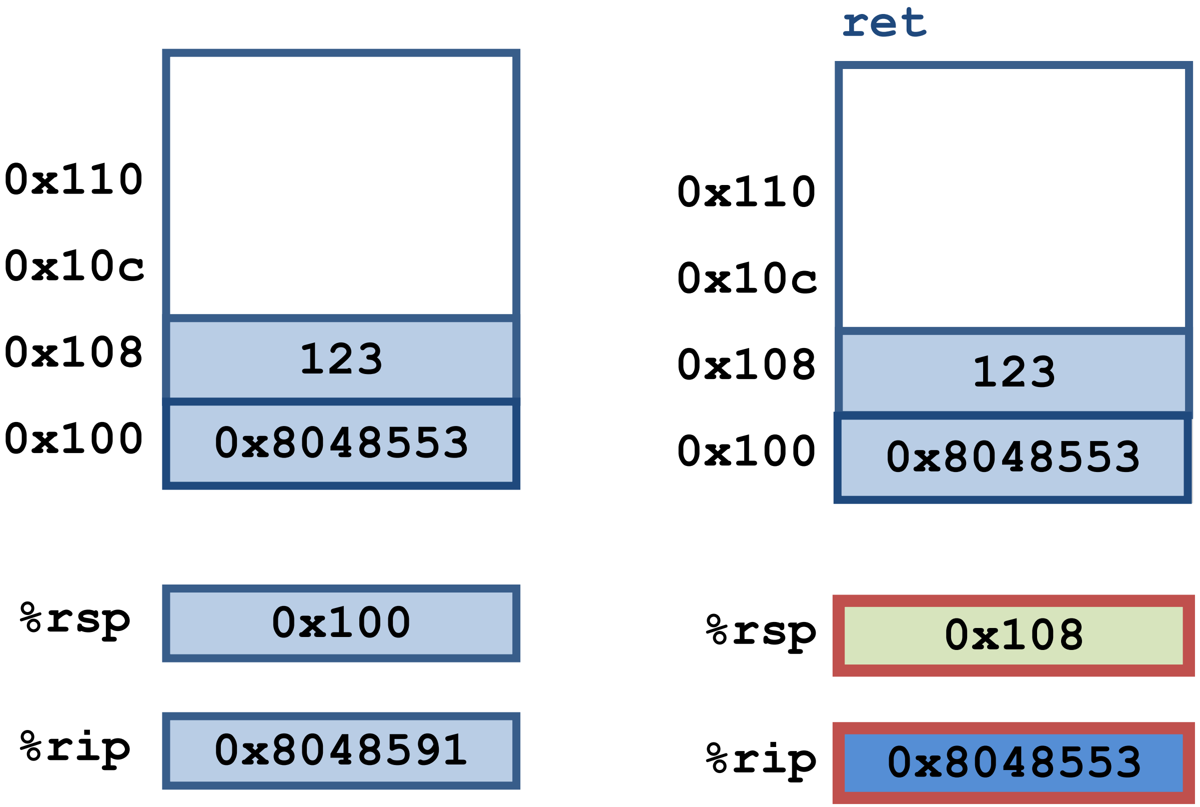
Procedure call example

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



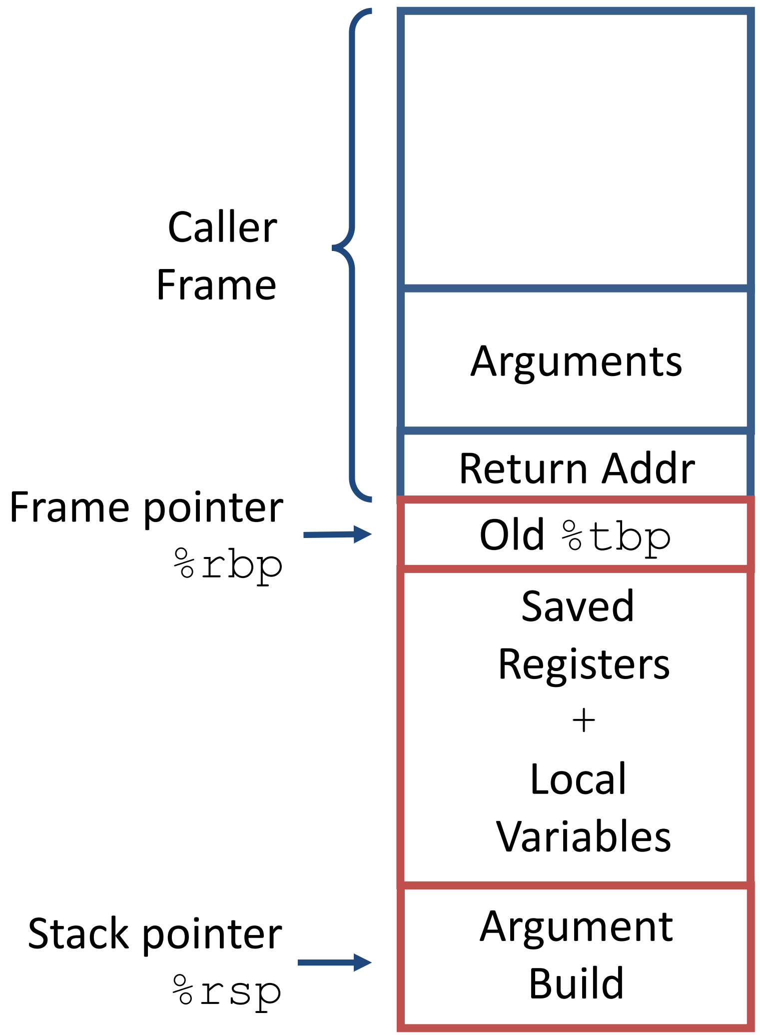
Procedure return example

```
8048591:     c3                        ret
```



Full x86_64/Linux stack frame

- **Current stack frame ("top" to bottom)**
  - Argument build
    Parameters for function about to call
  - 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



Today

- Procedure call and return
- **x86-64 calling conventions**

Recursive factorial

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

```
r_fact:
      pushq   %rbx
      movl    %edi, %ebx
      movl    $1, %eax
      cmpl    $1, %edi
      jle     .L2
      leal    -1(%rdi), %edi
      call    r_fact
      imull   %ebx, %eax
.L2
      popq    %rbx
      ret
```

- Registers
  - `%rax/%eax` used without first saving
  - `%rbx/%ebx` used, but saved at beginning and restored in the end

x86-64 integer registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

Register saving conventions

- **When a procedure `foo` calls `bar`:**
  - `foo` is the ***caller***
  - `bar` is the ***callee***

- **Can a register be used for temporary storage?**

```
foo:
    ...
    movl   $15213, %edx
    call   bar
    addl   %edx, %eax
    ...
    ret
```

```
bar:
    ...
    movl   8(%rsp), %edx
    addl   $91125, %edx
    ...
    ret
```

  - Contents of register `%edx` overwritten by `bar`

Register saving conventions

- **When a procedure `foo` calls `bar`:**
  - `foo` is the ***caller***
  - `bar` 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

Calling convention

Calling method (*Caller*)

- Push parameters in order last to first
- Push object instance
- Call (jump to) method

Called method (*Callee*)

- Save registers on stack
- Execute body of method
- Copy result (if any) to eax/rax
- Restore registers from stack
- Return from method (jump back)

Calling method (*Caller*), continued:

- Remove object instance from the stack
- Remove parameters from the stack
- Use method result

x86-64 integer registers

`%rax`	Return value, #varargs	`%r8`	Argument #5
`%rbx`	Callee saved; base ptr	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Static chain ptr
`%rdx`	Argument #3(and 2^{nd} return)	`%r11`	Used for linking
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack Pointer	`%r14`	Callee saved
`%rbp`	Callee saved, frame ptr	`%r15`	Callee saved

x86-64 registers

- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer**
  - Eliminates 32-bit need to update `%ebp/%rbp`
- **Other registers**
  - 6+1 callee saved
  - 2 or 3 have special uses

x86-64 integer registers

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

x86-64 stack frame example

```
long sum = 0;
/* swap a[i] and a[i+1] */
void swap_ele_su(long a[], int i)
{
  swap(&a[i], &a[i+1]);
  sum += a[i];
}
```

```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movslq  %esi,%rbx
    movq    %r12, -8(%rsp)
    movq    %rdi, %r12
    leaq    (%rdi,%rbx,8),%rdi
    subq    $16, %rsp
    leaq    8(%rdi),%rsi
    call    swap
    movq    (%r12,%rbx,8),%rax
    addq    %rax, sum(%rip)
    movq    (%rsp),%rbx
    movq    8(%rsp), %r12
    addq    $16, %rsp
    ret
```

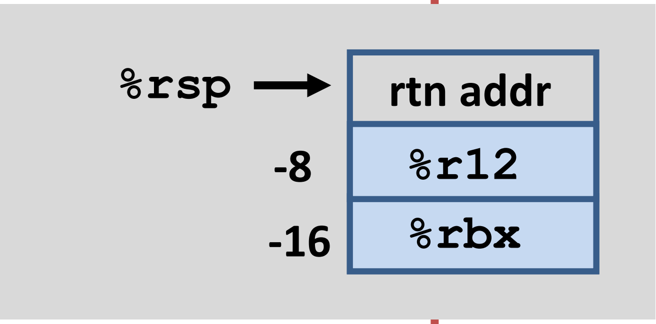
- Keeps values of **`a`** and **`i`** in callee save registers
- Must set up stack frame to save these registers

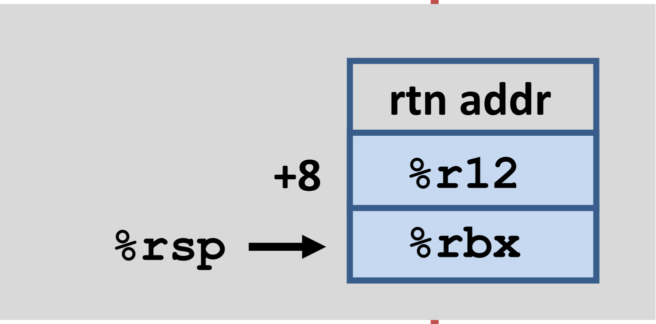
Understanding the x86-64 stack frame

```
swap_ele_su:
  movq   %rbx, -16(%rsp)          # save %rbx
  movslq %esi,%rbx                # extend and save i
  movq   %r12, -8(%rsp)           # save %r12
  movq   %rdi, %r12               # save a
  leaq   (%rdi,%rbx,8),%rdi       # &a[i]
  subq   $16, %rsp                # allocate stack frame
  leaq   8(%rdi),%rsi             #       &a[i+1]
  call   swap                     # call swap()
  movq   (%r12,%rbx,8),%rax       # a[i]
  addq   %rax, sum(%rip)          # sum += a[i]
  movq   (%rsp),%rbx              # restore %rbx
  movq   8(%rsp), %r12            # restore %r12
  addq   $16, %rsp                # deallocate stack frame
  ret
```

Understanding the x86-64 stack frame

```
swap_ele_su:
  movq    %rbx, -16(%rsp)          # save %rbx

  movq    %r12, -8(%rsp)           # save %r12


  subq    $16, %rsp                # allocate stack frame




  movq    (%rsp),%rbx              # restore %rbx
  movq    8(%rsp), %r12            # restore %r12
  addq    $16, %rsp                # deallocate stack frame
```





Interesting features of the stack frame

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Do by decrementing the stack pointer

- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

x86-64 long swap

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

```
swap:
    movq    (%rdi), %rdx
    movq    (%rsi), %rax
    movq    %rax, (%rdi)
    movq    %rdx, (%rsi)
    ret
```

- Operands 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 non-leaf w/o stack frame

```
long scount = 0;
/* swap a[i] and a[i+1] */
void swap_ele_su(long a[], int i)
{
  swap(&a[i], &a[i+1]);
  scount++;
}
```

```
swap_ele_su:
  movslq   %esi, %rsi
  leaq     (%rdi,%rsi,8),%rdi
  leaq     8(%rdi), %rsi
  call     swap
  incq     scount(%rip)
  ret
```

- Call method is implemented using `push` and `jmp`
  - `call method`   `push rip`
                         `jmp method`
- Ret method is implement using `pop`
  - `ret`   `pop rip`

The address pushed by `call` is the address of the next instruction to resume execution after the ***called*** method has finished

X86-64 procedure summary

- Use of registers
  - Callee save, argument passing, other …
- Many tricky optimizations
  - What kind of stack frame to use (if at all)
  - Rarely need a base (frame) pointer
  - Calling with jump
  - Various allocation techniques
- But the ***conventions*** still hold