Procedures & The Stack II

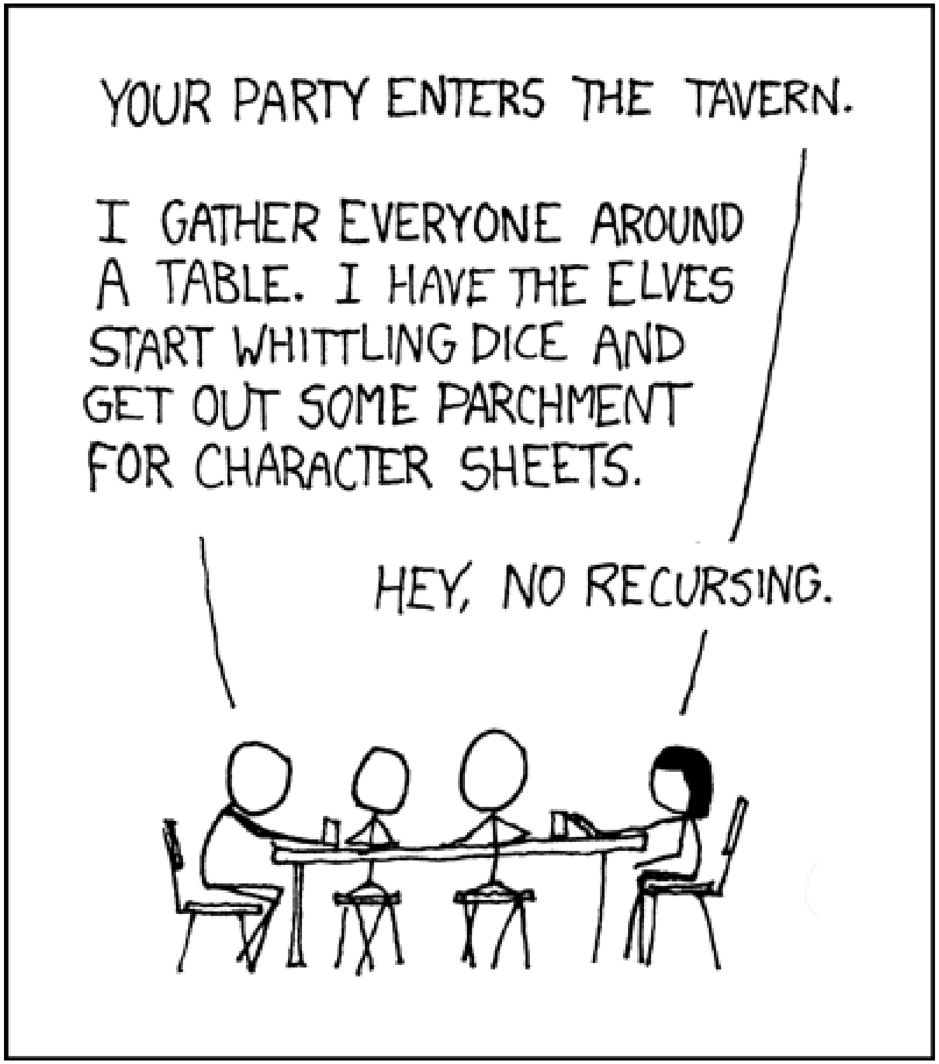
CSE 351 Autumn 2016

Instructor:

Justin Hsia

Teaching Assistants:

Chris Ma
Hunter Zahn
John Kaltenbach
Kevin Bi
Sachin Mehta
Suraj Bhat
Thomas Neuman
Waylon Huang
Xi Liu
Yufang Sun



http://xkcd.com/244/

Administrivia

- Lab 2 due Friday

- **Midterm** on Nov. 2 in lecture
  - Make a cheat sheet! – two-sided letter page, *handwritten*
  - Historically my exams have averages of 65-70%
  - Check Piazza this week for announcements & practice problems
- **Midterm review session**
  - 5-7pm on Monday, Oct. 31 in EEB 105
- Look for additional staff office hours as well

Example: `increment`

```
long increment(long *p, long val) {
    long x = *p;
    long y = x + val;
    *p = y;
    return x;
}
```

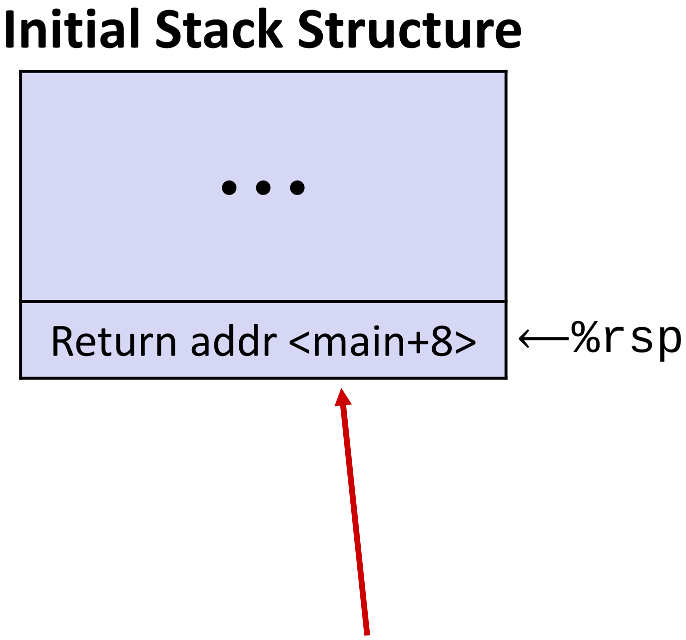
```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Register	Use(s)
`%rdi`	1st arg (`p`)
`%rsi`	2nd arg (`val`), `y`
`%rax`	`x`, return value

Procedure Call Example (initial state)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```



Initial Stack Structure

• • •

Return addr <main+8>

←%rsp

- Return address on stack is the address of instruction immediately *following* the call to "`call_incr`"
    - Shown here as `main`, but could be anything)

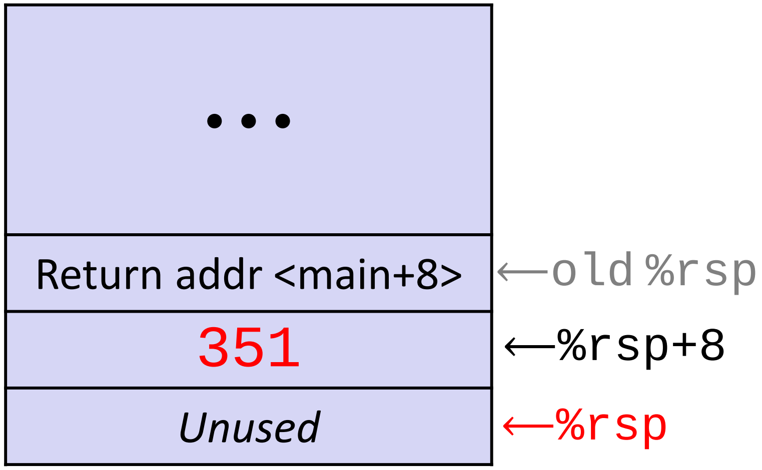
Procedure Call Example (step 1)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Allocate space for local vars

Stack Structure



- Setup space for local variables
  - Only v1 needs space on the stack
- Compiler allocated extra space
  - Often does this for a variety of reasons, including alignment

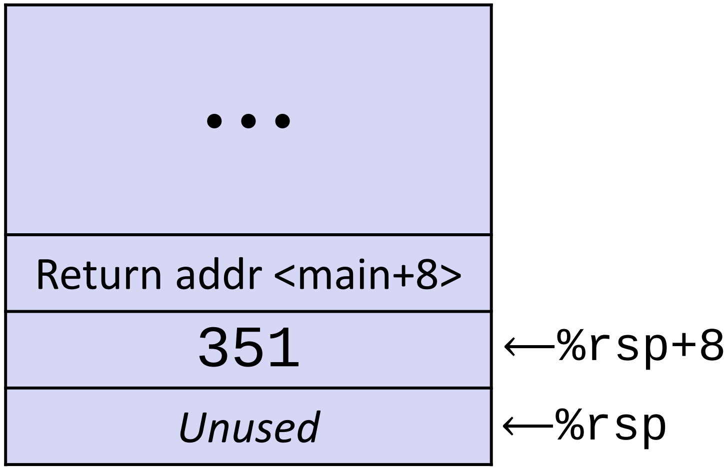
Procedure Call Example (step 2)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

Set up parameters for call to `increment`

Stack Structure



• • •

Return addr <main+8>

351 ⟵%rsp+8

Unused ⟵%rsp

Aside: `movl` is used because 100 is a small positive value that fits in 32 bits. High order bits of `rsi` get set to zero automatically. It takes *one less byte* to encode a `movl` than a `movq`.

Register	Use(s)
%rdi	&v1
%rsi	100

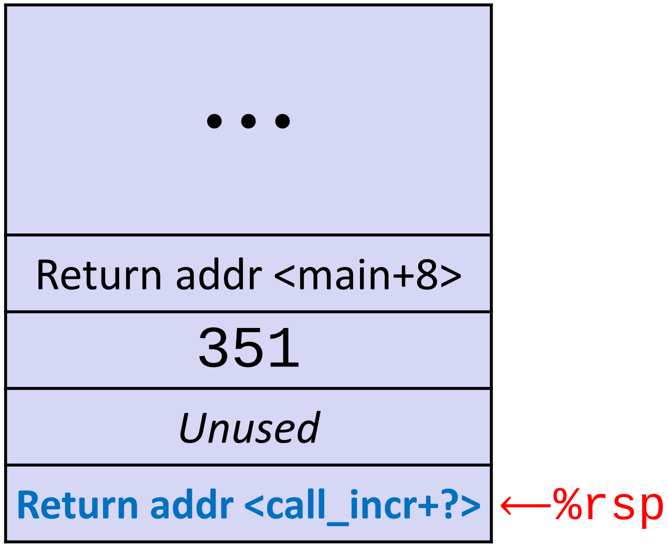
Procedure Call Example (step 3)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax
  addq    %rax, %rsi
  movq    %rsi, (%rdi)
  ret
```

Stack Structure



- State while inside `increment`
  - **Return address** on top of stack is address of the `addq` instruction immediately following call to `increment`

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`100`
`%rax`	

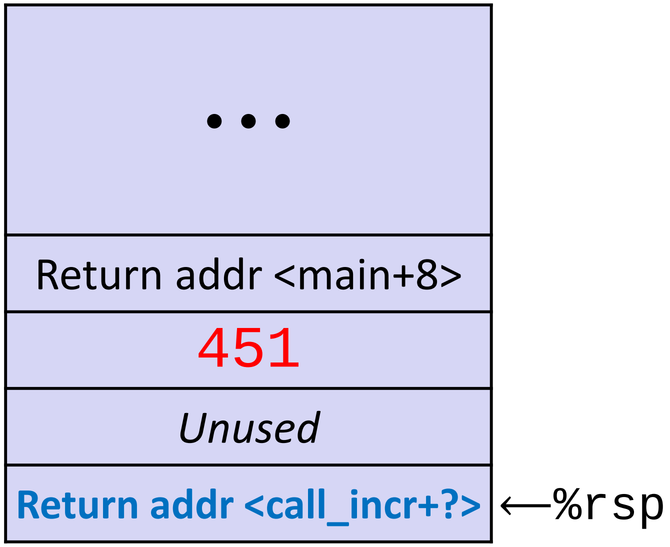
Procedure Call Example (step 4)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

```
increment:
  movq    (%rdi), %rax # x = *p
  addq    %rax, %rsi   # y = x+100
  movq    %rsi, (%rdi) # *p = y
  ret
```

Stack Structure



- State while inside `increment`
  - *After* code in body has been executed

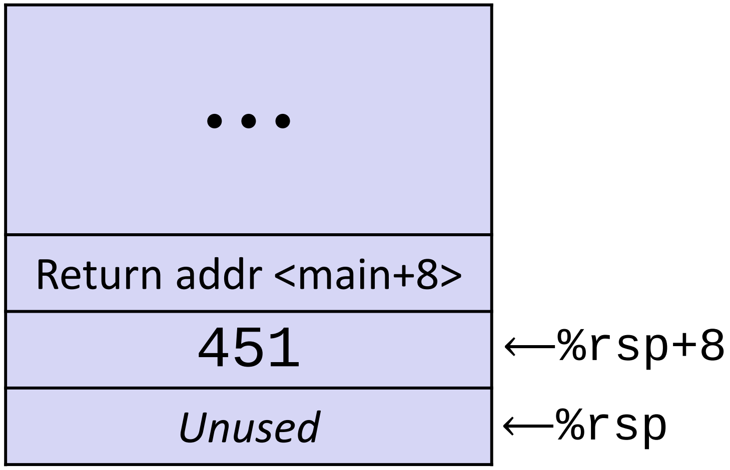
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	351

Procedure Call Example (step 5)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



- After returning from call to `increment`
  - Registers and memory have been modified and return address has been popped off stack

Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`451`
`%rax`	`351`

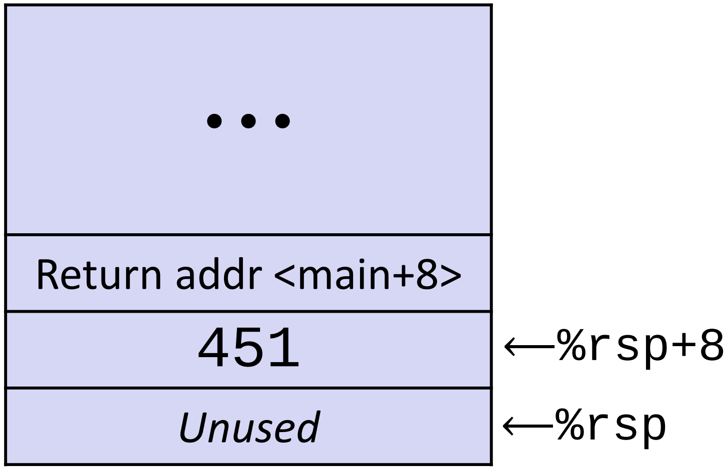
Procedure Call Example (step 6)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq     $16, %rsp
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     8(%rsp), %rax
  addq     $16, %rsp
  ret
```

← Update `%rax` to contain `v1+v2`

Stack Structure



• • •

Return addr <main+8>

451 ←%rsp+8

Unused ←%rsp

Register	Use(s)
%rdi	&v1
%rsi	451
%rax	451+351

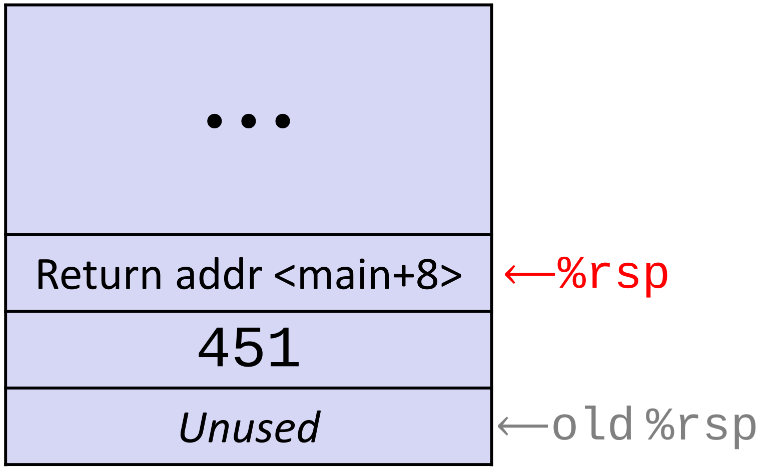
Procedure Call Example (step 7)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

← De-allocate space for local vars

Stack Structure



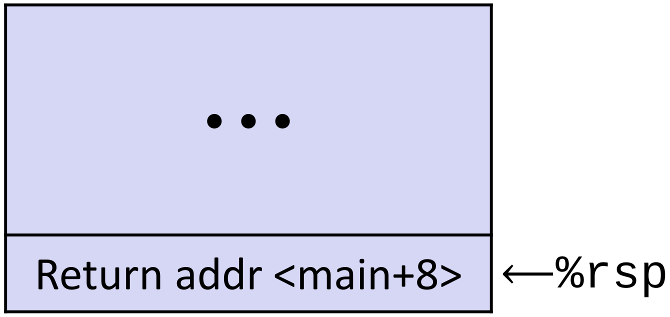
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 8)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure



• • •

Return addr <main+8>

⟵%rsp

- State *just before* returning from call to `call_incr`

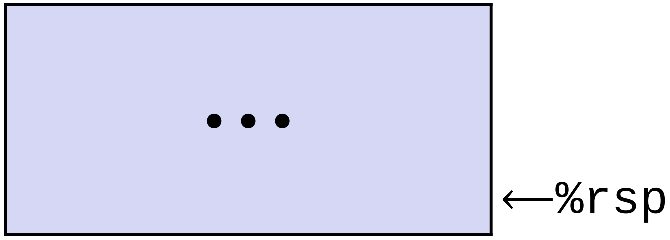
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Procedure Call Example (step 9)

```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Final Stack Structure



- State immediately *after* returning from call to `call_incr`
  - Return addr has been popped off stack
  - Control has returned to the instruction immediately following the call to `call_incr` (not shown here)

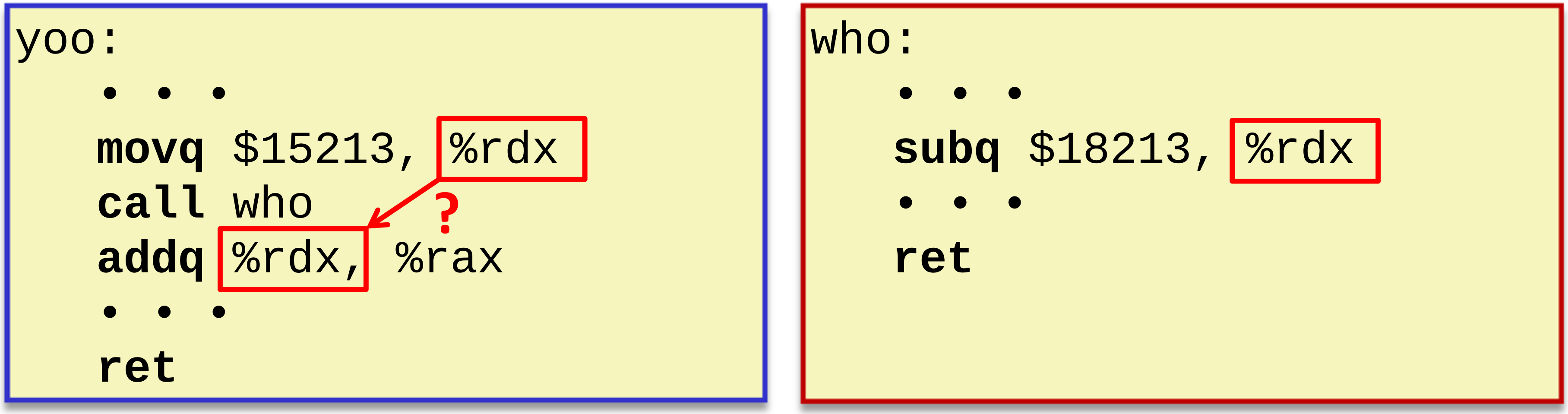
Register	Use(s)
`%rdi`	`&v1`
`%rsi`	`451`
`%rax`	`802`

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Register Saving Conventions

- When procedure `yoo` calls `who`:
  - `yoo` is the ***caller***
  - `who` is the ***callee***
- Can registers be used for temporary storage?



```
yoo:
   • • •
   movq $15213, %rdx
   call who
   addq %rdx, %rax
   • • •
   ret
```

```
who:
   • • •
   subq $18213, %rdx
   • • •
   ret
```

  - No! Contents of register `%rdx` overwritten by `who`!
  - This could be trouble – something should be done. Either:
    - ***Caller*** should save `%rdx` before the call (and restore it after the call)
    - ***Callee*** should save `%rdx` before using it (and restore it before returning)

Register Saving Conventions

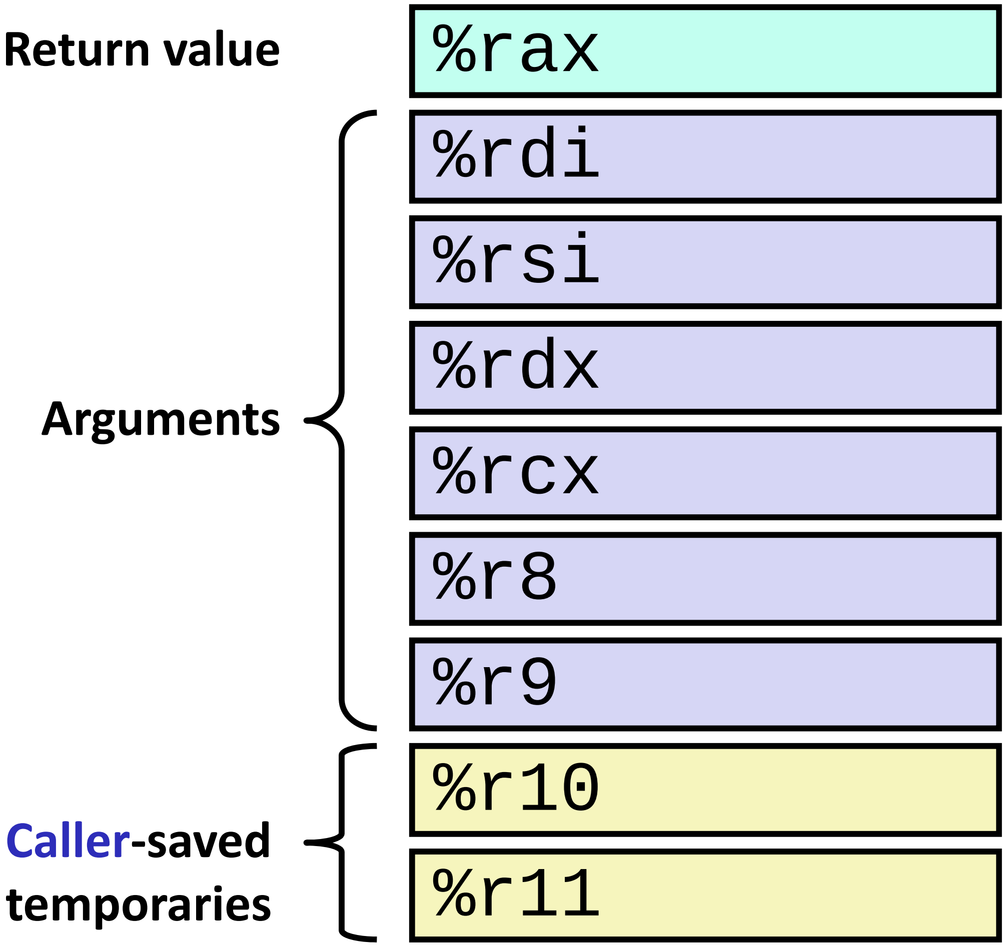
- ***"Caller-saved" registers***
  - It is the caller's responsibility to save any important data in these registers before calling another procedure (i.e. the **callee** can freely change data in these registers)
  - Caller saves values in its stack frame before calling **Callee**, then restores values after the call
- ***"Callee-saved" registers***
  - It is the callee's responsibility to save any data in these registers before using the registers (i.e. the **caller** assumes the data will be the same across the **callee** procedure call)
  - **Callee** saves values in its stack frame before using, then restores them before returning to **caller**

Silly Register Convention Analogy

1) Parents (***caller***) leave for the weekend and give the keys to the house to their child (***callee***)
   - Being suspicious, they put away/hid the valuables (***caller***-*saved*) before leaving
   - Warn child to leave the bedrooms untouched: "These rooms better look the same when we return!"
2) Child decides to throw a wild party (*computation*), spanning the entire house
   - To avoid being disowned, child moves all of the stuff from the bedrooms to the backyard shed (***callee***-*saved*) before the guests trash the house
   - Child cleans up house after the party and moves stuff back to bedrooms
3) Parents return home and are satisfied with the state of the house
   - Move valuables back and continue with their lives

x86-64 Linux Register Usage, part 1

- **`%rax`**
  - Return value
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%rdi, ..., %r9`**
  - Arguments
  - Also **caller**-saved & restored
  - Can be modified by procedure
- **`%r10, %r11`**
  - **Caller**-saved & restored
  - Can be modified by procedure



Return value

`%rax`

Arguments

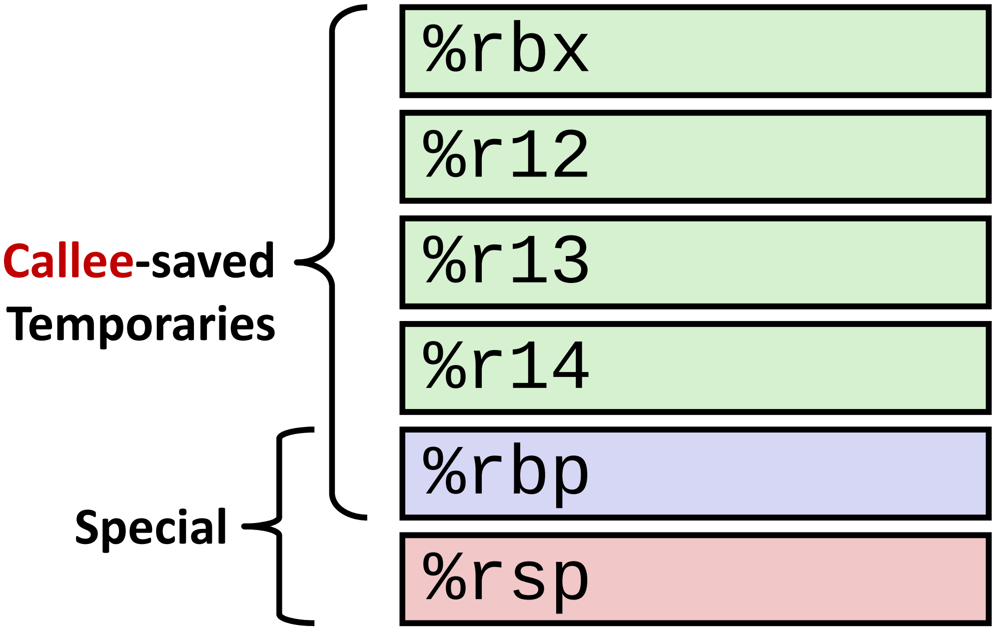
`%rdi`
`%rsi`
`%rdx`
`%rcx`
`%r8`
`%r9`

Caller-saved temporaries

`%r10`
`%r11`

x86-64 Linux Register Usage, part 2

- `%rbx, %r12, %r13, %r14`
  - Callee-saved
  - Callee must save & restore
- `%rbp`
  - Callee-saved
  - Callee must save & restore
  - May be used as frame pointer
  - Can mix & match
- `%rsp`
  - Special form of callee save
  - Restored to original value upon exit from procedure



Callee-saved
Temporaries

Special

x86-64 64-bit Registers: Usage Conventions

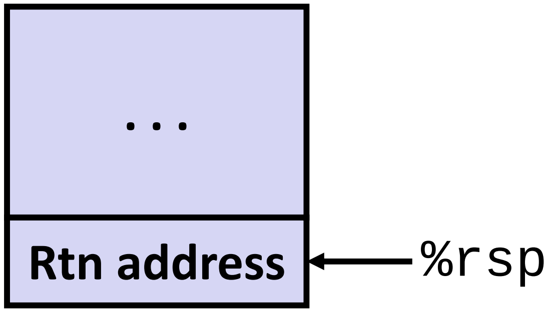
`%rax`	Return value - Caller saved
`%rbx`	Callee saved
`%rcx`	Argument #4 - Caller saved
`%rdx`	Argument #3 - Caller saved
`%rsi`	Argument #2 - Caller saved
`%rdi`	Argument #1 - Caller saved
`%rsp`	Stack pointer
`%rbp`	Callee saved
`%r8`	Argument #5 - Caller saved
`%r9`	Argument #6 - Caller saved
`%r10`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

Callee-Saved Example (step 1)

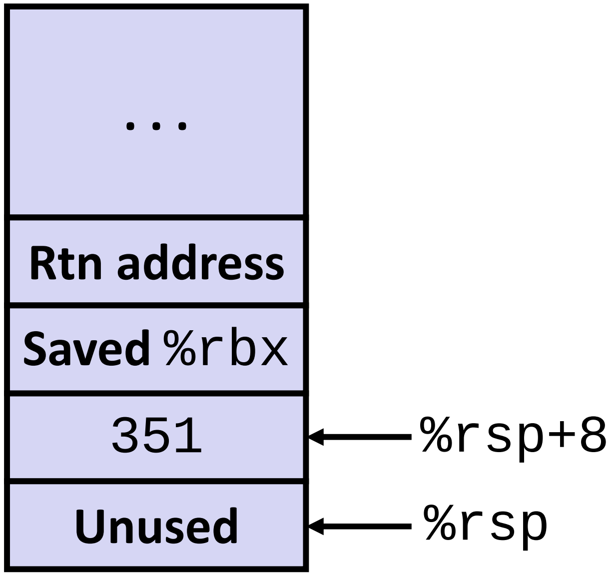
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq    %rbx
  subq     $16, %rsp
  movq     %rdi, %rbx
  movq     $351, 8(%rsp)
  movl     $100, %esi
  leaq     8(%rsp), %rdi
  call     increment
  addq     %rbx, %rax
  addq     $16, %rsp
  popq     %rbx
  ret
```

Initial Stack Structure



Resulting Stack Structure

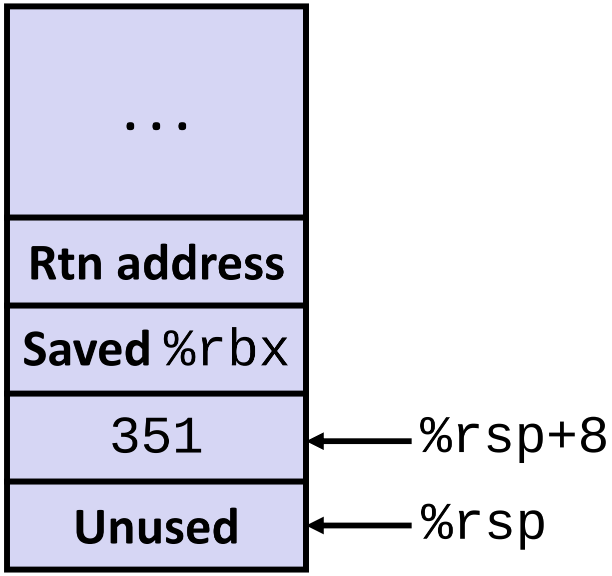


Callee-Saved Example (step 2)

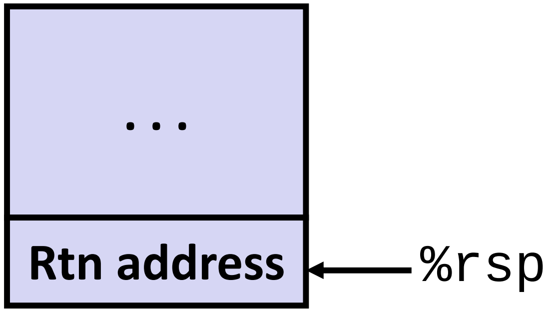
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Stack Structure



Pre-return Stack Structure



Why Caller *and* Callee Saved?

- We want *one* calling convention to simply separate implementation details between caller and callee

- In general, neither caller-save nor callee-save is "best":
  - If caller isn't using a register, caller-save is better
  - If callee doesn't need a register, callee-save is better
  - If "do need to save", callee-save generally makes smaller programs
    - Functions are called from multiple places

- So… "some of each" and compiler tries to "pick registers" that minimize amount of saving/restoring

Register Conventions Summary

- **Caller**-saved register values need to be pushed onto the stack before making a procedure call *only if the Caller needs that value later*
  - **Callee** may change those register values
- **Callee**-saved register values need to be pushed onto the stack *only if the Callee intends to use those registers*
  - **Caller** expects unchanged values in those registers

- Don’t forget to restore/pop the values later!

Procedures

- Stack Structure
- Calling Conventions
  - Passing control
  - Passing data
  - Managing local data
- Register Saving Conventions
- Illustration of Recursion

Recursive Function

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Compiler Explorer:

https://godbolt.org/g/4ZJbz1

- Compiled with `-O1` for brevity instead of `-Og`
- Try `-O2` instead!

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Base Case

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x	Argument
%rax	Return value	Return value

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Trick because some AMD hardware doesn't like jumping to `ret`

Recursive Function: Callee Register Save

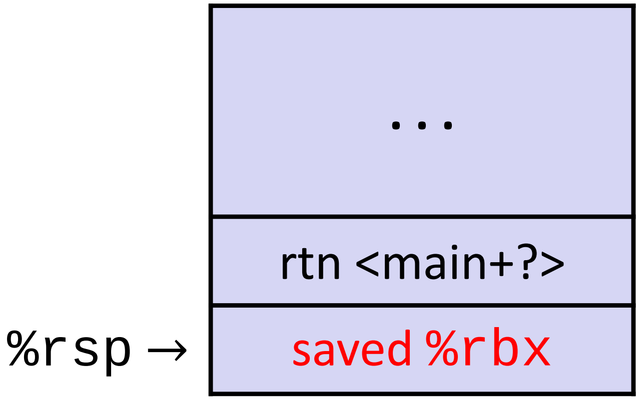
```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
`%rdi`	`x`	Argument

Need original value of `x` *after* recursive call to `pcount_r`.

“Save” by putting in `%rbx` (**callee** saved), but need to save old value of `%rbx` before you change it.

The Stack



. . .

rtn <main+?>

`%rsp` → saved `%rbx`

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

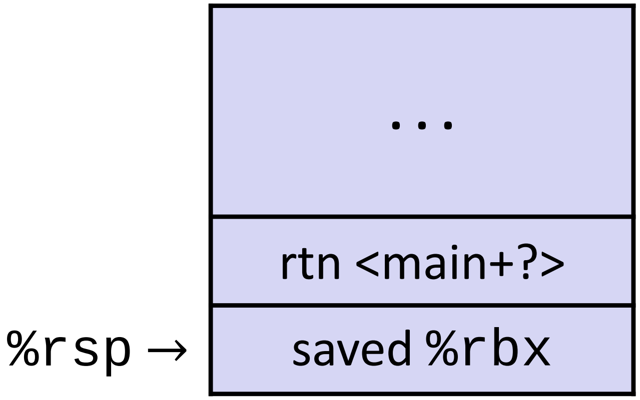
Recursive Function: Call Setup

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rdi	x (new)	Argument
%rbx	x (old)	Callee saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

The Stack



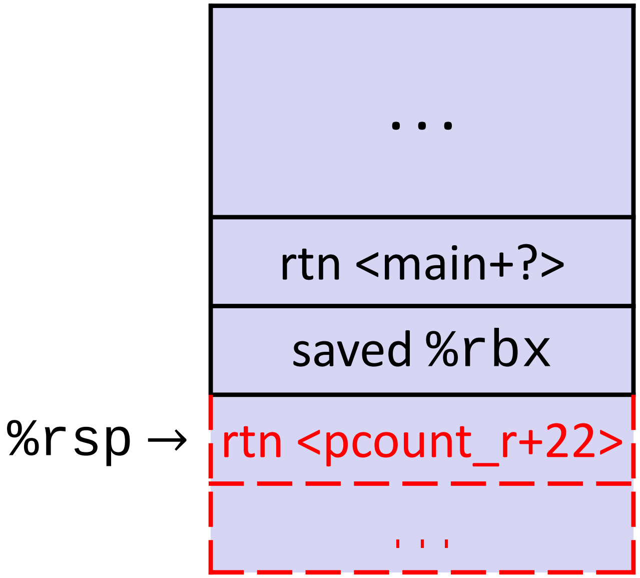
Recursive Function: Call

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Recursive call return value	Return value
%rbx	x (old)	Callee saved

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

The Stack



. . .

rtn <main+?>

saved %rbx

%rsp →

rtn <pcount_r+22>

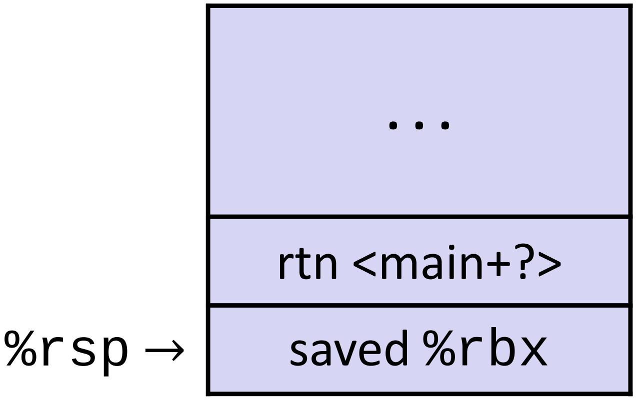
. . .

Recursive Function: Result

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	x&1	Callee saved

The Stack



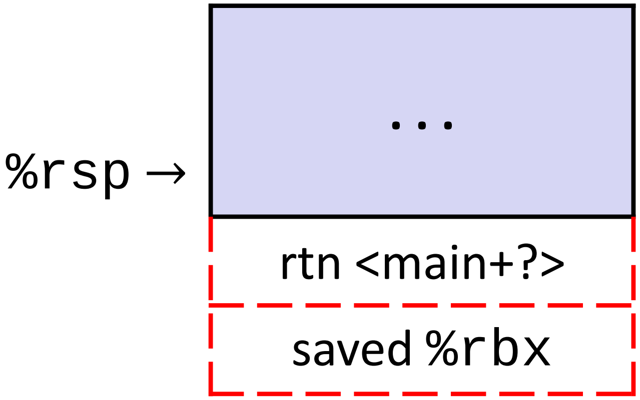
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Recursive Function: Completion

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
  if (x == 0)
    return 0;
  else
    return (x&1)+pcount_r(x >> 1);
}
```

Register	Use(s)	Type
%rax	Return value	Return value
%rbx	Previous %rbx value	Callee restored

The Stack



```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep ret
```

Observations About Recursion

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
    - Unless the code explicitly does so (e.g., buffer overflow)
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out (LIFO)
- Also works for mutual recursion (P calls Q; Q calls P)

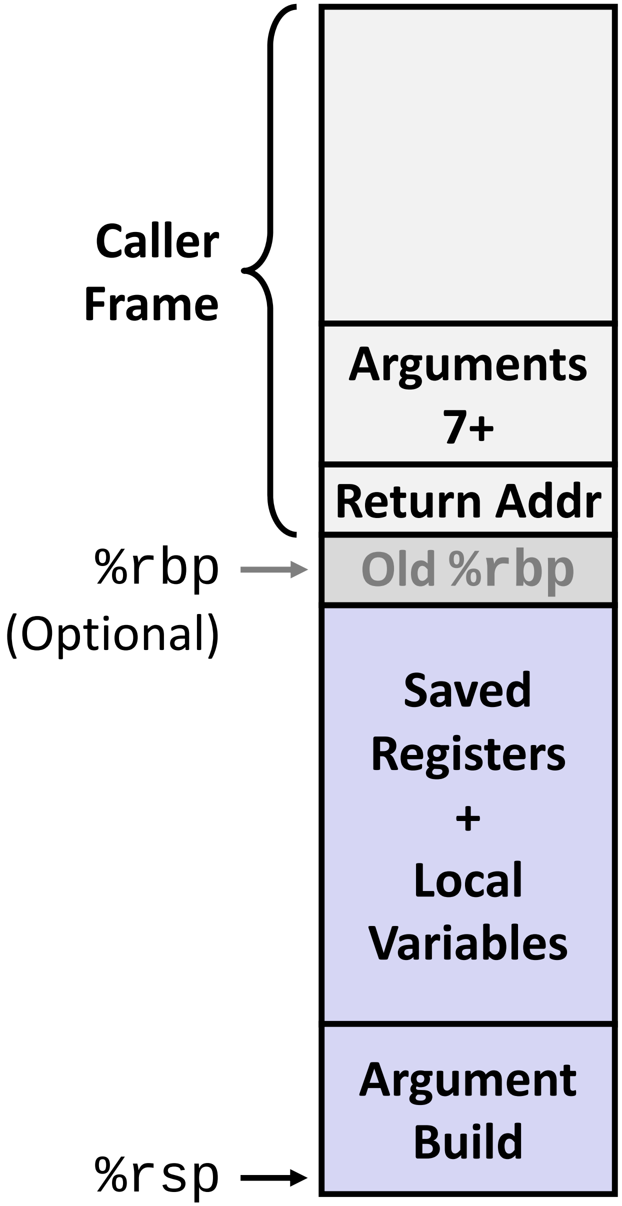
x86-64 Stack Frames

- Many x86-64 procedures need no stack frame at all
  - Only return address is pushed onto the stack when calling another procedure

- A procedure *does need* a stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers
  - Has local variables that are arrays or structs
  - Uses & to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Is using **caller**-saved registers and then calls a procedure
  - Modifies/uses **callee**-saved registers

x86-64 Procedure Summary

- Important Points
  - 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
- Heavy use of registers
  - Faster than using memory
  - Use limited by data size and conventions
- Minimize use of the Stack



BONUS SLIDES

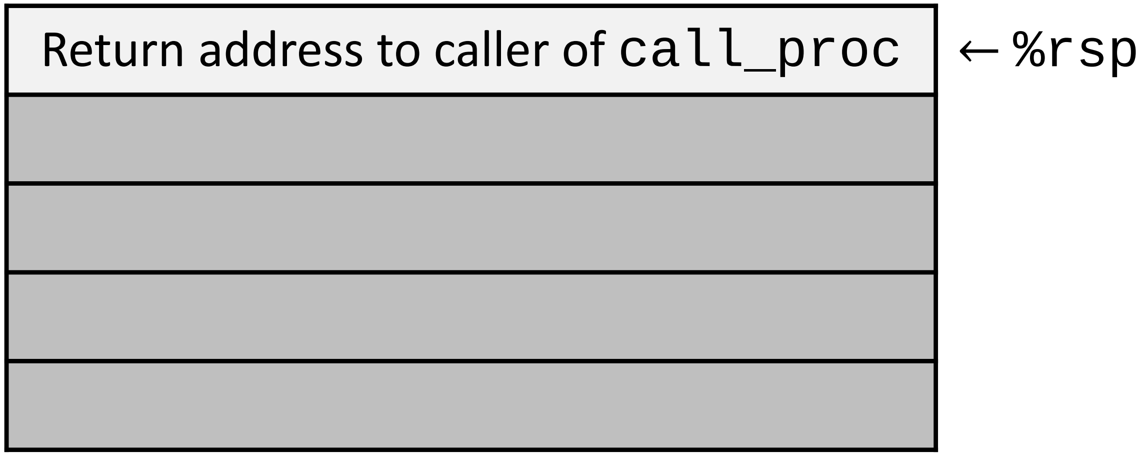
One more x86-64 example that shows passing of more than 6 arguments and passing addresses of local variables. The following example, along with a brief re-cap of x86-64 calling conventions is in the following video:

- https://courses.cs.washington.edu/courses/cse351/videos/05/056.mp4
- Alternate (but similar) version: https://godbolt.org/g/E7UFJ7

x86-64 Example (1)

```
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) # x1
  movl  $2,24(%rsp) # x2
  movw  $3,28(%rsp) # x3
  movb  $4,31(%rsp) # x4
  • • •
```



Return address to caller of `call_proc`

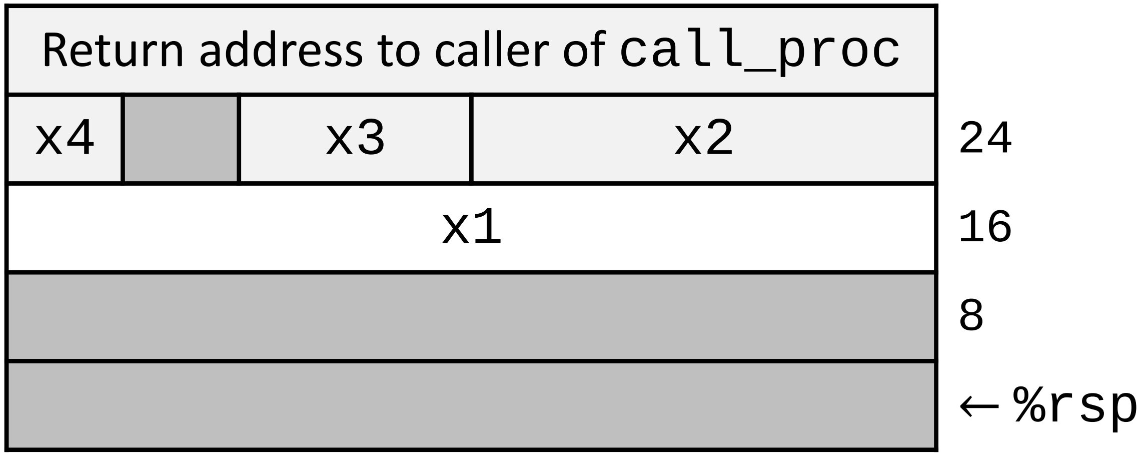
← `%rsp`

Note: Details may vary depending on compiler!

x86-64 Example (2) – Allocate local vars

```
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) # x1
  movl  $2,24(%rsp) # x2
  movw  $3,28(%rsp) # x3
  movb  $4,31(%rsp) # x4
  • • •
```

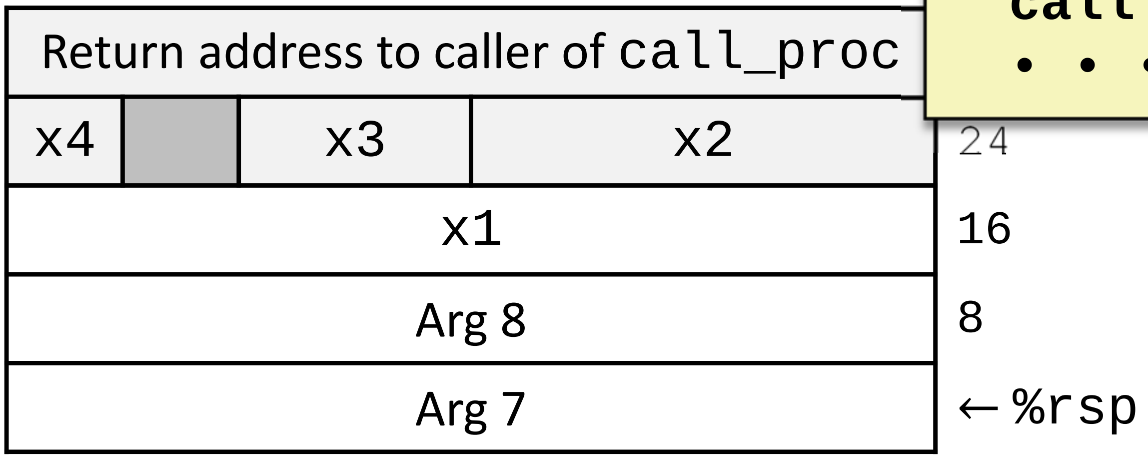


x86-64 Example (3) – setup params to `proc`

```
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 # %rcx=&x2
  leaq  16(%rsp),%rsi # %rsi=&x1
  leaq  31(%rsp),%rax # %rax=&x4
  movq  %rax,8(%rsp)  # arg8=&4
  movl  $4,(%rsp)     # arg7=4
  leaq  28(%rsp),%r9  # %r9=&x3
  movl  $3,%r8d     # %r8 = 3
  movl  $2,%edx     # %rdx = 2
  movq  $1,%rdi     # %rdi = 1
  call  proc
  • • •
```

Same instructions as in video, just a different order.



Return address to caller of `call_proc`

x4 | ▨ | x3 | x2 — 24

x1 — 16

Arg 8 — 8

Arg 7 — ← `%rsp`

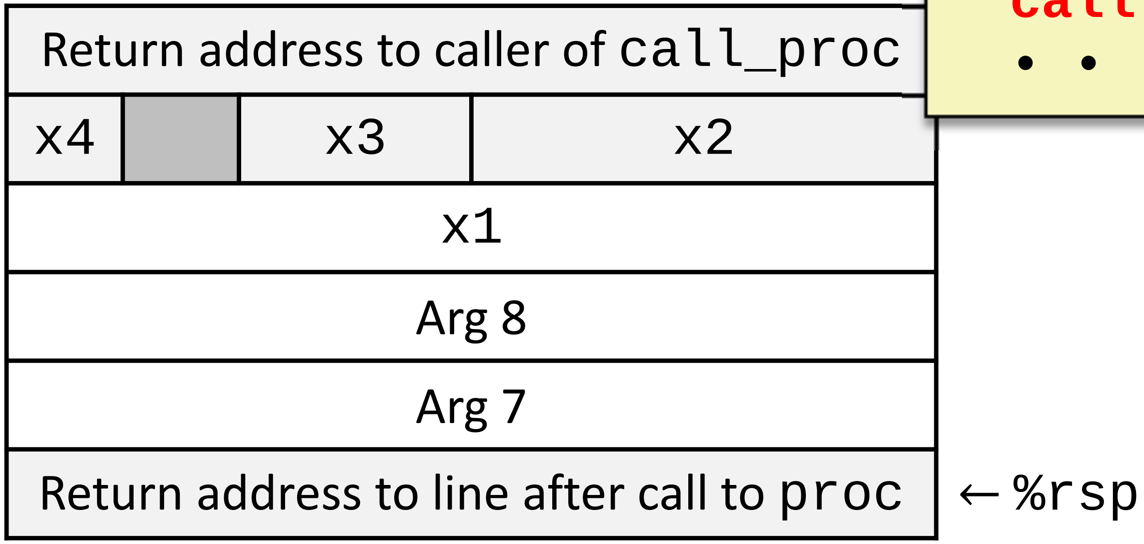
Argument order:

- Diane's Silk Dress Cost $8 9
- %rdi, %rsi, %rdx, %rcx, %r8, %r9

x86-64 Example (4) – call `proc`

```
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
  • • •
```



Return address to caller of `call_proc`

x4 | ▨ | x3 | x2

x1

Arg 8

Arg 7

Return address to line after call to `proc` ← `%rsp`

x86-64 Example (5) – after call to `proc`

```
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 # %eax=x3
  movsbl 31(%rsp),%edx # %edx=x4
  subl   %edx,%eax     # %eax=x3-x4
  cltq
  movslq 24(%rsp),%rdx # %rdx=x2
  addq   16(%rsp),%rdx #%rdx=x1+x2
  imulq  %rdx,%rax   # %rax=rax*rdx
  addq   $32,%rsp
  ret
```

<table>
<tr><td colspan="4">Return address to caller of <code>call_proc</code></td><td></td></tr>
<tr><td>x4</td><td>▨</td><td>x3</td><td>x2</td><td>24</td></tr>
<tr><td colspan="4">x1</td><td>16</td></tr>
<tr><td colspan="4">Arg 8</td><td>8</td></tr>
<tr><td colspan="4">Arg 7</td><td>← %rsp</td></tr>
</table>

`movs__`:
- Move and *sign* extend

`cltq`:
- Sign extend `%eax` into `%rax`
- *(special-case to save space)*

x86-64 Example (6) – de-allocate local vars

```
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
```

Return address to caller of `call_proc`	← `%rsp`