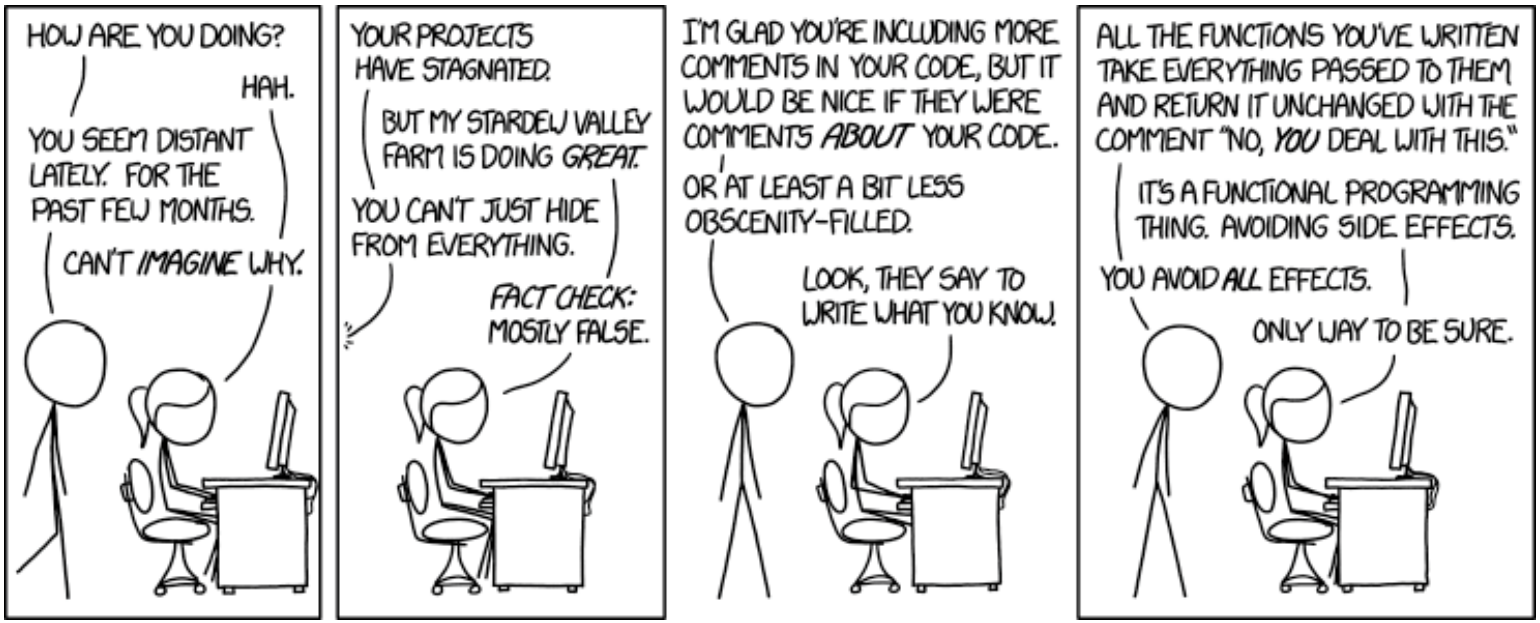
Procedures II

CSE 351 Spring 2019

Instructor: Ruth Anderson

Teaching Assistants:

Gavin Cai, Britt Henderson, Sophie Tian, Casey Xing, Jack Eggleston, Richard Jiang, Connie Wang, Chin Yeoh, John Feltrup, Jack Skalitzky, Sam Wolfson



http://xkcd.com/1790/

Administrivia

- Lab 2 (x86-64) due Wednesday (5/01)
- Homework 3, due Wednesday (5/8)
  - On midterm material, but due after the midterm

- **Midterm** (Fri 5/03, 4:30-5:30pm in KNE 130)

Example: increment

adding val to value store at p

rdi rsi

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

written this way to correspond to assembly

```
increment:
  movq    (%rdi), %rax    # move x=*p to the return value %rax
  addq    %rax, %rsi      # set val = val + x
  movq    %rsi, (%rdi)    # set *p = val
  ret
```

Dianes silk dress costed $89 : %rdi, %rsi, %rdx, %rcx, %r8, %r9

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

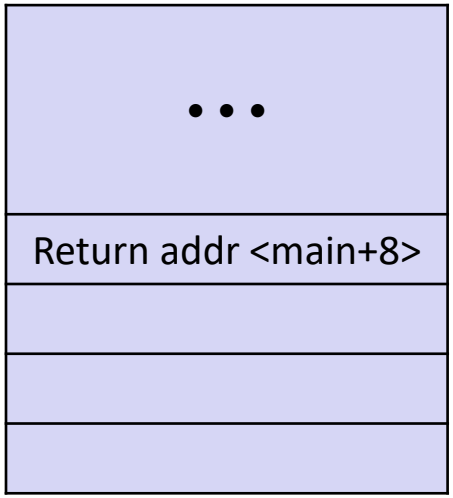
Procedure Call Example - Handout

```
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

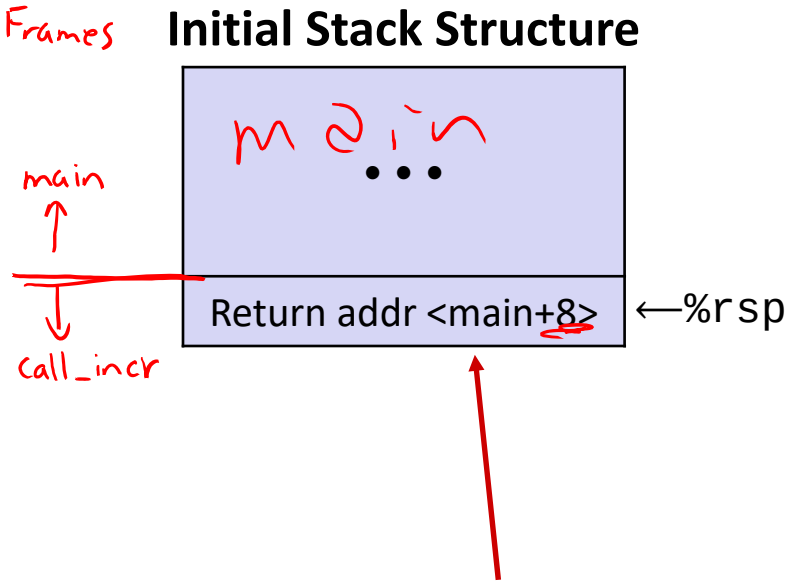


Register	Use(s)
%rdi	
%rsi	
%rax	

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
```



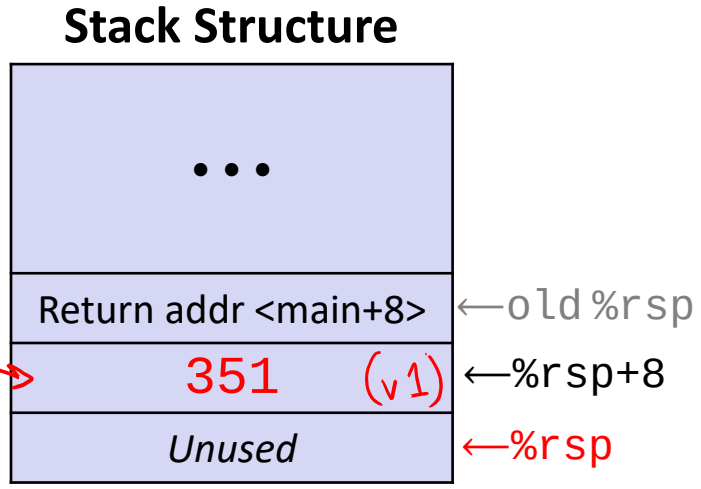
- Return address on stack is the address of instruction immediately *following* the call to "`call_incr`"
  - Shown here as `main`, but could be anything)
  - Pushed onto stack by `call call_incr`

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



- 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

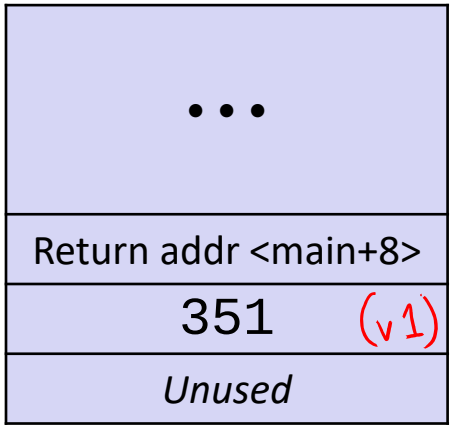
Procedure Call Example (step 2)

```
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      # set val
  leaq    8(%rsp), %rdi   # set p
  call    increment
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Set up parameters for call to `increment`

Stack Structure



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`.

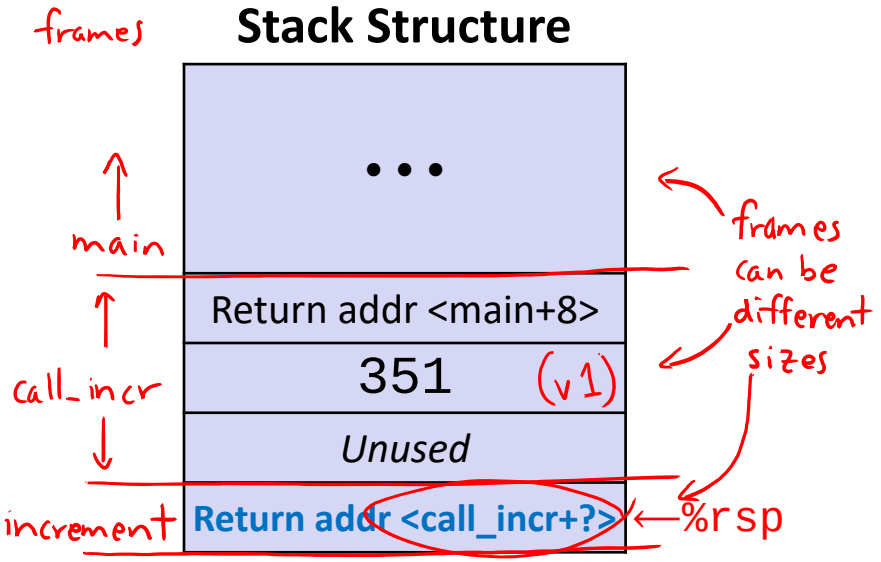
Register	Use(s)
%rdi	&v1
%rsi	100

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
```



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

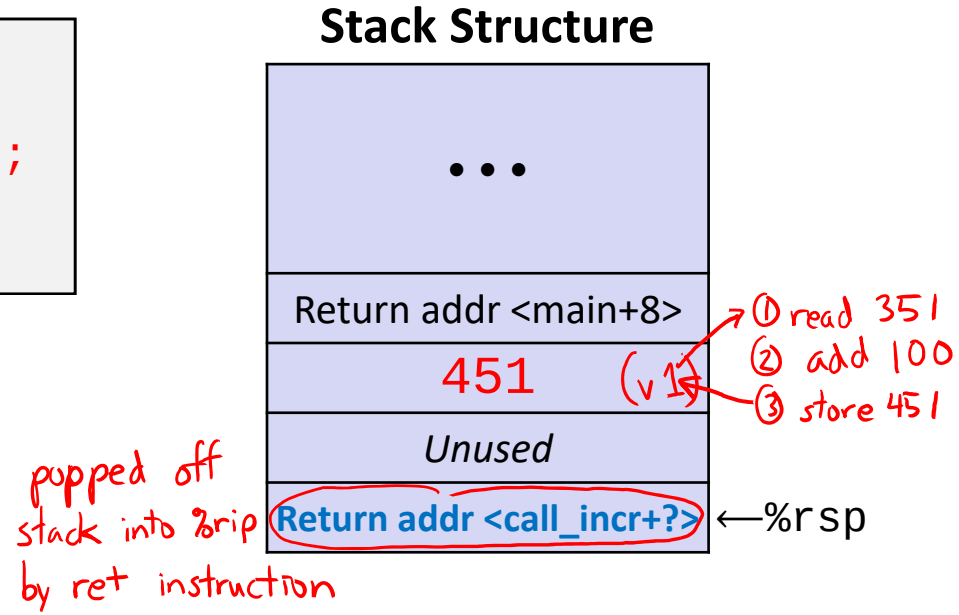
Register	Use(s)
%rdi	&v1
%rsi	100
%rax	

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:
① movq    (%rdi), %rax # x = *p
② addq    %rax, %rsi   # y = x+100
③ movq    %rsi, (%rdi) # *p = y
  ret
```



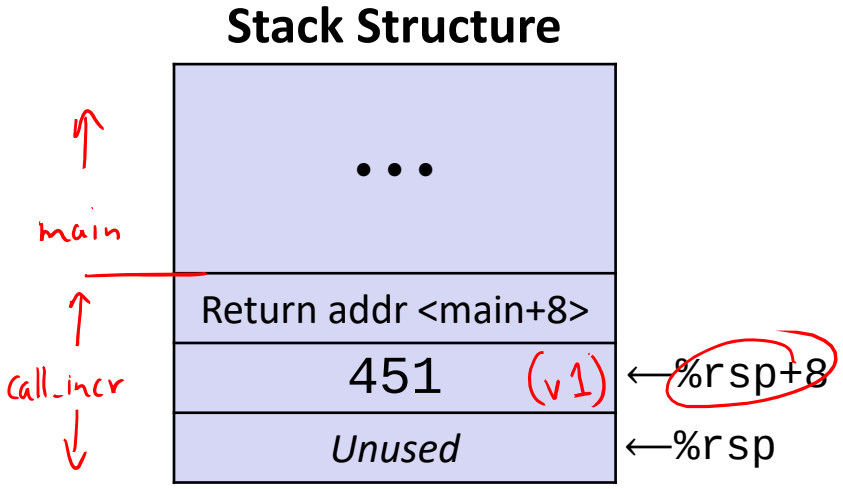
- State while inside `increment`
  - *After* code in body has been executed

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
```



- 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 (v2)

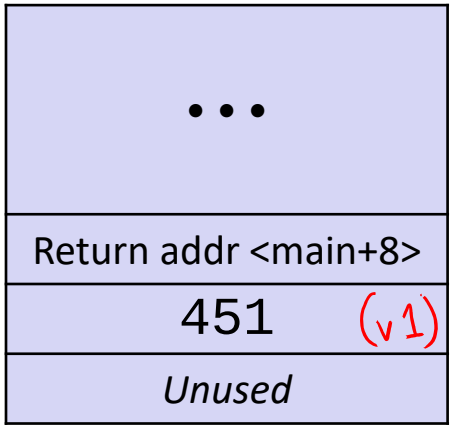
Procedure Call Example (step 6)

```
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



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

Procedure Call Example (step 7)

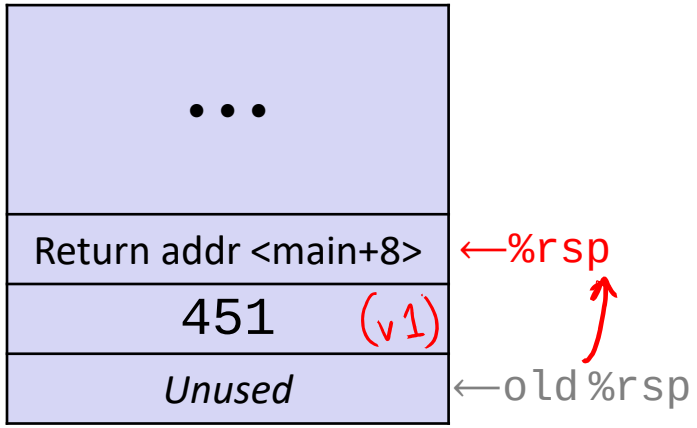
```
long call_incr() {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return v1 + v2;
}
```

```
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

(make sure %rsp points to return addr before ret)

Stack Structure

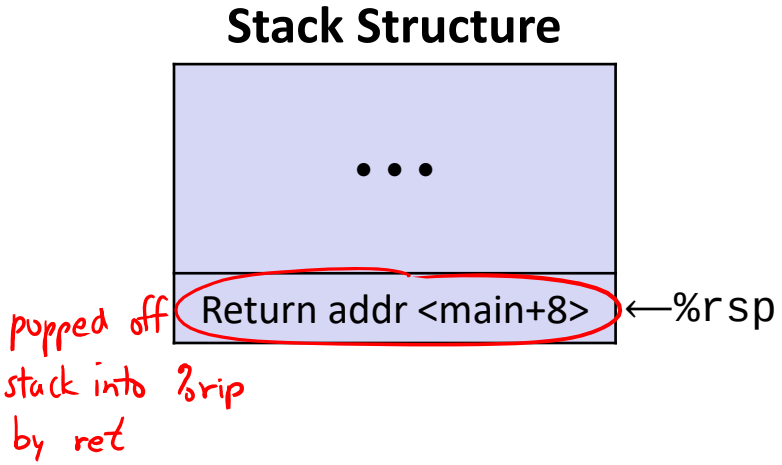


Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

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
```



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

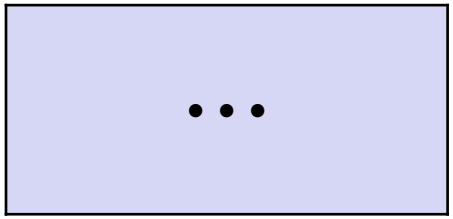
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

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

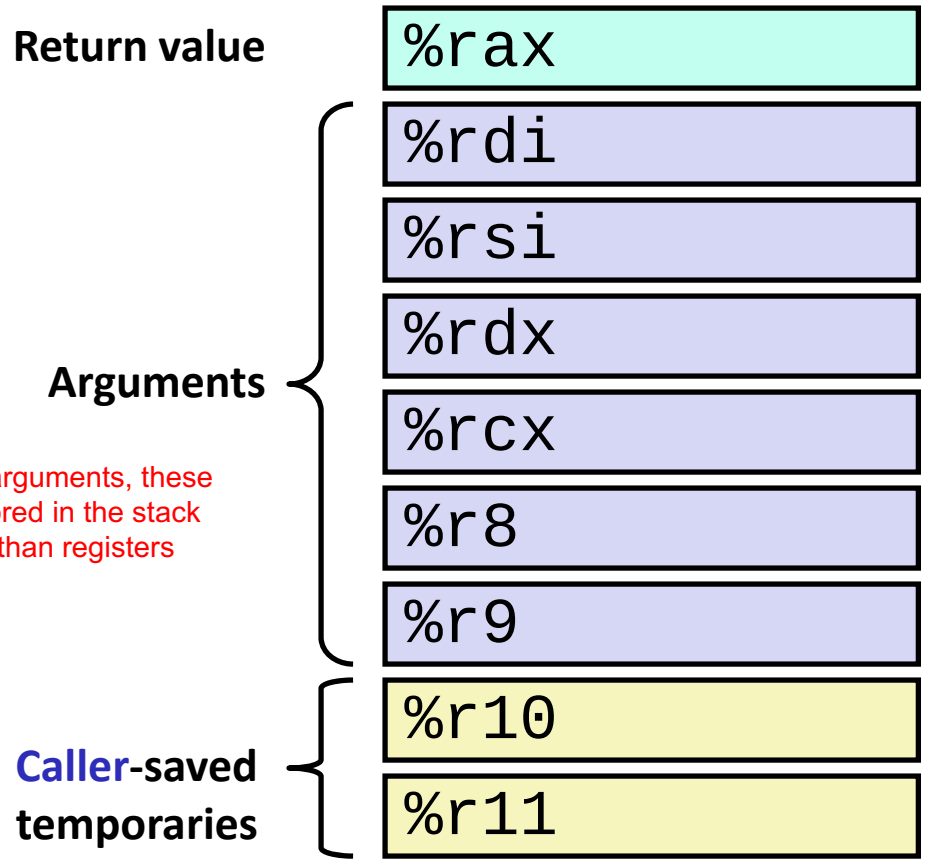
- ***"Caller-saved" registers***
  - It is the **caller**'s responsibility to save any important data in these registers before calling another procedure (*i.e.* the **callee** can freely change data in these registers)
  - **Caller** saves values in its stack frame before calling **Callee**, then restores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

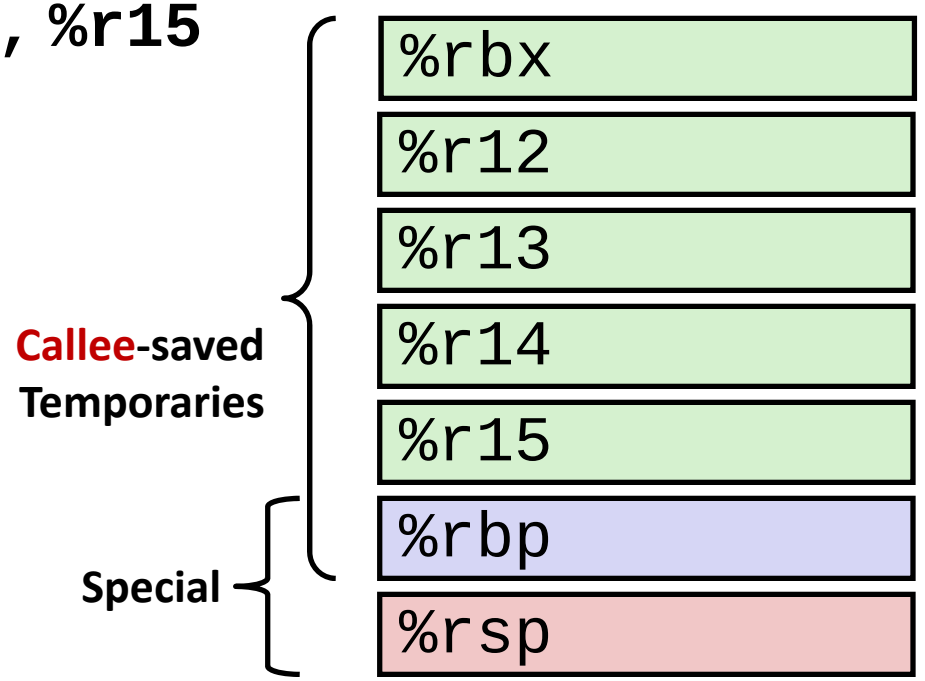
x86-64 Linux Register Usage, part 1

- **`%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



x86-64 Linux Register Usage, part 2

- **`%rbx, %r12, %r13, %r14, %r15`**
  - **Callee**-saved
  - **Callee** must save & restore
- **`%rbp`**
  - **Callee**-saved
  - **Callee** must save & restore
  - May be used as frame pointer
  - Can mix & match
- **`%rsp`** Stack pointer; points to top of the stack
  - Special form of **callee** save
  - Restored to original value upon exit from procedure



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)

focused on this interaction: main → call_incr2 → increment

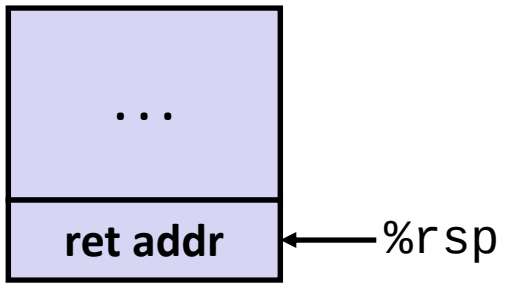
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

need x (in %rdi) after procedure call

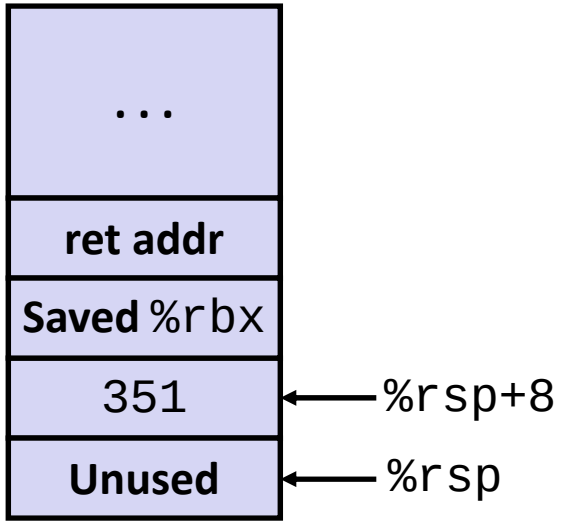
```
call_incr2:
  pushq   %rbx          ← save old %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx    ← change %rbx
  movq    $351, 8(%rsp)
  movl    $100, %esi
  leaq    8(%rsp), %rdi
  call    increment     across procedure call
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

assumed the same

Initial Stack Structure



Resulting Stack Structure



Callee-Saved Example (step 2)

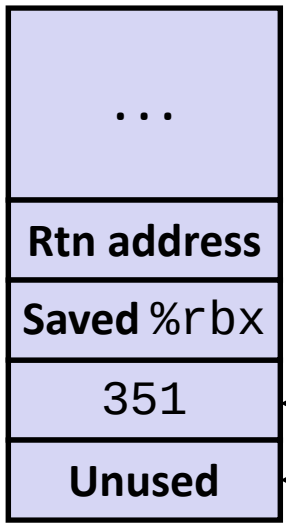
```
long call_incr2(long x) {
    long v1 = 351;
    long v2 = increment(&v1, 100);
    return x + v2;
}
```

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

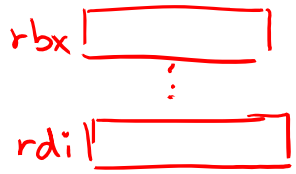
stack discipline:
add/sub
push/pop
must be symmetric within procedure

Memory

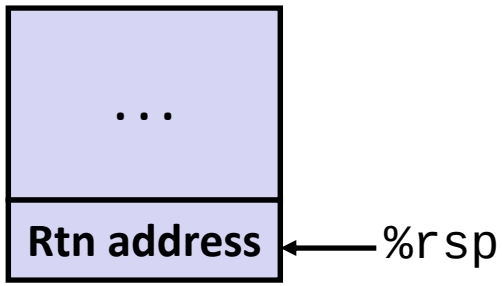
Stack Structure



Registers



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

logical right shift

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

stop once all 1's shifted off

value of LSB

shift off LSB and recurse

counts all 1's in binary representation of x

Compiler Explorer:

https://godbolt.org/z/xFCrsw

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

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  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

prepare return val of 0

```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

jump to .L if x & x = 0

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

(don't worry about it)

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);
}
```

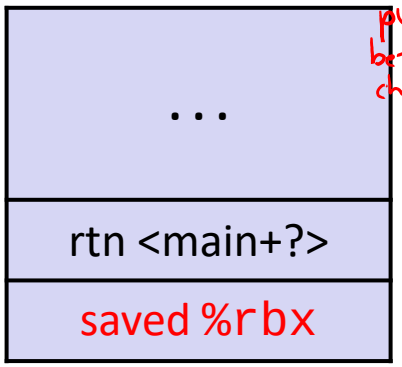
need x across procedure call

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



push before changing

store "x" for this stack frame

pop/restore before returning

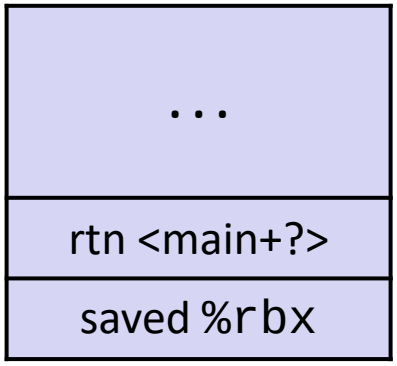
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

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

The Stack




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    $1, %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

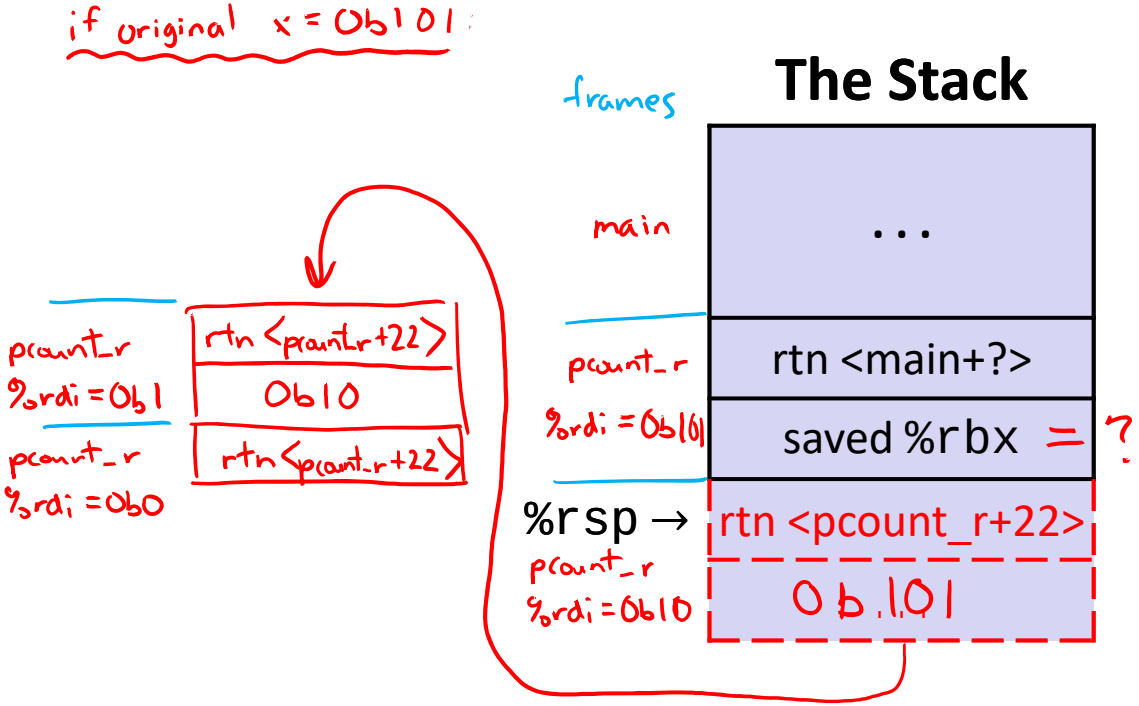
implicit

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
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

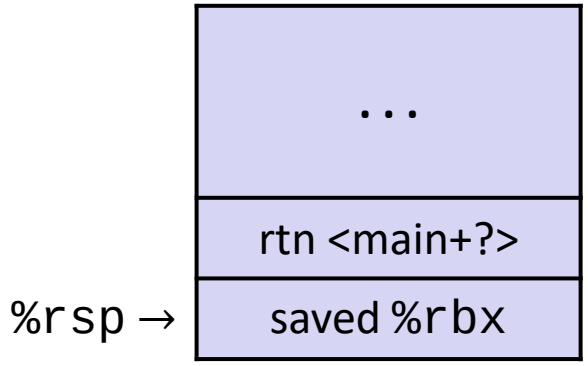


Recursive Function: Result

```
/* 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
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

across

x

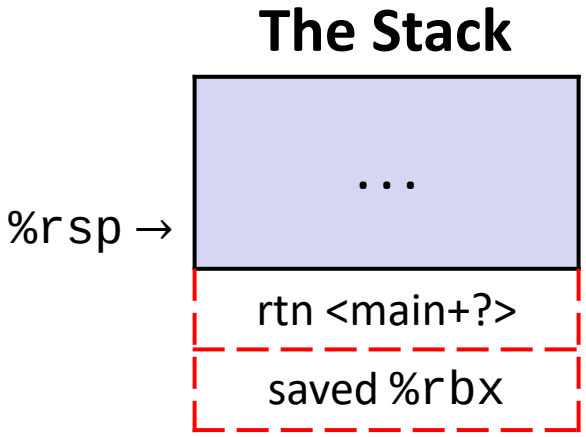
x

assumed the same

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




```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  jne     .L8
  rep ret
.L8
  pushq   %rbx
  movq    %rdi, %rbx
  shrq    %rdi
  call    pcount_r
  andl    $1, %ebx
  addq    %rbx, %rax
  popq    %rbx
  ret
```

restore before returning

Observations About Recursion

The stack naturally provides a recursive framework

- Works without any special consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return address
  - 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 have a minimal stack frame
  - Only return address is pushed onto the stack when procedure is called

- A procedure *needs* to grow its stack frame when it:
  - Has too many local variables to hold in **caller**-saved registers e.g. > 6 args
  - Has local variables that are arrays or structs e.g. that cannot fit in a single register
  - 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