

The Hardware/Software Interface

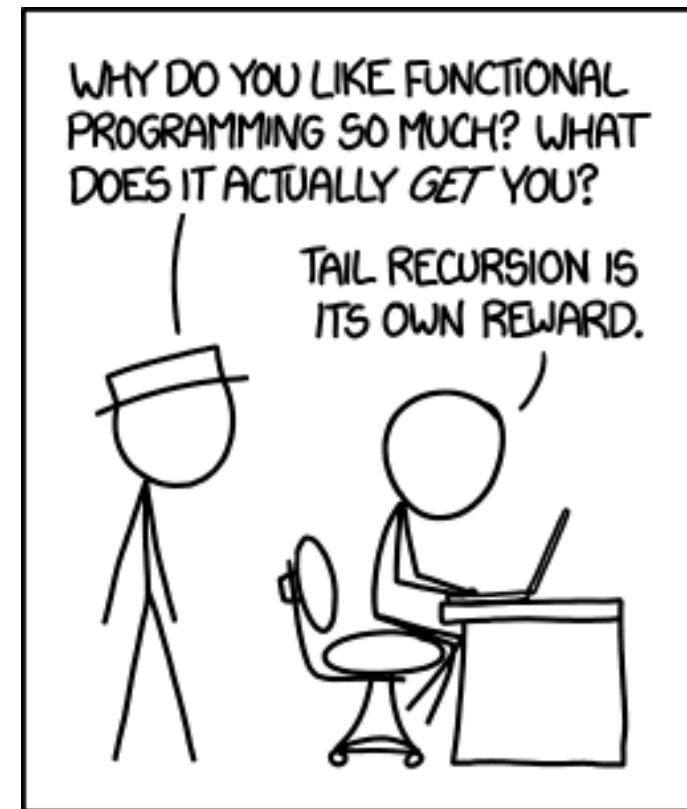
Procedures II

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus	Divya Ramu
Grace Zhou	Jessie Sun
Jiuyang Lyu	Kanishka Singh
Kurt Gu	Liander Rainbolt
Mendel Carroll	Ming Yan
Naama Amiel	Pollux Chen
Rose Maresh	Soham Bhosale
Violet Monserate	



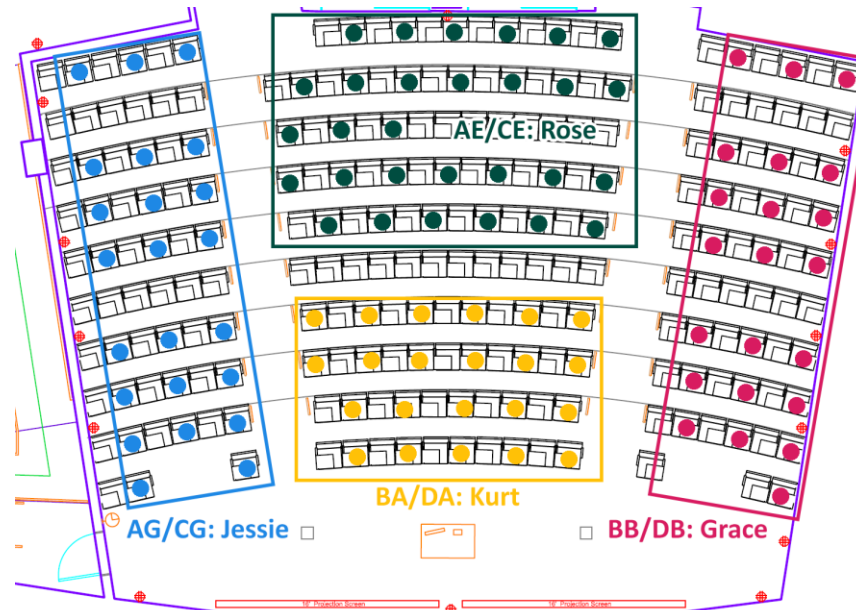
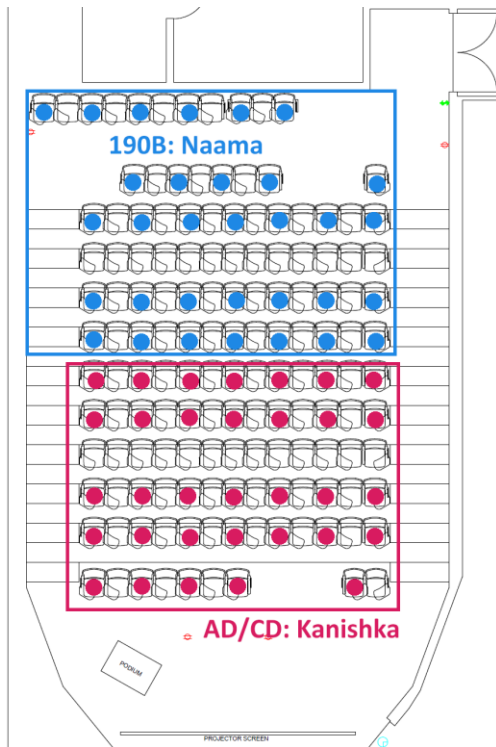
<http://xkcd.com/1270/>

Relevant Course Information

- ❖ Lab 1b grades released later this week
 - Regrade requests open ~24 hours after grade release (rounded to 12:00 am), close ~72 hours after grade release (rounded to 11:59 pm)
- ❖ Lab 2 due Friday (10/24)
 - Since you are submitting a text file (`defuser.txt`), there won't be any Gradescope autograder output about compilation this time – check the Code tab after submission to make sure that everything looks right
 - Extra credit (bonus) needs to be submitted to the extra credit assignment

Relevant Course Information (cont.)

- ❖ Midterm: 10/27, 5:30 pm, location dependent on section
 - Practice with midterm reference sheet, cheat sheet (letter-size)
 - Review session this Friday (10/24) 4:30-6:20pm in CSE2 G01
 - Details on course website exams page



SIG 134, CSE2 G20, ARC 147



Lecture Outline (1/3)

- ❖ **Stack Frame Details**
- ❖ x86-64 Register Saving Conventions
- ❖ Recursion

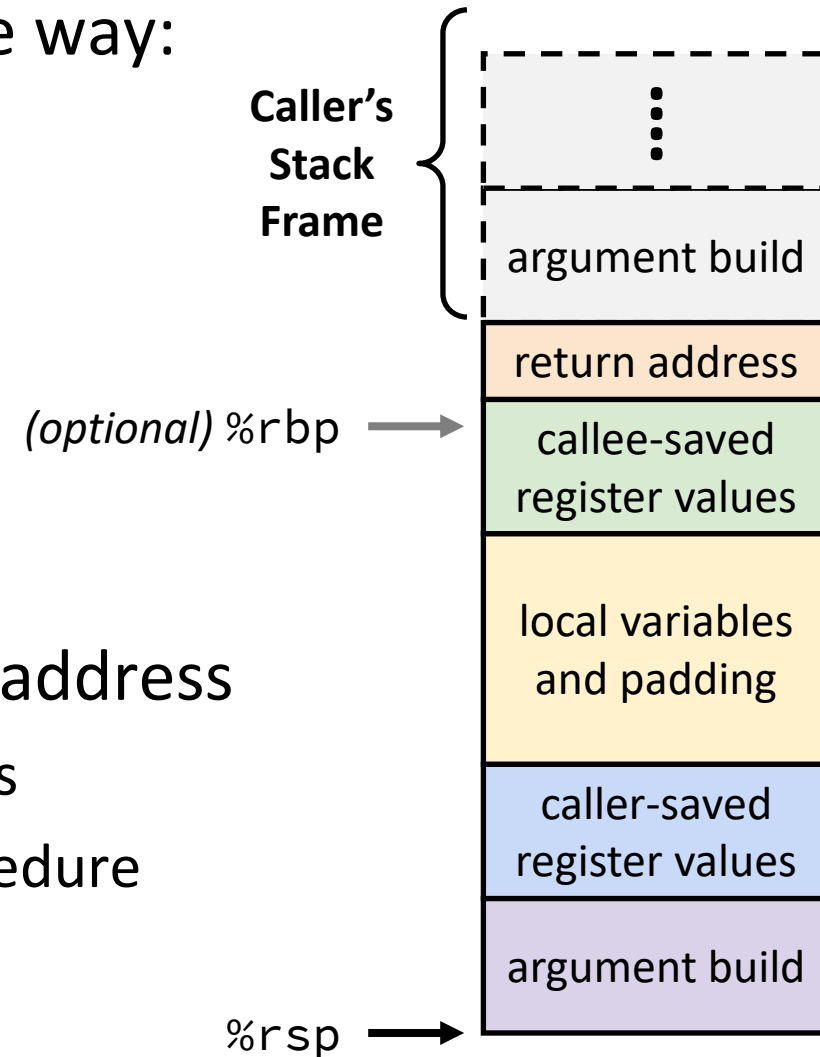
x86-64 Stack Frame Structure

❖ Each stack frame organized in the same way:

- 1) Return address pushed by `call`
- 2) Callee-saved registers
- 3) Local variables
- 4) Caller-saved registers
- 5) Argument build (arguments 7+)

❖ The only required section is the return address

- Minimal stack frame is just a return address
- Others depend on the specifics of the procedure
 - Avoid using memory if you can



Stack Frame Details Example

- ❖ **Callee:** mem_add adds val to current value pointed to by p

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

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

- ❖ **Caller:** Calls mem_add, returns arbitrary value

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

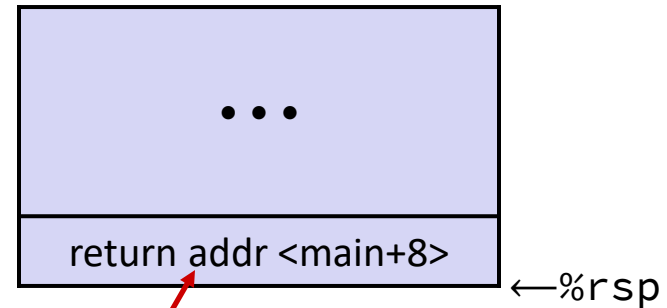
```
call_mem_add:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    mem_add  
    addq    8(%rsp), %rax  
    addq    $16, %rsp  
    ret
```

Stack Frame Details Example (Initial State)

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

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

Initial Stack Structure



- ❖ Return address is to instruction *after* `call call_mem_add`
 - Here `main`, but could be anything

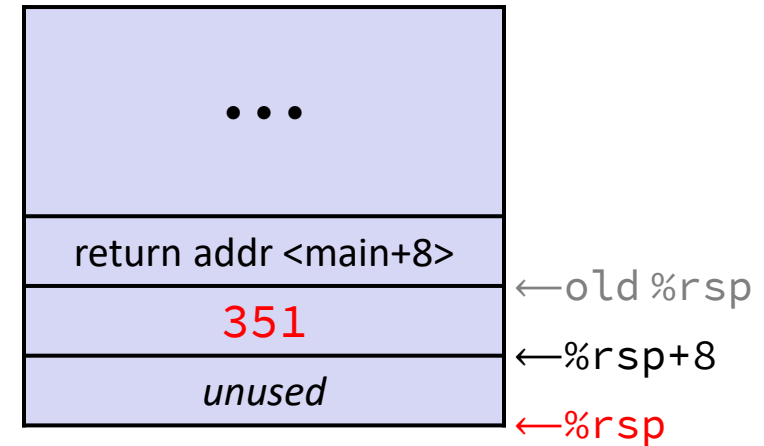
Stack Frame Details Example (Step 1)

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

```
call_mem_add:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    mem_add  
    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 commonly allocates extra space for a variety of reasons, including alignment

Stack Frame Details Example (Step 2)

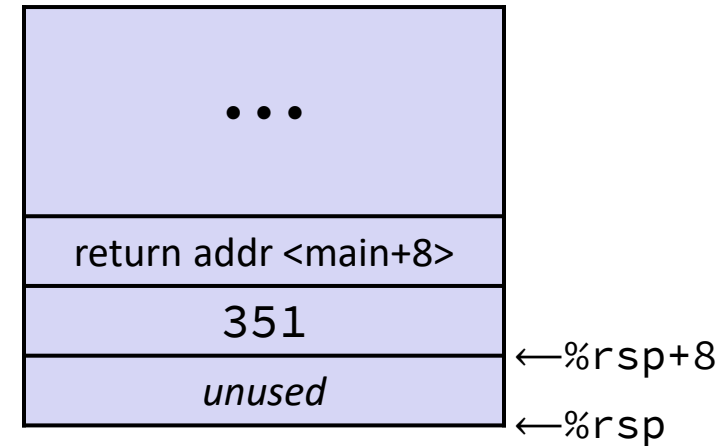
```
long call_mem_add() {  
    long v1 = 351;  
    long v2 = mem_add(&v1, 100);  
    return v1 + v2;  
}
```

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

} Set up parameters for call
to increment

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

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`.

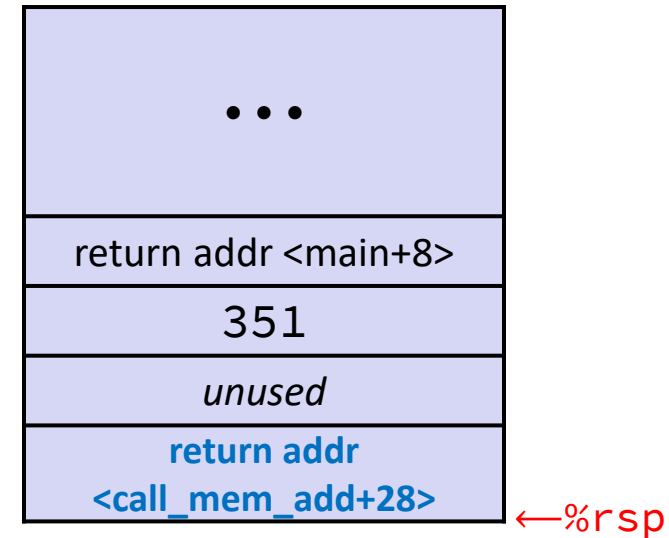
Stack Frame Details Example (Step 3)

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

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

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

Stack Structure



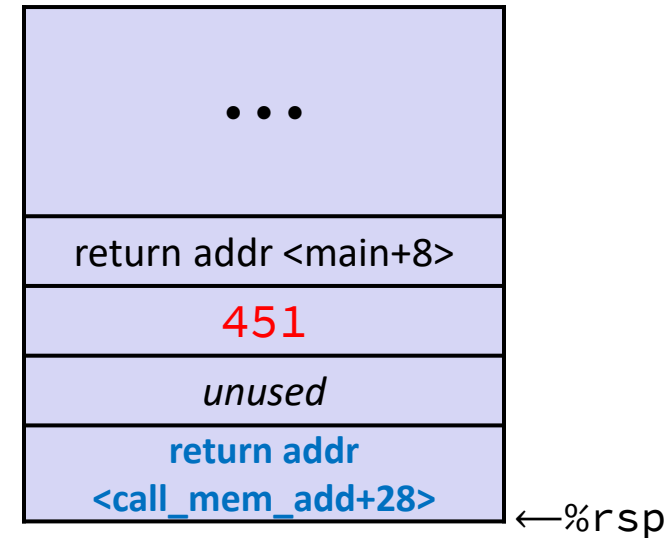
- ❖ Entering mem_add
 - **Return address** on top of stack is address of the addq instruction after call mem_add

Stack Frame Details Example (Step 4)

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

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

Stack Structure



- ❖ State *after* body of `mem_add` has been executed

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

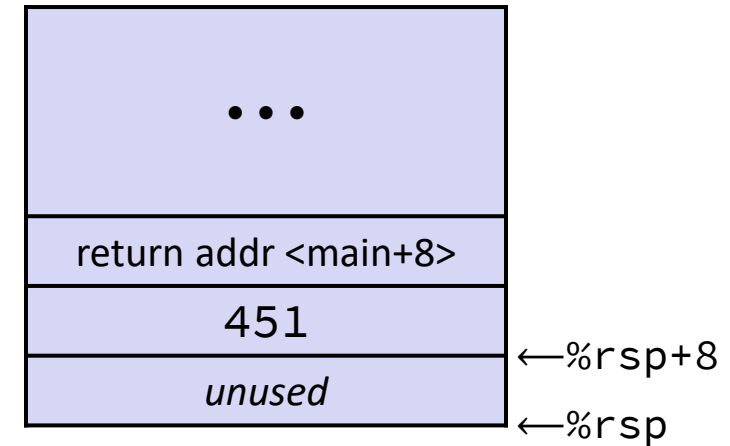
Stack Frame Details Example (Step 5)

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

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

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

Stack Structure



- ❖ After returning from `mem_add`
 - Registers and memory have been modified and return address has been popped off stack

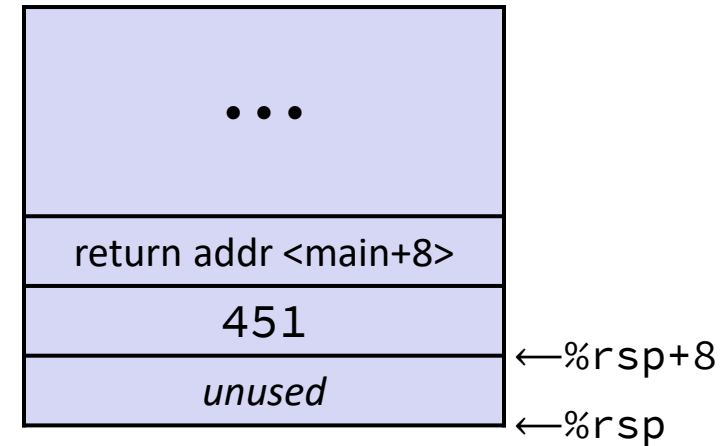
Stack Frame Details Example (Step 6)

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

```
call_mem_add:  
    subq    $16, %rsp  
    movq    $351, 8(%rsp)  
    movl    $100, %esi  
    leaq    8(%rsp), %rdi  
    call    mem_add  
    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

Stack Frame Details Example (Step 7)

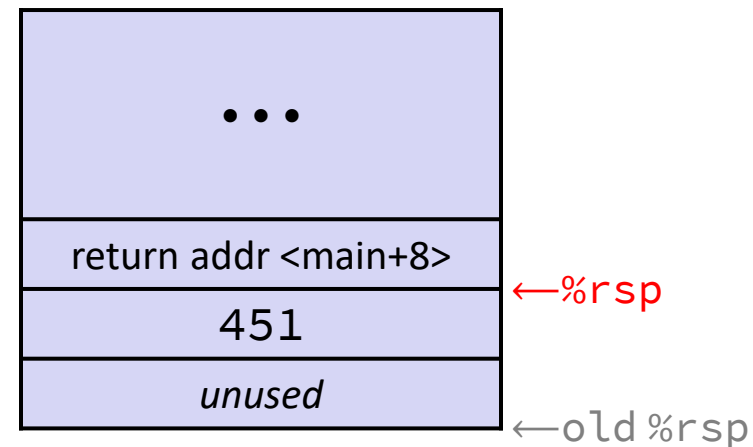
```
long call_mem_add() {  
    long v1 = 351;  
    long v2 = mem_add(&v1, 100);  
    return v1 + v2;  
}
```

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

← De-allocate space for local vars

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

Stack Structure



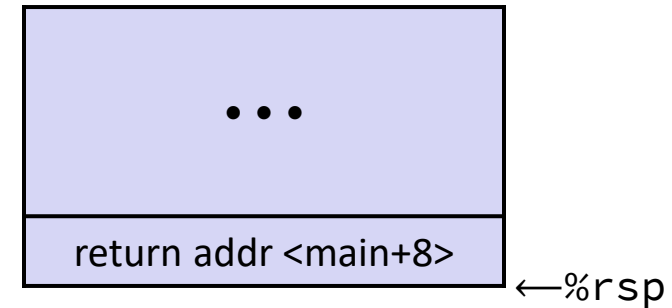
Stack Frame Details Example (Step 8)

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

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

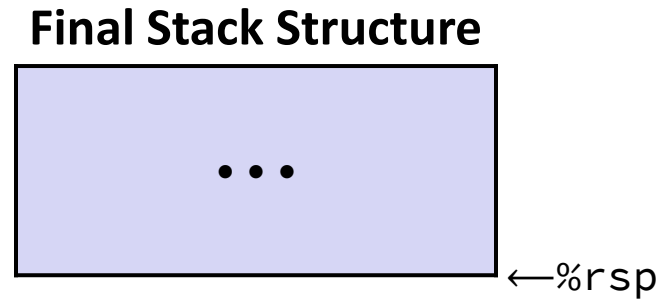
Register	Use(s)
%rdi	&v1
%rsi	451
%rax	802

Stack Structure



❖ State *just before* returning from `call_mem_add`

Stack Frame Details Example (Final State)



- ❖ State *after* returning from `call_mem_add`
 - Return address has been popped off stack (stack frame fully deallocated)
 - Return value in %rax

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

Polling Questions

- ❖ In the following function, how big is the stack frame?
Which instruction(s) pertain to the **local variables** and **saved registers** portions of its stack frame?

```
call_mem_add2:
1  pushq    %rbx
2  subq     $16, %rsp
3  movq     %rdi, %rbx
4  movq     $351, 8(%rsp)
5  movl     $100, %esi
6  leaq     8(%rsp), %rdi
7  call     mem_add
8  addq     %rbx, %rax
9  addq     $16, %rsp
10 popq     %rbx
11 ret
```

Lecture Outline (2/3)

- ❖ Stack Frame Details
- ❖ **x86-64 Register Saving Conventions**
- ❖ Recursion

Register Saving Convention Motivation

- ❖ Assume procedure foo (caller) calls bar (callee):

```
foo:
...
movq $351, %rdx
call bar
addq %rdx, %rax
...
ret
```

```
bar:
...
movq $333, %rdx
...
ret
```

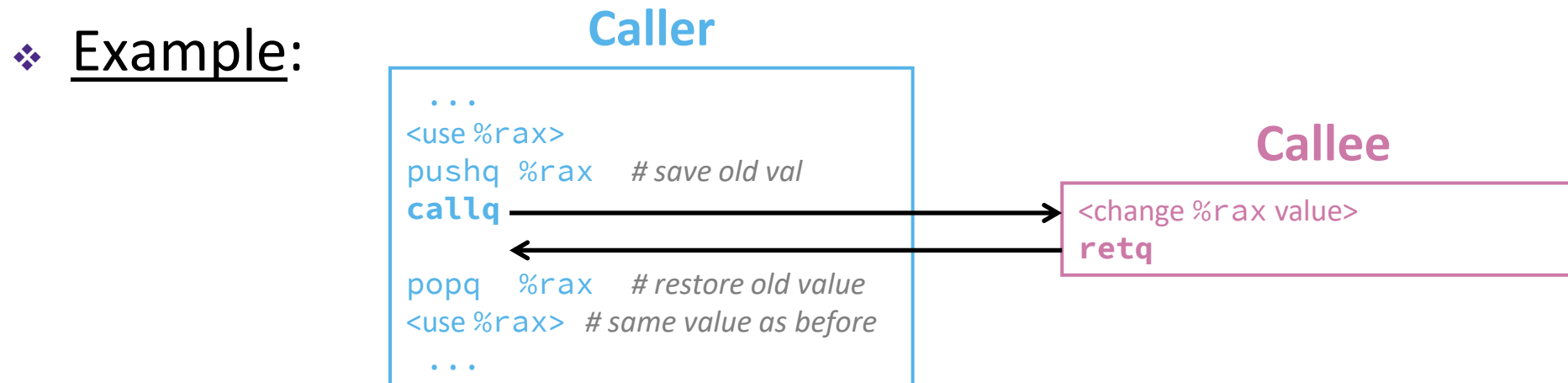
- ❖ What guarantees does foo have about register values across this call?
 - None! Contents of register %rdx may be overwritten by bar!
 - This could be trouble – something should be done. Either:
 - Caller should save %rdx before the call (and restore it after the call)
 - Callee should save %rdx before using it (and restore it before returning)

Register Saving Conventions (Review)

- ❖ ***Register saving conventions*** dictate how we should deal with register reuse to avoid accidentally destroying another procedure's data
 - All procedures have access to *all* of the general-purpose registers
 - Registers will be designated as either **callee**-saved or **caller**-saved based on who's responsibility it is to save (and restore) the old value
- ❖ Two options to save register values:
 - 1) Push the old value onto the stack
 - 2) Copy the old value into a register of the opposite type

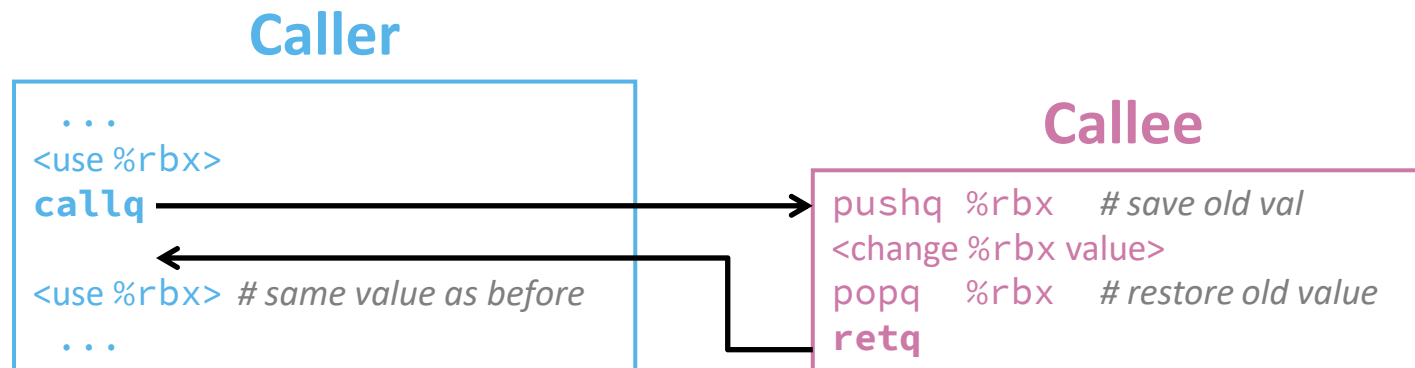
Caller-Saved Registers (Review)

- ❖ It is the caller's responsibility to save any important/needed data in these registers before calling another procedure
 - Saving is done just before calling the callee and restoring is done right after the call
 - The callee can freely change data in these registers
- ❖ In x86-64:
 - Calling convention-related registers: %rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9
 - Others: %r10, %r11



Callee-Saved Registers (Review)

- ❖ It is the **callee**'s responsibility to save any data in these registers before using the registers
 - Saving is done early in procedure (before use) and restoring is done just before returning to **caller**
 - The **caller** assumes the data will be the same across the **callee** procedure call
- ❖ In x86-64:
 - "Pointer" registers restored before frame deallocates: %rsp, %rbp
 - Others: %rbx, %r12, %r13, %r14, %r15
- ❖ Example:



x86-64 Linux Register Usage (Review)

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

Register Saving Example

```
long call_mem_add2(long x) {  
    long v1 = 351;  
    long v2 = mem_add(&v1, 100);  
    return x + v2;  
}
```

```
call_mem_add2:  
    pushq    %rbx  
    subq     $16, %rsp  
    movq     %rdi, %rbx  
    movq     $351, 8(%rsp)  
    movl     $100, %esi  
    leaq     8(%rsp), %rdi  
    call     mem_add  
    addq     %rbx, %rax  
    addq     $16, %rsp  
    popq     %rbx  
    ret
```

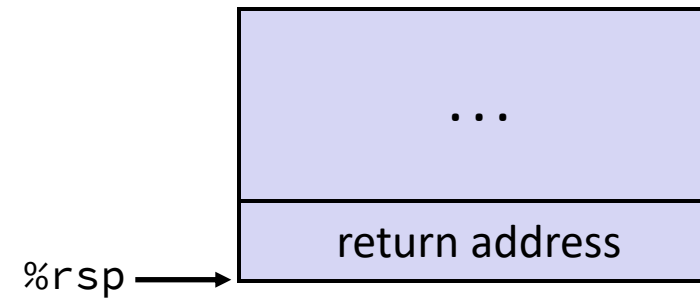
- ❖ Modified version of `call_mem_add` from stack frames details example
 - Now need `x` (in `%rdi`) after procedure call
 - Chooses to save `%rdi` by copying into `%rbx`
 - Chooses to save `%rbx` (used in procedure) by pushing to stack

Register Saving Example: Pre-Call

```
long call_mem_add2(long x) {  
    long v1 = 351;  
    long v2 = mem_add(&v1, 100);  
    return x + v2;  
}
```

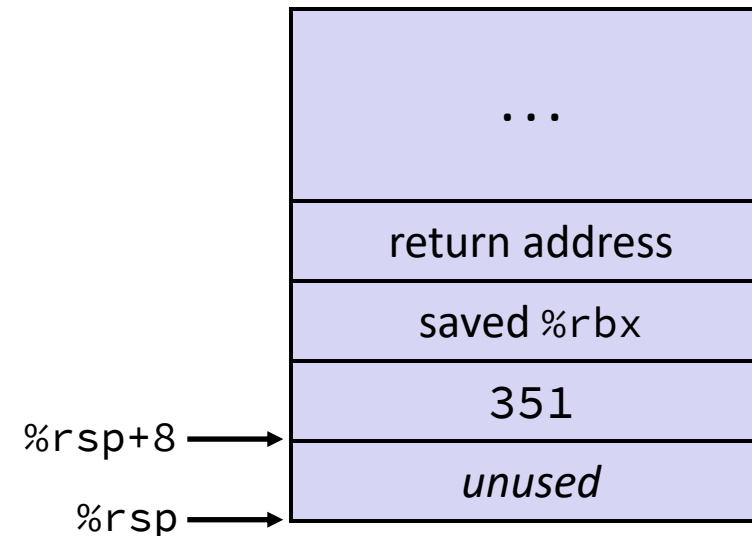
```
call_mem_add2:  
    pushq    %rbx  
    subq     $16, %rsp  
    movq     %rdi, %rbx  
    movq     $351, 8(%rsp)  
    movl     $100, %esi  
    leaq     8(%rsp), %rdi  
    call     mem_add  
    addq     %rbx, %rax  
    addq     $16, %rsp  
    popq     %rbx  
    ret
```

Initial Stack Structure



Register	Value
<code>%rdi</code>	<code>x</code>
<code>%rbx</code>	<code>???</code>

Pre-Call Stack Structure



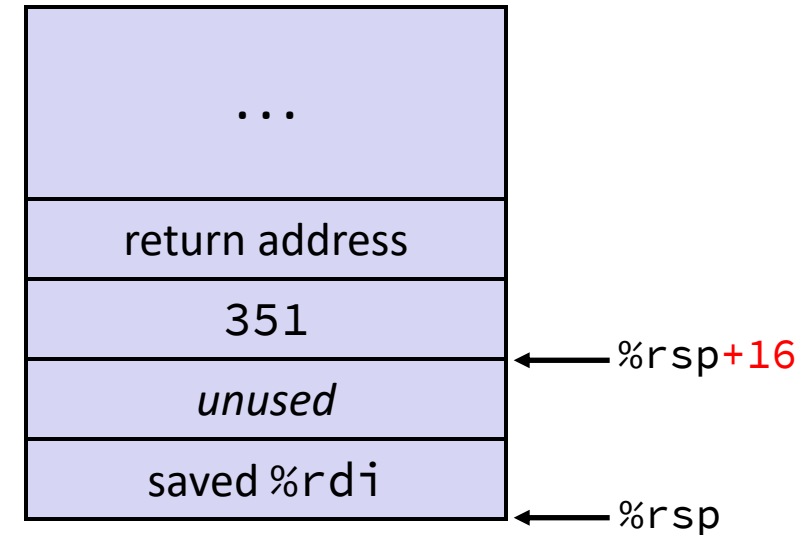
Register	Value
<code>%rdi</code>	<code>&v1</code>
<code>%rbx</code>	<code>x</code>

Alternate Register Saving Example

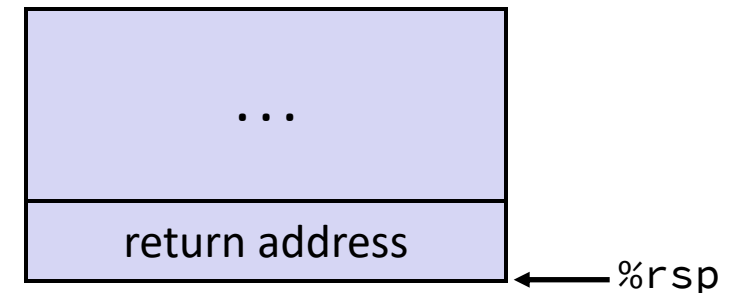
- ❖ Alternate formulation of `call_mem_add2` that saves `%rdi` to the stack instead:

```
call_mem_add3:
    subq    $16, %rsp
    pushq   %rdi
    movq    $351, 16(%rsp)
    movl    $100, %esi
    leaq    16(%rsp), %rdi
    call    mem_add
    popq    %rdi
    addq    %rdi, %rax
    addq    $16, %rsp
    ret
```

Stack Structure



Pre-return Stack Structure



Why Caller and Callee Saved?

- ❖ Neither caller-saved nor callee-saved is always “best”:
 - If caller isn’t using a register, caller-saved is better
 - If callee doesn’t need a register, callee-saved is better
 - If “do need to save”, callee-saved generally makes smaller programs
 - Functions are called from multiple places
- ❖ We want *one* calling convention to simply separate implementation details between caller and callee
 - So “some of each” and compiler tries to “pick registers” that minimize amount of saving/restoring

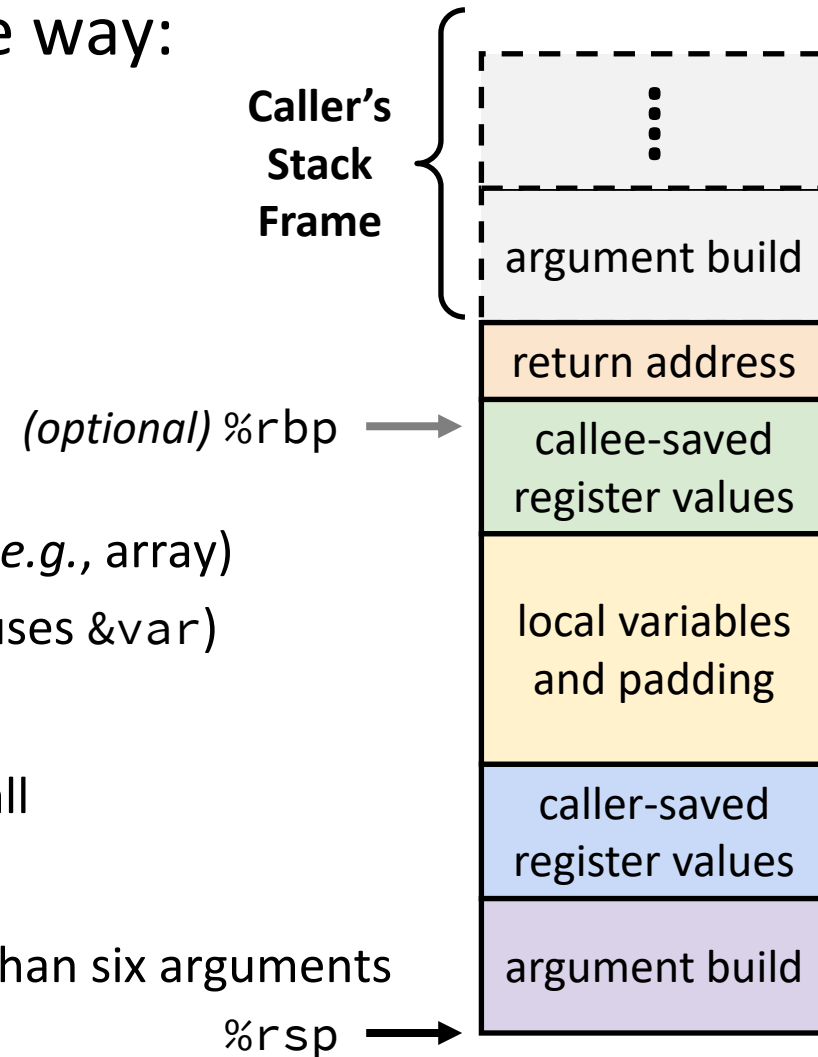
x86-64 Stack Frame Memory Usage (Review)

- ❖ 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 local variables that are arrays or structs
 - Passing in a pointer to a local variable during a function call (e.g. `func(&var)`)
 - 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 Stack Frame Structure (Complete)

❖ Each stack frame organized in the same way:

- 1) Return address pushed by `call`
 - The address of the instruction after `call`
- 2) Callee-saved registers
 - Only if procedure modifies/uses them
- 3) Local variables
 - Unavoidable if variable is too big for a register (*e.g.*, array)
 - Unavoidable if variable needs an address (*i.e.*, uses `&var`)
- 4) Caller-saved registers
 - Only if values are needed *across* a procedure call
- 5) Argument build
 - Only if procedure calls a procedure with more than six arguments



Lecture Outline (3/3)

- ❖ Stack Frame Details
- ❖ x86-64 Register Saving Conventions
- ❖ **Recursion**

Recursion (Review)

- ❖ Works without any additional special considerations!
 - Stack frames mean that each function call has private storage
 - Saved return address, registers, and local variables
 - Register saving conventions generally prevent one function call from corrupting another's data
 - Lifetime of each recursive call matches Last-In, First-Out (LIFO) structure of the stack
- ❖ Tracing recursive code can be a bit harder
 - All return addresses and stack frames will look similar/the same

Recursive Example: Popcount

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

```
pcount_r:
    testq    %rdi, %rdi
    jne      .L8
    movl     $0, %eax
    ret
.L8:
    pushq    %rbx
    movq     %rdi, %rbx
    andl     $0x1, %ebx
    shrq     $1, %rdi
    call     pcount_r
    addq     %rbx, %rax
    popq     %rbx
    ret
```

- ❖ Counts the 1's in the binary representation of x
 - $5 == 0b101$
 - `pcount_r(5) => 2`
 - <https://godbolt.org/z/xnd5rqeh9>
 - Compiled with `-Og` for more natural instruction ordering
- ❖ Register usage:
 - Need $x \& 1$ (`%rbx`) after return from `pcount_r(x >> 1)`
 - Chooses to save `%rbx` by pushing to stack (only in recursive case)

Recursive Example: Follow-along version

```

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

```

*You have a version of this on your handout!
Follow along as we trace the code*

```

pcount_r:
    testq    %rdi, %rdi
    jne      .L8
    movl     $0, %eax
    ret
.L8:
    pushq    %rbx
    movq     %rdi, %rbx
    andl     $0x1, %ebx
    shrq     $1, %rdi
    call     pcount_r
    addq     %rbx, %rax
    popq     %rbx
    ret

```

Register	Value (x=5)	Value (x=2)	Value (x=1)	Value (x=0)
%rdi				
%rbx				
%rax				

The Stack

...
return addr <main+?>

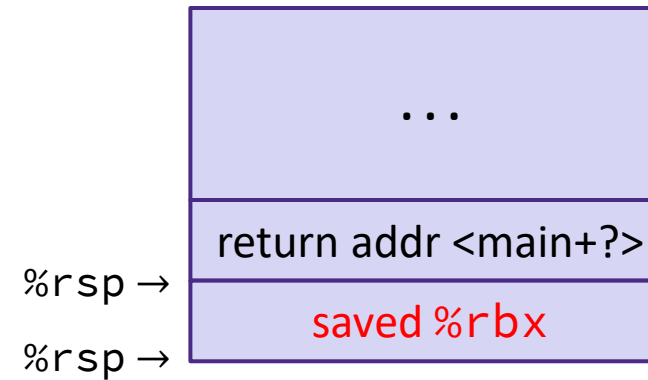
Recursive Example: Register Saving (x = 5)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rdi	5 (x)	caller-saved (reg)
%rbx	5 (old x)	callee-saved (stack)

The Stack



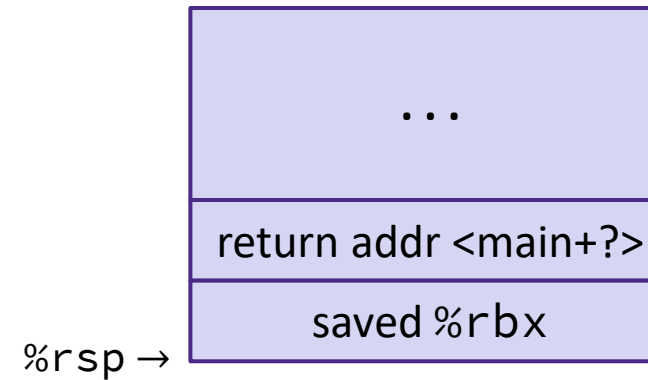
Recursive Example: Call Setup (x = 5)

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

```
pcount_r:
    testq    %rdi, %rdi
    jne      .L8
    movl     $0, %eax
    ret
.L8:
    pushq    %rbx
    movq     %rdi, %rbx
    andl     $0x1, %ebx
    shrq     $1, %rdi
    call     pcount_r
    addq     %rbx, %rax
    popq     %rbx
    ret
```

Register	Value	Save Status
%rdi	5 (x)	caller-saved (reg)
%rbx	1 (x&1)	callee-saved (stack)

The Stack



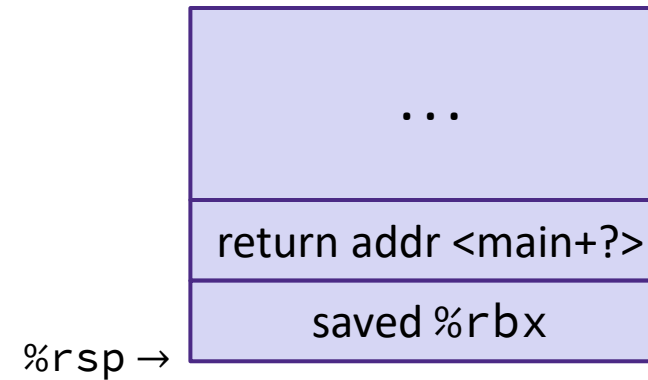
Recursive Example: Call Setup (x = 5)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rdi	2 (x>>1)	caller-saved (reg)
%rbx	1 (x&1)	callee-saved (stack)

The Stack



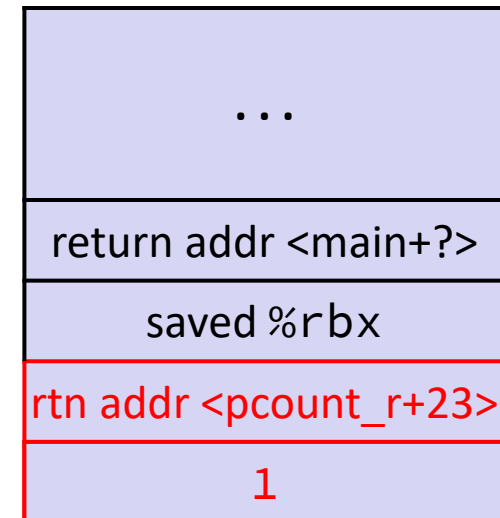
Recursive Example: Recursive Call (x = 2)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rdi	1 (x>>1)	caller-saved (reg)
%rbx	0 (x&1)	callee-saved (stack)

The Stack



Recursive Example: Recursive Call (x = 1)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne     .L8  
    movl    $0, %eax  
    ret  
.L8:  
    pushq   %rbx  
    movq    %rdi, %rbx  
    andl    $0x1, %ebx  
    shrq    $1, %rdi  
    call    pcount_r  
    addq    %rbx, %rax  
    popq    %rbx  
    ret
```

Register	Value	Save Status
%rdi	0 (x>>1)	caller-saved (reg)
%rbx	1 (x&1)	callee-saved (stack)

The Stack

...
return addr <main+?>
saved %rbx
rtn addr <pcount_r+23>
1
rtn addr <pcount_r+23>
0

Recursive Example: Base Case (x = 0)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rdi	0	n/a
%rbx	1	n/a
%rax	0 (ret val)	n/a

The Stack

...
return addr <main+?>
saved %rbx
rtn addr <pcount_r+23>
1
rtn addr <pcount_r+23>
0
rtn addr <pcount_r+23>

Recursive Example: Result (x = 1)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rbx	1 (x&1)	callee-restored
%rax	1 (ret val)	n/a

The Stack

...
return addr <main+?>
saved %rbx
rtn addr <pcount_r+22>
1
rtn addr <pcount_r+22>
0

Recursive Example: Result (x = 1)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rbx	0 (x&1)	callee-restored
%rax	1 (ret val)	n/a

The Stack

...
return addr <main+?>
saved %rbx
rtn addr <pcount_r+22>
1
rtn addr <pcount_r+22>

Recursive Example: Result (x = 2)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rbx	0 (x&1)	callee-restored
%rax	1 (ret val)	n/a

The Stack

...
return addr <main+?>
saved %rbx
rtn addr <pcount_r+22>
1

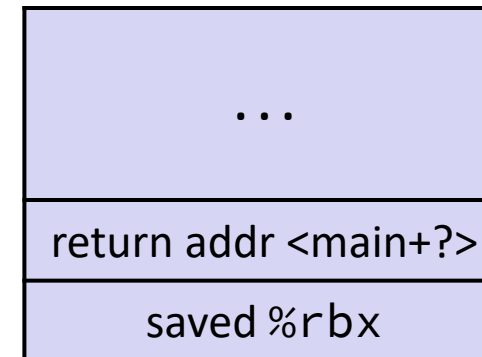
Recursive Example: Result (x = 5)

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rbx	1 (x&1)	callee-restored
%rax	2 (ret val)	n/a

The Stack



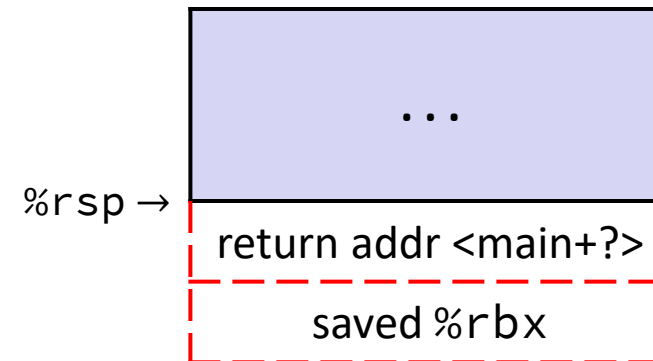
Recursive Example: Completion

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

```
pcount_r:  
    testq    %rdi, %rdi  
    jne      .L8  
    movl     $0, %eax  
    ret  
.L8:  
    pushq    %rbx  
    movq     %rdi, %rbx  
    andl     $0x1, %ebx  
    shrq     $1, %rdi  
    call     pcount_r  
    addq     %rbx, %rax  
    popq     %rbx  
    ret
```

Register	Value	Save Status
%rbx	<i>prev value</i>	callee-restored
%rax	2 (ret val)	n/a

The Stack



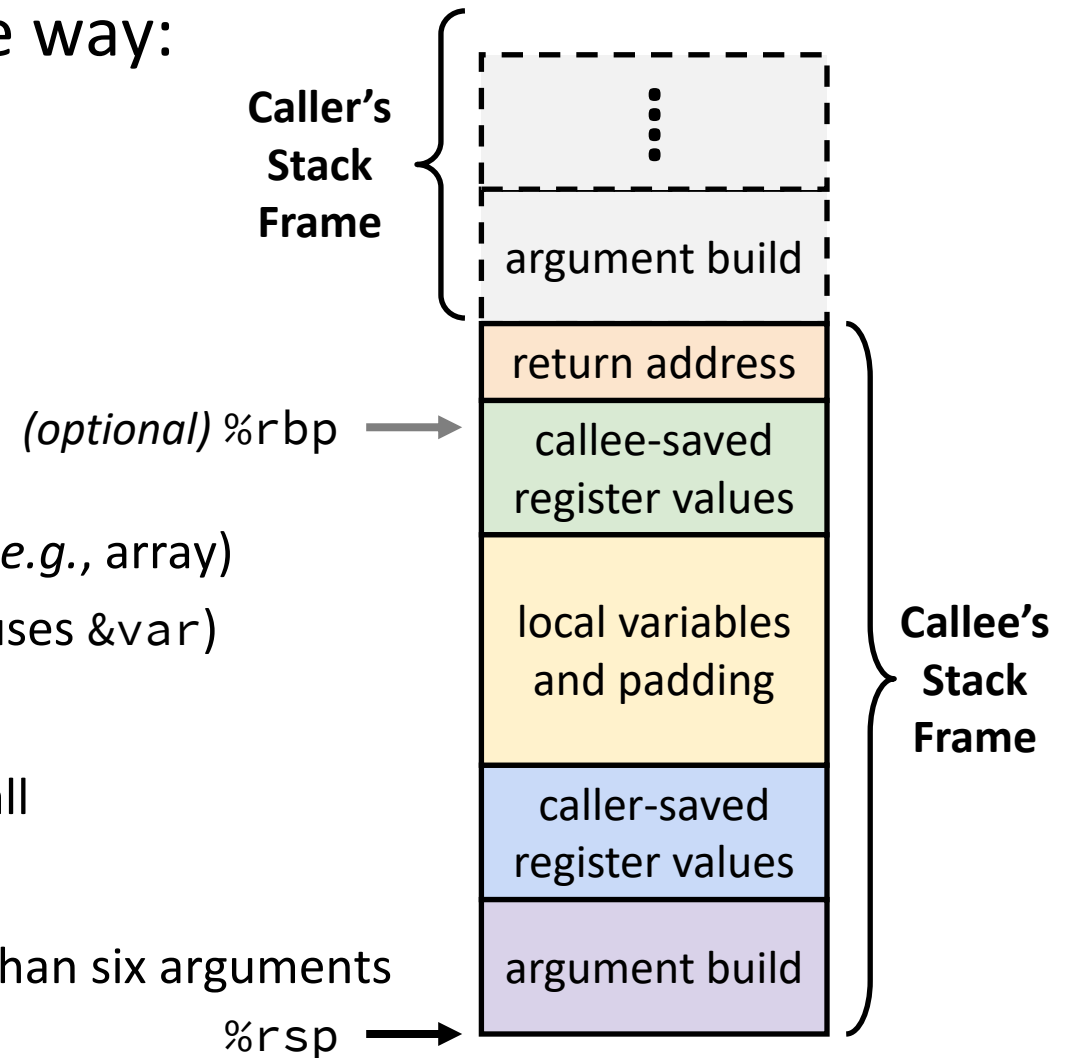
GDB Demo #2

- ❖ Let's examine the `pcount_r` stack frames on a real machine!
 - Using `pcount.c` from the course website
- ❖ You will need to use GDB to get through Lab 2
 - Useful debugger in this class and beyond!
- ❖ Pay attention to:
 - Checking the current stack frames (`backtrace`)
 - Getting stack frame information (`info frame <#>`)
 - Examining memory (`x`)

Summary (1/3)

❖ Each stack frame organized in the same way:

- 1) Return address pushed by `call`
 - The address of the instruction after `call`
- 2) Callee-saved registers
 - Only if procedure modifies/uses them
- 3) Local variables
 - Unavoidable if variable is too big for a register (*e.g.*, array)
 - Unavoidable if variable needs an address (*i.e.*, uses `&var`)
- 4) Caller-saved registers
 - Only if values are needed *across* a procedure call
- 5) Argument build
 - Only if procedure calls a procedure with more than six arguments



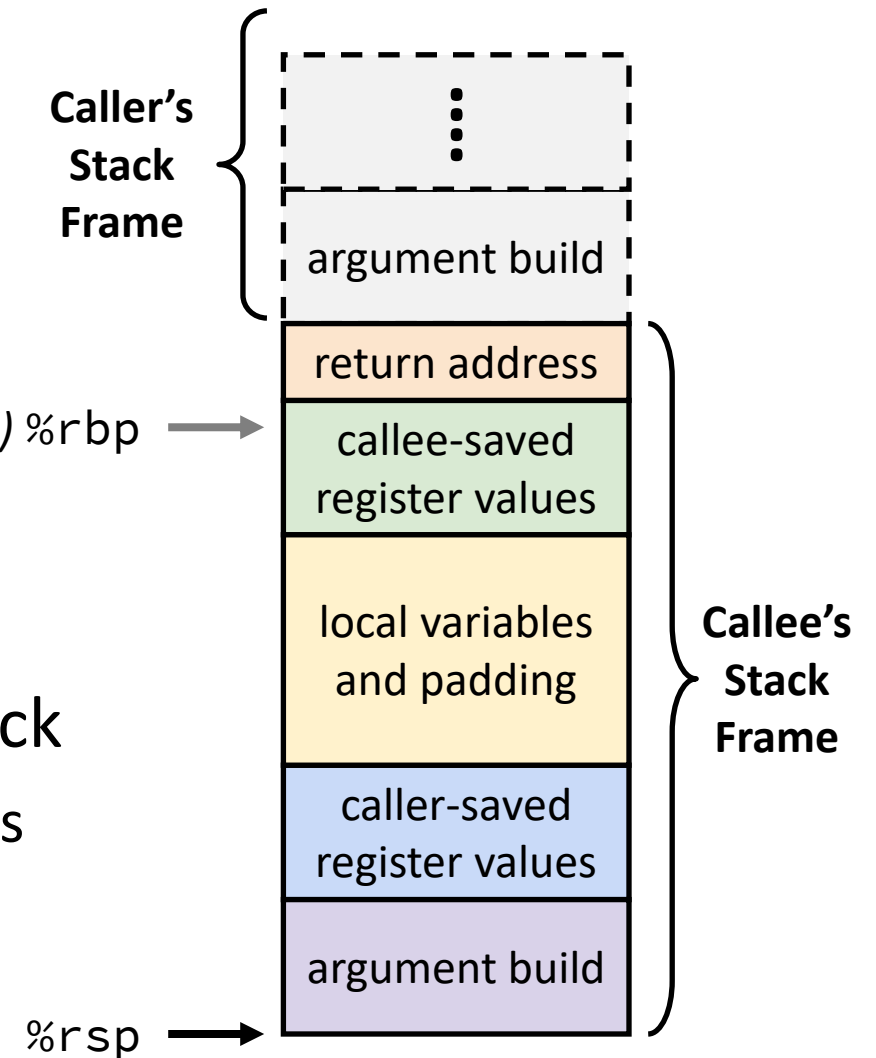
Summary (2/3)

❖ Important Points

- Procedures are a **combination of *instructions* and *conventions***
 - Conventions prevent functions from disrupting each other
- Stack is the right data structure
 - “Last in, first out” matches lifetime of procedures *(optional)* %rbp
- Recursion handled by normal calling conventions

❖ Generally want to minimize the use of the stack

- Lean heavily on registers, which are faster to access



Summary (3/3)

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