# Machine Level Programming: Procedure Calls including Recursion

CSCI 237: Computer Organization
17th Lecture, Friday, Oct. 17

**Kelly Shaw**

1

1


2

2

## Administrative Details

- Read CSAPP 3.8
- Weekly quiz open is due today at 2:30
- Midterm in lab on Wednesday 10/23
  - Closed book and closed notes
  - Review session on Monday 10/21 7-8:30pm in Physics 205
  - Sample midterm and solutions on Glow
- Apply to be a TA!
- TA feedback forms
- CS Colloquium talk today at 2:35pm in Wege
  - What I did last summer part 2

3

3

## Last Time: Machine-Level Programming: Procedures

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

4

4

## Today: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

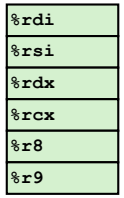
5

5

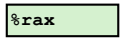
## Procedure Data Flow (Review)

**Registers**

- First 6 arguments




- Return value




**Stack**


- Only allocate stack space when needed (e.g., more than 6 integral arguments)!

6

6

## Data Flow Examples

```
void multstore
 (long x, long y, long *dest)
{
    long t = mult2(x, y);
    *dest = t;
}
```

```
0000000000400540 <multstore>:
  # x in %rdi, y in %rsi, dest in %rdx
  •••
  400541: mov    %rdx,%rbx        # Save dest
  400544: callq  400550 <mult2>   # mult2(x,y)
  # t in %rax
  400549: mov    %rax,(%rbx)      # Save at dest
  •••
```

```
long mult2
  (long a, long b)
{
  long s = a * b;
  return s;
}
```

```
0000000000400550 <mult2>:
  # a in %rdi, b in %rsi
  400550:  mov    %rdi,%rax       # a
  400553:  imul   %rsi,%rax       # a * b
  # s in %rax
  400557:  retq                   # Return
```

7

7

## Today: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

8

8

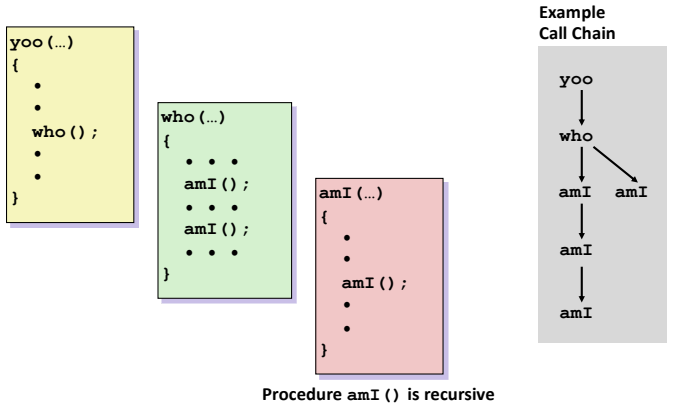
## Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java
  - Code must be "Reentrant"
    - There can be multiple simultaneous instantiations of a single procedure
  - We need some place to store the state of each instantiation
    - Arguments
    - Local variables
    - Return address
- Stack discipline
  - The state for a given procedure is only needed for a limited time
    - From when the procedure is called to when the return occurs
  - Callee returns before caller does
- Stack allocated in Frames
  - A frame stores the state for single procedure instantiation

9

9

## Call Chain Example

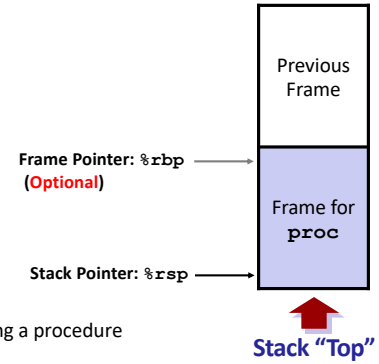



10

10

## Stack Frames

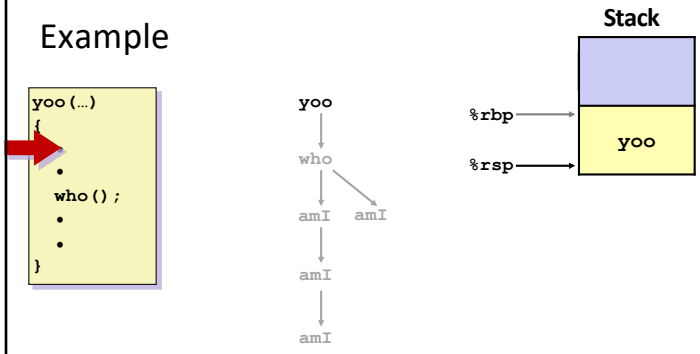
- Contents:
  - Return information
  - Local storage (if needed)
  - Temporary space (if needed)
- Management
  - Space is allocated when entering a procedure
    - "Set-up" code
    - Includes push by `call` instruction
  - Space is deallocated when a return occurs
    - "Finish" code
    - Includes pop by `ret` instruction




11

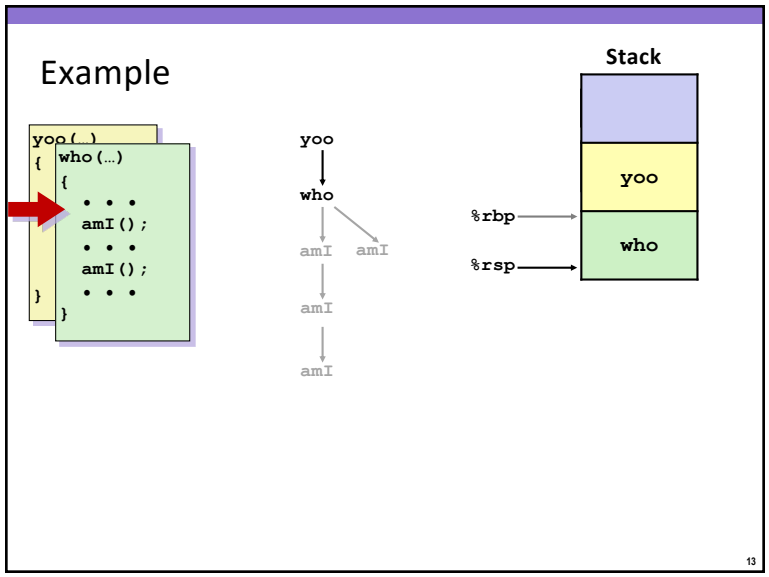
11

## Example

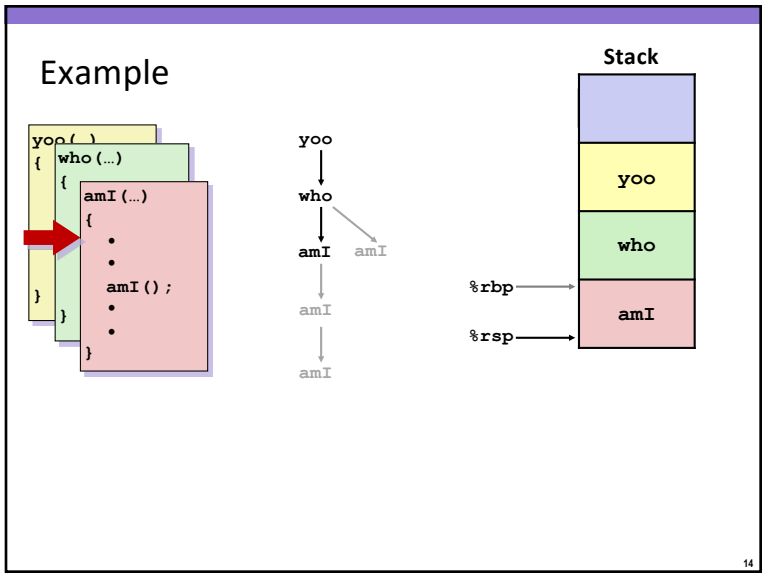



12

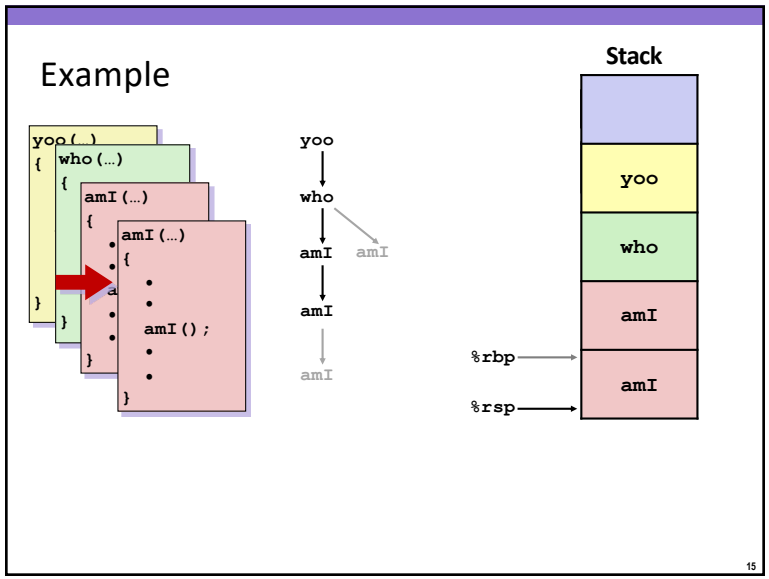
12


Example
Stack
yoo ( )
{
who (…)
{
• • •
amI ();
• • •
amI ();
• • •
}
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
13

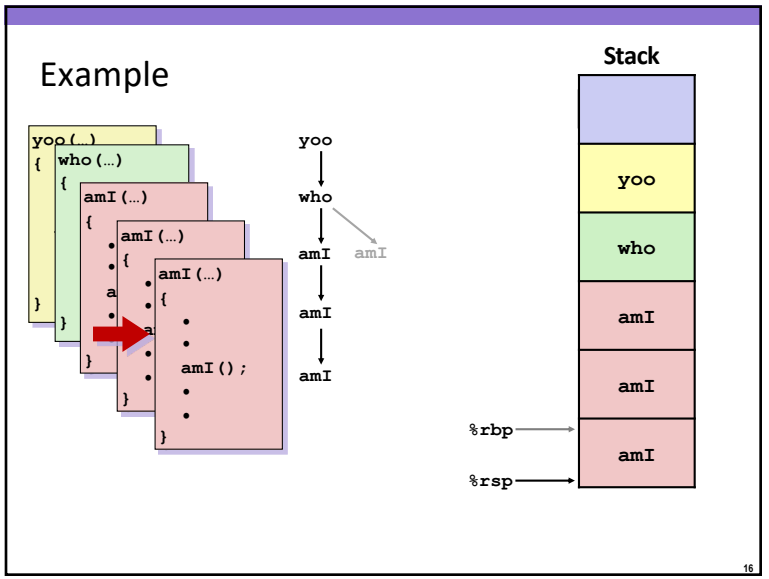
13


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
•
•
amI ();
•
•
}
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
14

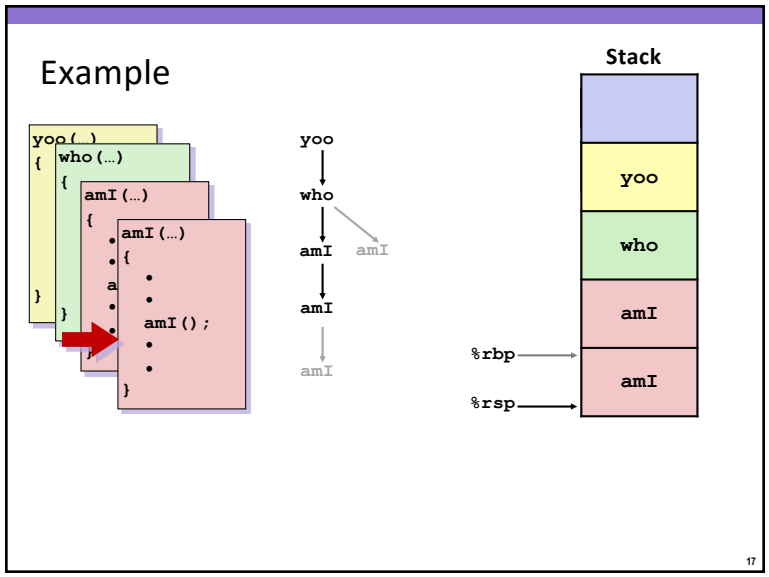
14


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI (…)
{
•
•
amI ();
•
•
}
}
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
amI
15

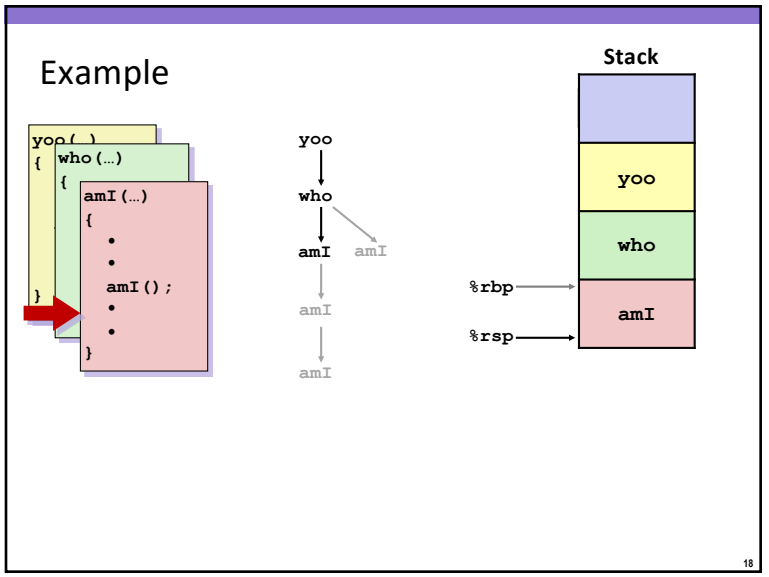
15


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI (…)
{
amI (…)
{
•
•
amI ();
•
•
}
}
}
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
amI
amI
16

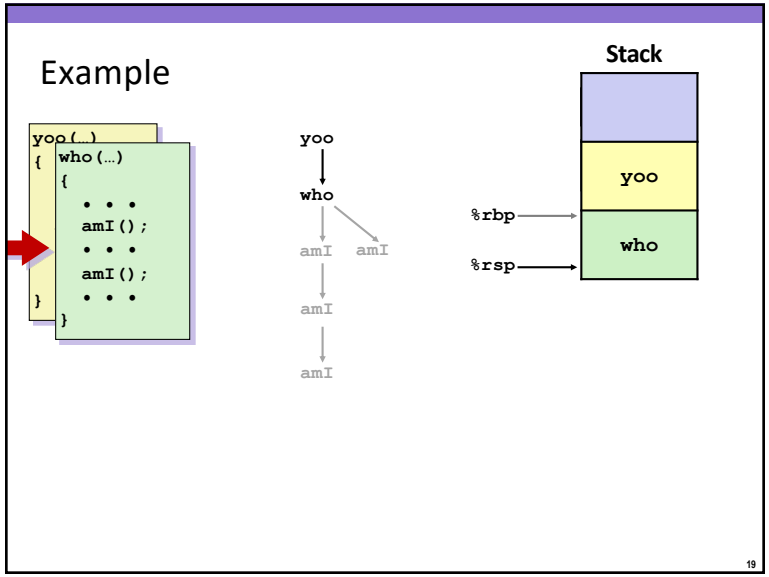
16


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
amI
17

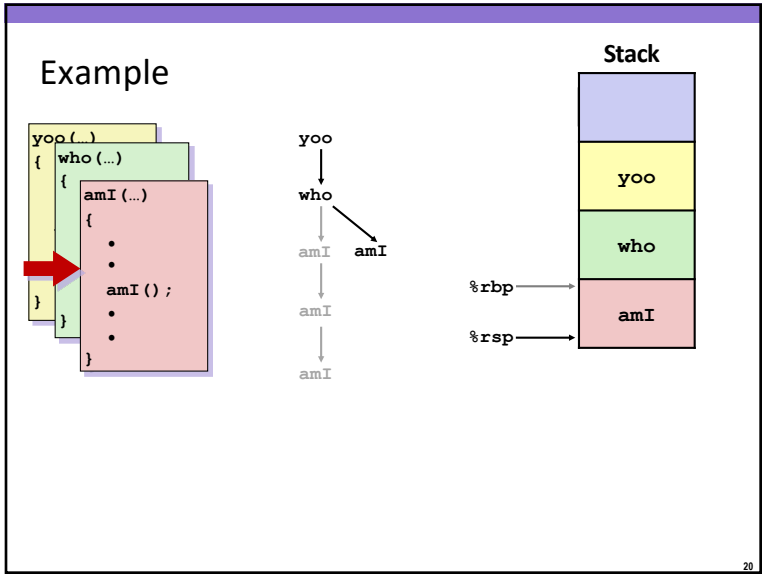
17


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
18

18

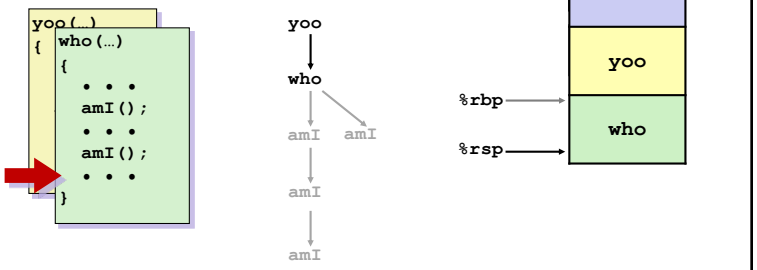

Example
Stack
yoo ( )
{
who (…)
{
• • •
amI ();
• • •
amI ();
• • •
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
19

19


Example
Stack
yoo ( )
{
who (…)
{
amI (…)
{
amI ();
}
yoo
who
amI
amI
amI
amI
%rbp
%rsp
yoo
who
amI
20

20

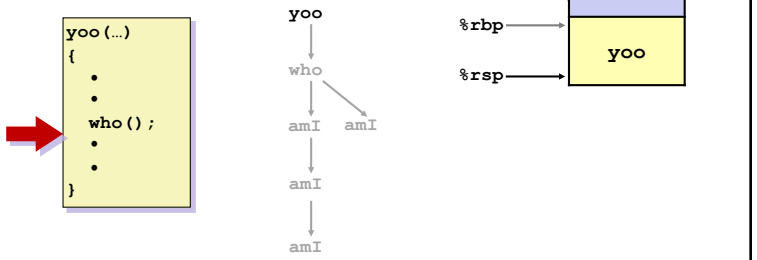
## Example




21

21

## Example

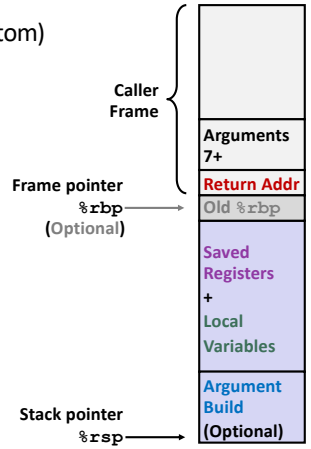



22

22

## 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 (optional)
- Caller Stack Frame
  - Return address
    - Pushed by `call` instruction
  - Arguments for this call




23

23

## Example: `incr`

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

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

| Register | Use(s) |
|---|---|
| %rdi | Argument p |
| %rsi | Argument val, y |
| %rax | x, Return value |

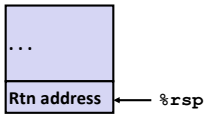
24

24

## Example: Calling `incr` #1

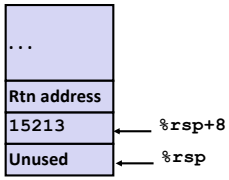
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Initial Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Resulting Stack Structure



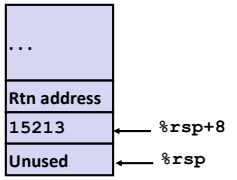

25

25

## Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

| Register | Use(s) |
|---|---|
| %rdi | &v1 |
| %rsi | 3000 |

26

26

## Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure


**Aside 1: `movl $3000, %esi`**
- Remember, movl -> %exx zeros out high order 32 bits.
- Why use movl instead of movq? 2 bytes shorter.

```
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

| | |
|---|---|
| %rdi | &v1 |
| %rsi | 3000 |

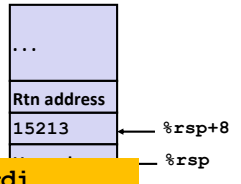
27

27

## Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Stack Structure




**Aside 2: `leaq 8(%rsp), %rdi`**
- Computes %rsp+8
- Actually used for what it is meant!

```
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

| | |
|---|---|
| %rsi | 3000 |

28

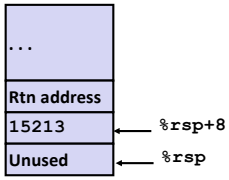
28

## Example: Calling `incr` #2

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure




| Register | Use(s) |
|---|---|
| %rdi | &v1 |
| %rsi | 3000 |

29

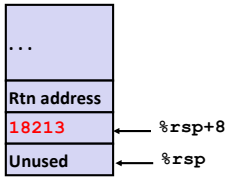
29

## Example: Calling `incr` #3

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure




| Register | Use(s) |
|---|---|
| %rdi | &v1 |
| %rsi | 3000, 18213 |
| %rax | 15213 |

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

30

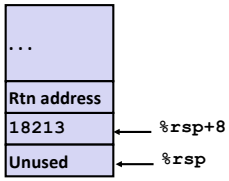
30

## Example: Calling `incr` #4

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure




| Register | Use(s) |
|---|---|
| %rax | 33426 (Return value) |

31

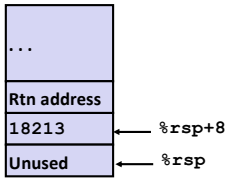
31

## Example: Calling `incr` #5a

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

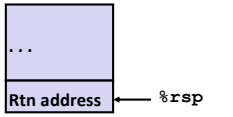
```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

Stack Structure




| Register | Use(s) |
|---|---|
| %rax | 33426 (Return value) |

Updated Stack Structure



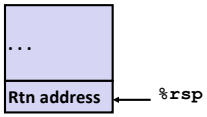

32

32

## Example: Calling `incr` #5b

```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

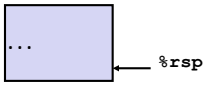
Updated Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

| Register | Use(s) |
|---|---|
| %rax | 33426 (Return value) |

Final Stack Structure



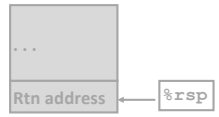

33

33

## Example: Calling `incr` #5b

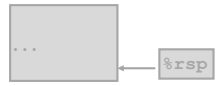
```
long call_incr() {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return v1+v2;
}
```

Updated Stack Structure




```
call_incr:
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    8(%rsp), %rax
  addq    $16, %rsp
  ret
```

**Why not just push $15213 onto stack? pushq allocates 8 bytes, not 16. (GCC tries to align stack frames to 16 bytes by default.)**




34

34

## Practice on Your Own

- Convert the following x86-64 assembly code to C.

```
fcn:
  leaq (%rdi, %rsi, 8), %rdi
  pushq %rdi
  callq calc
  popq %rdi
  movq %rax, (%rdi)
  movq $1, %rax
  retq
calc:
  movq (%rdi), %rdi
  addq %rdi, %rdi
  movq %rdi, %rax
  retq
```

Given how calc is written, was pushing %rdi onto the stack required?

35

35

## Today: Machine-Level Programming: Procedures and Arrays

- Procedures
  - Stack Structure
  - Calling Conventions
    - Passing control
    - Passing data
    - Managing local data
  - Register saving conventions
  - Illustration of Recursion

36

36

## Register Saving Conventions

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

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

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

  - Contents of register **%rdx** overwritten by **who**
  - This could be trouble → something should be done!
    - Need some coordination

37

37

## Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the caller
  - **who** is the callee
- Can register be used for temporary storage?
- Conventions
  - "Caller Saved"
    - Caller saves temporary values in its frame before the call
  - "Callee Saved"
    - Callee saves temporary values in its frame before using
    - Callee restores them before returning to caller

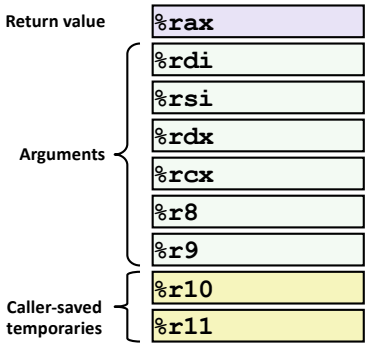
For this class, conventions will be followed strictly.

38

38

## x86-64 Linux Register Usage #1

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

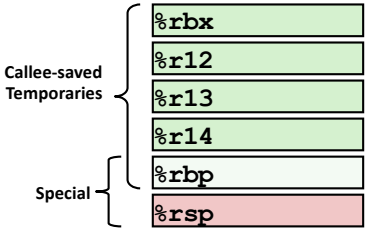



39

39

## x86-64 Linux Register Usage #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




40

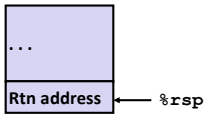
40

## Callee-Saved Example #1

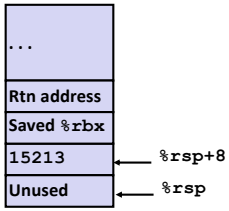
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Initial Stack Structure




Resulting Stack Structure




41

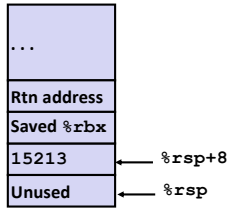
41

## Callee-Saved Example #2

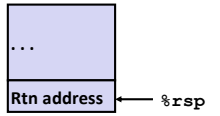
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```

```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

Resulting Stack Structure




Pre-return Stack Structure




42

42

## Using Callee-Saved Registers vs. Stack

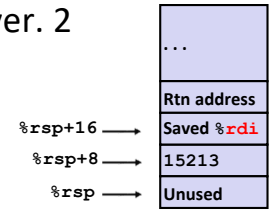
- Suppose you have a value in a **caller-saved** register X but need to make a function call
- If you don't need the value in X after the function call, don't do anything
- If you do need the value in X after the function call, you need to **either**:
  - Store value in X on to the stack and then restore value to register from the stack after function call
  - Store value in **callee-saved** register Y to stack, copy X to register Y, use Y after the function call returns, restore original value of Y by copying value from stack to register Y before returning from your function (previous example)

43

43

## Callee-Saved Example #1 ver. 2

```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x+v2;
}
```




```
call_incr2:
  pushq   %rbx
  subq    $16, %rsp
  movq    %rdi, %rbx
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  addq    %rbx, %rax
  addq    $16, %rsp
  popq    %rbx
  ret
```

```
call_incr2:
  pushq   %rdi
  subq    $16, %rsp
  movq    $15213, 8(%rsp)
  movl    $3000, %esi
  leaq    8(%rsp), %rdi
  call    incr
  movq    16(%rsp), %rdi
  addq    %rdi, %rax
  addq    $16, %rsp
  popq    %rdi
  ret
```

44

44

## Practice on Your Own

- Which values in fcn() would need to be stored either in callee-saved registers or on the stack?

```
long fcn(long a, long *b, long c)
{
   long tmp = a + c;

   long result = foo(tmp);

   *(b+1) = result - a;

   return tmp;
}
```

45

45