CS429: Computer Organization and Architecture

Instruction Set Architecture IV

Dr. Bill Young
Department of Computer Science
University of Texas at Austin

Last updated: October 9, 2019 at 18:14

Mechanisms in Procedures

Passing Control

- To beginning of procedure code
- Back to return point

Passing Data

- Procedure arguments
- Return value

Memory Management

- Allocate during procedure execution
- Deallocate upon return

Mechanisms all implemented with machine instructions

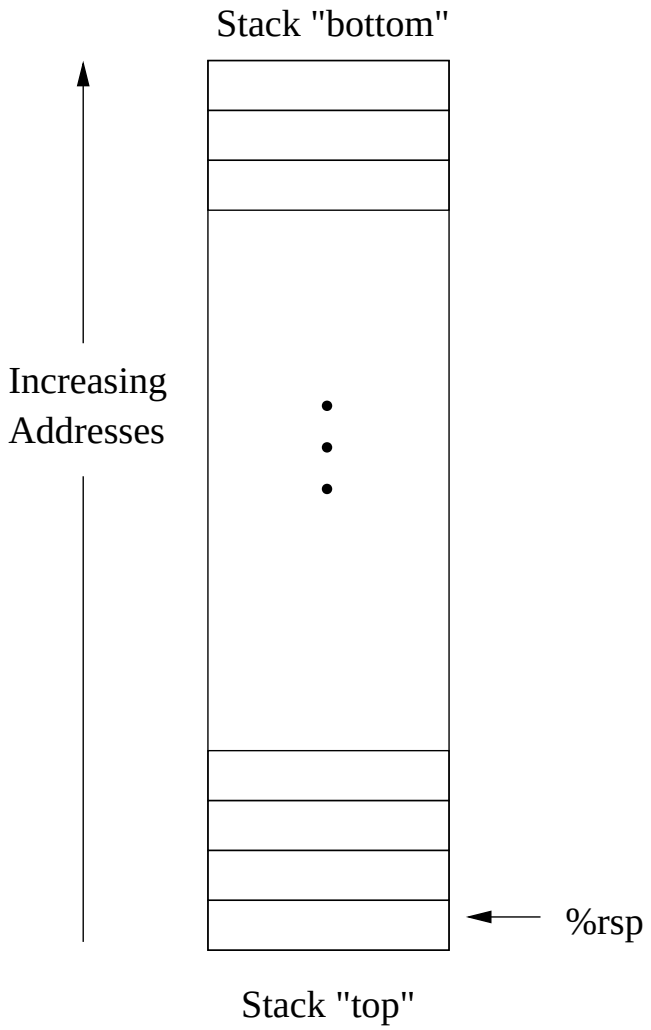
x86-64 implementation of a procedure uses only those mechanisms required.

```
P(...) {
   ...
   y = Q(x);
   printf(y);
   ...
}
```

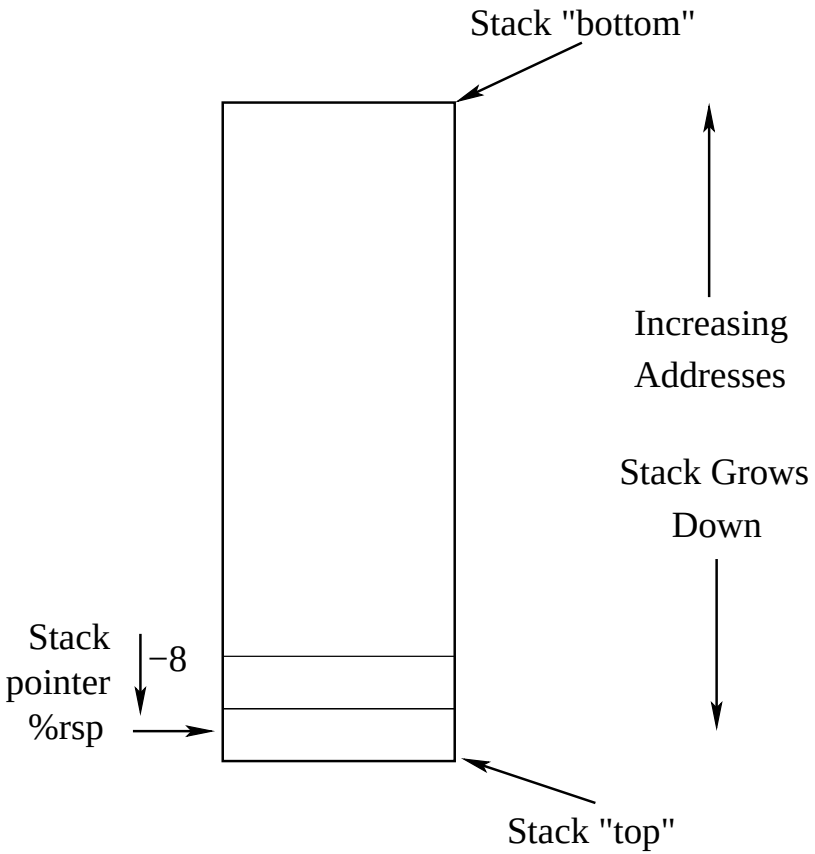
```
int Q(int i)
{
    int t = 3*i;
    int v[10];
    ...
    return v[t];
}
```

x86-64 Stack

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



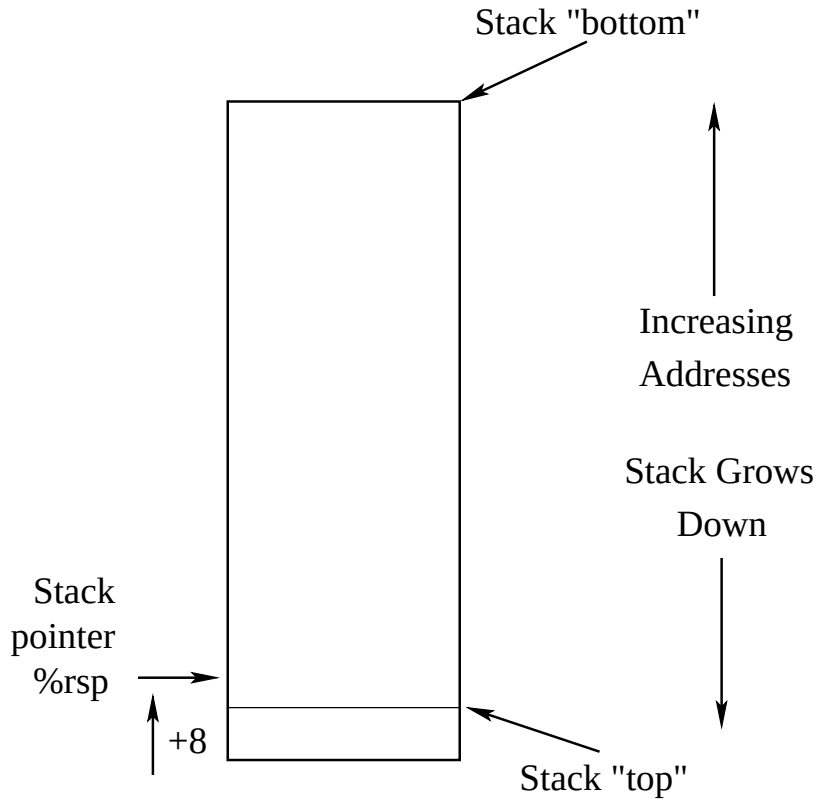
x86-64 Stack Pushing



Pushing

- `pushq Src`
- Decrement `%rsp` by 8
- Write operand at address given by `%rsp`

x86-64 Stack Popping



Popping

- `popq Dest`
- Read operand at address given by `%rsp`
- Increment `%rsp` by 8
- Write to Dest

Code Examples

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

```
0000000000400540 <multstore>:
  400540:   push   %rbx                # Save %rbx
  400541:   mov    %rdx,%rbx           # Save dest
  400544:   callq  400580 <mult2>      # mult2(x, y)
  40054D:   mov    %rax, (%rbx)        # Save at dest
  400550:   pop    %rbx                # Restore %rbx
  400552:   retq                       # Return
```

Code Examples (2)

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

```
0000000000400580 <mult2>:
   400580:   mov   %rdi, %rax      # a
   400583:   imul  %rsi, %rax      # a * b
   400587:   retq                  # Return
```

Procedure Control Flow

Use stack to support procedure call and return

Procedure call: `call label`

- Push return address on stack
- Jump to label

Return address:

- Address of next instruction right after call

Procedure return: `ret`

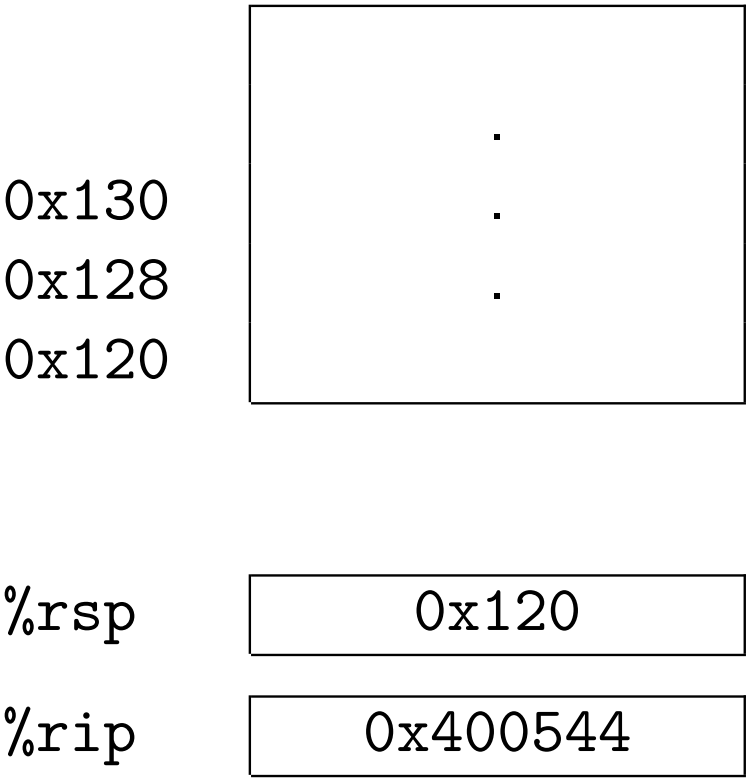
- Pop address from stack
- Jump to address

Control Flow Example #1

Poised to execute `call` in `multstore`.

```
0000000000400540 <multstore>:
                 ...
  400544:   callq 400580 <mult2>
  40054D:   mov   %rax, (%rbx)
                 ...
```

```
0000000000400580 <mult2>:
   400580:   mov   %rdi, %rax
                 ...
   400587:   retq
```

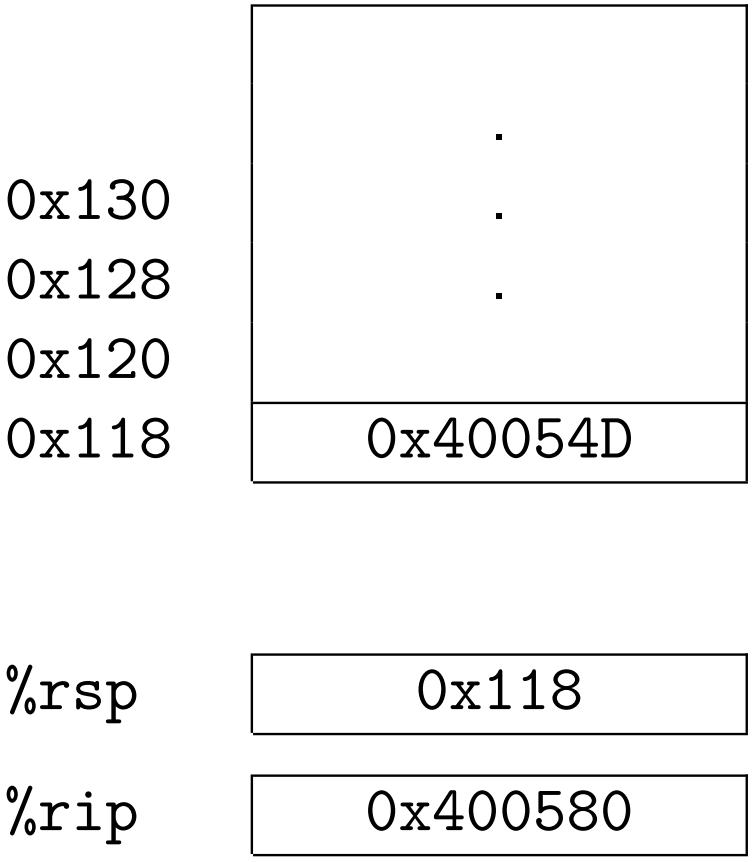


Control Flow Example #2

After `call` in `multstore`. Now in `mult2`.

```
0000000000400540 <multstore>:
                 ...
  400544:   callq  400580 <mult2>
  40054D:   mov    %rax, (%rbx)
                 ...
```

```
0000000000400580 <mult2>:
   400580:   mov   %rdi, %rax
                 ...
   400587:   retq
```

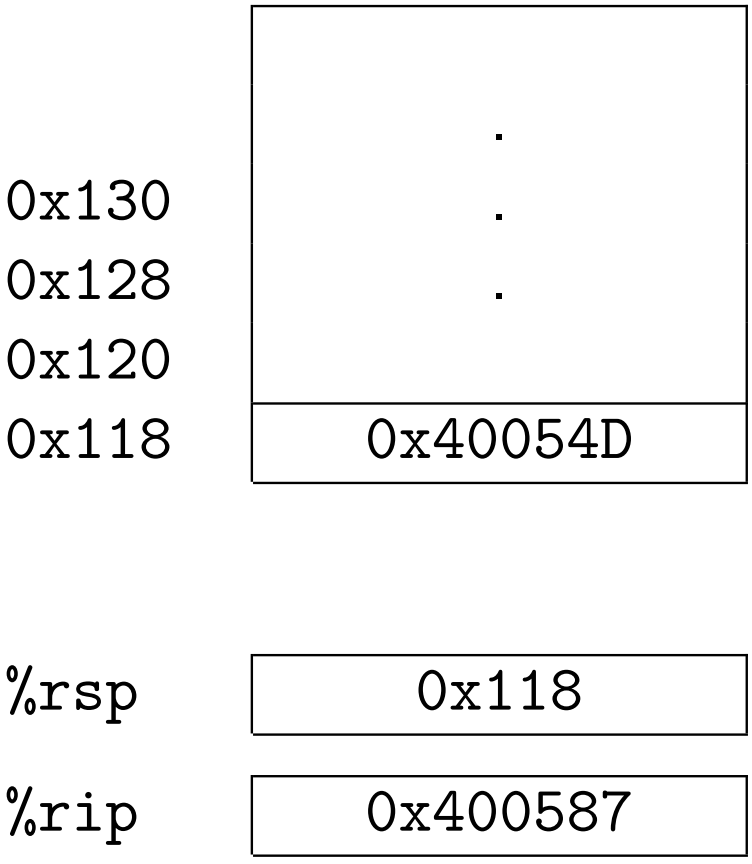


Control Flow Example #3

Execute through `mult2` to `retq`.

```
0000000000400540 <multstore>:
                 ...
  400544:   callq 400580 <mult2>
  40054D:   mov   %rax, (%rbx)
                 ...
```

```
0000000000400580 <mult2>:
   400580:   mov   %rdi, %rax
                 ...
   400587:   retq
```

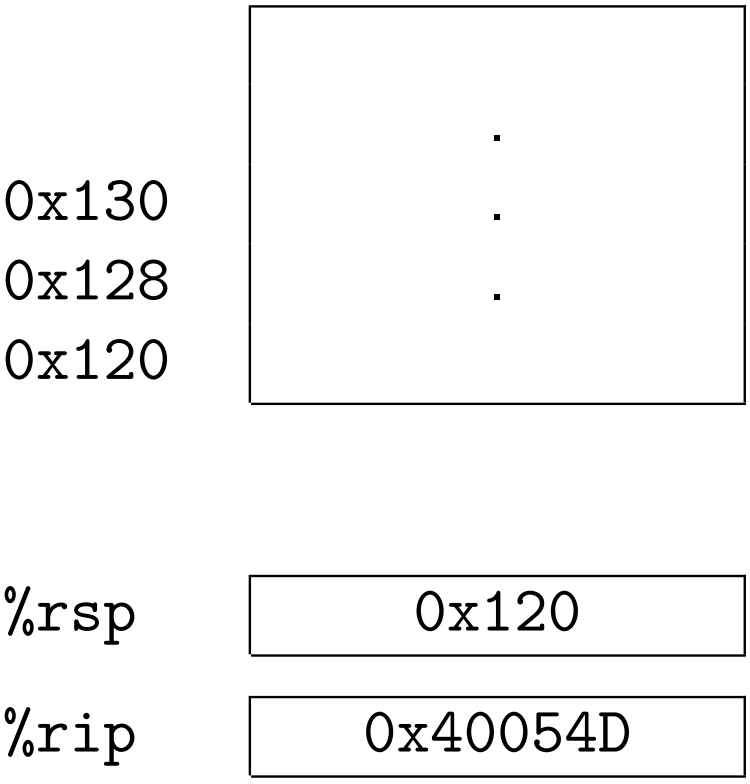


Control Flow Example #4

After `retq` in `mult2`. Back in `multstore`.

```
0000000000400540 <multstore>:
                 ...
  400544:   callq 400580 <mult2>
  40054D:   mov   %rax, (%rbx)
                 ...
```

```
0000000000400580 <mult2>:
   400580:   mov   %rdi, %rax
                 ...
   400587:   retq
```



Procedure Data Flow

Registers: First 6 arguments

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

Mnemonic to remember the order: "Diane's silk dress cost $89." Args past 6 are pushed on the stack in reverse order.

Return value

%rax

Stack

...
Arg n
...
Arg 8
Arg 7

← %rsp

Only allocate stack space when needed.

Stack-Based Languages

Languages that support recursion

- e.g., C, Pascal, Java
- Code must be "reentrant": multiple simultaneous instantiations of single procedure
- Need some place to store state of each instantation: arguments, local variables, return pointer

Stack discipline

- State for given procedure needed for a limited time: from call to return
- Callee returns before caller does

Stack allocated in Frames

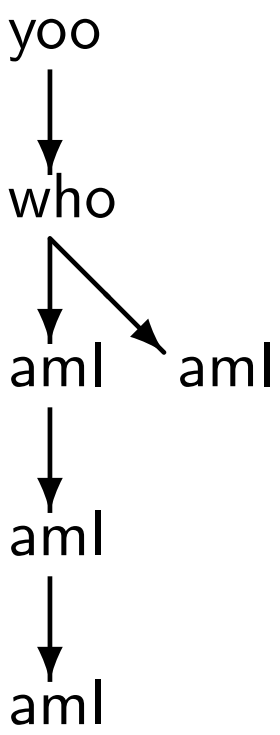
- State for a single procedure instantiation

Call Chain Example

Code Structure

```
yoo(...) {
    ...
    who();
    ...
}

who(...) {
    ...
    aml();
    ...
    aml();
    ...
}

aml(...) {
    ...
    aml();
    ...
}
```

Procedure aml is recursive.



Stack Frames

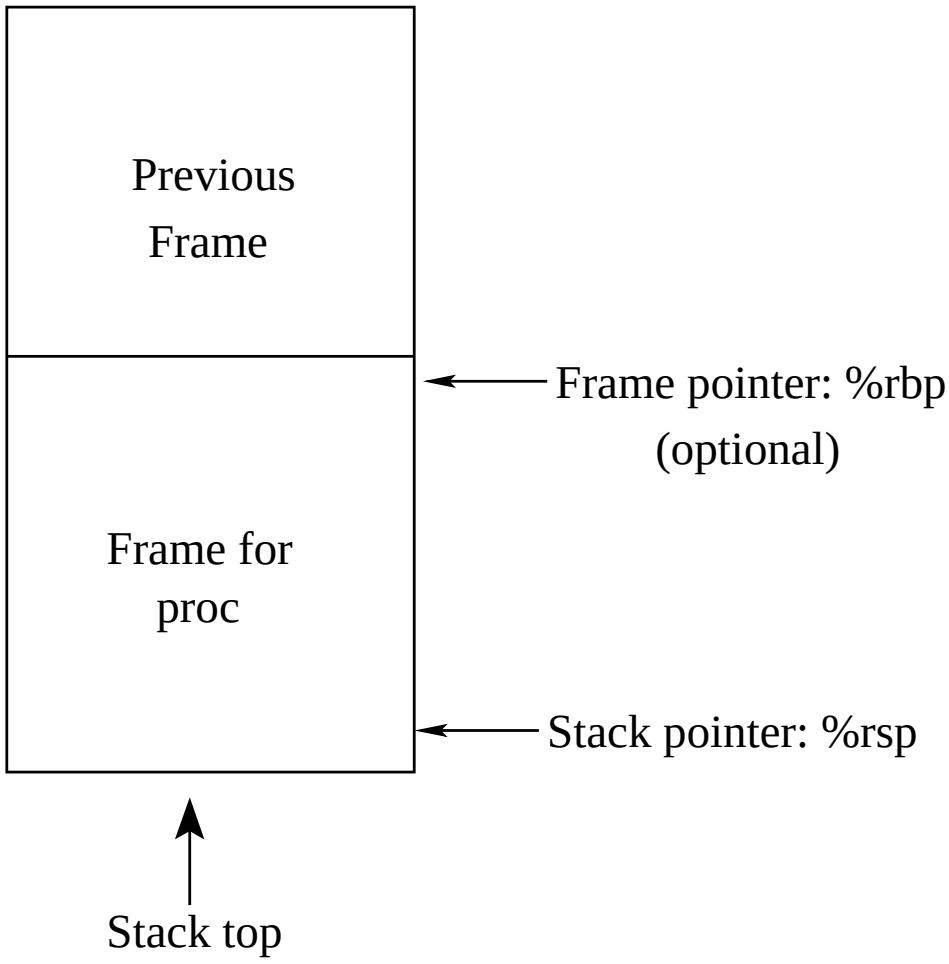
In IA32, almost every procedure call generated a stack frame. In x86-64, most calls do not generate an explicit frame.

Contents

- Return information
- Local storage (if needed)
- Temporary space (if needed)

Management

- Space allocated when enter procedure
  - "Set-up" code
  - Includes push by `call` instruction
- Deallocated when returning
  - "Finish" code
  - Includes pop by `ret` instruction

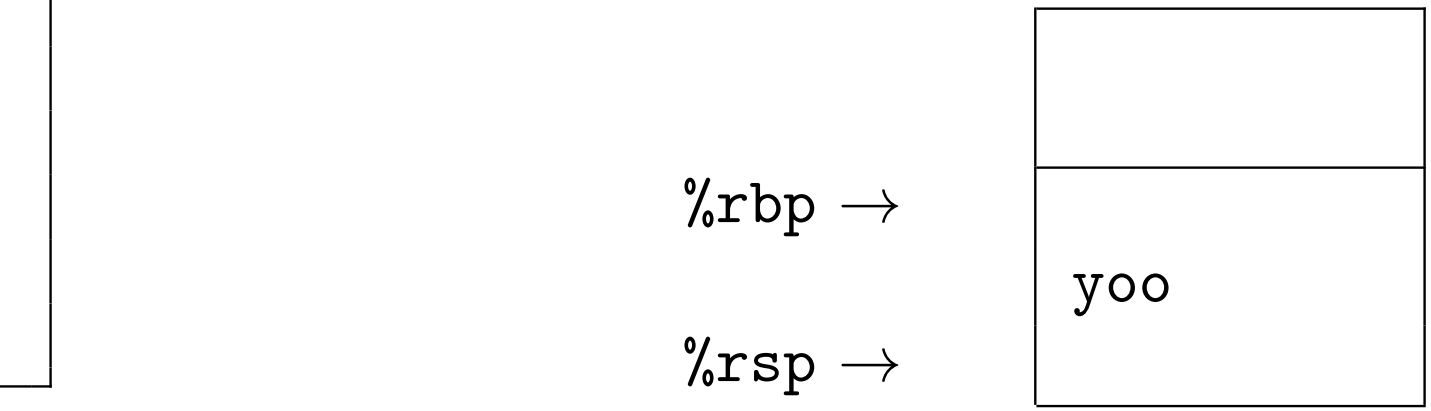


Example (1)

yoo

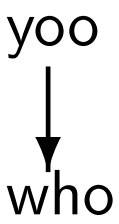
```
/* Executing
    here: */
yoo(...)
{
   ...
   who();
   ...
}
```

Stack

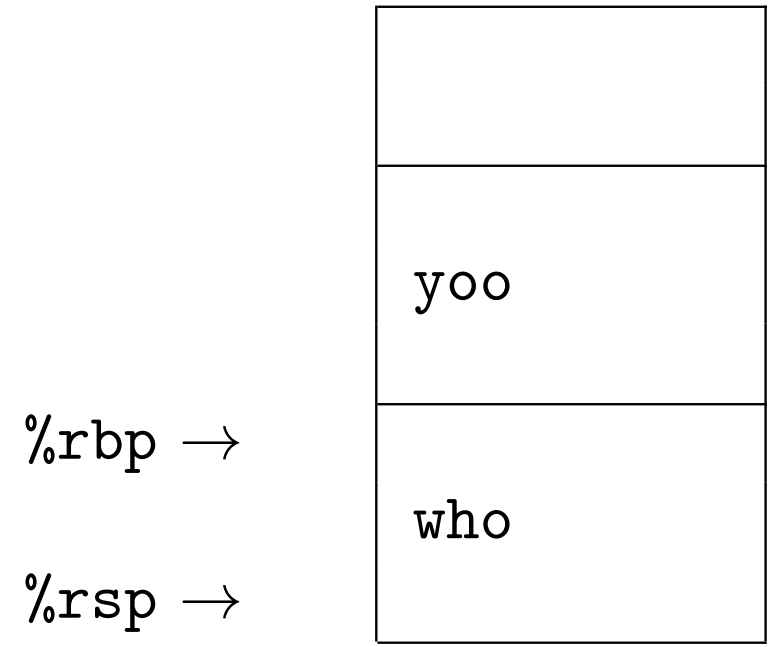


Example (2)

```
yoo(...)
{
   ...
   who();
   ...
}

/* Executing
    here: */
who(...)
{
   ...
   amI();
   ...
}
```



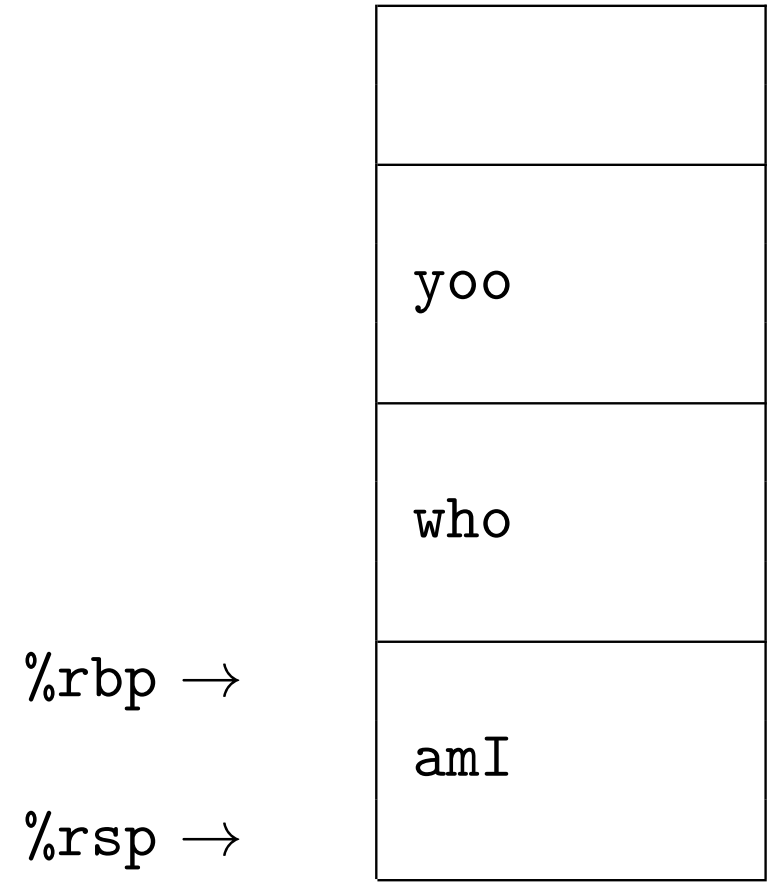
Stack



Example (3)

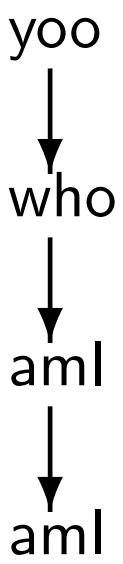
```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```

Stack



Example (4)

```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```



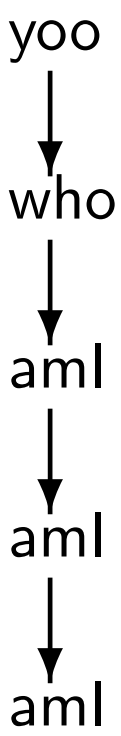
Stack

%rbp →

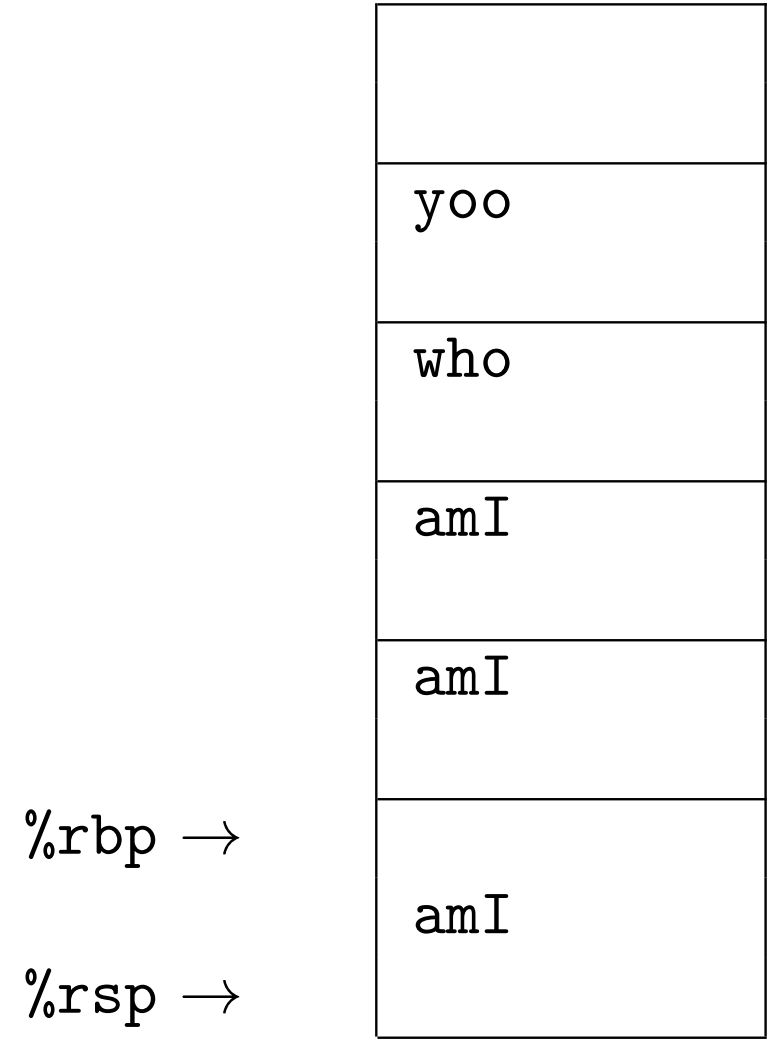
%rsp →

Example (5)

```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```



Stack



Example (6)

```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```

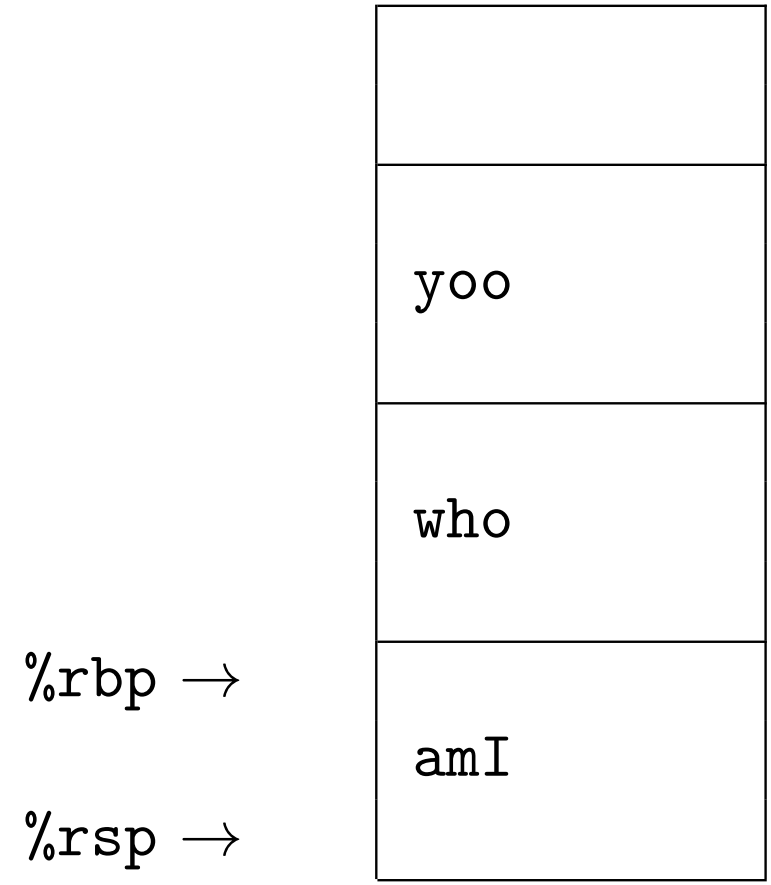
yoo
↓
who
↓
amI
↓
amI

Stack

Example (7)

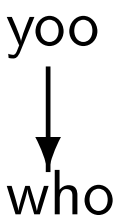
```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```

Stack

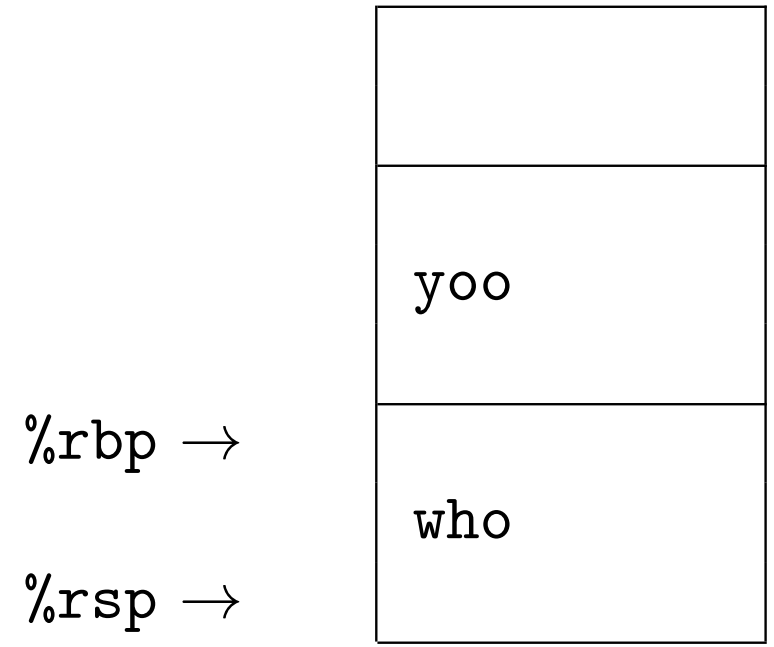


Example (8)

```
yoo(...)
{
   ...
   who();
   ...
}

/* Executing
    here: */
who(...)
{
   ...
   amI();
   ...
}
```

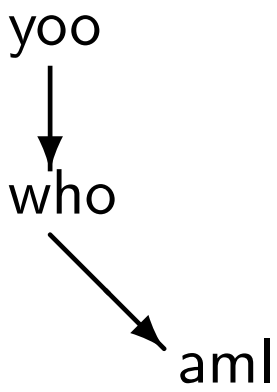


Stack

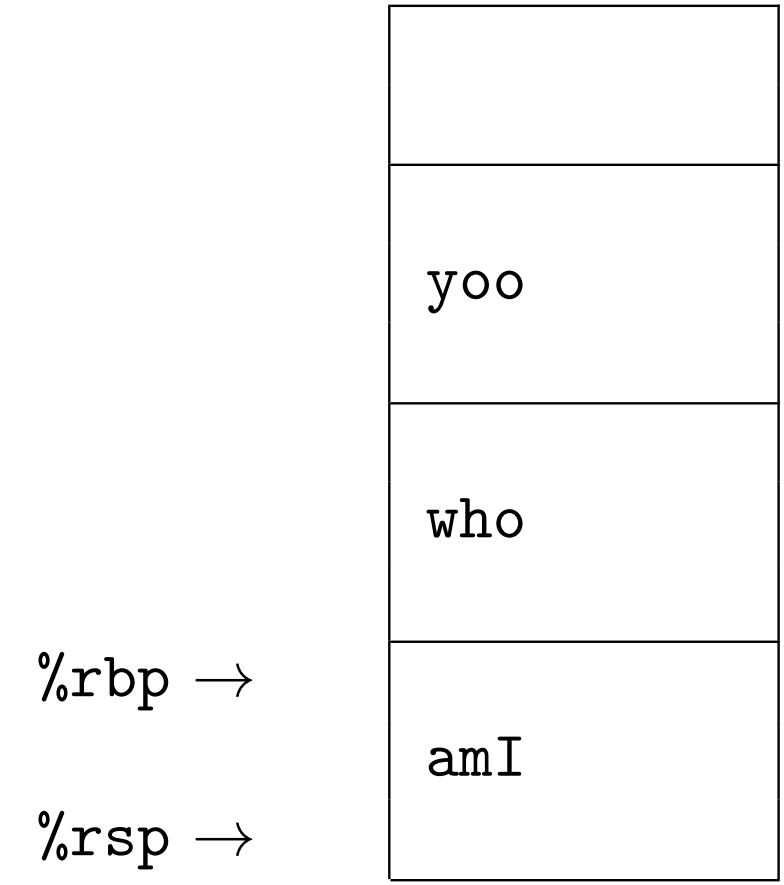


Example (9)

```
yoo(...)
{
   ...
}

who(...)
{
   ...
}

/* Executing
    here: */
amI(...)
{
   ...
   amI();
   ...
}
```

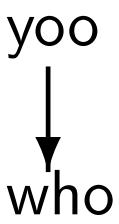


Stack

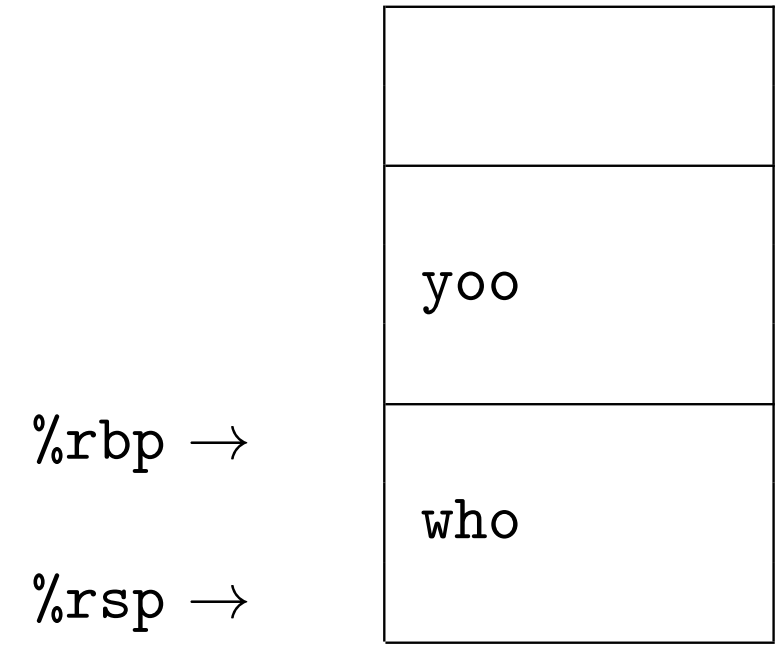


Example (10)

```
yoo(...)
{
   ...
   who();
   ...
}

/* Executing
    here: */
who(...)
{
   ...
   amI();
   ...
}
```



Stack

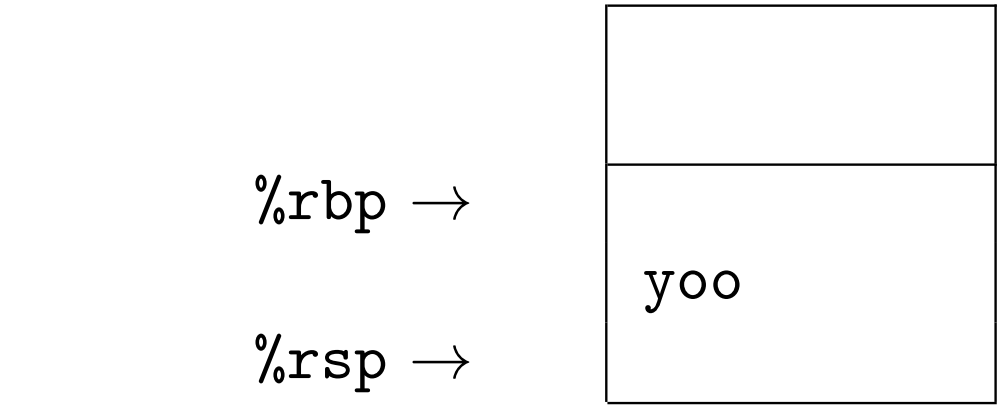


Example (11)

yoo

```
/* Executing
    here: */
yoo(...)
{
   ...
   who();
   ...
}
```

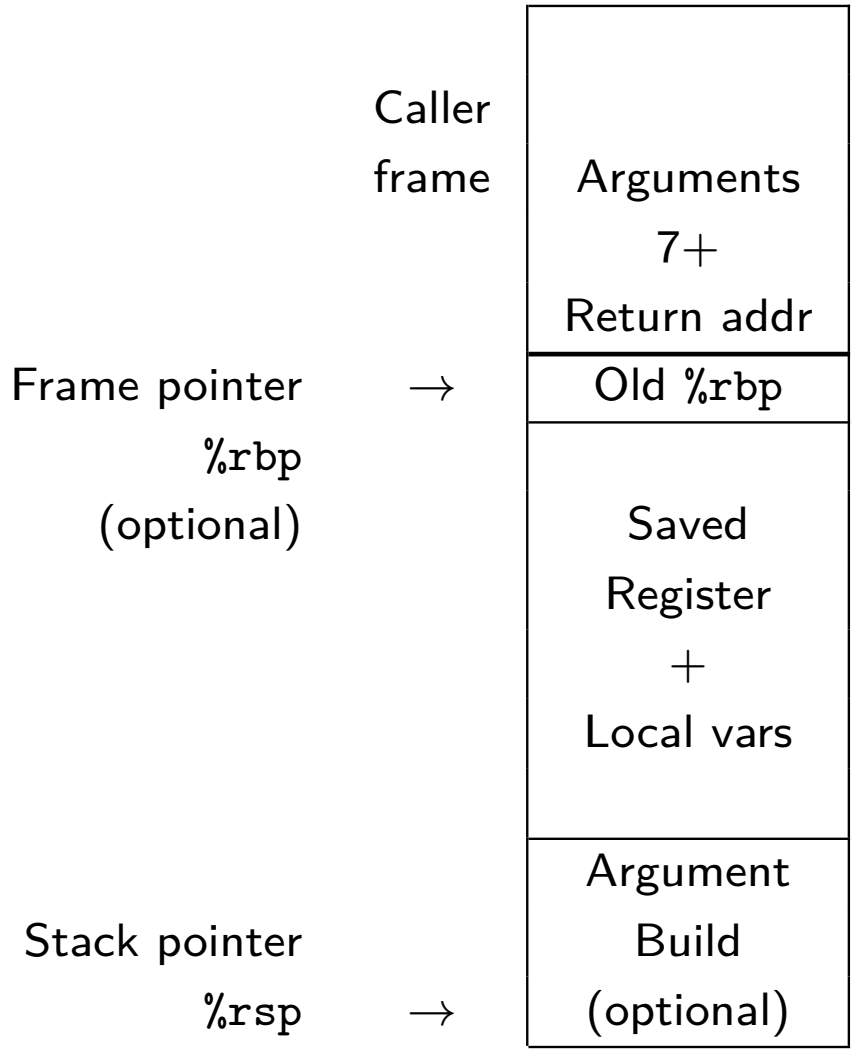
Stack



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

Building a Stack Frame

IA32 routines almost always constructed an explicit stack frame; x86-64 routines usually don't.

If you do build a stack frame:

```
proc:
      pushq %rbp              # save caller's frame base
      movq  %rsp, %rbp        # set callee's frame base

        ...                   # body of routine proc

      movq  %rbp, %rsp        # discard callee's frame
      popq  %rbp              # reset caller's frame base
      ret
```

Reserve %rbp for this purpose if you're doing this.

What does this do?

Example: incr

This example doesn't use explicit frames!

```
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
```

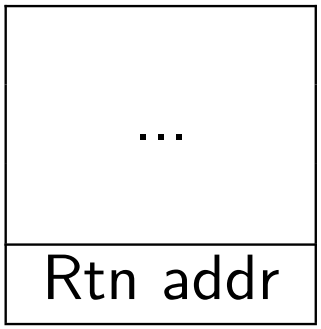
Register	Use(s)
%rdi	Argument p
%rsi	Argument val, y
%rax	x, return value

Example: Calling incr #1

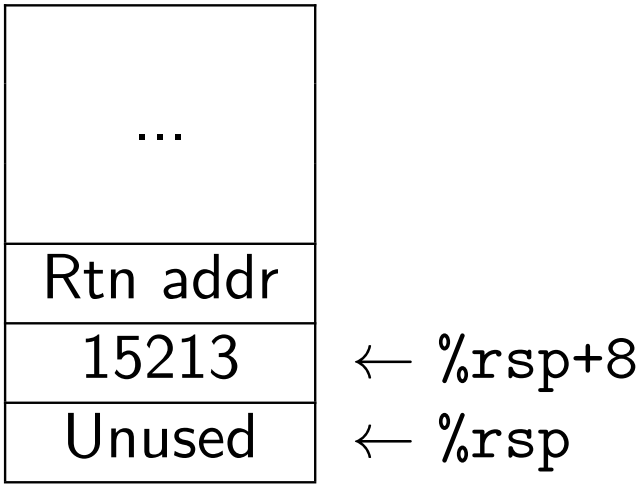
```
long call_incr() {

    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
```

Initial Stack Structure



Resulting Stack Structure

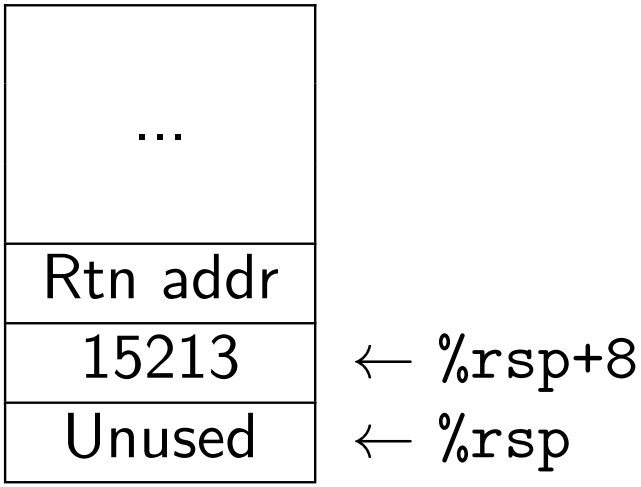


Example: Calling incr #2

```
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



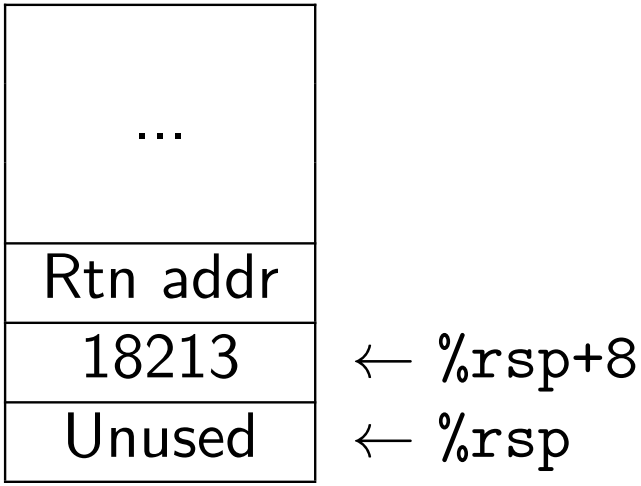
Register	Use(s)
%rdi	&v1
%rsi	3000

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



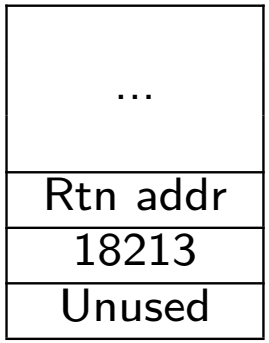
Register	Use(s)
%rdi	&v1
%rsi	3000

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



← %rsp+8

← %rsp

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

Updated Stack Structure

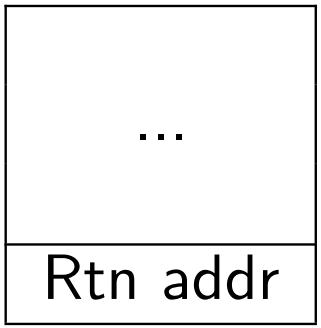
← %rsp

Example: Calling incr #5

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

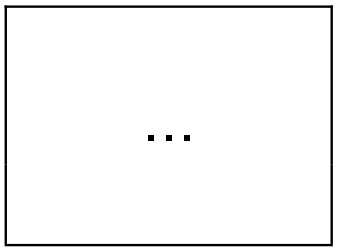
```
call_incr:
    subq    $16, %rsp
    movq    $15213, 8(%rsp)
    movl    $3000, %esi
    leaq    8(%rsp), %rdi
    call    incr
    addq    8(%rsp), %rax
    addq    $16, %rsp

    ret
```

Updated Stack Structure



Register	Use(s)
%rax	Return value

Final Stack Structure



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 are overwritten by who
- This could be trouble; something should be done!
- Need to coordinate between caller and callee.

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

x86-64 Linux Caller-saved Registers

`%rax`

- Return value
- Also caller-saved
- Can be modified by procedure

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

- Arguments
- Also caller-saved
- Can be modified by procedure

`%r10, %r11`

- Caller-saved
- Can be modified by procedure

Return value	`%rax`
Argument 1	`%rdi`
Argument 2	`%rsi`
Argument 3	`%rdx`
Argument 4	`%rcx`
Argument 5	`%r8`
Argument 6	`%r9`
Caller-saved	`%r10`
temporaries	`%r11`

x86-64 Linux Callee-saved Registers

`%rbx`, `%r12`, `%r13`, `%r14`, `%r15`

- Callee-saved
- Callee must save and restore

`%rbp`

- Callee-saved
- Callee must save and restore
- May be used as frame pointer
- Can mix and match

`%rsp`

- Special form of callee-saved
- Restored to original value upon exit from procedure

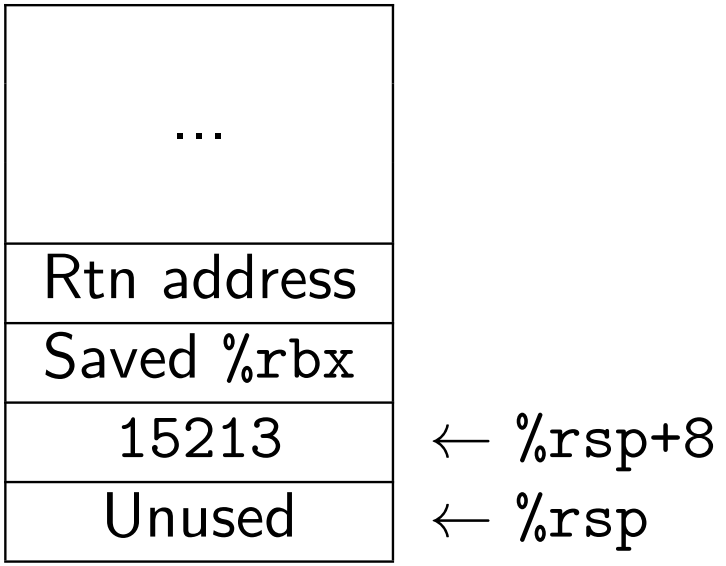
Callee-Saved Example #1

```
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

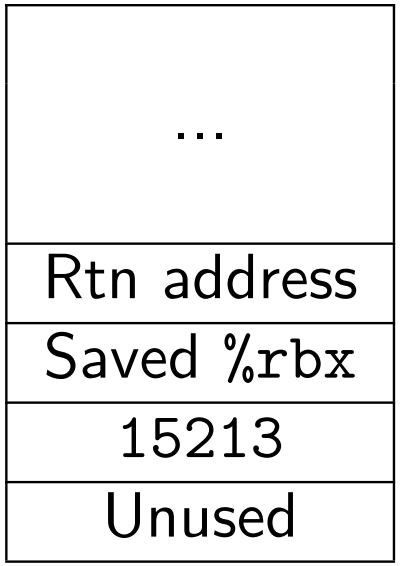


Callee-Saved Example #2

```
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

Recursive Function

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

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

Recursive Function Terminal Case

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

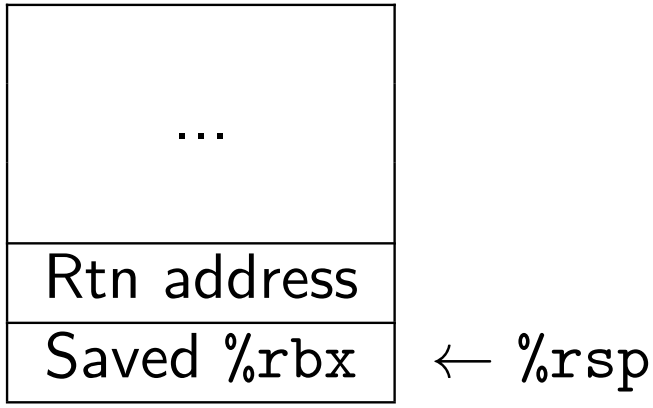
Register	Use(s)
%rdi	Argument x
%rax	Return value

```
pcount_r:

    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6

    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

Recursive Function Register Save

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

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6

    pushq   %rbx

    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi
    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```



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

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 >> 1	Rec. argument
%rax	x & 1	Caller-saved

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx

    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi

    call    pcount_r
    addq    %rbx, %rax
    popq    %rbx
.L6:
    ret
```

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
%rbx	x & 1	Callee-saved
%rax		Recursive call return value

```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi

    call    pcount_r

    addq    %rbx, %rax
    popq    %rbx
.L6:
    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
%rbx	x & 1	Callee-saved
%rax		Return value

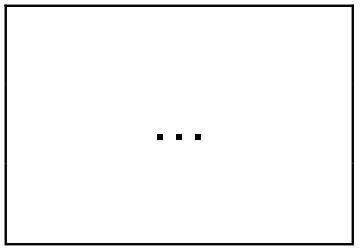
```
pcount_r:
    movl    $0, %eax
    testq   %rdi, %rdi
    je      .L6
    pushq   %rbx
    movq    %rdi, %rbx
    andl    $1, %ebx
    shrq    $1, %rdi
    call    pcount_r

    addq    %rbx, %rax

    popq    %rbx
.L6:
    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

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

    popq    %rbx
.L6:
    ret
```



← %rsp

Observations About Recursion

Handled without Special Consideration

- Stack frames mean that each function call has private storage
  - Saved registers and local variables
  - Saved return pointer
- Register saving conventions prevent one function call from corrupting another's data
  - Unless the C 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

Also works for mutual recursion

- P calls Q; Q calls P

x86-64 Procedure Summary

Important Points

- Stack is the right data structure for procedure call / return; if P calls Q, then Q returns before P

Recursion (and mutual recursion) are handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result return in `%rax`

Pointers are addresses of values (on stack or global)