[Chap.3-5] Machine-level Representation of Programs

Young Ik Eom (yieom@skku.edu, 031-290-7120)
Distributing Computing Laboratory
Sungkyunkwan University
http://dclab.skku.ac.kr

SKK UNIVERSITY

Contents

- Introduction
- Program encodings
- Data formats
- Intel processors
- Accessing information
- Primitive instructions
- Data movement instructions
- Arithmetic and logic instructions
- Control instructions
- Procedures
- …

Procedures

Procedure call

- Handled by manipulating program stack
  - Control transfer
  - Passing of data (parameters and return value)
  - Allocating space for local variables (on entry/call) and deallocating them (on exit/return)
  - Saving some register values
  - Saving return address

Procedures

Procedure call

- Stack frame
  - The portion of the stack allocated for a single procedure call
  - Stack pointer (**%rsp**)
    - ✓ Moves while the procedure is executing

Procedures

Procedure call

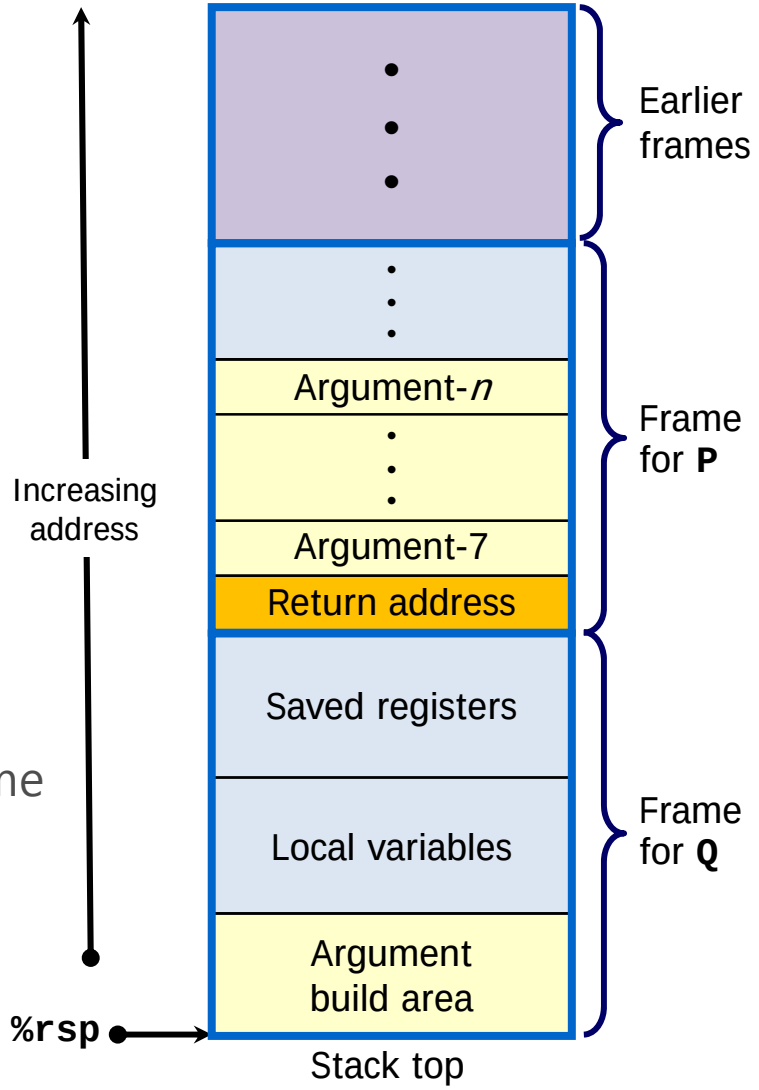
Stack frame

- When P calls Q
  - ✓ The arguments to Q are stored in some registers and P's stack frame
  - ✓ The return address is pushed onto the stack (forming the end of P's stack frame)
  - ✓ Q's stack frame starts with some register values that should be saved in the Q's frame
  - ✓ Some local variables of Q are followed in the Q's frame

param 저장 순서

rdi - rsi - rdx - rcx - r8 - r9

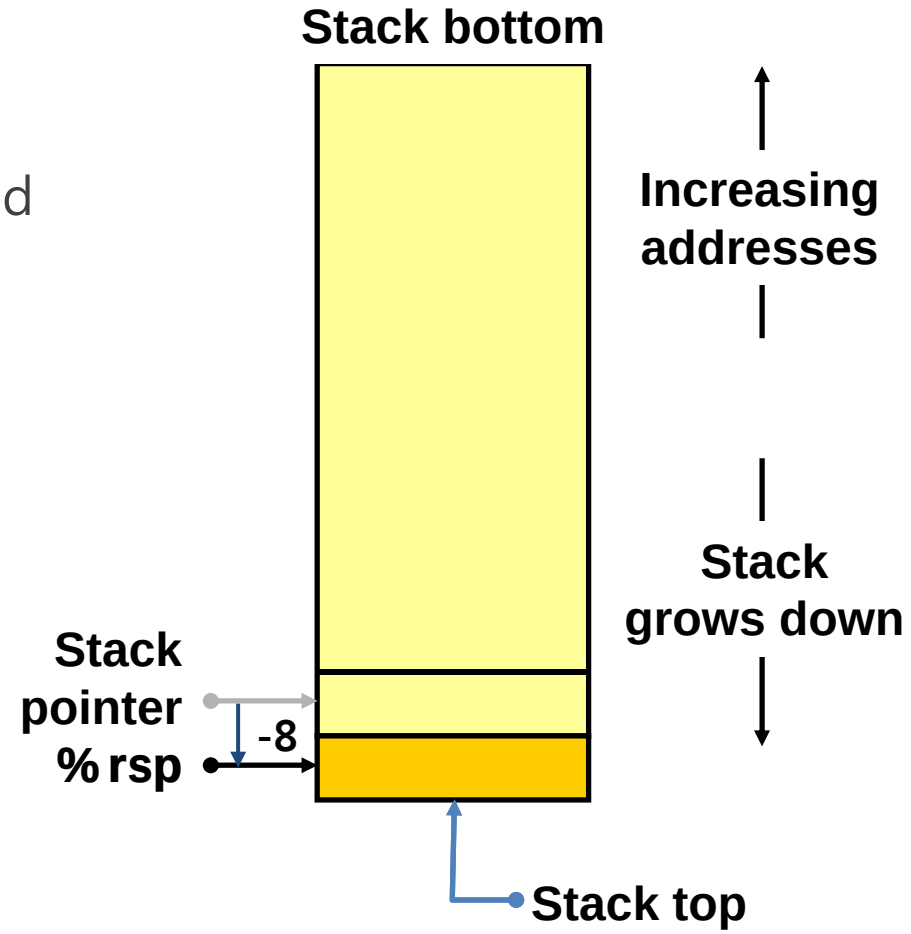
개 이상부턴 → stack



Procedures

Pushing onto the stack

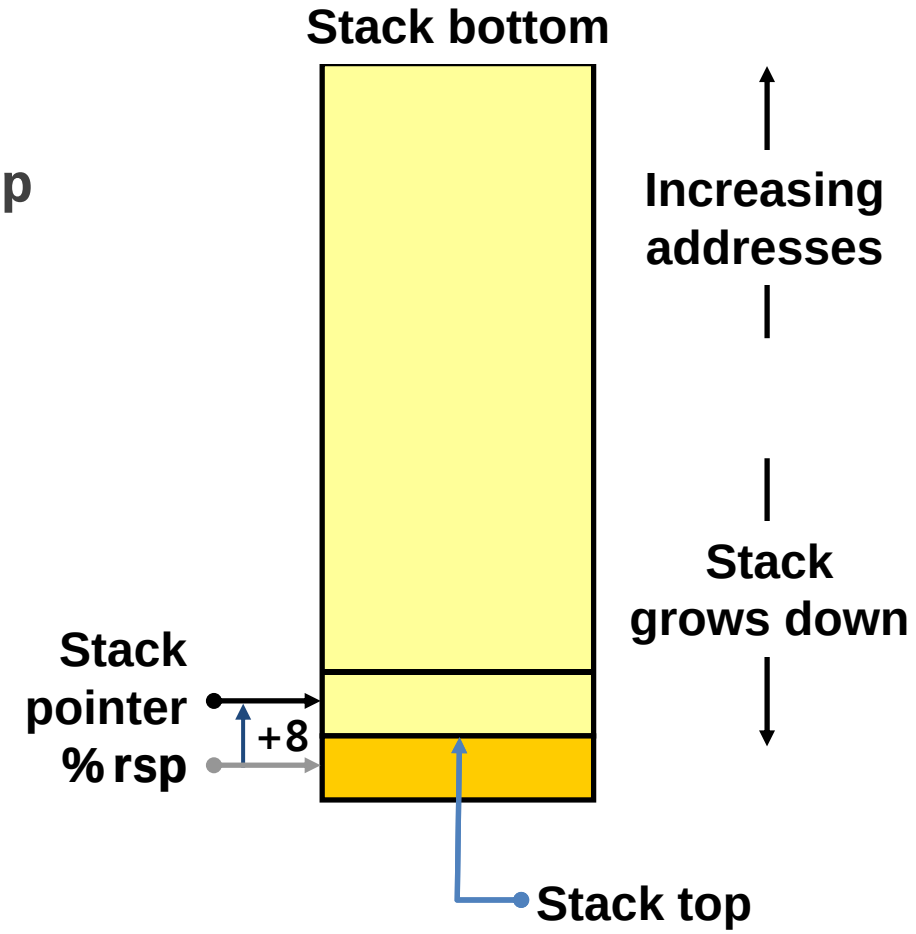
- **pushq src**
  - Decrement **%rsp** by 8
  - Fetch operand at **src** and write it at address given by **%rsp**



Procedures

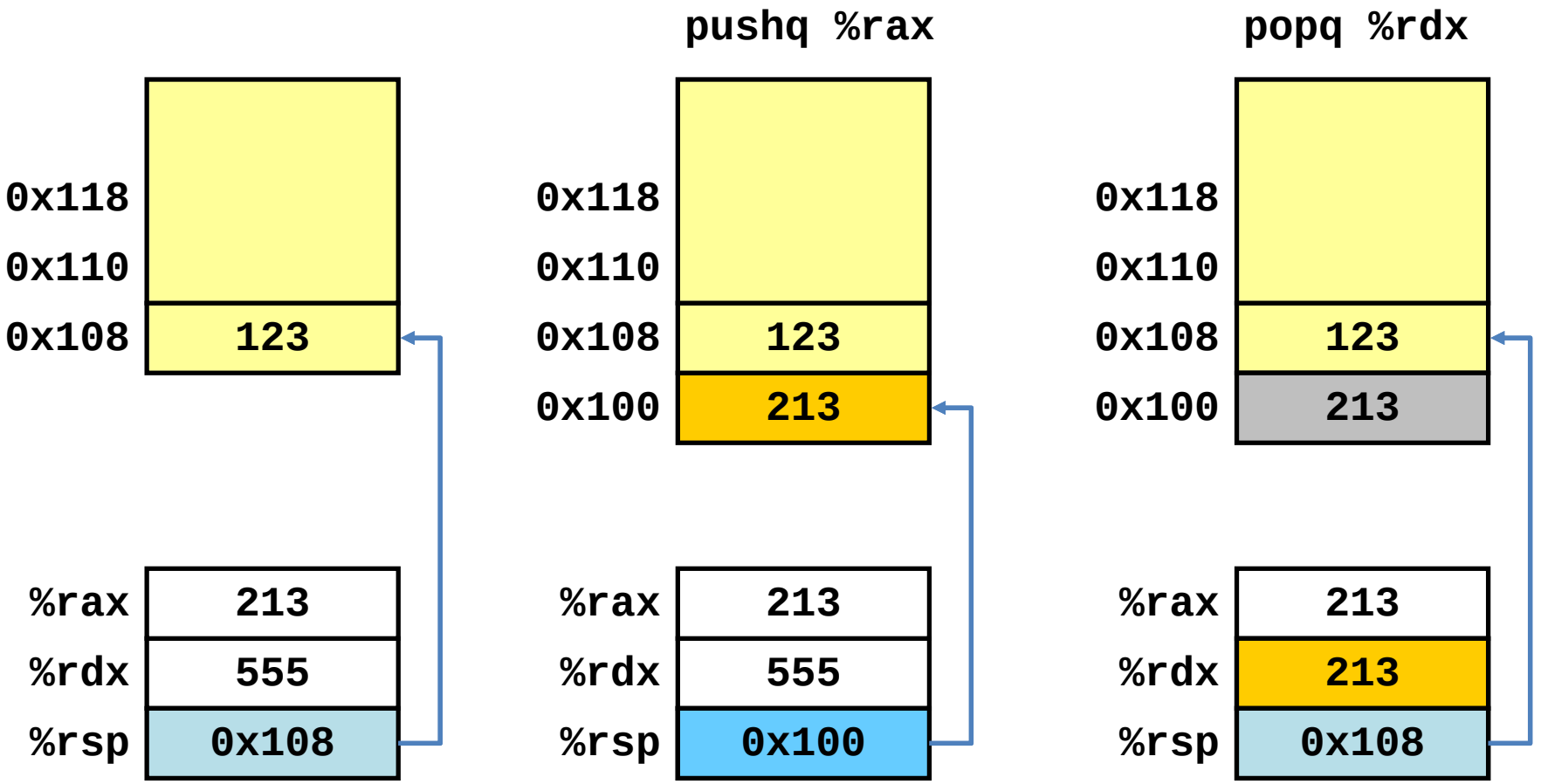
Popping off the stack

- **popq dst**
  - Read operand at address given by **%rsp** and write to **dst**
  - Increment **%rsp** by 8



Procedures

Stack operation example



Procedures

Transferring control

- **call** instruction
  - Pushes a return address onto the stack and jumps to the target
    - ✓ Direct call
      - Target as a label
    - ✓ Indirect call
      - Target given by a * followed by an operand specifier

Instruction		Description
`call`	*Label*	Procedure call
`call`	**Operand*	Procedure call
`ret`		Return from call

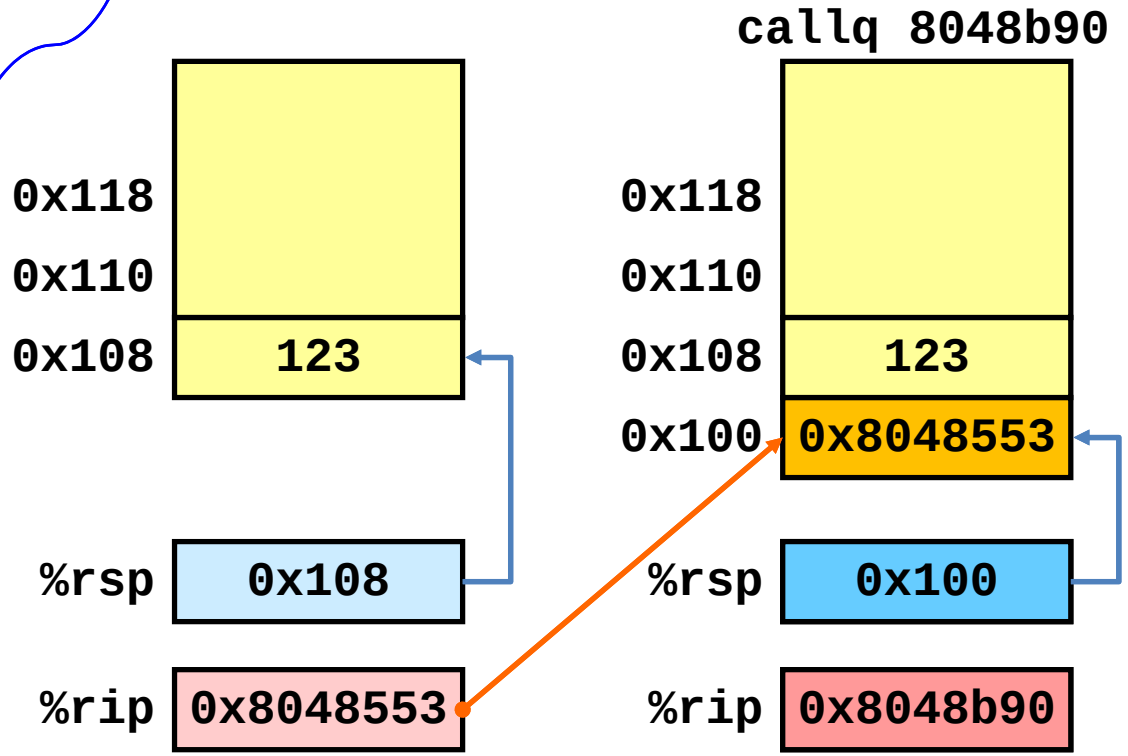
- **ret** instruction
  - Pops an address off the stack and jumps to the location

Procedures

Transferring control: Example) call

```
804854e:  e8 3d 06 00 00        callq  8048b90 <main>
8048553:  48 89 c2              mov    %rax,%rdx
```

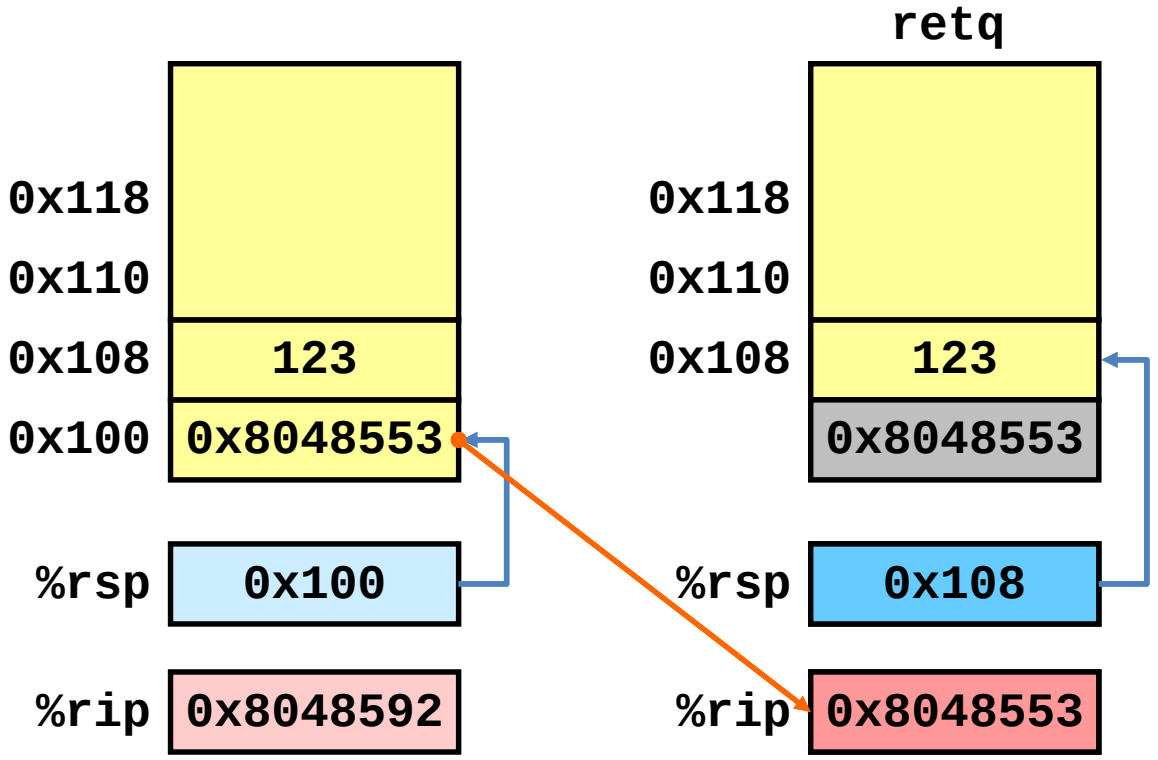
```
  0x08048553
+ 0x0000063d
= 0x08048b90
```



Procedures

Transferring control: Example) return

```
8048591:   c3                                   retq
```



Procedures

- Example) Stack frame

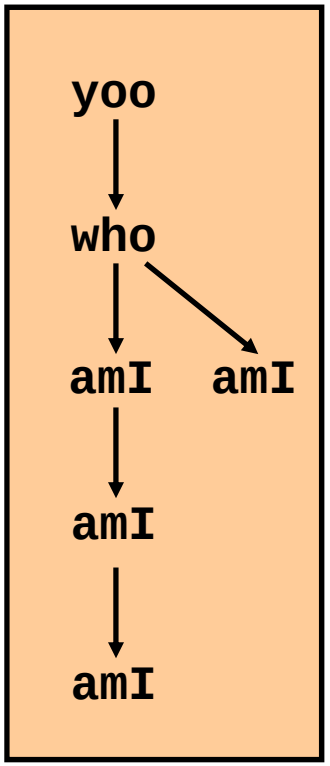
```
yoo(…)
{
    •
    •
   who();
    •
    •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

```
amI(…)
{
    •
    •
   amI();
    •
    •
}
```

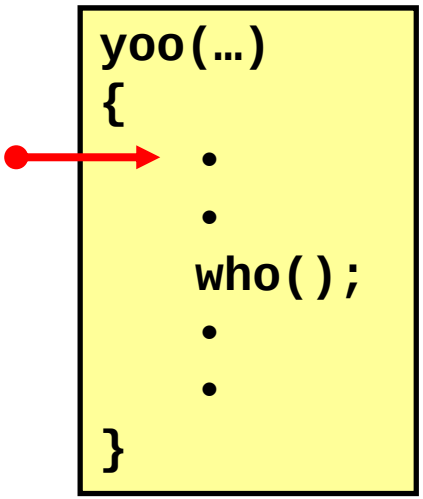
Code structure

Call Chain



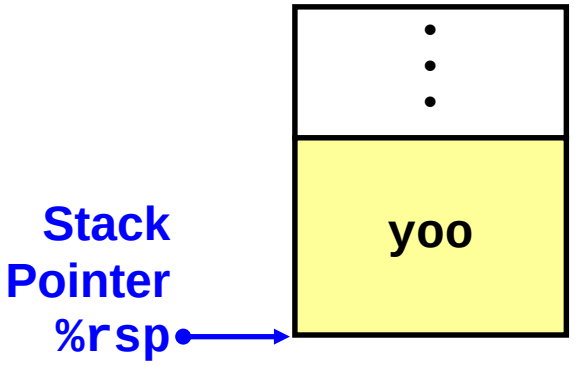
Procedures

- Example) Stack frame



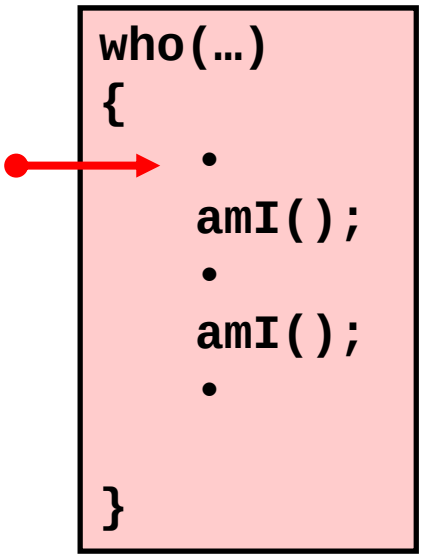
Call Chain

yoo

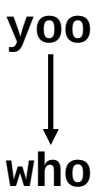


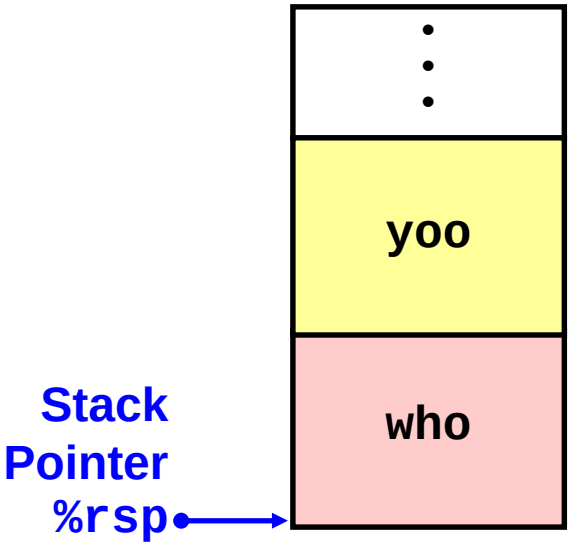
Procedures

- Example) Stack frame



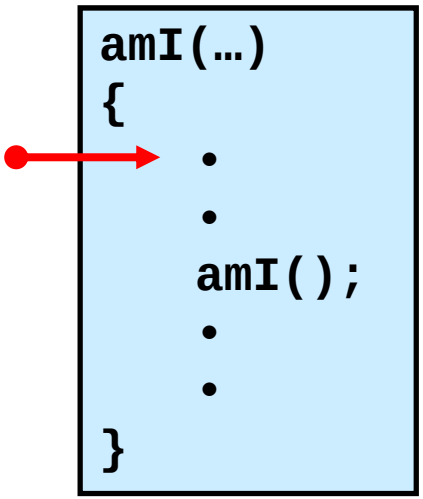
Call Chain



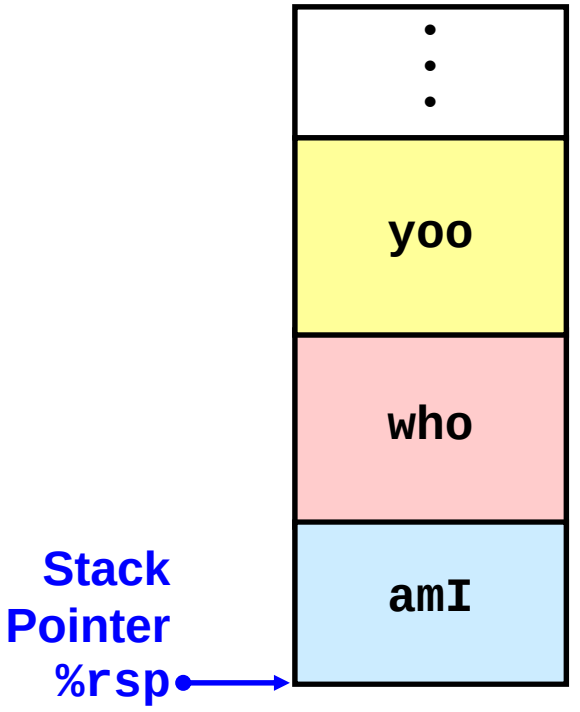


Procedures

- Example) Stack frame

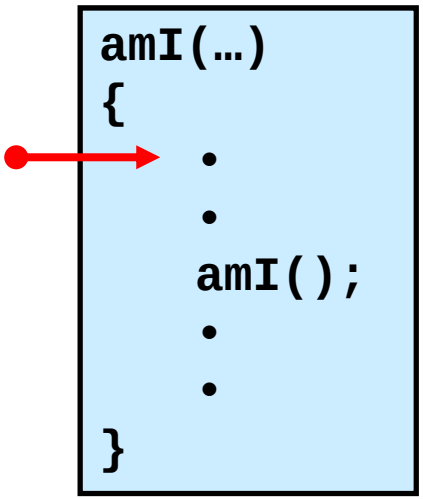


Call Chain



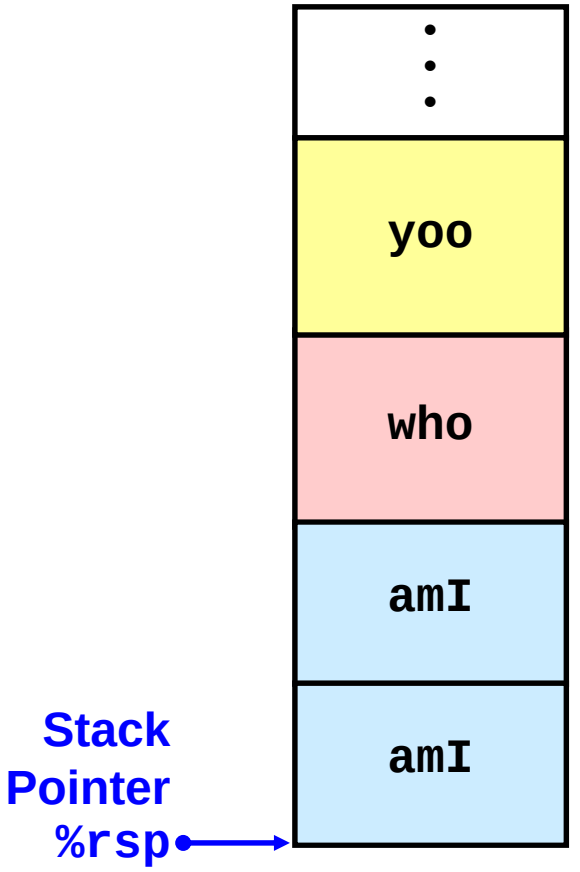
Procedures

- Example) Stack frame



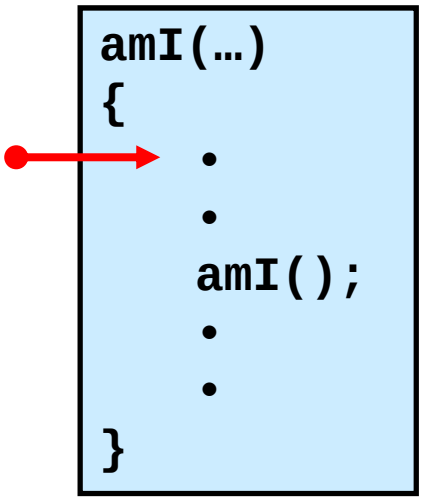
Call Chain

yoo
who
amI
amI



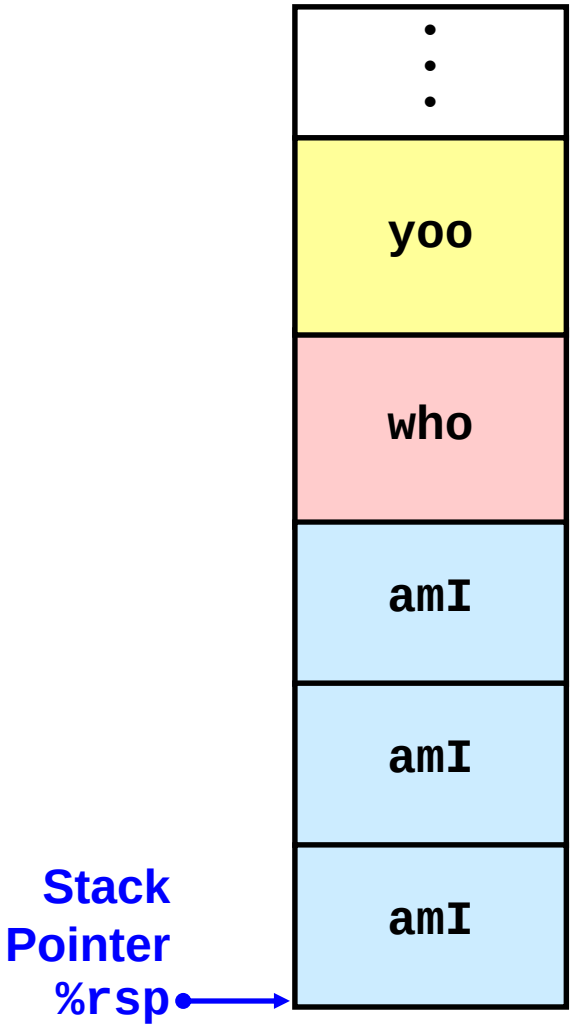
Procedures

- Example) Stack frame



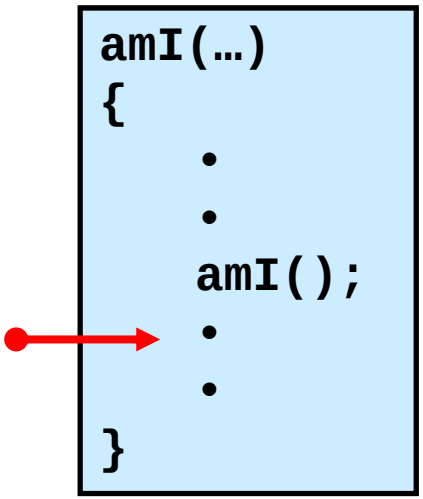
Call Chain

yoo
who
amI
amI
amI

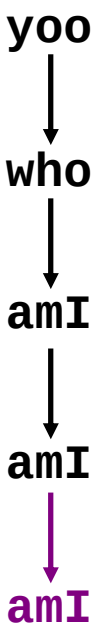


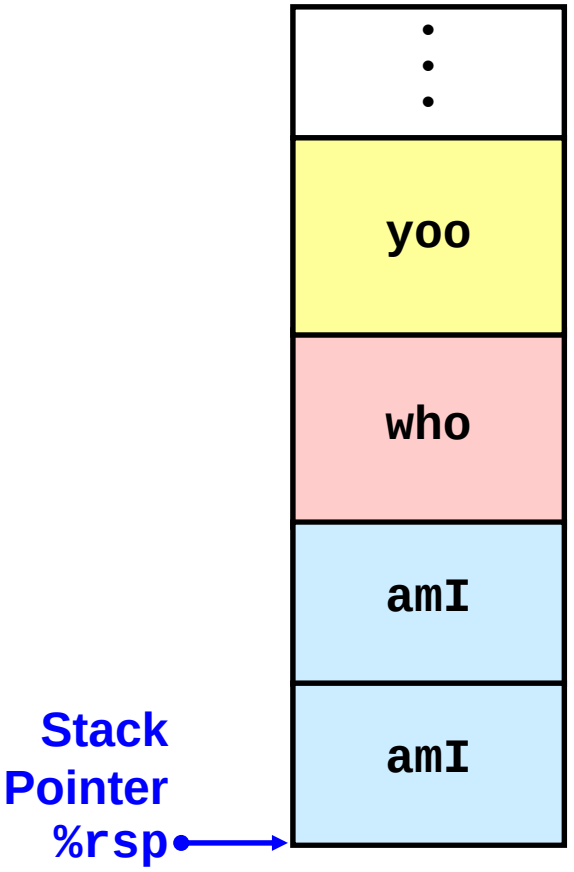
Procedures

- Example) Stack frame



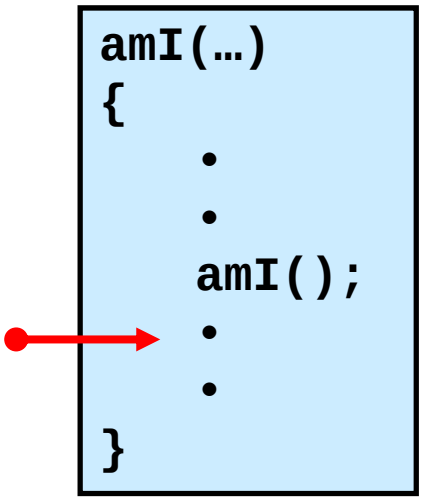
Call Chain



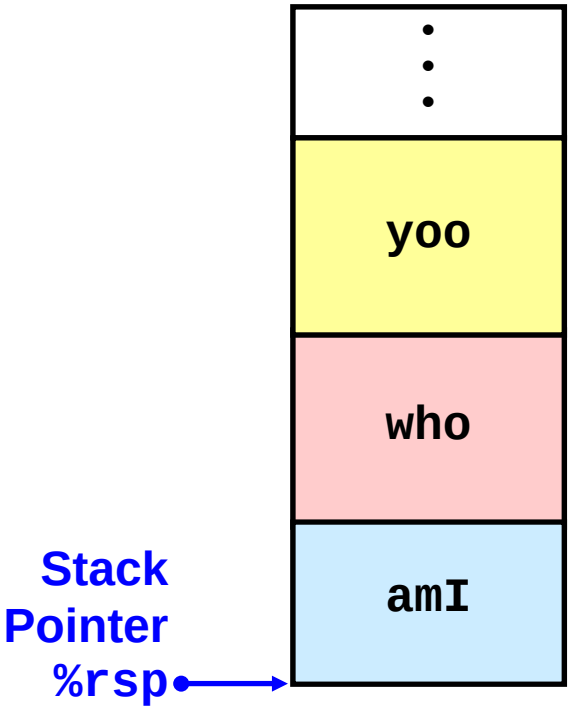


Procedures

- Example) Stack frame

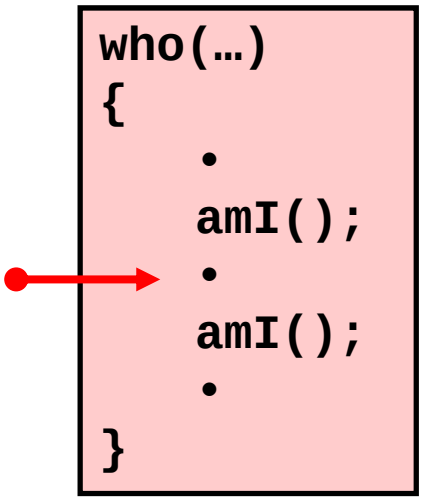


Call Chain

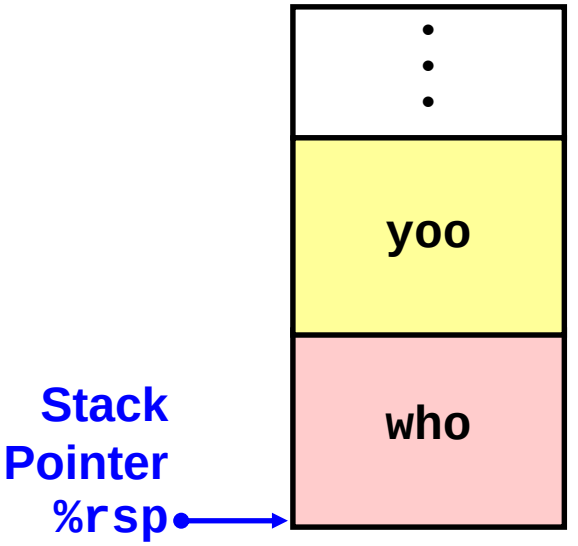


Procedures

- Example) Stack frame

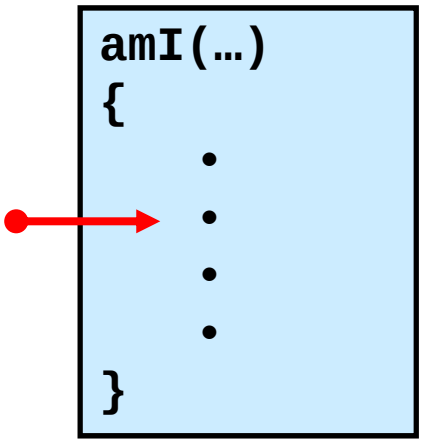


Call Chain

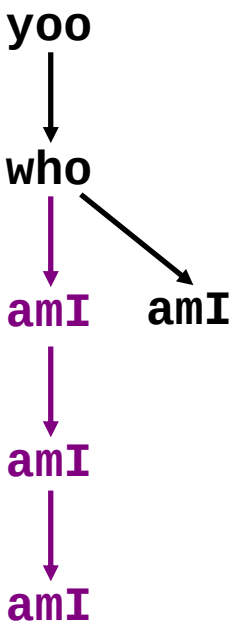


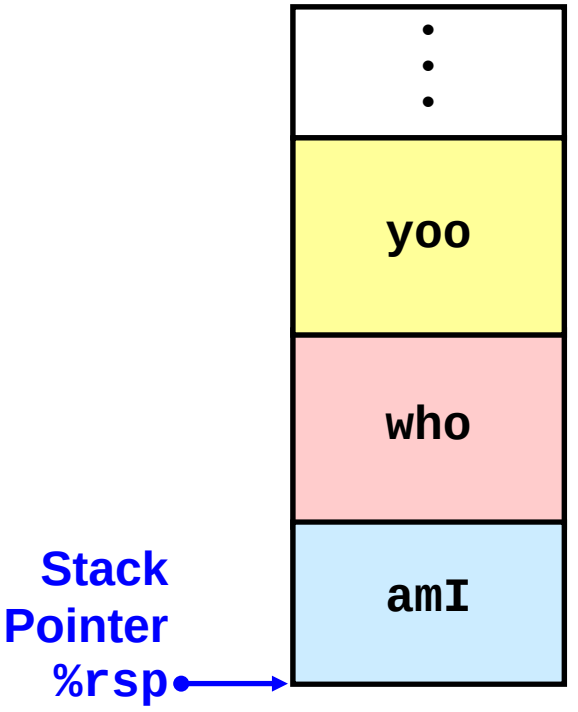
Procedures

- Example) Stack frame



Call Chain



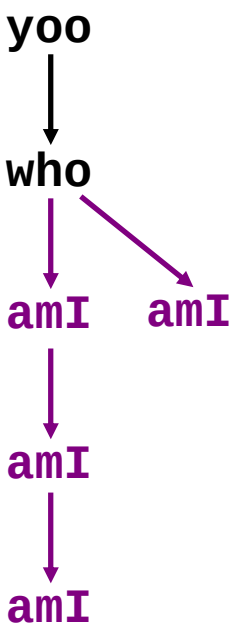


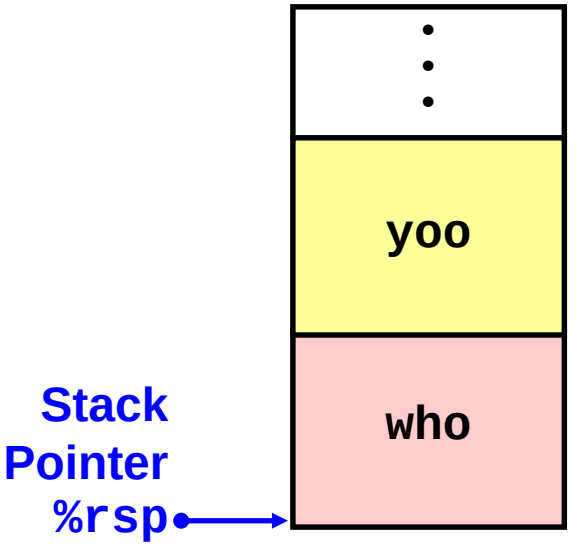
Procedures

- Example) Stack frame

```
who(…)
{
    •
    amI();
    •
    amI();
    •
}
```

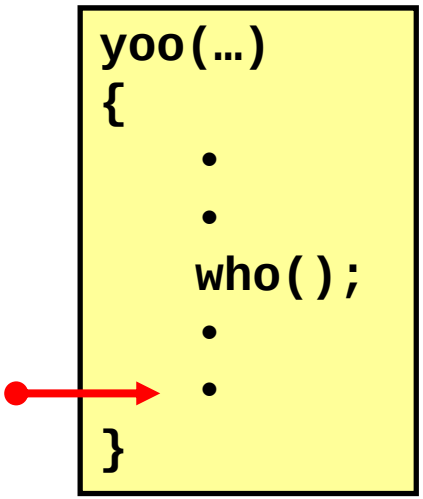
Call Chain



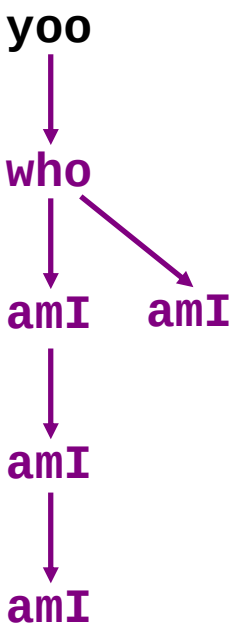


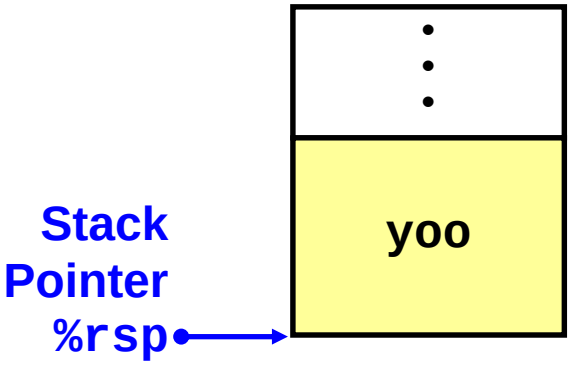
Procedures

- Example) Stack frame



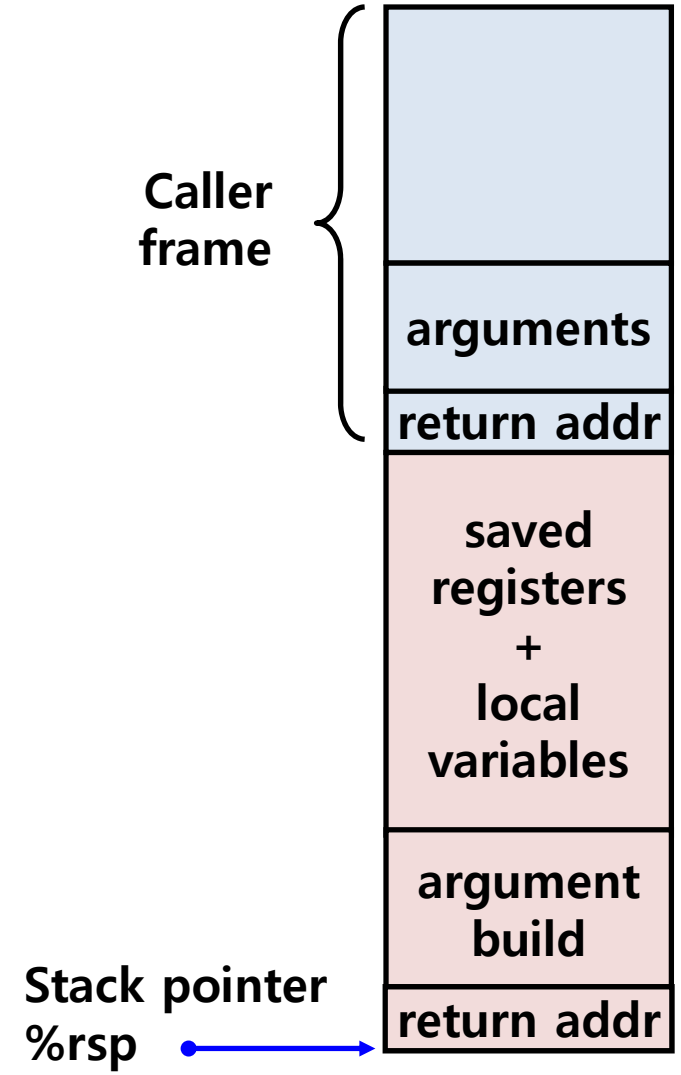
Call Chain





Procedures

- Stack frame revisited



Procedures

Data transfer

- Most of data passing to and from procedures take place via registers
  - With x86-64, up to 6 integral (i.e., integer and pointer) arguments can be passed via registers
- When a function has more than 6 integral arguments, the other ones are passed on the stack
- Return values are passed through the register **%rax**

Procedures

Data transfer

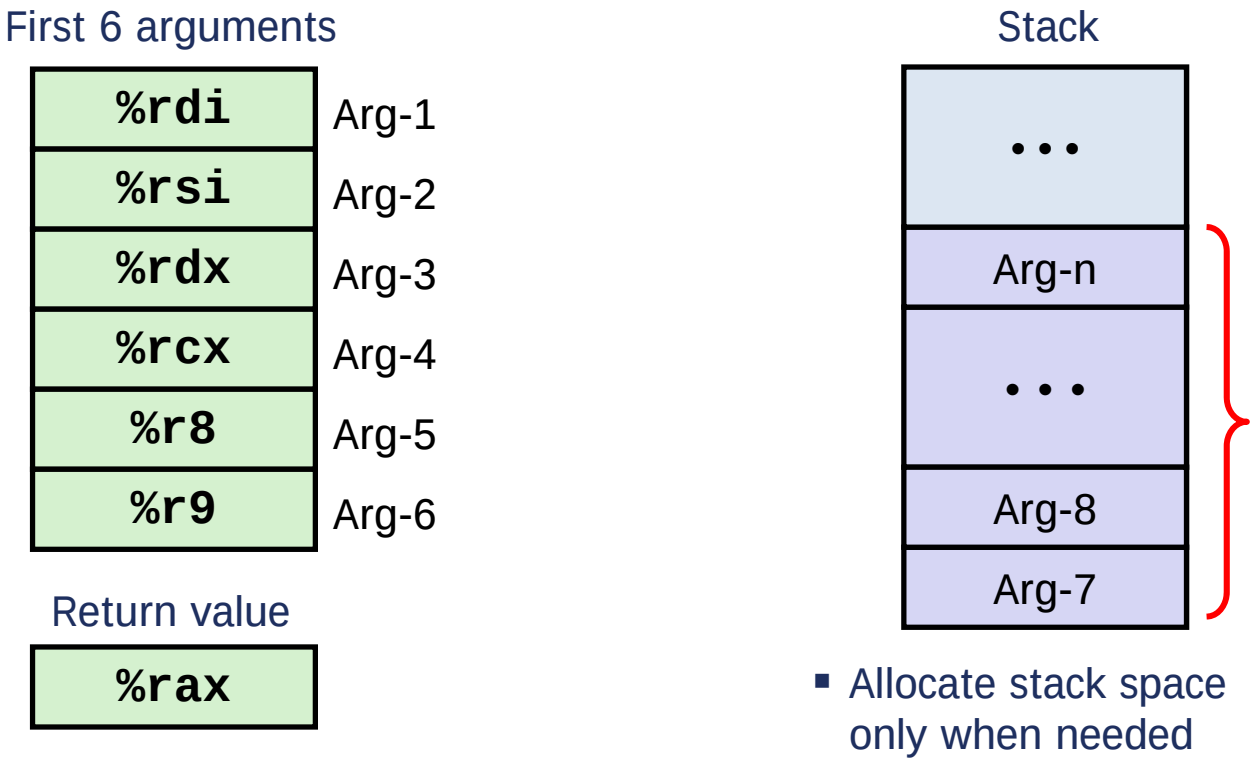
- Argument passing by registers
  - ✓ Up to first 6 integral arguments

Operand size (bits)	Argument Number 1	2	3	4	5	6
64	%rdi	%rsi	%rdx	%rcx	%r8	%r9
32	%edi	%esi	%edx	%ecx	%r8d	%r9d
16	%di	%si	%dx	%cx	%r8w	%r9w
8	%dil	%sil	%dl	%cl	%r8b	%r9b

Procedures

Data transfer

- For more than 6 integral arguments
  - Argument passing on the stack



- Allocate stack space only when needed

Procedures

Data transfer

- Example) Argument passing

```
void proc(long a1,  long *a1p,
          int a2,   int *a2p,
          short a3, short *a3p,
          char a4, char *a4p)
{
   *a1p += a1;
   *a2p += a2;
   *a3p += a3;
   *a4p += a4;
}
```

```
a1   in %rdi    (64 bits)
a1p  in %rsi    (64 bits)
a2   in %edx    (32 bits)
a2p  in %rcx    (64 bits)
a3   in %r8w    (16 bits)
a3p  in %r9     (64 bits)
a4   at %rsp+8  (8 bits)
a4p  at %rsp+16 (64 bits)
```

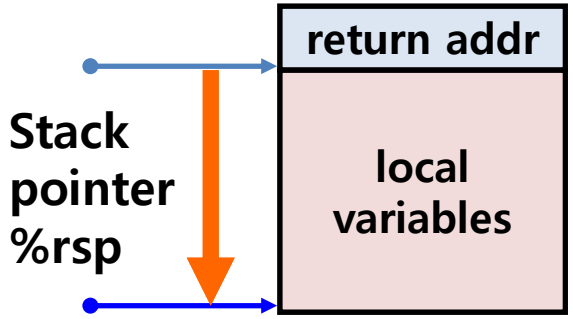
```
proc:
  addq   %rdi,(%rsi)    # *a1p += a1 (64 bits)
  addl   %edx,(%rcx)    # *a2p += a2 (32 bits)
  addw   %r8w,(%r9)     # *a3p += a3 (16 bits)
  movq   16(%rsp),%rax  # Fetch a4p  (64 bits)
  movb   8(%rsp),%dl    # Fetch a4   (8 bits)
  addb   %dl,(%rax)     # *a4p += a4 (8 bits)
  ret
```

Procedures

Local storage on the stack

- Generally, local data are stored in processor registers
- But, there are some cases that local data should be stored in memory
  - When there are not enough registers to hold all of the local data
  - When the address operator & is applied to a local variable
    - ✓ Address of the local data must be generated
  - When some of the local variables are arrays or structures

- A procedure allocates space on the stack frame by decrementing the stack pointer



Procedures

Local storage on the stack

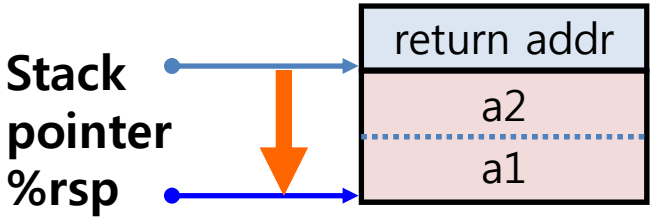
- Example)

```
long s_add(long *xp, long *yp)
{
    long x = *xp;
    long y = *yp;
    *xp = y;
    *yp = x;
    return x + y;
}
```

```
long caller()
{
    long a1 = 534;
    long a2 = 1057;
    long sum = s_add(&a1, &a2);
    long diff = a1 - a2;
    return sum * diff;
}
```

```
    long caller()

caller:
 subq $16,%rsp          Allocate 16 bytes for stack frame
 movq $534,(%rsp)       Store 534 in a1
 movq $1057,8(%rsp)     Store 1057 in a2
 leaq 8(%rsp),%rsi      Compute &a2 as second argument
 movq %rsp,%rdi         Compute &a1 as first argument
 call s_add             Call s_add(&a1,&a2)
 movq (%rsp),%rdx       Get a1
 subq 8(%rsp),%rdx      Compute diff = a1 - a2
 imulq %rdx,%rax        Compute sum * diff
 addq $16,%rsp          Deallocate stack frame
 ret                    Return
```



Procedures

Local storage in registers

- Register usage conventions
  - When one procedure (the caller) calls another (the callee), the callee may not overwrite some register values that the caller planned to use later
- Example) When **yoo()** calls **who()**
  - How about register **%rdx** ?

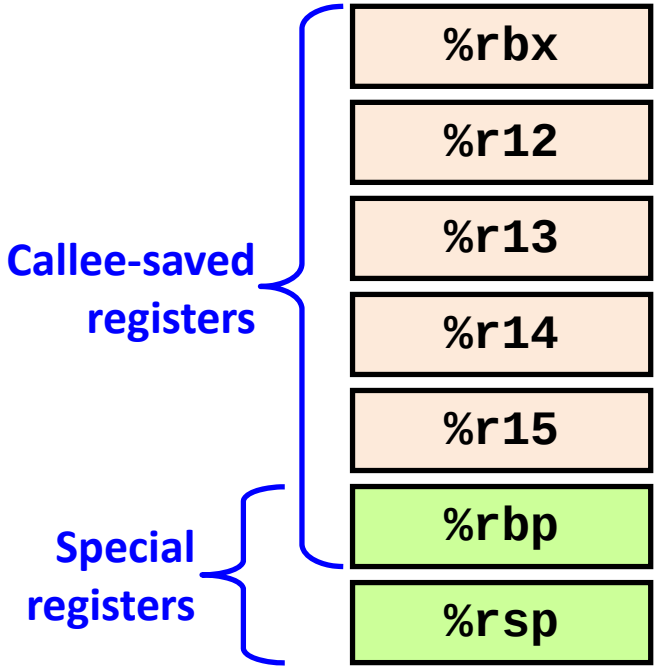
```
yoo:
    • • •
    movl $15213,%edx
    call who
    addq %rdx,%rax
    • • •
    ret
```

```
who:
    • • •
    movq 8(%rsp),%rdx
    addq $91125,%rdx
    • • •
    ret
```

Procedures

Local storage in registers

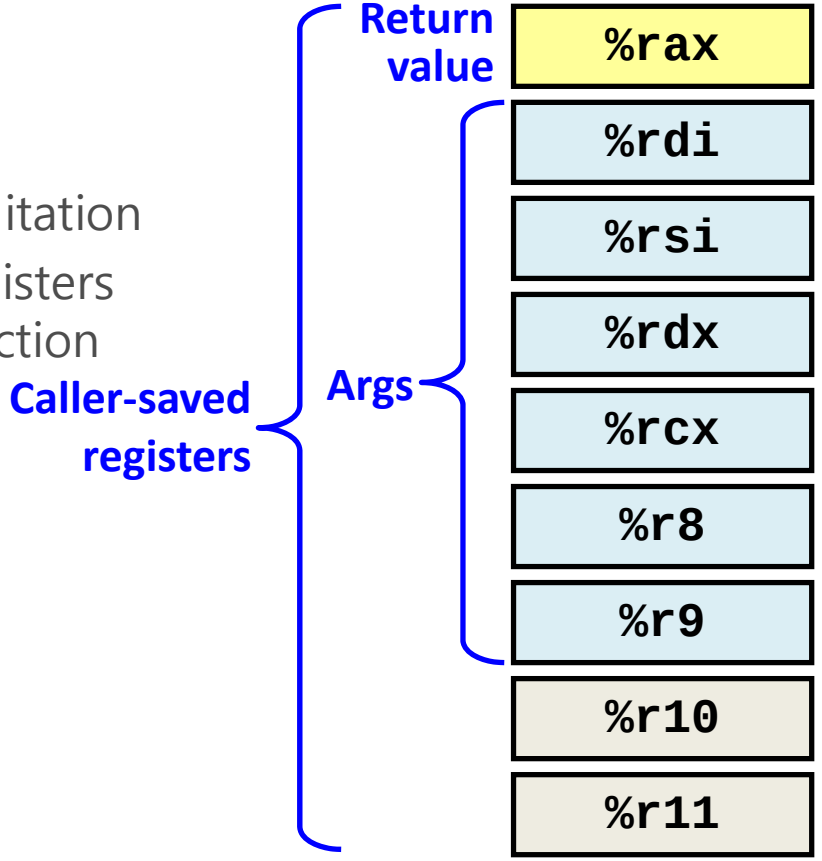
- Register usage conventions
  - Callee-saved registers
    - ✓ Callee must save these registers before using them and restore them before returning
    - ✓ **%rbx**, **%rbp**, **%r12 ~ %r15**
  - Special (callee-saved) register
    - ✓ **%rsp**



Procedures

Local storage in registers

- Register usage conventions
  - Caller-saved registers
    - ✓ All other registers
    - ✓ Callee can use (overwrite) these registers with no limitation
    - ✓ Caller must save these registers before calling another function



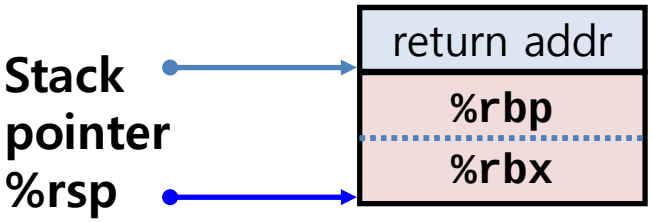
Procedures

Local storage in registers

- Example)

```
long P(long x, long y)
{
   long u = Q(y);
   long v = Q(x);
   return u + v;
}
```

```
    long P(long x, long y)
    x in %rdi, y in %rsi
P:
    pushq %rbp          Save %rbp
    pushq %rbx          Save %rbx
    subq $8,%rsp        Align stack frame
    movq %rdi,%rbp      Save x
    movq %rsi,%rdi      Move y to first argument
    call Q              Call Q(y)
    movq %rax,%rbx      Save result
    movq %rbp,%rdi      Move x to first argument
    call Q              Call Q(x)
    addq %rbx,%rax      Add saved Q(y) to Q(x)
    addq $8,%rsp        Deallocate last part of stack
    popq %rbx           Restore %rbx
    popq %rbp           Restore %rbp
    ret
```

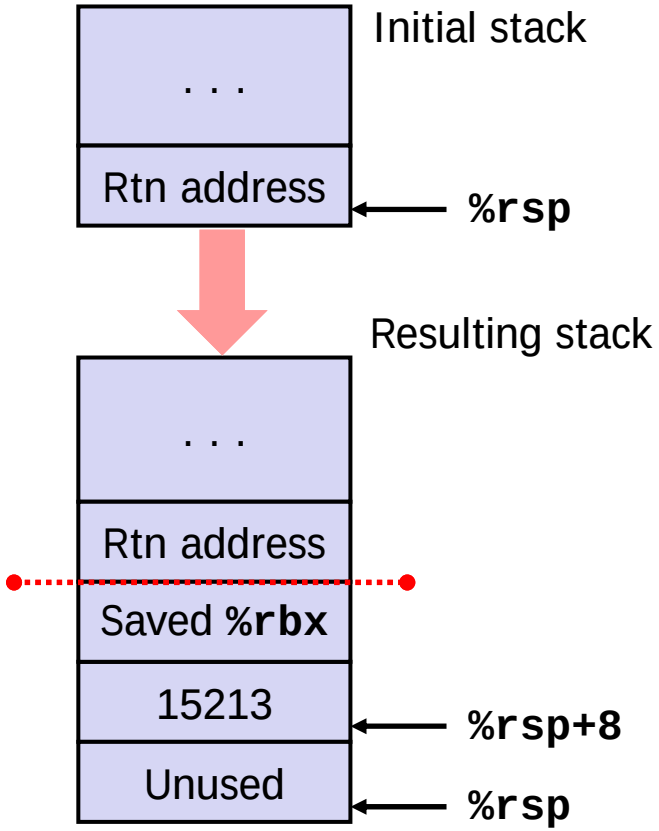


Procedures

Example) Local data and register saving

```
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
```

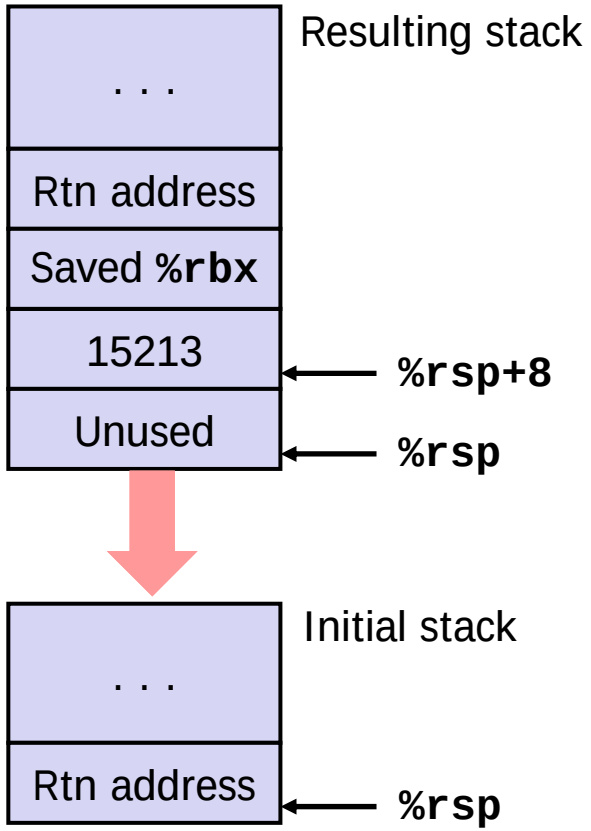


Procedures

Example) Local data and register saving

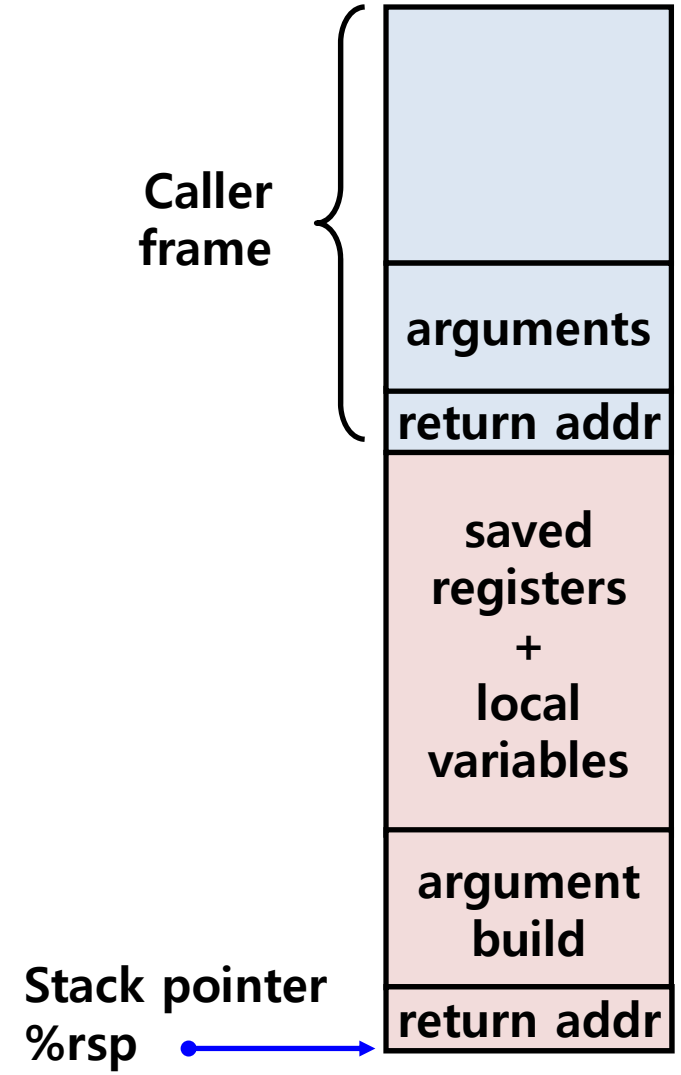
```
long call_incr2(long x) {
    long v1 = 15213;
    long v2 = incr(&v1, 3000);
    return x + v2;
}
```

```
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
```



Procedures

- Stack frame revisited



Summary