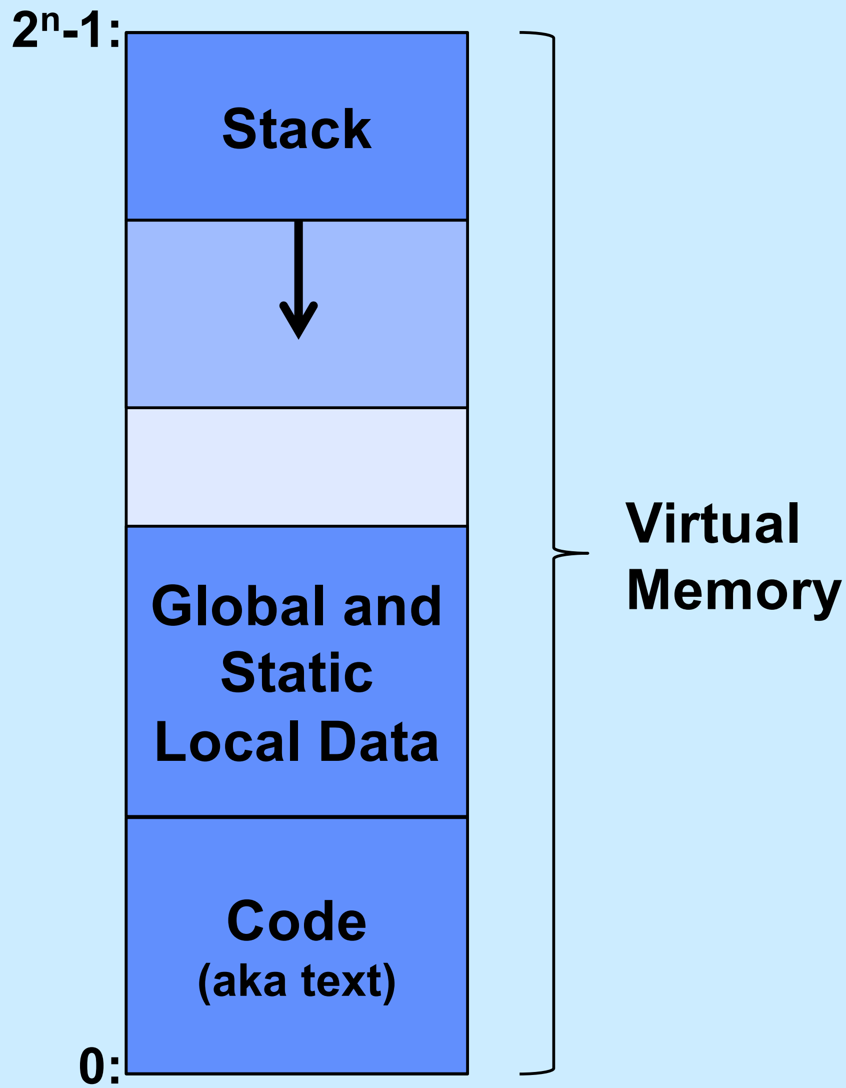
CS 33

Machine Programming (4)

Digression (Again): Where Stuff Is (Roughly)



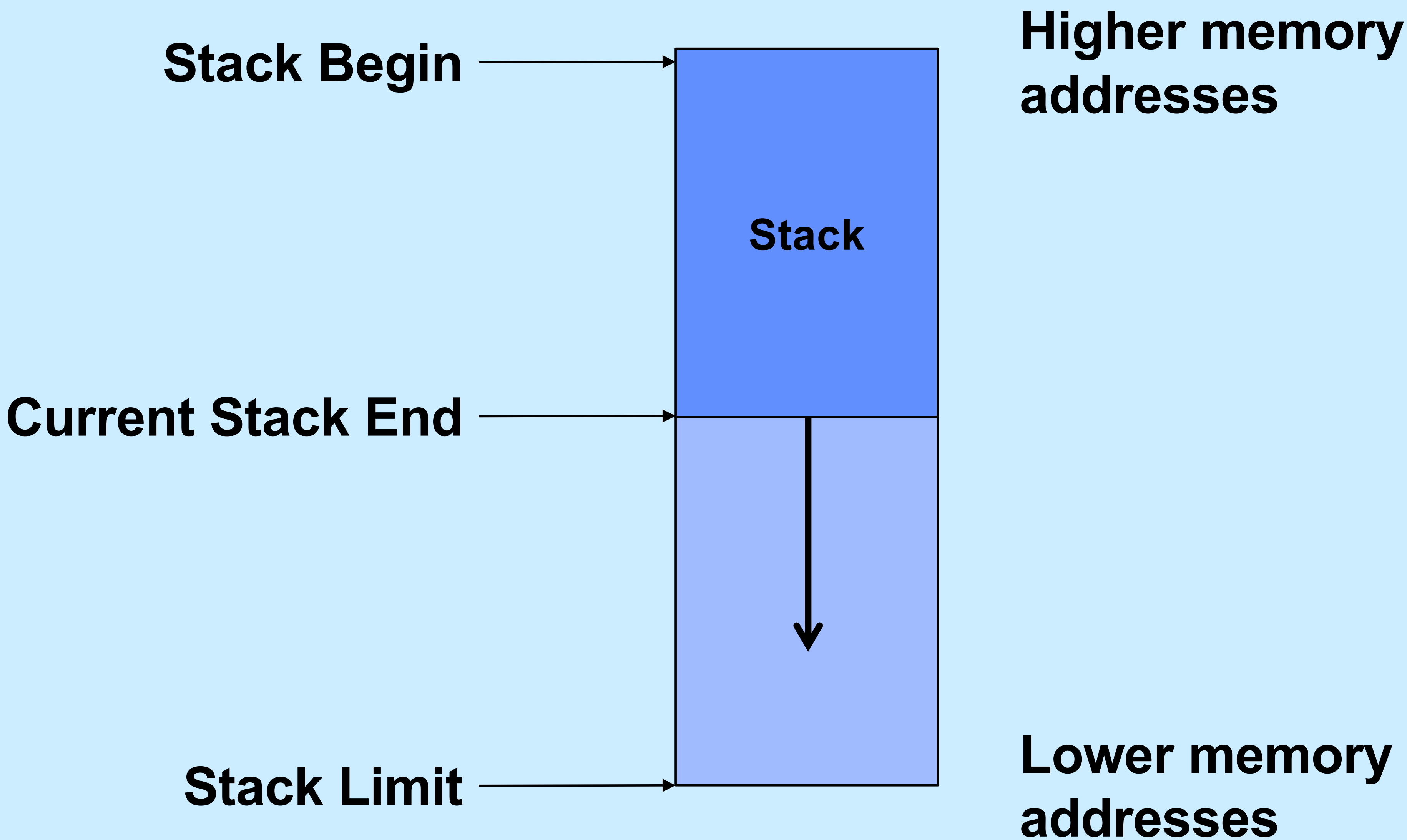
Function Call and Return

- Function A calls function B
- Function B calls function C

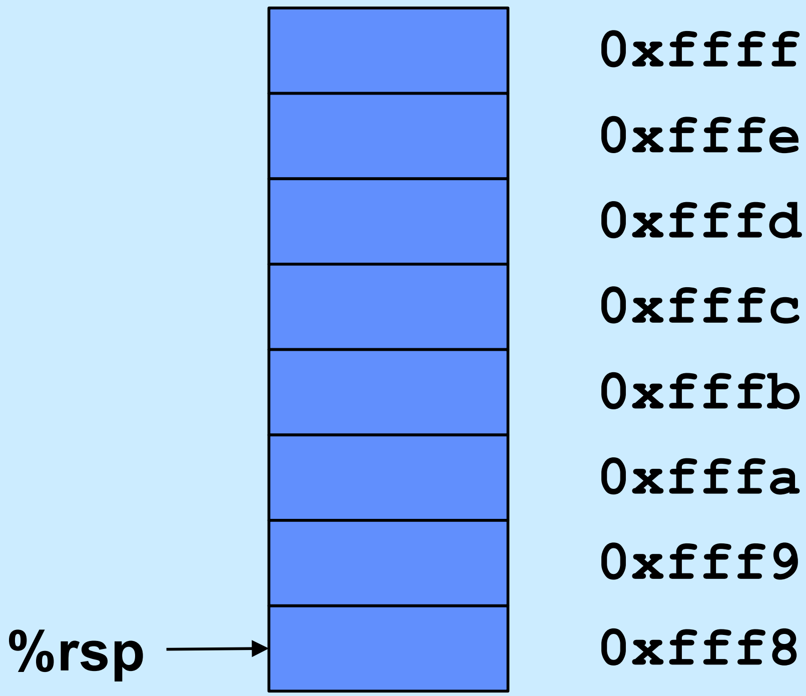
... several million instructions later

- C returns
  – how does it know to return to B?
- B returns
  – how does it know to return to A?

The Runtime Stack

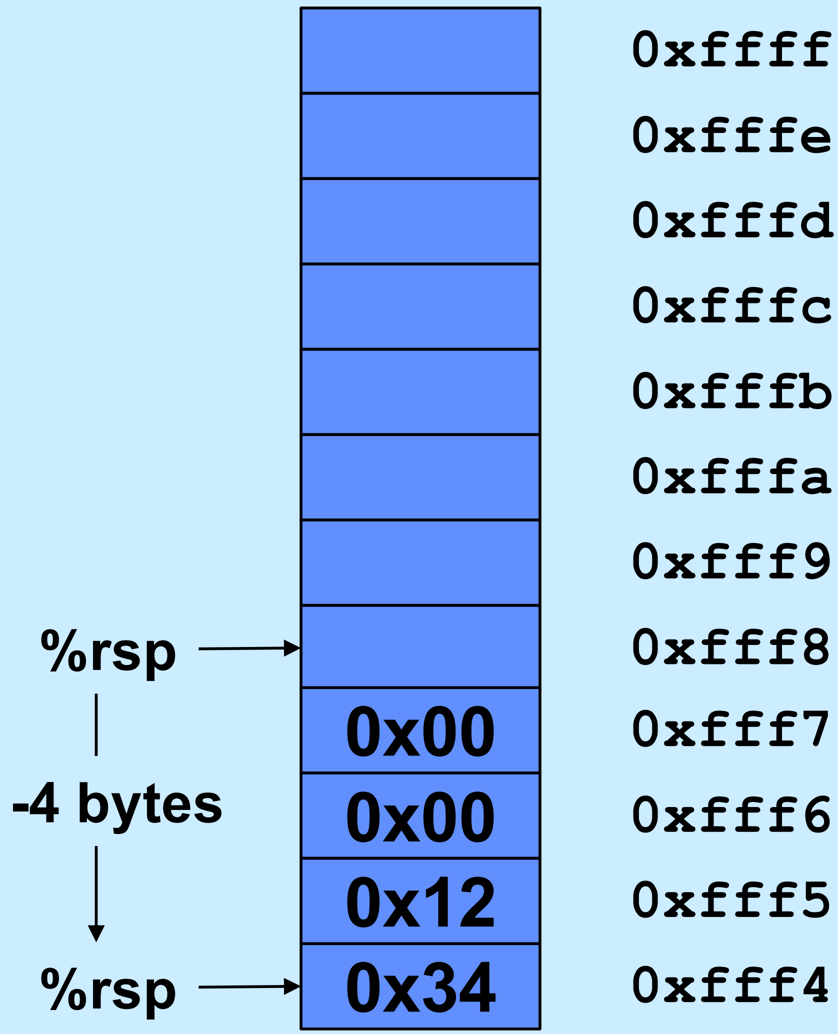


Stack Operations



Push

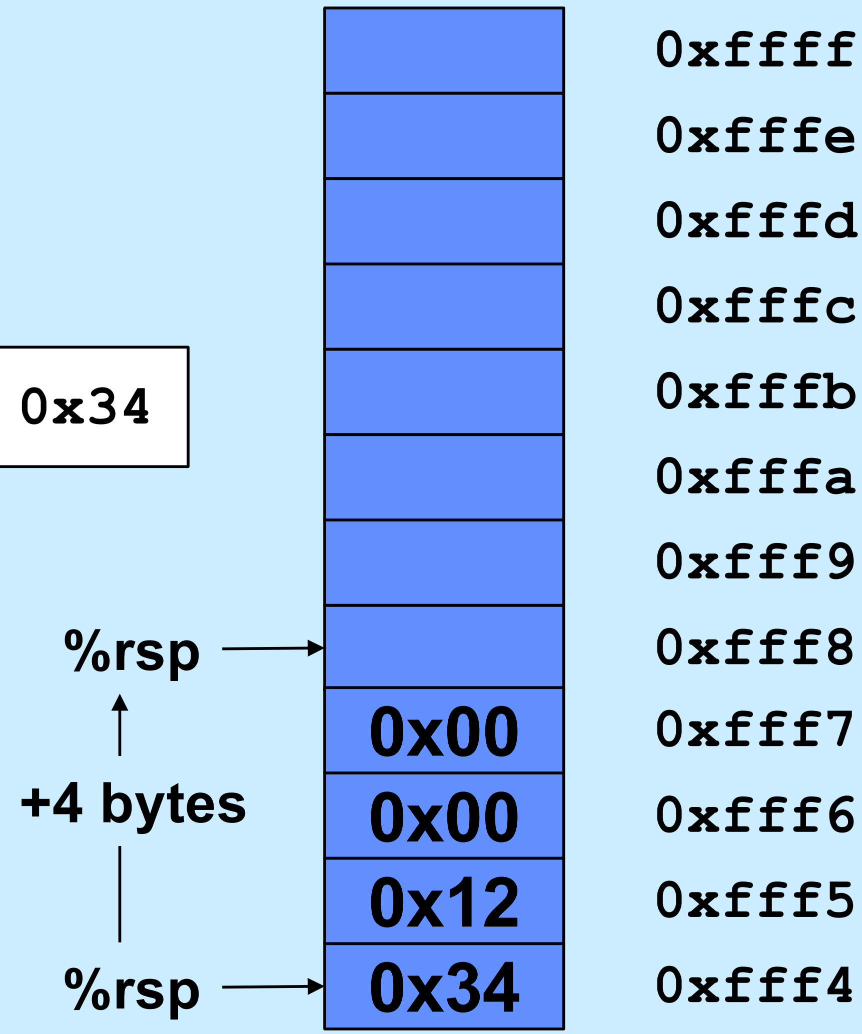
```
pushl $0x1234
```



Pop

```
popl %r8d
```

%r8d:				
	0x00	0x00	0x12	0x34



Call and Return

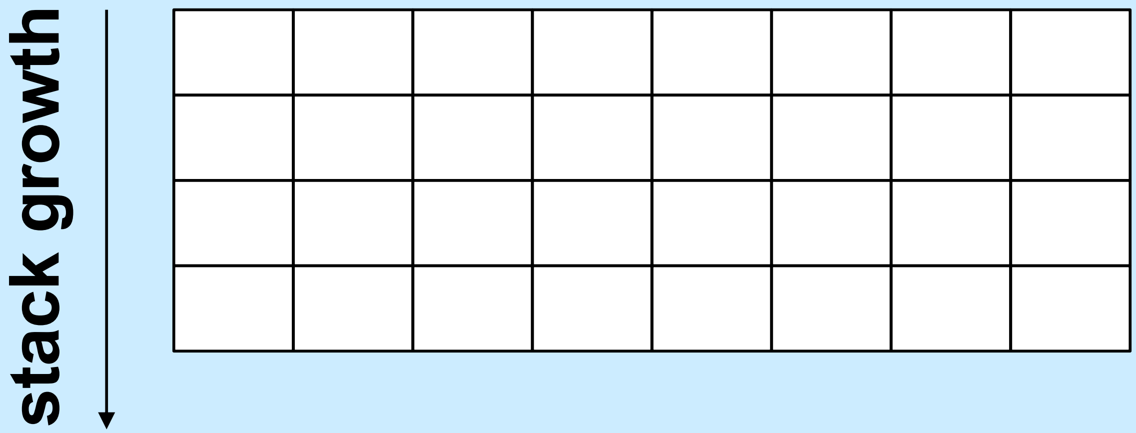
```
0x1000: call func
0x1004: addq $3, %rax
```

```
0x2000: func:
    ... ...
0x2200: movq $6, %rax
0x2203: ret
```

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```



0xffff10018
0xffff10010
0xffff10008
0xffff10000

								%rax
00	00	00	00	00	00	10	00	%rip
00	00	00	0f	ff	f1	00	00	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

								%rax
00	00	00	00	00	00	20	00	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	22	03	%rip
00	00	00	0f	ff	f0	ff	f8	%rsp

Call and Return

```
0x2000:  func:
   ...   ...
0x2200:  movq $6, %rax
0x2203:  ret
```

```
0x1000:  call func
0x1004:  addq $3, %rax
```

stack growth

								0xffff10018
								0xffff10010
								0xffff10008
								0xffff10000
00	00	00	00	00	00	10	04	0xffff0fff8

00	00	00	00	00	00	00	06	%rax
00	00	00	00	00	00	10	04	%rip
00	00	00	0f	ff	f1	00	00	%rsp

Arguments and Local Variables

```
int mainfunc() {
   long array[3] =
      {2,117,-6};
   long sum =
      ASum(array, 3);
   ...
   return sum;
}
```

```
long ASum(long *a,
         unsigned long size) {
   long i, sum = 0;
   for (i=0; i<size; i++)
      sum += a[i];
   return sum;
}
```

- **Local variables usually allocated on stack**
- **Arguments to functions pushed onto stack**

- **Local variables may be put in registers (and thus not on stack)**

Arguments and Local Variables

```
mainfunc:
    pushq %rbp                   # save old %rbp
    movq %rsp, %rbp              # set %rbp to point to stack frame
    subq $32, %rsp               # alloc. space for locals (array and sum)
    movq $2, -32(%rbp)           # initialize array[0]
    movq $117, -24(%rbp)         # initialize array[1]
    movq $-6, -16(%rbp)          # initialize array[2]
    pushq $3                     # push arg 2
    leaq -32(%rbp), %rax         # array address is put in %rax
    pushq %rax                   # push arg 1
    call ASum
    addq $16, %rsp               # pop args
    movq %rax, -8(%rbp)          # copy return value to sum
    addq $32, %rsp               # pop locals
    popq %rbp                    # pop and restore old %rbp
    ret
```

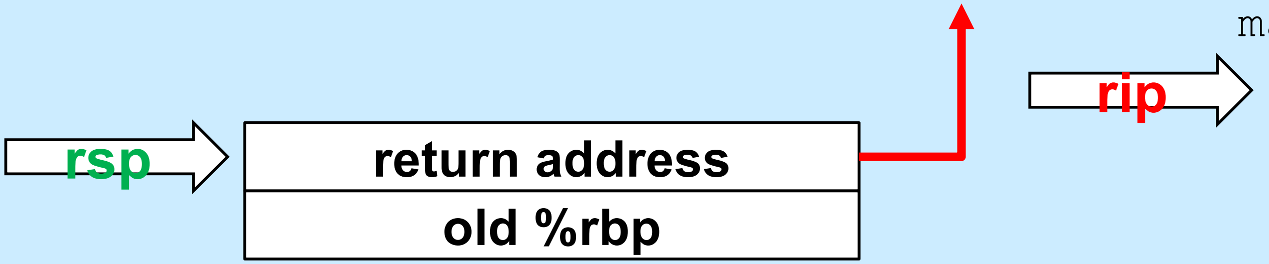
Arguments and Local Variables

```
ASum:
   pushq %rbp                  # save old %rbp
   movq %rsp, %rbp             # set %rbp to point to stack frame
   movq $0, %rcx               # i in %rcx
   movq $0, %rax               # sum in %rax
   movq 16(%rbp), %rdx         # copy arg 1 (array) into %rdx
loop:
   cmpq 24(%rbp), %rcx         # i < size?
   jge done
   addq (%rdx,%rcx,8), %rax    # sum += a[i]
   incq %rcx                   # i++
   ja loop
done:
   popq %rbp                   # pop and restore %rbp
   ret
```

Enter mainfunc

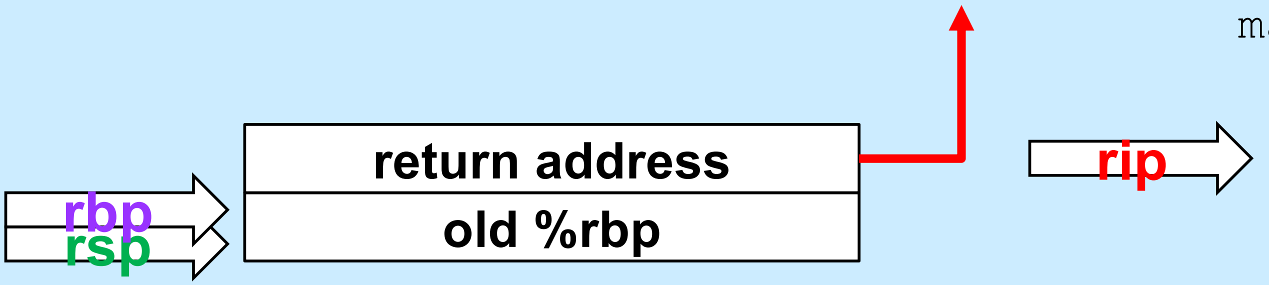

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Enter mainfunc



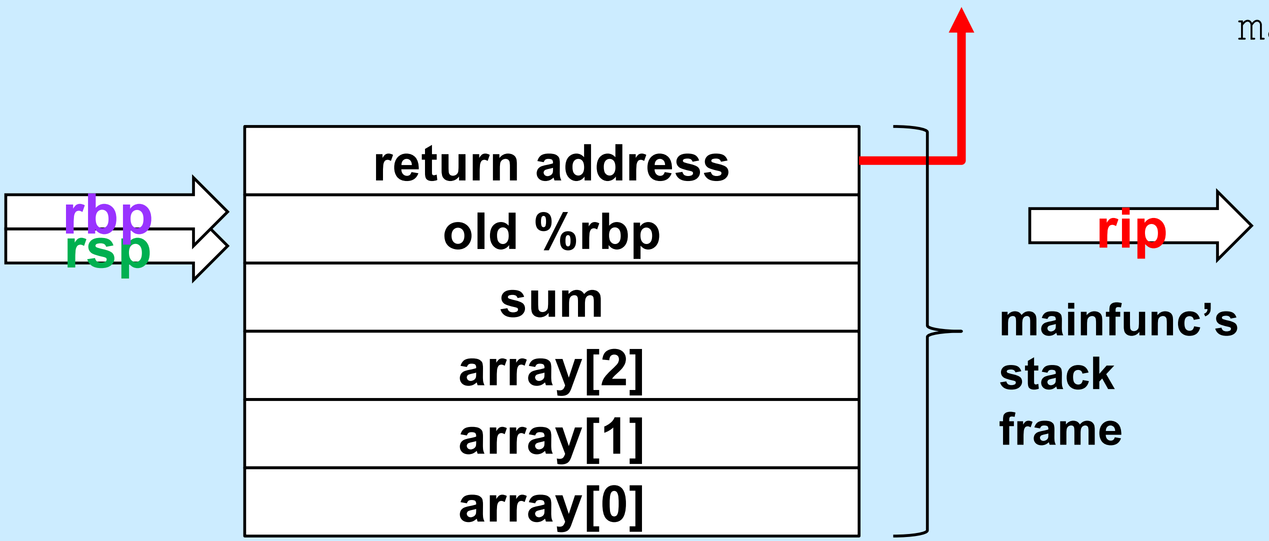

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Setup Frame



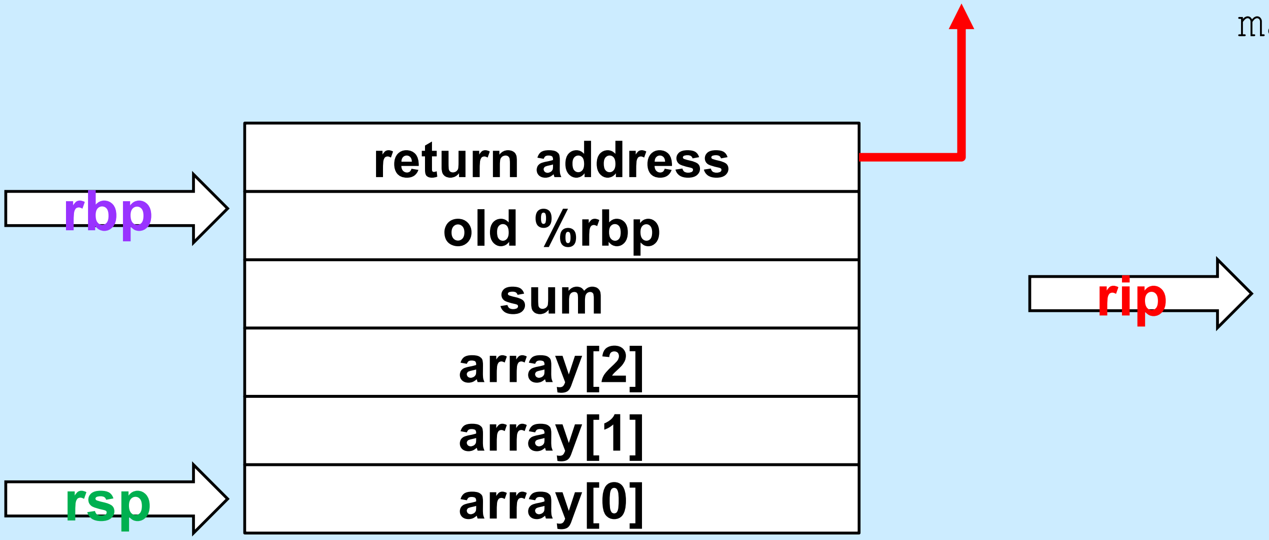

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Allocate Local Variables



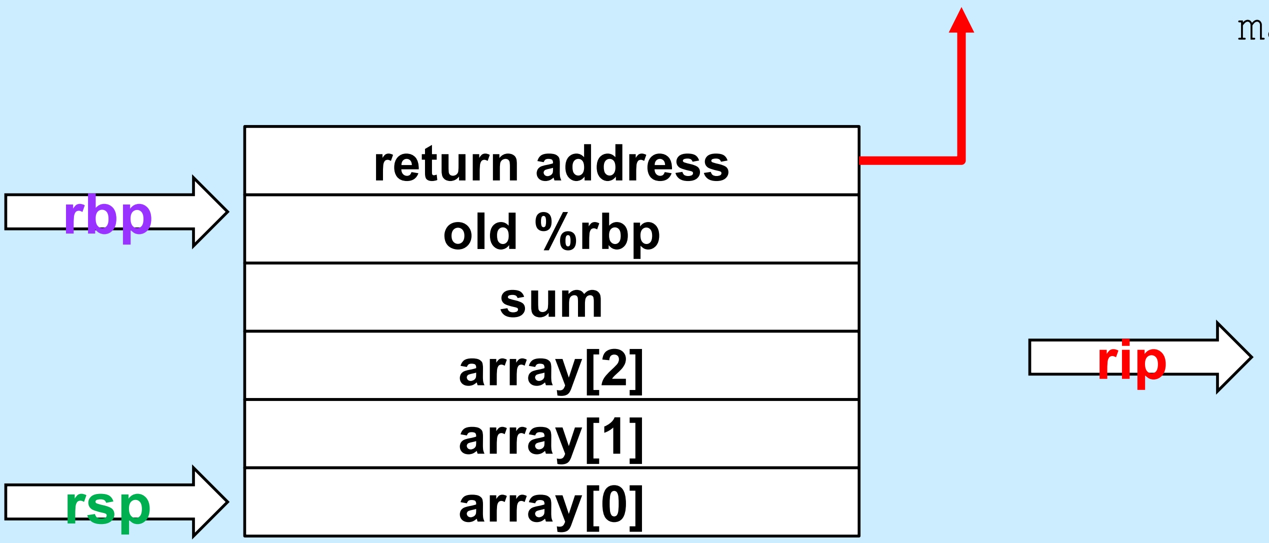

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Initialize Local Array



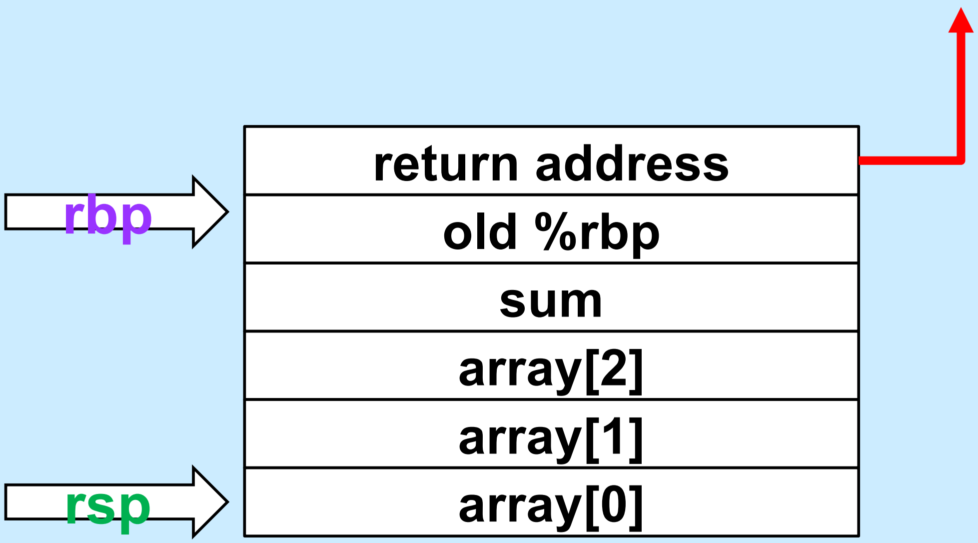

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Initialize Local Array



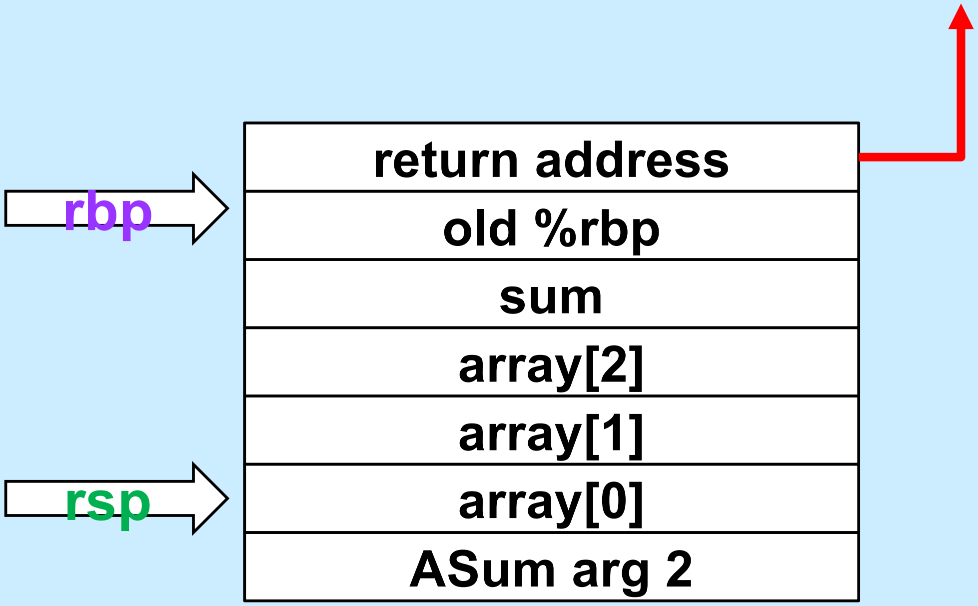

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

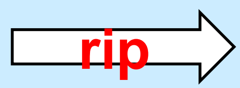
Initialize Local Array




```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

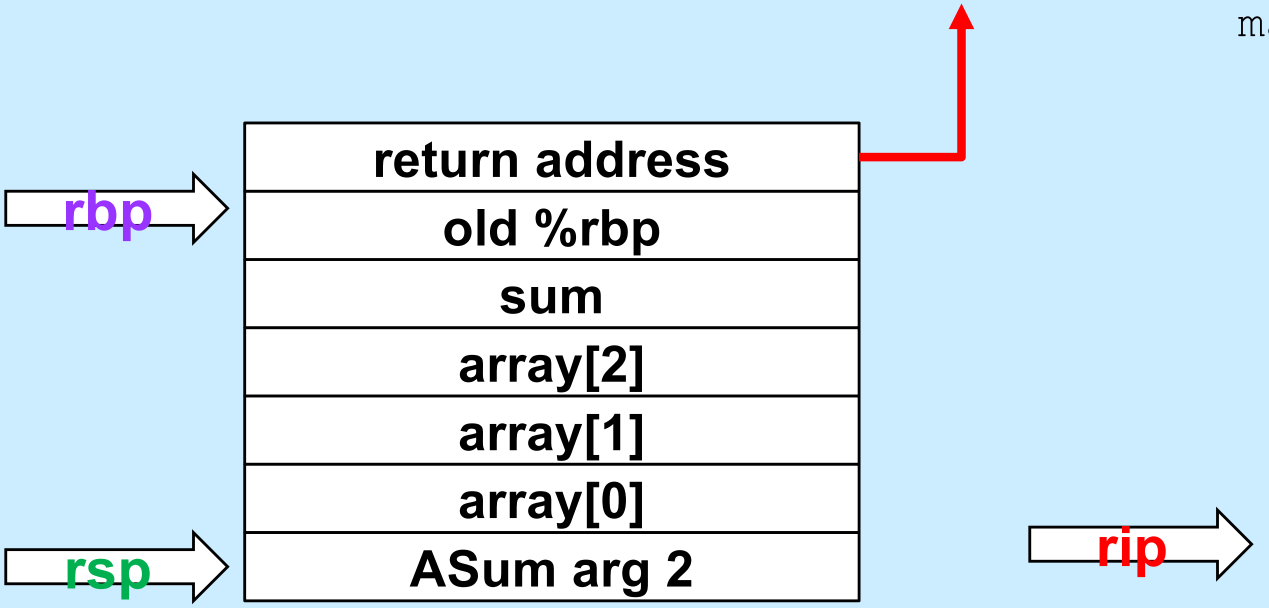
Push Second Argument





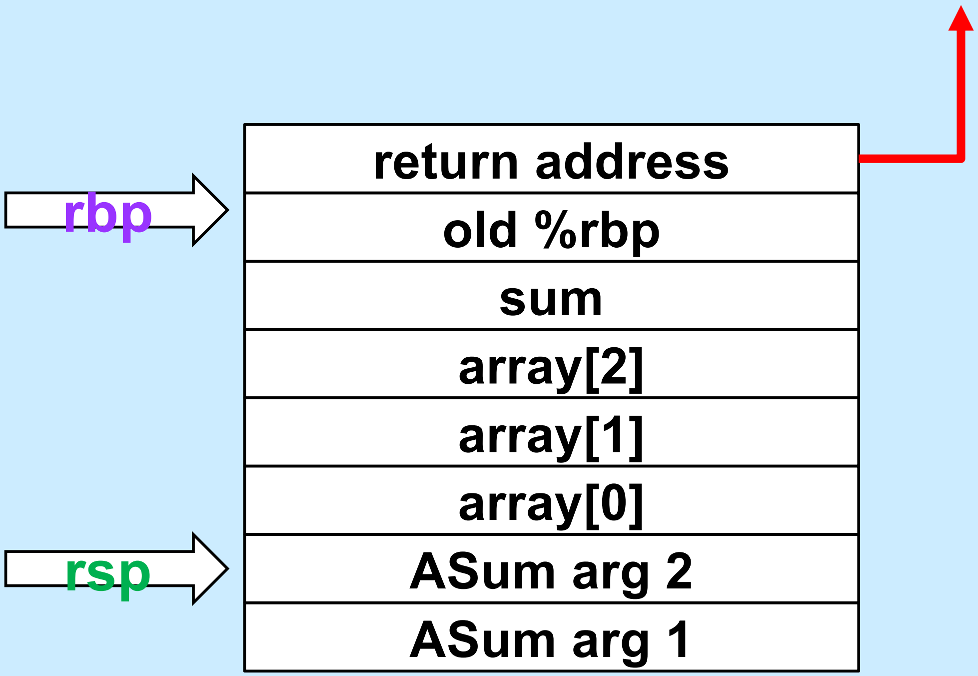

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

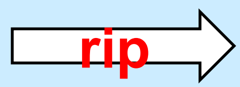
Get Array Address




```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

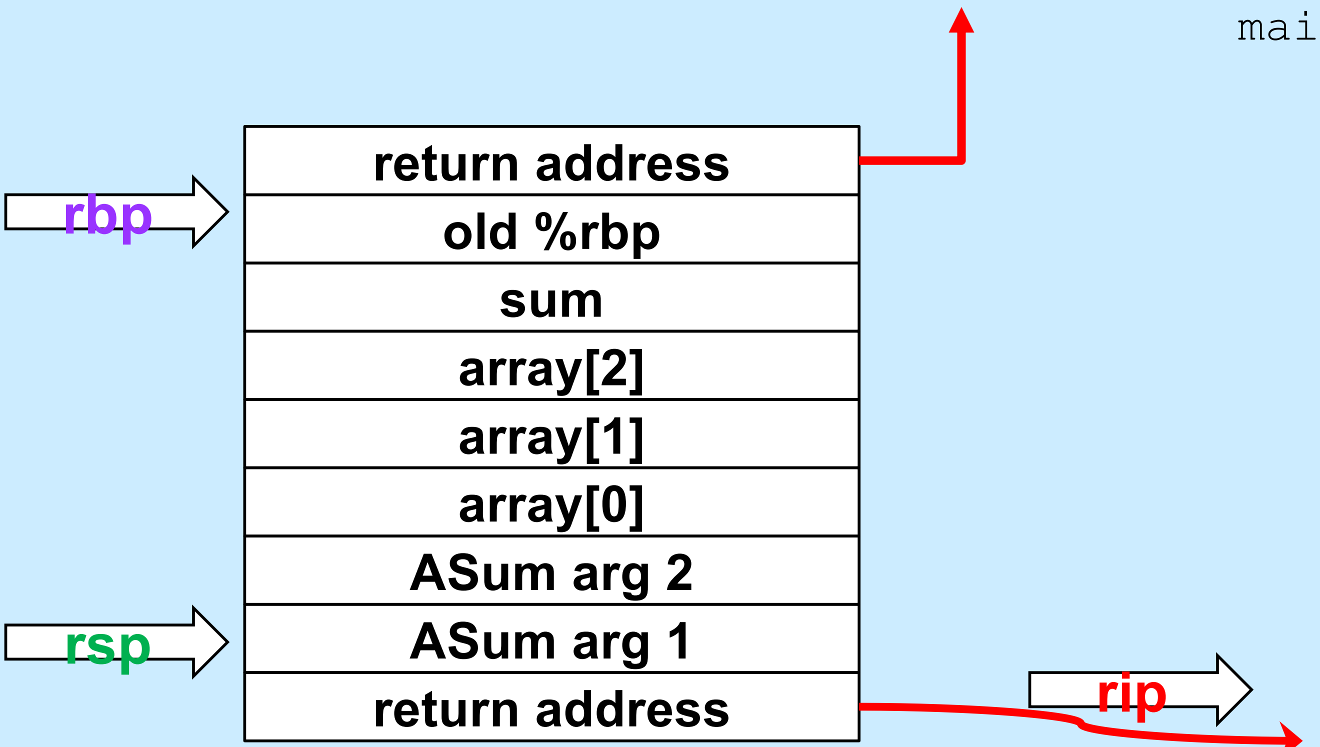
Push First Argument





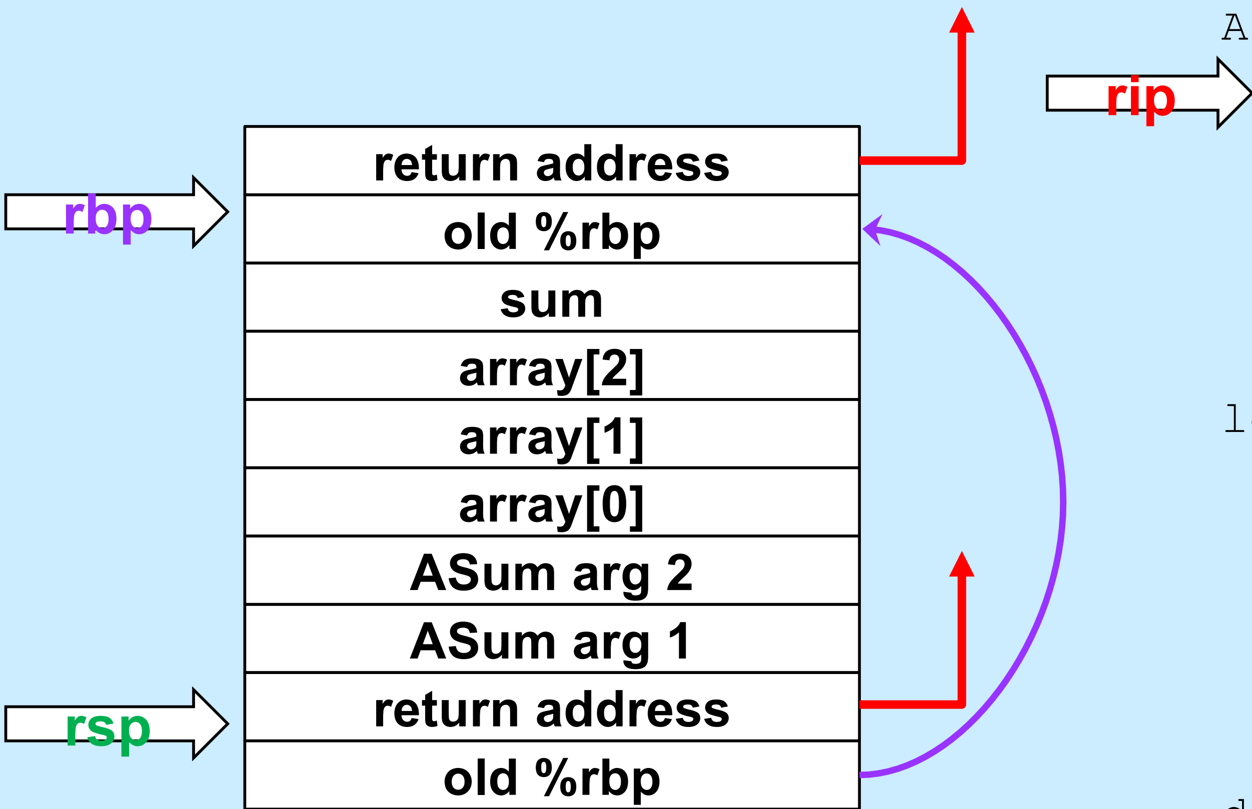

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

Call ASum



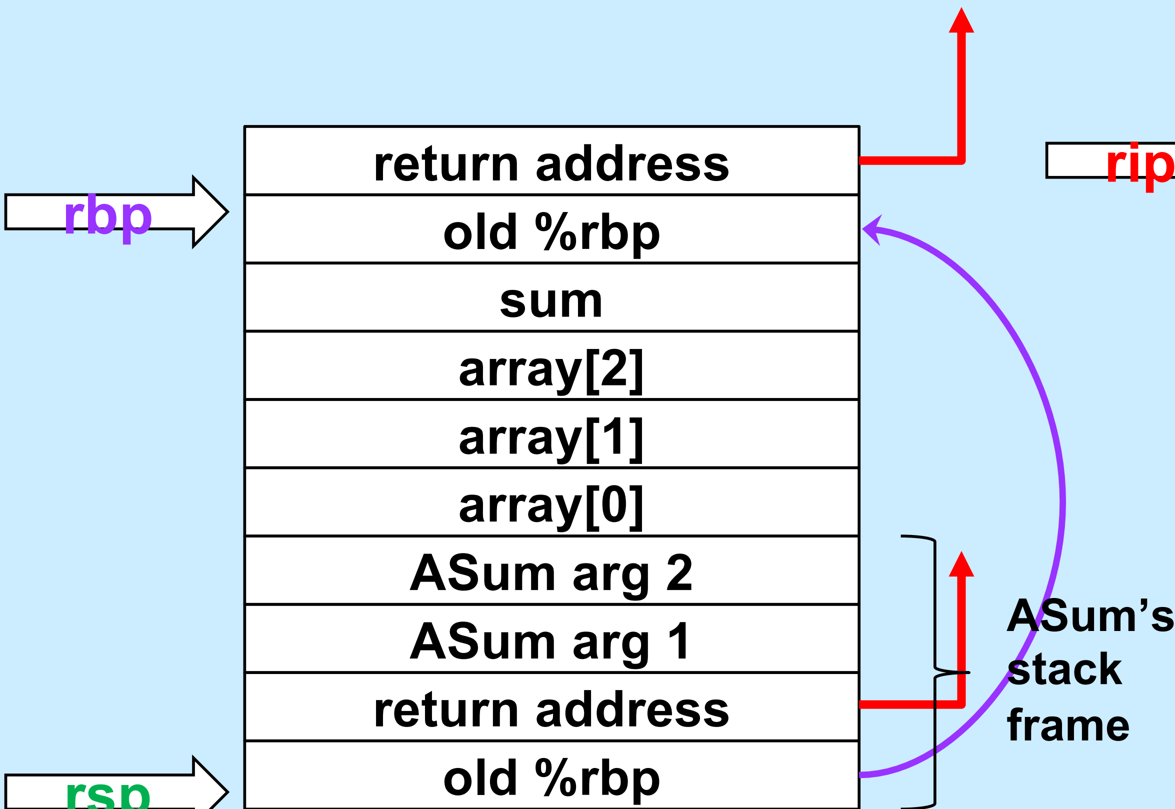

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```


Enter ASum



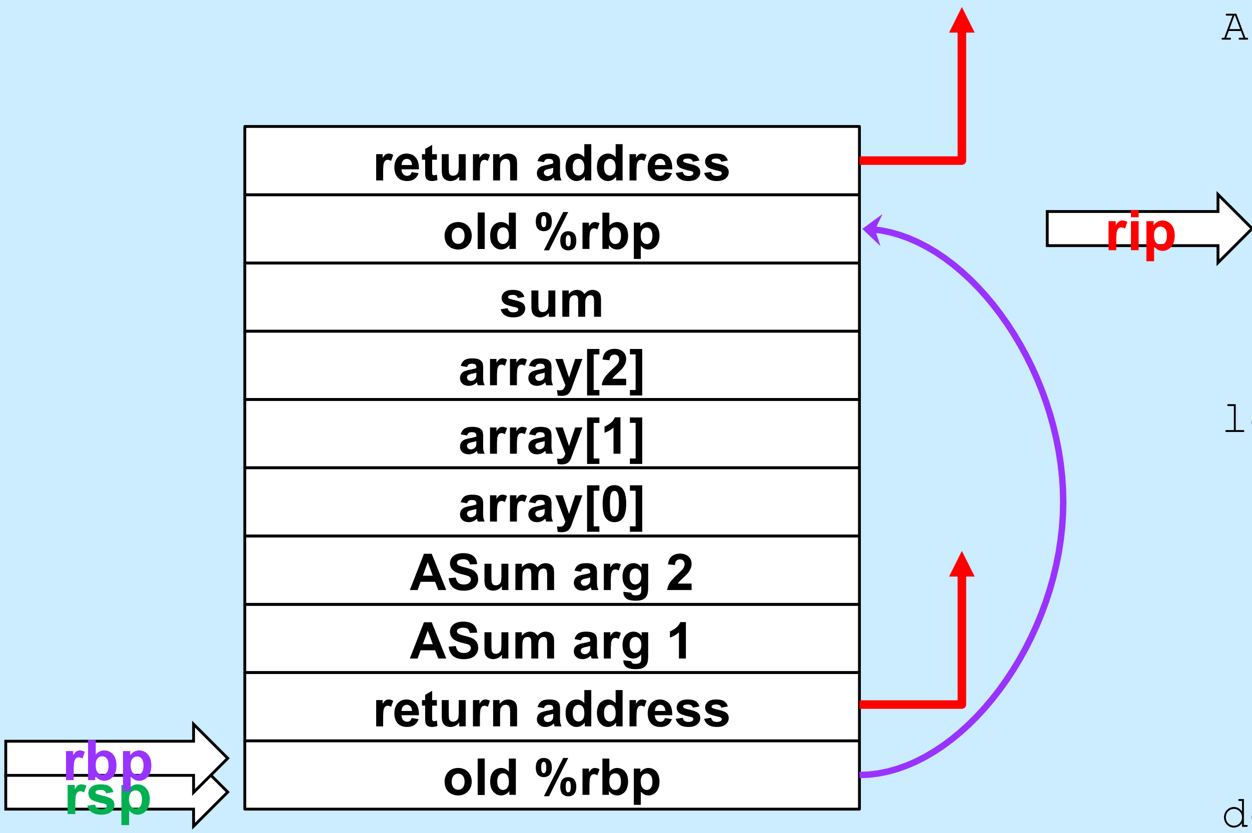

```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```

Setup Frame




```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```

Execute the Function




```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```

Quiz 1

What's at 24(%rbp)?

a) a local variable

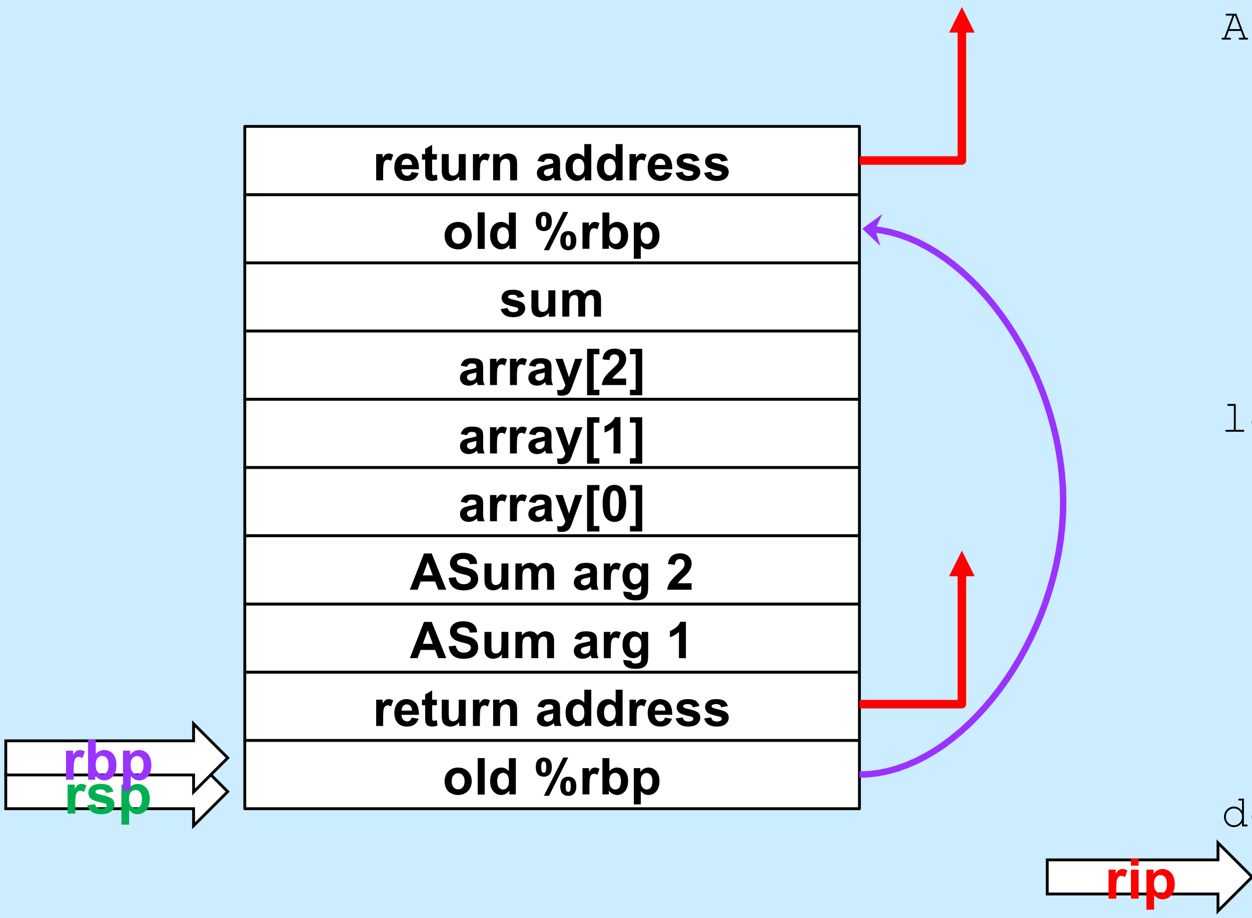
b) the first argument to ASum

c) the second argument to ASum

d) something else

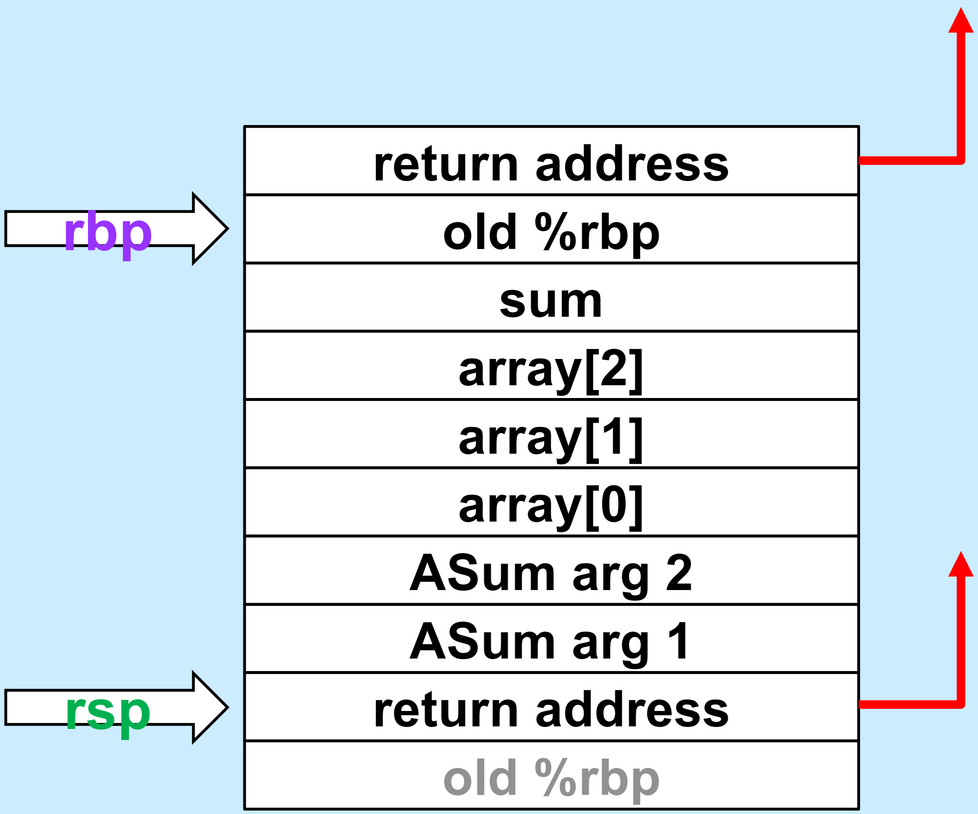
```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```


Prepare to Return

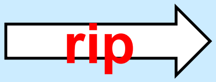



```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```

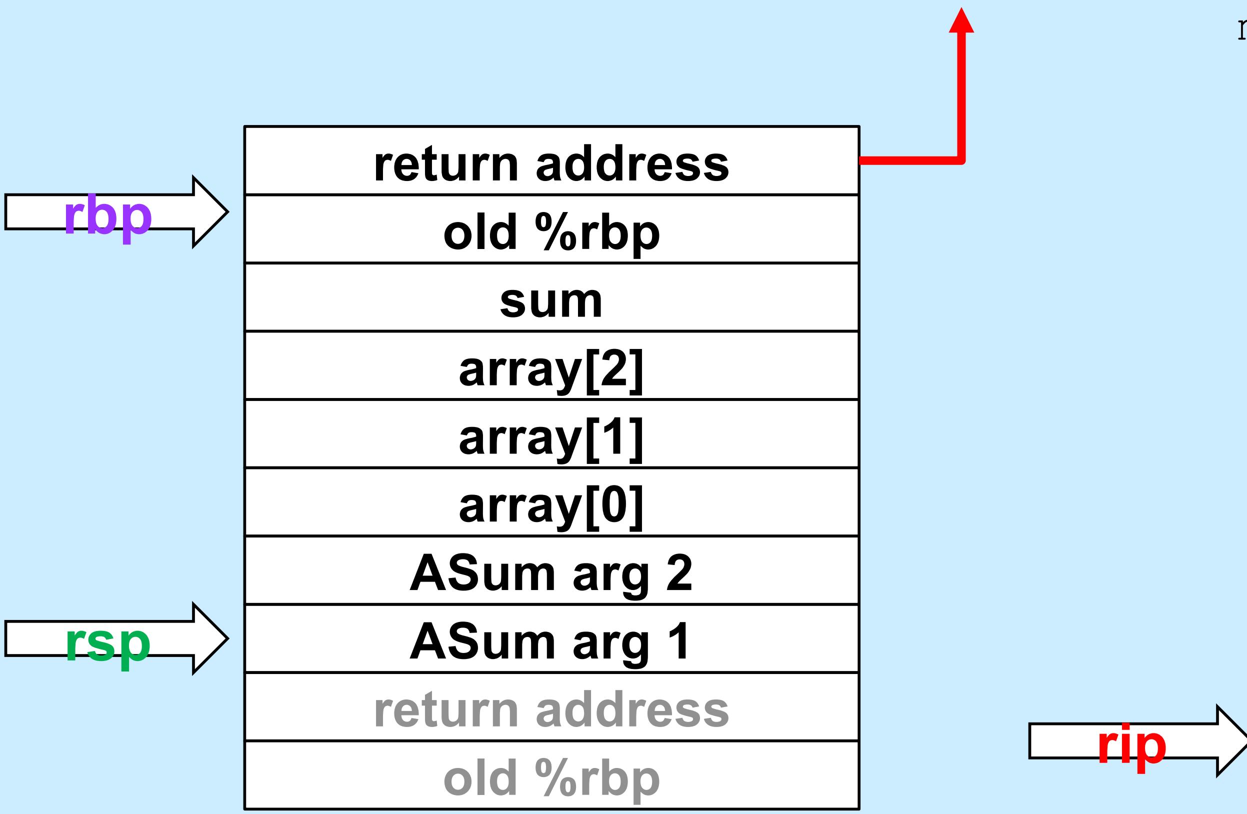

Return




```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```

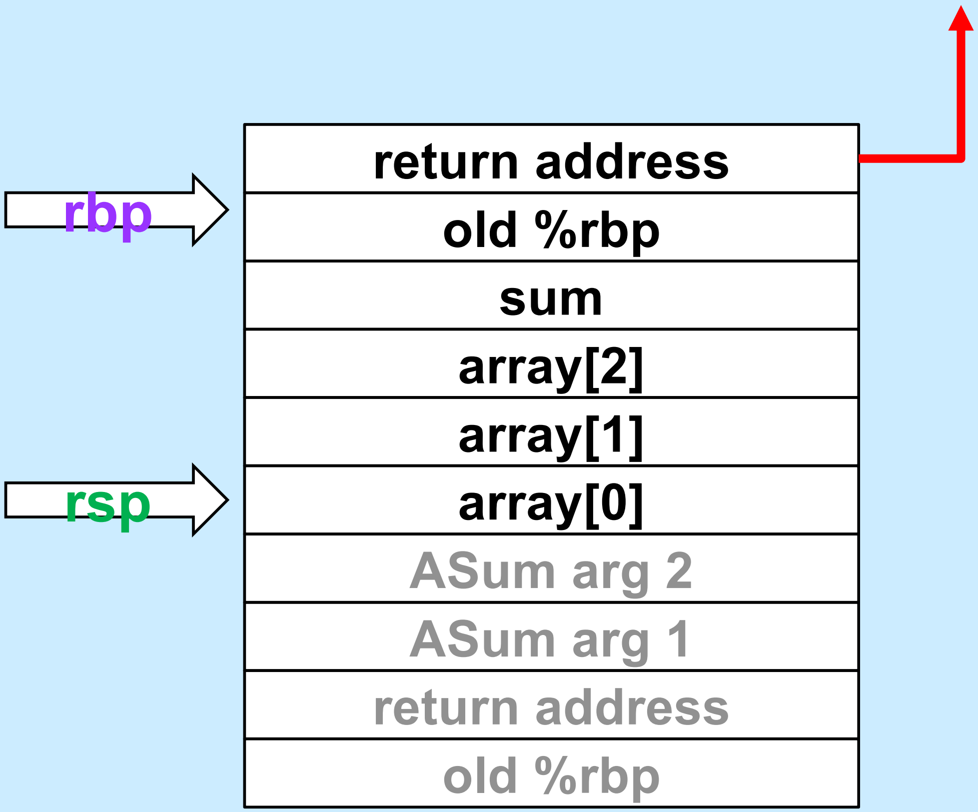


Pop Arguments

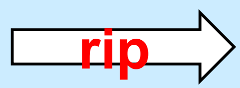



```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

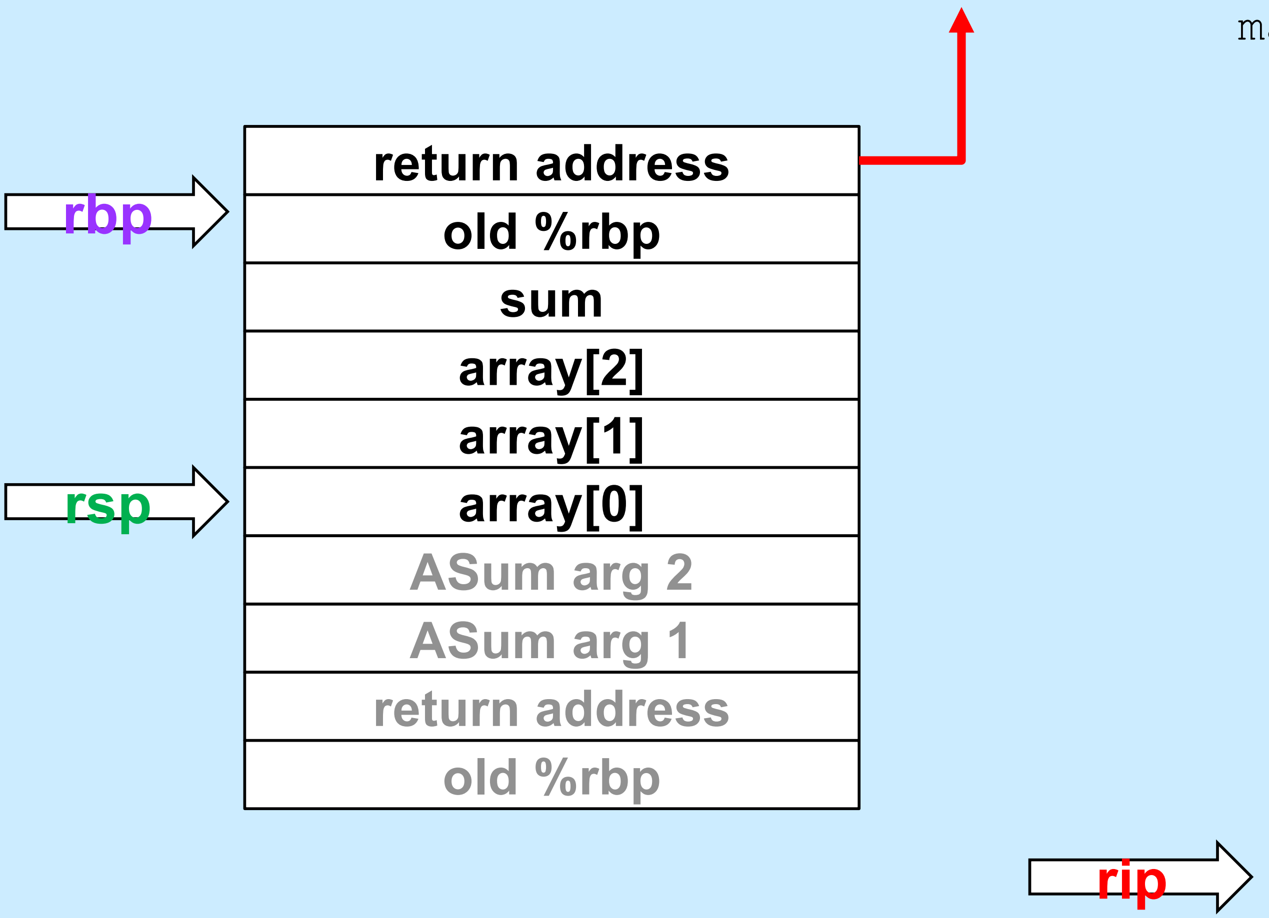
Save Return Value




```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

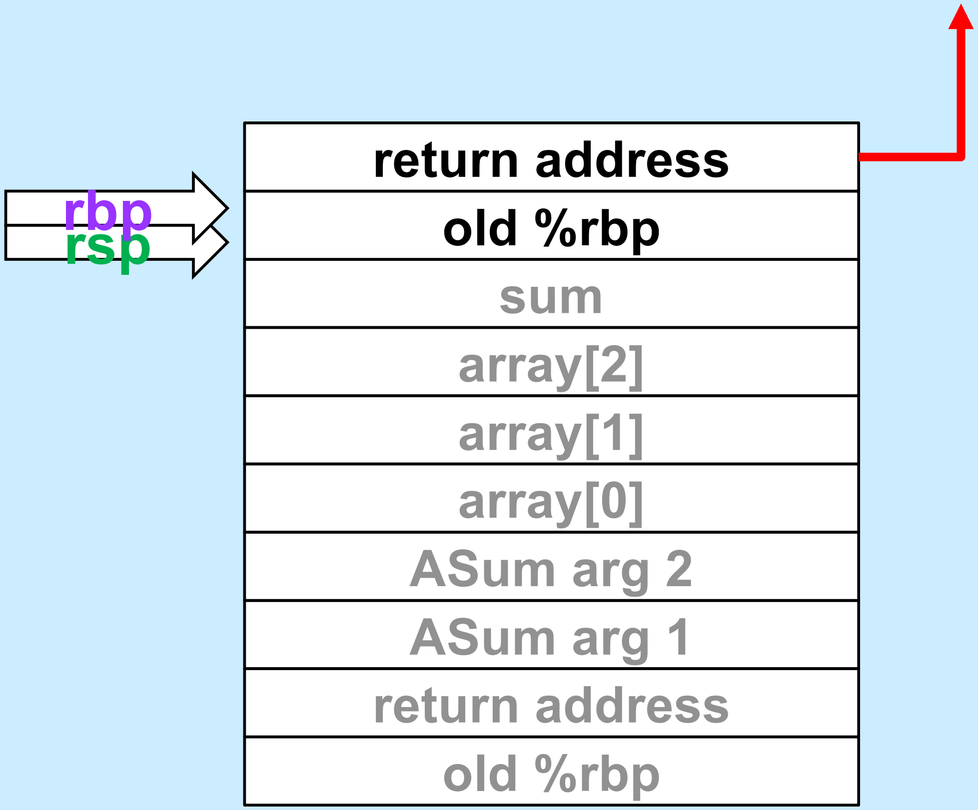


Pop Local Variables




```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

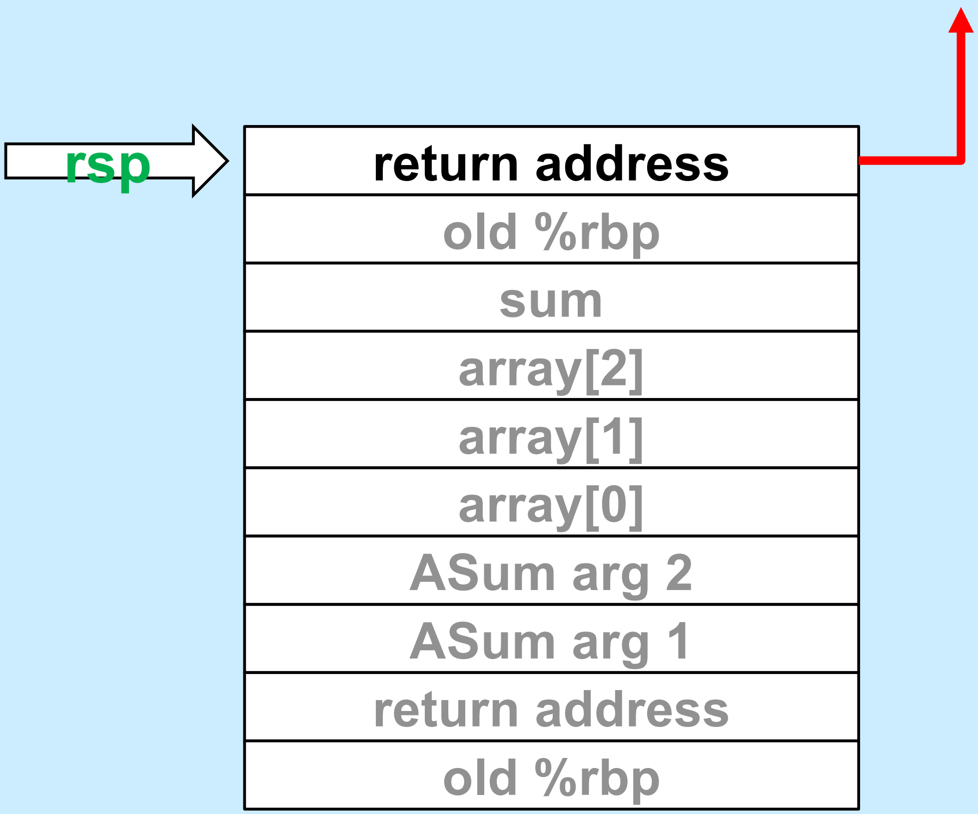
Prepare to Return



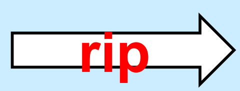

```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```

rip

Return




```
mainfunc:
    pushq %rbp
    movq %rsp, %rbp
    subq $32, %rsp
    movq $2, -32(%rbp)
    movq $117, -24(%rbp)
    movq $-6, -16(%rbp)
    pushq $3
    leaq -32(%rbp), %rax
    pushq %rax
    call ASum
    addq $16, %rsp
    movq %rax, -8(%rbp)
    addq $32, %rsp
    popq %rbp
    ret
```



Using Registers

- **ASum modifies registers:**
  - **%rsp**
  - **%rbp**
  - **%rcx**
  - **%rax**
  - **%rdx**
- **Suppose its caller uses these registers**

```
...
movq $33, %rcx
movq $167, %rdx
pushq $6
pushq array
call ASum
  # assumes unmodified %rcx and %rdx
addq $16, %rsp
addq %rax,%rcx    # %rcx was modified!
addq %rdx, %rcx   # %rdx was modified!
```

```
ASum:
   pushq %rbp
   movq %rsp, %rbp
   movq $0, %rcx
   movq $0, %rax
   movq 16(%rbp), %rdx
loop:
   cmpq 24(%rbp), %rcx
   jge done
   addq (%rdx,%rcx,8), %rax
   incq  %rcx
   ja loop
done:
   popq %rbp
   ret
```

Register Values Across Function Calls

- **ASum modifies registers:**
  - **%rsp**
  - **%rbp**
  - **%rcx**
  - **%rax**
  - **%rdx**
- **May the caller of ASum depend on its registers being the same on return?**
  - **ASum saves and restores %rbp and makes no net changes to %rsp**
    - » **their values are unmodified on return to its caller**
  - **%rax, %rcx, and %rdx are not saved and restored**
    - » **their values might be different on return**

```
ASum:
    pushq %rbp
    movq %rsp, %rbp
    movq $0, %rcx
    movq $0, %rax
    movq 16(%rbp), %rdx
loop:
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq  %rcx
    ja loop
done:
    popq %rbp
    ret
```


Register-Saving Conventions

- **Caller-save registers**
  - if the caller wants their values to be the same on return from function calls, it must save and restore them

```
pushq %rcx
call func
popq %rcx
```

- **Callee-save registers**
  - if the callee wants to use these registers, it must first save them, then restore their values before returning

```
func:
    pushq %rbx
    movq $6, %rbx
    ...
    popq %rbx
```


x86-64 General-Purpose Registers: Usage Conventions

`%rax`	Return value	`%r8`	Caller saved
`%rbx`	Callee saved	`%r9`	Caller saved
`%rcx`	Caller saved	`%r10`	Caller saved
`%rdx`	Caller saved	`%r11`	Caller Saved
`%rsi`	Caller saved	`%r12`	Callee saved
`%rdi`	Caller saved	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Base pointer	`%r15`	Callee saved

Recursive Function

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

- Registers
  - %rax, %rdx used without first saving
  - %rbx used, but saved at beginning & restored at end

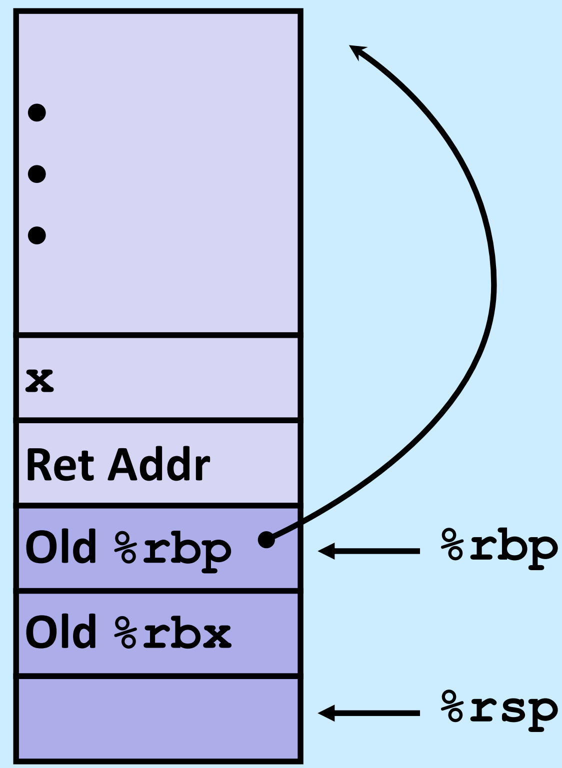
```
pcount_r:
    pushq %rbp
    movq  %rsp, %rbp
    pushq %rbx
    movq  16(%rbp), %rbx
    movq  $0, %rax
    testq %rbx, %rbx
    je    .L3
    movq  %rbx, %rax
    shrq  $1, %rax
    pushq %rax
    call  pcount_r
    addq  $8, %rsp
    movq  %rbx, %rdx
    andq  $1, %rdx
    leaq  (%rdx,%rax), %rax
.L3:
    popq  %rbx
    popq  %rbp
    ret
```

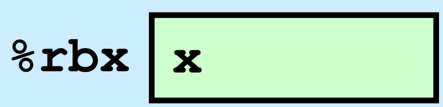
Recursive Call #1

```
pcount_r:
    pushq %rbp
    movq  %rsp, %rbp
    pushq %rbx
    movq  16(%rbp), %rbx
     • • •
```

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

- **Actions**
  - **save old value of `%rbx` on stack**
  - **store x in `%rbx`**





Recursive Call #2

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

```
    . . .
   movq  $0, %rax
   testq %rbx, %rbx
   je    .L3
    . . .
.L3:
    . . .
   ret
```

- **Actions**
  - if x == 0, return
    - » with `%rax` set to 0

`%rbx`	`x`

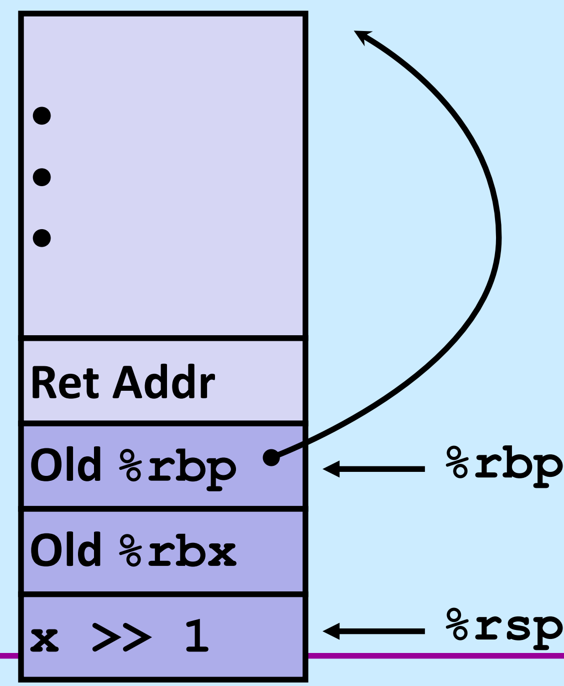
Recursive Call #3

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

```
  • • •
movq  %rbx, %rax
shrq  $1, %rax
pushq %rax
call  pcount_r
  • • •
```

- **Actions**
  - push x >> 1 on stack as arg
  - make recursive call
- **Effect**
  - `%rax` set to function result
  - `%rbx` still has value of x

`%rbx`	`x`



Recursive Call #4

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

```
  • • •
movq    %rbx, %rdx
addq    $8, %rsp
andq    $1, %rdx
leaq    (%rdx,%rax), %rax
  • • •
```

- Assume
  - `%rax` holds value from recursive call
  - `%rbx` holds x
- Actions
  - pop argument from stack
  - compute (x & 1) + computed value
- Effect
  - `%rax` set to function result

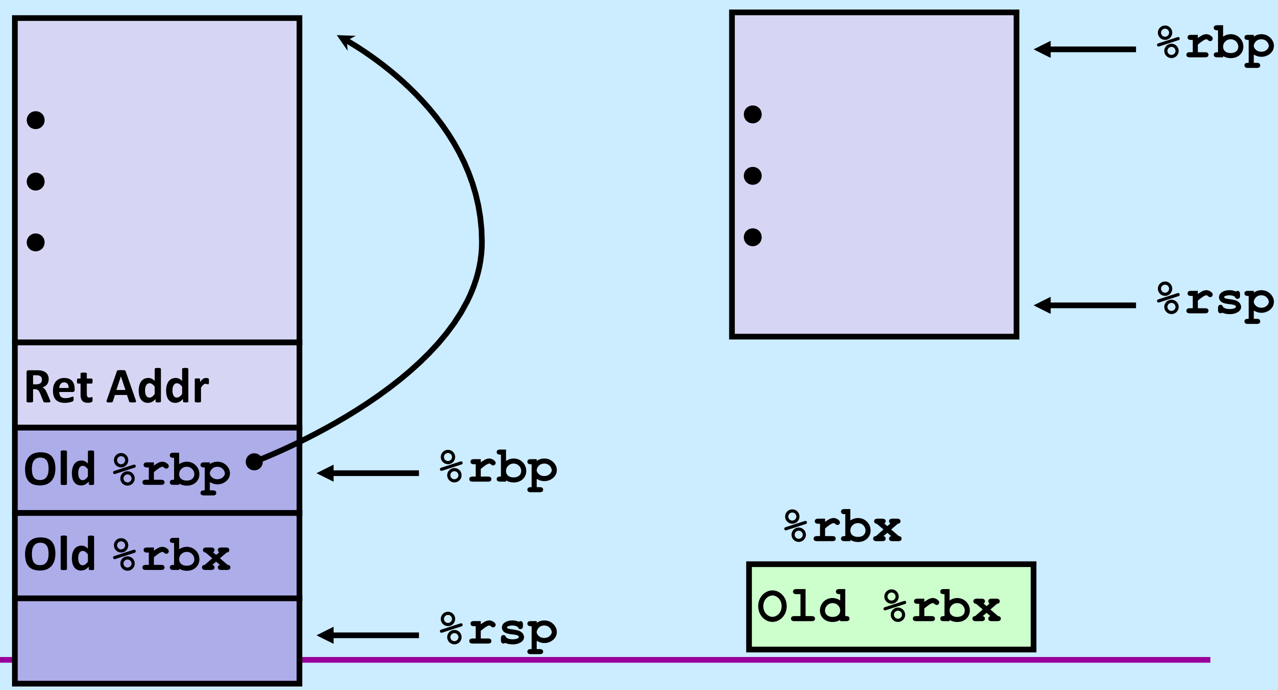
%rbx	x

Recursive Call #5

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

```
        • • •
L3:
    popq        %rbx
    popq        %rbp
    ret
```

- **Actions**
  - restore values of %rbx and %rbp
  - return (and pop return address)



Observations About Recursion

- Handled without special consideration
  - stack frames mean that each function call has private storage
    - » saved registers & local variables
    - » saved return pointer
  - register-saving conventions prevent one function call from corrupting another's data
  - 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

Passing Arguments in Registers

- **Observations**
  - accessing registers is much faster than accessing primary memory
    - if arguments were in registers rather than on the stack, speed would increase
  - most functions have just a few arguments
- **Actions**
  - change calling conventions so that the first six arguments are passed in registers
    - in caller-save registers
  - any additional arguments are pushed on the stack

Why Bother with a Base Pointer?

- It (%rbp) points to the beginning of the stack frame
  – making it easy for people to figure out where things are in the frame
  – but people don’t execute the code ...
- The stack pointer always points somewhere within the stack frame
  – it moves about, but the compiler knows where it is pointing
    » a local variable might be at 8(%rsp) for one instruction, but at 16(%rsp) for a subsequent one
    » tough for people, but easy for the compiler
- Thus the base pointer is superfluous
  – it can be used as a general-purpose register

x86-64 General-Purpose Registers: Updated Usage Conventions

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

Recursive Function (Improved)

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

```
pcount_r:
   movq    $0, %rax
   testq   %rdi, %rdi
   je      .L8
   pushq   %rbx
   movq    %rdi, %rbx
   shrq    $1, %rdi
   call    pcount_r
   andq    $1, %rbx
   addq    %rbx, %rax
   popq    %rbx
.L8:
   ret
```

- **Registers**
  - the single argument (x) is passed in %rdi
  - %rbx is a callee-saved register and thus is saved and restored
  - %rax is caller-saved
  - %rbp isn't used

Summary

- What's pushed on the stack
  - return address
  - saved registers
    - » caller-saved by the caller
    - » callee-saved by the callee
  - local variables
  - function parameters
    - » those too large to be in registers (structs)
    - » those beyond the six that we have registers for
  - large return values (structs)
    - » caller allocates space on stack
    - » callee copies return value to that space

Quiz 2

Suppose function A is compiled using the convention that %rbp is used as the base pointer, pointing to the beginning of the stack frame. Function B is compiled using the convention that there's no need for a base pointer. Will there be any problems if A calls B or if B calls A?

a) Neither case will work

b) A calling B works, but B calling A doesn't

c) B calling A works, but A calling B doesn't

d) Both work

Tail Recursion

```
int factorial(int x) {
  if (x == 1)
    return x;
  else
    return
      x*factorial(x-1);
}
```

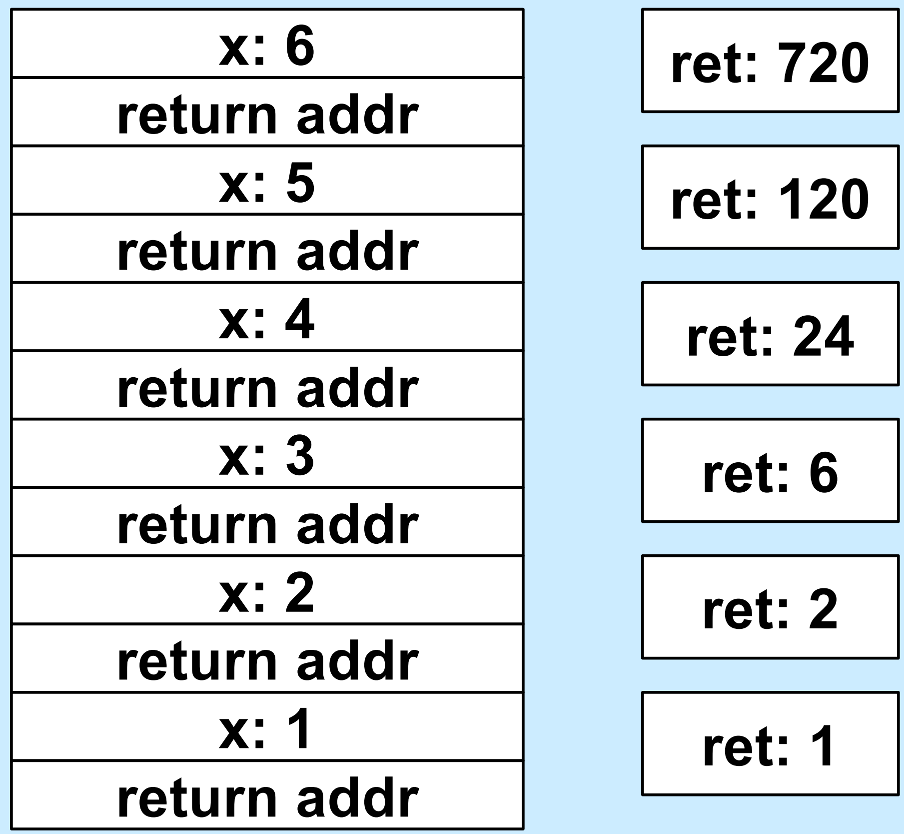
```
int factorial(int x) {
  return f2(x, 1);
}

int f2(int a1, int a2) {
  if (a1 == 1)
    return a2;
  else
    return
      f2(a1-1, a1*a2);
}
```

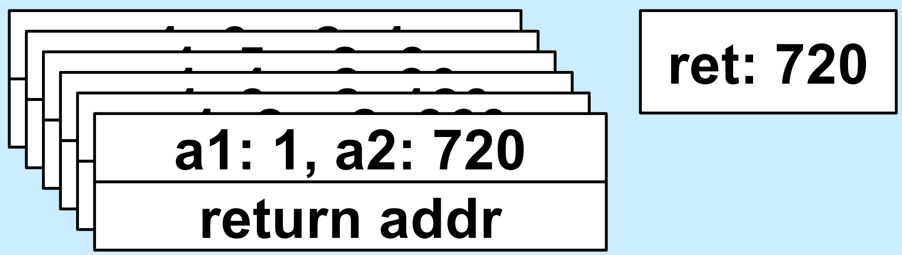

No Tail Recursion (1)

x: 6
return addr
x: 5
return addr
x: 4
return addr
x: 3
return addr
x: 2
return addr
x: 1
return addr

No Tail Recursion (2)



Tail Recursion



Code: gcc –O1

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        subq    $8, %rsp
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
        addq    $8, %rsp
.L5:
        ret
```

Not Using the Stack ...

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        call    f2          # recursive call!
.L5:
        ret
```

Not Recursive!

```
f2:
        movl    %esi, %eax
        cmpl    $1, %edi
        je      .L5
        movl    %edi, %esi
        imull   %eax, %esi
        subl    $1, %edi
        ja      f2          # goto!
.L5:
        ret
```


Code: gcc –O2

```
f2:
        cmpl    $1, %edi
        movl    %esi, %eax
        je      .L8
.L12:
        imull   %edi, %eax
        subl    $1, %edi
        cmpl    $1, %edi
        jne     .L12
.L8:
        ret
```

loop!