

# CS 33

## Machine Programming (5)

# Arguments and Local Variables (C Code)

```
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 (1)

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 (2)

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
```

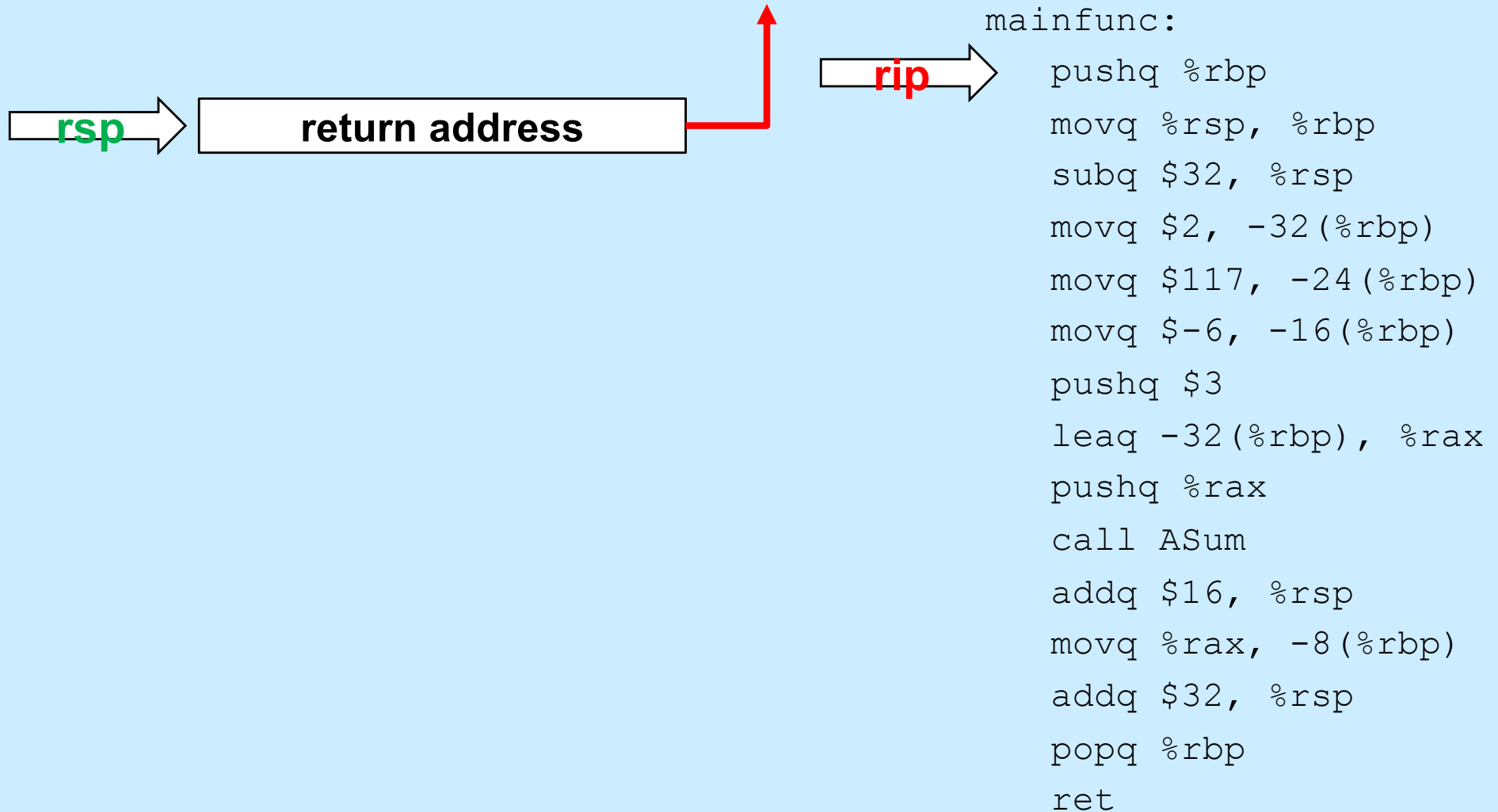
loop:

```
    cmpq 24(%rbp), %rcx       # i < size?
    jge done
    addq (%rdx,%rcx,8), %rax   # sum += a[i]
    incq %rcx                 # i++
    ja loop
```

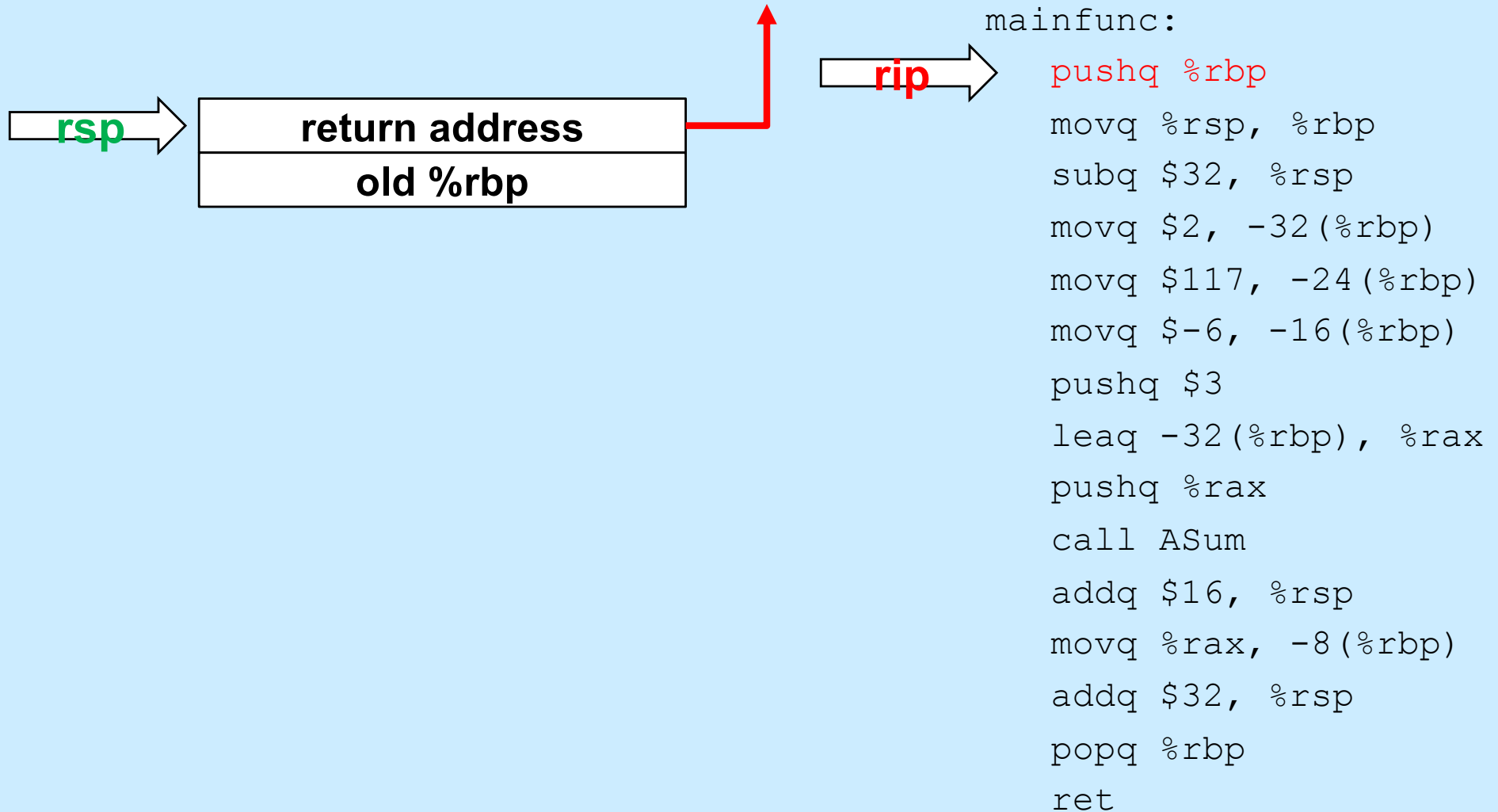
done:

```
    popq %rbp                # pop and restore %rbp
    ret
```

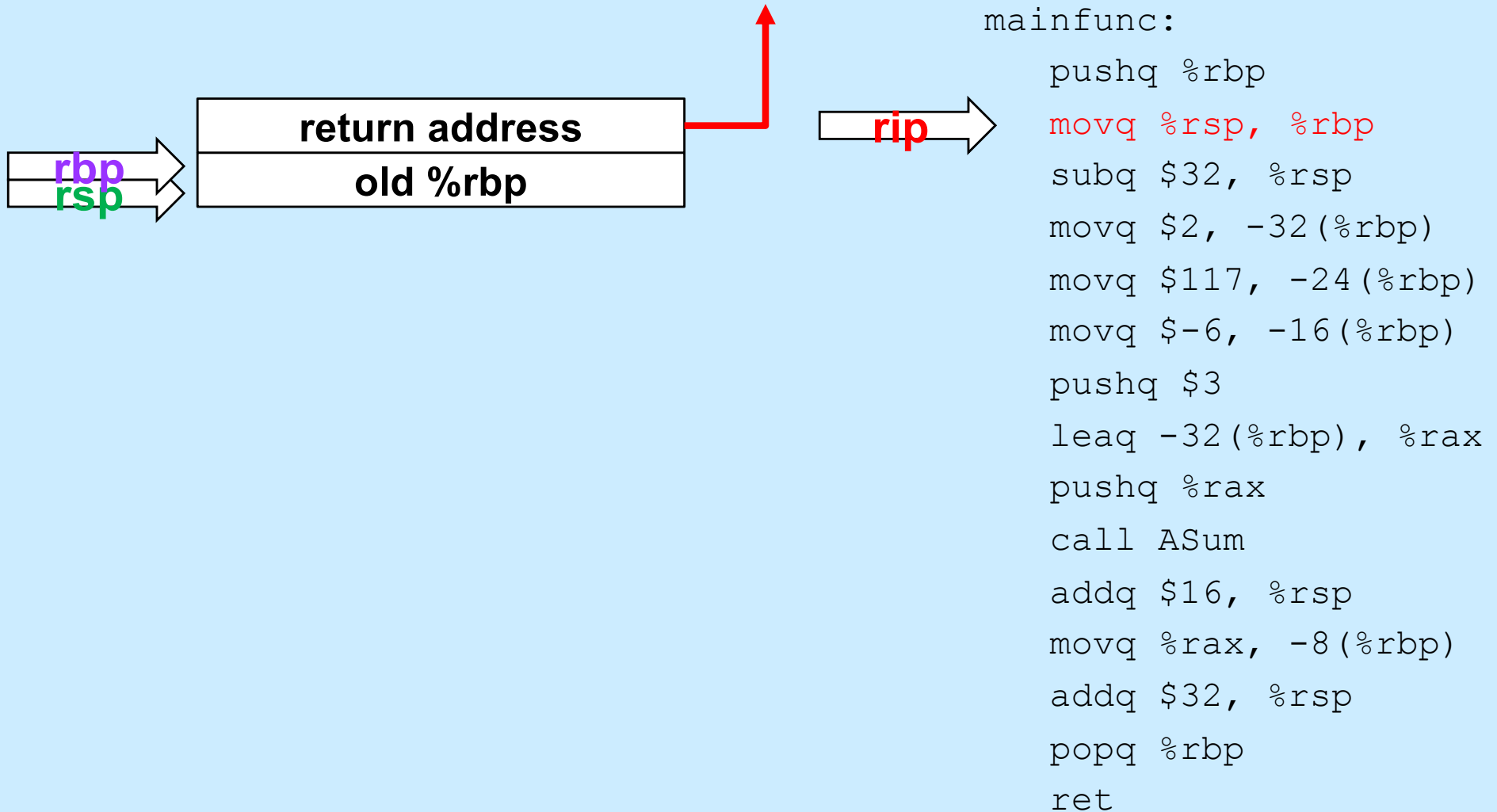
# Enter mainfunc



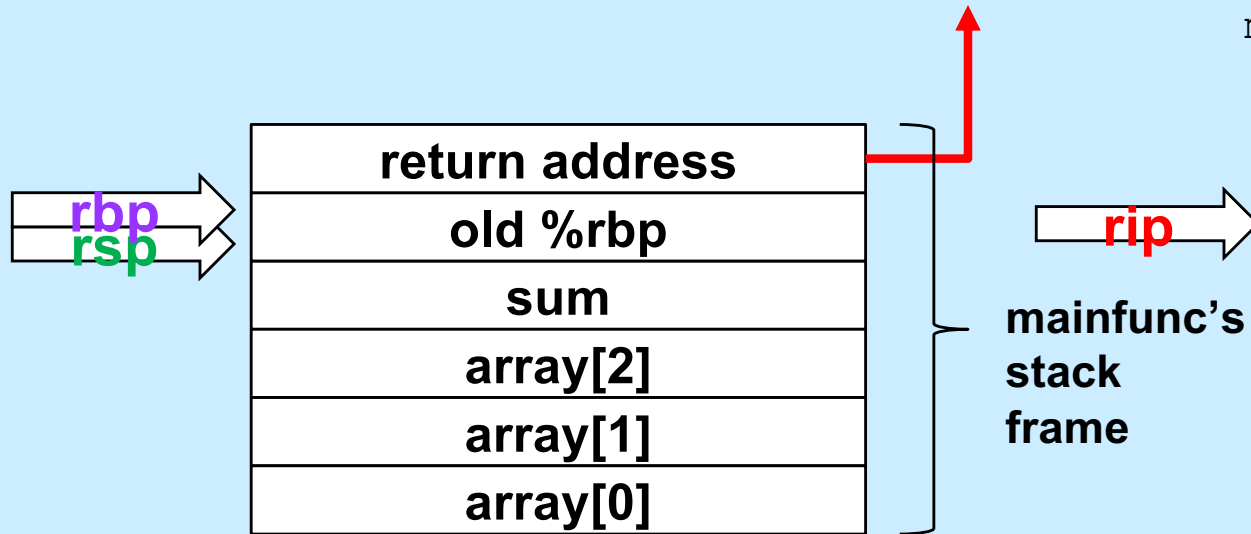
# Enter mainfunc



# Setup Frame



# Allocate Local Variables

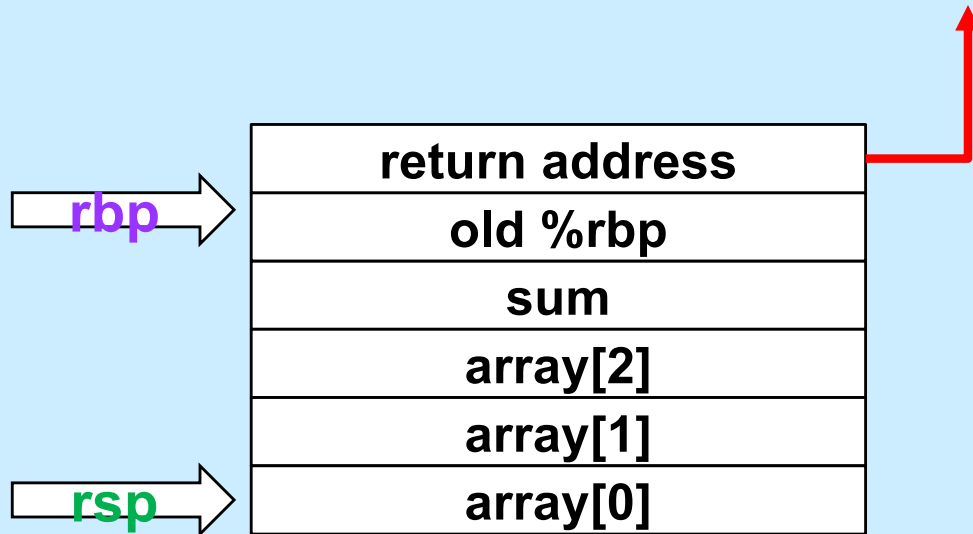


`mainfunc:`

```
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
```



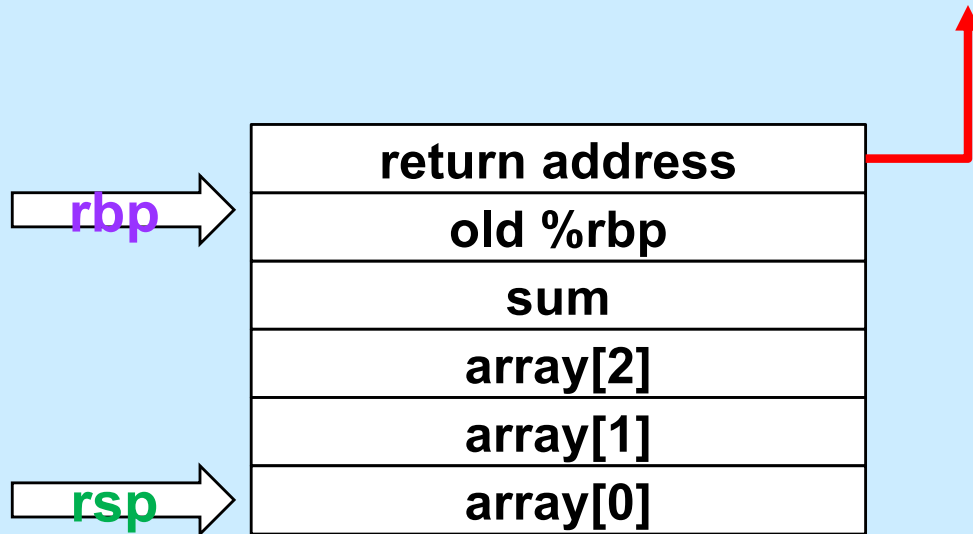
# Initialize Local Array



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
```

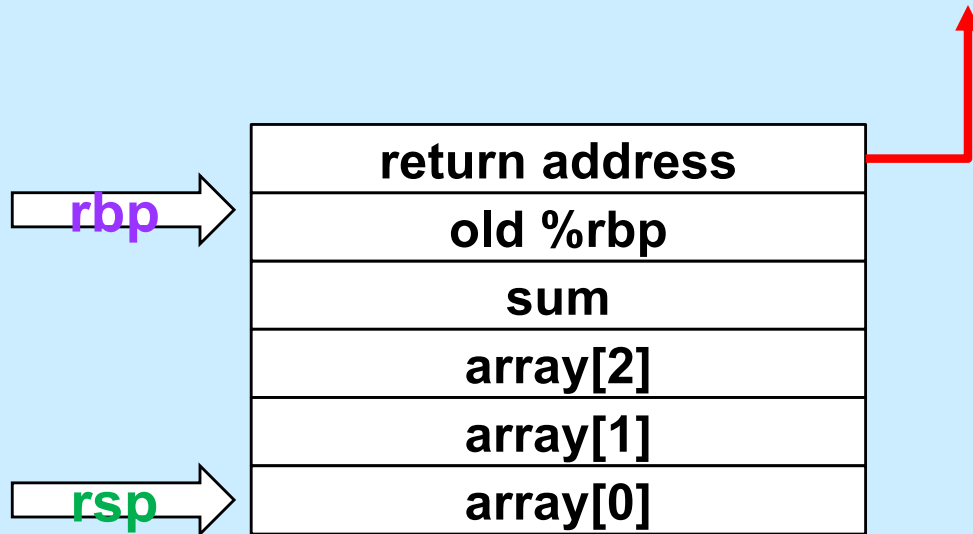
# Initialize Local Array



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
```

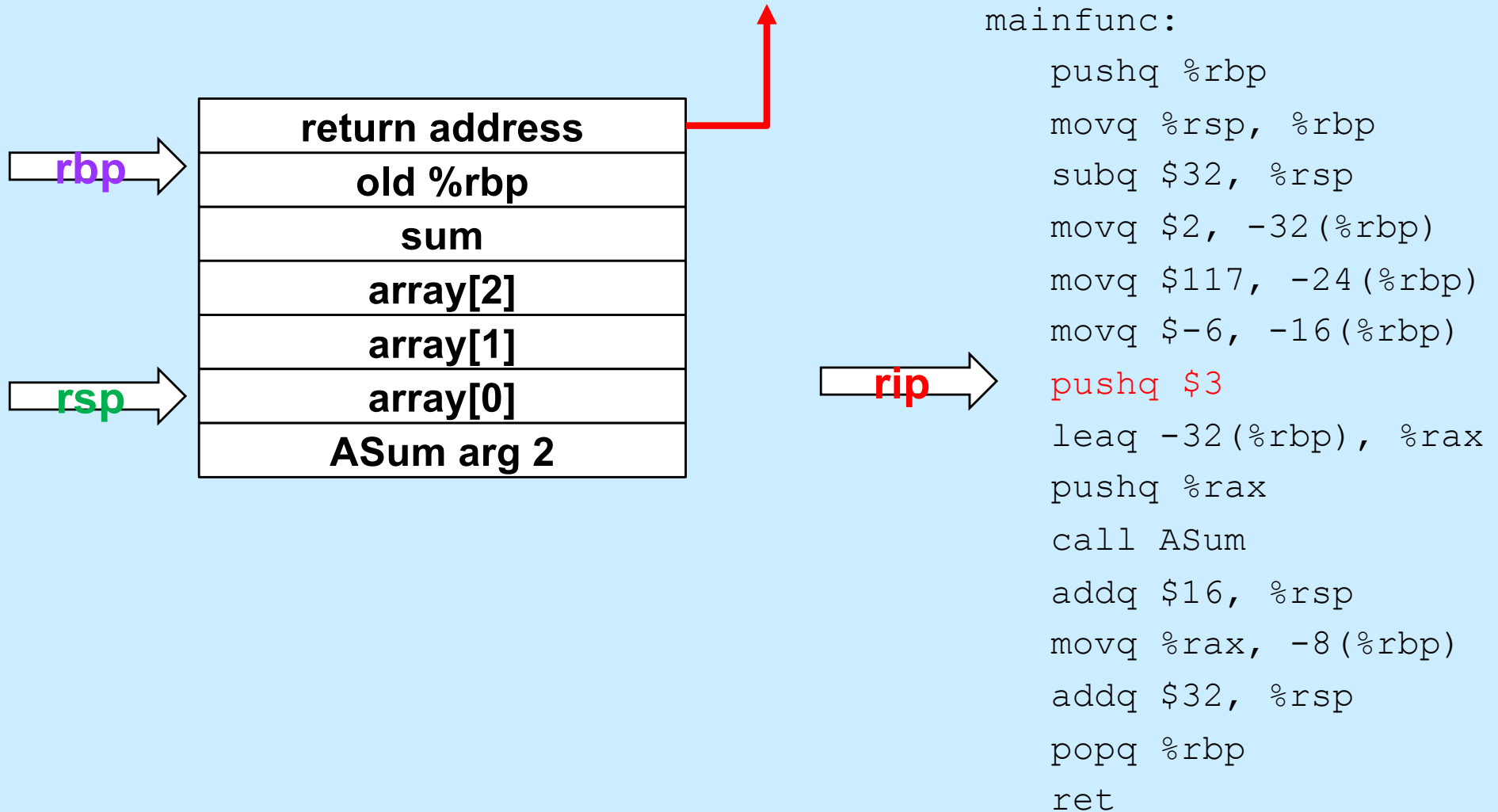
# Initialize Local Array



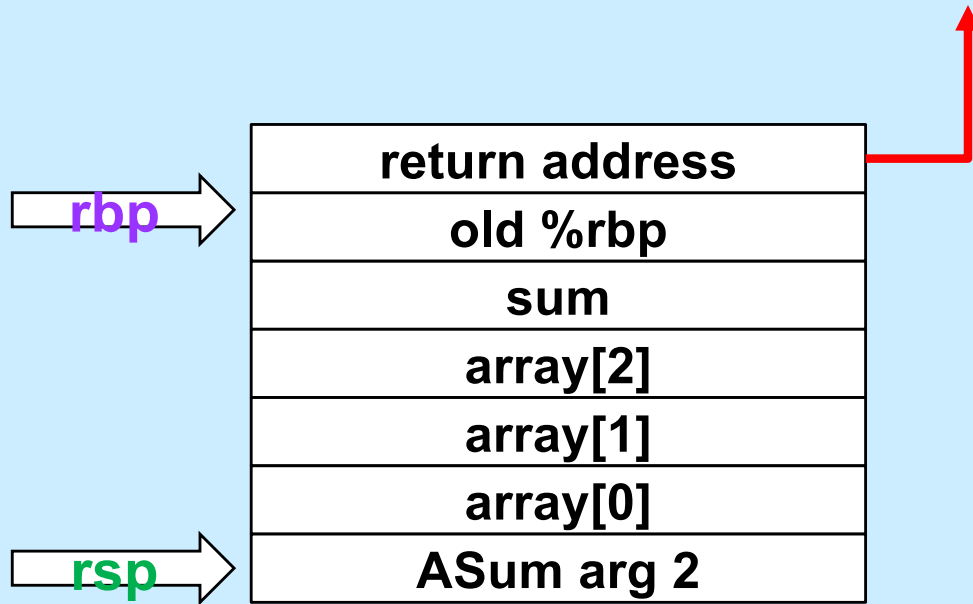
mainfunc:

```
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



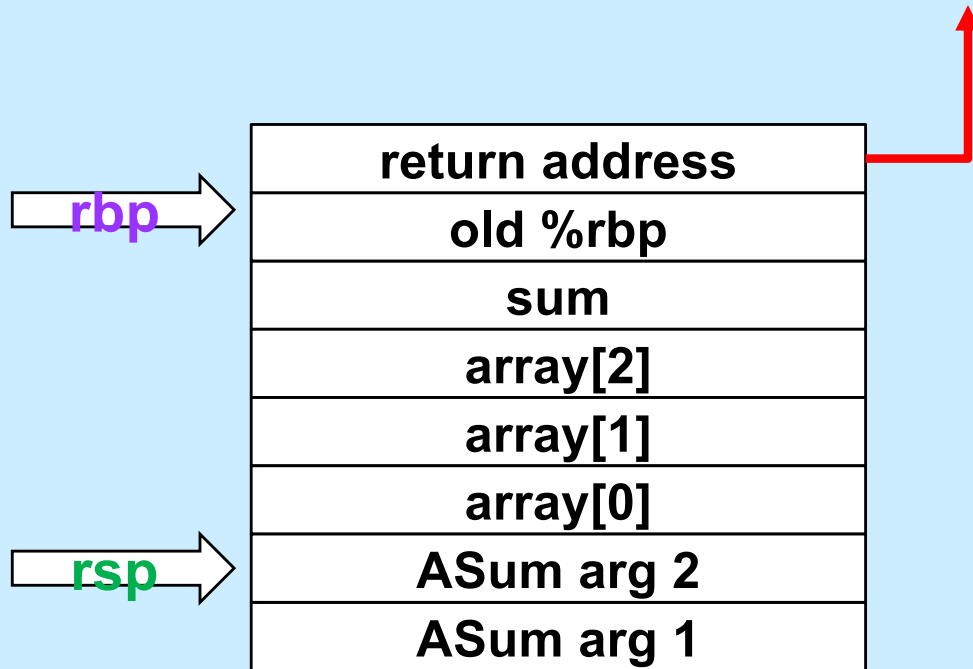
# Get Array Address



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
```

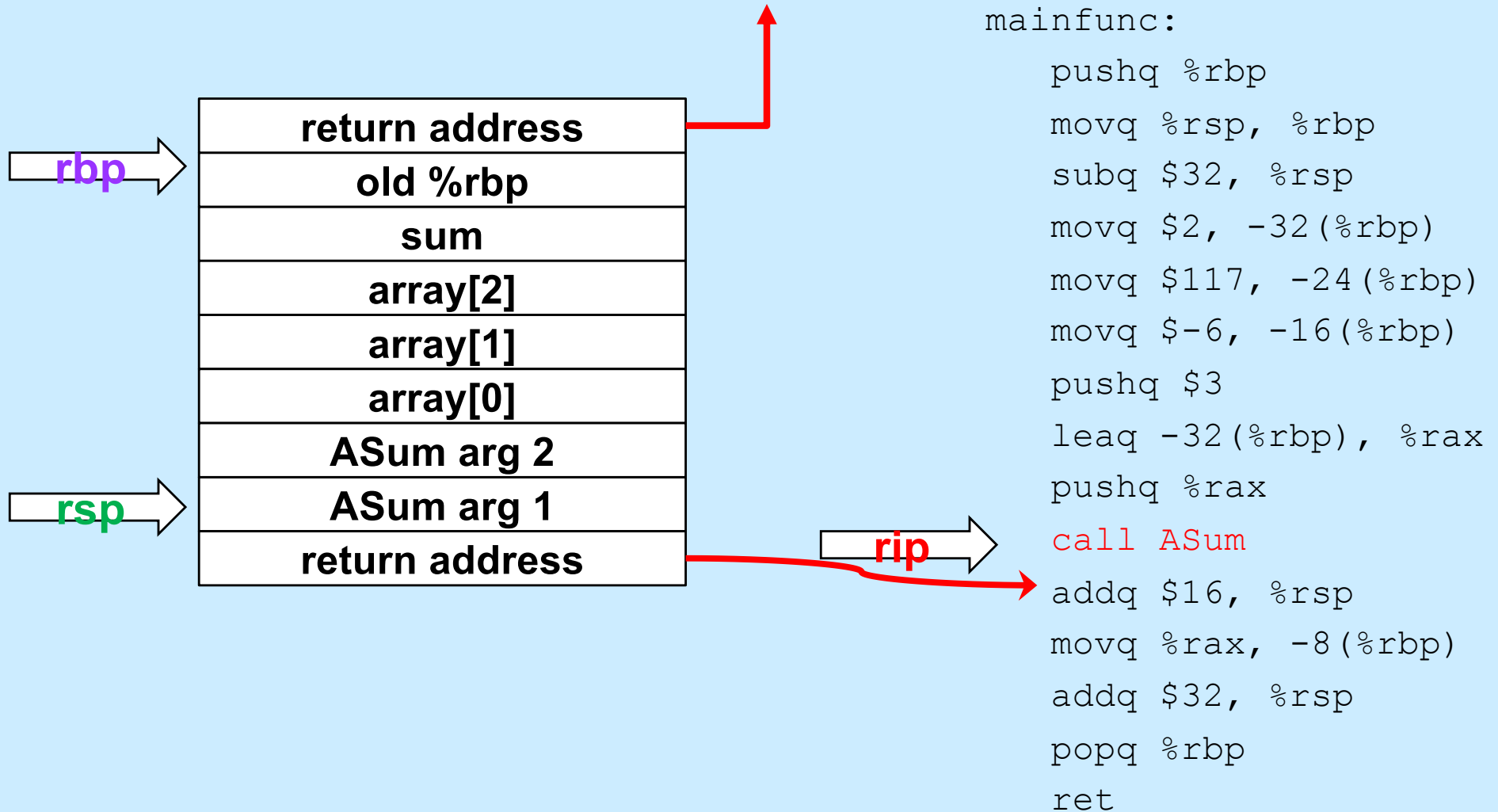
# Push First Argument



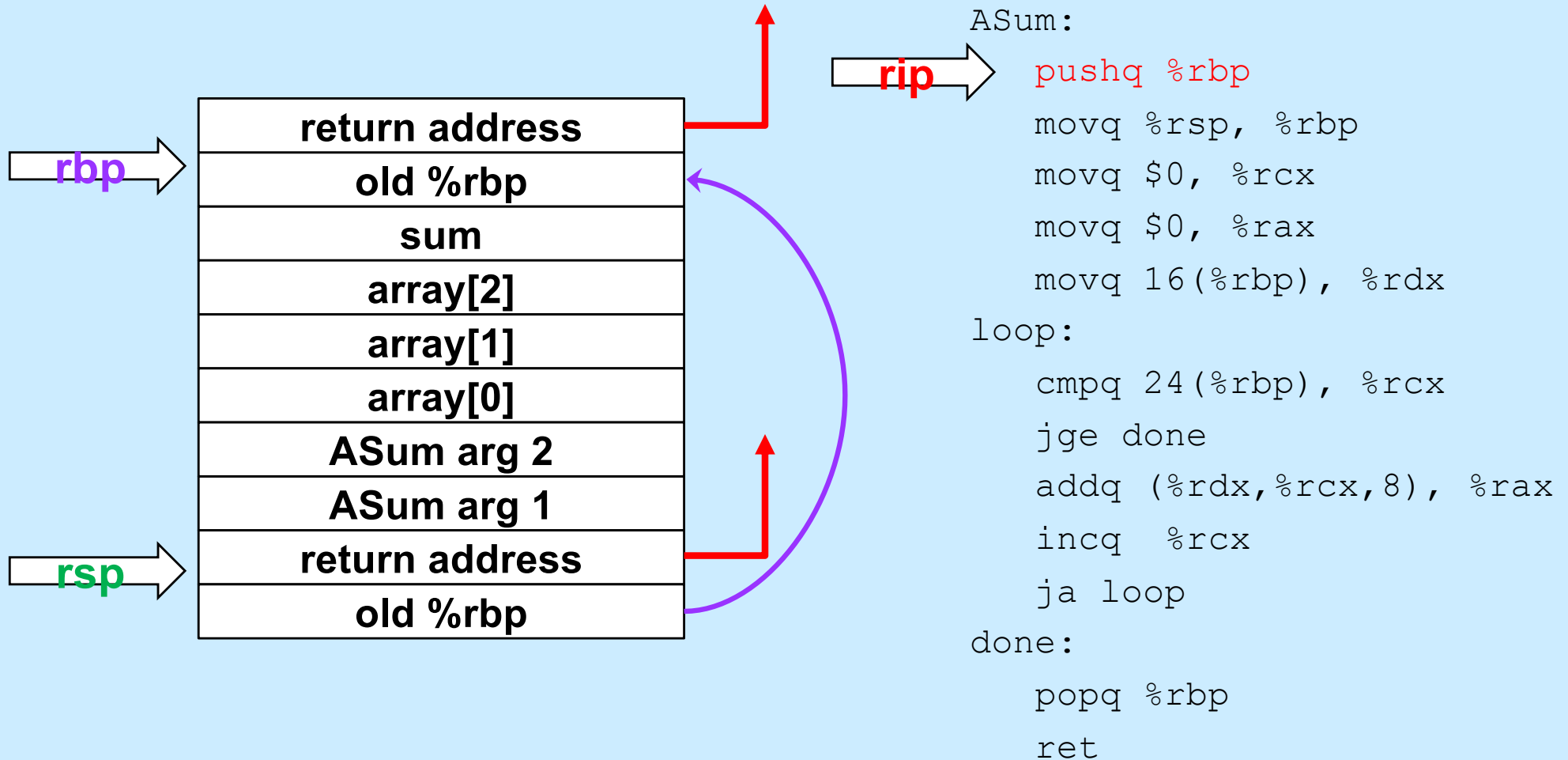
mainfunc:

```
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

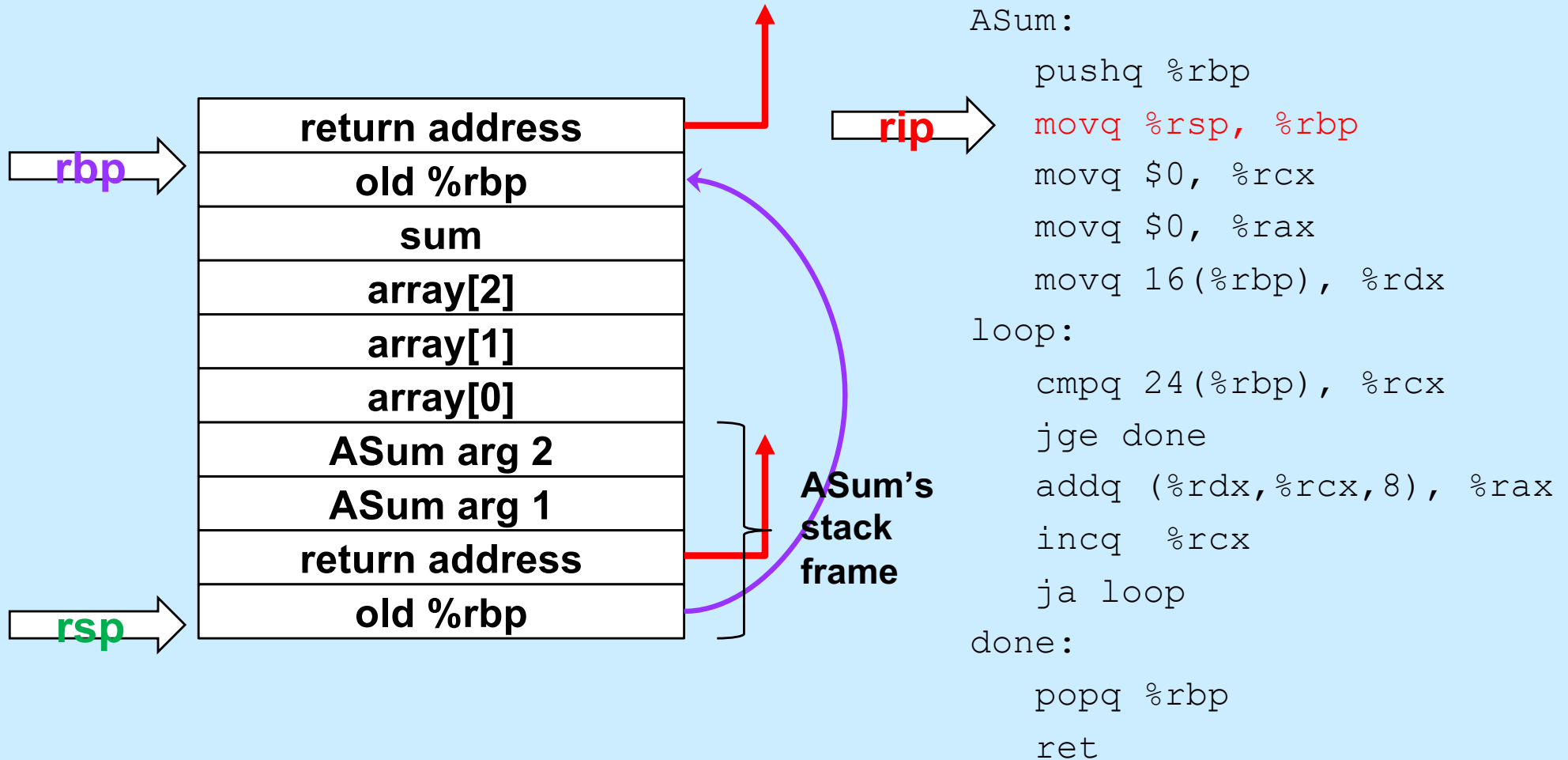


# Enter ASum

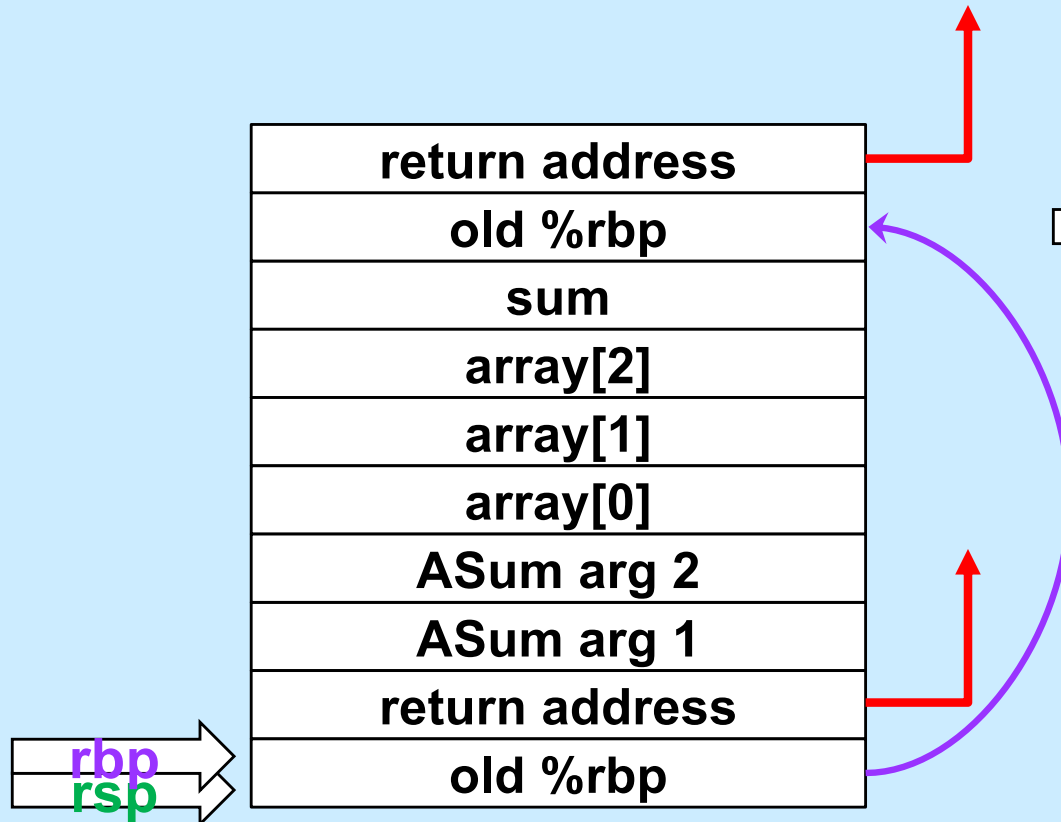




# Setup Frame



# 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 16(%rbp) (after the second instruction is executed)?**

- 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
```

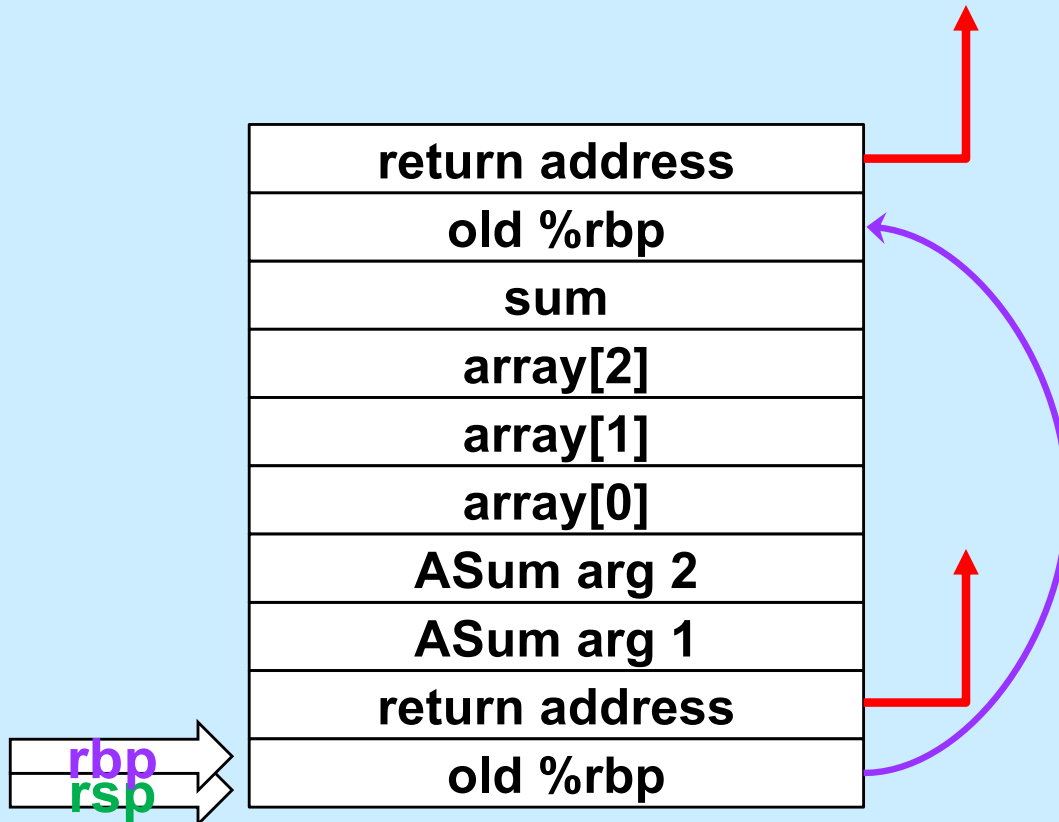
loop:

```
    cmpq 24(%rbp), %rcx
    jge done
    addq (%rdx,%rcx,8), %rax
    incq %rcx
    ja loop
```

done:

```
    popq %rbp
    ret
```

# Prepare to Return

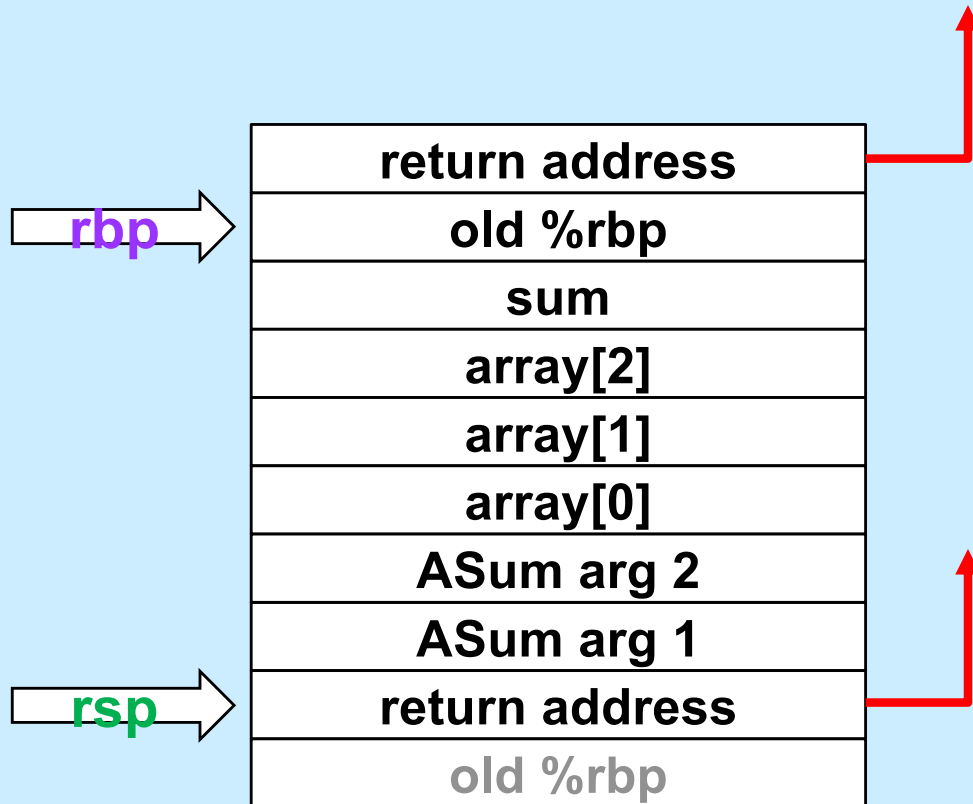


ASum:

```
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
```

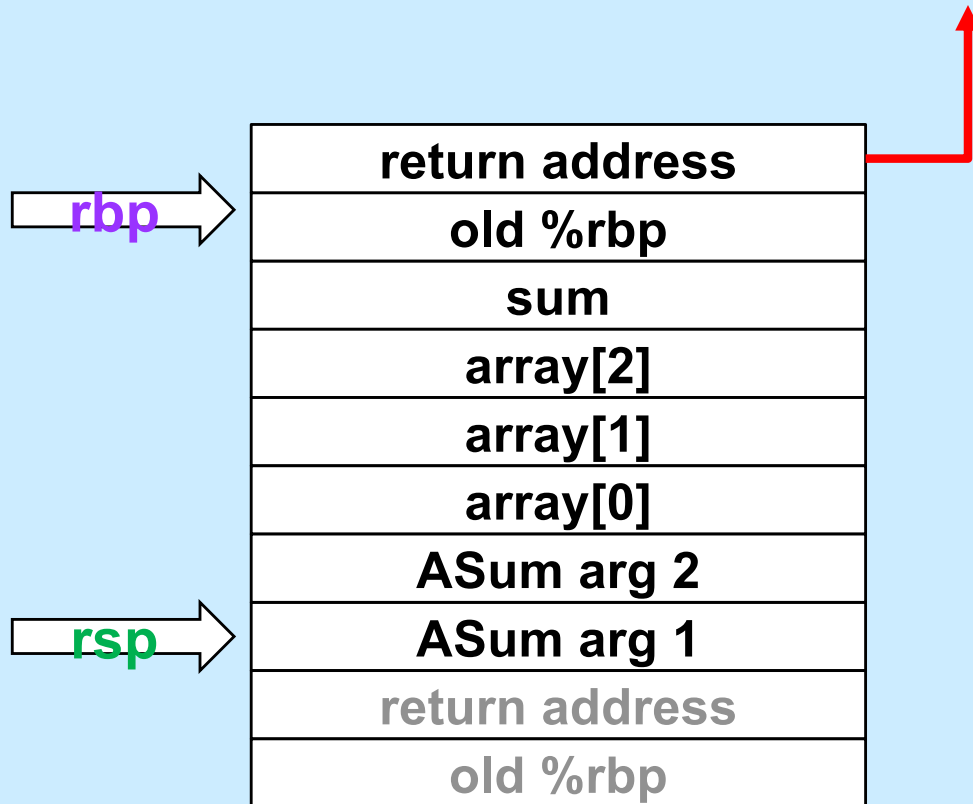
`done:`

```
    popq %rbp
```



```
    ret
```

# Pop Arguments

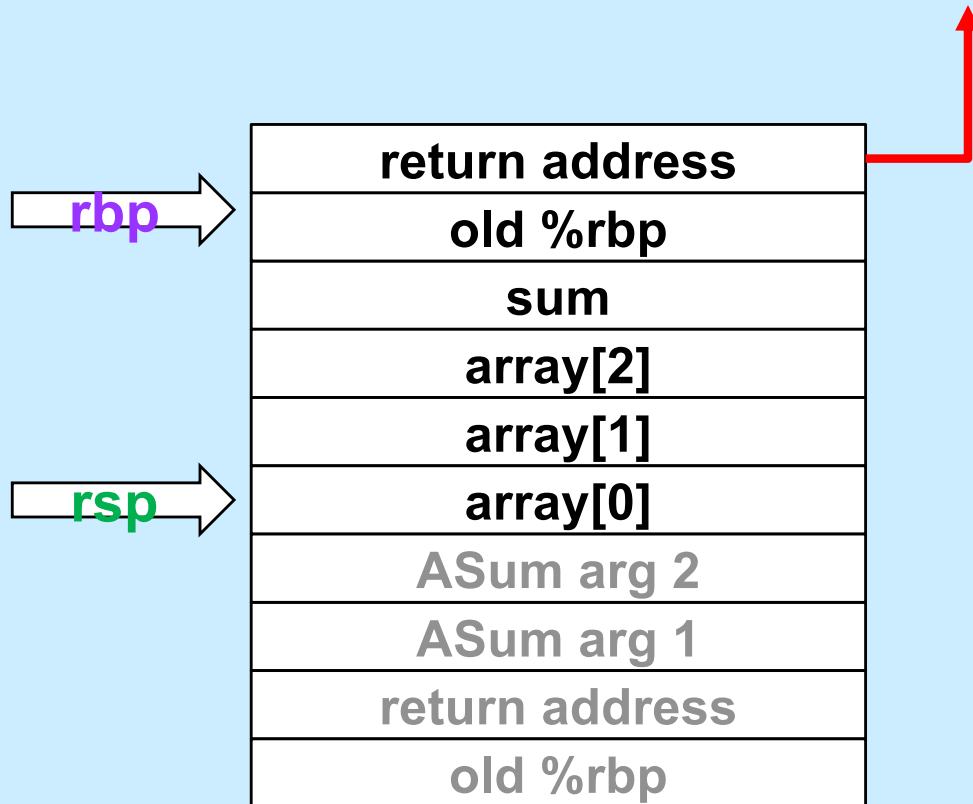


mainfunc:

```
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

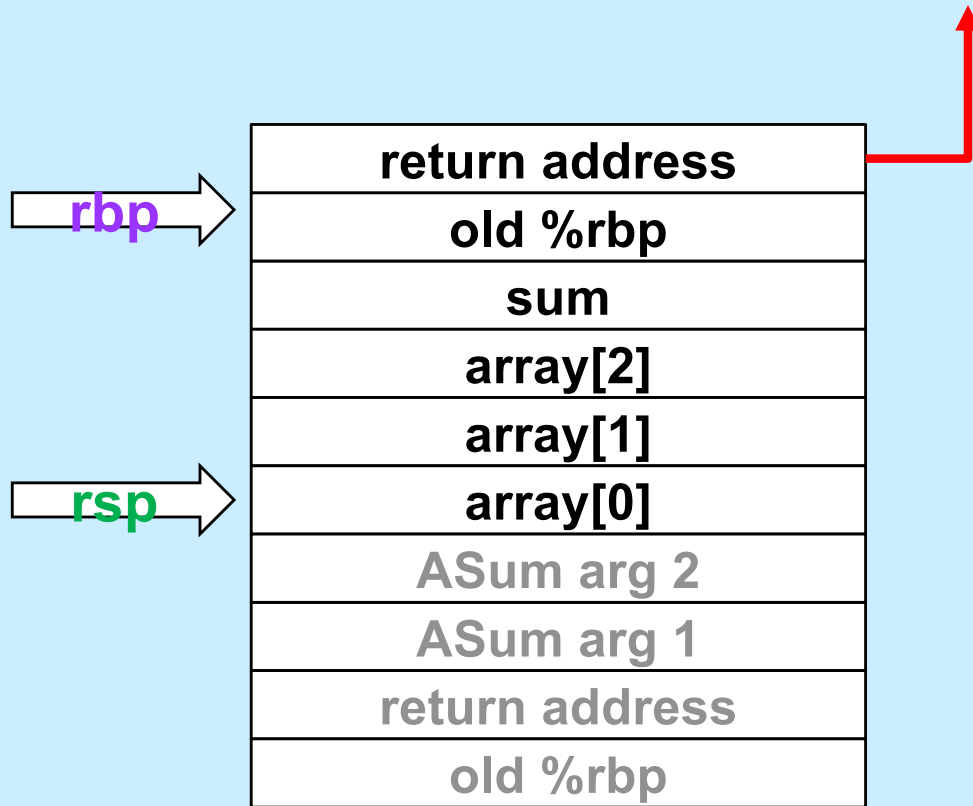


mainfunc:

```
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



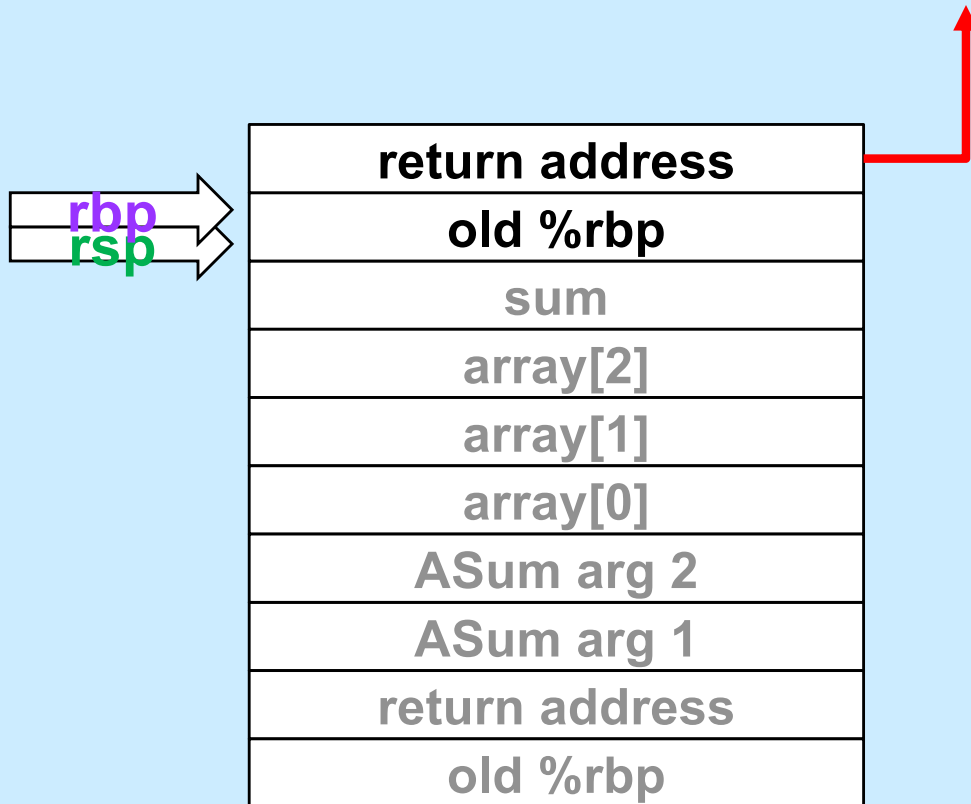
mainfunc:

```
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
```

**rip** →



# Prepare to Return

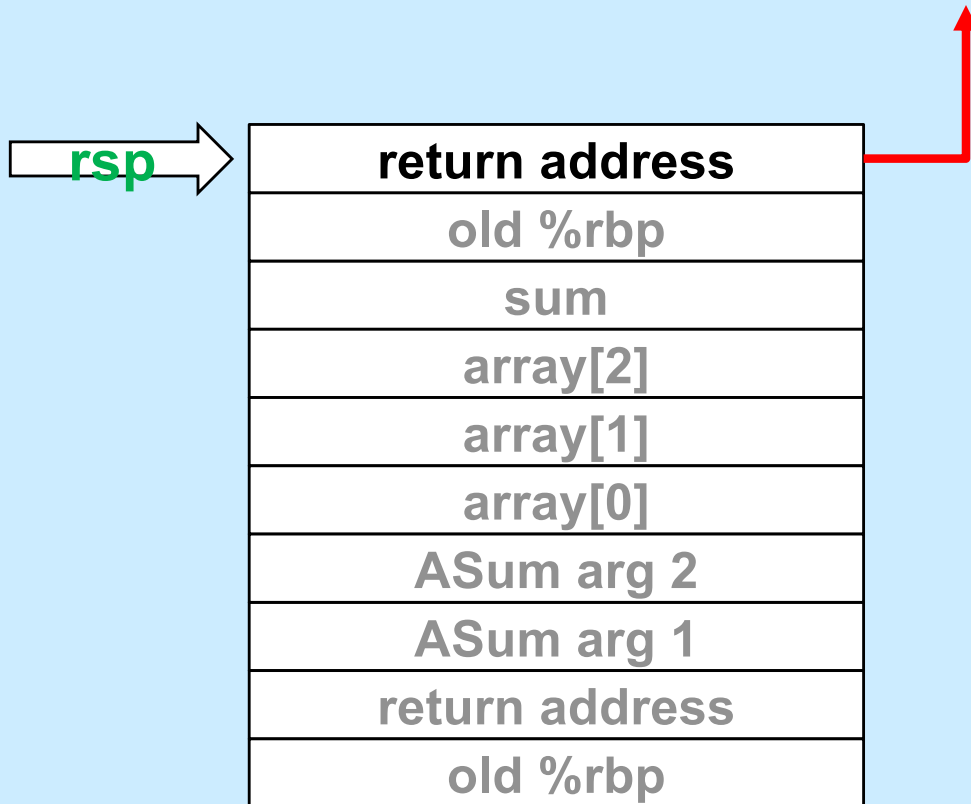


mainfunc:

```
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
```



# 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

<b>%rax</b>	Return value
<b>%rbx</b>	Callee saved
<b>%rcx</b>	Caller saved
<b>%rdx</b>	Caller saved
<b>%rsi</b>	Caller saved
<b>%rdi</b>	Caller saved
<b>%rsp</b>	Stack pointer
<b>%rbp</b>	Base pointer

<b>%r8</b>	Caller saved
<b>%r9</b>	Caller saved
<b>%r10</b>	Caller saved
<b>%r11</b>	Caller Saved
<b>%r12</b>	Callee saved
<b>%r13</b>	Callee saved
<b>%r14</b>	Callee saved
<b>%r15</b>	Callee saved

# 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
-

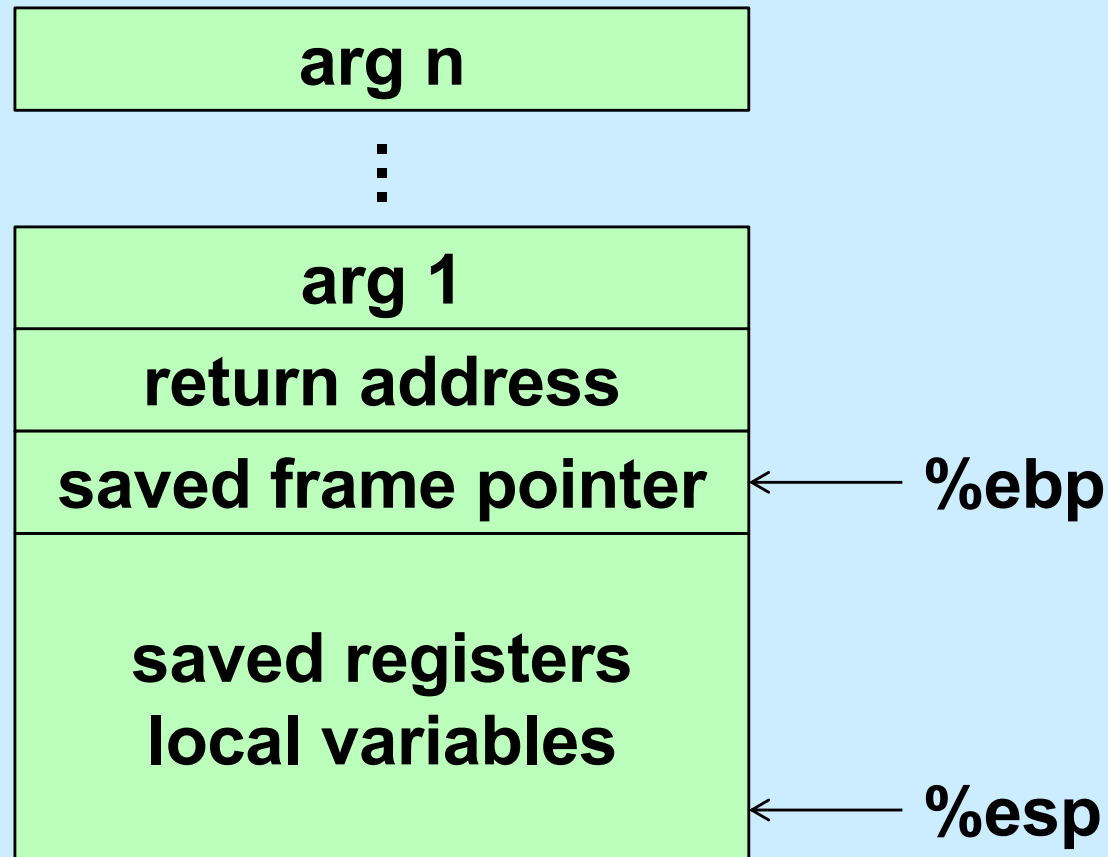


# x86-64 General-Purpose Registers: Updated Usage Conventions

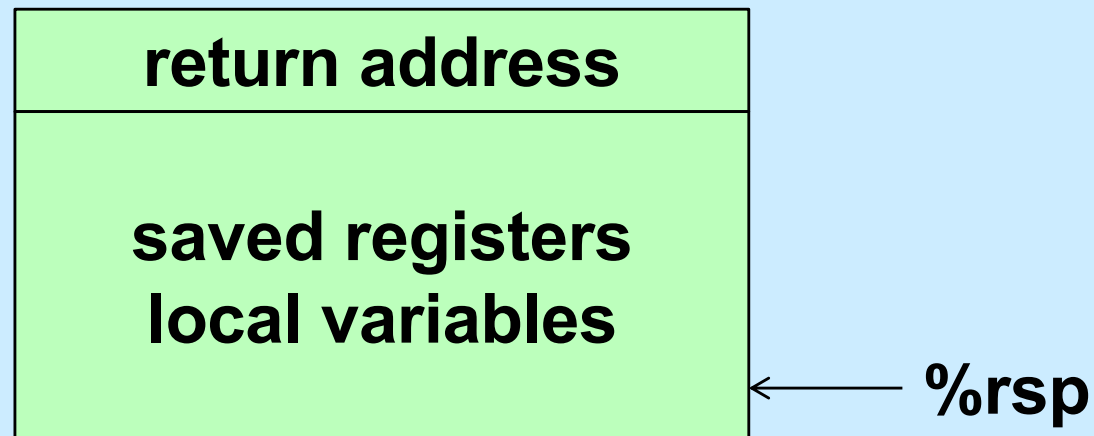
<b>%rax</b>	Return value
<b>%rbx</b>	Callee saved
<b>%rcx</b>	Argument #4
<b>%rdx</b>	Argument #3
<b>%rsi</b>	Argument #2
<b>%rdi</b>	Argument #1
<b>%rsp</b>	Stack pointer
<b>%rbp</b>	Callee saved

<b>%r8</b>	Argument #5
<b>%r9</b>	Argument #6
<b>%r10</b>	Caller saved
<b>%r11</b>	Caller Saved
<b>%r12</b>	Callee saved
<b>%r13</b>	Callee saved
<b>%r14</b>	Callee saved
<b>%r15</b>	Callee saved

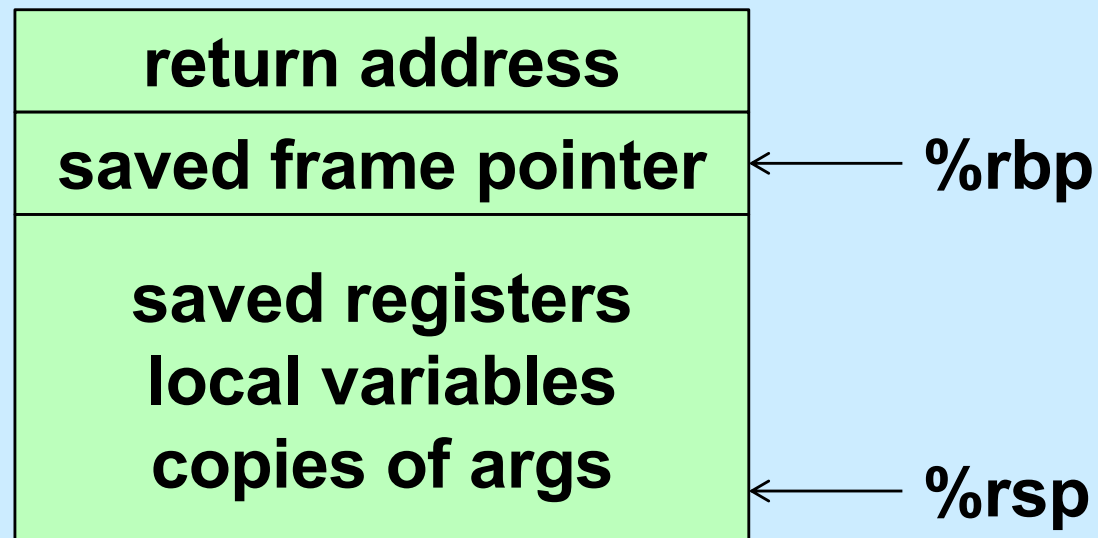
# The IA32 Stack Frame



# The x86-64 Stack Frame



# The -O0 x86-64 Stack Frame (Buffer)



# 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**

# Exploiting the Stack

## Buffer-Overflow Attacks

# String Library Code

- **Implementation of Unix function `gets()`**

```
/* Get string from stdin */
char *gets(char *dest)
{
    int c = getchar();
    char *p = dest;
    while (c != EOF && c != '\n') {
        *p++ = c;
        c = getchar();
    }
    *p = '\0';
    return dest;
}
```

- no way to specify limit on number of characters to read
- **Similar problems with other library functions**
  - `strcpy`, `strcat`: copy strings of arbitrary length
  - `scanf`, `fscanf`, `sscanf`, when given `%s` conversion specification



# Vulnerable Buffer Code

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
int main() {  
    echo();  
  
    return 0;  
}
```

```
unix>./echo  
123  
123
```

```
unix>./echo  
123456789ABCDEF01234567  
123456789ABCDEF01234567
```

```
unix>./echo  
123456789ABCDEF012345678  
Segmentation Fault
```

# Buffer-Overflow Disassembly

echo:

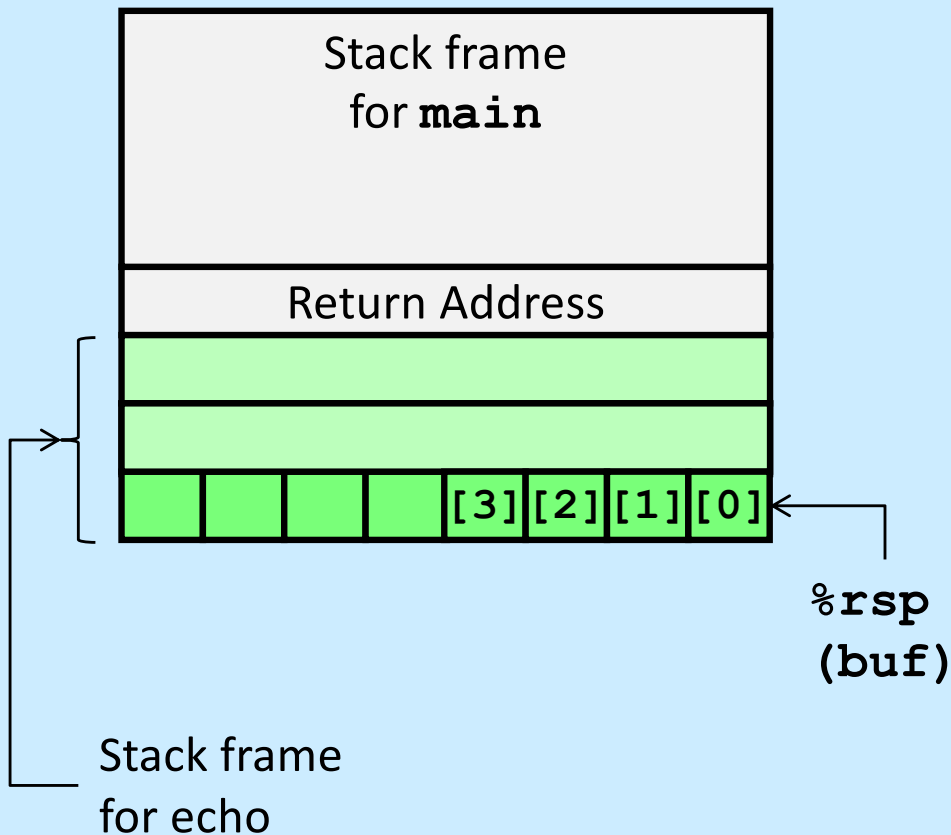
```
000000000040054c <echo>:
  40054c:      48 83 ec 18      sub     $0x18,%rsp
  400550:      48 89 e7         mov     %rsp,%rdi
  400553:      e8 d8 fe ff ff   callq   400430 <gets@plt>
  400558:      48 89 e7         mov     %rsp,%rdi
  40055b:      e8 b0 fe ff ff   callq   400410 <puts@plt>
  400560:      48 83 c4 18      add     $0x18,%rsp
  400564:      c3              retq
```

main:

```
0000000000400565 <main>:
  400565:      48 83 ec 08      sub     $0x8,%rsp
  400569:      b8 00 00 00 00    mov     $0x0,%eax
  40056e:      e8 d9 ff ff ff   callq   40054c <echo>
  400573:      b8 00 00 00 00    mov     $0x0,%eax
  400578:      48 83 c4 08      add     $0x8,%rsp
  40057c:      c3              retq
```

# Buffer-Overflow Stack

*Before call to gets*

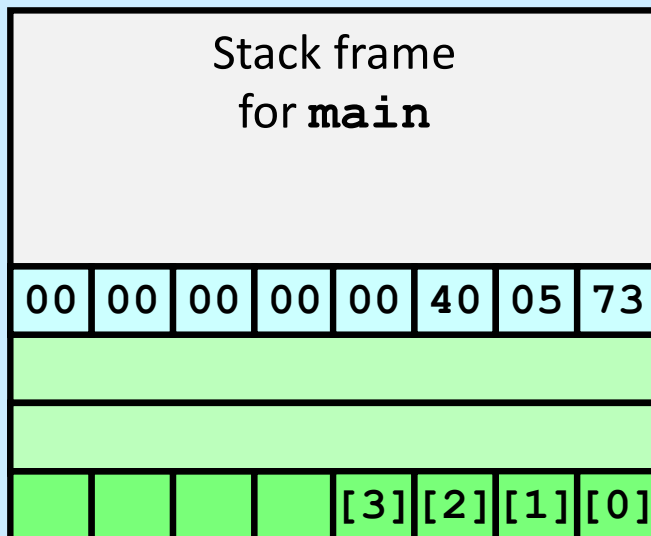


```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Too small! */  
    gets(buf);  
    puts(buf);  
}
```

```
echo:  
    subq    $24, %rsp  
    movq    %rsp, %rdi  
    call    gets  
    movq    %rsp, %rdi  
    call    puts  
    addq    $24, %rsp  
    ret
```

# Buffer Overflow Stack Example

```
unix> gdb echo
(gdb) break echo
Breakpoint 1 at 0x40054c
(gdb) run
Breakpoint 1, 0x000000000040054c in echo ()
(gdb) print /x $rsp
$1 = 0x7fffffffef988
(gdb) print /x *(unsigned *)$rsp
$2 = 0x400573
```

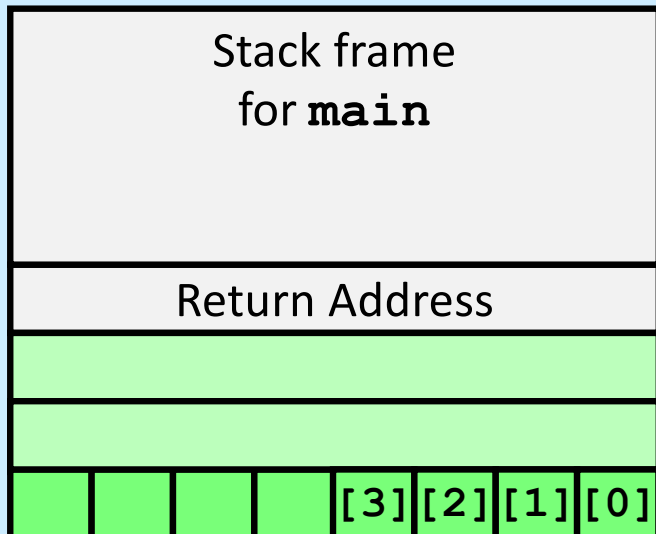


40056e:	e8 d9 ff ff ff	callq	40054c <echo>
400573:	b8 00 00 00 00	mov	\$0x0,%eax

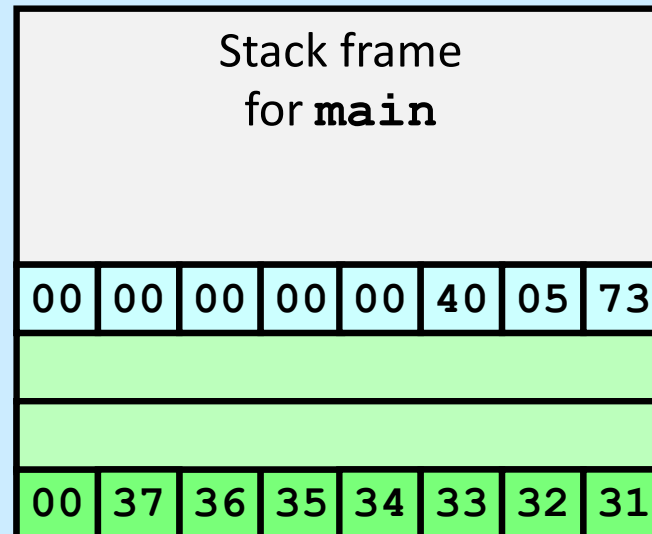
---

# Buffer Overflow Example #1

*Before call to gets*



*Input 1234567*



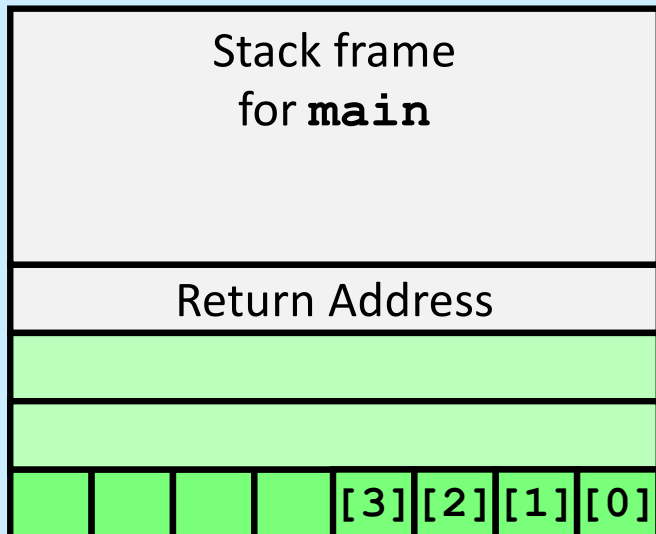
Overflow buf, but no problem

```
40056e:      e8 d9 ff ff ff      callq  40054c <echo>
400573:      b8 00 00 00 00      mov     $0x0,%eax
```

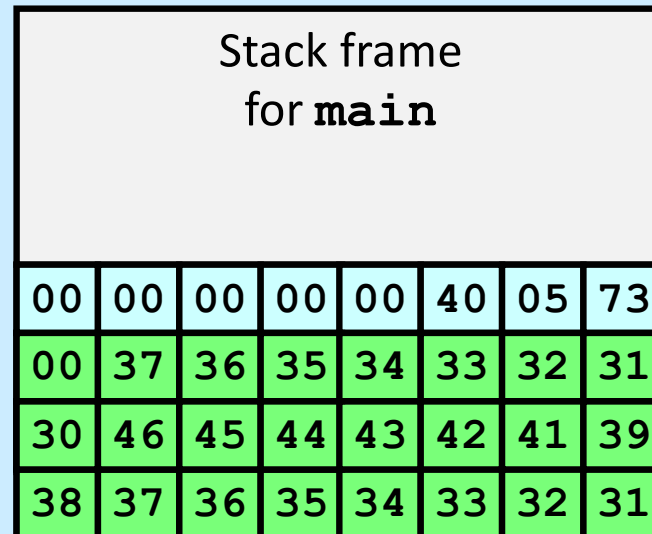
---

# Buffer Overflow Example #2

*Before call to gets*



*Input 123456789ABCDEF01234567*



Still no problem

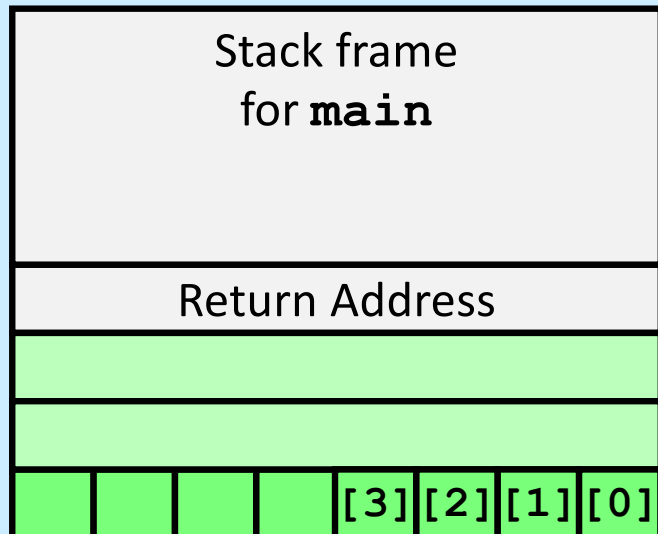
```

40056e:      e8 d9 ff ff ff      callq  40054c <echo>
400573:      b8 00 00 00 00      mov     $0x0,%eax
  
```

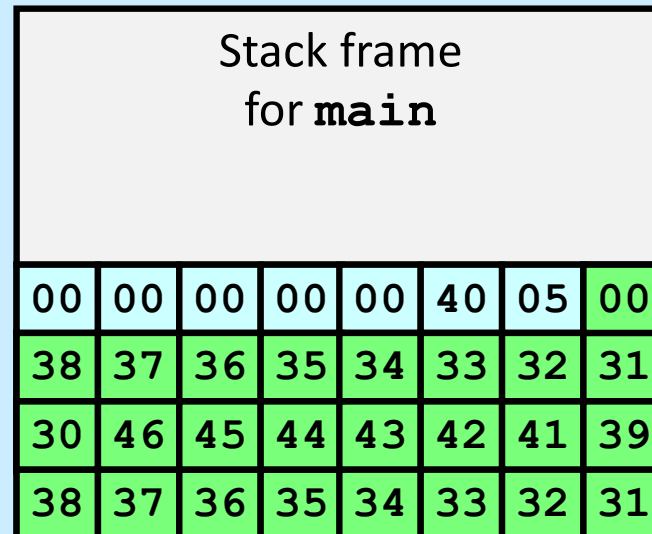
---

# Buffer Overflow Example #3

*Before call to gets*



*Input 123456789ABCDEF012345678*



**Return address corrupted**

```

40056e:      e8 d9 ff ff ff    callq  40054c <echo>
400573:      b8 00 00 00 00    mov     $0x0,%eax
  
```

---

# Avoiding Overflow Vulnerability

```
/* Echo Line */  
void echo()  
{  
    char buf[4]; /* Way too small! */  
    fgets(buf, 4, stdin);  
    puts(buf);  
}
```

- **Use library functions that limit string lengths**
  - **fgets** instead of **gets**
  - **strncpy** instead of **strcpy**
  - **don't use scanf with %s conversion specification**
    - » **use fgets to read the string**
    - » **or use %ns where n is a suitable integer**



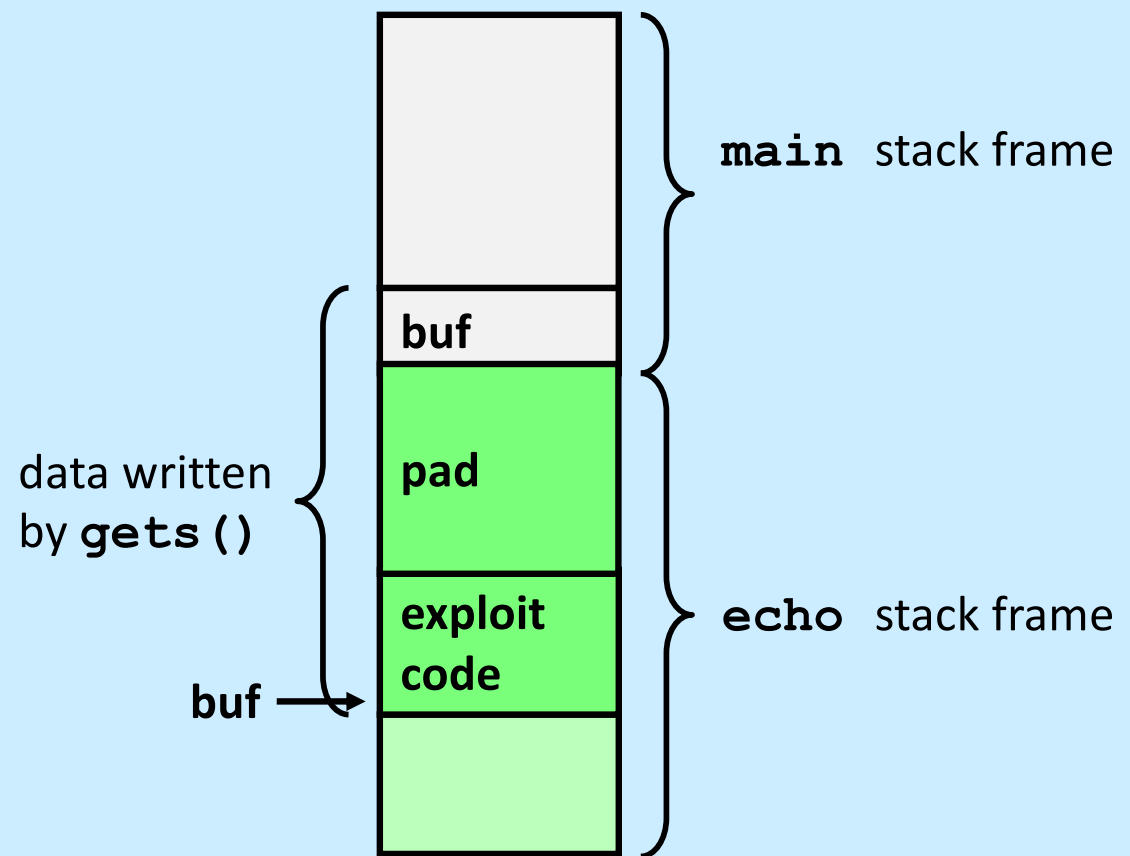
# Malicious Use of Buffer Overflow

Stack after call to `gets()`

```
void main() {  
    echo();  
    ...  
}
```

return address A

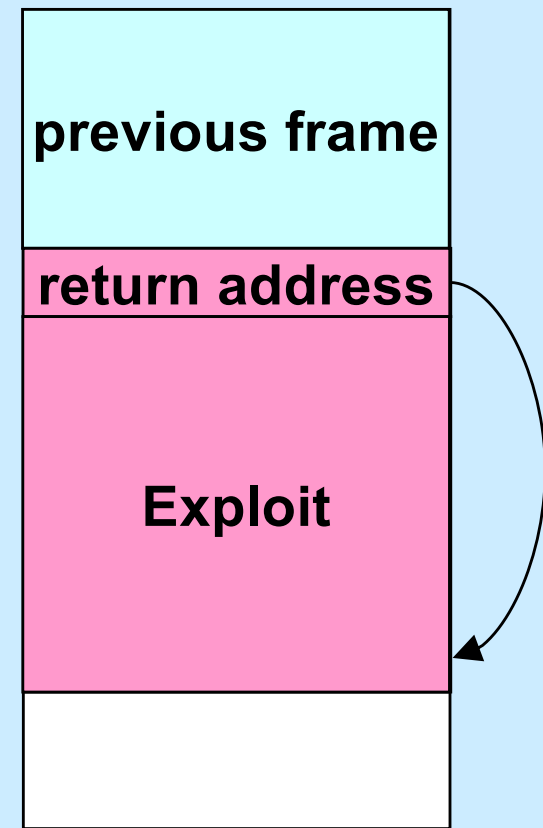
```
int echo() {  
    char buf[80];  
    gets(buf);  
    ...  
    return ...;  
}
```



- Input string contains byte representation of executable code
- Overwrite return address A with address of buffer `buf`
- When `echo()` executes `ret`, will jump to exploit code

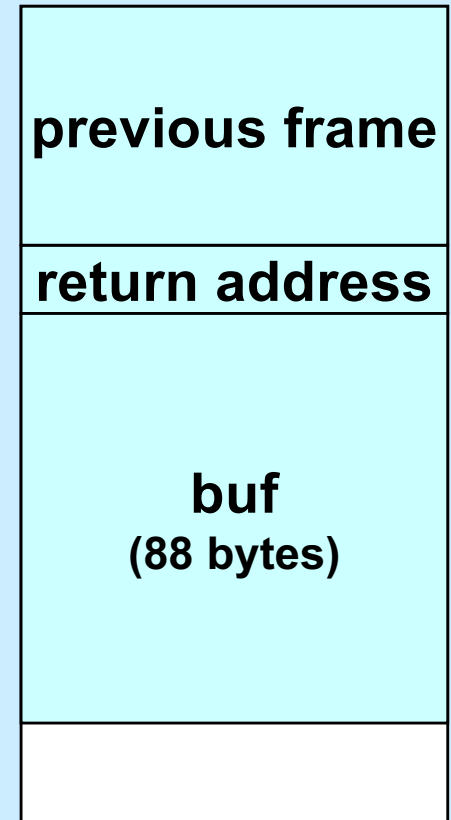
```
int main( ) {  
    char buf[80];  
    gets(buf);  
    puts(buf);  
    return 0;  
}
```

```
main:  
    subq    $88, %rsp    # grow stack  
    movq    %rsp, %rdi   # setup arg  
    call    gets  
    movq    %rsp, %rdi   # setup arg  
    call    puts  
    movl    $0, %eax     # set return value  
    addq    $88, %rsp    # pop stack  
    ret
```



# Crafting the Exploit ...

- **Code + padding**
  - 96 bytes long
    - » 88 bytes for buf
    - » 8 bytes for return address



## Code (in C):

```
void exploit() {  
    write(1, "hacked by twd",  
          strlen("hacked by twd\n"));  
    exit(0);  
}
```

# Quiz 3

The exploit code will be read into memory starting at location **0x7fffffffe948**. What value should be put into the return-address portion of the stack frame?

- a) 0
- b) 0x7fffffffe9a0
- c) 0x7fffffffe948
- d) it doesn't matter what value goes there

