程序的机器级表示（5）

谢旻晖

2025.11

Addressing Mode（寻址方式）

Type	Form	Operand value	Name
Immediate	$Imm	Imm	Immediate
Register	E_a	$R[E_a]$	Register
Indexed	Imm	M[Imm]	Absolute
Indexed	(E_a)	$M[R[E_a]]$	Indirect
Indexed	$Imm(E_b)$	$M[Imm+ R[E_b]]$	Base+displacement
Indexed	(E_b, E_i)	$M[R[E_b]+ R[E_i]]$	Indexed
Indexed	$Imm(E_b, E_i)$	$M[Imm+ R[E_b]+ R[E_i]]$	Scaled indexed
Indexed	$(, E_i, s)$	$M[R[E_i]*s]$	Scaled indexed
Indexed	(E_b, E_i, s)	$M[R[E_b]+ R[E_i]*s]$	Scaled indexed
Indexed	$Imm(E_b, E_i, s)$	$M[Imm+ R[E_b]+ R[E_i]*s]$	Scaled indexed

Address	Value
0x100	0xFF
0x104	0xAB
0x108	0x13
0x10C	0x11

Register	Value
%eax	0x100
%ecx	0x1
%edx	0x3

Operand	Value
%eax	0x100
(%eax)	0xFF
$0x108	0x108
0x108	0x13
260(%ecx,%edx)	(0x108) 0x13
(%eax,%edx,4)	(0x10C) 0x11

leaq off(base, i, scale), rax

Rax = base + scale * I + off

---

movq off(base, i, scale), rax

Rax = *(base + scale * I + off)

movq %rax, %rbx

rbx = rax

movq (%rax), %rbx

rbx = *(rax)

movq (,%rax, 8), %rbx leaq (,%rax, 8), %rbx

rbx = *(8*rax) rbx=8*rax

主观题 10分

设置

填写C语言代码：

```
Long test (long x, long y) {
    long val = _____;
    if (_____) {
        if (_____) {
            val = _____;
        else
            val = _____;
    } else if (_____)
        val = _____;
    return val;
}
```

```
# x @ %rdi, y @ %rsi
Test:
  leaq    0(,%rdi,8), %rax
  testq   %rsi, %rsi
  jle     .L2
  movq    %rsi, %rax
  subq    %rdi, %rax
  movq    %rdi, %rdx
  andq    %rsi, %rdx
  cmpq    %rsi, %rdi
  cmovge %rdx, %rax
.L2:
  addq    %rsi, %rdi
  cmpq    $-2, %rsi
  cmovle %rdi, %rax
  ret
```

作答

通知！！！！

期中考试时间：11月12日（周三）晚上上机时间考试

地点：待定，群里通知

考试内容：

第一章开始，考到简单的汇编指令

（可能不包括循环/条件/过程调用等，

具体视柴老师班讲课进度，后面再具体给通知）

Procedure Call

Outline

- Procedure call
- Stack frame
- Calling conventions
- Recursive

Execution within Procedure/Function

- Data Movement (e.g., movl $-17, (%esp))
- Arithmetic Operations (e.g., incl 8(%eax))
- Logical Operations (e.g., xorl 8(%esp), %eax)
- Condition Codes (e.g., cmpl %eax, %edx)
- Jump Instructions (e.g., jg .L5)

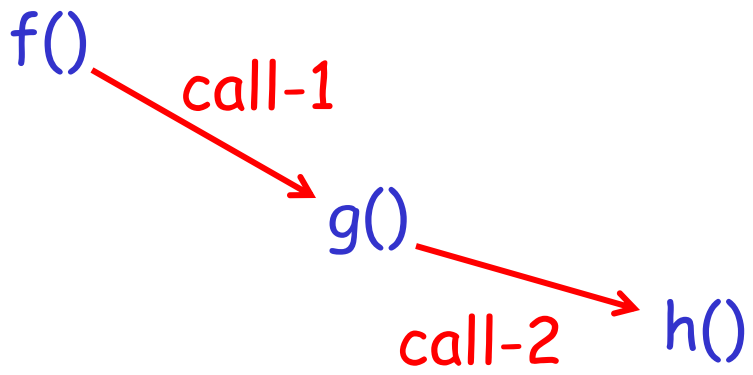
How to execute cross procedures?

Procedure/Function call

- Another type of unconditional JUMP
  - SAME:
    - Control from one part to another
  - DIFF:
    - Return：往返 vs. 单程
    - Passing data (arguments, return values)
    - Local variable
    - Registers：保存离开时的状态

Basic Concept

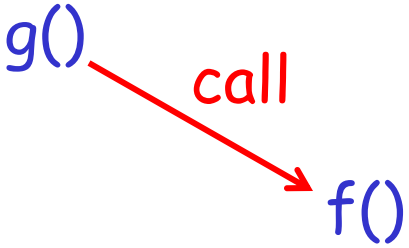
- Terminology
  - Caller
  - Callee



- call-1
  - Caller: f
  - Callee: g
- call-2
  - Caller: g
  - Callee: h

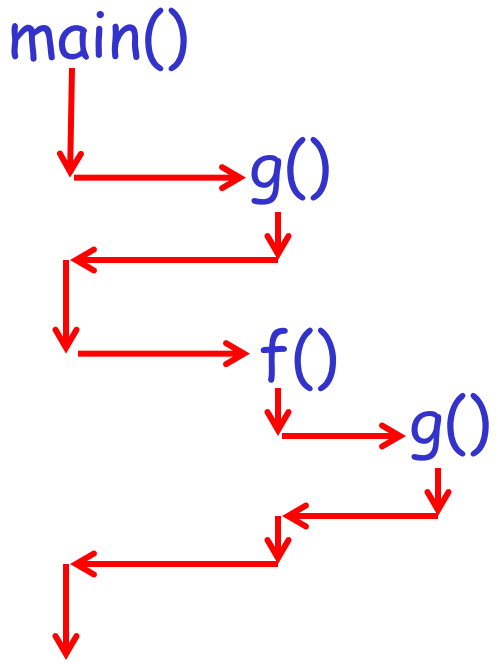
Basic Concept

- Terminology
  - Caller: g
  - Callee: f



- Control Flow

e.g. int g() { return 1 ; }
int f() { return g() ; }
int main() { g(); return f(); }



Procedure/Function Implementation

- Invoke callee
- Return to caller
- Passing data
- Registers
- Local variable

Procedure/Function Implementation

- Invoke callee: call (new instructions)
- Return to caller
- Passing data
- Registers
- Local variable

Invoke Callee

- Instruction
  - **`call` *`label`* (direct)**
  - **`call` *`*operand`* (indirect)**
- Behavior description (by hardware)
  - Save *return address* in the stack (call下一条指令)
  - Jump to the *entry* of callee

call = push + jmp

```
push retaddr
jmp callee
```

Execution of call and ret

```
//beginning of function sum
1. 08048394 <sum>:
2.   8048394:  55                  push  %ebp
     . . .
3.   80483a4:                      ret
     . . .
//call to sum from main
4.   80483dc:  e8 b3 ff ff ff      call  8048394<sum>
5.   80483e1:  83 c4 14            add   $0x14, %esp
```

Executing call

%eip	0x080483dc
%esp	0xff9bc960
ret	-

After call

%eip	0x08048394
%esp	0xff9bc95c
ret	0x080483e1

Procedure/Function Implementation

- ✓ Invoke callee: call (new instructions)
- ? Return to caller: ret (new instructions)
- ? Passing data
- ? Registers
- ? Local variable

Return to Caller

- Instruction
  - **`ret`**
- Behavior description (by hardware)
  - Pop *return address* from stack
  - Jump to *return address* in caller

ret = pop + jmp

```
pop retaddr
jmp retaddr
```

Execution of call and ret

```
//beginning of function sum
1. 08048394 <sum>:
2.   8048394:   55                push   %ebp
     . . .
3.   80483a4:                     ret
     . . .
//call to sum from main
4.   80483dc:   e8 b3 ff ff ff   call   8048394<sum>
5.   80483e1:   83 c4 14         add    $0x14, %esp
```

executing ret

%eip	0x080483a4
%esp	0xff9bc95c
ret	0x080483e1

After ret

%eip	0x080483e1
%esp	0xff9bc960
ret	-

Procedure/Function Implementation

- ✓ Invoke callee: call (new instructions)
- ✓ Return to caller: ret (new instructions)
- ? Passing data: stack, register
- ? Registers
- ? Local variable

课堂练习

•下面的代码片段经常出现在库函数的编译版本中：

```
1    call next
2next:
3    popl %eax
```

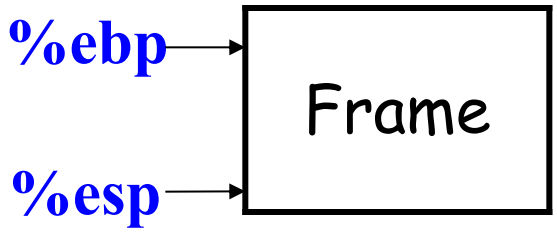
A)Eax被设置成什么值？

B)解释为什么这个调用没有与之匹配的ret指令

C)这段代码完成了什么功能？

Stack Frame Structure

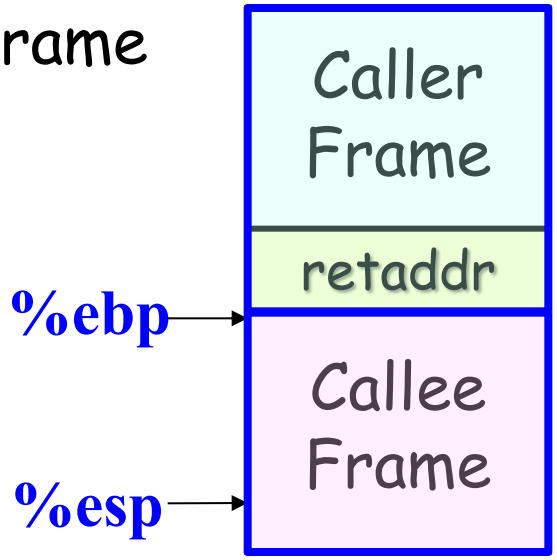
- The portion of stack allocated for a procedure
- A stack frame is delimited by
  - The frame pointer %ebp
  - The stack pointer %esp



- The stack pointer can move when the procedure is executing (dynamic)
- The frame pointer is static (%ebp)

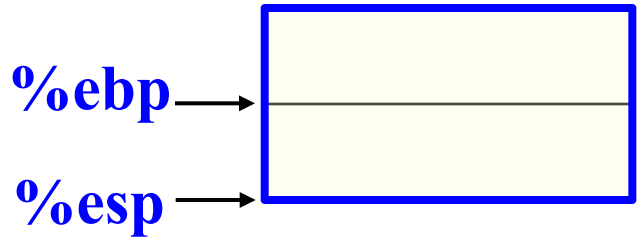
Stack Frame Structure

- 主函数、子函数共享同一个system stack
- call: save *return address* in the stack
- ret: pop *return address* from stack
  - The end of caller's stack frame



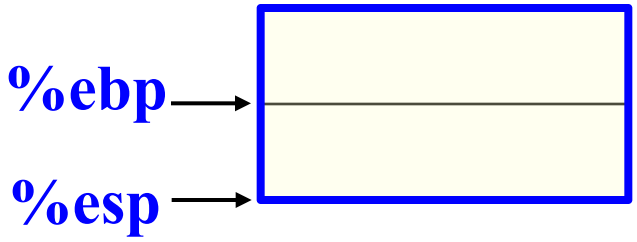
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained
    1. call callee



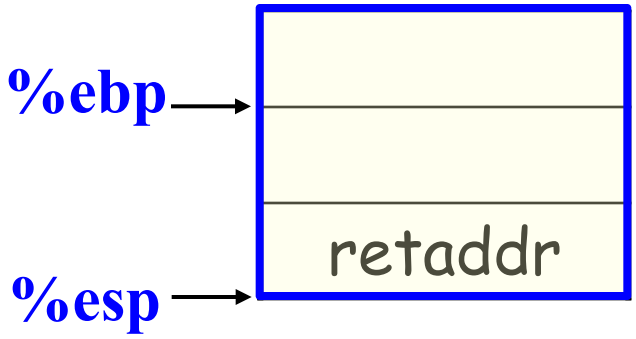
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

2. push %ebp
3. mov %esp, %ebp

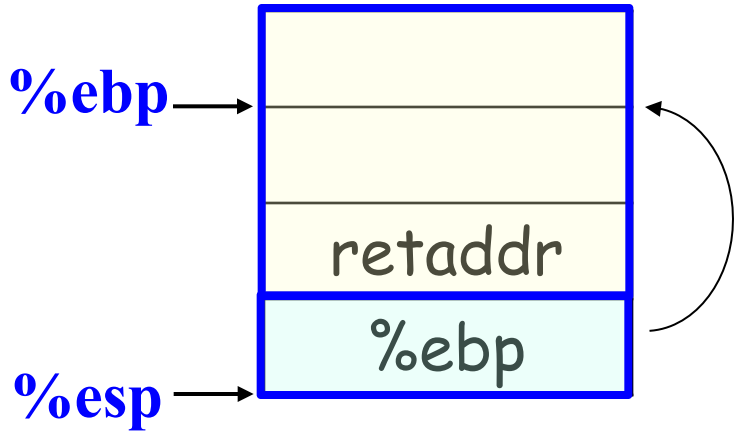


Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained
  1. call callee

  callee:

  2. push %ebp
  3. mov %esp, %ebp



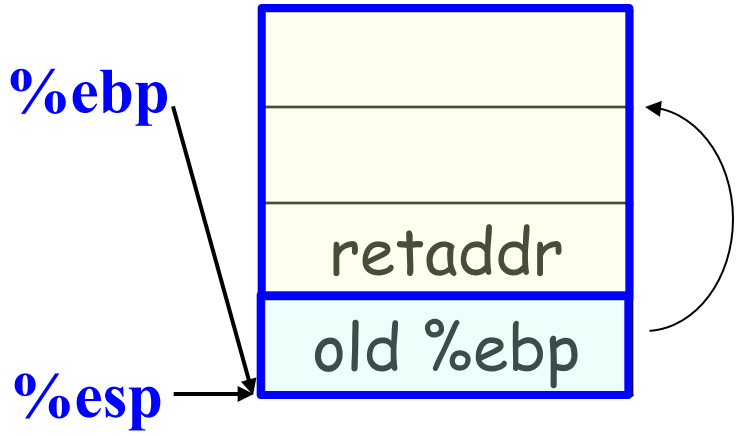
Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

2. push %ebp
3. mov %esp, %ebp



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

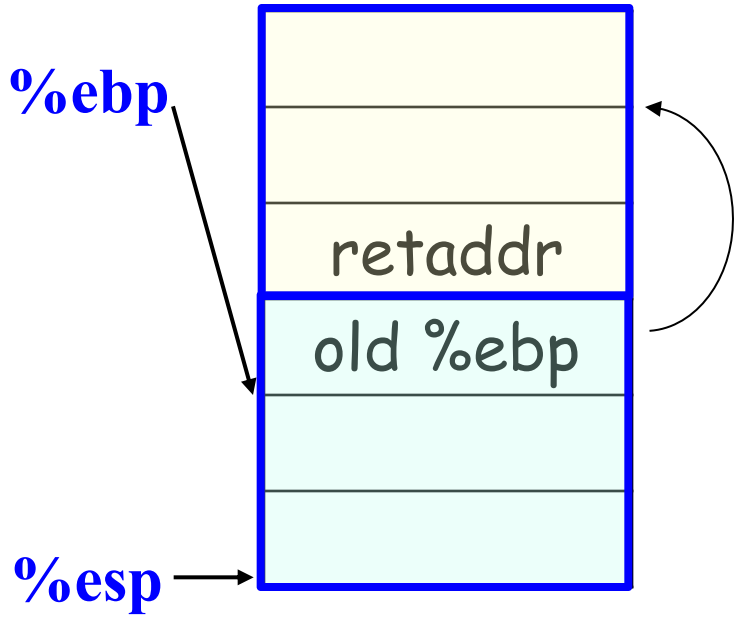
2. push %ebp
3. mov %esp, %ebp

. . .

n-2. mov %ebp, %esp

n-1. pop %ebp

n. ret



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

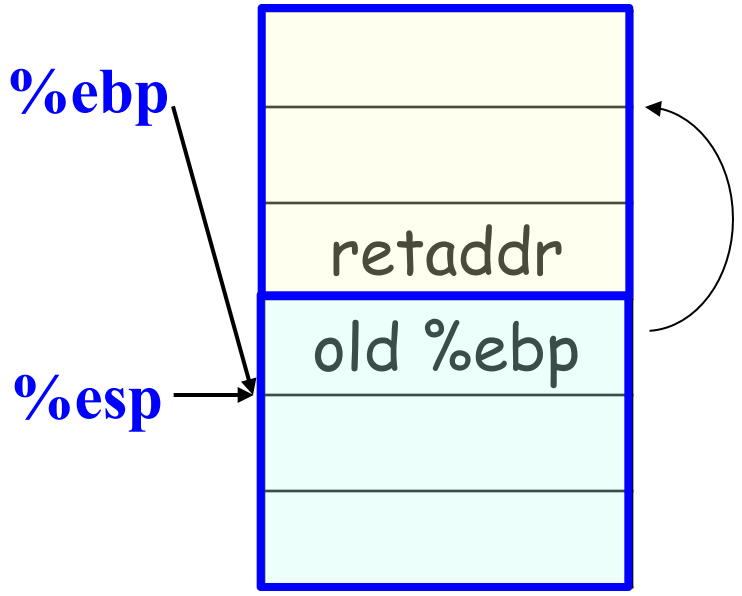
2. push %ebp
3. mov %esp, %ebp

. . .

n-2. mov %ebp, %esp

n-1. pop %ebp

n. ret



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

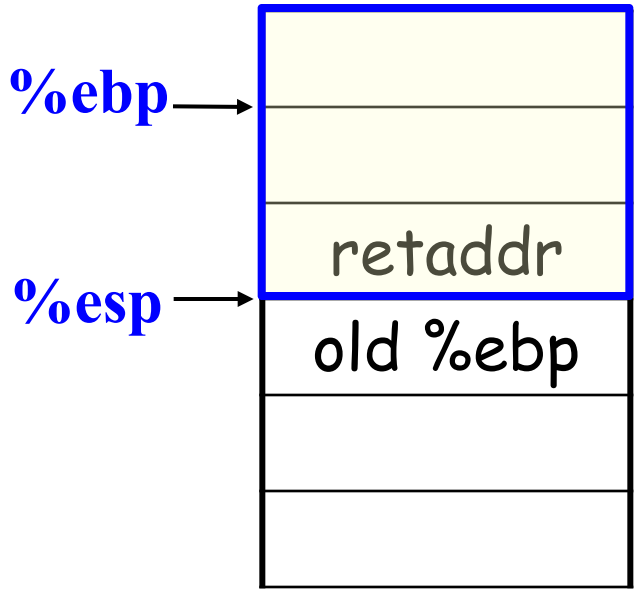
2. push %ebp
3. mov %esp, %ebp

. . .

n-2. mov %ebp, %esp

n-1. pop %ebp

n. ret



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained

1. call callee

callee:

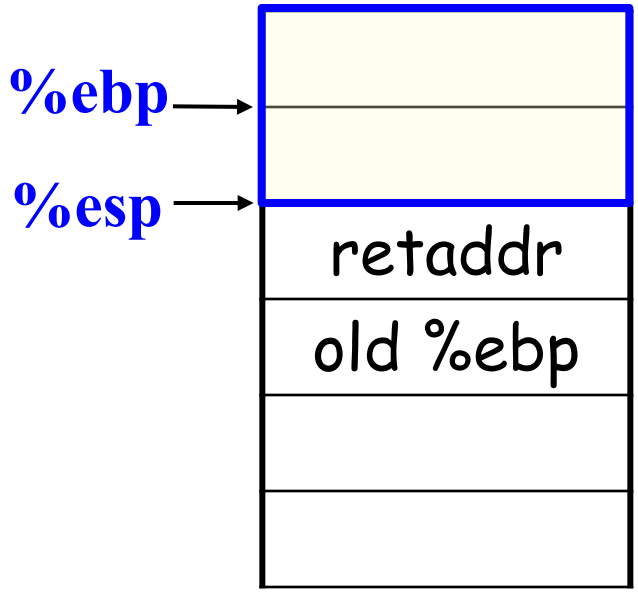
2. push %ebp
3. mov %esp, %ebp

. . .

n-2. mov %ebp, %esp

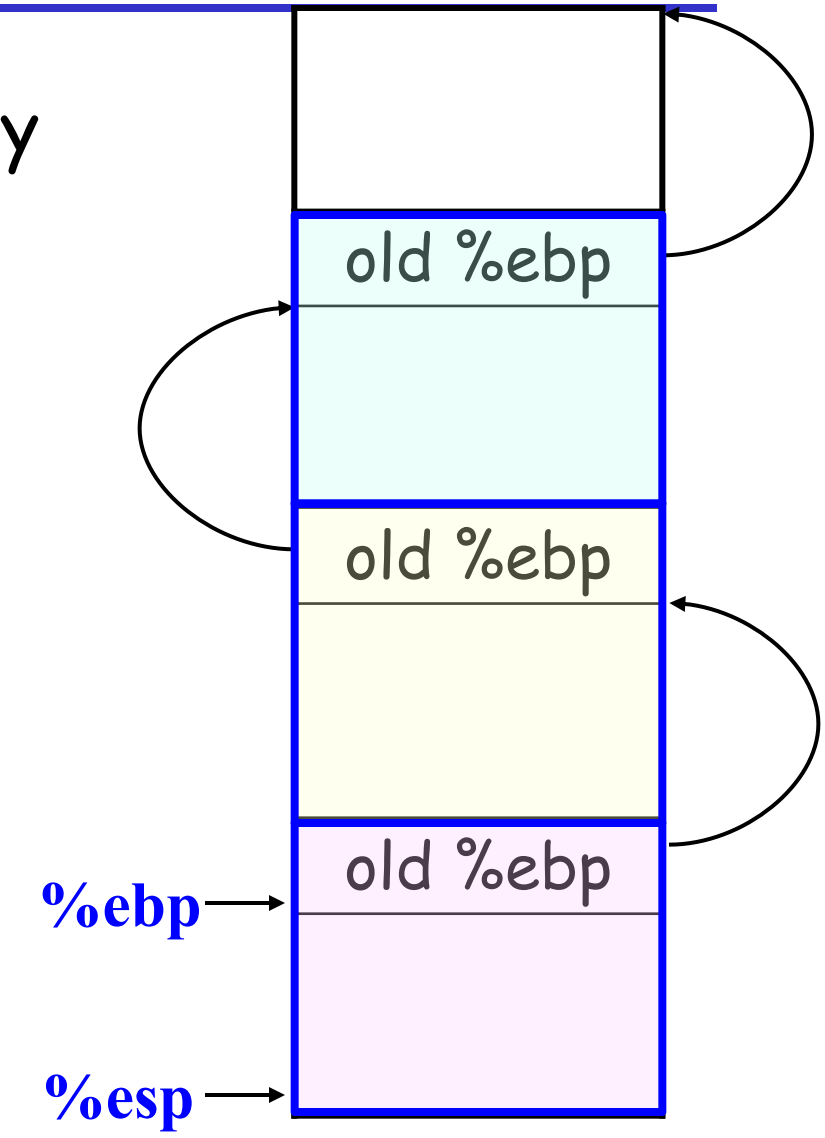
n-1. pop %ebp

n. ret



Frame Chain

- Pointers (%ebp/%esp) only delimit topmost frame
- Frames are chained



Restore Caller %ebp

- Instruction
  - **`leave`**
- Behavior description (by hardware)
  - Adjust %esp to callee %ebp
  - Pop *caller %ebp* from stack

leave = mov + pop

```
mov %ebp, %esp
pop %ebp
```

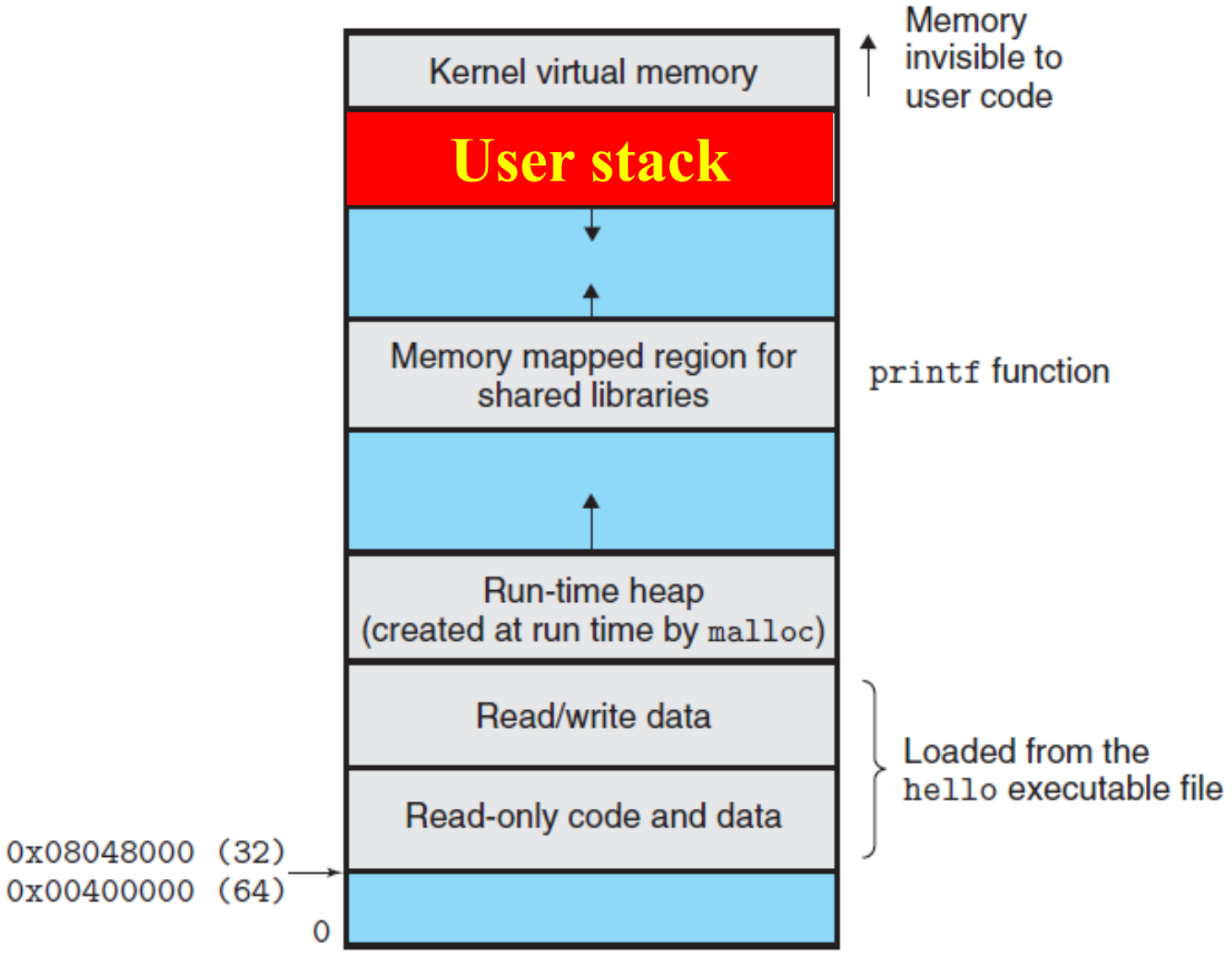
上次讲到这里

Execution of call and ret

```
//beginning of function sum
1. 08048394 <sum>:
2.   8048394:   55                  push   %ebp
     . . .
3.   80483a4:                       ret
     . . .
//call to sum from main
4.   80483dc:   e8 b3 ff ff ff      call   8048394<sum>
5.   80483e1:   83 c4 14            add    $0x14, %esp
```

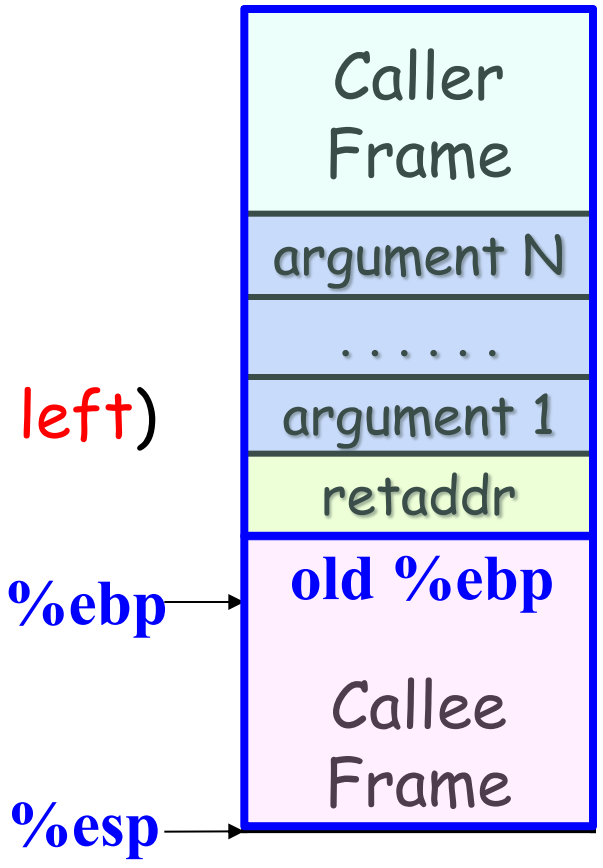
Executing call		After call		After ret	
%eip	0x080483dc	%eip	0x08048394	%eip	0x080483e1
%esp	0xff9bc960	%esp	0xff9bc95c	%esp	0xff9bc960
ret	-	ret	0x080483e1	ret	-

Memory Layout



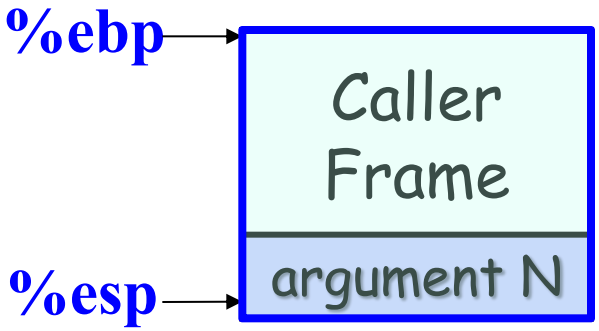
Passing Data: Arguments

- Pushed by Caller
  - Saved in caller frame
  - Just upon of return address
  - From Nth to 1st (from right to left)
- Used by Callee
  - Relative to %ebp
  - Offset: 8 + 4*i + %ebp



Passing Data: Arguments

push argument N

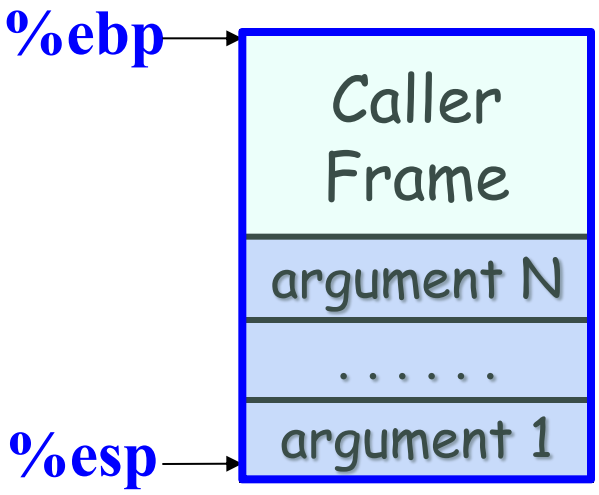


Passing Data: Arguments

push argument N

. . .

push argument 1



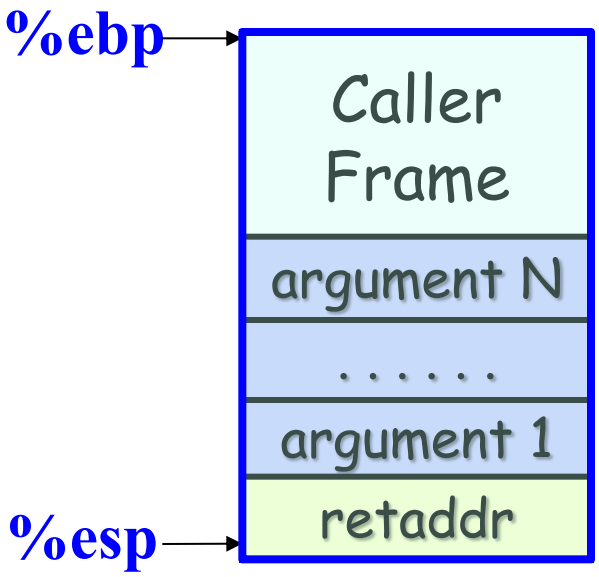
Passing Data: Arguments

push argument N

. . .

push argument 1

call callee



Passing Data: Arguments

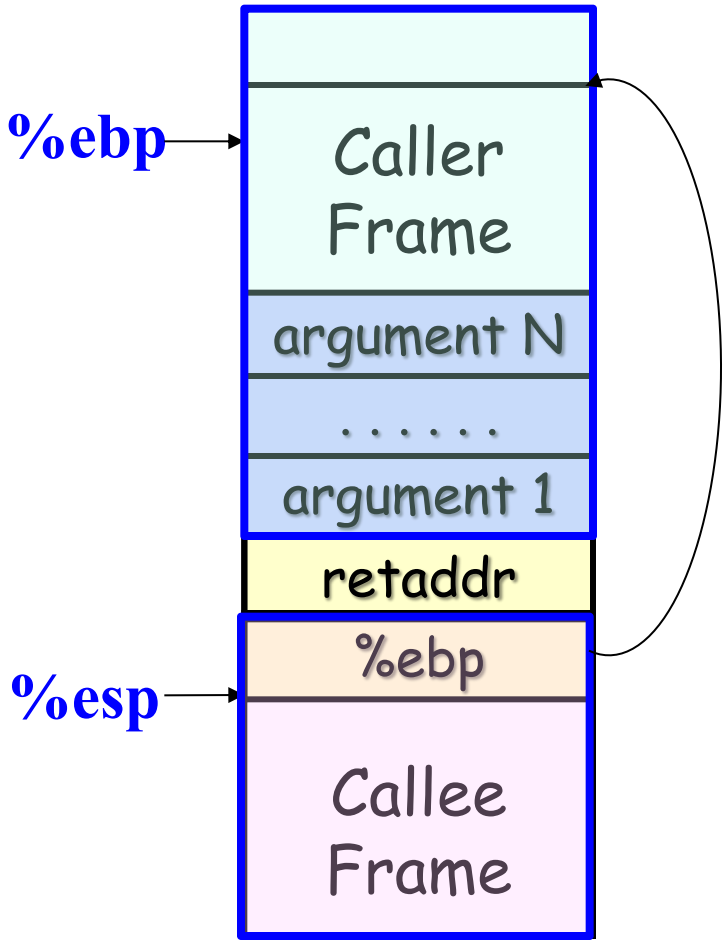
push argument N

. . .

push argument 1

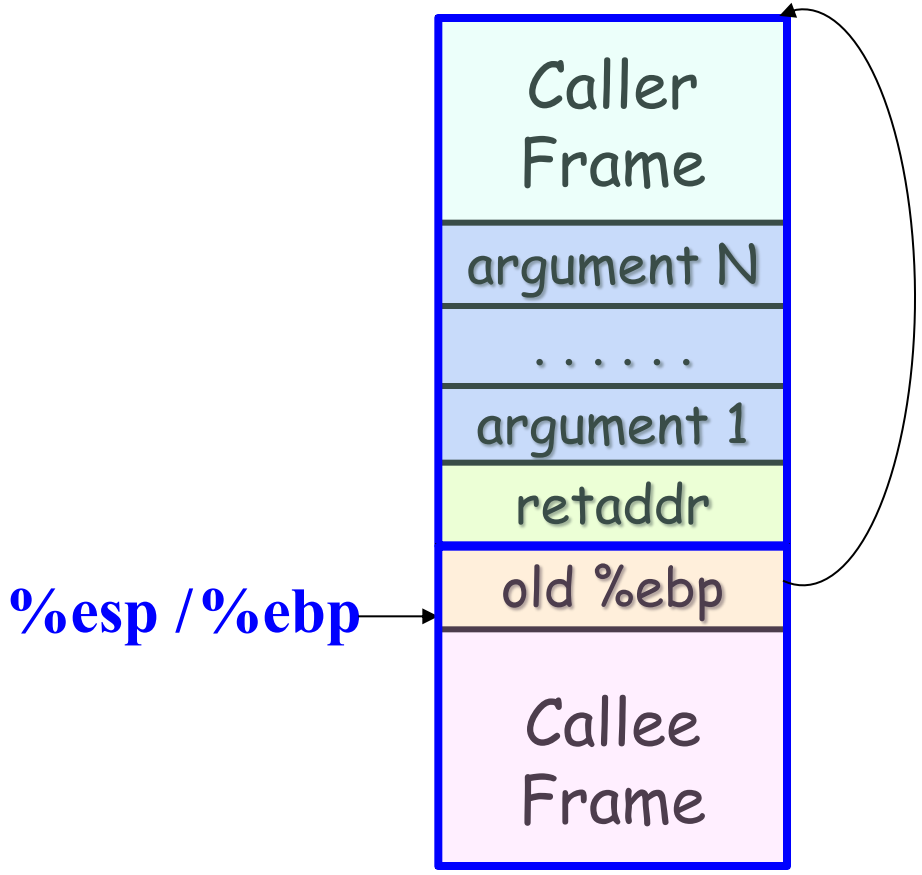
call callee

push %ebp



Passing Data: Arguments

```
push argument N
. . .
push argument 1
call callee
push %ebp
mov %esp, %ebp
. . .
```



Passing Data: Arguments

- X86 64位，传参数先用寄存器
  - %rdi，%rsi，%rdx，%rcx，%r8，%r9 用作函数**参数**，依次对应第1**参数**，第2**参数**....第6个参数（从左到右）
  - 剩余参数还和以前的算法一样，从右往左依次入栈
  - 例如func(p1, p2, p3, p4, p5, p6, p7, p8)
    - p1: %rdi, p2: %rsi, p3: %rdx, p4: %rcx, p5: %r8, p6: %r9
    - 栈：(低地址）←ret addr, p7, p8←（高地址）

Passing Data: Return Value

- Specific register to keep the return value
  - %eax / %rax is used to pass the result of callee to caller

课堂练习

- C函数incrprob有三个参数q, x, t，每个都可能是有符号数，也可能是无符号数，该函数主体为：
  - *t += x;
  - *q += *t;
- 基础知识：
  - 64位汇编传参顺序: rdi, rsi, rdx, rcx, r8, r9
  - movslq表示把32位数填充进64位数，并做符号扩展

1 incrprob:

2addl (%rdx), %edi

3movl %edi, (%rdx)

4movslq %edi, %rdi

5addq %rdi, (%rsi)

6ret

求三个参数的顺序和可能的类型，确定incrproc所有可能的函数原型(LP64)。

C TYPE	ILP32	LP64	LLP64	ILP64
char	8	8	8	8
short	16	16	16	16
float	32	32	32	32
double	64	64	64	64
int	32	32	32	64
long	32	64	64	64
long long	64	64	64	64
pointer	32	64	64	64

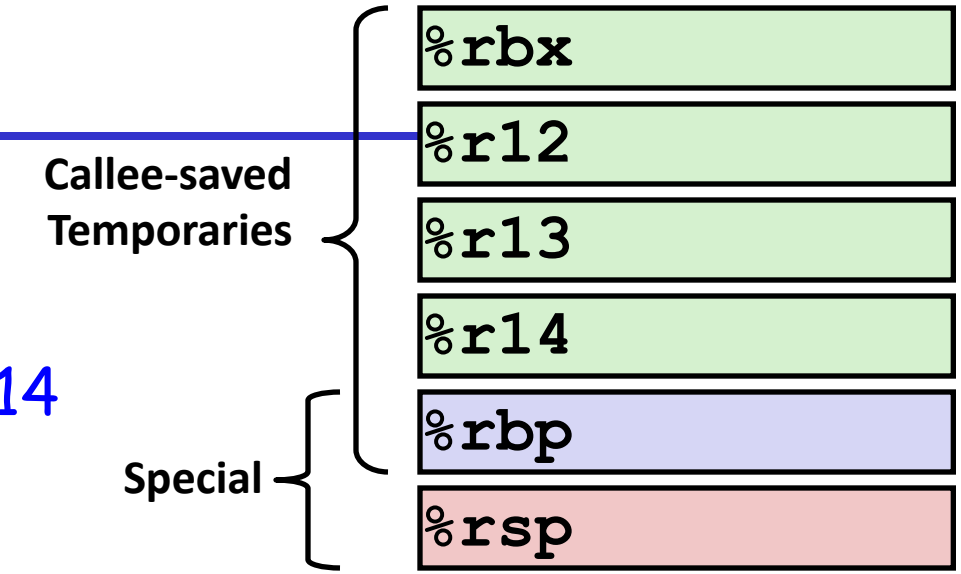
Procedure/Function Implementation

- ✓ Invoke callee: call (new instructions)
- ✓ Return to caller: ret (new instructions)
- ✓ Passing data: stack, register
- ? Registers: calling convention
- ? Local variable

Calling Convention

- Registers act as a single resource shared by all of the procedures
  - Only 1 procedure can be active
  - Partition registers between caller and callee (必须遵守惯例)
    - Caller-save register
    - Callee-save register
  - Only consider the registers used by the procedure

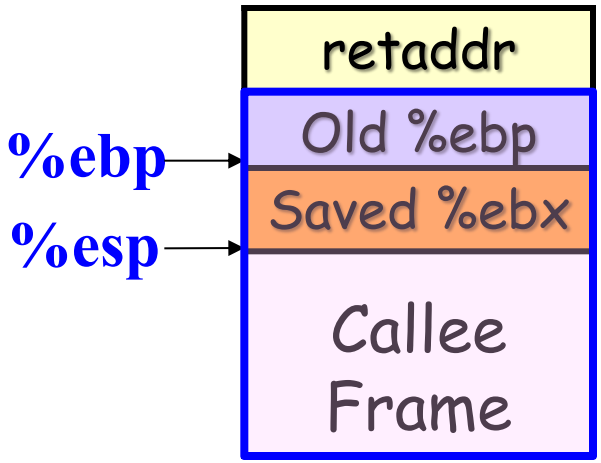
Calling Convention



- Callee-save registers
  - %rbx, %rbp, %r12~%r14
  - Saved by callee
  - Caller can use these registers freely
  - Callee must save them before using
  - Callee must restore them before return

Callee-save Registers

```
call callee
push %ebp
mov %esp, %ebp
push %ebx
. . .
```



Calling Convention

- Caller-save registers
  - 所有其他寄存器（除了%rsp不需要保存）
  - Saved by caller
  - Callee can use these registers freely
  - The contents in these registers may be changed after return
  - Caller must restore them if it tries to use them after calling

Caller-save Registers

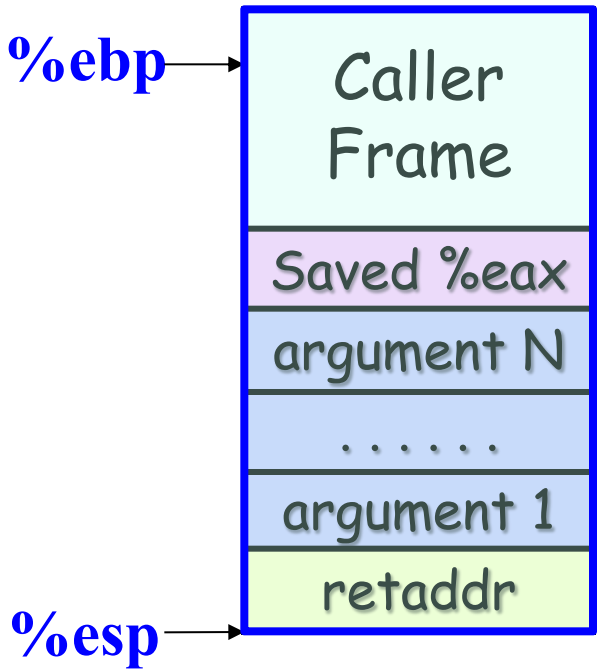
push %eax

push argument N

. . .

push argument 1

call callee



Procedure/Function Implementation

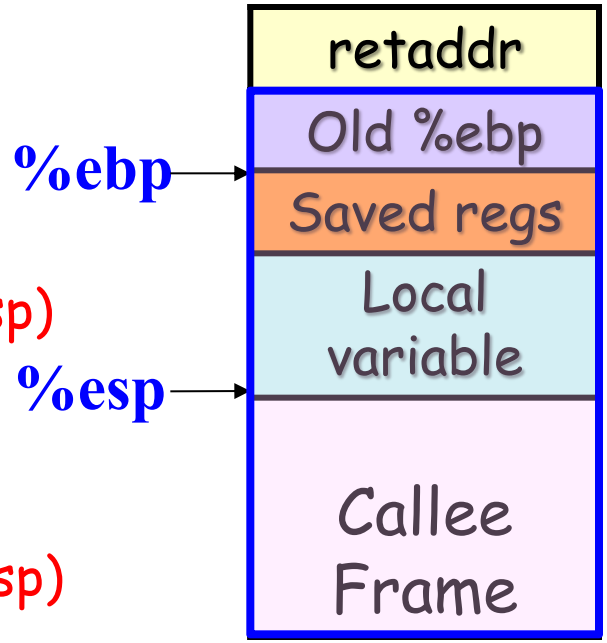
- ✓ Invoke callee: call (new instructions)
- ✓ Return to caller: ret (new instructions)
- ✓ Passing data: stack, register
- ✓ Registers: calling convention
- ? Local variable: stack

Local Variable

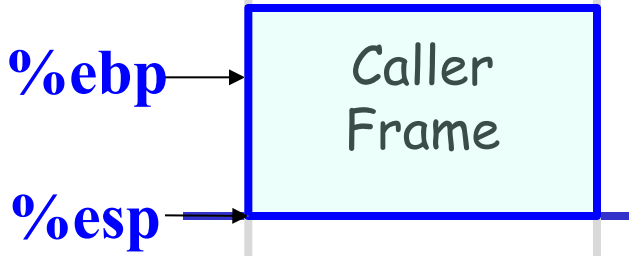
- Why not store local variables in registers ?
  - No enough registers
  - Array and structures (e.g., a[2])
  - Need address (e.g., &a)

Local Variable

- Allocation
  - Below saved regs or old %ebp
  - move/sub %esp, (e.g., subl $4, %esp)
- De-allocation
  - move/add %esp, (e.g., addl $4, %esp)
- Usage
  - Relative to %esp/%ebp, (e.g., movl %eax, 8(%esp))

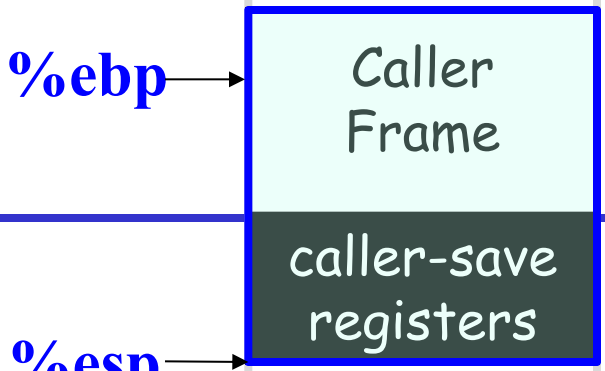


Put it Together



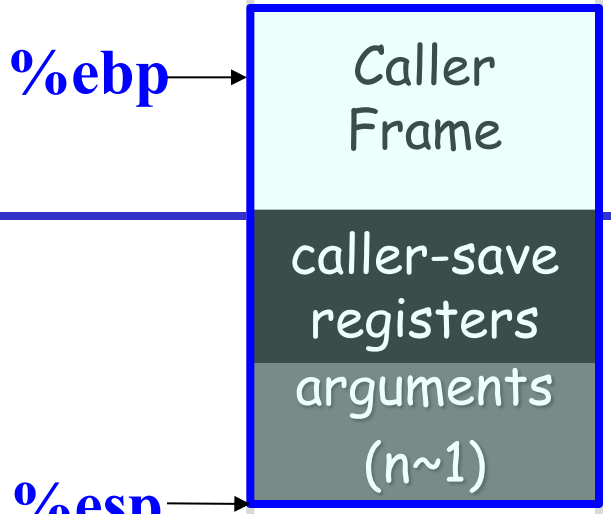
Put it Together

1. Save caller-save registers (%rbx, %rbp, %r12~15, %rsp以外的寄存器)



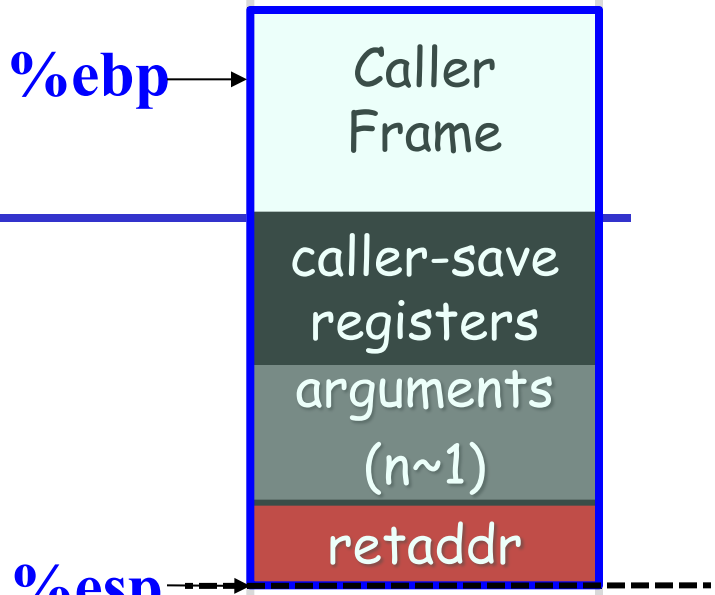
Put it Together

1. Save caller-save registers
2. Push actual arguments from right to left



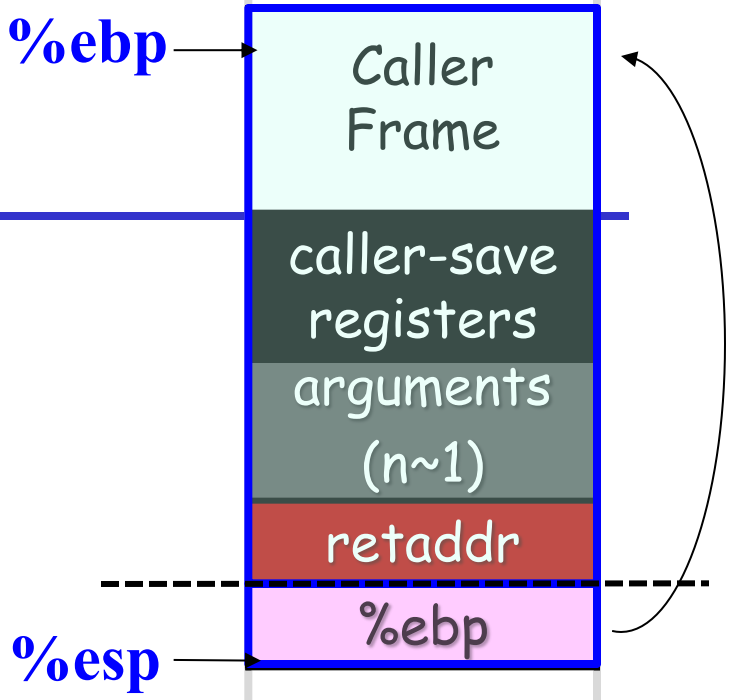
Put it Together

1. Save caller-save registers
2. Push actual arguments from right to left
3. Call instruction
   - Save return address
   - Transfer control to callee



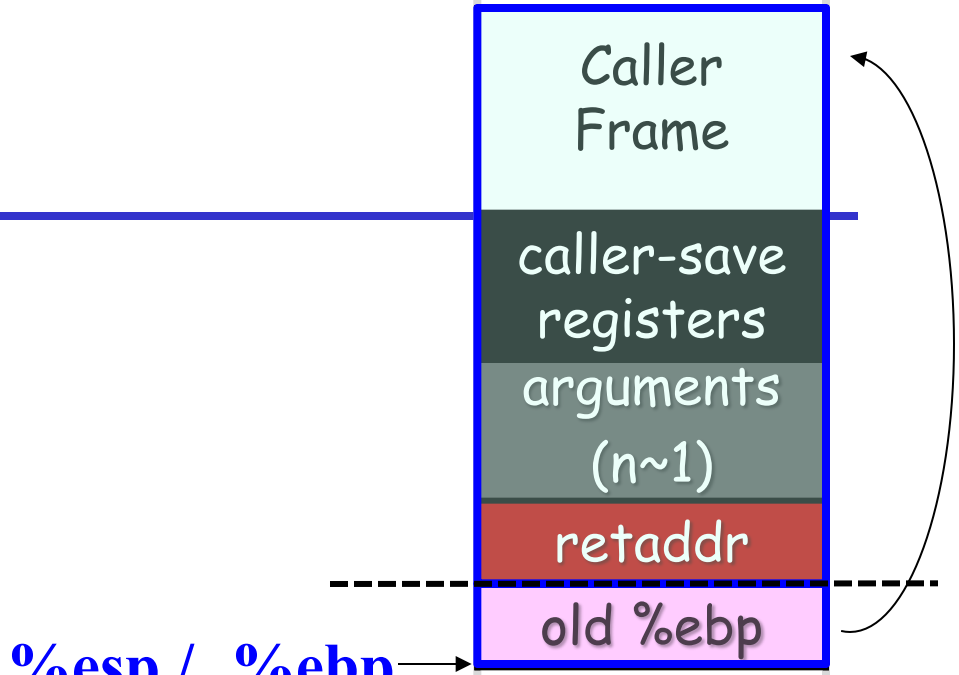
Put it Together

4. Save caller %ebp



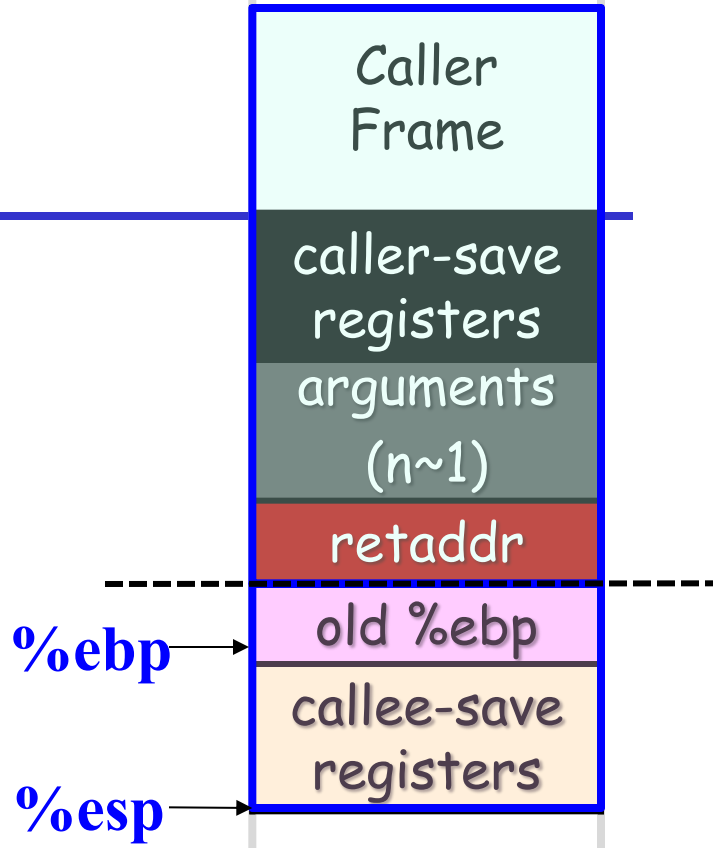
Put it Together

4. Save caller %ebp
5. Set callee %ebp



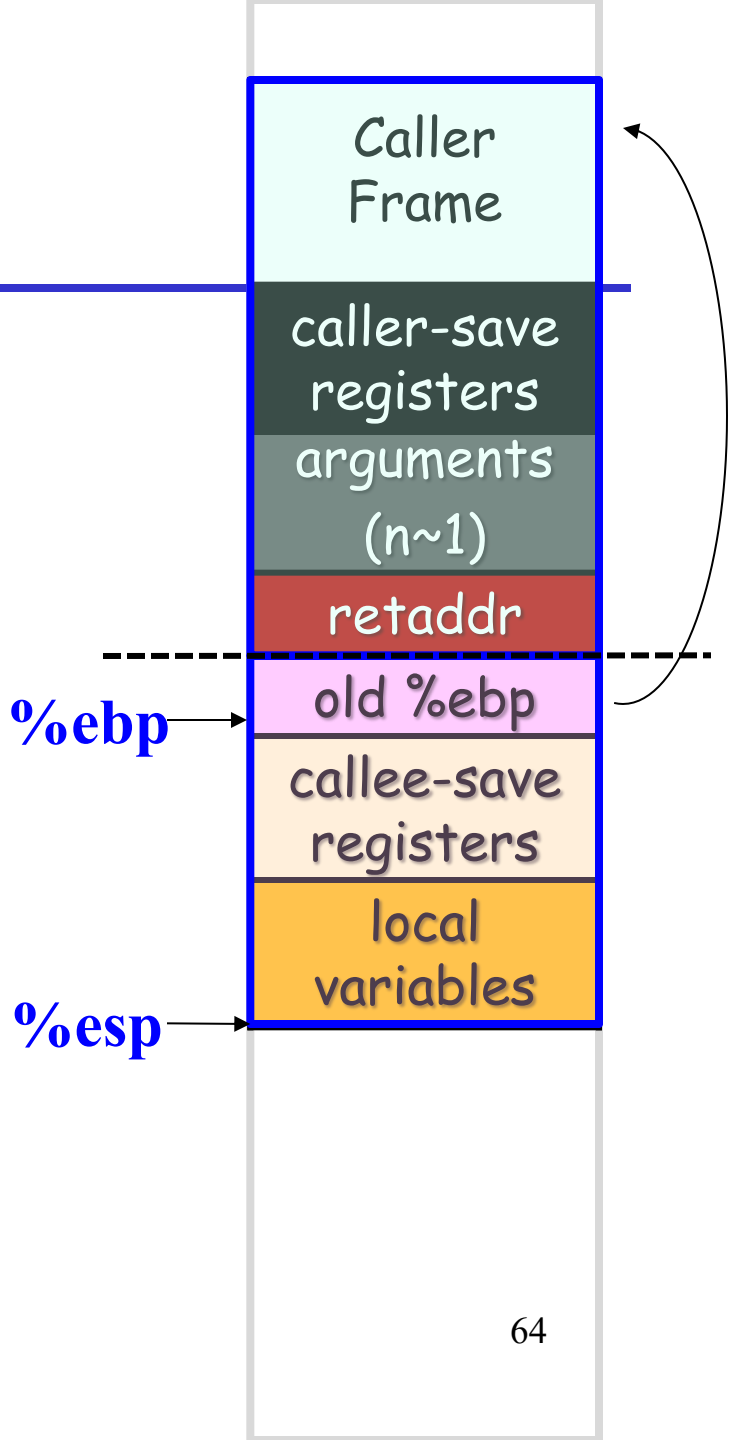
Put it Together

4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers (%rbx, %rbp, %r12~15)



Put it Together

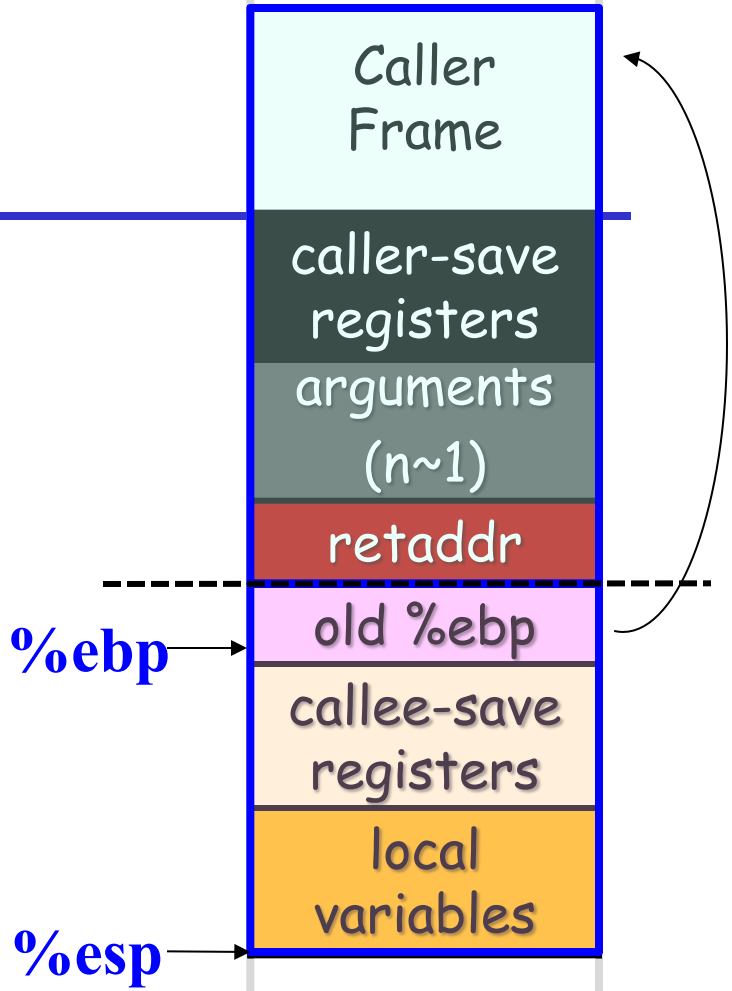
4. Save caller %ebp
5. Set callee %ebp
6. Save callee-save registers
   (%rbx, %rbp, %r12~15)
7. Allocate space for local variable



Put it Together

. . .

n-4. save return value in %eax

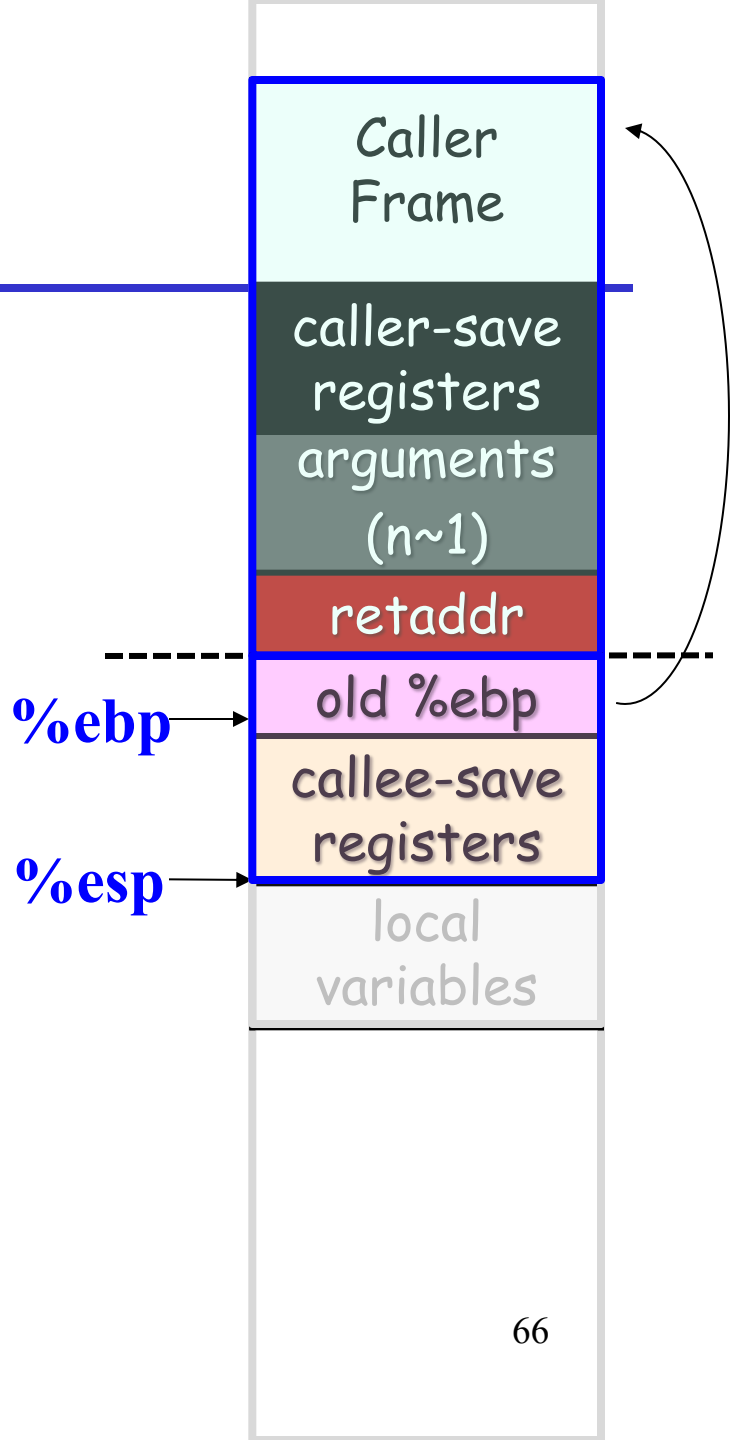


Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable



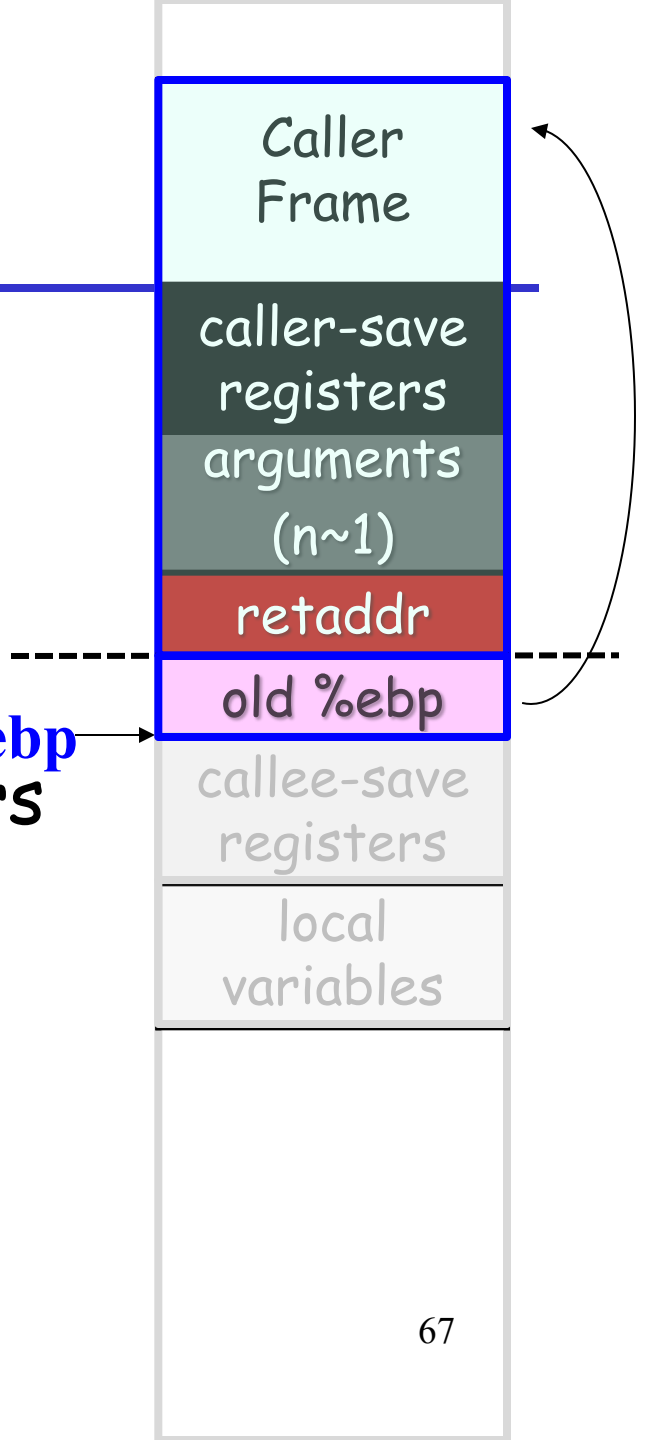
Put it Together

. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers



Put it Together

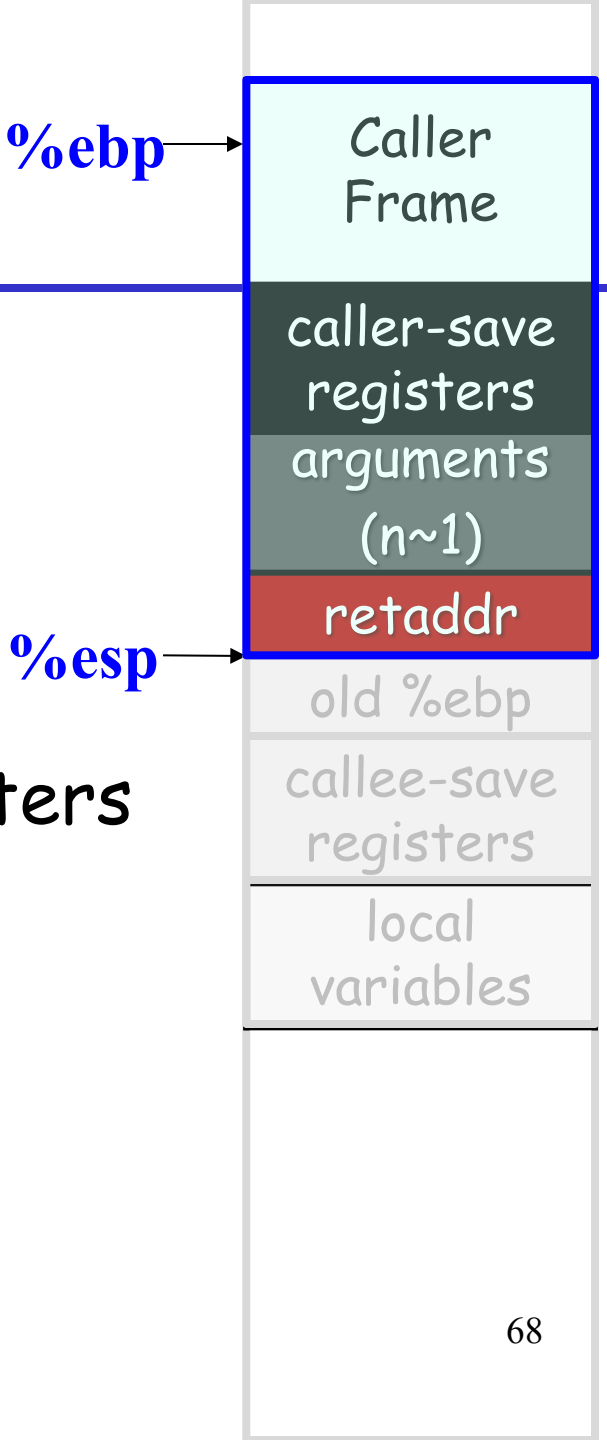
. . .

n-4. save return value in %eax

n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp



Put it Together

. . .

n-4. save return value in %eax

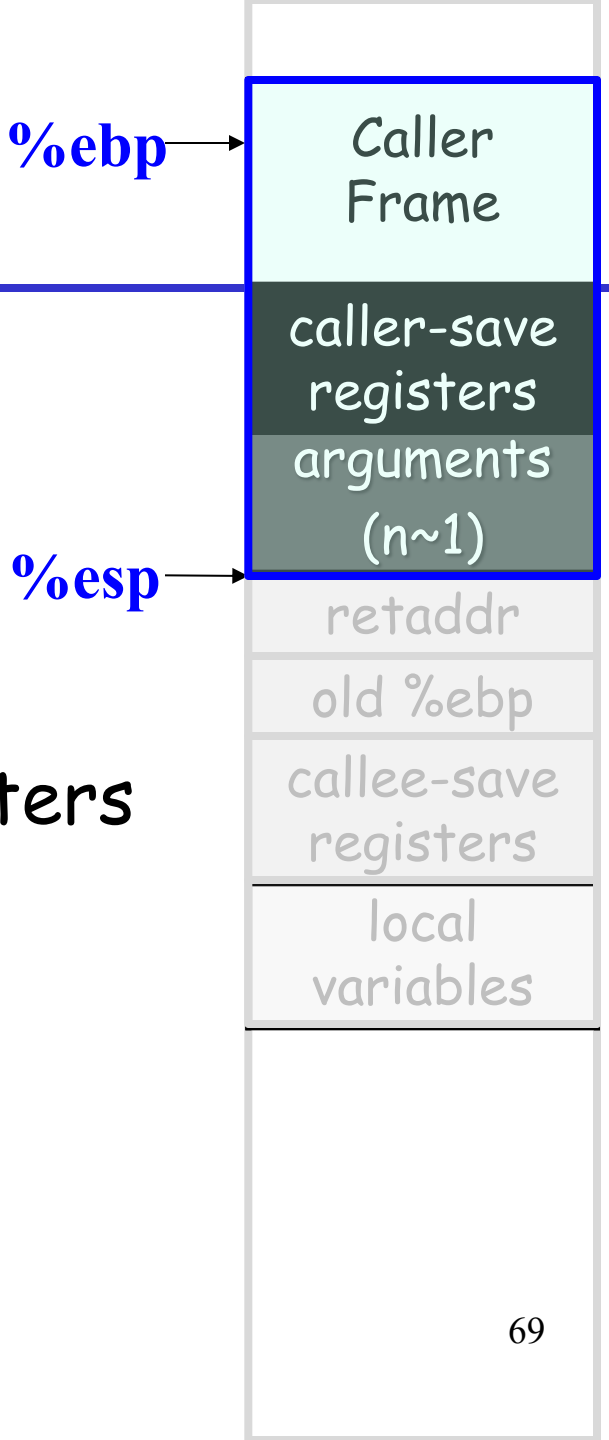
n-3. de-allocate local variable

n-2. Restore callee-save registers

n-1. Restore caller %ebp

n. Ret instruction

- pop return address
- Transfer control to caller



Example

```
1 int swap_add(int *xp, int *yp)
2 {
3         int x = *xp;
4         int y = *yp;
5
6         *xp = y;
7         *yp = x;
8         return x + y;
9 }
10
```

Example

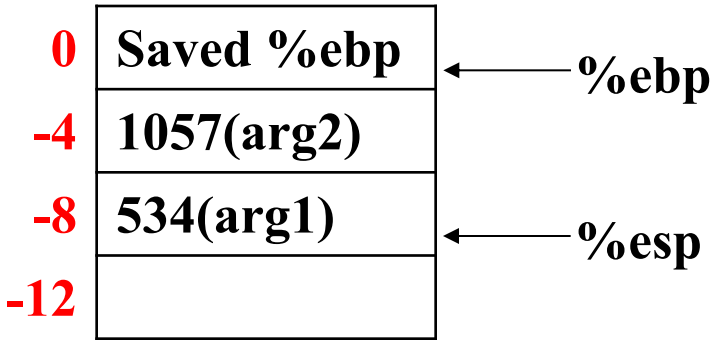
```
11 int caller()
12 {
13        int arg1 = 534;
14        int arg2 = 1057;
15        int sum = swap_add(&arg1, &arg2);
16        int diff = arg1 - arg2;
17
18        return sum * diff;
19 }
```

Example

Before

int sum = swap_add(&arg1, &arg2);

Stack frame for caller



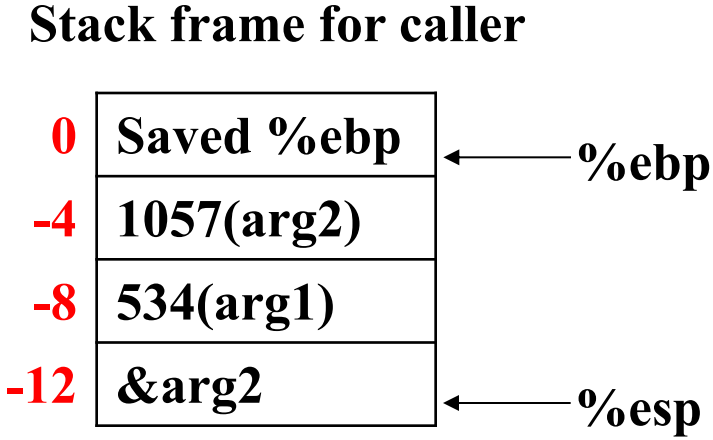
Parameter Passing

1 **leal -4(%ebp),%eax**

Compute &arg2

2 **pushl %eax**

Push &arg2



Parameter Passing

1 leal -4(%ebp),%eax

Compute &arg2

2 pushl %eax

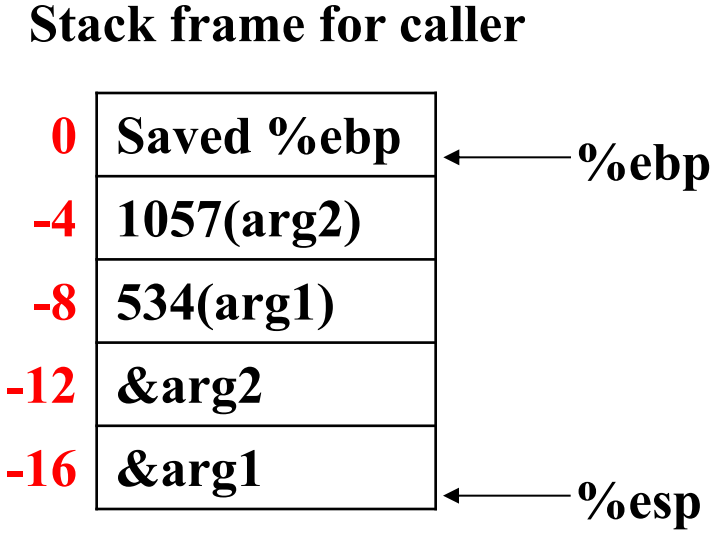
Push &arg2

3 leal -8(%ebp),%eax

Compute &arg1

4 pushl %eax

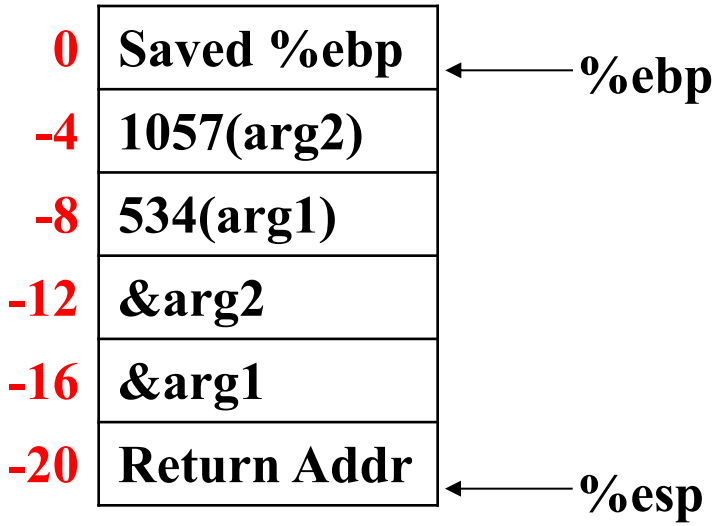
Push &arg1



Call Instruction

```
1   leal -4(%ebp),%eax
                    Compute &arg2
2   pushl %eax
                    Push &arg2
3   leal -8(%ebp),%eax
                    Compute &arg1
4   pushl %eax
                    Push &arg1
5    call swap_add
                    Call the swap_add function
```

Stack frame for caller

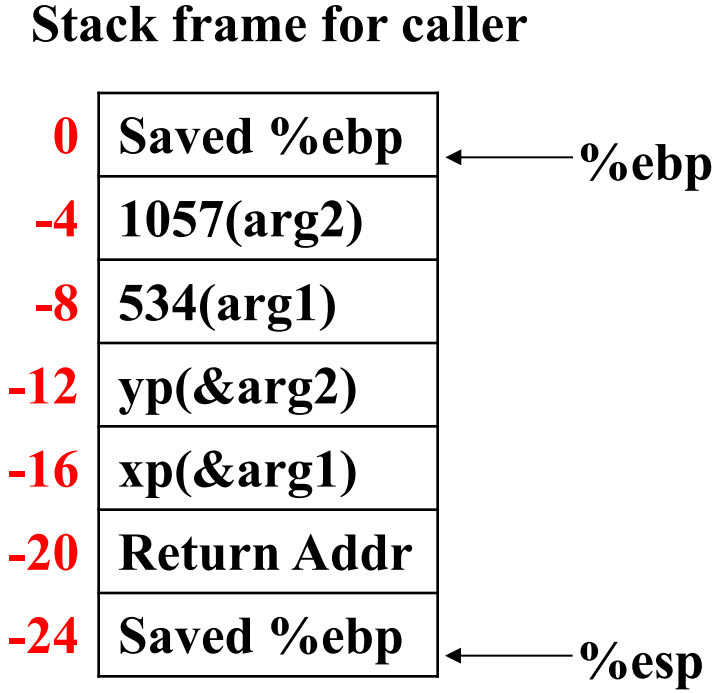


Setup code in swap_add

swap_add:

1 pushl %ebp

***Save old* %ebp**



Setup code in swap_add

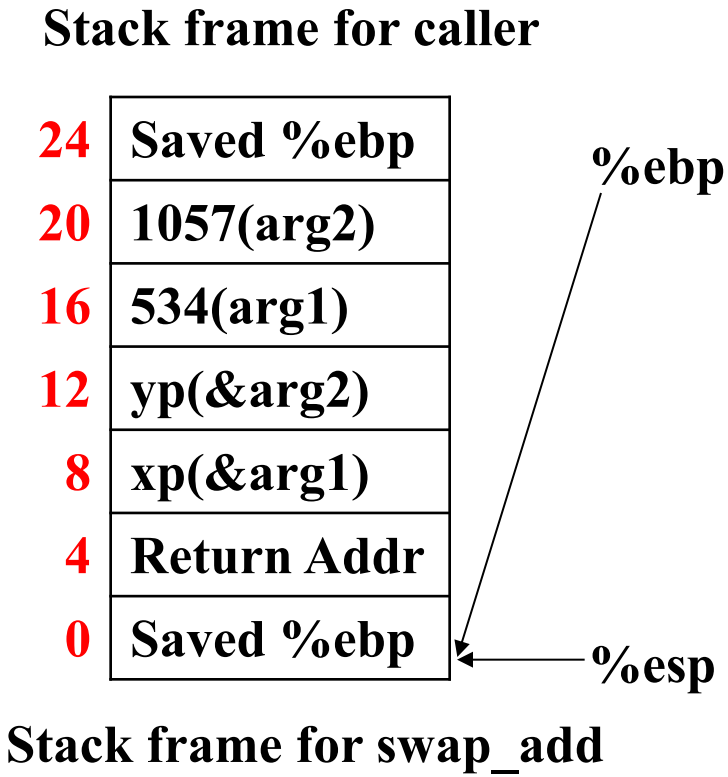
swap_add:

1 pushl %ebp

***Save old* %ebp**

2 movl %esp,%ebp

Set* %ebp *as frame pointer



Setup code in swap_add

swap_add:

1 pushl %ebp

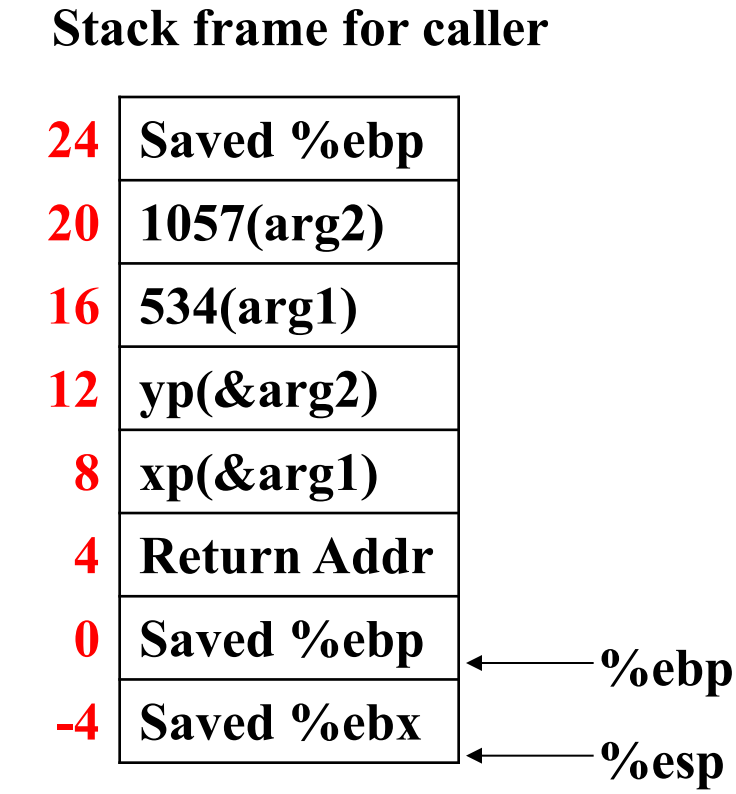
Save old %ebp

2 movl %esp,%ebp

Set %ebp *as frame pointer*

3 pushl %ebx

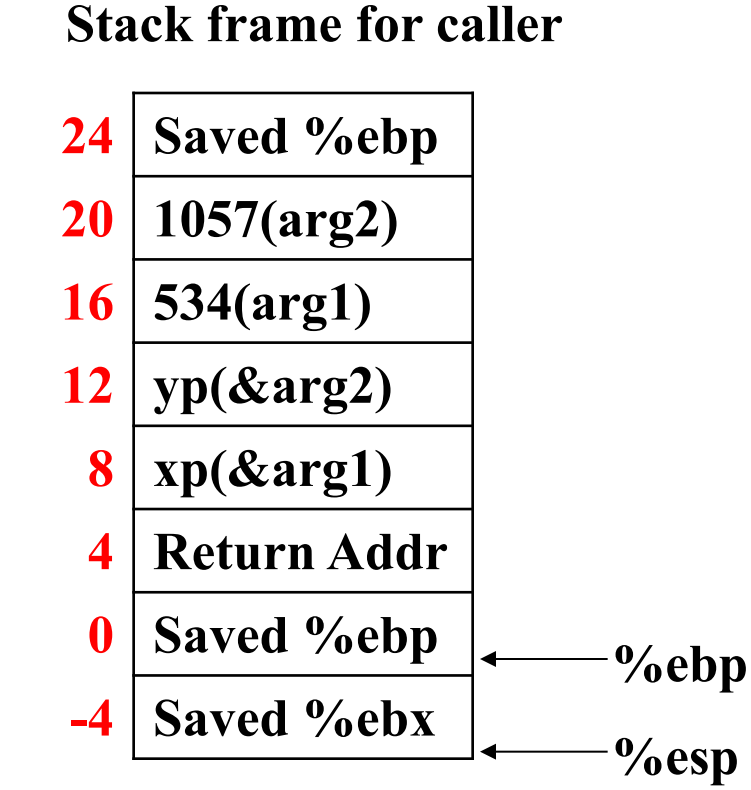
Save %ebx



Body code in swap_add

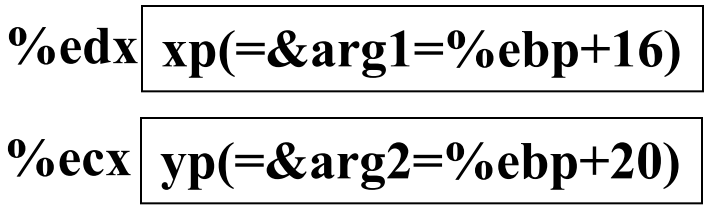
5 movl 8(%ebp),%edx

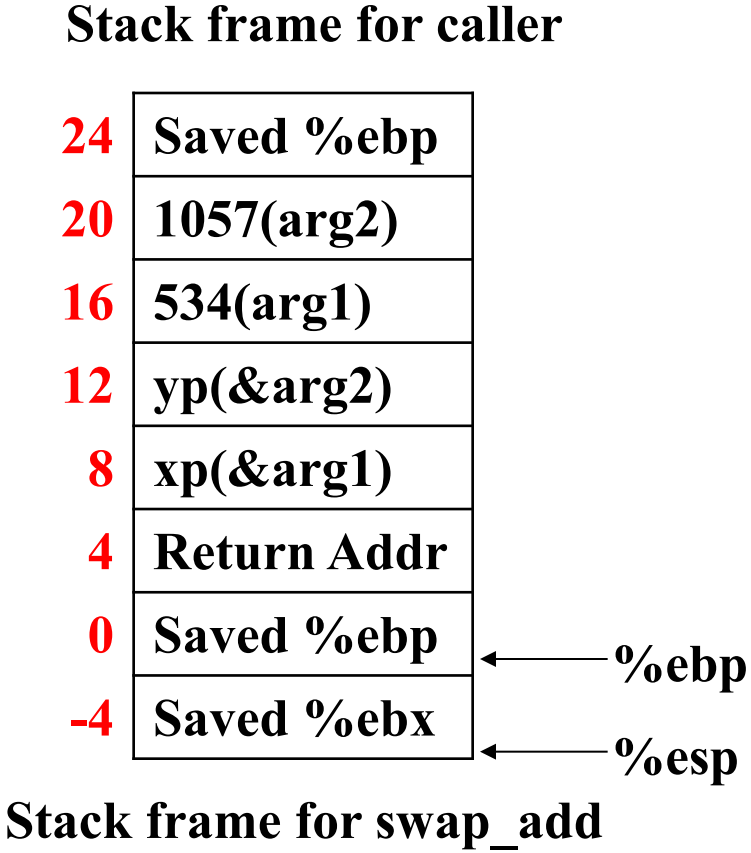
Get xp



Body code in swap_add

```
5   movl 8(%ebp),%edx
                        Get xp
6   movl 12(%ebp),%ecx
                        Get yp
```



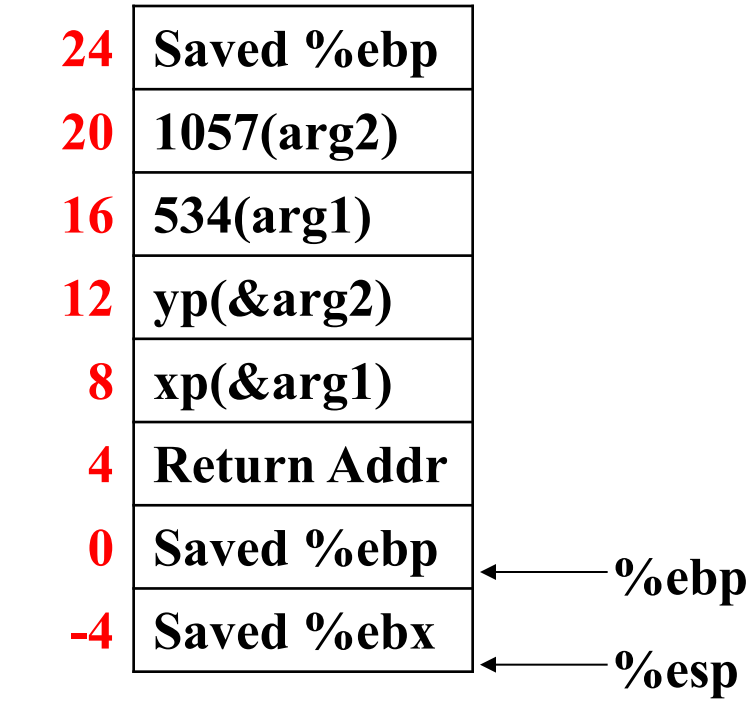


Body code in swap_add

```
5  movl 8(%ebp),%edx
                        Get xp
6  movl 12(%ebp),%ecx
                        Get yp
7  movl (%edx),%ebx
                        Get x
8  movl (%ecx),%eax
                        Get y
```

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	534
%eax	1057

Stack frame for caller

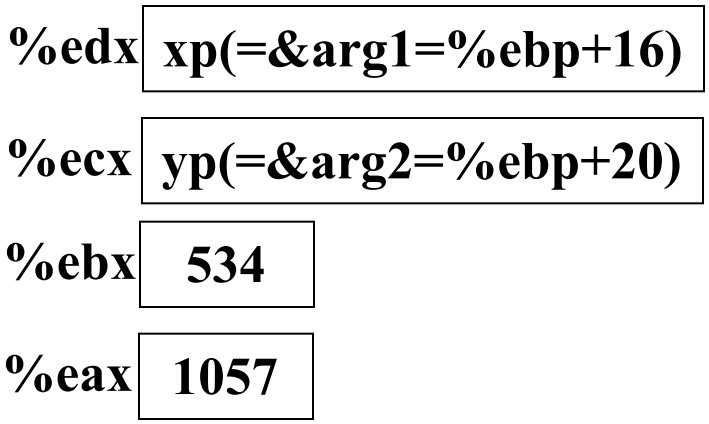


Stack frame for swap_add

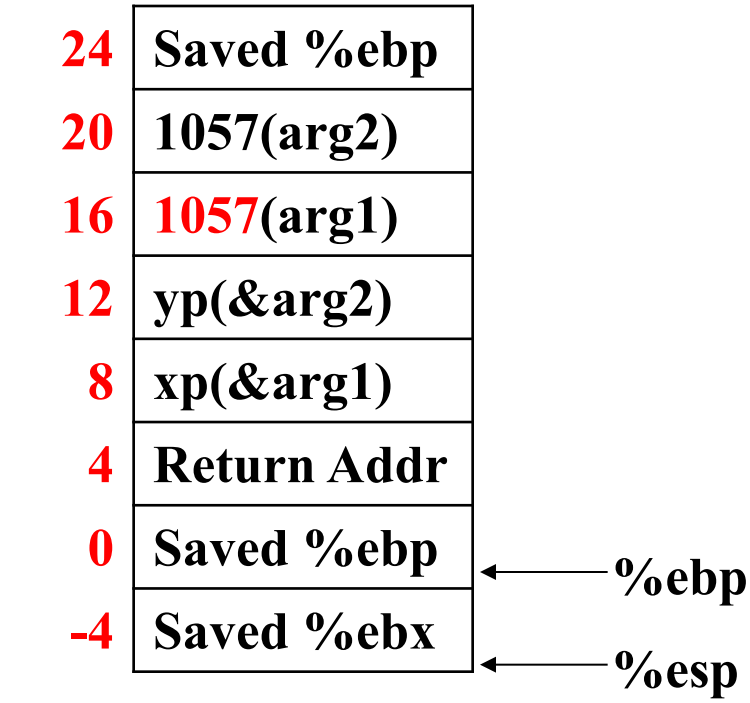
Body code in swap_add

9 movl %eax, (%edx)

Store y at *xp



Stack frame for caller



Stack frame for swap_add

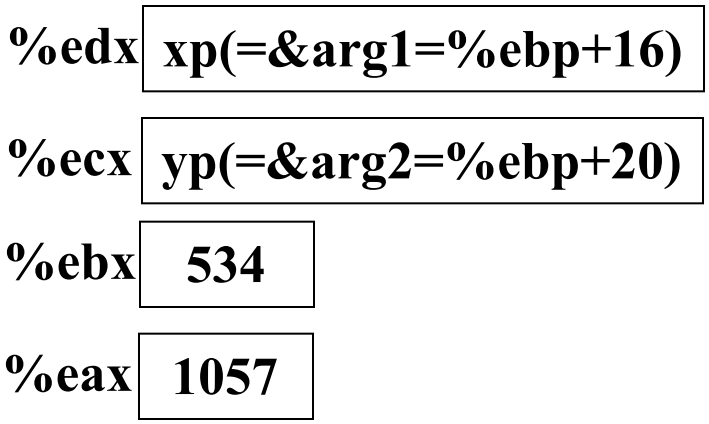
Body code in swap_add

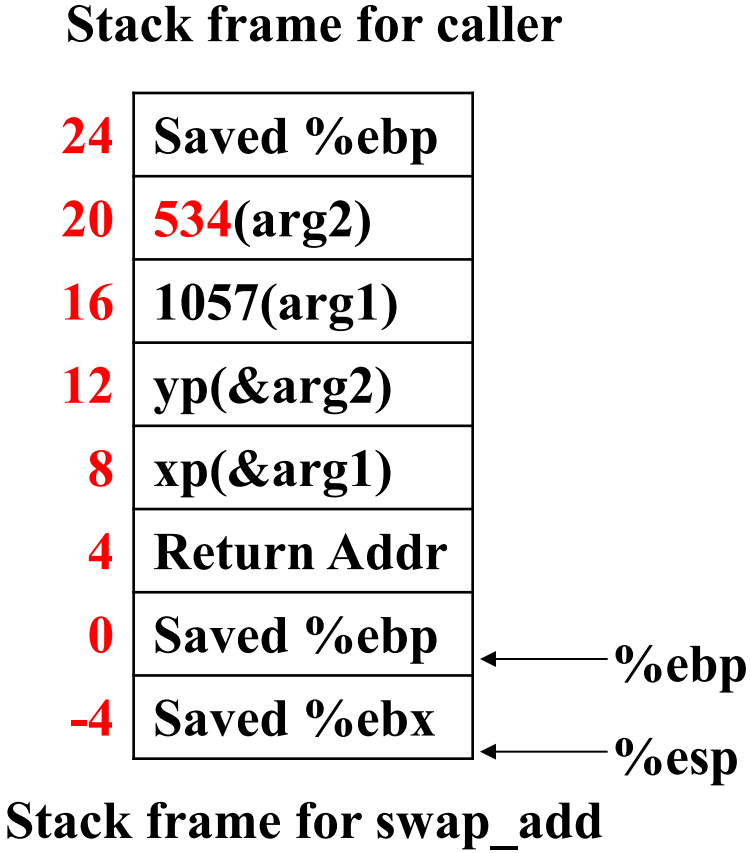
9 movl %eax, (%edx)

Store y at *xp

10 movl %ebx, (%ecx)

Store x at *yp



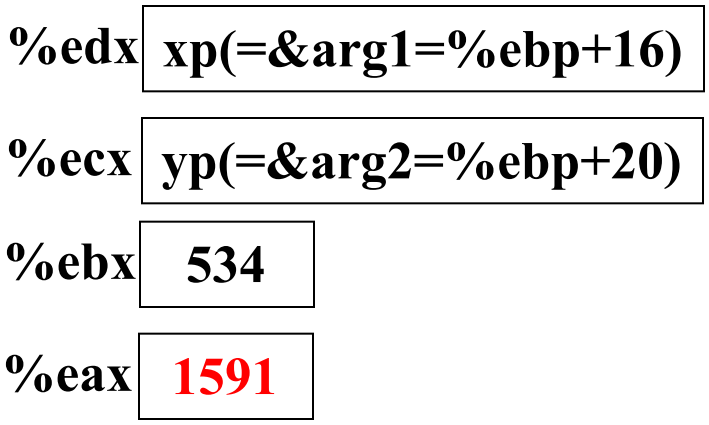


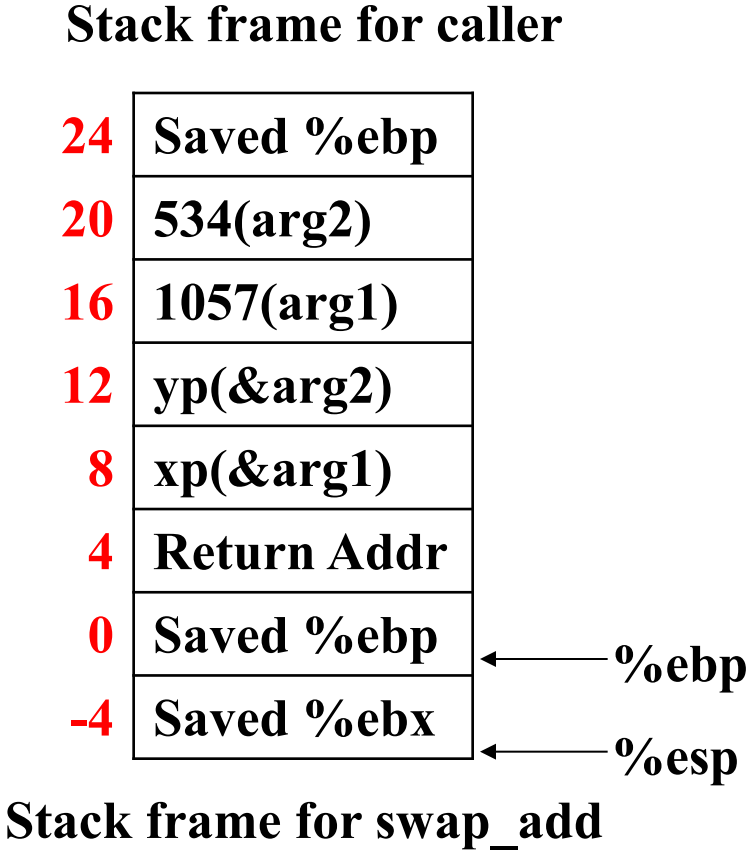
Body code in swap_add

```
9   movl %eax, (%edx)
```
*Store y at *xp*

```
10  movl %ebx, (%ecx)
```
*Store x at *yp*

```
11  addl %ebx,%eax
```
Set return value = x+y

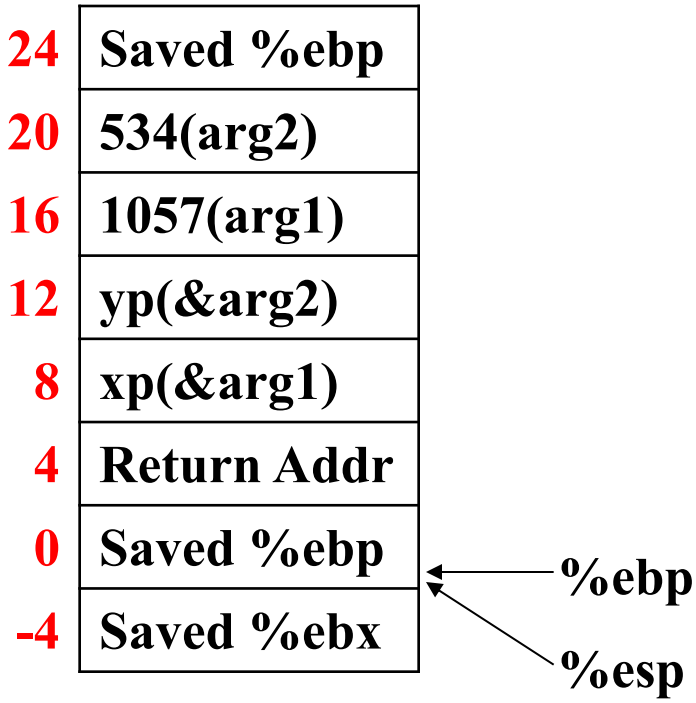




Finishing code in swap_add

12 popl %ebx *Restore* %ebx

Stack frame for caller

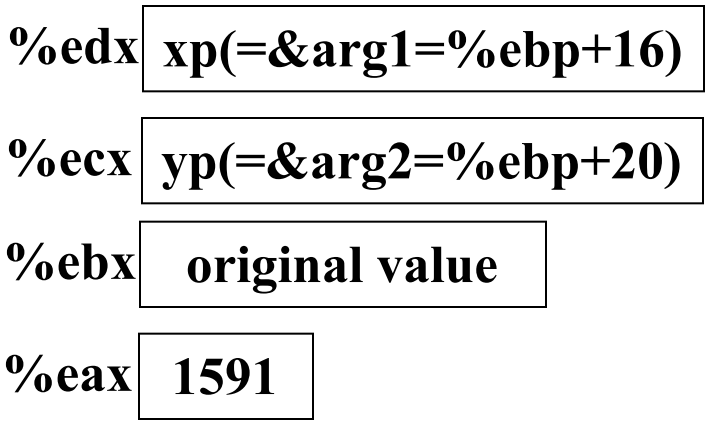


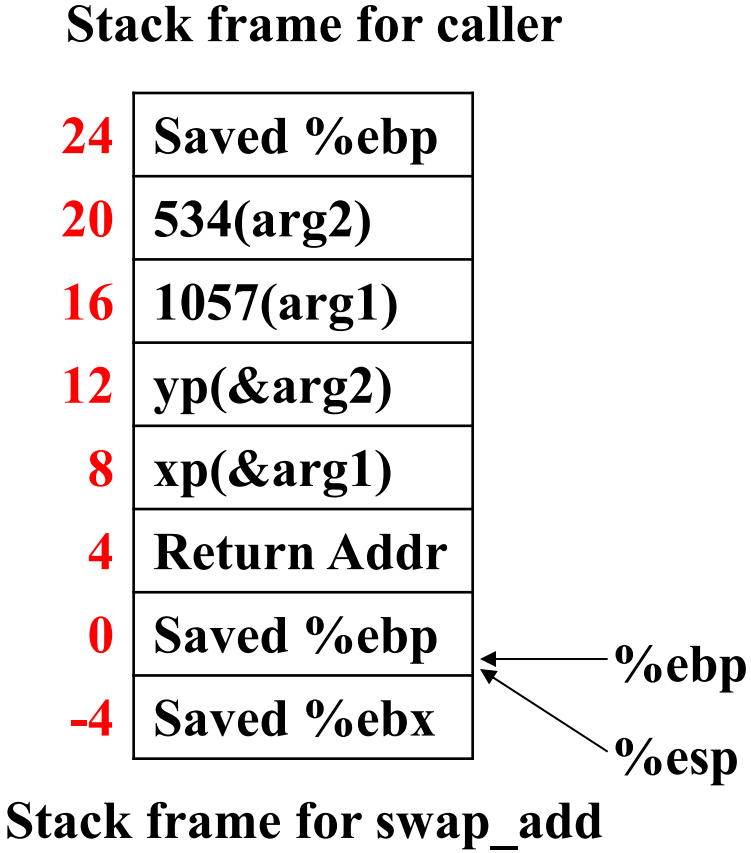
Stack frame for swap_add

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Finishing code in swap_add

```
12  popl %ebx        Restore %ebx
13  movl %ebp, %esp
                     Restore %esp
```

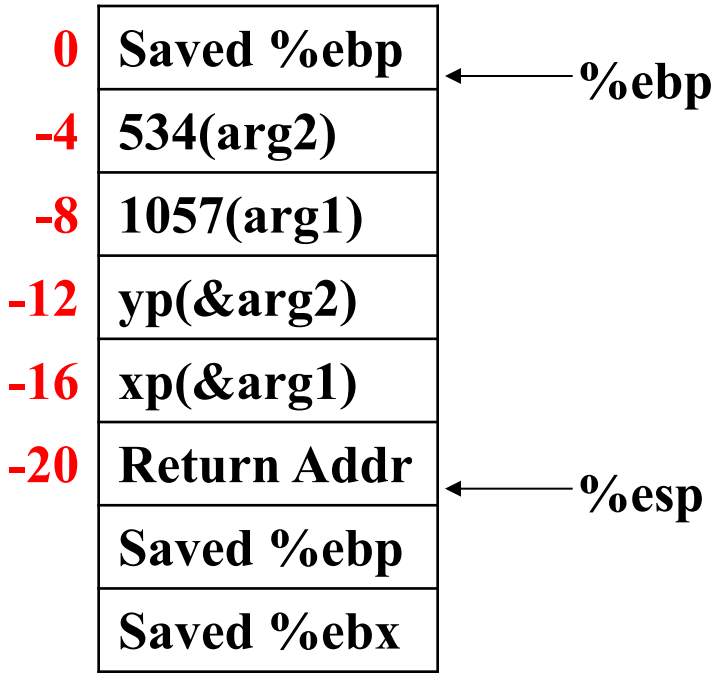




Finishing code in swap_add

```
12  popl %ebx       Restore %ebx
13  movl %ebp, %esp
                    Restore %esp
14  popl %ebp       Restore %ebp
```

Stack frame for caller



%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

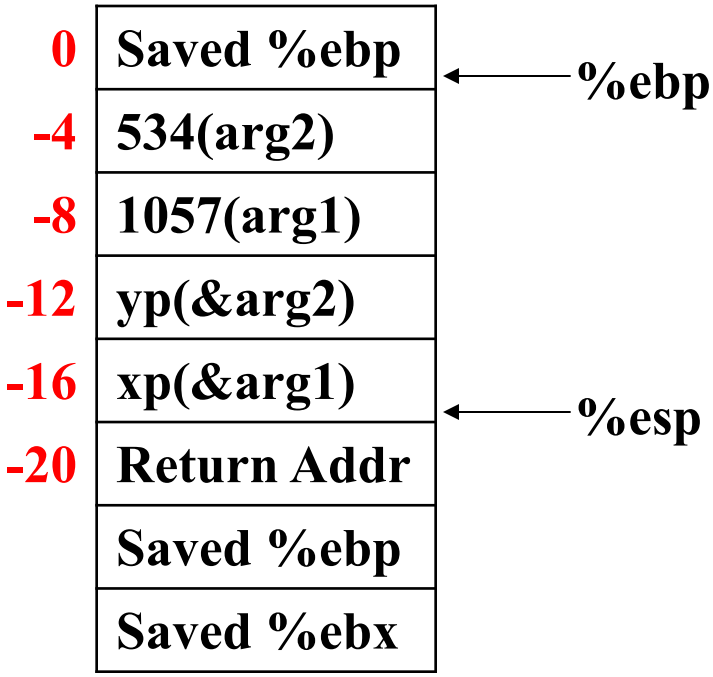
Finishing code in swap_add

```
12  popl %ebx       Restore %ebx
13  movl %ebp, %esp
                    Restore %esp
14  popl %ebp       Restore %ebp
15  ret             Return to caller
```

- **Call by value**

%edx	xp(=&arg1=%ebp+16)
%ecx	yp(=&arg2=%ebp+20)
%ebx	original value
%eax	1591

Stack frame for caller



主观题 10分

```
int proc(void) {
        int x, y;
        scanf("%x %x", &y, &x);
        return x-y;
}
```

GCC产生如下汇编代码

```
1 proc
2Pushl %ebp
3Movl  %esp, %ebp
4Subl  $40, %esp
5Leal   -4(%ebp), %eax
6Movl  %eax, 8(%esp)
7Leal  -8(%ebp), %eax
8Movl  %eax, 4(%esp)
9Movl  $.LC0, (%esp)；常量字符串"%x %x"的地址
10Call scanf
11Movl -4(%ebp), %eax
12  Subl -8(%ebp), %eax
13  Movl %ebp, %esp,
14  Popl %ebp
15 Ret
```

过程proc开始时，esp值为0x800040, ebp值为0x00060，scanf输入值为0x46和0x53, "%x %x"地址为0x300070.

A.第3行ebp的值被设为多少？

B.第4行esp的值被设为多少？

C.局部变量x和y的地址是多少？

D.画出scanf返回后的栈图

E.指出proc函数未使用的栈区的地址

作答

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    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; 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)	Type
%rdi	x	Argument
%rax	Return value	Return value

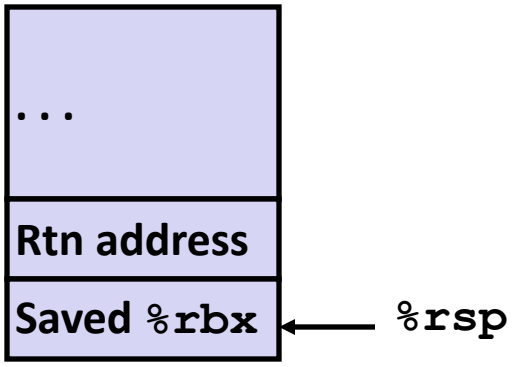
```
pcount_r:
  movl    $0, %eax
  testq   %rdi, %rdi
  je      .L6
  pushq   %rbx
  movq    %rdi, %rbx
  andl    $1, %ebx
  shrq    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

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);
}
```

Register	Use(s)	Type
%rdi	x	Argument

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



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);
}
```

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

Register	Use(s)	Type
%rdi	x >> 1	Rec. argument
%rbx	x & 1	Callee-saved

Recursive Function Call

```
/* 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    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Recursive call return value	

Recursive Function Result

```
/* 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    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

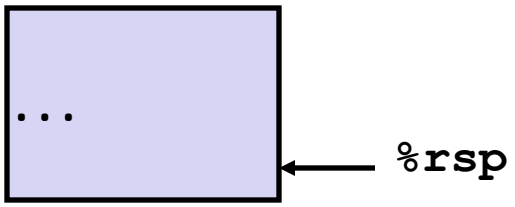
Register	Use(s)	Type
%rbx	x & 1	Callee-saved
%rax	Return value	

Recursive Function Completion

```
/* 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    %rdi
  call    pcount_r
  addq    %rbx, %rax
  popq    %rbx
.L6:
  rep; ret
```

Register	Use(s)	Type
%rax	Return value	Return value



主观题 10分

设置

一个具有通用结构的C函数如下：

```
long rfun (unsigned long x) {
    if ( _____ )
      return _____;
    unsigned long nx =_____;
    long rv = rfun(nx);
    return _____;
}
```

请填写C语言代码中缺失的表达式；rfun存储在被调用者保存寄存器%rbx中的值是_____。

GCC产生的汇编代码如下：

```
long rfun (unsigned long x)
x in %rdi
rfun:
  pushq     %rbx
  movq      %rdi, %rbx
  movl      $0, %eax
  testq     %rdi, %rdi
  je        .L2
  shrq      $2, $rdi
  call rfun
  addq      %rbx, %rax
.L2
  popq      %rbx
  ret
```

作答

课堂练习（课后）

- 斐波那契数列公式
  - F(0)=0, F(1)=1,
  - F(n)=F(n-1)+F(n-2)

- 请写出汇编函数求解斐波那契数列的第n项，然后画出主函数调用F(4) 的过程中，栈的状态（要求标注每个位置所有出现过的值）
  - 假设32位机，函数调用过程使用栈进行传参，函数F()求解过程中只保存了ebp的值，没有用到局部变量。
  - 在图中画出Stack Frame Chain

```
# int F(int n)
F:
    pushl %ebp          # 保存旧的 ebp
    movl  %esp, %ebp    # 建立新栈帧
    movl  8(%ebp), %eax # 取参数 n (参数位于 ebp+8)
    cmpl  $1, %eax
    jg    L_recur       # 如果 n > 1 则递归
    # base case: F(0)=0, F(1)=1
    movl  8(%ebp), %eax
    jmp   L_end

L_recur:
    movl  8(%ebp), %eax
    subl  $1, %eax
    pushl %eax          # 参数 n-1
    call  F
    movl  %eax, %ebx    # 保存 F(n-1)
    movl  8(%ebp), %eax
    subl  $2, %eax
    pushl %eax          # 参数 n-2
    call  F
    addl  %ebx, %eax    # eax = F(n-1) + F(n-2)

L_end:
    movl  %ebp, %esp
    popl  %ebp
    ret
```