练习 5：实现函数调用堆栈跟踪函数（需要编程）。

我们需要在 lab1 中实现 kdebug.c 中的函数 print_stackframe，可以通过函数 print_stackframe 来跟踪函数调用堆栈中记录的返回地址。如果能够正确实现此函数，可在 lab1 中执行 make qemu 后，在 qemu 模拟器中得到类似如下的输出：

```
⋮
ebp:0x00007b28 eip:0x00100992 args:0x00010094 0x00010094
0x00007b58 0x00100096
    kern/debug/kdebug.c:305: print_stackframe+22
ebp:0x00007b38 eip:0x00100c79 args:0x00000000 0x00000000
0x00000000 0x00007ba8
    kern/debug/kmonitor.c:125: mon_backtrace+10
ebp:0x00007b58 eip:0x00100096 args:0x00000000 0x00007b80
0xffff0000 0x00007b84
    kern/init/init.c:48: grade_backtrace2+33
ebp:0x00007b78 eip:0x001000bf args:0x00000000 0xffff0000
0x00007ba4 0x00000029
    kern/init/init.c:53: grade_backtrace1+38
ebp:0x00007b98 eip:0x001000dd args:0x00000000 0x00100000
0xffff0000 0x0000001d
    kern/init/init.c:58: grade_backtrace0+23
ebp:0x00007bb8 eip:0x00100102 args:0x0010353c 0x00103520
0x00001308 0x00000000
    kern/init/init.c:63: grade_backtrace+34
ebp:0x00007be8 eip:0x00100059 args:0x00000000 0x00000000
0x00000000 0x00007c53
    kern/init/init.c:28: kern_init+88
ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e
0x64e4d08e 0xfa7502a8
    <unknow>: -- 0x00007d72 --
⋮
```

请完成实验，看看输出是否与上述显示大致一致，并解释最后一行各个数值的含义。

提示：可阅读 3.3.1 小节“函数堆栈”，了解编译器如何建立函数调用关系的。在完成 lab1 编译后，查看 lab1/obj/bootblock.asm，了解 bootloader 源码与机器码的语句和地址等的对应关系；查看 lab1/obj/kernel.asm，了解 ucore OS 源码与机器码的语句和地址等的对应关系。

要求完成函数 kern/debug/kdebug.c::print_stackframe 的实现，提交改进后源代码包(可以编译执行)，并在实验报告中简要说明实现过程，并写出对上述问题的回答。

补充材料

显示完整的栈结构需要解析内核文件中的调试符号，这较为复杂和繁琐。代码中有一些辅助函数可以使用。例如，可以通过调用 print_debuginfo 函数完成查找对应函数名并打印至屏幕的功能。具体可以参见 kdebug.c 代码中的注释。

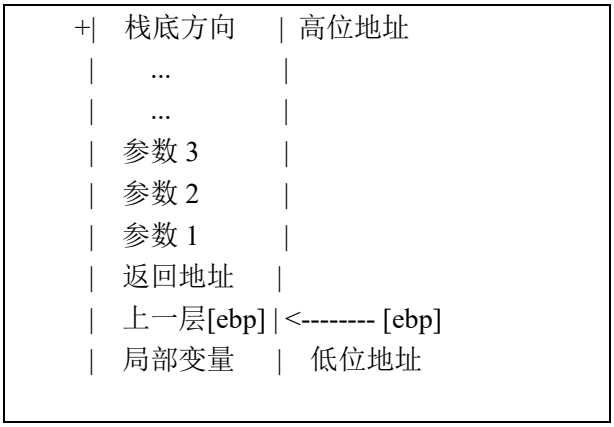
答：

（1）函数堆栈[1]

栈是一个很重要的编程概念（编译课和程序设计课都讲过相关内容），与编译器和编程语言有紧密的联系。理解调用栈最重要的两点是：栈的结构，EBP 寄存器的作用。一个函数调用动作可分解为：零到多个 PUSH 指令（用于参数入栈），一个 CALL 指令。CALL 指令内部其实还暗含了一个将返回地址（即 CALL 指令下一条指令的地址）压栈的动作（由硬件完成）。几乎所有本地编译器都会在每个函数体之前插入类似如下的汇编指令：

```
pushl   %ebp
movl   %esp , %ebp
```

这样在程序执行到一个函数的实际指令前，已经有以下数据顺序入栈：参数、返回地址、ebp 寄存器。由此得到类似如下的栈结构（参数入栈顺序跟调用方式有关，这里以 C 语言默认的 CDECL 为例）：



函数调用栈结构

这两条汇编指令的含义是：首先将 ebp 寄存器入栈，然后将栈顶指针 esp 赋值给 ebp。“mov ebp esp”这条指令表面上看是用 esp 覆盖 ebp 原来的值，其实不然。因为给 ebp 赋值之前，原 ebp 值已经被压栈（位于栈顶），而新的 ebp 又恰恰指向栈顶。此时 ebp 寄存器就已经处于一个非常重要的地位，该寄存器中存储着栈中的一个地址（原 ebp 入栈后的栈顶），从该地址为基准，向上（栈底方向）能获取返回地址、参数值，向下（栈顶方向）能获取函数局部变量值，而该地址处又存储着上一层函数调用时的 ebp 值。

一般而言，ss:[ebp+4]处为返回地址，ss:[ebp+8]处为第一个参数值（最后一个入栈的参数值，此处假设其占用 4 字节内存），ss:[ebp-4]处为第一个局部变量，ss:[ebp]处为上一层 ebp 值。由于 ebp 中的地址处总是“上一层函数调用时的 ebp 值”，而在每一层函数调用中，都能通过当时的 ebp 值“向上（栈底方向）”能获取返回地址、参数值，“向下（栈顶方向）”能获取函数局部变量值。如此形成递归，直至到达栈底。这就是函数调用栈。

举一个实际的例子[2]查看 ebp 与 esp 两个寄存器如何构建出完整的函数栈（其中 leave 等同于 movl %ebp, %esp，popl %ebp 两条指令）：

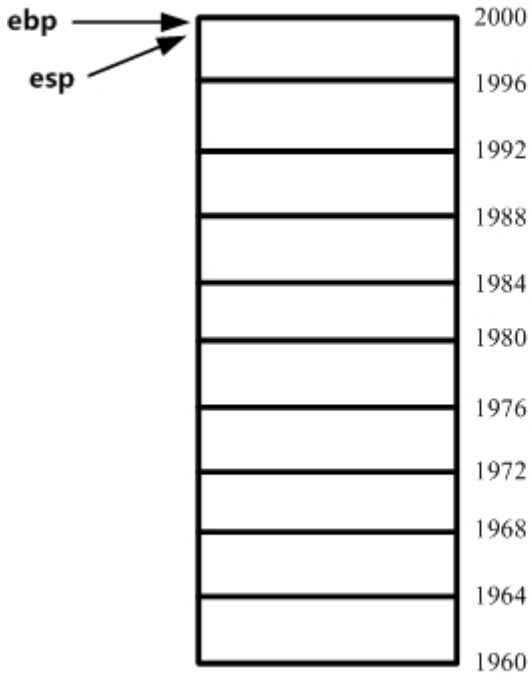
---

[1] 陈渝, 向勇. 操作系统实验指导[M]. 北京, 清华大学出版社, 2013:57-58.

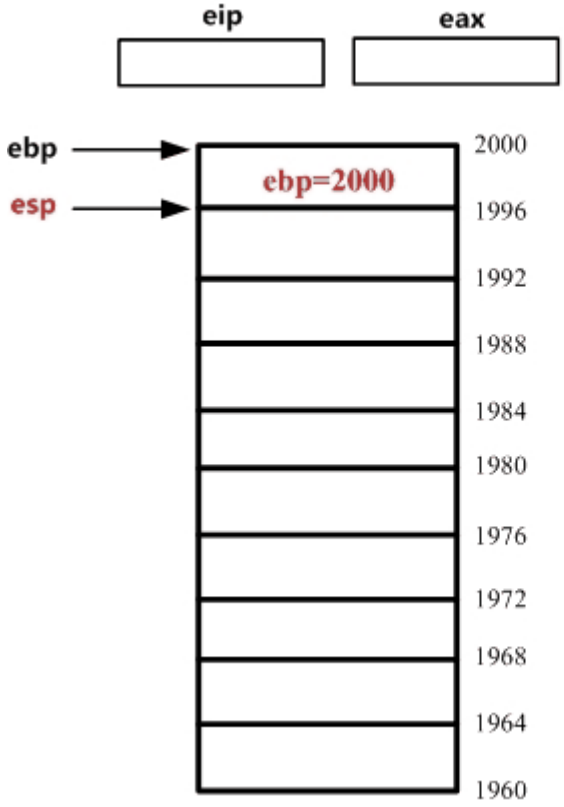
[2] nlskyfree. ucore lab1[OL]. https://www.jianshu.com/p/969e1bdcb471, 2018-07-13.

```c
int g(int x) {
    return x + 10;
}
int f(int x) {
    return g(x);
}
int main(void) {
    return f(20) + 8;
}
```

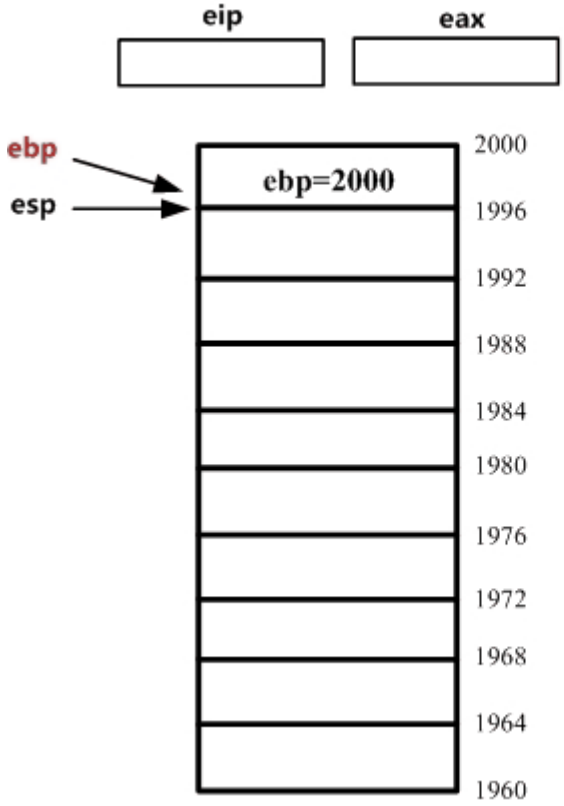
函数调用过程的示意图如下：



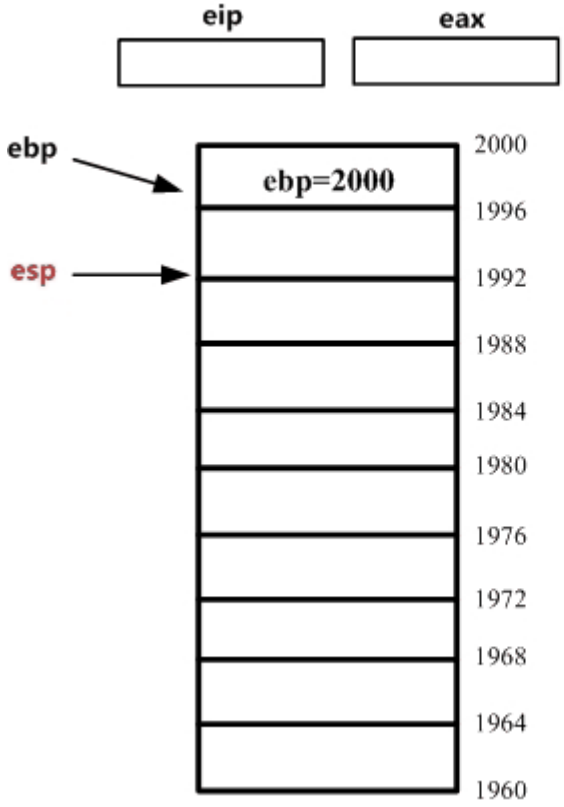

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
24          leave
25          ret
```



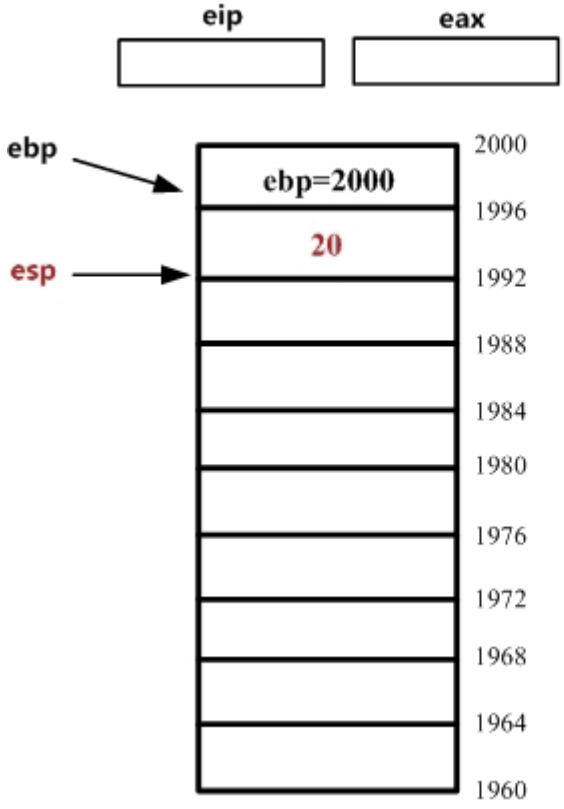

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



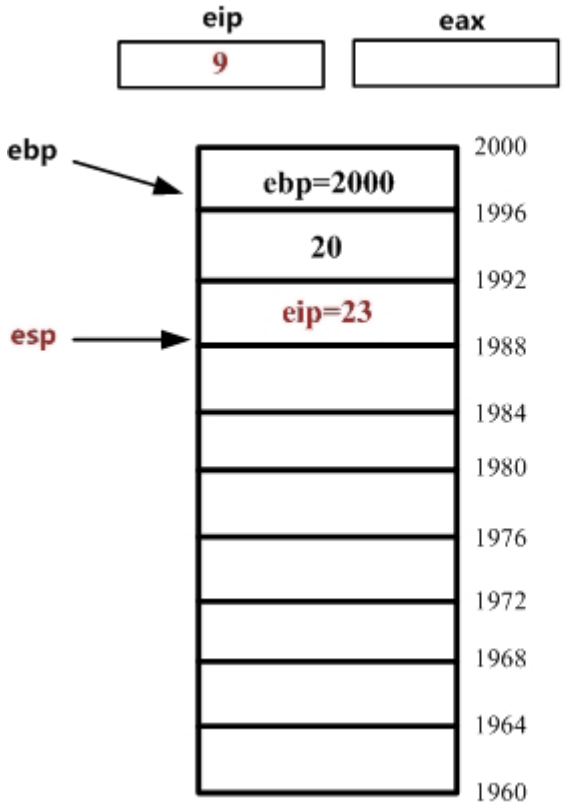

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



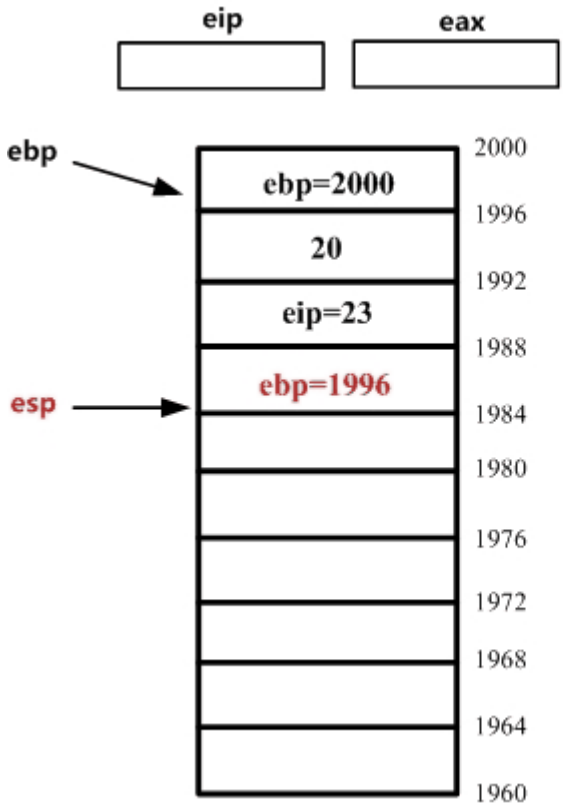

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



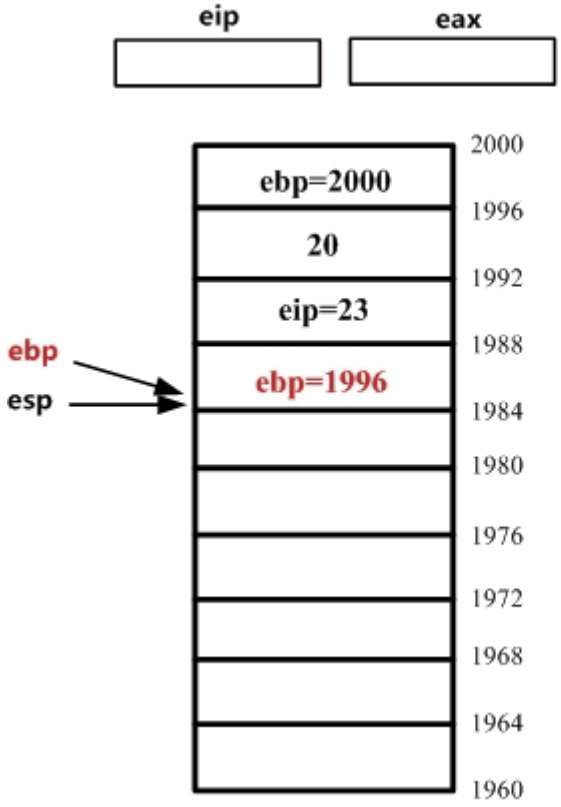

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



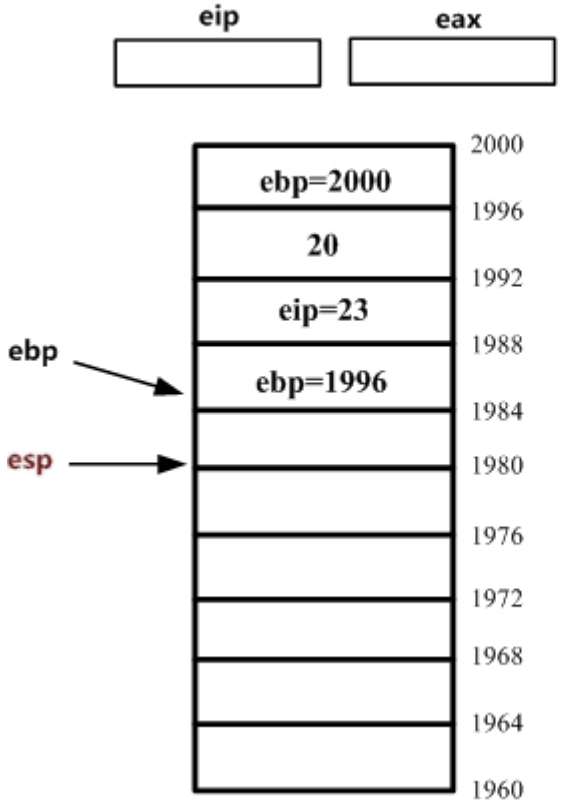

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



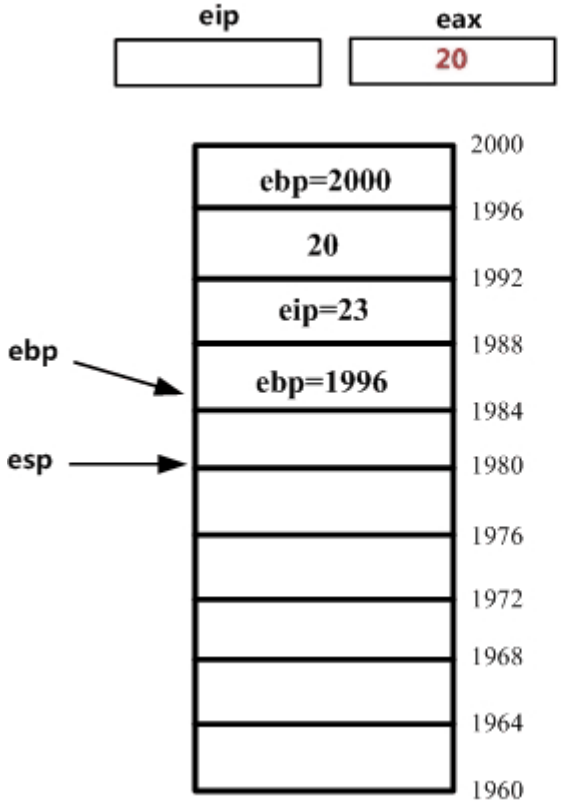

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



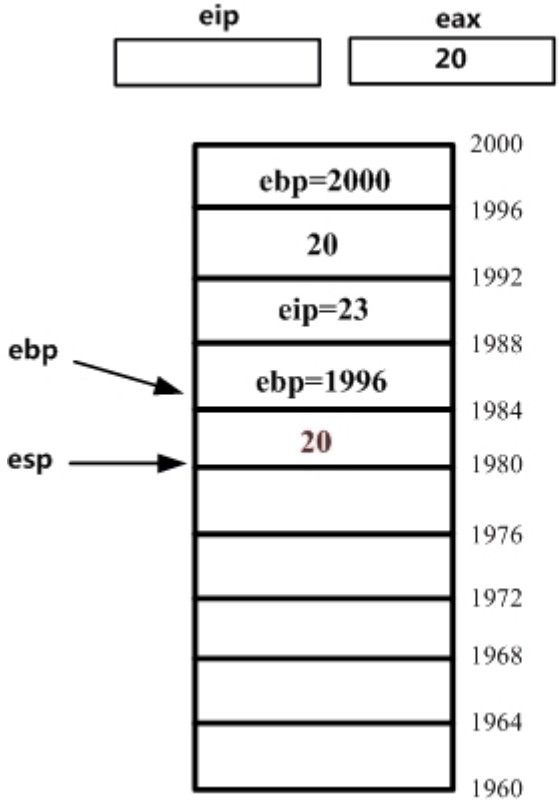

```
1    g:
2        pushl   %ebp
3        movl    %esp, %ebp
4        movl    8(%ebp), %eax
5        addl    $10, %eax
6        popl    %ebp
7        ret
8    f:
9        pushl   %ebp
10       movl    %esp, %ebp
11       subl    $4, %esp
12       movl    8(%ebp), %eax
13       movl    %eax, (%esp)
14       call    g
15       leave
16       ret
17   main:
18       pushl   %ebp
19       movl    %esp, %ebp
20       subl    $4, %esp
21       movl    $20, (%esp)
22       call    f:
23       addl    $8, %eax
```



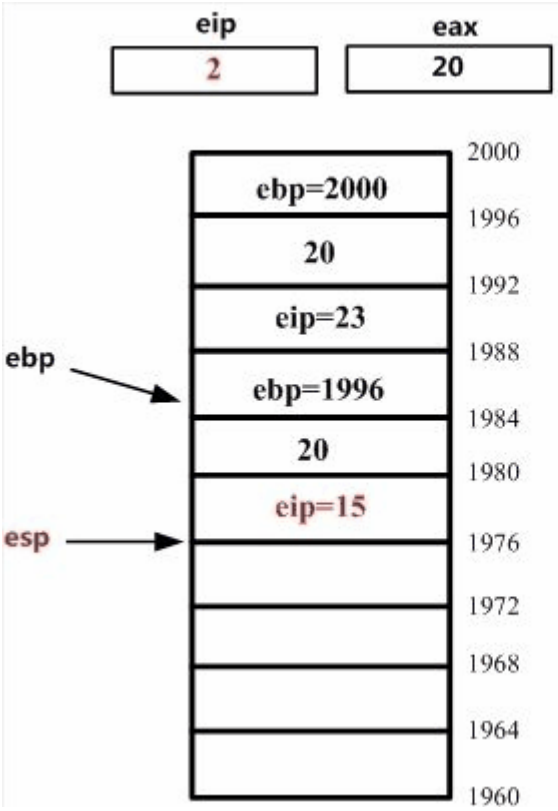

```
1    g:
2        pushl   %ebp
3        movl    %esp, %ebp
4        movl    8(%ebp), %eax
5        addl    $10, %eax
6        popl    %ebp
7        ret
8    f:
9        pushl   %ebp
10       movl    %esp, %ebp
11       subl    $4, %esp
12       movl    8(%ebp), %eax
13       movl    %eax, (%esp)
14       call    g
15       leave
16       ret
17   main:
18       pushl   %ebp
19       movl    %esp, %ebp
20       subl    $4, %esp
21       movl    $20, (%esp)
22       call    f:
23       addl    $8, %eax
```



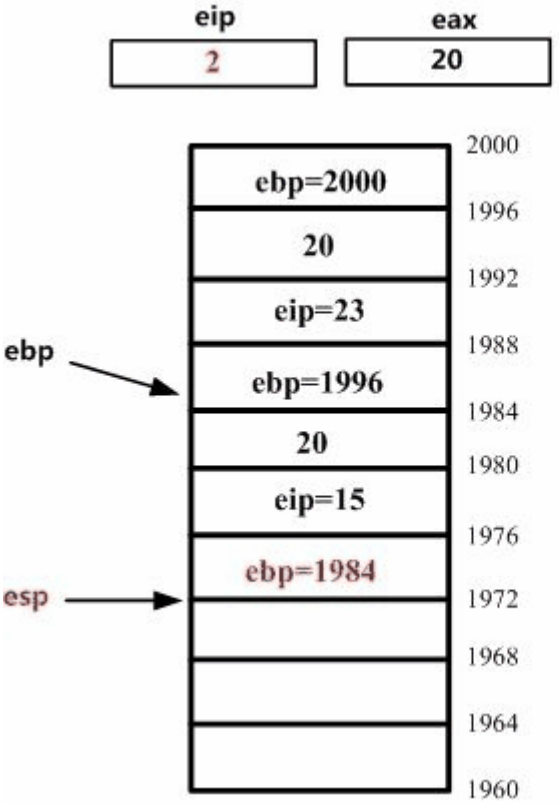

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %ea
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



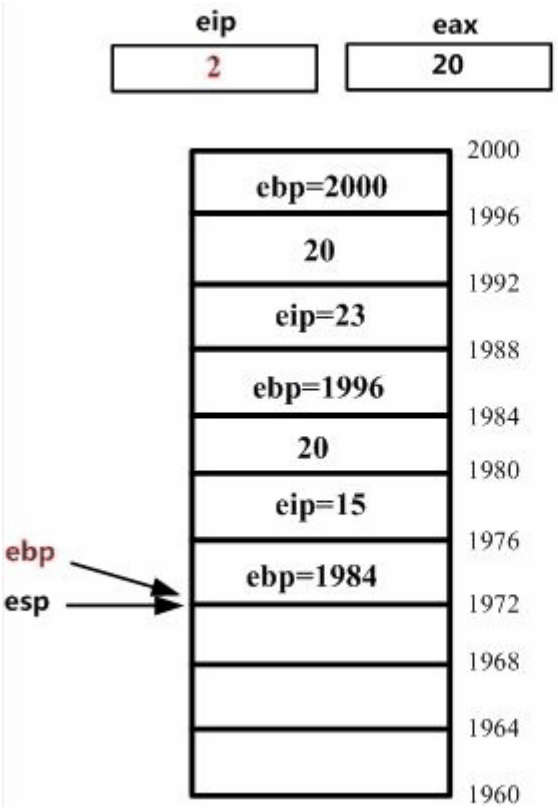

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



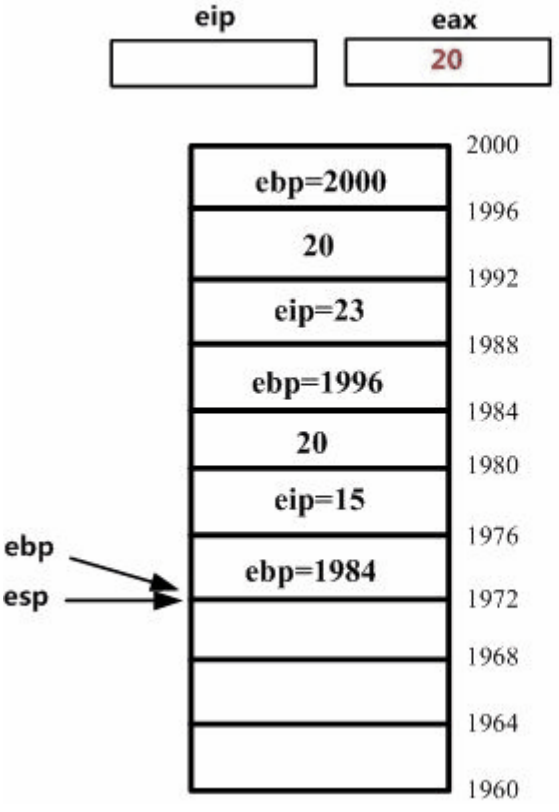

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



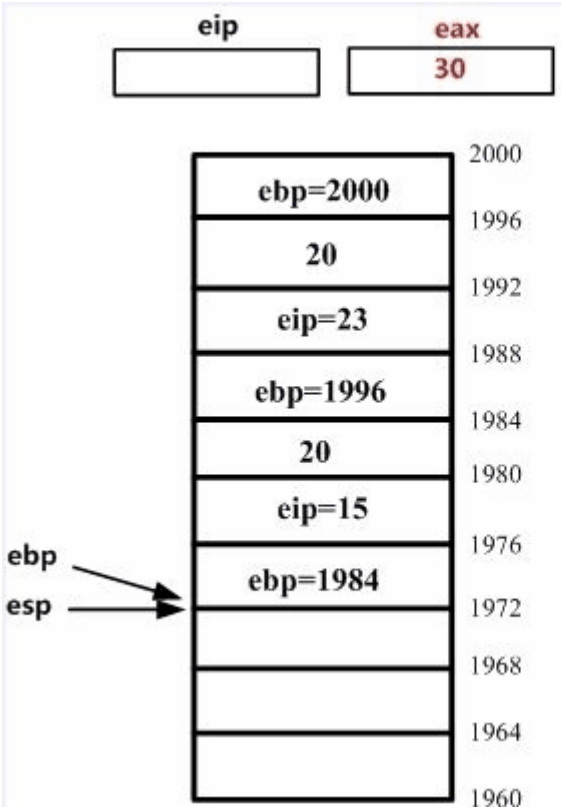

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



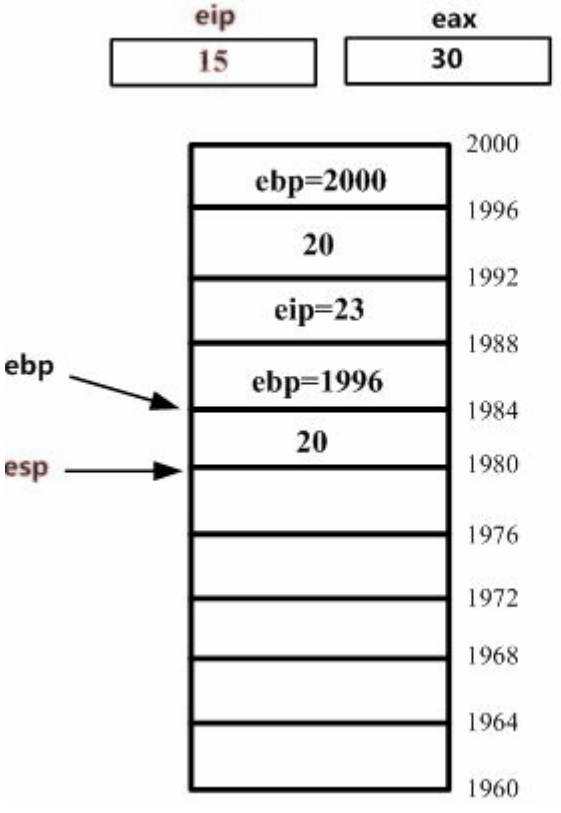

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



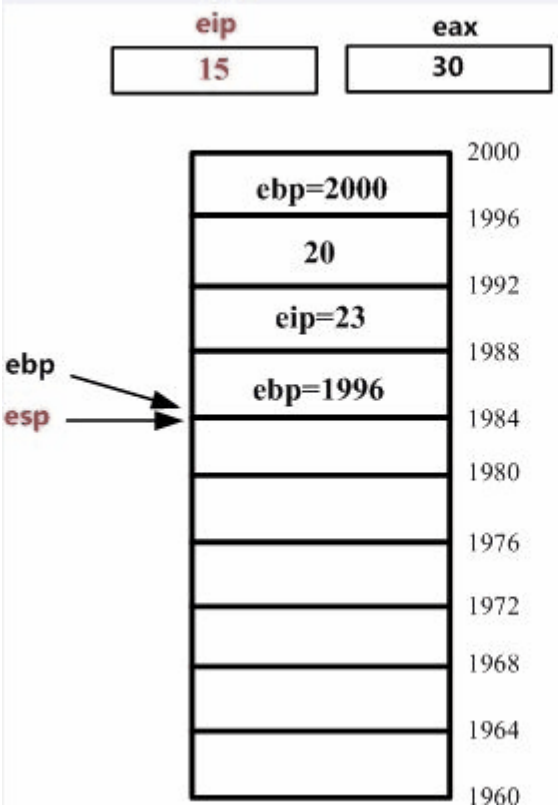

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %ea
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



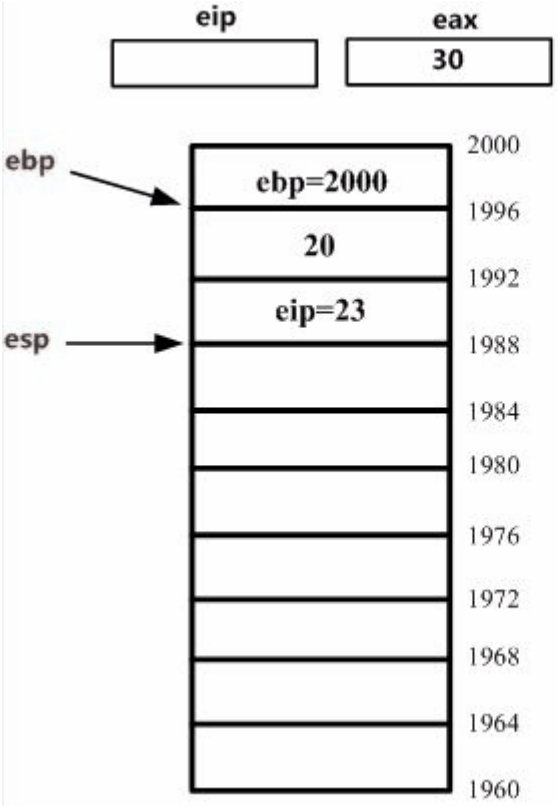

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



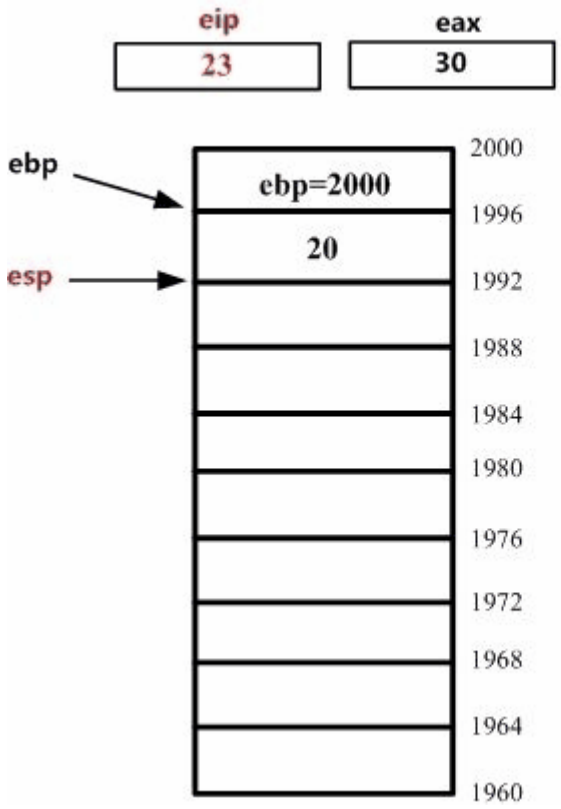

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



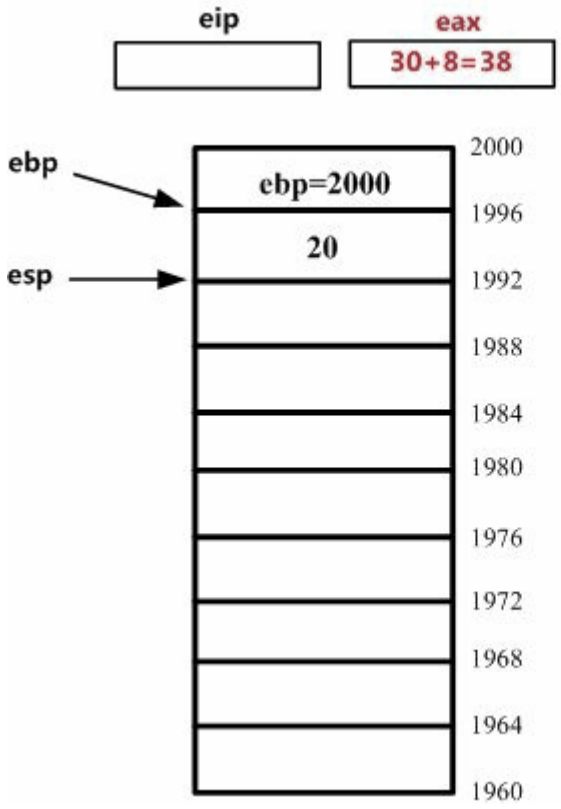

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



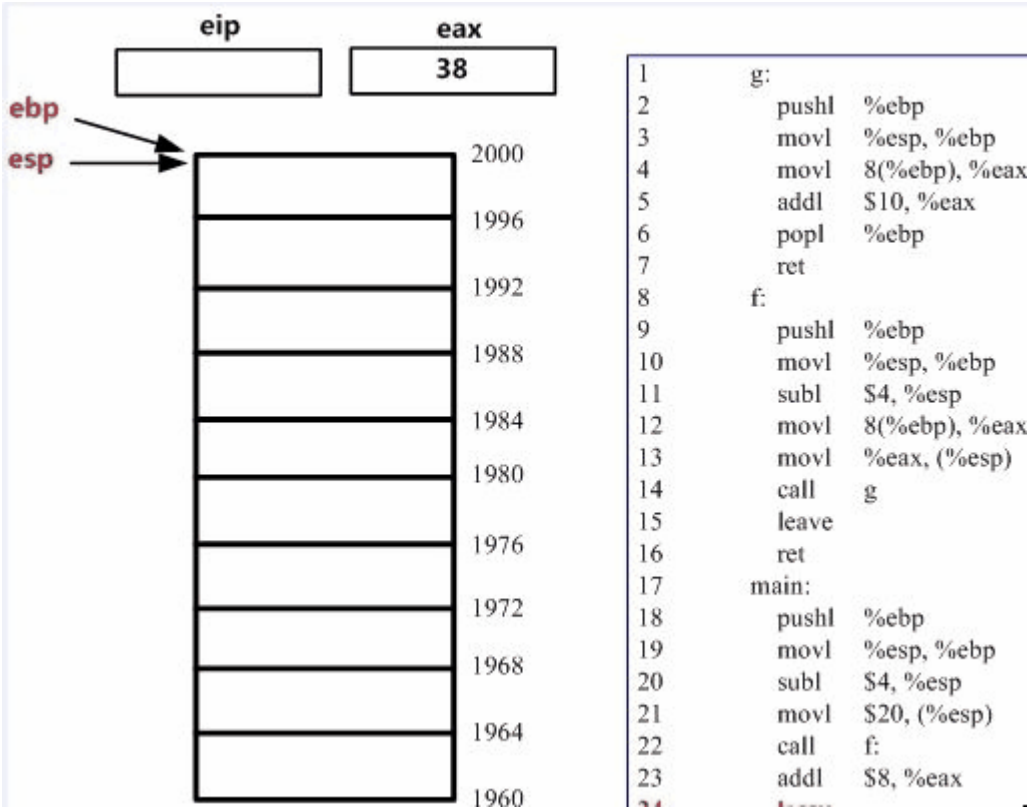

```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```




```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```




```
1       g:
2           pushl   %ebp
3           movl    %esp, %ebp
4           movl    8(%ebp), %eax
5           addl    $10, %eax
6           popl    %ebp
7           ret
8       f:
9           pushl   %ebp
10          movl    %esp, %ebp
11          subl    $4, %esp
12          movl    8(%ebp), %eax
13          movl    %eax, (%esp)
14          call    g
15          leave
16          ret
17      main:
18          pushl   %ebp
19          movl    %esp, %ebp
20          subl    $4, %esp
21          movl    $20, (%esp)
22          call    f:
23          addl    $8, %eax
```



（2）查看 lab1/obj 目录下的 bootblock.asm 和 kernel.asm

```
83
84  .code32                                             # Assemble for 32-bit mode
85  protcseg:
86      # Set up the protected-mode data segment registers
87      movw $PROT_MODE_DSEG, %ax                       # Our data segment selector
88      7c32:   66 b8 10 00             mov    $0x10,%ax
89      movw %ax, %ds                                   # -> DS: Data Segment
90      7c36:   8e d8                   mov    %eax,%ds
91      movw %ax, %es                                   # -> ES: Extra Segment
92      7c38:   8e c0                   mov    %eax,%es
93      movw %ax, %fs                                   # -> FS
94      7c3a:   8e e0                   mov    %eax,%fs
95      movw %ax, %gs                                   # -> GS
96      7c3c:   8e e8                   mov    %eax,%gs
97      movw %ax, %ss                                   # -> SS: Stack Segment
98      7c3e:   8e d0                   mov    %eax,%ss
99
100     # Set up the stack pointer and call into C. The stack region is from 0--start(0x7c00)
101     movl $0x0, %ebp
102     7c40:   bd 00 00 00 00          mov    $0x0,%ebp
103     movl $start, %esp
104     7c45:   bc 00 7c 00 00          mov    $0x7c00,%esp
105     call bootmain
106     7c4a:   e8 c1 00 00 00          call   7d10 <bootmain>
107
108 00007c4f <spin>:
109
110     # If bootmain returns (it shouldn't), loop.
111 spin:
112     jmp spin
```

```
12  void
13  kern_init(void){
14    100000:   55                      push   %ebp
15    100001:   89 e5                   mov    %esp,%ebp
16    100003:   83 ec 18                sub    $0x18,%esp
17      extern char edata[], end[];
18      memset(edata, 0, end - edata);
19    100006:   ba 80 fd 10 00          mov    $0x10fd80,%edx
20    10000b:   b8 16 ea 10 00          mov    $0x10ea16,%eax
21    100010:   29 c2                   sub    %eax,%edx
22    100012:   89 d0                   mov    %edx,%eax
23    100014:   83 ec 04                sub    $0x4,%esp
24    100017:   50                      push   %eax
25    100018:   6a 00                   push   $0x0
26    10001a:   68 16 ea 10 00          push   $0x10ea16
27    10001f:   e8 93 2d 00 00          call   102db7 <memset>
28    100024:   83 c4 10                add    $0x10,%esp
29
30      cons_init();                // init the console
31    100027:   e8 42 15 00 00          call   10156e <cons_init>
32
33      const char *message = "(THU.CST) os is loading ...";
34    10002c:   c7 45 f4 60 35 10 00    movl   $0x103560,-0xc(%ebp)
35      cprintf("%s\n\n", message);
36    100033:   83 ec 08                sub    $0x8,%esp
37    100036:   ff 75 f4                pushl  -0xc(%ebp)
38    100039:   68 7c 35 10 00          push   $0x10357c
39    10003e:   e8 0a 02 00 00          call   10024d <cprintf>
40    100043:   83 c4 10                add    $0x10,%esp
41
```

（3）实现调用堆栈跟踪函数 `print_stackframe`[3]

该函数位于 lab1/kern/debug/kdebug.c 文件中，根据注释实现该函数如下：

```
void
print_stackframe(void) {
    uint32_t ebp = read_ebp();  //读取 ebp 的值
    uint32_t eip = read_eip();  //读取 eip 的值

    int i, j;
    for (i = 0; ebp != 0 && i < STACKFRAME_DEPTH; i++) {
        cprintf("ebp:0x%08x eip:0x%08x args:", ebp, eip);
        uint32_t *args = (uint32_t *)ebp + 2;   //参数的首地址
        for (j = 0; j < 4; j++) {   //假设有 4 个参数，依次输出
```

[3] whl1729. 《ucore lab1 exercise5》实验报告[OL]. https://www.cnblogs.com/wuhualong/p/ucore_lab1_exercise5_report.html, 2019-03-04.

```
            cprintf("0x%08x ", args[j]);
        }
        cprintf("\n");
        print_debuginfo(eip - 1);   //打印 C 语言的函数名和行号
        eip = *((uint32_t *)ebp + 1);
        ebp = *((uint32_t *)ebp);   //转到上一个 ebp
    }
}
```

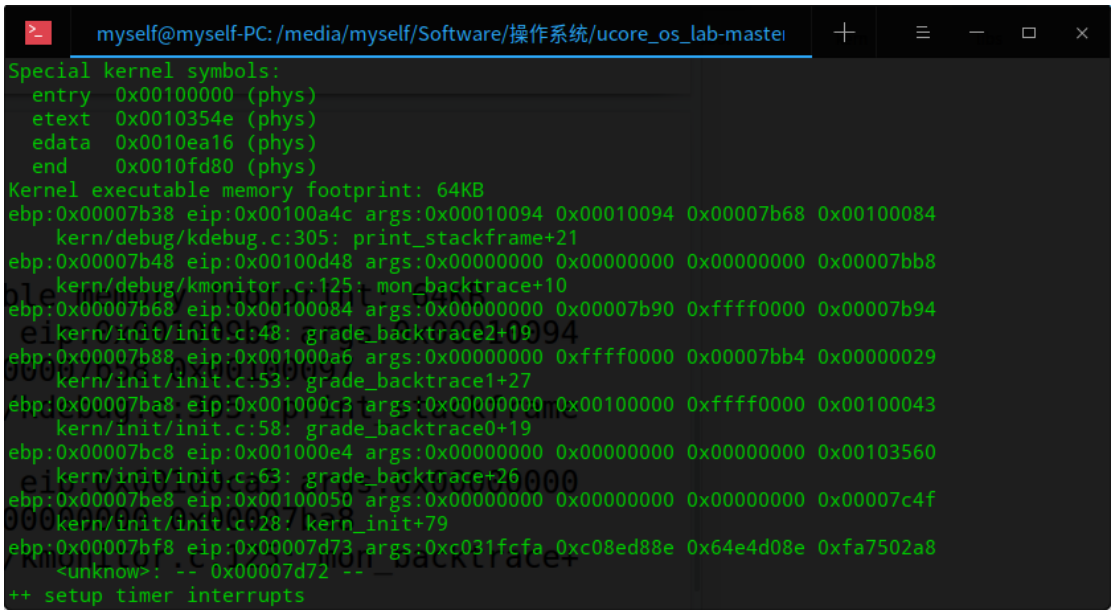
首先定义两个局部变量 ebp、esp 分别存放 ebp、esp 寄存器的值。这里将 ebp 定义为指针，是为了方便后面取 ebp 寄存器的值；其中 cprintf 是定义在 lab1/libs/stdio.h 中负责进行输出的函数。

调用 read_ebp 函数来获取执行 print_stackframe 函数时 ebp 寄存器的值，这里 read_ebp 必须定义为 inline 函数，否则获取的是执行 read_ebp 函数时的 ebp 寄存器的值；调用 read_eip 函数来获取当前指令的位置，也就是此时 eip 寄存器的值。这里 read_eip 必须定义为常规函数而不是 inline 函数，因为这样的话在调用 read_eip 时会把当前指令的下一条指令的地址（也就是 eip 寄存器的值）压栈，那么在进入 read_eip 函数内部后便可以从栈中获取到调用前 eip 寄存器的值。

由于变量 eip 存放的是下一条指令的地址，因此将变量 eip 的值减去 1，得到的指令地址就属于当前指令的范围了；由于只要输入的地址属于当前指令的起始和结束位置之间，print_debuginfo 都能搜索到当前指令，因此这里减去 1 即可。

以后变量 eip 的值就不能再调用 read_eip 来获取了（每次调用获取的值都是相同的），而应该从 ebp 寄存器指向栈中的位置再往上一个单位中获取；由于 ebp 寄存器指向栈中的位置存放的是调用者的 ebp 寄存器的值，之后通过不断回溯，直到 ebp 寄存器的值变为 0。

（4）make qemu



（5）解释最后一行各个数值的含义[4]

最后一行是 ebp:0x00007bf8 eip:0x00007d73 args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8，共有 ebp，eip 和 args 三类参数，下面分别给出解释。

ebp:0x0007bf8 此时 ebp 的值是 kern_init 函数的栈顶地址，从 obj/bootblock.asm 文件中知道整个栈的栈顶地址为 0x00007c00，ebp 指向的栈位置存放调用者的 ebp 寄存器的值，ebp+4 指向的栈位置存放返回地址的值，这意味着 kern_init 函数的调用者（也就是 bootmain 函数）没有传递任何输入参数给它！因为单是存放旧的 ebp、返回地址已经占用 8 字节了。

eip:0x00007d73 eip 的值是 kern_init 函数的返回地址，也就是 bootmain 函数调用 kern_init 对应的指令的下一条指令的地址。这与 obj/bootblock.asm 是相符合的。

```
302     7d6c:   25 ff ff ff 00          and    $0xffffff,%eax
303     7d71:   ff d0                   call   *%eax
304 }
305
306 static inline void
307 outw(uint16_t port, uint16_t data) {
308     asm volatile ("outw %0, %1" :: "a" (data), "d" (port));
309     7d73:   ba 00 8a ff ff          mov    $0xffff8a00,%edx
```

args:0xc031fcfa 0xc08ed88e 0x64e4d08e 0xfa7502a8 一般来说，args 存放的 4 个 dword 是对应 4 个输入参数的值。但这里比较特殊，由于 bootmain 函数调用 kern_init 并没传递任何输入参数，并且栈顶的位置恰好在 bootloader 第一条指令存放的地址的上面，而 args 恰好是 kern_int 的 ebp 寄存器指向的栈顶往上第 2~5 个单元，因此 args 存放的就是 bootloader 指令的前 16 个字节！可以对比 obj/bootblock.asm 文件来验证（验证时要注意系统是小端字节序）。

```
 9  # start address should be 0:7c00, in real mode, the beginning address of the running bootloader
10  .globl start
11  start:
12  .code16                                             # Assemble for 16-bit mode
13      cli                                             # Disable interrupts
14      7c00:   fa                      cli
15      cld                                             # String operations increment
16      7c01:   fc                      cld
17
18      # Set up the important data segment registers (DS, ES, SS).
19      xorw %ax, %ax                                   # Segment number zero
20      7c02:   31 c0                   xor    %eax,%eax
21      movw %ax, %ds                                   # -> Data Segment
22      7c04:   8e d8                   mov    %eax,%ds
23      movw %ax, %es                                   # -> Extra Segment
24      7c06:   8e c0                   mov    %eax,%es
25      movw %ax, %ss                                   # -> Stack Segment
26      7c08:   8e d0                   mov    %eax,%ss
```

[4] whl1729. 《ucore lab1 exercise5》实验报告[OL]. https://www.cnblogs.com/wuhualong/p/ucore_lab1_exercise5_report.html, 2019-03-04.