```
     long call_proc()
1    call_proc:
       Set up arguments to proc
2      subq    $32, %rsp          Allocate 32-byte stack frame
3      movq    $1, 24(%rsp)       Store 1 in &x1
4      movl    $2, 20(%rsp)       Store 2 in &x2
5      movw    $3, 18(%rsp)       Store 3 in &x3
6      movb    $4, 17(%rsp)       Store 4 in &x4
7      leaq    17(%rsp), %rax     Create &x4
8      movq    %rax, 8(%rsp)      Store &x4 as argument 8
9      movl    $4, (%rsp)         Store 4 as argument 7
10     leaq    18(%rsp), %r9      Pass &x3 as argument 6
11     movl    $3, %r8d           Pass 3 as argument 5
12     leaq    20(%rsp), %rcx     Pass &x2 as argument 4
13     movl    $2, %edx           Pass 2 as argument 3
14     leaq    24(%rsp), %rsi     Pass &x1 as argument 2
15     movl    $1, %edi           Pass 1 as argument 1
       Call proc
16     call    proc
       Retrieve changes to memory
17     movslq  20(%rsp), %rdx     Get x2 and convert to long
18     addq    24(%rsp), %rdx     Compute x1+x2
19     movswl  18(%rsp), %eax     Get x3 and convert to int
20     movsbl  17(%rsp), %ecx     Get x4 and convert to int
21     subl    %ecx, %eax         Compute x3-x4
22     cltq                       Convert to long
23     imulq   %rdx, %rax         Compute (x1+x2) * (x3-x4)
24     addq    $32, %rsp          Deallocate stack frame
25     ret                        Return
```

b）调用函数生成的汇编代码

图 3-32　（续）

看看 call_proc 的汇编代码(图 3-32b)，可以看到代码中一大部分(第 2～15 行)是为调用 proc 做准备。其中包括为局部变量和函数参数建立栈帧，将函数参数加载至寄存器。如图 3-33 所示，在栈上分配局部变量 x1～x4，它们具有不同的大小：24～31(x1)，20～23(x2)，18～19(x3)和 17(s3)。用 leaq 指令生成到这些位置的指针(第 7、10、12 和 14 行)。参数 7(值为 4)和 8(指向 x4 的位置的指针)存放在栈中相对于栈指针偏移量为 0 和 8 的地方。

当调用过程 proc 时，程序会开始执行图 3-29b 中的代码。如图 3-30 所示，参数 7 和 8 现在位于相对于栈指针偏移量为 8 和 16 的地方，因为返回地址这时已经被压入栈中了。

当程序返回 call_proc 时，代码会取出 4 个局部变量(第 17～20 行)，并执行最终的计算。在程序结束前，把栈指针加 32，释放这个栈帧。

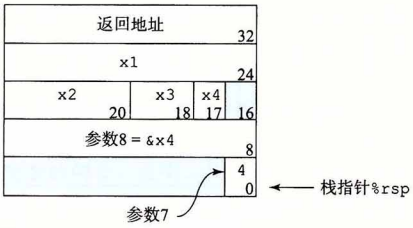

图 3-33　函数 call_proc 的栈帧。该栈帧包含局部变量和两个要传递给函数 proc 的参数

3.7.5　寄存器中的局部存储空间

寄存器组是唯一被所有过程共享的资源。