汇编代码及注释

.file "lab.c" ;打开文件

.text ;

.globl add2 ;声明add2要被连接器用到（.global声明）

.type add2, @function ;定义函数add2

add2:

.LFB0: ;函数的开头生成的标签

.cfi\_startproc ;用在每个函数的开始，用于初始化一些内部数据

endbr64 ;终止64位间接分支

pushq %rbp ;将父函数的栈帧指针压栈

.cfi\_def\_cfa\_offset 16 ;此处距离CFA地址为16字节

.cfi\_offset 6, -16 ;把第6号寄存器原先的值保存在距离CFA - 16的位置

movq %rsp, %rbp ;使 %rsp 和 %rbp 指向同一位置，将栈中保存的父栈帧的 %rbp 的值赋值给 %rbp，并且 %rsp 上移一个位置指向父栈帧的结尾处

.cfi\_def\_cfa\_register 6 ;从这里开始, 使用rbp作为计算CFA的基址寄存器(前面用的是rsp)

movl %edi, -4(%rbp) ;约定%edi为第一个参数，保存到-4(%rbp)的位置

movl %esi, -8(%rbp) ;约定%esi为第一个参数，保存到-8(%rbp)的位置

movl -4(%rbp), %edx ;-4(%rbp)保存到%edx

movl -8(%rbp), %eax ;-8(%rbp)保存到%eax

addl %edx, %eax ;计算edx+eax

popq %rbp ;恢复 rip 寄存器，将程序控制权交给调用者

.cfi\_def\_cfa 7, 8 ;现在重新定义CFA, 它的值是第7号寄存器所指位置加8字节

ret ;弹出返回地址到eip中,从被调用函数返回到调用函数

.cfi\_endproc ;函数结束

.LFE0: ;函数的结尾生成的标签

.size add2, .-add2 ;设置与add2有关的空间大小

.globl add3 ;声明add3要被连接器用到（.global声明）

.type add3, @function ;定义函数add3

add3:

.LFB1: ;函数的开头生成的标签

.cfi\_startproc ;用在每个函数的开始，用于初始化一些内部数据

endbr64 ;终止64位间接分支

pushq %rbp ；将父函数的栈帧指针压栈

.cfi\_def\_cfa\_offset 16 ;此处距离CFA地址为16字节

.cfi\_offset 6, -16 ;把第6号寄存器原先的值保存在距离CFA - 16的位置

movq %rsp, %rbp ;使 %rsp 和 %rbp 指向同一位置，将栈中保存的父栈帧的 %rbp 的值赋值给 %rbp，并且 %rsp 上移一个位置指向父栈帧的结尾处

.cfi\_def\_cfa\_register 6 ； ;从这里开始, 使用rbp作为计算CFA的基址寄存器(前面用的是rsp)

subq $16, %rsp ;申请16字节的栈空间

movl %edi, -4(%rbp) ;约定%edi为参数，保存到-4(%rbp)的位置

movl %esi, -8(%rbp) ;约定%esi为参数，保存到-8(%rbp)的位置

movl %edx, -12(%rbp) ;约定%edi为参数，保存到-12(%rbp)的位置

movl -12(%rbp), %edx ;edx=[rbp-12]

movl -8(%rbp), %eax ;eax=[rbp-8]

movl %edx, %esi ;esi=edx

movl %eax, %edi ;edi=eax

call add2 ；调用函数add2

movl -4(%rbp), %edx ;edx=[rbp-4]

addl %edx, %eax ;计算edx+eax

leave ;执行完这两条指令后, ebp恢复为旧的ebp, 即指向调用者的基址, esp则指向返回地址。

.cfi\_def\_cfa 7, 8 ；;现在重新定义CFA, 它的值是第7号寄存器所指位置加8字节

ret ;弹出返回地址到eip中,从被调用函数返回到调用函数

.cfi\_endproc ;函数结束

.LFE1: ;函数的结尾生成的标签

.size add3, .-add3 ;设置与add2有关的空间大小

.globl main ;声明main要被连接器用到（.global声明）

.type main, @function ;定义函数main

main:

.LFB2: ;函数的开头生成的标签

.cfi\_startproc ;用在每个函数的开始，用于初始化一些内部数据

endbr64 ;终止64位间接分支

pushq %rbp ；将父函数的栈帧指针压栈

.cfi\_def\_cfa\_offset 16 ;此处距离CFA地址为16字节

.cfi\_offset 6, -16 ;把第6号寄存器原先的值保存在距离CFA - 16的位置

movq %rsp, %rbp ;使 %rsp 和 %rbp 指向同一位置，将栈中保存的父栈帧的 %rbp 的值赋值给 %rbp，并且 %rsp 上移一个位置指向父栈帧的结尾处

.cfi\_def\_cfa\_register 6 ;从这里开始, 使用rbp作为计算CFA的基址寄存器(前面用的是rsp)

subq $16, %rsp ;申请16字节的栈空间

movl $1, -8(%rbp) ;-8(%rbp)的位置存1

movl $2, -4(%rbp) ;-4(%rbp)的位置存2

movl -4(%rbp), %ecx ;-4(%rbp)保存到%ecx

movl -8(%rbp), %eax ;-8(%rbp)保存到%eax

movl $3, %edx ;%edx的位置存3

movl %ecx, %esi ;ecx=esi

movl %eax, %edi ;edi=eax

call add3 ;调用add3函数

leave ;执行完这两条指令后, ebp恢复为旧的ebp, 即指向调用者的基址, esp则指向返回地址。

.cfi\_def\_cfa 7, 8 ；现在重新定义CFA, 它的值是第7号寄存器所指位置加8字节

ret ;弹出返回地址到eip中,从被调用函数返回到调用函数

.cfi\_endproc ;函数结束

.LFE2: ;函数的结尾生成的标签

.size main, .-main

.ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0" ;认证为Ubuntu系统里的gcc编译

.section .note.GNU-stack,"",@progbits ;定义一个数据段，段名为note.GNU-stack, progbits为自定义数据段

.section .note.gnu.property,"a" ;定义一个数据段，段名为note.gnu.property, 缺省的标志为a

.align 8 ;按8字节对齐

.long 1f - 0f ;long声明

.long 4f - 1f ;long声明

.long 5 ;long声明

0:

.string "GNU" ;使用GNU操作系统

1:

.align 8

.long 0xc0000002 ;声明一个占32位空间的数

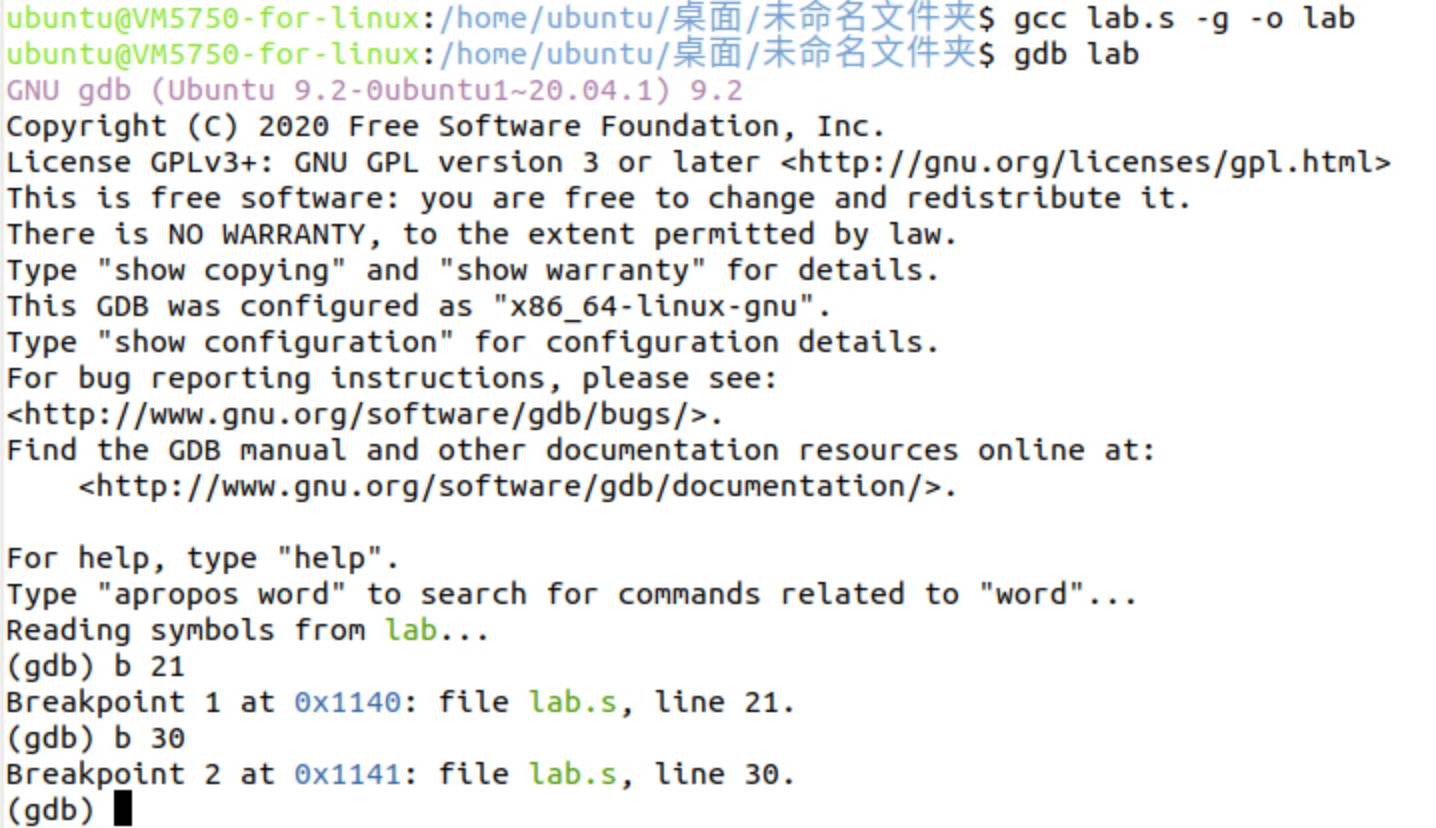
.long 3f - 2f ;long声明

2:

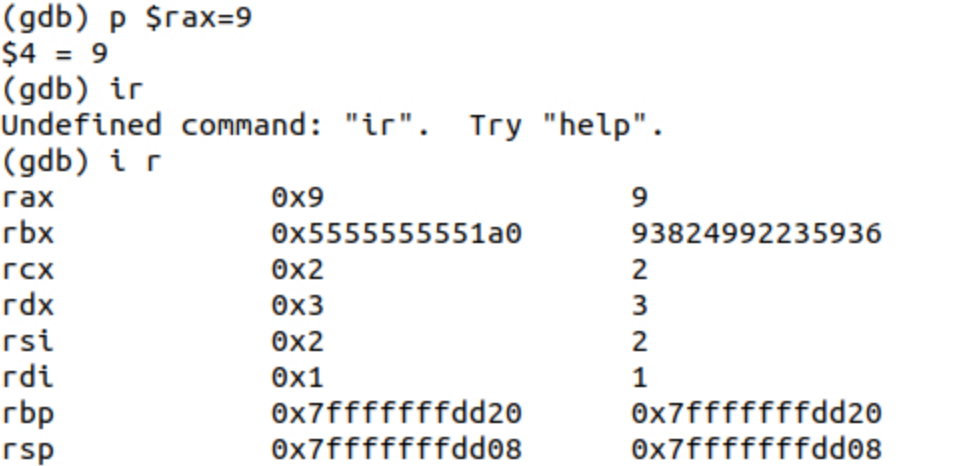
.long 0x3 ;long声明

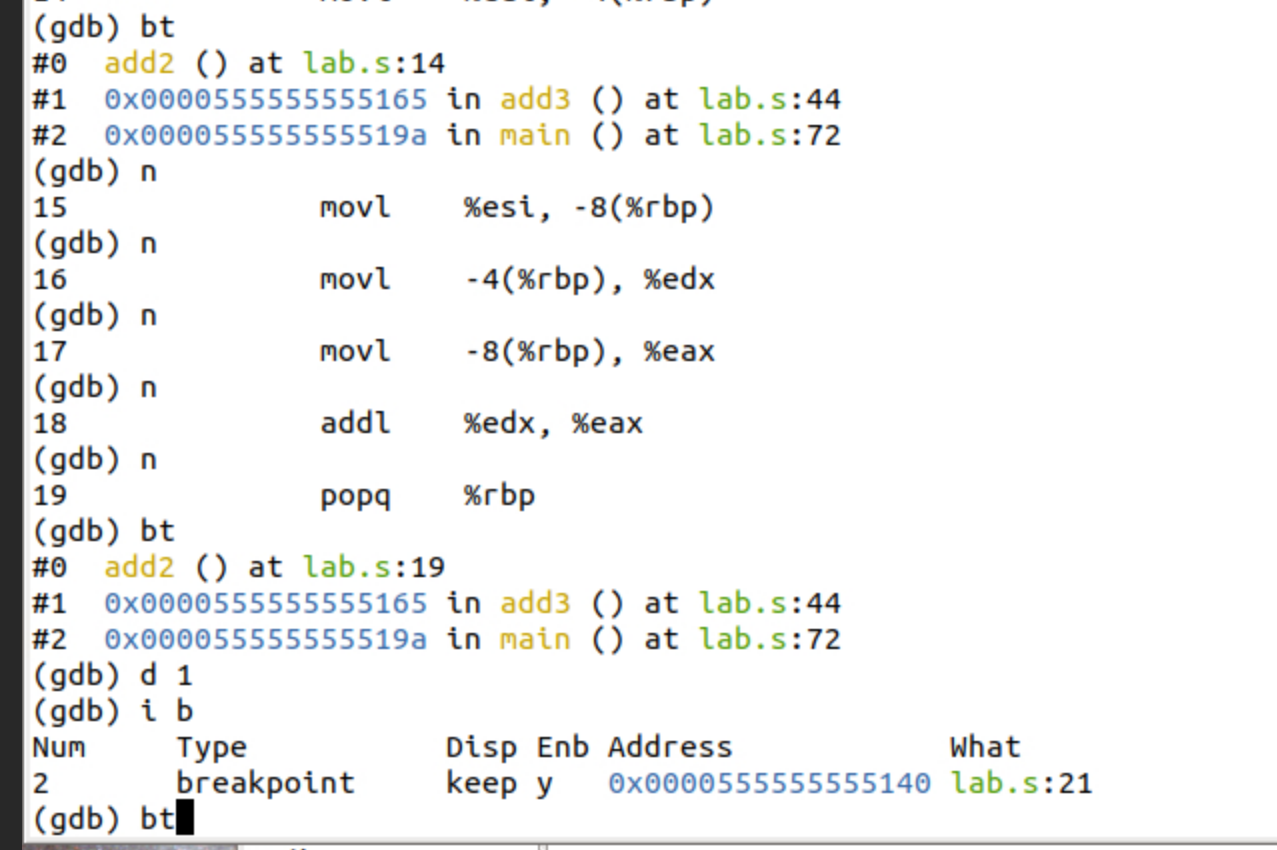
3:

gdb调试:









其中查看函数调用堆栈这一环节前后做了三遍：

第一次因为设置断点设在了add3的程序段，所以查看的栈是在add3函数下查看的（也可能是我理解的不对）；第二次仅仅使用了gdb调试指令但没有启动程序，所以使用查看寄存器或者栈时提示”The program has no register now”以及”No stack”，最后一次才让add2显示出来了。

附加题

源程序：

int funtest(int n1, int n2, int n3, int n4, int n5, int n6, int n7, int n8, int n9)

{

return n1 + n2 + n3 + n4 + n5 + n6 + n7 + n8 + n9;

}

int main()

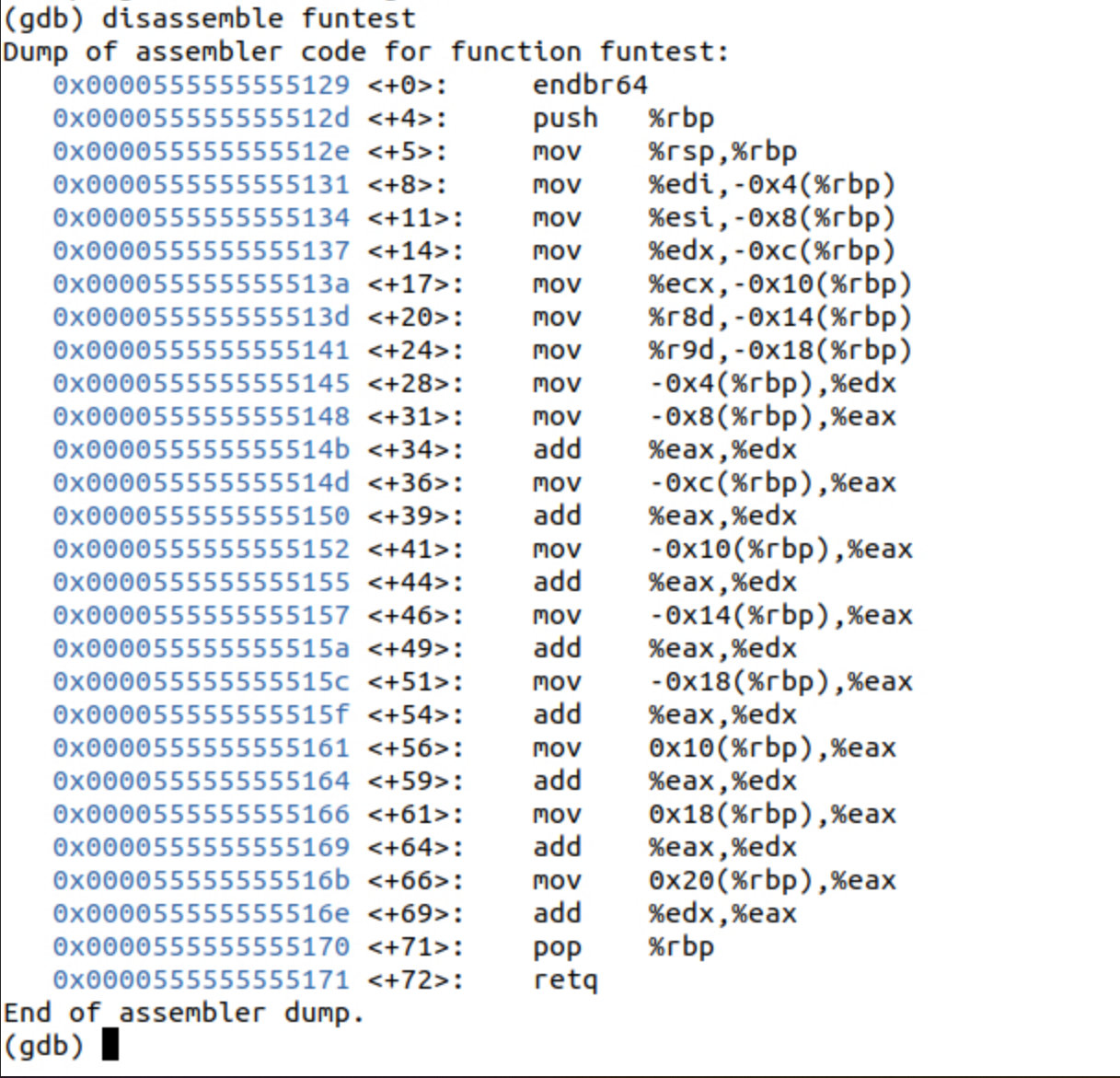
{

return funtest(1, 2, 3, 4, 5, 6, 7, 8, 9);

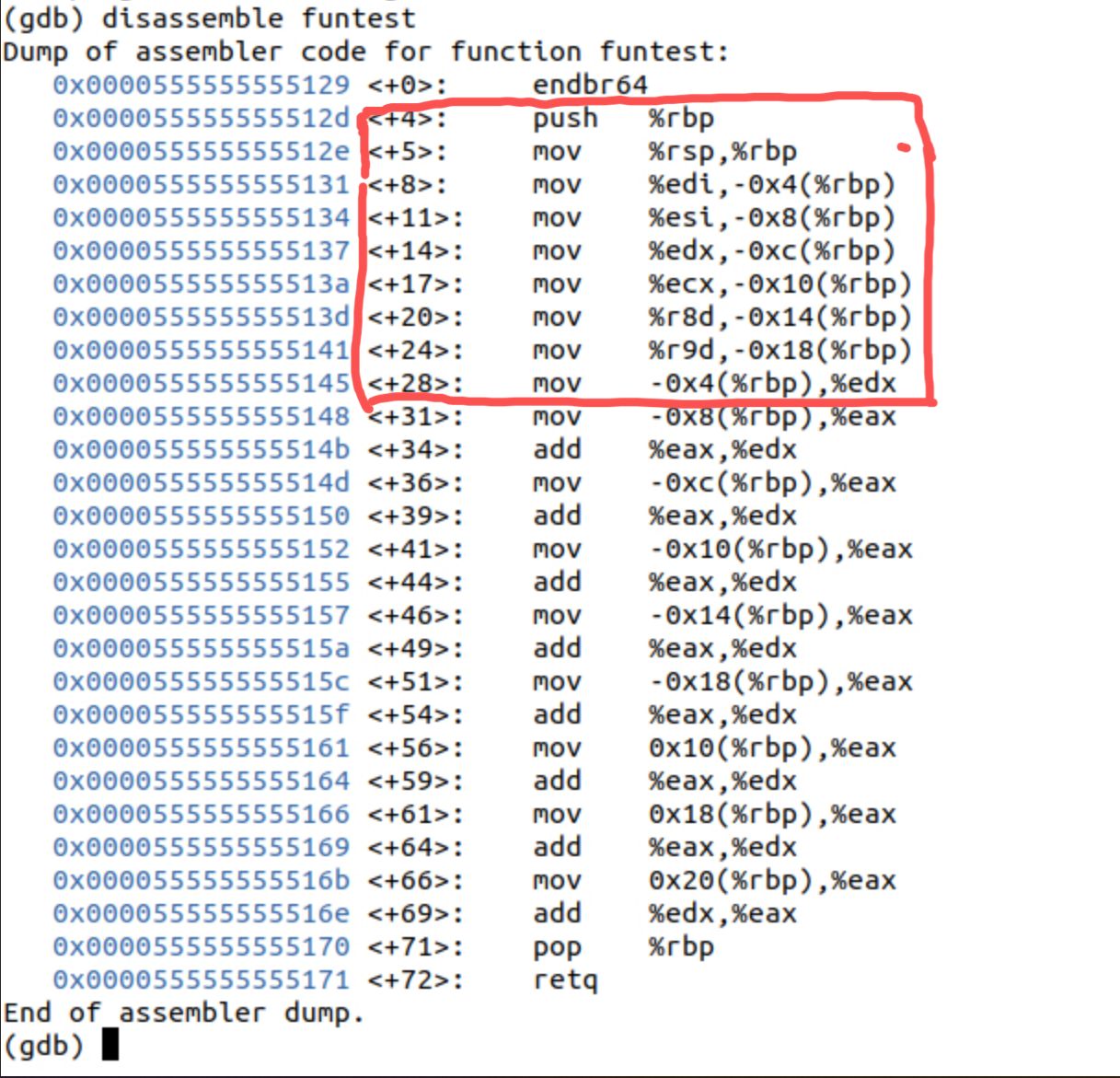
}

编译调试：





（上下为同一张图片）



分析上图可知：函数调用时传递了9个参数，其中一个用栈，其他8个用寄存器（存入rbp寄存器的对应位置）

执行“return n1 + n2 + n3 + n4 + n5 + n6 + n7 + n8 + n9;”指令时，mov,add指令依次执行

用eax寄存器存放需要新加入的数据（即rbp寄存器指定位置存放的数据），用edx存放当前加法的计算结果，依次对eax,edx中的数据作add操作，最终得到所需结果。