x86 和 x64 的栈框架分析

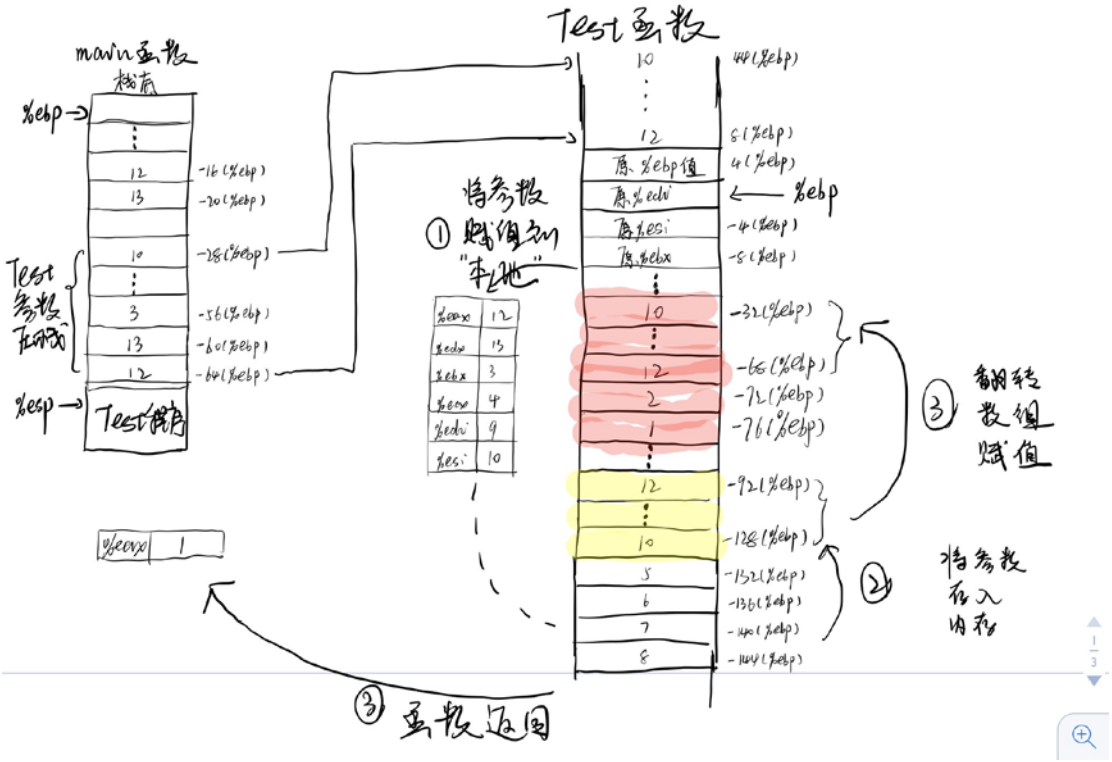
——1170300825 李大鑫

注：最主要的还是那两个手绘的图 _(:3」∠)_

程序代码

```
void test(char a,char b,char c,char d,char e,char f,char g,char h,char i,char j)
{
    int list[10];
    int ax = 1;
    int bx = 2;
    list[0] = a;
    list[1] = b;
    list[2] = c;
    list[3] = d;
    list[4] = e;
    list[5] = f;
    list[6] = g;
    list[7] = h;
    list[8] = i;
    list[9] = j;
}
int main()
{
    int a = 12;
    int b = 13;
    test(a,b,3,4,5,6,7,8,9,10);
    return 0;
}
```

X86 的栈框架分析



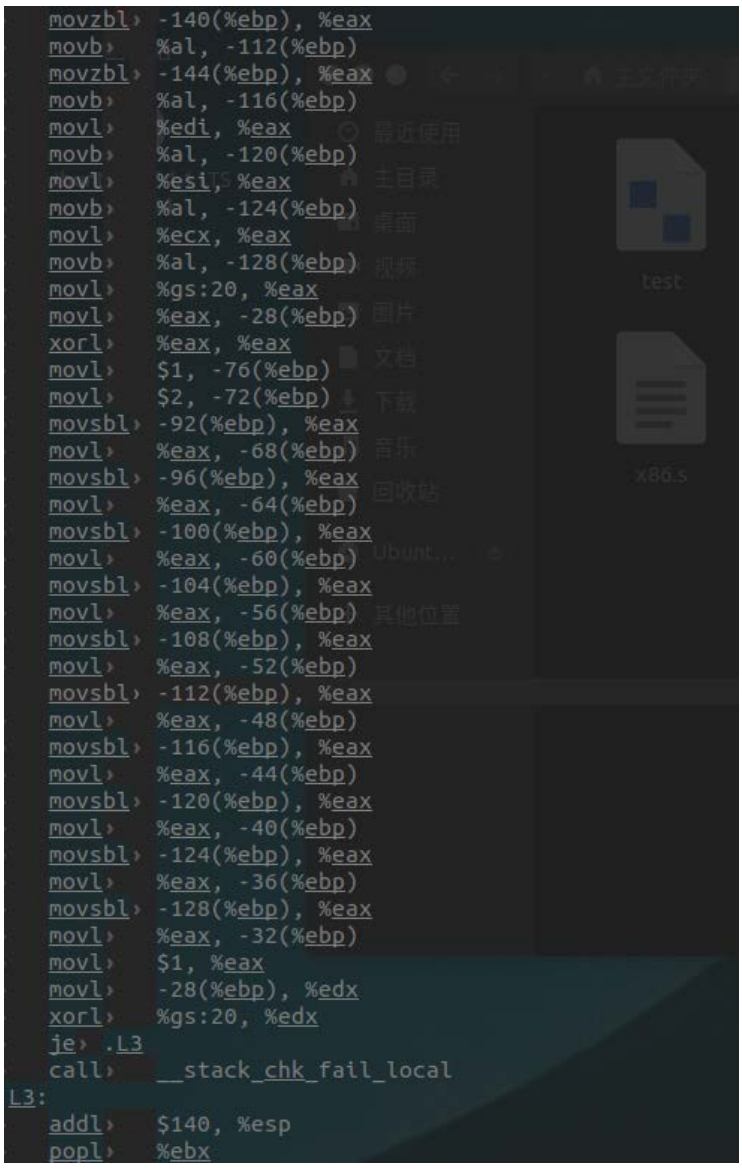
Main 函数

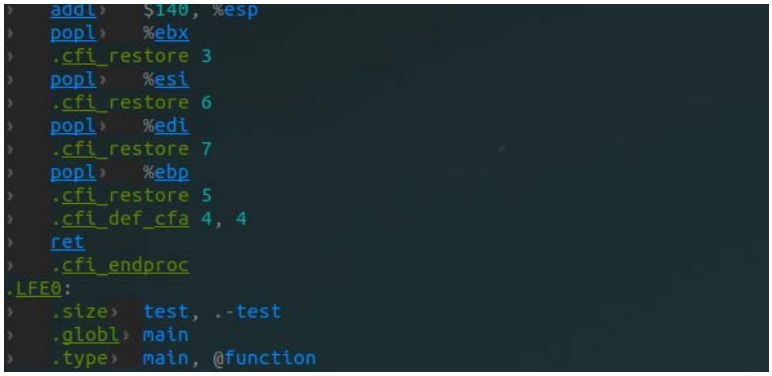
```
main:
.LFB1:
    .cfi_startproc
    leal    4(%esp), %ecx
    .cfi_def_cfa 1, 0
    andl    $-16, %esp
    pushl   -4(%ecx)
    pushl   %ebp
    .cfi_escape 0x10,0x5,0x2,0x75,0
    movl    %esp, %ebp
    pushl   %ecx
    .cfi_escape 0xf,0x3,0x75,0x7c,0x6
    subl    $20, %esp
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    $12, -16(%ebp)
    movl    $13, -12(%ebp)
    movl    -12(%ebp), %eax
    movsbl  %al, %edx
    movl    -16(%ebp), %eax
    movsbl  %al, %eax
    subl    $8, %esp
    pushl   $10
    pushl   $9
    pushl   $8
    pushl   $7
    pushl   $6
    pushl   $5
    pushl   $4
    pushl   $3
    pushl   %edx
    pushl   %eax
    call    test
    addl    $48, %esp
    movl    $0, %eax
    movl    -4(%ebp), %ecx
    .cfi_def_cfa 1, 0
    leave
    .cfi_restore 5
```

```
    leal    -4(%ecx), %esp
    .cfi_def_cfa 4, 4
    ret
    .cfi_endproc
.LFE1:
    .size   main, .-main
    .section    .text.__x86.get_pc_thunk.ax,"axG",@progbits,__x86.get_pc_thunk.ax,comd
    .globl  __x86.get_pc_thunk.ax
    .hidden __x86.get_pc_thunk.ax
    .type   __x86.get_pc_thunk.ax, @function
__x86.get_pc_thunk.ax:
.LFB2:
    .cfi_startproc
    movl    (%esp), %eax
    ret
    .cfi_endproc
.LFE2:
    .hidden __stack_chk_fail_local
    .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
    .section    .note.GNU-stack,"",@progbits
```

Test 函数

```
test:
.LFB0:
    .cfi_startproc
    pushl   %ebp
    .cfi_def_cfa_offset 8
    .cfi_offset 5, -8
    movl    %esp, %ebp
    .cfi_def_cfa_register 5
    pushl   %edi
    pushl   %esi
    pushl   %ebx
    subl    $140, %esp
    .cfi_offset 7, -12
    .cfi_offset 6, -16
    .cfi_offset 3, -20
    call    __x86.get_pc_thunk.ax
    addl    $_GLOBAL_OFFSET_TABLE_, %eax
    movl    8(%ebp), %eax
    movl    12(%ebp), %edx
    movl    16(%ebp), %ebx
    movl    20(%ebp), %ecx
    movl    %ecx, -132(%ebp)
    movl    24(%ebp), %esi
    movl    %esi, -136(%ebp)
    movl    28(%ebp), %edi
    movl    %edi, -140(%ebp)
    movl    32(%ebp), %ecx
    movl    %ecx, -144(%ebp)
    movl    36(%ebp), %edi
    movl    40(%ebp), %esi
    movl    44(%ebp), %ecx
    movb    %al, -92(%ebp)
    movl    %edx, %eax
    movb    %al, -96(%ebp)
    movl    %ebx, %eax
    movb    %al, -100(%ebp)
    movzbl  -132(%ebp), %eax
    movb    %al, -104(%ebp)
    movzbl  -136(%ebp), %eax
```





参数如何传递

由上图可以看出，32 位程序在调用函数 test 的过程中的参数传递的方法是：按照参数列表从右到左的顺序将参数依次压入栈中，因此栈中保存的数字从栈顶到栈底（从低地址到高地址）分别是 1,2,3，...10。

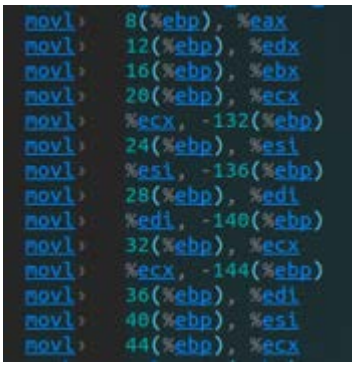
局部变量如何实现

分析在函数内部，栈的操作：

1）将十个参数保存到寄存器和本程序栈中。取参数的顺序是从按照参数列表从左到右的顺序。保存情况图示：

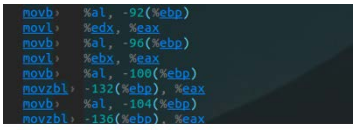
%eax	%edx	%ebx	%ecx	-132 (%ebp)	-136 (%ebp)	-140 (%ebp)	-144 (%ebp)	%edi	%esi
a	b	c	j	d	e	f	g	H	i

可以看出保存参数的策略是：先用寄存器，如果寄存器 eax，edx，ebx，ecx，不能完全保存，则保存到栈中。



(<不使用立即数>差异分析：试过不是用立即数而是先将参数赋值到变量中然后调用函数的版本，结果是直接将所有参数保存到栈中一端连续的空间然后直接进行接下来的两步，因为不使用立即数所以这些数字在调用之前已经被压入栈中，对于 32 位版本就不需要进行多余操作，只需要通过一个中间变量进行赋值到数组空间即可。而使用立即数的话，则需要先将立即数存储在寄存器和内存中，然后保存到栈中的一段连续空间)

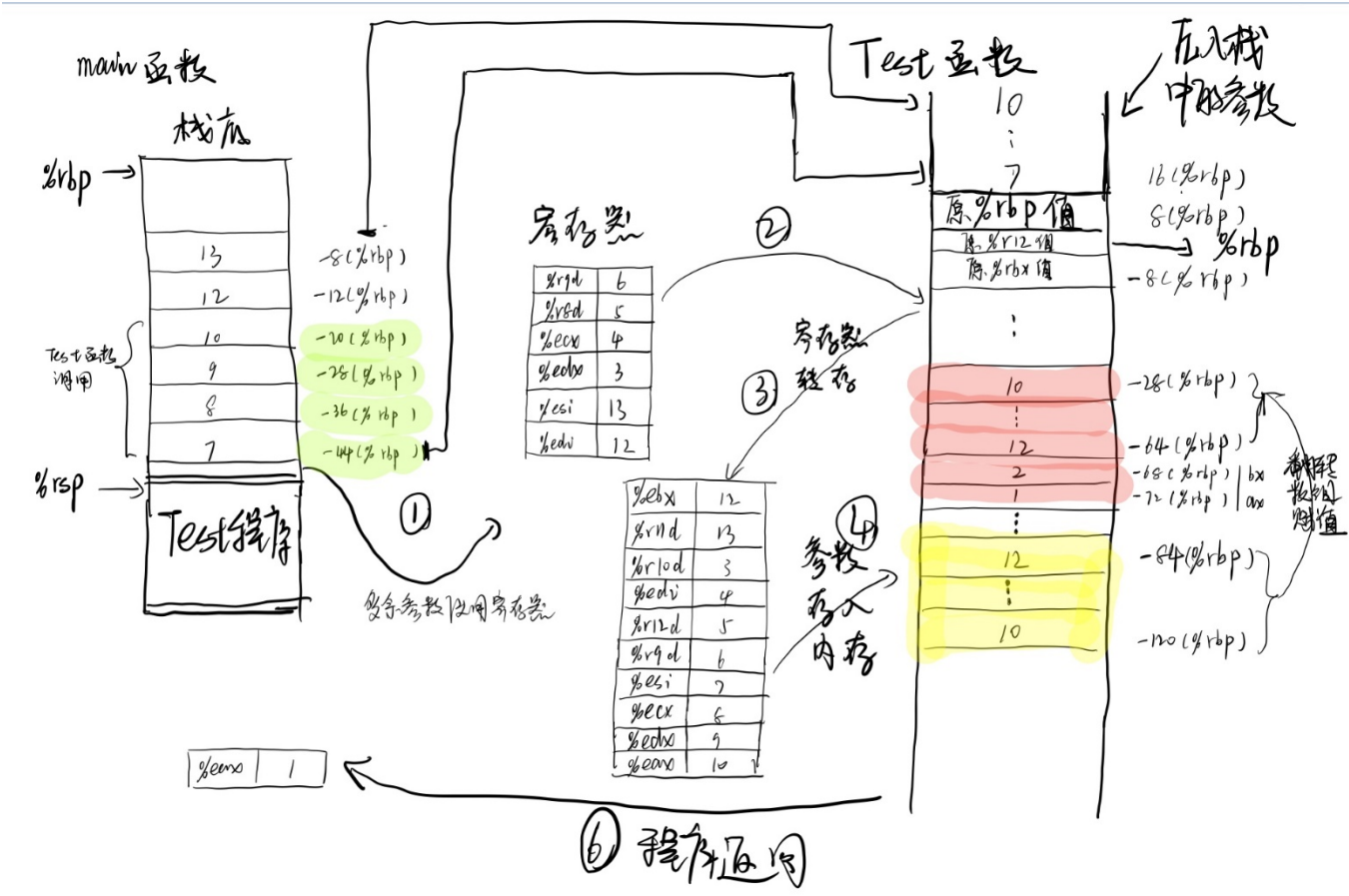
2）然后将上述参数保存到栈中，-92(%ebp) ~ -128(%ebp)。mov 的方法是现将 mov 到 eax，然后将 al 赋值到对应的栈地址。



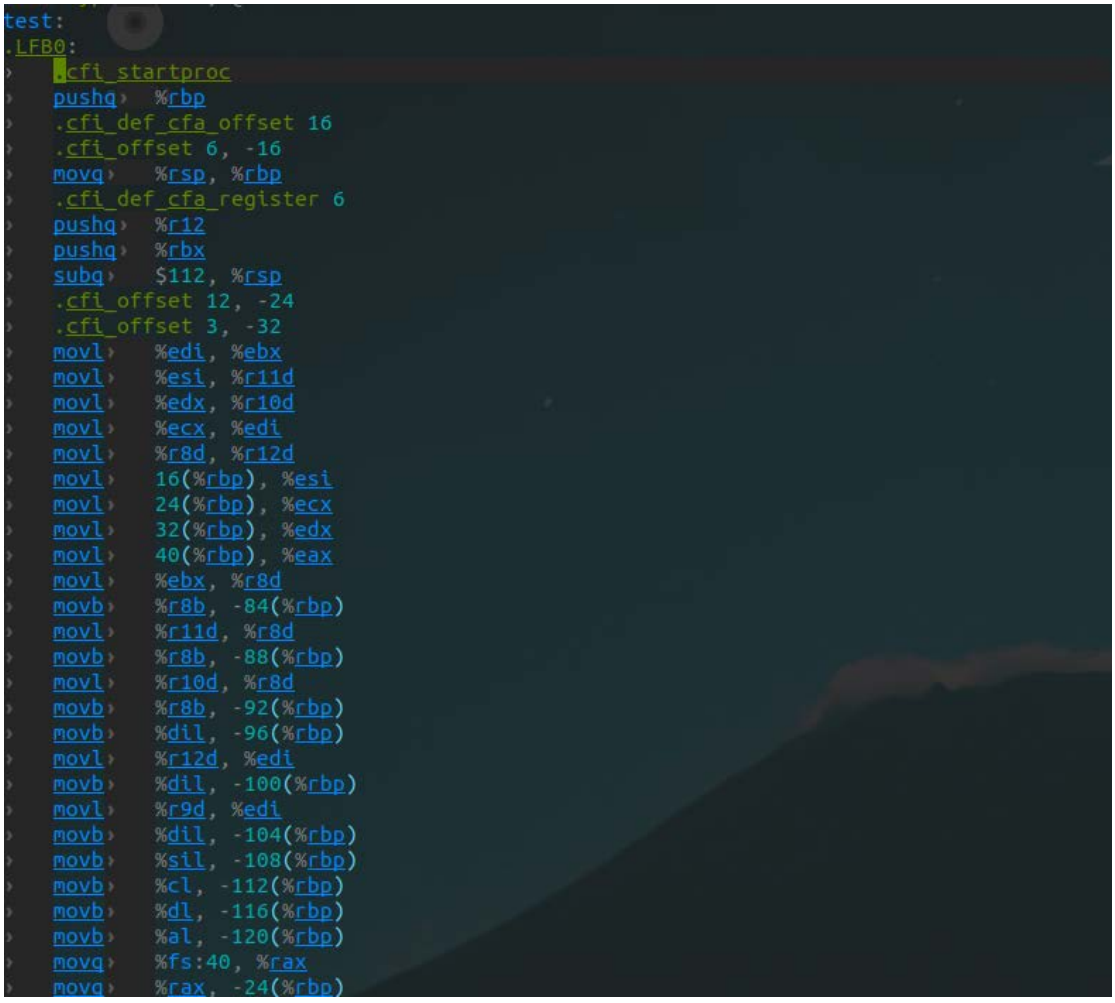
3）然后进行赋值，将 ax 和 bx 保存到-76(%ebp)和-72(%ebp)，数组在栈中是一段连续的空间，-68（%ebp）~-32（%ebp）。低地址在前高地址在后，按照顺序将存储在栈中-92(%ebp) ~ -128(%ebp)的数据 mov 到数组空间-68（%ebp）~-32（%ebp）。

```
movsbl  -92(%ebp), %eax
movl    %eax, -68(%ebp)
movsbl  -96(%ebp), %eax
movl    %eax, -64(%ebp)
movsbl  -100(%ebp), %eax
movl    %eax, -60(%ebp)
movsbl  -104(%ebp), %eax
movl    %eax, -56(%ebp)
movsbl  -108(%ebp), %eax
movl    %eax, -52(%ebp)
movsbl  -112(%ebp), %eax
movl    %eax, -48(%ebp)
movsbl  -116(%ebp), %eax
movl    %eax, -44(%ebp)
movsbl  -120(%ebp), %eax
movl    %eax, -40(%ebp)
movsbl  -124(%ebp), %eax
movl    %eax, -36(%ebp)
movsbl  -128(%ebp), %eax
movl    %eax, -32(%ebp)
```

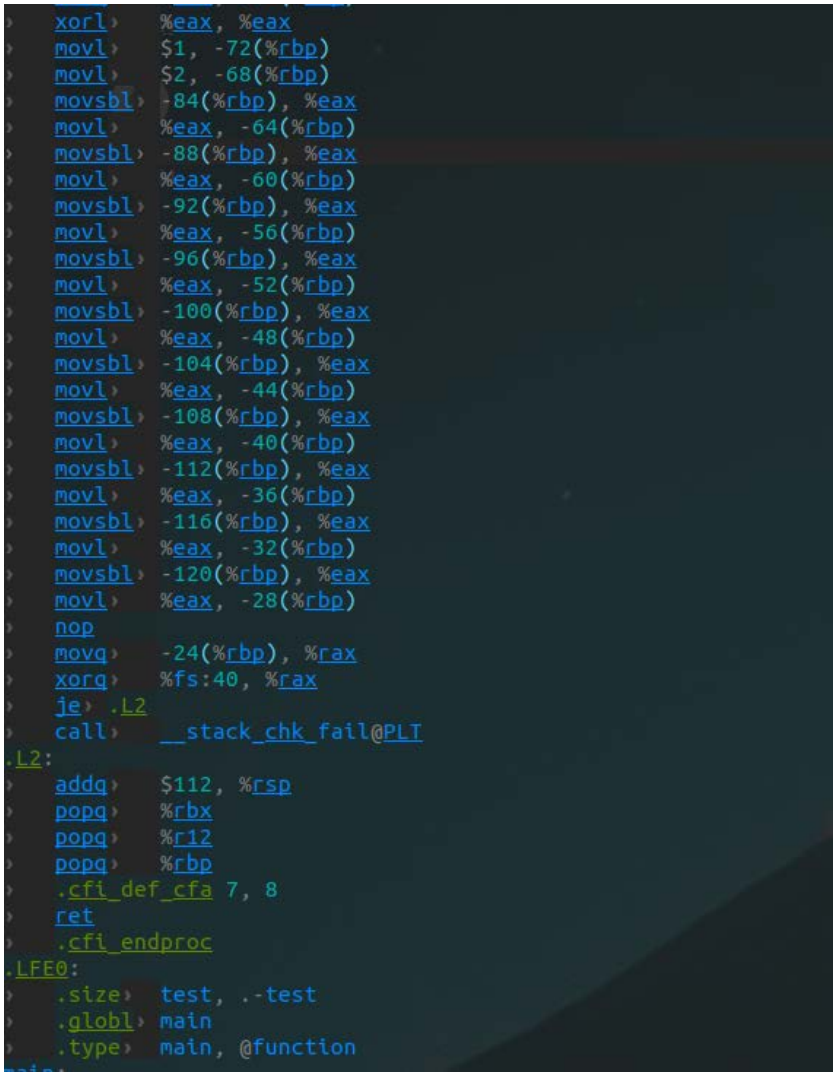
X64 的栈框架分析



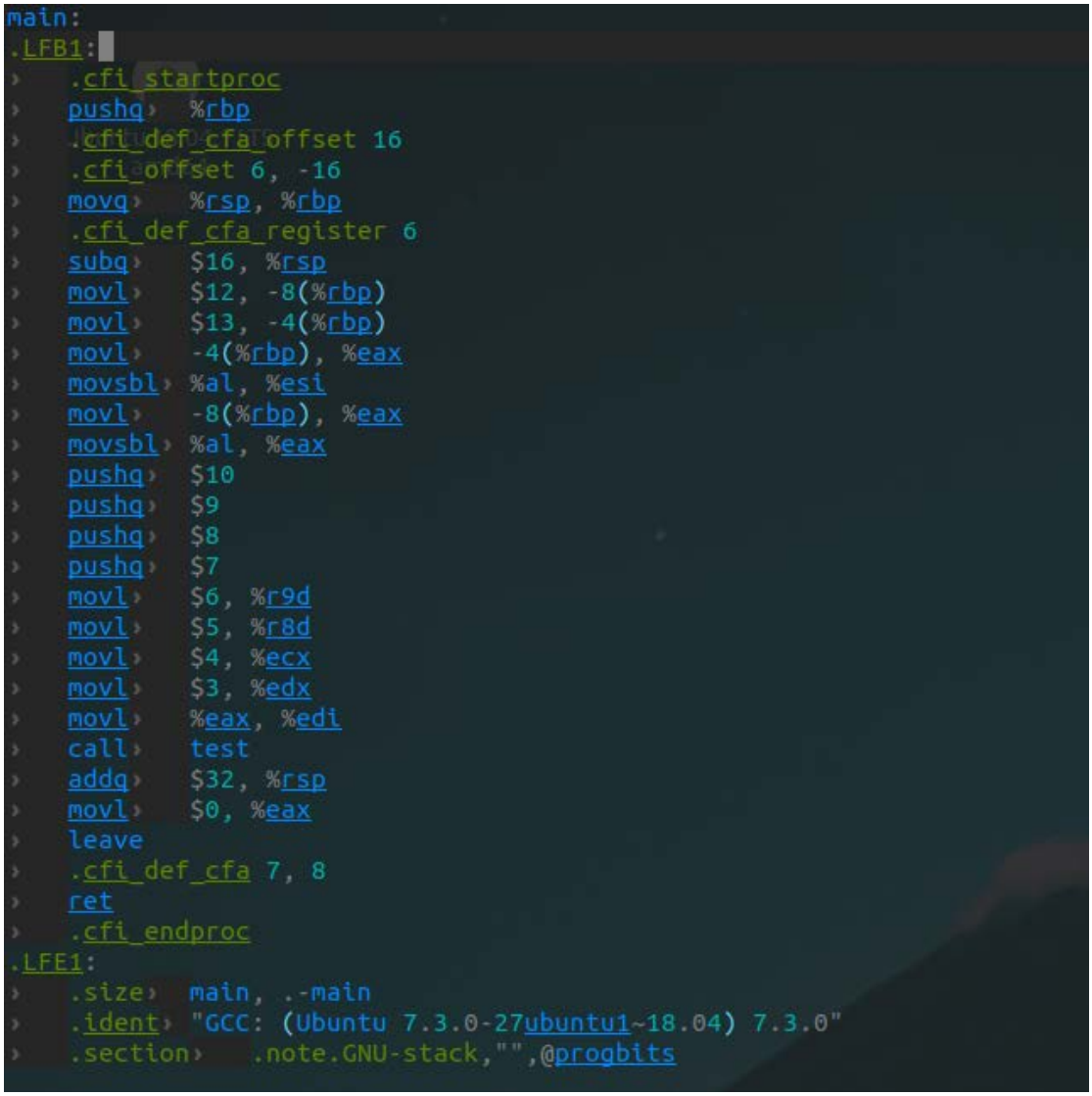
Test 函数



```
test:
.LFB0:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    pushq   %r12
    pushq   %rbx
    subq    $112, %rsp
    .cfi_offset 12, -24
    .cfi_offset 3, -32
    movl    %edi, %ebx
    movl    %esi, %r11d
    movl    %edx, %r10d
    movl    %ecx, %edi
    movl    %r8d, %r12d
    movl    16(%rbp), %esi
    movl    24(%rbp), %ecx
    movl    32(%rbp), %edx
    movl    40(%rbp), %eax
    movl    %ebx, %r8d
    movb    %r8b, -84(%rbp)
    movl    %r11d, %r8d
    movb    %r8b, -88(%rbp)
    movl    %r10d, %r8d
    movb    %r8b, -92(%rbp)
    movb    %dil, -96(%rbp)
    movl    %r12d, %edi
    movb    %dil, -100(%rbp)
    movl    %r9d, %edi
    movb    %dil, -104(%rbp)
    movb    %sil, -108(%rbp)
    movb    %cl, -112(%rbp)
    movb    %dl, -116(%rbp)
    movb    %al, -120(%rbp)
    movq    %fs:40, %rax
    movq    %rax, -24(%rbp)
```


xorl %eax, %eax
movl $1, -72(%rbp)
movl $2, -68(%rbp)
movsbl -84(%rbp), %eax
movl %eax, -64(%rbp)
movsbl -88(%rbp), %eax
movl %eax, -60(%rbp)
movsbl -92(%rbp), %eax
movl %eax, -56(%rbp)
movsbl -96(%rbp), %eax
movl %eax, -52(%rbp)
movsbl -100(%rbp), %eax
movl %eax, -48(%rbp)
movsbl -104(%rbp), %eax
movl %eax, -44(%rbp)
movsbl -108(%rbp), %eax
movl %eax, -40(%rbp)
movsbl -112(%rbp), %eax
movl %eax, -36(%rbp)
movsbl -116(%rbp), %eax
movl %eax, -32(%rbp)
movsbl -120(%rbp), %eax
movl %eax, -28(%rbp)
nop
movq -24(%rbp), %rax
xorq %fs:40, %rax
je .L2
call __stack_chk_fail@PLT
.L2:
addq $112, %rsp
popq %rbx
popq %r12
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size test, .-test
.globl main
.type main, @function

Main 函数



```
main:
.LFB1:
    .cfi_startproc
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register 6
    subq    $16, %rsp
    movl    $12, -8(%rbp)
    movl    $13, -4(%rbp)
    movl    -4(%rbp), %eax
    movsbl  %al, %esi
    movl    -8(%rbp), %eax
    movsbl  %al, %eax
    pushq   $10
    pushq   $9
    pushq   $8
    pushq   $7
    movl    $6, %r9d
    movl    $5, %r8d
    movl    $4, %ecx
    movl    $3, %edx
    movl    %eax, %edi
    call    test
    addq    $32, %rsp
    movl    $0, %eax
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE1:
    .size   main, .-main
    .ident  "GCC: (Ubuntu 7.3.0-27ubuntu1~18.04) 7.3.0"
    .section    .note.GNU-stack,"",@progbits
```

参数如何传递

64 位程序: 首先尝试将参数按照自左到右的顺序依次传入%edi,%esi,%edx,%ecx,%r8d.%r9d，如果 6 个寄存器装不下，就将多余的参数按照从右到左的顺序依次压入栈中。

局部变量如何实现

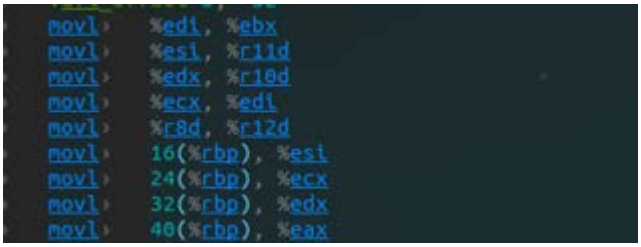
分析在函数内部，栈的操作:

1）将十个参数保存到寄存器和本程序栈中。取参数的顺序是从按照参数列表从左

到右的顺序。保存情况图示：

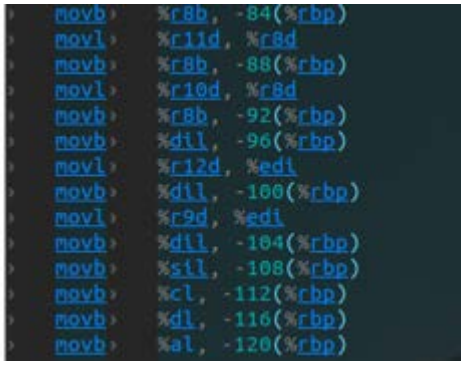
%ebx	%r11d	%r10d	%edi	%r12d	%9d	%esi	%ecx	%edx	%eax
a(12)	b(13)	c(3)	d(4)	e(5)	f(6)	g(7)	h(8)	i(9)	j(10)

可以看出保存参数的策略是：先用寄存器，如果寄存器 eax，edx，ebx，ecx，不能完全保存，则保存到栈中。

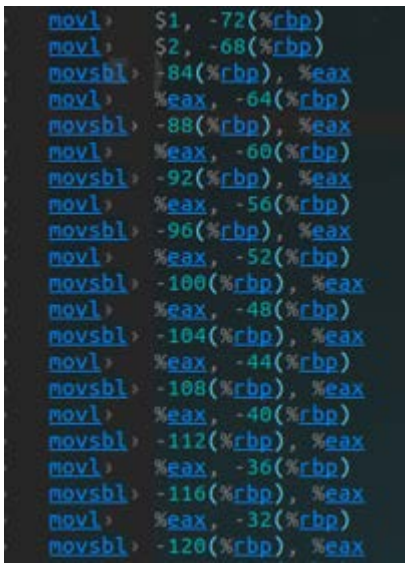


(<不使用立即数>差异分析：试过不是用立即数而是先将参数赋值到变量中然后调用函数的版本，结果是直接将所有参数保存到栈中一端连续的空间然后直接进行接下来的两步，因为不使用立即数所以这些数字在调用之前已经被压入栈中，对于 64 位版本需要先将存储在寄存器中的数存储到栈中的连续空间，然后通过一个中间变量把这段连续空间的值进行赋值到数组空间即可。而使用立即数的话，则需要先将立即数存储在寄存器和内存中，然后保存到栈中的一段连续空间)

2）然后将上述参数保存到栈中，-84(%rbp) ~ -120(%rbp)。mov 的方法是现将 mov 到 eax，然后将 al 赋值到对应的栈地址。



3）然后进行赋值，将 ax 和 bx 保存到-72(%rbp)和-68(%rbp)，数组在栈中是一段连续的空间，-64（%rbp）~-28（%rbp）。低地址在前高地址在后，按照顺序将存储在栈中-84(%rbp) ~ -120(%rbp)的数据 mov 到数组空间-64（%rbp）~-28（%rbp）。


movl $1, -72(%rbp)
movl $2, -68(%rbp)
movsbl -84(%rbp), %eax
movl %eax, -64(%rbp)
movsbl -88(%rbp), %eax
movl %eax, -60(%rbp)
movsbl -92(%rbp), %eax
movl %eax, -56(%rbp)
movsbl -96(%rbp), %eax
movl %eax, -52(%rbp)
movsbl -100(%rbp), %eax
movl %eax, -48(%rbp)
movsbl -104(%rbp), %eax
movl %eax, -44(%rbp)
movsbl -108(%rbp), %eax
movl %eax, -40(%rbp)
movsbl -112(%rbp), %eax
movl %eax, -36(%rbp)
movsbl -116(%rbp), %eax
movl %eax, -32(%rbp)
movsbl -120(%rbp), %eax