函数反汇编

http://www.ruanyifeng.com/blog/2018/01/assembly-language-primer.html

```
//————————————————————
int add_a_and_b(int a, int b) {
  return a + b;
}

int main() {
  return add_a_and_b(2, 3);
}
//————————————————————
```

gcc 将这个程序转成汇编语言：

```
//————————————————————
_add_a_and_b:        //为该函数建立一个新的帧。
  push   %ebx        //将 EBX 寄存器里面的值，写入_add_a_and_b这个帧。(ESP内地址-4)
  mov    %eax, [%esp+8]      //将ESP中地址+8得到之前的2，将2写入到EAX寄存器
  mov    %ebx, [%esp+12]     //将ESP中地址+12得到之前的3，将3写入到EBX寄存器
  add    %eax, %ebx      //将EAX中的“2”和EBX中的“3”相加，将结果“5”写入到EAX寄存器
  pop    %ebx    //取出 Stack 最近写入的值，再将这个值写回 EBX 寄存器。(ESP内地址+4)
  ret       //当前函数的帧被回收。回到刚才main函数中断的地方，继续往下执行。

_main:             //在 Stack 上为main建立一个帧
  push   3       //将3写入main这个帧。(ESP 寄存器里面的地址，减去4个字节)
  push   2       //将2写入main这个帧，紧贴着前面写入的3。(ESP 寄存器会再减去4个字节)
  call   _add_a_and_b      //调用add_a_and_b函数。main函数中断。
  add    %esp, 8     //ESP里地址 +8个字节，再写回 ESP 寄存器。（回收“2”，“3”）
  ret       //main函数运行结束，ret指令退出程序执行。
//————————————————————
```

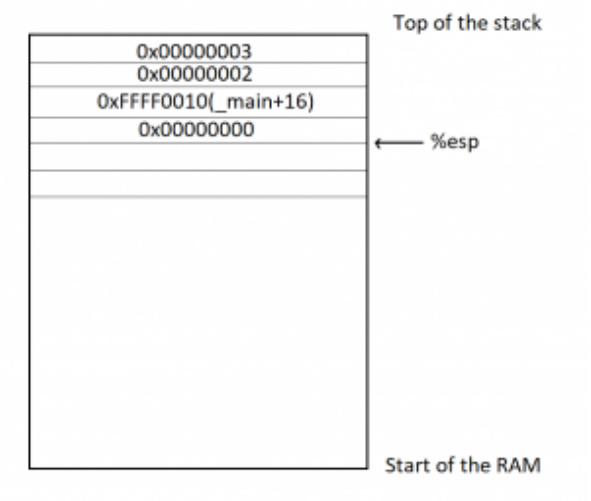
（ESP 寄存器有特定用途，保存当前 Stack 的地址）
（我们常常看到 32位 CPU、64位 CPU 这样的名称，其实指的就是寄存器的大小。32 位 CPU 的寄存器大小就是4个字节。）

```
_main:
```

程序从_main标签开始执行，这时会在 Stack 上为main建立一个帧，并将 Stack 所指向的地址，写入 ESP 寄存器。
后面如果有数据要写入main这个帧，就会写在 ESP 寄存器所保存的地址。

push　3
push指令用于将运算子放入 Stack，这里就是将3写入main这个帧。
(push指令其实有一个前置操作。它会先取出 ESP 寄存器里面的地址，将其减去4个字节，然后将新地址写入 ESP 寄存器。使用减法是因为 Stack 从高位向低位发展，4个字节则是因为3的类型是int，占用4个字节。得到新地址以后， 3 就会写入这个地址开始的四个字节。)

push　2
第二行也是一样，push指令将2写入main这个帧，位置紧贴着前面写入的3。这时，ESP 寄存器会再减去 4个字节（累计减去8）。



call　_add_a_and_b
call指令用来调用函数。调用add_a_and_b函数。这时，程序就会去找_add_a_and_b标签，并为该函数建立一个新的帧。

push　%ebx
将 EBX 寄存器里面的值，写入_add_a_and_b这个帧。这是因为后面要用到这个寄存器，就先把里面的值取出来，用完后再写回去。（这时，push指令会再将 ESP 寄存器里面的地址减去4个字节（累计减去12）。）

mov　%eax, [%esp+8]
mov指令用于将一个值写入某个寄存器。
这一行代码表示，先将 ESP 寄存器里面的地址加上8个字节，得到一个新的地址，然后按照这个地址在 Stack 取出数据。根据前面的步骤，可以推算出这里取出的是2，再将2写入 EAX 寄存器。

add　%eax, %ebx
add指令用于将两个运算子相加，并将结果写入第一个运算子。
上面的代码将 EAX 寄存器的值（即2）加上 EBX 寄存器的值（即3），得到结果5，

再将这个结果写入第一个运算子 EAX 寄存器。

```
pop    %ebx
```

pop指令用于取出 Stack 最近一个写入的值（即最低位地址的值），并将这个值写入运算子指定的位置。
上面的代码表示，取出 Stack 最近写入的值（即 EBX 寄存器的原始值），再将这个值写回 EBX 寄存器（因为加法已经做完了，EBX 寄存器用不到了）。
注意，pop指令还会将 ESP 寄存器里面的地址加4，即回收4个字节。

```
ret
```

ret指令用于终止当前函数的执行，将运行权交还给上层函数。也就是，当前函数的帧将被回收。
可以看到，该指令没有运算子。随着add_a_and_b函数终止执行，系统就回到刚才main函数中断的地方，继续往下执行。

```
add    %esp, 8
```

上面的代码表示，将 ESP 寄存器里面的地址，手动加上8个字节，再写回 ESP 寄存器。这是因为 ESP 寄存器的是 Stack 的写入开始地址，前面的pop操作已经回收了4个字节，这里再回收8个字节，等于全部回收。

```
ret
```

最后，main函数运行结束，ret指令退出程序执行。