ABI与内联汇编

王慧妍

why@nju.edu.cn

南京大学

软件学院

计算机软件研究所

本讲概述

I386，IA-32，x86-64，x64，。。。。。。

- 本讲内容
  - 一些背景
  - ABI
  - x86-64 体系结构与汇编语言
  - inline assembly

机器字长的发展

字长 (Word size)

In computing, a word is the natural unit of data used by a particular processor design. The *number of bits in a word* (the word size, word width, or word length)…

- 能直接进行整数/位运算的大小
- 指针的大小 (索引内存的范围)

8 位机 (6502)

- 16 bit PC 寄存器 (64 KiB 寻址能力，KiB 级内存，无乘除法/浮点数)
  - Apple II; Atari 2600; NES; ...

16位机（8086/8088）

- 我们需要更大的内存！更大的数据宽度！
  - 20 bit 地址线 (两个寄存器)
  - Intel 8086：经典中的经典→x86架构

32 位机 (Intel x86)

- 80386 处理器 4,096 倍的地址空间
  - 4 GiB 内存在 1980s 看起来是非常不可思议的

64 位机 (x86-64; AArch64; RV64; ...)

- 64 位地址空间能索引 17,179,869,184 GiB 内存
  - 我们的服务器有 128 GiB 内存
  - 目前看起来是非常够用的 (PML4)
    - 现在的处理器一般实现 48 bit 物理地址 (256 TiB)

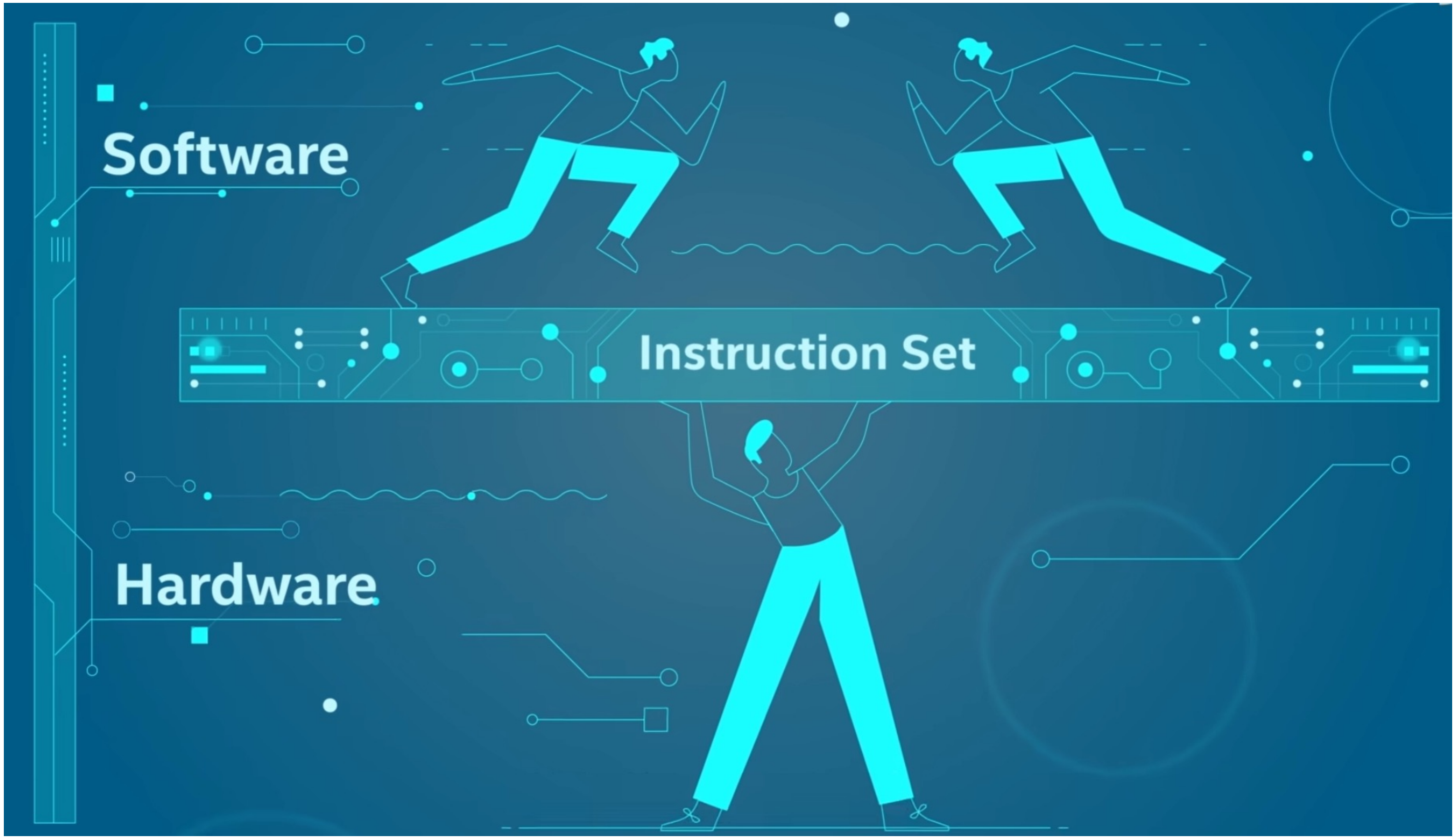
Fun Facts

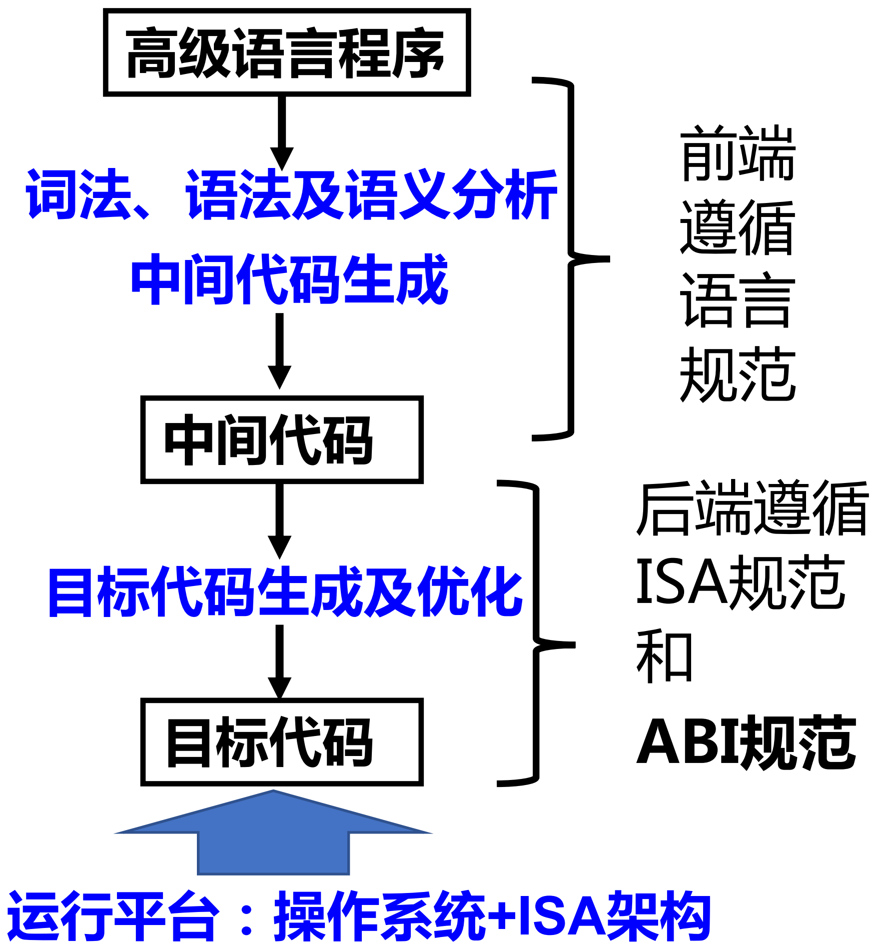
- `int`类型的长度
  - 8 bit computer: 8 bit
  - 16 bit computer: 16 bit
  - 32 bit computer: 32 bit
  - 64 bit computer: 32 bit
  - JVM (32/64 bit): 32 bit

- 在逻辑世界里描述日常世界，2,147,483,647 已经足够大了

处理器架构和指令集的演进

- x86，x64，amd64，…
- x86，ARM，MIPS，RISC，…

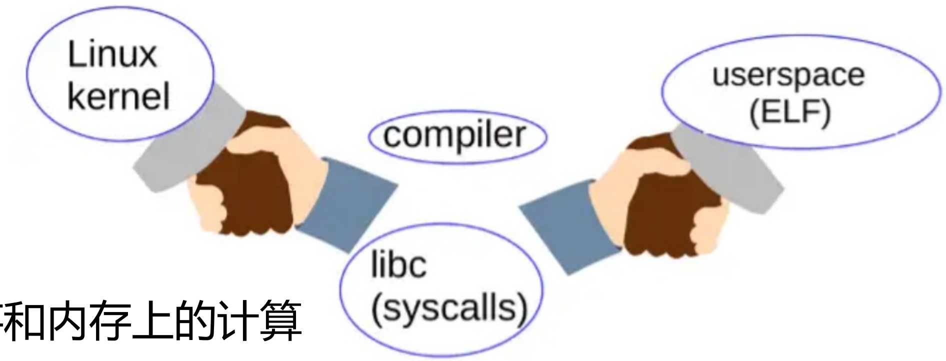



高级语言程序
词法、语法及语义分析
中间代码生成
中间代码
目标代码生成及优化
目标代码
运行平台：操作系统+ISA架构
前端
遵循
语言
规范
后端遵循
ISA规范
和
ABI规范

概念复习：ABI

程序的机器级表示

- 程序经历：.c → .o (编译)和 .o → a.out (链接)
  - 不同版本、不同编译器、不同语言的二进制文件都可以链接
    - 他们需要一个“共同语言”
- ABI（Application Binary Interface）
  - 区别于API (Application Programming Interface)
    - 程序源代码中的规范
  - 约定binary的行为（Linux ABI）
    - 二进制文件的格式
    - 函数调用、系统调用……
      - C语言规范只定义了运行时内存和内存上的计算
      - `printf`都无法实现，必须借助外部的库函数
    - 链接、加载的规范



System V ABI

Contents

Application Binary Interface (ABI)

- 例如：我们熟悉的`x86 calling convention`
    - `cdecl (Linux)`
    - `stdcall (Win32)`
    - 只要遵循标准的函数就可以互相调用

stdio.h ×

D: > gcc > mingw64 > x86_64-w64-mingw32 > include > stdio.h > fprintf(FILE *__restrict__, const char *__restrict__, ...)

```c
#pragma GCC diagnostic push
#pragma GCC diagnostic ignored "-Wshadow"
#endif
  __attribute__((__format__ (__MINGW_PRINTF_FORMAT, 2, 3))) __MINGW_ATTRIB_NONNULL(2)
  int __cdecl fprintf(FILE * __restrict__ _File,const char * __restrict__ _Format,...);
  __attribute__((__format__ (__MINGW_PRINTF_FORMAT, 1, 2))) __MINGW_ATTRIB_NONNULL(1)
  int __cdecl printf(const char * __restrict__ _Format,...);
  __attribute__((__format__ (__MINGW_PRINTF_FORMAT, 2, 3))) __MINGW_ATTRIB_NONNULL(2)
  int __cdecl sprintf(char * __restrict__ _Dest,const char * __restrict__ _Format,...) __M
```

例子：cdecl函数调用

- caller stack frame
  - 所有参数以数组的形式保存在堆栈上（所以才有“反序压栈”）
  - 然后是返回地址
  - 跳转到callee

- callee
  - EAX作为返回值
  - 其他寄存器都是callee save

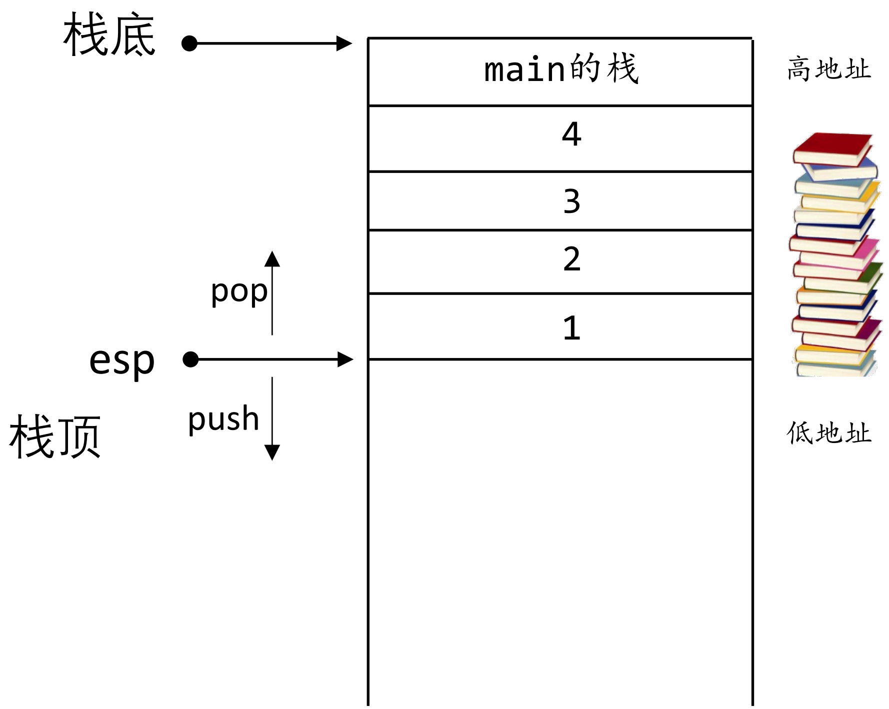
```
void bar(int *);
int foo(int x) {
   bar(&x);
   return x;
}
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```

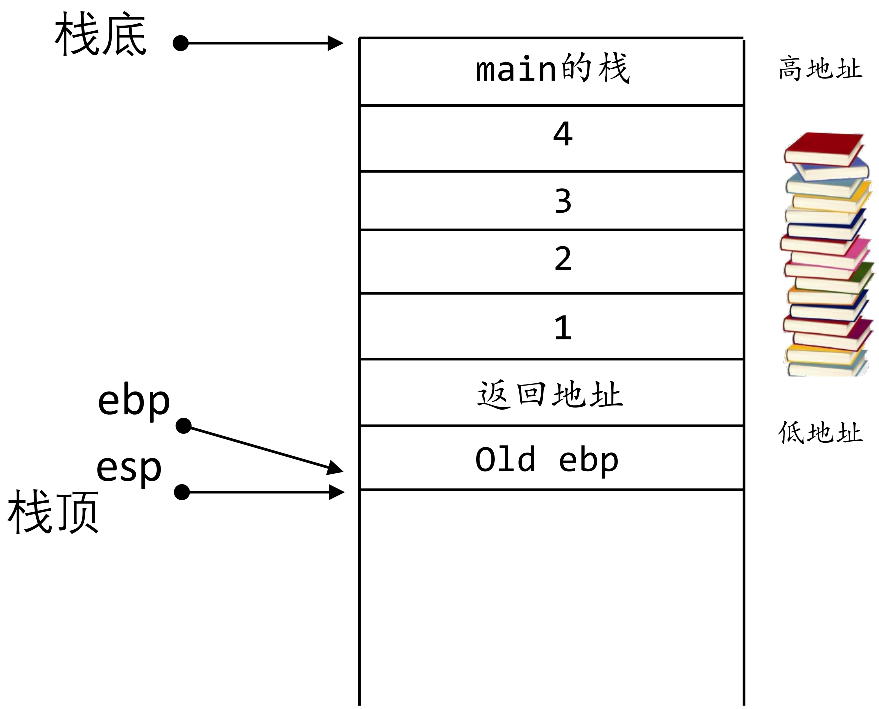


cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```



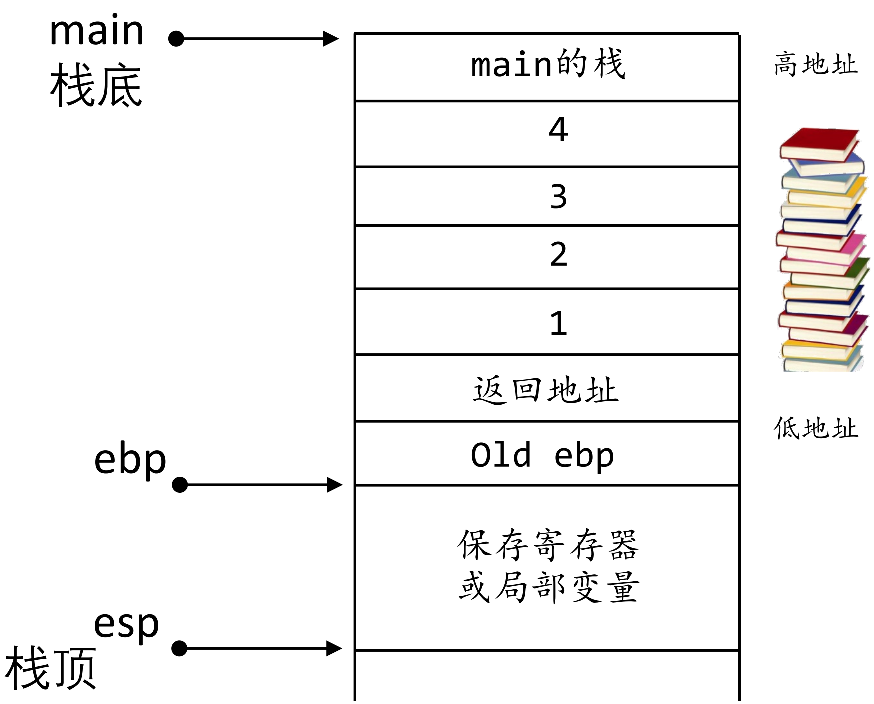

```
Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   55                      push   %ebp
   5:   89 e5                   mov    %esp,%ebp
   7:   e8 fc ff ff ff          call   8 <foo+0x8>
   c:   05 01 00 00 00          add    $0x1,%eax
  11:   8b 55 08                mov    0x8(%ebp),%edx
  14:   8b 45 0c                mov    0xc(%ebp),%eax
  17:   01 c2                   add    %eax,%edx
  19:   8b 45 10                mov    0x10(%ebp),%eax
  1c:   01 c2                   add    %eax,%edx
  1e:   8b 45 14                mov    0x14(%ebp),%eax
  21:   01 d0                   add    %edx,%eax
  23:   5d                      pop    %ebp
  24:   c3                      ret

00000025 <main>:
  25:   f3 0f 1e fb             endbr32
  29:   55                      push   %ebp
  2a:   89 e5                   mov    %esp,%ebp
  2c:   e8 fc ff ff ff          call   2d <main+0x8>
  31:   05 01 00 00 00          add    $0x1,%eax
  36:   6a 04                   push   $0x4
  38:   6a 03                   push   $0x3
  3a:   6a 02                   push   $0x2
  3c:   6a 01                   push   $0x1
  3e:   e8 fc ff ff ff          call   3f <main+0x1a>
  43:   83 c4 10                add    $0x10,%esp
  46:   b8 00 00 00 00          mov    $0x0,%eax
  4b:   c9                      leave
  4c:   c3                      ret
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```



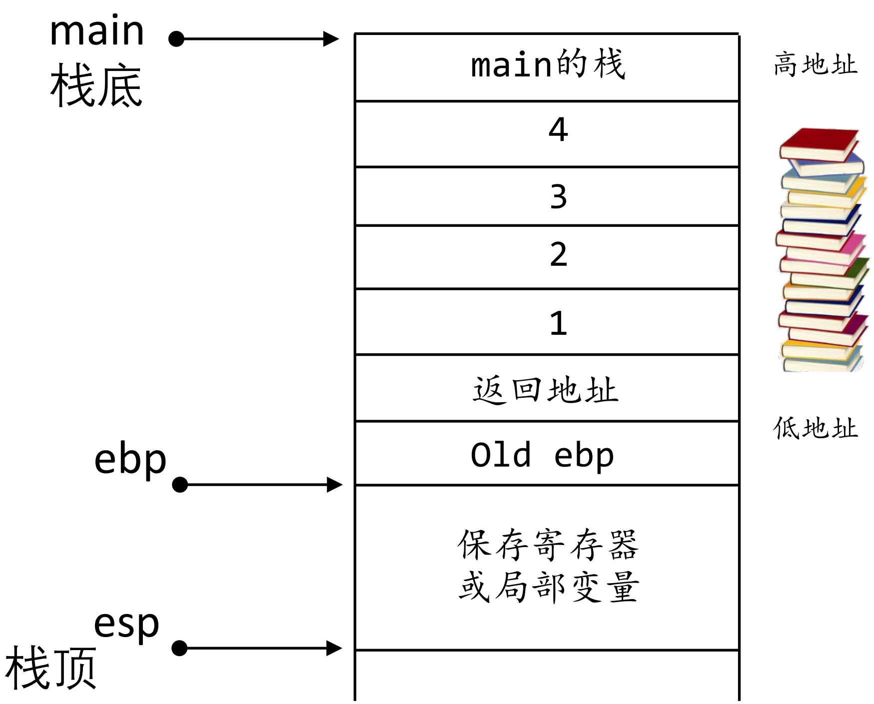

```
Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   55                      push   %ebp
   5:   89 e5                   mov    %esp,%ebp
   7:   e8 fc ff ff ff          call   8 <foo+0x8>
   c:   05 01 00 00 00          add    $0x1,%eax
  11:   8b 55 08                mov    0x8(%ebp),%edx
  14:   8b 45 0c                mov    0xc(%ebp),%eax
  17:   01 c2                   add    %eax,%edx
  19:   8b 45 10                mov    0x10(%ebp),%eax
  1c:   01 c2                   add    %eax,%edx
  1e:   8b 45 14                mov    0x14(%ebp),%eax
  21:   01 d0                   add    %edx,%eax
  23:   5d                      pop    %ebp
  24:   c3                      ret

00000025 <main>:
  25:   f3 0f 1e fb             endbr32
  29:   55                      push   %ebp
  2a:   89 e5                   mov    %esp,%ebp
  2c:   e8 fc ff ff ff          call   2d <main+0x8>
  31:   05 01 00 00 00          add    $0x1,%eax
  36:   6a 04                   push   $0x4
  38:   6a 03                   push   $0x3
  3a:   6a 02                   push   $0x2
  3c:   6a 01                   push   $0x1
  3e:   e8 fc ff ff ff          call   3f <main+0x1a>
  43:   83 c4 10                add    $0x10,%esp
  46:   b8 00 00 00 00          mov    $0x0,%eax
  4b:   c9                      leave
  4c:   c3                      ret
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```



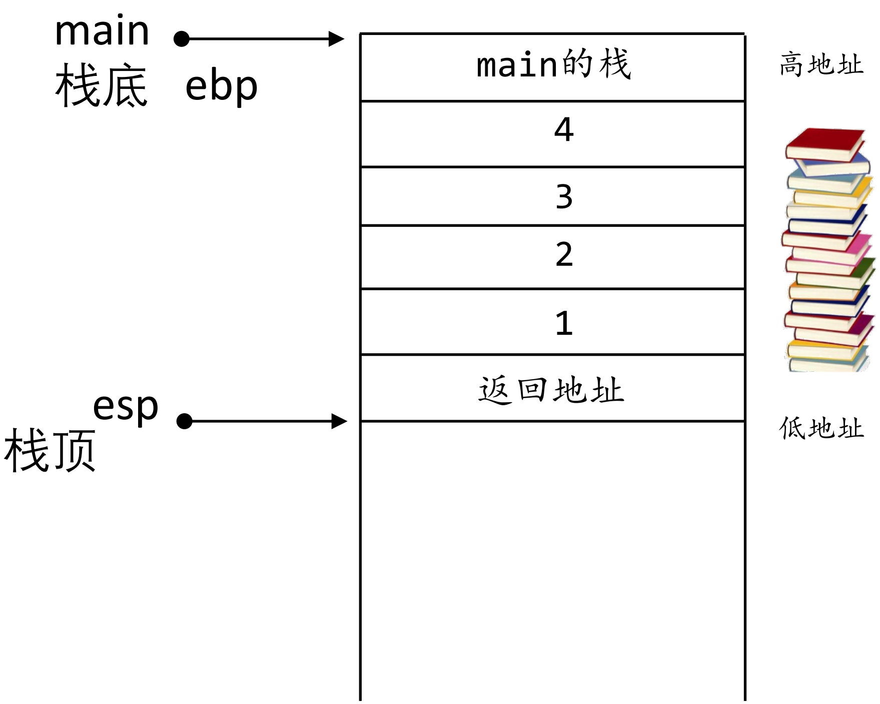

```
Disassembly of section .text:

00000000 <foo>:
   0:	f3 0f 1e fb          	endbr32
   4:	55                   	push   %ebp
   5:	89 e5                	mov    %esp,%ebp
   7:	e8 fc ff ff ff       	call   8 <foo+0x8>
   c:	05 01 00 00 00       	add    $0x1,%eax
  11:	8b 55 08             	mov    0x8(%ebp),%edx
  14:	8b 45 0c             	mov    0xc(%ebp),%eax
  17:	01 c2                	add    %eax,%edx
  19:	8b 45 10             	mov    0x10(%ebp),%eax
  1c:	01 c2                	add    %eax,%edx
  1e:	8b 45 14             	mov    0x14(%ebp),%eax
  21:	01 d0                	add    %edx,%eax
  23:	5d                   	pop    %ebp
  24:	c3                   	ret

00000025 <main>:
  25:	f3 0f 1e fb          	endbr32
  29:	55                   	push   %ebp
  2a:	89 e5                	mov    %esp,%ebp
  2c:	e8 fc ff ff ff       	call   2d <main+0x8>
  31:	05 01 00 00 00       	add    $0x1,%eax
  36:	6a 04                	push   $0x4
  38:	6a 03                	push   $0x3
  3a:	6a 02                	push   $0x2
  3c:	6a 01                	push   $0x1
  3e:	e8 fc ff ff ff       	call   3f <main+0x1a>
  43:	83 c4 10             	add    $0x10,%esp
  46:	b8 00 00 00 00       	mov    $0x0,%eax
  4b:	c9                   	leave
  4c:	c3                   	ret
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```



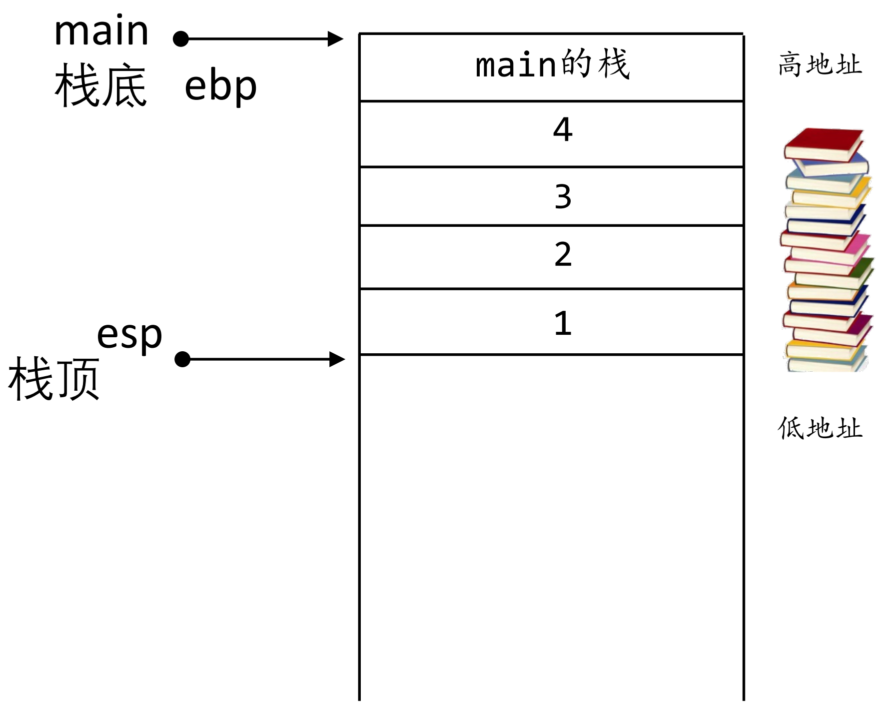

```
Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   55                      push   %ebp
   5:   89 e5                   mov    %esp,%ebp
   7:   e8 fc ff ff ff          call   8 <foo+0x8>
   c:   05 01 00 00 00          add    $0x1,%eax
  11:   8b 55 08                mov    0x8(%ebp),%edx
  14:   8b 45 0c                mov    0xc(%ebp),%eax
  17:   01 c2                   add    %eax,%edx
  19:   8b 45 10                mov    0x10(%ebp),%eax
  1c:   01 c2                   add    %eax,%edx
  1e:   8b 45 14                mov    0x14(%ebp),%eax
  21:   01 d0                   add    %edx,%eax
  23:   5d                      pop    %ebp
  24:   c3                      ret

00000025 <main>:
  25:   f3 0f 1e fb             endbr32
  29:   55                      push   %ebp
  2a:   89 e5                   mov    %esp,%ebp
  2c:   e8 fc ff ff ff          call   2d <main+0x8>
  31:   05 01 00 00 00          add    $0x1,%eax
  36:   6a 04                   push   $0x4
  38:   6a 03                   push   $0x3
  3a:   6a 02                   push   $0x2
  3c:   6a 01                   push   $0x1
  3e:   e8 fc ff ff ff          call   3f <main+0x1a>
  43:   83 c4 10                add    $0x10,%esp
  46:   b8 00 00 00 00          mov    $0x0,%eax
  4b:   c9                      leave
  4c:   c3                      ret
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```



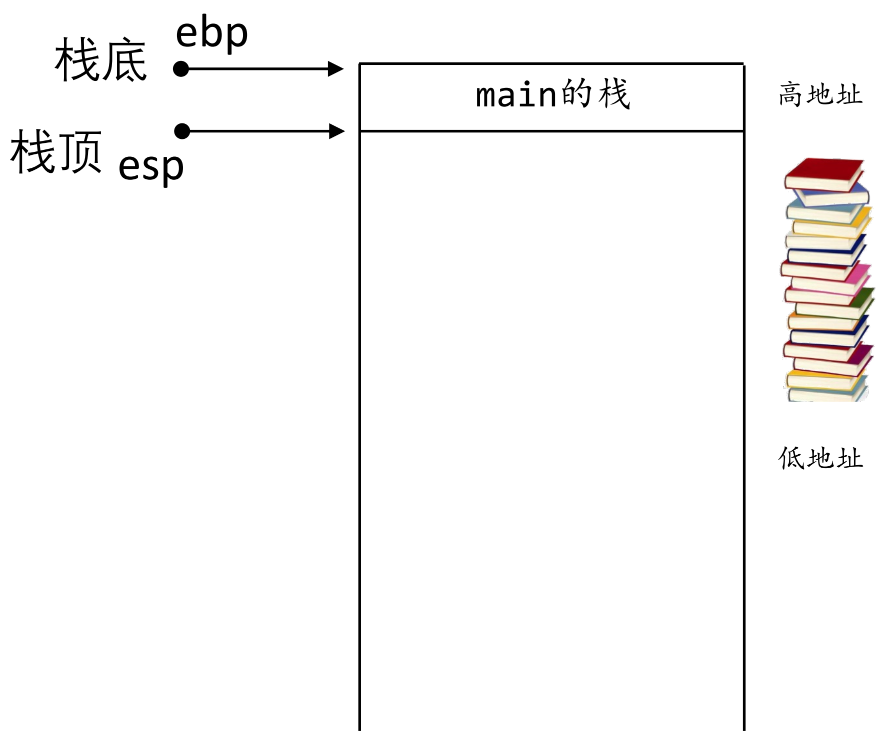

```
Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   55                      push   %ebp
   5:   89 e5                   mov    %esp,%ebp
   7:   e8 fc ff ff ff          call   8 <foo+0x8>
   c:   05 01 00 00 00          add    $0x1,%eax
  11:   8b 55 08                mov    0x8(%ebp),%edx
  14:   8b 45 0c                mov    0xc(%ebp),%eax
  17:   01 c2                   add    %eax,%edx
  19:   8b 45 10                mov    0x10(%ebp),%eax
  1c:   01 c2                   add    %eax,%edx
  1e:   8b 45 14                mov    0x14(%ebp),%eax
  21:   01 d0                   add    %edx,%eax
  23:   5d                      pop    %ebp
  24:   c3                      ret

00000025 <main>:
  25:   f3 0f 1e fb             endbr32
  29:   55                      push   %ebp
  2a:   89 e5                   mov    %esp,%ebp
  2c:   e8 fc ff ff ff          call   2d <main+0x8>
  31:   05 01 00 00 00          add    $0x1,%eax
  36:   6a 04                   push   $0x4
  38:   6a 03                   push   $0x3
  3a:   6a 02                   push   $0x2
  3c:   6a 01                   push   $0x1
  3e:   e8 fc ff ff ff          call   3f <main+0x1a>
  43:   83 c4 10                add    $0x10,%esp
  46:   b8 00 00 00 00          mov    $0x0,%eax
  4b:   c9                      leave
  4c:   c3                      ret
```

cdecl函数调用惯例

```c
int _cdecl foo(int m1, int m2, int m3, int m4);
```

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

```c
int main() { foo(1,2,3,4);}
```




```
Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   55                      push   %ebp
   5:   89 e5                   mov    %esp,%ebp
   7:   e8 fc ff ff ff          call   8 <foo+0x8>
   c:   05 01 00 00 00          add    $0x1,%eax
  11:   8b 55 08                mov    0x8(%ebp),%edx
  14:   8b 45 0c                mov    0xc(%ebp),%eax
  17:   01 c2                   add    %eax,%edx
  19:   8b 45 10                mov    0x10(%ebp),%eax
  1c:   01 c2                   add    %eax,%edx
  1e:   8b 45 14                mov    0x14(%ebp),%eax
  21:   01 d0                   add    %edx,%eax
  23:   5d                      pop    %ebp
  24:   c3                      ret

00000025 <main>:
  25:   f3 0f 1e fb             endbr32
  29:   55                      push   %ebp
  2a:   89 e5                   mov    %esp,%ebp
  2c:   e8 fc ff ff ff          call   2d <main+0x8>
  31:   05 01 00 00 00          add    $0x1,%eax
  36:   6a 04                   push   $0x4
  38:   6a 03                   push   $0x3
  3a:   6a 02                   push   $0x2
  3c:   6a 01                   push   $0x1
  3e:   e8 fc ff ff ff          call   3f <main+0x1a>
  43:   83 c4 10                add    $0x10,%esp
  46:   b8 00 00 00 00          mov    $0x0,%eax
  4b:   c9                      leave
  4c:   c3                      ret
```

参数传递	出栈方	名字修饰
从右往左顺序压参数入栈	函数调用方	函数名前加下划线

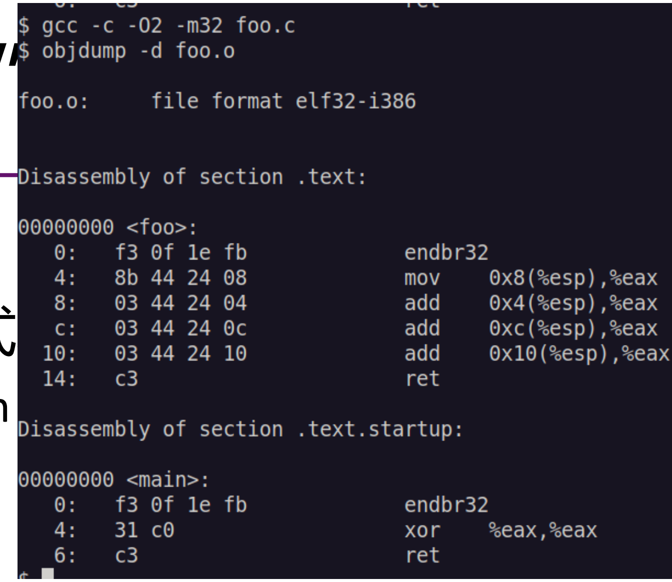
调用惯例

```c
int foo(int m1, int m2, int m3, int m4) {
    return m1 + m2 + m3 + m4;
}
```

	参数传递	出栈方	名字修饰
cdecl	从右往左顺序压参数入栈	函数调用方	下划线+函数名
stdcall	从右往左顺序压参数入栈	函数本身	下划线+函数名+@+参数字节数
fastcall	头两个DWORD类型或更少字节参数放入寄存器，其他参数从右往左顺序入栈	函数本身	@+函数名+@+参数字节数
pascal	从左往右顺序压参数入栈	函数本身	较复杂，见pascal文档

_foo@16

阅读汇编代码：“符号执行”

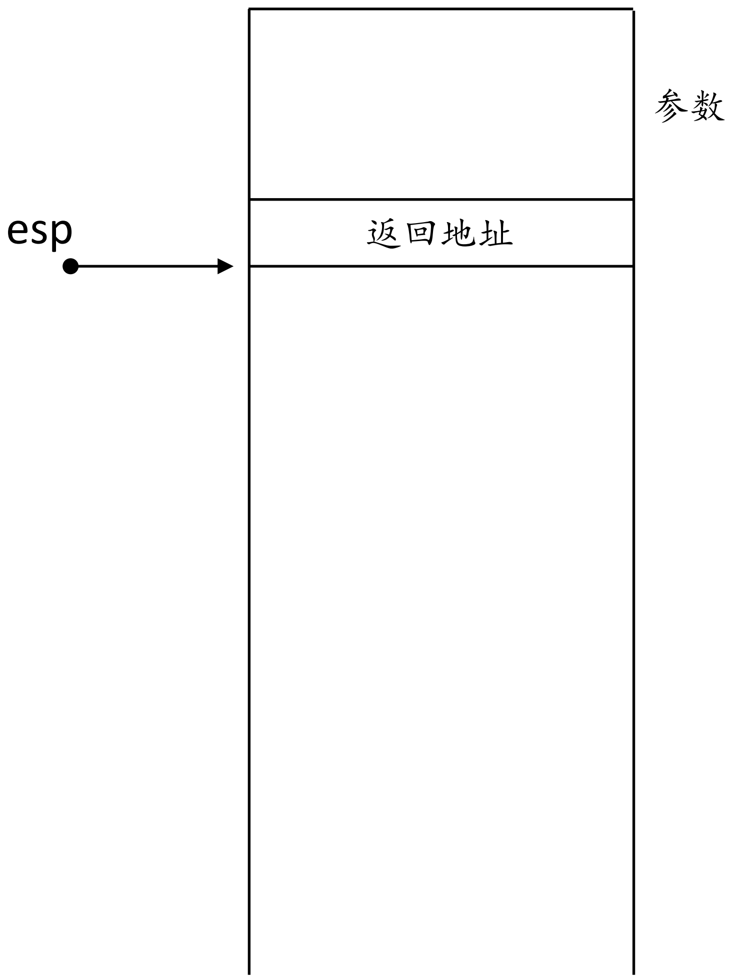


- 试着把内存/寄存器用数学符号表示出来
  - 然后假想地 “单步执行” 程序，用符号公式
    - James C. King. Symbolic execution and program *ACM*, 19(7), 1976.

- 编译选项：-m32 -O2 -fno-pic(便于大家理解)

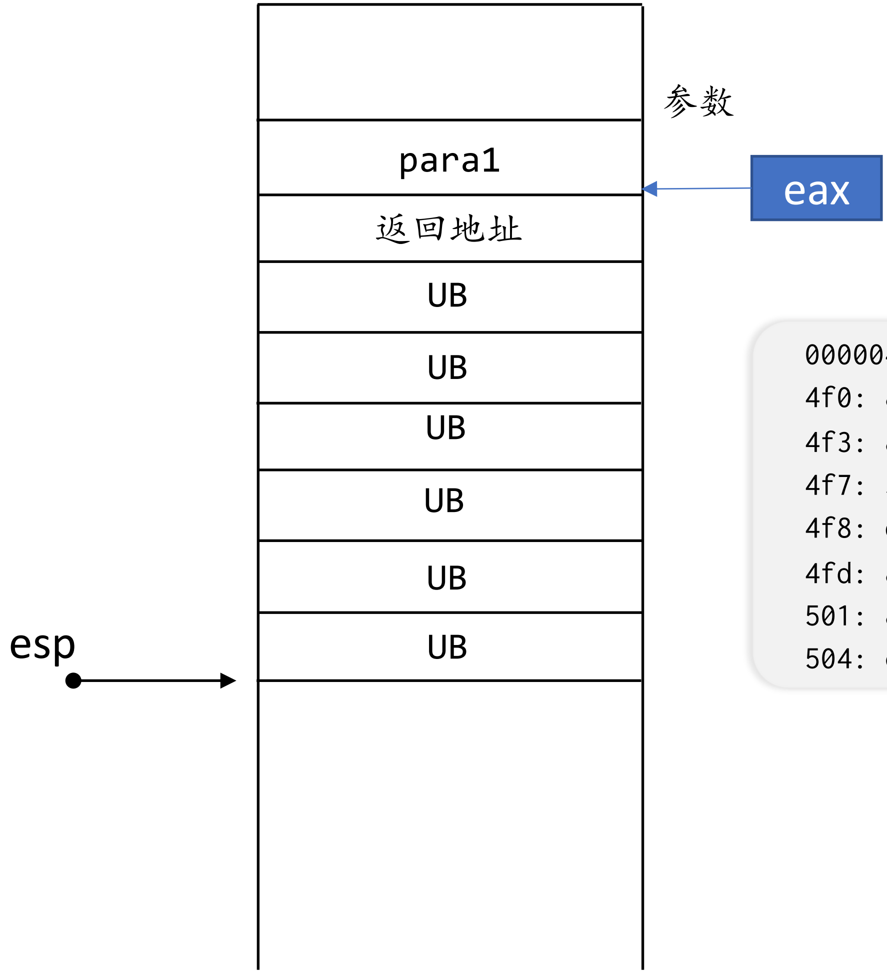
```
000004f0 <foo>:
4f0: 83 ec 18               sub $0x18,%esp
4f3: 8d 44 24 1c            lea 0x1c(%esp),%eax
4f7: 50                     push %eax
4f8: e8 13 00 00 00         call 510 <bar>
4fd: 8b 44 24 20            mov 0x20(%esp),%eax
501: 83 c4 1c               add $0x1c,%esp
504: c3                     ret
```

假装单步调试



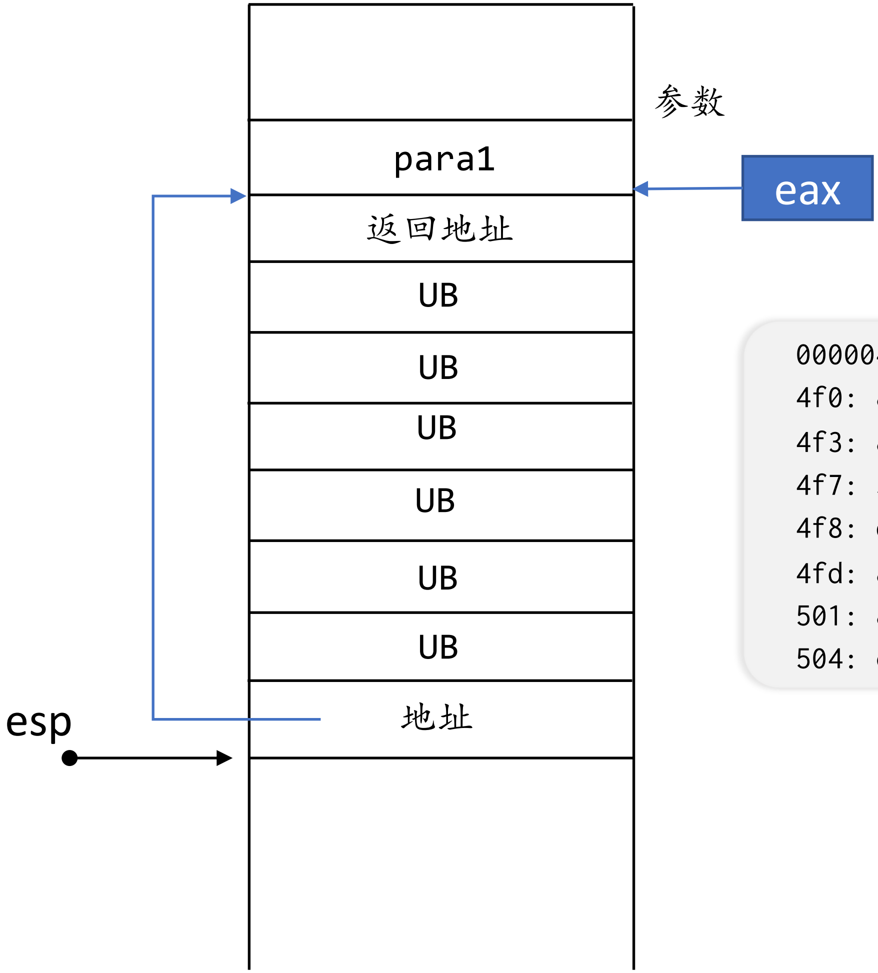

```
000004f0 <foo>:
4f0: 83 ec 18                       sub $0x18,%esp
4f3: 8d 44 24 1c                    lea 0x1c(%esp),%eax
4f7: 50                             push %eax
4f8: e8 13 00 00 00                 call 510 <bar>
4fd: 8b 44 24 20                    mov 0x20(%esp),%eax
501: 83 c4 1c                       add $0x1c,%esp
504: c3                             ret
```

假装单步调试



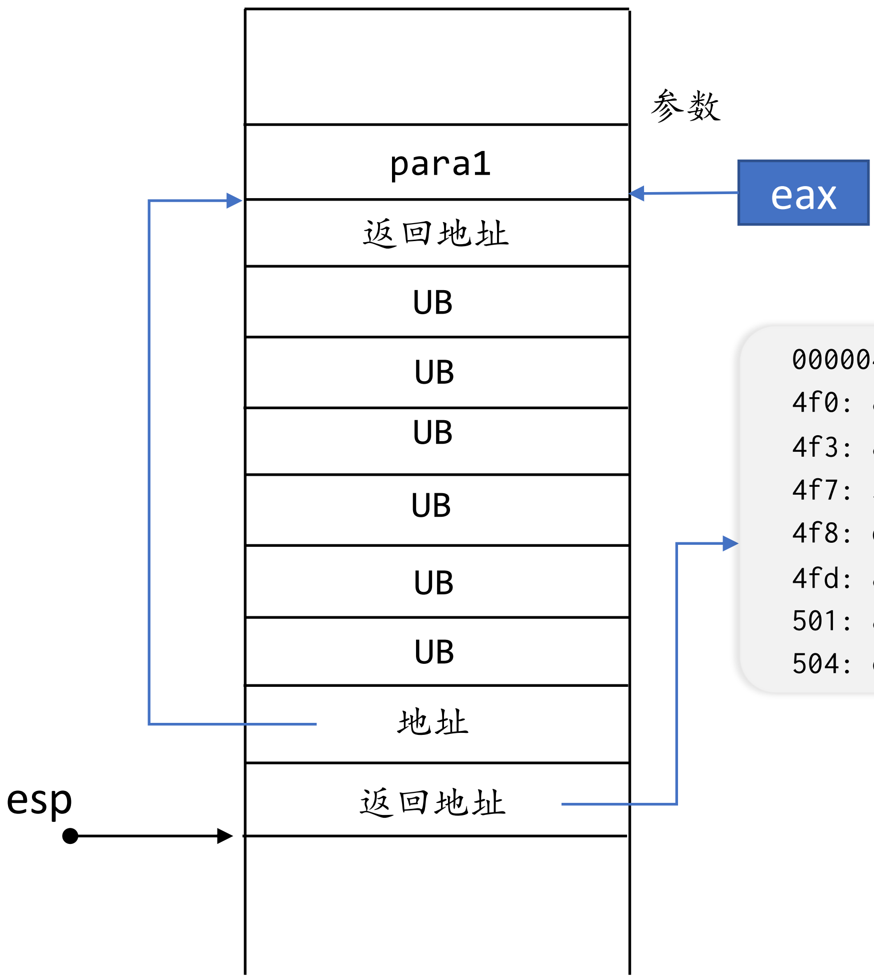

```
000004f0 <foo>:
4f0: 83 ec 18                  sub $0x18,%esp
4f3: 8d 44 24 1c               lea 0x1c(%esp),%eax
4f7: 50                        push %eax
4f8: e8 13 00 00 00            call 510 <bar>
4fd: 8b 44 24 20               mov 0x20(%esp),%eax
501: 83 c4 1c                  add $0x1c,%esp
504: c3                        ret
```

假装单步调试




```
000004f0 <foo>:
4f0: 83 ec 18                sub $0x18,%esp
4f3: 8d 44 24 1c             lea 0x1c(%esp),%eax
4f7: 50                      push %eax
4f8: e8 13 00 00 00          call 510 <bar>
4fd: 8b 44 24 20             mov 0x20(%esp),%eax
501: 83 c4 1c                add $0x1c,%esp
504: c3                      ret
```

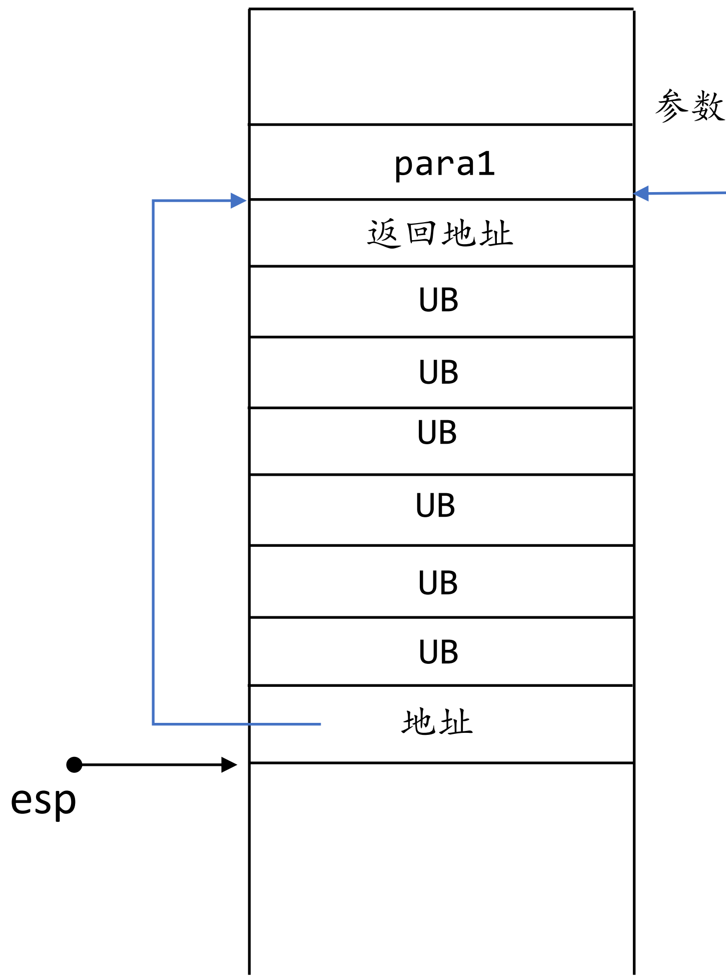
假装单步调试




```
000004f0 <foo>:
4f0: 83 ec 18                   sub $0x18,%esp
4f3: 8d 44 24 1c                lea 0x1c(%esp),%eax
4f7: 50                         push %eax
4f8: e8 13 00 00 00             call 510 <bar>
4fd: 8b 44 24 20                mov 0x20(%esp),%eax
501: 83 c4 1c                   add $0x1c,%esp
504: c3                         ret
```

```
foo (int x){
    bar(&x);
}
```

假装单步调试

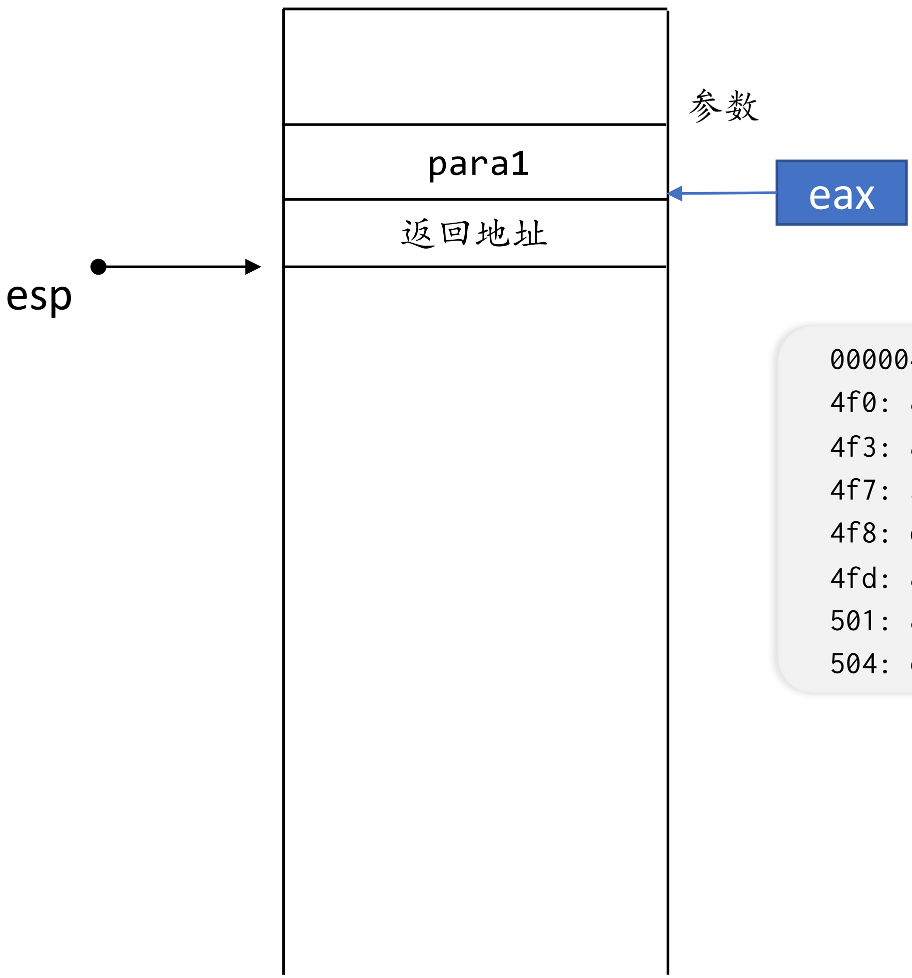


eax

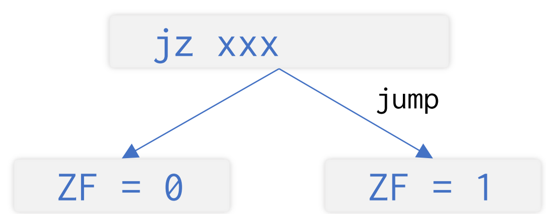
```
000004f0 <foo>:
4f0: 83 ec 18                 sub $0x18,%esp
4f3: 8d 44 24 1c              lea 0x1c(%esp),%eax
4f7: 50                       push %eax
4f8: e8 13 00 00 00           call 510 <bar>
4fd: 8b 44 24 20              mov 0x20(%esp),%eax
501: 83 c4 1c                 add $0x1c,%esp
504: c3                       ret
```

```
foo (int x){
    bar(&x);
}
```

假装单步调试




```
000004f0 <foo>:
4f0: 83 ec 18                sub $0x18,%esp
4f3: 8d 44 24 1c             lea 0x1c(%esp),%eax
4f7: 50                      push %eax
4f8: e8 13 00 00 00          call 510 <bar>
4fd: 8b 44 24 20             mov 0x20(%esp),%eax
501: 83 c4 1c                add $0x1c,%esp
504: c3                      ret
```



阅读汇编代码：“符号执行”

- 试着把内存/寄存器用数学符号表示出来
  - 然后假想地“单步执行”程序，用符号公式
    - James C. King. Symbolic execution and program *ACM*, 19(7), 1976.

```
$ gcc -c -O2 -m32 foo.c
$ objdump -d foo.o

foo.o:     file format elf32-i386


Disassembly of section .text:

00000000 <foo>:
   0:   f3 0f 1e fb             endbr32
   4:   8b 44 24 08             mov    0x8(%esp),%eax
   8:   03 44 24 04             add    0x4(%esp),%eax
   c:   03 44 24 0c             add    0xc(%esp),%eax
  10:   03 44 24 10             add    0x10(%esp),%eax
  14:   c3                      ret

Disassembly of section .text.startup:

00000000 <main>:
   0:   f3 0f 1e fb             endbr32
   4:   31 c0                   xor    %eax,%eax
   6:   c3                      ret
```

- 编译选项：-m32 -O2 -fno-pic(便于大家理解)

```
000004f0 <foo>:
4f0: 83 ec 18           sub $0x18,%esp
4f3: 8d 44 24 1c        lea 0x1c(%esp),%eax
4f7: 50                 push %eax
4f8: e8 13 00 00 00     call 510 <bar>
4fd: 8b 44 24 20        mov 0x20(%esp),%eax
501: 83 c4 1c           add $0x1c,%esp
504: c3                 ret
```

x86-64：寄存器与函数调用

寄存器 (1)：继承自 IA32

用途	64b	低32b	低16b	低8b	8-15b
返回值	%rax	%eax	%ax	%al	%ah
调用者保存	%rbx	%ebx	%bx	%bl	%bh
参数4	%rcx	%ecx	%cx	%cl	%ch
参数3	%rdx	%edx	%dx	%dl	%dh
参数2	%rsi	%esi	%si	%sil	
参数1	%rdi	%edi	%di	%dil	
调用者保存	%rbp	%ebp	%bp	%bpl	
栈顶	%rsp	%esp	%sp	%spl	

寄存器(2): 新增加的寄存器

- x86-64 扩充了很多寄存器！
  - 于是再也不用像 IA32 那样，用堆栈传递参数了！！

用途	64b	低32b	低16b	低8b	8-15b
参数5	%r8	%r8d	%r8w	%r8b	
参数6	%r9	%r9d	%r9w	%r9b	
调用者保存	%r10	%r10d	%r10w	%r10b	
链接	%r11	%r11d	%r11w	%r11b	
C unsued	%r12	%r12d	%r12w	%r12b	
调用者保存	%r13	%r13d	%r13w	%r13b	
调用者保存	%r14	%r14d	%r14w	%r14b	
调用者保存	%r15	%r15d	%r15w	%r15b	(没有)

A + B in x86-64

```c
int f(int a, int b) {
    return a + b;
}
```

```
00000510 <add_32>:
 510: 8b 44 24 08       mov 0x8(%esp),%eax
 514: 03 44 24 04       add 0x4(%esp),%eax
 518: c3                ret
```

```
0000000000000630 <add_64>:
 630: 8d 04 37          lea (%rdi,%rsi,1),%eax
 633: c3                retq
```

max in x86-64

```c
int max(int a, int b) {
   if (a > b) return a;
   else return b;
}
```

```
00000514 <max_32>:
 514: 8b 44 24 04        mov 0x4(%esp),%eax
 518: 3b 44 24 08        cmp 0x8(%esp),%eax
 51c: 7d 04              jg 522 <max+0xe>
 51e: 8b 44 24 08        mov 0x8(%esp),%eax
 522: c3                 ret
```

```
movl   4(%esp), %eax
movl   8(%esp), %edx
cmpl   %edx, %eax
cmovl  %edx, %eax
ret
```

```
0000000000000640 <max_64>:
 640: 39 f7              cmp %esi,%edi
 642: 89 f0              mov %esi,%eax
 644: 0f 4d c7           cmovge %edi,%eax
 647: c3                 retq
```

使用寄存器传递函数参数：优势

- 支持 6 个参数的传递：rdi, rsi, rdx, rcx, r8, r9
  - callee 可以随意修改这 6 个寄存器的值
  - 编译器有了更大的调度空间 (大幅减少堆栈内存访问)

- 例子：

```
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # XXXXXX <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```

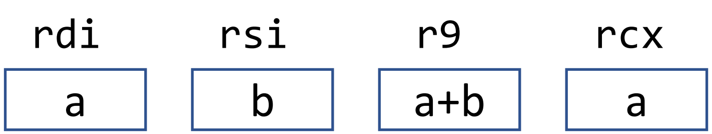
201010

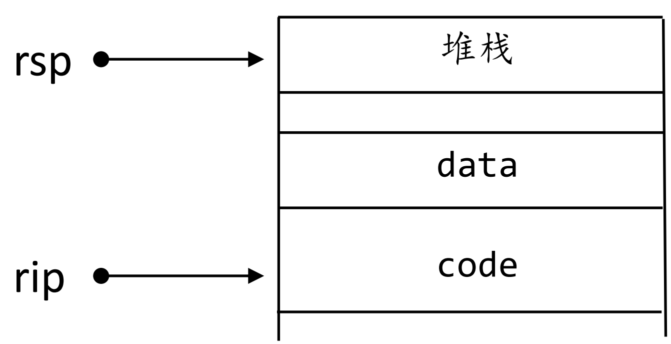
rdi	rsi	r9	rcx
a	b	a+b	a

```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # 201010 <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```



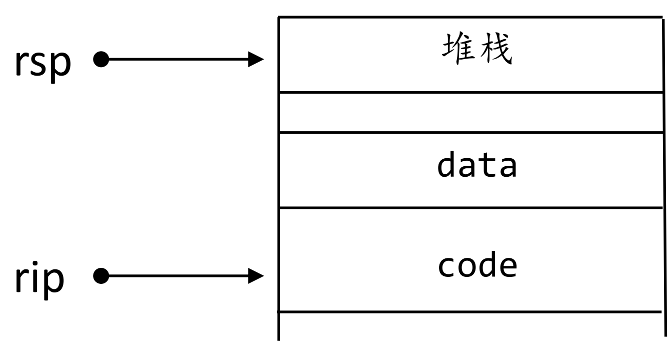


```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # 201010 <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```

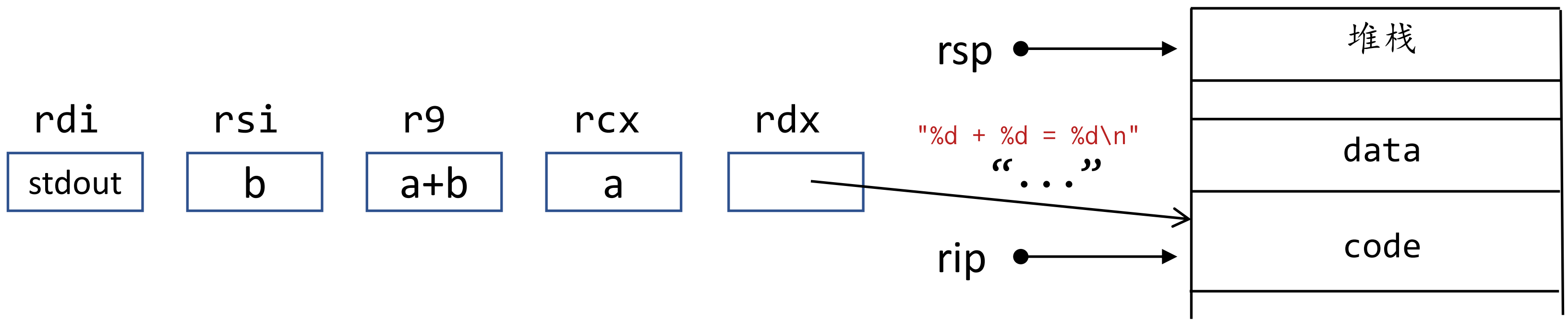
rdi	rsi	r9	rcx
stdout	b	a+b	a



```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

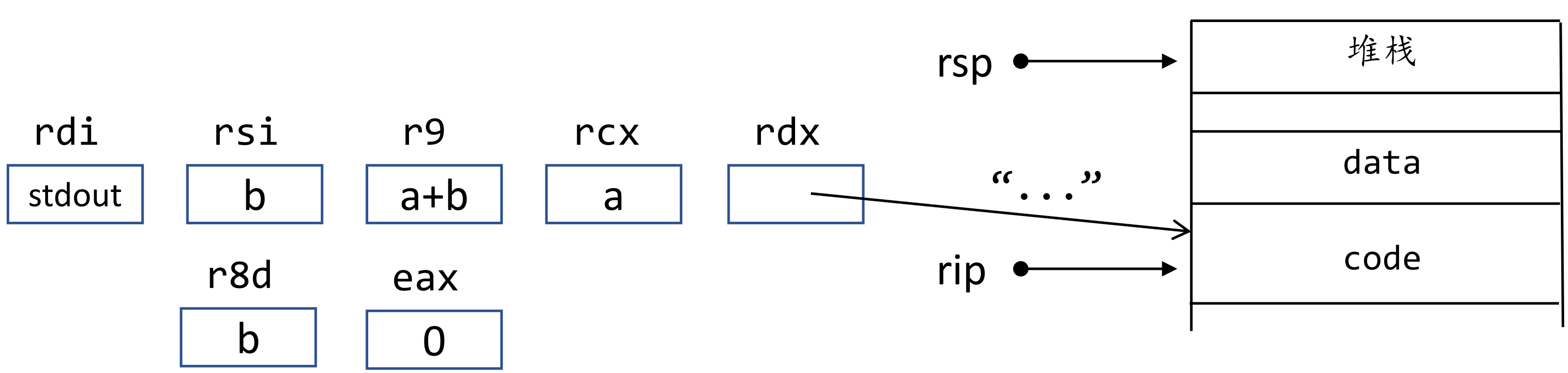
```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # 201010 <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```



```c
void plus(int a, int b) {
   fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

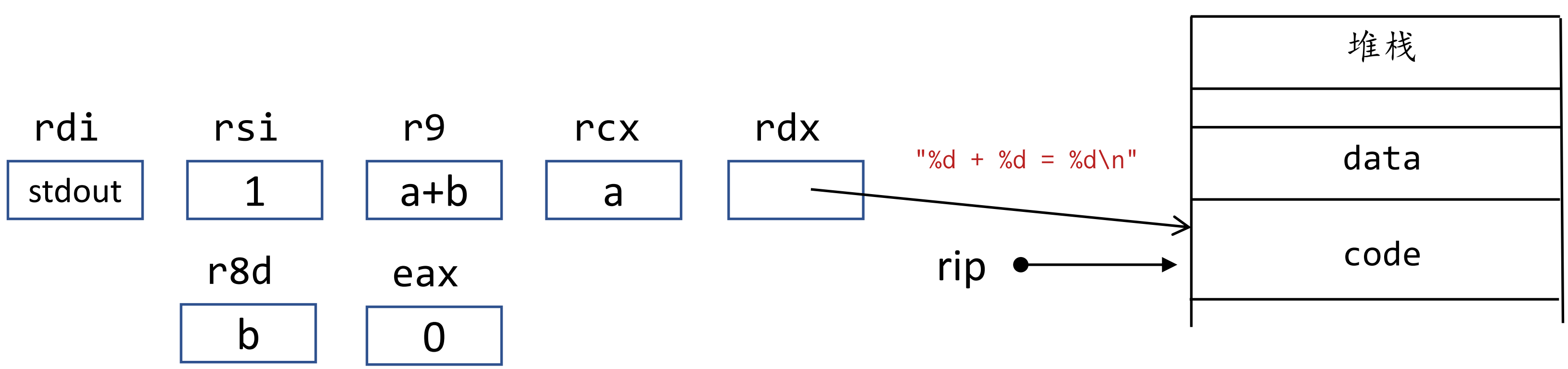
```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # 201010 <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```



```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

- 实际调用的是 __fprintf_chk@plt
  - 需要传递的参数：stdout, %d + %d = %d\n, a, b, a + b
  - calling convention: rdi, rsi, rdx, rcx, r8, r9

```
0000000000000700 <plus>:
 700: 44 8d 0c 37             lea (%rdi,%rsi,1),%r9d
 704: 89 f9                   mov %edi,%ecx
 706: 48 8b 3d 03 09 20 00    mov 0x200903(%rip),%rdi # 201010 <stdout@@GLIBC_2.2.5>
 70d: 48 8d 15 b0 00 00 00    lea 0xb0(%rip),%rdx # 7c4 <_IO_stdin_used+0x4>
 714: 41 89 f0                mov %esi,%r8d
 717: 31 c0                   xor %eax,%eax
 719: be 01 00 00 00          mov $0x1,%esi
 71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```



一些更多的分析

- plus的最后一条指令

```
71e: e9 5d fe ff ff          jmpq 580 <__fprintf_chk@plt>
```

  - 并不是调用的printf，而是调用的有堆栈检查的版本
    - 准备参数时有 mov $0x1,%esi

  - 直接jmp是因为函数末尾有一条ret指令
    - 借用了__fprintf_chk@plt的ret指令返回到plus的调用者
    - 如果有返回值，就会生成call指令；如果plus返回printf的结果，依然是jmp
    - 省的不止是一条指令
      - 连续的ret对分支预测是很大的挑战

对比32位printf

- 好读，但指令执行起来会稍慢一些

```c
void plus(int a, int b) {
    fprintf(stdout, "%d + %d = %d\n", a, b, a + b);
}
```

```
000005b4 <plus>:
 5b4: 83 ec 14              sub    $0x14,%esp
 5b7: 8b 44 24 18           mov    0x18(%esp),%eax
 5bb: 8b 54 24 1c           mov    0x1c(%esp),%edx
 5bf: 8d 0c 10              lea    (%eax,%edx,1),%ecx
 5c2: 51                    push   %ecx
 5c3: 52                    push   %edx
 5c4: 50                    push   %eax
 5c5: 68 60 06 00 00        push   $0x660
 5ca: 6a 01                 push   $0x1
 5cc: ff 35 00 00 00 00     pushl  0x0
 5d2: e8 fc ff ff ff        call   5d3 <plus+0x1f>
 5d7: 83 c4 2c              add    $0x2c,%esp
 5da: c3                    ret
 5db: 90                    nop
```

一个有趣的小问题

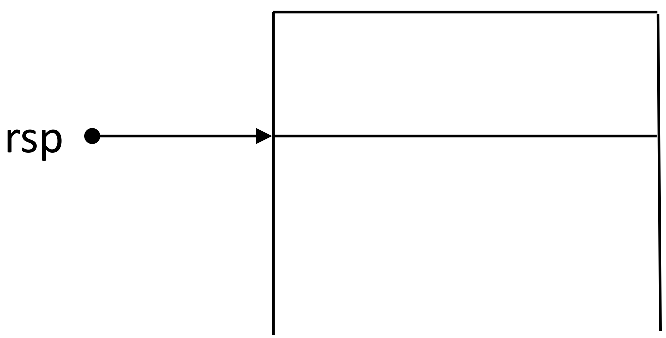
- x86-64 按寄存器传递参数
  - `void f(int x) {… &x …}` 会发生什么？
- 编译器会给参数分配内存，保证后续访问合法
  - 给编译器带来了轻微的负担
  - 但编译器并不觉得这是负担

```c
void bar(int *);
int foo(int x) {
    bar(&x);
    return x;
}
```

假装单步调试

```c
void bar(int *);
int foo(int x) {
    bar(&x);
    return x;
}
```

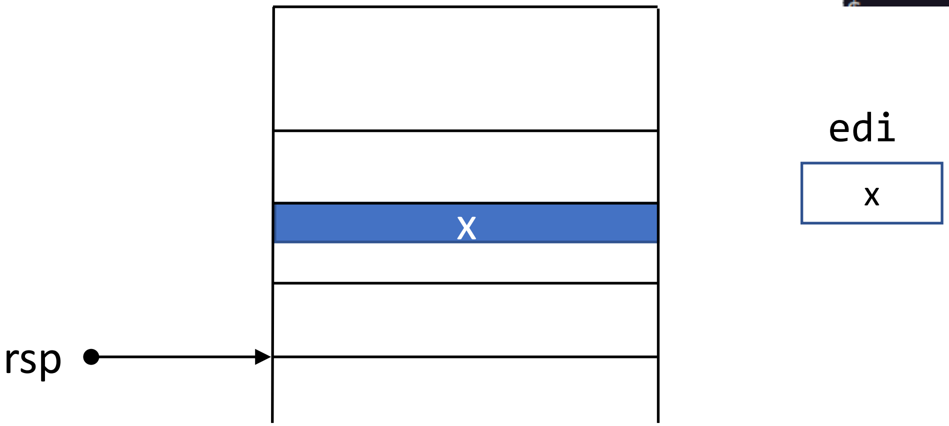
```
$ gcc -c -O2 foo.c
$ objdump -d foo.o

foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   48 83 ec 18             sub    $0x18,%rsp
   8:   89 7c 24 0c             mov    %edi,0xc(%rsp)
   c:   48 8d 7c 24 0c          lea    0xc(%rsp),%rdi
  11:   e8 00 00 00 00          call   16 <foo+0x16>
  16:   8b 44 24 0c             mov    0xc(%rsp),%eax
  1a:   48 83 c4 18             add    $0x18,%rsp
  1e:   c3                      ret
```



假装单步调试

```c
void bar(int *);
int foo(int x) {
    bar(&x);
    return x;
}
```

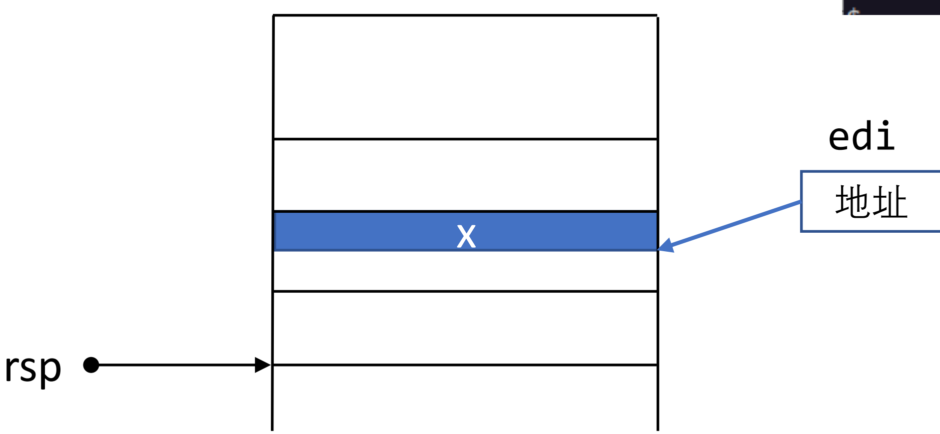
```
$ gcc -c -O2 foo.c
$ objdump -d foo.o

foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:	f3 0f 1e fa          	endbr64
   4:	48 83 ec 18          	sub    $0x18,%rsp
   8:	89 7c 24 0c          	mov    %edi,0xc(%rsp)
   c:	48 8d 7c 24 0c       	lea    0xc(%rsp),%rdi
  11:	e8 00 00 00 00       	call   16 <foo+0x16>
  16:	8b 44 24 0c          	mov    0xc(%rsp),%eax
  1a:	48 83 c4 18          	add    $0x18,%rsp
  1e:	c3                   	ret
```



假装单步调试

```c
void bar(int *);
int foo(int x) {
    bar(&x);
    return x;
}
```

```
$ gcc -c -O2 foo.c
$ objdump -d foo.o

foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:	f3 0f 1e fa          	endbr64
   4:	48 83 ec 18          	sub    $0x18,%rsp
   8:	89 7c 24 0c          	mov    %edi,0xc(%rsp)
   c:	48 8d 7c 24 0c       	lea    0xc(%rsp),%rdi
  11:	e8 00 00 00 00       	call   16 <foo+0x16>
  16:	8b 44 24 0c          	mov    0xc(%rsp),%eax
  1a:	48 83 c4 18          	add    $0x18,%rsp
  1e:	c3                   	ret
```



x86-64程序：更多的例子

swap in x86-64

- 总的来说，x86-64是更现代的体系结构，更精简的指令序列
  - `void swap(int *x, int *y);` 交换两个指针指向的数字

```
mov    0x8(%esp),%edx
mov    0xc(%esp),%eax
mov    (%edx),%ecx
mov    (%eax),%ebx
mov    %ebx,(%edx)
mov    %ecx,(%eax)
pop    %ebx
```

```
mov    (%rdi),%eax
mov    (%rsi),%edx
mov    %edx,(%rdi)
mov    %eax,(%rsi)
```

例子：循环

```c
int fact(int n) {
    int res = 1;
    do { res *= n; n--; } while (n > 0);
    return res;
}
```

```
        mov     $0x1,%eax
        nopl    (%rax)
.L1:    imul    %edi,%eax
        sub     $0x1,%edi
        test    %edi,%edi
        jg      .L1
        repz retq
```

- 两个诡异代码：
  - nopl (%rax) ：内存对齐（padding）
  - repz retq：防止连续分支指令

例子：递归

```
0000000000000704 <fib>:
 704: 55                push    %rbp
 705: 53                push    %rbx
 706: 89 fd             mov     %edi,%ebp
 708: 31 db             xor     %ebx,%ebx
 70a: 48 83 ec 08       sub     $0x8,%rsp
 70e: 83 fd 01          cmp     $0x1,%ebp
 711: 7e 0f             jle     722 <fib+0x1e>
 713: 8d 7d ff          lea     -0x1(%rbp),%edi
 716: 83 ed 02          sub     $0x2,%ebp
 719: e8 e6 ff ff ff    callq   704 <fib>
 71e: 01 c3             add     %eax,%ebx
 720: eb ec             jmp     70e <fib+0xa>
 722: 8d 43 01          lea     0x1(%rbx),%eax
 725: 5a                pop     %rdx
 726: 5b                pop     %rbx
 727: 5d                pop     %rbp
 728: c3                retq
```

Inline Assembly

在 C 代码中嵌入汇编

- 编译器：把 C 代码 “原样翻译” 成汇编代码
  - 那我们是不是可以在语句之间插入汇编呢？
    - 可以！编译器就做个 “复制粘贴”

```c
int foo(int x, int y) {
    x++; y++;
    asm ("endbr64;"
         ".byte 0xf3, 0x0f, 0x1e, 0xfa"
    );
    return x + y;
}
```

```
$ objdump -d foo.o

foo.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   89 7d fc                mov    %edi,-0x4(%rbp)
   b:   89 75 f8                mov    %esi,-0x8(%rbp)
   e:   83 45 fc 01             addl   $0x1,-0x4(%rbp)
  12:   83 45 f8 01             addl   $0x1,-0x8(%rbp)
  16:   f3 0f 1e fa             endbr64
  1a:   f3 0f 1e fa             endbr64
  1e:   8b 55 fc                mov    -0x4(%rbp),%edx
  21:   8b 45 f8                mov    -0x8(%rbp),%eax
  24:   01 d0                   add    %edx,%eax
  26:   5d                      pop    %rbp
  27:   c3                      ret
```

在汇编中访问 C 世界的表达式

```c
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
         "addl %1, %2; "
         "movl %2, %0; "
         : "=r"(z) // output
         : "r"(x), "r"(y) // input
    );
    return z;
}
```

gcc -O0

```
0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   55                      push   %rbp
   5:   48 89 e5                mov    %rsp,%rbp
   8:   89 7d ec                mov    %edi,-0x14(%rbp)
   b:   89 75 e8                mov    %esi,-0x18(%rbp)
   e:   83 45 ec 01             addl   $0x1,-0x14(%rbp)
  12:   83 45 e8 01             addl   $0x1,-0x18(%rbp)
  16:   8b 45 ec                mov    -0x14(%rbp),%eax
  19:   8b 55 e8                mov    -0x18(%rbp),%edx
  1c:   01 c2                   add    %eax,%edx
  1e:   89 d0                   mov    %edx,%eax
  20:   89 45 fc                mov    %eax,-0x4(%rbp)
  23:   8b 45 fc                mov    -0x4(%rbp),%eax
  26:   5d                      pop    %rbp
  27:   c3                      ret
```

在汇编中访问 C 世界的表达式

```c
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
         "addl %1, %2; "
         "movl %2, %0; "
         : "=r"(z) // output
         : "r"(x), "r"(y) // input
    );
    return z;
}
```

gcc –O2

```
0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   83 c7 01                add    $0x1,%edi
   7:   8d 46 01                lea    0x1(%rsi),%eax
   a:   01 f8                   add    %edi,%eax
   c:   89 c0                   mov    %eax,%eax
   e:   c3                      ret
```

在汇编中访问 C 世界的表达式

```c
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
         "addl %1, %2; "
         "movl %2, %0; "
         : "=&r"(z) // output
         : "r"(x), "r"(y) // input
    );
    return z;
}
```

gcc –O2

```
0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   83 c7 01                add    $0x1,%edi
   7:   83 c6 01                add    $0x1,%esi
   a:   01 fe                   add    %edi,%esi
   c:   89 f0                   mov    %esi,%eax
   e:   c3                      ret
```

与编译器交互

- 实际的编译器可能会
  - 将某个变量分配给某个寄存器
    - inline assembly 改写寄存器就会导致错误
    - → clobber list

```
asm asm-qualifiers ( AssemblerTemplate
                 : OutputOperands
                 [ : InputOperands
                 [ : Clobbers ] ])

asm asm-qualifiers ( AssemblerTemplate
                      : OutputOperands
                      : InputOperands
                      : Clobbers
                      : GotoLabels)
```

```c
#include <stdio.h>

int inc1(int src) {
  int dst;

  asm("mov %1, %0\n\t"
      "add $1, %0"
      : "=r"(dst)
      : "r"(src));

  return dst;
}

int inc2(int src) {
  int dst;

  asm("mov %1, %0\n\t"
      "mov $3, %%eax\n\t"
      "add $1, %0"
      : "=r"(dst)
      : "r"(src));

  return dst;
}
```

```c
int inc3(int src) {
  int dst;

  asm("mov %1, %0\n\t"
      "mov $3, %%eax\n\t"
      "add $1, %0"
      : "=r"(dst)
      : "r"(src)
      : "%eax");

  return dst;
}

int main(i
  printf("
  printf("
  printf("
  return 0
}
```

```
inc1: 2
inc2: 4
inc3: 2
```

```
endbr64
push    %rbp
mov     %rsp,%rbp
mov     %edi,-0x14(%rbp)
mov     -0x14(%rbp),%eax
mov     %eax,%eax
mov     $0x3,%eax
add     $0x1,%eax
mov     %eax,-0x4(%rbp)
mov     -0x4(%rbp),%eax
pop     %rbp
ret
```

与编译器交互

- 实际的编译器可能会
  - 代码优化
    - 例如 assembly 的返回值没有用到，就可以删除
      - → volatile

```c
int foo(int x, int y) {
    int z;
    x++; y++;
    asm (
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return 0;
}
```

gcc –O2

```
0000000000000000 <foo>:
   0:	f3 0f 1e fa          	endbr64
   4:	31 c0                	xor    %eax,%eax
   6:	c3                   	ret
```

在汇编中访问 C 世界的表达式

```c
int foo(int x, int y) {
    int z;
    x++; y++;
    asm volatile(
        "addl %1, %2; "
        "movl %2, %0; "
        : "=r"(z) // output
        : "r"(x), "r"(y) // input
    );
    return 0;
}
```

gcc –O2

```
0000000000000000 <foo>:
   0:   f3 0f 1e fa             endbr64
   4:   83 c7 01                add    $0x1,%edi
   7:   83 c6 01                add    $0x1,%esi
   a:   01 fe                   add    %edi,%esi
   c:   89 f7                   mov    %esi,%edi
   e:   31 c0                   xor    %eax,%eax
  10:   c3                      ret
```

总结

对汇编感到很痛苦？

- 两个建议
  - 原理：想一想 YEMU 和 NEMU
  - 实践：“模拟调试”
    - 指令不过是 CPU 执行的基本单元

- 同时，付出也是必要的
  - 但你一旦掌握了分析汇编代码的方法
    - x86-64 也不可怕
    - AArch64 也不可怕

  - 学会查手册

End. (你找对手册了吗？)

[x86-64 Machine-Level Programming]

[How to Use Inline Assembly Language in C Code]