

X86-64 结构指南

为了代码生成项目，我们需要向学生简要介绍 x86-64 平台。

示例

考虑如下 Decaf 程序：

```
class Program {  
  
    int foo(int x) {  
        return x + 3;  
    }  
  
    void main() {  
        int y;  
  
        y = foo(callout("get_int_035"));  
        if (y == 15) {  
            callout("printf_035", "Indeed! 'tis 15!\n");  
        } else {  
            callout("printf_035", "What! %d\n", y);  
        }  
    }  
}
```

编译后的程序版本可能会如下：

```
foo:  
    enter    $0, $0  
    mov     16(%rbp), %rax  
    add     $3, %rax  
    leave  
    ret  
  
    .globl main  
main:  
    enter   $(8 * 3), $0  
  
    call   get_int_035  
    push  %rax  
  
    call   foo
```

```

    add    $8, %rsp
    mov    %rax, -8(%rbp)

    mov    -8(%rbp), %r10
    mov    $15, %r11
    cmp    %r10, %r11
    mov    $0, %r11
    mov    $1, %r10
    cmove  %r10, %r11
    mov    %r11, -16(%rbp)

    mov    -16(%rbp), %r10
    mov    $1, %r11
    cmp    %r10, %r11
    je     .fifteen

    push  -8(%rbp)
    push  $.what
    call  printf_035
    add   $(2 * 8), %rsp
    jmp   .fifteen_done

.fifteen:
    push  $.indeed
    call  printf_035
    add   $(1 * 8), %rsp
.fifteen_done:

    mov    $0, %rax
    leave
    ret

.indeed:
    .string "Indeed, \'tis 15!\n"

.what:
    .string "What! %d\n"

```

我们可以仔细剖析这段汇编代码，并且和 Decaf 代码联系起来一起思考。注意这不是这段程序唯一可能的汇编代码；它仅仅是表达可在项目阶段使用方法的一个示例。

```

foo:
    enter  $(8 * 0), $0
    ...
leave
ret

```

- 这是一个函数定义的标准版式。第一行命名了一个函数接入口标记。接下来的enter

指令构建了**栈框架**。在函数处理完其真实工作后，`leave`指令回复调用者的栈框架，`ret`则将控制权返还给调用者。

- 注意 `enter` 的一个操作数是一个静态算法表达。这些表达可以通过编译器估算出来并且转换为最终输出的常数。

```
mov    16(%rbp), %rax
add    $3, %rax
```

- `foo` 的目的是对其实参加三，并返回值。函数的实参存储在调用者的框架中，在 `%rbp` 的正四字偏移量处。第 `k` 个实参存储在 `(8+8k)(%rbp)`，这样 `mov` 指令将第一个实参的值存储到 `%rax` 寄存器中。接下来的指令将 `%rax` 的字符值或立即数值增加 3。注意立即数总是以 '\$' 作为前缀。

- 通过**调用惯例**，一个函数必须将其返回值放在 `%rax` 寄存器中，所以 `foo` 成功的返回了 `x+3`。

```
.globl main
main:
    enter    $(8 * 3), $0
    ...
    mov     $0, %rax
    leave
    ret
```

- `.global main` 指令使 `main` 符号可在不止这一个模块可以访问。这一点很重要因为我们所反链接的 C 运行时间库，需要找到一个 `main` 函数来开始启动程序。

- `enter` 指令在栈中分配了三个四字空间：一个用以存储本地变量，两个用于存储传递给函数的实参。

- 在程序末端，我们将 `%rax` 设置为 0，表明程序成功结束。

```
call    get_int_035
push    %rax
```

- 我们调用 `get_int_035`，函数从标准输入中读取一个整数并返回这个整数。该函数没有实参。

- 返回的整数存储在 `%rax` 中，我们将其压入栈，用以 `foo` 的一个实参。注意我们已经在此处做了一些优化：另外一种可用的方法是将返回值存储在本地变量中，然后在载回并将其作为一个实参。

```
call    foo
add     $8, %rsp
mov     %rax, -8(%rbp)
```

- 对于这个实参，我们已经将其压入栈，我们调用 `foo`。

- 一旦 `foo` 返回了，我们就需要清除早些时候压入栈的实参，以清空栈。这里，我们增加 `%rsp`；我们页可以执行一次 `pop` 指令。

- 最后，我们将存储在 `%rax` 中的返回值存储在本地临时变量中。本地变量存储在 `%rbp` 的负偏移中。

```

mov     -8(%rbp), %r10
mov     $15, %r11
cmp     %r10, %r11
mov     $0, %r11
mov     $1, %r10
cmove   %r10, %r11
mov     %r11, -16(%rbp)

```

• 这一串代码序列说明了如何只用两个寄存器和临时存储器就实现比较操作的。我们以载入要比较的数在两个存储器开始，如 `foo` 的返回值和字符 15。这一步是很必要的，因为比较指令值只对寄存器操作数有效。

- 然后，我们用 `cmp` 指令执行实际比较操作。比较的结果是改变内部寄存器标记。
- 我们的目标是在本地变量中存储一个布尔值—1 或者 0—作为操作的结果值。为了实现这样的目标，我们在寄存器 `%r10` 和 `%r11` 中存储两个可能的值，1 和 0。
- 然后我们使用 `cmove` 指令（意为 `c-mov-e`，或者相等则有条件转存）要依靠之前比较所设置的标记，来确定输出值应该是 0 还是 1。指令将值存储在 `%r11` 中。
- 最后，我们将 `%r11` 中的布尔值存储到 -16 (`%rbp`) 的本地变量中。

```

mov     -16(%rbp), %r10
mov     $1, %r11
cmp     %r10, %r11
je      .fifteen
...
jmp     .fifteen_done
.fifteen:
...
.fifteen_done:

```

• 这是有条件语句的标准线性结构。我们将一个布尔变量和 1 进行比较，在执行 `je` 指令（相等则跳转），指令在成功比较后跳转到其目标块。如果比较失败，`je` 和空指令执行的一样。

• 我们在目标块末端做一个标记，并且在失败块末端跳转到标记处。按照惯例，这些标记并不定义函数，是为开始一个程序阶段而命名的。

```

.indeed:
    .string "Indeed, \'tis 15!\n"

.what:
    .string "What! %d\n"

```

- 这些对静态字符串定义在程序中。他们的用途是作为函数参数用。

参考资料

这个分发资料仅仅涉及了x86-64 指令集和结构中丰富可能性的一个子集。要更全面的介绍（仍然是易读的），请参考[AMD64 结构程序员手册，卷一：编程运用](#)。

寄存器

gcc 接受的汇编语法，寄存器名都以%作前缀。对于项目的第一部分，我们仅仅使用x86-64 的 16 个通用目的寄存器中的五个。所有的这些寄存器都为 64 比特。

寄存器	目的	跨越调用保存
%rax	返回值	不
%rsp	栈指针	是
%rbp	基址指针	是
%r10	暂存值	不
%r11		

指令集

这里表现的每个记忆操作代码都是一个指令族。每个指令族中，都有变量采用不同的参数类型（寄存器，立即值或者存储器地址）或者/并且有不同的参数大小（比特，字，双字，或四字）。前者可以通过参数的前缀来识别，后者通过记忆后缀的单字符可选项来识别。举例来说，一条将立即数 3 写入 64 比特%rax 寄存器的 mov 指令可以写为

```
movq    $3,%rax
```

立即操作数都以\$作为前缀。没有前缀的操作数视为存储器地址，并且应该避免出现这样的局面，因为他们的值是未知的。

对于修改操作数的指令，操作时在第二次使用时才修改值。这和 Microsoft 和 Borland 的编译器有所不同，这些编译器一般用于 DOS 和 Windows。

操作代码	描述
复制值	
mov src,dest	将寄存器，立即数或者存储器地址值拷入寄存器或者存储器地址
cmove %src,%dest	在最近比较操作有相应值的时候将寄存器%src 拷贝到寄存器%des（cmove：等于，cmovne：不等于，cmovg：大于，cmovl：小于，cmovge：大于等于，cmovle：小于等于）。
cmovne %src,%dest	
cmovg %src,%dest	
cmovl %src,%dest	
cmovge %src,%dest	
cmovle %src,%dest	
栈管理	
enter \$x,\$0	开始通过将%rbp 的值压入栈，建立程序栈框

	架, 将%esp 的当前值存入%ebp, 最后自减%esp 为 x 留出四字大小的本地变量空间。
leave	通过将%rsp 和%rbp 的旧值来移出栈框架的本地变量。
push src	自减%rsp, 将 src 写入%rsp 指向的存储器位置。这里, src 可以为寄存器, 立即数和存储器地址。
pop dest	将%rsp 指向位置的值存入 dest, 自减%rsp。这里, dest 可为寄存器或者存储器位置。
控制流	
jmp target	无条件跳转到 target, target 只能为存储器位置 (如, 卷标)
je target	如果最近的比较有相应的结果则跳转到 jump (je: 等于; jne: 不等于)。
jne target	
算术和逻辑	
add src, dest	将 src 加到 dest
sub src, dest	将 dest 减去 src
imul src, dest	将 dest 乘以 src
idiv src, dest	用 dest 除以 src。
shr src, dest	将 dest 左移或右移 src 位
shl src, dest	
ror src, dest	将 dest 循环左移或右移 src 位
cmp src, dest	对 dest 小于, 等于或者大于 src 置相应的标志

栈组织

局部和全局变量都存储在栈中, 栈一般为%rbp 和%rsp 寄存器偏移地址定义的存储区域。在创建栈框架过程中的程序调用结果可以存储这个符号的本地变量和临时中间值。栈的组织如下:

位置	内容	框架
8n+16(%rbp)	参数 n	前一个
.....	
16(%rbp)	参数 0	
8(%rbp)	返回地址	当前
0(%rbp)	前一个%rbp 值	
-8(%rbp)	本地和临时值	
0(%rbp)		

调用惯例

调用者将参数反序压入栈。最后，将返回地址和转换控制压入被调用者。被调用者将返回值存入%rax，并且要整理其本地变量和清除栈中的返回地址。但是不负责清除参数。指令 `call`, `enterl`, `leave` 和 `ret` 可以让遵守这些调用管理简单些。

在基于X86-64的Linux环境下，C语言使用的标准调用惯例有一点不同；具体见[系统V应用二进制接口-AMD64结构处理器补充](#)。特别是通过传递最初几个在寄存器而不是栈中的参数来优化调用。结果，程序还是不能直接被任意C程序调用。所以我们提供两个函数，`printf_035` 和 `get_int_035`，这些函数是特别用来适应这些简化惯例的。