微机原理与接口技术

姓名：陈致蓬

单位：中南大学自动化学院

电话： 15200328617

Email ： ZP.Chen@csu.edu.cn

Homepage:

https://www.scholarmate.com/psnweb/homepage

QQ ： 315566683

第8章 汇编

8.1 汇编梗概

8.2 汇编指令

8.3 汇编程序编写

8.1 汇编梗概

8.1.1 计算机执行的程序

计算机程序语言的分类：

- 高级语言
- 低级语言
- 机器语言

计算机本质上只能识别和执行二进制语言（即机器语言），因此任何程序的执行都不是直接执行，而是执行其编译后的二进制文件。

8.1.1 计算机执行的程序

计算机运行高级程序语言的过程：

- 将高级程序语言程序翻译成汇编程序
- 将汇编程序编译为二进制文件
- 将二进制文件链接为可执行文件
- 将可执行文件（指令）读入到内存按顺序执行

汇编语言是源代码和二进制代码之间的桥梁，它与二进制代码一一对应，同时又具备了可读性。可以说，它就是文本化的二进制代码。在发明高级语言之前，它一直是人类使用的程序语言。

8.1.1 计算机执行的程序

示例：

从左到右依次为 C 语言源程序、汇编程序和二进制语言（机器语言）

```
#include <
int main()
{
    printf
    return
}
```

```
0000000100003f60 <_main>:
100003f60: 55                          pushq   %rbp
100003f61: 48 89 e5                    movq    %rsp, %rbp
100003f64: 48 83 ec 10                 subq    $16, %rsp
100003f68: c7 45 fc 00 00 00 00        movl    $0, -4(%rbp)
100003f6f: 48 8d 3d 30 00 00 00        leaq    48(%rip), %rdi  # 100003fa6 <dyld_stub_binder+0x1000
03fa6>
100003f76: b0 00                       movb    $0, %al
100003f78: e8 09 00 00 00              callq   0x100003f86 <dyld_stub_binder+0x100003f86>
100003f7d: 31 c0                       xorl    %eax, %eax
100003f7f: 48 83 c4 10                 addq    $16, %rsp
100003f83: 5d                          popq    %rbp
100003f84: c3                          retq
```

```
00 00 00
00 00 00
00
```

8.1.1 计算机执行的程序

高级语言的编写规则与设备无关，而汇编语言的编写则依平台变化而不同。这是因为汇编代码其实就是一条一条的指令，然而不同的机器上的指令集体系结构是不同的。

例如：在 x86_64 平台上，你需要用到它提供的它自己的复杂指令集；在 arm 机器上，则要用到它自身的精简指令集。

8.1 汇编梗概
8.1.2 寄存器（以 x86_64 为例）

x86_64 平台具有 16 个寄存器，由于变量长度不一，并非所有变量都需要用到寄存器的所有位，因此一个 64 位寄存器又可以拆为 64 个寄存器，用于存放不同长度的变量。

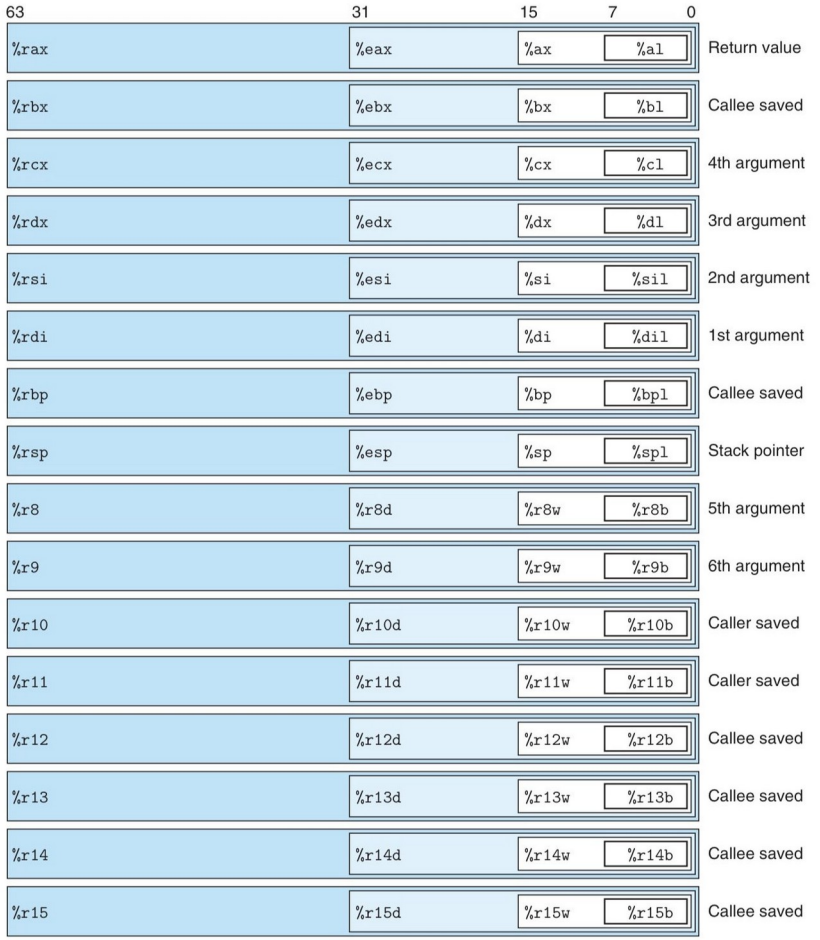
寄存器用途不一，但主要还是用于存储变量或地址的值。

8-byte register	Bytes 0-3	Bytes 0-1	Byte 0
%rax	%eax	%ax	%al
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rbx	%ebx	%bx	%bl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rsp	%esp	%sp	%spl
%rbp	%ebp	%bp	%bpl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

8.1.2 寄存器（以 x86_64 为例）

这 16 个寄存器中，有的要承担特殊的任务，如果 %rsp 用于指向栈顶，%rax(及 %eax 等) 要存储函数返回的值，有 6 个寄存器要用来保存函数的参数值，而有的则没有限制。

其它寄存器还有 caller-saved （调用者保存），callee-saved （被调用者保存）的特性



8.1 汇编梗概

8.1.3 特殊寄存器—— PC 寄存器

PC(Program counter) 寄存器，在汇编语言中以 %rip 表示，是非常关键的寄存器。指令序列都保存在内存中，每一条指令都有相应的（虚拟）地址。 PC 寄存器就存储着正在执行的指令的下一条要执行的指令。因此，每执行一条指令，下一条指令的地址就被放在 PC 寄存器中了。

```
100003f60: 55
100003f61: 48 89 e5
100003f64: 48 83 ec 10
100003f68: c7 45 fc 00 00 00 00
100003f6f: 48 8d 3d 30 00 00 00
03fa6>
100003f76: b0 00
100003f78: e8 09 00 00 00
100003f7d: 31 c0
100003f7f: 48 83 c4 10
100003f83: 5d
100003f84: c3
```

8.1.3 特殊寄存器—— PC 寄存器

PC 寄存器并不会显式地出现在汇编代码中，其值的变化都是暗地里进行的。

当然，也可以将想要执行的指令放到 PC 寄存器中改变程序的执行顺序（跳转指令和函数调用就是如此原理实现的）。

```
100003f60: 55
100003f61: 48 89 e5
100003f64: 48 83 ec 10
100003f68: c7 45 fc 00 00 00 00
100003f6f: 48 8d 3d 30 00 00 00
03fa6>
100003f76: b0 00
100003f78: e8 09 00 00 00
100003f7d: 31 c0
100003f7f: 48 83 c4 10
100003f83: 5d
100003f84: c3
```

8.2 汇编指令

8.2.1 指令

汇编代码是很多条指令的序列，一个指令可以完成一个 CPU 操作。

一条指令由操作码和 0 ～ 2 个操作数构成。操作码指定了当前指令要执行的操作，如将两数相加，操作数则是操作码的作用对象。

因此，指令的长度不固定，短则 1 个字节，长则 15 字节。

8.2 汇编指令
8.2.2 操作数

操作数可以是立即数、寄存器、内存地址。下面是表示这三种操作数的方法（来自 **x64 data sheet**）：

Type	From	Operand Value	Name
Immediate	*$Imm*	*Imm*	Immediate
Register	E_a	$R[E_a]$	Register
Memory	*Imm*	*M[Imm]*	Absolute
Memory	(E_a)	$M[R[E_b]]$	Absolute
Memory	$Imm(E_b, E_i, s)$	$M[Imm + R[E_b] + (R[E_i] \times s)]$	Scaled indexed

More information about operand specifiers can be found on pages 169-170 of the textbook.

8.2.2 操作数

例如：

- $5 是立即数，它的值 5 ；
- %rax 是寄存器，它的值是寄存器 %rax 中的值；
- 0xf7 是内存地址，它的值是内存中地址为 0x07 的某种类型的值；
- (%rax) 也是内存的地址，只不过，该地址保留在寄存器 %rax 中；
- 0xf7(%rax, %rbp, 4) 也是内存的地址，所有的内存寻址方式都可以写成这种类型。

8.2 汇编指令
8.2.3 操作码

操作码主要分为：

- 算术逻辑类，对操作数进行算数运算和逻辑运算。
- 数据传输类，在存贮器和寄存器、寄存器和输入输出端口之间传送数据。
- 控制类，控制程序跳转、终端等。

8.2 汇编指令
8.2.4 汇编示例

示例 1 ：一个什么也没有做的函数，没有参数也没有返回值。

```
void foo(){
    return;
}
```

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 5d                            popq    %rbp
       5: c3                            retq
```

8.2.4 汇编示例

示例 1 ：一个什么也没有做的函数，没有参数也没有返回值。

红框中 0000000000000000 （16 进制）代表函数的地址（虚拟地址），第二部分 <_foo> 中的 _foo 为函数名，为原函数名加上一个’ _ ’。

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 5d                            popq    %rbp
       5: c3                            retq
```

8.2.4 汇编示例

示例 1 ：一个什么也没有做的函数，没有参数也没有返回值。

红框中的数字 0 、 1 、 4 、 5 代表各条指令的地址（ 16 进制）。由于各指令的长度不同，因此个指令间的地址间隔亦不相同，如第二条指令长度为三字节，因此第二和第三条指令之间的地址间隔为 3 。

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 5d                            popq    %rbp
       5: c3                            retq
```

8.2.4 汇编示例

示例 1 ：一个什么也没有做的函数，没有参数也没有返回值。

红框中的 16 进制数代表指令的机器码。

蓝框中的为汇编指令。

```
0000000000000000 <_foo>:
       0:  55                    pushq   %rbp
       1:  48 89 e5              movq    %rsp, %rbp
       4:  5d                    popq    %rbp
       5:  c3                    retq
```

8.2.4 汇编示例

示例 2 ：带返回值的函数。

```
int foo(){
    return 0;
}
```

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 31 c0                         xorl    %eax, %eax
       6: 5d                            popq    %rbp
       7: c3                            retq
```

8.2.4 汇编示例

示例 2 ：带返回值的函数。

相较于示例 1 中，示例 2 中多了红框中的指令。这是将 %eax 与自身作按位异或，其结果是 %eax 的值变为 0(而且是 int 类型) 。而 %eax 恰好是保存返回值的。

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 31 c0                         xorl    %eax, %eax
       6: 5d                            popq    %rbp
       7: c3                            retq
```

8.2.4 汇编示例

示例 3 ：返回两个参数和的函数。

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

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 8d 04 37                      leal    (%rdi,%rsi), %eax
       7: 5d                            popq    %rbp
       8: c3                            retq
```

8.2.4 汇编示例

示例 3 ：返回两个参数和的函数。

相较于示例 1 ，示例 3 中的 leal 指令将 %rdi 和 %rdi 的值相加，赋给 %eax 。 %rdi 和 %rsi 分别储存第一个和第二个参数， %eax 储存返回变量。

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 8d 04 37                      leal    (%rdi,%rsi), %eax
       7: 5d                            popq    %rbp
       8: c3                            retq
```

8.2.4 汇编示例

示例 4 ：返回八个参数和的函数。

```
int foo(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8){
    return a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8;
}
```

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 8d 04 37                      leal    (%rdi,%rsi), %eax
       7: 01 d0                         addl    %edx, %eax
       9: 01 c8                         addl    %ecx, %eax
       b: 44 01 c0                      addl    %r8d, %eax
       e: 44 01 c8                      addl    %r9d, %eax
      11: 03 45 10                      addl    16(%rbp), %eax
      14: 03 45 18                      addl    24(%rbp), %eax
      17: 5d                            popq    %rbp
      18: c3                            retq
```

8.2.4 汇编示例

示例 4 ：返回 8 个参数和的函数。

```
0000000000000000 <_foo>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 8d 04 37                      leal    (%rdi,%rsi), %eax
       7: 01 d0                         addl    %edx, %eax
       9: 01 c8                         addl    %ecx, %eax
       b: 44 01 c0                      addl    %r8d, %eax
       e: 44 01 c8                      addl    %r9d, %eax
      11: 03 45 10                      addl    16(%rbp), %eax
      14: 03 45 18                      addl    24(%rbp), %eax
      17: 5d                            popq    %rbp
      18: c3                            retq
```

8.2.4 汇编示例

示例 4 ：返回 8 个参数和的函数。

6 个参数传递的寄存器 %rdi, %rsi, %edx, %ecx, %r8d, %r9d 依次出现，验证了前 6 个参数保存在这些寄存器内。而红框内最后两行则是将 %rbp 上面第 16 字节和 24 字节的 doubleword 型数据加到 %eax 上，说明第 7 、 8 个参数存储在这两个地方。因此若多余 6 个参数的话，多余的参数会保存在栈中来进行传递。

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数。

```
int callee(int a1, int a2, int a3, int a4, int a5, int a6, int a7, int a8){
    return a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8;
}

int caller(){
    return callee(1, 2, 3, 4, 5, 6, 7, 8) + 9;
}
```

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数。

```
0000000000000000 <_callee>:
       0: 55                            pushq   %rbp
       1: 48 89 e5                      movq    %rsp, %rbp
       4: 8d 04 37                      leal    (%rdi,%rsi), %eax
       7: 01 d0                         addl    %edx, %eax
       9: 01 c8                         addl    %ecx, %eax
       b: 44 01 c0                      addl    %r8d, %eax
       e: 44 01 c8                      addl    %r9d, %eax
      11: 03 45 10                      addl    16(%rbp), %eax
      14: 03 45 18                      addl    24(%rbp), %eax
      17: 5d                            popq    %rbp
      18: c3                            retq
      19: 0f 1f 80 00 00 00 00          nopl    (%rax)
```

```
0000000000000020 <_caller>:
      20: 55                            pushq   %rbp
      21: 48 89 e5                      movq    %rsp, %rbp
      24: bf 01 00 00 00                movl    $1, %edi
      29: be 02 00 00 00                movl    $2, %esi
      2e: ba 03 00 00 00                movl    $3, %edx
      33: b9 04 00 00 00                movl    $4, %ecx
      38: 41 b8 05 00 00 00             movl    $5, %r8d
      3e: 41 b9 06 00 00 00             movl    $6, %r9d
      44: 6a 08                         pushq   $8
      46: 6a 07                         pushq   $7
      48: e8 00 00 00 00                callq   0x4d <_caller+0x2d>
      4d: 48 83 c4 10                   addq    $16, %rsp
      51: 83 c0 09                      addl    $9, %eax
      54: 5d                            popq    %rbp
      55: c3                            retq
```

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数（链接后）。

```
0000000100003f60 <_callee>:
100003f60: 55                          pushq   %rbp
100003f61: 48 89 e5                    movq    %rsp, %rbp
100003f64: 8d 04 37                    leal    (%rdi,%rsi), %eax
100003f67: 01 d0                       addl    %edx, %eax
100003f69: 01 c8                       addl    %ecx, %eax
100003f6b: 44 01 c0                    addl    %r8d, %eax
100003f6e: 44 01 c8                    addl    %r9d, %eax
100003f71: 03 45 10                    addl    16(%rbp), %eax
100003f74: 03 45 18                    addl    24(%rbp), %eax
100003f77: 5d                          popq    %rbp
100003f78: c3                          retq
100003f79: 0f 1f 80 00 00 00 00        nopl    (%rax)
```

```
0000000100003f80 <_main>:
100003f80: 55                          pushq   %rbp
100003f81: 48 89 e5                    movq    %rsp, %rbp
100003f84: bf 01 00 00 00              movl    $1, %edi
100003f89: be 02 00 00 00              movl    $2, %esi
100003f8e: ba 03 00 00 00              movl    $3, %edx
100003f93: b9 04 00 00 00              movl    $4, %ecx
100003f98: 41 b8 05 00 00 00           movl    $5, %r8d
100003f9e: 41 b9 06 00 00 00           movl    $6, %r9d
100003fa4: 6a 08                       pushq   $8
100003fa6: 6a 07                       pushq   $7
100003fa8: e8 b3 ff ff ff              callq   0x100003f60 <_callee>
100003fad: 48 83 c4 10                 addq    $16, %rsp
100003fb1: 83 c0 09                    addl    $9, %eax
100003fb4: 5d                          popq    %rbp
100003fb5: c3                          retq
```

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数（链接后）。

立即数 $1 ~ 6 分别按顺序装入对应寄存器，之后，先后将立即数 $8 和 $7 压栈， %rsp 在这个过程中隐式地减少了 8+8=16 个字节（因为装下一个 quadword 需要 8 个字节）。从这可以看到多出的参数是从后往前依次压栈的。所以示例 4 中 16(%rbp) 中保存着 $7 ， 24(%rbp) 保存着 $8 。

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数（链接后）。

callq 的操作数 0x100003f60 <_callee> 正是 callee 的地址。

这个过程隐式地发生了将“返回地址”压栈。所谓返回地址就是调用指令的下一条指令的地址，如这里 (链接后的程序) 的返回地址是 100003fad 。将返回地址压栈的目的是使子函数执行完毕后能继续执行下一条执行。压栈意味着 %rsp 将减少 8 个字节。将 PC 寄存器设置为被调用函数的地址，这里是 0x100003f60 。

8.2.4 汇编示例

示例 5 ：调用返回八个参数和的函数（链接后）。

参数 7 的地址与 %rbp 隔着“返回地址”和“ %rbp 的旧值”两个值，共 16 个字节，而参数 8 除了隔着这 16 字节，还隔着参数所占的 8 字节。因此参数 7 与 %rbp 的地址相差 16 个字节？参数 8 与 %rbp 的地址相差 24 字节。

谢谢大家！