计算机系统讲座(1)

计算机系统漫游

信息的表示和处理

程序的机器级表示

优化程序性能

第1章 计算机系统漫游

- 编译过程：gcc -o hello hello.c

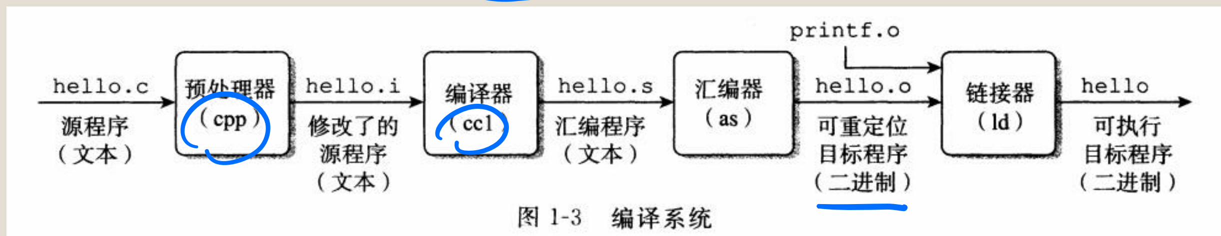


图 1-3 编译系统

gcc -E hello.c -o hello.i
gcc -S hello.i
gcc -c hello.s
gcc hello.o printf.o -o hello

- 运行：./hello
- 超标量：处理器可以达到比一个周期一条指令更快的执行效率
- 超线程：允许一个CPU执行多个控制流的技术
- 抽象：

图 1-18 计算机系统提供的一些抽象。计算机系统中的一个重大主题就是提供不同层次的抽象表示，来隐藏实际实现的复杂性

2.1 信息存储

23. C 程序中定义 int x=-3，则&x 处依次存放（小端模式）的十六进制数据为_ FD FF FF FF _。

- 区分好十六进制和十进制
- 数据类型大小区分
- 小端法(Intel)：最低有效字节在最前面的方式
- 大端法：最高有效字节在最前面的方式
- 字符串与大小端无关
- printf函数第一个参数：格式串；其余参数：要打印的值
- &、|、~、^
- 掩码：一个字中选出的位的集合
- ||、&&、!
- 短路求值：a&&5/a；p&&*p++
- 移位运算

C声明		字节数	
有符号	无符号	32位	64位
[signed] char	unsigned char	1	1
short	unsigned short	2	2
int	unsigned	4	4
long	unsigned long	4	8
int32_t	uint32_t	4	4
int64_t	uint64_t	8	8
char *		4	8
float		4	4
double		8	8

图 2-3 基本 C 数据类型的典型大小(以字节为单位)。分配的字节数受程序是如何编译的影响而变化。本图给出的是 32 位和 64 位程序的典型值

21. 判断整型变量 n 的位 7 为 1 的 C 语言表达式是__________。

2.2 整数表示

- 无符号数的编码
- 补码编码：最高有效位负权
- 无符号数与有符号数的转换结果保持位值不变，只是改变了解释这些位的方式（CPU无法区分有符号数和无符号数）
- 一个运算中，一个运算数是有符号的、一个无符号的，那么C语言会隐式将有符号参数强制类型转换为无符号数
- 先改变大小，再有符号和无符号转换
- 隐式强制类型转换和无符号数据类型造成的错误

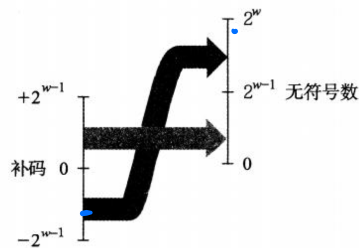


图 2-17　从补码到无符号数的转换。函数 $T2U$ 将负数转换为大的正数

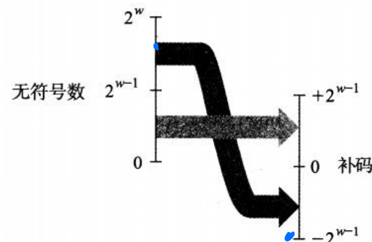
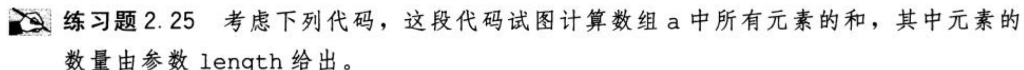


图 2-18　从无符号数到补码的转换。函数 $U2T$ 把大于 $2^{w-1}-1$ 的数字转换为负值

1. C 语言中整数-1 与无符号 0 比较，其结果是（ A ）
   A. 大于　B. 小于　C. 可能大于可能小于　D. 无法比较

32. （ X ）C 语言中，关系表达式：127 > (unsigned char)128U 是成立的。

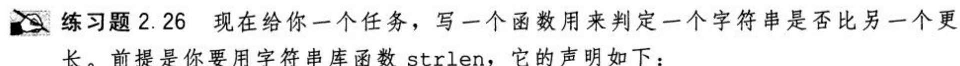
2.2 整数表示



练习题 2.25 考虑下列代码，这段代码试图计算数组 a 中所有元素的和，其中元素的数量由参数 length 给出。

```
1   /* WARNING: This is buggy code */
2   float sum_elements(float a[], unsigned length) {
3       int i;
4       float result = 0;
5
6       for (i = 0; i <= length-1; i++)
7           result += a[i];
8       return result;
9   }
```

当参数 length 等于 0 时，运行这段代码应该返回 0.0。但实际上，运行时会遇到一个内存错误。请解释为什么会发生这样的情况，并且说明如何修改代码。



练习题 2.26 现在给你一个任务，写一个函数用来判定一个字符串是否比另一个更长。前提是你要用字符串库函数 strlen，它的声明如下：

```
/* Prototype for library function strlen */
size_t strlen(const char *s);
```

最开始你写的函数是这样的：

```
/* Determine whether string s is longer than string t */
/* WARNING: This function is buggy */
int strlonger(char *s, char *t) {
    return strlen(s) - strlen(t) > 0;
}
```

当你在一些示例数据上测试这个函数时，一切似乎都是正确的。进一步研究发现在头文件 stdio.h 中数据类型 size_t 是定义成 unsigned int 的。

A. 在什么情况下，这个函数会产生不正确的结果？

B. 解释为什么会出现这样不正确的结果。

C. 说明如何修改这段代码好让它能可靠地工作。

2.25 设计这个问题是要说明从有符号数到无符号数的隐式强制类型转换很容易引起错误。将参数 length 作为一个无符号数来传递看上去是件相当自然的事情，因为没有人会想到使用一个长度为负数的值。停止条件 i<=length-1 看上去也很自然。但是把这两点组合到一起，将产生意想不到的结果！

因为参数 length 是无符号的，计算 0−1 将使用无符号运算，这等价于模数加法。结果得到 *UMax*。≤比较同样使用无符号数比较，而因为任何数都是小于或者等于 *UMax* 的，所以这个比较总是为真！因此，代码将试图访问数组 a 的非法元素。

有两种方法可以改正这段代码，其一是将 length 声明为 int 类型，其二是将 for 循环的测试条件改为 i<length。

2.26 这个例子说明了无符号运算的一个细微的特性，同时也是我们执行无符号运算时不会意识到的属性。这会导致一些非常棘手的错误。

A. 在什么情况下，这个函数会产生不正确的结果？当 s 比 t 短的时候，该函数会不正确地返回 1。

B. 解释为什么会出现这样不正确的结果。由于 strlen 被定义为产生一个无符号的结果，差和比较都采用无符号运算来计算。当 s 比 t 短的时候，strlen(s)-strlen(t) 的差会为负，但是变成了一个很大的无符号数，且大于 0。

C. 说明如何修改这段代码好让它能可靠地工作。将测试语句改成：

```
return strlen(s) > strlen(t);
```

2.3 整数运算

- 无符号数加法溢出：当且仅当$x+y<x$（或者等价地$x+y<y$）
- 补码加法溢出：正溢出：$x>0,y>0,x+y<=0$

  负溢出：$x<0,y<0,x+y>=0$
- CPU可以判断加法是否溢出

2.4 浮点数

- IEEE754：$V=(-1)^s \times M \times 2^E$　　s:符号 M:尾数 E:阶码
- float:1,8,23； double:1,11,52
- 规格化： exp位模式不全为0也不全为1

  $E=e-Bias$ (e:无符号数， $Bias=2^{k-1}-1$ （127， 1023））

  $M=1+f$ (隐含以1开头) [1,2)
- 非规格化的值： exp位模式全0

  $E=1-Bias$

  $M=f$
- 特殊值： exp位模式全1

  小数域全为0：无穷

  小数域非零： NaN

2.4 浮点数

- 例：15213_{10}
- 舍入：向偶数舍入：倾向于最低有效位为0

方 式	1.40	1.60	1.50	2.50	−1.50
向偶数舍入	1	2	2	2	−2
向零舍入	1	1	1	2	−1
向下舍入	1	1	1	2	−2
向上舍入	2	2	2	3	−1

图 2-37 以美元舍入为例说明舍入方式(第一种方法是舍入到一个最接近的值，而其他三种方法向上或向下限定结果，单位为美元)

22. 按照“向偶数舍入”的规则，二进制小数 101.110_2 舍入到最接近的 1/2（小数点右边 1 位）后的二进制为 110.0_2 。

2.4 浮点数

- int、float、double转换：
- （1）int->float：数字不会溢出，但可能被舍入
- （2）int或float->double：能够保留精确的数值
- （3）double->float：可能溢出，可能舍入
- （4）float或double->int：向0舍入，可能溢出

3.2 程序编码

- C语言不可见，但机器代码可见：

  （1）程序计数器(PC)：用%rip表示（%rip不是通用寄存器）

  （2）通用寄存器

  （3）条件码寄存器

  （4）向量寄存器

- 查看机器代码文件内容：用反汇编器 objdump -d mstore.o

3.3-3.4 数据格式、访问信息

- b,w,l,q
- 操作数三种类型：
  （1）立即数：$
  （2）寄存器： $R[r_a]$
  （3）内存引用： $Imm(r_b,r_i,s)$：
  $Imm+R[r_b]+R[r_i]*s$

63	31	15	7	0	
%rax	%eax	%ax	%al		返回值
%rbx	%ebx	%bx	%bl		被调用者保存
%rcx	%ecx	%cx	%cl		第4个参数
%rdx	%edx	%dx	%dl		第3个参数
%rsi	%esi	%si	%sil		第2个参数
%rdi	%edi	%di	%dil		第1个参数
%rbp	%ebp	%bp	%bpl		被调用者保存
%rsp	%esp	%sp	%spl		栈指针
%r8	%r8d	%r8w	%r8b		第5个参数
%r9	%r9d	%r9w	%r9b		第6个参数
%r10	%r10d	%r10w	%r10b		调用者保存
%r11	%r11d	%r11w	%r11b		调用者保存
%r12	%r12d	%r12w	%r12b		被调用者保存
%r13	%r13d	%r13w	%r13b		被调用者保存
%r14	%r14d	%r14w	%r14b		被调用者保存
%r15	%r15d	%r15w	%r15b		被调用者保存

图 3-2 整数寄存器。所有 16 个寄存器的低位部分都可以作为字节、字(16 位)、双字(32 位)和四字(64 位)数字来访问

3.4.2 数据传送指令

2. 在 x86-64 中，有初始值%rax =0x1122334455667788，执行下述指令后 rax 寄存器的值是(A)

movl $0xaa11, %rax

A. 0xaa11 B. 0x112233445566aa11

C. 0x112233440000aa11 D. 0x11223344ffffaa11

- MOV指令只会更新目的操作数指定的那些寄存器字节或内存位置，唯一例外的是movl指令以寄存器为目的时，会把该寄存器高位4字节置为0

movzlq ×

2. 关于汇编指令 movl $0x1234, 8(%rax,%rbx,8)，错误的说法是(D)

A. 目的操作数是内存操作数

B. 内存操作数的地址是 8+(%rax+%rbx×8)

C. 操作数宽度是 4 字节

D. 可以改写成 mov $0x1234, 8(%rax, %rbx,8)，效果一样

指令		效果	描述
MOV	S, D	D←S	传送
movb			传送字节
movw			传送字
movl			传送双字
movq			传送四字
movabsq	I, R	R←I	传送绝对的四字

图 3-4 简单的数据传送指令

指令		效果	描述
MOVZ	S, R	R←零扩展(S)	以零扩展进行传送
movzbw			将做了零扩展的字节传送到字
movzbl			将做了零扩展的字节传送到双字
movzwl			将做了零扩展的字传送到双字
movzbq			将做了零扩展的字节传送到四字
movzwq			将做了零扩展的字传送到四字

图 3-5 零扩展数据传送指令。这些指令以寄存器或内存地址作为源，以寄存器作为目的

指令		效果	描述
MOVS	S, R	R←符号扩展(S)	传送符号扩展的字节
movsbw			将做了符号扩展的字节传送到字
movsbl			将做了符号扩展的字节传送到双字
movswl			将做了符号扩展的字传送到双字
movsbq			将做了符号扩展的字节传送到四字
movswq			将做了符号扩展的字传送到四字
movslq			将做了符号扩展的双字传送到四字
cltq		%rax ←符号扩展(%eax)	把%eax 符号扩展到%rax

图 3-6 符号扩展数据传送指令。MOVS 指令以寄存器或内存地址作为源，以寄存器作为目的。cltq 指令只作用于寄存器%eax 和%rax

3.4.4 压入和弹出数据

- 栈向低地址方向生长，栈顶元素的地址是最低的，栈指针%rsp保存着栈顶元素的地址
- 程序可以用标准的内存寻址方法访问栈内的任意位置
- push A 先将栈指针减8，再写入
- pop A 先读出，再将栈指针加8

3.5 算术和逻辑操作

- 加载有效地址lea：将第一个操作数的有效地址写入到目的寄存器
  算术运算、产生指针
- 一元操作
- 二元操作
- 移位
- 特殊的算术操作

指令	效果	描述
leaq S, D	D ← &S	加载有效地址
INC D	D ← D + 1	加1
DEC D	D ← D - 1	减1
NEG D	D ← -D	取负
NOT D	D ← ~D	取补
ADD S, D	D ← D + S	加
SUB S, D	D ← D - S	减
IMUL S, D	D ← D * S	乘
XOR S, D	D ← D ^ S	异或
OR S, D	D ← D | S	或
AND S, D	D ← D & S	与
SAL k, D	D ← D << k	左移
SHL k, D	D ← D << k	左移（等同于SAL）
SAR k, D	$D \leftarrow D >>_A k$	算术右移
SHR k, D	$D \leftarrow D >>_L k$	逻辑右移

指令	效果	描述
imulq S	R[%rdx]: R[%rax]←S×R[%rax]	有符号全乘法
mulq S	R[%rdx]: R[%rax]←S×R[%rax]	无符号全乘法
clto	R[%rdx]: R[%rax]←符号扩展(R[%rax])	转换为八字
idivq S	R[%rdx]←R[%rdx]: R[%rax] mod S R[%rdx]←R[%rdx]: R[%rax]÷S	有符号除法
divq S	R[%rdx]←R[%rdx]: R[%rax] mod S R[%rdx]←R[%rdx]: R[%rax]÷S	无符号除法

图 3-12 特殊的算术操作。这些操作提供了有符号和无符号数的全 128 位乘法和除法。一对寄存器%rdx 和%rax 组成一个 128 位的八字

3.6 控制

- CF：进位标志，检查无符号操作的溢出
- ZF：零标志，最近的操作得出的结果为0
- SF：符号标志，最近的操作得到的结果为负数
- OF：溢出标志，补码溢出——正溢出或负溢出
- 算术逻辑运算
- CMP：S2-S1
- TEST：S1&S2
- SET、JMP、CMOV
- PC相对

指令		同义名	效果	设置条件
sete	*D*	setz	*D* ← ZF	相等/零
setne	*D*	setnz	*D* ← ~ZF	不等/非零
sets	*D*		*D* ← SF	负数
setns	*D*		*D* ← ~SF	非负数
setg	*D*	setnle	*D* ← ~(SF ^ OF) & ~ZF	大于（有符号>）
setge	*D*	setnl	*D* ← ~(SF ^ OF)	大于等于（有符号>=）
setl	*D*	setnge	*D* ← SF ^ OF	小于（有符号<）
setle	*D*	setng	*D* ← (SF ^ OF) | ZF	小于等于（有符号<=）
seta	*D*	setnbe	*D* ← ~CF & ~ZF	超过（无符号>）
setae	*D*	setnb	*D* ← ~CF	超过或相等（无符号>=）
setb	*D*	setnae	*D* ← CF	低于（无符号<）
setbe	*D*	setna	*D* ← CF | ZF	低于或相等（无符号<=）

图 3-14　SET 指令。每条指令根据条件码的某种组合，将一个字节设置为 0 或者 1。有些指令有“同义名”，也就是同一条机器指令有别的名字

指令		同义名	跳转条件	描述
jmp	*Label*		1	直接跳转
jmp	**Operand*		1	间接跳转
je	*Label*	jz	ZF	相等/零
jne	*Label*	jnz	~ZF	不相等/非零
js	*Label*		SF	负数
jns	*Label*		~SF	非负数
jg	*Label*	jnle	~(SF ^ OF) & ~ZF	大于（有符号>）
jge	*Label*	jnl	~(SF ^ OF)	大于或等于（有符号>=）
jl	*Label*	jnge	SF ^ OF	小于（有符号<）
jle	*Label*	jng	(SF ^ OF) | ZF	小于或等于（有符号<=）
ja	*Label*	jnbe	~CF & ~ZF	超过（无符号>）
jae	*Label*	jnb	~CF	超过或相等（无符号>=）
jb	*Label*	jnae	CF	低于（无符号<）
jbe	*Label*	jna	CF | ZF	低于或相等（无符号<=）

图 3-15　jump 指令。当跳转条件满足时，这些指令会跳转到一条带标号的目的地。有些指令有“同义名”，也就是同一条机器指令的别名

指令		同义名	传送条件	描述
cmove	*S*, *R*	cmovz	ZF	相等/零
cmovne	*S*, *R*	cmovnz	~ZF	不相等/非零
cmovs	*S*, *R*		SF	负数
cmovns	*S*, *R*		~SF	非负数
cmovg	*S*, *R*	cmovnle	~(SF ^ OF) & ~ZF	大于（有符号>）
cmovge	*S*, *R*	cmovnl	~(SF ^ OF)	大于或等于（有符号>=）
cmovl	*S*, *R*	cmovnge	SF ^ OF	小于（有符号<）
cmovle	*S*, *R*	cmovng	(SF ^ OF) | ZF	小于或等于（有符号<=）
cmova	*S*, *R*	cmovnbe	~CF & ~ZF	超过（无符号>）
cmovae	*S*, *R*	cmovnb	~CF	超过或相等（无符号>=）
cmovb	*S*, *R*	cmovnae	CF	低于（无符号<）
cmovbe	*S*, *R*	cmovna	CF | ZF	低于或相等（无符号<=）

图 3-18　条件传送指令。当传送条件满足时，指令把源值 *S* 复制到目的 *R*。有些指令是“同义名”，即同一条机器指令的不同名字

3.6 控制

- if
- do while
- while
- for
- 跳转表：.rodata节

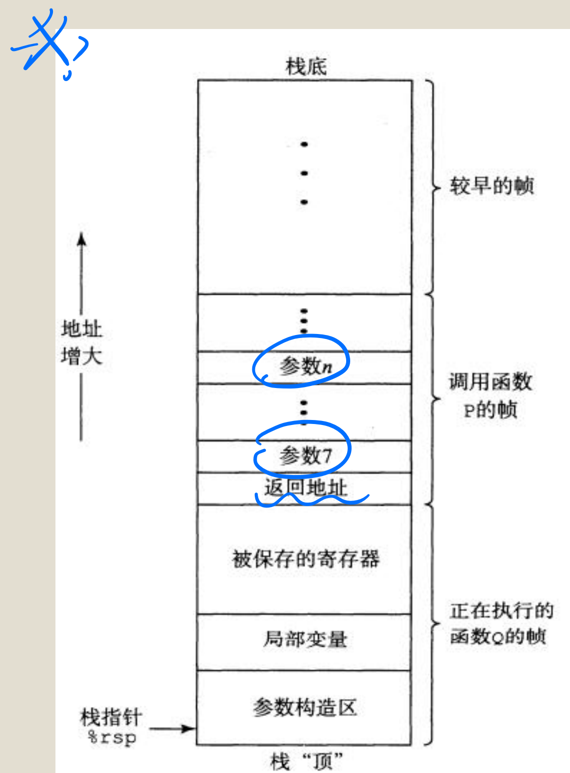
```
1        .section        .rodata
2        .align 8          Align address to multiple of 8
3     .L4:
4        .quad    .L3      Case 100: loc_A
5        .quad    .L8      Case 101: loc_def
6        .quad    .L5      Case 102: loc_B
7        .quad    .L6      Case 103: loc_C
8        .quad    .L7      Case 104: loc_D
9        .quad    .L8      Case 105: loc_def
10       .quad    .L7      Case 106: loc_D
```

```
   void switch_eg(long x, long n, long *dest)
   x in %rdi, n in %rsi, dest in %rdx
1  switch_eg:
2    subq    $100, %rsi              Compute index = n-100
3    cmpq    $6, %rsi                Compare index:6
4    ja      .L8                     If >, goto loc_def
5    jmp     *.L4(,%rsi,8)           Goto *jt[index]
6  .L3:                            loc_A:
7    leaq    (%rdi,%rdi,2), %rax       3*x
8    leaq    (%rdi,%rax,4), %rdi       val = 13*x
9    jmp     .L2                       Goto done
10 .L5:                            loc_B:
11   addq    $10, %rdi                 x = x + 10
12 .L6:                            loc_C:
13   addq    $11, %rdi                 val = x + 11
14   jmp     .L2                       Goto done
15 .L7:                            loc_D:
16   imulq   %rdi, %rdi                val = x * x
17   jmp     .L2                       Goto done
18 .L8:                            loc_def:
19   movl    $0, %edi                  val = 0
20 .L2:                            done:
21   movq    %rdi, (%rdx)              *dest = val
22   ret                               Return
```

图 3-23　图 3-22 中 switch 语句示例的汇编代码

3.7 过程

- 寄存器：%rdi %rsi %rdx %rcx %r8 %r9
- 参数7的位置
- call、ret
- 局部数据必须放在内存中的情况：
  （1）寄存器不足够放所有的本地数据
  （2）对一个局部变量使用地址运算符’&’
  因此必须能够为它产生一个地址
  （3）某些局部变量是数组或结构，必须能够通过数组或结构引用被访问到
- 管理变长栈帧，可以使用%rbp作为帧指针



3.7 过程

- 被调用者保存寄存器：%rbx、%rbp、%r12-%r15
- 调用者保存寄存器：其他除了%rsp

```
long P(long x, long y)
{
    long u = Q(y);
    long v = Q(x);
    return u + v;
}
```

a）调用函数

```
long P(long x, long y)
x in %rdi, y in %rsi
1   P:
2     pushq   %rbp          Save %rbp
3     pushq   %rbx          Save %rbx
4     subq    $8, %rsp      Align stack frame
5     movq    %rdi, %rbp    Save x
6     movq    %rsi, %rdi    Move y to first argument
7     call    Q             Call Q(y)
8     movq    %rax, %rbx    Save result
9     movq    %rbp, %rdi    Move x to first argument
10    call    Q             Call Q(x)
11    addq    %rbx, %rax    Add saved Q(y) to Q(x)
12    addq    $8, %rsp      Deallocate last part of stack
13    popq    %rbx          Restore %rbx
14    popq    %rbp          Restore %rbp
15    ret
```

b）调用函数生成的汇编代码

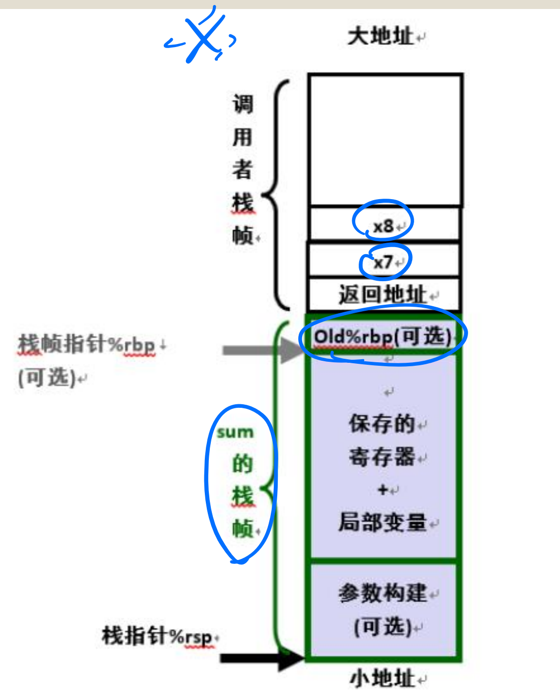
图 3-34　展示被调用者保存寄存器使用的代码。在第一次调用中，必须保存 x 的值，第二次调用中，必须保存 Q(y)的值

3.7 过程

41. 从汇编的角度阐述：函数 **int sum(int x1,int x2,int x3,int x4,int x5,int x6,int x7,int x8)**，调用和返回的过程中，参数、返回值、控制是如何传递的？并画出 **sum** 函数的栈帧（**X86-64** 形式）。

41 题（每点 1 分，图 2 分，满分 5 分）

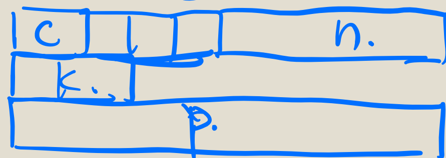
- 整型参数 x1~x6 分别用%rdi, %rsi, %rdx, %rcx, %r8, %r9 传递
  或：整型参数 x1~x6 分别用%edi, , %esi, %edx, %ecx, %r8d, %r9d 传递
- 参数 x7 x8 用栈传递；
- 返回值用%rax（%eax）传递
- call 指令将返回地址入栈、并将控制转移到被调用函数
- ret 指令将返回地址出栈、修改 RIP 的数值，将控制转移到调用者程序。



3.8-3.9 数组、结构体、联合

- 对齐：长度为K的基本类型成员的偏移必须是K的倍数
  结构体的整体对齐要求值为所有元素的最大对齐要求值

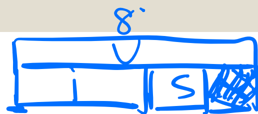
22. C语言程序定义了结构体 **struct noname{char c; int n; short k; char *p;};**若该程序编译成 **64** 位可执行程序，则 **sizeof(noname)**的值是__24.________。



20. x86-64 中，某 C 程序定义了结构体

```
struct SS {
        double v;
        int i;
        short s;
    } aa[10];
```

16.



则执行 sizeof(aa)的值是（ D ）

A. 14　　B. 80　　C. 140　　D. 160

3.10 缓冲区溢出攻击

- 攻击原理：向程序输入缓冲区写入特定的数据，例如在gets读入字符串时，使位于栈中的缓冲区数据溢出，用特定的内容覆盖栈中的内容，例如函数返回地址等，使得程序在读入字符串，结束函数gets从栈中读取返回地址时，错误地返回到特定的位置，执行特定的代码，达到攻击的目的。
- 防范方法:

(1)代码中避免溢出漏洞：例如使用限制字符串长度的库函数。

(2)随机栈偏移：程序启动后，在栈中分配随机数量的空间，将移动整个程序使用的栈空间地址。

(3)限制可执行代码的区域

(4)进行栈破坏检查——金丝雀

3.10 缓冲区溢出攻击

43. 下列 C 程序存在安全漏洞，请给出攻击方法。如何修复或防范？

```
int getbuf(char *s) {
    char    buf[32];
    strcpy( buf, s );
}
```

攻击方法：采用基于缓冲区溢出攻击，让输入字符串 s 的字符个数大于 32，可导致 getbuf 函数返回到无关的代码处，或返回到指定的攻击代码处。

修复:限制字符串操作的长度可编码缓冲区溢出的攻击，如：

```
int getbuf_s(char *s) {
    char    buf[32];
    if(strlen(s)<sizeof(buf))
        strcpy( buf, s );
}
```

系统级的防范措施：栈空间地址的随机偏移、将栈 stack 标记为不可执行、或在栈中某个位置放入特定的金丝雀值。

46. 有下列 C 函数：

```
long arith(long x, long y, long z)
{
    long t1 = ________(1)________;
    long t2 = ________(2)________;
    long t3 = ________(3)________;
    long t4 = ________(4)________;
    ____________(5)____________;
}
```

函数 arith 的汇编代码如下：

```
arith:
    orq    %rsi, %rdi
    sarq   $3, %rdi
    notq   %rdi
    movq   %rdx, %rax
    subq   %rdi, %rax
    retq
```

请填写出上述 C 语言代码中缺失的部分

（1）___x|y_________ （2）___t1>>3_________ （3）___~t2_________

（4）____z-t3________ （5）___return t4_________

27. 阅读如下函数 myproc 的汇编代码，在横线上说明对应语句的功能（5 分）

```
myproc:
    movl    $0, %edx            ________①________
    movl    $0, %eax
    jmp  .L2
.L3:
    addq(%rdi,%rdx,8), %rax    ________②________
    addq$1, %rdx               ________③________
.L2:
    cmpq    %rsi, %rdx          ________④________
    jl   .L3                    ________⑤________
    ret
```

28. 写出 27 题函数 myproc 的 C 语言源程序（5 分）

27 题(5 分)

①寄存器%edx 赋值 0

②内存%rdi+%rdx*8 处的 8 字节数值与寄存器%rax 数值相加，结果保存在%rax 中

③将寄存器%rdx 的数值+1

④比较指令，用%rdx-%rsi 的结果设置标志位

⑤如果小于，则跳转至.L3

28 题(5 分)

答：

```
long myproc0(long x[], long n){
    long val = 0;
    long i;

    for(i=0;i<n; i++){
        val += x[i];
    }
    return val;
}
```

第5章 优化程序性能

运算	整数			浮点数		
	延迟	发射	容量	延迟	发射	容量
加法	1	1	4	3	1	1
乘法	3	1	1	5	1	2
除法	3~30	3~30	1	3~15	3~15	1

图 5-12 参考机的操作的延迟、发射时间和容量特性。延迟表明执行实际运算所需要的时钟周期总数，而发射时间表明两次运算之间间隔的最小周期数。容量表明同时能发射多少个这样的操作。除法需要的时间依赖于数据值

- CPE：每个元素的周期数
- T=CPE*n+经常开销
- 延迟：表示完成运算所需要的总时间
- 发射时间：连续两个同类型运算之间需要的最小间隔（时钟周期数）
- 发射时间为1的功能单元被称为完全流水线化的
- 容量：该运算的功能单元数
- 延迟界限：当一系列操作必须按照严格顺序执行时，代码中的数据相关限制了处理器利用指令级并行的能力时，延迟界限能够限制程序性能
- 吞吐量界限：发射时间/容量。这个界限是程序性能的终极限制

0.5

第5章 优化程序性能

- 代码移动，通过将代码从循环中溢出减少计算执行的频率
- 用简单方法替代复杂操作，如移位、加替代乘法或除法
- 共享共用子表达式，重用表达式的一部分
- 减少过程调用，小函数使用inline形式声明
- 消除不必要的内存引用
- 循环体展开减少循环次数
- 使用局部变量作为累积量
- 通过多个累积量、重新结合的方法，提高指令级并行性
- 尽量缩短关键路径
- 用条件表达式替代if-else语句（用功能性的风格重写条件操作，有利于编译器采用条件数据传送实现，而非使用分支结构实现）
- 使用速度更快的CPU指令，例如SSE等SIMD指令