CSAPP 笔记

apl

2025 年 4 月 14 日

目录

1 环境与工具

1.1 Linux 操作系统

1.1.1 什么是操作系统

现代计算机由一个或多个处理器、主存、磁盘以及各种输入输出设备组成。操作系统是管理这些组件，并将硬件抽象为统一软件接口的一层软件。

1.1.2 用户和用户组

Linux 系统是多用户多任务的分时操作系统。根用户账号是 `root`，拥有最高管理权限，为系统安全通常不直接使用根用户登录。每个用户都属于一个用户组，系统可对用户组中的用户进行集中管理。

1.1.3 文件和目录权限

在 Linux 中，文件和目录具有访问权限。每个文件包含九个权限位，分别定义所有者、所有者所在用户组和其他用户的访问权限。

```
-rwxr--r-- 1 user user 54130 Jan 8 12:14 README.md
```

最前面的 - 表示文件，`d` 表示目录，`l` 表示符号链接，后面九位每三位为分隔，`r` 表示读取，`w` 表示写入，`x` 表示执行，- 表示无权限。

1.1.4 文件和目录操作命令

1. **`ls` 命令：**用于列出指定目录下的子目录及文件名称，格式为 `ls [选项] 路径`，如 `ls -a` 可连同隐藏文件一起列出，`ls -l` 可列出文件和目录的属性与权限等详细数据。
2. **`cd` 命令：**改变当前工作目录，格式为`cd 路径`。
3. **`pwd` 命令：**显示当前工作目录。
4. **`mkdir` 命令：**创建新的目录，格式为 `mkdir [选项] 路径`，`-p` 可同时创建多级目录，如 `mkdir -p /home/user/documents/work/project`。
5. **`rmdir` 命令：**删除空的目录，格式为 `rmdir [选项] 路径`。
6. **`cp` 命令：**复制文件或目录，格式为 `cp [选项] 源路径 1 源路径 2 ……源路径 n 目标路径`，`-r` 选项用于递归复制目录下的所有子文件和子文件夹，`-f` 选项可强行覆盖目标文件。
7. **`rm` 命令：**删除文件或目录，格式为 `rm [选项] 路径 1 路径 2 ……路径 n`，删除文件夹需使用`-r` 选项，`-f` 选项可强行删除目标文件。
8. **`mv` 命令：**移动文件与目录，格式为 `mv 源路径 1 源路径 2 ……源路径 n 目标路径`，也可用于文件重命名，如 `mv file1 file5` 把 file1 重命名为 file5。

9. `chmod` **命令：**改变文件权限，格式为 `chmod 模式 路径`，模式如`[ugoa...][[+-=][rwx]...]`，`u` 表示文件拥有者，`g` 表示同用户组，`o` 表示其他用户，`a` 表示三者皆是，`+` 增加权限，`-`取消权限，`=`唯一设定权限。

10. `ln` **命令：**创建文件链接，格式为 `ln [选项] 源路径 目标路径`。Linux 系统允许创建文件链接，分为硬链接（类似文件别名）和软链接（类似快捷方式）。`-s` 选项用于创建软链接，`-f` 选项可强制执行。

11. `tar` **命令：**文件打包和拆包，如 `tar czf file.tar.gz file/`可将 `file` 文件夹下的所有内容压缩为 `file.tar.gz`，使用 `gzip` 压缩算法。如 `tar xzf file.tar.gz` 可将 `file.tar.gz` 解压缩到当前文件夹（拆包时参数 `z` 可省略）。

压缩算法	参数	后缀
gzip	z	.tar.gz
bzip2	j	.tar.bz2
Lempress-Ziv	Z	.tar.Z

表 1: 几种常见压缩算法参数

12. `zip/unzip` **命令：**文件打包和解包，如 `zip -q -r html.zip /home/html` 和 `unzip html.zip`。

13. `cat` **命令：**将文件以文本方式输出至控制台。

14. `hexdump` **命令：**将文件以编码形式输出至控制台。

15. `head` **命令：**显示文件前几行的数据。

16. `tail` **命令：**输出文件尾部的数据。

17. `objdump` **命令：**查看 `elf` 格式文件。

18. `more` **命令：**分页显示文件

19. `find` **命令：**根据条件搜索文件，如 `find ./test1 -name a.txt` 可在当前目录下的 `test1` 子目录中搜索 `a.txt` 文件。

20. `grep` **命令：**根据条件搜索文件内容（常用于搜索文本文件），如 `grep hello file.txt` 可在 `file.txt` 中查找字符串 `hello` 并打印匹配的行。

1.2 文本编辑器 vi/vim

1.2.1 VIM 的工作模式

1. **普通模式：**用于导航、删除、复制等，是默认模式

2. **查找模式：**用于搜索文本

3. **插入模式：**用于输入文本
4. **命令模式：**用于执行保存、退出等高级命令
5. **可视模式：**用于选择文本块

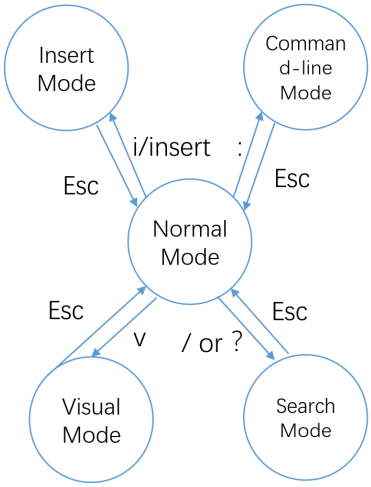

图 1 vim 的工作模式

1.2.2 各模式下的操作

1. **普通模式：**可使用上下左右键、`Home`、`End`、`PageUp`、`PageDown`、`gg`、`Shift + g` 等键移动光标，`yy` 复制当前行，`dd` 剪切当前行，`p` 粘贴，`u` 撤销，`Ctrl + r` 恢复撤销操作。
2. **插入模式：**按 `i` 键或 `Insert` 键进入，支持方向键等移动光标，按 `Insert` 键可切换插入和替换模式。
3. **查找模式：**按/或? 键进入，输入查找字符串并回车查找，按 `n` 键查找下一项，`Shift + n` 查找上一项。
4. **命令模式：**按: 键进入，输入 `w` 保存，`q` 退出，`wq` 保存并退出等命令。
5. **可视模式：**按 `v` 键进入，移动光标选择文本块，`y` 键复制，`d` 键剪切，普通模式下 `p` 键粘贴。

1.3 GNU 工具链

1.3.1 GNU 工具链简介

GNU 工具链是一组开源编程工具，由 GNU 计划提供，包含用于编译、调试和构建软件的实用程序，如 GCC（支持多种编程语言的编译器集合）、GNU Binutils（包括汇编器、连接器等）、GDB（调试程序的工具）、Make（用于构建和管理项目的工具，通过 Makefile 文件描述构建规则）。

1.3.2 构建程序

1. **单个源文件：**可使用 `gcc main.c` 编译并链接main.c，默认生成 `a.out`；也可使用 `gcc -o main main.c` 指定生成文件名为 `main`。

2. **多个源文件：**如编译a.c和b.c，可使用 `gcc a.c b.c -o p` 直接编译链接生成 `p` 文件；也可先 `gcc -c a.c b.c` 生成目标文件，再 `gcc -o p a.o b.o` 链接生成 `p` 文件。

1.3.3 增量编译

只编译修改的文件再链接，可提高编译效率。如只编译b.c，使用 `gcc -c b.c`，再 `gcc -o p a.o b.o` 链接。

1.3.4 引用自定义头文件与链接第三方库

引用自定义头文件时，使用`-I` 指定头文件搜索目录，如 `gcc -c a.c -Iinc`。链接第三方库时，使用`-l` 指定库名称，如 `gcc -o p a.o -lm` 链接数学库；若库不在默认搜索路径，使用`-L` 指定路径。

1.3.5 自动化编译

1. **编写 shell 脚本：**编写脚本可简化编译过程，但存在局限性，如处理复杂项目时脚本复杂、无法增量编译等。

2. **使用 Makefile:** 在工程根目录创建 Makefile 文件，编写规则后使用 `make` 命令编译。Makefile 中可定义变量，如 `CC`（编译器）、`CFLAGS`（编译选项）等，规则包含目标、依赖和命令，`make` 会根据依赖关系和时间戳确定编译操作。

1.3.6 在编译程序时增加调试信息

编译时添加`-g` 选项，如`CFLAGS = -Iinc -g`，可使编译器在生成目标文件时包含调试信息，便于使用 GDB 调试程序，如设置断点、查看变量值等。

1.4 代码版本管理

1.4.1 创建本地仓库与提交文件

使用 `mkdir` 创建本地工作目录，`cd`进入目录，`git init` 创建空的本地仓库。创建文件后使用 `git add` 将文件转换为 staged 状态，再用 `git commit` 提交至版本库，提交时可编辑提交说明。

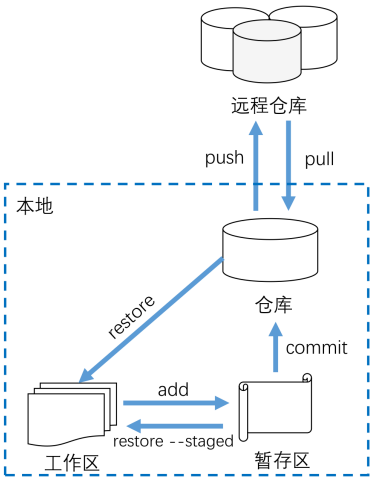

图 2 git 示意图

1.4.2 查看提交日志与代码还原

使用 `git log` 命令可查看提交日志，包含提交记录标识、作者、日期和提交说明等信息。使用 `git restore` 命令可进行代码还原，如 `git restore main.c` 用最近一次提交覆盖工作区文件，也可指定提交记录标识还原。

1.4.3 远程仓库操作

使用 `git remote add origin` 远程仓库地址关联远程仓库，首次推送使用 `git push -u origin master`，后续只需 `git push`。也可使用 `git clone` 命令克隆远程仓库至本地。

2 信息的存储

2.1 使用比特表示信息

2.1.1 万物皆比特

由于信号易存储在双稳态单元中，并且可以在存在噪声和不准确的信道中可靠地传输。信息都可以使用二进制的编码进行表示，计算机通过二进制来发送指令，以及表示和处理各种数字、字符串等。

2.1.2 字节数据编码

1 Byte = 8 bits。二进制表示范围是 00000000_2 到 11111111_2；十进制表示范围是 0_{10} 到 255_{10}；十六进制表示范围是 00_{16} 到 FF_{16}，十六进制以 16 为基数，计数符号为 0 - 9 和 A - F 。在 C 语言中，$FA1D37B_{16}$ 可表示为 0xFA1D37B 或 0xfa1d37b。

十进制	十六进制	二进制
0	00	00000000
1	01	00000001
2	02	00000010
3	03	00000011
4	04	00000100
5	05	00000101
6	06	00000110
7	07	00000111
8	08	00001000
9	09	00001001
10	0A	00001010
11	0B	00001011
12	0C	00001100
13	0D	00001101
14	0E	00001110
15	0F	00001111

表 2: 字节数据编码对应表

2.2 位运算

2.2.1 布尔代数

- And（与）：$A\&B = 1$ 当且仅当 $A = 1$ 且 $B = 1$。
- Or（或）：$A|B = 1$ 当 $A = 1$ 或者 $B = 1$。
- Not（非）：$\sim A = 1$ 当 $A = 0$。
- Exclusive - Or（Xor，异或）：$A\hat{}B = 1$ 当 $A = 1$ 或者 $B = 1$，但不同时为 1，即相同为 0，不同为 1。

2.2.2 C 语言中的位运算

C 语言定义了四个位运算符号：

C 表达式	二进制表达式	二进制结果	十六进制结果
$\sim$ 0x41	$\sim$[0100 0001]	[10111110]	0xBE
$\sim$ 0x00	$\sim$ [00000000]	[11111111]	0xFF
0x69&0x55	[01101001]&[01010101]	[01000001]	0x41
0x69 | 0x55	[01101001]|[01010101]	[01111101]	0x7D

表 3: C 语言位运算示例

2.2.3 异或运算的应用：数据交换

在 C 语言中，可以利用异或运算实现不使用额外变量交换两个数：

```
void funny(int *x, int *y)
{
    *x = *x ^ *y; /* #1 */
    *y = *x ^ *y; /* #2 */
    *x = *x ^ *y; /* #3 */
}
```

交换过程如下：

步骤	*x	*y
开始	A	B
1	A^B	B
2	A^B	(A^B)^B = A^(B^B) = A
3	(A^B)^A = (B^A)^A = B^(A^A) = B	A
结束	B	A

表 4: 异或运算交换数据过程

2.2.4 C 语言中的逻辑运算

C 语言定义了三种逻辑运算：||（逻辑或）、&&（逻辑与）、!（逻辑非），具有短路效应。例如：

- x && 5/x 可以用于避免除 0 运算。
- p && *p++ 可以避免空指针运算。
- 5 || x = y 赋值语句将不会被执行。

2.2.5 C 语言中的移位运算

C 语言中有逻辑移位和算术移位。右移运算有逻辑移位（左侧补 0）和算术移位（左侧补原最高位值）两种操作。对于无符号数，右移是逻辑的；对于有符号数，几乎所有的编译器针对有符号数的右移都采用的是算术右移

Operation	Values	
Argument x	[01100011]	[10010101]
x << 4	[0011*0000*]	[0101*0000*]
x >> 4 (logical)	[*0000*0110]	[*0000*1001]
x >> 4 (arithmetic)	[*0000*0110]	[*1111*1001]

图 3 移位运算

2.2.6 未定义行为

C 语言规范中没有被明确定义的行为称为未定义行为（UB），编程时应避免使用未定义行为，但有符号数算术右移除外。例如，移位 k，当 k 大于等于变量位长时，值直接变为 0，在 GCC 中的实现如下：

```
int aval = 0x0EDCBA98 >> 36;
movl $0, -8(%ebp)   // 值直接变为 0
unsigned uval = 0xFEDCBA98u << 40;
movl $0, -4(%ebp)   // 值直接变为 0
```

2.2.7 运算优先级

移位运算符的优先级低于加减乘除，例如 $-1 << 2+3 << 4$，正确的运算顺序为 $(1 << (2+3)) << 4$ 。

2.3 信息的存储和表示

2.3.1 字长

字长是指针数据的大小（虚拟地址宽度）。32 位 (4 字节) 计算机字长限制了地址空间为 4GiB（2^{32} 字节）；64 位字长 (8 字节) 寻址能力达到了 18EiB

2.3.2 C 语言中的各数据类型位宽

计算机和编译器支持多种数据类型，或是小于字长，或大于字长，但长度都是整数个字节。

数据类型	32 位系统	64 位系统	x86 - 64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	8	8
float	4	4	4
double	8	8	8
pointer	4	8	8

表 5: C 语言数据类型位宽

2.3.3 字节序

有小端序（Little endian）和大端序（Big endian）两种。小端序低地址存放低位数据，高地址存放高位数据；大端序低地址存放高位数据，高地址存放低位数据。例如，对于 0x1234567 ：

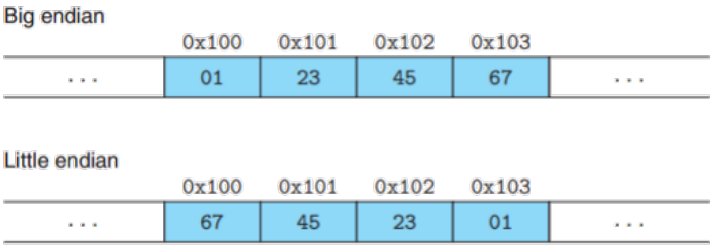

图 4 小端序和大端序

2.3.4 探索数据在存储器中的存储方式

通过以下代码可以打印各变量的字节表示形式：

```
#include <stdio.h>
typedef unsigned char *byte_pointer;    //unsigned char 占用一个字节，可以做到逐字节访问内存
void show_bytes(byte_pointer start,int len){
    int i;
    for(i = 0; i < len; i++)
        printf("%.2x ",start[i]);
    printf("\n");
}
void show_int(int x){
    show_bytes((byte_pointer)&x, sizeof(int));
}
void show_float(float x){
    show_bytes((byte_pointer)&x, sizeof(float));
}
void show_pointer(void *x){
    show_bytes((byte_pointer)x, sizeof(void*));
}
```

在 Linux32/64（小端）、Win32（小端）和 Sun（32 位，大端）系统下的测试数字 12345（0x00003039）结果如下。对于指针的存储，不同的编译器和计算机可能会分配不同的地址，甚至每一次运行时得到的结果都不相同。

机器	值	类型	字节（从左到右地址升高）
Linux 32	12345	int	39 30 00 00
Windows	12345	int	39 30 00 00
Sun	12345	int	00 00 30 39
Linux 64	12345	int	39 30 00 00
Linux 32	12345.0	float	00 e4 40 46
Windows	12345.0	float	00 e4 40 46
Sun	12345.0	float	46 40 e4 00
Linux 64	12345.0	float	00 e4 40 46
Linux 32	& ival	int *	e4 f9 ff bf
Windows	& ival	int *	b4 cc 22 00
Sun	& ival	int *	ef ff fa 0c
Linux 64	& ival	int *	b8 11 e5 ff ff 7f 00 00

表 6: 不同系统下数据存储测试结果

2.3.5 字符串的表示

C 语言的字符串使用 char 数组表示，每个字符都被编码成 ASCII 码，是一个 7 比特的字符编码集（扩展集为 8 比特）。字符“0”的编码是 0x30，数字字符 i 的编码是 0x30 $+i$ 。字符串的结尾应为空字符，即 ASCII 编码为 0 。字符串的表示与字节序无关，大小端兼容，因为每个 char 只占用一个字节，单独存储时和大小端无关

2.3.6 小知识：PE 和 ELF 格式

Windows 操作系统下常用的可执行文件格式是 PE（Portable Executable）；Unix 家族（含 Linux）操作系统下可执行文件格式为 ELF（Executable and Linkable Format）。

3 整数

3.1 编码

3.1.1 整数的表示

不同 C 语言数据类型在 32 位和 64 位机器上有不同的取值范围：

C 语言数据类型	32 位机器	64 位机器
char	$-128 \sim 127$	$-128 \sim 127$
unsigned char	$0 \sim 255$	$0 \sim 255$
short	$-32768 \sim 32767$	$-32768 \sim 32767$
unsigned short	$0 \sim 65535$	$0 \sim 65535$
int	$-2147483648 \sim 2147483647$	$-2147483648 \sim 2147483647$
unsigned int	$0 \sim 4294967295$	$0 \sim 4294967295$
long	$-2147483648 \sim 2147483647$	$-9223372036854775808 \sim 9223372036854775807$
unsigned long	$0 \sim 4294967295$	$0 \sim 18446744073709551615$

表 7: C 语言数据类型的取值范围

3.1.2 二进制转无符号数和有符号数（补码）

假设整数数据类型有 w 位，二进制编码用位向量 $\vec{x} = (x_{w-1}, x_{w-2}, \cdots, x_0)$ 表示。无符号数编码定义为：$B2U_w(\vec{x}) = \sum_{i=0}^{w-1} x_i 2^i$ 有符号数（补码）编码定义为：$B2T_w(\vec{x}) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$

例如：

$$B2U_4([0001]) = 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$

$$B2U_4([1011]) = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 11$$

$$B2T_4([0001]) = -0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$

$$B2T_4([1011]) = -1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = -5$$

3.1.3 将整数转换为补码

如果数字值非负，补码等于该值对应的二进制数（位长不足补 0）；如果数字值为负，其绝对值的二进制数逐位取反，并加 1，符号位置 1 。例如，假设整数类型位长为 4，整数 5 的补码是 0101，整数 -5 的补码是 1011。

3.1.4 可表示的整数范围

无符号数范围：$U_{min} = 0$，$U_{max} = 2^w - 1$

有符号数范围：$T_{min} = -2^{w-1}$，$T_{max} = 2^{w-1} - 1$

关系满足 $|T_{Min}| = T_{Max} + 1$，$U_{max} = 2 \times T_{Max} + 1$。例如：

Value	Word size $w=8$	$w=16$	$w=32$
U_{Max}	0xFF (255)	0xFFFF (65535)	0xFFFFFFFF (4294967295)
T_{Min}	0x80 (-128)	0x8000 (-32768)	0x80000000 (-2147483648)
T_{Max}	0x7F (127)	0x7FFF (32767)	0x7FFFFFFF (2147483647)
-1	0xFF	0xFFFF	0xFFFFFFFF
0	0x00	0x0000	0x00000000

表 8: 不同字长下的重要数字

3.1.5 反码和原码

反码：是符号位的权重为 $-(2^{w-1}-1)$。
原码：符号位的作用仅用于决定正/负，1 负 2 正。

3.2 变换

3.2.1 有符号数转无符号数

C 语言允许将有符号数转换为无符号数，转换时编码本身没有发生变化。非负数转换后值不变，负数转换为一个大整数。例如：

$$ux = \begin{cases} x, & x \geq 0 \\ x + 2^w, & x < 0 \end{cases}$$

3.2.2 无符号数转有符号数

无符号数转有符号数时编码保持不变，转换规则为：

$$x = \begin{cases} ux, & ux \leq T_{Max} \\ ux - 2^w, & ux > T_{Max} \end{cases}$$

3.2.3 C 语言中的有符号数和无符号数

缺省情况下，所有的整数常量都是有符号数。如果需要声明无符号数常量，需要增加一个后缀“U”，如 0U，4294967259U。C 语言中有显式转换和隐式转换。隐式转换时，编译器会产生 Warning，不推荐使用。例如：

```
int tx, ty;
unsigned ux, uy;
tx = (int)ux; // 显式转换
uy = (unsigned)ty; // 显式转换
tx = ux; // 隐式转换
uy = ty; // 隐式转换
```

3.2.4 C 语言中的表达式求值

如果在一个表达式中混用无符号数和有符号数，有符号数会被隐式转换成无符号数，比较运算也采用此规则。例如：

Constant1	Relation	Constant2	Evaluation
0	==	0U	unsigned
-1	<	0	signed
-1	>	0U	unsigned
2147483647	>	-2147483648	signed
2147483647U	<	-2147483648	unsigned
-1	>	-2	signed
(unsigned)-1	>	-2	unsigned

表 9: 表达式求值示例（以 32 位字长为例）

3.2.5 位扩展

无符号数位扩展是零扩展，扩展后最高位的空位补 0；
有符号数位扩展是符号位扩展，扩展后最高位的空位用原编码的符号位填充。例如：

变量	Decimal	Hex	Binary
short int x	12345	30 39	00110000 00111001
int ix	12345	00 00 30 39	00000000 00000000110000 00111001
short int y	-12345	CF C7	11001111 11000111
int iy	-12345	FF FF CF C7	11111111 11111111 11001111 11000111

表 10: 位扩展示例

3.2.6 位截断

无符号数和有符号数的位截断都是将高位的位直接丢弃，只保留低位的位。无符号数截断后的值为 $x \bmod 2^k$（k 为截断后保留的位数）；有符号数截断后的值为将截断后的无符号数转换为有符号数。例如：

```
int x = 53191;
short sx = (short)x; // 截断
int y = sx; // 再扩展
```

这里 $x = 53191$，二进制为 00000000000000001100111111000111，截断为 16 位后 sx 二进制为 1100111111000111，再扩展为 32 位 y 二进制为 11111111111111111100111111000111，值为 -12345。

3.3 运算

3.3.1 无符号数加法

对于无符号数 x 和 y，如果 $x+y<2^w$，则正常相加；如果 $x+y\geq 2^w$，则结果为 $(x+y) \bmod 2^w$，即产生了溢出。例如：

x	y	$x+y$	$x+y \bmod 2^4$
0	0	0	0
5	3	8	8
11	13	24	8
15	15	30	14

表 11: 无符号数加法示例（$w=4$）

3.3.2 有符号数加法

对于有符号数 x 和 y，可能会出现正溢出、正常和负溢出三种情况。

1. 正溢出时 $x+y\geq 2^{w-1}$，结果为 $x+y-2^w$；
2. 正常时 $-2^{w-1}\leq x+y<2^{w-1}$，结果为 $x+y$；
3. 负溢出时 $x+y<-2^{w-1}$，结果为 $x+y+2^w$。

例如：

x	y	x + y	补码加法结果
1	2	3	3
5	5	10	-6
-5	-5	-10	6
-1	-2	-3	-3

表 12: 有符号数加法示例（$w=4$）

3.3.3 无符号数乘法

无符号数 x 和 y 相乘，结果为 $(x\times y) \bmod 2^w$。例如在 $w=4$ 时，$x=3, y=5$，$x\times y=15$，结果就是 15；若 $x=5, y=6$，$x\times y=30$，结果为 $30 \bmod 16=14$。

3.3.4 有符号数乘法

有符号数 x 和 y 相乘，先按无符号数相乘得到结果 $p=x\times y$，然后将 p 截断为 w 位，再将截断后的无符号数转换为有符号数。例如在 $w=4$ 时，$x=-3, y=2$，无符号相乘 x 看作 13，$y=2$，$p=26$，截断为 4 位后为 10，转换为有符号数为 -6。

3.3.5 移位运算

在大多数计算机上移位运算和加法运算比乘法运算快得多。左移 k 位相当于乘以 2^k。对于无符号数右移 k 位使用逻辑右移，相当于除以 2^k，采用的是向下取整；对于有符号数右移 k 位，采用算术右移，相当于除以 2^k 并向下取整。例如：

```
int x = 12;
int y = x << 2; // y = 12 * 4 = 48
int z = x >> 2; // z = 12 / 4 = 3
```

对于有符号数除以 2^k，为了实现向零取整，可以在右移前加上一个偏置量 $(1 << k) - 1$。相当于右移 k 位后再加 1。代码实现如下：

```
int divide_power2(int x, int k) {
    int bias = (x < 0)? ((1 << k) - 1) : 0;
    return (x + bias) >> k;
}
```

4 浮点数

4.1 传统小数的二进制表示

在二进制中，小数点左侧每一位的权重为 2^i，右侧为 $\frac{1}{2^i}$（位置计数法），一个二进制数 b 可以表示为 $b = \sum_{i=-n}^{m} 2^i \times b_i$。例如：$5\frac{3}{4} = 101.11_2$ $2\frac{7}{8} = 010.111_2$

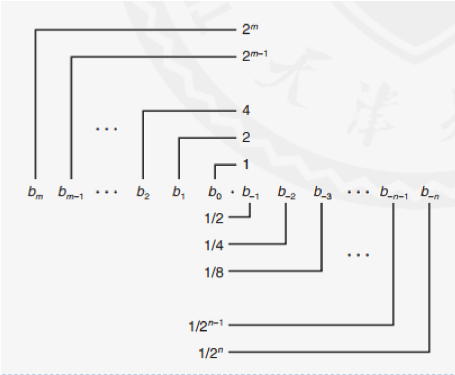

图 5 小数的二进制表示

并且，除以 2 可通过逻辑右移实现，乘以 2 可通过左移实现。像 $0.11111\cdots_2$ 这样的数字十分接近但小于 1，可表示为 $1.0 - \varepsilon$。同时，并非所有有理数都能精确地用有限位二进制小数表示，例如 $\frac{1}{3} = 0.0101010101[01]\cdots_2$，只能精确表示 $\frac{x}{2^k}$ 这种形式的数字。

4.2 IEEE 浮点数的表示

浮点数的表示形式为 $(-1)^s \times M \times 2^E$，其中：

- 符号位 s 决定了数字的正负。

- 尾数 M 通常在 $[1.0, 2.0)$ 或 $[0.0, 1.0)$。
- 阶码 E 是浮点数的权重，为 2 的 E 次幂。

图 6 浮点数编码

编码方式为：最高位是符号位 s，exp 编码后得到 E（exp $\neq E$），frac 编码后得到 M（frac $\neq M$）。

4.2.1 几种精度的浮点数

常见的浮点数精度有：

精度类型	总位数	符号位	exp 位数	frac 位数
单精度	32 位	1 位	8 位	23 位
双精度	64 位	1 位	11 位	52 位
扩展精度（仅 Intel 支持）	80 位	1 位	15 位	63 或 64 位

表 13: 不同精度浮点数的位宽分布

4.2.2 规格化数

当 exp $\neq 000\cdots0$ 且 exp $\neq 111\cdots1$ 时：

- 尾数编码为包含一个隐式前置的 1，即 $M = 1.x_1x_2\cdots$，其中 $x_1x_2\cdots$ 为 frac 域的各位的编码。
- 阶码为一个有偏置的指数，$E = \text{Exp} - \text{Bias}$，Exp 为 exp 域无符号数编码值，$\text{bias} = 2^{k-1}-1$，$k$ 是 exp 的位宽。
- 例如，对于单精度浮点数，bias $= 127$（Exp : $1\cdots254$，E : $-126\cdots127$）；双精度浮点数，bias $= 1023$（Exp : $1\cdots2046$，E : $-1022\cdots1023$）。隐式前置的整数 1 始终存在，因此在 frac 中不需要包含。

以 float $f = 15213.0$ 为例：

$$
\begin{aligned}
f = 15213_{10} &= 11011011011010000000000_2 \\
&= 1.1101101101101101_2 \times 2^{13} \\
\text{尾数} \quad M &= 1.\mathbf{1101101101101101}_2 \\
\text{frac} &= \mathbf{1101101101101}0000000000_2 \\
\text{阶码} \quad E &= 13, \\
\text{Bias} &= 2^{k-1} - 1 = 127 \quad (k = 8), \\
\text{Exp} &= 140 = 10001100_2
\end{aligned}
$$

结果为 0 10001100 11011011011010000000000。

4.2.3 非规格化数

当 $\exp = 000\cdots0$ 时：

- 阶码 $E = -\text{Bias} + 1$（而不是 $E = 0 - \text{Bias}$）。
- 尾数编码为包含一个隐式前置的 0，即 $M = 0.x_1x_2\cdots$。
- 当 $\exp = 000\cdots0$，且 $\text{frac} = 000\cdots0$ 时，表示 0，要注意 $+0$ 和 -0 的区别。
- 当 $\exp = 000\cdots0$，且 $\text{frac} \neq 000\cdots0$ 时，表示非常接近于 0.0 的数字，这些数字是等间距的。

4.2.4 特殊值

当 $\exp = 111\cdots1$ 时：

- 当 $\exp = 111\cdots1$ 且 $\text{frac} = 000\cdots0$ 时，表示无穷 ∞，意味着运算出现了溢出，有正向溢出和负向溢出，例如 $1.0/0.0 = -1.0/-0.0 = +\infty$，$1.0/-0.0 = -\infty$。
- 当 $\exp = 111\cdots1$ 且 $\text{frac} \neq 000\cdots0$ 时，表示不是一个数字（NaN），表示数值无法确定，例如 $-1, \infty - \infty, \infty \times 0$。

4.2.5 IEEE 编码的特殊属性

1. 浮点数 0 和整数 0 编码相同，所有位都为 0
2. 几乎可以用无符号整数比较的方法实现浮点数的比较运算，但首先要比较符号位，同时要考虑 $-0 = 0$ 和 NaN 的问题 (NaN 比其他值都大)

4.2.6 需要关注的数字

不同类型的浮点数有一些特殊的数字：

描述	exp	frac	单精度值（十进制）	双精度值（十进制）
零	$00\cdots00$	$0\cdots00$	0.0	0.0
最小非规格化数	$00\cdots00$	$0\cdots01$	$2^{-23} \times 2^{-126}$	$2^{-52} \times 2^{-1022}$
最大非规格化数	$00\cdots00$	$1\cdots11$	$(1-\varepsilon) \times 2^{-126}$	$(1-\varepsilon) \times 2^{-1022}$
最小规格化数	$00\cdots01$	$0\cdots00$	1×2^{-126}	1×2^{-1022}
一	$01\cdots11$	$0\cdots00$	1.0	1.0
最大规格化数	$11\cdots10$	$1\cdots11$	$(2-\varepsilon) \times 2^{127}$	$(2-\varepsilon) \times 2^{1023}$

表 14: 不同精度浮点数的特殊值

4.2.7 举例：一个微型的浮点数编码系统

以 8 位浮点数编码为例，最高位为符号位，接下来是 4 位 exp，偏置 Bias 为 7，最后 3 位是 frac，与 IEEE 规范具有相同的形式，有规格化数、非规格化数，以及 0、NaN 和无穷的编码。

s	exp	frac	E	值
0	0000	000	-6	0
0	0000	001	-6	$\frac{1}{8} \times \frac{1}{64} = \frac{1}{512}$（最接近 0）
⋮	⋮	⋮	⋮	⋮
0	1111	000	n/a	inf

表 15: 8 位浮点数编码示例

4.2.8 总结

单精度浮点数可分为以下几类：

1. 规格化数：exp $\neq$ 0000000 且 exp $\neq$ 11111111。
2. 非规格化数：exp = 00000000，frac $\neq$ 0000000。
3. 无穷：exp = 11111111，frac = 00000000。
4. NaN：exp = 11111111，frac $\neq$ 00000000。

4.3 几种舍入模式

常见的舍入模式有：

1. 向下舍入：舍入结果接近但不会大于实际结果。
2. 向上舍入：舍入结果接近但不会小于实际结果。
3. 向 0 舍入：舍入结果向 0 的方向靠近，如果为正数，舍入结果不大于实际结果；如果为负数，舍入结果不小于实际结果。
4. 向偶数舍入：浮点数运算默认的舍入模式，其他的舍入模式都会统计偏差，一组正数的总和将始终被高估或低估。

向偶数舍入适用于舍入至小数点后任何位置，当数字正好处在四舍五入的中间时，向最低位为偶数的方向舍入。例如：

值	结果	说明
1.2349999	1.23	比中间值小，四舍
1.2350001	1.24	比中间值大，五入
1.2350000	1.24	中间，向上舍入（偶数方向）
1.2450000	1.24	中间，向下舍入（偶数方向）

表 16: 向偶数舍入示例

在二进制数中，偶数方向意味着舍入后最后一位为 0，中间意味着待舍入的部分为 $100\cdots_2$。

4.4 浮点数运算

浮点数运算的基本思想是先计算出精确的值，然后将结果调整至目标的精度。如果阶码值过大，可能会导致溢出，可能会进行舍入以满足尾数的位宽。例如：

$$x +^f y = \text{Round}(x + y)$$
$$x \times^f y = \text{Round}(x \times y)$$

4.4.1 浮点数乘法

对于浮点数 $(-1)^{s_1} M_1 2^{E_1} \times (-1)^{s_2} M_2 2^{E_2}$：

- 精确结果：$M_1 \times M_2$，阶码 $E = E_1 + E_2$，符号位 $s = s_1^{s_2}$。
- 修正：如果 $M \geq 2$，右移 M，并增大 E 的值；如果 E 超出范围，发生溢出；对 M 进行舍入以满足 frac 的位宽精度要求。在实际实现中，尾数相乘的细节较为繁琐。

4.4.2 浮点数加法

对于浮点数 $(-1)^{s_1} M_1 2^{E_1} + (-1)^{s_2} M_2 2^{E_2}$（假设 $E_1 > E_2$）：

- 精确结果：$(-1)^s M_2^E$，其中符号位 s 和尾数 M 是有符号数对齐后相加的结果，阶码 $E = E_2$。
- 修正：如果 $M \geq 2$，右移 M，并增大 E 的值；如果 $M < 1$，左移 M k 位，然后 E 减去 k；如果 E 超出范围，发生溢出；对 M 进行舍入以满足 frac 的位宽精度要求。

4.5 C 语言中的浮点数

C 语言标准确保支持两种精度的浮点数，即 float（单精度）和 double（双精度）。在进行类型转换时：

- double/float 转 int：截断尾数部分，向 0 舍入。标准中未定义越界和 NaN 的情况，通常设置为 T_{Min} 和 T_{Max}。

- int 转 double：只要 int 的位宽小于等于 53 位，就能精确转换。
- int 转 float：会根据舍入模式进行舍入。

以下是一些 C 语言中浮点数操作的示例（假设 d 和 f 分别是 float 和 double 且不是 NaN 和无穷）：

表达式	结果
`x == (int)(float) x`	否：有效数字为 24 位
`x == (int)(double) x`	是：有效数字为 53 位
`f == (float)(double) f`	是：提高精度
`d == ( float ) d`	否：丢失精度
`f == -(-f)`	是：仅改变符号位
`2/3 == 2/3.0`	否：$2/3 == 0$
`if(d < 0.0) ((d*2) < 0.0)`	是
`d * d >= 0.0`	是
`(d + f) - d == f`	否：不满足结合律

表 17: C 语言中浮点数操作示例结果

在浮点数运算中，存在精度和准确度的问题。例如 $0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 \neq 1$，但 $0.2+0.2+0.2+0.2+0.2 = 1$。在使用浮点数进行比较时，如`if (d1 - d2 == 0)`，需要谨慎处理，因为浮点数的精度限制可能导致结果不准确。在进行浮点数运算时，要充分考虑精度损失对计算结果的影响。

5 程序的机器级表示: 基础知识

5.1 Intel 处理器体系结构的历史

5.1.1 复杂指令集计算机 (CISC)

Intel x86 系列属于复杂指令集计算机，其指令多、格式复杂。理论上 CISC 的性能较难与精简指令集计算机 (RISC) 相比。在低功耗场景下，其速度会受到影响。

5.1.2 摩尔定律

摩尔定律指出，单位面积上可以容纳的晶体管数量几乎每两年增加一倍

5.2 C 语言, 汇编语言和机器语言

5.2.1 定义

1. 体系结构（指令集体系结构，ISA）：编写汇编代码时需要理解的处理器设计部分，如指令集规范、寄存器组织等。

2. 微体系结构：体系结构的具体实现，例如高速缓存大小、核心频率等。

3. 机器语言是处理器可以直接执行的字节级程序，汇编语言是文本形式的机器语言。

常见的指令集体系结构有 Intel 的 x86、IA32、Itanium、x86 - 64，以及 ARM（用于几乎所有移动电话）。

5.2.2 程序员可见的状态

1. 程序计数器（PC，在 x86 - 64 中称为 RIP），存储下一条要执行指令的地址

2. 条件码，存储最近一次算术逻辑运算的状态信息，用于条件分支

3. 存储器，基于字节寻址，存储程序、用户数据和栈数据

4. 寄存器文件，频繁用于存储程序数据

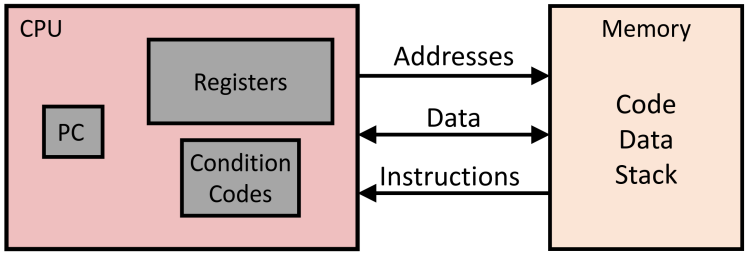

图 7

5.2.3 汇编语言的数据类型

1. 整数：1、2、4 或 8 字节

2. 地址：无类型的指针

3. 浮点数：4、8 或 10 字节

4. 代码：指令的字节序列编码

与 C 语言不同，汇编语言没有聚合类型，数组或结构体在汇编语言中表现为在内存中连续分配的字节。

5.2.4 汇编语言的操作

汇编语言可以对寄存器或存储器数据执行算术/逻辑运算，在寄存器和存储器间传输数据（包括将数据从存储器加载至寄存器以及将寄存器的数据存储至存储器），还能进行转移控制，如无条件跳转至/从过程、条件分支等。

5.2.5 举例: 机器指令

对于 C 语言代码 `*dest = t;`，对应的汇编代码 `movq %rax, (%rbx)`，将 8 字节数据移动至存储器。其操作数 t 存储在寄存器 `%rax`中，dest 存储在寄存器 `%rbx` 中，*dest 表示内存地址 `M[%rbx]`。对应目标码（机器指令）为`0x40059e: 48 89 03`，该指令为 3 字节，存储于地址 0x40059e。

5.2.6 反汇编

反汇编器是探索目标码的有用工具，它可以分析指令的编码序列，根据目标码重新生成汇编代码，并且可以对任何可执行程序文件和 `.o` 文件进行反汇编。例如使用 `objdump -d sum` 对 sum 文件进行反汇编，或者在 gdb 调试器中使用 `gdb sum` 以及 `disassemble sumstore` 反汇编 sumstore 函数。反汇编程序会分析字节并重构为汇编代码。

6 程序的机器级表示: 基本操作

6.1 x86 寄存器

6.1.1 x86 - 64 寄存器

x86 - 64 架构下，每个寄存器的低 4/2/1 字节都有唯一的标识。其包含多个寄存器，如 `%rax`、`%rbx`、`%rcx` 等，可以用于存储计算结果、函数参数传递、循环计数等。其中 `%rsp` 是栈指针寄存器，有使用限制。寄存器具体如下：

63 - 31 位	31 - 15 位	15 - 7 位	7 - 0 位
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bp
%rsp	%esp	%sp	%spl
%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

表 18: x86 - 64 寄存器各部分标识

6.1.2 IA32 (x86 - 32) 寄存器

IA32（x86 - 32）架构下的寄存器有 %eax、%ecx、%edx 等，其各部分标识如下：

31 - 15 位	15 - 7 位	7 - 0 位
%eax	%ax	%al
%ecx	%cx	%cl
%edx	%dx	%dl
%ebx	%bx	%bl
%esi	%si	-
%edi	%di	-
%esp	%sp	-
%ebp	%bp	-

表 19: IA32 (x86 - 32) 寄存器各部分标识

6.2 数据移动指令

6.2.1 汇编语言格式

汇编语言格式为 [label :] [opcode] [operand 1] [, operand 2]，即

标号: 操作码 操作数1 [, 操作数2]

不同风格的汇编语句结构有差异。例如，在 ATT assembly 中：

```
l1: movq $5, %rax
addq $-16, (%rax)
```

在 Intel assembly 中：

```
l1: mov rax, 5
add QWORD PTR[rax], -16
```

6.2.2 操作数类型

1. **立即数**：整数常量，如 $0x400、$-533 ，和 C 语言中的常数类似，但需要加前缀 $，其被编码为 1、2、4 或 8 个字节。

2. **寄存器**：十六个整数寄存器之一，如 %rax、%r13 。其中 %rsp 有特殊用途，通常不使用，其他寄存器在一些特殊的指令中也会有特殊用途。

3. **存储器**：指向的内存中 8 个连续字节，由寄存器给出地址，例如 (%rax) ，还有很多其他的“寻址模式”。

6.2.3 movq 指令操作数的几种组合

源操作数 Src	目标操作数 Dest	示例	等价 C 语言
立即数	寄存器	`movq $0x4, %rax`	`temp = 0x4;`
立即数	存储器	`movq $-147, (%rax)`	`*p = -147;`
寄存器	寄存器	`movq %rax, %rdx`	`temp2 = temp1;`
寄存器	存储器	`movq %rax, (%rdx)`	`*p = temp;`
存储器	寄存器	`movq (%rax), %rdx`	`temp = *p;`

表 20: movq 指令操作数组合及等价 C 语言表达

6.2.4 数据格式

不同 C 语言类型声明对应不同的数据类型、操作码后缀和大小，具体如下：

C 语言类型声明	数据类型	操作码后缀	大小 (bytes)
`char`	Byte	b	1
`short`	Word	w	2
`int`	Double Word	l	4
`long`	Quad Word	q	8
`char *`	Quad Word	q	8
`float`	Single precision	s	4
`double`	Double precision	l	8

表 21: C 语言类型与数据格式对应关系

6.2.5 几种简单的存储器寻址模式

1. **间接寻址：** `(R)` 表示 `Mem[Reg[R]]` ，寄存器 `R` 指向了存储器的地址，和 C 语言中的指针作用相同，例如 `movq (%rcx),%rax` 。

2. **基地址 + 偏移量寻址：** `D(R)` 表示 `Mem[Reg[R]+D]` ，寄存器 `R` 指定了存储器区域的开始位置，常数 `D` 是偏移量，例如 `movq 8(%rbp),%rdx` 。

6.2.6 举例：简单寻址模式（swap 函数）

以 `swap` 函数为例：

```
void swap(long *xp, long *yp) {
    long t0 = *xp;
    long t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

其汇编代码如下：

```
swap:
    movq (%rdi), %rax       # t0 = *xp
    movq (%rsi), %rdx       # t1 = *yp
    movq %rdx, (%rdi)       # *xp = t1
    movq %rax, (%rsi)       # *yp = t0
    ret
```

在该函数中，%rdi 存放 xp 的值，%rsi 存放 yp 的值，%rax 用于存放 t0 ，%rdx 用于存放 t1 。通过逐步执行这些指令，实现两个指针所指向的值的交换。

6.2.7 完整的存储器寻址模式

最通用的形式为 D(Rb, Ri, S) 表示 Mem[Reg[Rb]+S * Reg[Ri]+D] ，其中：

1. D：常数偏移量，可以为 1、2、4 或 8 字节整数。
2. Rb：基地址寄存器，是 16 个寄存器之一。
3. Ri：变址寄存器，除 %rsp 外的其他寄存器。
4. S：比例因子，可以为 1、2、4 或 8。

还有一些特殊形式，如 (Rb, Ri) 表示 Mem[Reg[Rb] + Reg[Ri]] ，
D(Rb, Ri) 表示 Mem[Reg[Rb] + Reg[Ri] + D] ，(Rb, Ri, S) 表示 Mem[Reg[Rb] + S * Reg[Ri]] 。

6.2.8 小练习：地址计算

假设 %rdx = 0xf000，%rcx = 0x0100 ，不同地址表达式的计算结果如下：

表达式	地址计算	地址
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

表 22: 不同地址表达式的计算结果

6.3 算术、逻辑运算指令

6.3.1 地址计算指令（leaq）

leaq 指令格式为 leaq Src, Dst，其中 Src 是寻址模式表达式，该指令将表达式计算的地址写入 Dst 。其用途包括计算地址（计算过程中不需要引用存储器），例如 p = & x[i] ；还可计算模式为 x + k*y（k = 1, 2, 4 或 8）的表达式。例如，对于函数：

```
long m12(long x)
{
    return x*12;
}
```

编译后的汇编指令为：

```
leaq    (%rdi,%rdi,2), %rax    # t <- x+x*2
salq    $2, %rax               # return t<<2
```

6.3.2 一些算术运算指令（两操作数指令）

两操作数指令（双目运算）的格式及计算方式如下：

格式	计算	说明
addq Src, Dest	Dest = Dest + Src	-
subq Src, Dest	Dest = Dest - Src	-
imulq Src, Dest	Dest = Dest * Src	-
salq Src, Dest	Dest = Dest << Src	也叫 shlq
sarq Src, Dest	Dest = Dest >> Src	算术右移
shrq Src, Dest	Dest = Dest >> Src	逻辑右移
xorq Src, Dest	Dest = Dest ^ Src	-
andq Src, Dest	Dest = Dest & Src	-
orq Src, Dest	Dest = Dest | Src	-

表 23: 两操作数算术运算指令

需要注意操作数的顺序，并且有符号数和无符号数指令没有区别。

6.3.3 一些算术运算指令（单操作数指令）

单操作数指令（单目运算）的格式及计算方式如下：

格式	计算
incq Dest	Dest = Dest + 1
decq Dest	Dest = Dest - 1
negq Dest	Dest = - Dest
notq Dest	Dest = ~Dest

表 24: 单操作数算术运算指令

更多指令可查阅教材。

6.3.4 需要关注的指令及举例：算术运算

需要关注的指令有 leaq（地址计算）、salq（左移）、imulq（乘法）等。以函数：

```c
long arith(long x, long y, long z)
{
    long t1 = x+y;
    long t2 = z+t1;
    long t3 = x+4;
    long t4 = y * 48;
    long t5 = t3 + t4;
    long rval = t2 * t5;
    return rval;
}
```

```asm
arith:
    addq %rsi, %rdi       # t1 = x + y
    addq %rdi, %rdx       # t2 = z + t1
    leaq 4(%rdi), %rax    # t3 = x + 4
    leaq (%rsi,%rsi,2), %rcx  # t4a = y + 2*y
    salq $4, %rcx         # t4 = t4a << 4
    addq %rcx, %rax       # t5 = t3 + t4
    imulq %rax, %rdx      # rval = t2 * t5
    movq %rdx, %rax       # return rval
    ret
```

7 程序的机器级表示: 控制

7.1 条件码

7.1.1 处理器状态与条件码概述

在 x86 - 64 架构中，处理器状态包含多个部分，存储当前程序的执行信息

1. 临时数据，`%rax` 等
2. 运行时栈的位置（栈顶），`%rsp`
3. 当前代码控制点的位置（即将要执行的指令地址），`%rip`
4. 条件码，包括 CF、ZF、SF、OF

条件码寄存器是简单的位寄存器，通过算术运算、比较指令和测试指令等进行设置。

7.1.2 条件码含义

1. CF: 进位标志 (Carry Flag)。最近的操作使无符号数最高位产生了进位，可用来检查无符号操作的溢出
2. ZF: 零标志 (Zero Flag)。最近的操作得出的结果为 0
3. SF: 符号标志 (Sign Flag)。最近的操作得到的结果为负数
4. OF: 溢出标志 (Overflow Flag)。最近的操作导致一个有符号数补码溢出

7.1.3 条件码的设置方式

1. **算术运算隐式设置：** `addq Src, Dest`，相当于 $t = a + b$

   这类算术运算指令会隐式设置条件码。若运算时出现无符号数运算溢出，CF 被置位；若 $t == 0$，ZF 被置位；若有符号数 $t < 0$（，SF 被置位；若有符号数运算出现溢出，即 `(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)`，OF 被置位。需要注意的是，`\leaq` 指令不修改条件码。

2. **比较指令显式设置：** `cmpq Src2, Src1`（如 `cmpq b,a`）

   指令类似于计算 $a-b$，但不将结果写入目标寄存器，而是根据计算结果显式设置条件码。当运算时出现超出最高位的借位（用于无符号数比较），CF 被置位；若 $a==b$，ZF 被置位；若 $(a-b)<0$（看做有符号数），SF 被置位；若有符号数运算出现溢出，即 `(a > 0 && b < 0 && (a - b) < 0) || ( a < 0 && b > 0 && (a - b) > 0)`，OF 被置位。

3. **测试指令显式设置：** `testq Src2, Src1`（如 `testq b,a`）

   指令类似于计算 $a\&b$，同样不将结果写入目标寄存器，而是根据 $a\&b$ 的结果设置条件码。常用于对一个操作数的某几个位进行掩码检测，当 $a\&b==0$ 时，ZF 被置位；若 $(a\&b)<0$，SF 被置位。

7.1.4 读取条件码与 SetX 指令

SetX 指令可根据条件码表达式将目标寄存器的最后一个字节修改为 0 或 1，且不会影响目标寄存器最高 7 个字节的值。常见的 SetX 指令如下：

SetX 指令	条件	描述
`sete`	`ZF`	相等/为零
`setne`	`~ZF`	不相等/不为零
`sets`	`SF`	负数
`setns`	`~SF`	非负数
`setg`	`~(SF^OF) & ~ZF`	大于（有符号）
`setge`	`~(SF^OF)`	大于或等于（有符号）
`setl`	`(SF^OF)`	小于（有符号）
`setle`	`(SF^OF)|ZF`	小于或等于（有符号）
`seta`	`~CF & ~ZF`	高于（无符号）
`setb`	`CF`	低于（无符号）

表 25: SetX 指令与条件码关系

以函数 gt 为例：

```c
int gt(long x, long y){
    return x > y;
}
```

```
gt:
    cmpq    %rsi, %rdi      # Compare x:y
    setg    %al             # Set when >
    movzbl  %al, %eax       # Set rest of %rax
    ↪  zero
    ret
```

在 x86 - 64 指令集中，32 位操作指令会将目标寄存器的高 32 位清 0。

7.2 条件分支

7.2.1 跳转指令（jX 指令）

jX 指令根据条件码跳转到代码的其他位置执行，常见的 jX 指令如下：

jX 指令	条件	描述
jmp	1	无条件跳转
je	ZF	相等/为零则跳转
jne	~ZF	不相等/不为零则跳转
js	SF	负数则跳转
jns	~SF	非负数则跳转
jg	~(SF^OF)\&~ZF	大于（有符号）则跳转
jge	~(SF^OF)	大于或等于（有符号）则跳转
jl	(SF^OF)	小于（有符号）则跳转
jle	(SF^OF)|ZF	小于或等于（有符号）则跳转
ja	~CF\&~ZF	高于（无符号）则跳转
jb	CF	低于（无符号）则跳转

表 26: jX 指令与条件码关系

7.2.2 条件分支示例（早期模式）

对于 C 语言函数：

```c
long absdiff(long x, long y){
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

```c
// goto version
long absdiff_j(long x, long y){
    long result;
    int ntest = x <= y;
    if (ntest)
        goto Else;
    result = x - y;
    goto Done;
    Else:
        result = y - x;
    Done:
        return result;
}
```

使用 `gcc -Og -S -fno-if-conversion control.c` 生成的汇编代码如下（`%rdi`表示 x，`%rsi`表示 y，`%rax`表示 return 的值）：

```
absdiff:
    cmpq    %rsi, %rdi      # x:y
    jle     .L4             # x <= y 跳转 L4
    movq    %rdi, %rax      # ret <- x
```

```
    subq    %rsi, %rax      # ret -= y
    ret
.L4:                        # x <= y
    movq    %rsi, %rax      # ret <- y
    subq    %rdi, %rax      # ret -= x
    ret
```

7.2.3 条件表达式的翻译（使用分支）

对于条件表达式 `val = Test ? Then_Expr : Else_Expr;`，可以使用 `goto` 语句版本实现：

```
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
    val = Else_Expr;
Done:
```

这种方式为 `Then` 和 `Else` 表达式创建独立的代码块，根据条件选择合适的一个代码块并执行。

7.2.4 条件数据移动指令

在计算机处理器中，流水线技术允许最多有三条指令同时执行，提高了指令执行效率。然而，条件分支会破坏流水线的指令流，影响处理器性能。而条件数据移动指令不需要改变控制流，在某些情况下能避免这种性能损耗。

条件数据移动指令功能为 `if (Test) {Dest <- Src}` 。GCC 在编译时会尝试使用该指令翻译条件分支，但仅在保证逻辑安全的时候使用。以 `absdiff` 函数为例，使用条件数据移动指令的汇编代码如下：

```
absdiff:
    movq    %rdi, %rax      # x
    subq    %rsi, %rax      # result = x - y
    movq    %rsi, %rdx
    subq    %rdi, %rdx      # eval = y - x
    cmpq    %rsi, %rdi      # x:y
    cmovle  %rdx, %rax      # if x<=y, result = eval
    ret
```

不过，在一些情况下不适合使用条件数据移动指令，比如存在大量计算、有风险的计算或有副作用的计算时。例如 `val = Test(x) ? Hard1(x) : Hard2(x);` 以及 `val = p ? *p : 0;` 还有 `val = x > 0 ? x*=7 : x+=3;` 等情况。

7.3 循环

7.3.1 Do - While 循环示例与编译结果

以计算无符号长整型数 `x` 编码中“1”的个数的函数为例：

```
long pcount_do(unsigned long x){
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

其使用 goto 语句等价表示为：

```
long pcount_goto(unsigned long x){
    long result = 0;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
        return result;
}
```

编译后的汇编代码如下：

```
movl        $0, %eax    # result = 0
.L2:                    # loop:
    movq    %rdi, %rdx
    andl    $1, %edx    # t = x & 0x1
    addq    %rdx, %rax  # result += t
    shrq    %rdi        # x >>= 1
    jne     .L2         # if (x) goto loop
    rep; ret
```

其中，\%rdi 用于传递参数 x，\%rax 用于存储 result 。

7.3.2 While 循环通用的翻译方式

1. “跳转到中间”翻译方法（使用 -Og 编译优化选项）

   对于 while (Test) Body 结构，翻译为 goto 版本如下：

   ```
   goto test;
   loop:
       Body
   test:
       if (Test)
       goto loop;
   done:
   ```

   以计算无符号长整型数 x 编码中“1”的个数的 while 循环函数为例：

```
// while 版本
long pcount_while(unsigned long x){
    long result = 0;
    while (x) {
        result += x & 0x1;
        x >>= 1;
    }
    return result;
}
```

```
// goto 版本
long pcount_goto_jtm(unsigned long x){
    long result = 0;
    goto test;
    loop:
        result += x & 0x1;
        x >>= 1;
    test:
        if(x) goto loop;
        return result;
}
```

与 do - while 循环相比，该方式在循环开始前先跳转至循环条件检测的位置。

2. **另一种翻译方法（使用 -O1 编译优化选项）**

对于 while (Test) Body 结构，翻译为 goto 版本为：

```
    if (!Test)
        goto done;
    loop:
        Body
    if (Test)
        goto loop;
    done:
```

以计算无符号长整型数 x 编码中“1”的个数的 while 循环函数为例，对应的此版本代码为：

```
long pcount_goto_dw(unsigned long x){
    long result = 0;
    if (!x) goto done;
    loop:
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    done:
        return result;
}
```

与 do - while 循环相比，这种方式在循环开始前先检测循环条件，再进入循环。

7.3.3 For 循环一般形式与转换

For 循环的一般形式为 for (Init; Test; Update) Body，可以转换为 while 循环和 goto 版本。例如：

```
#define WSIZE 8 * sizeof(int)
long pcount_for(unsigned long x){
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

```
// goto version
long pcount_for_gt(unsigned x) {
    int i;
    int result = 0;
    i = 0;
    if (!(i < WSIZE)) goto done;  //可以
    ↪ 被编译器优化
    loop:
    unsigned bit = (x >> i) & 0x1;
    result += bit;
    i++;
    if (i < WSIZE)
        goto loop;
    done:
    return result;
}
```

转换为 while 循环如下：

```
long pcount_for_while(unsigned long x){
    size_t i;
    long result = 0;
    i = 0;
    while (i < WSIZE)
    {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```

7.4 switch 语句

7.4.1 switch 的一般形式及跳转表

switch 语句的一般形式如下：

```
switch(x) {
    case val_0:
        Block 0;
    case val_1:
        Block 1;
      ......
    case val_(n-1):
        Block n-1;
}
```

对应的跳转表结构为：

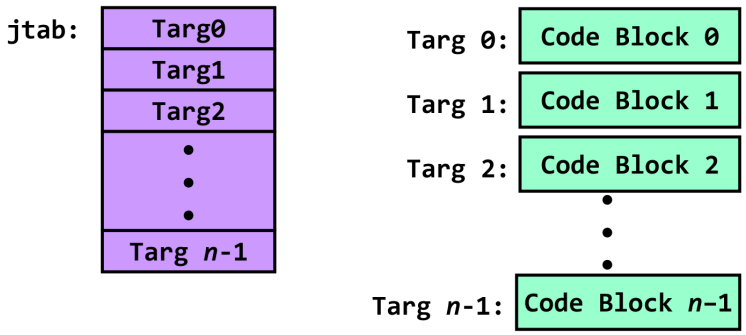

图 8 跳转表

翻译后（扩展 C）的形式为 `goto *JTab[op];`，可通过跳转表快速根据 `x` 的值跳转到相应的代码块执行。

7.4.2 分析跳转表

以 `switch_eg` 函数为例：

```
long switch_eg(long x, long y, long z) {
    long w = 1;
    switch(x) {
        ...
    }
    return w;
}
```

其汇编代码如下：

```
setup:
switch_eg:
    movq    %rdx, %rcx
    cmpq    $6, %rdi        # x:6
    ja      .L8             # Use default if x > 6
    jmp     *.L4(,%rdi,8)   # goto *JTab[x]
```

其中，`jmp *.L4(,%rdi,8)` 表示间接跳转到由 `.L4` 开始的跳转表中索引为 `x`（`%rdi` 存储 `x` 的值）的位置。当 `x` 大于 6 时，会跳转到 `.L8` 执行默认情况。注意，`w` 并没有在 `switch` 开始前初始化。

跳转表在汇编代码中的定义如下：

```
.section .rodata    # read-only data
    .align 8
.L4:
    .quad   .L8     # x = 0
    .quad   .L3     # x = 1
    .quad   .L5     # x = 2
    .quad   .L9     # x = 3
    .quad   .L8     # x = 4
    .quad   .L7     # x = 5
    .quad   .L7     # x = 6
```

跳转表的基地址是 .L4，每个跳转目标需要 8 个字节（因为在 64 位系统中，地址是 8 个字节）。间接跳转时，缩放因子必须是 8 的整倍数，通过计算地址 .L4 + x * 8 来获得跳转目标的位置（仅限于 0 <= x <= 6 的情况）。

7.4.3 代码块实现

1. 代码块（x == 1）

```
.L3:                                    case 1:
    movq    %rsi, %rax # y                  w = y*z;
    imulq   %rdx, %rax # y*z                break;
    ret
```

当 x 等于 1 时，执行此代码块，实现 w = y*z 的操作，其中 %rdi 为参数 x，%rsi 为参数 y，%rdx 为参数 z，%rax 用于存储返回值。

2. 处理落入另一个 case 的情况（x == 2,x == 3 ）

```
.L5:                    #Case2          case 2:
    movq    %rsi, %rax                      w = y/z;
    cqto                                    /* Fall Through */
    idivq   %rcx        # y/z           case 3:
    jmp     .L6         # goto merge        w += z;
.L9:                    # Case 3            break;
    movl    $1, %eax    # w = 1
.L6:                    # merge:
    addq    %rcx, %rax  # w += z
    ret
```

对于 case 2 和 case 3，case 2 执行完 w = y/z 后会继续执行 case 3 的代码，即跳转到 merge 标签处执行 w += z。

3. 代码块（x == 5,x == 6 , 缺省）

```
.L7:                    # Case 5,6      case 5:
    movl    $1, %eax    # w = 1         case 6:
    subq    %rdx, %rax  # w -= z            w -= z;
    ret                                     break;
.L8:                    # Default:      default:
    movl    $2, %eax    # 2                 w = 2;
    ret
```

当 x 等于 5 或 6 时，执行 w -= z; 当 x 不满足任何 case 条件（即默认情况）时，执行 w = 2。

7.4.4 特殊情况处理

1. **没有从 0 处开始的情况**

   对于 switch 语句中 case 值不是从 0 开始的情况，如：

```
void switch_eg_2(long x) {
    switch(x) {
        case 10000:
        ......
        case 10002:
        ......
        case 10005:
        ......
        default:
        ......
    }
}
```

   汇编代码会进行相应调整：

```
switch_eg_2:
    leaq    -$10000(%rdi), %rsi     # %rsi = %rdi - 10000
    cmpq    $6, %rsi                # x:6
    ja      .L8                     # Use default
    jmp     *.L4(,%rsi,8)           # goto *JTab[x]
```

   通过 leaq -$10000(%rdi), %rsi 将 x 的值进行调整，使其可以适配从 0 开始索引的跳转表。

2. **稀疏的 switch 语句**

   当 switch 语句的 case 值分布非常稀疏时，如：

```
switch(x) {
    case 0:
            ...
    case 1000:
            ...
    case 92027:
            ...
}
```

   编译器可能会将其翻译为二分查找的语句（时间复杂度为 $O(\log n)$），而不是退化为逐个比较的 if - else - if - else ... 形式（时间复杂度为 $O(n)$），从而提高效率。

7.4.5 总结

switch 语句通过条件跳转、间接跳转（利用跳转表实现）等机制实现其功能。编译器会根据 switch 语句的具体情况，如 case 值的连续性、数量等，自动生成合适的代码序列，以实现

更复杂的控制逻辑。大规模的 switch 语句通常使用跳转表实现，而稀疏的 switch 语句则可以使用决策树（二分查找）来优化实现。

8 程序的机器级表示：过程

在计算机系统中，程序的机器级表示是理解程序如何在底层运行的关键。过程（函数调用）是程序设计中的重要概念，它涉及到控制流的转移、数据的传递以及栈的管理等多个方面。下面将详细介绍程序机器级表示中过程相关的知识。

8.1 栈的结构

8.1.1 x86 - 64 的栈概述

在 x86 - 64 架构的计算机系统中，栈是一种非常重要的数据结构，用于存储过程调用的上下文信息。栈的一个显著特点是它向低地址方向生长，这意味着当有新的数据入栈时，栈指针会向低地址移动。寄存器 %rsp 用于存储栈的最低地址，也就是栈指针，它始终指向栈顶的位置。

栈的基本特性总结如下：

栈的特性
栈底（Stack “Bottom”）位于高地址端，这是栈的起始位置。
栈顶（Stack “Top”）位于低地址端，随着数据的入栈和出栈，栈顶的位置会不断变化。
栈向下生长（Stack Grows Down），即向低地址方向扩展。当有新的数据需要入栈时，栈指针 %rsp 会减
栈指针（Stack Pointer）%rsp 始终指向栈顶，通过操作 %rsp 可以实现数据的入栈和出栈操作。

表 27: x86 - 64 栈的基本特性

8.1.2 x86 - 64 的栈：入栈操作

入栈操作 pushq src 是将一个 64 位的数据从源操作数 src 压入栈中。具体步骤如下：

1. 从 src 中取出操作数（Fetch operand at src）：首先，处理器会从源操作数 src 所在的位置读取要入栈的数据。这个源操作数可以是寄存器或者内存地址。
2. %rsp 减 8（Decrement %rsp by 8）：由于在 x86 - 64 架构中，每个数据单元是 64 位（8 字节），所以在入栈时，栈指针 %rsp 需要减去 8，以指向新的栈顶位置。
3. 将操作数的值写入 %rsp 指向的地址（Write operand at address given by %rsp）：最后，将从 src 中取出的数据写入到新的栈顶位置，也就是 %rsp 所指向的内存地址。

下面是一个简单的入栈操作示例代码：

```
pushq %rax ; 将寄存器 %rax 中的值入栈
```

在这个示例中，首先处理器会读取寄存器 %rax 中的值，然后将栈指针 %rsp 减 8，最后将 %rax 中的值写入到新的栈顶位置。

8.1.3 x86 - 64 的栈：出栈操作

出栈操作 `popq dest` 是将栈顶的 64 位数据弹出并写入到目标操作数 `dest` 中。具体步骤如下：

1. 从 `%rsp` 指向的地址中取出值（Fetch value in `%rsp`）：处理器会从当前栈指针 `%rsp` 所指向的内存地址读取栈顶的数据。
2. 将值写入 `dest`（Write value to operand `dest`）：将读取到的栈顶数据写入到目标操作数 `dest` 中，目标操作数可以是寄存器或者内存地址。
3. `%rsp` 加 8（Increment `%rsp` by 8）：由于已经将栈顶的数据弹出，栈指针需要加 8，以指向新的栈顶位置。

下面是一个简单的出栈操作示例代码：

```
popq %rbx ; 将栈顶的值弹出并写入寄存器 %rbx
```

在这个示例中，处理器会先读取栈顶的值，然后将该值写入到寄存器 `%rbx` 中，最后将栈指针 `%rsp` 加 8。

8.2 过程调用规范

8.2.1 控制流转移

过程调用是程序中实现代码复用和模块化的重要手段。在 x86 - 64 架构中，过程调用通过 `call label` 指令实现，而过程返回则使用 `ret` 指令。

`call label` 指令的执行过程如下：

1. 将返回地址压入栈：返回地址是紧接在 `call` 指令之后的指令所在的地址。当执行 `call` 指令时，处理器会将这个返回地址压入栈中，以便在过程执行完毕后能够返回到正确的位置继续执行。
2. 跳转至 `label` 标签处的过程代码开始位置：处理器会跳转到 `label` 所指定的地址处，开始执行被调用过程的代码。

`ret` 指令的执行过程如下：

1. 从栈中弹出返回地址：处理器会从栈顶弹出之前压入的返回地址。
2. 跳转至该地址：根据弹出的返回地址，处理器会跳转到相应的位置继续执行程序。

下面是一个完整的过程调用示例代码：

```
; 示例代码：控制流转移
void multstore(long x, long y, long *dest) {
    long t = mult2(x, y);
    *dest = t;
}
0000000000400540 <multstore>:
    400540: push %rbx         # Save %rbx
    400541: mov %rdx,%rbx     # Save dest
    400544: callq 400550 <mult2> # mult2(x,y)
    400549: mov %rax,(%rbx)   # Save at dest
    40054c: pop %rbx          # Restore %rbx
    40054d: retq              # Return

long mult2(long a, long b) {
    long s = a * b;
    return s;
}
0000000000400550 <mult2>:
    400550: mov %rdi,%rax     # a
    400553: imul %rsi,%rax    # a * b
    400557: retq              # Return
```

在这个示例中，multstore 函数调用了 mult2 函数。当执行到 callq 400550 <mult2> 指令时，会将下一条指令（400549）的地址压入栈中，然后跳转到 mult2 函数的起始地址（400550）开始执行。当 mult2 函数执行完毕，执行 retq 指令时，会从栈中弹出返回地址（400549），然后跳转到该地址继续执行 multstore 函数的后续代码。

8.2.2 数据传递

在 x86 - 64 架构中，数据传递遵循一定的规则。前 6 个参数通过寄存器传递，具体如下：

用于传递参数的寄存器
%rdi：用于传递第一个参数。
%rsi：用于传递第二个参数。
%rdx：用于传递第三个参数。
%rcx：用于传递第四个参数。
%r8：用于传递第五个参数。
%r9：用于传递第六个参数。

表 28: x86 - 64 中传递前 6 个参数的寄存器

返回值通过 %rax 寄存器传递。如果参数数量超过 6 个，剩余的参数则通过栈传递。并且，只有在需要时栈才会分配空间来存储这些额外的参数。

下面是一个参数传递的示例代码：

```
long add_five_numbers(long a, long b, long c, long d, long e) {
```

```
    return a + b + c + d + e;
}

int main() {
    long result = add_five_numbers(1, 2, 3, 4, 5);
    return 0;
}
```

对应的汇编代码中，参数 1 会被传递到 %rdi，参数 2 会被传递到 %rsi，参数 3 会被传递到 %rdx，参数 4 会被传递到 %rcx，参数 5 会被传递到 %r8。函数执行完毕后，返回值会存储在 %rax 中。

8.2.3 存储管理

在过程执行时，会为该过程分配栈帧空间。栈帧是栈中的一个区域，用于存储过程调用的相关信息，包括参数、局部变量、返回地址等。过程返回前，会释放栈帧空间，以便后续的过程调用使用。

栈帧的管理主要通过栈指针 %rsp 和可选的帧指针 %rbp 进行。栈指针 %rsp 始终指向栈顶，而帧指针 %rbp 可以指向栈帧的底部，方便访问栈帧中的数据。

栈帧的内容通常包括以下几个部分：

栈帧内容	
参数（Arguments）	即
本地变量（Local variables）	
保存的寄存器上下文信息（Saved register context）	保存过程中使用的寄存器的值，以便恢复
指向调用者的栈帧底部的指针（Old frame pointer，可选）	用于回溯调用栈，可用于访问调用者栈帧中
返回地址（Return address）	调用 call 指令时入栈，用

表 29: x86 - 64/Linux 栈帧内容

8.2.4 寄存器使用惯例

在 x86 - 64 架构中，寄存器分为“调用者保护”和“被调用者保护”两类，这是为了保证在过程调用过程中寄存器的值不会被意外修改。

“调用者保护”的寄存器，调用者在调用前把临时数据保存到自己的栈帧中。这意味着，如果一个过程（调用者）需要使用这些寄存器，并且希望在调用其他过程后这些寄存器的值仍然保持不变，那么它需要在调用之前将这些寄存器的值保存到自己的栈帧中，在调用返回后再恢复这些值。

“被调用者保护”的寄存器，被调用者在使用前将临时数据保存至自己的栈帧中，并在返回前恢复这些数据。也就是说，当一个过程（被调用者）需要使用这些寄存器时，它需要先将这些寄存器的值保存到自己的栈帧中，在使用完后再恢复这些值，以确保调用者看到的这些寄存器的值没有被改变。

具体的寄存器分类如下：

寄存器类别	寄存器列表
调用者保护（Caller - saved）	%rax, %rdi, %rsi, %rdx, %rcx, %r8, %r9, %r10, %r11
被调用者保护（Callee - saved）	%rbx, %r12, %r13, %r14, %rbp, %rsp

表 30: x86 - 64 Linux 中寄存器的保护类别

下面是一个寄存器使用惯例的示例代码：

```
void function1() {
    long temp = 10;
    function2();
    // 此时 temp 的值仍然是 10，因为 %rax 是调用者保护寄存器
}

void function2() {
    long result = 20;
    // 可以安全地使用 %rax 寄存器，因为调用者已经保存了 %rax 的值
    // ...
}
```

在这个示例中，function1 是调用者，function2 是被调用者。如果 function1 需要使用 %rax 寄存器，并且希望在调用 function2 后 %rax 的值不变，那么它需要在调用之前保存 %rax 的值。而 function2 可以安全地使用 %rax 寄存器，因为调用者已经处理了 %rax 的保存和恢复。

8.3 递归

递归是一种函数调用自身的编程技术，它在解决一些复杂问题时非常有用。下面以计算无符号长整型数中 1 的个数的递归函数 pcount_r 为例进行说明。

8.3.1 递归函数的代码实现

```
/* Recursive popcount */
long pcount_r(unsigned long x) {
    if (x == 0)
        return 0;
    else
        return (x & 1) + pcount_r(x >> 1);
}
```

这个函数的功能是计算无符号长整型数 x 中二进制表示下 1 的个数。如果 x 为 0，则返回 0；否则，返回 x 的最低位（x & 1）加上 x 右移一位后剩余部分中 1 的个数（通过递归调用 pcount_r(x >> 1) 计算）。

8.3.2 递归函数的汇编代码分析

对应的汇编代码如下：

```
pcount_r:
    movl $0, %eax
    testq %rdi, %rdi
    je .L6
    pushq %rbx
    movq %rdi, %rbx
    andl $1, %ebx
    shrq %rdi # (by 1)
    call pcount_r
    addq %rbx, %rax
    popq %rbx
.L6:
    rep; ret
```

下面对这段汇编代码进行详细分析：

1. `movl $0, %eax`：将寄存器 `%eax` 初始化为 0，用于存储最终的结果。
2. `testq %rdi, %rdi`：对寄存器 `%rdi` 进行按位与操作，用于判断 `%rdi` 是否为 0。如果为 0，则设置相应的标志位。
3. `je .L6`：如果 `%rdi` 为 0（即 `x` 为 0），则跳转到标签 `.L6` 处，直接返回 0。
4. `pushq %rbx`：将寄存器 `%rbx` 的值压入栈中，因为 `%rbx` 是被调用者保护寄存器，需要在使用前保存其值。
5. `movq %rdi, %rbx`：将参数 `x`（存储在 `%rdi` 中）复制到寄存器 `%rbx` 中。
6. `andl $1, %ebx`：对 `%ebx` 的最低位进行按位与操作，得到 `x` 的最低位。
7. `shrq %rdi`：将 `%rdi` 中的值右移一位，相当于 `x >> 1`。
8. `call pcount_r`：递归调用 `pcount_r` 函数，计算 `x >> 1` 中 1 的个数。
9. `addq %rbx, %rax`：将 `x` 的最低位（存储在 `%rbx` 中）加到结果寄存器 `%rax` 中。
10. `popq %rbx`：从栈中弹出之前保存的 `%rbx` 的值，恢复 `%rbx` 的原始值。
11. `rep; ret`：返回调用者，将结果存储在 `%rax` 中。

递归函数的处理和普通函数调用类似，栈帧为每次函数调用提供私有的存储空间，用于保存寄存器和局部变量、返回地址等。寄存器使用惯例保证了一次函数调用不会破坏其他函数的数据，栈的后进先出特性也适用于递归和相互递归的情况。