CSAPP 笔记

apl

2025 年 4 月 2 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

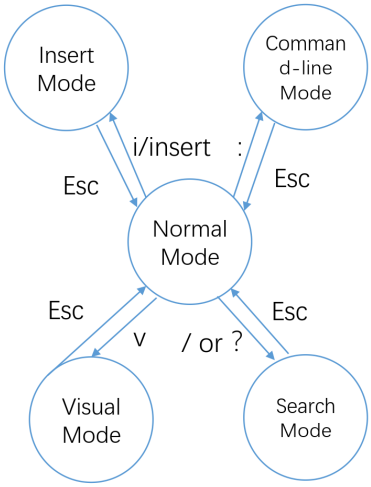

图 1 vim 的工作模式

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 Git 的基本概念

包括仓库（项目的版本控制库，可在本地或远程服务器）、提交（对代码库的保存，包含代码修改等）、暂存区（文件提交前的临时区域）、还原（用仓库记录还原工作目录文件更改）、推送（将本地仓库更改上传到远程仓库）、拉取（从远程仓库获取最新更改并应用到本地仓库）。

1.4.2 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。

1.4.3 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。

1.4.4 远程仓库操作

以 GITEE 为例，可创建空的远程仓库，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。

1.4.5 多人协作开发

多人协作开发时，开发者克隆远程仓库至本地，各自进行代码修改后提交。推送前需先拉取远程仓库最新代码，若有冲突需解决冲突后再推送。冲突多数可由 git 自动解决，无法自动解决时需人工干预编辑冲突区域代码。

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：

```c
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 2 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。

C 语言数据类型	典型 32 位系统	典型 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。小端序如 Intel，低地址存放低位数据，高地址存放高位数据；大端序如 IBM、Sun Microsystem（Oracle），低地址存放高位数据，高地址存放低位数据。例如，对于 0x1234567 ：

地址				
	低地址			高地址
大端序	$0x100:0x12$	$0x101:0x34$	$0x102:0x56$	$0x103:0x07$
小端序	$0x100:0x07$	$0x101:0x56$	$0x102:0x34$	$0x103:0x12$

表 6: 字节序示例

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：

```c
#include <stdio.h>
typedef unsigned char *byte_pointer;
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）系统下的测试结果如下：

机器	值	类型	字节（十六进制）
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 7: 不同系统下数据存储测试结果

2.3.5 指针的存储方法

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。例如，在 x86 - 64、Sun、IA32 环境下：

```
int B = -15213;
int *P = &B;
```

2.3.6 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。

2.3.7 程序的表示

不同类型的机器使用不同的且不兼容的指令和指令编码。在相同处理器不同的操作系统中，由于编码规范存在差异，同样代码所生成的程序也不是二进制兼容的，程序很少能够在不同类型机器和不同操作系统中实现二进制水平上移植。

2.3.8 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 8: 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 等价性与唯一性

有符号数和无符号数的非负值编码相同。每个编码都表示唯一的整数值，每个可表示的整数也有唯一的位编码，因此可以反向映射，即 $U2B(x) = B2U^{-1}(x)$，$T2B(x) = B2T^{-1}(x)$。

3.1.4 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）；否则，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。

3.1.5 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（补码）范围：$T_{min} = -2^{w-1}$，$T_{max} = 2^{w-1} - 1$

3.1.6 一些重要的数字

不同字长下的重要数字如下：

Value	Word size $w = 8$	Word size $w = 16$	Word size $w = 32$
UMax	0xFF (255)	0xFFFF (65535)	0xFFFFFFFF (4294967295)
TMin	0x80 (-128)	0x8000 (-32768)	0x80000000 (-2147483648)
TMax	0x7F (127)	0x7FFF (32767)	0x7FFFFFFF (2147483647)
-1	0xFF	0xFFFF	0xFFFFFFFF
0	0x00	0x0000	0x00000000

表 9: 不同字长下的重要数字

3.1.7 可表示整数范围的关系

$|T_{Min}| = T_{Max} + 1$，$U_{max} = 2 \times T_{Max} + 1$。有符号数 -1 的补码和 U_{max} 的编码相同，所有位都为 1；0 的编码方式在两种表示中都相同，所有位都为 0。

3.1.8 反码和原码

反码和补码的定义类似，区别是符号位的权重为 $-(2^{w-1} - 1)$。原码和补码的区别是，符号位的作用仅用于决定其他位的位权为正/负。

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。

3.2.4 C 语言中的有符号数和无符号数之间的转换

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
```

3.2.5 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 10: 表达式求值示例（以 32 位字长为例）

3.2.6 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 11: 位扩展示例

3.2.7 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。可以通过判断 $s < x$ 或 $s < y$ 来检测溢出（$s = x + y$）。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 12: 无符号数加法示例（$w = 4$）

3.3.2 有符号数加法（补码加法）

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。正溢出时 $x+y \geq 2^{w-1}$，结果为 $x + y - 2^w$；正常时 $-2^{w-1} \leq x + y < 2^{w-1}$，结果为 $x + y$；负溢出时 $x + y < -2^{w-1}$，结果为 $x + y + 2^w$。可以通过以下规则检测溢出：- 正溢出：$x > 0, y > 0, s \leq 0$ - 负溢出：$x < 0, y < 0, s \geq 0$ 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 13: 有符号数加法示例（$w = 4$）

3.3.3 无符号数求反

对于无符号数 x，其反数为 $2^w - x$（$x > 0$），0 的反数为 0。例如在 $w = 4$ 时，$x = 5$，反数为 $16 - 5 = 11$。

3.3.4 有符号数求反（补码求反）

对于有符号数 x，当 $x > T_{min}$ 时，反数为 $-x$；当 $x = T_{min}$ 时，反数为 T_{min} 本身。例如在 $w = 4$ 时，$T_{min} = -8$，$x = -3$ 反数为 3，$x = -8$ 反数为 -8。

3.3.5 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 $30 \bmod 16 = 14$。

3.3.6 有符号数乘法（补码乘法）

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。

3.3.7 移位运算与乘法除法的关系

左移 k 位相当于乘以 2^k。对于无符号数右移 k 位相当于除以 2^k，采用的是向下取整；对于有符号数右移 k 位，在大多数编译器下（算术右移）相当于除以 2^k 并向零取整。例如：

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

3.3.8 除以 2 的幂的优化

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。例如：

```
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：$5\frac{3}{4} = 101.11_2$ $2\frac{7}{8} = 010.111_2$

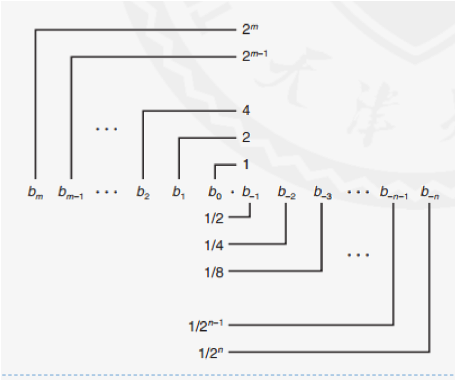

图 3 小数的二进制表示

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 4 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 14: 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，$\text{bias} = 127$（$\text{Exp} : 1\cdots254$，$E : -126\cdots127$）；双精度浮点数，$\text{bias} = 1023$（$\text{Exp} : 1\cdots2046$，$E : -1022\cdots1023$）。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：

$$
\begin{aligned}
f = 15213_{10} &= 11101101101010000000000_2 \\
&= 1.1101101101101101_2 \times 2^{13} \\
\text{尾数} \quad M &= 1.\mathbf{1101101101101101}_2 \\
\text{frac} &= \mathbf{1101101101101}0000000000_2 \\
\text{阶码} \quad E &= 13, \\
\text{Bias} &= 2^{k-1} - 1 = 127 \quad (k = 8), \\
\text{Exp} &= 140 = 10001100_2
\end{aligned}
$$

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 15: 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 16: 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 ≠ 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 17: 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 18: 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

5.1.3 Intel 64 位体系结构发展的历史

2001 年，Intel 试图从 IA32 转变为 IA64（安腾），但性能令人失望。2003 年，AMD 提出 x86 - 64 位体系结构（现称为 AMD64）。2004 年，Intel 提出 EM64T 体系结构，实现对 IA32 的 64 位扩展，几乎与 x86 - 64 相同。2019 年，英特尔宣布放弃 IA64 架构，目前除低端 x86 处理器外，其他处理器均支持 x86 - 64，但仍有许多程序在 32 位模式下运行。

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）是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. 微体系结构是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

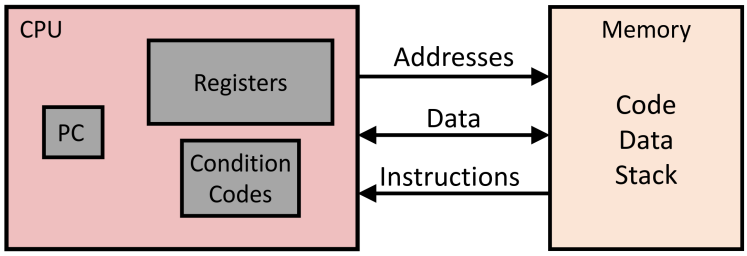

图 5

5.2.3 将 C 语言代码转换为机器语言

将 C 语言代码转换为机器语言的过程包括：C 程序（如 p1.c、p2.c）通过编译器（如使用 `gcc -S`）生成汇编程序（p1.s、p2.s），再由汇编器（使用 'gcc' 或 'as'）生成目标程序（p1.o、p2.o），最后通过链接器（使用 'gcc' 或 'ld'）生成可执行程序（p）。例如，使用 `gcc -Og p1.c p2.c -o p`，其中 '-Og' 是基本的编译优化选项（最新版本 GCC 支持）。

5.2.4 将 C 语言代码编译为汇编代码

以 'sum.c' 文件中的代码为例：

```
long plus(long x, long y);
void sumstore(long x, long y, long *dest)
{
    long t = plus(x, y);
    *dest = t;
}
```

使用 `gcc -Og -S sum.c` 可生成 'sum.s' 文件，得到的 x86 - 64 汇编代码如下：

```
sumstore:
    pushq %rbx
    movq %rdx, %rbx
    call plus
    movq %rax, (%rbx)
    popq %rbx
    ret
```

需要注意，由于编译选项和 gcc 版本不同，可能会得到不同的编译结果。

5.2.5 汇编语言的特征: 数据类型

汇编语言中的数据类型包括 1、2、4 或 8 字节的整数、地址（无类型的指针）、4、8 或 10 字节的浮点数以及指令的字节序列编码（代码）。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。

5.2.6 汇编语言的特征: 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。

5.2.7 链接器

链接器将 ‘.s‘ 文件翻译为 ‘.o‘ 文件，对每条指令进行二进制编码，生成几乎完整的可执行代码，但缺少不同文件的链接信息。链接器实现了不同文件间的引用，并与静态链接库结合（例如代码中的 ‘malloc‘、‘printf‘ 等），某些库需要动态链接，链接在程序开始执行时进行。

5.2.8 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)`，在 x86 - 64 中，这是一条将 8 字节数据（四字）移动至存储器的指令。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]`。该指令为 3 字节，存储于地址 0x40059e。

5.2.9 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 `disassemble sumstore` 反汇编 ‘sumstore‘ 函数。任何可解释为可执行代码的文件都可以被反汇编，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。其中`\%rsp`有特殊用途，一般不在汇编里使用。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 19: 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 20: 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]

。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

操作数类型包含：

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 C Analog
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 21: 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 22: 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(R_{b}, R_{i}, S)` 表示 `Mem[Reg[R_{b}]+S * Reg[R_{i}]+D]`，其中：

1. D：常数偏移量，可以为 1、2、4 或 8 字节整数。
2. R_{b}：基地址寄存器，是 16 个寄存器之一。
3. R_{i}：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 (R_{b}, R_{i}) 表示 Mem[Reg[R_{b}] + Reg[R_{i}]] ，D(R_{b}, R_{i}) 表示 Mem[Reg[R_{b}]+Reg[R_{i}]+D] ，(R_{b}, R_{i}, S) 表示 Mem[Reg[R_{b}]+S * Reg[R_{i}]] 。

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 23: 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 24: 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 25: 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：

```
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;
}
```

其汇编代码为：

```
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
```

此函数的汇编代码通过一系列指令实现了函数中各变量的计算。首先将 `x` 和 `y` 相加得到 `t1`，再把 `t1` 与 `z` 相加得到 `t2`。对于 `t3`，使用 `leaq` 指令计算 `x + 4`。计算 `t4` 时，先通过 `leaq` 计算 `y + 2*y` 得到 `t4a`，再左移 4 位得到 `t4`。然后将 `t3` 和 `t4` 相加得到 `t5`，最后将 `t2` 和 `t5` 相乘得到结果 `rval` 并返回。

6.4 控制转移指令

6.4.1 条件码寄存器

CPU 中有一组单个位的条件码寄存器，常用的有：

- `CF`（进位标志）：无符号运算产生进位时设置。
- `ZF`（零标志）：结果为零时设置。
- `SF`（符号标志）：结果为负时设置。
- `OF`（溢出标志）：有符号运算溢出时设置。

很多算术和逻辑指令都会设置这些条件码，例如 `addq`、`subq` 等。

6.4.2 比较和测试指令

比较指令 `cmpq` 和测试指令 `testq` 主要用于设置条件码，而不改变任何寄存器或内存的值。`cmpq Src2, Src1` 相当于计算 `Src1 - Src2` 并设置条件码，例如：

```
cmpq %rdi, %rsi  # 设置条件码，基于 %rsi - %rdi
```

`testq Src2, Src1` 相当于计算 `Src1 & Src2` 并设置条件码，例如：

```
testq %rdi, %rdi  # 判断 %rdi 是否为零
```

6.4.3 条件跳转指令

条件跳转指令根据条件码的值来决定是否跳转。例如：

- `je`（等于跳转）：当 `ZF = 1` 时跳转。

- jne（不等于跳转）：当 ZF = 0 时跳转。
- jg（大于跳转）：有符号比较中，当 SF = OF 且 ZF = 0 时跳转。
- jge（大于等于跳转）：有符号比较中，当 SF = OF 时跳转。
- jl（小于跳转）：有符号比较中，当 SF != OF 时跳转。
- jle（小于等于跳转）：有符号比较中，当 SF != OF 或 ZF = 1 时跳转。
- ja（高于跳转）：无符号比较中，当 CF = 0 且 ZF = 0 时跳转。
- jae（高于等于跳转）：无符号比较中，当 CF = 0 时跳转。
- jb（低于跳转）：无符号比较中，当 CF = 1 时跳转。
- jbe（低于等于跳转）：无符号比较中，当 CF = 1 或 ZF = 1 时跳转。

示例代码：

```
cmpq %rdi, %rsi
jg greater  # 如果 %rsi > %rdi 则跳转到 greater 标签处
# 不满足条件时执行的代码
jmp end
greater:
# 满足条件时执行的代码
end:
```

6.4.4 无条件跳转指令

无条件跳转指令 jmp 会直接跳转到指定的位置。可以是直接跳转（给出标签），也可以是间接跳转（通过寄存器或内存地址）。直接跳转示例：

```
jmp target  # 跳转到 target 标签处
target:
# 跳转后的代码
```

间接跳转示例：

```
jmp *%rax  # 跳转到 %rax 所指向的地址处
```

6.4.5 循环和分支结构的实现

在汇编中，循环和分支结构通常通过条件码和跳转指令来实现。

while 循环示例 C 语言的 while 循环：

```c
long fact_while(long n) {
    long result = 1;
    while (n > 1) {
        result *= n;
        n--;
    }
    return result;
}
```

对应的汇编代码：

```asm
fact_while:
movl $1, %eax  # result = 1
cmpq $1, %rdi  # 比较 n 和 1
jle done       # 如果 n <= 1 则跳转到 done
loop:
imulq %rdi, %rax  # result *= n
subq $1, %rdi     # n--
cmpq $1, %rdi     # 比较 n 和 1
jg loop          # 如果 n > 1 则跳转到 loop
done:
movq %rax, %rdi  # 返回 result
ret
```

if - else 分支示例 C 语言的 if - else 语句：

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

对应的汇编代码：

```asm
absdiff:
cmpq %rsi, %rdi  # 比较 x 和 y
jl less          # 如果 x < y 则跳转到 less
subq %rsi, %rdi  # x - y
movq %rdi, %rax  # 返回结果
ret
less:
subq %rdi, %rsi  # y - x
movq %rsi, %rax  # 返回结果
ret
```

6.5 过程调用和栈管理

6.5.1 栈的基本概念

栈是一种后进先出（LIFO）的数据结构，在 x86 - 64 中，栈从高地址向低地址增长，\%rsp 寄存器指向栈顶。栈主要用于存储局部变量、保存返回地址和寄存器状态等。

6.5.2 过程调用指令

call 指令用于调用一个过程，它会将下一条指令的地址（返回地址）压入栈中，并跳转到被调用过程的起始地址。例如：

```
call proc  # 调用 proc 过程
# 调用返回后执行的代码
```

ret 指令用于从过程中返回，它会从栈中弹出返回地址，并跳转到该地址继续执行。

6.5.3 栈帧结构

每个过程在栈上都有一个栈帧（stack frame），栈帧包含局部变量、保存的寄存器值等。栈帧的底部由 \%rbp 寄存器指向（基指针），栈帧的顶部由 \%rsp 寄存器指向（栈指针）。

进入一个过程时，通常会保存当前的 \%rbp 值，并将 \%rsp 的值赋给 \%rbp，以建立新的栈帧。例如：

```
pushq %rbp     # 保存旧的 %rbp
movq %rsp, %rbp  # 建立新的栈帧
# 过程体代码
popq %rbp      # 恢复旧的 %rbp
ret            # 返回
```

6.5.4 参数传递

在 x86 - 64 中，前六个整数或指针参数依次通过 \%rdi、\%rsi、\%rdx、\%rcx、\%r8、\%r9 寄存器传递，更多的参数则通过栈传递。返回值通常通过 \%rax 寄存器返回。

例如，一个接收两个参数的函数：

```
long add(long a, long b) {
    return a + b;
}
```

调用该函数的汇编代码示例：

```
movq $3, %rdi  # 参数 a = 3
movq $5, %rsi  # 参数 b = 5
call add       # 调用 add 函数
# 返回值在 %rax 中
```