南京邮电大学

实 验 报 告

（2025/ 2026 学年 第 二 学期）

课程名称	计算机系统基础I
实验名称	数据表示实验
实验时间	2026 年 5 月 21 日
指导单位	基础教学中心
指导教师	

学生姓名		班级学号	
学院(系)	计算机学院	专　业	软件工程

实 验 报 告

实验名称	数据表示实验			指导教师	
实验类型	设计	实验学时	2	实验时间	2026 5 21

一. 实验目的和要求

实验目的：

学习掌握整数和字符的表示；C语言整数、字符等变量在寄存器和存储器中的存放；掌握按位运算和逻辑运算、移位运算、位扩展和位截断运算；掌握整数加减、乘除运算；掌握浮点数IEEE754表示的表示以及浮点数的加减乘除运算。

实验要求

下载实验压缩包，bits.c文件包含15个编程谜题中每个谜题的框架，用第2章中学到的知识来完成这些程序中缺失的部分，具体实验流程请参考实验手册。编程要求如下：

- 除浮点数函数实现外，只能使用顺序程序结构，禁用if, do, while, for, switch 等。
- 有限操作类型，！ ～ & ^ | + << >> 各函数不一样
- 禁用（！=、==、&&、|| 等组合操作符）
- 常量值范围 0~255
- 禁用强制类型转换
- 禁用整型外的任何其它数据类型
- 禁用定义和宏
- 不得使用函数
- 具体要求可参看bits.c各函数框架的注释

二、实验环境(实验设备)

硬件：微型计算机

软件：linux 操作系统、Gcc编译套件

三、实验代码及结果

中文五号宋体，英文五号Times new roman字体，1.25倍行距

说明：这部分内容主要包括：

1、给出源代码，并加上必要注释；

2、给出程序测试结果以及测试分数。

代码：

```
#include<stdio.h>
typedef unsigned char *byte_pointer;
void show_bytes (byte_pointer start, size_t len)
{
size_t i;
for (i =0;i< len; i++) printf(" 0x%.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 *));
}
void show_double (double x) {
show_bytes((byte_pointer) &x, sizeof (double));
}
void show_char (char x) {
show_bytes((byte_pointer) &x, sizeof (char));
}
void show_short (short x) {
show_bytes((byte_pointer) &x, sizeof (short));
2 }
void show_long (long x) {
show_bytes((byte_pointer) &x, sizeof (long));
}
void test(int x)
{
int y=x;
float f=(float) x;
int *u= &x;
double t=(double) x;
char c=(char) x;
short s=(short) x;
long l=(long) x;
show_int(y);
show_float(f);
show_pointer(u);
show_double(t);
show_char(c);
show_short(s);
show_long(l);
}
int main()
{
```

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

中文五号宋体，英文五号Times new roman字体，1.25倍行距

说明：这部分内容主要包括：在编程、调试或测试过程中遇到的问题及解决方法、本次实验的心得体会、进一步改进的设想等。

(一)实验中遇到的主要问题及解决方法

1. bitMask 函数的移位边界问题

在实现 bitMask 时，第一版写法是 (~0) << (highbit + 1)，当 highbit == 31 时会触发 << 32，属于 C 标准的未定义行为，btest 在边界输入上给出了错误结果。解决方法是将单次移位拆成两次：先 << highbit，再 << 1，每次移位都不超过 31 位，行为完全可预测。这让我意识到位运算并不是纯数学，底层的硬件指令（SHL/SAR）有自己的约束边界。

2. isAsciiDigit 的 TMin 边界 bug

最初用 (x + ~0x2F) >> 31 检查 x >= 0x30 的符号位，btest 在 x = 0x80000000（TMin）时报错。原因是 TMin 加 ~0x2F 在 32 位补码里环绕到正数区，符号位翻成 0，误判为合法 ASCII 数字。改用「高半字节是否等于 3」+ 「低半字节是否小于 10」的拆字节方案后，避开了所有算术溢出，btest 通过所有测试用例。

3. floatFloat2Int 遗漏 NaN 和 ±∞ 的特殊处理

最初没有单独判断 exp == 255 的情况，btest 对 NaN 输入报错。回头看 IEEE 754 才注意到阶码全 1（exp == 0xFF）同时代表 ±∞ 和 NaN，真实阶 E = 128 >= 31，刚好被「超出 int 范围」这一条统一捕获，不需要单独写分支。这个规律是 IEEE 754 标准设计层面的内在一致性，理解它之后代码反而变简单了。

（二）实验心得

这次 DataLab 是第一次在比特级别上直接操作数据，整体下来有几个真实的收获。

最深的一个是理解了补码「取反加一」不只是口诀，而是由「加法器需要同时处理正负数」这一工程约束推导出来的必然结果。negate、absVal 的正确写法，以及 absVal 的(x + mask) ^ mask 这个不带分支的绝对值公式，都从这一性质自然推出来。

浮点数部分理解了 IEEE 754 的「字段分工」之后，floatScale2 和 floatFloat2Int 的逻辑几乎是逐字段翻译成代码。之前觉得浮点数很神秘，这次拆开看，符号、阶码、尾数三个字段各自管各自的事，思路清晰了很多。顺带也明白了 JavaScript 里 0.1 + 0.2 !== 0.3 不是 bug，而是二进制下 0.1 本身就是无限循环小数，IEEE 754 在有限位数里做了舍入。

另一个收获是「用位掩码代替分支」的思维方式。题目禁止 if，逼着我去想怎么用算术表达式承载控制流：x >> 31 得到全 0 或全 1 的掩码，再用 & 和 ^ 做条件选择。这种技巧在性能敏感的代码里很常用，现在至少能看懂编译器做分支消除时在干什么。

（三）意见与建议（没有可省略）

无

五.支撑毕业要求指标点
六、指导教师评语 (含学生能力达成度的评价)

成 绩		批阅人		日 期	

评分细则	评分项	优秀	良好	中等	合格	不合格
	遵守实验室规章制度					
	学习态度					
	算法思想准备情况					
	程序设计能力					
	解决问题能力					
	课题功能实现情况					
	算法设计合理性					
	算法效能评价					
	回答问题准确度					
	报告书写认真程度					
	内容详实程度					
	文字表达熟练程度					

	其它评价意见					
	本次实验能力达成评价（总成绩）					