2023—2024 第二学期 csappl 期末试卷（A）答案

单项选择题

1．B

2．C

数	字长w			
	8	16	32	64
$UMax_w$	0xFF 255	0xFFFF 65 535	0xFFFFFFFF 4 294 967 295	0xFFFFFFFFFFFFFFFF 18 446 744 073 709 551 615
$TMin_w$	0x80 −128	0x8000 −32 768	0x80000000 −2 147 483 648	0x8000000000000000 −9 223 372 036 854 775 808
$TMax_w$	0x7F 127	0x7FFF 32 767	0x7FFFFFFF 2 147 483 647	0x7FFFFFFFFFFFFFFF 9 223 372 036 854 775 807
−1 0	0xFF 0x00	0xFFFF 0x0000	0xFFFFFFFF 0x00000000	0xFFFFFFFFFFFFFFFF 0x0000000000000000

图 2-14　重要的数字。图中给出了数值和十六进制表示

结论1：补码所对应的真值范围是不对称的，负数个数比正数多一个。

结论2：最大的无符号数值刚好比补码的最大值的两倍大一个。

注意：-1 和UMax 有同样的位表示——一个全1 的串
数值0在两种表示方式中都是全0 的串

3．D　32768 为 2 的 15 次方，补码表示为 100..0，作为有符号短整型解释时，其值为 D

考虑这样的问题（分别写出有符号+无符号数）：

计算机中8位二进制数11111111对应的值是多少？

-1　　255

计算机中8位二进制数10000000对应的值是多少？

-128　　128

计算机中8位二进制数01000001对应的值是多少？

65　　65

C语言标准规定：若运算中同时有无符号和带符号整数，则按无符号整数运算

关系 表达式	类型	结果	说明
0 = = 0U	无	1	00…0B = 00…0B
-1 < 0	带	1	11…1B (-1) < 00…0B (0)
-1 < 0U	无	0*	11…1B (2^{32}-1) > 00…0B(0)
2147483647 > -2147483647 - 1	带	1	011…1B (2^{31}-1) > 100…0B (-2^{31})
2147483647U > -2147483647 - 1	无	0*	011…1B (2^{31}-1) < 100…0B(2^{31})
2147483647 > (int) 2147483648U	带	1*	011…1B (2^{31}-1) > 100…0B (-2^{31})
-1 > -2	带	1	11…1B (-1) > 11…10B (-2)
(unsigned) -1 > -2	无	1	11…1B (2^{32}-1) > 11…10B (2^{32}-2)

4．A

1. **核心概念：偏向偏差**
   指舍入操作在大量重复后，对数值产生**系统性高估或低估**（如始终“舍”或始终“入”，导致结果长期偏离真实值）。

2. **各选项舍入逻辑与偏向性**

   - **A. 向偶数舍入（银行家舍入）：**
     规则是“若舍入位为中间值（如二进制 100...0 、十进制 5 ），则看保留位的最后一位：若为偶数则舍去，若为奇数则进1”。
     例：舍入到整数位， 2.5 → 舍入为 2 （保留位最后一位是偶数）； 3.5 → 舍入为 4 （保留位最后一位是奇数）。
     这种“遇中间值时平衡进/舍”的逻辑，使**长期统计中进、舍概率趋于均等**，从根本上避免系统性偏向。

   - **B. 向零舍入：**
     规则是“直接截断超出保留位的部分”（正数向下舍、负数向上舍）。
     例： 2.8 → 2 ， -2.8 → -2 。
     对正数长期低估、对负数长期高估，存在系统性偏向。

   - **C. 向下舍入：**
     规则是“始终向更小的数舍入”（正数截断、负数更负）。
     例： 2.8 → 2 ， -2.8 → -3 。
     对所有数长期低估，偏向性极强。

   - **D. 向上舍入：**
     规则是“始终向更大的数舍入”（正数进1、负数截断）。
     例： 2.1 → 3 ， -2.1 → -2 。
     对所有数长期高估，偏向性极强。

3. **结论**
   只有**向偶数舍入**通过“中间值时平衡进/舍”的机制，消除了系统性偏向；其他舍入方式（向零、向下、向上）均存在固定方向的偏向。

5． A

提供一个可以参考的例子

-12.75

1. Denormalize: -12.75

2. Convert integer part:

12 = 8 + 4 = 1100_2

3. Convert fractional part:

.75 = .5 + .25 = $.11_2$

4. Put parts together and normalize:

1100.11 = 1.10011 x 2^3

5. Convert exponent: 127 + 3 = 128 + 2 = $1000\ 0010_2$

1	1000 0010	100 1100 0000 0000 0000 0000

The Hex rep. is C14C0000H

6． C

可重定位目标文件格式

计算机系统基础I

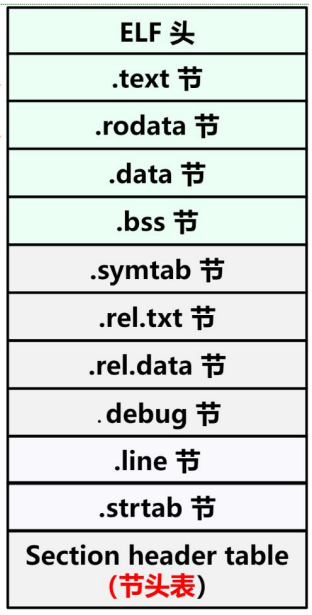
ELF 头
- ✓ 定义了ELF魔数、版本、小端/大端、操作系统平台、目标文件的类型、机器结构类型、节头表的起始位置和长度等

.text 节
- ✓ 编译后的代码部分

.rodata 节
- ✓ 只读数据，如 printf 格式串、switch 跳转表等

.data 节
- ✓ 已初始化的全局变量

.bss 节
- ✓ 未初始化全局变量，仅是占位符，不占据任何实际磁盘空间。区分初始化和非初始化是为了空间效率

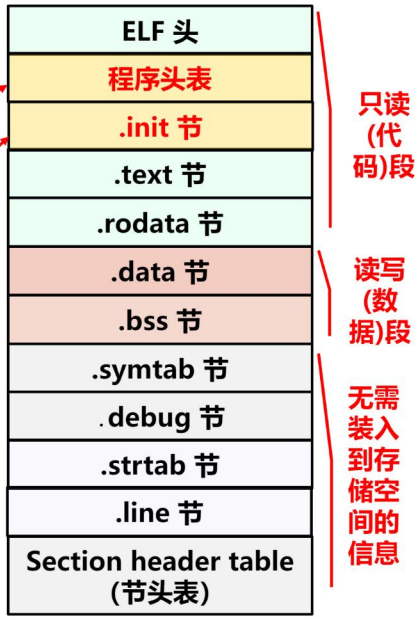


可执行目标文件格式

计算机系统基础I

与可重定位文件稍有不同：

- ELF头中字段e_entry给出执行程序时第一条指令的地址，而在可重定位文件中，此字段为0
- 多一个程序头表，也称段头表（segment header table），是一个结构数组
- 多一个.init节，用于定义_init函数，该函数用来进行可执行目标文件开始执行时的初始化工作
- 少两个.rel节（无需重定位）



7. C

1. 调用约定核心规则

x86-64 中，函数参数按**类型分组**传递到不同寄存器：

- **整数/指针类型**（如 `long`、`long*`）：前 6 个参数依次存入 `rdi, rsi, rdx, rcx, r8, r9`；
- **浮点类型**（如 `float`、`double`）：前 6 个参数依次存入 `xmm0, xmm1, xmm2, xmm3, xmm4, xmm5`；

2. 逐个分析实参的寄存器分配

函数调用为 `Test(a, &a, b, &b)`，共 4 个实参，需按"类型+顺序"匹配寄存器：

- **实参** `a`：类型 `long`（整数）→ 第 1 个整数参数 → 存入 `rdi`；
- **实参** `&a`：类型 `long*`（指针，属于整数类）→ 第 2 个整数参数 → 存入 `rsi`；
- **实参** `b`：类型 `float`（浮点）→ 第 1 个浮点参数 → 存入 `xmm0`；
- **实参** `&b`：类型 `float*`（指针，属于整数类）→ 第 3 个整数参数 → 存入 `rdx`（前两个整数参数已用 `rdi`、`rsi`，第三个整数参数对应 `rdx`）；

8. D

解析：要解决静态链接的符号解析问题，需理解**静态链接器的处理规则**：

1. **核心规则**：
   静态链接器按**命令行参数顺序**处理目标文件（ .o ）和库（ .a ）；对库（ .a ）仅在"当前存在未解析符号"时，才从中提取对应目标文件；且**库只会被扫描一次**（除非重复指定）。
2. 分析依赖链 p.o → libx.a → liby.a → libx.a → p.o ：
   - p.o 依赖 libx.a 的符号；
   - libx.a 依赖 liby.a 的符号；
   - liby.a 依赖 libx.a 的符号（形成**库间循环依赖**）。
3. 逐个排除选项：
   - **选项A**： gcc p.o libx.a liby.a
     处理顺序为 p.o → libx.a → liby.a 。 liby.a 依赖 libx.a 的符号，但 libx.a 已被处理过，链接器不会"回头"扫描 libx.a ，导致 liby.a 的依赖符号无法解析。
   - **选项B**： gcc p.o libx.a liby.a p.o
     最后重复 p.o 无意义（ p.o 是目标文件，已处理过，无新符号）；且 liby.a 依赖 libx.a 的符号仍因 libx.a 仅处理一次而无法解析。
   - **选项C**： gcc p.o libx.a liby.a libx.a p.o
     重复 p.o 多余（目标文件无需重复）；虽重复 libx.a 能解决循环依赖，但参数数量多于选项D，不满足"最小命令行"。
   - **选项D**： gcc p.o libx.a liby.a libx.a
     处理顺序为 p.o → libx.a → liby.a → libx.a ：
     - p.o 未解析符号由第一个 libx.a 解决；
     - libx.a 未解析符号由 liby.a 解决；
     - liby.a 未解析符号（依赖 libx.a ）由**第二次扫描 libx.a** 解决； 所有循环依赖的符号均被解析，且参数数量最少（仅4个参数）。

知识拓展：静态链接的"循环依赖"处理

当库间存在循环依赖（如 A→B→A ），需通过**重复指定库**让链接器多次扫描。原理是：第一次扫描库A时，解决前期未解析符号；扫描库B时，库B依赖A的符号会触发"再次扫描A"（因A在命令行中后续重复出现），从而打破循环。这种设计是静态链接"单遍扫描+按需提取"机制的妥协，保证了灵活性与性能平衡。

- 假设调用关系如下：
  func.o → libx.a 和 liby.a 中的函数
  libx.a → libz.a 中的函数
  libx.a 和 liby.a 之间、liby.a 和 libz.a 相互独立
  则以下几个命令行都是可行的：
  – gcc -static –o myfunc func.o libx.a liby.a libz.a
  – gcc -static –o myfunc func.o liby.a libx.a libz.a
  – gcc -static –o myfunc func.o libx.a libz.a liby.a
- 假设调用关系如下：
  func.o → libx.a 和 liby.a 中的函数
  libx.a → liby.a 同时 liby.a → libx.a
  则以下命令行可行：
  – gcc -static –o myfunc func.o libx.a liby.a libx.a
  – gcc -static –o myfunc func.o liby.a libx.a liby.a

9. D

步骤1：理解 cmpq %rax, %rdx 的作用

`cmpq` 指令通过计算 `rdx - rax` 来设置标志位（不修改操作数），用于后续条件跳转判断。本次执行中：

- `rdx = 80`，`rax = 68` → `rdx - rax = 12`（无符号运算，结果非零且无借位）。

步骤2：分析 jbe 的跳转条件

`jbe`（Jump if Below or Equal）是**无符号数的"低于或等于"**跳转，触发条件为 `CF=1`（**借位标志，表示 `rdx < rax`**）或 `ZF=1`（**零标志，表示 `rdx = rax`**）。

结合 `cmpq` 结果：

- `rdx - rax = 12 ≠ 0` → `ZF=0`；
- `rdx > rax`（无借位）→ `CF=0`；因此 `CF∨ZF = 0`，**`jbe` 的跳转条件不满足**，指令不会跳转。

步骤3：确定"不跳转"时的执行地址

`jbe` 是**2字节指令**（机器码 `76 f8` 占2字节），其所在地址为 `804857e`。
"不跳转"时，CPU会执行**下一条指令**，即 `jbe` 指令的下一条指令地址 = `jbe地址 + 指令长度` = `804857e + 2 = 8048580`。

知识拓展：相对跳转的寻址逻辑

若 `jbe` 条件满足，跳转目标由**"下一条指令地址 + 偏移量"**计算：

- 偏移量由机器码的第二个字节（`f8`）决定，`f8` 是补码表示的 `-8`；
- 下一条指令地址为 `804857e + 2 = 8048580`；
- 跳转目标 = `8048580 + (-8) = 8048578`（对应选项C，但本题条件不满足，故不跳转）。

综上，`jbe` 条件不满足时执行下一条指令，地址为 `8048580`，答案选 **D**。

10. C 更正题目为 0x8000fff2 单元

1. 小端字节序规则

小端法（Little-Endian）的核心是：**低位字节存储在低地址，高位字节存储在高地址。**

2. int型变量的内存分布

`int` 类型在多数系统中占 **4字节（32位）**，变量 `x` 的地址为 `0x8000fff0`，因此其4个字节的存储地址依次为：

- 第0字节（最低位字节）：`0x8000fff0`
- 第1字节：`0x8000fff1`
- 第2字节：`0x8000fff2`
- 第3字节（最高位字节）：`0x8000fff3`

3. 拆分 `x=0x80901020` 的字节

十六进制数 `0x80901020` 可拆分为4个独立字节（每2个十六进制位对应1个字节）：

- 最低位字节（第0字节）：`0x20`（对应地址 `0x8000fff0`）
- 第1字节：`0x10`（对应地址 `0x8000fff1`）
- 第2字节：`0x90`（对应地址 `0x8000fff2`）
- 最高位字节（第3字节）：`0x80`（对应地址 `0x8000fff3`）

二、分析题

1．7fd5 0 1 1 0

解析：

addw %bx, %ax 是**x86架构的16位加法指令**，功能是将源操作数 %bx （16位）加到目的操作数 %ax （16位），结果存入 %ax ，并更新标志寄存器（ZF、CF、OF、SF）。需分**数值计算**和**标志位判断**两步分析：

1. **数值计算（16位加法）**：
   - ax = 0xffd0 （十六进制）→ 十进制为 65488 （无符号）或 -48 （16位补码，因最高位为1，0x10000 - 0xffd0 = 0x30 → 48 ，故为 -48 ）。
   - bx = 0x8005 （十六进制）→ 十进制为 32773 （无符号）或 -32763 （16位补码，因最高位为1，0x10000 - 0x8005 = 0x7ffb → 32763 ，故为 -32763 ）。
   - 执行加法： 0xffd0 + 0x8005 = 0x17fd5 （十六进制），但 addw 是16位指令，仅保留**低16位**，即 0x7fd5 （十进制 32725 ），因此 ax 最终值为 0x7fd5 。
2. **标志位判断**：
   - **ZF（零标志）**：结果为 0x7fd5 ≠ 0 → ZF = 0 。
   - **CF（进位标志）**：无符号加法中，若结果超过16位（即产生进位）则置1。 0xffd0 + 0x8005 = 0x17fd5 （有高位进位 1 ）→ CF = 1 。
   - **OF（溢出标志）**：有符号加法中，若“两个同符号数相加后结果符号不同”则溢出。 ax = -48 （负）、 bx = -32763 （负），相加结果 0x7fd5 （正，因最高位为 0 ）→ 符号从负变正，发生溢出 → OF = 1 。
   - **SF（符号标志）**：结果的最高位（第15位）为 0 （ 0x7fd5 二进制为 0111 1111 1101 0101 ）→ SF = 0 。

OF 溢出标志计算：Cn^Cn−1

在**有符号数加法**中，溢出的本质是"结果超出有符号数能表示的范围"。硬件通过对比 **最高有效位的进位（C_n）** 和 **次高位的进位（C_{n-1}）** 是否"不同"，来判断是否溢出。

- C_n：加法时**最高位（如16位加法的第15位）**产生的进位（有进位则为1，无则为0）。
- C_{n-1}：加法时**次高位（如16位加法的第14位）**产生的进位。
- ^（异或）：若 C_n 和 C_{n-1} 状态不同（一个有进位、一个无进位），则 OF=1（溢出）；若状态相同，则 OF=0（未溢出）。

知识拓展：

1. addw 指令本质：
   属于x86汇编的**字（16位）加法指令**，格式为 addw 源操作数, 目的操作数 ，功能是 目的操作数 = 目的操作数 + 源操作数 ，并更新标志位（ZF、CF、OF、SF等）。

2. **标志位的工程意义：**
   - **ZF**：用于判断"运算结果是否为0"，常配合分支指令（如 je ）实现条件跳转。
   - **CF**：无符号数运算的"进位/借位"标志，用于多字节加法/减法的进位链。
   - **OF**：有符号数运算的"溢出"标志，确保数值未超出有符号数范围（如16位有符号数范围为 -32768 ~ 32767 ）。
   - **SF**：直接反映结果的符号（最高位值），辅助判断有符号数的正负。
3. **补码加法的一致性：**
   补码设计的核心优势是**加减法统一**——无论操作数正负，只需按二进制加法执行，由硬件自动更新标志位，无需额外区分"加法/减法逻辑"。

2.加法器为什么要生成并保存条件标志位（OF,CF,SF,ZF等）？这四个标志位分别代表什么含义?

- 机器在运行过程中需要通过条件标志位来判定运算后的状态。

- **OF**：溢出标志位（判断有符号数是否溢出）
- **CF**：进位/借位标志位（判断无符号数是否溢出）
- **SF**：符号位
- **ZF**：零标志位

无符号加溢出条件：CF=1

带符号加溢出条件：OF=1

2． .data 16 8 .rel.data 4011C8

3．0x4c7 0x100 0x1ef 0xc0 0xc0f3

4．

In FP, 除数为0的结果是 +/- ∞, 不是溢出异常.（整数除0为异常）

为什么要这样处理？ **∞ ： infinity**

- **可以利用+∞/-∞作比较。 例如：X/0>Y可作为有效比较**

How to represent +∞/-∞?

- **Exponent : all ones (11111111B = 255)**
- **Significand: all zeros**

+∞ : 0 11111111 00000000000000000000000

-∞ : 1 11111111 00000000000000000000000

Operations

5.0 / 0 = +∞, -5.0 / 0 = -∞

5+(+∞) = +∞, (+∞)+(+∞) = +∞

5 - (+∞) = -∞, (-∞) - (+∞) = -∞ etc

答案：**整数除法对"除0"无合法定义，触发硬件异常；浮点数通过IEEE754标准的"特殊值"机制（∞、NaN）处理除0，属于合法运算。**

解析：

1. **整数运算的刚性逻辑**：
   整数域中，"除以0"无数学定义（商不存在且余数无法满足"余数＜除数"）。CPU执行整数除法指令（如 `IDIV`）时，若检测到除数为0，会触发**算术异常（如中断或陷阱）**，强制终止非法操作，防止错误结果传播。
2. **浮点数的柔性设计（IEEE754标准）**：
   浮点数引入"特殊值"处理边界场景：
   - 非零浮点数÷0 → 结果为**有符号无穷大（+∞/-∞）**（如 `5.0/0.0=+∞`，`-3.0/0.0=-∞`），属于合法浮点数表示，不触发异常。
   - 0.0÷0.0 → 结果为**NaN（非数）**（标记无效运算），同样是合法格式，仅需程序逻辑处理（如标记错误），不中断执行。
3. **设计目标差异**：
   整数运算追求"精确性"，非法操作必须报错；浮点数运算追求"鲁棒性"，通过特殊值让程序在数值边界仍能运行（如科学计算中用∞表示趋势）。

三、综合应用题

1．gcc －o sum main.c sum.c
预处理，编译，汇编，链接

2．是是是否
Sum.o main.o main.o 否
外部符号 局部符号 全局符号 否

3．.rodata

4．4　　val = 16 result = −1764

5．03 00 00 00　　f9 ff ff ff

6．5　　38 00 00 00　　解法：4011e7−4011af

7．rip：4011e7rsp：0x7ff..ffddf8 栈顶单元内容：4011af

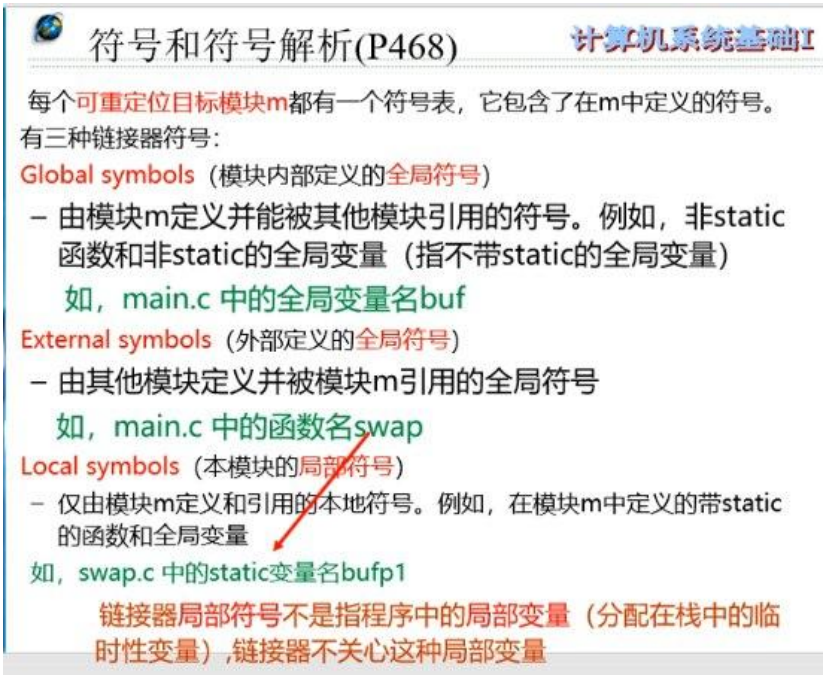
8．6 为 array[i]中内容

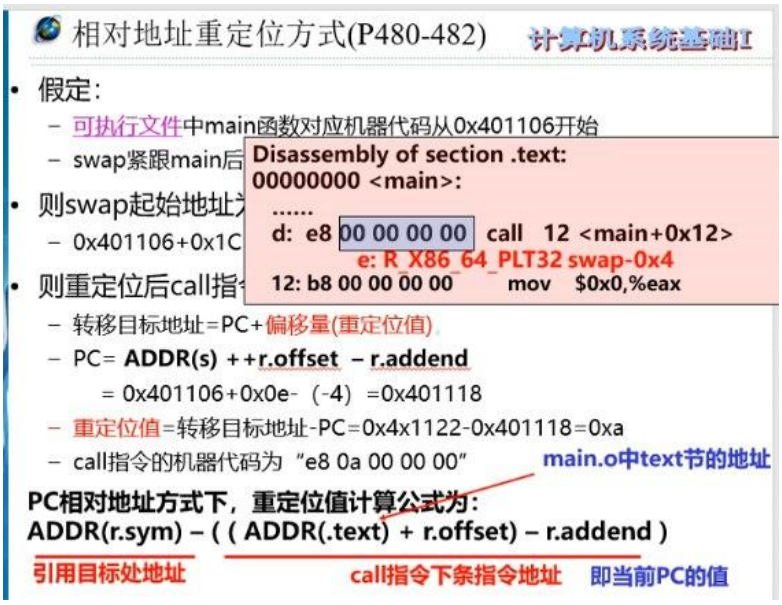
9．esi：4 ecx：−4

10．imul 0x2e7c(%rip),%eax

mov %eax,0x2e76(%rip)

11．栈破坏检测，插入金丝雀值，防止栈出错





实验题略