ICS 问题求解

PKU 04832363：计算机系统导论讨论班

2021 年秋

北京大学
PEKING UNIVERSITY

编译于 2021-10-19 16:06

汇编：王畅
勘误：请写信到 wchang@pku.edu.cn

说明：以下内容是 2021 年秋计算机系统导论班（04832363，班号 16）使用的材料。主要的练习题均来源于往年计算机系统导论课程，编者仅做了汇编和微小的改动。考题只选择了 2018 年及之前的内容，不选用更新的题目是因为 2019 年和 2020 年的试卷将留给大家复习备考时整套使用。有关 Lab 的指导则为编者自行组织。后面附有部分题目的分析和答案，“部分”选择的要旨是对较难的题目作详细解释。

目录

1 位级表示 1

2 位级表示—往年考题 5

3 Datalab 回顾 11

4 汇编语言 15

5 汇编语言—往年考题 25

6 Bomblab 回顾 41

7 体系结构初步 43

8 部分参考答案 45

1 位级表示

要点

- ▷ 知道整数、浮点数在系统的存储方式（字节序）。
- ▷ 熟练掌握整数、浮点数的位级表示规则，快速完成其同十进制数的相互转换。
- ▷ 理解整数、浮点数规范中的各类特殊数的计算方法及其性质。
- ▷ 运用浮点数舍入的规则进行运算，知道类型转换的基本规则，能针对整数、浮点数的一些常见“反常情况”进行判断。

1. 在 x86-64 机器上，定义 `unsigned int A = 0x123456`。请画出 A 在内存中的存储方式：

...	低地址	A				高地址	...
...						...	

定义 `unsigned short B[2] = {0x1234, 0x5678}`。请画出 B 在内存中的存储方式：

...	低地址	B				高地址	...
...						...	

2. 在 x86-64 机器上，有下列 C 代码：

```c
int main() {
    unsigned int A = 0x11112222;
    unsigned int B = 0x33336666;
    void *x = (void *)&A;
    void *y = 2 + (void *)&B;
    unsigned short P = *(unsigned short *)x;
    unsigned short Q = *(unsigned short *)y;
    printf("0x%04x", P + Q);
    return 0;
}
```

运行该代码，结果是什么？

3. 在 x86-64 机器上，有下列 C 代码：

```
int main() {
    char A[12] = "11224455";
    char B[12] = "11445577";
    void *x = (void *)&A;
    void *y = 2 + (void *)&B;
    unsigned short P = *(unsigned short *)x;
    unsigned short Q = *(unsigned short *)y;
    printf("0x%04x", Q - P);
    return 0;
}
```

运行该代码，结果是什么？

4. 在 x86-64 机器上，有如下的定义：

```
int x = ...; // 表达式 A
int y = ...; // 表达式 B
unsigned int ux = x;
unsigned int uy = y;
```

判断下表中的表达式是否等价：

序号	表达式 A	表达式 B
1	`x > y`	`ux > uy`
2	`(x > 0) || (x < ux)`	`1`
3	`x ^ y ^ x ^ y ^ x`	`x`
4	`((x >> 1) << 1) <= x`	`1`
5	`((x / 2) * 2) <= x`	`1`
6	`x ^ y ^ (~x) - y`	`y ^ x ^ (~y) - x`
7	`(x == 1) && (ux - 2 < 2)`	`(x == 1) && ((!!ux) - 2 < 2)`

提示 减法的运算优先级比按位异或高。布尔运算的结果都是有符号数。

5. 下列代码的目的是将字符串 A 的内容复制到字符串 B，覆盖 B 原有的内容，并输出 “Hello World”；但实际运行输出是 “Buggy Codes”。尝试找到代码中的错误。

```
int main() {
    char A[12] = "Hello World";
    char B[12] = "Buggy Codes";
    int pos;
    for (pos = 0; pos - sizeof(B) < 0; pos++)
        B[pos] = A[pos];
    printf("%s\n", B);
}
```

6. 假设某浮点数格式为 1 位符号、3 位阶码、4 位小数。下表给出了用该格式表达的浮点数 $(-1)^S M \cdot 2^E$ 与其二进制表示的关系。完成下表。

描述	二进制表示	M（写成分数）	E	f
负零		/	/	`-0.0`
/	`01000101`			
最小的非规格化负数				
最大的规格化正数				
一				`1.0`
/				`5.5`
$+\infty$		/	/	/

7. 假设浮点数格式 A 为 1 位符号、3 位阶码、4 位小数，浮点数格式 B 为 1 位符号、4 位阶码、3 位小数。回答下列问题。

(1) 格式 A 中有多少个二进制表示对应于正无穷大？

(2) 考虑能精确表示的实数的最大绝对值。A 比 B 大还是比 B 小，还是两者一样？

(3) 考虑能精确表示的实数的最小非零绝对值。A 比 B 大还是比 B 小，还是两者一样？

(4) 考虑能精确表示的实数的个数。A 比 B 多还是比 B 少，还是两者一样？

8. 判断下列说法的正确性。

(1) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + 1 > b`。

(2) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + b > b + b`。

(3) 对于任意的单精度浮点数 `a` 和 `b`，如果 `a > b`，那么 `a + 1 > b + 1`。

(4) 对于任意的双精度浮点数 `d`，如果 `d < 0`，那么 `d * d > 0`。

(5) 对于任意的双精度浮点数 `d`，如果 `d < 0`，那么 `d * 2 < 0`。

(6) 对于任意的双精度浮点数 `d`，`d == d`。

(7) 将 `float` 转换成 `int` 时，既有可能造成舍入，又有可能造成溢出。

9. 遵循 IEEE 754 浮点数标准，考虑下列代码：

```
for (int x = 0; ; x++) {
    float f = x;
    if (x != (int)f) {
        printf("%d", x);
        break;
    }
}
```

试问代码的运行结果是什么？或者死循环？

10. 遵循 IEEE 754 浮点数标准，考虑下列代码：

```
int x = 33554466; // 2^25 + 34
int y = x + 8;
for ( ; x < y; x++) {
    float f = x;
    printf("%d ", x - (int)f);
}
```

写出程序的运行结果。

2 位级表示—往年考题

1. (2018) 下列哪种类型转换既可能导致溢出，又可能导致舍入？

A. `int` 转 `float`

B. `float` 转 `int`

C. `int` 转 `double`

D. `float` 转 `double`

2. (2018) 在采用小端法存储机器上运行下面的代码，输出的结果是？（`int`、`unsigned` 为 32 位长，`short` 为 16 位长，0~9 的 ASCII 码是 `0x30~0x39`）

```
char *s = "2018";
int *p1 = (int *)s;
short s1 = (*p1) >> 12;
unsigned u1 = (unsigned) s1;
printf("0x%x\n", u1);
```

A. `0x00002303`

B. `0x00032303`

C. `0xffff8313`

D. `0x00008313`

3. (2018) 考虑如下函数

```
void XOR(int x, int y) {
    y = x ^ y;
    x = x ^ y;
    y = x ^ y;
    printf(x, y);
}
```

则 `XOR`(a, b) 的输出结果是什么？

A. a, b

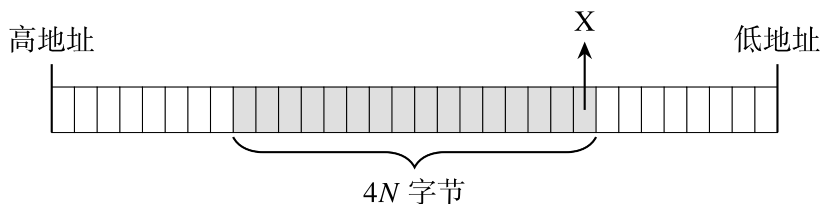
B. b, a

C. $b, 0$

D. $b, a \wedge b$

4. (2017) 假定一个特殊设计的计算机，将 `int` 型数据的长度从 4 字节扩展为 $4N$ 字节，采用大

端法。现将该 `int` 型所能表示的最小负数写入内存中，如下图所示。其中每个小矩形代表一个字节，请问 X 位置这个字节中的值是多少？



A. `00000000`
B. `01111111`
C. `10000000`
D. `11111111`

5. (2017) 以下说法正确的是：
A. 负数加上负数结果都为负数
B. 正数加上正数结果都为正数
C. 用 `&` 和 `~` 可以表示所有的逻辑与或非操作
D. 用 `&` 和 `|` 可以表示所有的逻辑与或非操作

6. (2017, 2016) 若我们采用基于 IEEE 浮点格式的浮点数表示方法，阶码字段（exp）占据 k 位，小数字段（frac）占据 n 位，则最小的规格化正数是：
A. $(1-2^{-n})\cdot 2^{-2^{k-1}+2}$
B. $2^{-2^{k-1}+2}$
C. $2^{-n}\cdot 2^{-2^{k-1}+2}$
D. $(1-2^{-n})\cdot 2^{-2^{k}+1}$

7. (2016) 假定编译器规定 `int` 和 `short` 型长度分别为 32 位和 16 位，执行下列语句：`unsigned short x = 65530; unsigned int y = x;`，得到 `y` 的机器数是 ______________。（用 16 进制表示，勿省略前导的 0）

8. (2016) 一个 C 语言程序在一台 32 位机器上运行。程序中定义了三个变量 x, y, z，其中 x 和 z 为 `int` 型，y 为 `short` 型。当 $x = 127, y = -9$ 时，执行赋值语句 $z = x + y$ 后，z 的值是 ______________。（用 16 进制表示，勿省略前导的 0）

9. (2016) 若按 IEEE 浮点标准的单精度浮点数（符号位 1 位，阶码字段 exp 占据 8 位，小数字段 frac 占据 23 位）表示 −8.25，结果是 ______________。（用 16 进制表示）

10. (2015) 给定一个实数，会因为该实数表示成单精度浮点数而发生误差。不考虑 `NaN` 和 `Inf` 的情况，该绝对误差的最大值为：
A. 2^{103}
B. 2^{104}
C. 2^{230}
D. 2^{231}

11. (2016) 现有一个二进制浮点的表示规则，其中 E 为指数部分（3 比特），bias 为 3；M 为小数部分（5 比特），采用二进制补码表示形式，且取值 $0.5 \leqslant |M| < 1$，s 是浮点的符号位。该形式包含一个值为 1 的隐藏位。问 $+5_{10}$ 在该表示下的值是下列哪一个？

A. `010001100`

B. `010100100`

C. `011011010`

D. `011110101`

12. (2015) 关于浮点数，以下说法正确的是：

A. 给定任意浮点数 a, b, x，如果 $a > b$ 成立 (指求值为 1)，则一定有 $a + x > b + x$ 成立

B. 给定任意浮点数 a, b, x，如果 $a > b$ 不成立 (指求值为 0)，则一定有 $a + x > b + x$ 不成立

C. 不考虑结果为 `NaN`，`Inf` 或运算过程发生溢出的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果

D. 不考虑结果为 `NaN`，`Inf` 的情况，高精度浮点数一定得到比低精度浮点数更精确或相同的结果

13. (2015) 在 32 位平台上，按 C90 标准判定以下语句中结果为假的是：

A. `return INT_MIN < INT_MAX;`

B. `return -2147483648 < 2147483647;`

C. `int a = -2147483648; return a < 2147483647;`

D. `return -2147483647 - 1 < 2147483647;`

提示 C90 中的类型转换顺序：`int -> long -> unsigned -> unsigned long`，$2^{31} = 2147483648$。

14. (2014) 设整数均为 32 位，其中 `unsigned x = 0x00000001; int y = 0x80000000; int z = 0x80000001;`。以下表达式求值为 1 的是：

A. `(-1) < x`

B. `(-y) > -1`

C. `~y + y == -1`

D. `(z << 4) > (z * 16)`

15. (2014) 下面说法正确的是：

A. 数 0 的反码表示是唯一的

B. 数 0 的补码表示不是唯一的

C. `1000_1111_1110_1111_1100_0000_0000_0000` 表示的唯一整数是 `0x8FEFC000`

D. `1000_1111_1110_1111_1100_0000_0000_0000` 如果是单精度浮点数，则其值是 $-(1.110111111)_2 \cdot 2^{31-127}$

16. (2016, 2014) 下面表达式中为真的是：

A. `(unsigned)-1 < -2`

B. `2147483647 > (int)2147483648U`

C. `(0x80005942 >> 4) == 0x09005942`

D. `2147483647 + 1 != 2147483648`

17. (2014) 下面关于 IEEE 浮点数标准说法正确的是哪个？

A. 在位数一定的情况下，不论怎么分配 exponent bits 和 fraction bits，所能表示的数的个数是不变的。

B. 若甲类浮点数有 10 位，乙类浮点数有 11 位，那么甲所能表示的最大数一定比乙小。

C. 若甲类浮点数有 10 位，乙类浮点数有 11 位，那么甲所能表示的最小正数一定比乙小。

D. “`0111000`”可能是 7 位浮点数的 `NaN` 表示。

18. (2014) 假设有下面 x 和 y 的程序定义：

```
int x = a >> 2;
int y = (x + a) / 4;
```

那么有 ________ 个位于闭区间 $[-8, 8]$ 内的整数 a 能使得 x 和 y 相等。

19. (2013) 对于 IEEE 浮点数，如果减少 1 位指数位，将其用于小数部分，下列叙述正确的是哪个？

A. 能表示更多数量的实数值，但实数值取值范围比原来小了。

B. 能表示的实数数量没有变化，但数值的精度更高了。

C. 能表示的最大实数变小，最小的实数变大，但数值的精度更高。

D. 以上说法都不正确。

20. (2017) 考虑有一种基于 IEEE 浮点格式的 9 位浮点表示格式 A。格式 A 有 1 个符号位、k 个阶码位、n 个小数位。现在已知 -9 的位模式可以表示为 `101100010`。回答以下问题。（注：阶码偏移量为 $2^{k-1}-1$）

(1) 求 k 和 n 的值。

(2) 基于格式 A，请填写下表。值的表示可以写成整数（如 16），或者写成分数（如 17/64）。

描述	二进制表示	值
最大的非规格化数		
最小的正规格化数		
最大的规格化数		

(3) 假设格式 A 变为 1 个符号位、$k+1$ 个阶码位、$n-1$ 个小数位，那么能表示的实数数量会怎样变化？数值的精度会怎样变化？（回答增加、降低或不变即可）

21. (2016) 在 64 位机器上，判断表达式是否对代码生成的变量恒成立（指求值为 1）。

```
/* random_int() 函数返回一个随机的 int 类型值 */
int x = random_int();
int y = random_int();
int z = random_int();
unsigned ux = (unsigned)x;
long lx = (long)x; /* long 为 64 位 */
long ly = (long)y;
double dx = (double)x;
```

```c
    double dy = (double)y;
    double dz = (double)z;
```

(1) `(x >= 0) || (3 * x < 0)`

(2) `(x >= 0) || (x < ux)`

(3) `((x >> 1) << 1) <= x`

(4) `((x - y) << 3) + (x >> 1) - y == 8 * x - 9 * y + x / 2`

(5) `(x - y > 0) == ((y + ~x + 1) >> 31 == 1)`

(6) `dx + dy == (double) (y + x)`

(7) `dx + dy + dz == dz + dy + dx`

(8) `(int)((lx + ly) >> 1) == ((x & y) + ((x ^ y) >> 1))`

22. (2016) 假设 C 语言中新定义了一种数据类型 T, 该类型为 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，其字段划分如下：符号位 (s) 1-bit、阶码字段 (exp) 6-bit、小数字段 (frac) 5-bit。

(1) 若将该格式下能表示的所有正规格数从小到大依次排列，则相邻两数之间差值的最小值为 ____________，最大值为 ____________。

(2) 现定义了如下变量：

```c
    T a = -15.875;
    T b = (1 << 28) + (1 << 24) + (1 << 22);
    T c = a * b;
```

给出各变量的二进制表示。

23. (2015) 对于下面的每一个表达式，请选择以下选项中的**一个或多个**，使得该表达式恒成立，如果没有满足条件的选项则填写 none：A. < B. > C. == D. != E. none。

题目中出现的变量定义如下（浮点数保证不是 `NaN` 或者 `Inf`）：`int x, y; unsigned ux = x; double d;`。

(1) 如果 `x > 0`，则 `x + 1` ____________ `0`

(2) 如果 `x > y`，则 `ux` ____________ `y`

(3) 如果 `((x << 31) >> 31) < 0`，则 `x & 1` ____________ `0`

(4) 如果 `((unsigned char)x >> 1) < 64`，则 `(char)x` ____________ `0`

(5) 如果 `d < 0`，则 `d * 2` ____________ `0`

(6) 如果 `d < 0`，则 `d * d` ____________ `0`

(7) `x ^ y ^ (~x) - y` ____________ `y ^ x ^ (~y) - x`

(8) `(((!!ux)) << 31) >> 31)` ____________ `(((!!x) << 31) >> 31)`

24. (2015) 考虑一种 12-bit 长的浮点数，此浮点数遵循 IEEE 浮点数格式，浮点数的字段划分如下：符号位 (s) 1-bit、阶码字段 (exp) 4-bit、小数字段 (frac) 7-bit。回答下列问题。

(1) 请写出在下列区间中包含多少个用上面规则精确表示的浮点数：$[1,2)$; $[2,3)$。

(2) 填写下表。

描述	二进制表示
最大的非规格化数	
最小的正规格化数	
$17\frac{1}{16}$	
$-\frac{1}{8192}$	
$20\frac{3}{8}$	
$-\infty$	

25. (2014) 回答下列问题。

(1) 假设下列 `unsigned` 和 `int` 数均为 5 位（有符号整型用补码运算表示）：`int y = -7; unsigned z = y;`。确定 `y`，`z` 和最小有符号整数的十进制表示和二进制表示。

(2) 按照 IEEE 浮点数标准，首先将下列两个数表示 $(-1)^s M \cdot 2^E$ 的形式，然后写出其二进制表示：$0.375, -12.5$。

3 Datalab 回顾

要点

- ▷ 了解位级操作中常见的小模块，包括 de Morgan 律、加法器、减法器、popcount 等，会运用常见策略处理非常规的位运算编程要求，如掩码设计、模拟、分治、位级表示展开等。
- ▷ 了解各种运算符的优先顺序。知道 ANSI C 与 C99 等标准的不同。初步了解整数在发生强制转换时的基本规则。
- ▷ 熟练运用 IEEE 754 标准，会使用该标准直接构造浮点数。
- ▷ 推荐的课外参考书：*Hacker's Delight* 和 *Matters Computational: Ideas, Algorithms, Source Code*，以及一个链接 `https://graphics.stanford.edu/~seander/bithacks.html`，里面介绍了一些位运算魔法。Datalab 和往年有一些比较过分的考题大量出自前一本书。因为 datalab 已经 due，故列出来供大家参考。

我们下面通过几个新例子来回顾 datalab 中用到的重要技巧，供大家回顾、反思和练习，以便于确保从 datalab 中学到了东西。当然，解法不唯一。

首先是大家已经熟练使用的掩码。它是指一串二进制数字，通过与目标数字的按位操作，达到屏蔽指定位而抽取信息的需求。例如，我们要取得某个二进制数 `x` 的最高位，可以使用 `(x >> 31) & 1`，这里 `1` 就是掩码。又如，获得 `x` 的所有偶数位，可以使用 `x & 0xcccccccc`，这里 `0xcccccccc` 就是掩码，等等。这段话中我们回顾了 `allOddBits` 的做法。

在第一个正式例子中，我们回顾 `bitNot`, `bitXor` 两个题的基本思路。

3.1 例 (de Morgan 律) 我们知道 `~`, `&`, `|` 可以表达“所有”的逻辑表达式，比方说异或 `x ^ y = (x & ~y) | (~x & y)`。而实际上 `~`, `|` 就足以做到这件事，因为我们有 `x & y = ~(~x | ~y)`。这样的情况称为**连接词的完备集**，以后大家还会反复学到。

当然，其实单纯一个与非或者一个或非也够了。假设**或非**用 $\downarrow$ 表示（即 $x \downarrow y$ = `~(x | y)`），请你完成以下代码：唯一允许的位运算操作符是 $\downarrow$。

```c
unsigned xor_with_nor(unsigned x, unsigned y) {
    return ______________________; // return x ^ y with only NOR
}
```

在第二个例子中，我们总结和 `isLessOrEqual`, `sm2tc`, `counter1To5` 几个题有关的重要技巧。

3.2 例 (选择器) 试用位级运算的技术计算表达式 `cond ? t : f`。根据注释中的提示尝试补全下面的代码，假设 `cond` 的输入总是 1 或者 0。

```
int conditional(int cond, int t, int f) {
    /* Compute a mask that equals 0x00000000
     * or 0xFFFFFFFF depending on the value of cond */
    int mask = ____________________;
    /* Use the mask to toggle between returning t or returning f */
    return ____________________;
}
```

这个例子的意义是，如果你一定想要用 `if`，可以用这个办法避开 datalab 的限制。更显然地，减法、常数乘法都是事实上可以用的，此外你也可以回顾你在实验题中是怎样表达 == 的。

```
int equal(int x, int y) {
    return _________________________; // return x == y
}

int minus(int x, int y) {
    return _________________________; // return x - y
}
```

下面的例子则和 `fullSub`，`satAdd` 以及 `trueFiveEighths` 有关。

3.3 例 (加法器与减法器) 你可能以前听说过，如果要计算两个无符号数的平均 $\lfloor \frac{x+y}{2} \rfloor$ 而不发生溢出可以用：`(x & y) + ((x ^ y) >> 1)`。不难看出其原理：`x ^ y` 是半加（不考虑进位）的部分，不会发生溢出，右移即可；另一部分则包括进位，当且仅当两位均为 1 时发生，而如果将这个进位保留在原地，恰好就是除以 2 的效果。这样的思想来自于计算机的加法器。

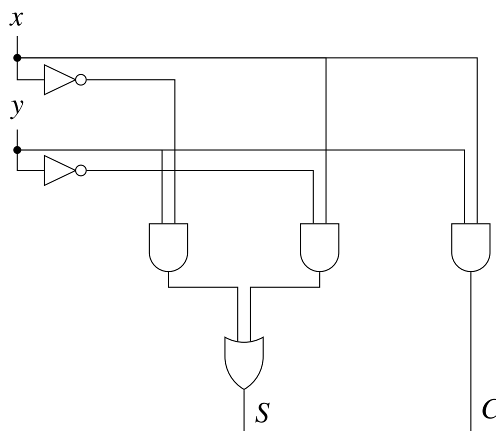


图 3.1: 半加器的结构，它计算两个一位整数的和 S 及其进位 C。可以看出 $S = x \oplus y, C = x \& y$。

假如我们要运算两个两位数相加，不难看出，只需要将第一位的进位 C 连接到下一位的加法中即可，即下一位是 $x \oplus y \oplus C$，以此类推。

如果要将加法改成减法，大家都很清楚怎么办，因为 `x - y = x + (~y + 1)`。

一般来说，有符号数的处理比无符号数稍微需要一些讨论。根据上面的方法，请你尝试给出两个有符号数的平均 $\lfloor \frac{x+y}{2} \rfloor$ 而不发生溢出的算法；对 $\lceil \frac{x+y}{2} \rceil$ 做同样的事。

与此相关，请问如何用位运算检查 $x+y$ 是否溢出了？ 提示〉 检查符号位。

```
int addOK(int x, int y) {
    _________________________________;
    _________________________________;
    // You can add more lines
    return ____; // Determine if we can compute x + y without overflow
}
```

剩下的若干例子中，我们仔细回顾 countConsecutive1 和 palindrome 这两个比较难的问题需要用到的分治技巧。

3.4 例 (计数) 让我们来决定一个给定数的二进制表示中，1 的个数是偶数还是奇数；奇数返回 1，否则返回 0。允许使用 datalab 整数部分规定的所有运算符。

这里我们采用**分治**的策略，首先考察比较短的数。例如，当所处理的数只有两位时，采用的代码十分显然：

```
int bitParity2bit(int x) {
    int bit1 = 0b01 & x;
    int bit2 = 0b01 & (x >> 1);
    return bit1 ^ bit2;
}
```

将异或理解为 $\mathbb{F}_2$ 上的加法是非常有益的。

对于四位整数，我们两位两位操作，并让操作的过程某种意义上“并行”进行。首先分别确定 1、2 和 3、4 位的奇偶性，所得结果再设法异或一次。（思考：mask2 可以改成其他数吗？）

```
int bitParity4bit(int x) {
    int mask = 0b0101;
    int halfParity = (mask & x) ^ (mask & (x >> 1));
    int mask2 = 0b0011;
    return (mask2 & halfParity) ^ (mask2 & (halfParity >> 2));
}
```

现在，请你尝试根据上面的提示，补全下面针对八位整数的算法（要求使用操作符数目不超过 12 个）。

```
int bitParity8bit(int x) {
    int mask = ____________________;
    int quarterParity = ____________________;
    int mask2 = ____________________;
    int halfParity = ____________________;
    int mask3 = ____________________;
    return ____________________;
}
```

最后，试完成针对十六位或更长整数的算法（注意，根据 datalab 要求，立即数有效长度不能超过 8 位）。

作为模仿练习，请你对问题“决定一个给定数的二进制表示中有多少个 1”做同样的事。

```
/* Let's count how many bits are set in a number */
int bitCount8bit(int x) {
    int mask = ____________________;
    int quarterSum = ____________________;
    int mask2 = ____________________;
    int halfSum = ____________________;
    int mask3 = ____________________;
    return ____________________ + ____________________;
}
```

3.5 例 (模拟操作 2) 对于一个无符号整数，我们希望将它的位反过来（记为 rev(·)），例如 `0x01234567` 倒过来就是 `0xE6A2C480`。

这个某种意义上是一个广为流传的~~面试题~~。我们沿用**分治**的思想。假设二进制数 x 有 $2m$ 位 $(\underbrace{b_{2m-1}\cdots b_m}_{x_h}\underbrace{b_{m-1}\cdots b_0}_{x_l})_2$，那么显然有

$$\mathrm{rev}(x) = \mathrm{rev}(x_l)\,\mathrm{rev}(x_h).$$

根据这个思路，请你补全以下代码：

```
unsigned reverseBits(unsigned n) {
    n = (n >> 16) | (n << 16);
    n = ((n & __________) >> 8) | ((n & __________) << 8);
    n = ____________________ | ____________________;
    n = ________________________________________;
    n = ________________________________________;
    return n;
}
```

读过两个分治的操作之后，请思考：给一个整数，如何计算它有多少个前导零？允许的操作符包括所有的位级运算，以及加法和 `!`。 提示 二分查找。

3.6 例 (算术运算) 考虑这样一个问题：对一个无符号整数 x, 计算 $x \bmod 5$。除了加法之外不能用其他算术操作。

方法上这很标准，假设 $x = (b_{31}\cdots b_1 b_0)_2$，通过计算 $2^i \pmod 5$ 的周期，我们知道

$$x \equiv \sum_{i=0}^{31} b_i 2^i \equiv 1b_0 + 2b_1 + 4b_2 + 3b_3 + 1b_4 + \cdots + 3b_{31} \pmod 5.$$

技术上，对一个四位无符号整数 $(b_3b_2b_1b_0)_2$ 计算 $b_0 + 2b_1 + 4b_2 + 3b_3$ 可以非常直接：比如首先取得各位，然后用左移配合加法计算结果。你也可以思考有什么更省操作符数目的方法。

注意，当你需要求取 $x \bmod 2^i$ 时，总是可以简单使用 `x & ((1 << i) - 1)`。

4 汇编语言

要点

- ▷ 知道 x86-64 中指令、程序计数器、寄存器、条件码、内存等概念，记住重要的寄存器名称、含义和使用规范，清楚掌握数据传送指令和操作数的正确用法。
- ▷ 掌握条件分支、条件传送、各种循环、跳转表的翻译方式，能熟练地在汇编语言中识别控制流结构，快速完成机器码、汇编语言、C 代码的相互转换。
- ▷ 理解 x86-64 系统栈空间的分布和管理方式，知道过程调用中的重要寄存器和相关保存指令，会复述过程调用的整个过程并绘制栈空间的变化情况。
- ▷ 知道结构体、联合体在内存中的存储情况，掌握用对齐规则访问结构体和联合体，以及计算其实际大小的方法。会处理复杂的指针和函数指针问题。

1. 判断下列 x86-64 ATT 操作数格式是否合法。

(1) `8(%rax, , 2)`

(2) `$30(%rax, %rax, 2)`

(3) `0x30`

(4) `13(, %rdi, 4)`

(5) `(%rsi, %rdi, 6)`

(6) `%ecx`

(7) `(%ecx)`

(8) `(%rbp, %rsp)`

2. 假设 `%rax`、`%rbx` 的初始值都是 0。根据下列一段汇编代码，写出每执行一步后两个寄存器的值。

```
movabsq   $0x0123456789ABCDEF, %rax
movw      %ax, %bx
movswq    %bx, %rbx
movl      %ebx, %eax
movabsq   $0x0123456789ABCDEF, %rax
cltq
```

3. 下列操作不等价的是：

A. `movzbq` 和 `movzbl`

B. `movzwq` 和 `movzwl`

C. `movl` 和 `movslq`

D. `movslq %eax, %rax` 和 `cltq`

4. 判断下列 x86-64 ATT 数据传送指令是否合法。

(1) `movl $0x400010, $0x800010`

(2) `movl $0x400010, 0x800010`

(3) `movl 0x400010, 0x800010`

(4) `movq $-4, (%rsp)`

(5) `movq $0x123456789AB, %rax`

(6) `movabsq $0x123456789AB, %rdi`

(7) `movabsq $0x123456789AB, 16(%rcx)`

(8) `movq 8(%rsp), %rip`

5. 在 32 位机器中有如下定义 `int array[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};`。

某一时刻，`%ecx` 存着第一个元素的地址，`%ebx` 值为 3，那么下列操作中，哪一个将 `array[3]` 移入了 `%eax`？

A. `leal 12(%ecx), %eax`

B. `leal (%ecx, %ebx, 4), %eax`

C. `movl (%ecx, %ebx, 4), %eax`

D. `movl 8(%ecx, %ebx, 2), %eax`

6. 下面的汇编代码对应一个 C 函数，其原型为 `long func(long a, long b);`。将其翻译为 C 代码。

```
// a in %rdi, b in %rsi
func:
    movq   %rdi, %rax
    salq   $4, %rax
    subq   %rdi, %rax
    movq   %rax, %rdi
    leaq   0(, %rsi, 8), %rax
    subq   %rsi, %rax
    addq   %rdi, %rax
    ret
```

7. 指令 `setg %al` 会让寄存器 `%al` 得到：

A. `~(SF ^ OF) & ~ZF`

B. `~(SF | OF) & ~ZF`

C. `~(SF | OF)`

D. `~(SF ^ OF)`

8. 下面的汇编代码对应一个 C 函数，其原型为 `long func(long a, long b);`。将其翻译为 C 代码。

```
  // a in %rdi, b in %rsi
  func:
      movl    $1, %eax
      jmp     .L2
  .L4:
      testb   $1, %sil
      je      .L3
      imulq   %rdi, %rax
  .L3:
      sarq    %rsi
      imulq   %rdi, %rdi
  .L2:
      testq   %rsi, %rsi
      jg      .L4
      rep ret
```

9. 对于下列四个函数，假设 gcc 开了编译优化，判断 gcc 是否会将其编译为条件传送。

(1)
```
long f1(long a, long b) { return (++a > --b) ? a : b; }
```

(2)
```
long f2(long a, long b) { return (*a > *b) ? --(*a) : (*b)--; }
```

(3)
```
long f3(long a, long b) { return a ? *a : (b ? *b : 0); }
```

(4)
```
long f4(long a, long b) { return (a > b) ? a++ : ++b; }
```

10. 根据下面的汇编指令补充机器码中缺失的字节。

机器码	汇编指令
loop:	
4004d0: 48 89 f8	mov %rdi, %rax
4004d3: eb ______	jmp 4004d8 <loop+0x8>
4004d5: 48 d1 f8	sar %rax
4004d8: 48 85 c0	test %rax, %rax
4004db: 7f ______	jg 4004d5 <loop+0x5>
4004dd: f3 c3	repz retq

11. 使用 GDB 查看某个可执行文件，发现其一段内存为：

```
  0x400598: 0x0000000000400488 0x0000000000400488
  0x4005a8: 0x000000000040048b 0x0000000000400493
```

```
0x4005b8: 0x000000000040049a 0x0000000000400482
0x4005c8: 0x000000000040049a 0x0000000000400498
```

根据以下汇编代码，

```
0x400474: cmp   $0x7, %edi
0x400477: ja    0x40049a
0x400479: mov   %edi, %edi
0x40047b: jmpq  *0x400598(, %rdi, 8)
0x400482: mov   $0x15213, %eax
0x400487: retq
0x400488: sub   $0x5, %edx
0x40048b: lea   0x0(, %rdx, 4), %eax
0x400492: retq
0x400493: mov   $0x2, %edx
0x400498: and   %edx, %esi
0x40049a: lea   0x4(%rsi), %eax
0x40049d: retq
```

补全主函数的 C 代码。

```
// a in %rdi, b in %rsi, c in %rdx
int main(int a, int b, int c) {
    int res = 4;
    switch (a) {
        case 0:
        case 1:
            __________;
        case ___:
            res = __________;
            break;
        case ___:
            res = __________;
            break;
        case 3:
            __________;
        case 7:
            __________;
        default:
            __________;
    }
    return res;
}
```

12. 将下列汇编代码翻译成 C 代码。

```
func:
    movq   %rsi, %rax
```

```
    testq  %rdi, %rdi
    jne    .L7
    rep ret
.L7:
    subq   $8, %rsp
    imulq  %rdi, %rax
    movq   %rax, %rsi
    subq   $1, %rdi
    call   func
    addq   $8, %rsp
    ret
```

```
long func(long n, long m) {
    if (__________)
        return __________;
    return func(__________, __________);
}
```

13. 将下列 C 代码翻译为汇编代码。

```
void callee(long *a, long *b) {
    if (a == b) return;
    *a ^= *b;
    *b ^= *a;
    *a ^= *b;
}
void caller(long n, long arr[]) {
    for (long i = 0; i < n/2; i++)
        callee(&arr[i], &arr[n-i]);
}
```

汇编代码为:

```
callee:
    cmpq   %rsi, %rdi
    je     .L1
    movq   (%rsi), %rax
    xorq   (%rdi), %rax
    movq   ____________
    xorq   (%rsi), %rax
    movq   %rax, (%rsi)
    xorq   %rax, (%rdi)
.L1:
    rep ret

caller:
    pushq  %r12
```

```
    pushq  %rbp
    pushq  %rbx
    movq   %rdi, %rbp
    movq   %rsi, %r12
    movl   $0, %ebx
    jmp    .L4
.L5:
    movq   %rbp, %rax
    subq   %rbx, %rax
    ____   (%r12, %rax, ____), %rsi
    ____   (%r12, %rbx, ____), %rdi
    call   callee
    addq   $1, %rbx
.L4:
    movq   %rbp, %rax
    shrq   $63, %rax
    addq   %rbp, %rax
    sarq   %rax
    cmpq   %rbx, %rax
    jg     .L5
    popq   ____
    popq   ____
    popq   ____
    ret
```

对于上面代码，在 x86-64、操作系统为 Linux 的情况下，假设 main 在 `0x4000ac` 处调用 caller，caller 在 `0x400088` 处调用 callee；调用函数（`call xx`）的代码长度为 5。在 main 即将调用 caller 时，部分寄存器的情况见下表左侧。请在下图右侧画出控制流第一次走到 `.L1` 时，堆栈的结构。

寄存器	调用前的值	地址	内容（不确定的空格不用填）
%rsp	0xffffffffff80	0xf...f88~8f	
%rax	0x0	0xf...f80~87	
%rbx	0x15	0xf...f78~7f	
%rbp	0x18	0xf...f70~77	
%r12	0x213	0xf...f68~6f	
%rsi	0x0	0xf...f60~67	
%rdi	0x0	0xf...f58~5f	
		0xf...f50~57	

14. 在 x86-64、Linux 操作系统下有如下 C 定义：

```c
struct A {
    char CC1[6];
    int II1;
```

```
    long LL1;
    char CC2[10];
    long LL2;
    int II2;
};
```

(1) sizeof(A) = ______。

(2) 将 A 重排后，令结构体尽可能小，那么得到的新的结构体大小为 ______ 字节。

15. 在 x86-64、LINUX 操作系统下，考虑如下的 C 定义：

```
typedef union {
    char c[7];
    short h;
} union_e;

typedef struct {
    char d[3];
    union_e u;
    int i;
} struct_e;

struct_e s;
```

(1) s.u.c 的首地址相对于 s 的首地址的偏移量是 ______ 字节。

(2) sizeof(union_e) = ______ 字节。

(3) s.i 的首地址相对于 s 的首地址的偏移量是 ______ 字节。

(4) sizeof(struct_e) = ______ 字节。

(5) 若只将 i 的类型改成 short，那么 sizeof(struct_e) = ______ 字节。

(6) 若只将 h 的类型改成 int，那么 sizeof(union_e) = ______ 字节。

(7) 若将 i 的类型改成 short、h 的类型改成 int，则 sizeof(union_e) = ______ 字节，sizeof(struct_e) = ______ 字节。

(8) 若只将 short h 的定义删除，那么 (1)~(4) 问的答案分别是 ____、____、____、____ 字节。

16. 以下提供了一段代码的 C 语言、汇编语言以及运行到某一时刻栈的情况。

```
0000000000400596 <func>:
    400596: sub    $0x28, %rsp
    40059a: mov    %fs:0x28, %rax
    4005a3: mov    %rax, 0x18(%rsp)
    4005a8: xor    %eax, %eax
    4005aa: mov    (%rdi), %rax
    4005ad: mov    0x8(%rdi), %rdx
```

```
    4005b1: cmp    %rdx, %rax
    4005b4: jge    _____(1)_____
    4005b6: mov    %rdx, (%rdi)
    4005b9: mov    %rax, 0x8(%rdi)
    4005bd: mov    0x8(%rdi), %rax
    4005c1: test   %rax, %rax
    4005c4: jne    4005cb <func+0x35>
    4005c6: mov    (%rdi), %rax
    4005c9: jmp    _____(2)_____
    4005cb: mov    (%rdi), %rdx
    4005ce: sub    %rax, %rdx
    4005d1: mov    %rdx, (%rsp)
    4005d5: mov    %rax, 0x8(%rsp)
    4005da: mov    _____(3)_____, %rdi
    4005dd: callq  400596 <func>
    4005e2: mov    0x18(%rsp), %rcx
    4005e7: xor    _____(4)_____, %rcx
    4005f0: _____(5)_____     4005f7 <func+0x61>
    4005f2: callq  400460 <__stack_chk_fail@plt>
    4005f7: add    _____(6)_____, %rsp
    4005fb: retq

00000000004005fc <main>:
    4005fc: sub    $0x28, %rsp
    400600: mov    %fs:0x28, %rax
    400609: mov    %rax, 0x18(%rsp)
    40060e: xor    %eax, %eax
    400610: movq   0x69, (%rsp)
    400618: movq   0xfc, 0x8(%rsp)
    400621: mov    %rsp, %rdi
    400624: callq  400596 <func>
    400629: mov    %rax, %rsi
    40062c: mov    $0x4006e4, %edi
    400631: mov    $0x0, %eax
    400636: callq  400470 <printf@plt>
    40063b: mov    0x18(%rsp), %rdx
    400640: xor    _____(4)_____, %rdx
    400649: _____(5)_____     400650 <main+0x54>
    40064b: callq  400460 <__stack_chk_fail@plt>
    400650: mov    $0x0, %eax
    400655: add    _____(6)_____, %rsp
    400659: retq
```

C 语言代码：

```
    typedef struct{
        long a;
```

```
        long b;
    } pair_type;

    long func(pair_type *p) {
        if (p -> a < p -> b) {
            long temp = p -> a;
            p -> a = p -> b;
            p->b = temp;
        }
        if (______(7)______) {
            return p->a;
        }
        pair_type np;
        np.a = ______(8)______;
        np.b = ______(9)______;
        return func(&np);
    }

    int main(int argc, char* argv[]) {
        pair_type np;
        np.a = ______(10)______;
        np.b = ______(11)______;
        printf("%ld", func(&np));
    }
```

堆栈情况如下（从高地址向低地址列举）:

```
 1. 0x0000000000000000
 2. 0xc76d5add7bbeaa00
 3. 0x00007fffffffdf60
 4. ____________________
 5. ____________________
 6. 0x0000000000400629
 7. ____________________
 8. ____________________
 9. 0x0000000000000001
10. 0x0000000000000069
11. 0x0000000000000093
12. ____________________
13. 0x00000000ff000000
14. ____________________
15. 0x0000000000000000
16. ____________________
17. ____________________
18. ____________________
```

```
19. 0x0000000000000000
20. ____________________
21. ____________________
22. 0x000000000000002a
23. 0x000000000000003f
24. 0x00000000004005e2
```

一些可能用到的字符的 ASCII 码如下：

换行	空格	"	%	(	)	,	0	A	a
0x0a	0x20	0x22	0x25	0x28	0x29	0x2c	0x30	0x41	0x61

回答下列问题。

(1) gdb 下使用命令 `x/4b 0x4006e4` 后（即查看 `0x4006e4` 开始的 4 个字节，用 16 进制表示）得到的输出结果是 ____________。

(2) 互相翻译 C 语言代码和汇编代码，补充缺失的空格（标号相同的为同一格）。

(3) 补充栈的内容。使用 16 进制，可以不写前导多余的 0；对于给定已知条件后仍无法确定的值，填写“不确定”；已知程序运行过程中寄存器 `%fs` 的值没有改变。

(4) 程序运行结果是 ____________。

5 汇编语言—往年考题

1. (2018) 在 x86-64 下，以下哪个选项的说法是错误的?

A. `movl` 指令以寄存器作为目的时，会将该寄存器的高位 4 字节设置为 0

B. `cltq` 指令的作用是将 `%eax` 符号扩展到 `%rax`

C. `movabsq` 指令只能以寄存器作为目的

D. `movswq` 指令的作用是将零扩展的字传送到四字节目的

2. (2018) 下列关于程序控制结构的机器代码实现的说法中，正确的是:

A. 使用条件跳转语句实现的程序片段比使用条件赋值语句实现的同一程序片段的运行效率高

B. 使用条件跳转语句实现的程序片段与使用条件赋值语句实现的同一程序片段虽然效率可能不同，但在 C 语言的层面上看总是有着相同的行为

C. 一些 `switch` 语句不会被 gcc 用跳转表的方式实现

D. 以上说法都不正确

3. (2018) 下列关于条件码的叙述中，不正确的是:

A. 所有算术指令都会改变条件码

B. 所有比较指令都会改变条件码

C. 所有与数据传送有关的指令都会改变条件码

D. 条件码一般不会直接读取，但可以直接修改

4. (2017) 在下列的 x86-64 汇编代码中，错误的是:

A. `movq %rax, (%rsp)`

B. `movl $0xFF, (%ebx)`

C. `movsbl (%rdi), %eax`

D. `leaq (%rdx, 1), %rdx`

5. (2017) 在下列关于条件传送的说法中，正确的是:

A. 条件传送可以用来传送字节、字、双字和四字的数据

B. C 语言中的“`?:`”条件表达式都可以编译成条件传送

C. 使用条件传送总可以提高代码的执行效率

D. 条件传送指令不需要用后缀 (例如 `b`, `w`, `l`, `q`) 来表明操作数的长度

6. (2017) 在下列指令中，其执行会影响条件码中的 CF 位的是:

A. `jmp NEXT`

B. `jc NEXT`
C. `inc %bx`
D. `shl $1, %ax`

7. (2016) 下列关于比较指令 `cmp` 说法中，正确的是：
A. 专用于有符号数比较
B. 专用于无符号数比较
C. 专用于串比较
D. 不区分比较的对象是有符号数还是无符号数

8. (2016) 在如下代码段的跳转指令中，目的地址是：

```
400020: 74 F0 je   ________
400022: 5d    pop  %rbp
```

A. `400010`
B. `400012`
C. `400110`
D. `400112`

9. (2016) 对于如下的 C 语言中的条件转移指令，它所对应的汇编代码中至少包含几条条件转移指令：`if (a > 0 && a != 1 || a < 0 && a != -1) b = a;`？
A. 2 条
B. 3 条
C. 4 条
D. 5 条

10. (2016) 将 `%ax` 清零，下列指令不能实现该效果的是：
A. `sub %ax, %ax`
B. `xor %ax, %ax`
C. `test %ax, %ax`
D. `and $0, %ax`

11. (2016) 在如下 `switch` 语句翻译得到的跳转表中，哪些标号没有出现在分支中？

```
addq    $1, %rdi
cmpq    $8, %rdi
ja      .L2
jmp     *.L4(, %rdi, 8)
.L4:
    .quad .L9
    .quad .L5
    .quad .L6
    .quad. L7
    .quad .L2
    .quad .L7
```

```
        .quad .L8
        .quad .L2
        .quad .L5
```

A. 3, 6
B. −1, 4
C. 0, 7
D. 2, 4

12. (2016) 已知短整型数组 S 的起始地址和下标 i 分别存放在寄存器 %rdx 和 %rcx，将 &S[i] 存放在寄存器 %rax 中所对应的汇编代码是：

A. `leaq (%rdx, %rcx, 1), %rax`
B. `movw (%rdx, %rcx, 2), %rax`
C. `leaq (%rdx, %rcx, 2), %rax`
D. `movw (%rdx, %rcx, 1), %rax`

13. (2015) 下列寻址模式中，正确的是：

A. `(%eax, , 4)`
B. `(%eax, %esp, 3)`
C. `123`
D. `$1(%ebx, %ebp, 1)`

14. (2015) 假设某条 C 语言 switch 语句编译后产生了如下的汇编代码及跳转表：

```
    movl    8(%ebp), %eax
    subl    $48, %eax
    cmpl    $8, %eax
    ja      .L2
    jmp     *.L7(, %eax, 4)
    .L7:
        .long .L3
        .long .L2
        .long .L2
        .long .L5
        .long .L4
        .long .L5
        .long .L6
        .long .L2
        .long .L3
```

在源程序中，下面的哪些标号出现过？

A. '2', '7'
B. 1
C. '3'
D. 5

15. (2015, 2014) 下列的指令组中，哪一组指令只改变条件码，而不改变寄存器的值？

A. `CMP, SUB`
B. `TEST, AND`
C. `CMP, TEST`
D. `LEAL, CMP`

16. (2014) 简单的 `switch` 语句常采用跳转表的方式实现，在 x86-64 系统中，下述最有可能是正确的 `switch` 分支跳转汇编指令的是哪个？

A. `jmp .L3(, %eax, 4)`
B. `jmp .L3(, %eax, 8)`
C. `jmp *.L3(, %eax, 4)`
D. `jmp *.L3(, %eax, 8)`

17. (2018) 以下代码的输出结果不可能是

```
union {
    double d;
    struct {
        int i;
        char c[4];
    } s;
} u;
u.d = 1;
printf("%d\n", u.s.c[2]);
```

A. 0
B. −16
C. 240
D. 191

18. (2018) 下列关于 C 语言中的结构体（struct）以及联合（union）的说法中，正确的是：

A. 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列不一定会使之内存占用最小
B. 对于任意 struct，将其成员按照其实际占用内存大小从小到大的顺序进行排列一定不会使之内存占用最大
C. 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列不一定会使之内存占用最小
D. 对于任意 union，将其成员按照其实际占用内存大小从小到大的顺序进行排列一定不会使之内存占用最大

19. (2017) 有如下代码段：

```
int func(int x, int y);
int (*p) (int a, int b);
p = func;
```

```
p(0, 0);
```

对应的下列 x86-64 过程调用正确的是：

A. `call *%rax`
B. `call *(%rax)`
C. `call (%rax)`
D. `call func`

20. (2017) 有定义：`int A[3][2] = {{1,2}, {3,3}, {2,1}};`，则 `A[2]` 是：

A. `&A + 16`
B. `A + 16`
C. `*A + 4`
D. `*A + 2`

21. (2015) 已知下面的数据结构，假设在 Linux/IA32 下要求对齐，这个结构的总的大小是多少个字节？如果重新排列其中的字段，最少可以达到多少个字节？

```
struct {
    char a;
    double *b;
    double c;
    short d;
    long long e;
    short f;
};
```

A. 32, 28
B. 36, 32
C. 28, 26
D. 26, 26

22. (2015) 在"大端法"下，已知如下的 C 语言数据结构：`union { char c[2]; int i; };`。当 `c` 的值为 `0x01, 0x23` 时，`i` 的值为：

A. `0x0123`
B. `0x2301`
C. `0x01230000`
D. 不确定

23. (2014) 有如下定义的结构，在 x86-64 下，下述结论中错误的是？

```
struct {
    char c;
    union {
        char vc;
        double value;
        int vi;
```

```
        } u;
        int i;
    } sa;
```

A. `sizeof(sa) == 24`
B. `(&sa.i - &sa.u.vi) == 8`
C. `(&sa.u.vc - &sa.c) == 8`
D. 优化成员变量的顺序，可以做到 `sizeof(sa) == 16`

24. (2013) 32 位 x86、Windows 操作系统下定义 struct `S` 包含: `double a, int b, char c`, 请问 `S` 在内存空间中最多和最少分别能占据多少个字节（32 位 Windows 系统按 1、4、8 的原则对齐 `char, int, double`)?

A. 16, 13
B. 16, 16
C. 24, 13
D. 24, 16

25. (2018) 假设在 64 位 Linux 机器上有数据结构定义如下：

```
    typedef struct s1 {
        char cc[N];
        int ii[N];
        int *ip;
    } S1;

    S1 t1[N];
```

(1) 分别确定当 $N = 3, 4, 5$ 时，`sizeof(S1)` 和 `sizeof(T1)` 的输出。

(2) 当 $N = 4$ 时，该数据结构初始化代码如下:

```
    void init(int n) {
        int i;
        for (i = 0; i < n; i++) {
            t1[i].ip = &(t1[i].ii[i]);
        }
    }
```

根据上述代码，填写下面汇编中缺失的内容：

```
    init:
        movl    $0, %ecx
        jmp     .L2

    .L3:
        movslq  %ecx, %rax
        leaq    (______, %rax, 8), %rsi
        leaq    0(, %rsi, 4), %rdx
```

```
    addq    $t1+4, %rdx
    salq    ______, %rax
    movq    ______, ______
    addl    $1, %ecx

.L2
    cmpl    ______, %ecx
    jl      .L3
    rep ret
```

(3) 当 $N = 3$ 时，函数 fun 的汇编代码如下：

```
fun:
    movslq   %esi, %rax
    movslq   %edi, %rdi
    leaq     (%rdi, %rdi), %rdx
    leaq     (%rdx, %rdi), %r8
    leaq     (%r8, %r8), %rcx
    addq     %rcx, %rax
    movl     %esi, t1+4(, %rax, 4)
    addq     %rdx, %rdi
    leaq     0(, %rdi, 8), %rax
    movq     t1+16(%rax), %rax
    movl     %esi, (%rax)
    ret
```

根据上述代码，填写函数 fun 的 C 语言代码：

```
void fun(int x, int y) {
    __________ = __________;
    __________ = __________;
}
```

26. (2017) 分析下面 C 语言程序和相应的 x86-64 汇编程序，请填写缺失的内容。

```
#include <stdio.h>
#include "string.h"

void myprint(char *str) {
    char buffer[16];
    __________(buffer, str);
    printf("%s \n", buffer);
}

void alert(void) {
    printf("__________\n");
}
```

```
int main(int argc, char *argv[]) {
    myprint("1234567123456712345671234567\xaa\x84\x04\x08");
    return 0;
}
```

```
    .section .rodata
.LC0:
    .string  "____________"
    .text
    .globl   myprint
    .type    myprint, @function
myprint:
.LFB0:
    .cfi_startproc
    pushq    %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset 6, -16
    movq     %rsp, %rbp
    .cfi_def_cfa_register 6
    subq     $48, ____________
    movq     %rdi, -40(%rbp)
    movq     %fs:40, ____________
    movq     %rax, -8(%rbp)
    xorl     %eax, %eax
    movq     ____________, %rdx
    leaq     -32(%rbp), %rax
    movq     %rdx, ____________
    ____________________________________
    call     strcpy
    _____    -32(%rbp), %rax
    movq     %rax, %rsi
    movl     $.LC0, %edi
    movl     $0, %eax
    call     printf
    nop
    _____    ____________, %rax
    xorq     ____________, ____________
    je       ____________
    call     __stack_chk_fail
.L2:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc

.LFE0:
```

```
	.size	myprint, .-myprint
	.section .rodata
.LC1:
	.string	"Where am I?"
	.text
	.globl	alert
	.type	alert, @function
alert:
.LFB1:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	movl	$.LC1, %edi
	call	puts
	nop
	popq	%rbp
	.cfi_def_cfa 7, 8
	______________________________
	.cfi_endproc

.LFE1:
	.size	alert, .-alert
	.section .rodata
	.align 8
.LC2:
	.string	"1234567123456712345671234567\252\____\004\b"
	.text
	.globl	main
	.type	main, @function
main:
	.LFB2:
	.cfi_startproc
	pushq	%rbp
	.cfi_def_cfa_offset 16
	.cfi_offset 6, -16
	movq	%rsp, %rbp
	.cfi_def_cfa_register 6
	subq	$16, %rsp
	movl	%edi, -4(%rbp)
	movq	%rsi, -16(%rbp)
	movl	$.LC2, %edi
	______________________________
	movl	$0, %eax
```

```
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
.LFE2:
    .size   main, .-main
```

27. (2015) 在 32 位机器中，有如下声明。

```
union ELE {
    struct {
        int x;
        int *p;
    } e1;
    struct {
        union ELE *next;
        int y;
    } e2;
};
```

(1) 上面的 union 有几个字节?

(2) 假设编译器为 process 函数产生了如下了代码，请补充完整下面的过程。已知只有一个不需要任何强制类型转换且不违反任何类型限制的答案。

```
movl    8(%ebp), %eax
movl    (%eax), %ecx
movl    4(%ecx), %edx
movl    (%edx), %edx
subl    4(%eax), %edx
movl    %edx, (%ecx)
```

```
void process(union ELE *up) {
    up->__________ = __________ - __________;
}
```

(3) 假设整型变量 n 在栈中 %ebp + 8 位置，ELE* 型变量 up 在栈中 %ebp + 12 位置。已知以 *up 为头元素（记为第 0 个），由声明中的 next 指针连接形成了一个链表，现在希望将第 n 个元素（假设链表足够长）的 x 的值放入 %eax 中。以下完成该功能的汇编代码有错，请找出所有错误并改正。

```
    xorl    %ecx, %ecx
    movl    8(%edx), %ebp
    movl    12(%ebp), %eax
LOOP:
    movl    (%eax), %eax
    add     $1, %ecx
    test    %ecx, %edx
```

```
        jne      LOOP
        movl     (%eax), %eax
```

(4) 阅读下列代码，回答后面的问题。

```
typedef struct {
    short x[A][B];
    int y;
} str1;

typedef struct {
    char array[B];
    int t;
    short s[B];
    int u;
} str2;

void setVal(str1 *p, str2 *q) {
    int v1 = q->t;
    int v2 = q->u;
    p->y = v1 + v2;
}
```

编译器为 setVal 产生下面的代码：

```
movl     12(%ebp), %eax
movl     28(%eax), %edx
addl     8(%eax), %edx
movl     8(%ebp), %eax
movl     %edx, 44(%eax)
```

给出 A， B 的值。

28. (2014) 一个函数如下，其中部分代码被隐去。

```
int f(int n, int m) {
    if (m > 0) {
        if (____________________) {
            int r = ____________________;
            return ____________________;
        }
        else if (____________________) {
            return 1;
        }
    }
    return 0;
}
```

如下是通过 gcc -g -O2 命令编译后，在 gdb 中通过 disas f 命令得到的反汇编代码，其中

有两个汇编指令不全。

```
0x00000000004004e0 <f+0>:     mov     %rbx, -0x10(%rsp)
0x00000000004004e5 <f+5>:     mov     ____________________
0x00000000004004ea <f+10>:    xor     %eax, %eax
0x00000000004004ec <f+12>:    sub     $0x10, %rsp
0x00000000004004f0 <f+16>:    test    %esi, %esi
0x00000000004004f2 <f+18>:    mov     %edi, %ebp
0x00000000004004f4 <f+20>:    mov     %esi, %ebx
0x00000000004004f6 <f+22>:    jle     0x400513 <f+51>
0x00000000004004f8 <f+24>:    cmp     $0x1, %edi
0x00000000004004fb <f+27>:    jle     0x400521 <f+65>
0x00000000004004fd <f+29>:    lea     -0x1(%rbp), %edi
0x0000000000400500 <f+32>:    callq   0x4004e0 <f>
0x0000000000400505 <f+37>:    lea     -0x1(%rax, %rbx, 1), %edx
0x0000000000400509 <f+41>:    mov     %edx, %eax
0x000000000040050b <f+43>:    sar     $0x1f, %edx
0x000000000040050e <f+46>:    idiv    %ebp
0x0000000000400510 <f+48>:    lea     0x1(%rdx), %eax
0x0000000000400513 <f+51>:    mov     ____________________
0x0000000000400517 <f+55>:    mov     0x8(%rsp), %rbp
0x000000000040051c <f+60>:    add     $0x10, %rsp
0x0000000000400520 <f+64>:    retq
0x0000000000400521 <f+65>:    sete    %al
0x0000000000400524 <f+68>:    movzbl  %al, %eax
0x0000000000400527 <f+71>:    jmp     0x400513 <f+51>
```

(1) 补全 C 代码和汇编指令。

(2) 已知在调用函数 `f(4, 3)` 时，我们在函数 `f` 中指令 `retq` 处设置了断点，下面列出的是程序在第一次运行到断点处暂停时时，相关通用寄存器的值。

- ▷ %rax 0x3
- ▷ %rcx 0x3
- ▷ %rdx 0x309c552970
- ▷ %rsi 0x3
- ▷ %rdi 0x1
- ▷ %rbp 0x2
- ▷ %rsp 0x7fffffffe340
- ▷ %rip 0x400520

请根据你对函数及其汇编代码的理解，填写当前栈中的内容。如果某些内存位置处内容不确定，请填写 x。

- ▷ 0x7fffffffe38c ____________________
- ▷ 0x7fffffffe388 ____________________
- ▷ 0x7fffffffe384 ____________________
- ▷ 0x7fffffffe380 ____________________

- ▷ 0x7fffffffe37c ____________________
- ▷ 0x7fffffffe378 ____________________
- ▷ 0x7fffffffe374 ____________________
- ▷ 0x7fffffffe370 ____________________
- ▷ 0x7fffffffe36c ____________________
- ▷ 0x7fffffffe368 ____________________
- ▷ 0x7fffffffe364 ____________________
- ▷ 0x7fffffffe360 ____________________
- ▷ 0x7fffffffe35c ____________________
- ▷ 0x7fffffffe358 ____________________
- ▷ 0x7fffffffe354 ____________________
- ▷ 0x7fffffffe350 ____________________
- ▷ 0x7fffffffe34c ____________________
- ▷ 0x7fffffffe348 ____________________
- ▷ 0x7fffffffe344 ____________________
- ▷ 0x7fffffffe340 ____________________
- ▷ 0x7fffffffe33c ____________________
- ▷ 0x7fffffffe338 ____________________
- ▷ 0x7fffffffe334 ____________________
- ▷ 0x7fffffffe330 ____________________
- ▷ 0x7fffffffe32c ____________________
- ▷ 0x7fffffffe328 ____________________
- ▷ 0x7fffffffe324 ____________________
- ▷ 0x7fffffffe320 ____________________

29. (2014) 阅读下面的汇编代码，根据汇编代码填写 C 代码中缺失的部分，然后描述该程序的功能。

```
        pushl    %ebp
        movl     %esp, %ebp
        movl     $0x0, %ecx
        cmpl     $0x0, 8(%ebp)
        jle      .L1
    .L2
        movl     $0x0, %edx
        movl     8(%ebp), %eax
        divl     $0x0a
        addl     %edx, %ecx
        movl     %eax, 8(%ebp)
        cmpl     $0x0, 8(%ebp)
        jg       .L2
    .L1
        movl     0x0, %edx
        movl     %ecx, %eax
```

```
    divl    0x3
    cmpl    0x0, %edx
    jne     .L3
    movl    0x1, %eax
    jmp     .L4
.L3
    movl    0x0, %eax
.L4
```

```
int fun(_____ x) {
    int bit_sum = 0;
    while (_______________) {
        ____________________;
        ____________________;
    }
    if (_______________) {
        return 1;
    } else {
        return 0;
    }
}
```

30. (2013) 阅读下面的 C 代码：

```
/**
 * int_sqrt - rough approximation to sqrt
 * @x: integer of which to calculate the sqrt
 *
 * A very rough approximation to the sqrt() function.
 */
unsigned long int_sqrt(unsigned long x) {
    unsigned long b, m, y = 0;
    if (x <= 1)
        return x;
    m = 1UL << (BITS_PER_LONG - 2);
    while (m != 0) {
        b = y + m;
        y >>= 1;
        if (x >= b) {
            x -= b;
            y += m;
        }
        m >>= 2;
    }
    return y;
}
```

已知在 64 位的机器上 BITS_PER_LONG 的定义为 long 类型的位宽。请根据代码填写下面的汇编指令：

```
4004c4: push   %rbp
4004c5: mov    %rsp, %rbp
4004c8: mov    %rdi, -0x28(%rbp)
4004cc: movq   __________, -0x8(%rbp)
4004d4: cmpq   $0x1, -0x28(%rbp)
4004d9: ja     __________ <int_sqrt+??>
4004db: mov    -0x28(%rbp), %rax
4004df: jmp    __________ <int_sqrt+??>
4004e1: movl   $0x0, -0x10(%rbp)
4004e8: movl   __________, -0xc(%rbp)
4004ef: jmp    __________ <int_sqrt+??>
4004f1: mov    -0x10(%rbp), %rax
4004f5: mov    -0x8(%rbp), %rdx
4004f9: lea    __________, %rax
4004fd: mov    %rax, -0x18(%rbp)
400501: shrq   -0x8(%rbp)
400505: mov    -0x28(%rbp), %rax
400509: cmp    -0x18(%rbp), %rax
40050d: jb     __________ <int_sqrt+??>
40050f: mov    -0x18(%rbp), %rax
400513: sub    %rax, -0x28(%rbp)
400517: mov    -0x10(%rbp), %rax
40051b: add    %rax, -0x8(%rbp)
40051f: shrq   __________, -0x10(%rbp)
400524: cmpq   $0x0, -0x10(%rbp)
400529: jne    __________ <int_sqrt+??>
40052b: mov    -0x8(%rbp), __________
40052f: leaveq
400530: retq
```

31. (2013) 某单参数函数 f 的主体的汇编代码如下：

```
4004c4: push   %rbp
4004c5: mov    %rsp, %rbp
4004c8: sub    $0x10, %rsp
4004cc: mov    %edi, -0x4(%rbp)
4004cf: cmpl   $0x1, -0x4(%rbp)
4004d3: ja     4004dc <f+0x18>
4004d5: mov    $0x1, %eax
4004da: jmp    40052d <f+0x69>
4004dc: mov    -0x4(%rbp), %eax
4004df: and    $0x1, %eax
4004e2: test   %eax, %eax
4004e4: jne    4004f5 <f+0x31>
```

```
4004e6: mov    0x200440(%rip), %eax        # 60092c <x.1604>
4004ec: add    $0x1, %eax
4004ef: mov    %eax, 0x200437(%rip)        # 60092c <x.1604>
4004f5: mov    -0x4(%rbp), %eax
4004f8: and    $0x1, %eax
4004fb: test   %al, %al
4004fd: je     40050e <f+0x4a>
4004ff: mov    0x20042b(%rip), %eax        # 600930 <y.1605>
400505: add    $0x1, %eax
400508: mov    %eax, 0x200422(%rip)        # 600930 <y.1605>
40050e: mov    -0x4(%rbp), %eax
400511: sub    $0x1, %eax
400514: mov    %eax, %edi
400516: callq  4004c4 <f>
40051b: mov    0x20040f(%rip), %edx        # 600930 <y.1605>
400521: lea    (%rax, %rdx, 1), %edx
400524: mov    0x200402(%rip), %eax        # 60092c <x.1604>
40052a: lea    (%rdx, %rax, 1), %eax
40052d: leaveq
40052e: retq
```

对应的 C 代码为：

```
#define N ______________
#define M ______________
struct P1 { char c[N]; char *d[N]; char e[N]; } P1;
struct P2 {int i[M]; char j[M]; short k[M]; } P2;

unsigned int f(unsigned int n) {
    _____ unsigned int x = sizeof(P1);
    _____ unsigned int y = sizeof(P2);
    if (______________) return 1;
    if (______________) x++;
    if (______________) y++;
    return ______________;
}
```

(1) 补全上面的空缺。

(2) 执行 `printf("%x, %x\n", f(2), f(2));` 得到的输出是 ____________。

6 Bomblab 回顾

要点

- ▹ 熟练掌握汇编语言语法，会阅读汇编语言中的各种程序结构，能和 C 语言进行快速转换。
- ▹ 知道如何使用 GDB 进行调试。

1. 写出使用 gcc 编译源代码 `lab.c`、生成可执行文件 `lab`、采用二级编译优化的命令。

2. 写出使用 gcc 编译源代码 `foo.c`、生成汇编语言文件 `foo.s` 的命令。

3. 将可执行文件逆向工程为汇编代码的工具是：

A. gcc
B. gdb
C. objdump
D. hexedit
E. gedit

4. Gdb 中，单步指令执行的命令是：

A. `r`
B. `b`
C. `p`
D. `finish`
E. `si`
F. `disas`

5. (2013) 在完成本课程的 bomblab 的时候，通常先执行 `gdb bomb` 启动调试，然后执行 ____ `explode_bomb` 命令以防引爆炸弹，之后在进行其他必要的设置后，最后执行 ____ 命令以便开始执行程序。上述两个空格对应的命令是：

A. `st, ru`
B. `br, go`
C. `br, ru`
D. `st, go`

7 体系结构初步

要点

- ▷ 了解体系结构的简要发展史，熟悉 RISC、CISC 的区别和应用。了解基本的门电路、算术运算器、选择器、触发器和寄存器的原理和实现。
- ▷ 熟悉 Y86 “体系结构”中，各种指令的编码规则（包括操作数、指令子类型等），会将汇编代码和指令编码相互转换。简单了解 MIPS 体系结构。

1. 下列描述更符合（早期）RISC 还是 CISC？

(1) 指令机器码长度固定。

(2) 指令类型多、功能丰富。

(3) 不采用条件码。

(4) 实现同一功能，需要的汇编代码较多。

(5) 译码电路复杂。

(6) 访存模式多样。

(7) 参数、返回地址都使用寄存器进行保存。

(8) x86-64。

(9) MIPS。

(10) 广泛用于嵌入式系统。

(11) 已知某个体系结构使用 `add R1, R2, R3` 来完成加法运算。当要将数据从寄存器 `S` 移动至寄存器 `D` 时，需要使用 `add S, #ZR, D` 进行操作（`#ZR` 是一个恒为 0 的寄存器）。

(12) 已知某个体系结构提供了 `xlat` 指令，它以一个固定的寄存器 `A` 为基地址，以另一个固定的寄存器 `B` 为偏移量，在 `A` 对应的数组中取出下标为 `B` 的项的内容，放回寄存器 `A` 中。

8 部分参考答案

参考答案一

1. 根据小端法可知分别为 0x56 0x34 0x12 0x00、0x34 0x12 0x78 0x56。

2. 根据小端法，A 在内存中从高地址到低地址分别是 11 11 22 22，得到 P = 0x2222，同理 Q = 0x3333，结果为 0x5555。

3. 答案为 0x0303，分析方法和前一题一样。

4. 1 取 x = 1, y = -1 即不正确；2 取 x = -1 即不正确；3 正确，利用异或的交换律、结合律，以及 x ^ x == 0（视为模 2 加法即可）；4 正确，即使是对负数；5 不正确，负奇数该运算向 0 舍入；6 正确，(~x) - y 也就是 (~x) + (~y) + 1，注意运算优先级；7 不正确，!!ux 是有符号数。

5. sizeof 返回的结果是无符号数，因此循环条件恒不成立，不会执行循环体。

6. 这是基本练习，填完的表格如下：

描述	二进制表示	M（写成分数）	E	f
负零	10000000	/	/	-0.0
/	01000101	21/16	1	1/8
最小的非规格化负数	10001111	15/16	-2	-15/64
最大的规格化正数	01101111	31/16	3	31/2
一	00110000	1	0	1.0
/	01010110	11/8	2	5.5
$+\infty$	01110000	/	/	/

7. 根据无穷大的定义，类 IEEE 754 的浮点数都只有 1 个无穷大。一般阶码越多，则最大绝对值越大，所以 B 大；A 大，理由同前；小数长度越多，则 NaN 越多，而总的可能表示的模式数一样，所以 B 更多。

8. 1 正确；2 取 `a = INF, b = FLT_MAX`；3 取 `a = 16777220, b = 16777218`；4 取最大的非规格化负数；5 正确；6 `nan != nan`；7 正确。

9. $16777217 = 2^{24} + 1$，这时写成科学计数法小数点后面要 24 位，超出了单精度浮点的表示范围，会发生舍入。

10. 写出 $2^{25} + 34$ 的科学计数法表示后按要求舍入即可，这些数舍入到单精度浮点数的结果计算如下。我们用下划线表示单精度浮点数小数表示的极限位置，后面的需要进行舍入，根据四舍六入五成双规则即可得到答案（注意最后一位相当于 1，倒数第二位相当于 2）。

1. `1.0000000000000000000001000_10 * 2^25`，差 2。
2. `1.0000000000000000000001000_11 * 2^25`，差 −1。
3. `1.0000000000000000000001001_00 * 2^25`，差 0。
4. `1.0000000000000000000001001_01 * 2^25`，差 1。
5. `1.0000000000000000000001001_10 * 2^25`，差 −2
6. `1.0000000000000000000001011_11 * 2^25`，差 −1。
7. `1.0000000000000000000001010_00 * 2^25`，差 0。
8. `1.0000000000000000000001010_01 * 2^25`，差 1。

参考答案二

1. B。

5. C，连接词的完备集。

10. A。计算过程：浮点数阶码 8 位，即最大指数为 127，尾数为 23 位，即最低位表示的数为 $2^{-23}2^{127} = 2^{104}$。那么误差在刚好落在最低位的一半的时候最大，即 2^{103}。

11. D。计算 $0.5 \leqslant \left|\frac{5}{2^e} - 1\right| < 1 \implies e = 4, \mathrm{EXP} = 7 = \texttt{111}_2$，注意此时 M 为负数，$|M| = \texttt{0.01010}_2$，用补码表示得 $\texttt{1.10101}_2$。

13. B。此题和 C90 标准有关，分析参看 https://stackoverflow.com/questions/9941261/warning-this-decimal-constant-is-unsigned-only-in-iso-c90。

15. D，注意 C 可能是有符号整数。

18. B，可以直接计算，注意不同运算的舍入规则。

23. 分别为 D、D、BD、E、AD、E、C、C。

参考答案四

1. 1 不正确；2 不正确，`$` 只用来表示立即数；3 正确，是内存地址 `0x30`；4 正确；5 不正确，缩放比例只能是 1、2、4、8；6 正确；7 不正确，x86-64 不允许将除了 64 位寄存器以外的寄存器作为寻址模式基地址；8 不正确，`%rsp` 不能作为操作数！

2. 注意 `movl` 会清零高 32 位。如果 `movabsq` 的立即数是负数，主要要取绝对值（本题中不需要）。答案为：

1. `0x0123456789ABCDEF`, `0x0000000000000000`
2. `0x0123456789ABCDEF`, `0x000000000000CDEF`
3. `0x0123456789ABCDEF`, `0xFFFFFFFFFFFFCDEF`
4. `0x00000000FFFFCDEF`, `0xFFFFFFFFFFFFCDEF`
5. `0x0123456789ABCDEF`, `0xFFFFFFFFFFFFCDEF`
6. `0xFFFFFFFF89ABCDEF`, `0xFFFFFFFFFFFFCDEF`

3. C。A、B 中，`movzbl/movzwl` 都生成了四字节，把高位设为 0。D 中 `cltq` 是对 `%eax` 的符号拓展。而 C 中 `movl` 和 `movzlq` 等价。

4. 1 不正确，目的不能是立即数；2 正确；3 不正确，两个操作数不能同时是内存地址；4 正确；5 不正确，要用 `movabsq`；6 正确；7 不正确，`movabsq` 的目标地址必须是整数寄存器；8 不正确，不能用 `mov` 向 `%rip` 中传入数据，否则读程序计数器产生破坏。

5. C，注意前两个选项只是计算了地址，而 `int` 每个元素 4 字节，D 错误。

6. 如下所示:

```c
long func(long a, long b) {
    return a * 15 + b * 7;
}
```

7. A。如果 `a > b`，那么 `a - b` 或者为正，或者发生上溢变负；同时 `a` 不等于 `b`。

8. 如下所示（快速幂）:

```c
long func(long a, long b) {
    long ans = 1;
    while (b > 0) {
        if (b & 1)
            ans = ans * a;
        b = b >> 1;
        a = a * a;
    }
    return ans;
}
```

9. 1、4 会被编译成条件传送。1 由于比较前计算出的 a 与 b 就是传送的目标，因此会被编译成条件传送；2 由于比较结果会导致 a 与 b 指向的元素发生不同的改变，因此会被编译成条件跳转；3 由于指针 a 可能无效，因此会被编译为条件跳转；4 会被编译成条件传送，因为 a 和 b 都是局部变量，返回的时候对 a 和 b 的操作都是无影响的。

10. 03; f8。第五行跳转位置为 0xf8(-8) + 0x4004dd = 0x4004d5，注意进行跳转之前，PC 指向该指令的下一条指令。

11. 做法是首先根据跳转表, 确定各标号的起始地址, 然后再作汇编代码的对应. 答案如下所示：

```
case 0:
case 1:
    c = c - 5;
case 2:
    res = 4 * c; // or res *= c
    break;
case 5:
    res = 86547; // or 0x15213
    break;
case 3:
    c = 2;
case 7:
    b = b & c;
default:
    res += b; // or res = b + 4
```

13. 阅读 callee 汇编代码可知，首先 *b 进入 %rax，随后 %rax 变为 *a ^ *b。根据 C 代码确定下一步要将结果转入 *a 中，所以汇编语言代码的空应填写 %rax, %(rdi)。caller 汇编代码的阅读不需要太仔细 (就做题而言)，极易填写。注意这里体现的典型的 for 循环结构，快速识别 %ebx 为循环变量，.L4 后为循环条件检查。后者需要计算 n / 2，要留意编译器选择用移位作除法，但是对于负数需要作修正 (舍入方式不同)。sarq 等移位操作若只有一个操作数，则移位量默认为 1。

此处栈空间的填写难度不大。首先，main 即将发生调用时，%rsp 值为 0xf...f80，接下来 call 后需要在栈中设置返回地址，这个返回地址为 0x4000ac + 0x5 = 0x4000b1。所以栈的第三个空填此。注意栈是向下增长的，所以前两个空不确定。接下来 caller 要保存 %r12, %rbp, %rbx 三个寄存器，因此先后填入 0x213, 0x18, 0x15。最后，callee 被调用，需要再次填入返回地址 0x400088 + 0x5 = 0x40008d。最后一个空显然不确定。

14. 解题时不妨用框图的形式把整个结构的内存布局画出来并标好偏移量。CC1 从位置 0 开始；整数 4 字节对齐，所以 II1 从位置 8 开始；长整数 8 字节对齐，所以 LL1 必须从位置 16 开始；而后的字符数组 CC2 延展到位置 34，仍由长整数 8 字节对齐，LL2 只好从位置 40 开始；最后的整数从位置 48 开始，现在总大小为 52 字节。在结构体后面再放一个相同的 A，发现需要再补 4 字节才能使得第二个 A 的 LL1 对齐，所以第一问为 56 字节。第二问：可以重排顺序为 LL1 LL2 II1 II2 CC1 CC2，刚好没有空白空间，得到的大小为 40 字节。显然这个可以用

贪心法来做（先放对齐要求高的元素）。

15. 本题的主要难点是确定 `union`，`struct` 本身的对齐，方法一般是在结构或联合后面再放一个相同的对象，构成数组，看需要补多少字节才能使得第二个元素对齐（对齐要求的是每个基本数据类型对齐！）。经验规则是按其中对齐要求最严格的元素进行这一工作。

根据对齐规则，题给 `union` 只需要 7 字节就能使得内部对齐，但是后面接一个同样的对象时 `short` 需要 2 字节对齐，所以它实际上为 8 字节，且要求 2 字节对齐。这样我们就知道 `struct` 的大小为 $3+\underline{1}+8+4=16$ 字节。由此，前 4 空答案为 4、8、12、16。同理可以得到后面几问的答案，为 14、8、8、16、(3、7、12、16)。

16. 这是典型的汇编综合题，需要同时结合 C 代码和汇编代码来观察。

第一步先通读 C 代码，确定代码大意，便于在汇编中找一些对应的片段。然后来填写汇编代码。主要难点在厘清 C 语言变量，内存和寄存器的映射关系，确定代码的用途。先看主函数，前三行为金丝雀。接下来，`0x69`，`0xfc` 分别进入 `%rsp`，`%rsp + 8` 对应的内存位置中。根据主函数唯一的 `&np` 取局部变量地址，以及紧接着 `%rsp` 被作为参数传入 `func`，知道这就是 `np` 结构体的初始化工作。所以 C 代码的第 10、11 空分别填写 105、252。

接着，`func` 的返回值被存入 `%rsi`, 一个立即数（看上去像地址）被存入 `%rdi`，然后调用 `printf` 函数。结合 C 代码知道 `%rdi` 是字符串 `"%ld"`（属于 `char*` 类型）的地址。所以由给出的 ASCII 码表计算得到问答题 (1) 应填写 `0x25`，`0x6c`，`0x64`，`0x00`。特别留意字符串的最后有一个 `0x00` 表示结尾！主函数的最后几个空是检查金丝雀是否被破坏，易知金丝雀的原值在 `%fs:0x28`，而在栈上对应于地址 `%rsp + 0x18`（这个之后还要用到）。检查金丝雀是否被破坏是容易的，`40063b` 先把后者放到 `%rcx` 中，所以要比较 `%fs:0x28`，`%rdx`，这就是空 (4) 的答案。接下来要条件跳转，因为如果条件成立会正常返回，所以空 (5) 写 `je`。空 (6) 是函数返回释放栈空间，因为一开始分配了 `0x28` 个字节，所以这个空也要填 `0x28`。

现在分析 `func` 函数。前三行是金丝雀以及寄存器清零。从地址 `4005aa` 开始分析，首先 `p->a`，`p->b` 分别进入 `%rax`，`%rdx` 中。接下来比较 `p->a >= p->b`。结合 C 代码，如果这成立，就不需要经过交换 `p->a`，`p->b` 的过程。立刻看出，`4005b6`，`4005b9` 两行完成了交换工作（没有出现 `temp` 的对应寄存器映射）。由于不确定交换之后 `%rax` 到底对应 `p->a`，`p->b` 中的哪个，所以其要重新移入 `p->b` 到 `p->a` 中。所以 `4005bd` 就是空 (1) 的跳转目标。接下来用 `test` 指令判断 `%rax` 是否为 0。注意 C 代码，如果某个条件成立，就要 `return p->a`。因为 `4005cb` 把 `p->a` 放到 `%rax` 中，所以空 (2) 会跳到函数结尾返回，也就是说跳转目标是 `4005e2`（返回之前要检查金丝雀是否被破坏）。由此也可知只有 `p->b` 为零时才返回，空 (7) 填写 `p->b == 0`。

紧接着处理 `func` 的递归调用。结合 C 代码知道要生成新的 `np`，而且 `np.a`，`np.b` 分别在栈地址的 `%rsp`，`%rsp + 8` 处。因为这时 `p->a` 在 `%rdx` 中，`p->b` 在 `%rax` 中，所以由 `4005ce`，`4005d1` 两行立刻知道 (8)、(9) 两空分别写 `p->a - p->b`，`p->b`。

补全 C 代码后，可以看出这是辗转相除法，输入是 252、105，它们的最大公约数是 21，故问答题的 (4) 回答 21。

最后填写栈。对于递归调用，我们应该先正确划分各次调用的栈的边界。第一步，由于主函数和 `func` 一开始都 `subq 0x28, %rsp`，所以栈空间都有 40 字节或 5 个 64 位地址。注意 6 号位置是 `0x400629`，恰好是主函数调用 `func` 的返回地址，所以我们知道 1–5 号位置是主

函数的栈帧，6 是从 `func` 返回主函数的返回地址；7–11 是第一次调用 `func` 的栈帧，12 是从第二次递归调用返回第一次调用 `func` 的返回地址，13–17 是第二次调用 `func` 的栈帧，以此类推。所以 12、18 都填 `0x4005e2`（`func` 中递归调用的下一条指令的位置）。

第二步，根据前面的分析，每个栈的栈帧的底部向上数第一个和第二个数都是存储局部变量 `np`（无论是主函数还是 `func`），所以位置 4、5 似乎分别应该写上 `0xfc, 0x69`。但是，注意后面的递归调用中，这两个位置的数会被交换以保证 `p->a >= p->b`！所以这两个空要倒过来，4、5 分别写上 `0x69, 0xfc`。第三步，由前面的分析，金丝雀值一直存在 `%rsp + 0x18`，也就是栈帧自底向上第四个位置，又题目条件说 `%fs` 寄存器没有改变，所以所有的金丝雀值都是 `0xc76d5add7bbeaa00`，这样 8、14、20 都是这个值。

对于栈的 10、11 号位置，前面的分析知道它们应该是 `252 - 105, 105 = 0x93, 0x69`（顺序未定），而交换后保证上一个要小于下一个，所以 10、11 给出的情况印证我们的分析。同理，16、17 空是 `0x93 - 0x69 = 0x2a, 0x69`，上一个要小于下一个。虽然 22、23 号位置不用填写，但是我们务必注意，栈在这个状态时，刚刚进行下一次递归调用，这两个数还没有被下一次递归调用修改顺序（尽管实际上不需要修改顺序）。

栈的剩下位置，7、21 都是不确定的，因为只是分配了没有修改过。（注意栈帧开头应该不是旧的帧指针，因为如果要这样需要显式 `push`。）

参考答案五

3. 此题有错，A、B 都不准确，有一些 SIMD 指令并不会改变条件码；C 明显是错的；D 基本正确，确实有指令可以直接写条件码。

4. 此题有错，B、D 都不对，其中 B 不能用 32 位寄存器寻址，D 中少了一个逗号。

9. B，这个条件相当于 `(a == 0 || a == 1 || a == -1)`。

14. C，从 `subl` 可看出待比较的数是字符型。

17. D。题目没有说明大小端，如果是大端法则为 A。若为小端法，注意 `char` 在不同的编译器上可能是有符号型或者是无符号型，所以计算知 –16 和 240 都有可能。

19. 此题有错，A、D 都可能产生（和优化选项有关）。

参考答案六

参考答案七