

复旦大学计算机科学技术学院  
2020-2021 学年第一学期期中考试试卷

课程名称: 计算机系统基础 课程代码: COMP130156.01/COMP130143.02

开课院系: 计算机科学技术学院 考试形式: 开卷

姓名: \_\_\_\_\_ 学号: \_\_\_\_\_ 专业: \_\_\_\_\_

**提示:** 请同学们秉持诚实守信宗旨, 谨守考试纪律, 摒弃考试作弊。学生如有违反学校考试纪律的行为, 学校将按《复旦大学学生纪律处分条例》规定予以严肃处理。

题号	1	2	3	4	5	6	7	8	总分
得分									

### 1. (16 分)

Assume we are running code on a 6-bit machine using two's complement arithmetic for signed integers. A "short" integer is encoded using 3 bits. Fill in the empty boxes in the table below. The following definitions are used in the table:

```
short sy = -3;
int y = sy;
int x = -17;
unsigned ux = x;
```

Note: You need not fill in entries marked with “—”.

Expression	Decimal Representation	Binary Representation
Zero	0	
-	-6	
-		01 0010
ux		
y		
x>>1		

TMax		
-TMin		
TMin+TMin		

### 2. (18 分)

Consider the following 8-bit floating point representation based on the IEEE floating point format:

- There is a sign bit in the most significant bit.
- The next 3 bits are the exponent. The exponent bias is  $2^{3-1} - 1 = 3$ .
- The last 4 bits are the fraction.
- The representation encodes numbers of the form:  $V = (-1)^s \cdot M \cdot 2^E$ , where  $M$  is the significand and  $E$  is the biased exponent.

The rules are like those in the IEEE standard (normalized, denormalized, representation of 0, infinity, and NAN). FILL in the table below. Here are the instructions for each field:

- **Binary:** The 8 bit binary representation.
- **M:** The value of the significand. This should be a number of the form  $x$  or  $x/y$ , where  $x$  is an integer, and  $y$  is an integral power of 2. Examples include 0,  $3/4$ .
- **E:** The integer value of the exponent.
- **Value:** The numeric value represented.

Note: you need not fill in entries marked with “—”.

Description	Binary	M	E	Value
Minus zero				-0.0
—	0 100 0101			
Smallest denormalized (negative)				
Largest normalized (positive)				
One				1.0
—				5.5
Positive infinity		—	—	+∞

### 3. (6 分)

Consider the following C functions and assembly code:

```
int fun7(int a)
{
    return a * 30;
}

int fun8(int a)
{
    return a * 34;
}

int fun9(int a)
{
    return a * 18;
}
```

Which of the functions compiled into the assembly code shown?

### 4. (6 分)

Consider the following C functions and assembly code:

```
int fun4(int *ap, int *bp)
{
    int a = *ap;
    int b = *bp;
    return a+b;
}

int fun5(int *ap, int *bp)
{
    int b = *bp;
    *bp += *ap;
    return b;
}

int fun6(int *ap, int *bp)
{
    int a = *ap;
    *bp += *ap;
    return a;
}
```

Which of the functions compiled into the assembly code shown?

### 5. (10 分)

Consider the source code below, where M and N are constants declared with #define.

```
int array1[M][N];
int array2[N][M];

int copy(int i, int j)
{
    array1[i][j] = array2[j][i];
}
```

Suppose the above code generates the following assembly code:

```
copy:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
    movl 8(%ebp),%ecx
    movl 12(%ebp),%ebx
    leal (%ecx,%ecx,8),%edx
    sall $2,%edx
    movl %ebx,%eax
    sall $4,%eax
    subl %ebx,%eax
    sall $2,%eax
    movl array2(%eax,%ecx,4),%eax
    movl %eax,array1(%edx,%ebx,4)
    popl %ebx
    movl %ebp,%esp
    popl %ebp
    ret
```

What are the values of M and N?

M =

N =

### 6. (10 分)

Consider the following assembly representation of a function foo containing a for loop:

```
foo:  
    pushl %ebp  
    movl %esp,%ebp  
    pushl %ebx  
    movl 8(%ebp),%ebx  
    leal 2(%ebx),%edx  
    xorl %ecx,%ecx  
    cmpl %ebx,%ecx  
    jge .L4  
.L6:  
    leal 5(%ecx,%edx),%edx  
    leal 3(%ecx),%eax  
    imull %eax,%edx  
    incl %ecx  
    cmpl %ebx,%ecx  
    jl .L6  
.L4:  
    movl %edx,%eax  
    popl %ebx  
    movl %ebp,%esp  
    popl %ebp  
    ret
```

Fill in the blanks to provide the functionality of the loop:

```
int foo(int a)  
{  
  
    int i;  
    int result = _____;  
  
    for( _____; _____; i++ ) {  
  
        _____;  
    }  
    _____;  
  
    return result;  
}
```

### 7. (16 分)

Consider the following C declarations:

```
typedef struct {  
    short code;  
    int start;  
    char raw[3];  
    double data;  
} OldSensorData;  
  
typedef struct {  
    short code;  
    short start;  
    char raw[5];  
    short sense;  
    short ext;  
    double data;  
} NewSensorData;
```

- A. Using the templates below (allowing a maximum of 24 bytes), indicate the allocation of data for structs of type OldSensorDataNewSensorData. Mark off and label the areas for each individual element (arrays may be labeled as a single element). Cross hatch the parts that are allocated, but not used (to satisfy alignment).

Assume the Linux alignment rules discussed in class. Clearly indicate the right hand boundary of the data structure with a vertical line.

OldSensorData:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
+---+																							

NewSensorData:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
+---+																							

B. Now consider the following C code fragment:

```
void foo(OldSensorData *oldData)
{
    NewSensorData *newData;

    /* this zeros out all the space allocated for oldData */
    bzero((void *)oldData, sizeof(oldData));

    oldData->code = 0x104f;
    oldData->start = 0x80501ab8;
    oldData->raw[0] = 0xe1;
    oldData->raw[1] = 0xe2;
    oldData->raw[2] = 0x8f;
    oldData->raw[-5] = 0xff;
    oldData->data = 1.5;

    newData = (NewSensorData *) oldData;

    ...
}
```

Once this code has run, we begin to access the elements of newData. Below, give the value of each element of newData that is listed. Assume that this code is run on a Little-Endian machine such as a Linux/x86 machine. You must give your answer in hexadecimal format. Be careful about byte ordering!

- (a) newData->start = 0x \_\_\_\_\_
- (b) newData->raw[0] = 0x \_\_\_\_\_
- (c) newData->raw[2] = 0x \_\_\_\_\_
- (d) newData->raw[4] = 0x \_\_\_\_\_
- (e) newData->sense = 0x \_\_\_\_\_

### Problem 8. (18 分):

The problem concerns the following C code. This program reads a string on standard input and prints an integer in hexadecimal format based on the input string it read.

```
#include <stdio.h>

/* Read a string from stdin into buf */
int evil_read_string()
{
    int buf[2];
    scanf("%s",buf);
    return buf[1];
}

int main()
{
    printf("0x%x\n", evil_read_string());
}
```

Here is the corresponding machine code on a Linux/x86 machine:

```
08048414 <evil_read_string>:
08048414: 55                      push  %ebp
08048415: 89 e5                  mov   %esp,%ebp
08048417: 83 ec 14                sub   $0x14,%esp
0804841a: 53                      push  %ebx
0804841b: 83 c4 f8                add   $0xffffffff8,%esp
0804841e: 8d 5d f8                lea   0xffffffff8(%ebp),%ebx
08048421: 53                      push  %ebx
08048422: 68 b8 84 04 08          push  $0x80484b8      address arg for scanf
08048427: e8 e0 fe ff ff          push  $0x4(%ebx),%eax
0804842c: 8b 43 04                mov   0x4(%ebx),%eax
0804842f: 8b 5d e8                mov   0xffffffe8(%ebp),%ebx
08048432: 89 ec                  mov   %ebp,%esp
08048434: 5d                      pop   %ebp
08048435: c3                      ret

08048438 <main>:
08048438: 55                      push  %ebp
08048439: 89 e5                  mov   %esp,%ebp
0804843b: 83 ec 08                sub   $0x8,%esp
0804843e: 83 c4 f8                add   $0xffffffff8,%esp
08048441: e8 ce ff ff ff          call  8048414 <evil_read_string>
08048446: 50                      push  %eax
08048447: 68 bb 84 04 08          push  $0x80484bb      integer arg for printf
0804844c: e8 eb fe ff ff          push  $0x4(%eax),%eax
08048451: 89 ec                  mov   %ebp,%esp
08048453: 5d                      pop   %ebp
08048454: c3                      ret
```

This problem tests your understanding of the stack discipline and byte ordering. Here are some notes to help you work the problem:

- `scanf("%s", buf)` reads an input string from the standard input stream (`stdin`) and stores it at address `buf` (including the terminating '\0' character). It does not check the size of the destination buffer.
- `printf("0x%lx", i)` prints the integer `i` in hexadecimal format preceded by "0x".
- Recall that Linux/x86 machines are Little Endian.
- You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'd'	0x64	'v'	0x76
'r'	0x72	'i'	0x69
'.'	0x2e	'l'	0x6c
'e'	0x65	'\0'	0x00
		's'	0x73

A. Suppose we run this program on a Linux/x86 machine, and give it the string "dr.evil" as input on `stdin`.

Here is a template for the stack, showing the locations of `buf[0]` and `buf[1]`. Fill in the value of `buf[1]` (in hexadecimal) and indicate where `ebp` points just after `scanf` returns to `evilreadstring`.

|<- buf[0]->|<-buf[1] ->|  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

What is the 4-byte integer (in hex) printed by the `printf` inside `main`?

0x\_\_\_\_\_

B. Suppose now we give it the input "dr.evil.lives" (again on a Linux/x86 machine).

- (a) List the contents of the following memory locations just after `scanf` returns to `evilreadstring`.  
Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

buf[0] = 0x\_\_\_\_\_

buf[3] = 0x\_\_\_\_\_

(b) Immediately before the `ret` instruction at address 0x08048435 executes, what is the value of the frame pointer register `ebp`?

ebp = 0x\_\_\_\_\_

You can use the following template of the stack as *scratch space*. Note: this does not have to be filled out to receive full credit.

<- buf[0] ->|<- buf[1] ->|  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+  
| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |  
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+