Problem 1

Please read the following code and answer the following questions.

```
#include <stdio.h>

typedef unsigned char *byte_pointer;
void show_bytes(byte_pointer start, int len)
{
    for (int i = 0; i < len; i++)
        printf("0x%.2x ", start[i]);
    printf("\n");
}

struct s {
    char *p;
    short v;
    char arr[3];
    int a;
};

int main(void)
{
    int i, x = 0x1234567;
    char *charp, one[8];
    char two[4][4];

    for (i = 0; i < 8; i++)
        one[i] = i;

    for (i = 0, charp = two; i < sizeof(two); i++)
        charp[i] = i;

    printf("x: ");
    show_bytes((byte_pointer)&x, 2);

    int *ip = (int *)(one + 5);
    ip[-1] = 0x11;
    printf("new one: ");
    show_bytes((byte_pointer)one, 8);

    void *vp = (void *)(two + 2) + 2;
    short *sp = (short *)vp;
    *sp = 0xff;
    printf("new two: ");
    show_bytes((byte_pointer)two, 16);

    struct s *ss = (struct s *)two;
    vp = &(ss->arr[ss->arr[0]]);
    printf("new s: ");
    show_bytes((byte_pointer)vp, 3);

    return 0;
}
```

Suppose the following code is executed on a 64-bit little-endian machine.

1. What is the size of **`struct s`**?

   24

2. Please fill in the blanks about the output of this program.

```
x: 0x67 0x45
new one: 0x00 0x11 0x00 0x00 0x00 0x05 0x06 0x07
new two: 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0xff 0x00 0x0c 0x0d
0x0e 0x0f
new s: 0x09 0xff 0x00
```

Problem 2

Suppose the following code is executed on a 64-bit machine.

```
struct data {
    unsigned char *p;
    int i;
    short s[3];
    union {
        char j;
        int k;
    } u;
    char c;
};

struct data d[2];
```

Suppose the address of global variable **d** is **0x555555755040**, please answer the following questions.

Variable	Start address
d[0]	0x555555755040
d[1]	0x555555755060
d[0].p	0x555555755040
d[0].i	0x555555755048
d[0].s[1]	0x55555575504e
d[0].u.j	0x555555755054
d[0].u.k	0x555555755054
d[0].c	0x555555755058

Problem 3

Here is a snippet of a C program written by one of the TAs of ICS. Some lines in function **blocks_sum** and the definition of **struct block** are hided. (64-bits little endian machine).

```
unsigned long blocks_sum(struct block *block_array, int array_size){
        int i;
        unsigned long sum = 0;
        struct block *bp;

        for (i = 0; i < array_size; i++) {
                bp = block_array + i;
                if (bp->type == 0)
                        /* Hided code 1*/
                else
                        /* Hided code 2*/
        }

        return sum;
}
```

Compiling this codes with -O1 GCC option (-O means compiler optimization level) yields the assembly below:

```
 1  blocks_sum:
 2          testl   %esi, %esi
 3          jle     .L6
 4          movq    %rdi, %rdx
 5          leal    -1(%rsi), %eax
 6          leaq    (%rax,%rax,2), %rax
 7          leaq    24(%rdi,%rax,8), %rdi
 8          movl    $0, %eax
 9          jmp     .L5
10  .L3:
11          movsbl  16(%rdx), %ecx
12          movl    8(%rdx), %esi
13          sarl    %cl, %esi
14          addq    %rsi, %rax
15  .L4:
16          addq    $24, %rdx
```

```
17          cmpq    %rdi, %rdx
18          je      .L8
19  .L5:
20          cmpb    $0, (%rdx)
21          jne     .L3
22          movq    8(%rdx), %rsi
23          movsbl  16(%rdx), %ecx
24          movl    (%rsi), %esi
25          shrl    %cl, %esi
26          movl    %esi, %esi
27          addq    %rsi, %rax
28          jmp     .L4
29  .L8:
30          rep ret
31  .L6:
32          movl    $0, %eax
33          ret
```

1. FOR-LOOP: In this level of optimization, GCC translates the for-loop in a different way from what we express in C. Please answer:
   a) What line range in the assembly is the translation of "Hided code 1"?
      Line 22 ~ 27
   b) What line range in the assembly is the translation of "Hided code 2"?
      Line 11 ~ 14
2. STRUCT and UNION
   a) What is the size of **`struct block`**?
      24
   b) Write a possible definition of **`struct block`** according to the C, assembly codes above.

```
struct block {
       char type; (must be of size 1)
       union {
              unsigned int *p; (must be a pointer of an 4-byte unsigned type)
              int v; (must be a signed type of size 4)
       };
       char shift_size; (must be an signed type of size 1)
};
```

Problem 4

One of the students in ICS course writes a toy program as follows.

```
int lock_flag = 0;

void lock() {
    while (__sync_lock_test_and_set(&lock_flag, 1)) {}
}

void unlock() {
    lock_flag = 0;
}

struct account {
     union {
          char name[3];
          short id;
     } u;
     int balance;
};

void transfer(struct account *a, struct account *b, int amount)
{
     lock();
     a->balance -= amount;
     b->balance += amount;
     unlock();
}
```

In this code, the GCC built-in function:

```
type __sync_lock_test_and_set (type *ptr, type value, ...)
```

is an atomic exchange operation. It writes value into ***ptr**, and returns the previous contents of ***ptr**. And here is a runtime disassembly of these functions.

```
lock:
   0x000055555555464a    push   %rbp
   0x000055555555464b    mov    %rsp,%rbp
   0x000055555555464e    nop
   0x000055555555464f    mov    $0x1,%eax
   0x0000555555554654    xchg   %eax,0x2009ba(%rip)
   0x000055555555465a    test   %eax,%eax
   0x000055555555465c    jne    0x55555555464f <lock+5>
   0x000055555555465e    nop
   0x000055555555465f    pop    %rbp
   0x0000555555554660    retq
unlock:
   0x0000555555554661    push   %rbp
   0x0000555555554662    mov    %rsp,%rbp
   0x0000555555554665    movl   $0x0,[1]0x2009a5(%rip)
   0x000055555555466f    nop
   0x0000555555554670    pop    %rbp
   0x0000555555554671    retq
transfer:
   0x0000555555554672    push   %rbp
   0x0000555555554673    mov    %rsp,%rbp
   0x0000555555554676    sub    $0x18,%rsp
   0x000055555555467a    mov    %rdi,-0x8(%rbp)
   0x000055555555467e    mov    %rsi,-0x10(%rbp)
   0x0000555555554682    mov    %edx,-0x14(%rbp)
   0x0000555555554685    mov    $0x0,%eax
   0x000055555555468a    callq  0x55555555464a <lock>
   0x000055555555468f    mov    -0x8(%rbp),%rax
   0x0000555555554693    [2]mov    0x4(%rax),%eax
   0x0000555555554696    sub    -0x14(%rbp),%eax
   0x0000555555554699    mov    %eax,%edx
   0x000055555555469b    mov    -0x8(%rbp),%rax
   0x000055555555469f    mov    %edx,0x4(%rax)
   0x00005555555546a2    mov    -0x10(%rbp),%rax
   0x00005555555546a6    mov    0x4(%rax),%edx
   0x00005555555546a9    mov    -0x14(%rbp),%eax
   0x00005555555546ac    [3]add    %eax,%edx
   0x00005555555546ae    mov    -0x10(%rbp),%rax
   0x00005555555546b2    mov    %edx,0x4(%rax)
   0x00005555555546b5    mov    $0x0,%eax
   0x00005555555546ba    callq  0x555555554661 <unlock>
   0x00005555555546bf    nop
   0x00005555555546c0    [4]leaveq
   0x00005555555546c1    retq
```

1. Please fill in the blanks above.
2. Assume that just **AFTER** the execution of **`push %rbp (0x0000555555554672)`**, the value of **rbp**, **rsp** is as follows.

register	value
rbp	**0x7fffffffe0e0**
rsp	**0x7fffffffe0b0**

Please fill in the blanks below.

Case	rbp value	rsp value
Before 0x000055555555464a	**0x7fffffffe0b0**	**0x7fffffffe090**
After 0x0000555555554696	**0x7fffffffe0b0**	**0x7fffffffe098**
After 0x0000555555554670	**0x7fffffffe0b0**	**0x7fffffffe090**