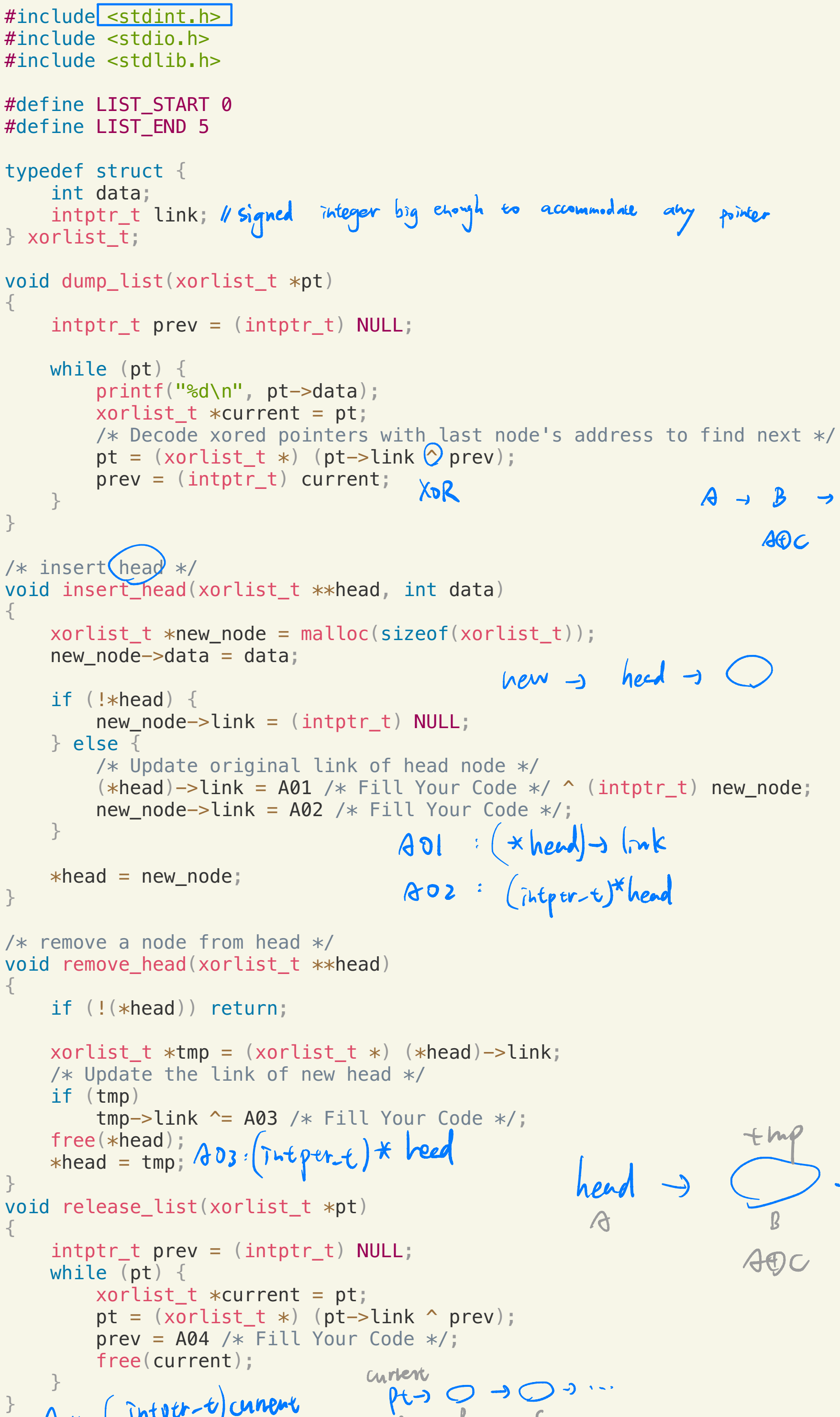
A.



```c
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#define LIST_START 0
#define LIST_END 5

typedef struct {
    int data;
    intptr_t link; // signed integer big enough to accommodate any pointer
} xorlist_t;

void dump_list(xorlist_t *pt)
{
    intptr_t prev = (intptr_t) NULL;

    while (pt) {
        printf("%d\n", pt->data);
        xorlist_t *current = pt;
        /* Decode xored pointers with last node's address to find next */
        pt = (xorlist_t *) (pt->link ^ prev);
        prev = (intptr_t) current;
    }
}

/* insert head */
void insert_head(xorlist_t **head, int data)
{
    xorlist_t *new_node = malloc(sizeof(xorlist_t));
    new_node->data = data;

    if (!*head) {
        new_node->link = (intptr_t) NULL;
    } else {
        /* Update original link of head node */
        (*head)->link = A01 /* Fill Your Code */ ^ (intptr_t) new_node;
        new_node->link = A02 /* Fill Your Code */;
    }

    *head = new_node;
}

/* remove a node from head */
void remove_head(xorlist_t **head)
{
    if (!(*head)) return;

    xorlist_t *tmp = (xorlist_t *) (*head)->link;
    /* Update the link of new head */
    if (tmp)
        tmp->link ^= A03 /* Fill Your Code */;
    free(*head);
    *head = tmp;
}
void release_list(xorlist_t *pt)
{
    intptr_t prev = (intptr_t) NULL;
    while (pt) {
        xorlist_t *current = pt;
        pt = (xorlist_t *) (pt->link ^ prev);
        prev = A04 /* Fill Your Code */;
        free(current);
    }
}
```

XoR

A → B → C

A⊕C   B⊕D

new → head → ○

A01 : (*head)→ link

A02 : (intptr_t)*head

A03 : (intptr_t) * head

tmp

head → ○ → ○ ...

A   B   C

A⊕C

A04: (intptr_t) current

current

pt→ ○ → ○ → ...

A   B   C

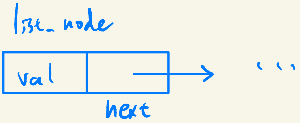
A⊕C

```
int main()
{
    xorlist_t *head = malloc(sizeof(xorlist_t)), *tail;
    xorlist_t *pt = head;
    intptr_t last_node = (intptr_t) NULL;
    for (int c = LIST_START; c < LIST_END; ++c) {
        xorlist_t *new_node = malloc(sizeof(xorlist_t));
        *pt = (xorlist_t){.data = c, .link = (intptr_t) new_node ^ last_node};
        last_node = (intptr_t) pt;
        pt = new_node;
    }
    *pt = (xorlist_t){.data = LIST_END, .link = last_node ^ (intptr_t) NULL};
    tail = pt;

    insert_head(&head, 99);
    dump_list(head);

    remove_head(&tail);
    dump_list(tail);

    release_list(head);

    return 0;
}
```

B.

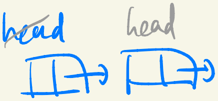
?

```
#include <assert.h>
#include <stdlib.h>

struct list_node {
    int val;
    struct list_node *next;
};
```



```
struct list_node *reverse(B01 /* Fill Your Code */)
{
    struct list_node *next = NULL, *ret;
    while (*head) {
        ret = malloc(sizeof(struct list_node));
        ret->val = B02 /* Fill Your Code */;
        ret->next = B03 /* Fill Your Code */;
        next = B04 /* /* Fill Your Code */;
        *head = (*head)->next;
    }
    return ret;
}
```

B01: struct list_node ** head



B02: (*head)-> val

B03: next

B04: ret

```
/* Assume that NEW_LL_1234() properly malloc's a linked list
 * 1 -> 2 -> 3 -> 4, and returns a pointer that points to the first
 * list_node in the linked list. Assume that before test_reverse
 * returns, head and ret will be properly freed.
 */
void test_reverse()
{
    struct list_node *head = NEW_LL_1234();
    assert(head->val == 1);        /* returns True */
    assert(head->next->val == 2);  /* returns True */
    struct list_node *ret = reverse(&head);
    assert(head != ret);     /* ret is a new copy of the original list */
    assert(ret->val == 4);   /* should return True */
}
```

C.

ABCD → DCBA

```
/* Return a value in which the order of the bytes in 4-byte arguments is reversed.
*/
static inline uint32_t end_bswap32(uint32_t __x)
{
    return (__x >> 24) | (__x >> 8 & C01) | (__x << 8 & C02) |(__x << 24);
}
```

?

即 unsigned int

$0 \sim 2^{32} - 1 \equiv$ 0x00000000 ~ 0xFFFFFFFF

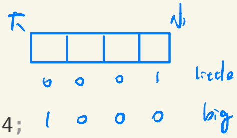
bit or

0x FF00

0x FF0000

```
/* Host to Big Endian 32-bit */
static inline uint32_t end_htobe32(uint32_t n)
{
    union {
        int i;
        char c;
    } u = {1};
    return u.c ? C03 : C04;
}
```

0001



end_bswap32(n) n

>> : 位元右移

<< : 位元左移

000A
00B0
0C00
or) D000

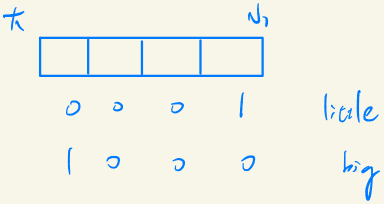
```
/* Host to Little Endian 32-Obit */
static inline uint32_t end_htole32(uint32_t n)
{
    union {
        int i;
        char c;
    } u = {1};
    return u.c ? C05 : C06;
}
```

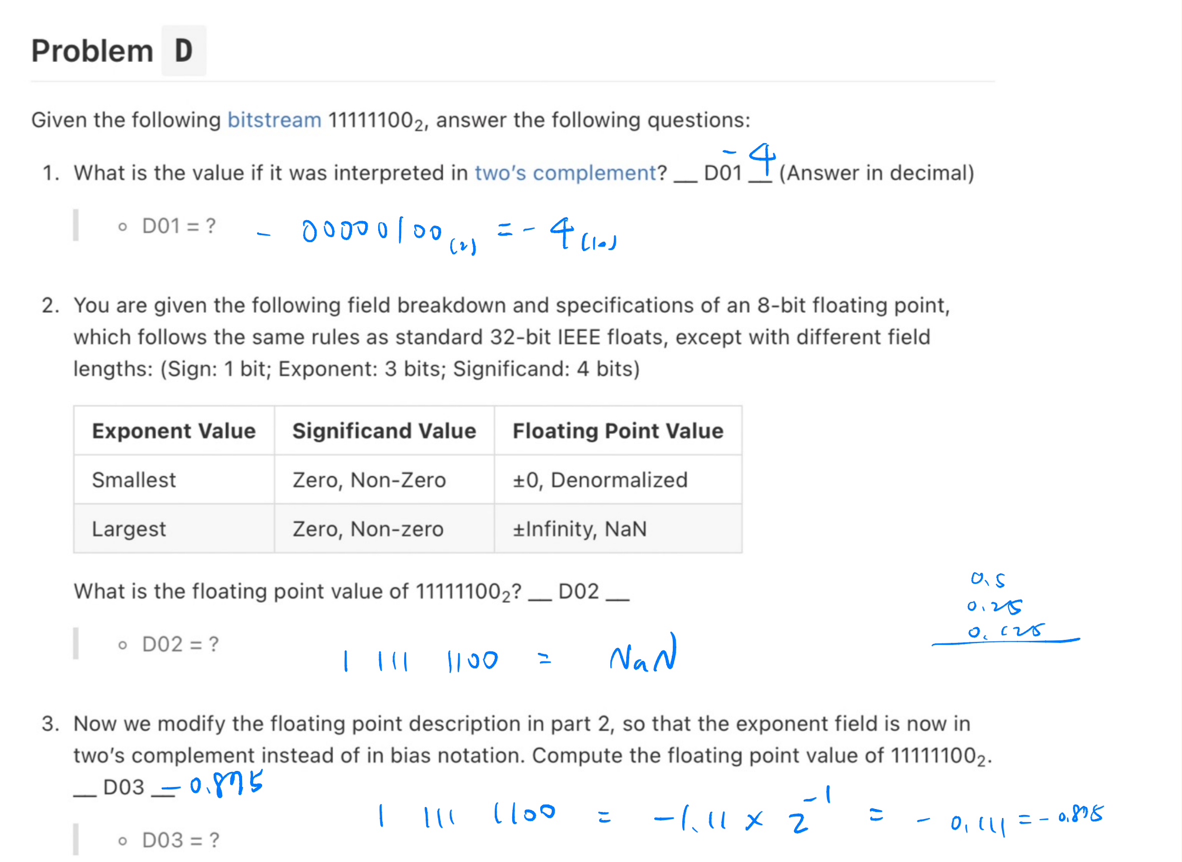
n   end_bswap32(n)

00 00 FF 00
and) ? A B C
B 0

00 FF 00 00
and) B C D 0
C 0 0

0001





Problem D

Given the following bitstream 11111100_2, answer the following questions:

1. What is the value if it was interpreted in two's complement? __ D01 __ -4 (Answer in decimal)

   > - D01 = ? $-\ 00000100_{(2)} = -4_{(10)}$

2. You are given the following field breakdown and specifications of an 8-bit floating point, which follows the same rules as standard 32-bit IEEE floats, except with different field lengths: (Sign: 1 bit; Exponent: 3 bits; Significand: 4 bits)

Exponent Value	Significand Value	Floating Point Value
Smallest	Zero, Non-Zero	±0, Denormalized
Largest	Zero, Non-zero	±Infinity, NaN

   What is the floating point value of 11111100_2? __ D02 __

   0.5
   0.25
   0.125

   > - D02 = ?   1 111 1100 = NaN

3. Now we modify the floating point description in part 2, so that the exponent field is now in two's complement instead of in bias notation. Compute the floating point value of 11111100_2. __ D03 __ -0.875

   > - D03 = ? $1\ 111\ 1100 = -1.11 \times 2^{-1} = -0.111 = -0.875$

E.

$(2-2^{-10}) \times 2^{12}$

You received a sequence of IEEE standard 16-bit floating point numbers from a trusted source. Note that a 16-bit floating point is 1 sign bit, 5 exponent bits, and 10 mantissa bits. The bias for the exponent is −15. Unfortunately, some of the data was corrupted during communication, rendering it unreadable. For the following problems, we will use x to refer to a bit that was corrupted (in other words, we don't know what the sender wanted that bit to be). For example, if I received the data `0b0xx1`, the sender sent one of `0b0001`, `0b0101`, `0b0011`, or `0b0111`.

1 You receive the data `0b1110xxxxxxxxxxxx`. What is the decimal value of the smallest number the sender could have sent (i.e. it is less than all of the other possibilities)? __ E01 __ -8188 You must provide the decimal form, do not leave as a power of 2.

- E01 = ?   1 110xx xx···x → 1 11011 11···1 $= -1.11\cdots1 \times 2^{12}$

2 You receive the data `0b0x1x0x1x0x1x0x1x`. What is the hexadecimal encoding of the biggest number the sender could have sent? __ E02 __

- E02 = ?   0x7777

$-1.11\cdots1 = -(2-2^{-10})$

0 x1x0x 1x0x1x0x 1x

→ 0 11101 1101110111

$-(2-2^{-10}) \cdot 2^{12} = -(2^{13}-2^{2})$

$= -\ 8188$

F.

1 8 23

F01 : 23

F02 : 0x 7FFFFF

```
static inline float u2f(unsigned u)
{
    return *(float *) &u;
}

int float_to_int(float_bits f)
{
    const unsigned sign = f >> 31;
    const unsigned exp = f >> F01 /* Fill Your Code */ & 0xFF;
    const unsigned frac = f & F02 /* Fill Your Code */;
    const unsigned bias = 0x7F; // 127
    int result;
    if (exp < bias) {
        /* the float number is less than 1 */
        result = 0;
    } else if (exp >= 31 + bias) {
        /* overflow */
        result = 0x80000000;
    } else {
        /* normal */
        unsigned E = exp - bias;
        unsigned M = frac | 0x800000;
        if (E > F03 /* Fill Your Code */) {
            result = F04 /* Fill Your Code */;
        } else {
            /* round to zero */
            result = F05 /* Fill Your Code */;
        }
    }
    return sign ? -result : result;
}
```

✓

F03: 23

F04: M << (E - 23)

F05: M >> (23 - E)

G.

G01 : 1

```
int is_positive(int x)
{
    const int mask = G01 /* Fill Your Code */;
    /* place x's MSB in the least bit
     * if x negative, tmp is 11111111 // hexadecimal
     * if positive/0, tmp is 00000000
     */
    int tmp = x >> 31;        x = |||...|
                                   32bits
    /* keep just the least bit by AND to mask(00000001)
     * if x negative, tmp is 00000001
     * if positive/0, tmp is 00000000
     */
    tmp &= mask;

    /* if x is 0 -- now it is 1, else it is 0 */
    x = !x; // handle the 0

    /* if x is negative or '0', make x to 1
     * if x is positive, make x to 0
     */
    G02 /* Fill Your Code */;       x = x | tmp;

    /* if x positive, make it 1.
     * if negative/0, make it 0.
     */
    x = !x;

    /* return 1 if x > 0, else return 0 */
    return (bool)x;
}
```

~ bitwise NOT

! logical operator NOT

(return 0 or 1)

10000000

or 01000000

11000000

11110000

11111111

000…0 11111

0101 0101 … 0101

0011 0011 … 0011