1. **Memory Allocator**
Consider a dynamic memory allocator with the following properties.

- **Double word** (8 bytes) aligned.
- **Implicit** free list is used for free block organization.
- All blocks have a header of size 4 bytes.
- The size of a block (including the header) is stored in the header.
- **bit 0** (least significant bit) in the header indicates the use of the current block: 1 for allocated, 0 for free.

Answer the following questions regarding this allocator:

(a) Minimum block size =

(b) Maximum block size =

(c) For the following memory allocation, what is the size of the payload and padding that will be used in the allocated block? Block splitting might happen based on the size of the request.
You should assume the following: A free block of size **32 bytes** is chosen by the allocator to satisfy the below malloc request. The header of this block is at the memory location **0x8090A0B4**.

```
char *p = malloc(8);

Payload = _____ bytes

Padding = _____ bytes

Memory address stored in the pointer p (in hexadecimal):

0x_____
```

(d) The contents of the header of a block in the allocator is **0x000000A9**.

i. Is the block allocated or free?

ii. What is the size of the block (in decimal)?

iii. Is the contents of the header of this block valid with respect to this allocator? Remember, for a block to be valid with respect to an allocator, its size should satisfy the alignment requirement of the allocator. Yes OR No

2. Assembly review

Consider the following C functions and their corresponding assembly functions that were generated on a Linux/x86 machine.

```
/* copy string x to buf */
void foo(char *x) {
    int buf[1];
    strcpy((char *)buf, x);
}

void callfoo() {
    foo("abcdefghi");
}
```

```
080484f4 <foo>:
080484f4:       55              pushl  %ebp
080484f5:       89 e5           movl   %esp,%ebp
080484f7:       83 ec 18        subl   $0x18,%esp
080484fa:       8b 45 08        movl   0x8(%ebp),%eax
080484fd:       83 c4 f8        addl   $0xfffffff8,%esp
08048500:       50              pushl  %eax
08048501:       8d 45 fc        leal   0xfffffffc(%ebp),%eax
08048504:       50              pushl  %eax
08048505:       e8 ba fe ff ff  call   80483c4 <strcpy>
0804850a:       89 ec           movl   %ebp,%esp
0804850c:       5d              popl   %ebp
0804850d:       c3              ret

08048510 <callfoo>:
08048510:       55              pushl  %ebp
08048511:       89 e5           movl   %esp,%ebp
08048513:       83 ec 08        subl   $0x8,%esp
08048516:       83 c4 f4        addl   $0xfffffff4,%esp
08048519:       68 9c 85 04 08  pushl  $0x804859c   # push string address
0804851e:       e8 d1 ff ff ff  call   80484f4 <foo>
08048523:       89 ec           movl   %ebp,%esp
08048525:       5d              popl   %ebp
08048526:       c3              ret
```

3. Dynamic Memory Allocation - Implementation

Consider an allocator that uses an **implicit free list**. Each memory block, either allocated or free, has a size that is a multiple of **eight bytes**. Thus, only the **29 higher order bits** in the **header** and **footer** are needed to record block size, which **includes** the header and footer and is represented in units of **bytes**. The usage of the remaining **3 lower order bits** is as follows:

- **bit 0** indicates the use of the current block: 1 for allocated, 0 for free.
- **bit 1** indicates the use of the previous adjacent block: 1 for allocated, 0 for free.
- **bit 2** is unused and is always set to be 0.

Five helper routines are defined to facilitate the implementation of `free(void *p)`. The functionality of each routine is explained in the comment above the function definition. Fill in the body of the helper routines, the code section label (i.e., A, B, or C) that implement the corresponding functionality correctly. The **tilde (~) operator** in C is the **ones' complement** operator.

```
/* given a pointer p to an allocated block, i.e., p is a
pointer returned by some previous malloc()/realloc() call;
returns the pointer to the header of the block*/
void * header(void* p)
{
    void *ptr;

    _______;
    return ptr;
}

A. ptr = p-1
B. ptr = (void *)((int *)p-1)
C. ptr = (void *)((int *)p-4)

/* given a pointer to a valid block header or footer,
returns the size of the block */
int size(void *hp)
{
    int result;

    _______;
    return result;
}

A. result=(*hp)&(~7)
B. result=((*(char *)hp)&(~5))<<2
C. result=(*(int *)hp)&(~7)
```

Some useful things to help you solve this problem:

- `strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating '\0' character) to address `dst`. It does **not** check the size of the destination buffer.
- 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
'a'	0x61	'f'	0x66
'b'	0x62	'g'	0x67
'c'	0x63	'h'	0x68
'd'	0x64	'i'	0x69
'e'	0x65	'\0'	0x00

Now consider what happens on a Linux/x86 machine when `callfoo()` calls `foo()` with the input string "`abcdefghi`".

(a) List the contents of the following memory locations immediately **after** `strcpy` returns to `foo`. Each answer should be an **unsigned 4-byte integer** that would be output when printed using a printf (e.g., `0x0840A1B1`).

`buf[0]` = 0x_____

`buf[1]` = 0x_____

`buf[2]` = 0x_____

(b) Immediately **before** the `ret` instruction at address `0x0804850d` executes, what is the value of the frame pointer register `%ebp`?

`%ebp` = 0x_____

(c) Immediately **after** the `ret` instruction at address `0x0804850d` executes, what is the value of the program counter register `%eip`?

`%eip` = 0x_____