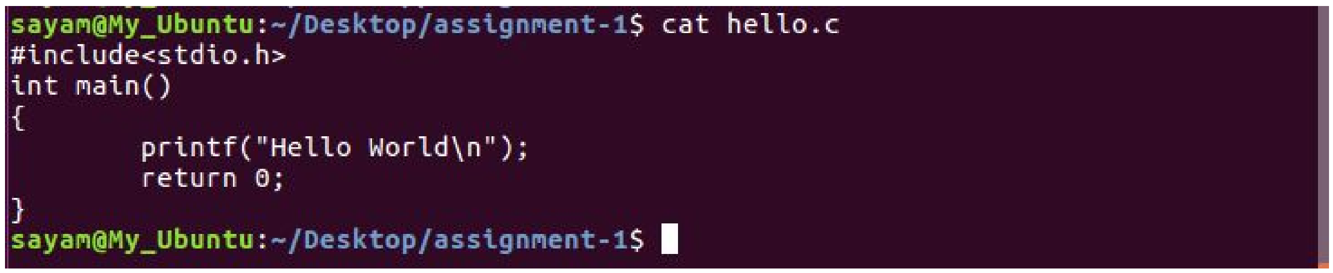
Computer Organizations and Systems Assignment -1

Submitted By -

Sayam Kumar
S20180010158 Sec-A

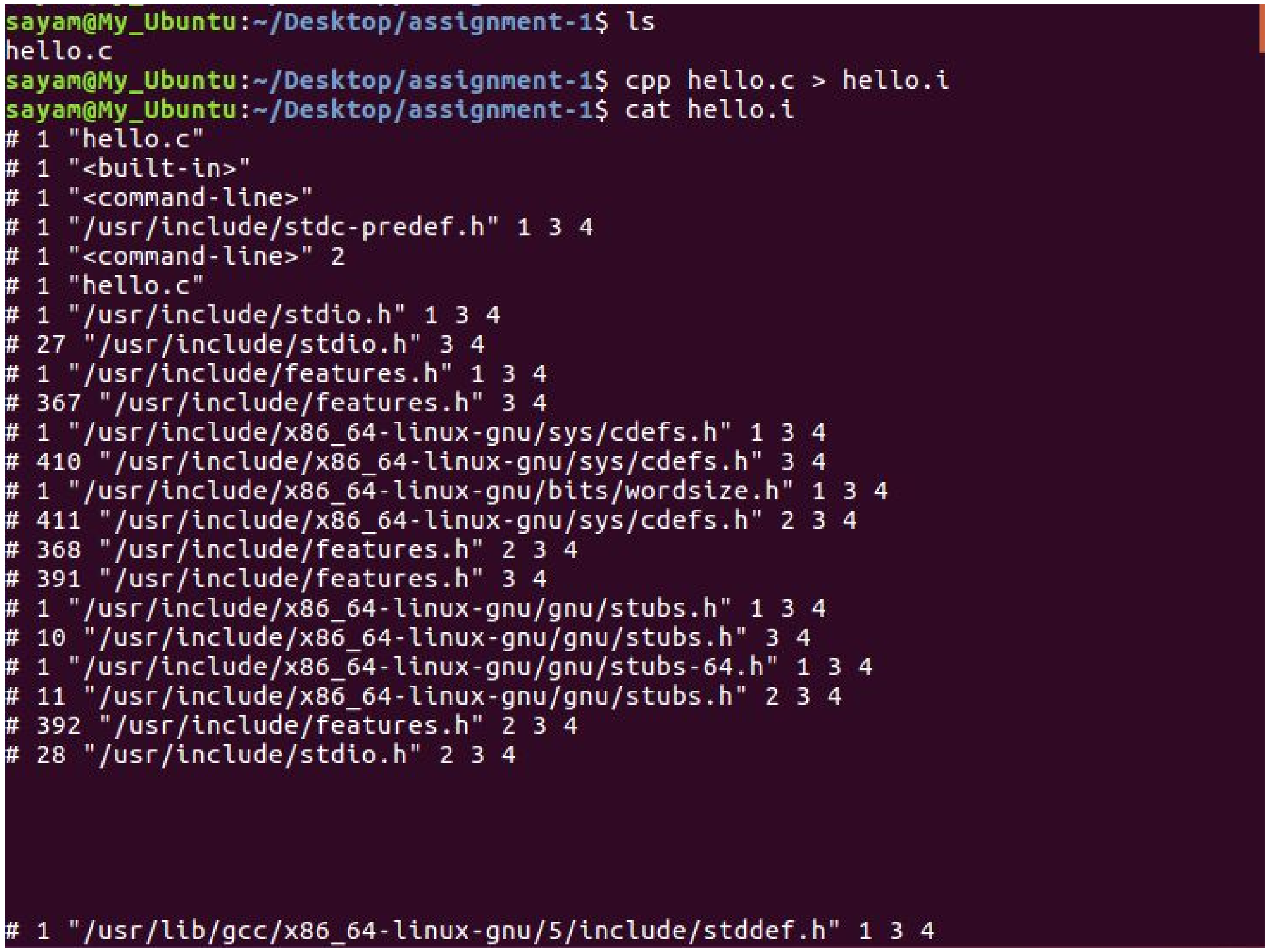
Question-1. Show all the steps for execution of object file for hello world program with screenshots after successfully loading into RAM.
Answer - Below is the screenshot of my hello world program.



```
sayam@My_Ubuntu:~/Desktop/assignment-1$ cat hello.c
#include<stdio.h>
int main()
{
        printf("Hello World\n");
        return 0;
}
sayam@My_Ubuntu:~/Desktop/assignment-1$ 
```

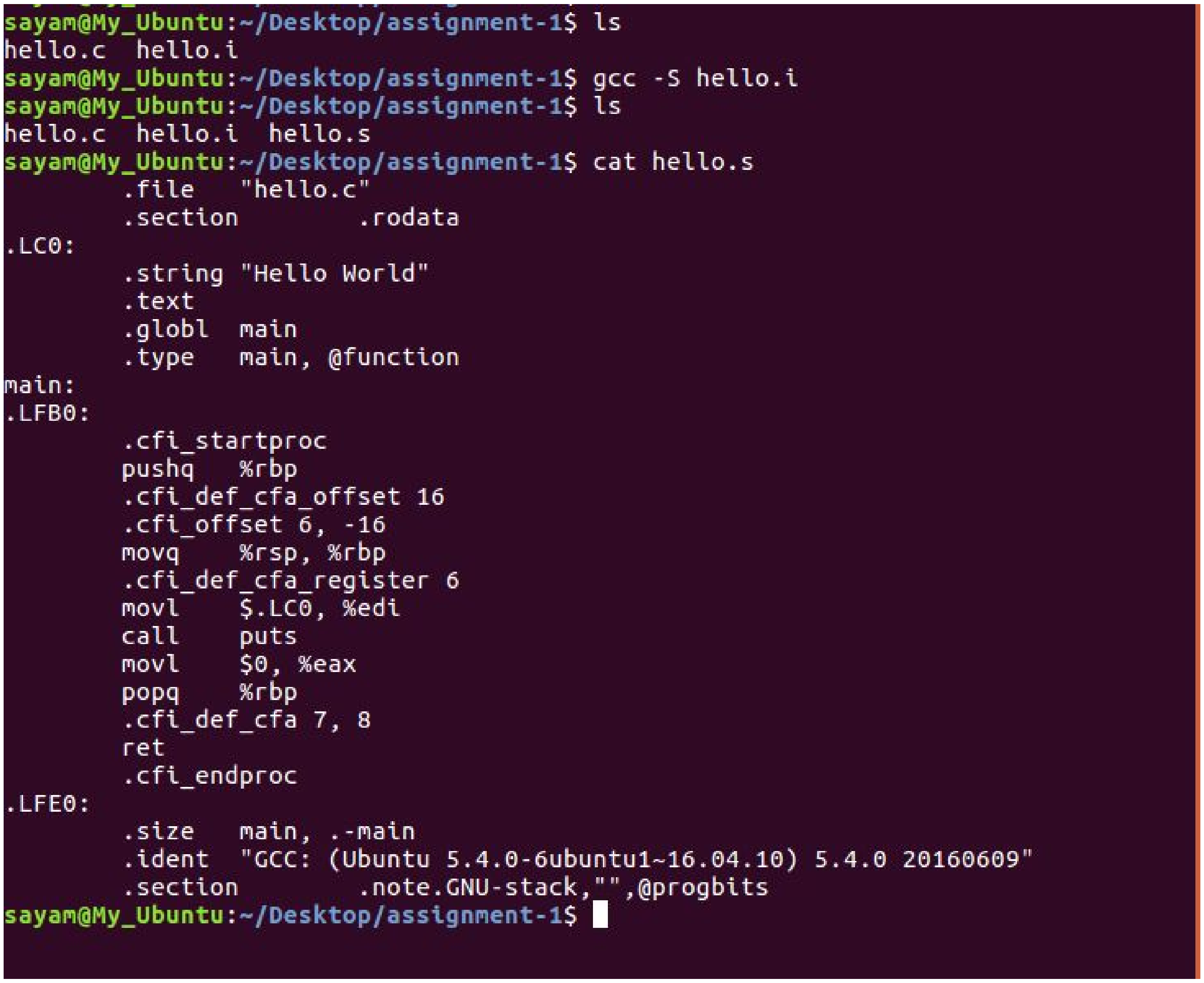
There are four major steps before successful execution of our C program. These are Pre-processing, compilation, assembling and linking. All these four stages are explained in detail in the forthcoming pages.

1. Pre-processing - Through pre-processing, all the declarations present inside the header files gets included in the C program. Also, pre-processing removes the comments before compilation of the program and completes the task of macro expansion. The return 0 statement verifies the successful execution of program to the parent process ie shell. The hello.i file generated after command **cpp hello.c > hello.i** includes all declarations and macros replaced. The same is shown in the screenshot below. The hello world program written by us gets appended to the last of hello.i file.



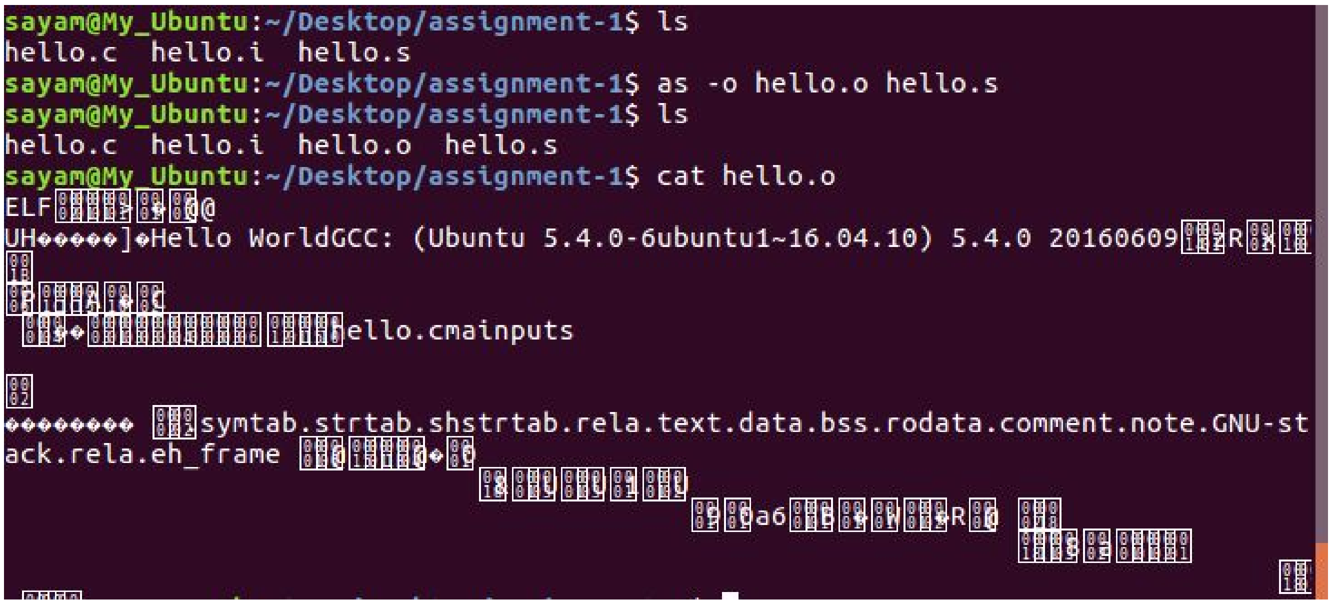
```
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c
sayam@My_Ubuntu:~/Desktop/assignment-1$ cpp hello.c > hello.i
sayam@My_Ubuntu:~/Desktop/assignment-1$ cat hello.i
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 367 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 1 3 4
# 410 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/bits/wordsize.h" 1 3 4
# 411 "/usr/include/x86_64-linux-gnu/sys/cdefs.h" 2 3 4
# 368 "/usr/include/features.h" 2 3 4
# 391 "/usr/include/features.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 1 3 4
# 10 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 3 4
# 1 "/usr/include/x86_64-linux-gnu/gnu/stubs-64.h" 1 3 4
# 11 "/usr/include/x86_64-linux-gnu/gnu/stubs.h" 2 3 4
# 392 "/usr/include/features.h" 2 3 4
# 28 "/usr/include/stdio.h" 2 3 4






# 1 "/usr/lib/gcc/x86_64-linux-gnu/5/include/stddef.h" 1 3 4
```

2. Compilation - Through compilation, the code in hello.i gets compiled. The major process involved in compilation are parsing, lexing of the code and optimization of the unused variables. The -S flag indicates that compilation should stop before assembling the code. This will help us to dive deeply into the process of execution of the hello world program. The hello.s file generated after compilation is an assembly file written in low level language. This also contains information regarding the instruction set of the microprocessor.



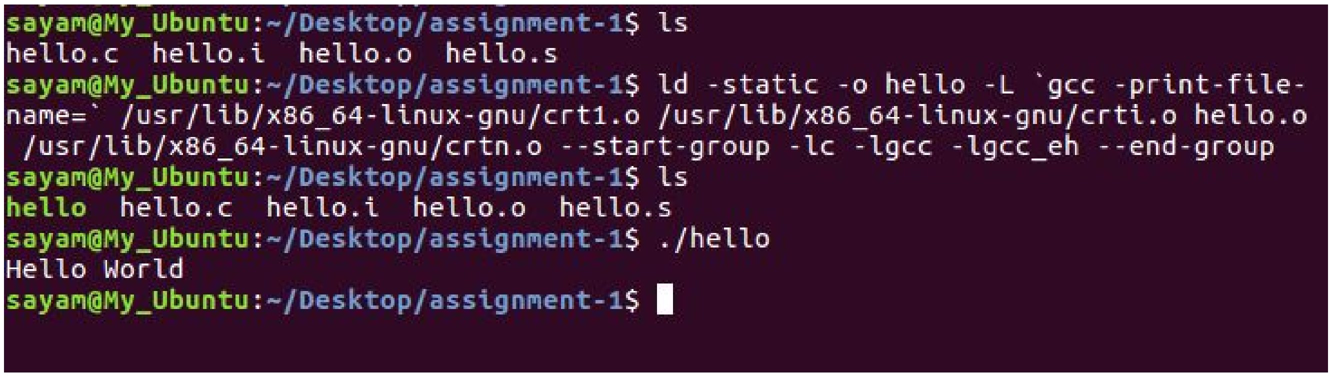
```
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c  hello.i
sayam@My_Ubuntu:~/Desktop/assignment-1$ gcc -S hello.i
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c  hello.i  hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ cat hello.s
        .file   "hello.c"
        .section        .rodata
.LC0:
        .string "Hello World"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609"
        .section        .note.GNU-stack,"",@progbits
sayam@My_Ubuntu:~/Desktop/assignment-1$
```

3. Assembling - Through assembling, the assembly code in hello.s file gets changed into object code. This object code is in the form of binary language which is ready to be served to the microprocessor. The command for generating hello.o file is **as -o hello.o hello.s** . The ELF(Executable and Linkable format) is a common standard file format for executables and object codes.



```
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c  hello.i  hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ as -o hello.o hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c  hello.i  hello.o  hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ cat hello.o
ELF[illegible]@
UH�����]�Hello WorldGCC: (Ubuntu 5.4.0-6ubuntu1~16.04.10) 5.4.0 20160609[illegible]R[illegible]
[illegible]
[illegible]
 [illegible]hello.cmainputs

[illegible]
�������� [illegible]symtab.strtab.shstrtab.rela.text.data.bss.rodata.comment.note.GNU-st
ack.rela.eh_frame [illegible]
                                                      [illegible]
                                                                            [illegible]a6[illegible]R[illegible]
                                                                                              [illegible]
```

4. Linking - Through linking, we link all the library files. For fixing the error in loading, I have taken the help of the website shown to us. These crt files help in generating stack in memory and then successfully loads the program in memory. After execution of **./hello,** the output of “**Hello World**” is shown correctly.



```
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello.c  hello.i  hello.o  hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ ld -static -o hello -L `gcc -print-file-
name=` /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o hello.o
 /usr/lib/x86_64-linux-gnu/crtn.o --start-group -lc -lgcc -lgcc_eh --end-group
sayam@My_Ubuntu:~/Desktop/assignment-1$ ls
hello  hello.c  hello.i  hello.o  hello.s
sayam@My_Ubuntu:~/Desktop/assignment-1$ ./hello
Hello World
sayam@My_Ubuntu:~/Desktop/assignment-1$
```

Question-2. Write a C program to convert any decimal representation into binary format.

Answer - The logic is simple. To convert any integer into binary format, we need to take the remainder by 2 of the number and divide the number.This process is repeatedly done. If we want to convert any float into binary format, we separate the integer and fractional part and then convert into binary format. The C program for this problem is given below -

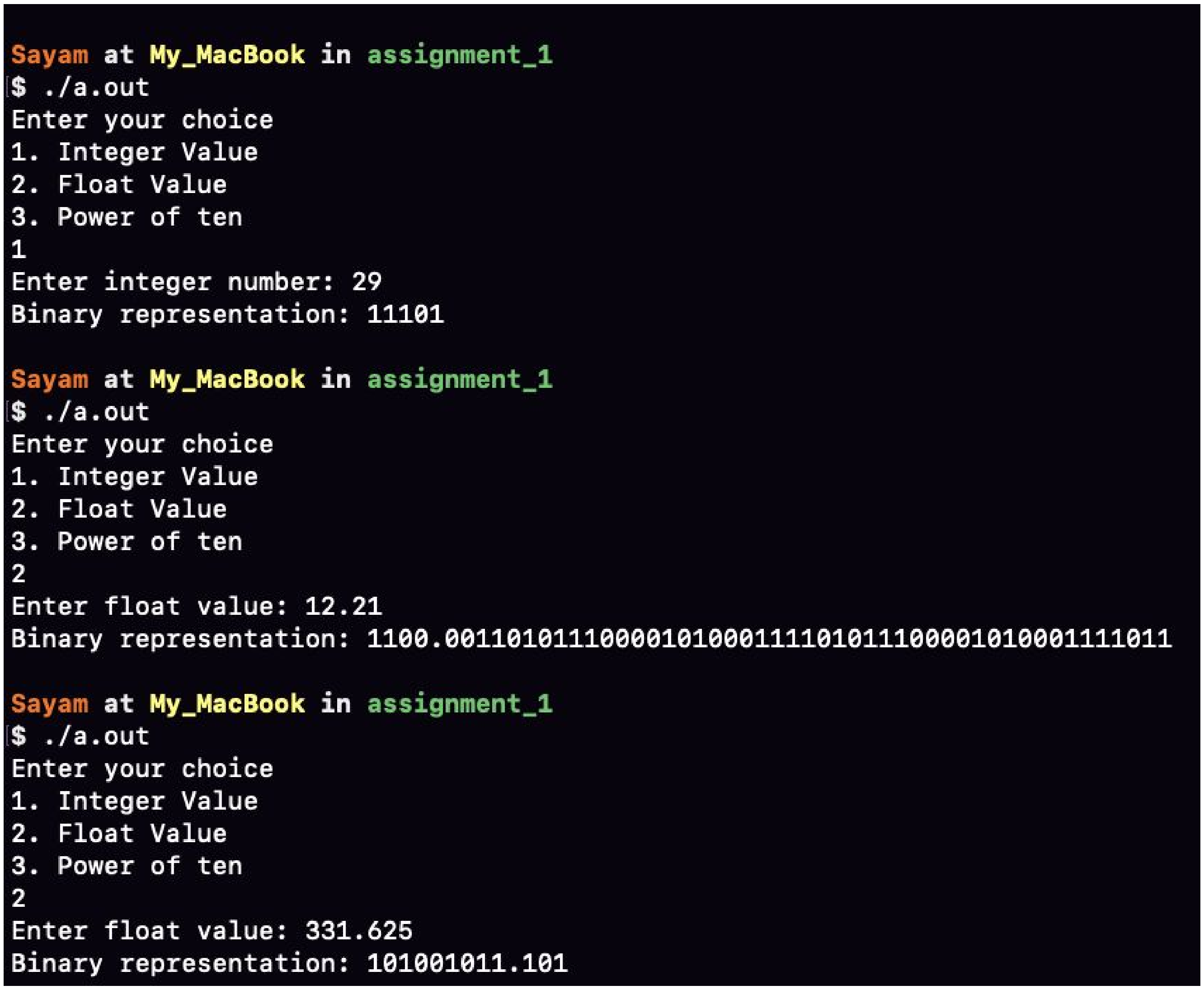
```
#include<stdio.h>

#include<math.h>

void DecimalToBinary(unsigned long long int num)

{

   if (num==0)

   {

       printf("0");

       return;


   int length = ceil(log(num)/log(2)) + 1, i=0;

   char BinaryString[length+1];

   while(num>0)

   {

       BinaryString[i] = num%2 + '0';

       i++;

       num = num>>1; //dividing by two

   }

   for(int j=i-1;j>=0;j--)

   {

       printf("%c",BinaryString[j]);

   }

}


void FloatToBinary(double num)
```

```
{
   unsigned long long int integer_part = (int)num;
   double fractional_part = num-integer_part;
   DecimalToBinary(integer_part);
   printf(".");
   if (num==0)
   {
       printf("0");
       return;
   }
   while(fractional_part!=0 && fractional_part!=1)
   {
       fractional_part *=2.0;
       printf("%d", (int)fractional_part);
       fractional_part = fractional_part - (int)fractional_part;
   }
}

unsigned long long int power(int p) //calculating power in log(n)
time
{
   unsigned long long int answer = 1;
   int x = 10;
   while (p > 0) {
       if (p & 1)
           answer = answer * x;
       p = p >> 1;
       x = x * x;
   }
   return answer;
}
```

```c
int main()
{
    int choice;
    printf("Enter your choice\n1. Integer Value\n2. Float Value\n3. Power of ten\n");
    scanf("%d", &choice);
    switch (choice)
    {
        case 1:
            printf("Enter an integer number: ");
            unsigned long long int num;
            scanf("%lld", &num);
            printf("Binary representation: ");
            DecimalToBinary(num);
            printf("\n");
            break;

        case 2:
            printf("Enter float value: ");
            double float_num;
            scanf("%lf", &float_num);
            printf("Binary representation: ");
            FloatToBinary(float_num);
            printf("\n");
            break;

        case 3:
            printf("Enter a number: ");
            double base;
            scanf("%lf", &base);
            printf("Enter a power of 10: ");
            int p;
            scanf("%d", &p);
            double answer = base*power(p);
            printf("Binary representation: ");
            FloatToBinary(answer);
            printf("\n");
            break;

        default:
            printf("Invalid choice entered\n");
    }
}
```

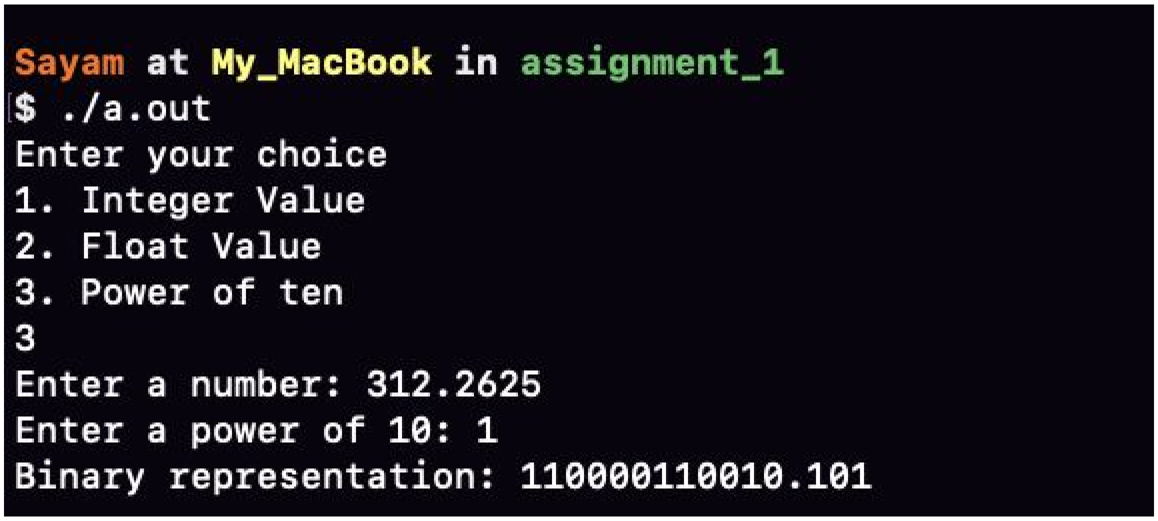
Validation through inputs -

1. Testing cases 1 and 2.



```
Sayam at My_MacBook in assignment_1
$ ./a.out
Enter your choice
1. Integer Value
2. Float Value
3. Power of ten
1
Enter integer number: 29
Binary representation: 11101

Sayam at My_MacBook in assignment_1
$ ./a.out
Enter your choice
1. Integer Value
2. Float Value
3. Power of ten
2
Enter float value: 12.21
Binary representation: 1100.00110101110000101000111101011100001010001111011

Sayam at My_MacBook in assignment_1
$ ./a.out
Enter your choice
1. Integer Value
2. Float Value
3. Power of ten
2
Enter float value: 331.625
Binary representation: 101001011.101
```

2. Testing case 3.



```
Sayam at My_MacBook in assignment_1
$ ./a.out
Enter your choice
1. Integer Value
2. Float Value
3. Power of ten
3
Enter a number: 312.2625
Enter a power of 10: 1
Binary representation: 110000110010.101
```

Question-3. Write a C program to convert a binary number representation into decimal format(only integers).

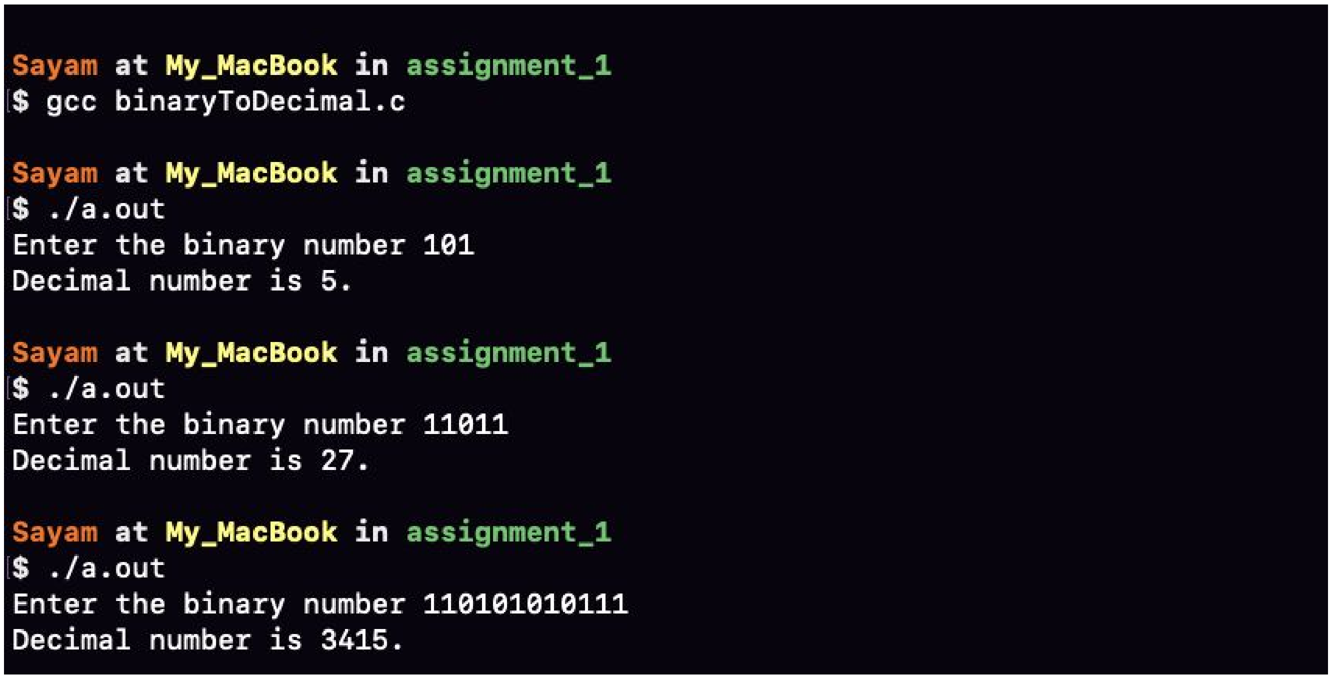
Answer - The logic is simple again. First we validate the given string if it comprises of only 0's and 1's. And then we keep on multiplying the bits with higher powers of two, to get decimal representation. The key factor here is when we multiply a variable to get higher powers of two. The most optimal way to do this is by implementing left shift binary operator.

The C program for this problem is given below -

```c
#include<stdio.h>
#include<string.h>
void BinaryToDecimal(char s[], int length)
{
   unsigned long long int sum=0, base=1;
   for(int i=length-1;i>=0;i--)
   {
       sum = sum + (s[i]-'0')*base;
       base = base<<1; // multiplying base with 2 by left shift.s
   }
   printf("Decimal number is %lld.\n", sum);
}
```

```
int main()
{
   char str[70];
   printf("Enter the binary number ");
   scanf("%s", str);
   int d=strlen(str), flag=0;
   for(int i=0;i<d;i++)
   {
       if (str[i]!='0' && str[i]!='1')
       {
          flag=1;
           break;
       }
   }
   if (flag==1)
   {
       printf("Wrong format entered!\n");
       return 0;
   }
   BinaryToDecimal(str, strlen(str));
}
```

Validation through inputs -



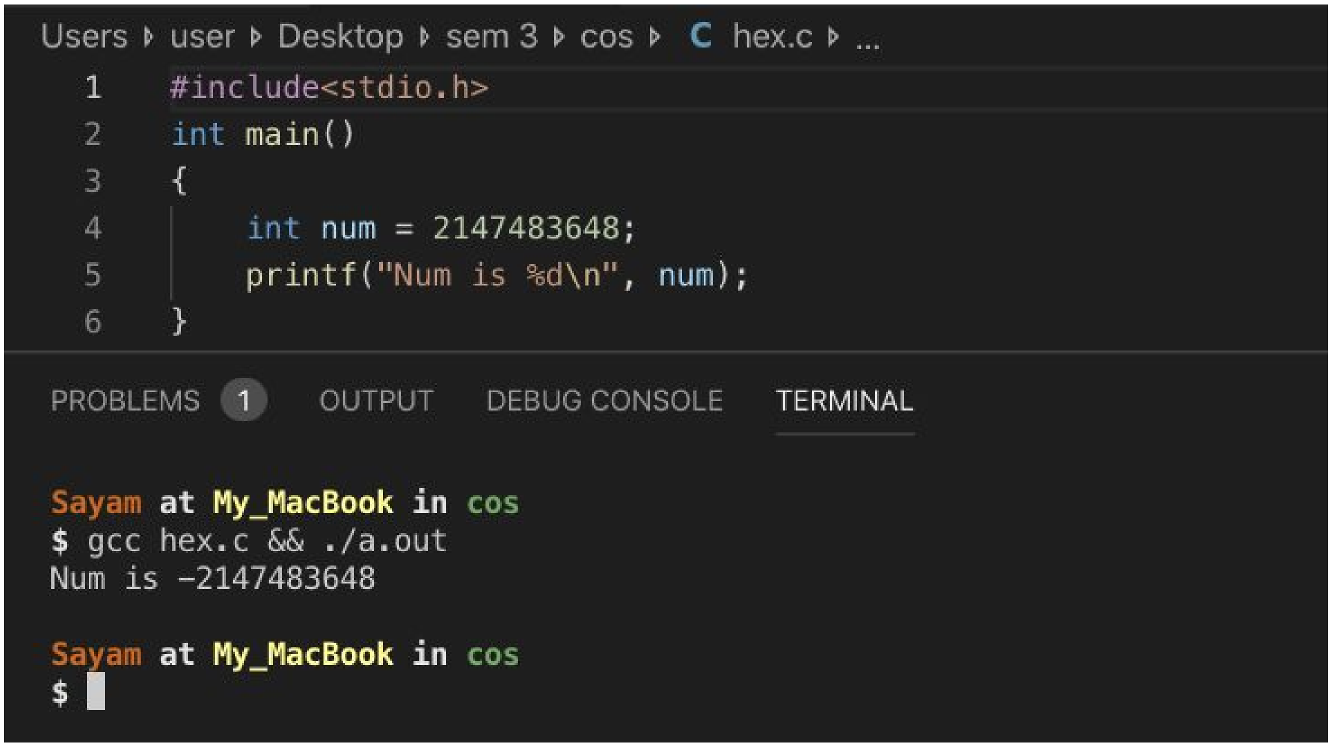
```
Sayam at My_MacBook in assignment_1
[$ gcc binaryToDecimal.c

Sayam at My_MacBook in assignment_1
[$ ./a.out
Enter the binary number 101
Decimal number is 5.

Sayam at My_MacBook in assignment_1
[$ ./a.out
Enter the binary number 11011
Decimal number is 27.

Sayam at My_MacBook in assignment_1
[$ ./a.out
Enter the binary number 110101010111
Decimal number is 3415.
```

Question-4. Consider a hexadecimal number 0x87654321 stored in standard int data type of the C language. Now, as its decimal conversion is 2271560481, it is out of the range for int data type. Then how is C language again converting it back to the same hexadecimal number?

Answer - The range of int data type in the C language is from -2,147,483,648 to 2,147,483,647. Any number greater than 2,147,483,647 will get circularly back to -2,147,483,648 and start again.

Example- 2,147,483,648 is just 1 bigger than 2,147,483,647 gets back to -2,147,483,648.

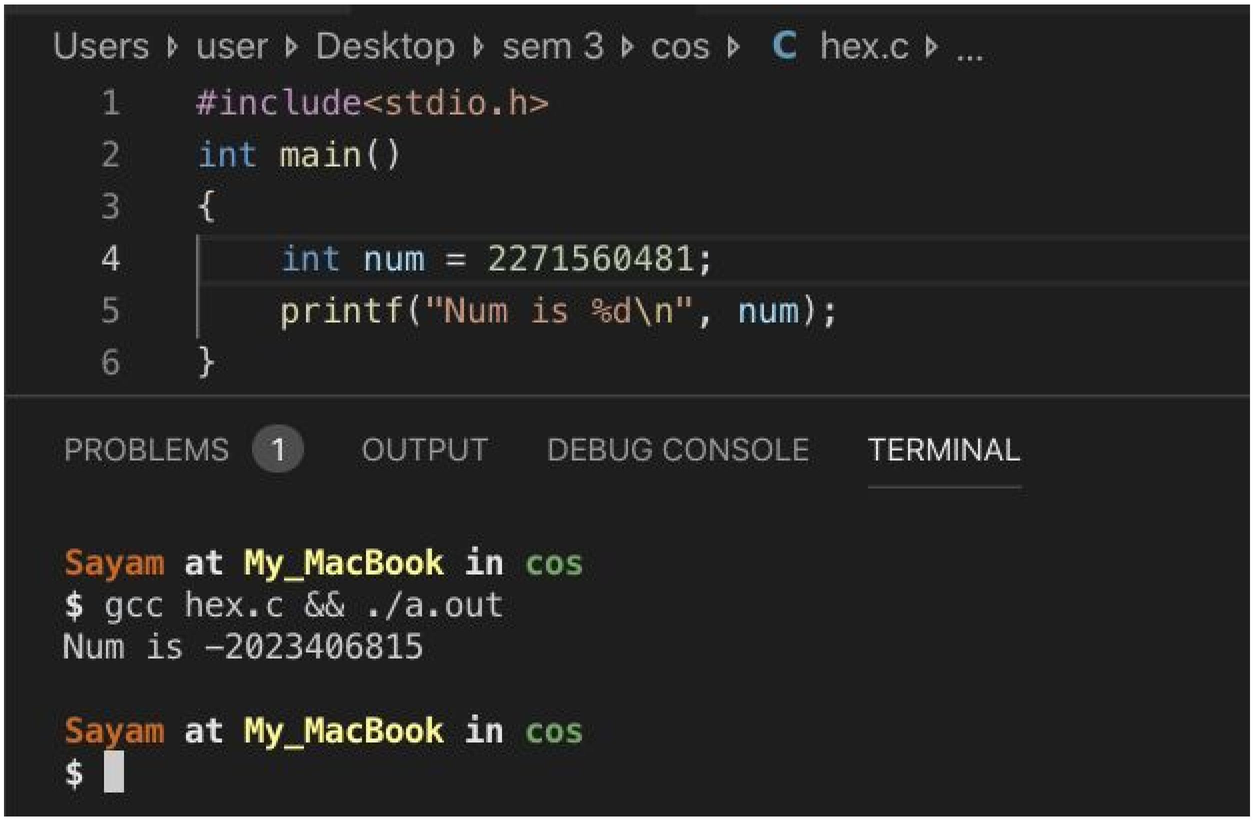
Proof-



```c
#include<stdio.h>
int main()
{
    int num = 2147483648;
    printf("Num is %d\n", num);
}
```

```
Sayam at My_MacBook in cos
$ gcc hex.c && ./a.out
Num is -2147483648

Sayam at My_MacBook in cos
$ 
```

Now the decimal representation of 0x87654321 is 2271560481. As this decimal number is out of range, so its cyclic conversion will be stored as -2023406815. The same problem is indicated by VS Code editor. Proof -



```c
#include<stdio.h>
int main()
{
    int num = 2271560481;
    printf("Num is %d\n", num);
}
```

```
Sayam at My_MacBook in cos
$ gcc hex.c && ./a.out
Num is -2023406815

Sayam at My_MacBook in cos
$ 
```

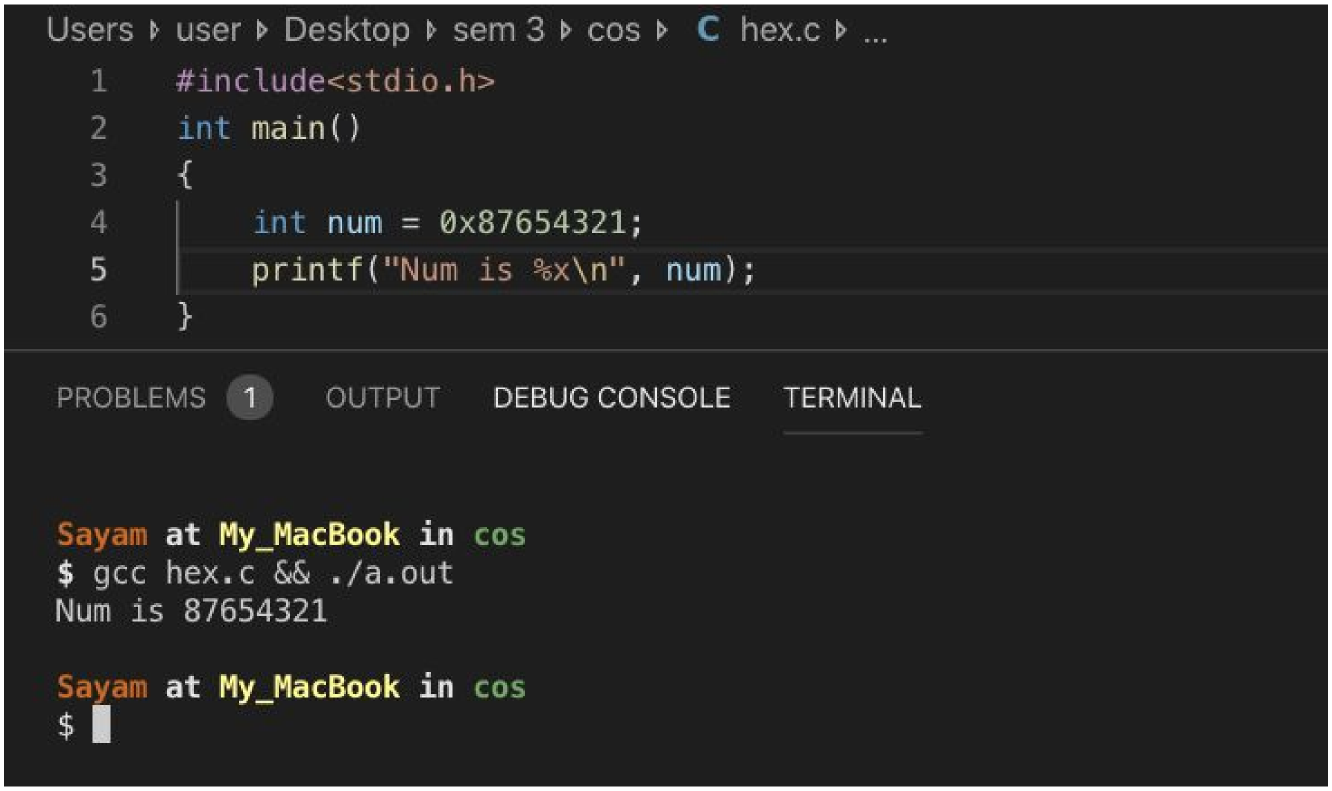
Now, we know that every negative number is stored as 2’s complement in memory. So, for calculating the 2’s complement, first calculate 1’s complement and add 1 to it. For 1’s complement, take the inversion of binary representation by changing 0 to 1 and vica versa. After we have obtained 2’s complement, changing it again to hexadecimal format gives 0x87654321.

The 2’s complement of -2023406815 is 10000111011001010100001100100001. Now to convert this binary string in hex, take the corresponding 4 bits and check its hex representation and repeat the whole process. The 4 bit split up is shown below -

```
 1000 0111 0110 0101 0100 0011 0010 0001
   8     7     6     5      4     3     2     1
```

So, the final output is still the same.

Proof -



```c
#include<stdio.h>
int main()
{
    int num = 0x87654321;
    printf("Num is %x\n", num);
}
```

```
Sayam at My_MacBook in cos
$ gcc hex.c && ./a.out
Num is 87654321

Sayam at My_MacBook in cos
$ 
```

Thank You!