College of

Nanotechnology, Sciences, and Engineering Department of Computer Science

ICSI 333 SYSTEM FUNDAMENTALS CQUPT, SPRING 2025

LAB 1.1 - 1.2

The total grade for both labs is 100 points.

You must follow the programming and documentation guidelines.

**DESCRIPTION**

You will work in two labs to complete this assignment. At the end, you must write a C program that performs the conversion of a decimal number into IEEE 754 Single Precision representation. A user inputs the real number, then the program should output three binary strings that show IEEE 754 binary sign bit, 24-bit mantissa (including hidden bit), and binary biased exponent.

The IEEE 754 standard for floating point numbers is explained in the lecture Number Systems available on Blackboard. In addition, you may find useful the instructions given in the article [How to Convert a Number from Decimal to IEEE 754 Floating](https://www.wikihow.com/Convert-a-Number-from-Decimal-to-IEEE-754-Floating-Point-Representation) [Point Representation](https://www.wikihow.com/Convert-a-Number-from-Decimal-to-IEEE-754-Floating-Point-Representation) or search for more information online.

Special requirements:

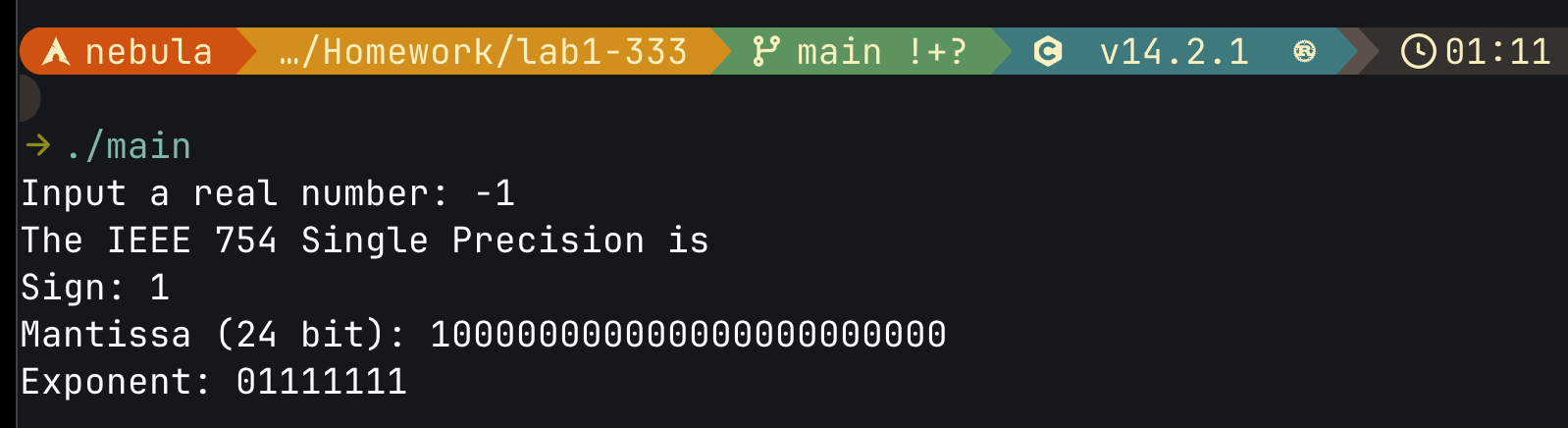
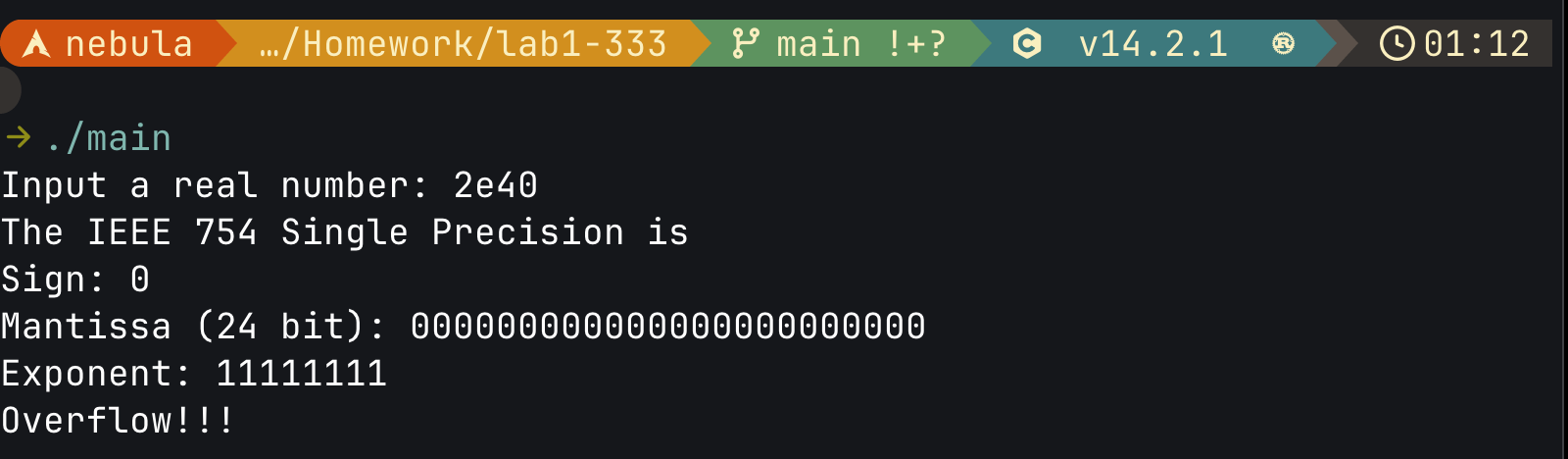
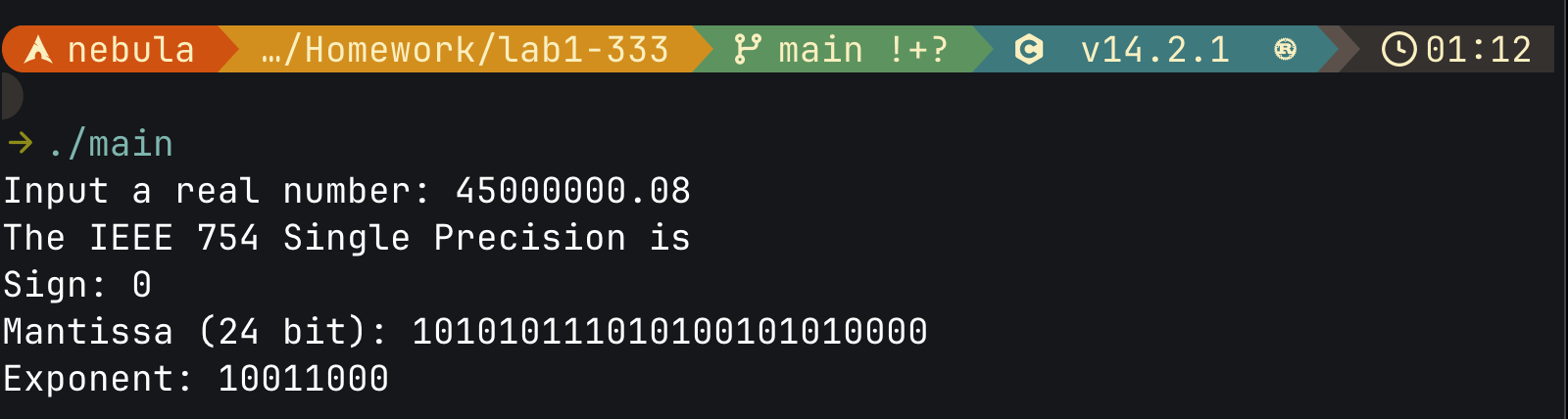
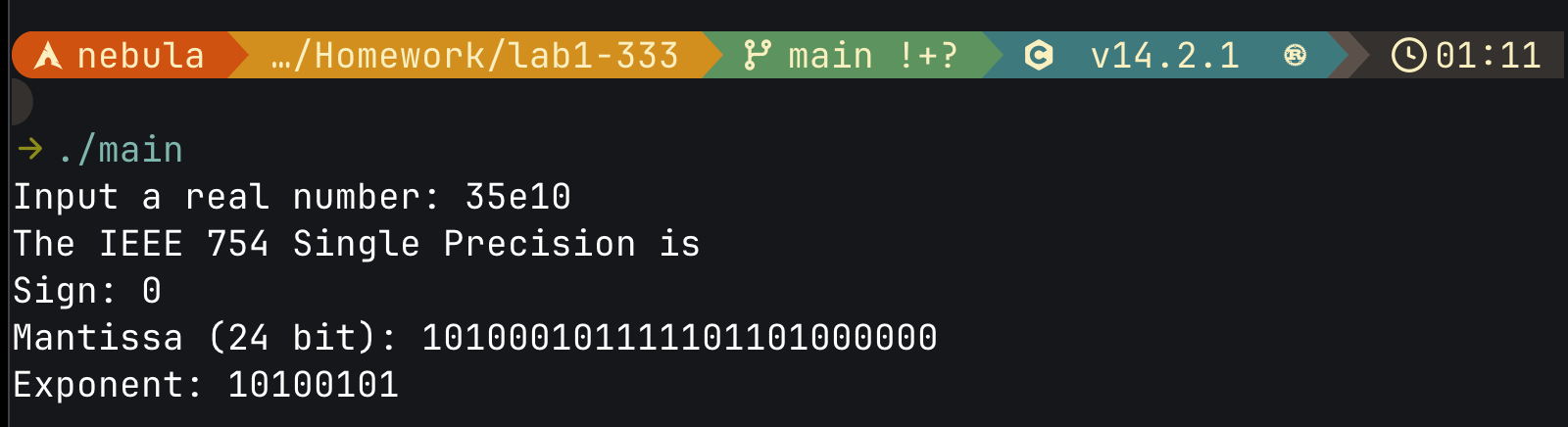
* You **should not use any standard library function** to convert a decimal number into binary format.

# You should not use any bitwise operators.

* Binary numbers in this project should be in the form of an array or string.
* For the sake of simplicity, the absolute value of the user’s input should not exceed the maximum value of the signed long integer type.

Work on the project task by task, starting with small numbers. Once you have correctly working code, elaborate on critical points.

**PROGRAM EXECUTION**



**PROGRAMMING SUGGESTIONS**

* + Work on your project step-by-step. If you cannot fulfil all requirements, but successfully solves some tasks your will be graded anyway. If your code does not compile by gcc, you will have a zero or very low grade.
  + Use a char array to store binary digits. Print arrays when you need output.
  + Use function for every logically separated piece of code. For example, converting whole numbers to binary, calculating the mantissa, etc. Pass the array to the functions by reference.
  + Remember that your code cannot process any possible number, apply a correctness check for inputs and function results.
  + Your program reads inputs from stdin (keyboard) and writes outputs to stdout (terminal window).
  + After each call to the function printf, include the following C statement: fflush(stdout);. Example: printf("Number = %d\n", number); fflush(stdout);
  + Remember that no bitwise operators or standard library functions converting to binary are allowed in this project.

# Do not copy any piece of code from online resources or another person. All works will be checked for plagiarism and academic integrity will be strongly enforced.

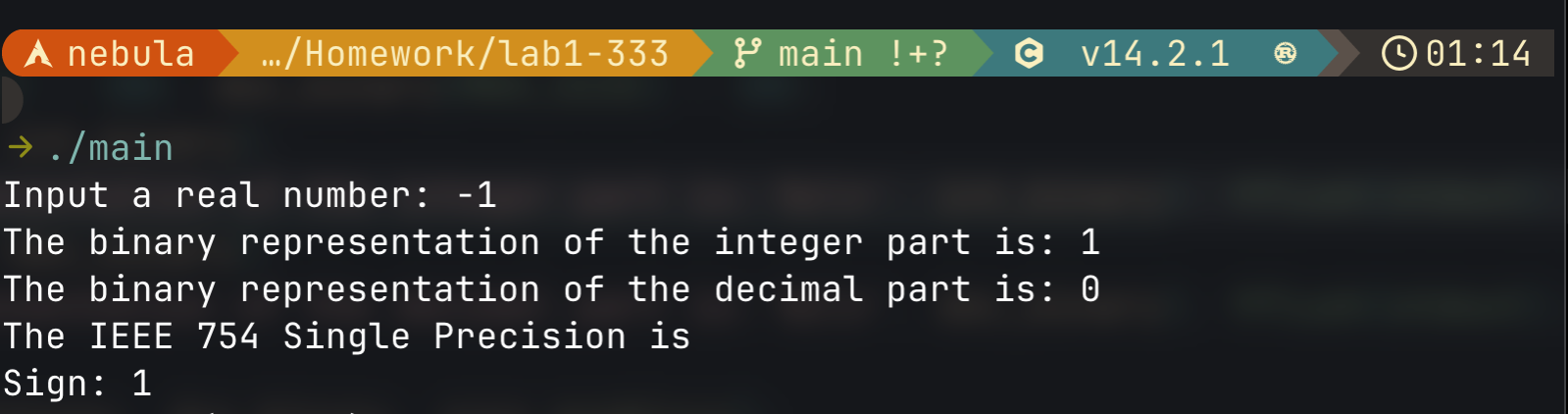
**RECOMMENDED STEPS OF DEVELOPMENT**

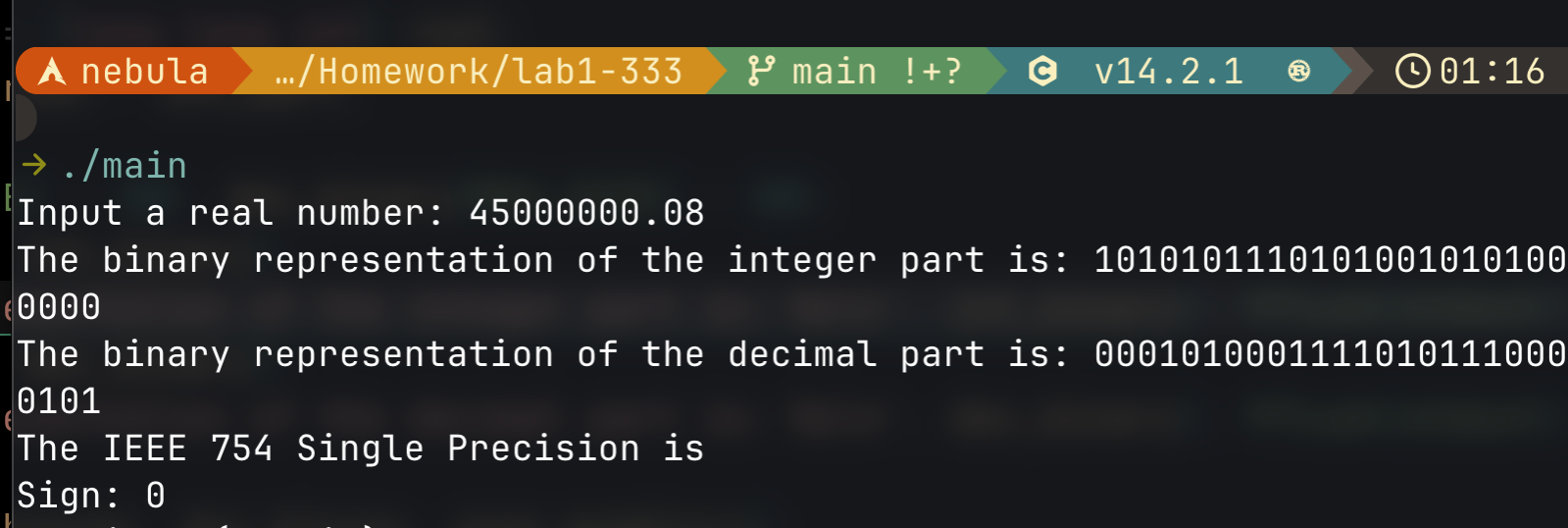
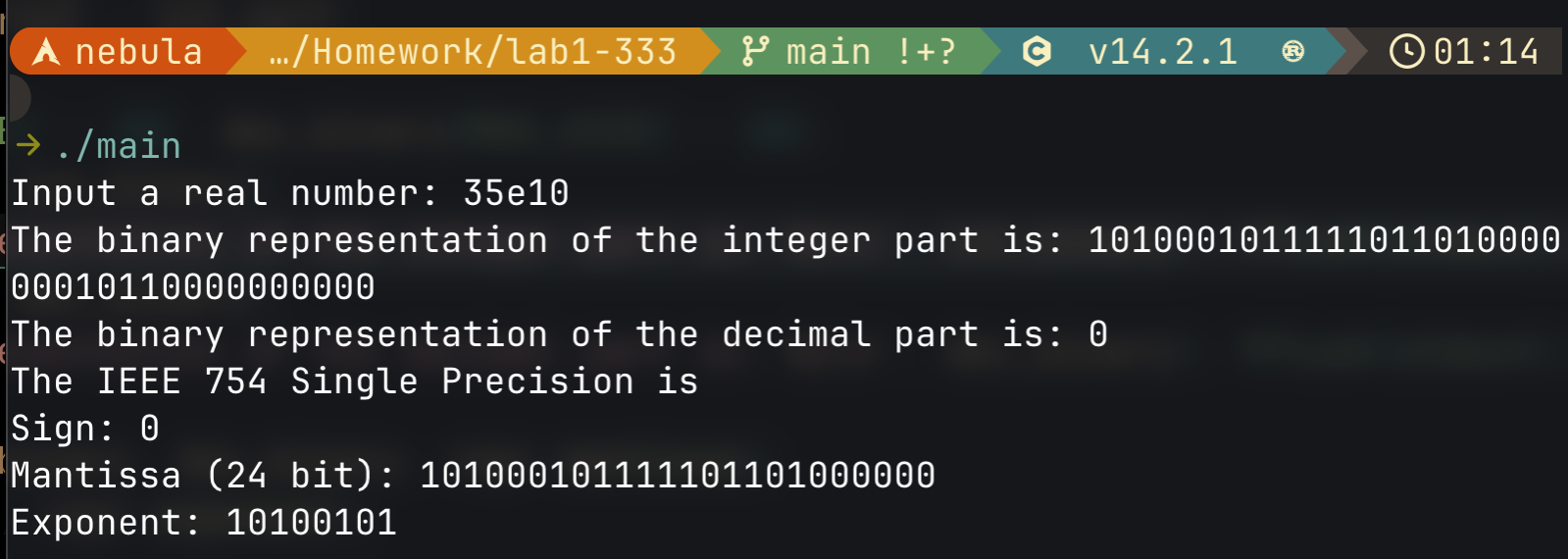
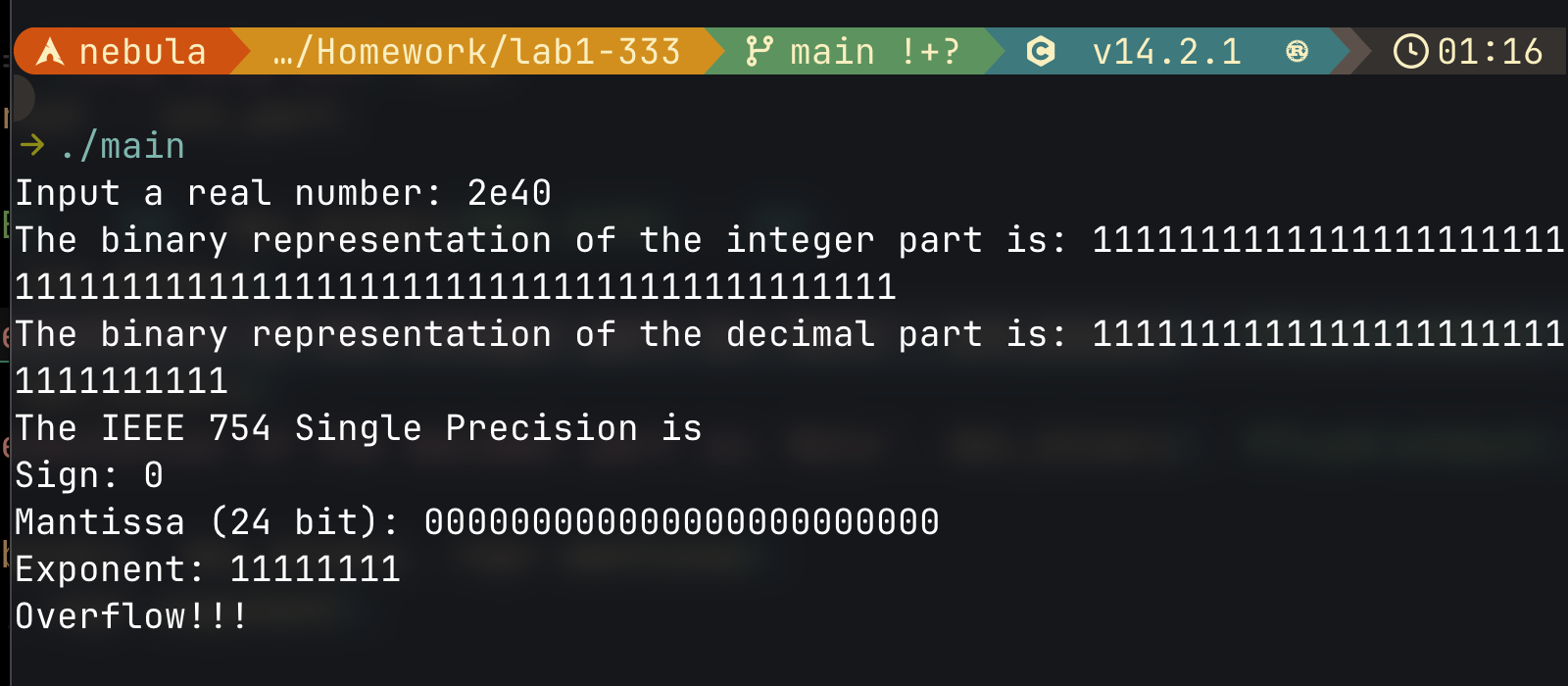
TASK #1. INPUT AND CONVERSION TO BINARY

In this part, you implement algorithms for converting real numbers into binary representation (see lecture Number Systems). Remember that the whole and fractional parts need different algorithms.

The two’s complement is not applicable in IEEE 754. The sign bit for mantissa is the most left bit of the representation and

can be handled independently.

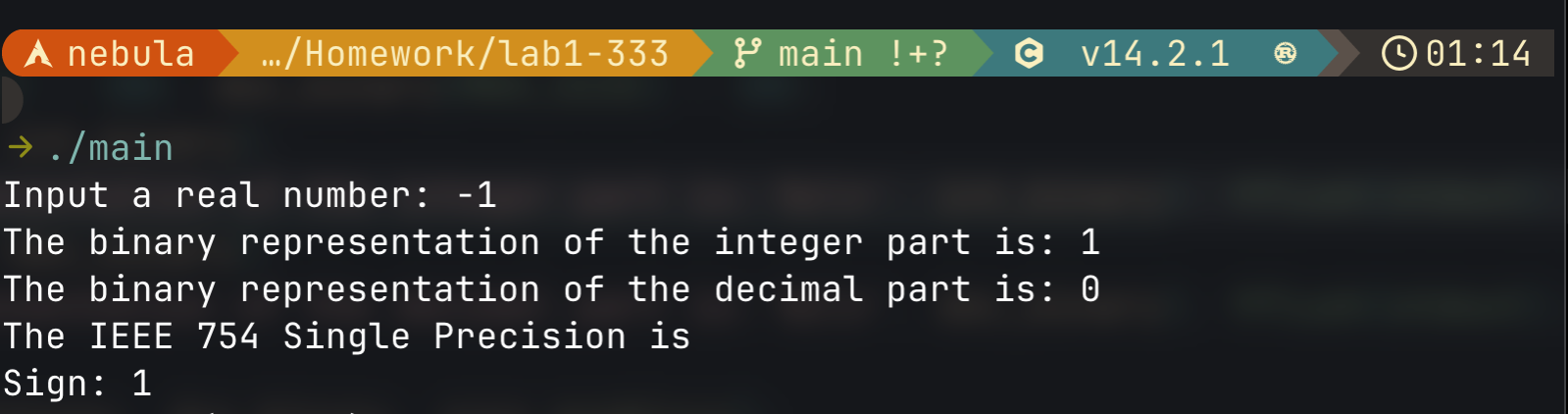


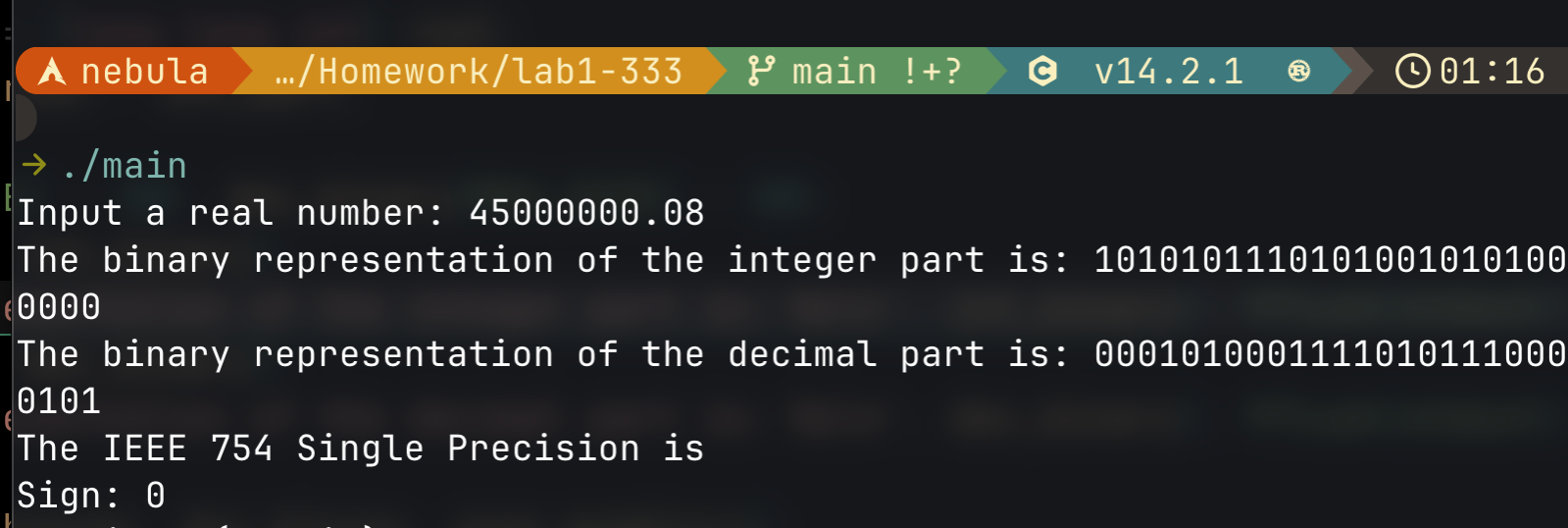
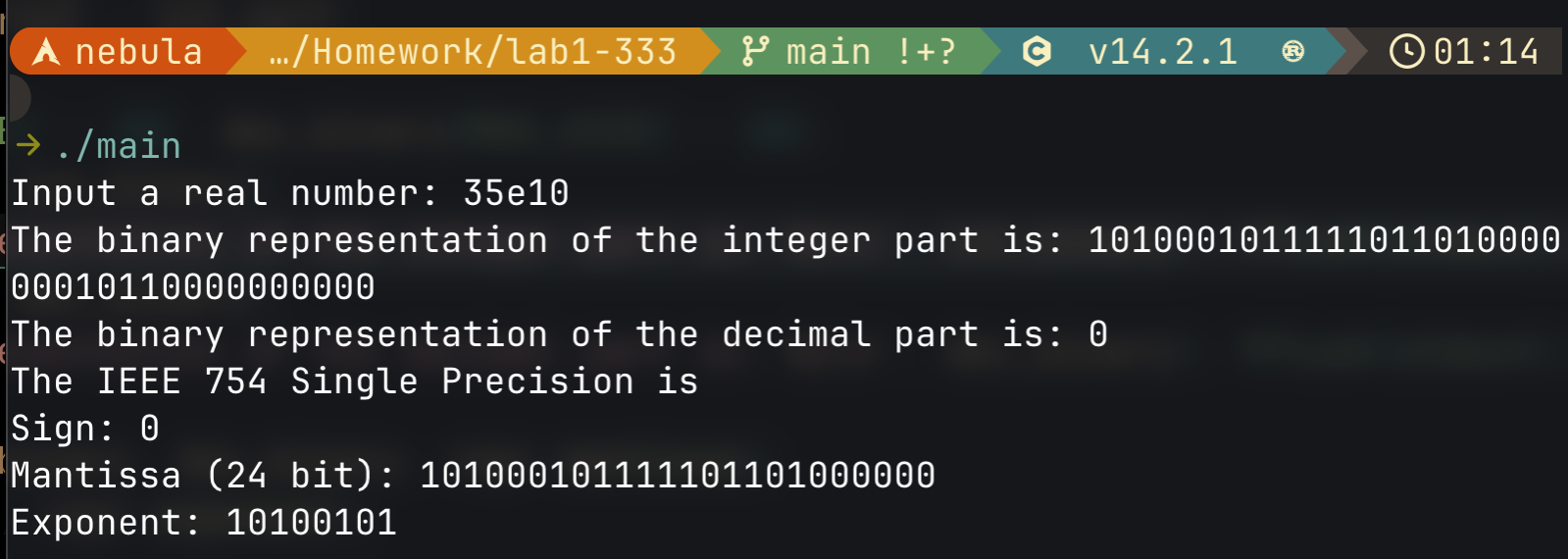
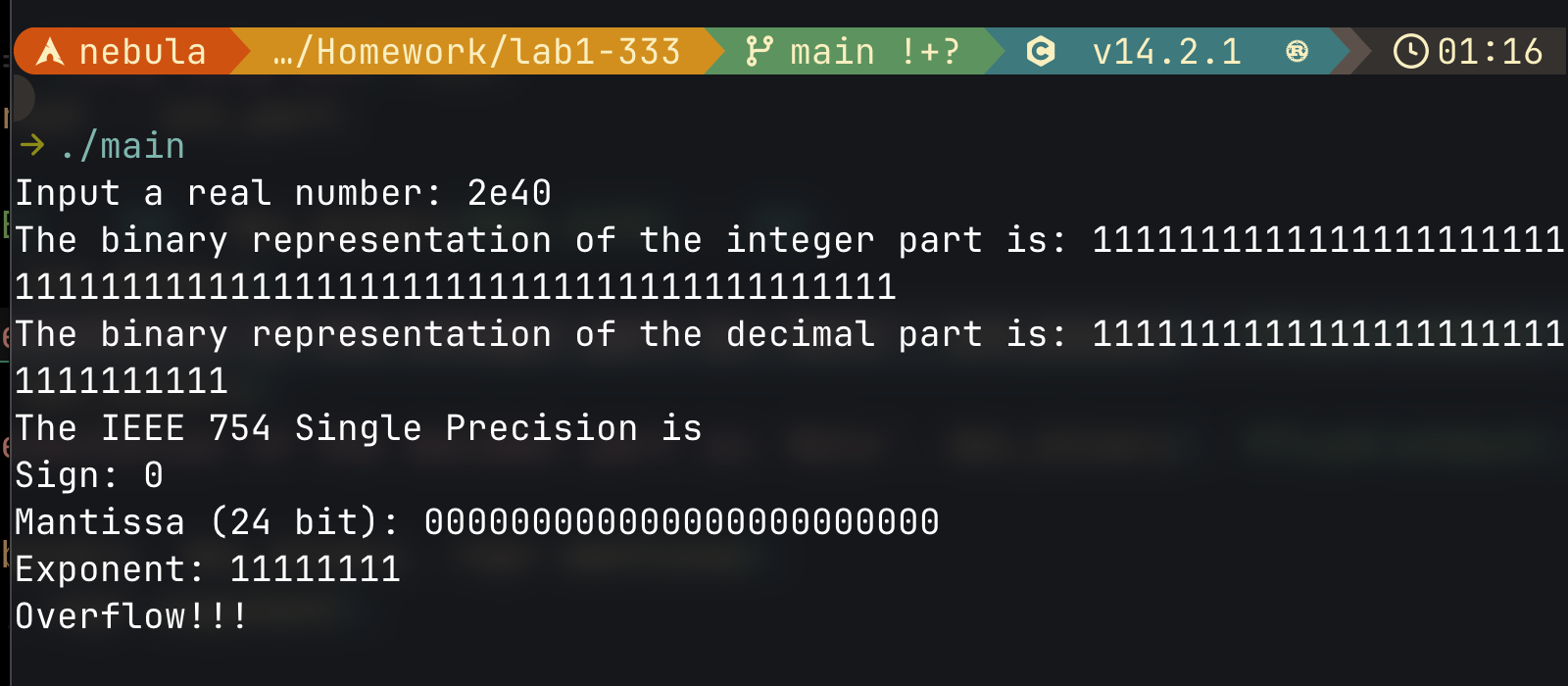


In IEEE 754 mantissa is a 24-bit binary number between (1)2 and (10)2 but the first 1 is not shown, so the total number of bits used for mantissa in IEEE 754 is 23.

To avoid ambiguity your program must show 24 bits.

If the total length of the whole and fractional part is more than 24 bits, the mantissa should be rounded to a 24-bit binary number. Note, the rounding may impact the exponent.

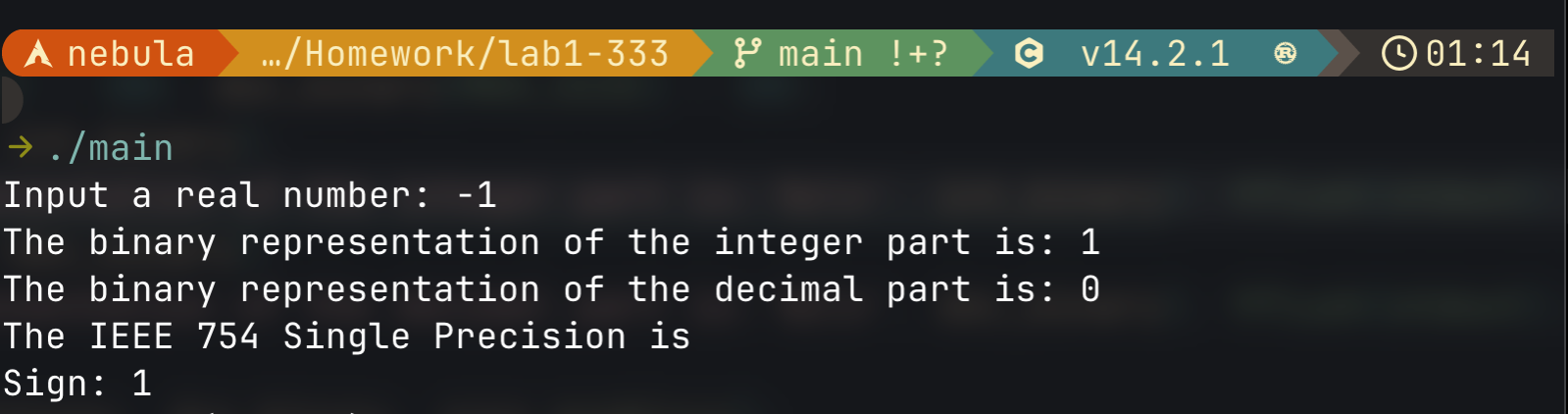


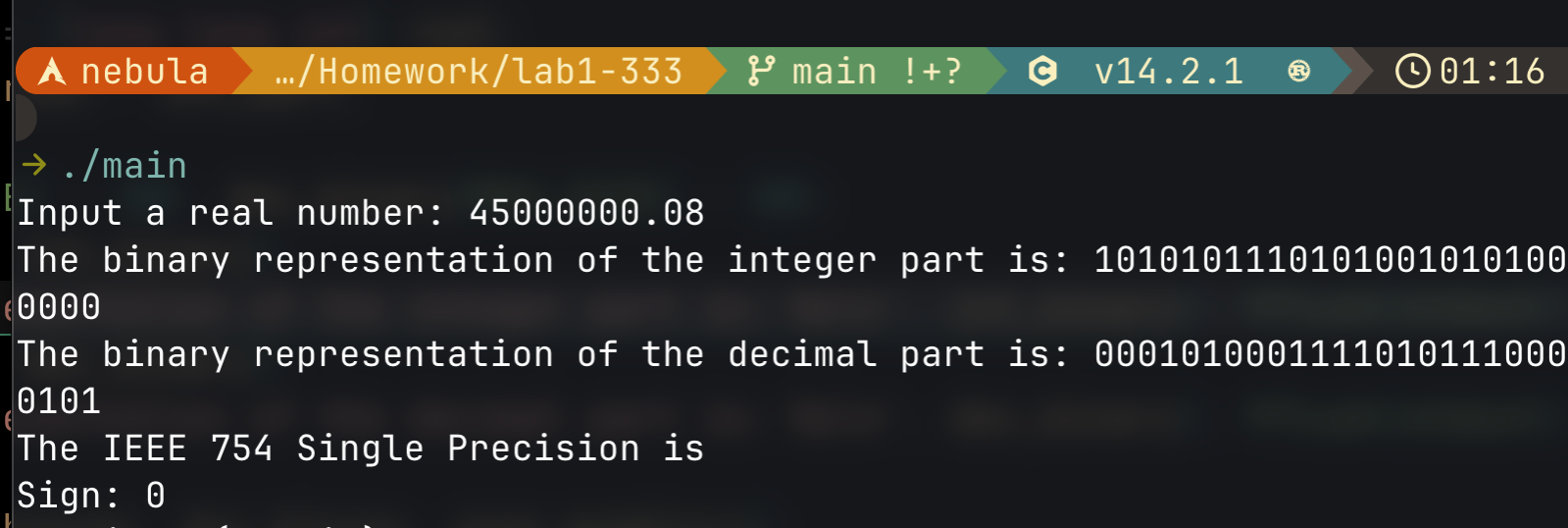
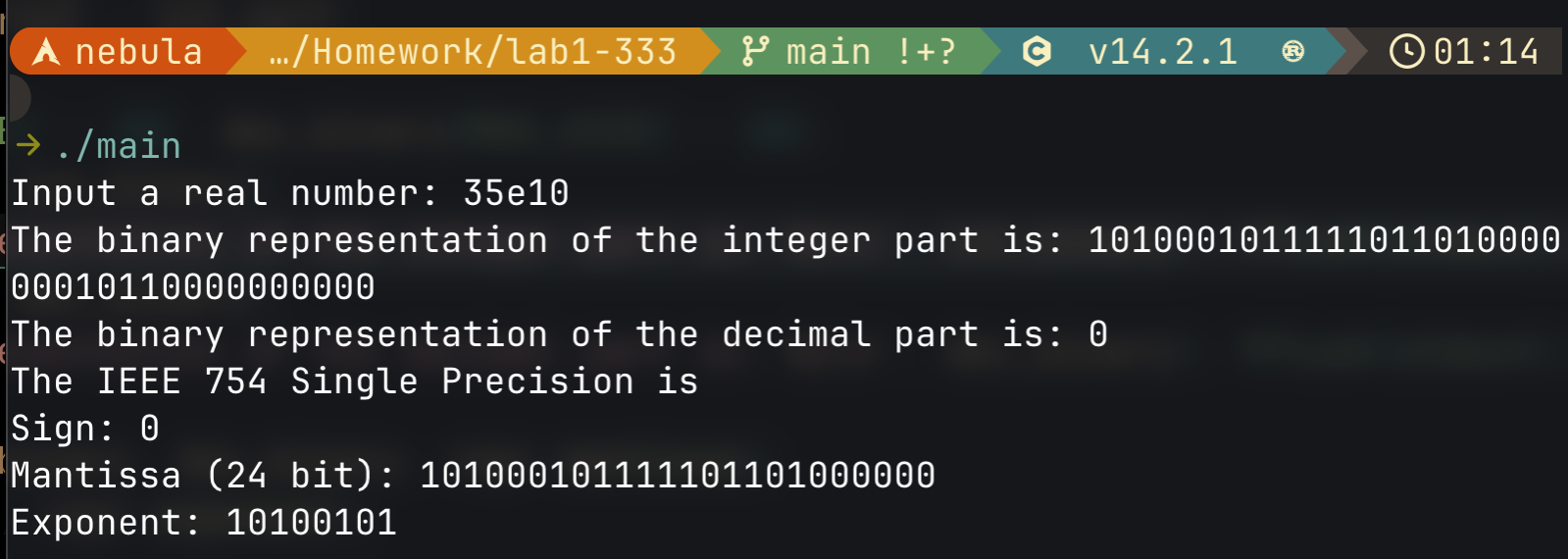
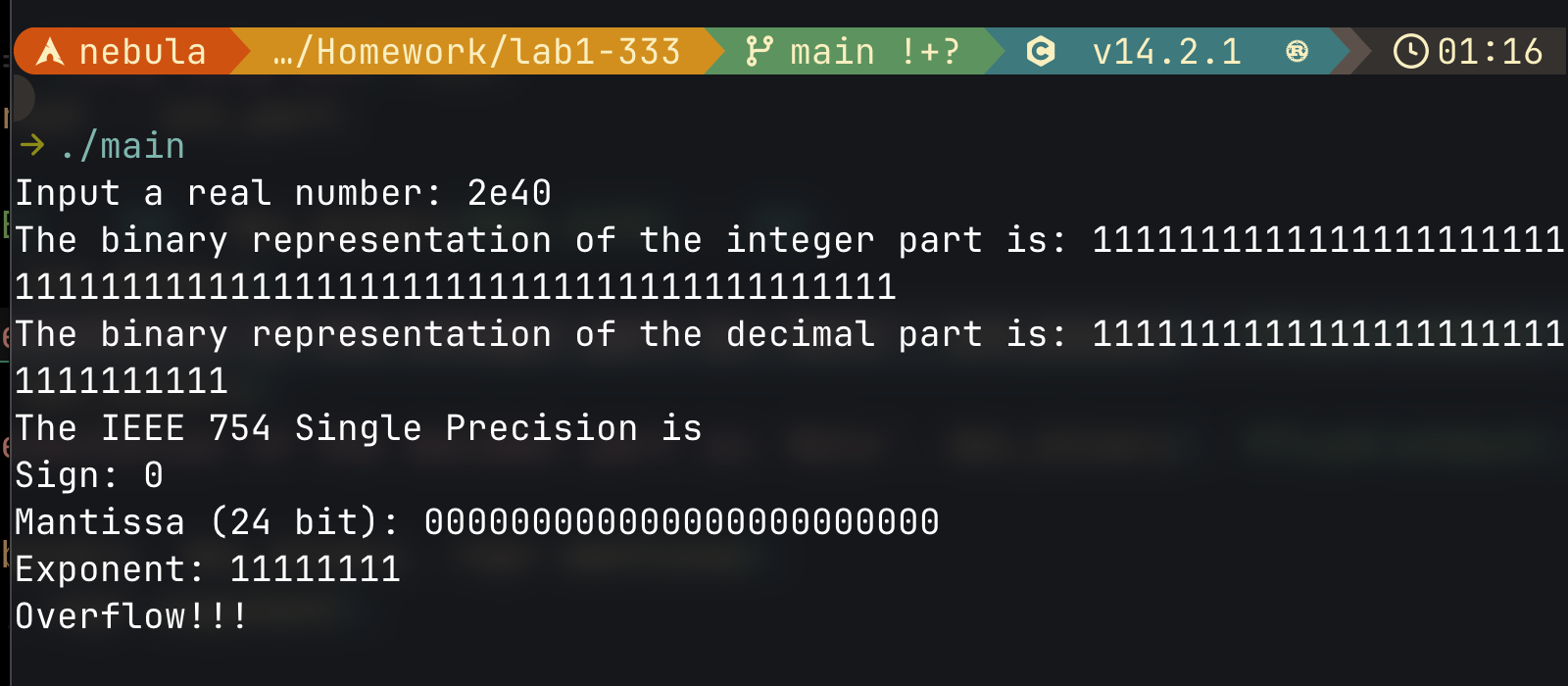


TASK #3. EXPONENT

The exponent is a power of two the mantissa should be multiplied by to calculate the value of the IEEE 754 number. It could be positive or negative, but an 8-bit code is a biased exponent which is 127 greater than the actual exponent. For example, to find the value of the IEEE 754 number the mantissa should be multiplied by 2-10, then the biased exponent is -10 + 127 = 117 = (01110101)2.

The exponent of (11111111)2 is reserved to generate an error message. The exponent of (00000000)2 is reserved to represent decimal 0.0 along with the mantissa of all bits equal to 0.

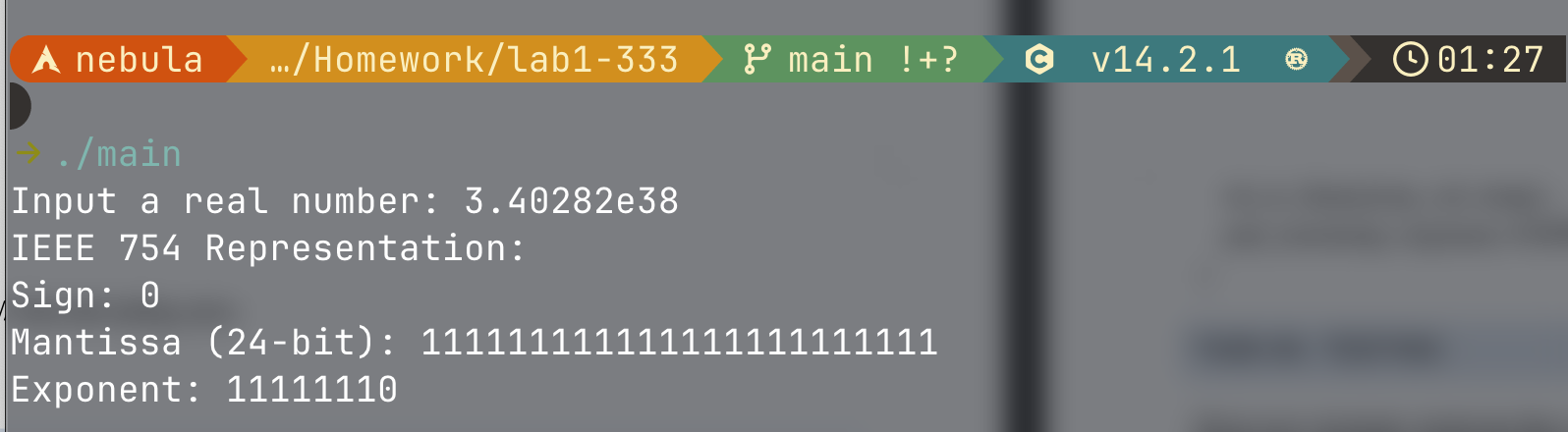


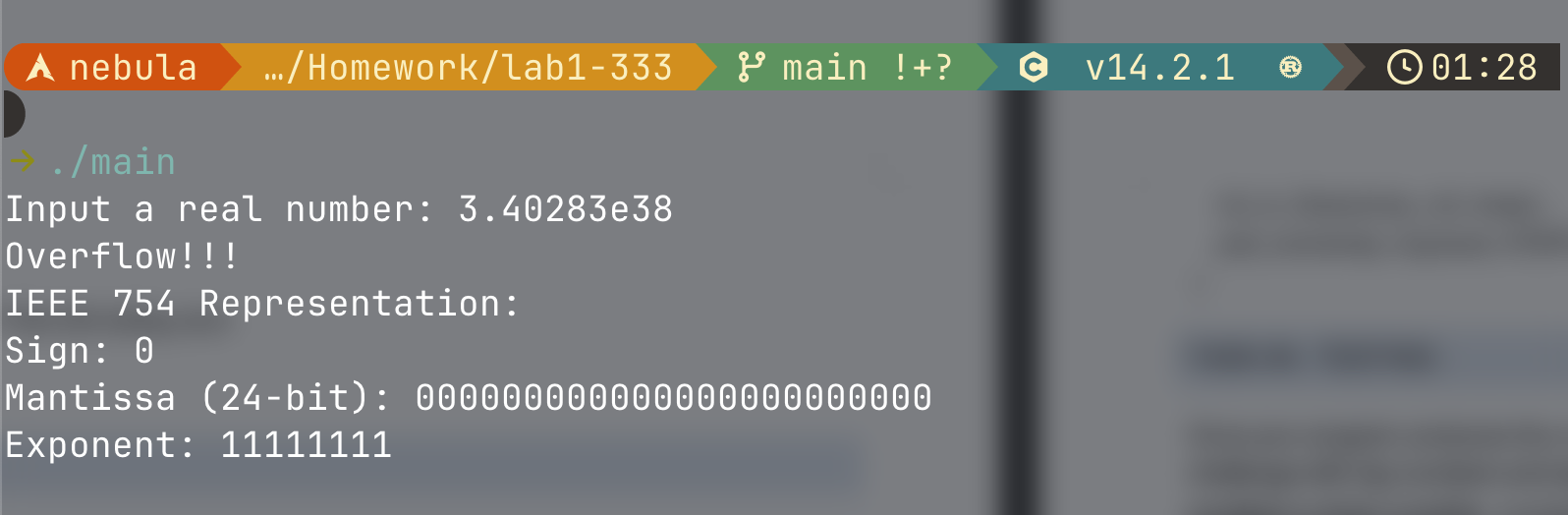
s

TASK #4. TESTING

Once your program produces the correct results for simple examples, like 1.0, 10.0, 0.1, -5.0, etc., you should give it a real challenge with big numbers and all critical points. Check how the program works for the biggest number possible, the smallest number possible, a number in exponential format, etc.

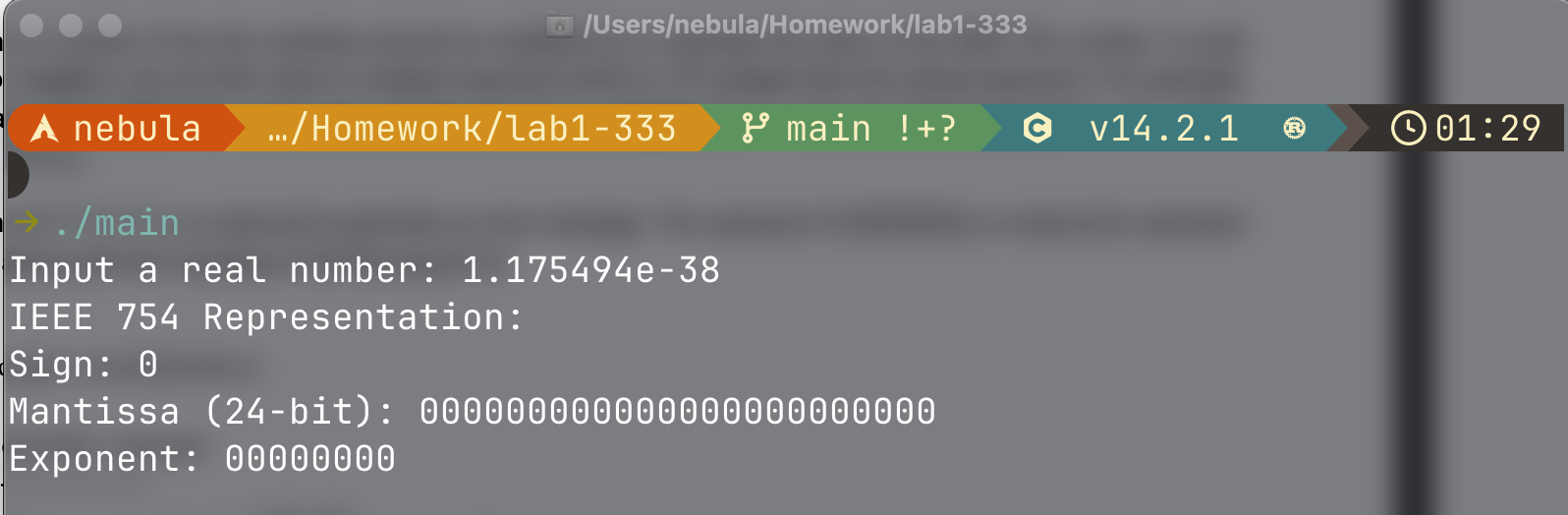
The biggest positive number possible: about 3.402823e+38.

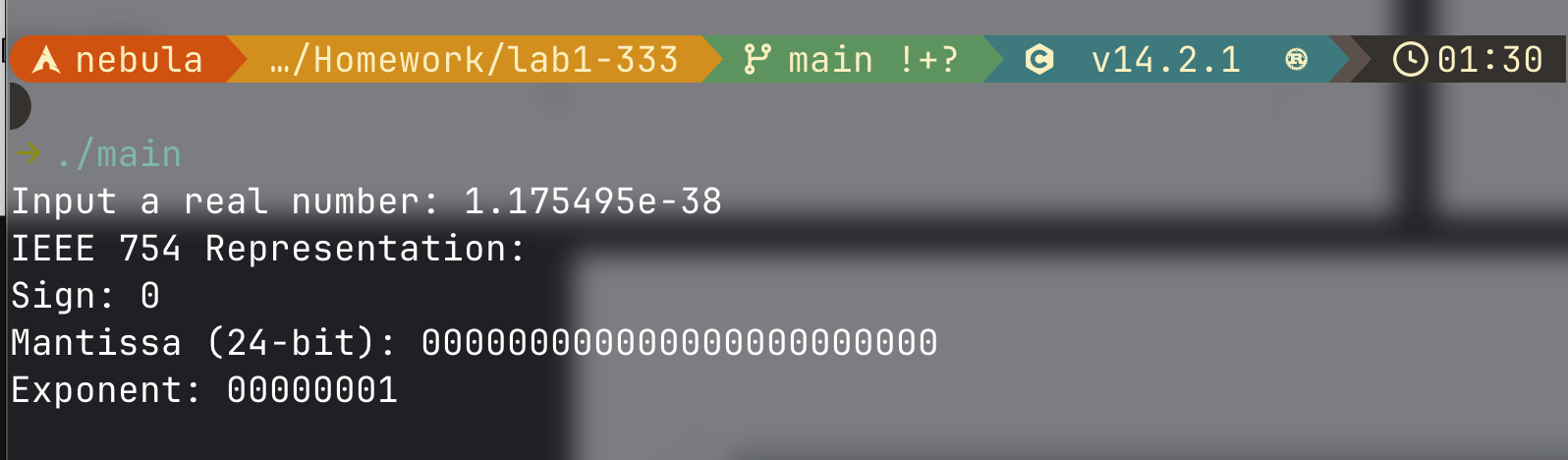




It overflows if it’s larger then the biggest number possible.

The smallest positive number possible: about 1.175494e−38.





Code:

/\*

\* Author: Gao Junran

\* Date: 2024-03-26

\* Description: This program converts a floating-point number into its IEEE 754 single-precision representation.

\* The program has been tested against https://baseconvert.com/ieee-754-floating-point for accuracy.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#define MANTISSA\_SIZE 24

#define EXPONENT\_SIZE 8

#define MAX\_SIZE 200

// Structure to store IEEE 754 representation

typedef struct {

char sign[2]; // Sign bit (0 for positive, 1 for negative)

char mantissa[MANTISSA\_SIZE + 1]; // 24-bit Mantissa

char exponent[EXPONENT\_SIZE + 1]; // 8-bit Exponent

} F32Repr;

/\*

\* Function: pad\_zero

\* -------------------

\* Pads a binary string with leading or trailing zeros to ensure a fixed size.

\*

\* source: The input binary string.

\* target: The output string with padding applied.

\* size: The fixed size to pad to.

\* isLeading: If 1, pads with leading zeros; if 0, pads with trailing zeros.

\*/

void pad\_zero(char \*source, char \*target, int size, int isLeading) {

int len = strlen(source);

if (len >= size) {

strcpy(target, source);

return;

}

int padCount = size - len;

if (isLeading) {

// Add leading zeros

for (int i = 0; i < padCount; i++) {

target[i] = '0';

}

strcpy(target + padCount, source);

} else {

// Copy source and add trailing zeros

strcpy(target, source);

for (int i = 0; i < padCount; i++) {

target[len + i] = '0';

}

target[size] = '\0';

}

}

/\*

\* Function: int\_to\_binary

\* ------------------------

\* Converts an integer to its binary representation.

\*

\* num: The integer to convert.

\* binary: The output binary string.

\*/

void int\_to\_binary(long long int num, char \*binary) {

int index = 0;

char temp[MAX\_SIZE] = {0};

while (num > 0) {

temp[index++] = (num % 2) + '0';

num /= 2;

}

int len = index;

for (int i = 0; i < len; i++) {

binary[i] = temp[len - i - 1]; // Reverse the binary string

}

binary[len] = '\0';

}

/\*

\* Function: dec\_to\_binary

\* ------------------------

\* Converts a fractional decimal to its binary representation.

\*

\* dec: The fractional part of a number.

\* binary: The output binary string.

\*/

void dec\_to\_binary(long double dec, char \*binary) {

int index = 0;

if (dec == 0.0) {

binary[index++] = '0';

}

while (dec > 0.0 && index < 32) { // Limit to 32-bit precision

dec \*= 2.0;

if (dec >= 1.0) {

binary[index++] = '1';

dec -= 1.0;

} else {

binary[index++] = '0';

}

}

binary[index] = '\0';

}

/\*

\* Function: calculate\_mantissa

\* -----------------------------

\* Computes the mantissa from integer and fractional binary parts.

\*

\* int\_binary: Binary representation of the integer part.

\* dec\_binary: Binary representation of the decimal part.

\* mantissa: The output mantissa string.

\*/

void calculate\_mantissa(char \*int\_binary, char \*dec\_binary, char \*mantissa) {

char binary[MAX\_SIZE] = {0};

strcpy(binary, int\_binary);

strcat(binary, dec\_binary);

// Normalize by removing leading zeros

while (binary[0] == '0' && binary[1] != '\0') {

memmove(binary, binary + 1, strlen(binary));

}

if (binary[0] == '\0') {

strcpy(binary, "1");

}

pad\_zero(binary, mantissa, MANTISSA\_SIZE, 0); // Pad with trailing zeros

mantissa[MANTISSA\_SIZE] = '\0';

}

/\*

\* Function: calculate\_exponent

\* -----------------------------

\* Computes the exponent value for IEEE 754 representation.

\*

\* real: The input floating-point number.

\* exponent: The output exponent string.

\*/

void calculate\_exponent(long double real, char \*exponent) {

int exp\_val = 0;

if (real >= 1.0) {

while (real >= 2.0) {

real /= 2.0;

exp\_val++;

}

} else {

while (real < 1.0) {

real \*= 2.0;

exp\_val--;

}

}

exp\_val += 127; // Biasing exponent

if (exp\_val > 255) {

exp\_val = 255; // Clamp overflow

}

char temp[MAX\_SIZE];

int\_to\_binary(exp\_val, temp);

pad\_zero(temp, exponent, EXPONENT\_SIZE, 1); // Pad with leading zeros

}

int main() {

printf("Input a real number: "); fflush(stdout);

long double real;

scanf("%Lf", &real);

F32Repr repr;

repr.sign[0] = (real >= 0.0) ? '0' : '1'; // Determine sign bit

repr.sign[1] = '\0';

real = fabs(real);

long long int int\_part = (long long int) real;

long double dec\_part = real - int\_part;

char int\_binary[MAX\_SIZE] = {0}, dec\_binary[MAX\_SIZE] = {0};

int\_to\_binary(int\_part, int\_binary);

dec\_to\_binary(dec\_part, dec\_binary);

calculate\_mantissa(int\_binary, dec\_binary, repr.mantissa);

calculate\_exponent(real, repr.exponent);

if (strcmp(repr.exponent, "11111111") == 0) { // Check for overflow

memset(repr.mantissa, '0', MANTISSA\_SIZE);

repr.mantissa[MANTISSA\_SIZE] = '\0';

printf("Overflow!!!\n");

}

printf("IEEE 754 Representation:\nSign: %s\nMantissa (24-bit): %s\nExponent: %s\n", repr.sign, repr.mantissa, repr.exponent);

return 0;

}