VNUHCM-UNIVERSITY OF SCIENCE

FACULTY OF INFORMATION TECHNOLOGY

CSC10009 – COMPUTER SYSTEM

Binary arithmetic

Lecturer
Mr. Lê Quốc Hòa

Class
23CLC08

Students
23127255 - Nguyễn Thọ Tài

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHTN
ĐHQG-HCM

October 18th, 2024

Contents

1 Preface

1.1 About

This project implements basic arithmetic operations (addition, subtraction, multiplication, and division) on 8-bit integers using bitwise operations in C++. The integers are first converted into a bitset (array of bits) and then various operations are performed directly on these bitsets. The project demonstrates low-level bitwise manipulation, two's complement representation for negative numbers, and implements custom functions to handle shifting, addition, subtraction, multiplication, and division.

The purpose of the project is to understand how arithmetic operations can be implemented at the bit level and to gain insights into how computers handle integer arithmetic internally.

1.2 Environment

1.2.1 Stats

- Programming language: C++11
- IDE: CLion
- Build: G++
- OS: Windows OS

1.2.2 Build

g++.exe main.cpp -o main
Then run "**./main**" (Linux) or "**main.exe**" (Windows)

1.3 Work Percentage

Task	**Description**	**Percentage**
Input	Getting input from console and validate if the input is in the range (-128, 127)	100%
Get bit & store it	Get bits of input then store it to bitset (char[8]) out those two	100%
Addition	Add 2 signed numbers (2' complemented)	100%
	Result is still in range (-128, 127) when overflowed	100%
Subtract	Subtract 2 signed numbers by turn the second bitset to its opposite sign by 2's complemented it	100%
Multiply	Convert two bitset to their positive, then multiply them normally	100%
	Update the result to the correct sign after-match	100%
Divide	Convert two bitset to their positive, then perform a long divide algorithm	100%
	Dealing with remainder for modulo funionality	100%
	Update the result to the correct sign after-match	100%

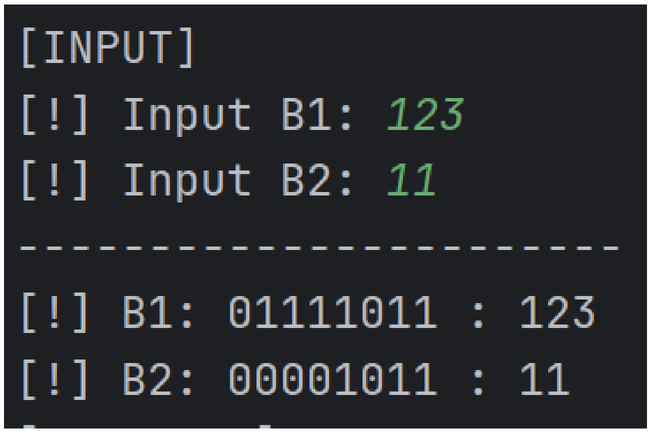
2 Operator

2.1 Converting int8 to bitset (char[8])

The conversion from an 8-bit integer (signed char) to a bitset (array of bits) is handled by the `toBitset(char x)` function. This function iterates through each bit of the integer, shifting and masking the bits to fill the array of size 8.

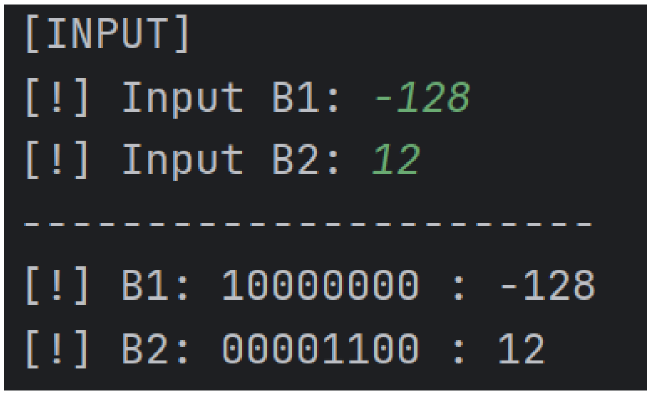
- **Purpose**: To represent an 8-bit integer as an array of bits (`char[8]`), allowing us to perform bitwise operations on it.
- **Operation**: The function shifts the integer `x` by `i` positions to the right, checks the least significant bit, and assigns it to the corresponding position in the array.
- **Example**:

```
char x = 5;
char* bitset = toBitset(x);
// bitset will be [1, 0, 1, 0, 0, 0, 0, 0]
```



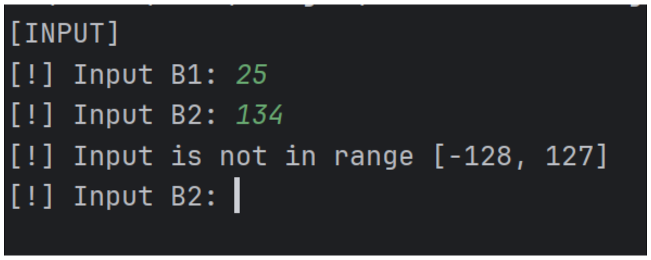
```
[INPUT]
[!] Input B1: 123
[!] Input B2: 11
------------------------
[!] B1: 01111011 : 123
[!] B2: 00001011 : 11
```

Figure 1: Present bits of positive numbers



```
[INPUT]
[!] Input B1: -128
[!] Input B2: 12
------------------------
[!] B1: 10000000 : -128
[!] B2: 00001100 : 12
```

Figure 2: Present bits of negative numbers



```
[INPUT]
[!] Input B1: 25
[!] Input B2: 134
[!] Input is not in range [-128, 127]
[!] Input B2:
```

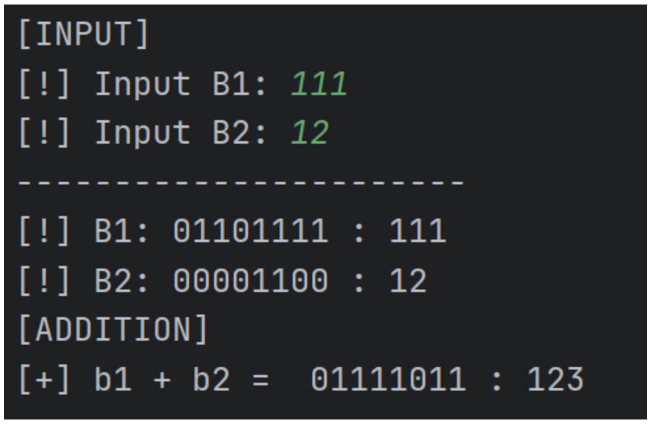
Figure 3: When input is not in range, it requested the user to reinput

2.2 Adding Two Bitsets

The `add` function performs addition on two bitsets representing 8-bit integers. It mimics how addition works in binary using bitwise XOR for summing and bitwise AND to calculate the carry.

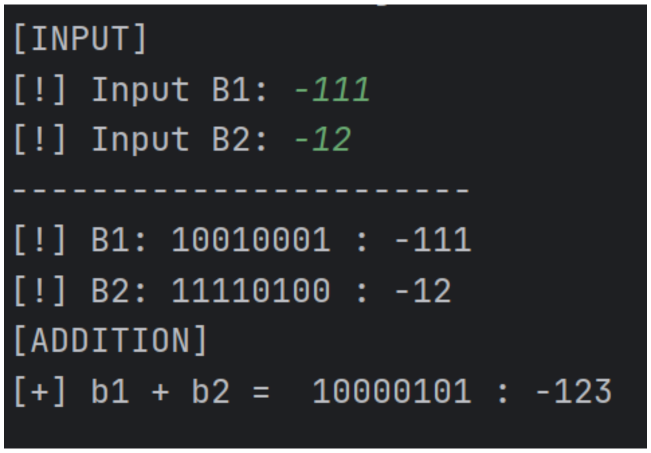
- **Purpose**: To add two numbers represented as bitsets and return their sum.
- **Operation**: The function iterates over each bit, computes the sum for each position using XOR, and calculates the carry using AND.
- **Carry Propagation**: The carry is propagated to the next bit until all bits have been processed.
- **Example**:

```
char* sum = add(bitset1, bitset2);
// sum = bitset1 + bitset2
```



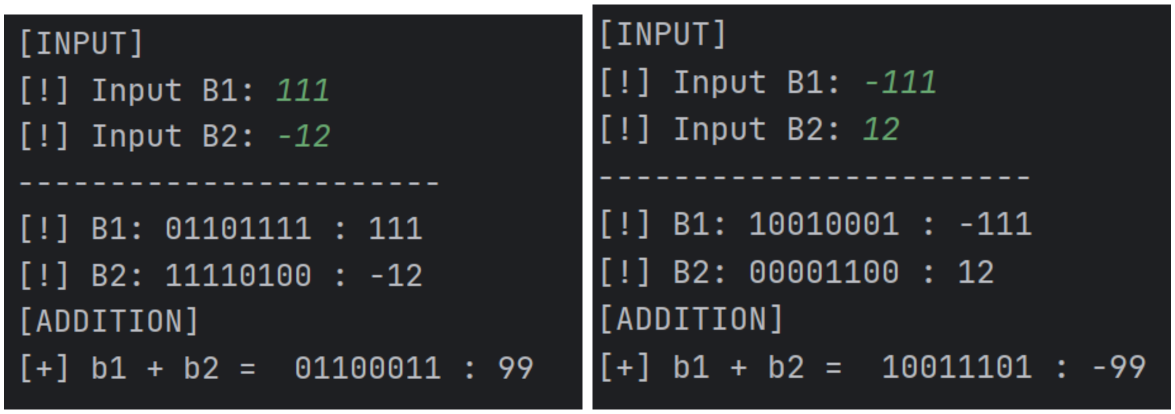
```
[INPUT]
[!] Input B1: 111
[!] Input B2: 12
------------------------
[!] B1: 01101111 : 111
[!] B2: 00001100 : 12
[ADDITION]
[+] b1 + b2 =  01111011 : 123
```

Figure 4: Adding two positive number, 111 & 12



```
[INPUT]
[!] Input B1: -111
[!] Input B2: -12
------------------------
[!] B1: 10010001 : -111
[!] B2: 11110100 : -12
[ADDITION]
[+] b1 + b2 =  10000101 : -123
```

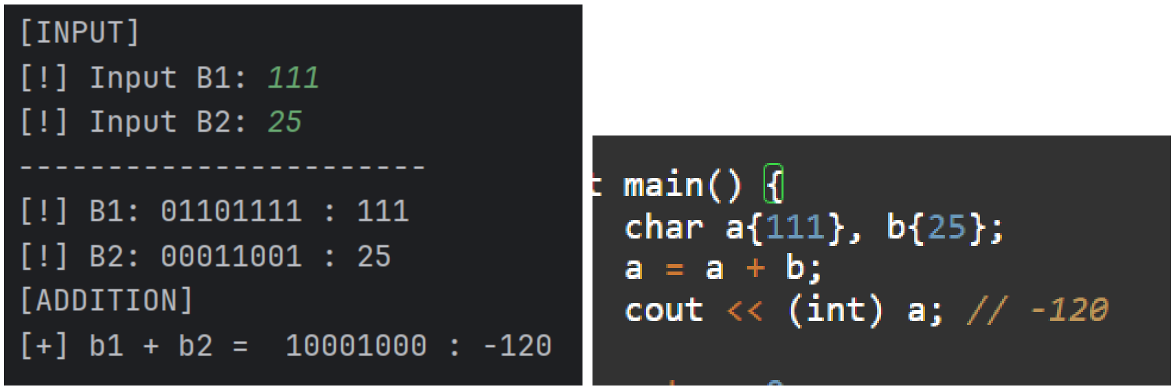
Figure 5: Adding two negative number, -111 & -12



```
[INPUT]
[!] Input B1: 111
[!] Input B2: -12
------------------------
[!] B1: 01101111 : 111
[!] B2: 11110100 : -12
[ADDITION]
[+] b1 + b2 =  01100011 : 99
```

```
[INPUT]
[!] Input B1: -111
[!] Input B2: 12
------------------------
[!] B1: 10010001 : -111
[!] B2: 00001100 : 12
[ADDITION]
[+] b1 + b2 =  10011101 : -99
```

Figure 6: Adding either positive or negetive, -111 + 12 & 111 + -12



```
[INPUT]
[!] Input B1: 111
[!] Input B2: 25
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00011001 : 25
[ADDITION]
[+] b1 + b2 =  10001000 : -120
```

```
t main() {
  char a{111}, b{25};
  a = a + b;
  cout << (int) a; // -120
```

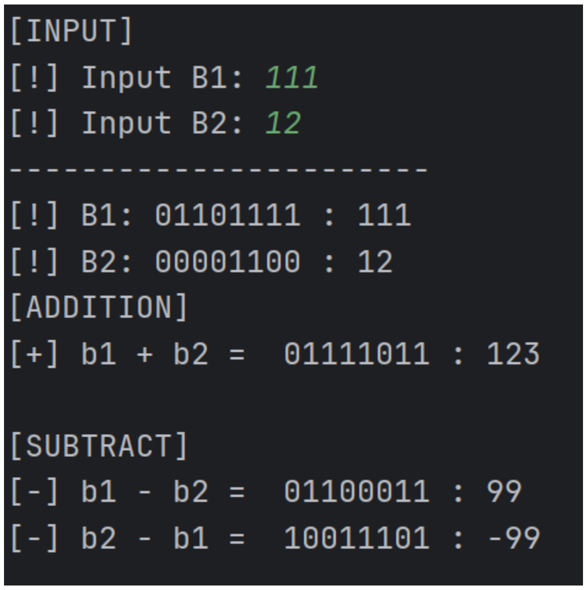
Figure 7: Overflowed result from adding 111 & 25

2.3 Subtracting Two Bitsets

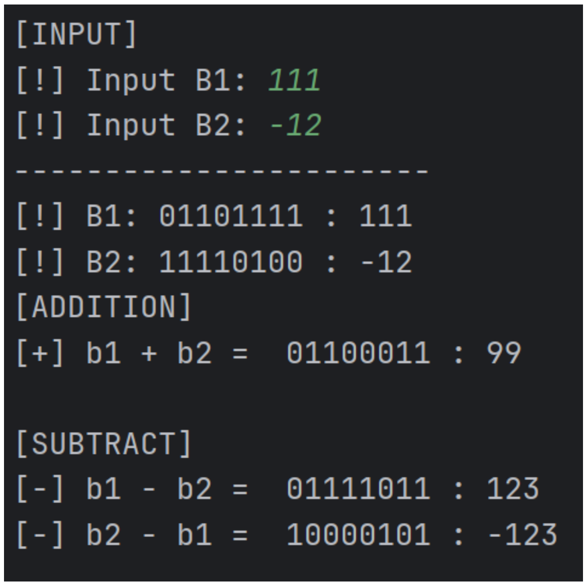
The `sub` function is responsible for subtracting one bitset from another. This is done by first calculating the two's complement of the second bitset (the one to be subtracted) and then adding it to the first bitset.

- **Purpose**: To subtract one bitset from another using the two's complement method.
- **Operation**:
  - Convert the second operand to its two's complement (flip the bits and add 1).
  - Perform binary addition between the first operand and the two's complement of the second operand.
- **Example**:

```
char* result = sub(bitset1, bitset2);
// result = bitset1 - bitset2
```



```
[INPUT]
[!] Input B1: 111
[!] Input B2: 12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00001100 : 12
[ADDITION]
[+] b1 + b2 =  01111011 : 123

[SUBTRACT]
[-] b1 - b2 =  01100011 : 99
[-] b2 - b1 =  10011101 : -99
```



```
[INPUT]
[!] Input B1: 111
[!] Input B2: -12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 11110100 : -12
[ADDITION]
[+] b1 + b2 =  01100011 : 99

[SUBTRACT]
[-] b1 - b2 =  01111011 : 123
[-] b2 - b1 =  10000101 : -123
```

Figure 8: Subtracting 111 and 12 (left), 111 and -12 (right)

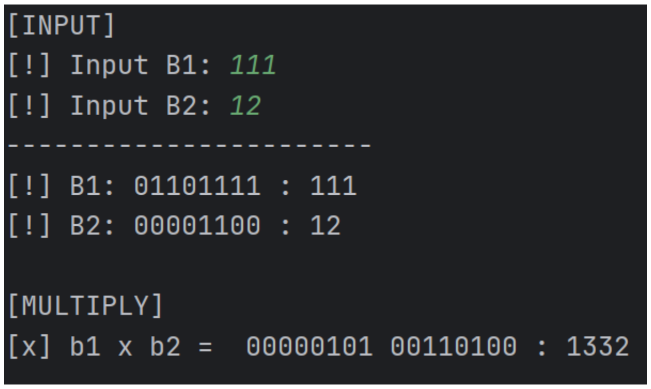
Because subtraction is just addition but the second bitset get 2'complemented, the overflow will behave similarly to addition.

2.4 Multiplying Two Bitsets

The `mul` function implements multiplication of two 8-bit integers using the bitwise shift-and-add algorithm, similar to how long multiplication works in binary. The result is stored in a 16-bit bitset.

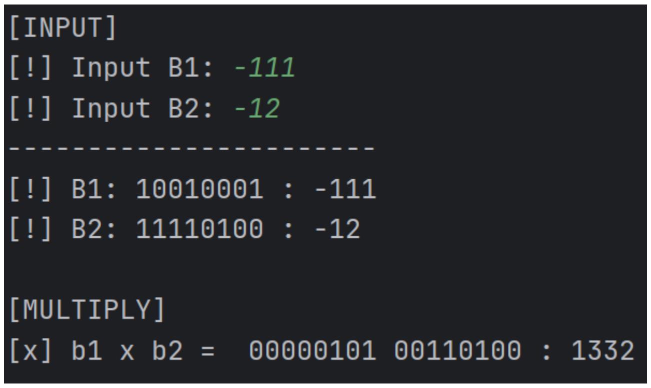
- **Purpose**: To multiply two numbers represented as bitsets.
- **Operation**:
  - Convert both bitsets to positive if either is negative.
  - For each bit in the second bitset, if the bit is set, add the first bitset to the result, appropriately shifted.
  - Finally, adjust the sign of the result based on the signs of the original inputs.
- **Example**:

```
char* product = mul(bitset1, bitset2);
// product = bitset1 * bitset2
```



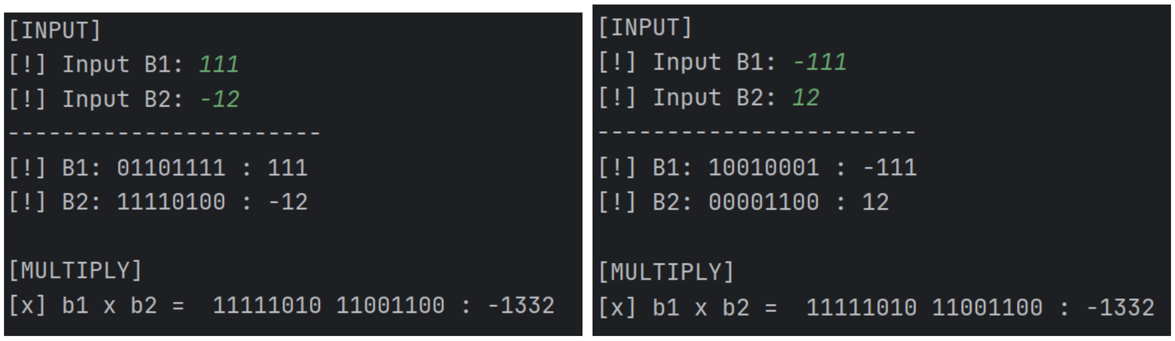
```
[INPUT]
[!] Input B1: 111
[!] Input B2: 12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00001100 : 12

[MULTIPLY]
[x] b1 x b2 =  00000101 00110100 : 1332
```

Figure 9: Multiply 111 with 12 resulted in a 16bit long bitset, and equal to 1332



```
[INPUT]
[!] Input B1: -111
[!] Input B2: -12
-----------------------
[!] B1: 10010001 : -111
[!] B2: 11110100 : -12

[MULTIPLY]
[x] b1 x b2 =  00000101 00110100 : 1332
```

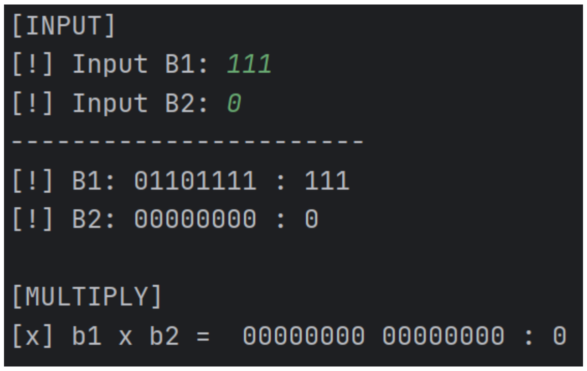
Figure 10: Multiply -111 with -12 resulted in a 16bit long bitset, and equal to 1332



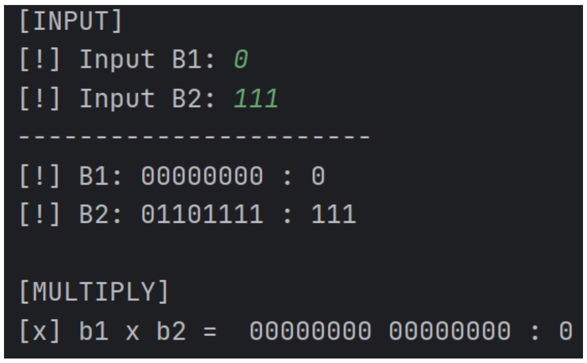
```
[INPUT]
[!] Input B1: 111
[!] Input B2: -12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 11110100 : -12

[MULTIPLY]
[x] b1 x b2 =  11111010 11001100 : -1332
```

```
[INPUT]
[!] Input B1: -111
[!] Input B2: 12
-----------------------
[!] B1: 10010001 : -111
[!] B2: 00001100 : 12

[MULTIPLY]
[x] b1 x b2 =  11111010 11001100 : -1332
```

Figure 11: Multiply -111 & 12 and 111 & -12, both equal to -1332



```
[INPUT]
[!] Input B1: 111
[!] Input B2: 0
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00000000 : 0

[MULTIPLY]
[x] b1 x b2 =  00000000 00000000 : 0
```



```
[INPUT]
[!] Input B1: 0
[!] Input B2: 111
-----------------------
[!] B1: 00000000 : 0
[!] B2: 01101111 : 111

[MULTIPLY]
[x] b1 x b2 =  00000000 00000000 : 0
```

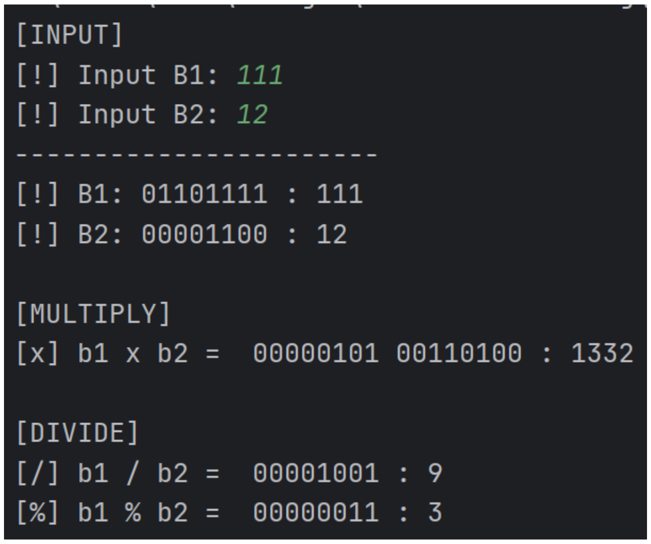
Figure 12: Multiply 111 with 0

2.5 Dividing Two Bitsets and Getting Their Modulo

The `div` function divides two bitsets and returns both the quotient and the remainder (modulo). It uses the long division algorithm in binary, shifting and subtracting until the remainder is smaller than the divisor.[2][1]

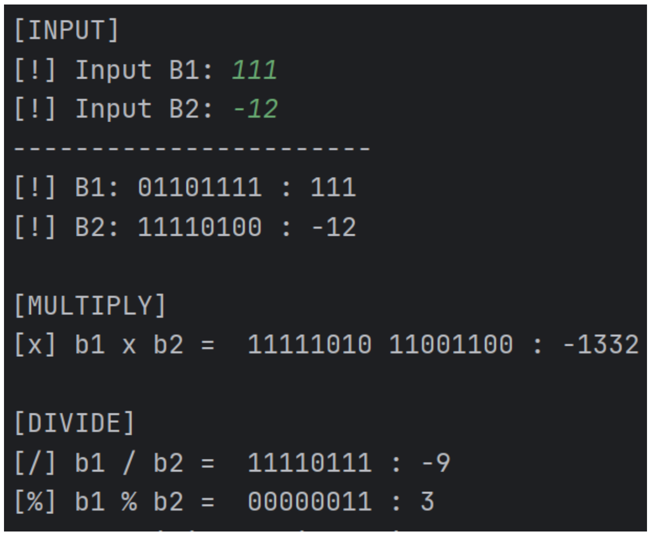
- **Purpose**: To divide two numbers represented as bitsets and return both the quotient and remainder.
- **Operation**:
  - Handle the sign by converting both numbers to positive
  - If dividend is larger than divisor, looping and doing shifting & subtracting until dividend is smaller.
  - While looping, checking if the left-shifting of the divisor copy is shifted to the correct place and the shifted divisor copy still smaller or equal to dividend. When the loop stopped, and doing a little check to right-shift back by 1 if the dividend is smaller than the divisor copy.
  - After the smaller loop, we have the amount of total left-shift - which is the position where we set the bit to 1 in the quotient bitset
  - At the final progress of the outer loop, we subtract the dividend by the divisor copy.
  - The remainder is what left of the dividend.
  - The quotient and remainder are adjusted for the correct sign.
- **Example**:

```
char* quotient = div(bitset1, bitset2, remainder);
// quotient = bitset1 / bitset2
// remainder = bitset1 % bitset2
```



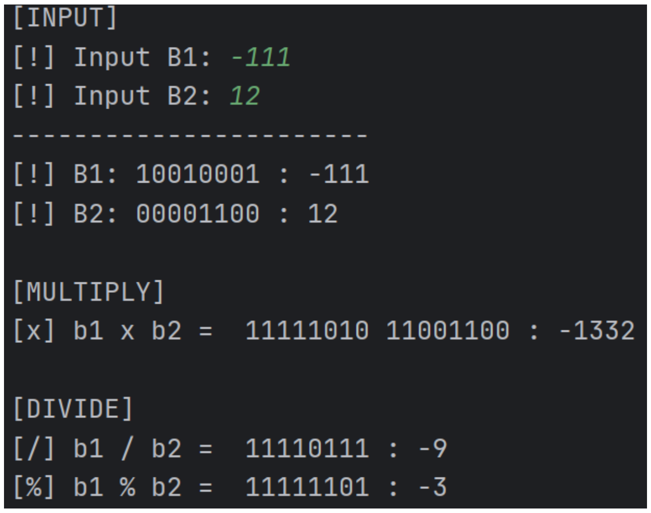
```
[INPUT]
[!] Input B1: 111
[!] Input B2: 12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00001100 : 12

[MULTIPLY]
[x] b1 x b2 =  00000101 00110100 : 1332

[DIVIDE]
[/] b1 / b2 =  00001001 : 9
[%] b1 % b2 =  00000011 : 3
```

Figure 13: Divide 111 by 12



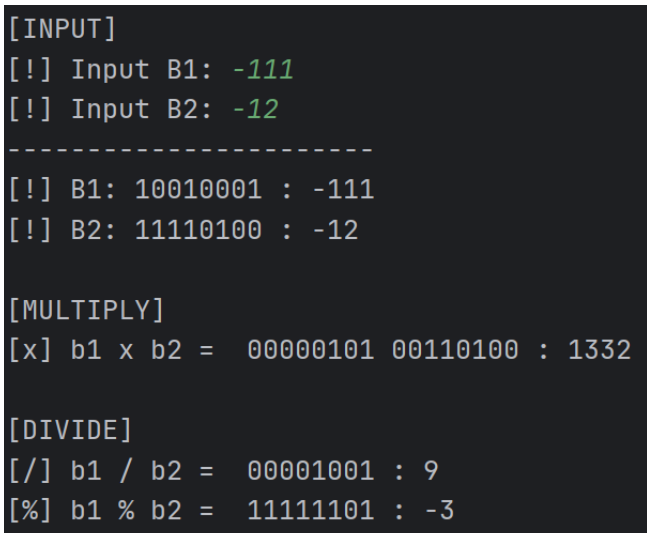
```
[INPUT]
[!] Input B1: 111
[!] Input B2: -12
-----------------------
[!] B1: 01101111 : 111
[!] B2: 11110100 : -12

[MULTIPLY]
[x] b1 x b2 =  11111010 11001100 : -1332

[DIVIDE]
[/] b1 / b2 =  11110111 : -9
[%] b1 % b2 =  00000011 : 3
```

Figure 14: Divide 111 by -12



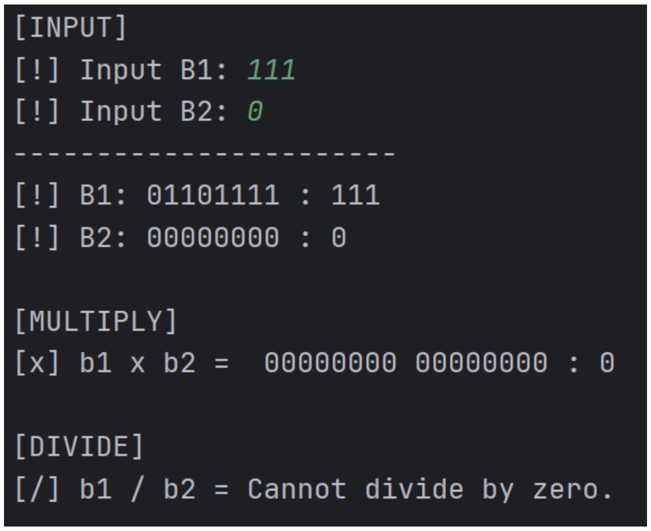
```
[INPUT]
[!] Input B1: -111
[!] Input B2: 12
-----------------------
[!] B1: 10010001 : -111
[!] B2: 00001100 : 12

[MULTIPLY]
[x] b1 x b2 =  11111010 11001100 : -1332

[DIVIDE]
[/] b1 / b2 =  11110111 : -9
[%] b1 % b2 =  11111101 : -3
```

Figure 15: Divide -111 by 12



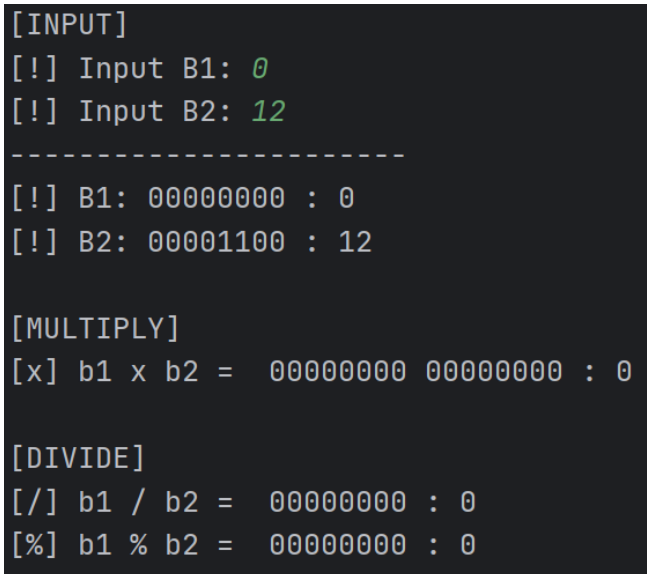
```
[INPUT]
[!] Input B1: -111
[!] Input B2: -12
-----------------------
[!] B1: 10010001 : -111
[!] B2: 11110100 : -12

[MULTIPLY]
[x] b1 x b2 =  00000101 00110100 : 1332

[DIVIDE]
[/] b1 / b2 =  00001001 : 9
[%] b1 % b2 =  11111101 : -3
```

Figure 16: Divide -111 by -12



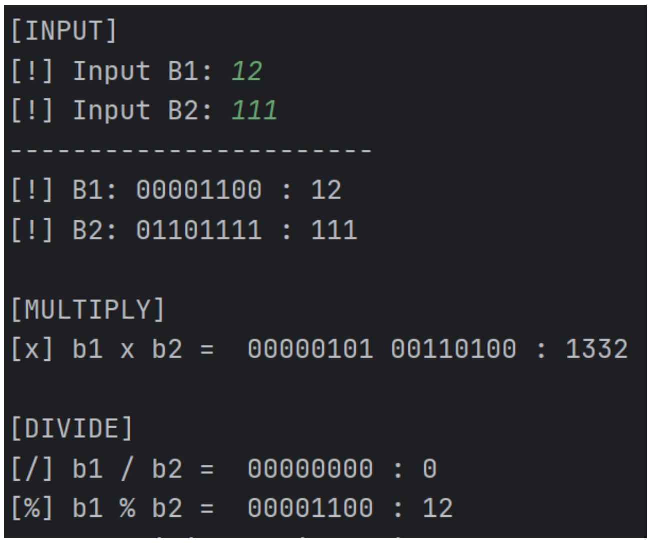
```
[INPUT]
[!] Input B1: 111
[!] Input B2: 0
-----------------------
[!] B1: 01101111 : 111
[!] B2: 00000000 : 0

[MULTIPLY]
[x] b1 x b2 =  00000000 00000000 : 0

[DIVIDE]
[/] b1 / b2 = Cannot divide by zero.
```

Figure 17: Divide by 0 resulted in a message saying "Cannot divide by zero."



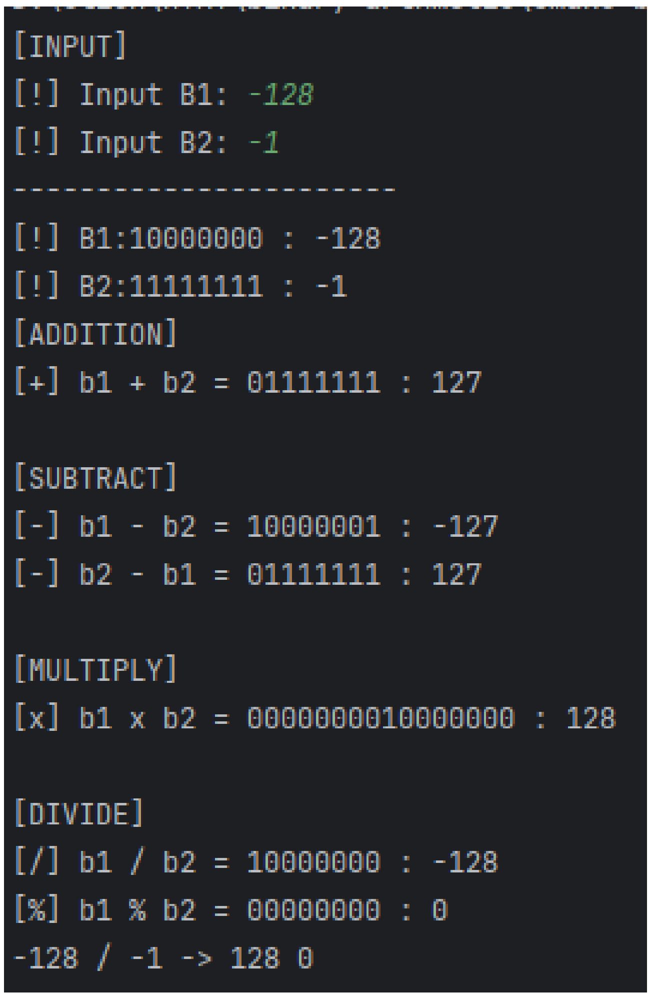
```
[INPUT]
[!] Input B1: 0
[!] Input B2: 12
-----------------------
[!] B1: 00000000 : 0
[!] B2: 00001100 : 12

[MULTIPLY]
[x] b1 x b2 =  00000000 00000000 : 0

[DIVIDE]
[/] b1 / b2 =  00000000 : 0
[%] b1 % b2 =  00000000 : 0
```

Figure 18: Dividend is zero.



```
[INPUT]
[!] Input B1: 12
[!] Input B2: 111
-----------------------
[!] B1: 00001100 : 12
[!] B2: 01101111 : 111

[MULTIPLY]
[x] b1 x b2 =  00000101 00110100 : 1332

[DIVIDE]
[/] b1 / b2 =  00000000 : 0
[%] b1 % b2 =  00001100 : 12
```

Figure 19: Divisor is larger than dividend resulted in remainder to be the dividend and the quotient to be 0



```
[INPUT]
[!] Input B1: -128
[!] Input B2: -1
-----------------------
[!] B1:10000000 : -128
[!] B2:11111111 : -1
[ADDITION]
[+] b1 + b2 = 01111111 : 127

[SUBTRACT]
[-] b1 - b2 = 10000001 : -127
[-] b2 - b1 = 01111111 : 127

[MULTIPLY]
[x] b1 x b2 = 0000000010000000 : 128

[DIVIDE]
[/] b1 / b2 = 10000000 : -128
[%] b1 % b2 = 00000000 : 0
-128 / -1 -> 128 0
```

Figure 20: The result is overflowed when dividend is -128 and divisor is -1. The result would be 128 and 0 but 128 got overflowed into -128

References

[1] Donald E Knuth. *The Art of Computer Programming*. Vol. 2: Seminumerical Algorithms Section 4.3.1: The Classical Algorithms. Addison-Wesley Professional, Apr. 1998.

[2] Luis Llamas. *How to divide integers in binary*. URL: `https://www.luisllamas.es/en/multiply-or-divide-binary/#binary-integer-division`.