# Lab 2: Manipulating Bits

[Lab 2: Manipulating Bits](#lab-2-manipulating-bits)  
 [1. Introduction](#Xe3d0fc0bea9a42ce7605565d0964033d7f6ee47)  
 [2. Lab Instructions or Steps](#X2e3707871fdaad114dfcbba591dd899555cc4fc)  
 [2.1 Step 1: Read the following instructions carefully](#Xad66b05824effcde1c6cedaf58dfd2cbde8a428)  
 [2.2 Step 2: Modify the following functions according the coding rules](#Xc67a25953e4cde5bbe22e5791315924e632a887)  
 [the pow2plus4 function](#the-pow2plus4-function)  
 [the bitAnd function](#the-bitand-function)  
 [the getByte function](#the-getbyte-function)  
 [the negate function](#the-negate-function)  
 [the isPositive function](#the-ispositive-function)  
 [3.Lab Device and Environment](#Xf63aca51f9a082ef37ee3a066bb20a9e10f6794)  
 [3.1Environment](#Xa65688011ab8ac028e996e754c6ea2091b24c2c)  
 [3.2 Lab Device](#Xa31b8a3f0121c6d19eebdbef2035a6a6277f37c)  
 [4. Results and Analysis](#X3032043b1260111f9b92ee54473079693240f69)  
 [4.1 Results](#X2eead686e7828242b99e884afcccf3c99671182)  
 [4.2 Analysis](#Xca492b8164e9849433fddf86eb2846b61267159)  
 [5.Appendix](#Xa718826a98947f221248872ee84a93c51a35c6d)

## 1. Introduction

The purpose of this lab is to become more familiar with bit-level representations and arithmetic of integers in chapter 2. We should do this by solving a series of programming puzzles.

## 2. Lab Instructions or Steps

Start by copying *bits.c* to a directory on a Linux machine in which you plan to do your work. You will be modifying this file.

The *bits.c* file contains a skeleton for each of the 5 programming puzzles. Your assignment is to complete each function skeleton using only straightline code for the integer puzzles (i.e., no loops or conditionals) and a limited number of C arithmetic and logical operators. Specifically, you are only allowed to use the following eight operators:

! ~ & ^ | + << >>

### 2.1 Step 1: Read the following instructions carefully

**CODING RULES:**

Replace the "return" statement in each function with one or more lines of C code that implements the function. Your code must conform to the following style:

int Funct(arg1, arg2, ...) {
  
 /\* brief description of how your implementation works \*/
  
 int var1 = Expr1;
  
 ...
  
 int varM = ExprM;
  
   
 varJ = ExprJ;
  
 ...
  
 varN = ExprN;
  
 return ExprR;
  
}

Each "Expr" is an expression using ONLY the following:

1. Integer constants 0 through 255 (0xFF), inclusive. You are not allowed to use big constants such as 0xffffffff.
2. Function arguments and local variables (no global variables).
3. Unary integer operations ! ~
4. Binary integer operations & ^ | + << >>

Some of the problems restrict the set of allowed operators even further.

Each "Expr" may consist of multiple operators. You are not restricted to

one operator per line.

You are expressly forbidden to:

1. Use any control constructs such as if, do, while, for, switch, etc.
2. Define or use any macros.
3. Define any additional functions in this file.
4. Call any functions.
5. Use any other operations, such as &&, ||, -, or ?:
6. Use any data type other than int. This implies that you

cannot use arrays, structs, or unions.

You may assume that your machine:

1. Performs right shifts arithmetically.
2. Has unpredictable behavior when shifting an integer by more

than the word size.

1. Uses 32-bit representations of integers.

### 2.2 Step 2: Modify the following functions according the coding rules

#### the pow2plus4 function

pow2plus4 - returns 2^x + 4, where 0 <= x <= 31

Legal ops: + <<\*

#### the bitAnd function

bitAnd - x&y using only ~ and |

Example: bitAnd(6, 5) = 4

Legal ops: ~ |

#### the getByte function

getByte - Extract byte n from word x

Bytes numbered from 0 (LSB) to 3 (MSB)

Examples: getByte(0x12345678,1) = 0x56

Legal ops: & << >>

#### the negate function

negate - return -x

Example: negate(1) = -1

Legal ops: ~ +

#### the isPositive function

isPositive - return 1 if x > 0, return 0 otherwise

Example: isPositive(-1) = 0

Legal ops: ! | >>

## 3.Lab Device and Environment

### 3.1Environment

root@AvA:/mnt/c/Users/XuanQuan# cat /proc/version  
Linux version 5.10.16.3-microsoft-standard-WSL2 (oe-user@oe-host) (x86\_64-msft-linux-gcc (GCC) 9.3.0, GNU ld (GNU Binutils) 2.34.0.20200220) #1 SMP Fri Apr 2 22:23:49 UTC 2021

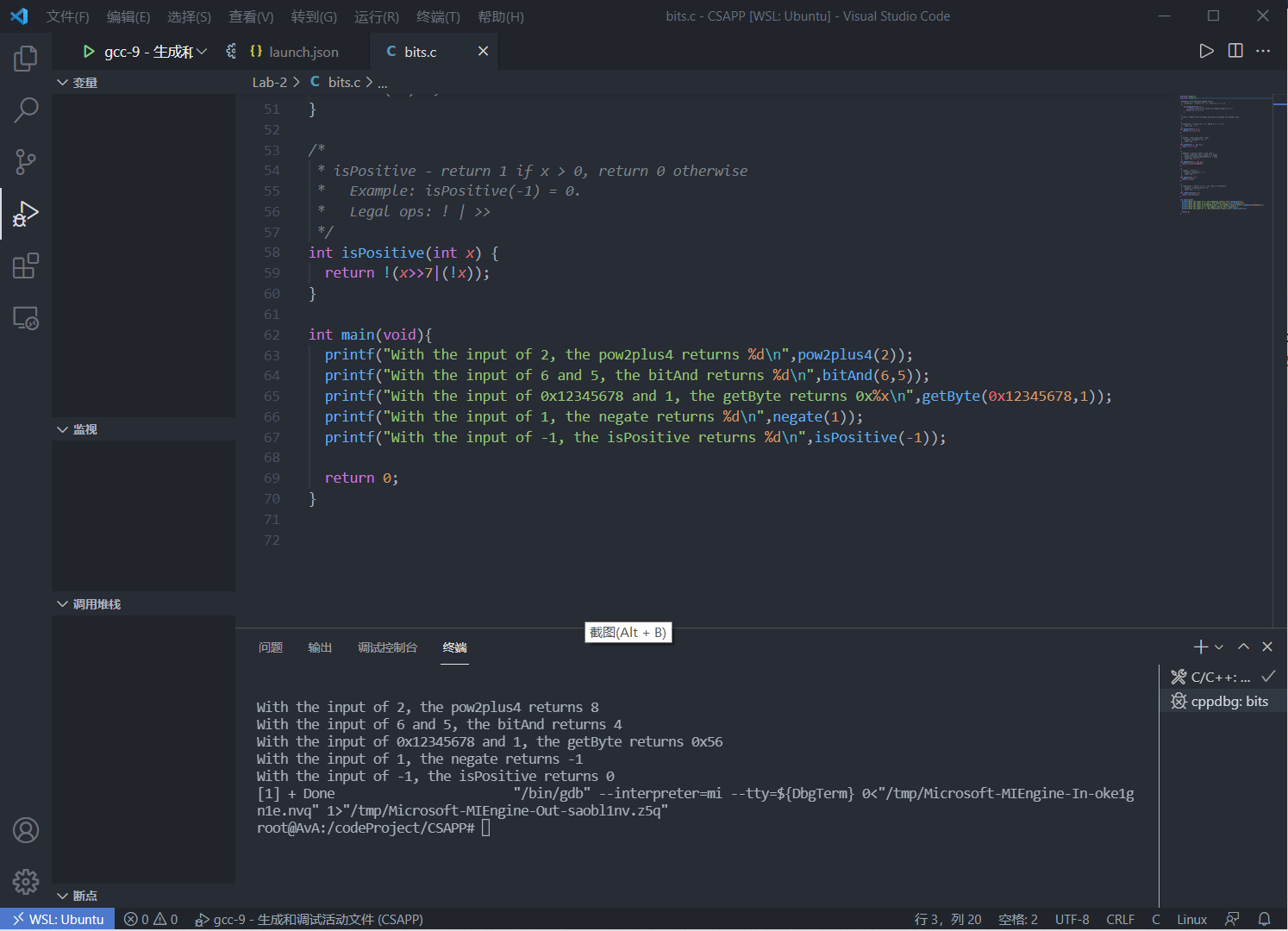
### 3.2 Lab Device

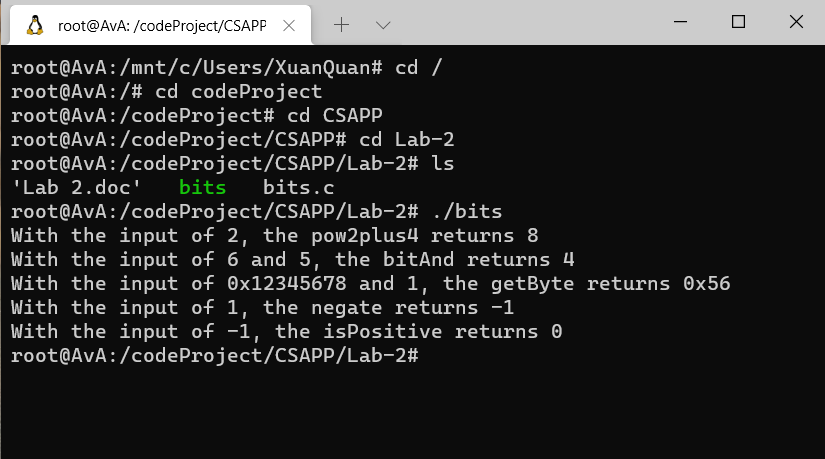
处理器 Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz 1.99 GHz  
机带 RAM 16.0 GB (15.8 GB 可用)  
系统类型 64 位操作系统, 基于 x64 的处理器

版本 Windows 10 家庭版  
版本号 21H1  
安装日期 ‎2021/‎7/‎30  
操作系统内部版本 19043.1266

## 4. Results and Analysis

### 4.1 Results





### 4.2 Analysis

The first function ask us to multiply x with 2 and add 4 with the operator + and <<, because the 1 << x is equivalent to 2^x , so the answer is (1<<x)+4

The second function ask us to realize AND(&) by using OR(|) and NOT(~). Analyzing the example, the binary representation of 6 and 5 is 0110 and 0101, we get 0100 which represents 4. Take the NOT results of 0110 and 0101 is 1001 and 1010 and we do a OR operation and get 1011, noting that is the NOT results of 0100, so we do a NOT operation and finally we get 0100.

The third function ask us to get byte n from word x, one byte is 8 bits, so we need to multiply n by 8, which is n<<3. And we right shift x by n<<3 to get rid of the bytes in the right side of n byte. To deal with the left side of n bye, we do an AND operation with 0xFF, and only the n byte & 0xFF can get 1, so the left side is removed.

The fourth function ask us to get the -x. Analyzing the example, the binary representation of 1 is 0001 and for -1 is 1111. Noting that the NOT result of 1 is 1110, so we add 1 to the binary representation we get 1111.

The fifth function ask us to judge whether x is positive. !x will return 1 when x is zero, otherwise it will return 0. If the highest order of binary representations is 1,that numeral is negative. Otherwise it is positive or zero. So we do x>>7. To separate zero from the result, we do a OR operation between x>>7 and !x. When the numeral is positive , it will return 0. Otherwise it will return 1 and we do a NOT operator finally.

## 5.Appendix

the program code bits.c is

#include <stdio.h>
  
   
/\*EXAMPLES OF ACCEPTABLE CODING STYLE:
  
 \* pow2plus1 - returns 2^x + 1, where 0 <= x <= 31
  
 \*/
  
 int pow2plus1(int x) {
  
 /\* exploit ability of shifts to compute powers of 2 \*/
  
 return (1 << x) + 1;
  
 }
  
   
/\*
  
 \* STEP 2: Modify the following functions according the coding rules.
  
 \*/
  
   
/\*
  
 \* pow2plus4 - returns 2^x + 4, where 0 <= x <= 31
  
 \* Legal ops: + <<
  
 \*/
  
int pow2plus4(int x) {
  
 return (1 << x) + 4;
  
}
  
   
/\*
  
 \* bitAnd - x&y using only ~ and |
  
 \* Example: bitAnd(6, 5) = 4
  
 \* Legal ops: ~ |
  
 \*/
  
int bitAnd(int x, int y) {
  
 return ~(~x | ~y);
  
}
  
   
/\*
  
 \* getByte - Extract byte n from word x
  
 \* Bytes numbered from 0 (LSB) to 3 (MSB)
  
 \* Examples: getByte(0x12345678,1) = 0x56
  
 \* Legal ops: & << >>
  
 \*/
  
int getByte(int x, int n) {
  
 return (x>>(n<<3))&0xff;
  
}
  
   
/\*
  
 \* negate - return -x
  
 \* Example: negate(1) = -1.
  
 \* Legal ops: ~ +
  
 \*/
  
int negate(int x) {
  
 return (~x)+1;
  
}
  
   
/\*
  
 \* isPositive - return 1 if x > 0, return 0 otherwise
  
 \* Example: isPositive(-1) = 0.
  
 \* Legal ops: ! | >>
  
 \*/
  
int isPositive(int x) {
  
 return !(x>>7|(!x));
  
}
  
   
int main(void){
  
 printf("With the input of 2, the pow2plus4 returns %d\n",pow2plus4(2));
  
 printf("With the input of 6 and 5, the bitAnd returns %d\n",bitAnd(6,5));
  
 printf("With the input of 0x12345678 and 1, the getByte returns 0x%x\n",getByte(0x12345678,1));
  
 printf("With the input of 1, the negate returns %d\n",negate(1));
  
 printf("With the input of -1, the isPositive returns %d\n",isPositive(-1));
  
   
 return 0;
  
}