天津大学

计算机系统基础上机实验报告

实验题目 1：位操作 bit-ops

学院名称 智能与计算学部

专　　业 工科试验班

学生姓名 张明君

学　　号 6319000359

年　　级 2019 级

班　　级 留学生班

时　　间 2020 年 6 月 19 日

实验 1：位操作

Bit Operations

1. 实验目的

进一步理解书中第二章《信息的表示和处理》部分的内容，深刻理解整数、浮点数的表示 和运算方法，掌握 GNU GCC 工具集的基本使用方法。

2. 实验内容

请按照要求补全 bits.c 中的函数，并进行验证。包括以下 6 个函数：理解

No	函数定义	说明

1	`int isAsciiDigit(int x)`	`/* isAsciiDigit - return 1 if`
		` *  0x30 <= x <= 0x39`
		` * (ASCII codes for characters '0' to '9')`
		` *  Example: isAsciiDigit(0x35) = 1.`
		` *  isAsciiDigit(0x3a) = 0.`
		` *  isAsciiDigit(0x05) = 0.`
		` *  Legal ops: ! ~ & ^ | + << >>`
		` *  Max ops: 15`
		` *  Rating: 3`
		` */`

2	`int anyEvenBit(int x)`	`/*`
		` * anyEvenBit - return 1 if any even-numbered`
		` * bit in word set to 1`
		` * Examples: anyEvenBit(0xA) = 0,`
		` *  anyEvenBit(0xE) = 1`
		` *  Legal ops: ! ~ & ^ | + << >>`
		` *  Max ops: 12`
		` *  Rating: 2`
		` */`

3	`int copyLSB(int x)`	`/*`
		` * copyLSB - set all bits of result to least`
		` * significant bit of x`
		` *  Example: copyLSB(5) = 0xFFFFFFFF,`
		` *  copyLSB(6) = 0x00000000`
		` *  Legal ops: ! ~ & ^ | + << >>`
		` *  Max ops: 5`
		` *  Rating: 2`

No	函数定义	说明
		*/
4	int leastBitPos(int x)	/* * leastBitPos - return a mask that marks * the position of the * least significant 1 bit. If x == 0, * return 0 * Example: leastBitPos(96) = 0x20 * Legal ops: ! ~ & ^ | + << >> * Max ops: 6 * Rating: 2 */
5	int divpwr2(int x,int n)	/* * divpwr2 - Compute x/(2^n), for 0 <= n <= 30 * Round toward zero * Examples: divpwr2(15,1) = 7, * divpwr2(-33,4) = -2 * Legal ops: ! ~ & ^ | + << >> * Max ops: 15 * Rating: 2 */
6	int bitCount(int x)	/* * bitCount - returns count of number of * 1's in word * Examples: bitCount(5) = 2, bitCount(7) = 3 * Legal ops: ! ~ & ^ | + << >> * Max ops: 40 * Rating: 4 */

3. 实验要求

1）在 Unbuntu18.04LTS 操作系统下，按照实验指导说明书，使用 gcc 工具集编译程序和测试

2）代码符合所给框架代码的规范（详见 bits.c 的开始位置注释内容）

3）需提交：源代码 bits.c、电子版实验报告全文。

4）本实验相关要求：注意：违背以下原则均视为程序不正确!!

程序内允许使用：	程序内禁止以下行为：
a. 运算符： ! ~ & ^ | + << >> b. 范围在 0 - 255 之间的常数	a. 声明和使用全局变量 b. 声明和使用定义宏

程序内允许使用：	程序内禁止以下行为：
c. 局部变量	c. 声明和调用其他的函数 d. 类型的强制转换 e. 使用许可范围之外的运算符 f. 使用控制跳转语句：if else switch do while for

4. 实验结果

INTEGER 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 form of casting.
7. Use any data type other than int. This implies that you cannot use arrays, structs, or unions.

You may assume that your machine:
1. Uses 2s complement, 32-bit representations of integers.
2. Performs right shifts arithmetically.
3. Has unpredictable behavior when shifting an integer by more than the word size.

FLOATING POINT CODING RULES

For the problems that require you to implent floating-point operations, the coding rules are less strict. You are allowed to use looping and conditional control. You are allowed to use both ints and unsigneds. You can use arbitrary integer and unsigned constants.

You are expressly forbidden to:
1. Define or use any macros.
2. Define any additional functions in this file.
3. Call any functions.
4. Use any form of casting.
5. Use any data type other than int or unsigned. This means that you cannot use arrays, structs, or unions.
6. Use any floating point data types, operations, or constants.

NOTES:
1. Use the dlc (data lab checker) compiler (described in the handout) to check the legality of your solutions.
2. Each function has a maximum number of operators (! ~ & ^ | + << >>) that you are allowed to use for your implementation of the function. The max operator count is checked by dlc. Note that '=' is not counted; you may use as many of these as you want without penalty.
3. Use the btest test harness to check your functions for correctness.
4. Use the BDD checker to formally verify your functions
5. The maximum number of ops for each function is given in the header comment for each function. If there are any inconsistencies between the maximum ops in the writeup and in this file, consider this file the authoritative source.

```
/*
 * STEP 2: Modify the following functions according the coding rules.
 *
 *     IMPORTANT. TO AVOID GRADING SURPRISES:
 *     1. Use the dlc compiler to check that your solutions conform
 *          to the coding rules.
 *     2. Use the BDD checker to formally verify that your solutions produce
 *          the correct answers.
 */
#endif
```

1.isAsciiDigit

```
/*
 * isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')
 *     Example: isAsciiDigit(0x35) = 1.
 *                      isAsciiDigit(0x3a) = 0.
 *                      isAsciiDigit(0x05) = 0.
 *     Legal ops: ! ~ & ^ | + << >>
 *     Max ops: 15
 *     Rating: 3
 */
int isAsciiDigit(int x) {
   int up_num = x + (~58 + 1);    //最高位为 1
   int low_num = x + (~48 + 1);      //最高位是 0

   int Is_up_legal = (!((up_num >> 31) + 1));
   int Is_down_legal = ((low_num >> 31) + 1);

   return      (Is_up_legal & Is_down_legal);
}
```

```
In this function, what we should to do is to judge weather the 'x' is bigger
than 48(inclusive) and smaller than 57(inclusive).
  first:
  we mapping the legal-x:
  up_num = x -58;
  low_num = x - 48;
  we will get (up_num < 0) and (low_num >= 0) if the "x" is legal.
  then:
  "(( up_num >> 31) + 1)" will be equal to 0 ,if (up_num < 0).
  "((low_num >> 31) + 1)" will be equal to 1 ,if (low_num >= 0).
```

```
  so,if (up_num < 0), "Is_up_legal" will be equal to 1.if (low_num >=
0),"Is_down_legal" will be equal to 1.
  last:
  only if ((up_num >> 31) + 1) == 0 and ((low_num >> 31) + 1), we return 1;
```

2.anyEvenBit

```
/*
 * anyEvenBit - return 1 if any even-numbered bit in word set to 1
 *      Examples anyEvenBit(0xA) = 0, anyEvenBit(0xE) = 1
 *      Legal ops: ! ~ & ^ | + << >>
 *      Max ops: 12
 *      Rating: 2
 */
int anyEvenBit(int x) {
   int x1 = x >> 8;
   int x2 = x1 >> 8;
   int x3 = x2 >> 8;
   return !!( ( x | x1 | x2 | x3 )& 0x55);
}
```

```
   Every time we move 8 bits, we do a|operation with the result of 3 times of movement
```

```
and X, which is equivalent to overlapping every 8 information to the lowest 8 bits,

At this time, we can know if its even digit is 1 by doing &amp;amp; operation with 01010101
```

3.copyLSB

```
/*
 * copyLSB - set all bits of result to least significant bit of x
 *      Example: copyLSB(5) = 0xFFFFFFFF, copyLSB(6) = 0x00000000
 *      Legal ops: ! ~ & ^ | + << >>
 *      Max ops: 5
 *      Rating: 2
 */
int copyLSB(int x) {
   int temp = x & 1;
   return (~temp + 1);
}
```

```
first, we could get least significant bit by used "x & 1".
  then, we could get result
```

4.leastBitPos

```
/*
 * leastBitPos - return a mask that marks the position of the
 *                       least significant 1 bit. If x == 0, return 0
 *      Example: leastBitPos(96) = 0x20
 *      Legal ops: ! ~ & ^ | + << >>
 *      Max ops: 6
 *      Rating: 2
 */
int leastBitPos(int x) {

   return (~x+1) & x;
}
```

we get the "-x",than "(-x)&x" is what we need.

5.divpwr2

```
/*
 * divpwr2 - Compute x/(2^n), for 0 <= n <= 30
 *    Round toward zero
 *      Examples: divpwr2(15,1) = 7, divpwr2(-33,4) = -2
 *      Legal ops: ! ~ & ^ | + << >>
 *      Max ops: 15
 *      Rating: 2
 */
int divpwr2(int x, int n) {
   int op = (x >> 31) + 1;
   int temp    =      x    >> n;
   int temp2 = temp << n;
   int Is_exact_division = !!(temp2 ^ x);
   return (temp + (       (!op) & (!!n) & (Is_exact_division)    )      );
}
```

In general, the "> > symbol represents X / (2 ^ n) and rounds down (positive and

negative).

So there are three situations to consider:

If it is negative, it needs to be rounded up.

Should be x if n = 0.

If you can divide, you don't need to round.

In this question, OP is the sign sign bit, positive number is 1, negative number is 0

Temp is the result of moving x right by N bits, which is quite direct x / (2 ^ n), rounding down.

Temp2 is the result of moving n bits to the left of temp. With (temp2 ^ x), you can judge whether the bits discarded by temp are all 0 (both divisible).

Is_ exact_ Division means that if it can be divided into 0, otherwise it is 1.

Because > > by default represents rounding down, we need to consider the above situation and use (! OP) & (! N) & (is)_ exact_ Division),

Meaning: satisfy (negative number) at the same time (n! =0) (not divisible) returns 1 in three cases, otherwise 0.

It also means that only when the above three conditions are met can + 1 be obtained on the premise of rounding down, otherwise the rounding down result is the normal rounding result.

6.bitCount

```
/*
 * bitCount - returns count of number of 1's in word
 *      Examples: bitCount(5) = 2, bitCount(7) = 3
 *      Legal ops: ! ~ & ^ | + << >>
 *      Max ops: 40
 *      Rating: 4
 */
int bitCount(int x) {

   int mask = (0x11 <<24) | (0x11 << 16) | (0x11 << 8) |(0x11) ;
   int temp = x & mask;
```

```
    x = x >> 1;
    temp = temp + (x & mask);
    x = x >> 1;
    temp = temp + (x & mask);
    x = x >> 1;
    temp = temp + (x & mask);

    return ((temp & 0xf) +
                ( (temp >> 4) & 0xf )+
                ( (temp >> 8) & 0xf )+
                ( (temp >> 12) & 0xf )+
                ( (temp >> 16) & 0xf )+
                ( (temp >> 20) & 0xf )+
                ( (temp >> 24) & 0xf )+
                ( (temp >> 28) & 0xf ));

}
```

First, it is divided into 4 blocks, each block moves to the right, and the number of 1 in each block is compared.