计算机系统结构(A)

实验 2

Log Creative

2021 年 6 月 27 日

一. make

(1) 本程序的编译使用哪个编译器？

答: gcc。

(2) 采用哪个命令，可以将所有程序全部编译？

答:

```
make all
```

(3) 采用哪个命令，可以将所有上次编译的结果全部删除？

答:

```
make clean
```

(4) 文件中第几行生成btest的目标文件？

答: 第11行：

```
$(CC) $(CFLAGS) $(LIBS) -o btest bits.c btest.c decl.c tests.c
```

(5) 文件中第几行生成fshow的目标文件？

答: 第14行：

```
$(CC) $(CFLAGS) -o fshow fshow.c
```

(6) 如果在Makefile文件中用要引用变量“FOO”，怎么表示？

答:

```
$(FOO)
```

二. 函数执行示例：

```
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$ ./fshow 34.5

Floating point value 34.5
Bit Representation 0x420a0000, sign = 0, exponent = 0x84, fraction = 0x0a0000
Normalized.  +1.0781250000 X 2^(5)
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$ ./ishow 20
Hex = 0x00000014,       Signed = 20,    Unsigned = 20
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$
```

通过了检查测试，并得到了全部分数：

```
logcreative@ubuntu: /mnt/hgfs/VMShared/linux/ComputerAr...
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$ ./dlc
 bits.c
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$ ./bte
st bits.c
Score   Rating  Errors  Function
 2      2       0       allOddBits
 4      4       0       isLessOrEqual
 4      4       0       logicalNeg
 5      5       0       floatScale2
 5      5       0       floatFloat2Int
Total points: 20/20
logcreative@ubuntu:/mnt/hgfs/VMShared/linux/ComputerArch/Project2/datalab$
```

`bits.c`

```c
/*
 * From CS:APP Data Lab
 *
 * Zilong Li - 518070910095
 *
 * bits.c - Source file with your solutions to the Lab.
 *          This is the file you will hand in to your instructor.
 *
 * WARNING: Do not include the <stdio.h> header; it confuses the dlc
 * compiler. You can still use printf for debugging without including
 * <stdio.h>, although you might get a compiler warning. In general,
 * it's not good practice to ignore compiler warnings, but in this
 * case it's OK.
 */

#if 0
/*
 * Instructions to Students:
 *
 * STEP 1: Read the following instructions carefully.
 */

You will provide your solution to the Data Lab by
editing the collection of functions in this source file.

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 if the shift amount
   is less than 0 or greater than 31.


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;
}

/*
 * pow2plus4 - returns 2^x + 4, where 0 <= x <= 31
 */
int pow2plus4(int x) {
   /* exploit ability of shifts to compute powers of 2 */
   int result = (1 << x);
   result += 4;
   return result;
}

FLOATING POINT CODING RULES

For the problems that require you to implement 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 can use any arithmetic,
logical, or comparison operations on int or unsigned data.

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 operations (integer, logical,
     or comparison) that you are allowed to use for your implementation
     of the function.  The max operator count is checked by dlc.
     Note that assignment ('=') 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
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 *   where bits are numbered from 0 (least significant) to 31 (most significant)
 *   Examples allOddBits(0xFFFFFFFD) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12 = Program ops: 12
 *   Rating: 2
 */
int allOddBits(int x) {

   int mask = 0xAA;
   int y = x & mask;
   x = x >> 8;
   y = y & (x & mask);
   x = x >> 8;
   y = y & (x & mask);
   x = x >> 8;
   y = y & (x & mask);
   return !(y ^ mask);

}


//2
/*
 * isLessOrEqual - if x <= y  then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
```

```
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24 > Program ops: 15
 *   Rating: 3
 */
int isLessOrEqual(int x, int y) {

   int diff = (x >> 31) ^ (y >> 31);
   int neg_sum = ~x + y;
   int compare = neg_sum + 1;
   compare = compare >> 31;
   return !(diff & (y >> 31)) & ((diff & (x >> 31)) | !compare);

}

//3
/*
 * logicalNeg - implement the ! operator, using all of
 *              the legal operators except !
 *   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12 > Program ops: 6
 *   Rating: 4
 */
int logicalNeg(int x) {

   int fone = x | (~x + 1);
   int fbit = ~fone;
   fbit = fbit >> 31;
   return fbit & 0x01;

}

//4
//float
/*
 * floatScale2 - Return bit-level equivalent of expression 2*f for
 *   floating point argument f.
 *   Both the argument and result are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representation of
 *   single-precision floating point values.
 *   When argument is NaN, return argument
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30 > Program ops: 25
 *   Rating: 4
 */

unsigned floatScale2(unsigned uf) {

   int index = uf & 0x7F800000;
   unsigned int tail = uf & 0x007FFFFF;
   if(!index & !tail) return uf;                // 0
   if(!(~((index<<1)>>24))) return uf;          // NaN, inf
   if(!index && tail){                          // denorm
      unsigned int ntail = uf << 1;
      return (uf & 0x80000000) + ntail;
   }
   index = index + 0x00800000;
   uf = uf & 0x807FFFFF;
   return uf + index;

}

//5
//float
```

```c
/*
 * floatFloat2Int - Return bit-level equivalent of expression (int) f
 *   for floating point argument f.
 *   Argument is passed as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point value.
 *   Anything out of range (including NaN and infinity) should return
 *   0x80000000u.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30 > Program ops: 28
 *   Rating: 4
 */
int floatFloat2Int(unsigned uf) {

   unsigned int sym = uf >> 31;

   int index = uf & 0x7F800000;
   unsigned int tail = uf & 0x007FFFFF;

   if(!index & !tail) return 0;                // 0

   index = index >> 23;
   index = index - 127;
   if(index < 0) return 0;
   if(index > 30 || !(~((index<<1)>>24)))       // NaN, inf
      return 0x80000000;

   tail = tail + 0x00800000;
   index = 23 - index;
   if(index < 0) tail = tail << (-index);
   else tail = tail >> index;

   if(sym) return ~tail+1;
   else return tail;

}
/*
 * floatPower2 - Return bit-level equivalent of the expression 2.0^x
 *   (2.0 raised to the power x) for any 32-bit integer x.
 *
 *   The unsigned value that is returned should have the identical bit
 *   representation as the single-precision floating-point number 2.0^x.
 *   If the result is too small to be represented as a denorm, return
 *   0. If too large, return +INF.
 *
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while
 *   Max ops: 30
 *   Rating: 4
 */
```
