Programming Assignment #2: Numbers and Bit Manipulations

Prof. Jae W. Lee (jaewlee@snu.ac.kr)
Department of Computer Science and Engineering
Seoul National University

TA (snu-arc-uarch-ta@googlegroups.com)

Contents

Caution

Neither 'fork' nor 'push' is allowed.

Your code will be uploaded to the web when pushed.
This allows everyone to access and read your code.

Anyone who forks or pushes will be disadvantaged.

Goal of this project

- **Understand and be familiarized with bit representation**
  - You are to implement several simple functions.
    - e.g., bit operation, bitwise float calculation.
  - Limited types and number of bit operators are allowed.

Environment setup

- **You will use GCC on Linux:**
  - Use same environment as PA1
  - please type:

```
sudo apt-get install gcc gcc-multilib
git clone
https://github.com/SNU-ARC/2023_fall_comarch_PA2
```

  - Add execution permission to 'dlc'

```
chmod +x ./dlc
```

```
bits.c  btest.c  decl.c  fshow.c  Makefile  tests.c
bits.h  btest.h  dlc     ishow.c  README
```

Explanation & Helper programs

- **bits.c / bits.h**
  - Problems to solve.
  - Header file is for test cases
- **tests.c**
  - Answer for function. But not following the coding rule.
- **btest.c / btest.h**
  - You can test your code with `btest`. This gives you test cases.
- **decl.c**
  - You can check the restrictions. But, please don't touch.
- **fshow.c / ishow.c**
  - Helper program for bit representation of float, int.
- **dlc**
  - Rule Checking program. (executable binary file)

Explanation & Helper programs

- **`./dlc bits.c`**
  - Check whether you followed the rules correctly.

```
PA2$ ./dlc bits.c
dlc:bits.c:243:bitFilter: Illegal constant (0xffffffff) (only 0x0 - 0xff allowed)
```

- **`./btest`**
  - Test your functions for correctness.

```
Score   Rating  Errors  Function
 1      1       0       onebitParity
 1      1       0       checkSubstraction
 1      1       0       twoscom2SignedVal
 1      1       0       nibbleReverse
 1      1       0       bitFilter
 1      1       0       addAndDivideBy4
 1      1       0       numZerosFirst
 1      1       0       absFloat
 1      1       0       floatFloor
ERROR: Test floatAverage(0[0x0],0[0x0]) failed...
...Gives 4194304[0x400000]. Should be 0[0x0]
Total points: 9/10
```

score you got (1 is max)

function being tested

input values

your result from bits.c

ground truth from test.c

total score you've got

Explanation & Helper programs

- **`./fshow [hexval]`**
  - After 'make', you can find fshow and ishow
  - Represent hex value as floating-point value

```
$> ./fshow 0x1234
```

```
Floating point value 6.530050844e-42
Bit Representation 0x00001234, sign = 0, exponent = 0x00, fraction = 0x001234
Denormalized.  +0.0005555153 X 2^(-126)
```

- **`./ishow [hexval]`**
  - After 'make', you can find fshow and ishow
  - Represent hex value as integer value

```
$> ./ishow 0x80000000
```

```
Hex = 0x80000000,       Signed = -2147483648,   Unsigned = 2147483648
```

Problems

- Please fill in the functions inside bits.c
- 10 problems are ready to solve
- After writing your own code, type 'make' to build your project

- `./dlc bits.c` for checking coding rule (validity check)
- `./btest` for checking answer (correctness check)

List of problems

- onebitParity
- checkSubstraction
- twoscom2SignedVal
- nibbleReverse
- bitFilter
- addAndDivideBy4
- numZerosFirst
- absFloat
- castFloat2Int
- compareFloat

Solving tips

- **There are restrictions on each problem.**
  - 'if' conditions, 'while' loops are also restricted for some problems.

```
 * onebitParity - returns 1 if x contains an odd number of 0's
 *   Examples: onebitParity(5) = 0, onebitParity(7) = 1
 *   Legal ops: & ^ << >>
 *   Max ops: 20
 *   Rating: 1
 */
int onebitParity(int x) {
  return 2;
}
```

- **After you write your code, you can check error with dlc**
  - ./dlc bits.c for check your code

```
dlc:bits.c:180:onebitParity: Illegal operator (|)
dlc:bits.c:182:onebitParity: Warning: 36 operators exceeds max of 20
```

- **Please refer to 'tests.c' to check the right answer**

```
int test_onebitParity(int x) {
  int result = 0;
  int i;
  for (i = 0; i < 32; i++)
    result ^= (x >> i) & 0x1;
  return result;
}
```

Q1. onebitParity

```
 * onebitParity - returns 1 if x contains an odd number of 0's
 *   Examples: onebitParity(5) = 0, onebitParity(7) = 1
 *   Legal ops: & ^ << >>
 *   Max ops: 20
 *   Rating: 1
 */
int onebitParity(int x) {
  return 2;
}
```

- **Parity is a simple form of error-checking**
  - Many ways to make & check parity, like counting 1 bits
- **For now, bitwise XOR operation per bit to make parity**
  - ex) $1101_{(2)} \longrightarrow 1\text{^}1\text{^}0\text{^}1 = 1$

Q2. checkSubstraction

```
 * checkSubstraction - Determine if can compute x-y without overflow
 *   Example: checkSubstraction(0x80000000,0x80000000) = 1,
 *            checkSubstraction(0x80000000,0x70000000) = 0,
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 1
 */
int checkSubstraction(int x, int y) {
  return 2;
}
```

- Check x - y is valid for 32 bits operation
- If possible, return 1. If not, return 0.
- Inputs can be any values represented by 32 bits.

Q3. twoscom2SignedVal

```
 * twoscom2SignedVal - Convert from two's complement to signed-magnitude
 *   where the MSB is the sign bit
 *   You can assume that x > TMin
 *   Example: twoscom2SignedVal(-5) = 0x80000005.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 1
 */
int twoscom2SignedVal(int x) {
  return 2;
}
```

- int x is two's complement for sure.
- Don't change sign bit, but represent absolute value of 'x' for all other bits
- Minimum value of input x is 0x80000001
- You can refer to tests.c

Q4. nibbleReverse

```
 * nibbleReverse - Reverse nibbles(4bits) in a 32-bit word
 *   Examples: nibbleReverse(0x80000002) = 0x20000008
 *             nibbleReverse(0x89ABCDEF) = 0xFEDCBA98
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 25
 *   Rating: 1
 */
int nibbleReverse(int x) {
    return 2;
}
```

- **4-bit reverse swap for 32-bit word.**
  - ex) 0x12345678 → 0x87654321

Q5. bitFilter

```
 * bitFilter - Generate a mask consisting of all 1's and filter input with it.
 * Examples: bitFilter(0xFF00, 11, 4) = 0x0F00,
 * bitFilter(0x2A00, 13, 9) = 0x2A00,
 * bitFilter(0x1300, 4, 2) = 0
 * Assume 0 <= lowbit <= 31, and 0 <= highbit <= 31
 * If lowbit > highbit, then mask should be all 0's
 *   Legal ops: & | << >>
 *   Max ops: 20
 *   Rating: 1
 */
int bitFilter(int input, int highbit, int lowbit) {
    return 2;
}
```

- **Make mask from low bit to high bit (inclusive)**
  - ex) bitFilter(0xFF00, 11, 4) → mask will be 0x0FF0
  - if lowbit > highbit, then mask will be 0
- **Mask input and return**
  - ex) bitFilter(0xFF00, 11, 4) == 0xFF00 & 0x0FF0 == 0x0F00

Q6. addAndDivideBy4

```
 * addAndDivideBy4 - adds two numbers and divide by 4 (round toward 0). But when overflow occurs
 *          while adding two numbers, returns the first operand.
 *   Examples: addAndDivideBy4(1073741824,1073741824) = 1073741824
 *             addAndDivideBy4(-2147483648,-1) = -2147483648
 *             addAndDivideBy4(32,9) = 10
 *             addAndDivideBy4(-22,9) = -3
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 1
 */
int addAndDivideBy4(int x, int y) {
  return 2;
}
```

- If (x+y) overflows, then return x
- else, return (x+y)/4

Q7. numZerosFirst

```
 * numZerosFirst - returns count of number of continuous 0's from first bits
 *   Example: numZerosFirst(0) = 32
 *   Example: numZerosFirst(0x80000000) = 0
 *   Example: numZerosFirst(0x40000000) = 1
 *   Example: numZerosFirst(0x00008000) = 16
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 50
 *   Rating: 1
 */
int numZerosFirst(int x) {
  return 2;
}
```

- **Count continuous 0s from first bit and return it.**
  - 0x80000000 ⟶ 0
  - 0x00000001 ⟶ 31

Q8. absFloat

```
 * absFloat - Return bit-level equivalent of absolute value of 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 representations of
 *   single-precision floating point values.
 *   When argument is NaN, return argument..
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 10
 *   Rating: 1
 */
unsigned absFloat(unsigned uf) {
  return 2;
}
```

- **Every float problems have no restriction on ops.**
  - You can use every operations including if and while
- **Return absolute value of float input.**
- **But there is infinity or NaN values in floating point representation; please check it and return input itself.**

Q9. castFloat2Int

```
 * castFloat2Int - 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
 *   Rating: 1
 */
int castFloat2Int(unsigned uf) {
  return 2;
}
```

- Cast floating point value to integer value
- return integer value
- If value goes out of range, then return 0x80000000

Q10. compareFloat

```
 * compareFloat - Compute f < g for floating point arguments f and g.
 *   Both the arguments are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representations of
 *   single-precision floating point values.
 *   If either argument is NaN, return 0.
 *   +0 and -0 are considered equal.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 1
 */
int compareFloat(unsigned uf, unsigned ug) {
    return 2;
}
```

- If f < g then return 1, else return 0
- If either input is NaN, then return 0

Submission

- **Write-up**
  - Briefly describe your implementation.
  - Filename: [student_id].txt (example: 2023-12345.txt)
  - Please use 'UTF-8' encoding if possible
  - **Please** submit it in **txt** format. Other formats are not accepted.
- **Compress your source code and write-up into a single zip file.**
  - Compress bits.c and your report.
  - Filename should be [student_id].zip (example: 2023-12345.zip).
  - **Please** submit it in **ZIP** format. Other formats are not accepted.
- **Submission deadline: by 23:59 on October 18, 2023**