Report For Computer Architecture Project 1

Name: Zhengtian Xu StudentID: 5140309178

October 25, 2016

1 Introduction

This is the report for the Project of Computer Architecture. In this project, I finished some bit-level representation of integers and floating point numbers. This project contains 15 programming "puzzles" and in this report I will give solutions and analysis for each puzzle as well as some problems I have overcame during this experiment.

In this experiment, we assume that our machine:

- Uses 2's complement, 32-bit representations of integers.
- Performs right shifts arithmetically.
- Has unpredictable behavior when shifting an integer by more than the word size.

All the solutions are in *bit.c*.

2 Project Environment

2.1 Environment

This project was run on Ubuntu 16.04 LTS.

2.2 Evaluation

- **btest:** This program checks the functional correctness of the functions in *bits.c*. To build and use it, type the following two commands:

```
unix> make
unix> ./btest
```

  It is helpful to work through the functions one at a time and test each one. To achieve this target, use the *-f* flag to instruct btest to test only a single function:

```
unix> ./btest -f bitAnd
```

Use the option flags -1, -2, and -3 to feed it specific function arguments :

```
unix> ./btest -f bitAnd -1 7 -2 0xf
```

- **dlc:** This is a modified version of an ANSI C compiler from the MIT CILK group that can be used to check for compliance with the coding rules for each puzzle. The typical usage is:

  ```
  unix> ./dlc bits.c
  ```

  The program runs silently unless it detects a problem, such as an illegal operator, too many operators, or non-straightline code in the integer puzzles. Running with the -e switch:

  ```
  unix> ./dlc -e bits.c
  ```

  causes *dlc* to print counts of the number of operators used by each function. Type *./dlc -help* for a list of command line options.

3 Prerequisite Knowledge

3.1 Bit Manipulations

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization. For most other tasks, modern programming languages allow the programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.[1]

3.2 Integers

An integer is a number with no fractional part; it can be positive, negative or zero. In this project, integers are represented in 32 bits.

Unsigned integer An unsigned integer is either positive or zero and an unsigned integer containing n bits can have a value between 0 and $2^n - 1$.

Signed integer Signed integers are stored in a computer using two's complement. In two's complement representation, positive numbers are simply represented as themselves, and negative numbers are represented by the two's complement of their absolute value[2]. An n-bit two's complement numeral system can represent every integer in the range $-(2^{n-1})$ to $+(2^{n-1} - 1)$.

3.3 Floating point

We use 32-bit floating point which is called IEEE 754 Floating Point Standard:

- 1 bit for $sign(s)$ of floating point number
- 8 bits for $exponent(E)$
- 23 bits for $fraction(F)$ (get 1 extra bit of precision if leading 1 is implicit)

The value of the floating point is $(-1)^S \times (1+F) \times 2^{E-127}$.

Representation for Denorms Normalized number have been defined , we briefly call them norms, we also have defined 0, infinity and NaN. We have used all combination except for this one, we can use this combination to represent denormalized numbers.

Table 1: Representation for Denorms

Exponent	Significand	Object
0	0	+/-0
0	nonzero	Denorms
1-254	anything implicit leading 1	Norms
255	0	+/- infinity
255	nonzero	NaN

4 Problems Statement

4.1 Bit Manipulations

In this problem, some puzzles about bit manipulations will be realized. Coding rules and styles must satisfy the following table.

Table 2: Bit-Level Manipulation Functions

Name	Description	Rating	Max Ops
bitAnd(x,y)	x & y using only $\|$ and $\sim$	1	8
getByte(x,n)	Get byte n from x	2	6
logicalShift(x,n)	Shift right logical	3	20
bitCount(x)	Count the number of 1's in x	4	40
bang(x)	Compute !n without using ! operator	4	12

4.2 Two's Complement Arithmetic

In this problem, some puzzles about two's complement arithmetic will be realized. Coding rules and styles must satisfy the following table.

Table 3: Two's Complement Arithmetic

Name	Description	Rating	Max Ops
tmin()	Most negative two's complement integer	1	4
fitsBits(x,n)	Does x fit in n bits?	2	15
divpwr2(x,n)	Compute $x/2n$	2	15
negate(x)	$-x$ without negation	2	5
isPositive(x)	$x > 0$?	3	8
isLessOrEqual(x,y)	$x <= y$?	3	24
ilog2(x)	Compute $\lfloor \log_2 x \rfloor$	4	90

4.3 Floating-Point Functions

In this problem, some puzzles about floating point functions will be realized. Coding rules and styles must satisfy the following table.

Table 4: Floating-Point Functions

Name	Description	Rating	Max Ops
float_neg(uf)	Compute -f	2	10
float_i2f(x)	Compute (float) x	4	30
float_twice(uf)	Computer 2*f	4	30

5 Experiment

5.1 bitAnd

5.1.1 Requirement

bitAnd - $x\&y$ using only $\sim$ and |

Example: bitAnd(6, 5) = 4

Legal ops: $\sim$ |

Max ops: 8

Rating: 1

5.1.2 Analysis

According to the *De Morgan's laws*,:

$$AB = \overline{\overline{A} + \overline{B}}$$

the operation And equals to the combination of the operation Or and Not. So given x and y, apply the *De Morgan' laws*, we can get the answer $\sim (\sim x| \sim y)$.

5.1.3 Solution Code

```
int bitAnd(int x, int y) {
        return ~(~x | ~y);
}
```

5.2 getByte

5.2.1 Requirement

getByte - Extract byte n from word x
Bytes numbered from 0 (LSB) to 3 (MSB)
Examples: getByte(0x12345678,1) = 0x56
Legal ops:! $\sim$ & ˆ | $+ << >>$
Max ops: 6
Rating: 2

5.2.2 Analysis

The number x has 4 bytes, so I shift n to the left by three bits to find the starting bit of the byte. And then shift x to the right by the number above and extract the byte by using *And* operation for the number and 0xff.

5.2.3 Solution Code

```
int getByte(int x, int n) {
        int shiftNumber;
        int newx;

        shiftNumber = n << 3;
        newx = x >> shiftNumber;
        return newx & 0xff;
}
```

5.3 logicalShift

5.3.1 Requirement

logicalShift - shift x to the right by n, using a logical shift

Can assume that $0 <= n <= 31$

Examples: logicalShift(0x87654321,4) = 0x08765432

Legal ops:! $\sim$ & ^ | $+ << >>$

Max ops: 20

Rating: 3

5.3.2 Analysis

This problem is divided into two different situations, one of which is x is negative and the other is non-negative. When the sign number is 0, if we shift x to the right, it will be no problem. However, if the sign bit is 1, if we shift x to the right, it will fill "empty" MSB bits with 1s.

Thus, I drop the leftmost n bits whether the sign bit represents positive or negative.

5.3.3 Solution Code

```
int logicalShift(int x, int n) {
        /* Make the left n number is 1*/
        int number;

        number = (1 << 31) >> n;
        /* I can't use minus operation, so first right shift n bits and then
           left shift one bit */
        number = number << 1;
        /* Use Not function for number because we want to maintain the last
            (32-n) bits */
        number = ~number;
        /* Shift x to the right by n bits without the sign bit */
        return (x >> n) & number;
}
```

5.4 bitCount

5.4.1 Requirement

bitCount - returns count of number of 1's in word

Examples: bitCount(5) = 2, bitCount(7) = 3

Legal ops: ! $\sim$ & ^ | $+ << >>$

Max ops: 40

Rating: 4

5.4.2 Analysis

If we detect each bit of x, the operation will exceed the limited quato. So we split it in 4 froups and calculate the number of 1. Use the method ehich called Divide and Conquer. Then I shrink the length using the same method.

In the first step we add together bits 0 and 1 and put the result in the two bit segment 0-1, add bits 2 and 3 and put the result in the two-bit segment 2-3 etc...

In the second step we add the two-bits 0-1 and 2-3 together and put the result in four-bit 0-3, add together two-bits 4-5 and 6-7 and put the result in four-bit 4-7 etc...

5.4.3 Solution Code

```
int bitCount(int x) {
    /* Construct the constants. */
    int result;
    int const1;
    int const2;
    int const3;
    int const4;
    int const5;

    /* 0x55555555 */
    const1 = (0x55) | (0x55 << 8);
    const1 = const1 | (const1 << 16);

    /* 0x33333333 */
    const2 = (0x33) | (0x33 << 8);
    const2 = const2 | (const2 << 16);

    /* 0x0f0f0f0f */
    const3 = (0x0f) | (0x0f << 8);
    const3 = const3 | (const3 << 16);

    /* 0x00ff00ff */
    const4 = (0xff) | (0xff << 16);

    /* 0x0000ffff */
    const5 = (0xff) | (0xff << 8);


    /* add every two bits */
    result=(x & const1)+((x >> 1) & const1);
    /* add every four bits */
    result=(result & const2) + ((result >> 2) & const2);

    /* add every eight bits */
    result=(result+(result>>4))&const3;

    /* add every sixteen bits */
    result=(result + (result >> 8)) & const4;

    /* add every thirty two bits */
    result=(result + (result >> 16)) & const5;
    return result;
}
```

5.5 bang

5.5.1 Requirement

bang - Compute $!x$ without using !

Examples: bang(3) = 0, bang(0) = 1

Legal ops: $\sim$ & ^ | $+ << >>$

Max ops: 12

Rating: 4

5.5.2 Analysis

The function ! represents that if the number is 0, then return 1, or return 0.

After observing the rule of two's complement and negative number, we can find that if the negative number of x has a different sign bit of x, then we can conclude that x is not equal to zero.We can compare the sign bit of x and its negative number because only zero has the same sign bit of the negative number of zero.

5.5.3 Solution Code

```
int bang(int x) {
        int negative;
        int signDifference;
        int result;
        /* First I find the negative number of x using two complement */
        negative = ~x + 1;
        /* If the sign bit is different, then return 0. Or return 1 */
        signDifference = (negative | x) >> 31;
        result = ~signDifference;
        /* To make the return number is one bit. */
        return (result & 0x01);
}
```

5.6 tmin

5.6.1 Requirement

tmin - return minimum two's complement integer

Legal ops: ! ~ & ^ | + << >>

Max ops: 4

Rating: 1

5.6.2 Analysis

The minimum two's complement integer is 0x80000000.

5.6.3 Solution Code

```
int tmin(void) {
        /* The minimum two's complement integer is 10000000... */
        return (0x01 << 31);
}
```

5.7 fitsBits

5.7.1 Requirement

fitsBits - return 1 if x can be represented as an

n-bit, two's complement integer.

$1 <= n <= 32$

Examples: fitsBits(5,3) = 0, fitsBits(-4,3) = 1

Legal ops: ! ~ & ^ | + << >>

Max ops: 15

Rating: 2

5.7.2 Analysis

This situation can be divided into two situations: one is x is negative and the other is non-negative.

- x **is negative:** The sign bit is 1, so if x can be representyed as an n-bit, two's complement integer, when we shift x by $n - 1$ bits, the number left is all 1s.

- x **is non-negative:** The sign bit is 0, so if x can be represented as an n-bit , two's complement integer, when we shift x by $n-1$ bits, the number left is all 0s.

5.7.3 Solution Code

```
int fitsBits(int x, int n) {
        int shift;
        int newx;

        /* Shift (n-1) bits which is n + 1 - 2 */
        shift = n + 0x01 + (~0x01);
        /* Shift x to the left by 'shift' bits and shift back again to see if
            the number left is all 0s or 1s */
        newx = (x >> shift);
        return (!newx) ^ (!(~newx));
}
```

5.8 divpwr2

5.8.1 Requirement

divpwr2 - Compute$x/(2^n)$, for $0 <= n <= 30$

Round toward zero

Examples: divpwr2(15,1) = 7, divpwr2(-33,4) = -2

Legal ops: $! \sim \& \,\hat{}\, | + << >>$

Max ops: 15

Rating: 2

5.8.2 Analysis

This situation can be divided into two situations:

- x **is non-negative**

  When x is a non-negative number, we can conclude that the answer of $\frac{x}{2^n}$ is to shift x to the right by n bits.

- x **is negative**

  When x is a negative number, we claim that the answer of $\frac{x}{2^n}$ is $\sim((\sim x+1) >> n)+1$

  Proof:

1. When the last n bits of x are not all 0s, then bits n-31 (bit 0 is the LSB) of both $\sim x$ and $\sim x+1$ are the same. Then we can prove that

$$\begin{aligned} \sim((\sim x+1) >> n)+1 &= \sim((\sim x) >> n)+1 \\ &= (x >> n)+1 \end{aligned}$$

2. When the last n bits of x are all 0s, then bits n-31 (bit 0 is the LSB) of both $\sim x+1$ and $\sim x+(1 << n)$ are the same. Then we can prove that

$$\begin{aligned} \sim((\sim x+1) >> n)+1 &= \sim((\sim x >> n)+1) \\ &= x >> n \end{aligned}$$

5.8.3 Solution Code

```
int divpwr2(int x, int n) {

        int sign;
        int m;
        int low;

        /* Check the sign bit to see if x is  negative */
        sign =!!(x >> 31);
        m = (1 << n) + ~0;
        low = m & x;
        return (x >> n) + ((!!low) & sign);
}
```

5.9 negate2

5.9.1 Requirement

negate - return -x

Example: negate(1) = -1.

Legal ops: ! $\sim$ & ^ | + << >>

Max ops: 5

Rating: 2

5.9.2 Analysis

This situation can be divided into two situations:

- **x is non-negative**

  Obviously, the negative of the number x is its two's complement number, which is $\sim x+1$.

- x **is negative**

  We claim that the answer is also $\sim x + 1$.

  Proof:

$$\begin{aligned} x + \sim x + 1 &= 0x11111111 + 1 \\ &= 0 \end{aligned}$$

  We know that in two's complement number, 0 has one certain form. So $\sim x + 1$ is the reverse number of x

5.9.3 Solution Code

```
int negate(int x) {
        /* return the two's complement number of x */
        return ~x + 1;
}
```

5.10 isPositive

5.10.1 Requirement

isPositive - return 1 if $x > 0$, return 0 otherwise
Example: isPositive(-1) = 0.
Legal ops: ! $\sim$ & ^ | + << >>
Max ops: 8
Rating: 3

5.10.2 Analysis

The method to check whether the number is positive is to check its sign bit and the number is not zero. We can exetract the sign bit and then check whether the number is zero.

5.10.3 Solution Code

```
int isPositive(int x) {
        int sign;
        int zero;
        /* Check whether the sign bit is 0 but drop the number 0 */
        sign = !!(x >> 31);
        /* If x = 0, then zero = 1 */
        zero = !x;
        /* Correct answer is sign = 0 and zero = 0 */
        return !(sign | zero);
}
```

5.11 isLessOrEqual

5.11.1 Requirement

isLessOrEqual - if$x <= y$ then return 1, else return 0

Example: isLessOrEqual(4,5) = 1.

Legal ops: ! $\sim$ & ^ | + << >>

Max ops: 24

Rating: 3

5.11.2 Analysis

We can check that whether the sign bits between two numbers are the same.

- **The sign bits are the same**
  If the sign bits are the same, then we can check whether $y - x >= 0$.If the former equation is satisfied, the answer is true. Otherwise, the answer is no.

- **The sign bits are different**
  If the sign bits are different, then we can check whether the sign bit of x represents that it is positive. If it is true, the answer is no. Otherwise, the answer is yes.

5.11.3 Solution Code

```
int isLessOrEqual(int x, int y) {
        int signx;
        int signy;
        int same;
        int sameSign;
        int notSameSign;

        signx = !!(x >> 31);
        signy = !!(y >> 31);
        /* Check whether x and y are the same sign*/
        same = !(signx ^ signy);
        /* If same, then check (x-y)'s sign. Otherwise if sign of x represents
            positive, return 0, or return 1 */
        sameSign = ((x + ~y) >> 31);
        notSameSign = signx;
        return (same & sameSign) | ((!same) & notSameSign);
}
```

5.12 ilog2

5.12.1 Requirement

ilog2 - return floor(log base 2 of x), where $x > 0$

Example: ilog2(16) = 4

Legal ops: ! ~ & ^ | + << >>

Max ops: 90

Rating: 4

5.12.2 Analysis

The key of this problem is to find the leftmost number 1 in the bit representation of x.

Since the biggest answer is 31, so we can represent it by using a number of 5 bits. In this problem, it is easy for us to determine the exact position of the number 1 by using the recurrence algorithm, and divided the original problem into more small problems. This is the thought of recurrence and we can using the following steps:

- Check whether the left 16 bits of the number x has the number one. If it has the number one, we drop the right 16 bits and using the same steps to check the left bits by decreasing the checking bits. At the same time, we use a value to record the shift bits.
- If it doesn't has the number one, we can just drop the leftmost 16 bits and check the left bits by decreasing the checking bits. However, we don't record the shift bits.
- Finally, the answer is the sum of the recorded values because we shift the leftmost one to the rightmost.

5.12.3 Solution Code

```c
int ilog2(int x) {
        int flag;
        int shift1;
        int shift2;
        int shift3;
        int shift4;
        int shift5;

        /* Whether x's left 16 bits has 1 */
        flag = !!(x >> 16);
        /* If has, then shift x to the right by 16 bits */
        shift1 = flag << 4;
        x = x >> shift1;

        /* Whether x's left 8 bits has 1 */
        flag = !!(x >> 8);
        /* If has, then shift x to the right by 8 bits */
        shift2 = flag << 3;
        x = x >> shift2;

        /* Whether x's left 4 bits has 1 */
        flag = !!(x >> 4);
        /* If has, then shift x to the right by 4 bits */
        shift3 = flag << 2;
        x = x >> shift3;

        /* Whether x's left 2 bits has 1 */
        flag = !!(x >> 2);
        /* If has, then shift x to the right by 2 bits */
        shift4 = flag << 1;
        x = x >> shift4;

        flag = !!(x >> 1);
        shift5 = flag;
        /* Return the sum which means the index of first "1" from left */
        return shift1 + shift2 + shift3 + shift4 + shift5;
}
```

5.13 float_neg

5.13.1 Requirement

float_neg - Return bit-level equivalent of expression -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: 2

5.13.2 Analysis

This problem requires us to change the floatint point to the its reverse floating point.

We know that the leftmost bit of the floating point is the sign bit, so we change the sign bit and then check whether the floating point is NaN. According to the *Table 1*, we know that if the exponent is 255, the floating point is NaN, so we should exclude the number.

5.13.3 Solution Code

```
unsigned float_neg(unsigned uf) {
        unsigned result;
        unsigned tmp;

        /* Reverse the sign */
        result = uf ^ 0x80000000;
        /* Extract the exponent and mantissa */
        tmp = uf & 0x7fffffff;
        /* Check NAN */
        if (tmp > 0x7f800000) return uf;
        else return result;
}
```

5.14 float_i2f

5.14.1 Requirement

float_i2f - Return bit-level equivalent of expression (float) x
Result is returned as unsigned int, but
it is to be interpreted as the bit-level representation of a
single-precision floating point values.
Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
Max ops: 30
Rating: 4

5.14.2 Analysis

This problem requires us to transform a integer to a floating point.
We know that a floating is consist of three parts, which are:

- The sign bit is the same as the sign bit of the integer.
- The exponent part is the position of the leftmost one in the integer.
- The significand part is 23 bits, so it is 23 bits from the begining of the leftmost 1 to the right.

In this problem, we should do the rounding.

- **Guard bit:** used to provide one F bit when shifting left to normalize a result (e.g., when normalizing F after division or subtraction)
- **Round bit:** used to improve rounding accuracy.
- **Sticky bit:** used to support Round to nearest even; is set to a 1 whenever a 1 bit shifts (right) through it (e.g., when aligning F uring addition/subtraction)

5.14.3 Solution Code

```
unsigned float_i2f(int x) {
        unsigned sign;
        unsigned exponent;
        unsigned significand;
        unsigned G, R, S;  /* Rounding */

        if (!x) return 0;
        if (x < 0) {sign = 0x80000000; x = ~x + 1;}
        else sign = 0;
        exponent = 31;

        while (!(x & 0x80000000) && exponent){
                x = x << 1;
                exponent = exponent + (~1+1);
        }
        exponent = (exponent + 127) << 23;
        significand = (x >> 8) + 0x00800000;
        /* do the rounding */
        G = (x & 0x01ff) >> 8;
        R = (x & 0x0ff) >> 7;
        S = !!(x & 0x7f);

        return sign + exponent + significand + ((R&&S) || (G&&R));
}
```

5.15 float_twice

5.15.1 Requirement

float_twice - 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
Rating: 4

5.15.2 Analysis

Considering three parts of floating point, which are sign, exponent and Significand. This problem can be divided into two situations:

- **The exponent part equals to zero**
  We should shift the uf to the left by one bit and add the sign bit.

- **The exponent part isn't equal to zero**
  We should check whether the floating point is NaN, if not, we can just add one to the exponent part.

5.15.3 Solution Code

```
unsigned float_twice(unsigned uf) {
        unsigned exp;
        unsigned sign;
        /* Extract the exponent */
        exp = uf & 0x7f800000;
        /* Extract the sign bit */
        sign = uf & 0x80000000;
        /* Check the exponent */
        if (exp){
                if (exp != 0x7f800000) uf = uf + 0x00800000;
        }
        else
                uf = (uf << 1) | sign;
        return uf;
}
```

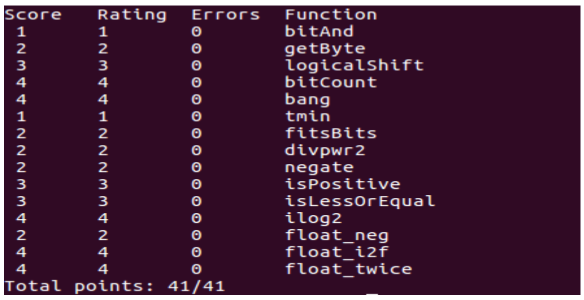
6 Result

6.1 Correctness

Use the autograding tools in the handout directory — *btest* to check the correctness of my project. The following picture can show the correctness.

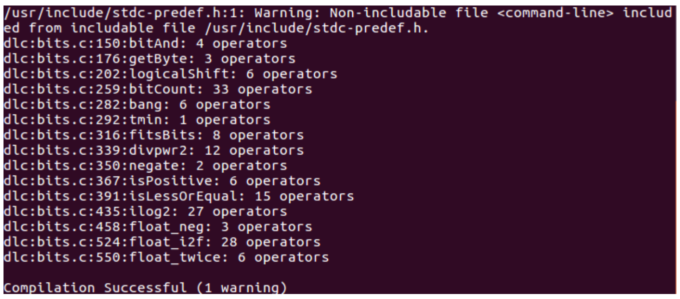
6.2 Compliance

Use the autograding tools in the handout directory — *dlc* to check the compliance with the coding rules for each puzzles. The following picture can show the compliance.



```
Score   Rating  Errors  Function
 1       1       0       bitAnd
 2       2       0       getByte
 3       3       0       logicalShift
 4       4       0       bitCount
 4       4       0       bang
 1       1       0       tmin
 2       2       0       fitsBits
 2       2       0       divpwr2
 2       2       0       negate
 3       3       0       isPositive
 3       3       0       isLessOrEqual
 4       4       0       ilog2
 2       2       0       float_neg
 4       4       0       float_i2f
 4       4       0       float_twice
Total points: 41/41
```

Figure 1: The correctness of the project



```
/usr/include/stdc-predef.h:1: Warning: Non-includable file <command-line> includ
ed from includable file /usr/include/stdc-predef.h.
dlc:bits.c:150:bitAnd: 4 operators
dlc:bits.c:176:getByte: 3 operators
dlc:bits.c:202:logicalShift: 6 operators
dlc:bits.c:259:bitCount: 33 operators
dlc:bits.c:282:bang: 6 operators
dlc:bits.c:292:tmin: 1 operators
dlc:bits.c:316:fitsBits: 8 operators
dlc:bits.c:339:divpwr2: 12 operators
dlc:bits.c:350:negate: 2 operators
dlc:bits.c:367:isPositive: 6 operators
dlc:bits.c:391:isLessOrEqual: 15 operators
dlc:bits.c:435:ilog2: 27 operators
dlc:bits.c:458:float_neg: 3 operators
dlc:bits.c:524:float_i2f: 28 operators
dlc:bits.c:550:float_twice: 6 operators

Compilation Successful (1 warning)
```

Figure 2: The compliance of the project

7 Acknowledgement

My deepest gratitude goes first and foremost to Dr. Yanyan Shen, my teacher for Computer Architecture, for her teaching in this term.

Then I wish to express my sincere thanks to teaching assistant Huang Kaixin and Zhou Xian for assigning such a interesting project for us to exercise our skills in bit operation and coding.

In this project, I have leant some knowledge beyond textbooks and have a good command of the ability to code with bit operation. What'more, many problems about environment and coding will occur so I searched most of them on Google and read English material and some English website such as stackoverflow.

Actually, the *dlc* program benefits me a lot because it enforces a stricter form of C declarations than is the case for C++ or that is enforced by gcc. Before this I didn't even know that I have so much compliance error in my codes. I think I will remain the good style coding in the future.

Thanks are also due to a number of my friends who generously give me the benefit of their tremendous suggestions.

References

[1] Wikipedia contributors. *Bit Manipulations*. Wikipedia, The Free Encyclopedia.27 June 2016. 20 Oct 2016.

[2] Wikipedia contributors. *Two's complement*. Wikipedia, The Free Encyclopedia.13 October 2016. 20 Oct 2016.