暨南大學
JINAN UNIVERSITY

Undergraduate Lab Report

Course Title: Experiment of Computer Organization

Course No: 60080014

Student Name: _____

Student No: _____

School: International School

Department: _____

Major: Computer Science & Technology

Instructor: SUN Heng

Academic Year:202~202,Semester : 1st [√] 2nd[ ]

Academic Affairs Office of Jinan University

Date (dd/mm/yyyy)_____

Computer Organization Lab List

Student Name:_____ Student No:_____

ID	Lab Name	Type
1	Number Storage Lab	Individual
2	Manipulating Bits	Individual
3	Simulating Y86-64 Program	Individual
4	Performance Lab	Team
5	A Simple Real-life Control System	Team
6	System I/O	Individual

Undergraduate Lab Report of Jinan University

Course Title Experiment of Computer Organization Evaluation

Lab Name Manipulating Bits Instructor SUN Heng

Lab Address

Student Name Student No

College International School

Department Major CST

Date / / Afternoon

1. Introduction

The purpose of this lab is to become more familiar with bit-level representations and arithmetic of integers in chapter 2. You'll do this by solving a series of programming puzzles. This is an individual project.

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:

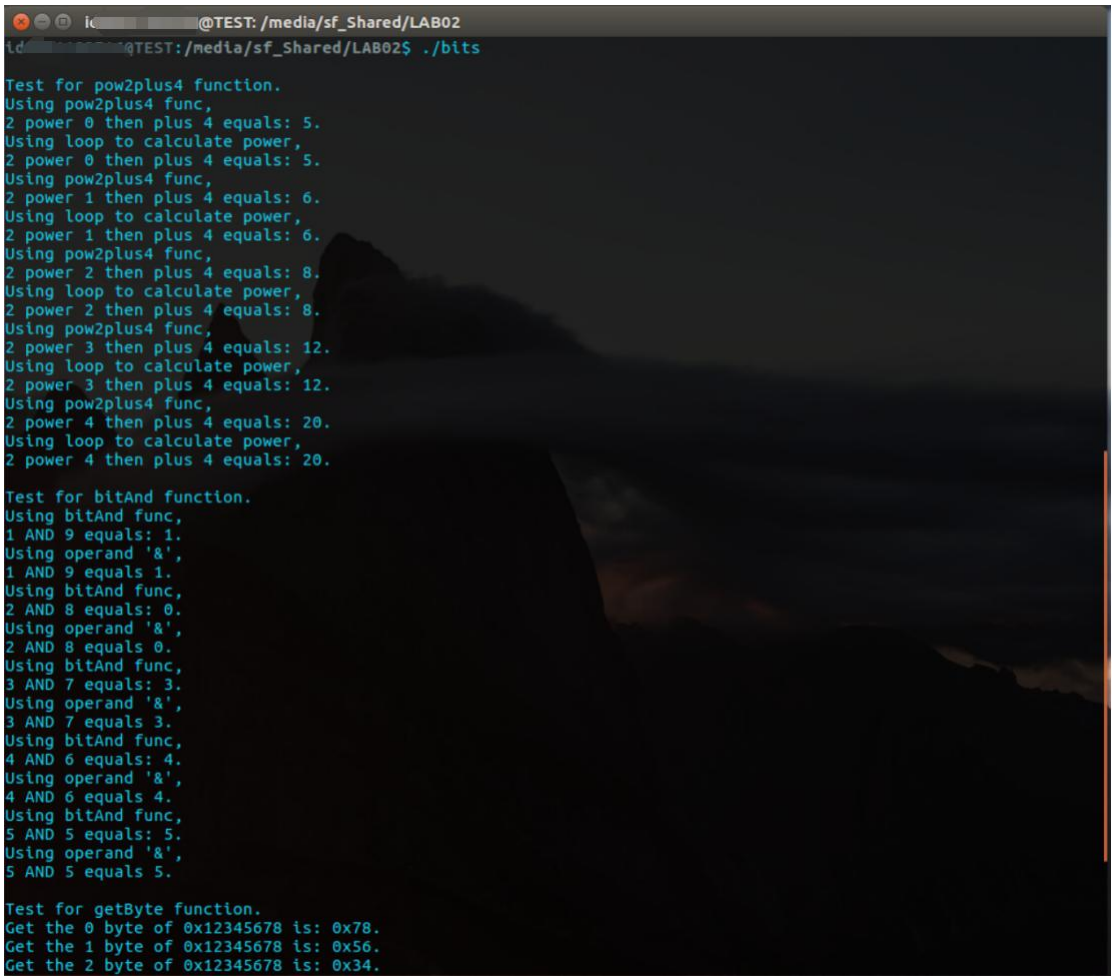
! ~ & ^ | + << >>

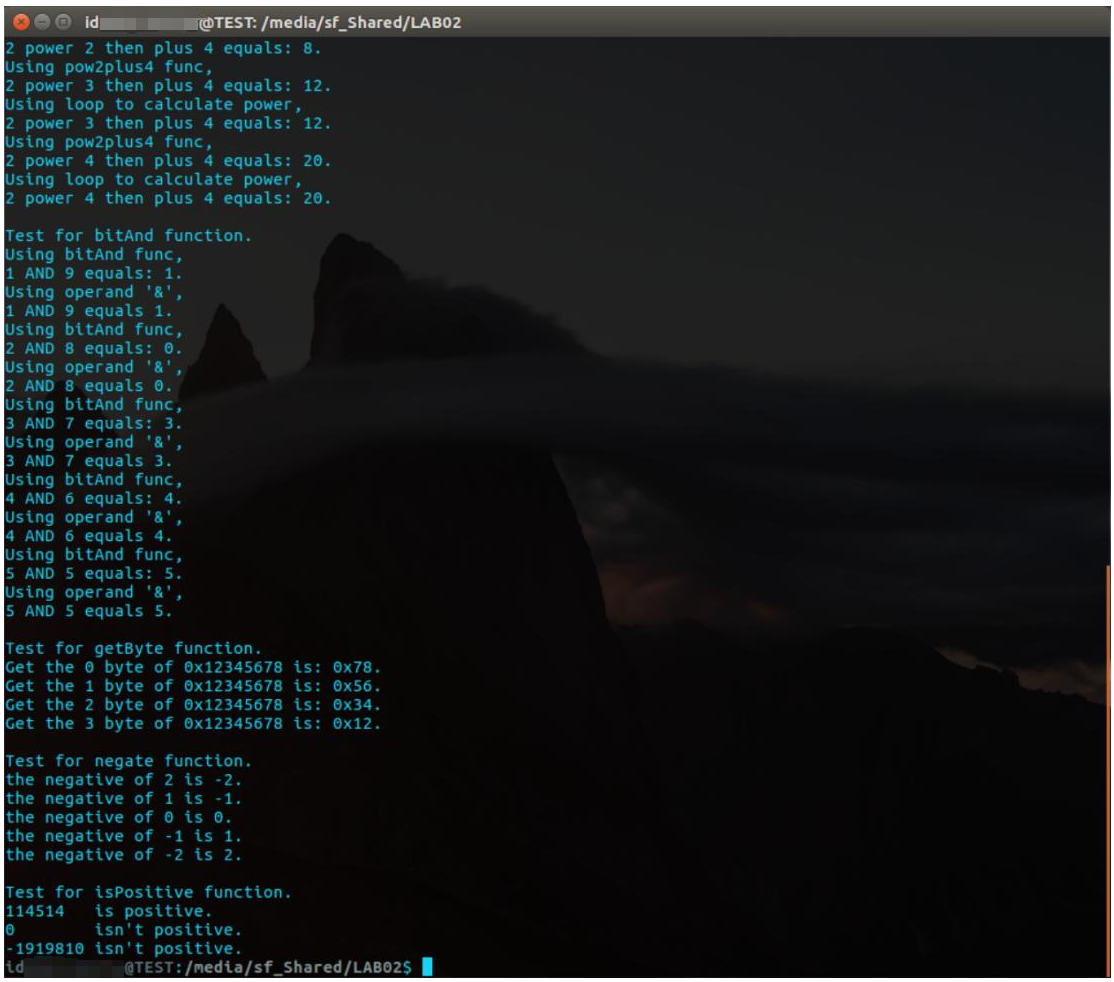
3. Lab Device or Environment

Ubuntu 16.04 (64-bit) with AMD Ryzen 9 5900HS CPU @ 3.30GHz and 4GB memory on virtual machine (Oracle VM VirtualBox)

4. Results and Analysis

Results:





Analysis:

The idea of functions implementation

(1) pow2plus4:

The pow2plus4 function requires returning a number that equals 2 ^ x + 4. From the example above that we get 2 power x plus 4 by shifting 2 left by x bits and add 4.

(2) bitAnd:

The bitAnd function requires achieving '&' by using '~' and '|' operands. Using De Morgan' s law, we can get ¬(x ∧ y) = (¬x) ∨ (¬y), so by '~' and '|', achieve operand '&' only need to take inverse of

(~x) | (~y).

(3) getByte:

The getByte function requires the implementation to get some bit (0-3 bits, 4 bits total) of an int number. Since numbers are stored in x86-64 storage by little endian way, getting the nth byte is different from the common reading way, such as the 0th byte of 0x12345678 is 0x78.

So the idea is to shift this number to the right by n bytes. In order to shift this number to the right by n bytes (namely, 8 * n bits), you can change x >> (n << 3), the value of n << 3 is equal to 8 * n, by performing the '&' operation with 0xFF (0000 · · · 0000 1111 1111), all the bits except the very right 8 bits will be set as 0, the resulting digit is the nth byte of the digit.

(4) negate:

The negate function returns the opposite number of parameter x, according to the way of number storage (Two's complement), we know the method to convert a number to its corresponding opposite number: Inverting every bit of the original number, then add 1 to it, we can get the -x of x.

(5) isPositive:

The isPositive function determines whether the parameter is positive. If so, it returns 1. If not, it returns 0. Implementing this function requires using only '!', '|' and '>>' operator.

First, according to the way negative numbers are stored in the computer, you can determine whether they are negative by shifting them 31 (0x1F) bits to the right, leaving only the original highest bit of the symbol bit. If the number is negative, the result is 1, if not, the result is 0.

Second, according to the C language '!' operator, you can see that this operation on a non-zero number will get a 0, and the operation on a zero will get an 1.

And finally, for the positive, negative and 0 cases, we can list a table to see these results and analyze if we can extract the special case for judging positive number.

isNegative: (x >> 31) or (x >> 0x1F)

isZero: !x

number	isNegative	isZero	isNegative | isZero	! (isNegative | isZero)
positive	0	0	0	1
negative	1	0	1	0
zero	0	1	1	0

5. Appendix (Program Code)

I fixed the program code and appended the main function to test the fixed functions above.

```
1. #include <stdio.h>
2.
3. /*
4.
```

```
5.   * STEP 1: Read the following instructions carefully.
6.
7.   */
8.
9.
10.
11. /* CODING RULES:
12.
13.  *
14.
15.  * Replace the "return" statement in each function with one
16.
17.  * or more lines of C code that implements the function. Your code
18.
19.  * must conform to the following style:
20.
21.  *
22.
23.  * int Funct(arg1, arg2, ...) {
24.
25.  *     brief description of how your implementation works
26.
27.  *     int var1 = Expr1;
28.
29.  *...  *int varM = ExprM;
30.
31.  *     *varJ = ExprJ;
32.
33.  *...  *varN = ExprN;
34.
35.  *     return ExprR;
36.
37.  *
38.
39.  * }
40.
41.  *
42.
43.  * Each "Expr" is an expression using ONLY the following:
44.
45.  * 1. Integer constants 0 through 255 (0xFF), inclusive. You are
46.
47.  *     not allowed to use big constants such as 0xffffffff.
48.
```

```
49.  * 2. Function arguments and local variables (no global variables).
50.
51.  * 3. Unary integer operations ! ~
52.
53.  * 4. Binary integer operations & ^ | + << >>
54.
55.  *
56.
57.  * Some of the problems restrict the set of allowed operators even further.
58.
59.  * Each "Expr" may consist of multiple operators. You are not restricted to
60.
61.  * one operator per line.
62.
63.  *
64.
65.  * You are expressly forbidden to:
66.
67.  * 1. Use any control constructs such as if, do, while, for, switch, etc.
68.
69.  * 2. Define or use any macros.
70.
71.  * 3. Define any additional functions in this file.
72.
73.  * 4. Call any functions.
74.
75.  * 5. Use any other operations, such as &&, ||, -, or ?:
76.
77.  * 6. Use any data type other than int.  This implies that you
78.
79.  *    cannot use arrays, structs, or unions.
80.
81.  *
82.
83.  *
84.
85.  * You may assume that your machine:
86.
87.  * 1. Performs right shifts arithmetically.
88.
89.  * 2. Has unpredictable behavior when shifting an integer by more
90.
91.  *    than the word size.
92.
```

```
 * 3. Uses 32-bit representations of integers.

 */


/*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; }


/*
```

```
137.  * bitAnd - x&y using only ~ and |
138.
139.  *   Example: bitAnd(6, 5) = 4
140.
141.  *   Legal ops: ~ |
142.
143.  */
144.
145. int bitAnd(int x, int y) { return ~((~x) | (~y)); }
146.
147.
148.
149. /*
150.
151.  * getByte - Extract byte n from word x
152.
153.  *   Bytes numbered from 0 (LSB) to 3 (MSB)
154.
155.  *   Examples: getByte(0x12345678,1) = 0x56
156.
157.  *   Legal ops: & << >>
158.
159.  */
160.
161. int getByte(int x, int n) { return (x >> (n << 3)) & (0xff); }
162.
163.
164.
165. /*
166.
167.  * negate - return -x
168.
169.  *   Example: negate(1) = -1.
170.
171.  *   Legal ops: ~ +
172.
173.  */
174.
175. int negate(int x) { return (~x) + 1; }
176.
177.
178.
179. /*
180.
```

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

 *   Example: isPositive(-1) = 0.

 *   Legal ops: ! | >>

 */

int isPositive(int x) { return !((x >> 0x1f) | (!x)); }


/*

 * test of the above function

 */

int main(int argc, char* argv[]) {



    /*test for pow2plus4*/

    printf("\nTest for pow2plus4 function.\n");

    for (int i = 0; i < 5; i++) {

        printf("Using pow2plus4 func,\n");

        printf("2 power %d then plus 4 equals: %d.\n", i, pow2plus4(i));

        printf("Using loop to calculate power,\n");

        int power = 1;

        for (int j = 0; j < i; j++) {

            power *= 2;

        }

```

```
225.         printf("2 power %d then plus 4 equals: %d.\n", i, power + 4);
226.
227.     }
228.
229.
230.
231.
232.
233.     /*test for bitAnd*/
234.
235.     printf("\nTest for bitAnd function.\n");
236.
237.     for (int i = 1; i < 6; i++) {
238.
239.         printf("Using bitAnd func,\n");
240.
241.         printf("%d AND %d equals: %d.\n", i, 10 - i, bitAnd(i, 10 - i));
242.
243.         printf("Using operand '&',\n");
244.
245.         printf("%d AND %d equals %d.\n", i, 10 - i, i & 10 - i);
246.
247.     }
248.
249.
250.
251.
252.
253.     /*test for getByte*/
254.
255.     printf("\nTest for getByte function.\n");
256.
257.     for (int i = 0; i < 4; i++) {
258.
259.         printf("Get the %d byte of 0x%x is: 0x%x.\n", i, 0x12345678,
260.
261.                 getByte(0x12345678, i));
262.
263.     }
264.
265.
266.
267.
268.
```

```
    /*test for negate*/

    printf("\nTest for negate function.\n");

    for (int i = 0; i < 5; i++) {

        printf("the negative of %d is %d.\n", 2 - i, negate(2 - i));

    }



    /*test for isPositive*/

    printf("\nTest for isPositive function.\n");

    int arr[3] = {114514, 0, -1919810};

    for (int i = 0; i < 3; i++) {

        if (isPositive(arr[i])) {

            printf("%-8d is positive.\n", arr[i]);

        } else {

            printf("%-8d isn't positive.\n", arr[i]);

        }

    }



    return 0;

}
```