COMPUTER ORGANIZATION
EECS 2021

LAB "A" REPORT
TRANSLATING DATA TO BINARY

The work in this report is my own. I have read and understood York University academic dishonesty policy and I did not violate the senate dishonesty policy in writing this report.

X _____

Camillo John (CJ) D'Alimonte
212754396
Lab Section 2
09/23/14
Professor Aboelaze

Abstract:

This laboratory/experiment was conducted in an attempt to learn the basics behind the process of translating data into binary. By modifying input using a high-level programming language (i.e. Java), I will be able to write different programs that will alter inputted data using different operations. The methods used consist of binary operations, logical and arithmetic shifting, hexadecimal conversions, and masking. By the end of the lab, I will come to the conclusion that I can modify data by manipulating its binary values by changing bit placement, specifically the value's most significant digits.

Equipment Used:

This lab was done through the UNIX environment at the Lassonde Lab. I used the Eclipse IDE for the Java applications and the UNIX terminal to input the command line arguments.

Methods/Procedures:

For LabA1, I needed to write a Java application that takes in an integer via a command-line argument and passes it to the static method toBinaryString(int) of the Integer class. My program needed to output the returned string. Since toBinaryString takes in an integer parameter and since the command-line argument is a string, I will have to parse the string argument into an integer using Integer.parseInt().

Pseudo Code	println (Integer.toBinaryString(Integer.parseInt(args[0])))

For the first part of LabA2, I needed to write a Java application that takes in an integer via a command-line argument and passes it to the static method toHexString(int) of the Integer class. My program needed to output the returned string. Since toHexString takes in an integer parameter and since the command-line argument is a string, I will have to parse the string argument into an integer using Integer.parseInt(). Once completed, I had to modify my program to expose the suppressed higher bits using the reverse(int) method from the Integer class. My last modification was to search for a method that will allow for the reversal to be done at a byte level. This method was the reverseBytes(int) from the Integer class.

Pseudo Code	int i → Integer.parseInt(args[0]); String s → (Integer.toHexString(i)); String reverse → Integer.toHexString(Integer.reverse(i)); String reversebyte → Integer.toHexString(Integer.reverseBytes(i)); println(s, reverse, reversebyte);

For LabA3, I needed to write a Java application that takes in two integers via args[0] and args[1] and manipulates them using Java's bitwise operators. My program needed to output the corresponding results. The bitwise operators are as follows:

```
int z = x & y    // and
int z = x | y    // or
int z = x ^ y    // xor
```

int z = ~x // not

Pseudo Code	int x → Integer.parseInt(args[0]); int y → Integer.parseInt(args[1]); int z → x & y; println (Integer.toBinaryString(z)); z → x | y; println (Integer.toBinaryString(z)); z → x ^ y; println (Integer.toBinaryString(z)); z→~x; println (Integer.toBinaryString(z));

For LabA4, I needed to write a Java application that takes in an integer and manipulates it using Java's logical and arithmetic shift operators. My program needed to output the corresponding results in both binary and decimal. The shift operators are as follows:

int z = x << 1 // logical left shift
int z = x >>> 1 // logical right shift
int z = x >> 1 // arithmetic right shift

The second part of the lab had me change the shift amount from 1 to 2

Pseudo Code	int x → Integer.parseInt(args[0]);
	int z → x << 2; // logical left shift
	println(z, toBinaryString(z));
	z → x >>> 2; // logical right shift
	println(z, toBinaryString(z))
	z → x >> 2; // arithmetic right shift
	println(z, toBinaryString(z));

For LabA5, I needed to write a Java application that parses the integer x and computes and outputs the value of bit #10. It does so by using logical shifts using the following logic:

int z = x << 21;
z = z >>> 31;

The second part of the lab introduces the concept of masks. I added this code fragment to my program.

int mask = 1024;
int y = x & mask;
y = y >> 10;

Pseudo Code	int x → Integer.parseInt(args[0]); int z → x << 21; z → z >>> 31; println (z); int mask → 1024; int y → x & mask; y → y >> 10; println (y);

For LabA6, I needed to write a Java application that parses the integer x and sets bit #10 of x (i.e. makes it 1) and clears bit #11 (i.e. makes it 0).

Pseudo Code	int x → Integer.parseInt(args[0]); println(x); int mask → 1024; int secondmask → 2048; int w → mask | x; int y → (~secondmask) & w; println (y);

For LabA7, I needed to write a Java application that parses the integer x and interchanges (i.e. swaps) bit #10 and bit #20.

Pseudo Code	int x --> Integer.parseInt(args[0]);
	int mask → 1024;
	int secondmask → 1048576;
	int z → x & mask;
	z → z >> 10;
	int w → secondmask & x;
	w → w >>20;
	if(w != z){
	x → mask ^ x;
	x → secondmask ^ x;
	println(x);

For LabA8, I needed to write a Java application that takes in an integer and manipulates it using the following command: int z = 1 + ~x. My program would output the result in both binary and decimal. This lab focuses on negative arguments.

Pseudo Code	int x → Integer.parseInt(args[0]); int z → 1 + ~x; println (x, toBinaryString(x));

Results:

The following terminal screen-shots provide a view of the different results I had during each of the 8 experiments.

```
jun01 58 % javac LabA1.java
jun01 59 % java LabA1 5
101
jun01 60 % java LabA1 212
11010100
jun01 61 % java LabA1 24
11000
jun01 62 %
```

LabA1

```
jun01 64 % java LabA2 205
cd
b3000000
cd000000
jun01 65 % java LabA2 195
c3
c3000000
c3000000
jun01 66 % java LabA2 57646
e12e
74870000
2ee10000
jun01 67 % java LabA2 6
6
60000000
6000000
jun01 68 %
```

LabA2

```
jun01 76 % java LabA3 205 38
11001101
100110

100
11101111
11101011
11111111111111111111111100110010
jun01 77 % java LabA3 34 25
100010
11001

0
111011
111011
11111111111111111111111111011101
jun01 78 % java LabA3 3535 6756
110111001111
1101001100100

100001000100
1111111101111
1011110101011
11111111111111111111001000110000
jun01 79 %
```

LabA3

```
jun01 84 % java LabA4 45
180
10110100
11
1011
11
1011
jun01 85 % java LabA4 5654
22616
101100001011000
1413
10110000101
1413
10110000101
jun01 86 %
```

LabA4

```
jun01 88 % java LabA5 5000
0
0
jun01 89 % java LabA5 6000
1
1
jun01 90 % java LabA5 1024
1
1
```

LabA5

```
jun15 91 % java LabA6 45335
1011000100010111
1011010100010111
jun15 92 % java LabA6 12134
10111101100110
10011101100110
jun15 93 %
```

LabA6

```
jun01 100 % pico LabA7.java
jun01 101 % javac LabA7.java
jun01 102 % java LabA7 67
1000011
1000011
jun01 103 % java LabA7 45646546
10101110001000001011010010
10101010001000011011010010
jun01 104 %
```

LabA7

```
jun01 106 % java LabA8 -5
5
101
jun01 107 % java LabA8 113
-113
11111111111111111111111110001111
jun01 108 % java LabA8 -34
34
100010
jun01 109 %
```

LabA8

Discussion:

My analysis of the results is separated for each experiment.

LabA1:

I concluded that bit number 0 of any even integer must be zero. Hence, the binary conversion of 24 is 11000. I noticed that if the binary representation of an integer ends (at the least significant end) with 00, then the integer is a multiple of 4. If the binary representation of the integer ends with 01, then the number modulo 4 is 1. For example, if you divide that number by 4, the remainder is 1. This pattern can also be expressed as 4k+1.

LabA2:

The method that allows me to reverse at the byte level is the reverseBytes(int) method from the Integer class. When I used 195 as my input, both the reverse() and reverseBytes methods outputted the same result: c3000000.

LabA3:

I was able to predict the output of my program with input x = 205 and y = 38 using Java's bitwise operations. Given 205 in binary is 11001101 and 38 in binary is 100110, I can use my knowledge of binary operations to correctly predict my program's output.

11001101

00100110

AND: 00000100 --> 100

OR: 11101111

XOR: 11101011

LabA4:

I was able to check my program's output by just checking the binary shifting on a piece of paper. For example, when I inputted 9 (Binary: 1001), the shift by two would result in both 1 values shifting over by two bits. The 1 in the eights column moves to the 32nd column while the 1 in the first column moves to the fourth column. Therefore, 32 + 4 = 36 and the new binary value is 100100. I wasn't able to report any difference in either of the right shifts.

LabA5:

Since an argument of 5000 would yield a 0 output and an argument of 6000 would yield an output of 1, I needed to program 1024 to behave as a mask. When I tested my program, 1024

indeed behaved like a mask as my input turned out to be 1. I was also able to verify that this mask-based approach would output the state of bit #10, in this case the mask-based y and the shift-based z were the same.

LabA6 and LabA7:

Both these labs were tricky for me as I didn't fully understand the concept of masking. In the beginning, I started off testing both labs with inadequately sized input so it made testing and debugging a lot harder. Not until I started entering large enough inputs did I see that my code was working correctly. Both labs took me a considerable amount of time to complete.

LabA8:

I concluded the given an integer x, int z = 1 + ~x will output the opposite value of x. For example, it the input is 5, then the program will output -5 in both binary and decimal. If the input is -17, then the program will output 17. The value of ~x is NOT x (the opposite – 1). Since the -1 and 1 cancel out, then we are left with the opposite value of x.

Conclusion:

I was able to better my understand on how data is converted into binary representation. I now know the different binary operations that are used in the conversion process. I was introduced to the concept of shift operations and their usefulness in manipulating binary strings. I want to continue reading more references on the usage of masks since that was the only concept that I found difficult to grasp.

References:

No external references were used for this lab other than the lab material provided to us at the beginning of the lab.

Appendix:

The source code for all 8 experiments is included below.

LabA1:

```
public class LabA1 {

        public static void main (String [] args){

                System.out.println(Integer.toBinaryString(Integer.parseInt(args[0])));

                } }
```

LabA2:

```
public class LabA2 {
        public static void main (String [] args){
        int i = Integer.parseInt(args[0]);
        String s = (Integer.toHexString(i));
        String reverse = Integer.toHexString(Integer.reverse(i));
        String reversebyte = Integer.toHexString(Integer.reverseBytes(i));
        System.out.println(s);
        System.out.println(reverse);
        System.out.println(reversebyte);
        } }
```

LabA3:

```
public class LabA3 {
        public static void main (String [] args){
                int x = Integer.parseInt (args[0]);
                System.out.println(Integer.toBinaryString(x));
                int y = Integer.parseInt (args[1]);
                System.out.println(Integer.toBinaryString(y));
                System.out.println("");
                int z = x & y;
                System.out.println (Integer.toBinaryString(z));
                z = x | y;
                System.out.println (Integer.toBinaryString(z));
                z = x ^ y;
                System.out.println (Integer.toBinaryString(z));
```

```
            z = ~x;
        System.out.println (Integer.toBinaryString(z));
    } }
```

LabA4:

```
public class LabA4 {
    public static void main (String [] args){
            int x = Integer.parseInt(args[0]);
            int z = x << 2; // logical left shift
            System.out.println (z);
            System.out.println (Integer.toBinaryString(z));
            z = x >>> 2; // logical right shift
            System.out.println (z);
            System.out.println (Integer.toBinaryString(z));
            z = x >> 2; // arithmetic right shift
            System.out.println (z);
        System.out.println (Integer.toBinaryString(z));
    } }
```

LabA5:

```
public class LabA5 {
    public static void main (String [] args){
            int x = Integer.parseInt(args[0]);
            int z = x << 21;
            z = z >>> 31;
            System.out.println (z);
```

```
                int mask = 1024;
                int y = x & mask;
                y = y >> 10;
                System.out.println (y);
                        } }
```

LabA6:

```
public class LabA6 {
        public static void main (String args []) {
                int x = Integer.parseInt(args[0]);
                System.out.println(Integer.toBinaryString(x));
                int mask = 1024;
                int secondmask = 2048;
                int w = mask | x;
                int y = (~secondmask) & w;
                System.out.println(Integer.toBinaryString(y));
                } }
```

LabA7:

```
public class LabA7{
        public static void main (String args[]){
                int x = Integer.parseInt(args[0]);
                System.out.println(Integer.toBinaryString(x));
                int mask = 1024;
                int secondmask = 1048576;
                int z = x & mask;
                z = z >> 10;
```

```
                int w = secondmask & x;
                w = w >>20;
                if(w != z)  {
                x = mask ^ x;
                x = secondmask ^ x;
                }
                System.out.println(Integer.toBinaryString(x));
                } }
```

LabA8:

```
public class LabA8 {
        public static void main (String args [] ){
                int x = Integer.parseInt(args[0]);
                int z = 1 + ~x;
                System.out.println (z);
                System.out.println (Integer.toBinaryString(z));
                }  }
```