Robert Kulesza
November 3, 2018
Professor Erkan
Computer Organization & Architecture

Project One

For project one, there were two problems to be solved. The first was to test whether or not the value stored in memory location x3030 was even or odd. If the value was even then 0 should be stored at the memory location x3031. If the value was odd then 1 should be stored at the memory location x3031. The second problem was to count the number of 1's in a value at memory location x3030 and store it at memory location x3032. In the problem description were a few restrictions. The program cannot overwrite the original value in memory location x3030, must be written in LC-3 binary, and must originate at memory location x3000.

For the solution to problem 1, the program used a bitwise-and operation to calculate whether or not the value stored at x3030 was even or odd. The bitwise-and operation will return 1 wherever the corresponding bits are both 1. It will return 0 wherever that condition is not met. Also, it is important to remember that in a binary number system, even numbers end in zero and odd numbers end in 1. If a program executes a bitwise-and operation on an even number and 1 then it will return 0. This is because the even number ends in the digit 0 and 1 ends in 1. If you conduct a bitwise-and operation on an odd number and 1 then it will return 1. This is because the odd number ends in the digit 1 and 1 also ends in the digit 1. Thus the result of this operation matches the output requested in problem one. Finally, the result of this operation must then be stored at location x3031.

For the solution to problem 2, the program changed the input value to positive via a 1's complement if it was negative. Subsequently the program used a "test operand" that doubled on each iteration of a loop. The "test operand" began with the value 1 and thus on each iteration it became the subsequent power of 2. The loop breaks when the "test operand" becomes larger than the input. During each iteration of the loop, the "test operand" is operated against the input in a bitwise-and operation. Keep in mind that a positive power of 2 always starts with exactly one 1 in its binary representation followed by all zeros. The result of a bitwise-and operation between the input and a positive power of 2 is 0 for all bits before and after that single 1-bit in the power of 2. Thus that single bit determines whether the operation returns 0 or the power of 2 used in the calculation. If the input number has a 1 corresponding to the 1-bit in the power of 2, then the result of the bitwise-and operation is the power of 2 used in the computation. Otherwise the result is 0. During every iteration of the loop, a bitwise-and was calculated between the "test operand" which is a power of 2 and the input value. By doing this, the program tested whether or not any of the digits of the input value were zero or one. If the digit was zero, nothing happened. If the digit was one, then the value of the "ones-counter" incremented. In that way the "ones-counter" kept track of the quantity of 1s in the binary representation of the input value. At the end of the loop, if the input value was originally negative, then the program subtracts the value in "ones-counter" from 16 and puts that result back into "ones-counter." This works because the program counted the ones in the ones-complement of the original value. In other words, the "ones-counter" in this case counted the number of zeros in the original value. To get the amount of ones in the original input all that is needed is to subtract the total number of zeros from 16. Finally, the value in the "ones-counter" matched the requested output and could be stored at memory address x3032 as per the instructions.

```
0011 0000 0000 0000 ;.orig x3000                                        ;problem 1
0010 0000 0010 1111 ;                     LD R0 x2f
0101 0010 0010 0001 ;                     AND R1 R0 X1
0011 0010 0010 1110 ;                     ST R1 x2e
0101 1101 1010 0000 ;                     AND R6 R6 x0          ;problem 2
0001 0000 0010 0000 ;                     ADD R0 R0 x0
0000 1000 0000 0001 ;                     BRn NEGIN
0000 1110 0000 0011 ;                     BR NOT_NEGIN
0001 1101 1010 1000 ;NEGIN                ADD R6 R6 x8
0001 1101 1010 1000 ;                     ADD R6 R6 x8
1001 0000 0011 1111 ;                     NOT R0 R0
0101 0010 0110 0000 ;NOT_NEGIN            AND R1 R1 x0
0101 0100 1010 0000 ;                     AND R2 R2 x0
0001 1000 1010 0000 ;LOOP                 ADD R4 R2 x0
0101 0110 1110 0000 ;                     AND R3 R3 x0
0001 0110 1110 0001 ;                     ADD R3 R3 x1
0001 1001 0010 0000 ;LOOP2                ADD R4 R4 x0
0000 1100 0000 0011 ;                     BRnz END_LOOP2
0001 0110 1100 0011 ;                     ADD R3 R3 R3
0001 1001 0011 1111 ;                     ADD R4 R4 x-1
0000 1111 1111 1011 ;                     BR LOOP2
1001 1000 0011 1111 ;END_LOOP2            NOT R4 R0
0001 1001 0010 0001 ;                     ADD R4 R4 x1
0001 1000 1100 0100 ;                     ADD R4 R3 R4
0000 0010 0000 0111 ;                     BRp END_LOOP
0101 0110 1100 0000 ;                     AND R3 R3 R0
0000 0010 0000 0010 ;                     BRp ADDONE
0001 0010 0100 0011 ;                     ADD R1 R1 R3
0000 1110 0000 0001 ;                     BR COND
0001 0010 0110 0001 ;ADDONE               ADD R1 R1 x1
0001 0100 1010 0001 ;COND                 ADD R2 R2 x1
0000 1111 1110 1101 ;                     BR LOOP
0001 1101 1010 0000 ;END_LOOP             ADD R6 R6 x0
0000 0100 0000 0011 ;                     BRz  DONE
1001 0010 0111 1111 ;                     NOT R1 R1
0001 0010 0110 0001 ;                     ADD R1 R1 x1
0001 0011 1000 0001 ;                     ADD R1 R6 R1
0011 0010 0000 1101 ;DONE                         ST R1 xd
1111 0000 0010 0101 ;                     TRAP X25 (.end)
```

In conclusion, both problems were solved using the repeated use of a bitwise-and operation. That said, there are other ways to solve these problems. In higher level programming languages, I would use the modulo or the remainder operation to detect if something was even or odd. And I would use a similar method to count the 1s in the binary representation of the input value at x3030. Despite the fact that there are bitwise operations in higher level languages as well. This algorithm seems to be considerably faster than division.