HW7 - Due Fri. Dec 10th

Use the "homework8" Dropbox in Learn@UW to turn in all your solutions. One copy per group is sufficient, please make sure you include your name and your partner's name in the document you upload.

Question 1. (2 points)

We want the following program fragment to shift R3 to the left by four bits, but it has an error in it. Identify the error and explain how to fix it.

```
.ORIG
AND   R2, R2, #0
ADD   R2, R2, #4
LOOP  BRz   DONE
ADD   R2, R2, #-1
ADD   R3, R3, R3
BR    LOOP
DONE  HALT
.END
```

ANSWER: ADD R2, R2, #-1 and R3, R3 should interchange their places, the loop is going to be taken as long as R2 is non-zero, however in the given program, ADD R3, R3, R3 will change the N,Z,P bits and BRz will not stop unless the R3 is wholly shifted 16 times.

Question 2. (4 points)

Consider the following LC-3 assembly language program. Assuming that the memory locations DATA get filled before the program executes, what is the relationship between the final values at DATA and the initial values at DATA?

```
.ORIG x3000
LEA  R0, DATA  ; R0 points to the head of the DATA array
AND  R1, R1, #0 ; clear R1
ADD  R1, R1, #9 ; R1 keeps track of the outer loop counter, initialized to 9
LOOP1 ADD  R2, R0, #0 ; R2 traverse over the DATA, initialized to the head of the DATA array
ADD  R3, R1, #0 ; R3 keeps track of the inner loop counter, initialized to 9
LOOP2 JSR  SUB1 ; jumps to sub1
ADD  R4, R4, #0
BRzp  LABEL ; if (DATA[i+1] >= DATA[i]) goto LABEL
JSR  SUB2 ; if (DATA[i+1] < DATA[i]) jump to sub2
```
LABEL     ADD R2, R2, #1 ; point to the next location in memory
ADD R3, R3, #1  ; decrement the inner loop counter
BRp LOOP2
ADD R1, R1, #1  ; decrement the outer loop counter
BRp LOOP1
HALT
DATA .BLKW #10
SUB1 LDR R5, R2, #0 ; R5 <- DATA[i]
NOT R5, R5 ; R5 <- NOT (R5)
ADD R5, R5, #1 ; R5 <- R5 + 1 ; the result is 2’s complement of DATA[i]
LDR R6, R2, #1 ; R6 <- DATA[i+1]
ADD R4, R5, R6 ; R4 = DATA[i+1] - DATA[i]
RET ; return
SUB2 LDR R4, R2, #0 ; R4 <- DATA[i]
LDR R5, R2, #1 ; R5 <- DATA[i+1]
STR R4, R2, #1 ;
STR R5, R2, #0 ; DATA[i] <-> DATA[i+1]; exchange the values in DATA[i]
RET ; return
.END

Answer: The final values at DATA are sorted, the algorithm provided is actually the bubble-sort.

Question 3 (6 points)

The assembly process is a two-pass process. The First Pass creates a symbol table.

a. What is the use of the symbol table? Does the table have entries for pseudo-ops like .ORIG and .END? why/why not?
A symbol table is a data structure used by a language translator such as a compiler or interpreter, where each identifier in a program's source code is associated with information relating to its declaration or appearance in the source, such as its type, scope level and sometimes its location. [Wikipedia]

No, .ORIG and .END are assembler directives. They are information that the assembler uses and do not find their way on to the compiled binary. Hence they do not have symbol table entries.

b. Create a symbol table for the code in Question 2.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP1</td>
<td>0x3003</td>
</tr>
<tr>
<td>LOOP2</td>
<td>0x3005</td>
</tr>
<tr>
<td>LABEL</td>
<td>0x3009</td>
</tr>
<tr>
<td>DATA</td>
<td>0x300F</td>
</tr>
<tr>
<td>SUB1</td>
<td>0x3010</td>
</tr>
<tr>
<td>SUB2</td>
<td>0x3016</td>
</tr>
</tbody>
</table>

**Question 4 (8 points)**

Assume that you have the following table in your program:

```
MASKS .FILL  x0001
      .FILL x0002
      .FILL x0004
      .FILL x0008
      .FILL x0010
      .FILL x0020
      .FILL x0040
      .FILL x0080
      .FILL x0100
      .FILL x0200
      .FILL x0400
      .FILL x0800
      .FILL x1000
```
a. Write a subroutine CLEAR in LC-3 assembly language that clears a bit in R0 using the table above. The index of the bit to clear is specified in R1. R0 and R1 are inputs to the subroutine.

```
CLEAR
    LEA    R3, MASKS ;
    LDR    R3, R3, R1 ; R3 point to the correct mask
    NOT    R3, R3 ; to clear a bit we have to invert the mask to make all the
                    ; bits except the one we wish to clear becomes one
    AND    R0, R0, R3 ; then AND it with the number
```

b. Write a similar subroutine SET that sets the specified bit instead of clearing it.

```
SET
    LEA    R3, MASKS
    LDR    R3, R3, R1
    NOT    R3, R3 ; To set a bit we have to OR that number with mask, but
                    ; LC-3 assembly does not have OR function, so in
                    ; following lines we implement R0 OR R4 by using AND
                    ; and NOT: R0+R4 = NOT(NOT(R0) AND NOT(R4))
    NOT    R0, R0 ;
    AND    R0, R0, R3 ;
    NOT    R0, R0 ;
```

**Question 5. (6 points)**

Figure out what the following programs do.

(a) (3 points)

```
.ORIG    x3000
```
Answer: if the value at A is prime Result contains a 1, else Result contains 0.

(b) (3 points)

.ORIG x3000
AND R5, R5, #0 ; Clear R5
AND R3, R3, #0 ; Clear R3
ADD R3, R3, #8 ; R3=8 (loop counter)
LEA R0, BB ; R0 = 0xBB
LDR  R1, R0, #1  ; R1 = 0x4000
LDR  R1, R1, #0  ; R1 = M[0x4000]
ADD  R2, R1, #0  ; R2 = M[0x4000]
AGAIN ADD R2, R2, R2  ; shift R2 to the left 8 times so the low-order 8 bits
                      ; are now in the position of high-order 8 bits and the
                      ; low order 8-bits in the shifted number are zero.
ADD  R3, R3, #-1
BRp  AGAIN
LDR  R4, R0, #0  ; R4 = 0xFF00
AND  R1, R1, R4  ; R1 = M[0x4000] & 0xFF00 (returns the
                 ; high-order 8 bits in the position of high-order 8
                 ; bits)
NOT  R1, R1
ADD  R1, R1, #1
ADD  R2, R2, R1
BRnp NO
ADD  R5, R5, #1

NO TRAP x25
BB  .FILL xFF00
    .FILL x4000
    .END

Answer:
It puts 1 in R5 if the high byte of the data in location x4000 is equal to the low byte of
the data in location x4000.

**Question 6 (4 points)**

You are given two pieces of program. They have been written by different programmers to store
x0015 into memory location x3000.

a.

```
.ORIG  x5000
AND  R0, R0, #0
ADD  R0, R0, #15
ADD  R0, R0, #6
STI  R0, PTR
<rest of the program.....>
HALT
```
(1) Why there are two ‘ADD’ instructions in module (a)? (1 point)
The range of the immediate value in ADD instruction is only from +15 to -16. +15 is the maximum number we can add at a time. Therefore the addition needs two steps.

(2) Explain the fundamental differences in their approaches? (1 point)
The first approach uses instructions to dynamically initialize the value stored in some memory location. The initialization is done at run time. The second approach uses the pseudo-op provided by the assembler to statically initialize the memory content. The initialization is done at assembly time.

(3) Give one example where module a’s approach is preferred. (2 points)
The advantage of the second approach is simplicity. However, it requires that the programmer knows what the value should be before running the program. Sometimes, the initial value can only be calculated by running part of the program. Furthermore, sometimes the necessity of initializing some memory location is also determined at run time.
In both scenarios, you can use the first approach to come up with a subroutine and reuse it when it is necessary. (you will have to modify it a little bit).