ECE 220 Summer 2017 Midterm 1

On-campus and Western Hemisphere students

- Absolutely no interaction between students or any external help is allowed!

- You are not allowed to use any applications on your desktop outside the Exam VM.
- You can use any applications inside the Exam VM.

- You are permitted one double-sided page of **hand-written** notes.
- LC-3 instructions are provided at the end of this exam booklet.

- Read entire problem description before starting to work on the solution.

- IMPORTANT: Once done, you must commit your work to Subversion:

```
cd ~/midterm1
svn commit -m "I am done"
```

- To verify your submission, point web browser inside the Exam VM to the following page: **https://subversion.ews.illinois.edu/svn/su17-ece220/NETID/midterm1** where NETID is your actual NetID.

- If your attempt to commit/verify fails, make sure you are still connected to the UIUC network via the VPN client. If your connection to UIUC network gets lost during the exam, you must reconnect again.

Good luck!

Part 1: I/O and Stack

Problem Statement

Write a subroutine called `IS_VALID` that reads characters entered by the user from the keyboard *without using any TRAPs* and returns 1 if the input consists of a "valid" sequence of characters, or 0 otherwise. "Valid sequence" means that each lowercase symbol has a matching uppercase symbol (only alphabetical characters are used) and the pairs of symbols are properly "nested". Here are some example inputs and corresponding outputs:

Input	Output
abcCBA	1
abBcdDCA	1
zaAddbBDcCDZ	1

Input	Output
wxyzbaAB	0
aAB	0
zaAcCfeE	0

Implementation Requirements

Your code should read input from the keyboard *and* echo it to the screen, *without using any TRAPs*. If you do not remember how to implement this without TRAPs, you can use TRAPs, but there will be a penalty of 10% for use of I/O TRAPs.

You may assume that the input is terminated when Enter key character (xD) is encountered and that the input will consist of only the lowercase, uppercase, and the newline characters. No other characters will be used and there is no need for input error checking.

You must write a subroutine, `IS_VALID`, that accepts input from the keyboard and continues processing it until Enter key character is encountered. It then returns 1 in R0 if the entered string is "valid", or 0 otherwise. You must use provided `PUSH` and `POP` subroutines. Algorithm for `IS_VALID` subroutine is provided below.

Complete your code in `part1.asm` in your `part1` folder. Do not forget to commit your work!

Algorithm for `IS_VALID`

```
0. Assume that the input is valid, set R0<-1
1. Read one input character from the keyboard
2. Check entered character:
   a) If the character is lowercase, push it onto stack
   b) If the character is uppercase:
       - pop from stack; if the popped character is
         - the matching lowercase character, then continue to step 1
         - else sequence is not valid, clear R0 and continue to step 1
   c) If the current character is 'Enter key' character:
      - if the stack is not empty, clear R0
      - return
```

Grading rubric

Item	Grade %
Code assembles, runs, and halts	5%
Main program contains a proper call to `IS_VALID`	5%
All registers that are modified by `IS_VALID` are restored to their original values on return	5%
All registers are properly initialized	5%
`IS_VALID` acquires input string from the keyboard until user hits ENTER key	10%
`IS_VALID` reads characters from the keyboard *without* using any TRAPs	10%
`IS_VALID` echoes input characters to the display *without* using any TRAPs	10%
`IS_VALID` uses stack subroutines PUSH and POP and implements the above algorithm	35%
On exit, `IS_VALID` returns result in R0	5%
Code uses as few as possible iterative and conditional constructs	5%
Subroutine is well-documented (description of functionality, register table, comments, proper source code formatting, etc.)	5%

Supplied part1.asm code

```
.ORIG x3000
; main code goes here

    HALT

; IS_VALID subroutine implementation goes here

    RET

ASCII_ENTER .FILL xD

ASCII_la .FILL x61  ; ASCII value for 'a'
ASCII_lz .FILL x7A  ; ASCII value for 'z'
ASCII_uA .FILL x41  ; ASCII value for 'A'
ASCII_uZ .FILL x5A  ; ASCII value for 'Z'

KBSR .FILL xFE00
KBDR .FILL xFE02
DSR  .FILL xFE04
DDR  .FILL xFE06

; Do Not Write Below This Line!
; ---------------------------------

; PUSH onto the stack
; IN: R0
; OUT: R5 (0-success, 1-fail/overflow)

; POP from the stack
; OUT: R0, R5 (0-success, 1-fail/underflow)
;

.END
```

Part 2: Subroutines

Problem Statement

Using *linear congruential generator* (LCG) algorithm, write a program that generates a sequence of *N* "pseudorandom" numbers and stores them in memory starting from address x4000.

Algorithm

LCG algorithm works as follows. The generator is defined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m$$

where X_i is the sequence of pseudorandom values, m is the "modulus" ($m > 0$), a is the "multiplier" ($0 < \mathrm{a} < m$), c is the "increment" ($0 \leq c < m$), and X_0 is the "seed", or "start value" of the sequence of pseudo-random numbers ($0 \leq X_0 < m$).

For example, for $a = 4, c = 1, m = 9, X_0 = 0$ and $N = 9$, the following sequence should be generated: 1, 5, 3, 4, 8, 6, 7, 2, 0.

Implementation Requirements

Your implementation must consist of two subroutines: `RNG` and `LCG`. Your main program must call `RNG` subroutine with the following parameters: R0<- *a*, R1 <- *c*, R2 <- *m*, R3 <- *seed*, R4 <- *N*, and R5 <- *address* of the output array. `RNG` subroutine is responsible for calling `LCG` subroutine *N* times and storing the random numbers returned by `LCG` into consecutive memory locations starting from the address passed to `RNG` in register R5. `LCG` subroutine must use the following parameters: R0 <- *a*, R1 <- *c*, R2 <- *m*, R3 <- *seed*. It must return newly generated random number in R3.

`MULT` and `DIVIDE` subroutines are provided and should be used by `LCG` subroutine to compute the product and remainder. Do not modify them.

Complete your code in `part2.asm` in your `part12` folder. Do not forget to commit your work!

Grading rubric

Item	Grade %
Code assembles, runs, and halts	5%
Main program contains a proper call to `RNG`	5%
All registers that are modified by `RNG` and `LNG` are restored to their original values on return	5%
All registers are properly initialized	5%
`RNG` and `LNG` use parameters as specified in the above implementation requirements	5%
`RNG` calls `LNG` *N* times and stores the results in consecutive memory locations starting from the address passed to `RNG` in R5	15%
`LNG` correctly implements the above algorithm	35%
`LNG` makes proper calls to `DIVIDE` and `MULT` subroutines	10%
`LNG` returns result in R3	5%
Code uses as few as possible iterative and conditional constructs	5%
Subroutines are well-documented (description of functionality, register table, comments, proper source code formatting, etc.)	5%

Supplied part2.asm code

```
.ORIG x3000

; main code goes here
; IMPLEMENT ME!
    ; setup arguments and call RNG

    HALT

; LCG model parameters
a .FILL #4
c .FILL #1
m .FILL #9
seed .FILL #0  ; aka X0

; number and address of random numbers to generate
N .FILL #9
addr .FILL x4000

; RNG subroutine implementation
;
RNG
    ; IMPLEMENT ME
    RET

; LCG subroutine implementation
;
LCG
    ; IMPLEMENT ME
    RET


; Do Not Write Below This Line!
; ---------------------------------
;
; DIVIDE - divides R1 by R2 and returns R0 and R3
; IN:  R1: numerator (dividend, N)
;      R2: denominator (divisor, D)
;      (R1 and R2 must be strictly > 0)
; OUT: R0: quotient, Q (Q = N / D)
;      R3: remainder, R

; MULT multiplies two numbers
; IN: R1, R2 (R2 must be strictly > 0)
; OUT: R0 <- R1 * R2
```

NOTES: RTL corresponds to execution (after fetch!); JSRR not shown