ECE 220 Summer 2016 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/su16-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_BALANCED` that reads a string of characters entered by the user from the keyboard *without using any TRAPs* and returns 1 if the input string consists of a "balanced" sequence of parentheses, or 0 otherwise. "Balanced parentheses" means that each opening symbol "(" has a corresponding closing symbol ")" and the pairs of parentheses are properly nested. Here are some example inputs and corresponding outputs:

Input	Output
(()()()())	1
(((())))	1
(()((())()))	1

Input	Output
((((((())	0
())	0
(()()(()	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 each use of an I/O TRAP.

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

You must write a subroutine IS_BALANCED that accepts input string from the keyboard and if it detects unbalanced parentheses, immediately returns 0 in R0. Otherwise, it continues checking until Enter key character is encountered and returns 1 in R0 if the entered string is "balanced", or 0 otherwise. You must use provided `PUSH` and `POP` subroutines.

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

Algorithm

```
1. Read input character from the keyboard
2. Check entered character:
   a) If the current character is an opening bracket '(', push it to stack
   b) If the current character is a closing bracket ')':
       - pop from stack; if the popped character is
         - the matching opening bracket, then continue to step 1
         - else parenthesis are not balanced, stop and return 0
   c) If the current character is 'Enter key' character:
      - if the stack is not empty, return 0
      - else return 1
```

Grading rubric

Item	Grade %
Code assembles, runs, and halts	5%
Main program contains a proper call to `IS_BALANCED`	5%
All registers that are modified by `IS_BALANCED` are restored to their original values on return	5%
All registers are properly initialized	5%
`IS_BALANCED` acquires input string from the keyboard until user hits ENTER key or when unbalanced sequence of brackets is encountered	10%
`IS_BALANCED` reads characters from the keyboard *without* using any TRAPs	15%
`IS_BALANCED` echoes input characters to the display *without* using any TRAPs	15%
`IS_BALANCED` uses stack subroutines PUSH and POP and implements the above algorithm	25%
On exit, `IS_BALANCED` 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_BALANCED subroutine implementation goes here

    RET

ASCII_ENTER .FILL xD
ASCII_OPEN .FILL x28   ; ASCII value for '('
ASCII_CLOSE.FILL x29   ; ASCII value for ')'
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

Write a program that computes all prime numbers on the interval [P, Q] where P and Q are user-supplied values and P < Q. A *prime number* is an integer positive number greater than 1 that has no positive divisors other than 1 and itself. For example, 5 is prime because 1 and 5 are its only positive integer factors, whereas 6 is not prime because it has the divisors 2 and 3 in addition to 1 and 6.

Algorithm

A simple method for verifying if a given number N is prime consists of testing whether the given number N is a multiple of any integer between 2 and square root of N. This routine consists of dividing N by each integer *m* that is greater than 1 and less than or equal to the square root of N. If the result of any of these divisions is an integer, then N is not a prime, otherwise it is a prime. For example, for N = 37, the trial divisions are by *m* = 2, 3, 4, 5, and 6. None of these numbers divides 37, so 37 is prime. Since computing square root in LC-3 assembly is non-trivial, you can check for *m* = 2, 3, …, N/2.

There are 168 prime numbers less than 1000:

2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983, 991, 997

Implementation Requirements

Your program should read values P and Q from memory locations x4000 and x4001. The resulting table of prime numbers on the interval from P to Q should be stored in memory starting from address x4002. You can assume that P and Q are positive numbers larger than 1 and less than 1000.

Your program must consist of three subroutines: `IS_PRIME`, `COMPUTE_PRIMES`, and `DIVIDE`. Your main code must call `COMPUTE_PRIMES` with the following arguments: R0 <- P, R1 <- Q, and R2 <- x4002 (address starting from which all prime numbers will be stored in memory). On exit, this subroutine returns the number of found prime numbers in R6. <u>Note that this subroutine is already fully implemented for you and you must NOT modify it.</u>

`IS_PRIME` is called by `COMPUTE_PRIMES` to check if a given number passed to it in R0 is a prime number. It should return 1 in R5 if the number is prime or 0 otherwise. `IS_PRIME` should use the provided `DIVIDE` subroutine.

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 COMPUTE_PRIMES	5%
All registers that are modified by IS_PRIME are restored to their original values on return	5%
All registers are properly initialized	5%
IS_PRIME accepts input in R0	5%
IS_PRIME correctly implements the above algorithm	40%
IS_PRIME checks all possible divisors from 2 to N/2	10%
IS_PRIME makes proper call to DIVIDE subroutine	10%
IS_PRIME returns result in R5	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 part2.asm code

```
.ORIG x3000
; main code goes here

; IMPLEMENT ME!
    ; setup arguments and call COMPUTE_PRIMES
    HALT

P .FILL x4000
Q .FILL x4001
R .FILL x4002

; COMPUTE_PRIMES subroutine implementation is provided - DO NOT MODIFY IT
; IN: R0 <- P, R1 <- Q, ([P, Q] interval), R2 <- address
; OUR: R6 <- count

; IS_PRIME subroutine implementation goes here
IS_PRIME

; 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
```

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

Instruction	Encoding	Assembly	RTL
ADD	0001 · DR · SR1 · 0 · 00 · SR2	ADD DR, SR1, SR2	DR ← SR1 + SR2, Setcc
ADD	0001 · DR · SR1 · 1 · imm5	ADD DR, SR1, *imm5*	DR ← SR1 + SEXT(imm5), Setcc
AND	0101 · DR · SR1 · 0 · 00 · SR2	AND DR, SR1, SR2	DR ← SR1 AND SR2, Setcc
AND	0101 · DR · SR1 · 1 · imm5	AND DR, SR1, *imm5*	DR ← SR1 AND SEXT(imm5), Setcc
BR	0000 · n · z · p · PCoffset9	BR{nzp} *PCoffset9*	((n AND N) OR (z AND Z) OR (p AND P)): PC ← PC + SEXT(PCoffset9)
JMP	1100 · 000 · BaseR · 000000	JMP BaseR	PC ← BaseR
JSR	0100 · 1 · PCoffset11	JSR *PCoffset11*	R7 ← PC, PC ← PC + SEXT(PCoffset11)
TRAP	1111 · 0000 · trapvect8	TRAP *trapvect8*	R7 ← PC, PC ← M[ZEXT(trapvect8)]
LD	0010 · DR · PCoffset9	LD DR, *PCoffset9*	DR ← M[PC + SEXT(PCoffset9)], Setcc
LDI	1010 · DR · PCoffset9	LDI DR, *PCoffset9*	DR ← M[M[PC + SEXT(PCoffset9)]], Setcc
LDR	0110 · DR · BaseR · offset6	LDR DR, BaseR, *offset6*	DR ← M[BaseR + SEXT(offset6)], Setcc
LEA	1110 · DR · PCoffset9	LEA DR, *PCoffset9*	DR ← PC + SEXT(PCoffset9), Setcc
NOT	1001 · DR · SR · 111111	NOT DR, SR	DR ← NOT SR, Setcc
ST	0011 · SR · PCoffset9	ST SR, *PCoffset9*	M[PC + SEXT(PCoffset9)] ← SR
STI	1011 · SR · PCoffset9	STI SR, *PCoffset9*	M[M[PC + SEXT(PCoffset9)]] ← SR
STR	0111 · SR · BaseR · offset6	STR SR, BaseR, *offset6*	M[BaseR + SEXT(offset6)] ← SR