EE 306, Fall 2016
Programming Lab 4
Due: Saturday, November 19, 11:59 PM

You must do every programming assignment by yourself. You are permitted to get help ONLY from the TAs and the instructor. **Absolutely no late assignments will be accepted.**

Setup

The starter files for Lab 4 can be found here. Once you download the lab zip file, you can extract it the same way you did for Lab 0. This time the lab directory contains a file called `lab4.asm`. You will write all the code for the programming lab in this file. You can find the simulator here.

Overview

Nim is a simple two-player game that probably originated in China (although the name is thought be German, from *nimm* the German word for "take"). There are many variations of this game with respect to the type of counters used (stones, matches, apples, etc.), the number of counters in each row, and the number of rows in the game board.

In our variation of Nim, the game board consists of three rows of rocks. Row A contains 3 rocks, Row B contains 5 rocks, and Row C contains 8 rocks.

The rules are as follows:

- Each player takes turns removing one or more rocks from a single row.
- A player cannot remove rocks from more than one row in a single turn.
- The game ends when a player removes the last rock from the game board. The player who removes the last rock loses.

Assignment

Your goal is to write an LC-3 program that lets you play the game of Nim. The program must start at x3000. At the beginning of the game you should display the initial state of the game board. Before each row of rocks you should output the name of the row, for instance "Row A:". You should use the ASCII character lowercase "o" (ASCII code x006F) to represent a rock. The initial state of the game board should look as follows:

```
ROW A: ooo
ROW B: ooooo
ROW C: oooooooo
```

Player 1 always goes first, and play alternates between Player 1 and Player 2. At the beginning of each turn you should output which player's turn it is, and prompt the player for her move. For Player 1 this should look as follows:

```
Player 1, choose a row and number of rocks:
```

To specify which row and how many rocks to remove, the player should input a letter followed by a number (they do NOT need to press Enter after inputting a move). The letter (A, B, or C) specifies the row, and the number (from 1 to the number of rocks in the chosen row) specifies how many rocks to remove. Your program must make sure the player's move has a valid row and number of rocks. If the player's move is invalid, you should output an error message and prompt the same player for a move. For example, if it is Player 1's turn:

```
Player 1, choose a row and number of rocks: D4
Invalid move. Try again.
Player 1, choose a row and number of rocks: A9
Invalid move. Try again.
Player 1, choose a row and number of rocks: A*
Invalid move. Try again.
Player 1, choose a row and number of rocks: &4
Invalid move. Try again.
Player 1, choose a row and number of rocks:
```

Your program should keep prompting the player until a valid move is chosen. Be sure your program echoes the player's input to the screen as they type it. After you have echoed the player's input, you should output a newline character (ASCII code x000A) to move the cursor to the next line.

After a player has chosen a valid move, you should check for a winner. If there is one, display the appropriate banner declaring the winner. If there is no winner, your program should update the state of the game board to reflect the move, re-display the updated game board, and continue with the next player's turn. When a player has removed the last rock from the game board, the game is over. At this point, your program should display the winner and then halt. For example, if Player 2 removes the last rock, your program should output the following:

```
Player 1 Wins.
```

Sample run

An example of the input/output for a game being played is shown on the next page. **To receive full credit for your program, your input/output format MUST EXACTLY MATCH the format in the example including all spaces and newlines.**

```
ROW A: ooo
ROW B: ooooo
ROW C: oooooooo
Player 1, choose a row and number of rocks: B2

ROW A: ooo
ROW B: ooo
ROW C: oooooooo
Player 2, choose a row and number of rocks: A1

ROW A: oo
ROW B: ooo
ROW C: oooooooo
Player 1, choose a row and number of rocks: C6

ROW A: oo
ROW B: ooo
ROW C: oo
Player 2, choose a row and number of rocks: G1
Invalid move. Try again.
Player 2, choose a row and number of rocks: B3

ROW A: oo
ROW B:
ROW C: oo
Player 1, choose a row and number of rocks: A3
Invalid move. Try again.
Player 1, choose a row and number of rocks: C2

ROW A: oo
ROW B:
ROW C:
Player 2, choose a row and number of rocks: A1

ROW A: o
ROW B:
ROW C:
Player 1, choose a row and number of rocks: A*
Invalid move. Try again.
Player 1, choose a row and number of rocks: &4
Invalid move. Try again.
Player 1, choose a row and number of rocks: A1

Player 2 Wins.
----- Halting the processor -----
```

Hints and Suggestions

- Remember, all input and output functions use ASCII characters. You are responsible for making any conversions that are necessary.
- For character input from the keyboard in this assignment, you should use TRAP x20 (GETC). To echo the characters onto the screen, you should follow each TRAP x20 with a TRAP x21 (OUT). Recall that TRAP x23 displays a banner to prompt the person at the keyboard to input a character. You do not need that banner since your program has its own style of prompt. Therefore you should use TRAP x20 which does the same as TRAP x23 except it does not print a banner on the screen to prompt for input.
- You should use subroutines where appropriate.
- In each subroutine you write, you should save and restore any registers that you use. This will avoid a major headache during debugging.
- A legitimate turn must contain the row, specified as A, B, or C (i.e., capital letter) followed by a number that is not larger than the number of rocks still remaining in that row.

Submitting

Please follow the same submission instructions as Lab 0 *carefully* (the submission part starts at step 4). Make sure you are using lab4 every place that lab0 was used. One of the common mistakes we saw with lab 0 was renaming the lab0 folder to eid_lab0. You should only rename the zip file **after** it is created, and not the directory inside. Make sure you delete all files that the assembler generates (lab4.bin, lab4.obj, etc.) before submitting. Also, make sure you do not just compress the lab4.asm file, and instead compress the entire lab3 directory.