Project Instructions

Introduction

In the first four labs, you wrote small pieces of RISC-V assembly code to implement conditionals (if-statements), loops, functions, and arrays. In the project, we're asking you to use those skills together to build a game based on **Sokoban** **(https://en.wikipedia.org/wiki/Sokoban)** .

To complete the project, you will need to implement:

- The base requirements, which allow you to play the game.
- Two enhancements. You may use the ones we suggest or suggest your own.
- A user guide that describes how to run and play the game.

Your project is due *Friday, October 18 @ 5:00 p.m.* Submit both your **.s (assembly program)** **(https://q.utoronto.ca/courses/366497/assignments/1308180)** and **.pdf (user guide)** **(https://q.utoronto.ca/courses/366497/assignments/1308181)** to their respective assignments on Quercus.

Background

Sokoban **(https://en.wikipedia.org/wiki/Sokoban)** is a puzzle game. The original version is played on a grid representing a warehouse, where every square is a floor or wall. The character occupies a floor space, and boxes are scattered across the grid, also on floor spaces. Targets (also called storage locations) are scattered on the floor. Neither the character nor the boxes can enter a wall, and the character can push one box at a time. The goal is to push all of the boxes onto the targets.

We will mimic the core game-play. You will use the console to interact with a player by (a) printing the board on the console using symbols (of your choice) to represent the character, boxes, walls, and target and (b) accepting the next move to make from the player. You will also print messages, such as congratulations sent on success or an inability to make the requested move, on the console.

Starter Code

We are providing starter code that provides an example of how to store information about the game state and which provides a non-random number generator. Your first task is to read this code to see what facilities it provides:

- _*start* is not implemented. It contains a sequence of TODO instructions that will guide you through the base functionality.

- Above *_start*, there are several memory locations reserved to store the size of the board and the location of the player, box, and target. You may wish to update these declarations later but should be efficient about your use of memory. Use bytes, for example, instead of words unless you need the additional range provided by words.
- The *notrand* function provides a non-random number every time it is called that you can use to set initial locations for the player, box, and target.

Base Requirements

Successfully implementing the base requirements will give you a score of 6 / 10. The base requirements are intended to allow someone to play the game. Your program must:

- Generate a pseudorandom location for the character, 1 box, and 1 target. The box and target must be placed in such a way that the game is solvable. You may choose the algorithm you use to generate a pseudorandom number, but you must use an existing algorithm. Please do not create your own. Further, we require that you *cite the algorithm* by leaving a comment in the source code. (See, as an example, the **ACM style guide's directions for web resources (https://www.acm.org/publications/authors/reference-formatting)** .) Your citation should include at least the name of the algorithm and its creator(s) and the name of the publication or link to the URL which describes the algorithm.
- Print a gameboard on the console that correctly represents the current locations of the character, box, target, and walls. The gameboard size is defined by variables in the program, and you should not assume that it will always be 8x8. You should document (more on this later!) how to interpret the symbols you have chosen to represent the gameboard in the console.
- Read input from the console and move the character and box appropriately based on that input. That means that the character, box, and walls must behave according to normal sokoban rules. You should document how input should be provide by the user.
- Restart (by either providing a new randomly generated grid or resetting to the original) if the user requests it via the console. You may choose -- and should document -- how the user signals that they want a restart.
- Inform the user on the console when the box has successfully been placed on the target.

Your program must be written in RISC-V assembly and will be submitted to Quercus as a .s file. It should not be written in a higher-level language and compiled to assembly, and you should review the policy on AI use in the course infosheet.

Enhancements

To obtain full marks on the project, you must implement any two enhancements. Each successfully implemented enhancement will add up to 2 points to your score, to a maximum of 10 / 10. You may receive less than the full 2 points for a partially completed enhancement.

Important: At the top of your sourcecode file, add a comment (a) stating precisely which enhancements you are implementing, (b) where we can find them (line number or label name), and (c) how they are implemented. This will help the TAs find your enhancements and to gauge whether they are well designed. If they are unable to find your enhancement, it'll be marked as not being present.

The ideas below are our suggestions. If you'd like to implement a different enhancement, please post your suggested enhancement **to this thread on piazza (https://piazza.com/class/lz8qzgs7pjy84/post/92)** . Do so at least one week prior to the deadline. We'll respond with "Yes, here are some formal requirements ..." or "No, sorry." If a feature has been approved, you may use it, whether or not you are the one who proposed it.

Potential enhancements

- Increase difficulty by increasing the number of boxes and targets.
  - This requires a change to how the boxes and targets are stored.
  - The number of boxes and targets should be bounded only by the size of the grid. That is: you may not assume there is a hard maximum of, say, 64, because someone could update the grid size.
- Provide a multi-player (competitive) mode.
  - For full credit, the program should prompt for the number of player prior to starting the game.
  - Each player should be given a chance to play the same puzzle each round.
  - The program should track how many moves each player needs to solve the puzzle and should display the cumulative standings (ordered by number of moves required) as a single leaderboard after the entire round is complete.
- Generate internal walls in addition to the character, box, and target. To get full credit for this enhancement, at least 1/4 of the internal gameboard must contain a wall. Those walls must be randomly generated but the game must remain solvable.
  - Note: This enhancement is far more challenging than the other two suggestions!

User Guide

This project allows you to make important decisions regarding the user interface and how to interact with the program. For example:

- How will the game represent walls or the character?
- How can the user change the gameboard size?
- How will the game indicate that the character cannot move in the direction indicated by the user?
- How does the user indicate that they want to restart?
- How will the game celebrate the player successfully solving the puzzle?

Your enhancements also need to be described. For example:

- If multi-player mode is implemented, how will players know that it is their turn?

- If internal walls are being built, how many walls will be placed?

Finally, we provided some instructions for running your starter code -- but your users will not be reading this assignment handout and will not be familiar with cpulator. You must provide instructions to help them load and run the game.

This guide needs to be written with a *user* in mind. The user wants to play the game, not understand how your code is written. The user will not be familiar with the simulator. And the user wants to quickly find the information they need, if there is a problem.

The user guide should be submitted as a PDF.

To get you started, here are some tips for writing **clear rules** **(https://www.fairway3games.com/writing-rules-a-recipe/)** . And here are some examples of game manuals: **Oregon Trail** **(https://oldgamesdownload.com/wp-content/uploads/The_Oregon_Trail_AppleII_Manual_EN.pdf)** , **Zork** **(http://mirrors.ibiblio.org/interactive-fiction/infocom/demos/ztuu.pdf)** , and **Lode Runner** **(https://www.mocagh.org/broderbund/loderunner-manual.pdf)** .

Grading

The Program

The assembly project is graded out of 10 marks. First, we evaluate the base requirements:

6/6: All base features are implemented correctly, and the algorithm used to implement randomness was cited.

5/6: All base features are implemented correctly. Algorithm chosen to generate random number was not cited.

4/6: Most base features are implemented correctly. There was a small error: <explanation>

3/6: The program can print to the console but does not implement sokoban movement.

0/6: No submission or unable to see any behaviour in console.

And then we evaluate two enhancements at 2 marks each. If the base requirements are not met, we may be unable to evaluate enhancements:

2/2: Enhancement works as expected.

1/2: Enhancement provides partial functionality.

0/2: Enhancement not implemented.

User Guide

In addition to submitting your code (a .s file), you must submit a PDF that contains the *user* *guide* for your game. The guide will be marked out of 6 marks. We will be focusing on three qualities of your writing, each worth 2 marks.

Since you may do well in some aspects of one quality and less well than others, your grade may fall between levels. For example, consider the "Structure and Organization" quality. Your guide may be well organized into sections (2/2), and the text may be organized appropriately into paragraphs and lists (2/2). However, your visual aids may not be referenced in the text at all (0/2). The grader may, depending on the importance of the various criteria in your particular writing, assign a grade of 1.5, 1.0, or even 0.5.

Structure and Organization

2/2: All important sections are included. The guide is organized into appropriate sections presented in a logical order (e.g., "How to Start" is before "Gameplay".) Ideas are introduced in the correct order (i.e., without referencing ideas defined later). Visual aids are placed near the text that references them and should be captioned. The text is organized into paragraphs or lists, as appropriate.

1/2: The guide is mostly organized into appropriate, logically ordered sections, but some sections are missing, could be reordered, or could be split. Some ideas are introduced too early. Some visual aids are placed far from where they are referenced or are not referenced or captioned. The text sometimes uses a list where a paragraph would be more appropriate.

0/2: The guide is not logically organized and should be split into sections. Ideas are frequently introduced in ways that require the user to jump back and forth through the document. Visual aids are not placed near relevant text, are not referenced or captioned, or aren't present when they would be useful. The text frequently uses lists instead of organizing the text into paragraphs.

Writing Mechanics

2/2: Sentences are concise (typically < 35 words), clear (easy to understand), and complete (containing a subject, verb, and appropriate phrases or clauses). Pronoun antecedents are clear. Verbs used are precise (e.g., "Click on the `Compile and Load' button" instead "Use the Compile and Load button").

1/2: Some sentences are overly long, complex, or incomplete. Pronoun antecedents are frequently unclear. Some verbs are imprecise.

0/2: Sentences are long, complex, and/or incomplete. Pronoun antecedents are unclear. Verbs are imprecise.

Understanding Audience Expectations

2/2: Terms are defined when first used. Only essential information is presented. All of the information a user needs is presented. Section headings are used to make relevant information easy to find. Visual aids are used are appropriate for the user and not merely decorative. Gender-neutral language is used consistently.

1/2: Some terms are not defined. Some inessential information is presented. Some information a user needs is missing. Some section headings are unclear or do not accurately present the material in the section. Some visual aids are not appropriate for the user. Some gendered language is used.

0/2: Terms are introduced without definition. Essential information is difficult to find or missing. Section headings do not help identify where to find relevant information. Visual aids are not appropriate for the user or are entirely decorative. Gendered language is used.