COMP 273, Winter 2022, Assignment 4

School of Computer Science
McGill University
Available On: Mars 25th, 2022
Due Date: April 11th, 2022, 23:59
(Late policy: 10 % off per day late, up to 2 days)

Before You Start

- Discussion is welcomed but do not share your codes!
- Start the assignment earlier as this gives you more time to write and debug your programs, and the TAs will have more time to help you.
- Make sure you understand the class content before you actually start writing the programs. This will save your time!
- Read the instructions (including the submission instructions at the end, and the instructions in the provided source files) carefully. For your submission, don't modify the existing code unless you are asked to. We are using auto-graders so follow the input/output format described in this handout. Failing to follow the instruction might result in a penalty.
- Follow the calling convention you learned from the class. E.g., how to pass arguments to subroutines, how to return values from them, what registers should be saved and restored, etc. This should not be a problem if you fully understand what you learned from the class.
- Learn how to debug because it's an essential skill for your future career if you want to be a good programmer or computer scientist. This includes knowing the debugging tools well, and having a good strategy for debugging, such as where to set the breakpoints, and what registers/variables should be carefully monitored.

1 Tower of Hanoi (25 marks)

The tower of Hanoi is a puzzle invented by French mathematician Édouard Lucas. The puzzle consists of three rods: A, B and C, and n disks (labeled with numbers $1..n$) with different radii stacked on pod A. The disks are arranged from top to bottom in the order of decreasing radius. Figure 1 shows an example with three disks. You can move the disk between rods, but smaller disks have to be on top of bigger disks, and you can only move one disk at one time. The goal is to move all the disks from rod A to rod C with the minimal number of moves.

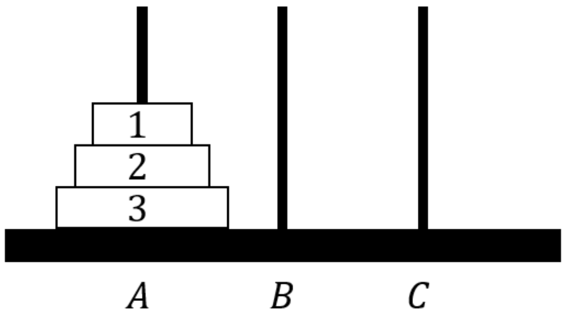


Figure 1: An example of the tower of Hanoi with $n = 3$

Alice is trying to solve this puzzle (with 3 disks). She cannot solve it directly, so she asks Bob to move disk 1 and 2 from rod A to B, then she can simply move disk 3 from A to C, and ask Bob again to move two disks from B to C, and problem solved! These two tasks are still a bit difficult for Bob, so for the first task, he asks Charlie to move the smallest disk from A to C so that he can move the second disk from A to B, and then ask charlie to move the smallest disk from C to B. Similar for the second task, he asks Charlie to move the smallest disk from B to A before he moves the second disk from B to C, and asks Charlie to move the smallest disk from A to C. All the tasks for Charlie are simple enough so he doesn't need help from others. By asking others to solve the sub-problems, Alice successfully solves the puzzle:

1. Charlie moves disk 1 from A to C;
2. Bob moves disk 2 from A to B;
3. Charlie moves disk 1 from C to B;
4. Alice moves disk 3 from A to C;
5. Charlies moves disk 1 from B to A;
6. Bob moves disk 2 from B to C
7. Charlies moves disk 1 from A to C.

This idea can be formalized using an recursive algorithm shown in Algorithm 1, which gives the solution to move n disks from the *source* rod to the *target* rod, with the help of the *auxiliary* rod.

Implement the recursive algorithm described above and complete `hanoi.asm`. The program should read an integer n (already implemented), i.e., the number of disks. The output should be the steps to solve the problem, one step per line, printed to the standard output. Each line should have the format of *Step i: move disk disk_label from rod_label to rod_label.* Here is an example of the output for the problem with $n = 3$:

```
Step 1: move disk 1 from A to C
Step 2: move disk 2 from A to B
Step 3: move disk 1 from C to B
```

Algorithm 1 Recursive algorithm for the tower of Hanoi

```
procedure MOVE($n$, $source$, $target$, $auxiliary$) :
    if n=1 then
        move disk 1 from $source$ to $target$
    else
        MOVE($n - 1$, $source$, $auxiliary$, $target$)
        move disk $n$ from $source$ to $target$
        MOVE($n - 1$, $auxiliary$, $target$, $source$)
    end if
end procedure
```

```
Step 4: move disk 3 from A to C
Step 5: move disk 1 from B to A
Step 6: move disk 2 from B to C
Step 7: move disk 1 from A to C
```

You can assume that the input n is a valid integer and $1 \leq n \leq 15$. Make sure you strictly follow the output format!

2 Game of Life (25 marks)

The game of life is a two-dimensional cellular automaton invented by British mathematician John H. Conway. The game simulates the evolution of a grid of cells. Each cell has two possible states: live or dead. The state of the cell in the next generation (let's say the $(i+1)$th generation) is decided by its current state and the current states of its eight surrounding cells (the ith generation):

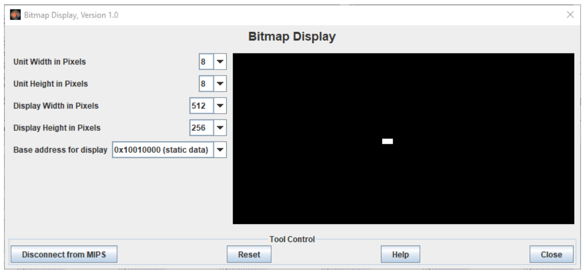
- A live cell will die if it has fewer than two live neighbours or more than three live neighbours ;
- A live cell will stay alive if it has two or three live neighbours;
- a dead cell will come to life if it has exactly three live neighbours.

Try the simulation tool at https://playgameoflife.com/ to gain further understanding of the game. Complete `life.asm` to simulate the game of life. The program should read the initial states of the cells (i.e., the cells at the 0th generation) from `life-input.txt`, read an integer n from the standard input (already implemented), simulate the evolution for n generations, and write the states of the cells at the nth generation in `life-output.txt`. The input file has multiple lines with the same lengths, each line is a string consists of characters "0"s and "1" representing the states of cells in the row ("0" means a dead cell, "1" means a live cell) ended with a newline character ("\n"). There are not any space or other delimiters in the string. The output file should have the same format and the same number of rows and columns as the input file. You can assume that $0 \leq n \leq 50$ and the numbers of rows and columns are no more than 100. Check `life-sample-input.txt` for the sample input and `life-sample-output-1.txt` for the corresponding sample output with $n = 1$, and `life-sample-output-2.txt` for $n = 2$. Be careful when dealing with the cells on the boundary. You can assume that there are no cells outside the boundary, or equivalently only dead cells outside the boundary.

3 Snake

Let's write something fun with all the things we have learned! The game *Snake* should be a good start if you try to write a video game. In the file `snake.asm`, we provide most of the code, but you need to fill in some missing parts to make it a real game.

Where is the keyboard and monitor? You need to go to the menu "Tools" →"Keyboard and Display MMIO Simulator", and click "Connect to MIPS" to have your "keyboard". For the display, you need to go to the menu "Tools" →"Bitmap Display", change both unit height and width to "8", and click "Connect to MIPS". The tool will display a buffer (*displayBuffer* in the code) in the data segment (by interpreting the buffer as an array of RGB colors), and this is your "monitor". Now, assemble and run the code. You should see a moving "snake" in display, similar to Figure 2.



Unit Width in Pixels	8
Unit Height in Pixels	8
Display Width in Pixels	512
Display Height in Pixels	256
Base address for display	0x10010000 (static data)

Figure 2: What you should see before writing anything.

Before we start writing the code, let's have a look at some important things already in the source file:

- *BLACK*, *RED*, *GREEN*...: these are constants that you can write into *displayBuffer*.
- *EMPTY*, *SNAKE*, *FOOD*...: things in the world. They are represented by different colors, e.g., the empty space is black (the background color). These can also be written into *displayBuffer*.
- *DIR_RIGHT*, *DIR_DOWN*, *DIR_LEFT*, *DIR_UP*: direction constants. The value of the *direction* variable must be one of these.
- *STATE_NORMAL*, *STATE_PAUSE*...: game state constants. The value of the *state* variable must be one of these.
- *MAX_NUM_PILLS*: the maximum number of pills (total numbers of the red pills and blue pills) in the scene.
- *displayBuffer*: an array (32x64) of RGB values (in words) that will be displayed in the "Bitmap Display" tool. A function named *pos2offset* has been provided to compute the offset of a pixel in this buffer given its coordinates (x, y). In our coordinate system, the origin is at the top left pixel, the x-axis points to the right and the y-axis points to the bottom. To draw a pixel, you can first compute its offset using *pos2offset*, and change the value of the word at that offset in *displayBuffer*, e.g., change the value to *FOOD* to draw a food, change it to *RED_PILL* to draw a red pill.

- *snakeSegment*: an array that stores the offsets of the snake segments in the display buffer. See the source file for details.
- *snakeLength*: the current length of the snake.
- *headX*, *headY*: the position of the snake head.
- *numPills*: the total numbers of the red pills and blue pills in the scene.
- *direction*: moving direction of the snake head. See the source file for details.
- *state*: game state. See the source file for details.
- *score*: current score. Increased by one every time the snake eats a regular food.
- *timeInterval*: time interval for controlling the game speed. This is implemented by sleeping for *timeInterval* milliseconds in the game loop.

Now let's complete the program step by step!

3.1 Handle Keyboard Input using MMIO (20 marks)

In the *main* process, the code handling keyboard input is missing. Write some code to handle the keyboard input using MMIO and do the following:

- Change the moving direction of the snake head when the inputs are "w" (up), "a" (left), "s" (down) or "d" (right). This can be done by modifying the *direction* variable. Note that the snake cannot go to the reverse direction, e.g., if the current direction is right, you cannot change the direction to left.
- If the inputs are "q" (quit/exit), "r" (restart), or "p" (pause/resume), change the state of the game. This can be done by changing the *state* variable. Note that pressing "p" should pause the game if the game is running, and resume the game if it is paused.

Note that you don't want to wait until the device is ready to read the data, which is different from the polling method you have seen in the class, because this will "block" the main process of the game and the game will be not responding until you press a key. So for the keyboard handling of this program, you query whether the input (keyboard) is ready, and if the input is ready, read the input; otherwise, continue with the normal execution of the program without reading the input.

After you finish this step, you should be able to control the movement of the snake and the game execution using the keyboard.

3.2 Spawn Pills (15 marks)

You might already notice that there are no red and blue pills spawned. This is because we are missing the code for spawning the pills in the subroutine *spawnFood*. Add some code to do the following:

- Spawn the red pills with 20% chance. Remember to update *numPills*.
- Spawn the blue pills with 15% chance. Remember to update *numPills*.

A subroutine called *randInt* is already provided to you for generating a random integer. Note that it should not spawn any pill if there are already *MAX_NUM_PILLS* pills in the current scene. And you can't spawn a pill at the position where there is already something else (i.e., you can only spawn a pill at a empty place). Check the code for spawning regular food if you don't have any idea. Finished? Now you can try some pills! Eating the red pill increases the game speed by 20%, and eating the blue pill decreases the game speed by 20%.

3.3 Add Walls (15 marks)

Playing in an empty space is boring. You can see at the beginning of the code, there is definition of *WALL* tile type, but there is no code adding the walls. Create a map file to specify the location of the wall tiles. Write code in *initMap* to load the file you create and add the walls to the scene by setting pixel values to *WALL* in *displayBuffer* (Figure 3). The layout of the walls you add should have enough complexity (e.g., not just a straight line). Be creative! You are free to have your own way to describe the wall layout in your map file. You also need to add a README file to explain how to modify the file to change to wall layout so we know how to modify it. Of course, you need to upload your README file and the map file.

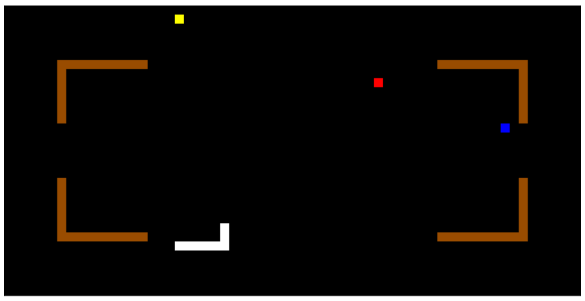


Figure 3: A world with food, pills and walls

Submission Instructions

- Do not wait until the last minute to submit your work.
- You need to submit the ASM files, the map file, and the README file.
- Make sure you upload the correct files (and the correct version). Double-check by downloading the file you upload.
- Do not change the filenames of the ASM files.
- Do not upload your files separately. Instead, compress them into a single ZIP file and rename it with your student ID.
- Make sure your code can be assembled and we can run your code. Code that can not be assembled or throws exceptions frequently might result in very low marks.