CS 447 and COE 147 Computer Organization
Spring 2012

Programming Project 2
Assigned: March 12. Due: March 27 by 11:59 PM.

1 Description

Put on your seat belts! It's time to step into your DeLorean time machine, set the date to 1978, and accelerate to 88 MPH. Bam! It's 1978! The Bee Gees rule, the Incredible Hulk is number one, the Vax is king and satin disco jackets are de rigueur. And, best of all, it's the beginning of the arcade video game revolution. The pinball machine will never be the same!

As you practice your best dance moves under the spinning glitter ball to Stayin' Alive, you have a brilliant idea for an arcade game, called Snake-n-frogs (SNF), that will most certainly land you a job at Atari. The objective of your game is to navigate a snake through a field eating frogs as quickly as possible. Of course, as the snake eats frogs, it is nourished and grows longer, making it progressively more difficult to navigate. If the snake ever hits itself, then the game is lost. When all the frogs are gone, the game is over. When the game ends, it prints the number of frogs eaten and the elapsed time that the game was played.

Your job is simple: implement SNF in MIPS with the Mars LED display simulator[1] (used in lab, see the course web site). The game should follow the rules below.

2 Game Rules

The game board is arranged as a grid of 64 columns by 64 rows, as shown in Figure 1. A position in the grid is denoted by a coordinate (x, y). The x coordinate is the column position, such that $0 \leqslant x \leqslant 63$. The y coordinate is the row position, such that $0 \leqslant y \leqslant 63$. The game board is a grid of light emitting diodes (LEDs), which can be turned off (black), red, yellow and green. The game board includes an arrow keypad, with up, down, left, right and center keys.

2.1 Snake Behavior and Movement

The snake has the following behavior:

- The snake can travel in the directions up, down, left and right on the game board. The snake cannot travel diagonally.
- The snake is composed of a series of segments. Each segment is a single *(x,y)* coordinate. There is a head segment that begins the snake and a tail segment that ends the snake. In between the head and tail, there is a series of segments for the body. The segments are consecutive, but can "turn corners" at 90 degree angles (Figure 1).
- The snake is represented by setting the color of the LEDs at each segment to yellow.
- Every $200ms$, the snake moves one position in the current direction of movement. Thus, each segment in the snake "moves" up, down, left or right one LED, according to the direction of movement (see the hints).
- The snake's direction of movement can be changed by the player pressing the up, down, left or right arrows keys. The direction is changed according to which key was pressed. For realism, the game should respond quickly to keypresses, preferably without any noticeable delay.

[1] MIPS was actually invented in 1981, but let's suspend our disbelief and pretend it was invented in 1977.

- When the snake's current direction of movement is up or down, its direction can only be changed to left or right (i.e., ignore up and down arrow key presses when the snake is already moving up or down).
- When the snake's current direction of movement is left or right, its direction can only be changed to up or down (i.e., ignore left and right arrow key presses when the snake is already moving left or right).
- If the snake reaches the border of the game board (e.g., column 63 when moving right), it reappears on the opposite side (e.g., column 0). In effect, the snake "wraps around" the game board (see the hints). This simplifies the game logic.
- When the snake moves over a frog, the frog is eaten. The snake grows in length by one segment in the direction of movement.

2.2 Frog Behavior

A frog has the following behavior:

- A frog is a single LED, which is colored green.
- A frog is stationary and does not move (see extra credit).
- A frog's position on the display is determined randomly at game start (see game start) by picking a random (x, y) coordinate on the game board. If this coordinate already has a frog, or has the snake, the coordinate is discarded (i.e., don't put a frog *on* the snake).
- When the snake moves over a frog, the frog disappears as it was eaten.

2.3 Game Start

The initial conditions at game start are:

- The snake's length is initially 8 segments, including its head, body and tail.
- The snake starts in row 31, with its head, body and tail at the segments:
  $[head = (7, 31), body = \{(6, 31), (5, 31), (4, 31), (3, 31), (2, 31), (1, 31)\}, tail = (0, 31)]$
- The game board is initially filled with *up to* 32 randomly placed frogs. When placing frogs, if a coordinate has to be discarded, then simply go on to the next frog. Thus, the game may initially have fewer than 32 frogs (this simplification avoids trying to find 32 empty positions).

2.4 Game End

The game ends when certain conditions are satisfied, and it prints the score and total playing time:

- If the snake hits itself (i.e., the head hits the body or tail), then the game is over.
- If all frogs are eaten, then the game is over.
- When the game is over, the total time to play the game, say x and the number of frogs eaten, say y, are displayed in the Run I/O window.
- The output must match:
  `Game over.`
  `The playing time was` x `ms.`
  `The game score was` y `frogs.`
- The playing time is displayed in milliseconds to simplify the game logic (use MARS time system call).

3 Animation

To animate the game, you must move the snake. This can be accomplished by turning on/off LEDs at particular positions. For instance, suppose the snake is in a straight line and moving to the right with its head segment *(30,12)* and it's tail segment *(20,12)*. On the next animation step (i.e., after $200ms$ has elapsed), the new head segment is *(31,12)* and the new tail segment is *(21,12)*. Thus, to animate the snake, turn on the LED at coordinate *(31,12)* by setting it to yellow, and turn off the LED at coordinate *(20,12)* by setting it to black. Note that this approach changes only two LEDs: the new head position (turned on to yellow) and the old tail position (turned off to black). It is an efficient way to perform the animation very quickly to achieve the desired effect.

4 Mars LED Display Simulator

The project needs a special version of Mars available from the CS 447 and COE 147 web site. This version of Mars has an LED Display Simulator developed and extended by former graduate students in the CS Department. For off-campus access to the simulator, you'll need the user name and password announced in class and recitation.

4.1 Display

The LED Display Simulator has a grid of LEDs. An LED is a light. Each LED has a position (x, y), where $0 \leqslant x \leqslant 63$ and $0 \leqslant y \leqslant 63$. The upper left corner is $(0, 0)$ and the lower right corner is $(63, 63)$. An LED can be turned on/off by writing a 2-bit value to a memory location that corresponds to the LED. An LED can be set to one of three colors: green (11_2), yellow (10_2) or red (01_2). The LED can also be turned off (00_2). For this project, we provide two functions to manipulate LEDs:

`void _setLED(int x, int y, int color)`
Set LED at (x, y) to green (`color`= 3), yellow (`color`= 2), red (`color`= 1) or off (`color`= 2).

`int _getLED(int x, int y)`
Return color of LED at (x, y).

These functions are available from the project's web site.

4.2 Keypad

The game must use the keypad provided by the LED display. The keypad is the set of arrow keys on the left side of the LED Display Simulator. The keypad has up (▲), down (▼), left (◀) and right (▶) buttons. The keypad also has a center button ('`b`').

Each arrow and center button in the keypad has a keyboard character equivalent. When you click a keypad button or press the equivalent keyboard character, a value is stored to memory location `0xFFFF0004`. To indicate a button click, the value `1` is written to memory location `0xFFFF0000`. When your program loads this address, memory location `0xFFFF0000` is set to `0`.

Important: The program *must* read the status memory location (`0xFFFF0000`) before trying to read the key value memory location (`0xFFFF0004`). The behavior is undefined, if you read the key value memory location without first reading the status location.

The keypad button and keyboard character equivalents are:

'`w`' is ▲ button, which stores `0xE0` at memory address `0xFFFF0004`
'`s`' is ▼ button, which stores `0xE1` at memory address `0xFFFF0004`
'`a`' is ◀ button, which stores `0xE2` at memory address `0xFFFF0004`
'`d`' is ▶ button, which stores `0xE3` at memory address `0xFFFF0004`

'b' is `b` button, which stores `0x42` at memory address `0xFFFF0004`

An example program that uses the keypad is available from the project web site.

4.3 Simulator Versions

This project will need the MARS 64x64 LED Display Simulator. This version of Mars adds the center button and the keyboard shortcuts for the keypad. It is based on Mars 4.1. The simulator is available from the web site:

`http://www.cs.pitt.edu/~childers/CS0447`

A special note for Mac OS users and JVM versions: We built this version of Mars with JVM 1.5. It should run on older Macs (Mac OS 10.5.x and older) and other machines with JVM 1.5 installed. You may wish to upgrade instead to JVM 1.6.

5 Turning in the Project

5.1 What to Submit

You must submit a compressed file (`.zip`) containing:

- `snake.asm` (the Snake-n-frogs game)
- `README.txt` (a help file - see next paragraph)

Put your name and e-mail address in both files at the top. Use `README.txt` to explain the algorithm you implemented for your programming assignment. Your explanation should be detailed enough that the teaching assistant can understand your approach without reading your source code.

If you have known problems/issues (bugs!) with your code (e.g., doesn't animate correctly, odd behavior, etc.), then you should clearly specify the problems in this file.

The explanation and bug list are critical to grading. So, if you're uncertain whether to include something, then err on the side of writing too much and just include it.

Your assembly language program must be properly documented and formatted. Use enough comments to explain your algorithm, implementation decisions and anything else necessary to make your code easily understandable.

The filename of your submission (the compressed file) should have the format:

`<your username>-pa02.zip`
Here's an example: `musfiq-pa02.zip`

Files submitted after March 27, 11:59 PM will not be graded. **It is strongly suggested that you submit your file well before the deadline.** That is, if you have a problem during submission at the last minute, it is very unlikely that anyone will respond to e-mail and help with the submission before the deadline.

5.2 Where to Submit

Follow the directions on your TA's web site for submission:

- Musfiq Rahman: `http://www.cs.pitt.edu/~musfiq/teaching/cs0447/index.htm`
- John Wenskovitch: `http://www.cs.pitt.edu/~jwenskovitch/cs0447.html`
- Jie Gu: `http://www.pitt.edu/~jig26/TA/index.htm`

6 Collaboration

In accordance with the policy on individual work for CS/CoE 447, this project is strictly an individual effort. Unlike labs, you must not collaborate with a partner. Please see the course web site (syllabus) for more information.

7 Extra Credit

This project has three extra credit options. If you attempt the extra credit, then you should attempt it in the order listed. To get extra credit for a step, the previous step(s) must work.

7.1 Walls

The classic snake game has "walls" that stop the snake (i.e., the game ends). In our version, let's do something a little different: When the snake hits a wall, its direction of movement is changed by 90 degrees relative to the current direction. For instance, if the snake is moving right and it hits a wall, then its new direction is changed to either up or down.

A wall is denoted by a red LED. Thus, if the head hits a red LED, then change the snake's direction. During game start, select random LEDs to turn red (this can be done at the same time frogs are placed).

There are a couple situations to be careful about. First, when the snake hits a wall, the wall should *not* change color. That is, the wall LED should remain red. It's easy to check for a wall by "validating" a move prior to making it: check that the snake's new head segment (coordinate) won't fall on a wall. You can use the `_getLED` function to read the color at the new coordinate. If there's no wall, the snake can move in the current direction. If a wall is encountered, change the direction. Second, it's possible that a snake can get stuck when several adjacent LEDs are red (e.g., in a U-shape). The game should terminate if the snake gets stuck (this is easy to do: try moving in one direction first; if this doesn't work, move in the opposite direction; if neither more works, then the snake is stuck).

7.2 Moving Frogs

In real life, frogs are smart. If they see a snake coming toward 'em, they'll move! So, we can approximate this behavior in SNF in the following way. Every so often (you pick how often), select a random frog to move. Simply move the frog by one LED in the directions left, right, up or down. To make this even more fun, allow the frog to move diagonally one LED as well. Be sure to "validate" the frog move before making it. The frog should not hop onto a wall, another frog, or worst of all, the snake.

A simple approach is the following. Select a random frog. Determine a random direction to move (include the diagonals!). Compute a new (x, y) coordinate for the frog. Check whether the coordinate contains any color other than black. If so, discard the frog move. Otherwise, make the move, setting the old LED position to black and the new LED position to green.

Think carefully about your strategy to move the frogs. If you pick poorly, it could harm the quality of the animation.

7.3 Game Boards

This option adds game boards that contain walls. There should be at least three game boards. The walls are determined ahead of time (i.e., by you) to create interesting obstacles and traps for the snake. A wall can be created from a series of red LEDs.

As each game board is won, move on to the next board. Re-initialize the snake and the frogs as described in Section 2.3 for each new board. The game over condition is changed to include ending the game when all boards are tried. The score is the total frogs eaten and the elapsed time for the

entire game. In essence, this option creates "game levels" that become progressively more difficult. To make this easier on the TA, use '`b`' as a short-cut to go to the next board. When '`b`' is pressed, the next board is tried. The player (TA!) should be allowed to press '`b`' at any time during play.

When you make the boards, be sure that the boards can be "won". For example, what happens when a box has a frog in it? (For your own amusement, try this: place the snake in a maze. This will cause it to move through the maze, assuming you sufficiently constrain the snake and implemented the wall behavior correctly.)

Hint: store the boards in memory to initialize the LED display with a data structure that indicates which LEDs should be turned to red.

8 Programming Hints

Tackle the project in small steps, rather than trying to do it all at once. Test each step carefully before moving on to the next one. Here are some recommended steps:

1. Think! Plan! Think some more! Write pseudo-code for your game strategy.
2. Identify a useful data structure for the snake (see below).
3. Implement the data structure and *thoroughly* test it before trying the animation.
4. Develop a function that moves the snake one position in a specified direction.
5. Develop this function piecewise. Start with a single direction, say left to right.
6. Does the snake wrap around when it hits the right side?
7. Add the three other directions. Does the snake wrap around?
8. Write and test a function to check for a keypress and to return the key, if pressed.
9. Inside a loop, call the keypress function. Does it detect a sequence of keypresses?
10. Inside this same loop, add code to detect when $200ms$ has elapsed. Did it work?
11. Now, whenever $200ms$ has elapsed, move the snake in the current direction.
12. Connect the keypresses to the snake's animation and current direction. Did you test that illegal keypresses are not permitted and the snake's direction changes correctly?
13. OK, you're close now. Write a routine to pick a random (x, y) coordinate.
14. Write a loop that randomly places frogs on the display by turning LEDs to green (use your random coordinate function, of course).
15. Try moving the snake over a green LED. It should move over it. Did you update the score?
16. Add support to handle the game over conditions.
17. Display the game over messages.
18. You're done! Try the extra credit (it's fun, really).

The critical aspect of this project is to use a good data structure for the snake. A "first-in, first-out queue" will work well (do you remember this from CS 445? See `http://en.wikipedia.org/wiki/FIFO_computing`). The queue is used to keep the coordinates for the snake's head, body and tail segments. The head is at the end of the queue and the tail is at the head of the queue. To manipulate the queue, you'll need three functions:

`_insert_q(x, y)`: Insert coordinate (x, y) at the end of the queue.
`_remove_q`: Removes and returns coordinate (x, y) at the head of the queue.
`_peek_end_q`: Returns (without removing) coordinate (x, y) at the end of the queue.

Here's how to use the queue. Every time the snake is moved, peek at the snake's current head segment (coordinate). Compute a new target segment (coordinate) for the head, according to the

direction and wrap-around. Insert this new coordinate into the queue. Get the color of the LED at the new coordinate. Turn the LED at the new position to yellow. Wrap-around can be easily handled by masking (use `andi` with an appropriate value to clear the upper bits).

If the previous color at the new coordinate is not green, then remove the tail segment from the queue (it is the head entry). Turn the LED for this segment (coordinate) to black. If the previous color at the new coordinate is green, then the snake just ate a frog. Add one to the game score. Do not remove the tail from the queue (thus, the snake grows by one segment). Check whether the snake ate all the frogs, and handle the game over condition, if so.

A "circular buffer" for the queue is probably the best implementation (see `http://en.wikipedia.org/wiki/Circular_buffer`). Be sure to size the buffer large enough to let the snake grow to its maximum length (in fact, you may wish to have buffer size that's a power of two to simplify address computations in the queue functions).

Lastly, note that the queue code is relatively simple. Each function is probably a dozen or so assembly language instructions with a circular buffer implementation. You might want to eschew error checking, such as overflowing the queue.

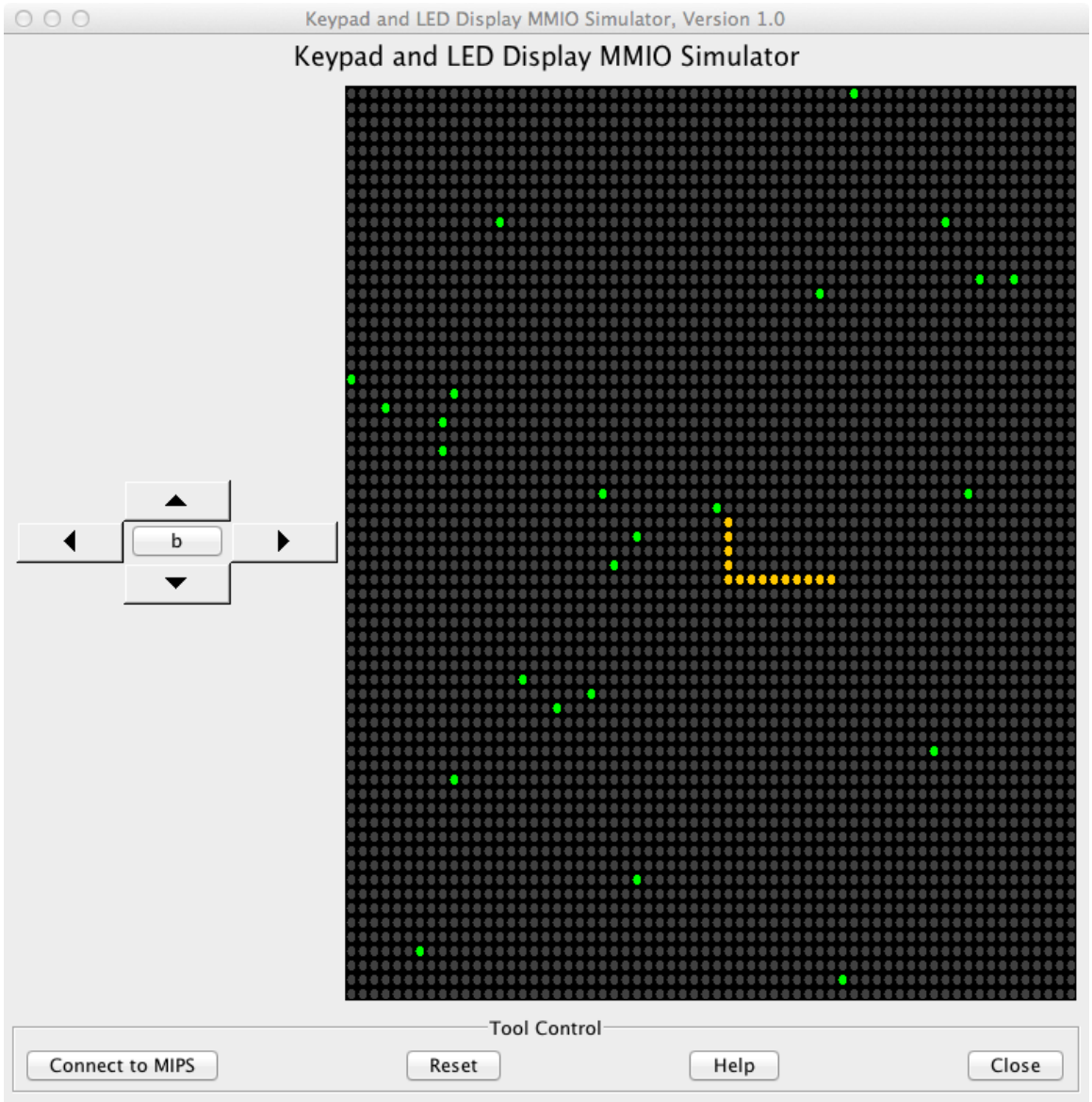

Figure 1: Example LED display with snake (yellow) and frogs (green) after playing for a while. The snake was initially moving down, and then turned right. It is continuing to the right.