CS/CoE 447 Computer Organization
Spring 2017

Programming Project
Assigned: February 13. Due: March 15 by 11:59 PM.

1 Description

As you settle into your DeLorean Time Machine, fasten your seat belt tightly. Set the way back date to October 25, 1978. Stomp on the accelerator until the speedo hits 88 MPH. Bam! Wham! It's 1978! Disco lives! Glitter is everywhere and the Bee Gees rule the world! The Incredible Hulk is number one and the Vax is king!

As you practice your best dance moves under the spinning glitter ball to Stayin' Alive, you have a smashing idea for an arcade game, called "Zombie", that will most certainly land you a job at Atari. You will impress Nolan Bushnell! The game is a maze, and the objective is to navigate a player from one corner of the maze to the opposite diagonal corner. Now, this isn't just *any* maze! The maze is haunted by zombies, which are trying to capture the player! These creatures are dangerous!! Avoid them! If the player is captured, the game is over and the world is doomed. Can you successfully make it through the maze?

The game is played on a 64x64 grid (i.e., rows and columns). A maze is imposed on this grid. The player starts in the upper left corner at position (0,0). The player can be navigated through the maze with arrow keys. The player must reach position (63,63). The zombies move around in the maze, trying to capture the player. The grid is divided into four quadrants, and each quadrant is haunted by one zombie. Because zombies tend to be a bit territorial, a zombie cannot move outside of its quadrant. Zombies move through the maze and obey walls like the player. The zombie in the same quadrant as the player is *attracted* by the player and *moves in the direction* of the player. The zombies in the other quadrants move *randomly* around their quadrants. If a zombie encounters the player, the player is captured and the game ends. If the player reaches position (63,63) without being captured, then the game is won. When the game ends, the total number of moves made the player is reported as the score.

For this project, you will implement Zombie in MIPS with the MARS LED display simulator (used in lab, see Courseweb for your recitation), according to the game 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). 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 upper left corner of the game board is coordinate $(0, 0)$ and the lower right is $(63, 63)$. The game board is a grid of light emitting diodes (LEDs), which can be turned off (black), red, orange or green. The game board includes an arrow keypad with up (▲), down (▼), left (◀), right (▶) and center (b) keys.

2.1 Zombie Behavior

The zombies have the following behavior:

- There are four quadrants of the game board grid (see Figure 1(b)). Each quadrant is 32x32. There are upper-left, upper-right, lower-left and lower-right quadrants.
- There is one zombie per quadrant.

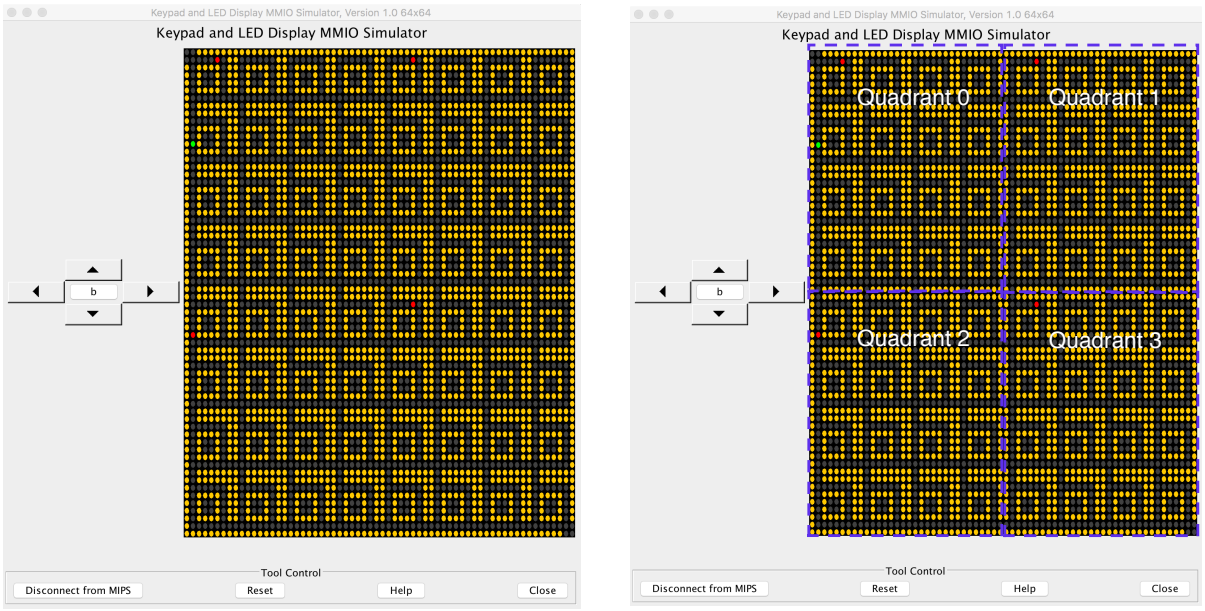

Figure 1: (a) maze with player and zombies. (b) grid quadrants.

- When the player is not in a zombie's quadrant, the zombie randomly walks through the maze in his/her quadrant.
- When the player is in a zombie's quadrant, the zombie moves toward the player. While the movement toward the payer may be random, it should be apparent the zombie is moving toward the player. For example, a player at a smaller x-position than the zombie might cause the zombie to move to the left (from the zombie's current (x, y) position to $(x-1, y)$ position).
- A zombie cannot move through a wall.
- By their very nature, zombies are curious creatures, so they search out new directions, preferably without backtracking (i.e., *reversing* direction). Therefore, when a zombie moves, it should select a direction at random, excluding the reverse direction. If the direction doesn't have a wall, the move is valid. A zombie can backtrack only when no other move is valid.
- Each zombie moves one position (LED) every $500ms$.
- Zombies are red (an LED for a zombie is set to the red color).
- When a zombie moves onto the player, the game is over, and the player is captured.

2.2 Player Behavior and Movement

- The player is drawn as a single LED set to green.
- The player is moved by pressing the left (◂), right (▸), up (▴) or down (▾) arrow keys. The player moves in the direction of the key pressed.
- The player must obey walls of the maze. Thus, if a player is moved in a direction that is occupied by a wall, the keypress is ignored.
- The player starts in position (0,0) in the upper-left quadrant of the grid.
- The game is won when the player reaches position (63,63) in the lower-right quadrant.

- The game is lost if the player is captured by a zombie. Note, moving the player *onto* an zombie causes the player to be captured! Likewise, an zombie moving onto the player cause the player to be captured. The point of the game is to *avoid* the zombies!!

2.3 Game Start, Completion and Scoring

- When invoked, the program first draws the maze. The maze will be given to you.
- Next, the zombies are placed on the board at any non-wall coordinate in his/her quadrant.
- The player is placed in position (0,0).
- The game itself and animation starts when the player presses the center (`b`) key.
- If the player is captured, the game should print "Sorry. You were captured." to the console. The program should then terminate. If the player reaches position (63,63), the game should print "Success! You won! Your score is *xyz* moves." *xyz* is the number of valid moves that the player made to get from (0,0) to (63,63).

3 Mars LED Display Simulator

The project needs a special version of Mars available from CourseWeb. This version of Mars has an LED Display Simulator developed and extended by former graduate students in the CS Department.

3.1 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). Details about the LED Display are available from the LED Display Simulator manual (see your TA's CourseWeb site).

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`= 0).

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

These functions are available from CourseWeb.

3.2 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. See the LED Display Simulator manual for a few more details.

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

Note: There have been reports that Mars locks up if the mapped keyboard keys are pressed too quickly.

Details about the keypad and how to use it are described in the LED Display Simulator manual. This document is available from CourseWeb. Additionally, an example program that uses the keypad is available from CourseWeb.

3.3 Simulator Versions

This project will need the Mars LED Display Simulator. This version of Mars adds the center button and the keyboard shortcuts for the keypad. The simulator is available from CourseWeb.

A special note for Mac OS users and JVM versions: You should use JVM 1.6 or later. We have had some trouble with the keyboard shortcuts for the keypad on Mac OS 10.5.8, which appears to be related to issues with Mars on Mac OS. If you encounter this problem, where the arrows work but the keyboard shortcuts don't work, then try a different computer (e.g., a Windows machine) to see whether the problem persists. Please let your instructor or TA know if you encounter any other problems.

4 Turning in the Project

4.1 What to Submit

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

- `<your pitt username>-zombies.asm` (the Zombie game)
- `<your pitt username>-README.txt` (a help file – see next paragraph)

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

> `<your username>-project1.zip`
> Here's an example: `xyz30-project1.zip`, which contains `xyz30-zombies.asm` and `xyz30-README.txt`.

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 reasonably documented and formatted. Use enough comments to explain your algorithm, implementation decisions and anything else necessary to make your code easily understandable.

Deadline: Files submitted after March 15, 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.

4.2 Where to Submit

Follow the directions from your TA for submission via CourseWeb.

5 Collaboration

In accordance with the policy on individual work for CS/CoE 0447, 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.

6 Programming Hints

6.1 How to get started?

Tackle the project in small steps, rather than trying to do it all at once. Test each step carefully before starting the next one. Here are some recommended steps; you may find it convenient to do the steps in a different order, but this should help you get started.

1. Think! Plan! Think some more! Write pseudo-code for your game strategy.
2. Think about what information you need to keep. For example, you will probably need to know the player's (x, y) position. You may wish to keep this information in two registers. As another example, you will need to know each zombie's (x, y) position as well. You may wish to keep this information in an array, where each array entry holds a zombie's (x, y). Other information will be useful as well, such as a zombie's current direction of movement and the minimum (x_{min}, y_{min}) and the maximum (x_{max}, y_{max}) of the zombie's quadrant.
3. Implement the game in small steps. You may want to start by drawing the maze.
4. Next, you may want to try implementing the player. Save the zombies for later.
5. To implement the player, you can use a "polling loop" (see pseudo-code below), which continuously reads the keypress status memory location. When this location has a 1, a key was pressed. Read the memory location that has the value of the key. Based on the value, call a function (or jump to some code) to handle the pressed key. For instance, you might want a `pressed_left_arrow()` function, which is called when the left arrow is pressed.
6. When a key is pressed, you need to update the (x, y) position of the player. For instance, suppose the right arrow is pressed. The next possible position for the player is $(x + 1, y)$. Before moving the player, read the LED value at $(x+1, y)$. If the LED value is "off" (0), then the move is valid. If the LED value is "red", then the player moved onto a zombie and the game is lost. If the LED value is "orange", then the move is invalid and should be discarded (the player is trying to move onto a wall). Test this navigation carefully. Can you move the player through the maze from the upper-left corner to the lower-right corner???
7. Be careful: When the player is moved from the current (x, y) position to a new (x', y') position, the LED at (x, y) should be turned off (0) and the LED at (x', y') should be set to green.
8. Next, animate the zombies! This is surprisingly simple! In the polling loop, check how much time has elapsed since the zombies were last moved. If the elapsed time is more than $500ms$ then move each zombie.
9. To move a zombie, write a function to select a potential new coodinate for the alien. Implement logic to randomly pick a direction (left, right, up, down). Given this direction, compute a new (x', y') using the zombie's current (x, y). Check that the move is valid. If valid, then move the zombie. Be sure to handle the case where the zombie captures the player.
10. When you write the function to move a zombie, you need to make sure the zombie remains in its quadrant. This can be done by checking that the new (x', y') position is within the boundary of the quadrant. If the new position falls outside of the quadrant, then the move is invalid and should be discarded. A new direction (possibly requiring backtracking) should then be selected. Keep trying all directions until a valid one is found. Note, there will *always* be a valid move because backtracking is allowed as a last resort.
11. Thoroughly test this basic functionality. Check that the zombies move around their quadrants.
12. Now, improve zombie movement. First, give preference to selecting a direction that doesn't require backtracking. This rule is easy! Try the three other possible directions (at random!) and only try backtracking when the other directions are not valid. Second, implement the

attraction of the zombie to the player (for the zombie in the same quadrant as the player). Attraction can be implemented by observing the player's position relative to the zombie and prioritizing direction of the zombie's movement in that direction. For instance, suppose the player is at (1,10) and the zombie is at (1,4). The zombie's y-coordinate (4) is smaller than the player's y-coordinate (10), and thus, the zombie should move down toward the player. Consider another possibility. The player is at (1,10) and the zombie is at (5,1). The zombie could either move toward the left, or possibly move down in this case.

13. To make the game work the best, move the zombies whenever the player is moved.
14. Add functionality to handle game start and end. Start the game when the center (`b`) is pressed. Add support to handle the game over condition by detecting the player reaches (63,63) or a zombie captures the player.
15. You're almost done! Next up, introduce the game scoring functionality. Simply record the start time of the game (when center is pressed). When the game ends, report the elapsed time as the current time minus the start time.
16. You're done!

6.2 What is the overall structure for the game?

Here is pseudo-code for the overall game. In essence, there is a "game loop" that checks for advancement of time steps and continually checks the keypad for a keypress, processing the keypresses.

```
time_step = read time;
loop {
  key_pressed = read keypress status;
  if (key_pressed) then {
    read key value;
    process key press:
       left key: move player left;
       right key: move player right;
       up: move player up;
       down: move player down;
    move zombies (see below);
  }
  current_time = read time;
  if (current_time - time_step >= 500ms), then {
    randomly move zombies in quadrant different than player;
    move zombie toward player in same quadrant as player;
    time_step = read time;
  }
}
```

6.3 How to move a zombie?

To move a zombie, you need to pick a direction (left, right, up or down) relative to the current position. The zombie is moved one step in the direction, assuming there is no wall in the way. For instance, suppose the zombie's current location is (x, y). Suppose you want to move the zombie to the left. The new coordinate of the zombie will be $(x - 1, y)$.

Before moving the zombie to a new location, a check should be made for a wall. Basically, the idea is to "validate" that moving to the new location is legal. In this example, `_getLED(x-1,y)` can be used to read the LED color at the new location $(x - 1, y)$. If the color is "off" (0), then there is no wall. To move the zombie, the current location is turned off. `_setLED(x,y,0)` will turn off the LED at $(x - 1, y)$. Next, the new location's color is set to red with `_setLED(x-1,y,1)`. Be sure to check whether the zombie is stepping onto the player (ending the game).

6.4 How to check the quadrant?

When a zombie moves around, it might try to move from its quadrant to an adjacent one. This should not be allowed. Prior to moving the zombie, you should check that the new position is within the zombie's quadrant. For each quadrant, keep track of the minimum coordinate, say (x_{min}, y_{min}) and the maximum coordinate, say (x_{max}, y_{max}). A position (x_{alien}, y_{alien}) is valid when $x_{min} \leqslant x_{alien} \leqslant x_{max}$ and $y_{min} \leqslant y_{alien} \leqslant y_{max}$.

6.5 How to pick a random direction?

There are many ways to select a random direction to move a zombie. A naïve way chooses random numbers, corresponding to directions, and tries to move the zombie in the selected direction. Of course, the rule about avoiding reversing direction needs to be obeyed. It turns out that enforcing this rule makes the random selection more challenging; you need to keep track of whether you've tried all moves prior to reversing direction. Perhaps a better strategy is the following one. Suppose we declare a byte array with 4 elements, called `direction`:

```
.data
direction: .byte 0,1,2,3
```

Let's use 0 for Up, 1 for Left, 2 for Right, and 3 for Down. Whenever we need to select a random direction, we initialize the array to hold the values 0, 1, 2 and 3. The *last element*, `direction[3]`, is set to the *reverse direction* of the zombie's current direction. For instance, suppose the zombie is moving Up. Thus, `direction[3]=3` in this example. If you use the suggested direction values, the reverse direction is the complement of the current direction.

Once `direction`'s elements are initialized, randomly permute the first 3 array elements. Suppose you have a function `permute(array, length)`, which will randomly shuffle the values in the array. Simply invoke the function, `permute(direction, 3)`, to permute elements 0, 1, and 2.

OK, now, we have a random sequence of directions. The final step iterates (in order) through the array. elements For each element, try to move the zombie in the given direction. If the move is valid, make the move and exit the loop. Note, the reverse direction is always valid and it will only be tried as a last resort with this approach.

6.6 How to permute an array?

Here is pseudo-code to permute an array of bytes with a given length.

```
procedure permute(int direction[], int length)
begin
  // permute the directions with length array elements
  for i = length downto 2 do begin
     // pick random integer in range [0..i-1]
     int r = random(0, i-1);

     // values i and r in the array
     byte tmp = direction[i-1];
     direction[i-1] = direction[r];
     direction[r] = tmp;
  end
end
```

Because we need to only permute an array with 3 elements, you can actually write out the individual steps of the loop to do the permutation (avoiding the need to loop).

6.7 How to make a zombie move toward the player?

OK, this is the fun part! The basic idea is to determine the zombie's position relative to the player's position, and move in the direction that has the shortest distance. To do this, we need to consider the *future* (x, y) position of the zombie. Because the zombie can possibly move in one of four directions, there are four possible future positions. We select the future position that is both *valid* and has the shortest distance.

Given a future position of the alien (x_{alien}, y_{alien}) and the player's position (x_{player}, y_{player}), we can use the direct line distance:

$$d = (x_{player} - x_{alien})^2 + (y_{player} - y_{alien})^2 \tag{1}$$

Using this equation, the distance can be computed for the four possible directions. The direction with the smallest d is selected for the zombie's next position, assuming the coordinate is valid.

When is the next position valid? Really, there's only one case to worry about! If the next position has a wall, then it is not valid and the zombie can't move in that direction. Alas, zombies don't have super powers to walk through walls (really, that's a *good* thing). You can use `_getLED` as suggested previously to check whether the zombie's next position has a wall.

If you followed the suggestion above to randomly move zombies, the process to move toward the player is not too difficult. Here is the idea. Put the reverse direction of the zombie at the end of the `direction` array. Fill in the rest of the array elements with the remaining directions. Sort the first 3 array elements according to the distance of each future position. How to sort three elements is left as an exercise for the reader. When you sort the directions, be sure to use the *distance* of the *future position* of the alien as the sort value!

Lastly, iterate through the elements (in order) of `direction` to try moving the zombie. If a direction is invalid (there is a wall at the future position), discard the direction and go on to the next array element. The reverse direction is tried as a last resort (it will always succeed).