

CS/COE 447 Computer Organization

Fall 2009

Programming Project #2

Assigned: October 26. Due: November 9 by 11:59PM.

Project Description

Crop circles, corn mazes! Yes, it's that time of the year when the corn is high and aliens land in corn fields, creating mazes and leaving behind pots of gold. People flock to the corn fields to "run the mazes" and collect the gold. What's more fun, the mazes are often haunted at Halloween, making them a spooky adventure to navigate.

In this project, we'll create our own simulated corn maze (sans ghosts and goblins, except for bugs in your program). The simulator will show a maze and let a player (you) navigate through the maze. The maze will have pots of gold sprinkled throughout it. The player's objective is to collect the gold in as few moves as possible.

For this project, we'll use an enhanced version of the LED display simulator. This enhanced display simulator has a keypad, with left, right, up and down arrow keys. The simulator has tri-colored LEDs. An LED can be set to red, orange, green, or black (off). In essence, the simulator is a simplified PSP Go (OK, it's a *greatly* simplified game console!).

Your program will let the user play a game. A game is represented as a maze with gold and a current player position. The maze has walls that block the movement of the player. The player's position is changed by keypad input, corresponding to one step left (left arrow), right (right arrow), up (up arrow), or down (down arrow). When a player moves over a pot of gold, the gold is collected and removed. A game is over when all the gold is gone. When a game finishes, the number of valid moves is printed and the user is prompted whether to play another game.

Project Details

The maze will be drawn on the LED display. A wall is shown as a sequence of red dots (LEDs). A pot of gold is shown as a single orange dot (LED). The player's current position is shown as a single green dot. When a player moves over an orange LED (gold pot), the LED is turned off when the player moves off it (i.e., the gold is "removed").

When a game ends, your program must print to the console (Run I/O window in Mars) the total number of *valid* moves made. This number is the game score. After the score is printed, your program should ask a "Yes/No" question (to the console) about whether to play a new game.

The program must support a small number of pre-determined mazes. It must start with the first maze and step through the mazes on each game (e.g., start with maze 0, when the user finishes it, then play maze 1, and so forth). The program should gracefully end with a message to say there are no more games when all games have been played.

Here is an example maze with the player's position, pots of gold and walls:

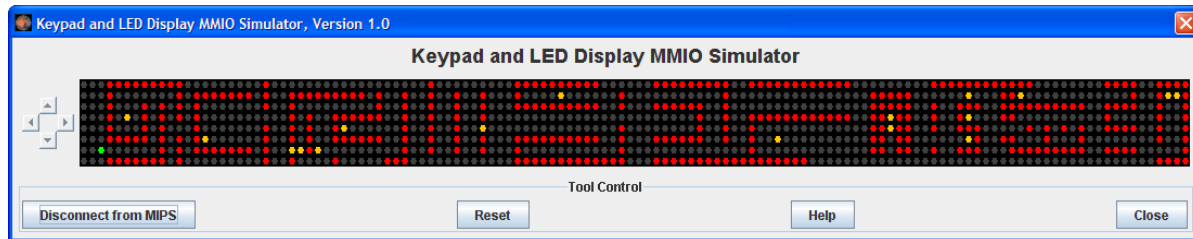


Table 1: Example Maze Game

Game Rules

Here is a summary of the game rules:

- The player's initial position is coordinate (0,0).
- A player cannot move on to a red dot.
- A player is moved one dot at a time with the keypad.
- The player cannot be moved off the game board, even if there's no wall on the border.
- When a player moves on to an orange dot, the dot should be changed to green. When the player moves off this spot, the dot should be turned off (i.e., the gold is removed). It should not be possible to collect gold from a spot more than once!!
- The game is over when all gold pots are removed.
- The game score is the number of valid moves. An attempted move that doesn't cause the player to change position should not increment the number of valid moves (e.g., there's a wall on the right side of the player and the right arrow key is pressed).
- When a game is over, the player is asked whether to play another game. If "No", the program exits. If "Yes" and there are games remaining to play, a new game begins. If "Yes" and there are no games remaining to play, the program prints an appropriate message and exits.
- When the program starts, it *must start with game 1 from the CS/COE 0447 web site*. You may add your own games.

The program must run with the Mars simulator and be implemented as a MIPS assembly language program. The programming project is an individual effort. You may ask other students how they are approaching the problem, but you must not work together on the coding.

Enhanced LED Display Simulator

The project requires an enhanced version of the LED display simulator from the CS/COE 447 web site. For off-campus access, you'll need the user name and password announced in class.

To run the display simulator, click on the Tools menu tab in Mars. Select the option "Keypad and LED Display MMIO Simulator". When the display opens, select Connect to MIPS. This action establishes communication between the MIPS simulator and the LED display simulator. Load your program and run it (try `arrowdemo.asm`). To end the LED display simulator, Disconnect and

Close it. You can clear the LED display with the Reset button. You may need to Reset and Connect whenever you load or assemble a program.

When the display simulator is connected to Mars, the program is running, and a new breakpoint is inserted, the breakpoint won't actually stop the program, even though it may be hit. This is a bug in Mars. The solution is simple: Insert all breakpoints prior to running your program. If you need to insert new breakpoints, then stop the program. Insert the breakpoints. Restart the program. It may also help to Disconnect the display simulator and then Connect it again.

If you have problems (bugs!) with the LED Display Simulator, let Dr. Childers and/or Santiago know. This simulator was written for this project. You might encounter a bug, but it is well tested.

Technical Specification of the Keypad and LED Display MMIO Simulator

The simulator models a display with 8 rows and 128 columns of LEDs. Each LED is associated with *two bits* in main memory. An LED is turned off (black) when its value is 00. The LED is red when its value is 01, orange when its value is 10, and green when its value is 11.

There are 1,024 LEDs, so 2,048 bits (256 bytes) in memory are needed. The values stored in the address range [0xFFFF0008, 0xFFFF0108] are used for the display. The bytes in the display's address range are mapped to specific LEDs. The display rows are numbered 0 to 7 (top to bottom) and the columns are 0 to 127 (left to right). The upper left corner is coordinate (row 0, column 0) and the lower right corner is coordinate (7, 127).

The upper left corner of the display is mapped to the lowest address (0xFFFF0008) and the lower right corner is mapped to the highest address (0xFFFF0108). Bits 7 and 6 in the byte at 0xFFFF0008 control the LED at (0,0). Bits 5 and 4 in the byte at 0xFFFF0008 control the LED at (0,1), bits 3 and 2 control the LED at (0,2), and bits 1 and 0 control the LED at (0,3). The byte at 0xFFFF0009 control the next four LEDs in the row, and so forth. The table below gives the mapping between LED coordinates and addresses. The table shows word addresses.

	Columns 0-31	Columns 32-63	Columns 64-95	Columns 96-127
Row 0	0xFFFF0008	0xFFFF0010	0xFFFF0018	0xFFFF0020
Row 1	0xFFFF0028	0xFFFF0030	0xFFFF0038	0xFFFF0040
Row 2	0xFFFF0048	0xFFFF0050	0xFFFF0058	0xFFFF0060
Row 3	0xFFFF0068	0xFFFF0070	0xFFFF0078	0xFFFF0080
Row 4	0xFFFF0088	0xFFFF0090	0xFFFF0098	0xFFFF00A0
Row 5	0xFFFF00A8	0xFFFF00B0	0xFFFF00B8	0xFFFF00C0
Row 6	0xFFFF00C8	0xFFFF00D0	0xFFFF00D8	0xFFFF00E0
Row 7	0xFFFF00E8	0xFFFF00F0	0xFFFF00F8	0xFFFF0100

The *least significant byte* of a word controls the LEDs at the low coordinates for the word. For example, the *least significant byte* of the word at 0xFFFF0050 controls row 2, columns 32 to 35. Within a byte, the *most significant bits* of the byte (bits 7 and 6) control the low coordinate for the

byte. For example, bits 7 and 6 in the byte at address 0xFFFF0050 are LED position (2, 32) and bits 1 and 0 are position (2, 35).

Your program needs to get input from the keypad. To get input, two memory locations are loaded. One memory location indicates *when* a key (arrow) has been pressed. The other memory location indicates *which* arrow key was pressed.

To get a value from the keypad takes two steps. In the first step, the keypad status is loaded from memory byte address 0xFFFF0000. If the byte loaded is 0, then an arrow key was not pressed. If the byte is 1, then an arrow key was pressed. When the loaded byte is 1, the second step is done. In the second step, a byte is loaded from memory byte address 0xFFFF0004 to get which arrow was pressed. The value read determines which arrow key was pressed. The values are:

Key	Value
Up arrow	0xE0
Down arrow	0xE1
Left arrow	0xE2
Right arrow	0xE3

Your program should use a “polling loop” to check for a key press. A “polling loop” repeatedly loads the byte at memory address 0xFFFF0000 until the loaded value is 1. Once a key is pressed (i.e., the loaded word is 0x1), your program should determine which arrow was pressed and update the game as appropriate. After the game is updated, go back to polling for input.

Suppose the user presses the Left arrow. The byte at memory address 0xFFFF0000 is set to 0x1 by the simulator. Your program loads the byte at this address and finds the byte is 1. Next, your program loads the byte at address 0xFFFF0004. Because the left arrow was pressed, the byte will be 0xE2. After the value at 0xFFFF0004 is read, the simulator resets the byte at 0xFFFF0000 to 0.

Turning in the Project

You must submit a compressed file (.zip or .tar.gz) containing:

- maze.asm (the game program)
- README.txt (a help file - see next paragraph)

Put your name and e-mail address in both files at the top. Use the README.txt file to explain the algorithm you implemented for your programming assignment. If you have known issues (e.g., bugs, etc.) with your code, you should specify those clearly in this file.

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

```
<your username>-pa02.zip (or .tar.gz)
```

For example: sab104-pa02.zip

Files submitted after November 9 at 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, we can not guarantee to respond to the last minute problem before the deadline.

Where to Submit

To submit your compressed file, you have to use anonymous FTP to `cs.pitt.edu` and go to `'incoming/CS0447-bock/pa-02/'` directory and put your file inside that directory. To use anonymous FTP, you can use any software that can do FTP (e.g., the PuTTY utilities) or use your favorite file explorer and type `ftp://cs.pitt.edu` in the address-bar. If it asks for username and password, just use `anonymous` as the username and leave the password field blank. Alternatively, you can type `ftp://anonymous@cs.pitt.edu` in the address bar.

NOTE: you will **not** be able make any kind of modification (rename, delete, copy and so on) to your file once it is submitted. If you want to make changes, you have to resubmit your file with a version number appended to the filename (but it is not recommended, please submit your final version). If you get problems in submitting your file there, let Santiago know by sending an email to `sab104@cs.pitt.edu`.

NOTE: Your assembly language code 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.

Project Hints

Think and plan carefully. This project may sound complex, but the *actual code is not complex*, if you make good use of functions and think about how the game should actually work. A well designed implementation should be about 200-250, maybe 300, lines of assembly (without comments or data declarations). It will take time to complete. Start early.

There are example game boards on the CS/COE 0447 web site. These examples are represented by strings. Each board has 8 strings with 128 characters. A character in a string gives the initial color at an LED position. 'x' is black, 'R' is red, 'O' is orange and 'G' is green. See the web site for more details.

You can use registers and the current values of the LEDs to record the game state. Two registers can track the (x, y) coordinate of the player. The LEDs record the location of walls and gold. For example, consider what happens when a player attempts a move to the right. Your program should load the LED value at the target position (the LED to the right of the current position) into a register. If the LED is red, then there's a wall in the way of the move. The move should be discarded. If the LED is black, then there is nothing in the way of the move and it should be accepted (i.e., the player is moved to the right and the current position updated). If the LED is orange, then there's gold in the spot and the move should be accepted. If the LED is green, something is terribly wrong! The only LED that should be green is the one at the player's position.

In addition to the two registers for the player's (x, y) position, you may also want to use two more registers for the game state. One register could track the number of gold pots remaining. When this register is decremented (on a move to an orange LED) and reaches 0, the game is over. Another register could record how many valid moves have been made. This register keeps the score, which is reported at the end of the game.

You may want to divide your program into some functions, such as:

- `void setLED(int x, int y, int color)` - sets color of the LED at (x, y) to color
- `int getLED(int x, int y)` - return color of the LED at (x, y).
- `void drawBoard(address board)` - initializes display as specified by the eight strings (128 * 8 = 1024 characters) at address board.
- `int checkMove(int newX, int newY)` - checks legality of move (newX, newY).
- `void makeMove(int newX, int newY)` - updates current player position to (newX, newY), number gold pots remaining and the score. Also, update the LED display itself.

You can develop and test these functions one at a time. This will simplify the program.