CS/CoE 447 Computer Organization
Spring 2016

Programming Project
Assigned: February 22. Due: March 18 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 "Bugs", that will most certainly land you a job at Atari. You will impress Nolan Bushnell! In the game, bugs are trying to invade from outerspace and take over the world. There is a brave bug-buster that has been tasked to rid the atmosphere of the bug horde. The bug-buster has exactly two minutes to accomplish this mission! Fortunately, the bug-buster is equipped with a phaser to scare away the bugs. Thus, the game player (user) controls the bug-buster and the phaser to rid Earth's atmosphere of the bugs in less than two minutes.

The game is played on a grid (i.e., rows and columns). The bug-buster is always in the bottom row (63) but can move horizontally (left/right). The bug-buster can fire the phaser from any x position in row 63. The bug-buster's movement and phaser are controlled by the game player. Bugs randomly appear in the top row (0) and move downward toward the bug-buster in a straight line. When fired, the bug-buster's phaser emits an electromagnetic pulse that moves in a line (in a column) from the bug-buster's position to the top of the game board. A pulse that hits an alien turns into a pulse wave that expands outward from the bug's position. A bug disappears when hit by a pulse. A pulse wave that reaches another bug causes similar beavior, causing a cascade effect that chases away many bugs. After two minutes, the game ends. If bugs remain, the game is lost (sadly, Earth is doomed). If no bugs remain, the game is won (happily, Earth is saved). At the end of the game, a score is reported. The score is printed as the number of bug hits and the number of phaser firings.

For this project, you will implement Bugs 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 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 Bug Behavior

The bugs have the following behavior:

- Bugs are placed on the game board at random coordinates $(x, 0)$, such that $0 \leqslant x \leqslant 63$.

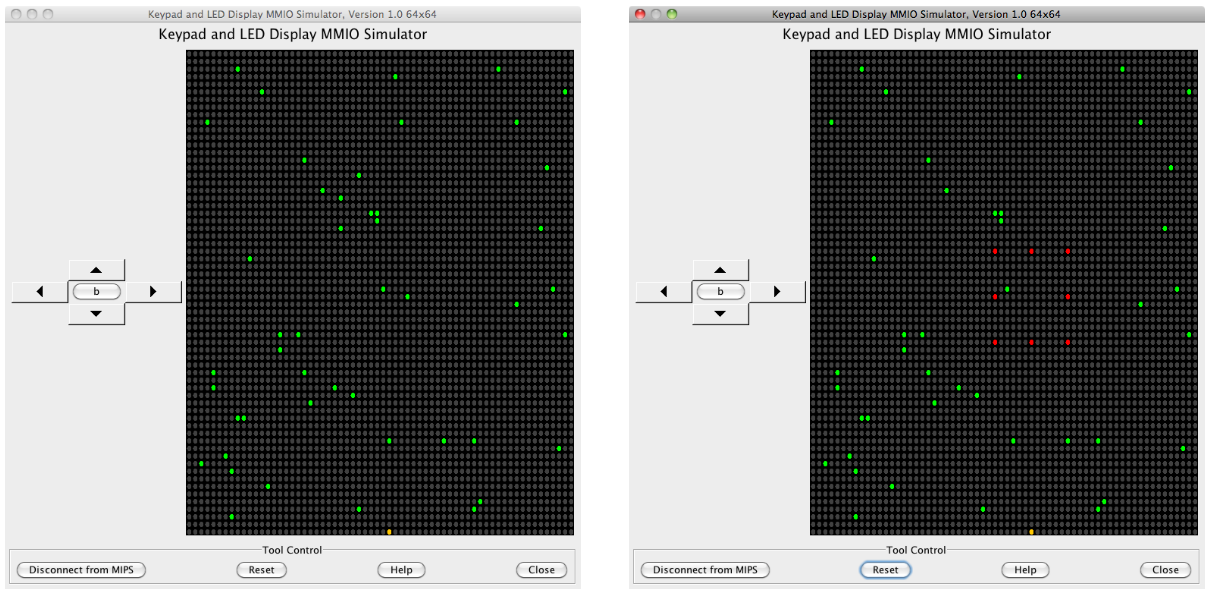
Keypad and LED Display MMIO Simulator, Version 1.0 64x64
Keypad and LED Display MMIO Simulator
b
Tool Control
Disconnect from MIPS
Reset
Help
Close
Keypad and LED Display MMIO Simulator, Version 1.0 64x64
Keypad and LED Display MMIO Simulator
b
Tool Control
Disconnect from MIPS
Reset
Help
Close

Figure 1: An exmaple game board (left) and a bug hit (right).

- Place only 3-4 bugs at a time. Add more bugs as the game progresses to increase the level of difficulty. By the end of two minutes, $\geqslant 64$ bugs should have been added.
- Multiple bugs are allowed in a column (e.g., there can be bugs at both $(30, 50)$ and $(30, 0)$).
- Bugs are animated. A bug moves from its initial position $(x, 0)$ to $(x, 62)$. The condition imposed on y ensures there is no bug in the bottom row (i.e., with the bug-buster).
- The bug moves down one LED (e.g., $(x, 10)$ to $(x, 11)$) every $100ms$.
- Bugs are green (an LED for a bug is set to the green color).

2.2 Bug-Buster Behavior and Movement

2.2.1 Bug-Buster Movement

- The bug-buster is drawn as a single LED set to orange.
- The bug-buster is moved by pressing the left (◂) or right (▸) arrow keys.
- The bug-buster moves left and right in row 63.
- The possible coordinates for the bug-buster are $(x, 63)$, where $0 \leqslant x \leqslant 63$.

2.2.2 Phaser Behavior

- The bug-buster's phaser is fired by pressing the up (▴) key.
- The phaser is fired from the bug-buster's current x position. The phaser may be fired multiple times in sequence from the same position, causing several pulses (one after another).
- When the phaser is fired, an electromagentic pulse is emitted. The pulse moves up one LED every $100ms$ in the column from which it was fired. The pulse is a red dot (LED) that moves in a column. For example, suppose the phaser was fired at $(32, 63)$. The pulse begins at position $(32, 62)$ and moves to position $(32, 61)$ after $100ms$.
- If the pulse reaches the top of the game board, it disappears.
- If the pulse reaches a bug, the pulse expands in a wave and the bug is removed.
- The wave expands in the diagonal, horizontal and vertical directions from the bug it hit. The wave's size is increased by one LED every $100ms$. For example, suppose the pulse hits a bug

at $(32, 20)$. After $100ms$, 8 LEDs are turned to red. The LEDs are $(31, 20)$, $(31, 19)$, $(31, 21)$, $(32, 19)$, $(32, 21)$, $(33, 20)$, $(33, 19)$, and $(33, 21)$. After another $100ms$, the wave progresses outward along the diagonals, horizontals and verticals from the previous coordinates.

- The wave has a maximum width and height of 20 LEDs. Thus, the wave reaches at most 10 LEDs in any direction from the bug's position.
- If a wave hits a bug, a new wave is started from that position. This behavior can cause a cascade of waves that remove many bugs.

2.3 Game Start, Completion and Scoring

- When invoked, the program initializes the game by placing 3-4 bugs at random positions in row 0 (i.e., (x,0)) and the bug-buster at coordinate $(32, 63)$.
- The game itself and animation starts when the player presses the center (`b`) key.
- Once the center key is pressed, the bug-buster has two minutes to scare aways the bugs.
- When two minutes are up, the game is over. The program prints the score and terminates.
- The game may be ended early by pressing the down (▾) key.
- The program prints "The game score is *bug-hits : phaser-firings.*" when the game completes. *bug-hits* is the number of bugs hit by a pulse or wave. *phaser-firings* is the number of times the phaser was fired by the player. A newline should be printed before and after the message.

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>-bugs.asm` (the Bugs 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-bugs.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 18, 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. Identify a useful data structure for the game events (see below).
3. Implement the data structure and *thoroughly* test it before trying the animation.
4. Develop a function to move a pulse from its current position to the next one. Call this function several times to fire a pulse in a fixed column.
5. Once the pulse works, add the wave. Put a bug (green LED) in the same fixed column as the pulse. Call the function several times until the pulse reaches the bug to check that the wave works correctly. Now, allow multiple pulses. You may need to change the pulse movement function to distinguish between different pulses.
6. Write and test a function to check for a keypress and to return the key, if pressed.
7. Inside a loop, call the keypress function. Does it detect a sequence of keypresses?
8. Add functionality to detect left (◂) and right (▸) key presses. Initially, set the LED at $(32, 63)$ to orange (bug-buster). Move the bug-buster left or right when an arrow key is pressed.
9. Add functionality to detect an up key (▴) press. Use your data structure to record the firing of the phaser (from the bug-buster's current x position). Allow the phaser to be fired multiple times, including in the same column.
10. Modify the keypress loop to check whether $100ms$ has elapsed. Call the *100ms* value the "time step". This is how often the animation is updated for the phaser.
11. Whenever $100ms$ has elapsed (one time step), move the pulses and waves recorded in your data structure. Update the data structure as appropriate (e.g., remove a pulse when it reaches row 0).
12. OK, you're close now. Write a function to pick a random $(x, 0)$ coordinate, such that $0 \leqslant x \leqslant 63$. Add a loop to randomly place 2-3 bugs on the display by turning LEDs to green. Try firing the phaser at the LEDs.
13. Now, add animation for the bugs. This animation works similarly to the phaser animation.
14. Introduce the game scoring logic. Count the number of bug hits.
15. 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 when two minutes have elapsed or the down (▾) key is pressed. Output a game over message and the score.
16. You're done!

6.2 What is a good data structure for bugs?

The critical aspect of this project is to use a good data structure for the game actions, or *events*. Events record and track actions in the game, such as the movement of pulses and waves. This game needs only a few event types:

1. *pulse move*: records position of pulse to be updated on next time step.
2. *wave move*: records position of a wave to be updated on next time step; a pulse move event is turned into a wave move event when a pulse hits a bug (or a bug hits a pulse by moving onto a pulse).

3. *bug move*: records position of a bug to be updated on next time step.
4. *game over*: ends the game; created when center button pressed.

Depending on your approach, you may need some other event types as well.

Associated with each event, there are a few pieces of information. An event needs a *type* (what kind of event), a coordinate (x, y), a *radius*, and a time *start*. Not all of this information is needed by each event, but it simplifies the data structure to record it. For example, the game over event type uses only *start* to record the start time of the game. The (x, y) coordinate is used by the wave, pulse and alien event types to track pulse, wave, and alien location. *radius* is used by the wave event type to record wave perimeter as a distance from the coordinate where a pulse hit a bug.

The information for each event type can be kept compactly; you'll probably need only two words to hold the information, if you "pack" multiple pieces of information as bytes into a word. For example, byte 0 of word 1 might be the event type, byte 1 of word 1 might be x, byte 2 of word 1 might be y, byte 3 of word 1 might be *radius*, etc.

The game events will be kept in your data structure. This data structure should allow inserting, removing and searching events as the game is played. A "first-in, first-out queue" will work well (do you remember this from CS 445? See `http://en.wikipedia.org/wiki/FIFO_computing`) to hold the events. Let's call this structure the "event queue". Insert new events in the queue and remove events from the queue to process them.

To manipulate the queue, you'll need three functions:

`_insert_q`: Insert event at the end of the queue.
`_remove_q`: Removes and returns an event from the head of the queue.
`_size_q`: Returns number of events in queue.

Here's how to use the queue. Every $100ms$, process the events in the queue. The "$100ms$" is the *time step.* All current events in the queue are processed on the current time step. Process the events one at a time. As the events are processed, new ones will be added to the queue and old ones will be removed. It is important that *events newly added on the current time step are not processed on the current time step.*

To process events, remove one from the queue. Check the event type. If the event is a pulse move, then move the pulse up one position. Turn off the LED at the current location (x, y). Check whether the pulse is in row 0 (i.e., $y = 0$). If so, skip any further processing for this event and go to the next one. Otherwise, check whether there is a bug (green LED) at the next location $(x, y - 1)$. If there's a bug in this location, create a wave event at $(x, y - 1)$ with $radius = 0$ and turn off the LED at $(x, y - 1)$. Insert the wave event in the queue. If the pulse didn't hit a bug, set the LED to red at $(x, y - 1)$. Add a new event for the next pulse move; use $(x, y - 1)$ as the new location. A similar process can be followed for the wave event type. In this case, on each wave event, increase the radius, check the perimeter distance, and update the LEDs along the wave perimeter. Be sure to check for a bug hit along the wave perimeter. The game over event is easily processed. Simply compare the time recorded in the event to the current time. If two minutes or more has elapsed, end the game. Otherwise, put the event back into the queue. A bug event type can be handled similarly to the pulse move event. Be careful: Depending on how you handle events, it is possible for a bug to move onto a pulse. In this case, a pulse hit has happened and must be handled (i.e., remove the associated pulse event and create a wave event for the hit).

A "circular buffer" is a good implementation for the queue (see `http://en.wikipedia.org/wiki/Circular_buffer`). Be sure to size the buffer large enough to allow many simultaneous events in the game (e.g., 64 pulses or waves). The queue code is relatively simple. Each function is probably a dozen assembly language instructions with a circular buffer implementation. It will help to use a queue size that is a power of two and a queue entry size that is one or two words. Be sure to check for queue overflow and handle it gracefully (end the game with an error message).

6.3 What is the overall structure for the game?

Here is pseudo-code for the overall game structure. 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: create pulse event at (player's x position, 62);
       center: create end game event;
  }
  current_time = read time;
  if (current_time - time_step >= 100ms), then {
    while events in queue {
      process events;
    }
    time_step = read time;
  }
}
```