CS 447 and COE 147 Computer Organization
Spring 2012

Programming Project 1
Assigned: February 2. Due: February 21 by 11:59 PM.

1 Description

In this programming project, you will implement a computer game called "WordSoup". The objective of the game is to guess a random word one letter at a time (e.g., in a style similar to Wheel of Fortune, without the glitz, glam and cheesiness!). Your program will randomly pick a word (from a small vocabulary) to ask the player to guess. The program prints the word with unguessed letters as underscores ('_') or asterisks ('*'). As letters are guessed, the underscores are filled in with the correct letter. Any letters with asterisks are never revealed to the player, even when guessed correctly (making the game more challenging!). These "hidden letters" are randomly chosen. The game is conducted as a series of "word rounds", where each round asks the player to guess a word. The game ends when the player says he/she wants to stop play.

A game tally—i.e., a total game score—is kept as the game progresses. The game tally is updated based on the outcome from each round. Each round has a score (point value awarded for winning the round). A round's score is initially set to the number of letters in the word. One point is deducted from the round's score everytime the player guesses incorrectly. At the end of a round (win or loss), the round's score is added to the game tally.

The player is prompted to enter a letter until all letters in the word are guessed or the player loses the round. If the player guesses a correct letter, then any letters matching the guess are filled in, unless hidden. If the player guesses incorrectly, then one point is subtracted from the round's score. If the round's score reaches zero, the player loses the round.

The game has three "short-cuts". The first short-cut ('?') gives a hint about the round's word. When the player types '?', the first remaining unguessed and unhidden letter from the left is filled in. This short-cut is ineffective for hidden letters; i.e., hidden letters must be guessed by the player. The player is allowed to use '?' three times. Each use of this short-cut deducts one point from the round's score. The second short-cut ('!') lets the player guess the entire word all at once. When the player types '!', the game prompts for the full word. If the word is guessed correctly, then the round's score is doubled and the round is won. If the player guesses incorrectly, then the round's score is doubled and negated. The round is lost. Lastly, a round can be forfeited by entering a period ('.'). When a round is forfeited, the round's score is 0.

When a round is done, the program prompts whether to continue. If the player answers 'y', (which means "yes") then the game continues. Otherwise, it ends and reports the game tally (i.e., the final total score).

2 Project Requirements

The project has several requirements, which must be followed:

- The program has a vocabulary of at least ten words, with varying word lengths.
- The first word is always "computer". Words after the first one are chosen randomly.
- Up to $\lfloor \frac{N}{2} \rfloor$ letters in a round's word are randomly chosen as "hidden letters". N is the number of letters (characters) in the word. The number of hidden letters is randomly chosen at the

start of the round. The hidden letters themselves are also chosen at the start of the round. Hidden letters are not revealed, even when guessed correctly.

- The number of letters in a word is a "round score".
- The player is prompted to enter a letter ('a' to 'z'), period ('.'), question mark ('?'), or exclamation point ('!'). The round's word is shown in the prompt. Unguessed letters are shown as underscores ('_') and hidden letters are shown as asterisks ('*'). There is a space between each letter in the word (see example output).
- Assume all input is valid and lowercase.
- Each incorrect guess decreases the round's score by 1 point.
- If the round score is decreased to ≤ 0, the round is lost.
- The score for a round is added to a game tally (total score).
- '.' is a short-cut to forfeit a round (i.e., give up). A forfeited round is worth 0 points.
- '?' is a short-cut to fill in the first unguessed letter (i.e., the first letter from the left with an underscore '_'). This short-cut skips hidden letters. For example, suppose the player guessed 's' and 'm'. After entering these letters, the word is "s m * _ _". The player then types '?' and an 'l' is revealed; the word is now displayed as "s m * l _". One point is deducted from the round's score whenever this hint is requested. The player may use '?' a maximum of three times. After three uses, the short-cut is ignored.
- '!' is a short-cut to guess the entire word. When used, the game prompts the player to enter a string for the word. If input string matches the word, then the round's score is doubled. If the input doesn't match, then the round's score is doubled and negated. After guessing the whole word, the round is done and the round's score is added to the game tally.
- After every input in a round, the round's score is printed (see sample output).
- When a round is finished, the program prints the correct word with all letters revealed, including hidden ones. The round's score and game tally are also printed.
- The player is prompted after a round with a y/n question whether to continue.
- When the program ends, it prints the game tally (total score).

The project output must be the same as shown in the example game play. This is required to make grading simpler. Similarly, the first word must be "computer" to make grading simpler. Be sure to put a blank line between each guess and to include the spaces between the underscores and aserisks (indicating a missing letter).

The game 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.

3 Turning in the Project

3.1 What to Submit

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

- `wordsoup.asm` (the WordSoup 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 play all rounds, odd display 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>-pa01.zip`
Here's an example: `musfiq-pa01.zip`

Files submitted after February 21, 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.

3.2 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`

4 Example Dialog between WordSoup and a Player

```
Welcome to WordSoup!
I am thinking of a word. The word is _ _ * _ _. Round score is 5.
Guess a letter?
m

Yes! The word is _ m * _ _. Round score is 5.
Guess a letter?
i

Yes! The word is _ m * _ _. Round score is 5.
Guess a letter?
s

Yes! The word is s m * _ _. Round score is 5.
Guess a letter?
o

No! The word is s m * _ _. Round score is 4.
Guess a letter?
a

No! The word is s m * _ _. Round score is 3.
Guess a letter?
e

Yes! The word is s m * _ e. Round score is 3.
Guess a letter?
l

You won the round! Your final guess was:
s m * l e
Correct word was:
smile
Your round score was 3. The game tally is 3.
Do you want to play again (y/n)?
n

Your final game tally is 3. Goodbye!
```

5 Programming Hints

Think before Coding: Write pseudo-code for the game prior to thinking about the assembly language. Use procedures to simplify the program. Keep in mind that the game basically has two loops: an outer loop that prompts whether to continue the game, and an inner loop that does a word round.

Selecting a Word, Hiding Letters and Storing a Word: The program randomly selects a word for each round from its vocabulary. Let's call this the "selected word". The selected word will be guessed by the player. After selected word is determined, the program randomly chooses $[0..\lfloor \frac{N}{2} \rfloor]$ letters in the word to be hidden. N is the number of letters in the word. For instance, suppose the selected word is "hello". In this case, $N = 5$ and [0..2] letters are chosen as hidden ones. Suppose 1 letter is picked to be hidden. Once the number of letters to hide is determined, the hidden letters themselves are chosen. So, in this case, a random number is picked between $[0..N-1]$. Suppose this number is 2. Thus, the first 'l' in "hello" is the hidden letter. The selected word would be initially displayed as "_ _ * _ _". See below on hints to pick randomly in MARS.

You will also need a way to represent the partially filled-in word as a round progresses. Let's call this word the "output word". When the user enters a character, compare it against the characters in the selected word. When a match is found, you put the character in the output word, according to the character's location in the selected word. If a character occurs multiple times in the word, then you should fill in all occurrences of the character. If the character matches a hidden position, then the character is not shown in the hidden position.

You can declare a string variable to hold the output word. Initially, fill in this variable with the appropriate number of underscores and null terminate it. In the example game dialog, the selected word is "smile", with the second position (letter 'i') picked as a hidden letter. The word is output initially as "_ _ * _ _ ". When a match is found, in the first step, the character "m" is put into the output word, producing "_ m * _ _ ". The output word (a string) and the prompts are displayed with system calls.

Representing your Vocabulary: An important aspect is how to store the vocabulary. The vocabulary is nothing more than a fixed array of words. The words could be stored as a series of strings and initialized in the data segment. You will need to know the number of characters in the selected word. The `mips12.asm` example (see website) shows how to count the number of characters. Alternatively, you might store each word's length as part of the vocabulary itself.

With the vocabulary as an array, get a random number at the beginning of each round. You map this random number to a word in the vocabulary (i.e., the random number is an array index). For example, if the vocabulary is "computer, smile, sunshine", then the random number 1 corresponds to the word "smile" (counting from 0). The example `weather4.asm` (see website) shows how to create an array of pointers to strings and how to map a number to a string address.

Detecting Matches: You need to detect matching characters from the user input in the selected word. This process is done with a loop that walks through the selected word and compares each character to the one entered by the user. If the character in the word matches, then store the character (byte) in the output word string, unless it's hidden.

Picking Randomly: The program needs to get a random number to select the positions in the word to hide. MARS has system calls that help. A random generator is typically "seeded" before it is used. This establishes the sequence of pseudo-random numbers that will be generated. A common approach uses the computer's current time (when the program is started) as a seed to

the random number generator. Once the random number generator is seeded, it can be called to get a random number in a specified range. There are three useful system calls in MARS. First, system call 30 returns the current computer system time. Second, system call 40 seeds the random number generator. Invoke this system call with the low order 32 bits of the current system time as the seed. Finally, system call 42 returns a random number in a specified range. These system calls work only with MARS.