Computer Science 230
Computer Architecture and Assembly Language
Fall 2020

Assignment 2

Due: Thursday, October 29th, 11:55 pm by conneX submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on the MIPS Assembler and Runtime Simulator (MARS) as was installed during Assignment #0. Assignment submissions prepared with the use of other MIPS assemblers or simulators will not be accepted. ***Solutions which prompt the user for input will not be accepted.***

Individual work

This assignment is to be completed by each individual student (i.e., no group work). Naturally you will want to discuss aspects of the problem with fellow students, and such discussion is encouraged. **However, sharing of code fragments is strictly forbidden without the express written permission of the course instructor.** If you are still unsure regarding what is permitted or have other questions about what constitutes appropriate collaboration, please contact me as soon as possible. (Code-similarity analysis tools will be used to examine submitted work.) The URLs of significant code fragments you have found and used in your solution must be cited in comments just before where such code has been used.

Objectives of this assignment

- Write and use procedures.
- Implement parameter passing using registers.
- Implement return values using registers.
- Use a stack for saving and restoring register values.
- Use the "Digital Lab Sim" tool in MARS to experiment with the visualization of morse code.

Morse code

Morse code is a method for transmitting messages over a distance by using a combination of long and short signals (i.e., long or short light flashes, long or short electrical pulses, etc.) Until recently, the ability to send and receive Morse code messages was an important skill for anyone involved in communication over radio (such as airplane pilots, military personnel, amateur-radio operators, etc.). For a bit more about Morse code you can read the Wikipedia article at:

`https://en.wikipedia.org/wiki/Morse_code`

Our assignment will use International Morse Code to display a message using the 26 letters of the English alphabet. A demonstration of parts of this assignment can be found at:

`https://webhome.csc.uvic.ca/~zastre/seng265-202005/a2.html`

The starter file provided to you is named `a2-morse.asm`. There is only one file for this assignment.

Your work to complete this assignment is in five parts, ordered from easy to more difficult:

a) Write the procedure `save_our_souls`
b) Write the procedure `morse_flash`
c) Write the procedure `flash_message`
d) Write the procedure `letter_to_code`
e) Write the procedure `encode_message`

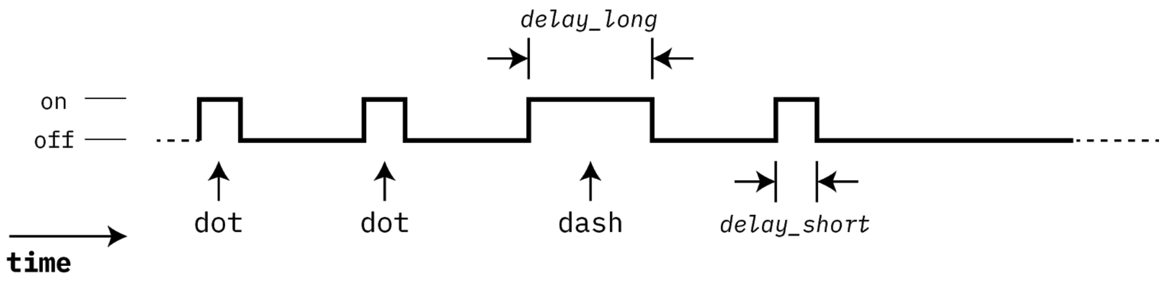
Part (a): Write procedures `save_our_souls`

`save_our_souls:`
parameters: *none*
return value: *none*

The MARS application has several additional tools, one of which is the "Digital Lab Sim". This simulates a lab board with two seven-segment digital displays plus a hex keypad. The procedures `seven_segment_on` and `seven_segment_off` contain the code necessary to turn on and off all seven segments of the right-most digital display. These procedures do not have parameters. You can use these procedures to make the display flash on and off. Feel free to play with this display (i.e., in a separate assembly file) but please do not modify those two procedures.

Also provided are two other procedures named `delay_long` and `delay_short`. These cause the simulator to "sleep" for a certain period of time, 600 milliseconds and 200 milliseconds respectively. The procedures do not have parameters. As everything is simulated, the delays might not have these exact durations on your computer; regardless, the long delay is significantly different from the short delay.

Your task is to write a procedure named `save_our_souls` which will flash the famous "SOS" morse code sequence (*dot dot dot dash dash dash dot dot dot*). Below is a timing diagram that attempts to relate the delays to the display being on or off; this shows a *dot dot dash dot* pattern. the length of flashes and delays. In real morse code the delay between dots is a bit shorter than the delay between dashes. However, in order to simplify programming, you can keep delays the same between the end of one dot/dash and the start of the next dot/dash.



For example, the pattern above (not including the final "off" part of the figure) would be:

```
jal seven_segment_on
jal delay_short
jal seven_segment_off
jal delay_long
jal seven_segment_on
jal delay_short
jal seven_segment_off
jal delay_long
jal seven_segment_on
jal delay_long
jal seven_segment_off
jal delay_long
jal seven_segment_on
jal delay_short
jal seven_segment_off
```

Please watch the video linked in the previous page which demonstrates how the "Digital Lab Sim" tool is connected to MARS.

Part (b): Write the procedure `morse_flash`

```
morse_flash:
```
parameters: one byte contained in the `$a0` register
return value: *none*

Later in part (d) of this assignment, you will write code to convert dots and dashes for a letter (e.g., ". . ." for 'S', and "- - -" for 'O') into a *one-byte equivalent*. For now, however, you are to write a procedure that takes a one-byte equivalent (passed into the procedure as the least-significant byte in a 32-bit register) and flashes the Digital Lab Sim display in a manner appropriate to the contents of that byte. **The procedure does not need to know the letter to be flashed**; all it needs is the byte itself

Below are examples of dot-dash sequences beside their one-byte equivalents (shown binary and hexadecimal notation):

`. . — .`	`0b01000010`	`0x42`
`— — —`	`0b00110111`	`0x37`
`. — .`	`0b00110010`	`0x32`
`—`	`0b00010001`	`0x11`

Each byte consists of a *high nybble* and a *low nybble*:

- The high nybble (left-most four bits) encodes the *length* of the sequence.
- The low nybble (right-most four bits) encodes the *dot-dash sequence* itself (where 0 is a "dot" and 1 is a "dash").

In the first example above (the Morse for 'F'), the high nybble encodes the number 4 (i.e., the length of the sequence), and the low nybble contains that sequence (0, 0, 1, 0). In the last example (the Morse for 'T'), the high nybble encodes the number 1 (i.e., the length of the sequence), and the low nybble contains that sequence (first three 0s are ignored, while last 1 is the dash). Notice that the low-nybble bits for sequences of length three, two and one will contain leading zeros; these leading zeros must be ignored (i.e., *they are not dots*).

The one-byte equivalent is to be turned into a series of calls to `leds_on` and `leds_off`, with a delay between calls visually distinguishing dots from dashes. This is to be done in a manner shown in the diagram provided in the `save_our_souls` section of this description.

There is a special one-byte value: `0xff`. It represents a space between words. For this, `morse_flash` must keep the display **off** for three calls to `delay_long`.

Part (c): Write the procedure **`flash_message`**

```
flash_message:
    parameters: data-memory address in $a0
    return value: none
```

Encoded messages are stored in data memory and consists of a sequence of one-byte equivalents. A sequence is terminated by the 0 value (i.e., null, just as in ending a string). For example, the one-byte equivalent sequence for "SOS" consists of the following four bytes:

```
0x30 0x37 0x30 0x00
```

The code provided for you in `a2_morse.asm` stores this sequence at the `test_buffer` memory location.

This procedure must use the address passed in `$a0` as the starting location of the sequence, and then loop through that sequence, calling `morse_flash` for each byte, until encountering the end of the sequence. Once the sequence ends, the procedure must return.

Part (d): Write the procedure **`letter_to_code`**

```
letter_to_code:
    parameters: data-memory address in $a0
    return value: one-byte equivalent stored in $v0
```

This may seem the most complicated of the assignment procedures. It must:

- Obtain the letter to be converted from input parameter `$a0`.
- Locate in the table of codes the dot-dash sequence for the letter (given to you in `a2_morse.asm`, starting at `codes` in data memory).
- Convert the dots ('.') and dashes ('–') and the length of the dot/dash sequence into the one-byte equivalent for the letter.

However, a couple of features of characters will help us here. For example, our letters are actually bytes (that is, ASCII characters) which can be directly compared with other bytes. Consider for example:

```
addi $t0, $zero, 65
addi $t1, $zero, 'A'     # must use single quotes!!!
beq $t0, $1, somewhere_over_the_rainbow
```

The comparison will always succeed. Why? Because the ASCII code for ‘A’ is 65. Note, however, that the ASCII code for ‘a’ is 97. **You may assume this procedure will be given messages with only upper-case letters (and spaces) for encoding.**

Another assist for you is that the table is aligned on an eight-byte boundary. Therefore even though some letters have longer Morse code sequences than others, each table entry has the same length. To indicate the end of a letter’s sequence we use our good friend, zero. For example, here is a snippet from the table:

```
...
.byte 'N', '-', '. ', 0, 0, 0, 0, 0
.byte 'O', '-', '-', '-', 0, 0, 0, 0
.byte 'P', '.', '-', '-', '.', 0, 0, 0
....
```

The sequence for “N” consists of a dash and a dot; that for “O” is three dashes; that for “P” a dot, dash, dash, and dot. All, however, end with one or many zeros such that each line consists of exactly eight bytes of data.

The roughest of pseudocode solutions is shown below to suggest an implementation strategy. The operation `mem[T]` refers to memory access of the location at `T` (i.e., as would occur via a load byte or `lb` instruction).

```
T = starting address of “code” array
while (mem[T] != 0)
     if mem[T] equals letter-to-be-converted:
          T = T + 1
          while mem[T] != 0:
               do something if mem[T] is a dot or
                 do something else if mem[T] is a dash
               T = T + 1
          finished (i.e., break out of outermost loop)
     T = T + 8

// At this point the encoding of letter is complete
```

Advanced tip: Strictly speaking the outermost loop is not needed, but only because of the byte-alignment used for the table. However, an acceptable and correct solution for this part (d) may include an outermost loop.

Part (e): Write the procedure **encode_message**

```
encode_message:
   parameters:
      data-memory address of message in $a0
      data-memory address of byte array to hold encoded message in $a1
   return value: none
```

This procedure is quite straightforward. The only tricky bit will be the correct use of the stack, and any bugs in `letter_to_code` – especially the way registers are saved and restored – can cause problems in `encode_message`. That said, all this procedure needs to do is (1) read each character in the original message, and for each character (2) convert it into one-byte equivalent by calling `letter_to_morse` and storing its result into the buffer. (A message is terminated with a zero.)

Once `encode_message` is completed, displaying the Morse code for a text message M made up of uppercase letters and spaces requires two steps:

1. encode the message M into some buffer B
2. call `flash_message` with the buffer B passed as a parameter

What you must submit

- Your completed work in the single source code file (`a2_morse.asm`); **do not change the name of this file!**
- Your work must use the provided skeleton `a2_morse.asm`. Any other kinds of solutions will not be accepted.

Evaluation

- 1 mark: Solution for `save_our_souls`
- 5 mark: Solution for `morse_flash`
- 5 marks: Solution for `flash_message`
- 5 marks: Solution for `letter_to_morse`
- 4 mark: Solution for `encode_message`

Therefore the total mark for this assignment is 20.

Some of the evaluation above will also take into account whether or not submitted code is properly formatted (i.e., indenting and commenting are suitably used), and the file correctly named.