Computer Science 230
Computer Architecture and Assembly Language
Spring 2018

Assignment 2

Due: Thursday March 1st, 11:55 pm by conneX submission
(Late submissions **not** accepted)

Programming environment

For this assignment you must ensure your work executes correctly on Arduino boards in ECS 249. If you have installed AVR Studio on your own computer then you are welcome to do much of the programming work on your machine. If this is the case, however, then you must allow enough time to ensure your solutions work on the lab machines. If your submission fails to work on a lab machine, the fault is very rarely that the lab workstations. "It worked on my machine!" will be treated as the equivalent of "The dog ate my homework!"

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 (Zastre).** 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 functions.
- Implement parameter passing using registers and stack.
- Implement return values using registers.
- Use stack frames where needed for parameter passing.
- Use the Arduino mega2560 board in the course lab to implement a Morse code display.

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 can be found at:

`https://youtu.be/PwtoJwlKhdw`

The starter file is named `a2_morse.asm`. There is only one file for this assignment.

Your work will be in five parts, ordered from easy to more difficult:

a) Write the functions `leds_on` and `leds_off`
b) Write the function `morse_flash`
c) Write the function `flash_message`
d) Write the function `letter_to_code`
e) Write the function `encode_message`

Part (a): Write functions `leds_on` and `leds_off`

`leds_on:`
parameters: one in r16, pass-by-value
return value: *none*

`leds_off:`
parameters: *none*
return value: *none*

The parameter to `leds_on` determines how many of the Ardujno board's six LEDs to turn on (i.e., the ones which you would have used during some lab exercises). Therefore if 6 is stored in r16, all six LEDs will be turned on. If 2 is stored in r16, then only two LEDs will be turned on (and you can choose which two those are).

`leds_off` turns off all LEDs. We could eliminate `leds_off` by saying that storing zero into r16 and calling `leds_on` turns off all LEDs. However, this assumes `leds_on` is correctly implemented. To reduce your frustration, please implement `leds_off`.

Part (b): Write the function `morse_flash`

```
morse_flash:
   parameters: one in r16, pass-by-value
   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 function that takes a one-byte equivalent (pass into the function within a register) and flashes the LEDs appropriate to the contents of that byte. The function doesn't need to know the letter to be flashed; all it needs is the byte.

Below are examples of dot-dash sequences beside their one-byte equivalents:

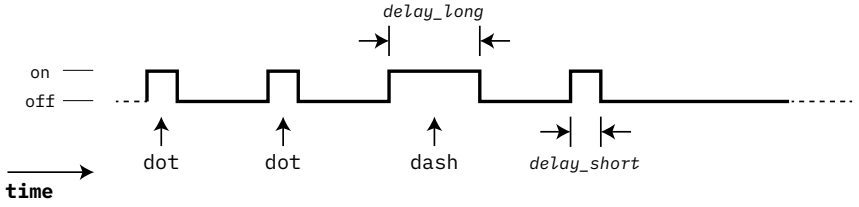
`. . – .`	`0b01000010`
`– – –`	`0b00110111`
`. – .`	`0b00110010`
`–`	`0b00010001`

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:



The previous diagram shows the timing pattern for ". . – ." The functions `delay_long` and `delay_short` are already provided for you in `a2_morse.asm`[1]. So a dot would be similar (in a pseudo-code manner) to:

```
leds_on
delay_short
leds_off
delay_long
```

while a dash would be:

```
leds_on
delay_long
leds_off
delay_long
```

Notice at the end of the timing diagram there appears to be an additional `delay_long` with the LEDs still off; this visually separates Morse code sequences.

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

Tip: When writing your implementation of `morse_flash` you may find the swap opcode helpful.

Part (c): Write the function `flash_message`

```
flash_message:
   parameters: encoded-message address on stack (two bytes);
      in effect pass-by-reference.
   return value: none
```

Encoded messages are stored in SRAM 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 `TESTADDRESS` memory location.

---

[1] The implementations of `delay_long` and `delay_short` are rather unsatisfying as they are just wrappers for repeated calls to `delay` and `delay_busywait`. In later assignments we will see how using timers and interrupts yield much more elegant implementations of such delays. For this assignment, however, your code must use only `delay_long` and `delay_short`.

This function must use the address passed on the stack 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 function may return. Given the form of parameter passing used, you will implement a stack-frame-like caller and callee sequence of instructions.

Part (d): Write the function `letter_to_code`

```
letter_to_code:
   parameters: letter on stack for conversion to one-byte equivalent; pass-by-value
   return value: one-byte equivalent stored in r0
```

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

- Obtain the letter to be converted from the stack.
- Locate in the table of codes the dot-dash sequence for the letter (given to you in `a2_morse.asm`, starting at `ITU_MORSE` in program memory).
- Convert the dots ('.') and dashes ('–') 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:

```
ldi r16, 65
ldi r17, 'A'     ; must use single quotes!!!
cp r16, r17
```

The comparison will set the Z flag. Why? Because the ASCII code for 'A' is 65. Note, however, that the ASCII code for 'a' is 97. **You may assume this function will be given messages with <u>only</u> 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:

```
...
.db "N", "-.", 0, 0, 0, 0, 0
.db "O", "---", 0, 0, 0, 0
.db "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[Z]` refers to memory access (either loading or storing, either SRAM or program memory) through using the Z register as the address.

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

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

Tip: Remember that `ld`, `lds`, and `ldd` are used for SRAM, while `lpm` is used for program memory (i.e., where `ITU_MORSE` is stored). Also remember that the AVR assembler treats program memory addresses as word addresses, so we must left-shift by one to convert such address by byte addresses.

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 and outermost loop.

Part (e): Write the function `encode_message`

`encode_message:`
parameters: address of message (i.e., sequence of upper-case letters plus spaces) to be encoded into one-byte-equivalent sequene; address of buffer in SRAM in which one-byte equivalent sequence is to be stored;
both addresses pushed onto the stack;
both parameters, in effect, are passed-by-reference
return value: *none*

This function is quite straightforward. The only tricky bit will be the correct use of the stack frame, 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 function 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

Given the form of parameter passing used, you will implement a stack-frame-like caller and callee sequence of instructions.

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 marks: Solution for `leds_on` and `leds_off`
- 4 mark: Solution for `morse_flash`
- 4 marks: Solution for `flash_message`
- 7 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.