CS2253 Programming Project 1
UTF-8 Encode/Decode

February 15, 2022

Due: March 14, 9:30am. Individual work.

1 Introduction

In this project, you will write an LC3 assembly-language program that is a (hand-written) translation, from C, of the UTF-8 encode/decode program. The C program is at `https://rosettacode.org/wiki/UTF-8_encode_and_decode#C` and solves the problem described at `https://rosettacode.org/wiki/UTF-8_encode_and_decode`.

The starting C program will be a good crash review of C, as the author has not been shy about using lots of C features in fairly few lines of code.

There is an easier and a harder version of the problem.

- In the easier version, you do not need to handle code points beyond xFFFF. So you can use 16-bit math in most cases (the C code could have been written [1] to use uint16_t instead of uint32_t).
- In the harder version, you need to handle the full range of code points, and also you need to handle MUTF-8, which means that code point 0 gets encoded differently[2]. (The given program appears to have a bug in its handling of code point 0 anyway.)

In both versions, you are to follow the C program fairly closely, to the point that the expressions in the C program should become fairly good comments when interspersed through your code. The given code has functions for `codepoint_len`, `utf8_len`, `to_utf8`, and `to_cp`. You should have subroutines with these names too.

You may want to read ahead in the textbook about arrays, pointers and structs. You already know about these from your previous C course, but the

[1] Your previous course might not have taught you about the uint16_t and uint32_t types. They are versions of unsigned ints that are guaranteed to take 16 and 32 bits, respectively.

[2] See `https://en.wikipedia.org/wiki/UTF-8#Modified_UTF-8`.

textbook also describes how these would look in LC-3, in the process of explaining the C. You may be somewhat confused by all the discussion of stack frames, though.

2 Details

Inspecting the C program, we see the following operators:
`>=`, `<=`, `&`, `~`, `*`, `>>`, `|`, `<<`. For the easier version of the problem, the only tricky ones seem to be * (multiplication, not dereferencing) and `>>` (logical right shift). However, you only multiply small positive numbers, so the "add in a loop" implementation of multiplication should be fine. For the harder version of the problem, some of the operations may need to be more than 32 bits — you will need to do some careful code analysis to determine this.

Right shift operator: Shifting bits right is not easy in the LC-3. In previous years, I suggested a complicated way to do this. This year, I recommend that a 16-bit logical right shift of `v` by 1 position be achieved something like this untested code:

```
uint16_t ans = (uint16_t) 0;
if (v & 0x8000) ans |= 0x4000;
if (v & 0x4000) ans |= 0x2000;
if (v & 0x2000) ans |= 0x1000;
// a dozen or so more omitted
if (v & 0x0002) ans |= 0x0001;
```

You can get a shift-by-k operation by repeating shift-by-1 k times. Or, more efficiently, you could modify the above code by changing the constants you OR with.

Subroutines: Don't use the stack for subroutines (or anything else). (This means that even if you find a C-to-LC3 compiler[3] you won't be able to use it to do your project.) Instead, use R0 and R1 for parameters and R0 for return values.)

You need subroutines for right shift, multiplication, outputting numbers (and bytes) in hexadecimal, plus subroutines for the four functions, excluding main, in the C program. You can make more subroutines if you wish.

You should be able to find LC-3 code online for outputting numbers in hexadecimal, so the code value is reduced to account for the fact that all you need to do is adapt this code. To avoid complaints of plagiarism, you need to put a prominent comment before any code you have borrowed, giving the original author's name (if known) and a web link to the place where you found this code.

[3] They exist!

The utf array: The C code has a data structure called `utf`. You are given a starter program that has this data structure in it. You must use it. (For the harder version of the project, you will need to adjust it.)

3 Grading

Simpler version: The values of the subtasks are as follows, for the simpler version of the project:

- rotating bits right: 10 points
- multiplication: 5 points
- hexadecimal output: 5 points
- subroutine corresponding to `codepoint_len`: 13
- subroutine corresponding to `utf8_len`: 13
- subroutine corresponding to `to_utf8`: 13
- subroutine corresponding to `to_cp`: 13
- code or subroutine corresponding to `main`: 13

It is not possible to get a grade above 85 for the easier version of the project.

Harder version: For the harder version of the project, the extension of necessary operators to 32 bits is worth an extra 15 and the change to MUTF-8 is worth 5. (Modify the code corresponding to `main` to include tests for code point 0 too.)

Note that it is possible to get up to 105 with the harder version.

Criteria: Code is evaluated on whether it works, is reasonably efficient, and meets other requirements (such as the use of subroutines). While working code will not be judged for matters of style, non-working code will be examined to see whether it appears to make sense. Bad naming, overly complex structure, poor commenting (and so forth) could make code appear to be nonsense, even if it truly were only a few lines away from working.

4 Submitting

Submit your .asm file on D2L by the deadline. Make sure that what you submit successfully assembles (comment out problematic lines, if necessary), and runs as well as it can.