CMPEN 331 – Computer Organization and Design
Lab 2_part2

You should copy-and-paste the file lab2_part.txt into the MARS editor, and then save the file as your own, for editing. Spend some time trying to see what it does.
The parts that need to be changed in main are between the comment lines Your part starts here and Your part ends here. You should also add more lines to the register assignment table, to describe what you did. Additional comments will probably be helpful. There are also two character strings that need to be changed, to put your names in the output.

Here are the Mars Messages and Run I/O panels, after assembling and running the starter program.

```
Assemble: assembling /.../lab2.asm

Assemble: operation completed successfully.

Go: running lab2.asm

Go: execution completed successfully.

CMPEN 331 Lab 2
Student's name
0 0x00000000 0x00000003 00000000000000000000000000000000 00000000000000000000000000000011
1 0x00000024 0x00000003 00000000000000000000000000100100 00000000000000000000000000000011
2 0x0000007e 0x00000003 00000000000000000000000001111110 00000000000000000000000000000011
3 0x0000007f 0x00000003 00000000000000000000000001111111 00000000000000000000000000000011
4 0x00000080 0x00000003 00000000000000000000000010000000 00000000000000000000000000000011
5 0x000000a2 0x00000003 00000000000000000000000010100010 00000000000000000000000000000011
6 0x00000627 0x00000003 00000000000000000000011000100111 00000000000000000000000000000011
7 0x000007ff 0x00000003 00000000000000000000011111111111 00000000000000000000000000000011
8 0x00000800 0x00000003 00000000000000000000100000000000 00000000000000000000000000000011
9 0x000020ac 0x00000003 00000000000000000010000010101100 00000000000000000000000000000011
10 0x00002233 0x00000003 00000000000000000010001000110011 00000000000000000000000000000011
11 0x0000ffff 0x00000003 00000000000000001111111111111111 00000000000000000000000000000011
12 0x00010000 0x00000003 00000000000000010000000000000000 00000000000000000000000000000011
```

```
13 0x00010348 0x00000003 00000000000000010000001101001000 00000000000000000000000000000011
14 0x00022e13 0x00000003 00000000000000100010111000010011 00000000000000000000000000000011
15 0x0010ffff 0x00000003 00000000000100001111111111111111 00000000000000000000000000000011
16 0x89abcdef 0x00000003 10001001101010111100110111101111 00000000000000000000000000000011
All done!

-- program is finished running –
```

Your output should be the same, except for the name, and the value of n that is printed (it's a constant 3 in the starter version).
The assignment is based on a short function used in the implementation of the UTF-8 data format. Here are some descriptions of UTF-8:

- http://en.wikipedia.org/wiki/UTF-8
  - The function in question is write_utf8() in the Sample Code section.
  - There are also some examples of the data formats in the Description section.
- http://www.ietf.org/rfc/rfc3629.txt
  - This is the official definition of UTF-8, from which we will take one diagram.
- http://www.cl.cam.ac.uk/~mgk25/unicode.html
  - UTF-8 and Unicode FAQ for Unix/Linux
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!), by Joel Spolsky
  - Useful background information, exactly as the title says.

The idea behind UTF-8 is to augment a character code with some additional bits to protect against certain kinds of communication failures. Here is the standard diagram:

```
   Char. number range   |        UTF-8 octet sequence
      (hexadecimal)     |              (binary)
 -----------------------+---------------------------------------
 0000 0000 - 0000 007F | 0xxxxxxx
 0000 0080 - 0000 07FF | 110xxxxx 10xxxxxx
 0000 0800 - 0000 FFFF | 1110xxxx 10xxxxxx 10xxxxxx
 0001 0000 - 0010 FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
```

The term "octet" means "8 bits", which everyone now thinks of as one byte. There were once computers whose byte size was not 8 bits, but they are all gone now.

The basic if-then-else structure, from write_utf8(), is

```
if (code_point < 0x80) {
 ... case 1
} else if (code_point <= 0x7FF) {
 ... case 2
} else if (code_point <= 0xFFFF) {
 ... case 3
} else if (code_point <= 0x10FFFF) {
 ... case 4
} else {
 ... case 5
}
```

In the starter version provided, we used the variable j instead of code_point. Each case should compute n from j.

It's going to be a lot easier if you sketch the solution in C, and then rewrite it into MIPS assembler, inserting the C version as a comment. Start with the if-then-else structure, and test that with some bogus values for n. Then, write each of the five cases separately; two of these are trivial, and the other three have a lot of features in common.

The MIPS assembly code is slightly easier to write if all the tests are < instead of a mixture of < and <= . Also, treat the registers as if they contain unsigned integers (when using a numeric instruction) or simple bit strings (when using logical and shift instructions).

In case 1, j fits in 7 bits, and it is expanded to 8 bits with a leading 0 bit, which yields the same value. In case 5, it's an error, so n is -1 or 0xFFFFFFFF; that's not the proper treatment of errors according to UTF-8, but it's certainly easier.

The following comments describe how the bits of j are to be rearranged to form the bits of n.

```
if (j < 0x80) {
  // j fits in 7 bits, expand to 8 bits
  // n =  j
} else if (j <= 0x7FF) {
  // j fits in 11 bits, expand to 16 bits
  // b = low 6 bits of j
  // a = next 5 bits of j
  // n = 110 a 10 b
  //                    5    6 bits in
  // j =            aaaaa bbbbbb
  //           3     5  2      6 bits out
```

```
        // n =     110 aaaaa 10 bbbbbb
      } else if (j <= 0xFFFF) {
        // j fits in 16 bits, expand to 24 bits
        // c = low 6 bits of j
        // b = next 6 bits of j
        // a = next 4 bits of j
        // n = 1110 a 10 b 10 c
        //                   4      6      6 bits in
        // j =            aaaa bbbbbb cccccc
        //        4    4  2      6  2      6 bits out
        // n = 1110 aaaa 10 bbbbbb 10 cccccc
      } else if (j <= 0x10FFFF) {
        // j fits in 21 bits, expand to 32 bits
        // d = low 6 bits of j
        // c = next 6 bits of j
        // b = next 6 bits of j
        // a = next 3 bits of j
        // n = 11110 a 10 b 10 c 10 d
        //                      3      6      6      6 bits in
        // j =                aaa bbbbbb cccccc dddddd
        //         5   3  2      6  2      6  2      6 bits out
        // n = 11110 aaa 10 bbbbbb 10 cccccc 10 dddddd
      } else {
        // j is outside the UTF-8 range of character codes
        // n = 0xFFFFFFFF
      }
```

Here is the output from a correct solution, using MARS. The name strings have not been changed - be sure you do that.

```
CMPEN 331 Lab 2
Student's name
0 0x00000000 0x00000000 00000000000000000000000000000000 00000000000000000000000000000000
1 0x00000024 0x00000024 00000000000000000000000000100100 00000000000000000000000000100100
2 0x0000007e 0x0000007e 00000000000000000000000001111110 00000000000000000000000001111110
3 0x0000007f 0x0000007f 00000000000000000000000001111111 00000000000000000000000001111111
4 0x00000080 0x0000c280 00000000000000000000000010000000 00000000000000001100001010000000
5 0x000000a2 0x0000c2a2 00000000000000000000000010100010 00000000000000001100001010100010
6 0x00000627 0x0000d8a7 00000000000000000000011000100111 00000000000000001101100010100111
7 0x000007ff 0x0000dfbf 00000000000000000000011111111111 00000000000000001101111110111111
```

```
8 0x00000800 0x00e0a080 00000000000000000000100000000000 00000000111000001010000010000000
9 0x000020ac 0x00e282ac 00000000000000000010000010101100 00000000111000101000001010101100
10 0x00002233 0x00e288b3 00000000000000000010001000110011 00000000111000101000100010110011
11 0x0000ffff 0x00efbfbf 00000000000000001111111111111111 00000000111011111011111110111111
12 0x00010000 0xf0908080 00000000000000010000000000000000 11110000100100001000000010000000
13 0x00010348 0xf0908d88 00000000000000010000001101001000 11110000100100001000110110001000
14 0x00022e13 0xf0a2b893 00000000000000100010111000010011 11110000101000101011100010010011
15 0x0010ffff 0xf48fbfbf 00000000000100001111111111111111 11110100100011111011111110111111
16 0x89abcdef 0xffffffff 10001001101010111100110111101111 11111111111111111111111111111111
All done!

-- program is finished running –
```

When your program is complete and you are satisfied it is right, or when you just ran out of time, put it in the Canvas Dropbox for Lab 2 part2. Use the filename Lab2_part2.asm. Your name should be in the file, and appear in the output.

The programs will be tested using MARS, and the TA will actually look at the program. It's important for your program to be correct, and to be readable. It should also be efficient, but an efficient wrong program is still wrong, and an unreadable program doesn't inspire confidence that you know what you are doing.

Hint:
Some of you may have an issue with printing the last line of array in Lab2 (0xffffffff). If you are using slt for implementing else-if structure, 0x89abcdef would be treated as a negative number. Simply replace slt with sltu, and your problem will be solved!