CPSC3300 Project 2

Due date: 11:59PM Oct. 27th

In this project you are required to build a basic mips32 assembler for well-formatted assembly inputs. Following are the useful instructions you need to follow:

1. An input file contains assembly code. All input instructions will be provided on separate lines with no spaces before.
2. Instructions are the instruction set (subset of mips) as mentioned in zybooks 4.1.
    a. The memory-reference instructions load word (lw) and store word (sw)
    b. The arithmetic-logical instructions add, sub, AND, OR, and slt
    c. The instructions branch equal (beq) and jump (j)
3. Instructions for translation are mentioned in zybooks 4.4.
4. Output should be a binary file, no preluded spacing. If there are errors in the input file, your program should print out which line in the input file has these errors.

Note: You should write this as binary, not as 0/1 ascii characters. Each language has a way to read/write raw binary to files. They will possibly not be human readable. https://inst.eecs.berkeley.edu/~cs61c/resources/MIPS_help.html is a good source for instruction format.

You may implement this in C/C++/Java/Python. We prefer C. Your program takes two command line arguments. The first one is the input file which contains the assembly code, and the second is the binary file to be generated. For example, let us assume we have the following assembly code in an input file `abc.s`

```
add $1, $2, $3
sub $2, $3, $4
and $5, $6, $40
bne $3, $4, -2
```

You will run your program, say, a.out, as

```
$ ./a.out abc.s output.dat
```

It should print out that in the input file, the third line has an error (register number is out of range), and the fourth line has an error (unsupported instruction).

We change it to the following code and now your program should be able to generate the intended binary file `output.dat`.

```
add $1, $2, $3
sub $2, $3, $4
and $5, $6, $4
beq $3, $4, -2
```

After the binary file is generated, you may use tools like `xxd` to check if you have generated the file correctly. For example,

`add $1, $2, $3` is translated to 00000000010000110000100000100000

`sub $2, $3, $4` is translated to 00000000011001000001000000100010

`and $5, $6, $4` is translated to 00000000110001000010100000100100

`beq $3, $4, -2` is translated to 00010000011001001111111111111110

And you can execute `xxd -b -c 4 output.dat` to get the following output

```
00000000: 00000000 01000011 00001000 00100000  .C.
00000004: 00000000 01100100 00010000 00100010  .d."
00000008: 00000000 11000100 00101000 00100100  ..($
0000000c: 00010000 01100100 11111111 11111110  .d..
```