Instructions

Deliverables

The deliverable for this assignment is a C program called `assembler.c`. Please commit this file to GitHub and submit your repository via Gradescope. Before you begin, please look over the `250_style_guide.pdf` document, available in the `#resources` channel on Slack. We will be enforcing the style guidelines more stringently from now on (you may skip the section on "Memory Errors and Leaks" for now).

How a C Program gets Compiled

Compiling a C program with `gcc` can be broken down into the following stages:

1. **Preprocessor:** expands `#include` and `#define` statements and the like
2. **Compilation:** converts C code into assembly instructions
3. **Assembler:** translates assembly instructions into binary machine code
4. **Linker:** adds in machine code from other sources (like library functions)

We can run these stages individually with the following commands:

```
gcc -E hello.c -o hello.i
gcc -S hello.i -o hello.s
gcc -c hello.s -o hello.o
gcc hello.o -o hello
```

After this sequence of commands, `hello.i` is the output of the preprocessor, `hello.s` is the output of the compilation-proper, `hello.o` is the output of the assembler, and `hello` is the output of the linker (the actual executable). If you open these in an editor, `hello.i` will still be somewhat recognizable: the code from `hello.c` is still in there, but it's mixed with lots of other stuff that came from `stdio.h`. `hello.s` should also be readable; it contains assembly code, which we'll learn lots more about later in the semester. On the other hand, `hello.o` and `hello` are probably completely incomprehensible, because these are binary files, and a text editor will attempt to render the ones and zeros as characters. We can view those ones and zeros with `xxd`:

```
xxd -b hello.o
```

but this still won't mean much to a human reader. However, this representation is how the computer's processor understands assembly instructions, so the `assembler` step that translates from human-readable assembly code to a machine-readable binary is a critical part of the compilation process.

Assembler

Your task in this assignment is to write an assembler for a much, much simpler language than C or x86 assembly. The language described below is the same one that is used by the CPU we will be building in a few weeks. With this assembler you will be able to translate programs you write in the CPU's assembly language into data that can be loaded into the CPU's program memory and actually run.

The language you are translating is specified in Table 1. Many of the details in the description (like registers and memory locations) may not make much sense yet; don't worry, you don't need to understand those parts until we build a CPU on homework 4.

Here are the specifications for this program:

- Your program should accept the name of an input file (containing assembly code) as the first command line argument, and the name of an output file (to which the machine code will be written) as the second.
- The program should translate each assembly code instruction into its `opcode` in hex, writing the results to the specified output file. The output file format should follow the specification described on this page:

  `http://www.cburch.com/logisim/docs/2.7/en/html/guide/mem/menu.html`

  Your output file should always begin with the line `v2.0 raw`, followed by lines that contain a single two-digit hexadecimal number. Hex digits include 0–9 and a–f, representing the number 0–15 in a single digit. We'll learn more about hex soon; feel free to jump ahead to section 4.1 in *Dive into Systems* if you're curious.
- You only need to assemble input files that are correctly formatted:
  - Instruction names are given entirely in capital letters.

Instruction	**Opcode**	**Description**
INPUT	`04`	Reads the data on the input line and stores it in register A.
OUTPUT	`07`	Copies the contents of register A to the OUT register.
JMP *val*	`0a`	Jumps to the location in memory that is specified by the byte that immediately follows the JMP instruction. In other words, the byte following the JMP instruction in the program is not another instruction, but a *value* that should be loaded into the PC next. Program execution will resume from this new PC value.
LOAD *val*	`0e`	Loads a value into register A. As with JMP, the value is specified by the byte that follows the LOAD instruction in memory.
INC	`14`	Increments the value of A by 1.
MOV	`17`	Copies the contents of register A to register B.
ADD	`1a`	Adds the values in A and B together, and stores the result in A.
HALT	`1d`	Halts execution—note that this can be accomplished by creating a self loop back to HALT, instead of returning control to the fetch state.
NOP	`1f`	No operation—a do-nothing operation (like `pass` in Python) that is simply a placeholder.

Table 1: The instruction set for the processor.

  - Each whitespace-separated token (word) is either a valid instruction or a 2-digit hexadecimal number.
  - Numbers only appear immediately following JMP or LOAD instructions, and every JMP or LOAD instruction is followed by a 2-digit hex number.

- Your program should gracefully handle the errors described below.

Error Messages

In the following cases, your program should exit with error code 1, and print the appropriate error message.

- If the user gave the wrong number of command-line arguments:

```
usage: ./assembler infile outfile
```

- If the source file could not be opened for reading (where `my_code.assembly` was the input file name):

  ```
  Error: unable to read source file: my_code.assembly.
  ```

- If the binary file could not be opened for writing (where `my_code.binary` was the output file name):

  ```
  Error: unable to write output file: my_code.binary.
  ```

- If the assembler encounters a token that does not match the specification (where "juMP" was the invalid token):

  ```
  $ ./assembler my_code.assembly my_code.binary
  Error: invalid token: juMP.
  ```

  Invalid tokens include otherwise-valid instructions that immediately follow JMP or LOAD instructions, or 2-digit hexadecimal numbers that don't.

Hints and Suggestions

- Declare an array of characters called `token` that will be used to store each word as it is read from the input file. You may assume that no tokens in the input file are more than 9 characters long, so a length-10 array will suffice. You can declare an array of ten characters with the following command:

  ```
  char token[10];
  ```

- Use `fscanf` with `%s` for reading the file. When reading into a `char` array, you don't need to use `&`. (We'll talk about why later in the semester.) For example:

  ```
  num_read = fscanf(infile, "%s", token);
  ```

- When validating hexadecimal numbers, you can compare characters by their ASCII value. For example, `'b' <= 'f'` evaluates to `true`.

- You can compare strings with the `strcmp` function from the `<string.h>` library. For example:

```
strcmp(token, "INPUT")
```

Note that `strcmp` returns 0 if the strings match, and non-zero if the strings differ!

- You can test your assembler by running it on the `incrementer.assembly` and `tripple.assembly` programs, and comparing to the correct outputs with `diff`. For example:

  ```
  ./assembler incrementer.assembly incrementer.out
  diff incrementer.binary incrementer.out
  ```

  When `diff` prints nothing, it means the two files were identical. Otherwise, it tells you which lines differ.