CS 421 **DUE: Wednesday, March 2 at 8:00 AM**
Spring 2022
Project 1
2/7/2022 **100 points**

In this project, you will write a C program to:

1. read a binary program file for an imaginary machine architecture,
2. translate this machine code into an assembly language program, to be written to a corresponding output text file, and
3. simulate the execution of the program on this imaginary machine, during which you must check for certain error conditions.

You may work with a partner on this project.

Our imaginary machine, the *Microputer*, has a very limited memory, register set and instruction set. Here are its details:

1. The size of the word for this architecture is 2 bytes.
2. It has 16 one-byte size registers, `R0` through `R15`, denoted by binary codes `0000` through `1111` respectively. The register contents are always unsigned, so they can store from `0x00` through `0xFF` (0-255 in decimal).
3. It has a Program Counter (`PC`) and Instruction Register (`IR`), both of which are two bytes (one word) in size.
4. It has 32 bytes of memory used ONLY for code. So, essentially, 16 instructions can be stored in memory. These 32 bytes are numbered `00000` through `11111`, in binary.
5. Since there is no memory available for data, all operations are performed through registers. There are two special (magical) instructions that allow data to be read from the standard input and written to the standard output. All instructions are one word in size. The complete instruction set architecture is given in the table below. Each instruction starts with a three-bit opcode (as the highest order bits), followed by the specific operands needed for that instruction. The number of bits for each component in an instruction format is specified in parentheses.

Instruction	Opcode	Format	Details
LDI	000	000(3) | Ri(4) | Imm(8) | Unused(1)	Loads immediate data Imm into Ri
ADD	001	001(3) | Ri(4) | Rj(4) | Rk(4) | Unused(1)	Adds Ri and Rj with result in Rk
AND	010	010(3) | Ri(4) | Rj(4) | Rk(4) | Unused(1)	ANDs Ri and Rj with result in Rk
OR	011	011(3) | Ri(4) | Rj(4) | Rk(4) | Unused(1)	ORs Ri and Rj with result in Rk
XOR	100	100(3) | Ri(4) | Rj(4) | Rk(4) | Unused(1)	XORs Ri and Rj with result in Rk
PRT	101	101(3) | Ri(4) | Unused(9)	Prints Ri to the standard output
RDD	110	110(3) | Ri(4) | Unused(9)	Reads lower 8 bits from standard input into Ri, clearing the standard input afterwards
BLE	111	111(3) | Ri(4) | Rj(4) | Addr(5)	Jumps to Addr if Ri is at most Rj. Exception results if Addr is not on a word boundary

Your program should do the following:

1. Accept an input file name and output file name as command line arguments. So, start your `main` function as:

   ```
   int main(int argc, char* argv[])
   {
   ```

   Then, you can check `argc` for the number of command line arguments and access each one using `argv`, e.g., `argv[...]`.

2. Read the binary instructions in the input file, create the output text file, translate these binary instructions into their equivalent assembly language instructions, and write these assembly instructions to the output file. So, for example, the bits (in hexadecimal) `0x2024` will be translated into the instruction `ADD R0 R1 R2`. Write each translated instruction to the output file, on a new line. When you write the instruction, prepend, before the instruction, the memory address where this instruction will be stored. e.g. `4: ADD R0 R1 R2`. The first instruction will always start at memory address 0. All memory addresses and immediate data are displayed in decimal in the assembly language file.

3. Simulate the execution of the complete program on the Microputer. This should be a faithful simulation. You should model the memory locations, the registers, the `PC` and the `IR`, and then simulate the program execution by changing the location of the `PC`, and decoding and executing the instruction pointed to by the `PC` until the program is complete. As part of your simulation, you must have at least the following functions:

   - ```
     void decode_instruction(char instruction[],
                             char* op,
                             char* ri,
                             char* rj,
                             char* rk,
                             short* imm,
                             char* addr)
     ```

     The idea is that `decode_instruction` function takes an instruction as input through the `instruction` parameter and returns several outputs through the remaining parameters.

   - `int execute_program(char program[])`

   Furthermore, all exceptions must be caught during runtime.

Note the following:

- All inputs from the standard input, and outputs to the standard output, are to be read/displayed in decimal.
     ```

- All inputs and outputs are restricted to one byte unsigned values. Any overflow should be truncated to the lower level bits.
- Code your program to avoid redundant blocks of code, as far as possible.
- Your program should be robust to user error, modular, well documented and cleanly designed.
- You should print out any exceptions to the standard output and terminate the program. Examples of exceptions might include invalid instruction formats and illegal memory access (out of bounds, not word boundaries, etc.).
- You should have an accompanying makefile that will build the executable, and do necessary cleanup. Note that your code must compile and execute without any warnings or errors on our Linux lab computers. Mimic the simple Makefile example in the notes.
- Three example files `inp1.dat`, `inp2.dat` and `inp3.dat`, and their respective translated assembly language files, `inp1.asm`, `inp2.asm` and `inp3.asm`, are provided for your reference.
  - For inputs of `5` and `10` on the standard input (all inputs in decimal), the value printed on the standard output for program `inp1.dat` is `15` (all outputs in decimal).
  - For inputs of `2` and `6` on the standard input (all inputs in decimal), the value printed on the standard output for program `inp2.dat` is `12` (all outputs in decimal).
  - For an input of `10` on the standard input (all inputs in decimal), the value printed on the standard output for program `inp3.dat` is `1424116` (all outputs in decimal).

Remote login to Linux: Since your code must compile and execute without any warnings or errors on our Linux lab computers, I recommend that you do your programming in the Linux environment. To open a Linux session remotely (e.g., from a Windows machine off campus), use the program called `PuTTY` and connect to `ssh.acs.uwosh.edu` on port `1022`, using the SSH protocol. If a dialog box containing a question about a host key comes up, then click "yes" to continue logging in. Note that it is also possible to use SSH at the Terminal window on Windows 10 and 11 (see the instructions on Canvas).

This is what comes up after I log in remotely from the terminal window on my Mac (instead of using `PuTTY`:

```
$ ssh -p 1022 summerss@ssh.acs.uwosh.edu
Warning: the ECDSA host key for '[ssh.acs.uwosh.edu]:1022' differs from the key
for the IP address '[141.233.181.105]:1022'
Offending key for IP in /Users/summerss/.ssh/known_hosts:4
Matching host key in /Users/summerss/.ssh/known_hosts:11
Are you sure you want to continue connecting (yes/no)? yes
summerss@ssh.acs.uwosh.edu's password:
```

```
Last login: Mon Jan 17 10:01:48 2022 from 23.112.236.203
$
```

Note that your password will not be printed to the screen but you still need to type it in correctly. When you are done, type the command `exit` or `logout` to close your remote session.

File transfer to/from Linux: If you would like to transfer files between your local machine and your Linux account, then use a program like WinSCP or FileZilla to connect to `ssh.acs.uwosh.edu` on port `1022` using the SFTP protocol. If you are using FileZilla, then make sure you connect through the Site Manager in the File menu. Note that the Quickconnect button may not let you change the protocol to SFTP, in which case your connection will fail.

SFTP is built into terminal on a Mac, so you could connect to your Linux account using the following command:

```
sftp -P 1022 youruserid@ssh.acs.uwosh.edu
```

Once connected, `ls`, `cd`, `pwd` still all work, so navigate to the directory then use `get <filename>` to retrieve it Note that `sftp` will retrieve files to your current working directory in terminal, so navigate to the appropriate directory before launching `sftp`. Unlike `ssh`, `sftp` uses `-P` to designate the port, not `-p`.

Formatting guidelines: Please ensure that you satisfy all of the following formatting guidelines:

1. At the very top of a `.c` file, put a comment block that looks something like:

   ```
   /*****************************************************************
    * Description: ??
    * Author: ??
    * Date: ??
    *****************************************************************/
   ```

2. Give your variables meaningful names. Non-constant variables should have names that start with a lowercase letter. Names for constant values should be in ALL CAPS, e.g., have

   `#define DECEMBER 31` instead of `#define December 31`

   This applies to `const` variables as well. Do not use single-character variable names except perhaps for loop variables.

3. No "magic numbers", i.e., do not have literal values other than `-1`, `0`, `1` or `2` in your code (use the `#define` pre-processor directive or `const`s).

4. Always put exactly one space before AND after binary operators, e.g., have:

`area = 3.14 * rad * rad;` instead of... `area=3.14*rad*rad;`

I have this rule just to make it easier for me to read your code.

5. Always put exactly one space after each comma in a list, e.g., have

   `int first, second, third;` instead of... `int first,second,third;`

6. Have a consistent indentation scheme throughout your program, e.g. have

```
if (x1 > x2) {
  if (x2 > x3) {
    if (x1 > x3) {
      // Good!
    }
  }
}
```

instead of...

```
if (x1 > x2)
  {
        if (x2 > x3) {
   if (x1 > x3)
     {
      // Bad!
  }
    }
}
```

7. When a line gets too long (more than 78 columns), break it at a reasonable place. Use indentation to make it obvious which lines are continuations, e.g., have:

```
printf("This is a message that is broken into two ");
printf("parts because it is too long.");
```

Submission: To complete this assignment, simply submit your C code file(s), along with the make file, zipped up in a single folder, to the Canvas submission folder **P1** by the deadline specified.