CS 4380

PROJECT 1

2 / 5 / 2022

1. Details

Please create the following files—names are in bold:

1. ***Proj1.asm*** *that contains your assembly language file.*
2. *Your source code in python.*
3. Any other files required for your program.
4. (optional) ***readme.txt*** that contains known bugs/problems in your other files or changes made to previous submissions.
5. Submit your project in a zip folder named ***<your_name>_p1.zip***.

2. Summary & Requirements

For this project you will implement a virtual machine that can perform arithmetical operations on values stored in memory.

You will write the output and bytecode for the assembler to both memory and a file; the easiest way to do this is by using Memory-Mapped I/O (MMIO)

You don't need to implement the stack yet. We won't use a heap for this course, so you don't need to implement the heap either.

Your program ***must***:

1. accept an assembly file or bytecode file as a command line parameter.
2. automatically terminate.
3. not crash when run with valid input.
4. have syntax and semantics that match instructions.
5. not have any additional assembly instructions.
6. suppress all debug output in the final versions of all files.

3. Program Part A: Memory

Place integers and characters into memory.

1. Place the following list of integers in memory:
    a. A = (1, 2, 3, 4, 5, 6)
    b. B = (300, 150, 50, 20, 10, 5)
    c. C = (500, 2, 5, 10)

SYNTAX: A = (1, 2, 3, …) should look like as follows in your .asm file for this project:

```
A1    .INT    1
A2    .INT    2
A3    .INT    3
…
```

2. Place your full name into memory (“Last, First”):
    a. Name = (‘L’, ‘a’, ‘s’, ‘t’, ‘,’, ‘\s’, ‘F’, ‘i’, ‘r’, ‘s’, ‘t’)
        i. Place each letter of your name, capitalizing the first letter of each name.
        ii. Include a comma and a space between names.

SYNTAX: Name = (G, r, e, g) should look like as follows in your .asm file for this project:

```
G    .BYT    ‘G’
r    .BYT    ‘r’
e    .BYT    ‘e’
g    .BYT    ‘g’
```

NOTE: If your name(s) use the same upper or lower case letter more than once you should only make that letter once. Bobby would only need to place Boby into memory because the lower case b’s are the same character.

4. Program Part B: Instruction Sets

Run calculations and store/print integers and characters to memory/file.

1. Print your full name “Last, First”.

   *Print a new line **twice**. First, to end the current line and start on the next line. Next to add a blank line between parts of this assignment. For example, it should look like this:*

   Last, First

   450, 500, …

2. Starting with the first two values in B:
   a. Add the first two values together (300 + 150 = 450) then store/print the result.
   b. For every other value add it to the previous result (450 + 50 = 500) then store/print the result.

   Print a new line twice.

3. Starting with the first two values in A:
   a. Multiply the first two values together (1 * 2 = 2) then store/print the result.
   b. For every other value multiply it by the previous result (2 * 3 = 6) then store/print the result.

   Print a new line twice

4. ***Take the final result from step 3*** and divide it by each value in B. Store/print each result.
   a. *You will round down (use integer division for this part).* Do not store floats/doubles!

   Print a new line twice

5. ***Take the final result from step 4*** and subtract each value in C. Store/print each result.

5. Grading.

Rubric: Project 1 (Spring 2022), Project 2 (Fall 2022)				
Criteria	**Ratings**			**Pts**
Assembler First Pass: Read and tokenize .asm files	**5 to >4.0 pts** **Full Marks** Read and tokenize .asm files	**4 to >0.0 pts** **Obtuse Code (or only partially implemented/functional)** 1. Code uses arrays of arrays for processing the tokens. However, the logic is much cleaner (and more maintainable if you process the tokens one line at a time. For example, instead of lines[row][column], save rows (unparsed data) as lines, but then process each line: line = lines[row], token = tokens[column] (assumes you tokenize the line and save the tokens in a list variable named tokens). 2. Etc.	**0 pts** **Not implemented**	5 pts
Assembler First Pass: Error Checking and Comments	**10 pts** **Full** Error Checking and Provides Feedback	**5 pts** **Partial** Error Checking, but no Feedback	**0 pts** **No Marks** Neither Error Checking nor Feedback	10 pts
Assembler First Pass: Generates Symbol Table Correctly	**20 to >19.0 pts** **Full Marks** Generates Symbol Table Correctly	**19 to >0.0 pts** **Partial**	**0 pts** **No Marks**	20 pts
Assembler Second Pass: Zeroes Out Unassigned Directives and Unused Operands	**5 pts** **Full Marks** Zeroes Out Unassigned Directives AND Unused Operands	**2.5 pts** **Partial** Zeroes Out Unassigned Directives OR Unused Operands	**0 pts** **No Marks** Doesn't Zero Out Unused Directives or Operands	5 pts

Rubric: Project 1 (Spring 2022), Project 2 (Fall 2022)				
Criteria	**Ratings**			**Pts**
Assembler Second Pass: Verifies Labels are Defined in Symbol Table (and Warns If Undefined)	**10 pts** **Full Marks** Verifies Labels are Defined in Symbol Table (and Warns If Undefined)	**5 pts** **Partial** Verifies Labels are Defined in Symbol Table, but Doesn't Warn If Undefined	**0 pts** **No Marks** Doesn't Check for Invalid Labels	10 pts
Assembler Second Pass: Generates the correct bytecode	**20 to >19.0 pts** **Full Marks** Bytecode exactly matches.	**19 to >0.0 pts** **Partial** One or more of the following: 1. Isn't setting first four bytes as initial PC or isn't setting it to the right value 2. First instruction isn't JMP MAIN 3. Instruction(s) work improperly 4. Etc.	**0 pts** **No Marks** Doesn't Check for Invalid Labels	20 pts
Virtual Machine: VM properly executes the .bin file without hanging.	**10 to >9.0 pts** **Full Marks** Output Exactly Matches	**9 to >0.0 pts** **Partial** One or more of the following: 1. Doesn't Generate Exact Output 2. Doesn't Generate Output when .bin provided (only when .asm) 3. Doesn't use R3 (like instructions to implement specifies) as the TRP register for TRP 1-4 4. 4. Uses STR, LDR when it should be STB, LDB 5. Uses STB, LDB when it should be STR, LDR 6. Etc.	**0 pts** **No Marks**	10 pts
Virtual Machine: Instructions are implemented properly	**10 to >9.5 pts** **Full Marks** Instructions are implemented properly	**9.5 to >0.0 pts** **Partial** Most instructions are implemented properly. For example: 1. LDB/STB uses 4 bytes (Instead of 1) 2. Etc.	**0 pts** **No Marks**	10 pts

Rubric: Project 1 (Spring 2022), Project 2 (Fall 2022)		
Criteria	**Ratings**	**Pts**
Assembly File: Assembly File's Code and Formatting is correct	**10 to >9.0 pts** **Full Marks** Assembly File is Formatted Correctly (Exactly matches the Syntax Document) **9 to >0.0 pts** **Partial** Assembly File is Partially Formatted Correctly (Mostly Matches the Syntax Document) **0 pts** **No Marks**	10 pts

The grade for this project is 100 points.

6. Hints.

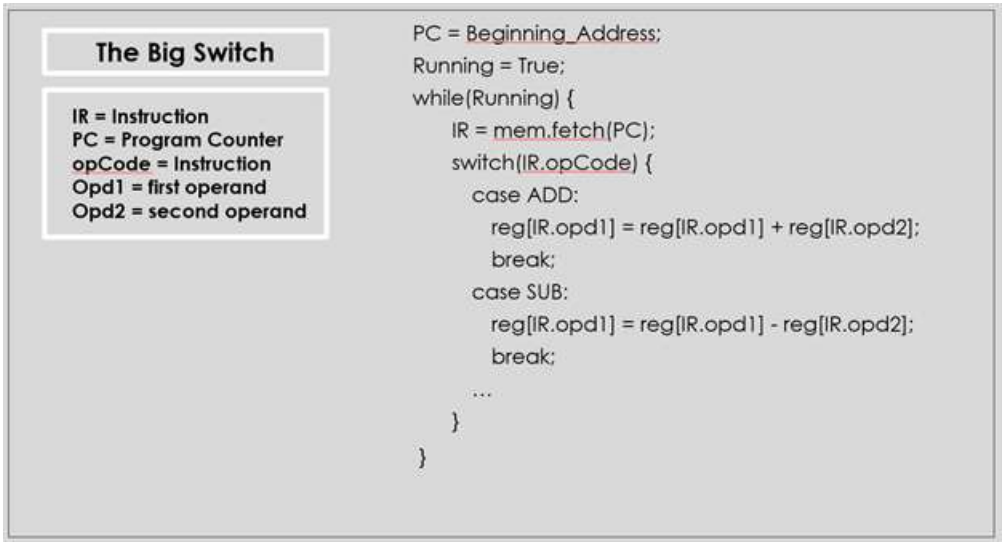
The following pages contain insight on how to complete this project.

1. Assembler Pass 1
    a. Read .asm file
    b. Group chars into Tokens. Recommend caching the tokens you parse, so they can be used for both the first and second pass. (Otherwise, you'll have to read the .asm file twice.)
    c. Check Syntax of Instructions.
        i. Directives:
            - `[Label]  .INT  [Integer-Constant]`
            - `[Label]  .BYT  [Byte-Constant]`
        ii. Instructions:
            - `[Label]  Opcode  [Operand1][,  Operand2]`
                - *Label is optional*
                - *Opcode is an instruction's mnemonic, such as ADD, SUB, MUL, etc.*
                - *Project 4 contains a zero-operand instruction, Projects 1-4 have some single operand instructions. Most instructions are two operand. Therefore, operand1 and operand2 are both optional.*
                - *Operands are one of:*
                    i. Registers
                    ii. Labels
                    iii. Immediate values
                    iv. Not used
        iii. Store defined Labels (and their offsets) in the Symbol Table.

2. Assembler Pass 2

    a. Check that the Referenced Labels are defined in the Symbol Table
        i. LDR R1, NUTMEG What is the address for NUTMEG?
        ii. JMP NEXT What is the address for NEXT?
            - *Sometimes a label is used BEFORE it is defined! (This is why we can't generate bytecode until the second pass.)*
    b. Generate Byte Code
        i. .BYT is one unsigned byte (0x00 – 0xFF), if no default value assign 0
        ii. .INT are signed and four bytes (0x80000000-0x7FFFFFFF), if no default value assign 0
        iii. Instructions – regardless of number of operands – are 12 bytes.
            1. Operation is first int
            2. Op1 is second int (if unused default to 0)
            3. Op2 is the third int (if unused default to 0)
        iv. Place the value of the program counter at the start of your program at the start of your bytecode. (ie. If your program starts at PC = 64, place 64 into the first four bytes)
            1. Offset 0 of the bytecode is the address of the initial PC
            2. This implies, since the initial PC is an int, that the first directive (or first instruction, if no directives) should start at offset 4, not offset 0!

3. The Big Switch



4. Advice:

    a. Less is more. Write succinct code!
        i. Don't implement unrequired methods.
        ii. Work within the scope of the project.

    b. Work in small increments!
        i. Slow and steady wins the race.

    c. This project requires a two-pass assembler!
        i. You can do it all in one pass for project one, but not having a two-pass assembler in projects 2-4 makes it unnecessarily difficult, if not impossible for the assembler to generate the correct bytecode.

    d. Include the following addressing modes:
        i. Direct EA = Address
        ii. Register EA = Register

    e. Print using the correct TRP!
        i. TRP 1 is for integers and TRP 3 is for characters.

    f. Support Comments!
        i. Characters after a semi-colon in your .asm file should be ignored as a comment.

    g. Don't test your program to succeed, test it to fail!
        i. Only when you fail to fail can you truly succeed.
        ii. Use TDD. Code a little, test a little (fail faster)
        iii. Alternate between your assembler and virtual machine to catch bugs early!

    h. Discuss your program with others to help both of you succeed!
        i. ***Don't*** share your source code for your Assembler or VM
        ii. ***Don't*** help anyone else write or fix code for their Assembler or VM
        iii. ***Don't*** plagiarize another student's .asm file!
        iv. ***Do*** share methodology, concepts, and your thought process
        v. You may share your .asm or .bin file with other classmates, but ***ONLY*** *to confirm your Assembler and VM are working correctly (according to the spec*)