Project Description

You are required to develop an emulator for an assembler. The assembler will execute assembly code given the following architecture and assembly language

Architecture:

1- Your system has the following general purpose registers that can store integer values: REG_A, REG_B, REG_C, REG_D, REG_E, REG_F, REG_G, REG_H

2- The system also has a specialized register called ACC which means accumulator

3- The system had input registers INPR1,INPR2,INPR3,INPR4,INPR5

4- The system had output registers OUTR1, OUTR2, OUTR3, OUTR4, OUTR5

5- And a memory (an array) of 64 locations. Each location can store an integer value.

The memory image is loaded from a text file that includes 64 integer values. So you will need to read the text file and store the 64 integer values into the memory (array).

Now you will need to read the code file. It is a file that has assembly code and execute its instructions one by one. Your assembler should throw an error if there is an unknown command or a mistake in the instruction.

Addressing:

The memory address 30 is written as follows: $30 whereas the value 30 is written as #30 line 30 is written like this L30, In other words if you want to say A equals the content of memory location 30 you would say A=$30 but if you want to say A equals value 30 you would say A=#30.If you want to address an instruction at line 40 then you would say L40.

Commands

The assembly language you are required to implement has the following operations:

1. **MOV operation**

The MOV command moves values from a location to another. It works like an assignment statement.

Examples:

a. MOV REG_A,REG_B means REG_A=REG_B: assign the value of REG_B to REG_A..

b. MOV REG_A,$50 means assign the value of memory location 50 to REG_A.
c. MOV $50, $30 means assign the value of location 30 to location 50.
d. MOV $30, REG_A means assign the value of register REG_A to memory location 30.
e. MOV REG_A, #30 means assign the value of register REG_A with value 30.
f. MOV $30,#30 means assign the value of memory location 30 the value 30.
g. MOV REG_A,INPR1 means assign the value of input register INPR1 to REG_A. Here as if you are reading a value from the user.
h. MOV OUTR1, REG_A means assign the value of REG_A to output register OUTR1. Here as if you are writing a value to an output peripheral.

2. <u>CMP operation</u>

CMP command compares two registers and/or memory locations and store the result of comparison into ACC. Lets say we are executing CMP REG_A, REG_B, If REG_A and REG_B are equal. ACC will be 0. If REG_A> REG_B then ACC will be 1. If REG_A< REG_B then ACC will be -1. Examples:

a) CMP REG_A, REG_B compare the value of register REG_A to the value of register REG_B. The comparison result is put in ACC.
b) CMP REG_A,$40 compare the value of register REG_A to the value of Memory location 40. The comparison result is put in ACC.
c) CMP REG_A.#40 compares the value of register REG_A with the value 40. The comparison result is put in ACC.

3. <u>BRH operation</u>

BRH command moves the execution from a place to another. In other words, if you are executing instruction number 15 now and the instruction id BRH L40 then the next instruction will not be instruction number 16, it will actually be instruction number 40.
Example:

a) BRH L10 : go to instruction at line 10
b) BRH foo: go to instruction named foo.

4. <u>BEQ operation</u>

Just like the BRH command except it is dictated by the value of register ACC (accumulator).
Example:

a) BEQ L10 : go to instruction at line 10 if ACC is 0
b) BEQ FOO : go to instruction named FOO if ACC is 0
c) BEQ L20 : go to instruction at line 20 if ACC is 0

5. <u>OUT operation</u>

OUT command is a command that shows the values of certain registers or memory location

Examples:
OUT REG_A : prints the value of REG_A on screen.
OUT REG_A, REG_B : prints the values of REG_A, REG_B on screen.
OUT ALLR : prints the values of all registers on screen
OUT $20: prints the value of memory location 20 on screen.
OUT $20,$30 prints the values of memory locations 20 and 30 on screen.
OUT ALLM prints the values of all memory locations on screen.
OUT ALL prints all values of all registers and memory on screen.
OUT INPR1 prints the value of input register INPR1 on screen.
OUT OUTR1 prints the value of output register OUTR1 on screen.

6. ADD operation

ADD operation always adds a value in a register or memory location to accumulator and stores the value in the accumulator.

Example:

a) ADD REG_A: this is equivalent to ACC=ACC+REG_A
b) ADD $30: this is equivalent to ACC=ACC+ memory location 30
c) ADD #30: this is equivalent to ACC=ACC+the value 30

So what if you want to get the summation of REG_A and REG_B?! You would do it in two steps like this:

```
MOV ACC,REG_A
ADD REG_B
```

Now what if you want to perform REG_C=REG_A+REG_B You would do it as follows:

```
MOV ACC, REG_A
ADD REG_B
MOV REG_C,ACC
```

7. SUB operation

SUB operation always subtracts a value in a register or memory location to accumulator and stores the value in the accumulator.

Example:

a) SUB REG_A: this is equivalent to ACC=ACC-REG_A
b) SUB $30: this is equivalent to ACC=ACC- memory location 30
c) SUB #30: this is equivalent to ACC=ACC-30

So what if you want to get the subtraction of REG_A and REG_B?! You would do it in two steps like this:

```
MOV ACC,REG_A
SUB REG_B
```

Now what if you want to perform REG_C=REG_A-REG_B You would do it as follows:

```
MOV ACC, REG_A
SUB REG_B
MOV REG_C,ACC
```

8. MLT operation

MLT operation always multiplies a value in a register or memory location with the accumulator and stores the value in the accumulator.

Example:

a) MLT REG_A: this is equivalent to ACC=ACC * REG_A
b) MLT $30: this is equivalent to ACC=ACC * memory location 30
c) MLT #30: this is equivalent to ACC=ACC * 30

So what if you want to get the multiplication of REG_A with REG_B?! You would do it in two steps like this:

```
MOV ACC,REG_A
MLT REG_B
```

Now what if you want to perform REG_C=REG_A * REG_B You would do it as follows:

```
MOV ACC, REG_A
MLT REG_B
MOV REG_C,ACC
```

9. DIV operation

DIV operation always divides the accumulator with a value in a register or memory location and stores the value in the accumulator.

Example:

a) DIV REG_A: this is equivalent to ACC=ACC / REG_A
b) DIV $30: this is equivalent to ACC=ACC / memory location 30

c) DIV #30: this is equivalent to ACC=ACC / 30

So what if you want to get the division of REG_A with REG_B?! You would do it in two steps like this:

```
MOV ACC,REG_A
DIV REG_B
```

Now what if you want to perform REG_C=REG_A / REG_B You would do it as follows:

```
MOV ACC, REG_A
DIV REG_B
MOV REG_C,ACC
```

MEMORY FILE EXAMPLE

A memory file is a text file that would have 64 different integer values like this:

```
5
7
8
3
56
1000
-345
234
9876
-23763
-15
767
8274
-3200
5646
.
.
.
```

CODE FILE EXAMPLE

Assume you want to write this code in the proposed assembly language:

```
Sum=0;
for (int i=1;i<=10;i++)
        SUM=SUM+I;
```

```
                MOV REG_A,#0
                MOV ACC,#0
                MOV REG_C,#0
LOOP :    MOV ACC, #1
          ADD REG_A
                MOV REG_A,ACC
ADD REG_C
                MOV REG_C,ACC
                CMP REG_A,#10
          BEQ END
                BRH LOOP
END:    OUT REG_C
```

SYMBOL TABLE

As seen in the previous code, you can actually give labels to instruction like the label "LOOP". While labels are convenient for programmers, it decreases the efficiency of executing a program. So you need to implement a symbol table that will translate the label into an instruction address. Every time you want to execute the code and you encounter a label you will look at the symbol table and jump directly to the instruction address instead of traversing the whole list of statements looking for the label.