Assignment Done By:

Mohit Sharma(2019CS10372)

Vivek Choudhary(2019CS10413)

Goal: Develop C++ program that interprets a subset of MIPS assembly language instructions.

1) Take a file input which contains the MIPS assembly language code.
2) Parse the input file and check for syntax errors and make space for instructions in memory and store them in desired format.
3) Execution of program and displaying content of register file after every instruction is executed.
4) At the end show the statistics of number of clock cycles and number of times each instruction is executed.

Approach:

* Main idea is that we have stored all given code in newly designed data structure("Instruction") and later at the time of each execution we used stored information from it.

* For memory part: We have broken the memory (2^20 bytes) in two parts first part for instruction and second for data and this division is done according to "input.txt". The second part for data section is made to start from position just next to instruction memory. So if there are 5 instructions in the program then memory till 19 (0 to 19=20(5*4)) is for instruction memory and after that it is for data section. And the memory in data section is dealt in 0-indexed manner. That is to access memory 20, numeric number 0 (not 1) is needed.

* Basically, we made a “register_map” (Hash map) to store the name of register and its corresponding position in “register_File” (where we will store the data of each registers on every execution).

* There is a structure named "Instruction" which stores the command to be executed and the parameters of that command (any particular register or the constant value).

* Read the given txt file. If error in syntax then report else tokenize all line one by one and store in "Instruction" structure.

* So, for every register or label it has checked whether it is valid register/label or not valid then perform operation else give error on which line this error is present. Same for all given instruction provided in assignment.

*So now we are ready with our whole instruction part. Good time to start executing whole program.

*It will start doing execution from line 1 according to the information stored in “Instructions”.

*For jump instructions like "j label" it will find the position for that label from “label_map” and continue from that position. Similar execution will happen in case of “bne” and “beq” instructions. These steps will go on until we find any error or we reach the end of program. After every instruction, value of registers is printed in hex format.

*At last, we have printed out the statistics of count each execution and total clock cycles.

* Stack pointer ($sp) is given an initial value of 2^20.

* Only integer values are considered to be stored in data section of memory. So to store any integer, it is converted to char *(having 4 bytes space) and then stored in memory. Similarly to retrieve and value back, 4 bytes space is taken from the memory at specified address and then it is converted to integer.

Boundary Cases:

Handled any kind of syntax error. (Considering only instructions given in assignment pdf as valid.)

Checked for overflows during calculation.

Checked for attempts to access any invalid or out of bounds position in the memory.

Infinite loop is allowed to continue to process.

To Run:

Put the test cases in a file named “input.txt” in the same folder in which the main.cpp file is present.

Compile the file with “g++ -o main main.cpp” command. Executable “main.exe” is created

Run the executable by typing “main.exe” in command line.

Output is printed in the console.

Memory Representation Of Different Instructions:

1) **add, sub, mul, slt**:

First 4 bits for instruction.

Then 5 bits for each register (total 3 registers are there).

2) **beq, bne**:

First 4 bits for instruction.

Then 5 bits each for first and second register arguments.

Then 18 bits for signed integer (storing instruction address to jump at).

3) **lw, sw**:

First 4 bits for instruction.

Then 5 bits each for first and second register arguments. (like in: lw $t0, 4($sp), 5 bits for $t0 and 5 for $sp, if second register is not there then it is taken as $zero (as in: lw $t0, 100)).

Then 18 bits for numerical part (4 in example 1 and 100 in example 2.

4) **j**:

First 4 bits for instruction.

Then next 28 bits for signed integer (that stores address of instruction to jump to)

5) **addi**:

First 4 bits to store instruction.

Then next 5 bits each for first and second register arguments.

Then next 18 bits to store signed integer value (the constant value).