COL216

ASSIGNMENT – 3

ABHINAV SINGHAL (2019CS50768) ABHISHEK KUMAR (2019CS10458)

Problem Statement: Develop an interpreter in C++ for a subset of MIPS assembly language instructions.

Major Design Considerations:

A) Input / Output: -

The input file to be executed is taken as a command line argument. The output file can also be given as the second argument in the command line (else by default a file called "output.txt" will be created). The output file consists of register values after each instruction.

Any errors will be displayed on the terminal. If syntax errors are present, all the syntax errors will be listed on the display. If there are no syntax errors but runtime errors are there, then the execution will stop at the first runtime error encountered and the error message will be duly displayed onto the terminal window.

B) Implementation: - There are essentially two main phases in our MIPS interpreter –

- **Parsing:**
  - The input file is opened and read line by line.
  - A struct called instruction is used to store an input line which has basically two fields: the operation type and the register ids (maximum 3) used in the line.
  - Bi-directional Hash-Maps have been used to store the labels used (the instruction number and the label identifier)
  - Once a line has been tokenized to get an object of type instruction, it is pushed into the memory allocated for instructions.
  - After entire file has been parsed, we begin execution if no syntax errors were raised during parsing.
- **Semantic Analysis/Execution:**
  - An identifier called PC (Program Counter) is maintained which keeps track of which address is to be executed in the current clock cycle.
  - Depending upon the kind of instruction, accordingly the registers are manipulated and if the statements are either beq, bne or j type then pc is set to the appropriate

instruction address, else it is simply incremented to the next instruction to be executed.

- This takes place till PC reaches the last instruction and the simulator executes the instruction.

- After each instruction is executed the state of all the registers is printed on the terminal

Syntactical and Runtime Errors are handled at appropriate places as and when required. If the execution was successful, relevant statistics such as number of clock cycles and number of times each instruction was executed is printed for the user's reference.

C) **Test-Cases:** - We have tested our implementation using multiple test cases, some of them shown below.

- **Test-Case:** (empty file)

- **Test-Case : (**Valid Test Case**)**

```
addi $t0, $t0, 50
addi $s0, $s0, 10
addi $t8, $t8, 5
lw $s1, 100(500)
sw $t0, 500(500)

slt $t9,$t0,$t2
beq $t9, $t1, loop
beq $t9, $t1, another

loop:
  add $t6, $t0, $s0
  addi $t6,$t6, 10
  add $t1, $t1, $t6
 beq $t8, $zero, end
sub $t8, $t8, $t2
 j loop

another:
   add $t0,$t0,$s0
   slt $t9,$t0, $t3
   beq $t9, $zero, loop
   beq $t9, $zero, another

end:
   sw $t1, 500
   lw $s5, 500
```

- **Test-Case:** (Infinite Loop: but no Runtime errors)

```
begin:
   addi $1,$1, 1
   j begin
```

- **Test-Case : (**Error: Label Not Found**)**

```
start: addi $t0, $t0, 50
addi $s0, $s0, 10

bne $t0,$t0,$s0
beq $t0,$0, end
j start

label:
   sw $s0, 1000
```

- **Test-Case :**

```
addi $t1,$t1,10
sw $t1,16
lw $t2,16
sw $t1,16
```

NOTE: Empty lines are ignored and are not treated as instructions.

~THANK YOU~