Computer Architecture COL-216
MIPS Compiler - Assignment 3

Aman Gupta,2019CS10673
Arpit Chauhan,2019CS10332
13/03/2021

Description

In this assignment, we have written a C++ program that reads a MIPS assembly language program as input and executes it as per MIPS ISA by maintaining internal data structures representing processor components such as Register File and Memory.

Input / Output

- The input to the program will be a text file containing MIPS instructions. Each instruction will be in a new line.
- The allowed instructions are: **add, sub, mul, beq, bne, slt, j, lw, sw, and addi.**
- If any instruction does not follow MIPS convention, then a corresponding error is thrown along with the line number.
- If all the instructions are syntactically correct, then the program starts executing the operations starting from the first instruction.
- The labels must also follow MIPS convention i.e. must start with a letter, maybe followed by letters, numbers, or "_", must be terminated by ":" and should be unique.
- While executing the instructions, the register file contents (32 register values in Hexadecimal format) are printed after executing each instruction.
- After all the instructions are executed, relevant statistics such as Number of clock cycles, Instruction memory used, Data memory used, and Number of times each instruction was executed, are printed.

Approach / Algorithm

- The Register File contains 32 integers registers , namely $zero and $r1 - $r31 . All of them are initialized to 0. The $zero is a reference register and is non-mutable.
- We have allotted 2^{19} Bytes of memory each for **Instructions** and **Data,** summing up to a total of 2^{20} Bytes. We have taken the memory to be **word addressable**.
- Each instruction occupies 4 bytes and is executed in one clock cycle.
- The C++ program reads the input text file line by line and checks the syntactic correctness of the instructions using RegEx and various other validators. If the instruction is valid, then the instruction type and its parameters are stored in a map(params) indexed by the line number. After all the instructions are loaded into the program, the execution phase starts.

- We maintain an iterator named **Program Counter**: pc, that always contains the address of the next instruction to be executed. This counter is updated after each instruction execution.
- The program ends when pc exceeds the address of the last instruction. Relevant statistics of the entire MIPS program are then printed.

ERROR HANDLING

All errors are reported with the appropriate line number and reason for the user to debug.

1. **Syntax Error**
   A syntax error is thrown in the following cases:
   - If an invalid instruction is provided to the code.
   - If the operands of any instruction do not follow the convention specified by MIPS ISA.
   - If the integer value of an I-type immediate value is larger than 2^16 bits.
   - If a label is not in the specified format or is repeated twice.
   - If the total number of instructions exceeds the allotted space for Instruction memory.

   Note: The extra whitespaces or indentation are ignored by the program
2. **Execution Error**
   An Execution error occurs in the following cases:
   - The instructions beq, bne, or j asks for a label that has not been initialized.
   - If the data memory exceeds the allotted space for data. Or asks for an invalid memory address. (Like not a multiple of four or negative)
   - If an instruction tries to modify the value of the zero register.

TESTING

- To test the code save the MIPS code in a text file in the same directory as the main file and then run the command {{ ./a.out (Name of Test File) }}. The output of the function will be saved in a file and any error if any will be reported on the console.
- We have done extensive testing as a part of the assignment to check the validity of the MIPS simulator. The test cases have been added to the Zip folder.
- We have attempted to handle syntax errors and execution errors of almost every type and tested them rigorously.

Test Cases 0-7 are various possible corner cases that would produce a syntax or execution error. And test cases 8-15 are executable and are used to test the semantic validity and the output of the compiler.

THE END