COL216 Assignment 3

Kuldeep Meena(2019CS10490)

March 13, 2021

Approach

1. Input Output Specification

Input: The C++ program takes an input file named example.asm with mips like code with following commands : add, sub, mul, beq, bne, slt, j, lw, sw, addi

Output: Register file content after each execution and relevant statistics of instruction execution

2. Some design decisions

- Only INT registers are available for storing data
- Each instruction and word to be stored or loaded can be adjusted in 4 bytes
- Only following commands are valid :add, sub, mul, beq, bne, slt, j, lw, sw,addi
- There is no dynamic memory allocation explicitly although can be done
- Stack grows downwards and $sp initially points just outside main memory
- I assume that instead of labels line number of the corresponding instruction will be used to jump to an instruction
- If there is an error the system stops execution with a print of last instruction during runtime
- The programmers are expected to handle infinite loops themselves

Working of algorithm

The algorithm works converts MIPS program into tokens using a very simplified lexer(which can be extended to complex versions using a program like FLEX for lexer generation) that divides each line of instruction into tokens namely instruction,register, lparen,rparen,etc. Once each line is tokenised and is token error free, since there are very few types of instructions possible in the given case

I have used a matching parser (which can be replaced by YACC or other helpers for complex grammer instuctions) which checks if the sentence of the program is a valid match of various possible valid sentences possible and if a sentence is possible it undergoes evaluation of the sentence. During the evaluation there is a check for any memory locations being used if memory location are null or memory location does not exist then appropriate errors are reported otherwise the instruction is executed and memory/register is updated as required.

Token generator takes an input by lines and to tokenise looks at one char at a time. Parser matches the produced token against some constant number of possible sentence productions and evaluation evaluates one instruction at a time. ALthough the lexer and parserMatcher takes only O(n) time the instructions execution time complexity is function of time complexity of the program to be interpreted. So time complexity is O(max(n,Time Complexity of example.asm))

Scope for improvement: The program works completely fine for assumed grammar and instruction commands but it is possible to improve by using a lexer and parser separately and using a well thought grammatical rules which I have not used in this assignment. I have focussed more on evaluation of instructions.

Some Test Cases

Because of long outputs and inputs I have attached the testCases in the file *testCases* please refer it to see the testCases

Testing Strategy

I have tried to do as exhaustive testing as possible and have also included some of the test cases that gives sense of the border cases for my algorithm.I have tested on several large test cases and checked the correctness. I have also included some test cases with wrong inputs and tested accordingly.I have tested for memory outbound and jump to unknown lines etc.