Design for COL216 Assignment 3

Shrey J. Patel, 2019CS10400
Aayush Goyal 2019CS10452

March 2021

1 Aim

To develop an interpreter for a subset of MIPS assembly language instructions. It should interpret instructions like **add, sub, mul, beq, bne, slt, j, li, lw, sw, addi**. It should also maintain 32 registers and 2^{20} Bytes of memory.

2 I/O specifications

- **Input:** MIPS assembly language program (text file).
- **Output:**
  1. **After every instruction:** Register File contents (32 register values in Hexadecimal format).
  2. **After execution completes:** Number of times each instruction was executed and Number of clock cycles.
- **To run the code:** Run the *make* command or *g++ code.cpp -o output* in terminal. This will create a executable named "output". Now run the command *./output file_name* or *./output file_name ⟩ output_filename*, where "file_name" is the name or relative path of the input file and output_filename is the name of the output file.

Note that we assume that every instruction is executed in a single clock cycle as mentioned in the assignment statement.

3 Approach

To make an interpreter for MIPS assembly language we first need to lex the whole input program. For this we have made a lexer function. This lexer function is given as input a string (which can be empty) and it breaks the string into different tokens. In all the lines we first have a instruction add, sub, mul, j which are followed by their arguments. Now the parameters for them are

space separated from the instruction. Further the parameters are comma (',') separated from each other. If our lexer function returns n tokens, that means 1^{st} and 2^{nd} were space or tab separated and the 2^{nd} to n^{th} tokens were comma separated. It also takes cares of all the redundant white spaces.

Now we read every line and then split every line into tokens using the above mentioned lexer function. For empty lines the lexer returns an empty vector and such lines are ignored. For lines containing labels like "loop:" the lexer returns a vector of size of 1 or if the input has it in the form "loop :" then it returns a vector of size 2 but we club them together again as "loop:".

Now the second task is to parse or process our code. For this we have made a parser function. Every time we encounter an instruction like add, sub, mul, etc. we check whether the parameters given are in required format (if not we throw appropriate error message and terminate the program) and then do the whatever the instruction requests.

4 Code Design and Implementation in C++

Elaborated in code.cpp file with appropriate comments. We have used the in-built unordered_map data structure of C++ for mapping string values to integers. Since we are mapping strings, it is very less probable that the hash values will clash and thus accessing values can be done in nearly O(1) time. Further the data structures used, algorithms, time and space complexity analysis is given below.

4.1 Data Structures

1. Memory: We have used an array named "Memory" to store the memory. Total allocated memory is 2^{20} Bytes and thus we can store 2^{18} data entries (because each data entry itself can occupy 4 Bytes memory)

2. Registers: We have stored the 32 registers in an unordered map. They map a string to an integer. Since in our case we are mapping using strings, the hash function would have negligible collisions in hash values. Thus the content of each register can be accessed efficiently in O(1) time.

3. Instructions: In MIPS the instructions are delimited by a new line. Thus we have maintained a vector of strings for storing every line of the input MIPS code.

4. Labels: For labels like "loop:" and others we have maintained an unordered map which maps the label name to the index i.e. the line number at which the corresponding label is stored in the instructions vector. This is required for branching we need to jump to the appropriate index where the label is present.

5. Operations: This stores the operations we have handled for the MIPS code. We have used an unordered map to to map the operation name

to an integer which denotes how many times the particular operation has been executed.

4.2 Algorithm

1. For Lexing: In MIPS, the instructions are delimited by space. Thus first we store every line in a vector of strings called "instructions". Now we made a lexer function which takes in as input a string and returns a vector of strings which contains tokens present in that string. In all the lines we first have a instruction add, sub, mul, j which are followed by their arguments. Now the parameters for them are space separated from the instruction. Further the parameters are comma (',') separated form each other. If our lexer function returns x tokens, that means 1^{st} and 2^{nd} were space or tab separated and the 2^{nd} to x^{th} tokens were comma separated. We maintained 2 booleans variables "first" and "second". The first becomes true when we encounter first character in the string, and then until we get a space or tab we consider that part of string as first token. Now after spaces and tabs when we encounter a character, second becomes true. Now we use comma as the delimiter for separating further tokens. In case of labels like "loop :" lexer returns a vector of size 2 but we modify the line in instructions vector to "loop:" so that after this lexer returns vector of size 1 (because in the parser we skip instructions with only one token)

2. For Parsing: We made a parser function for implementing the instructions. Every instruction is first broken into tokens using the lexer function. Now further depending on the number of tokens in that line we perform different operations, they are:

   (a) Size 4: This means that we have an operation like add, sub, mul, beq, bne, slt or addi. For each of these we check whether the further tokens are in required format of registers, integers or label names or not. If the required format of parameters is violated then appropriate error message is thrown otherwise the operation is executed.

   (b) Size 3: This means we have an operation like li, lw or sw and we get this information from the first token. Now each of the second token denote some register and third token should be a number or something of the form "x($yy)" for sw and lw. Further sw stores the value of register to the memory and lw loads the value stored at that memory into the register.

   (c) Size 2: This means that we have a jump 'j' operation. The first parameter must have the string "j" and the second parameter must be a label name. Using this instruction, we branch to that appropriate label.

   (d) Size 1: A single token means that it's just a label name, we don't have to do anything here. We already checked during lexing whether

it is a valid label name or not.

We have maintained an integer variable "itr", this itr or iterator keeps track of the instruction number we are currently executing (instruction number is the index at which the instruction is present in the "instructions" vector). We always execute the instruction present at itr index in the instructions vector. Thus for jumping to a label we just change the value of itr to index of that label. This index for accessing in O(1) is stored in a unordered map "labels".

4.3 Analysis

n = No. of characters in the input text file.

1) **Space Complexity**: $O(n + 2^{20})$
We have stored every line in an instructions vector. Apart from that we have used data structures which also take space proportional to the number of instructions, labels, registers. We have also made an array to implement the Memory part, this Memory array has a size of 2^{20} Bytes. So total space complexity is $O(n + 2^{20})$.

2) **Time Complexity**: $O(n)$
There are n characters and so the lexer takes O(n) time to read the data. Now further during parsing it depends in the time complexity of the input MIPS Program. If the input has an infinte while loop then it will be reflected during parsing and thus the C++ code will keep running forever.

4.4 Testing & Exception Handling

1. Testing:- We manually created our own test cases, both containing correct and incorrect codes, and provided them as input to the code.cpp . In case of correct input, we printed the state of the 32 registers after every instruction, and in the end we printed how many times each of the instructions were executed. While for incorrect expressions, we terminated the program with a appropriate error message which would be helpful to the user for debugging. We have basically tested out input codes with syntax errors, infinite loops, storing and loading from memory, and other. The example test cases are present in the submission folder.

2. Error handling:- We have thrown error at required places to the best of our MIPS syntax knowledge. Some examples of the error cases are:

   - Defining an instruction set for labels twice.
   - Defining label names as reserved keywords.
   - Colon required at the end of label name.

- Invalid instruction(s).
- Jumping to an invalid label name.
- Register name not valid or doesn't start with $.
- Immediate values are not given as integers for addi.
- Incorrect or unaligned memory address for lw and sw.
- Memory limit exceeded.
- Syntax errors for inappropriate usage of commas, newlines, tabs or whitespaces, thrown while lexing the input file.

Hence all the test cases have helped us to ensure that our code.cpp is correct. Getting correct outputs on some test cases can never be the right criteria to judge if an algorithm is correct or not but extensive checking can still ensure us that we have done the things correctly.