CSC3050 Assignment 1 Report

@October 13, 2023 @Yuzhe YANG 121090684

Big pictures and thoughts

This project tasked us with creating a MIPS assembler. The assembler's primary role is to convert MIPS instructions into binary machine codes.

MIPS instructions come in three types: R-type, I-type, and J-type. R-type instructions don't require a target address, immediate value, or branch displacement. I-type instructions have a 16-bit immediate field coding an immediate operand, a branch target offset (shifting to a label), or a displacement for a memory operand. J-type instructions involve jump instructions like `j` and `jal`.

So, my general approach is to categorize instructions into specific types first. In total, I classify all the instructions into 12 types by the format of instructions. For example, for the instruction format: `<instruction> $rd, $rt, ra`, it has instruction: `sll`, `sra`, `srl`.

The reason why I classify the instructions in this way rather than classify them into I, J, R type is because it is more convenient for me to detect all the elements in a single instruction line.

After I pre-processed all the input MIPS file, the program will read the file line by line and translate it into machine code.

High level implementation ideas

According to the requirements, there are 5 parts in my project:

1. `labelTable.py`: store the classified type of instruction
2. `phase1.py`: scan and pre-process the origin MIPS file
3. `phase2.py`: scan the processed MIPS file again and translate line by line
4. `function.py`: maintain some useful functions later I will use
5. `tester.py`: detect if the output file is matched with expected output

Implementation details

phase1

The given MIPS codes file consists of ".data" section and ".text" section and we only need to deal with the ".text" section. The ".text" section has labels, instructions and comments and we are required to ignore the comments.

In order to process the MIPS code, I wrote `pre_precess()` function in `fuction.py`, which will do:

1. Remove unwanted text like comments or blank space by using `rm()` function
2. Load `.text` part by using `load_text()` function
3. Find labels in the text, and record every labels' address by using `find_labels()` function
4. Remove labels from the text by using `rm_labels()` function

This function will return the pre-processed text and the dictionary of labels, which can let me directly call the function in `phase2.py`

labelTable

In `labelTabel.py`, I had divide all the instruction into 12 types by its format, which is stored in a dictionary. In this dictionary, its contains all the information about this instruction: `opcode`, `func`, `type` . Then I created a `Reg` array to store every register nam by its number order.

phase2

The program can be divided into these part:

1. Import Statements:
   - `import functions as f`: Imports a module named `functions` with the alias `f`.
   - `import labelTable as m`: Imports a module named `labelTable` with the alias `m`.
   - `import sys`: Imports the `sys` module for handling command-line arguments.
2. Command-line Arguments:
   - `file = sys.argv[1]`: Reads the first command-line argument as the input file name.
   - `output_file = sys.argv[2]`: Reads the second command-line argument as the output file name.
3. Pre-processing:
   - `text, label_dict = f.pre_process(file)`: Calls the `pre_process()` function from the `functions` module, passing the input file name. It returns processed text and a dictionary of labels.
4. Parsing and Translation:
   - The script then iterates through each line of the processed text.
   - It parses the elements of each line, including instructions, registers, immediate values, addresses, and types. In this procedure, it will remove all the blank space or comma, and store each element in an array.
   - Calls the `translate()` function from the `functions` module to get the MIPS instruction translated into machine code.

     ```
     def translate(type_num, inst, reg, address, imm, label, current):
         """
         Translates the given instruction into its corresponding binary format.

         Parameters:
         type_num (int): The type of instruction.
         inst (dict): The instruction dictionary containing the opcode and function fields.
     ```

```
    reg (list): The list of registers used in the instruction.
    address (dict): The dictionary containing the register and immediate fields used in the instruction.
    imm (int): The immediate value used in the instruction.
    label (int): The label value used in the instruction.
    current (int): The current instruction address.

    Returns:
    dict: The dictionary containing the binary format of the instruction.
    """
```

The function uses a series of conditional statements based on the `type_num` to determine the format of the MIPS instruction and then constructs a dictionary representing the binary format of that instruction. The `type_num` values appear to correspond to 12 MIPS instruction types I had divided before in `labelTabel.py`. The function adapts the translation based on these types.

For jump instruction, type 8 will calculate the absolute address based on the label by `address = START_ADDRESS + label*4`. Then it will call `Jtype_drop()` function to fit the 32bit address into 26 bit. For branch instruction, type 6 and type 7 will calculate the relative offset between PC counter and label address.

- Translates the machine code into a binary format based on the instruction type (R, I, or J). Then traverse the MIPS text line again until all the MIPS code has been translated into binary machine code.

tester

In this part, the tester will compare the output of `phase2.py` with the expected output.

In order to test the project, I wrote a shell file. Where you can change any test file name you want.

```
cd "/home/csc3050/hello_world/csc3050_toby/CSC3050_P1"
input_file="testfile6.asm"
output_file="output.txt"
expected_output_file="expectedoutput6.txt"

python phase1.py "$input_file"
python tester.py "$input_file" "$output_file" "$expected_output_file"
```

Also, you can directly run `python tester.py "$input_file" "$output_file" "$expected_output_file"` in the terminal after you navigate into proper folder.