# MIPS Assembler Project Report

## 1. Main idea

The target of this program is to translate the MIPS language written in a .asm file into machine code as a .txt file.

## 2. Program structure

This program, as submitted as a zip file, is expected to contain 3 files in total, namely: **MIPS.py**, **tester.py**, and **report.pdf**.

To test the program, it is expected to put all the testing files in the root folder. Then use a python editor to open it, execute tester.py, and follow the instructions. Additionally, it's also feasible to cd the address of this folder in the cmd shell first and invoke python to run it.

This program adopts a **functional programming paradigm**. In the MIPS.py, it contains 2 functions, phase1(), and phase2(); in the tester.py, it **imports MIPS.py** and serves as a main function to initiate the translation as well as examine the output.

## 3. Implementation

Generally, the translation process is executed line by line, that is, it reads one line from the MIPS instruction, breaking it down into different pieces, such as register names, immediate numbers, labels and etc., referring to their corresponding machine code separately, and then resembles them back together in sequence to form an output line of machine code.

However, since in MIPS language, there are some comments and 'jump' and 'branch' instructions, which requires the absolute address number of the target instruction, this program scans the whole code **two times** (or more exactly, **3** times) to finish the translation.

In phase1() function, the program reads the raw code and filters the comments and .data part. Next, it initiates and **PC** (program counter) and iterates over the lines again to find all the labels by str.find(“:”) method and records their corresponding addresses from the PC. **This information is stored in the “label_table” as a dictionary in the format of {‘label’: address}.**

```
# label mapping
        for i in range(len(linecode)):

            if linecode[i].find(':') == -1:
                program_counter += 4

            else:
        # the instruction is below the label
                if linecode[i].strip().endswith(':'):
                    label = linecode[i][:-1]

        # the label and the instruction are in the same line
                else:
                    strline = linecode[i]
                    label = strline[:strline.find(':')].strip()

                label_table[label] = program_counter + 4

```

```
Label table is:
 {'__builtin_memcpy_aligned_large': 4194304,
'__builtin_memcpy_bytes': 4194336,   … …
}
```

In phase2(), this program imports re (regular express) to help to break down the code into letters and numbers and other simple symbols for easier analysis. Note that the minus sign ‘-’ is specially included:

```
for line in code:
    mcode = 0
    linelist = re.findall(r"[\w\|-]+", line)
```

For each line, the program identifies its instruction type by looking up the operand’s name in the **instruction_map**, which is a **dictionary** contains the essential information for each instruction:

```
instruction_map = {

    # R type instructions
    # format: "instruction : [type, function_code, operands_tuple]

    "add": ["R", "100000", ('rd', 'rs', 'rt')],
    "div": ["R", "011010", ('rs', 'rt')],


    # I type instructions

    "addi": ["I", "001000", ('rt', 'rs', 'i')],
    "addiu": ["I", "001001", ('rt', 'rs', 'i')],


    # J type instructions

    "j": ["J", "000010"],
    "jal": ["J", "000011"]

}
```

(Some parts of the instruction_map)

Next, the program looks up the register's corresponding code in the **reg_table**, which is also a **dictionary**:

```
reg_table = {

    "zero": "00000",
    "at": "00001",
    "v0": "00010",
    "v1": "00011",
    "a0": "00100",

}
```

(Some parts of the reg_table)

Then the program **updates the preset values in the dictionary and resembles them according to the sequence identified in the tuple from the instruction_map**:

```
elif info_list[0] == 'I':
    i_dic = {'rs': '00000', 'rt': '00000', 'i': '0000000000000000'}
    i_tuple = info_list[2]

    ... ...

    else:
        i_dic[i_tuple[index-1]] = reg_table[linelist[index]]

    mcode = info_list[1] + i_dic["rs"] + i_dic["rt"] + i_dic["i"]
```

(Some partial logic)

Note that for the jump and branch instructions that involves label address, the program does specific arithmetic operations correspondingly:

```
if diff >= 0:
    immediate = bin(diff)[2:-2].zfill(16)
else:
    # 1110 = 111111110
    immediate = bin(diff & 0xffff)[2:-2].rjust(16, '1')
```

After the translation for each instruction line, the corresponding machine code is appended into a **list** called **machine_code**. After all the iterations are finished, the phase2() function returns machine_code.

So far, all the translation is finished, so does the duty of MIPS.py.

The output.txt is generated and written in the tester.py:

```
...
import MIPS

# ask the user to input files' names and check whether the file exist
test_file = input("Please enter the test file name: ")
...
...
output_file = input("Please enter the output file name: ")
f = open(output_file, 'w')

# conduct MIPS.py to do compiling
machine_code = MIPS.phase2(MIPS.phase1(test_file))
for line in machine_code:
    # print(line)
    f.write(line)
    f.write('\n')
    f.close()
```

Finally, the program erases some newline characters and spaces to compare the expected output and generated output.

```
All Passed! Congratulations!
```