Project Report

1. **Interpretation of the Project:**
   This project is aimed at writing an assembler that can translate a MIPS program into Machine Language, which contains only 0 and 1.
   The assembler should be used in command line. When it is used, two parameters should be passed to the assembler via command line. The final usage is "./assembler input_file.asm output_file.txt".
   Each MIPS instruction, when translated into Machine Language, is a line of 32 binary numbers. All the instructions that we have to translate starts with the name of the operation and then one or several register names, immediate values or labels. In this project, we have to identify these fields and translate them into appropriate binary numbers.
   Different form translating only single instructions, this project need to consider the context of each instruction, such as the value of PC when executing the program and the meanings of labels.

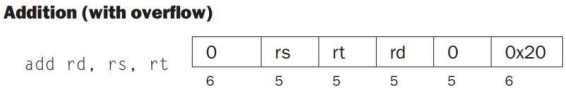
2. **General Idea of Implementation:**
   a) **Scan the file once and store the information about labels.**
      First of all, to deal with the situation of a label is referenced before it is defined, we have to know the definition of the label before we translate them. There is one obvious solution, which is to find the definition of the label when we reach its reference. However, it costs too much time. Therefore, we turn to the alternative solution: store the information about labels before we translate them. Then, the time complexity is O(length of the source file).
   b) **Read the source file line by line. Translate it when it contains an instruction. Keep track of PC while processing, in case we have to calculate offset.**
   c) **Write the result immediately after we translated one instruction.**
   d) **Stop the process when we have translated all the instructions.**

3. **Specific Methods When Implementing:**
   Initially, the most obvious method to handle them in my mind is to use a switch statement or multiple if statements to determine the corresponding read and write actions of each instruction, which looks like:

```
switch(operation_name){
    case "add": ... ; break;
    case "j": ... ; break;
    ...
}
```

```
if(operation_name == "add"){
    ...
} else if (operation_name == "j"){
    ...
} else ...
```

   However, writing such program can be painful.
   Hence, I began to think how to handle them systematically.
   I recognized several patterns of the MIPS code, which helped me to handle them systematically.
   a) Every instruction starts with a label or the name of the operation.
      1) This helps us to identify the operation.
   b) For a basic instruction like



   , regarding the left part, "add" is the operation name, "rd", "rs", "rt" are the names of three parameters, which is similar to the function definition in C, "void add(int rd, int rs, int rt)".
   When translating it, we refer to the definition of the operation, which are the left and right parts in the picture (later in the program we can read the definitions first and store them in case we refer to them).

We just need to substitute the “rs”, “rt”, “rd” fields in the right part with the value of the parameters with the same name, and leave the constant fields unchanged.

1) For example, when we translate “add $t0, $t1, $t2”, we look into each field from left to right:
   i. “0” of 6 bits, constant. Leave it as “000000”.
   ii. “rs”, a reference to the first parameter of 5 bits. Find the value of the parameter, “$t0”, which is number 8. Write it as “01000”.
   iii. “rt”, a reference to the second parameter of 5 bits. Find the value of the parameter, “$t1”, which is number 9. Write it as “01001”.
   iv. “rd”, a reference to the third parameter of 5 bits. Find the value of the parameter, “$t2”, which is number 10. Write it as “01010”.
   v. “0” of 5 bits, constant. Leave it as “00000”.
   vi. “0x20” of 6 bits, constant. Leave it as “100000”.
   vii. Link the fields. Obtain “000000|01001|01010|01000|00000|100000”.

c) So far, we can handle most of the instructions following the last pattern. But there might be other cases:

1)

Branch on equal

```
beq rs, rt, label
```

4	rs	rt	Offset
6	5	5	16

Case: In branch instructions, “Offset” field need to be calculated with PC and the definition of label.

Solution: Because we keep track of PC and have already stored the information of labels, we can calculate it easily.

2)

Load word

```
lw rt, address
```

0x23	rs	rt	Offset
6	5	5	16

Case: In L/S instructions, “rs” field is obtained by decoding the value of parameter “address”. “Offset” field is obtained by decoding it as well.

Solution: In this case, we just need to decode address first, read “rt”, and then write “0x23”, “rs”, “rt”, “Offset” in the end.

d) There are serval data types of the value of parameters:
   1) Register Name: “$t0”, “$zero”, …
   2) Decimal Number: “123” in “$lui $t0, 123”, …
   3) Label Name: “loop” in “j loop”, …

The type of const field is always hexadecimal number.

When reading the values of parameters and the value of constant fields, we need to carefully identify those types to translate them correctly.

4. Final Implementation:

a) Copy and paste from the text book to obtain the list of definitions of instructions to an Excel chart.

	A	B	C
1	0 rs rt rd 0 0x20	add rd, rs, rt	
2	0 rs rt rd 0 0x21	addu rd, rs, rt	
3	8 rs rt imm	addi rt, rs, imm	
4	9 rs rt imm	addiu rt, rs, imm	

b) **Check the definitions to make sure that the format suits my needs.**
For example, “address” in the field must be lower case, since my program will be detecting the “address” case-sensitively. Another example is that the definition, “0xe rs rt Imm | xori rt, rs, imm”, has an error that “Imm” and “imm” are different in terms of strings in C++, but they actually mean one thing, so “Imm” must be changed to “imm”.

c) **Format them using VS Code’s multi-cursor into C++ constants in the program.**

```
// Storing the raw information about all instructions, which will be
// deserialized later in the program.
const string INSTRUCTIONS[] = {
    "div rs rt",        "divu rs rt",       "mult rs rt ",      "multu rs rt",
    "madd rs rt",       "maddu rs rt",      "msub rs rt",       "msub rs rt",
    "teq rs rt",        "tne rs rt",        "tge rs rt",        "tgeu rs rt",
    "tlt rs rt",        "tltu rs rt",       "mfhi rd",          "mflo rd",
```

```
// Storing all the raw information about how to translate each instruction.
// Each instruction and its translation are stored at the same index.
const string PLACE_HOLDERS[] = {"0 rs rt 0 0×1a",
                                "0 rs rt 0 0×1b",
                                "0 rs rt 0 0×18",
                                "0 rs rt 0 0×19",
                                "0×1c rs rt 0 0",
```

d) **Implement the program flow:**
1) Open files.
2) Read the input file once to store information of labels.
3) Deserialize definitions of instructions.
4) Read the input file again to translate the input file instruction by instruction.
5) Exit.

5. Final Usage of this Project:

a) Compile the source file “MIPS_assembler.cpp” and obtain an executable file “assembler”.
As a reminder, the code is only tested in **Ubuntu** environment.

b) Switch the current folder to the folder that contains the executable file.

c) Run “**./assembler input_file.asm output_file.txt**” in terminal, where “input_file.asm” should be substituted by the path to the input file and “output_file.txt” the path to the output file.