1 Purpose

This lab involves implementing a basic assembler for the LC-3 assembly language. My goal is to create a toy assembler that is capable of translating the assembler code into machine code.

2 Principles

2.1 Reading Instructions

The first step to achieve our goal is to find a way reading the instructions. In my case, I will read an instruction section by section based on spaces or some other punctuation marks.

This method allows me to break each instruction into different parts and interpret each instruction according to the meaning of each part. Another advantage of this is that I can quickly find similarities between different instructions and design some general functions to handle the translation task of different instructions.

The code for reading one part of an instruction is as follows.

```
//Read one part of a instruction (between tow space characters)
int OnepartRead(const char* instruction , char output[], int index){
    int i;
    for (i = index; instruction[i] != ' ' && instruction[i] != '\0' &&
        instruction[i] != '\n'; i++){
        output[i - index] = instruction[i];
    }
    output[i - index] = '\0';
    return i+1;
}
```

According to the requirements of the assembly code format of the experiment, a single instruction can be thus decomposed into multiple components. This achievement marks the first step in breaking down a line of instructions into different elements (such as labels, opcodes, operands, etc.) and subsequently performing specific operations based on the unique characteristics of each component.

2.2 The First Pass

The process of translate the assembly program consists of two pass. The aim of the first pass is to build a symbol table of those label appeared in program. To build the symbol table, I designed two data structures for the label and the table.

```
//The data structure of label
typedef struct label{
    char name[20];    //The name of label
    unsigned short address;    //The address of label
}LABEL;

//The data structure of symbol table
typedef struct label_table{
    unsigned short start;    //The start address of the whole program
    LABEL table[30];    //Symbol table
    int num;    //The number of labels
}TABLE;
```

And then I can build the table while the first time the program traverses those instructions.

Since each label must appear once at the beginning of a line of code, I only need to read the first part of each code on the first traverse, and then determine that it is not part of the LC-3 instruction set, such a note must be a label, the last thing I need to do is to record its name and address.

To do this, I have to construct the LC-3 instruction set in memory, which is also quite simple. All I need is just a liner table which stores those 15 instructions.

There are few things need to be noticed. Because of the `.STRINGZ` instruction and `.BLKW` instruction may occupy more than one word in memory, we need to consider the change they make during the process of building the symbol table.

After doing all these things, I can implement the first pass.

```
int index = 0;    //The position of the instruction reading point
int index_table = 0;
char temp[100];

//First pass to build the Symbol table
PC = -1;
for (int i = 0; i < num_lines; i++){
    if (Find_label(lines[i])){
        index = OnepartRead(lines[i], SymbolTable.table[index_table].name
            , 0);
        SymbolTable.table[index_table].address = PC;
        index_table++;
    }
    index = OnepartRead(lines[i], temp, index);
    if (!strcmp(temp, ".STRINGZ")){
        for (int j = index ; lines[i][j] != '\0' && lines[i][j] != '\n' ;
             j++){
            if (lines[i][j] == '"') continue;
            PC++;
        }
    }//incrament the pc depending on the strings
    if (!strcmp(temp, ".BLKW")){
        int number;
        char N[0];
        index = OnepartRead(lines[i], temp, index);
        number = transform(temp, N, 0);
        for (int j = 1 ; j < number ; j++){
            PC++;
        }
    }//e.g. .BLKW 3
    PC++;
}
SymbolTable.num = index_table;    //Record the number of labels
```

The above code shows how to create a symbol table. And of course I considered the influence of `.STRINGZ` and `.BLKW`.

2.3 Translation of a Single Instruction

For the translation of a single instruction, the point is to classify the instructions, find the same or similar format instructions, and take roughly the same treatment for them.

In my case, I treat the `ADD` instruction and the `AND` instruction as the same class instruction, the memory access instructions (e.g. `LD`, `ST`, `LDR`, `STI`) as one type, and the pseudo (e.g. `.ORIG`, `.FILL`) instructions as one type.

When these instructions are classified, we can make corresponding operations according to each part of the instruction. For example, for the instructions `ADD R0, R0, #5`, I can take the first part of it, identify its opcode is `0001`, and then read the second part of it, identify it to register `R0`, and convert it to binary `000`. Then read the third part, convert to `000`, and finally read `#5`, convert it to the five-digit immediate number `00101`.

Since the AND directive has the same format as the ADD directive, I can handle them in the same way. Only in the process of implementing these functions, I also need to implement some additional functions such as base conversion, condition judgment and so on.

The instructions to access memory basically involve all of the above procedures, and I will show this part of my program. The following code shows the way I deal with the memory access instructions.

```
//Translate the LOAD and STORE instructions
void Instrucion_LS(const char instruction[], char machine_code[], char
    Part[], int index){
    char temp[100];
    char number[100];

    if (Part[2] == 'R'){
        //If the instruction is LDR ot STR
        //Read one more part of the instruction for twice
        for (int i = 0 ; i < 2 ; i++){
            index = OnepartRead(instruction, Part, index);
            transform(Part, temp, 3);
            strcat(machine_code, temp);
        }
        //Read the PC-Offset
        index = OnepartRead(instruction, Part, index);
        transform(Part, temp, 6);
        strcat(machine_code, temp);
        return ;
    }else{
        //The instruction is LD LDI ST or STI
        index = OnepartRead(instruction, Part, index);
        transform(Part, temp, 3);
        strcat(machine_code, temp);
        index = OnepartRead(instruction, Part, index);
        if (Part[0] == '#' || Part[0] == 'x'){
            //If the last part is a IMM
            transform(Part, temp, 9);
            strcat(machine_code, temp);
            return ;
        }else{
            //A label
            for (int i = 0; i < SymbolTable.num; i++){
                if (!strcmp(Part, SymbolTable.table[i].name)){
                    SplitNumber(SymbolTable.table[i].address - (PC + 1),
                        number); //Calculate the PC-Offeset
                    transform(number, temp, 9);
                    strcat(machine_code, temp);
                    return ;
                }
            }
        }
    }
}
```

I will explain some of the functions not mentioned in the above program in the next section.

3 Procedure

According to the textbook, we can translate the assembler program in a two pass process.

Step (1) Read the instructions from a `.asm` file.

Step (2) First pass to create the symbol table.

Step (3) Sectond pass to translate every instruction.

Step (4) Stores the machine code as a `.text` file.

In the process of translating a single instruction, the biggest problem is to realize the conversion between data in different bases. Since machine code is a 16-bit binary string, and assembly code may contain decimal or hexadecimal numbers, my program needs to be able to correctly identify them and convert them to a binary string of corresponding bits. For example, for the code `ADD R0, R7, xA`, my program needs to be able to recognize `R0`, `R7`, the two decimal numbers representing registers, and then convert them into the 3-bit binary string `000` and `111`, for `xA`, My code needs to be able to recognize this immediate hexadecimal number and convert it to the 5-bit binary number `01010`.

So I designed the following function `transform`, whose function is to recognize an incoming decimal or hexadecimal number, convert it into a binary string requiring an extended number, and return a corresponding decimal unsigned number.

```
unsigned short transform(const char input[], char output[], int expend){
    int index = 0;
    unsigned short Input = 0;     //16 bits to calculate the complement
    //Convert the number in array to a unsinged shor number which can be
        easily calculated
    if (input[index++] == 'x'){
        //Hex
        if (input[index++] == '-'){
            //Negative
            for (int i = index; input[i] != ' ' && input[i] != '\0' &&
                input[i] != '\n' && input[i] != ','; i++){
                 if (input[i] >= 'A'){
                     Input = Input * 16 - (input[i] - 'A' + 10);
                 }else{
                     Input = Input * 16 - (input[i] - '0');
                 }
            }
        }else{
            //Positive
            index--;
            for (int i = index; input[i] != ' ' && input[i] != '\0' &&
                input[i] != '\n' && input[i] != ','; i++){
                 if (input[i] >= 'A'){
                     Input = Input * 16 + (input[i] - 'A' + 10);
                 }else{
                     Input = Input * 16 + (input[i] - '0');
                 }
            }
        }
    }else{
        //Decimal from IMM or register
        if (input[index - 1] != '#' && input[index - 1] != 'R') index--;
        if (input[index++] == '-'){
            for (int i = index; input[i] != ' ' && input[i] != '\0' &&
                input[i] != '\n' && input[i] != ','; i++){
                 Input = Input * 10 - (input[i] - '0');
            }
        }else{
            index--;
            for (int i = index; input[i] != ' ' && input[i] != '\0' &&
                input[i] != '\n' && input[i] != ','; i++){
                 Input = Input * 10 + (input[i] - '0');
            }
        }
    }
```

```
    unsigned short temp = Input;
    char Temp[100];
    int i = 0;
    do    //Convert to binary
    {
        Temp[i] = (temp % 2) + '0';
        temp = temp / 2;
        i++;
    } while (i < 16);

    //Store the result
    for (i = 0; i < expend; i++){
        output[expend - 1 - i] = Temp[i];
    }
    output[i] = '\0';

    return Input;
}
```

Using this function, I can implement the register, immediate number and PC-Offeset translation in assembly code.

For the linear table I mentioned in section 2.2, I designed the following data structure.

```
typedef struct INSTRUCTION
{
    char ASM[7];
    char Opcode[5];
}Instructions;

Instructions instructions[INSTRUCTION_NUM] = {{"LEA", "1110"}, {"ADD", "
    0001"}, {"AND", "0101"}, {"JMP", "1100"}, {"JSR", "0100"}, {"JSRR", "
    0100"}, {"BR", "0000"}, {"BRZ", "0000"}, {"BRP", "0000"}, {"BRN", "
    0000"}, {"BRNZ", "0000"}, {"BRNP", "0000"}, {"BRZP", "0000"}, {"BRNZP"
    , "0000"}, {"LD", "0010"}, {"LDI", "1010"}, {"LDR", "0110"}, {"NOT", "
    1001"}, {"RET", "1100"}, {"RTI", "1000"}, {"ST", "0011"}, {"STI", "
    1011"}, {"STR", "0111"}, {"TRAP", "1111"}};
```

Now I just need to traverse through the table to determine what an instruction is and find its corresponding opcode.

4 Results

To test my program, I wrote a short assembler program including most of the characteristics of LC-3 instructions.

```
.ORIG x3000
AND R0, R0, #0
ADD R0, R0, #1
LD R0, LABEL1
ADD R1, R0, R0
AND R1, R1, R0
BRZ LABEL6
LABEL6 ADD R1, R1, R1
JSR LOOP
NOT R4, R0
STI R4, LABEL5
LEA R0, LABEL3
TRAP x22
```

```
LEA R0, LABEL4
TRAP x22
LDR R5, R0, #0
STR R5, R0, #1
TRAP x22
TRAP x25
LOOP ST R0, LABEL2
LD R0, LABEL5
ABC ADD R0, R0, #-1
BRP ABC
LD R0, LABEL2
RET
LABEL1 .FILL x4000
```

```
LABEL2 .BLKW 3
LABEL3 .STRINGZ "Hello
    world!"
LABEL4 .STRINGZ "USTC"
LABEL5 .FILL x4001
.END
```

Here is my program's translation of above assembler program.

```
0011000000000000          .ORIG x3000
0101000000100000          AND R0, R0, #0
0001000000100001          ADD R0, R0, #1
0010000000010101          LD R0, LABEL1
0001001000000000          ADD R1, R0, R0
0101001001000000          AND R1, R1, R0
0000010000000000          BRZ LABEL6
0001001001000001          LABEL6 ADD R1, R1, R1
0100100000001010          JSR LOOP
1001100000111111          NOT R4, R0
1011100000100100          STI R4, LABEL5
1110000000010001          LEA R0, LABEL3
1111000000100010          TRAP x22
1110000000011100          LEA R0, LABEL4
1111000000100010          TRAP x22
0110101000000000          LDR R5, R0, #0
0111101000000001          STR R5, R0, #1
1111000000100010          TRAP x22
1111000000100101          TRAP x25
0011000000000110          LOOP ST R0, LABEL2
0010000000011010          LD R0, LABEL5
0001000000111111          ABC ADD R0, R0, #-1
0000001111111110          BRP ABC
0010000000000010          LD R0, LABEL2
1100000111000000          RET
0100000000000000          LABEL1 .FILL x4000
0000000000000000          LABEL2 .BLKW 3
0000000000000000
0000000000000000
0000000001001000          LABEL3  'H'
0000000001100101                  'e'
0000000001101100                  'l'
0000000001101100                  'l'
0000000001101111                  'o'
0000000000100000                  ' '
0000000001110111                  'w'
0000000001101111                  'o'
0000000001110010                  'r'
0000000001101100                  'l'
0000000001100100                  'd'
0000000000100001                  '!'
0000000000000000          .STRINGZ "Hello world!"
0000000001010101          LABEL4  'U'
0000000001010011                  'S'
0000000001010100                  'T'
0000000001000011                  'C'
0000000000000000          .STRINGZ "USTC"
0100000000000001          LABEL5 .FILL x4001
                          .END
```

Load the machine code into the LC-3 simulator, the running result is `Hello world!USTCUUTC`, which meets the expectation.