Lab 7 Report

--**PB22111599 杨映川**

Purpose

Implement a basic assembler for the LC-3 assembly language.

Principles

1. Using the 2-pass method, the whole program will be traversed two times.
2. The 1st traverse is to locate all the labels and then store them into a specific table, which I used a `struct` array to achieve.

```
typedef struct node
{
    char name[MAX_LINE_LENGTH];
    int ad_incr;
} Label;
```

3. The 2nd traverse is to translate the assembly code line by line into the machine code.
4. For instructions like BR, LD, LDI, LDR, JSR that can be written either with numbers or labels, I used the function `NumOrLabel` in which includes both the `immediatenumberprocess` function and the `labelprocess` function that will execute according to the input.

```
void NumOrLabel(char *instruction, char *machine_code, int starting_point, int offset, int ad_incr)
{
    if ((instruction[starting_point] == 'x') || (instruction[starting_point] == '#'))
    { // number
        immediatenumberprocess(instruction, machine_code, starting_point, offset);
    }
    else // label
    {
        labelprocess(instruction, machine_code, starting_point, offset, ad_incr);
    }
}
```

5. For instructions like ADD, NOT, STR that can be followed by registers, I uesd the function `registerprocess` which will transform the number of the register into a 3-bit binary string snippet.

Procedure

1. Pseudo operations like .STRINGZ and .BLKW will generate a few lines of machine code, so inside one line of output[i] several '\n' are used to create new lines.
2. In order to improve the readability of the code, types of repetitive procedure are bagged together into one function. For example, the `NumOrLabel` function includes the `immediatenumberprocess` function and the `labelprocess` function. And inside each of the function there is a smaller function `BinCode` or `Value2Bin`.
3. The `BinCode` function will transform a string of numbers in either decimal or hexadecimal into its corresponding binary code string.

4. The `Value2Bin` function will transform a specific value, of course in decimal, into its corresponding binary code string.
5. The process above greatly decreases the difficulty to finish the whole program.
6. It is a big problem to cut off a piece of label inside one entire instruction, so the operations below are used to ensure the piece that is read in is correct.

```
char temp[MAX_LINE_LENGTH];
for (int i = 0; (instruction[i] != ' ') && (instruction[i] != 10) && (instruction[i] != 0); i++)
{
    temp[i] = instruction[i];
    temp[i + 1] = ' ';
    temp[i + 2] = 0;
}
int len = strlen(temp);
int labellength = 0;
```

Results

example 1

Using the file "test_in.asm" whose contents are:

```
.ORIG x3000
MAIN LD R1, DATA
TEST ADD R1, R1, #-10
BRZP TEST
ST R1, MEM
TRAP x25
DATA .FILL x1234
MEM .BLKW #1
.END
```

The output that is copied into the file "test_out.txt" is:

```
0011000000000000
0010001000000100
0001001001110110
0000011111111110
0011001000000010
1111000000100101
0001001000110100
0000000000000000
```

example 2

Using the file with the contents below:

ASM test_in.asm

```asm
.ORIG x3000
NOT R1, R2
MAIN LDI R1, DATA
LEA R0, MEM
LDR R5, R4, #5
TEST ADD R1, R1, #-10
BRZP TEST
ST R1, MEM
TRAP x25
DATA .FILL x1234
MEM .BLKW #2
.FILL xFF11
.STRINGZ "Idiot!"
.END
```

The output that is copied into another file is:

test_out.txt

```
0011000000000000
1001001010111111
1010001000000110
1110000000000110
0110101100000101
0001001001110110
0000011111111110
0011001000000010
1111000000100101
0001001000110100
0000000000000000
0000000000000000
1111111100010001
0000000001001001
0000000001100100
0000000001101001
0000000001101111
0000000001110100
0000000000100001
0000000000000000
```