Design suitable Data structures and implement Pass-I and Pass-II of a two-pass assembler for pseudo-machine. Implementation should consist of a few instructions from each category and few assembler directives. The output of Pass-I (intermediate code file and symbol table) should be input for Pass-II in c.

Features Included:

- **Instructions (IS)**: `MOVER`, `MOVEM`, `STOP`
- **Directives (AD)**: `START`, `END`
- **Declaratives (DL)**: `DS`, `DC`
- **Pass I**:
    - Generates Intermediate Code (`intermediate.txt`)
    - Generates Symbol Table (`symtab.txt`)
- **Pass II**:
    - Uses the above files to produce Machine Code (`machinecode.txt`)

Files Required

- `input.asm` (source code)
- `pass1.c` (Pass-I of the assembler)
- `pass2.c` (Pass-II of the assembler)

Input.asm :

START 100

A DS 1

MOVER AREG B

B DC 5

MOVEM AREG B

STOP

END

Program :

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct symbol {
    char label[20];
    int address;
    int index;
} symtab[50];

int symcount = 0, lc = 0;

char *optab[][3] = {
    {"STOP", "IS", "00"}, {"ADD", "IS", "01"}, {"SUB", "IS", "02"},
    {"MOVER", "IS", "04"}, {"MOVEM", "IS", "05"},
    {"START", "AD", "01"}, {"END", "AD", "02"},
    {"DS", "DL", "01"}, {"DC", "DL", "02"}
};

int getOpcodeIndex(char *mnemonic) {
    for (int i = 0; i < 9; i++) {
        if (strcmp(optab[i][0], mnemonic) == 0)
            return i;
    }
```

```
    return -1;
}

int getRegisterCode(char *reg) {
    if (strcmp(reg, "AREG") == 0) return 1;
    if (strcmp(reg, "BREG") == 0) return 2;
    if (strcmp(reg, "CREG") == 0) return 3;
    if (strcmp(reg, "DREG") == 0) return 4;
    return 0;
}

int search_symtab(char *label) {
    for (int i = 0; i < symcount; i++) {
        if (strcmp(symtab[i].label, label) == 0)
            return i;
    }
    return -1;
}

void add_symbol(char *label, int address) {
    int idx = search_symtab(label);
    if (idx != -1) {
        if (symtab[idx].address == -1)
            symtab[idx].address = address;
        return;
```

```
    }
    strcpy(symtab[symcount].label, label);
    symtab[symcount].address = address;
    symtab[symcount].index = symcount + 1;
    symcount++;
}

int main() {
    FILE *fp = fopen("input.asm", "r");
    FILE *ic = fopen("intermediate.txt", "w");
    FILE *st = fopen("symtab.txt", "w");

    char line[100], label[20], opcode[20], op1[20], op2[20];

    while (fgets(line, sizeof(line), fp)) {
        strcpy(label, "-"); strcpy(opcode, "-"); strcpy(op1, "-"); strcpy(op2, "-");

        int count = sscanf(line, "%s %s %s %s", label, opcode, op1, op2);

        // shift tokens if no label
        if (count == 3 && getOpcodeIndex(label) != -1) {
            strcpy(op2, op1);
            strcpy(op1, opcode);
            strcpy(opcode, label);
            strcpy(label, "-");
```

```
} else if (count == 2 && getOpcodeIndex(label) != -1) {
    strcpy(op1, opcode);
    strcpy(opcode, label);
    strcpy(label, "-");
}

int opIndex = getOpcodeIndex(opcode);

if (opIndex == -1) continue;

if (strcmp(optab[opIndex][0], "START") == 0) {
    lc = atoi(op1);
    fprintf(ic, "(AD,01) (C,%d)\n", lc);
    continue;
}

if (strcmp(optab[opIndex][0], "END") == 0) {
    fprintf(ic, "(AD,02)\n)");
    continue;
}

if (strcmp(label, "-") != 0)
    add_symbol(label, lc);

if (strcmp(optab[opIndex][0], "DS") == 0) {
```

```
            fprintf(ic, "(DL,01) (C,%s)\n", op1);
            lc += atoi(op1);
            continue;
        }

        if (strcmp(optab[opIndex][0], "DC") == 0) {
            fprintf(ic, "(DL,02) (C,%s)\n", op1);
            lc++;
            continue;
        }

        if (strcmp(optab[opIndex][1], "IS") == 0) {
            int reg = getRegisterCode(op1);
            int symIdx = search_symtab(op2);
            if (symIdx == -1) {
                add_symbol(op2, -1);
                symIdx = symcount - 1;
            }
            fprintf(ic, "(%s,%s) (R,%d) (S,%d)\n", optab[opIndex][1], optab[opIndex][2], reg,
symtab[symIdx].index);
            lc++;
        }
    }

    for (int i = 0; i < symcount; i++) {
        fprintf(st, "%d\t%s\t%d\n", symtab[i].index, symtab[i].label, symtab[i].address);
```

```
    }

    fclose(fp); fclose(ic); fclose(st);
    printf("Pass-I complete.\n");
    return 0;
}
```

Output :

Intermediate code.txt

(AD,01) (C,100)

(DL,01) (C,1)

(IS,04) (R,1) (S,2)

(DL,02) (C,5)

(IS,05) (R,1) (S,2)

Symtab.txt

1	A	100
2	B	102

ATOI Function :

The `atoi()` function in C is a standard library function used to convert a string (character array) representing an integer into an actual integer value.

Example:

START 1000 ; Initializes LC to 1000

...

WORD 5 ; Reserves 3 bytes and increments LC by 3

...

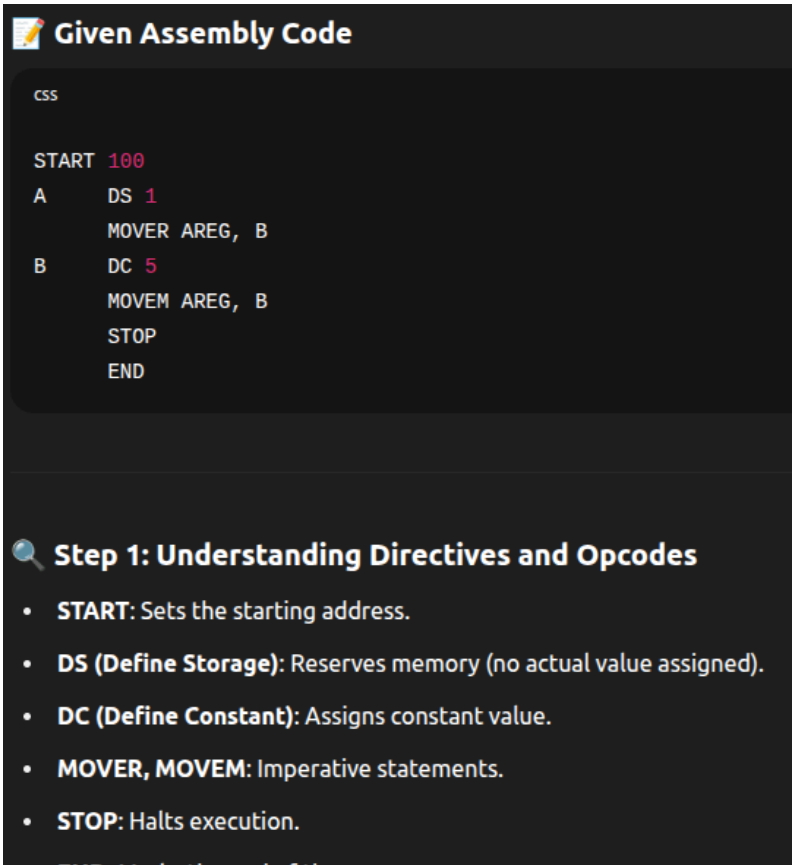
RESW 2 ; Reserves 6 bytes (2 * 3) and increments LC by 6

In this example, `atoi()` would be used to convert "1000", "5", and "2" into integers to be used by the assembler for address calculations.

To compile and run :

gcc pass1.c -o pass1

./pass1



Algorithm:

Algorithm: Pass-1 Assembler

Step 1: Initialize Variables

- Set `symcount = 0` (to track number of symbols).
- Set `lc = 0` (Location Counter).
- Define `optab` as the operation table (mnemonic, class, opcode).
- Define a symbol table array `symtab[]`.

Step 2: Open Required Files

- Open source file `input.asm` for reading.
- Open `intermediate.txt` for writing intermediate code.
- Open `symtab.txt` for writing the symbol table.

Step 3: Read Assembly Code Line by Line

- For each line in `input.asm`:
    1. **Initialize tokens**: Set `label`, `opcode`, `op1`, `op2` to `"-"`.
    2. **Parse the line** using `sscanf` to extract up to four words.
    3. **Adjust tokens**:
        - If label is actually a mnemonic (checked using `optab`), shift tokens accordingly to correctly identify the label, opcode, and operands.

Step 4: Process the Opcode

- Get index of `opcode` from `optab`.
- If **opcode is START**:
    - Set `lc = operand1`.
    - Write `(AD,01) (C,value)` to `intermediate.txt`.
- Else If **opcode is END**:
    - Write `(AD,02)` to `intermediate.txt`.
- Else If **opcode is DS**:
    - Add label to symbol table with current `lc`.
    - Write `(DL,01) (C,size)` to `intermediate.txt`.
    - Increment `lc` by size (from operand).
- Else If **opcode is DC**:

    - Add label to symbol table with current `lc`.
    - Write `(DL,02) (C,value)` to `intermediate.txt`.
    - Increment `lc` by 1.
- Else If **opcode is IS (Imperative Statement)**:
    - If label exists, add it to symbol table with current `lc`.
    - Get register code for operand1.
    - If operand2 (symbol) is not in symbol table:
        - Add it with address `-1` (forward reference).
    - Write `(IS,opcode) (R,reg_code) (S,symbol_index)` to `intermediate.txt`.
    - Increment `lc` by 1.

Step 5: Write Symbol Table

- For each entry in `symtab`:
    - Write `index label address` to `symtab.txt`.

Step 6: Close All Files

- Close input, intermediate, and symbol table files.

Step 7: End Program

- Print `"Pass-I complete"` to console.

4. `B DC 5`

- Label: `B`, constant 5
- Address = 102
- LC = 103
- **Symbol Table:** `B → 102`
- **IC:** `(DL,02) (C,5)`

---

5. `MOVEM AREG, B`

- Imperative Statement (IS), opcode 05

---

7. `END`

- Assembler Directive
- No effect on LC
- **IC:** `(AD,02)`

🗒️ Final Symbol Table

Index	Symbol	Address
1	A	100
2	B	102

📑 Intermediate Code (IC) Output

LC	IC
	(AD,01) (C,100)
100	(DL,01) (C,1)
101	(IS,04) (1) (S,2)
102	(DL,02) (C,5)