Pass-2 Program :

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

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

int symcount = 0;

int get_address_by_index(int idx) {
   for (int i = 0; i < symcount; i++) {
      if (symtab[i].index == idx)
         return symtab[i].address;
   }
   return -1;
}

void load_symtab() {
   FILE *fp = fopen("symtab.txt", "r");
   while (fscanf(fp, "%d %s %d", &symtab[symcount].index, symtab[symcount].label,
&symtab[symcount].address) != EOF) {
      symcount++;
```

```
    }
    fclose(fp);
}

int main() {
    FILE *ic = fopen("intermediate.txt", "r");
    FILE *mc = fopen("machine_code.txt", "w");

    load_symtab();

    char line[100];
    int lc = 100;

    while (fgets(line, sizeof(line), ic)) {
        int opcode, reg, symidx;
        char type[5];

        if (sscanf(line, "(%[^,],%d) (R,%d) (S,%d)", type, &opcode, &reg, &symidx) == 4) {
            int addr = get_address_by_index(symidx);
            fprintf(mc, "%d\t%02d\t%d\t%03d\n", lc, opcode, reg, addr);
            lc++;
        } else if (sscanf(line, "(%[^,],%d)", type, &opcode) == 2 && strcmp(type, "IS") == 0) {
            fprintf(mc, "%d\t%02d\t0\t000\n", lc, opcode);
            lc++;
        } else if (strstr(line, "(DL,02)")) {
            int constant;
            sscanf(line, "(DL,02) (C,%d)", &constant);
```

```
            fprintf(mc, "%d\t00\t0\t%03d\n", lc, constant);
            lc++;
        } else if (strstr(line, "(DL,01)")) {
            int size;
            sscanf(line, "(DL,01) (C,%d)", &size);
            for (int i = 0; i < size; i++) {
                fprintf(mc, "%d\t00\t0\t000\n", lc);
                lc++;
            }
        }
    }

    fclose(ic);
    fclose(mc);
    printf("Pass-II complete.\n");
    return 0;
}
```

Output :

machinecode.txt :

100	00	0	000
101	04	1	102
102	00	0	005
103	05	1	102

Algorithm of Pass 2

- **Initialize**:
    - Load the symbol table and literal table created during Pass 1.
    - Prepare for object code
    - **Process Each Line of Intermediate Code**:
    - For each line in the intermediate representation
    - a. **Parse the line** into label, mnemonic, and operand.
    - b. **Handle Pseudo-Ops** (e.g., START, END, RESW, RESB):
        - Process directives appropriately (e.g., START specifies the starting address).
    - c. **Process Mnemonics**:
        - Search the opcode table for the mnemonic.
        - If found, generate the corresponding machine code.
        - If the operand is a symbol, look it up in the symbol table to get its address.
        - If the operand is a literal, handle it accordingly.
    - d. **Handle Literals**:
        - Assign addresses to literals and include them in the object code.
    - e. **Handle Forward References**:
        - If an operand refers to a label not yet defined, leave a placeholder and backpatch it later.
    - f. **Generate Object Code**:
        - Assemble the object code instruction.
        - If the object code will not fit into the current text record, write the current text record to the object program and initialize a new one.
        - Add the object code to the text record.
- **Finalize**:
    - Write the last text record to the object program.
    - Write the end record to the object program.
    - Write the last listing line.

`%02d`: Prints a signed decimal integer with a minimum width of 2 characters, padding with leading zeros if necessary.

🎬 `%03d`: Prints a signed decimal integer with a minimum width of 3 characters, padding with leading zeros if necessary.

Breakdown of `(%[^,],%d)`

1. `(`: Matches the literal opening parenthesis （ in the input string.
2. `%[^,]`: This is a scanset conversion specifier. It reads characters into a string until it encounters a comma `,`. The `^` inside the brackets negates the set, so it matches any character except a comma.
3. `,`: Matches the literal comma `,` in the input string.
4. `%d`: Reads an integer (signed decimal) from the input string.
5. `)`: Matches the literal closing parenthesis ） in the input string.