System Programming Lab

Name: Swanand. M. Garge

DIV: D(D2)

Roll No.: 39

SRN: 202201589

ASSIGNMENT-2

Q. Design suitable data structures and implement Pass-1 of a two-pass assembler for hypothetical machine. Generate Literal table and Intermediate code file. Implementation should consider

1. **Sample instructions from each category and few assembler directives.**
2. **Forward references**
3. **Error handling: symbol used but not defined, invalid instruction/register etc**

Input File:

```
START 500
MOVEM AREG,10
READ ONES
ADD AREG,21
ADD DREG,='5'
SUB BREG,FIVE
PRINT BREG
LTORG
    ='1'
    ='2'
STOP
ONE DC 1
TWO DS 2
END
```

Directives:

```
START 01
END 02
ORIGIN 03
EQU 04
LTORG 05
```

Keywords:

```
STOP 00
ADD 01
SUB 02
MULT 03
MOVER 04
MOVEM 05
COMP 06
BC 07
DIV 08
READ 09
PRINT 10
```

Conditions:

```
LT 1
LE 2
EQ 3
GT 4
GE 5
ANY 6
```

Registers:

```
AREG 01
BREG 02
CREG 03
DREG 04
```

Code:

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

```
#define max_loc 250
#define max_imperatives 11

int read_asm_file();
int read_key_file();
int read_dir_file();
int read_regs_file();
int read_conditions();
int is_constant(const char *);
void setMemLoc();
int generateIC();
int is_constant(const char *);
int is_register(const char *);
int is_literal(const char *);

struct in_keywords_file
{
    char *mne;
    char opc[2];
} in_keys[max_imperatives];

struct found_imperatives
{
    char *mne;
    char opc[2];
    int memloc;
} f_imps[max_loc];

struct found_symbols
{
    char *name;
    int memloc;
    int f_index;
} f_symbols[max_loc];

struct in_reg_file
{
    char *mne;
    char opc[2];
} in_regs[4];

struct in_directives
{
    char *mne;
```

```
    char opc[2];
} in_dir[5];

struct found_dir
{
    char *mne;
} f_dir[max_loc];

struct pool_table
{
    int index;
} pool_l[max_loc];

struct found_literals
{
    char *val;
    int memLoc;
    int pool_index;
} f_lit[max_loc];

struct conditions
{
    char *mne;
    char opc[1];
} in_con[6];

char **asm_code_file;
FILE *file_ptr;

// declaraing counters
int imperatives_found_count = 0;
int directives_found_count = 0;
int symbol_found_count = 0;
int literals_found_count = 0;
int pool_table_count = 0;
int asm_file_word_count = 0;
int init_mem_loc = 0;
int f_imp_loc = 0;

int read_asm_file()
{
    char arr[20];
    char entered_file[100];
    int word_length;
    char **temp;
```

```
    printf("\nEnter ASM file name to open the file :");
    scanf("%s", &entered_file);

    file_ptr = fopen(entered_file, "r");

    if (file_ptr == NULL)
    {
        printf("\nError opening the file!!");
        return 0;
    }

    while (fscanf(file_ptr, "%s", arr) == 1)
    {
        word_length = strlen(arr);

        // Check for comma in the word
        char *comma_ptr = strchr(arr, ',');
        if (comma_ptr != NULL)
        {
            // Calculate the position of comma in the word
            int comma_position = comma_ptr - arr;

            asm_file_word_count++;
            temp = (char **)realloc(asm_code_file, asm_file_word_count *
sizeof(char *));

            if (temp == NULL)
            {
                printf("Error Allocating the Memory!!");
                return 0;
            }

            asm_code_file = temp;

            // Allocate memory for two separate parts
            asm_code_file[asm_file_word_count - 1] = (char
*)malloc((comma_position + 1) * sizeof(char));
            strncpy(asm_code_file[asm_file_word_count - 1], arr, comma_position);
            asm_code_file[asm_file_word_count - 1][comma_position] = '\0';

            asm_file_word_count++;
            temp = (char **)realloc(asm_code_file, asm_file_word_count *
sizeof(char *));
```

```
            if (temp == NULL)
            {
                printf("Error Allocating the Memory!!");
                return 0;
            }

            asm_code_file = temp;

            asm_code_file[asm_file_word_count - 1] = (char *)malloc((word_length
- comma_position) * sizeof(char));
            strcpy(asm_code_file[asm_file_word_count - 1], comma_ptr + 1); // +1
to skip the comma
        }
        else
        {
            asm_file_word_count++;
            temp = (char **)realloc(asm_code_file, asm_file_word_count *
sizeof(char *));

            if (temp == NULL)
            {
                printf("Error Allocating the Memory!!");
                return 0;
            }

            asm_code_file = temp;

            // No comma found, allocate memory normally
            asm_code_file[asm_file_word_count - 1] = (char *)malloc((word_length)
* sizeof(char));
            strcpy(asm_code_file[asm_file_word_count - 1], arr);
        }
    }
    asm_code_file[asm_file_word_count] = NULL;
}

int read_key_file()
{
    char arr1[10];
    char arr2[2];
    int word_len;
    int counter = 0;

    file_ptr = fopen("keywords.txt", "r");
    if (file_ptr == NULL)
```

```
    {
        printf("\nError Opening the File!!");
        return 0;
    }

    while (fscanf(file_ptr, "%s %s", arr1, arr2) == 2)
    {
        word_len = strlen(arr1);

        in_keys[counter].mne = (char *)malloc(word_len * sizeof(char));

        strcpy(in_keys[counter].mne, arr1);
        strcpy(in_keys[counter].opc, arr2);
        counter++;
    }
}

int read_dir_file()
{
    char arr[10];
    char arr2[2];
    int word_len;
    int counter = 0;

    file_ptr = fopen("directives.txt", "r");
    if (file_ptr == NULL)
    {
        printf("\nError Opening the File!!");
        return 0;
    }

    while (fscanf(file_ptr, "%s %s", arr, arr2) == 2)
    {
        word_len = strlen(arr);

        in_dir[counter].mne = (char *)malloc(word_len * sizeof(char));

        strcpy(in_dir[counter].mne, arr);
        strcpy(in_dir[counter].opc, arr2);
        counter++;
    }
}

int read_regs_file()
{
```

```
    char arr1[4];
    char arr2[2];
    int counter = 0;
    int word_len;

    file_ptr = fopen("regs.txt", "r");
    if (file_ptr == NULL)
    {
        printf("\nError opening the File!!");
        return 0;
    }

    while (fscanf(file_ptr, "%s %s", arr1, arr2) == 2)
    {
        word_len = strlen(arr1);

        in_regs[counter].mne = (char *)malloc(word_len * sizeof(char));

        strcpy(in_regs[counter].mne, arr1);
        strcpy(in_regs[counter].opc, arr2);

        counter++;
    }
}

int read_conditions()
{
    int counter = 0;
    char arr1[4];
    char arr2[2];
    int word_len;

    file_ptr = fopen("conditions.txt", "r");
    if (file_ptr == NULL)
    {
        printf("\nError Opening the File!!");
        return 0;
    }

    while (fscanf(file_ptr, "%s %s", arr1, arr2) == 2)
    {
        word_len = strlen(arr1);

        in_con[counter].mne = (char *)malloc(word_len * sizeof(char));
```

```
        strcpy(in_con[counter].mne, arr1);
        strcpy(in_con[counter].opc, arr2);

        counter++;
    }
}

void setMemLoc()
{
    int i;
    for (i = 0; i < asm_file_word_count; i++)
    {
        // 0 if true
        if (strcmp(asm_code_file[i], "START") == 0)
        {
            init_mem_loc = atoi(asm_code_file[i + 1]);
            break;
        }
    }
}

int is_literal(const char *str)
{
    char *ptr = strchr(str, '=');

    if (ptr == NULL)
    {
        return 0;
    }
    return 1;
}

int checkType()
{
    int i;
    int j;
    int counter;
    int word_len;
    int found_flag = 0;
    char *eq_ptr = NULL;
    int prs_lit[max_loc];
    memset(prs_lit, 0, sizeof(prs_lit));
    int current_pool_index = 0;

    for (i = 0; i < asm_file_word_count; i++)
```

```
    {
        eq_ptr = NULL;
        word_len = strlen(asm_code_file[i]);

        // check symbol
        // this block is the decalaration block
        // the declared symbol is added to the symbol_table
        if ((i + 1) < asm_file_word_count &&
            (strcmp(asm_code_file[i + 1], "DS") == 0 || strcmp(asm_code_file[i +
1], "DC") == 0))
        {
            f_symbols[symbol_found_count].name = (char *)malloc(word_len *
sizeof(char));

            strcpy(f_symbols[symbol_found_count].name, asm_code_file[i]);
            f_symbols[symbol_found_count].memloc = init_mem_loc;
            f_symbols[symbol_found_count].f_index = symbol_found_count;

            init_mem_loc++;
            symbol_found_count++;
            i++;
            continue;
        }

        // check for imperative
        for (j = 0; j < max_imperatives; j++)
        {
            if (strcmp(asm_code_file[i], in_keys[j].mne) == 0)
            {
                // Found an imperative
                f_imps[imperatives_found_count].mne = (char *)malloc((word_len) *
sizeof(char));
                strcpy(f_imps[imperatives_found_count].mne, in_keys[j].mne);

                strcpy(f_imps[imperatives_found_count].opc, in_keys[j].opc);
                f_imps[imperatives_found_count].memloc = init_mem_loc;

                imperatives_found_count++;

                // check for literal
                eq_ptr = strchr(asm_code_file[i + 2], '=');

                if (is_literal(asm_code_file[i + 2]) == 1 && !prs_lit[i + 2])
                {
```

```
                    f_lit[literals_found_count].val = (char
*)malloc((strlen(asm_code_file[i + 2])) * sizeof(char));

                    strcpy(f_lit[literals_found_count].val, asm_code_file[i +
2]);
                    f_lit[literals_found_count].memLoc = init_mem_loc;
                    f_lit[literals_found_count].pool_index = pool_table_count;

                    pool_l[pool_table_count].index = pool_table_count;

                    literals_found_count++;
                    pool_table_count++;
                    prs_lit[i + 2] = 1;
                }

                init_mem_loc++;
                eq_ptr = NULL;
                break; // Exit the loop after finding an imperative
            }
        }

        // check directive
        for (j = 0; j < 5; j++)
        {
            if (strcmp(asm_code_file[i], in_dir[j].mne) == 0)
            {
                f_dir[directives_found_count].mne = (char *)malloc(word_len *
sizeof(char));
                strcpy(f_dir[directives_found_count].mne, asm_code_file[i]);

                directives_found_count++;

                if (strcmp(asm_code_file[i], "LTORG") == 0)
                {

                    if (current_pool_index == -1)
                    {
                        // Increment pool index for the first literal
                        pool_table_count++;
                        current_pool_index = pool_table_count - 1;
                    }

                    for (int x = i + 1; x < asm_file_word_count; x++)
                    {
                        if (is_literal(asm_code_file[x]) == 0)
```

```
                        {
                            break;
                        }
                        else if (is_literal(asm_code_file[x]) == 1 &&
!prs_lit[x])
                        {

                            f_lit[literals_found_count].val = (char
*)malloc(strlen(asm_code_file[x]) * sizeof(char));
                            strcpy(f_lit[literals_found_count].val,
asm_code_file[x]);
                            f_lit[literals_found_count].pool_index =
current_pool_index;
                            f_lit[literals_found_count].memLoc = init_mem_loc;

                            pool_l[current_pool_index].index =
current_pool_index;

                            prs_lit[x] = 1;
                            literals_found_count++;
                            init_mem_loc++;
                        }
                    }

                    current_pool_index = -1; // Reset
                }
                break;
            }
        }

        // a literal is found
        // add it to the pool table and found literal table
        if (is_literal(asm_code_file[i]) == 1 && !prs_lit[i])
        {
            f_lit[literals_found_count].val = (char *)malloc(word_len *
sizeof(char));

            strcpy(f_lit[literals_found_count].val, asm_code_file[i]);
            f_lit[literals_found_count].memLoc = init_mem_loc;
            f_lit[literals_found_count].pool_index = pool_table_count;

            pool_l[pool_table_count].index = pool_table_count;

            literals_found_count++;
            pool_table_count++;
```

```
            prs_lit[i] = 1;
        }
    }
}

void printMenu()
{
    printf("\n1>Show IC\n2>Print Literal Table\n3>Exit");
    printf("\nEnter Your Choice :");
}


void printLit()
{
    printf("\nLiteral Table\n");
    printf("\nValue \tMemoryLoc   Pool Index");
    for (int i = 0; i < literals_found_count; i++)
    {
        printf("\n----------------------");
        printf("\n%s\t%d\t%d\n", f_lit[i].val, f_lit[i].memLoc,
f_lit[i].pool_index);
    }

    printf("\nPool Table:\t\n");
    printf("Literal\n Index");
    for (int i = 0; i < pool_table_count; i++)
    {
        printf("\n-------");
        printf("\n%d", pool_l[i].index);
    }
    printf("\n");
}

int printIC()
{
    FILE *fptr;
    char line[100];

    fptr = fopen("ic_gen.txt", "r");
    if (fptr == NULL)
    {
        printf("\nError Opening the File!!");
        return 0;
    }
```

```
    while (fgets(line, sizeof(line), fptr) != NULL)
    {
        printf("%s", line);
    }
}

int generateIC()
{
    FILE *fptr;
    int i;
    int j;
    int status_flag;
    int fl = 0;

    fptr = fopen("ic_gen.txt", "w");
    if (fptr == NULL)
    {
        printf("\nError Opening the File!!");
        return 0;
    }
    for (i = 0; i < asm_file_word_count; i++)
    {
        status_flag = 0;
        fl = 0;
        //  print Symbol
        for (j = 0; j < symbol_found_count; j++)
        {
            if (strcmp(asm_code_file[i], f_symbols[j].name) == 0)
            {
                if ((strcmp(asm_code_file[i + 1], "DS") == 0) ||
(strcmp(asm_code_file[i + 1], "DC") == 0))
                {
                    if (strcmp(asm_code_file[i + 1], "DS") == 0)
                    {
                        fprintf(fptr, "\n( DL,02 )");
                    }
                    else if (strcmp(asm_code_file[i + 1], "DC") == 0)
                    {
                        fprintf(fptr, "\n( DL,01 )");
                    }
                    i++;
                }
                else
                {
                    // this is the usage part
```

```
                fprintf(fptr, "\t( S,%d )", f_symbols[j].f_index);
            }
            status_flag = 1;
            fl = 1;
            break;
        }
        else
        {
            fl = 0;
        }
    }

    // print imperatives
    for (j = 0; j < max_imperatives; j++)
    {
        if (strcmp(asm_code_file[i], in_keys[j].mne) == 0)
        {
            // if yes print it
            fprintf(fptr, "\n( IS,%s )", in_keys[j].opc);
            status_flag = 1;
            // BC encountered
            if (j == 7)
            {
                for (int c = 0; c < 6; c++)
                {
                    if (strcmp(asm_code_file[i + 1], in_con[c].mne) == 0)
                    {
                        fprintf(fptr, "(%s)", in_con[c].opc);
                        break;
                    }
                }
            }
        }
    }

    // print directives
    for (j = 0; j < 5; j++)
    {
        if (strcmp(asm_code_file[i], in_dir[j].mne) == 0)
        {
            fprintf(fptr, "\n( AD,%s )", in_dir[j].opc);
            status_flag = 1;
            break;
        }
    }
```

```
        // print Literals
        for (j = 0; j < literals_found_count; j++)
        {
            if (strcmp(asm_code_file[i], f_lit[j].val) == 0)
            {
                fprintf(fptr, "\t( L,%s )\n", f_lit[j].val);
                status_flag = 1;
                break;
            }
        }

        // print registers
        for (j = 0; j < 4; j++)
        {
            if (strcmp(asm_code_file[i], in_regs[j].mne) == 0)
            {
                fprintf(fptr, "( %s )", in_regs[j].opc);
                status_flag = 1;
                break;
            }
        }

        // print constant
        if (is_constant(asm_code_file[i]) == 1)
        {
            fprintf(fptr, "( C,%s )", asm_code_file[i]);
            status_flag = 1;
            // continue;
        }

        if (status_flag == 0 && fl == 0)
        {
            fprintf(fptr, "\t(*ERROR-INVALID INPUT :%s)", asm_code_file[i]);
        }
    }
    fprintf(fptr, "\n");
    fclose(fptr);
}

int is_constant(const char *str)
{
    for (size_t i = 0; str[i] != '\0'; i++)
    {
        if (!isdigit(str[i]))
```

```
        {
            return 0; // If any non-digit character is found, return false
        }
    }
    return 1; // If all characters are digits, return true
}

int is_register(const char *str)
{
    for (int i = 0; i < 4; i++)
    {
        if (strcmp(str, in_regs[i].mne) == 0)
        {
            return 1; // if reg return 1
        }
    }
    // not a register
    return 0; // else return 0
}

int main()
{
    int ch;
    read_key_file();
    read_asm_file();
    read_dir_file();
    read_regs_file();
    read_conditions();
    setMemLoc();
    checkType();
    generateIC();

    do
    {
        printMenu();
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            system("cls");
            printIC();
            break;

        case 2:
```

```
            system("cls");
            printLit();
            break;

        case 3:
            exit(0);
            break;

        default:
            break;
        }
    } while (ch != 3);
}
```

Output:

Literal & Pool Table-

```
Literal Table

Value  MemoryLoc  Pool Index
------------------
='5'    506      0

------------------
='1'    511      1

------------------
='2'    512      2

Pool Table:
Literal
 Index
-------
0
-------
1
```

Intermediate Code-

```
( AD,01 )( C,500 )
( IS,05 )( 01 )( C,10 )
( IS,09 )       ( S,0 )
( IS,01 )( 01 )( C,21 )
( IS,01 )( 02 )( L,='5' )

( IS,10 )( 02 )
( AD,05 )( L,='1' )
( L,='2' )

( IS,00 )
( DL,01 )( C,1 )
( DL,02 )( C,2 )
( AD,02 )
```

ic_gen.txt file:

```
( AD,01 )( C,500 )
( IS,05 )( 01 )( C,10 )
( IS,09 ) ( S,0 )
( IS,01 )( 01 )( C,21 )
( IS,01 )( 02 )( L,='5' )
( IS,10 )( 02 )
( AD,05 )( L,='1' )
( L,='2' )
( IS,00 )
( DL,01 )( C,1 )
( DL,02 )( C,2 )
( AD,02 )
```