System Programming Lab

Name: Swanand Garge

Div : D(D2)

Roll no :39

SRN: 202201589

ASSIGNMENT-1

1. 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>
```

```c
#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 = -1;
```

```
    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
                        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 Symbol Table\n3>Exit");
    printf("\nEnter Your Choice :");
}

void printSym()
{
    printf("Symbol \nName\tAddress\n");
    for (int i = 0; i < symbol_found_count; i++)
    {
        if (f_symbols[i].memloc != -1)
        {
            printf("\n------------");
            printf("\n%s\t%d", f_symbols[i].name, f_symbols[i].memloc);
        }
    }
}

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, "( L,%s )\n", f_lit[j].val);
            status_flag = 1;
            break;
```

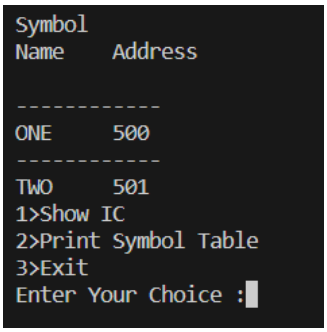
```
            }
        }

        // 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");
            printSym();
            break;

        case 3:
            exit(0);
            break;

        default:
```

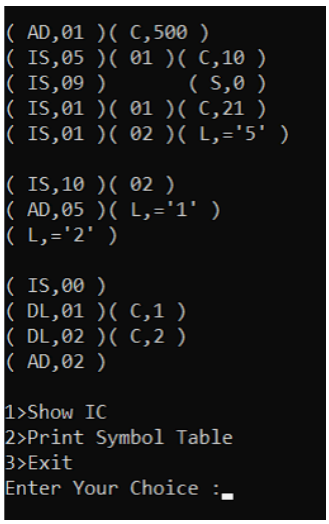
```
            break;
        }
    } while (ch != 4);
}
```

Output:

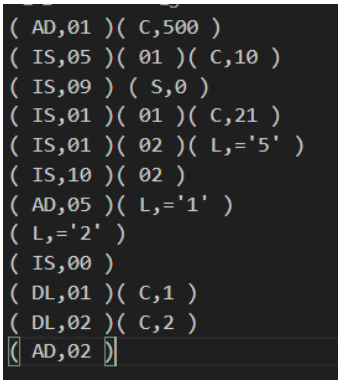
Symbol Table-



Intermediate Code-



ic_gen.txt file:



Code Explanation:

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

These lines include standard C libraries that the program will use. `stdio.h` for standard input/output, `stdlib.h` for memory allocation and other utilities, `string.h` for string manipulation functions, and `ctype.h` for character handling functions.

#define max_loc 250
#define max_imperatives 11

These lines define two macros: `max_loc` and `max_imperatives`. These macros are used to set maximum limits for arrays and loops in the program.

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 *);

These lines declare function prototypes for various functions that will be defined later in the code. These functions are responsible for reading input files, generating intermediate code, and performing various tasks.

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

This line defines a structure named `in_keywords_file` to hold keyword information. It has two members: `mne` (for the keyword itself) and `opc` (a two-character array for the opcode). An array `in_keys` of these structures is also declared to store keyword data.

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

This line defines a structure named `found_imperatives` to store information about found imperative instructions. It has members for the imperative's name (`mne`), opcode (`opc`), and memory location (`memloc`). An array `f_imps` of these structures is declared to store found imperative data.

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

This line defines a structure named `found_symbols` to store information about found symbols. It has members for the symbol's name (`name`), memory location (`memloc`), and an index (`f_index`). An array `f_symbols` of these structures is declared to store found symbol data.

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

This line defines a structure named `in_reg_file` to hold register information. It has members for the register's name (`mne`) and opcode (`opc`). An array `in_regs` of these structures is declared to store register data. It's assumed there are four registers.

```
struct in_directives
{
   char *mne;
   char opc[2];
} in_dir[5];
```

This line defines a structure named `in_directives` to store information about directives. It has members for the directive's name (`mne`) and opcode (`opc`). An array `in_dir` of these structures is declared to store directive data. It's assumed there are five directives.

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

This line defines a structure named `found_dir` to store information about found directives. It has a member for the directive's name (`mne`). An array `f_dir` of these structures is declared to store found directive data.

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

This line defines a structure named `pool_table` with a single member `index`. An array `pool_l` of these structures is declared to store pool table data.

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

This line defines a structure named `found_literals` to store information about found literals. It has members for the literal's value (`val`), memory location (`memLoc`), and pool index (`pool_index`). An array `f_lit` of these structures is declared to store found literal data.

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

This line defines a structure named `conditions` to store information about conditions. It has members for the condition's name (`mne`) and opcode (`opc`). An array `in_con` of these structures is declared to store condition data. It's assumed there are six conditions.

```
char **asm_code_file;
FILE *file_ptr;
```

These lines declare global variables `asm_code_file`, which is a double pointer to char (used for storing the tokenized input code), and `file_ptr`, which is a file pointer used for file operations.

```
// declaring 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;
```

These lines declare global counters for various elements found in the input code, including imperatives, directives, symbols, literals, pool tables, and others.

```
int read_asm_file()
{
   // ...
}
```

This defines the `read_asm_file` function, which is responsible for reading the assembly file provided by the user. It tokenizes the file contents and stores them in the `asm_code_file` array, handling comma-separated words.

```
int read_key_file()
{
   // ...
}
```

This defines the `read_key_file` function, which reads a file named "keywords.txt" containing keywords and their opcodes. It populates the `in_keys` array with this information.

```
int read_dir_file()
{
   // ...
}
```

This defines the `read_dir_file` function, which reads a file named "directives.txt" containing directives and their opcodes. It populates the `in_dir` array with this information.

```
int read_regs_file()
{
   // ...
}
```

This defines the `read_regs_file` function, which reads a file named "regs.txt" containing register names and their opcodes. It populates the `in_regs` array with this information.

```
int read_conditions()
{
   // ...
}
```

This defines the `read_conditions` function, which reads a file named "conditions.txt" containing condition names and their opcodes. It populates the `in_con` array with this information.

```
void setMemLoc()
{
   // ...
}
```

This defines the `setMemLoc` function, which sets the initial memory location based on the "START" directive in the input code.

```
int is_literal(const char *str)
{
    // ...
}
```

This defines the `is_literal` function, which checks if a given string represents a literal (e.g., "=5").

```
int checkType()
{
    // ...
}
```

This defines the `checkType` function, which processes the assembly code and extracts information about symbols, imperatives, directives, literals, and more. It populates various data structures with this information.

```
void printMenu()
{
    // ...
}
```

This defines the `printMenu` function, which displays a menu for the user to choose options, such as displaying intermediate code (IC), printing the symbol table, or exiting the program.

```
void printSym()
{
    // ...
}
```

This defines the `printSym` function, which prints the symbol table, showing symbol names and their associated addresses.

```
int printIC()
{
    // ...
}
```

This defines the `printIC` function, which reads and prints the generated intermediate code from the "ic_gen.txt" file.

```
int generateIC()
{
   // ...
}
```

This defines the `generateIC` function, which generates intermediate code (IC) based on the parsed assembly code and writes it to the "ic_gen.txt" file. The IC format consists of various tokens like symbols, imperatives, literals, directives, registers, and constants.

```
int is_constant(const char *str)
{
   // ...
}
```

This defines the `is_constant` function, which checks if a given string is a constant (contains only digits).

```
int is_register(const char *str)
{
   // ...
}
```

This defines the `is_register` function, which checks if a given string is a register by comparing it to the register names in `in_regs`.

```
int main ()
{
   // ...
}
```

This is the `main` function, the entry point of the program. It calls various functions to read configuration files, parse the input assembly code, generate IC, and interact with the user through a menu. Users can choose to display the IC, print the symbol table, or exit the program.

Overall, the code is organized into functions that perform specific tasks such as reading files, parsing code, and generating intermediate code. It also defines several data structures to store information about keywords, symbols, directives, and other elements encountered in the assembly-like code.