EXPERIMENT NO:-3
FILE DIRECTORIES

Aim:- To implement file organization techniques- single level, two-level & hierarchical file directories.

Theory:- Single level file directory is the simplest of all directory structures having only one directory that consist of all the files. Sometimes, it is said to be the root directory.

The problem in single level directory is different users may be accidentally using the same names for their files. To avoid this problem, each user need a private directory. In this way, names chosen by one user don't interfere with names chosen by a different user. which is accomplished by two level directory.

Hierarchichal directories create subdirectory and load same type of files into subdirectory, which is satisfactory for users with large number of files. Each can have as many directories as possible.

Algorithm for Single-Level Directory File Organization Technique:-

Step 0:- Start the program.
Step 1:- Create a directory structure to store a single directory and multiple files.
Step 2:- Enter a directory name.

Step 3:- Create file operation is performed.
3.1:- Accept filename.
3.2:- Increment file count.
3.3:- Update the file information table.
Step 4:- Delete file Operation is performed.
4.1:- Accept the filename to be deleted.
4.2:- Compare file names with names of existing files.
4.3:- If a match is found, then
4.3.1:- Delete the file by updating file information table.
4.3.2:- Decrement file count.
4.4:- Otherwise, display 'file not found'.
Step 5:- Search file operation is performed.
5.1:- Accept the name of the file.
5.2:- Compare filename with names of existing files.
5.3:- If a match is found, display 'file is found'.
5.4:- Otherwise, display 'file not found'.
Step 6:- Display files.
6.1:- Check if directory is empty.
6.2:- If yes, print 'Directory empty'.
6.3:- Otherwise, print information of each file from file information table.
Step 7:- Stop the program.

Algorithm for Two-Level File Directory:-

Step 0:- Start the program.
Step 1:- Create a Structure to store details of multiple directories & multiple files for each of directories.
Step 2:- Create directory
  2.1:- Accept the directory name
  2.2:- Increment directory count.
  2.3:- Update directory information table.
  2.4:- Display 'Directory Created'.
Step 3:- Create file.
  3.1:- Accept directory name.
  3.2:- Compare directory name with existing directory names.
  3.3:- If match is found, then
    3.3.1:- Accept file name
    3.3.2:- Increment file count for this directory.
    3.3.3:- Update corresponding file information table.
  3.4:- Otherwise, print 'directory not found'.
Step 4:- Delete file
  4.1:- Accept the directory name.
  4.2:- Compare the directory names with names
  4.3:- If match is found, then
    4.3.1:- Accept filename to be deleted.
    4.3.2:- Compare the filename with names of existing files in this directory.
    4.3.3:- If a match is found, then delete file

by updating corresponding file information table and decrement file count for this directory.

4.3.4:- Otherwise, display 'file not found!'.

4.4:- Otherwise display 'directory not found'.

Step 5:- Search file

5.1:- Accept the directory name.

5.2:- Compare directory name with names of existing directories.

5.3:- If match is found, then

5.3.1:- Accept name of file to be searched.

5.3.2:- Compare filename with names of existing files in this directory.

5.3.3:- If a match is found then display 'file ~~not~~ found'.

5.3.4:- Otherwise display 'file not found'.

5.4:- Otherwise display 'Directory Not found'.

Step 6:- Display files

6.1:- Accept the directory name.

6.2:- Compare the directory names with names of existing directories

6.3:- If match is found, then

6.3.1:- Check if directory is empty.

6.3.2:- If yes, print 'Directory empty'.

6.3.3:- Otherwise display file information in that directory.

6.4:- Otherwise display 'Directory Not found'.

Step 7:- Stop the program.

Result :- C programs for the simulation of different file organization techniques have been implemented.

EXPERIMENT NO:-4

PASS 1 OF TWO PASS ASSEMBLER.

Aim:- To implement pass 1 of two pass assembler.

Theory:- Pass 1 assign addresses to all statements in the program, save the addresses assigned to all labels for use in pass 2, perform some processing of assembler directives, including those for address assignment, such as BYTE, RESW, etc.

Algorithm:-

```
begin
    read first input line
    if OPCODE = 'START' then
        begin
            save #[OPERAND] as starting address
            initialize LOCCTR to starting address
            write line to intermediate file.
            read next input line.
        end (if START).
    else
        initialize LOCCTR to 0.
    while OPCODE ≠ 'END' do
        begin
            if this is not a comment line then
                begin
```

```
. if there is a symbol in LABEL field then,
  begin
      seach SYMTAB for LABEL.
      if found then
          set error flag (duplicate symbol)
      else
          insert (LABEL, LOCCTR) into SYMTAB
  end if symbol
seach OPTAB for OPCODE
if found then
      add 3 {instruction length} to LOCCTR.
else if  OPCODE = 'WORD' then
      add 3 to LOCCTR.
else if OPCODE = 'RESW' then
      add 3 * #[OPERAND] to LOCCTR.
else if  OPCODE = 'RESB' then
      add #[OPERAND] to LOCCTR.
else if  OPCODE = 'BYTE' then
    begin
        find length of constant in bytes.
        add length to LOCCTR.
    end {if BYTE}
else
      set error flag {invalid operation code}
end {if {not a comment}}
Write line to intermediate file.
read next input line.
end {while not END}
```

Write last line to intermediate file.
Save (LOCCTR - starting address) as program length.
end {Pass 1}.

Result:- C program for implementation of pass 1 of two pass assembler is successfully executed.

EXPERIMENT NO:-5

PASS 2 OF TWO PASS ASSEMBLER

Aim:- To implement pass 2 of a two pass assembler.

Theory:- Pass 2 assemble instructions and generate object program and look up addresses, generate data values defined by BYTE, WORD, perform processing of assembler directives not done during Pass 1, write object program and assembly listing.

Algorithm:-

```
begin
    read first input line {from intermediate file}
    if OPCODE = 'START' then
    begin
        write listing line
        read next input line
    end {if START}
    write Header record to object program
    initialize first Text record.
    while OPCODE ≠ 'END' do
    begin
        if this is not a comment line then
        begin
            seauch OPTAB for OPCODE
            if found then
```

```
            store symbol value as operand values.
            else
            begin
                store 0 as operand address.
                set error flag {undefined symbol}
            end
        end {if symbol}
        else
            store 0 as operand address
            assemble object code instruction.
        end {if ~OPCODE found}
    else if OPCODE = 'BYTE' OR 'WORD' then
        convert constant to object code.
    if object code will not fit into current
    Text record then
        begin
            write Text record to object program.
            initialize new Text record.
        end.
        add object code to Text Record
end (if not comment)
write listing line
read next input line
end {while not END}
Write last Text Record to object program.
Write End Record to object program.
Write last listing line.
end {Pass 2}.
```

Result :- C program for pass 2 of two pass assembler has been implemented successfully.

EXPERIMENT NO:-6

ONE PASS MACRO PROCESSOR

Aim:- To implement one pass macro processor.

Theory:- A one-pass macro processor uses only one pass for processing macro definitions and macro expansions. It can handle nested macro definitions. To implement one pass macro processor, the definition of a macro must appear in the source program before any statements that invoke macro.

Algorithm:-

```
begin
    EXPANDING := FALSE
    while OPCODE ≠ 'END' do
     begin
          GETLINE
          PROCESSLINE
     end
end
procedure PROCESSLINE
begin
    Seach NAMTAB for OPCODE
    if found then
       EXPAND
    else if OPCODE = 'MACRO' then
```

```
        DEFINE
    else write source line to expanded file
end.
procedure DEFINE
begin
    enter macro name into NAMTAB
    enter macro prototype into DEFTAB
    LEVEL := 1
    while LEVEL > 0 do
    begin
        GETLINE
        if this is not a comment line then
        begin
            substitute positional notation for parameters
            enter line into DEFTAB
            if OPCODE = 'MACRO' then
                LEVEL := LEVEL + 1
            else if OPCODE = 'MEND' then
                LEVEL := LEVEL - 1;
        end {if not comment}
    end {while}
    store in NAMTAB pointers to beginning and end
      of definition.
end {DEFINE}
procedure EXPAND
    begin
        EXPANDING := TRUE
        get first line of macro definition {prototype} from
```

DEFTAB:
set up arguments from macro invocation in ARGTAB
write macro invocation to expanded file as a comment.
while not end of macro definition do
begin
GETLINE
PROCESSLINE
end {while}
EXPANDING := FALSE
end {EXPAND}
procedure GETLINE
begin
if EXPANDING then
begin
get next line of macro definition from DEFTAB
Substitute arguments from ARGTAB for positional notation
end {if}
else
read next line from input file
end {GETLINE}.

Result:- C program for the implementation of one pass macro processor has been implemented.

EXPERIMENT NO:-7
ABSOLUTE LOADER

Aim:- To implement absolute loader.

Theory:- Absolute loader is a kind of loader in which relocated object files are created, loader accepts these files and places them at specified locations in memory. No relocation information is needed. It is obtained from the programmer or assembler.

Algorithm:-

```
begin
    Read Header record
    verify program name and length
    read first Text record.
    while record type ≠ E
    begin
        // if object code is in character form, convert it
        // into internal representation.
        move object code to specified location in
        memory.
        read next object program record.
    end
    jump to address specified in End record.
end.
```

Result:- C program for the implementation of absolute loader has been successfully implemented.

EXPERIMENT NO:-8
RELOCATING LOADER

Aim:- To implement relocating loader.

Theory:- Loaders that has the capability of performing relocation are called relocating loaders or relative loaders. The need for program location is an indirect consequence of change to larger and more powerful computers. There are two methods for specifying relocation in object program :- Modification record & Relocation Bit.

Algorithm:-

```
begin
    get PROGADDR from operating system
    While not end of input do
      begin
          read next record
          while record type ≠ 'E' do
          while record type = 'T'
          begin
              get length = second data
              mask bits(M) as third data
              For(i=0, i< length, i++)
                if Mi=1 then
                add PROGADDR at location PROGADR+
```

specified address.

else

move object code from record to location

PROGADDR + specified address.

read next record.

end

end

end.

Result:- C program for the implementation of relocating loader has been implemented successfully.

EXPERIMENT NO:-9
ADDITION & SUBTRACTION OF 16 BIT NUMBERS

Aim:- To perform addition & subtraction of 16-bit numbers using the kit.

Theory:- 8086 is a 16-bit register. We can simply take the numbers from memory to AX and BX registe. then add them using ADD instruction. When Carry is present store carry into memory, otherwise only store AX into memory.
For subtraction, we can simply take the numbers from memory to AX and BX register, then subtract them using SUB instruction.

Algorithm for Addition of 16-bit numbers:-

Step 0:- Start the program
Step 1:- Initialise the datasegment memory.
Step 2:- Initialise the extra segment memory.
Step 3:- Load the first number into AX register
Step 4:- Add two numbers.
Step 5:- Store the result in extra segment.
Step 6:- Stop the program.

Algorithm for Subtraction of 16 bit numbers :-

Step 0 :- Start the program.
Step 1 :- Initialise the data & extra segment memory.
Step 2 :- Subtract AX from OPR1.
Step 3 :- Store result in extra segment.
Step 4 :- Verify the result.
Step 5 :- Stop the program.

Result :- Assembly programs to perform addition & subtraction of 16-bit numbers have been implemented successfully.

Date _____

Expt. No. _____ Page No. _____

EXPERIMENT NO:-11
SEARCHING A NUMBER

Aim:- To write an assembly program to perform searching using 8086 emulator.

Theory:- We will be writing an assembly language program in 8086 microprocessor to search a number in a string of 5 bytes, store the offset where the element is found and number of iterations used to find the number.

Algorithm:-

Step 0:- Start the program.
Step 1:- Set array location to SI
Step 2:- Set count value to CX.
Step 3:- Set search value to SE.
Step 4:- Compare AL & BL.
Step 5:- Jump to FO if zero flag is set.
Step 6:- Else increment SI and decrement CX.
Step 7:- If CX not zero, jump to UP.
Step 8:- Else print msg2.
Step 9:- Jump to END1.
Step 10:- Stop the program.

Teacher's Signature _____