PART B
EXPERIMENT NUMBER 8

Aim: To design and implement the second pass of a two-pass assembler for IBM 360/370 Processor.

(PART B: TO BE COMPLETED BY STUDENTS)

(Students must submit the soft copy as per the following segments within two hours of the practical. The soft copy must be uploaded at the end of the practical)

Roll No. 50	**Name:** AMEY THAKUR
Class: Comps TE B	**Batch:** B3
Date of Experiment: 30/04/2021	**Date of Submission:** 30/04/2021
Grade:	

B.1 Software Code written by a student:
(Paste your code completed during the 2 hours of practice in the lab here)

- **SPCC-8.C**

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
        char a[10], ad[10], label[10], opcode[10], operand[10], symbol[10], ch;
        int st, diff, i, address, add, len, actual_len, finaddr, prevaddr, j=0;
        char mnemonic[15][15]={"LDA","STA","LDCH","STCH"};
        char code[15][15]={"33","44","53","57"};
        FILE *fp1,*fp2,*fp3,*fp4;

        fp1=fopen("ASSMLIST.DAT","w");
        fp2=fopen("SYMTAB.DAT","r");
        fp3=fopen("INTERMED.DAT","r");
        fp4=fopen("OBJCODE.DAT","w");
        fscanf(fp3,"%s%s%s",label,opcode,operand);

 while(strcmp(opcode,"END")!=0)
 {
        prevaddr=address;
        fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
```

```
}
finaddr=address;
fclose(fp3);
fp3=fopen("INTERMED.DAT","r");

fscanf(fp3,"%s%s%s",label,opcode,operand);
if(strcmp(opcode,"START")==0)
{
        fprintf(fp1,"\t%s\t%s\t%s\n",label,opcode,operand);
        fprintf(fp4,"H^%s^00%s^00%d\n",label,operand,finaddr);
        fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
        st=address;
        diff=prevaddr-st;
        fprintf(fp4,"T^00%d^%d",address,diff);
}
while(strcmp(opcode,"END")!=0)
{
        if(strcmp(opcode,"BYTE")==0)
        {
                fprintf(fp1,"%d\t%s\t%s\t%s\t",address,label,opcode,operand);
                len=strlen(operand);
                actual_len=len-3;
                fprintf(fp4,"^");
                for(i=2;i<(actual_len+2);i++)
                {
                        itoa(operand[i],ad,16);
                        fprintf(fp1,"%s",ad);
                        fprintf(fp4,"%s",ad);
                 }
                 fprintf(fp1,"\n");
        }
        else if(strcmp(opcode,"WORD")==0)
         {
                len=strlen(operand);
                itoa(atoi(operand),a,10);
  fprintf(fp1,"%d\t%s\t%s\t%s\t00000%s\n",address,label,opcode,operand,a);
  fprintf(fp4,"^00000%s",a);
 }
else if((strcmp(opcode,"RESB")==0)||(strcmp(opcode,"RESW")==0))
fprintf(fp1,"%d\t%s\t%s\t%s\n",address,label,opcode,operand);
else
{
        while(strcmp(opcode,mnemonic[j])!=0)
```

```
            j++;
           if(strcmp(operand,"COPY")==0)

fprintf(fp1,"%d\t%s\t%s\t%s\t%s0000\n",address,label,opcode,operand,code[j]);
    else
    {
     rewind(fp2);
     fscanf(fp2,"%s%d",symbol,&add);
      while(strcmp(operand,symbol)!=0)
       fscanf(fp2,"%s%d",symbol,&add);

fprintf(fp1,"%d\t%s\t%s\t%s\t%s%d\n",address,label,opcode,operand,code[j],add
);
     fprintf(fp4,"^%s%d",code[j],add);
    }
   }
   fscanf(fp3,"%d%s%s%s",&address,label,opcode,operand);
  }
  fprintf(fp1,"%d\t%s\t%s\t%s\n",address,label,opcode,operand);
  fprintf(fp4,"\nE^00%d",st);
  printf("\n Intermediate file is converted into object code");
  fcloseall();

  printf("\n\nThe contents of Intermediate file:\n\n\t");
  fp3=fopen("INTERMED.DAT","r");
  ch=fgetc(fp3);
  while(ch!=EOF)
  {
   printf("%c",ch);
   ch=fgetc(fp3);
  }
  printf("\n\nThe contents of Symbol Table :\n\n");
  fp2=fopen("SYMTAB.DAT","r");
  ch=fgetc(fp2);
  while(ch!=EOF)
  {
   printf("%c",ch);
   ch=fgetc(fp2);
  }
  printf("\n\nThe contents of Output file :\n\n");
  fp1=fopen("ASSMLIST.DAT","r");
  ch=fgetc(fp1);
  while(ch!=EOF)
```

```
 {
  printf("%c",ch);
  ch=fgetc(fp1);
 }
 printf("\n\nThe contents of Object code file :\n\n");
 fp4=fopen("OBJCODE.DAT","r");
 ch=fgetc(fp4);
 while(ch!=EOF)
 {
  printf("%c",ch);
  ch=fgetc(fp4);
 }
 fcloseall();
 getch();
}
```

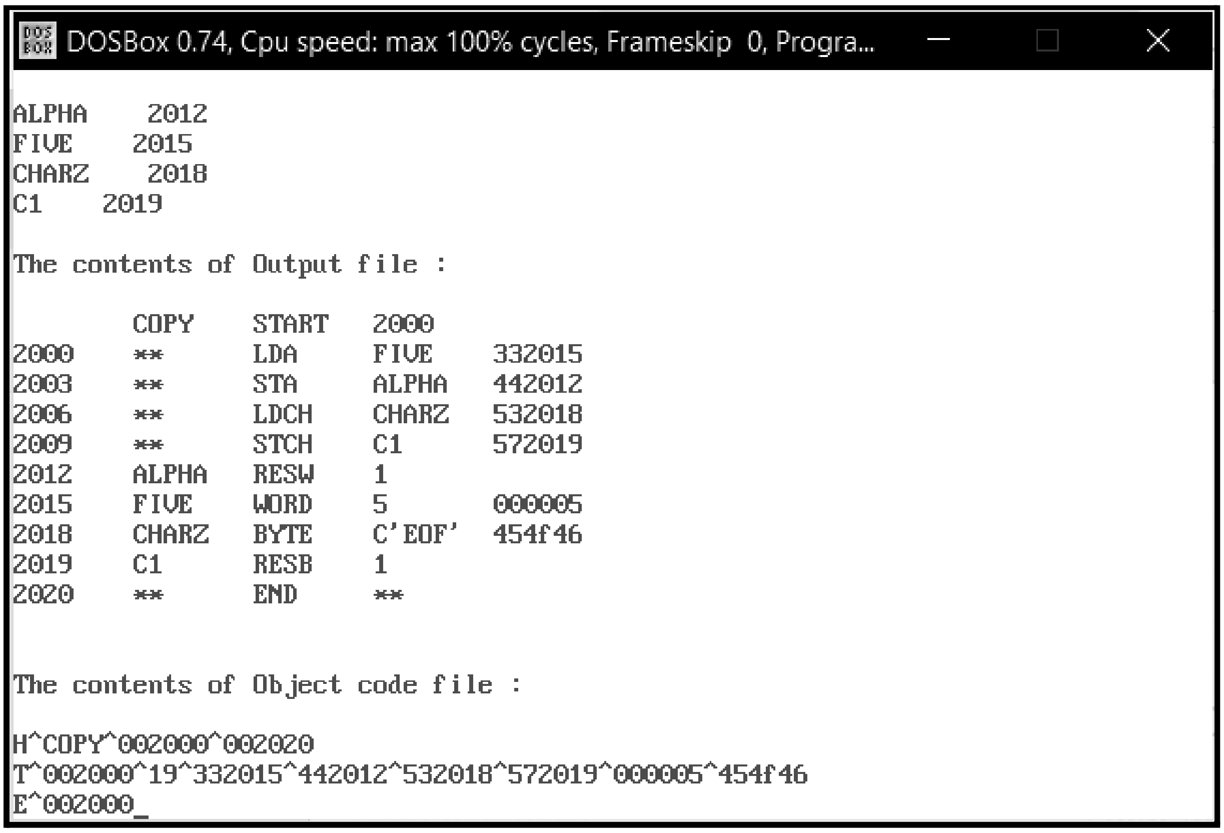
- **Symtab.dat**

```
ALPHA   2012
FIVE   2015
CHARZ   2018
C1   2019
```

- **Intermed.dat**

```
COPY   START   2000
2000   **   LDA   FIVE
2003   **   STA   ALPHA
2006   **   LDCH   CHARZ
2009   **   STCH   C1
2012   ALPHA   RESW   1
2015   FIVE   WORD   5
2018   CHARZ   BYTE   C'EOF'
2019   C1   RESB   1
2020   **   END   **
```

B.2 Input and Output:


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Progra...

```
ALPHA    2012
FIVE    2015
CHARZ    2018
C1    2019

The contents of Output file :

        COPY    START   2000
2000    **      LDA     FIVE    332015
2003    **      STA     ALPHA   442012
2006    **      LDCH    CHARZ   532018
2009    **      STCH    C1      572019
2012    ALPHA   RESW    1
2015    FIVE    WORD    5       000005
2018    CHARZ   BYTE    C'EOF'  454f46
2019    C1      RESB    1
2020    **      END     **


The contents of Object code file :

H^COPY^002000^002020
T^002000^19^332015^442012^532018^572019^000005^454f46
E^002000_
```

B.3 Observations and learning:
(Students are expected to comment on the output obtained with clear observations and learning for each task/ subpart assigned)

We have learnt about the two passes of the two-pass assembler.

B.4 Conclusion:
(Students must write the conclusion as per the attainment of individual outcome listed above and learning/observation noted in section B.3)

We have successfully implemented a c program for the second pass of a two-pass assembler.

B.5 Question of Curiosity
(To be answered by a student based on the practical performed and learning/ observations)

1. Give Example of Working of Two-Pass Assembler

Ans:

➔ Two-pass assembler: Assemblers typically make two or more passes through a source program to resolve forward references in a program. A forward reference is defined as a type of instruction in the code segment that is referencing the label of instruction, but the assembler has not yet encountered the definition of that instruction.

Pass 1:

➔ Assembler reads the entire source program and constructs a symbol table of names and labels used in the program, that is, the name of data fields and programs labels and their relative location (offset) within the segment. Pass 1 determines the amount of code to be generated for each instruction.

Pass 2:

➔ The assembler uses the symbol table that is constructed in Pass 1. Now it knows the length and relative of each data field and instruction, it can complete the object code for each instruction. It produces.OBJ (Object file),. LST (list file) and cross-reference (.CRF) files.

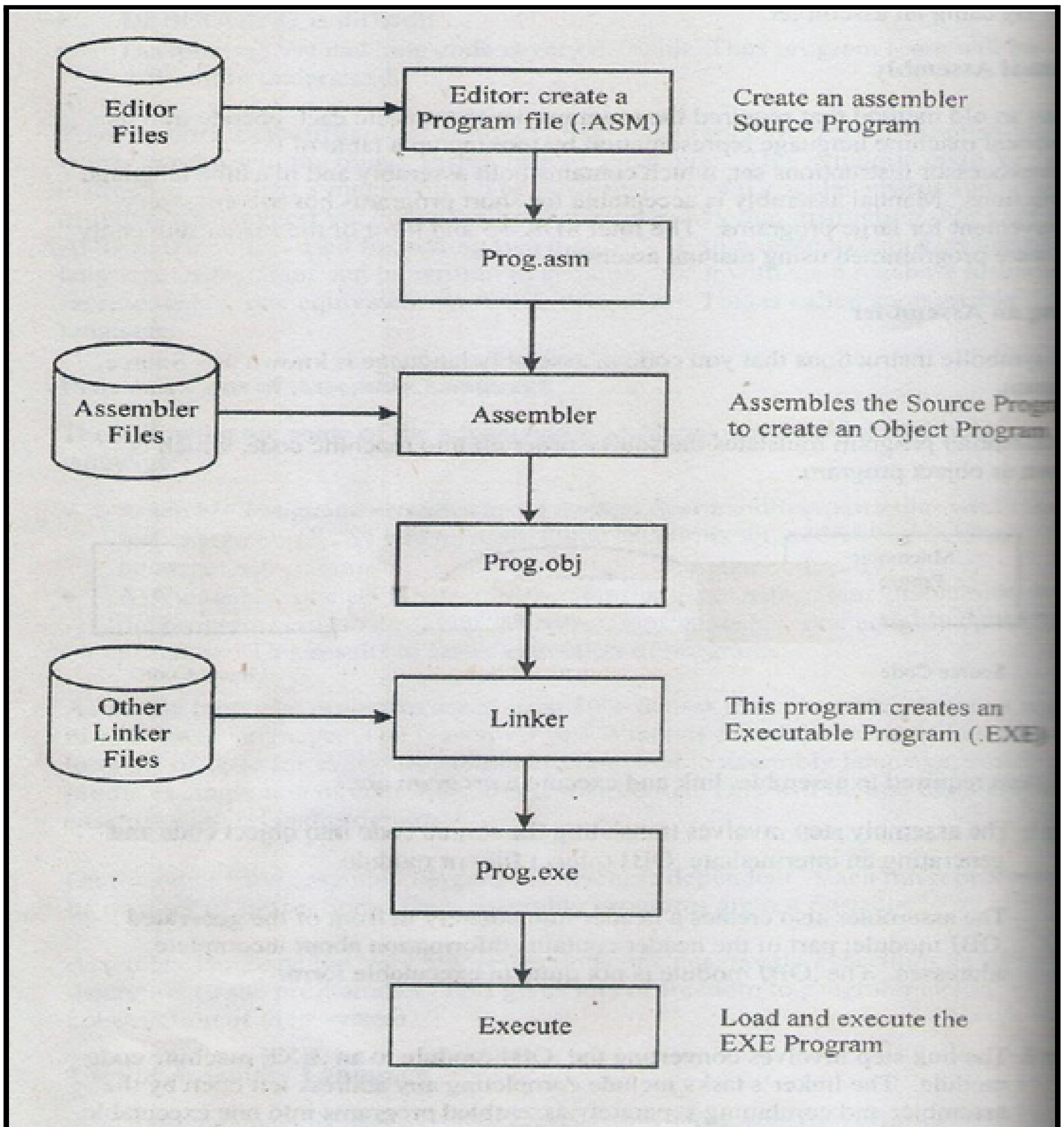

Editor Files
Editor: create a Program file (.ASM)
Create an assembler Source Program
Prog.asm
Assembler Files
Assembler
Assembles the Source Prog
to create an Object Program
Prog.obj
Other Linker Files
Linker
This program creates an Executable Program (.EXE)
Prog.exe
Execute
Load and execute the EXE Program

Figure 2: Program Assembly

2. Mention the Advantage of assemblers with Multiple Passes?

Ans:

➔ The main reason why most assemblers use a 2-pass system is to address the problem of forwarding references -- references to variables or subroutines that have not yet been encountered when parsing the source code. A strict 1-pass scanner cannot assemble source code that contains forward references. Pass 1 of the assembler scans the source, determining the size and address of all data and instructions; then pass 2 scans the source again, outputting the binary object code.

➔ Some assemblers have been written to use a 1.5 (one and a half) pass scheme, whereby the source is only scanned once, but any forward references are simply assumed to be of the largest size necessary to hold any native machine data type (for instance, assuming a reference to the address of an as-yet unencountered subroutine could be very far away, necessitating the use of a far branch). The unknown quantity is temporarily filled in as zero during pass 1 of the assembler, and the forward reference is added to a 'fix-up list'. After pass 1, the '.5' pass goes through the fix-up list and patches the output machine code with the values of all resolved forward references. This can result in sub-optimal opcode construction but allows for a very fast assembly phase.