PART A

(PART A : TO BE REFFERED BY STUDENTS)

Experiment No. 2 (B)

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

Objective: Develop a program to implement second pass:

- **a.** To generate Base table
- **b.** To generate machine code

Outcome: Students are able to design and implement pass 2 of two pass assembler.

Theory:

Pass 2: Purpose - To generate object program

1) Look up value of symbols (STGET)
2) Generate instruction (MOTGET2)
3) Generate data (for DC, DS)
4) Process pseudo ops (POT, GET2)

Data Structures:

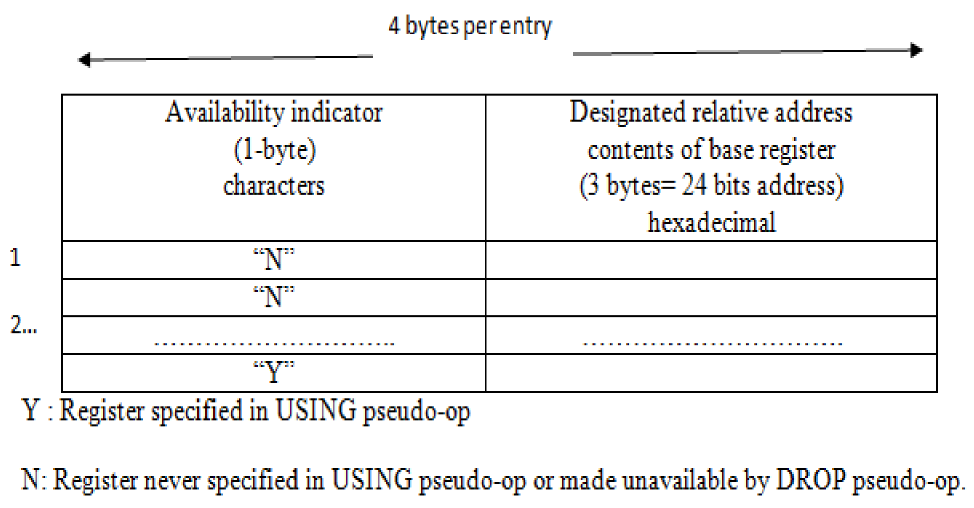
1) Copy of source program from Pass1
2) Location counter
3) MOT which gives the length, mnemonic format opcode
4) POT which gives mnemonic & action to be taken
5) Symbol table from Pass1
6) Base table which indicates the register to be used or base register
7) A work space INST to hold the instruction & its parts
8) A work space PRINT LINE, to produce printed listing
9) A work space PUNCH CARD for converting instruction into format needed by loader
10) An output deck of assembled instructions needed by loader.

Format of database:

Base Table:

Assembler uses this table to generate proper base register reference in machine instructions and to compute offset. Then the offset is calculated as:

offset= value of symbol from ST - contents of base register



4 bytes per entry

	Availability indicator (1-byte) characters	Designated relative address contents of base register (3 bytes= 24 bits address) hexadecimal
1	"N"	
	"N"	
2...	…………………………..	……………………………
	"Y"	

Y : Register specified in USING pseudo-op

N: Register never specified in USING pseudo-op or made unavailable by DROP pseudo-op.

Let us consider the same example as experiment no. 1 and the base table after statement 2:

Base register	Contents
15	0

After statement 7:

Base register	Contents
15	10

Code after pass2:

stmt no	Relative address	Statement		
3	0		A	1, 12 (0,15)
4	4		A	2, 16(0,15)
6	8		A	3, 10(0,15)
8	12		4	
9	16		5	
10		-		

Algorithm:

1. Initialize the location counter as: LC=0
2. Read the statement from source program
3. Examine the op-code field: If match found in MOT then
   a. From the MOT entry determine the length field i.e. L=length, binary op-code and format of the instruction.

   Different instruction format requires different processing as described below:

   1. **RR Instruction : (Register to Register)**
      Both of the register specification fields are evaluated and placed into second byte of RR instruction
   2. **RX Instruction : (Register to Index)**
      Both of the register and index fields are evaluated and processed similar to RR instruction. The storage address operand is evaluated to generate effective address

(EA). The BT is examined to find the base register. Then the displacement is determined as:

D=EA- Contents of base register.

The other instruction formats are processed in similar manner to RR and RX.

b. Finally the base register and displacement specification are assembled in third and fourth bytes of instruction.
c. The current value of location counter is incremented by length of instruction.

4. If match found in POT then
   a. If it is EQU pseudo-op then EQU card is printed in the listings.
   b. If it is USING pseudo-op then the corresponding BT entry is marked as available.
   c. If it is DROP pseudo-op then the corresponding BT entry is marked as unavailable.
   d. If it is DS or DC pseudo-op then various conversions are done depending on the data type and symbols are evaluated. Location counter is updated by length of data.
   e. END pseudo-op indicates end of source program and then pass2 is terminated. Before that if any literals are remaining then the code is generated for them.
5. After assembling the instruction it is put in the format required by loader.
6. Finally a listing is printed which consist of copy of source card, its storage location and hexadecimal representation.
7. Go to step 2.

PART B

(PART B : TO BE COMPLETED BY STUDENTS)

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

Roll. No. B48	Name: Aryan Unhale
Class: TE COMPS B	Batch: B3
Date of Experiment:28/1/25	Date of Submission: 6/3/25
Grade:	

B.1 Software Code written by student:

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

struct Symbol {
    char label[20];
    int address;
} symtab[50];
```

```
struct Opcode {
    char mnemonic[20];
    int opcode;
} optab[50];
```

```
int symCount = 0, opCount = 0;
```

```
// Function to load symbol table
void loadSymtab() {
    FILE *f = fopen("symtab2.txt", "r");
    if (f == NULL) {
        printf("Error: Symbol table not found!\n");
        exit(1);
    }
    while (fscanf(f, "%d %s", &symtab[symCount].address, symtab[symCount].label) != EOF) {
        symCount++;
    }
    fclose(f);
}
```

```
// Function to load opcode table
void loadOptab() {
    FILE *f = fopen("optab2.txt", "r");
    if (f == NULL) {
        printf("Error: Opcode table not found!\n");
        exit(1);
    }
    while (fscanf(f, "%s %d", optab[opCount].mnemonic, &optab[opCount].opcode) != EOF) {
        opCount++;
    }
    fclose(f);
}
```

```
// Function to find symbol address
```

```c
int getSymbolAddress(char *label) {
    for (int i = 0; i < symCount; i++) {
        if (strcmp(symtab[i].label, label) == 0) {
            return symtab[i].address;
        }
    }
    return -1;  // Not found
}
```

```c
// Function to find opcode value
int getOpcode(char *mnemonic) {
    for (int i = 0; i < opCount; i++) {
        if (strcmp(optab[i].mnemonic, mnemonic) == 0) {
            return optab[i].opcode;
        }
    }
    return -1;  // Not found
}
```

```c
int main() {
    FILE *f1, *f2;
    char label[20], mnemonic[20], operand[20];
    int opcode, address, symbolAddress;
```

```c
    // Load required tables
    loadSymtab();
    loadOptab();
```

```c
    f1 = fopen("input2.txt", "r");
    f2 = fopen("machine_code.txt", "w");
```

```c
    if (f1 == NULL || f2 == NULL) {
        printf("Error opening input or output files!\n");
        return 1;
    }
```

```c
    printf("Address\tMnemonic\tOperand\tMachine Code\n");
    fprintf(f2, "Address\tMnemonic\tOperand\tMachine Code\n");
```

```c
    while (fscanf(f1, "%s %s %s", label, mnemonic, operand) != EOF) {
        opcode = getOpcode(mnemonic);
        symbolAddress = getSymbolAddress(operand);
```

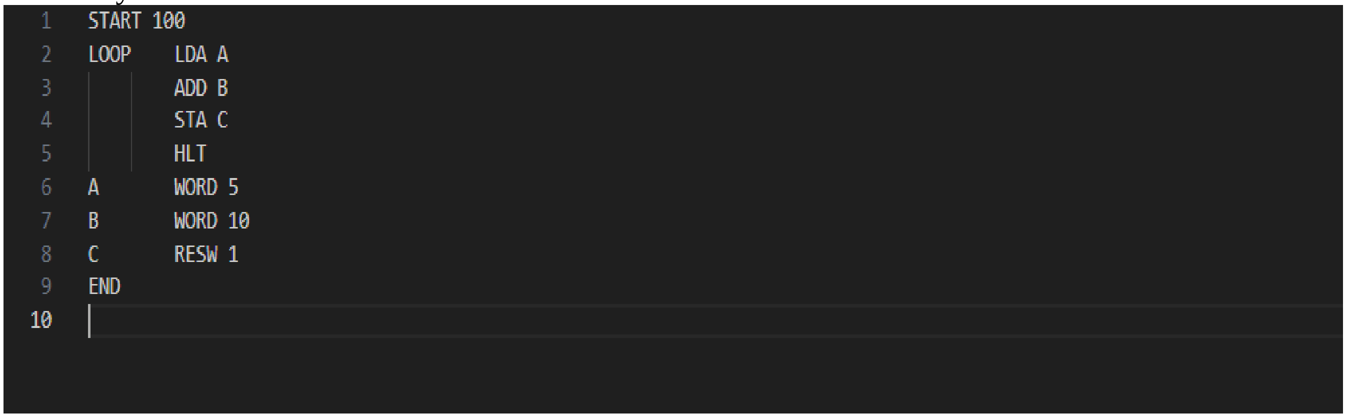
```c
        if (opcode != -1) {
            printf("%04X\t%s\t%s\t%02X%04X\n", address, mnemonic, operand, opcode, symbolAddress);
            fprintf(f2, "%04X\t%s\t%s\t%02X%04X\n", address, mnemonic, operand, opcode,
symbolAddress);
        } else {
            printf("%04X\t%s\t%s\tDATA\n", address, mnemonic, operand);
            fprintf(f2, "%04X\t%s\t%s\tDATA\n", address, mnemonic, operand);
        }
    }
```

```c
    fclose(f1);
    fclose(f2);
```

```c
    printf("Machine code successfully generated in machine_code.txt\n");
    return 0;
}
```

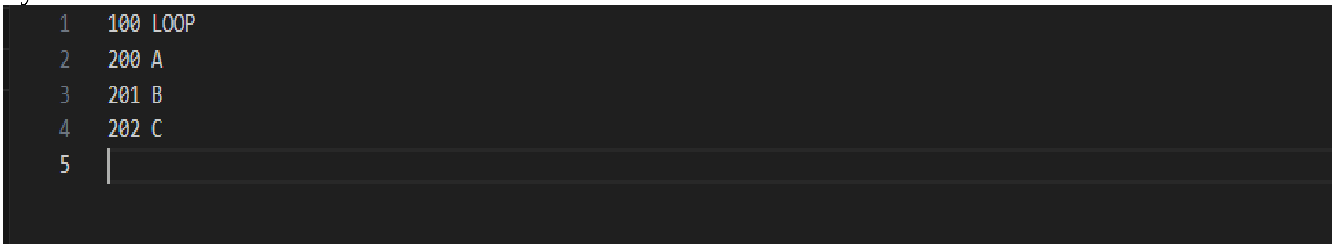
B.2 Input and Output:

Assembly code:



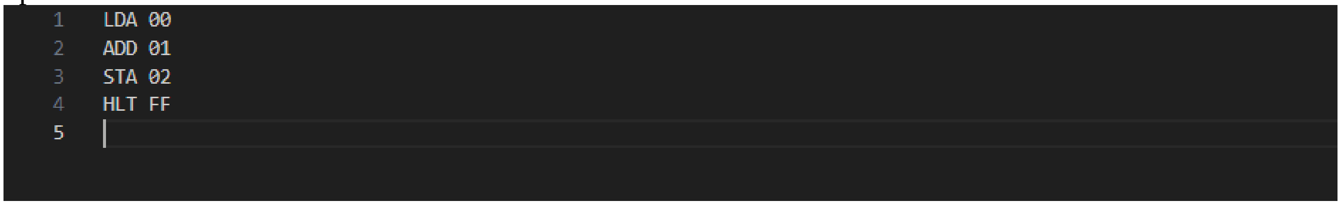
```
START 100
LOOP    LDA A
        ADD B
        STA C
        HLT
A       WORD 5
B       WORD 10
C       RESW 1
END
```

Symtab2.txt



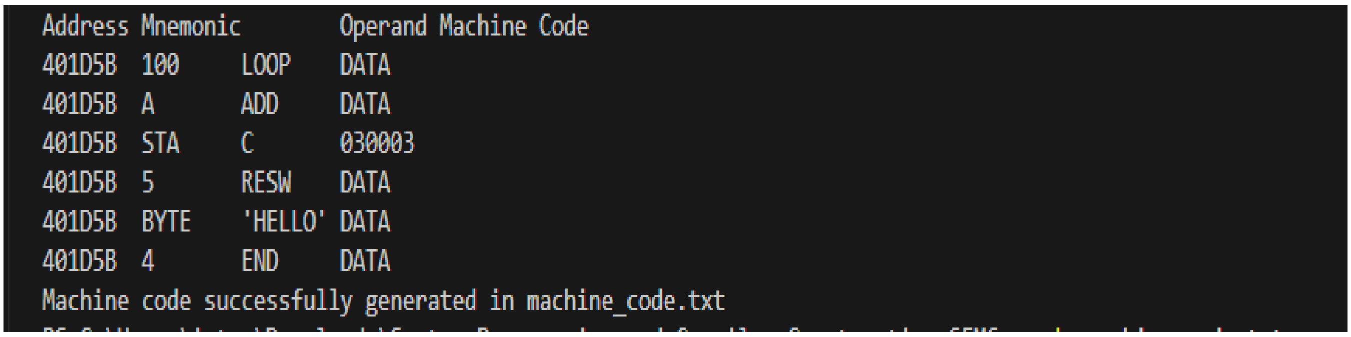
```
100 LOOP
200 A
201 B
202 C
```

optab2.txt



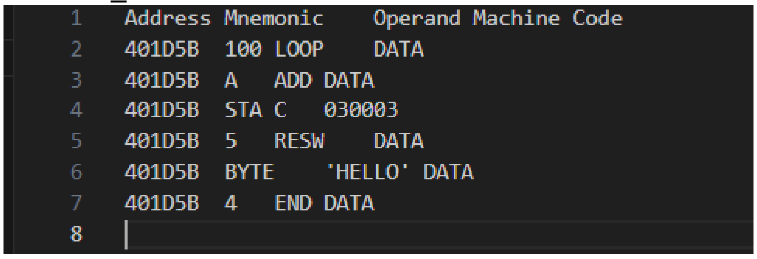
```
LDA 00
ADD 01
STA 02
HLT FF
```

OUTPUT:



```
Address Mnemonic        Operand Machine Code
401D5B  100     LOOP    DATA
401D5B  A       ADD     DATA
401D5B  STA     C       030003
401D5B  5       RESW    DATA
401D5B  BYTE    'HELLO' DATA
401D5B  4       END     DATA
Machine code successfully generated in machine_code.txt
```

Machine_code.txt



```
Address Mnemonic    Operand Machine Code
401D5B  100 LOOP    DATA
401D5B  A   ADD DATA
401D5B  STA C   030003
401D5B  5   RESW    DATA
401D5B  BYTE    'HELLO' DATA
401D5B  4   END DATA
```

B.3 Observations and learning:

In the second pass of a two-pass assembler, the main objective is to generate the object program by translating the intermediate code generated in the first pass into machine-readable code. The second pass utilizes various data structures such as the symbol table, machine operation table (MOT), and base table to perform the necessary transformations. It starts by examining the opcode field of each instruction, matching it with entries in the MOT to determine the instruction format and length. Different formats such as Register-to-Register (RR) or Register-to-Index (RX) require specific handling, including calculating the displacement for base register references, and converting symbolic addresses into effective addresses. The location counter is incremented after each instruction to ensure that the next instruction is placed in the correct memory location.

Additionally, during the second pass, pseudo-operations (POT) are processed. These include EQU (equating symbols), USING (indicating which register is used as the base register), and DROP (unmarking a base table entry). Data-conversion pseudo-operations like DS (define storage) and DC (define constants) are handled by evaluating symbols and updating the location counter accordingly. The final step involves formatting the assembled instructions into a form suitable for the loader, while also producing a listing of the source program, its storage location, and its hexadecimal machine code representation. This process ensures that the assembler correctly generates an object program that can be loaded and executed by the IBM 360/370 processor.

B.4 Conclusion:

In conclusion, the second pass of the assembler plays a crucial role in converting the intermediate assembly code into machine-readable instructions. By utilizing various data structures like the symbol table, base table, and machine operation table, it ensures that symbolic addresses are resolved, pseudo-operations are processed, and instructions are correctly formatted for the loader.

B.5 Question of Curiosity

A. Give Example of Working of Two Pass Assembler

1. **Pass 1:**
   - Create the symbol table by scanning the assembly code to assign addresses to all labels.
2. **Pass 2:**
   - Generate machine code using the symbol table to replace symbolic addresses with actual addresses.

Example-

Assembly Code:

```
LOOP LDA ='5'

     STA END

END  ADD LOOP
```

Symbol Table:

```
LOOP 1000

END  1009
```

MACHINE CODE

```
LOOP 00 5005

     0C 1009

END  18 1000
```

B. Mention the Advantage of assemblers with Multiple Passes?

Advantages of Assemblers with Multiple Passes:

Resolution of Forward References:

Pass 1: Collects and defines labels and addresses.

Pass 2: Uses this information to resolve any forward references, ensuring accurate address assignment.

Efficient Error Detection:

Multiple passes allow for more comprehensive error checking and reporting, leading to higher code reliability.

Optimized Code Generation:

The assembler can make more informed decisions about instruction placement and optimization strategies, resulting in better-optimized machine code.

These advantages contribute to creating more reliable and efficient assembly language programs.