

PART A

(PART A : TO BE REFERRED BY STUDENTS)

Experiment No. 1 (A)

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

Objective: Develop a program to implement first pass:

- a. To search instruction in MOT and return its length
- b. To search instruction in POT and return the routine called
- c. To generate Symbol table
- d. To generate literal table
- e. To generate intermediate code after pass 1

Outcome: Students are able to design and implement pass 1 of two pass assembler.

Theory:

An assembler performs the following functions

1. Generate instructions
 - a. Evaluate the mnemonic in the operator field to produce its machine code.
 - b. Evaluate subfields- find value of each symbol, process literals & assign address.
2. Process pseudo ops.

Pass 1: Purpose - To define symbols & literals

1. Determine length of machine instruction (MOTGET)
2. Keep track of location counter (LC)
3. Remember values of symbols until pass2 (STSTO)
4. Process some pseudo-ops. EQU
5. Remember literals (LITSTO)

Pass 1: Database

1. Source program
2. Location counter (LC) which stores location of each instruction
3. Machine Operation Table (MOT). This table indicates the symbolic mnemonic for each instruction and its length.
4. Pseudo Operation Table (POT). This table indicates the symbolic mnemonic and action taken for each pseudo-op in pass1.
5. Symbol Table (ST) which stores each label along with its value.
6. Literal Table (LT) which stores each literal and its corresponding address
7. A copy of input which will be used by pass2.

Format of databases

The Machine Operation Table (MOT) and Pseudo Operation Table (POT) are examples of fixed tables. During the assembly process the contents of this table are not filled in or altered

1. Machine-op Table (MOT)

← 6-bytes per entry →				
Mnemonic -codes (4 bytes) characters	Binary op-codes (1 byte) hexadecimal	Instruction length (2-bits) binary	Instruction format (3-bits) binary	Not used in this design (3 bits)
“Abbb”	5A	10	001	
“AHbb”	4A	10	001	
“ALbb”	5E	10	001	
“ALRb”	1E	01	000	
.....	

b : bank space

2. Pseudo-op Table (POT)

← 8-BYTES PER ENTRY →	
Pseudo-op (5-bytes) Characters	Address of routine to process pseudo-op (3 bytes= 24 bits address)
“DROPb”	P1DROP
“ENDbb”	P1END
“EQUbb”	P1EQU
“START”	P1START
“USING”	P1USING

Let us consider following source code and find the contents of symbol table and literal table.

Stmt no	Symbol	Op-code	Operands
1	SAMPLE	START	0
2		USING	*,15
3		A	1,FOUR
4		A	2,FIVE
5	TEMP	EQU	10
6		A	3,=F'3'
7		USING	TEMP,15
8	FOUR	DC	F'4'
9	FIVE	DC	F'5'
10		END	

3. Symbol Table:

Symbol	Value	Length	Relocation
“SAMPLEbbb”	0	1	“R”
“TEMPbbbb”	10	4	“A”
“FOURbbbb”	12	4	“R”
“FIVEbbbb”	16	4	“R”

4. Literal Table

Literal	Value	Length	Relocation
F'3'	20	4	"R"

5. Code after pass1:

Stmt no	Relative address	Statement		
1		SAMPLE	START	0
2			USING	*,15
3	0		A	1, _ (0,15)
4	4		A	2, _ (0,15)
5		-		
6	8		A	3, _ (0,15)
7		-		
8	12	FOUR	4	
9	16	FIVE	5	
10		-		

Algorithm:

- Initially location counter is set to relative address 0 i.e. LC=0
- Read the statement from source program
- Examine the op-code field: If match found in MOT then
 - From the MOT entry determine the length field i.e. L=length.
 - Examine the operand field to check whether literal is present or not. If any new literal is found then corresponding entry is done in LT.
 - Examine the label field for the presence of symbol. If label is present then it is entered in ST and current value of location counter is assigned to symbol.
 - The current value of location counter is incremented by length of instruction(L)
- If match found in POT then
 - If it is USING or DROP pseudo-op then first pass do nothing. It just writes a copy of these cards for pass 2.
 - If it is EQU pseudo-op then evaluate expression in operand field and assign value to the symbol present in label field.
 - If it is DS or DC pseudo-op then by examining the operand field find out number of bytes of storage required. Adjust the location counter for proper alignment.
 - If it is END pseudo-op then pass1 is terminated and control is passed to pass2. Before transferring the control, it assigns location to literals.
- A copy of source card is saved for pass 2.
- 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. B30	Name: Bhatt Pranjal
Class: TE COMPS B	Batch: B2
Date of Experiment:	Date of Submission:
Grade:	

B.1 Software Code written by student:

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

int main() {
    FILE *f1, *f2, *f3;
    int lc, sa, op1, o, len;
    char m1[20], la[20], op[20], otp[20];
```

```
// Open necessary files
f1 = fopen("input.txt", "r");
f3 = fopen("symtab.txt", "w");
```

```
if (f1 == NULL || f3 == NULL) {
    printf("Error opening file!\n");
    return 1;
}
```

```
// Read first line
fscanf(f1, "%s %s %d", la, m1, &op1);
if (strcmp(m1, "START") == 0) {
    sa = op1;
    lc = sa;
    printf("\t%s\t%s\t%d\n", la, m1, op1);
} else {
    lc = 0;
}
```

```
// Process each instruction
while (fscanf(f1, "%s %s %s", la, m1, op) != EOF) {
    printf("\n%d\t%s\t%s\t%s\n", lc, la, m1, op);
```

```
// If label is present, add to symbol table
if (strcmp(la, "-") != 0) {
    fprintf(f3, "%d\t%s\n", lc, la);
}
```

```
// Search opcode in optab.txt
f2 = fopen("optab.txt", "r");
if (f2 == NULL) {
    printf("Error opening optab.txt!\n");
    fclose(f1);
    fclose(f3);
    return 1;
}
```

```
int found = 0;
while (fscanf(f2, "%s %d", otp, &o) != EOF) {
    if (strcmp(m1, otp) == 0) {
        lc += 3;
```

```
        found = 1;
        break;
    }
}
fclose(f2);
```

```
// Handle assembler directives
if (!found) {
    if (strcmp(m1, "WORD") == 0) {
        lc += 3;
    } else if (strcmp(m1, "RESW") == 0) {
        op1 = atoi(op);
        lc += (3 * op1);
    } else if (strcmp(m1, "BYTE") == 0) {
        if (op[0] == 'X')
            lc += 1;
        else {
            len = strlen(op) - 2;
            lc += len;
        }
    } else if (strcmp(m1, "RESB") == 0) {
        op1 = atoi(op);
        lc += op1;
    }
}
}
```

```
printf("\nProgram length = %d\n", lc - sa);
```

```
// Close files
fclose(f1);
fclose(f3);
```

```
    return 0;
}
```

B.2 Input and Output:

```
PROBLEMS OUTPUT TERMINAL PORTS powershell + v [ ] ... ^ X

PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> gcc exp1.c -o exp1
PS C:\Users\intaz\Downloads\System Programming and Compiler Construction SEM6> ./exp1

0    LOOP    LDA    A

3    ADD     B     STA

3    C       WORD  5

6    RESW    2     BYTE

6    'HELLO' RESB   4

10   END     RESB   4

Program Length = 14
```

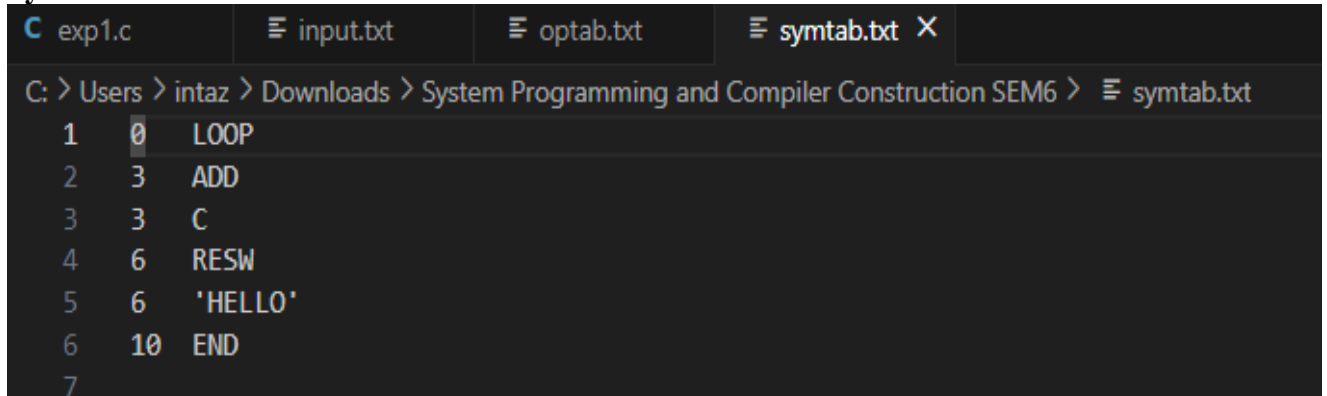
Assembly code: Input.txt

```
exp1.c input.txt X optab.txt
C: > Users > intaz > Downloads > System Programming and Compiler Construction SEM6 > input.txt
1    START 100
2    LOOP   LDA A
3          ADD B
4          STA C
5          WORD 5
6          RESW 2
7          BYTE 'HELLO'
8          RESB 4
9    END
10
```

Optab.txt

```
exp1.c input.txt optab.txt X
C: > Users > intaz > Downloads > System Programming and Compiler Construction SEM6 > optab.txt
1    LDA 3
2    ADD 3
3    STA 3
4
```

Symtab.txt



```
C: > Users > intaz > Downloads > System Programming and Compiler Construction SEM6 > symtab.txt
1 0 LOOP
2 3 ADD
3 3 C
4 6 RESW
5 6 'HELLO'
6 10 END
7
```

B.3 Observations and learning:

In this experiment, we implemented a two-pass assembler that generates the base table and machine code. During the first pass, we successfully constructed the Mnemonic Operation Table (MOT) and the Symbol Table. The MOT provided a reliable reference for translating assembly instructions into machine code, while the Symbol Table captured all the labels and their corresponding addresses. We also created the Segment Register Table to handle segment addresses accurately. The Forward Reference Table effectively tracked references to symbols that had not yet been defined, allowing us to resolve them during the second pass. The Cross Reference Table was useful for listing all symbols and their locations in the code, aiding in debugging and optimization.

B.4 Conclusion:

In conclusion, the experiment demonstrated the importance of structured data tables and a systematic approach in assembly language programming. By using a two-pass assembler and managing forward references effectively, we were able to generate accurate machine code and resolve references to undefined symbols, ensuring the overall success of the assembly process.

B.5 Question of Curiosity

A. Define Data Structures

1. Mnemonic Operation Table

Mnemonic Operation Table (MOT)

Description: Contains the mnemonics of the assembly language instructions along with their corresponding opcodes.

Example: `{ "LDA": "00", "STA": "01" }`

2. Symbol Table

Symbol Table

Description: Stores the labels and their corresponding addresses defined in the program.

Example: `{ "ALPHA": 1000, "BETA": 1004 }`

3. Segment Register Table

Segment Register Table

Description: Keeps track of segment registers and their corresponding segment addresses.

Example: `{ "CS": 0, "DS": 1000 }`

4. Forward Reference Table

Forward Reference Table

Description: Temporarily stores the references to symbols that have not yet been defined.

Example: `[{ symbol: "ALPHA", address: 1002 }, { symbol: "BETA", address: 1006 }]`

5. Cross Reference Table

Cross Reference Table

Description: Lists all symbols and the locations in the code where they are referenced.

Example: `{ "ALPHA": [1002, 1006], "BETA": [1010, 1014] }`

B. Comment on the Forward Reference Problem and Remedy.

Forward Reference Problem and Remedy

Forward Reference Problem: This occurs when an assembler encounters a symbol (e.g., a label) before it is defined. The assembler does not know the address of the symbol when it is first referenced.

Remedy:

1. **Two-Pass Assembler:** The assembler performs two passes over the source code. In the first pass, it constructs the symbol table with all defined symbols and their addresses. In the second pass, it generates the machine code using the symbol table.
2. **Forward Reference Table:** During the first pass, if a symbol is used before it is defined, the assembler adds it to the forward reference table. During the second pass, the assembler resolves these references once the symbol addresses are known.
3. **Backpatching:** After constructing the symbol table and generating machine code in the second pass, the assembler goes back and fills in the addresses for the forward references in the generated machine code.

These methods effectively manage the forward reference problem in assembly language programming.