An assembler is a program that accepts as input an assembly language program and produces its machine language equivalent along with information for the loader. The main purpose of this program is production of machine language.

Here we are developing the assembler for the microprocessor 8085. As we are not using loader the developed assembler will produce only the machine language program.

**GENERAL DESIGN PROCEDURE:**

Listed below are the 6 steps that have been followed:

1. Specify the problem
2. Specify the data structures
3. Define format of the data structures
4. Specify the algorithm
5. Look for modularity (i.e., capability of one program to be subdivided into independent programming units).

We consider the following program.

PROG: START 20F0H // STARTING OF THE PROGRAM

BEGIN: MVI A, =64H // COUNT

DCR A

JNZ BEGIN

HALT: HLT

END

**STAMENT OF THE PROBLEM:**

The assembler reads the first START instruction and note that it is a pseudo-op instruction (to the assembler) giving PROG as the name of the program and tells us that the program will start executing from the hex location 20F0. The next instruction, MVI is a machine-op. This tells the assembler to store the immediate data 64 into the accumulator. As this a machine-op, the assembler has to look for the bit configuration (which is called opcode) from the machine-op table provided by the 8085 Designer and put the opcode (of MVI A, DATA8 instruction) in the appropriate place of the machine language instruction. There is a Label (often called as symbol) present for this instruction to mark the instructions location, which might be used in the further program. For the further use and in order to mark the location of the label, we store the label and its location value in the Symbol Table. Next comes DCR instruction. As DCR is a machine-op, the assembler fetches opcode (of DCR A instruction) from the machine-op table and stores it in the appropriate place in the machine language instruction. Next comes up the JNZ instruction. The machine opcode cycle is again in effect and after storing the machine opcode, there is the usage of Label. A label maybe defined before the calling statement or after the calling statement. The name and location at which it is defined, is stored in the symbol table. The next instruction is HLT (a halt instruction), it is a machine instruction, so the opcode fetch cycle will run. And the last instruction is END, a pseudo-op and this will mark the end of the program. The END pseudo-op does not indicate the physical end of the program. It is a logical end of that particular program or sub routine. Other sub routine definitions may follow the end pseudo-op.

We have thus produced a Symbol Table and an intermediate output file as given below.

**SYMBOL TABLE:**

PROG 20F0

BEGIN 20F0

HALT 20F7

**INTERMEDIATE OUTPUT FILE:**

Typical line from P1Output.txt is:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Source Program Line No** | **Location Counter Value** | **Label** | **Mnemonic** | **Operand 1** | **Operand 2** | **Instruction Type** | **Comment** |
| 0002 | 20F0 | BEGIN | MVI | A | =64H | 8 | Initialize A |

Because symbols can appear before they are defined, it is convenient to make two passes over the input (as this example shows). The first pass is only to define the symbols and the second pass then uses the output generated by the first pass as an input and generates the final assembled program.

1. Generate instructions:
2. Evaluate the mnemonic in the operational field to produce the machine opcode.
3. Evaluate the subfields – find the values of each symbol, process literals.
4. Process Pseudo-ops.

We can group these tasks into two passes or sequential scans over the input (first pass is over input source file containing assembly language program and the second pass is over the intermediate output file generated by pass1), associated with each task there are more than one assembler modules.

**PASS 1:**

1. Determine pseudo-op or machine-op
2. Process pseudo-op and machine-op (get their value form their respective tables).
3. Keep track of LC.
4. Store the symbols in the Symbol Table.
5. Write pass 1 output to file.
6. Check Symbol Table for duplicates.

**PASS 2:**

1. Get the pass 1 output from the file.
2. Get the Symbol Table from the file.
3. Process machine-ops according to their type (total types = 9).
4. Process literals.
5. Check for errors (if any).
6. Write the assembled instructions to Pass2Output file.

**DATABASES:**

The second step in the design procedure is to establish the data bases that we have to work with.

Pass 1 Databases:

1. Input source program.
2. A works-space, SourceInst, which will be used to hold each instruction with its fields separated (e.g., Label name, mnemonic, operand 1, operand 2).
3. A location counter (LC), used to keep track of each instruction’s location.
4. A Machine-op table, that indicates the symbolic mnemonic and other fields for each instruction.
5. A pseudo-op table, that indicates the symbolic mnemonic for each pseudo-op.
6. A symbol table (ST), that is used to store each label and its corresponding value.
7. A copy of the output of pass 1 to be later used as an input by pass 2.

Pass 2 Databases:

1. A copy of output of pass 1 which is the input for pass 2.
2. Location counter (LC).
3. A Machine-op table, that indicates the symbolic mnemonic, Operand1, Operand2, Op-code, length of the instruction and Type of the instruction for each instruction.
4. A pseudo-op table, that indicates the symbolic mnemonic.
5. The symbol table prepared by pass 1, containing each label and its corresponding value.
6. A work-space, P1OutputInst, which will be used to hold each instruction with its field separated (e.g., Line number, location counter, label, mnemonic, operand 1, operand 2, opcode, type of the instruction, and comment (if present)).
7. A copy of pass 2 output for storing the assembled program.

**Format of Databases:**

The third step in the design procedure is to specify the format and content of each of the databases – a task that must be undertaken even before describing the specific algorithm underlying the assembler design. In reality, the algorithm, the databases, and format are all interlocked. Their specifications are in practical designs, circular, and are designed with some features of the format and algorithm that are used and continue to iterate their design until all the cases work successfully.

Both the passes require a machine-op table (MOT) which contains Mnemonic, operand 1, operand 2, opcode, Length of the instruction, Instruction type. They also require pseudo-op table (POT) which contains the pseudo-ops. Both these tables are *fixed tables.* The contents of these tables are not filled in or altered during the assembly process.

**Machine Operation Table (MOT)**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Mnemonics** | **Operand1** | **Operand2** | **Opcode** | **Length of Instruction** | **Type of Instruction** |
| String | String | String | String | Integer | Integer |
| 09  Characters | 11  Characters | 11  Characters | 02  Characters | Machine Dependent  Size(int) | Machine Dependent  Size(int) |
| 10 Bytes | 12 Bytes | 12 Bytes | 03 Bytes | 02 Bytes | 02 Bytes |
| ACI | Data8 | !! | CE | 2 | 3 |
| : | : | : | : | : | : |
| : | : | : | : | : | : |
| XTHL | ?? | !! | E3 | 1 | 1 |

Q. How did we add leading zeros to any operand?

Ans. First convert the operand string to integer by using “strtol” function. And then convert the number to Hex by giving using “sprintf” function and thus we get a number with leading zeros.

**/\*/\*-/\*\*\*/\*/\*/\*/\*/\*/ INSERT THE FLOW CHART OF ASSEMBLER FROM NOTEBOOK.**