**The Design of 8085 Assembler**

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.

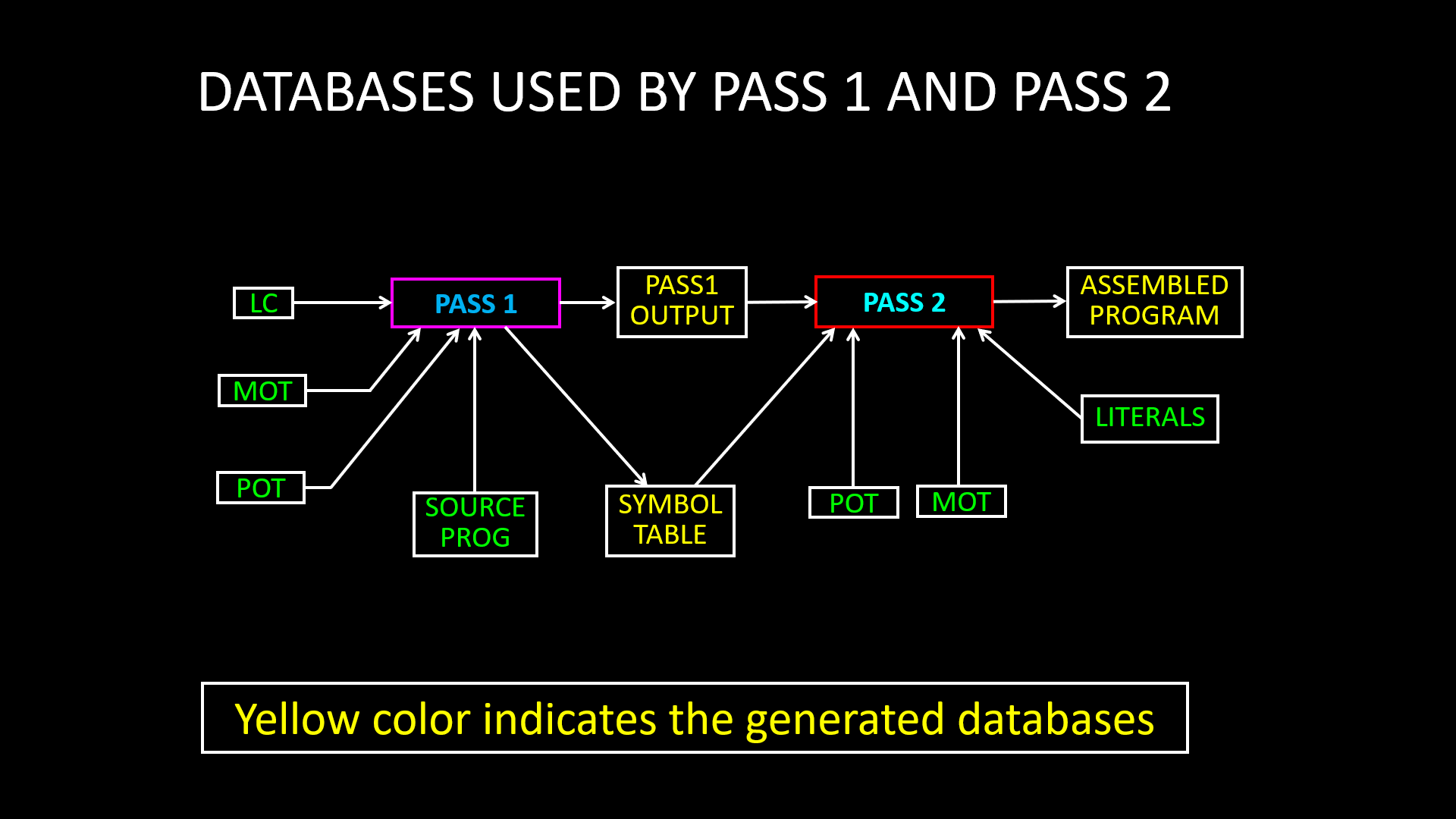
Pass 2 Databases:

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.

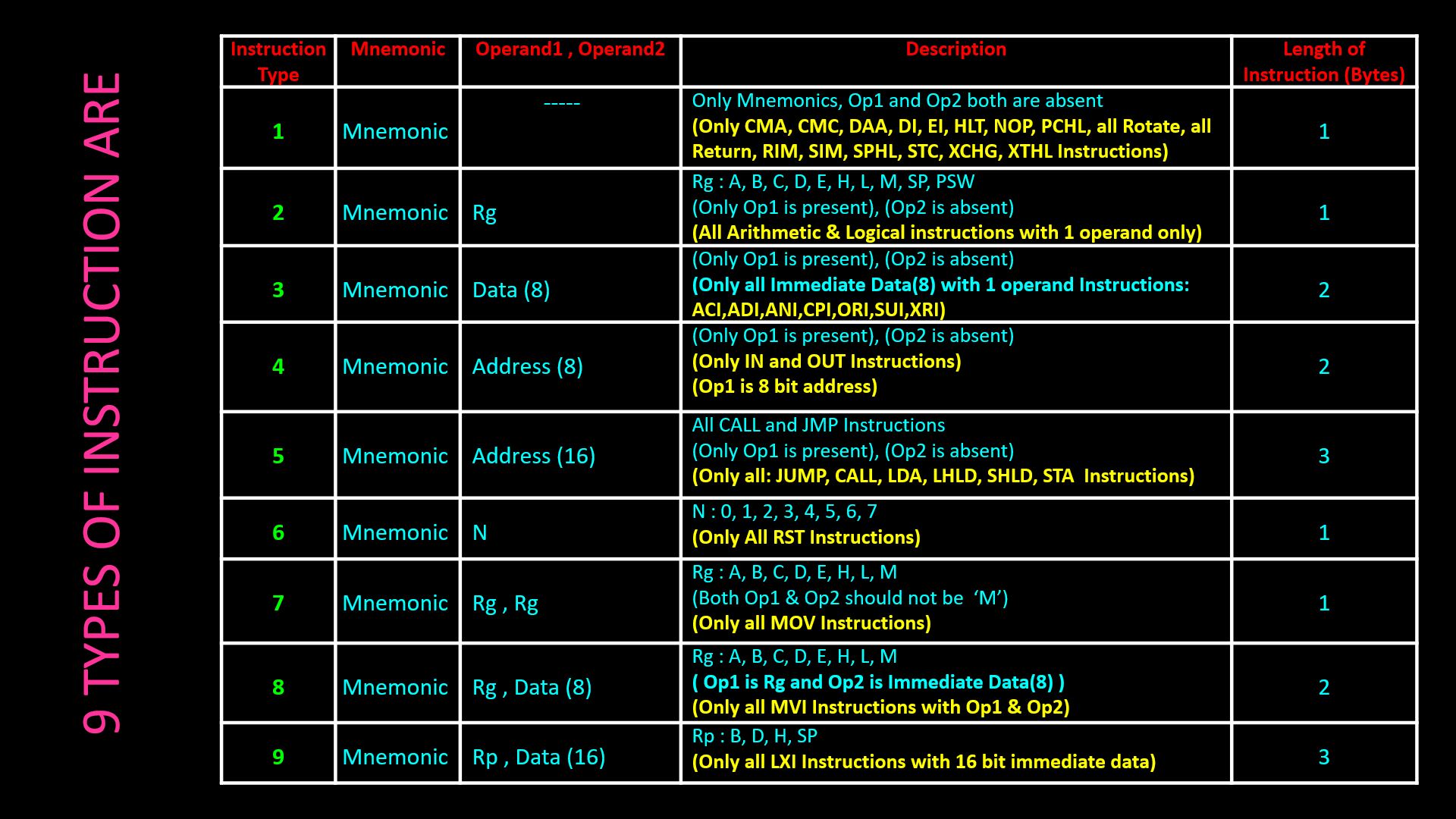
Both the passes require a machine-op table (MOT) which contains Mnemonic, operand 1, operand 2, opcode, Length of the instruction, Instruction type. They also require pseudo-op table (POT) which contains the pseudo-ops. Both these tables are *fixed tables.* The contents of these tables are not filled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



**Implementation Issues**

We have implemented the 8085-processor using c programming. In this section we discuss the detailed implementation procedure. The data structures that we used are arrays, structures, arrays of structures, and files. The exact format of used data structures is discussed in this section.

**Machine instructions are classified in 9 types.**



**Instruction Set for 8085**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sr. No.** | **Mnemonics** | **Operand 1** | **Operand 2** | **Opcode** | **Instruction**  **Length** | **Instruction**  **Type** |
| 1. | ACI | Data (8) |  | CE | 2 | 3 |
| 2. | ADC | A |  | 8F | 1 | 2 |
| 3. | ADC | B |  | 88 | 1 | 2 |
| 4. | ADC | C |  | 89 | 1 | 2 |
| 5. | ADC | D |  | 8A | 1 | 2 |
| 6. | ADC | E |  | 8B | **1** | 2 |
| 7. | ADC | H |  | 8C | 1 | 2 |
| 8. | ADC | L |  | 8D | 1 | 2 |
| 9. | ADC | M |  | 8E | 1 | 2 |
| 10. | ADD | A |  | 87 | 1 | 2 |
| 11. | ADD | B |  | 80 | 1 | 2 |
| 12. | ADD | C |  | 81 | 1 | 2 |
| 13. | ADD | D |  | 82 | 1 | 2 |
| 14. | ADD | E |  | 83 | 1 | 2 |
| 15. | ADD | H |  | 84 | 1 | 2 |
| 16. | ADD | L |  | 85 | 1 | 2 |
| 17. | ADD | M |  | 86 | 1 | 2 |
| 18. | ADI | Data (8) |  | C6 | 2 | 3 |
| 19. | ANA | A |  | A7 | 1 | 2 |
| 20. | ANA | B |  | A0 | 1 | 2 |
| 21. | ANA | C |  | A1 | 1 | 2 |
| 22. | ANA | D |  | A2 | 1 | 2 |
| 23. | ANA | E |  | A3 | 1 | 2 |
| 24. | ANA | H |  | A4 | 1 | 2 |
| 25. | ANA | L |  | A5 | 1 | 2 |
| 26. | ANA | M |  | A6 | 1 | 2 |
| 27. | ANI | Data (8) |  | E6 | 2 | 3 |
| 28. | CALL | Address (16) |  | CD | 3 | 5 |
| 29. | CC | Address (16) |  | DC | 3 | 5 |
| 30. | CM | Address (16) |  | FC | 3 | 5 |
| 31. | CMA |  |  | 2F | 1 | 1 |
| 32. | CMC |  |  | 3F | 1 | 1 |
| 33. | [CMP](https://electricalvoice.com/8-bit-logical-compare-instructions-8085-microprocessor/) | A |  | BF | 1 | 2 |
| 34. | CMP | B |  | B8 | 1 | 2 |
| 35. | CMP | C |  | B9 | 1 | 2 |
| 36. | CMP | D |  | BA | 1 | 2 |
| 37. | CMP | E |  | BB | 1 | 2 |
| 38. | CMP | H |  | BC | 1 | 2 |
| 39. | CMP | L |  | BD | 1 | 2 |
| 40. | CMP | M |  | BE | 1 | 2 |
| 41. | CNC | Address (16) |  | D4 | 3 | 5 |
| 42. | CNZ | Address (16) |  | C4 | 3 | 5 |
| 43. | CP | Address (16) |  | F4 | 3 | 5 |
| 44. | CPE | Address (16) |  | EC | 3 | 5 |
| 45. | [CPI](https://electricalvoice.com/8-bit-logical-compare-instructions-8085-microprocessor/) | Data (8) |  | FE | 2 | 3 |
| 46. | CPO | Address (16) |  | E4 | 3 | 5 |
| 47. | CZ | Address (16) |  | CC | 3 | 5 |
| 48. | DAA |  |  | 27 | 1 | 1 |
| 49. | DAD | B |  | 09 | 1 | 2 |
| 50. | DAD | D |  | 19 | 1 | 2 |
| 51. | DAD | H |  | 29 | 1 | 2 |
| 52. | DAD | SP |  | 39 | 1 | 2 |
| 53. | DCR | A |  | 3D | 1 | 2 |
| 54. | DCR | B |  | 05 | 1 | 2 |
| 55. | DCR | C |  | 0D | 1 | 2 |
| 56. | DCR | D |  | 15 | 1 | 2 |
| 57. | DCR | E |  | 1D | 1 | 2 |
| 58. | DCR | H |  | 25 | 1 | 2 |
| 59. | DCR | L |  | 2D | 1 | 2 |
| 60. | DCR | M |  | 35 | 1 | 2 |
| 61. | DCX | B |  | 0B | 1 | 2 |
| 62. | DCX | D |  | 1B | 1 | 2 |
| 63. | DCX | H |  | 2B | 1 | 2 |
| 64. | DCX | SP |  | 3B | 1 | 2 |
| 65. | DI |  |  | F3 | 1 | 1 |
| 66. | EI |  |  | FB | 1 | 1 |
| 67. | HLT |  |  | 76 | 1 | 1 |
| 68. | IN | PORT Address (8) |  | DB | 2 | 4 |
| 69. | INR | A |  | 3C | 1 | 2 |
| 70. | INR | B |  | 04 | 1 | 2 |
| 71. | INR | C |  | 0C | 1 | 2 |
| 72. | INR | D |  | 14 | 1 | 2 |
| 73. | INR | E |  | 1C | 1 | 2 |
| 74. | INR | H |  | 24 | 1 | 2 |
| 75. | INR | L |  | 2C | 1 | 2 |
| 76. | INR | M |  | 34 | 1 | 2 |
| 77. | INX | B |  | 03 | 1 | 2 |
| 78. | INX | D |  | 13 | 1 | 2 |
| 79. | INX | H |  | 23 | 1 | 2 |
| 80. | INX | SP |  | 33 | 1 | 2 |
| 81. | JC | Address (16) |  | DA | 3 | 5 |
| 82. | JM | Address (16) |  | FA | 3 | 5 |
| 83. | JMP | Address (16) |  | C3 | 3 | 5 |
| 84. | JNC | Address (16) |  | D2 | 3 | 5 |
| 85. | JNZ | Address (16) |  | C2 | 3 | 5 |
| 86. | JP | Address (16) |  | F2 | 3 | 5 |
| 87. | JPE | Address (16) |  | EA | 3 | 5 |
| 88. | JPO | Address (16) |  | E2 | 3 | 5 |
| 89. | JZ | Address (16) |  | CA | 3 | 5 |
| 90. | LDA | Address (16) |  | 3A | 3 | 5 |
| 91. | LDAX | B |  | 0A | 1 | 2 |
| 92. | LDAX | D |  | 1A | 1 | 2 |
| 93. | LHLD | Address (16) |  | 2A | 3 | 5 |
| 94. | LXI B | B | Data (16) | 01 | 3 | 9 |
| 95. | LXI D | D | Data (16) | 11 | 3 | 9 |
| 96. | LXI H | H | Data (16) | 21 | 3 | 9 |
| 97. | LXI SP | SP | Data (16) | 31 | 3 | 9 |
| 98. | MOV | A | A | 7F | 1 | 7 |
| 99. | MOV | A | B | 78 | 1 | 7 |
| 100. | MOV | A | C | 79 | 1 | 7 |
| 101. | MOV | A | D | 7A | 1 | 7 |
| 102. | MOV | A | E | 7B | 1 | 7 |
| 103. | MOV | A | H | 7C | 1 | 7 |
| 104. | MOV | A | L | 7D | 1 | 7 |
| 105. | MOV | A | M | 7E | 1 | 7 |
| 106. | MOV | B | A | 47 | 1 | 7 |
| 107. | MOV | B | B | 40 | 1 | 7 |
| 108. | MOV | B | C | 41 | 1 | 7 |
| 109. | MOV | B | D | 42 | 1 | 7 |
| 110. | MOV | B | E | 43 | 1 | 7 |
| 111. | MOV | B | H | 44 | 1 | 7 |
| 112. | MOV | B | L | 45 | 1 | 7 |
| 113. | MOV | B | M | 46 | 1 | 7 |
| 114. | MOV | C | A | 4F | 1 | 7 |
| 115. | MOV | C | B | 48 | 1 | 7 |
| 116. | MOV | C | C | 49 | 1 | 7 |
| 117. | MOV | C | D | 4A | 1 | 7 |
| 118. | MOV | C | E | 4B | 1 | 7 |
| 119. | MOV | C | H | 4C | 1 | 7 |
| 120. | MOV | C | L | 4D | 1 | 7 |
| 121. | MOV | C | M | 4E | 1 | 7 |
| 122. | MOV | D | A | 57 | 1 | 7 |
| 123. | MOV | D | B | 50 | 1 | 7 |
| 124. | MOV | D | C | 51 | 1 | 7 |
| 125. | MOV | D | D | 52 | 1 | 7 |
| 126. | MOV | D | E | 53 | 1 | 7 |
| 127. | MOV | D | H | 54 | 1 | 7 |
| 128. | MOV | D | L | 55 | 1 | 7 |
| 129. | MOV | D | M | 56 | 1 | 7 |
| 130. | MOV | E | A | 5F | 1 | 7 |
| 131. | MOV | E | B | 58 | 1 | 7 |
| 132. | MOV | E | C | 59 | 1 | 7 |
| 133. | MOV | E | D | 5A | 1 | 7 |
| 134. | MOV | E | E | 5B | 1 | 7 |
| 135. | MOV | E | H | 5C | 1 | 7 |
| 136. | MOV | E | L | 5D | 1 | 7 |
| 137. | MOV | E | M | 5E | 1 | 7 |
| 138. | MOV | H | A | 67 | 1 | 7 |
| 139. | MOV | H | B | 60 | 1 | 7 |
| 140. | MOV | H | C | 61 | 1 | 7 |
| 141. | MOV | H | D | 62 | 1 | 7 |
| 142. | MOV | H | E | 63 | 1 | 7 |
| 143. | MOV | H | H | 64 | 1 | 7 |
| 144. | MOV | H | L | 65 | 1 | 7 |
| 145. | MOV | H | M | 66 | 1 | 7 |
| 146. | MOV | L | A | 6F | 1 | 7 |
| 147. | MOV | L | B | 68 | 1 | 7 |
| 148. | MOV | L | C | 69 | 1 | 7 |
| 149. | MOV | L | D | 6A | 1 | 7 |
| 150. | MOV | L | E | 6B | 1 | 7 |
| 151. | MOV | L | H | 6C | 1 | 7 |
| 152. | MOV | L | L | 6D | 1 | 7 |
| 153. | MOV | L | M | 6E | 1 | 7 |
| 154. | MOV | M | A | 77 | 1 | 7 |
| 155. | MOV | M | B | 70 | 1 | 7 |
| 156. | MOV | M | C | 71 | 1 | 7 |
| 157. | MOV | M | D | 72 | 1 | 7 |
| 158. | MOV | M | E | 73 | 1 | 7 |
| 159. | MOV | M | H | 74 | 1 | 7 |
| 160. | MOV | M | L | 75 | 1 | 7 |
| 161. | MVI | A | Data (8) | 3E | 2 | 8 |
| 162. | MVI | B | Data (8) | 06 | 2 | 8 |
| 163. | MVI | C | Data (8) | 0E | 2 | 8 |
| 164. | MVI | D | Data (8) | 16 | 2 | 8 |
| 165. | MVI | E | Data (8) | 1E | 2 | 8 |
| 166. | MVI | H | Data (8) | 26 | 2 | 8 |
| 167. | MVI | L | Data (8) | 2E | 2 | 8 |
| 168. | MVI | M | Data (8) | 36 | 2 | 8 |
| 169. | NOP |  |  | 00 | 1 | 1 |
| 170. | ORA | A |  | B7 | 1 | 2 |
| 171. | ORA | B |  | B0 | 1 | 2 |
| 172. | ORA | C |  | B1 | 1 | 2 |
| 173. | ORA | D |  | B2 | 1 | 2 |
| 174. | ORA | E |  | B3 | 1 | 2 |
| 175. | ORA | H |  | B4 | 1 | 2 |
| 176. | ORA | L |  | B5 | 1 | 2 |
| 177. | ORA | M |  | B6 | 1 | 2 |
| 178. | ORI | Data (8) |  | F6 | 2 | 3 |
| 179. | OUT | PORT Address (8) |  | D3 | 2 | 4 |
| 180. | PCHL |  |  | E9 | 1 | 1 |
| 181. | POP | B |  | C1 | 1 | 2 |
| 182. | POP | D |  | D1 | 1 | 2 |
| 183. | POP | H |  | E1 | 1 | 2 |
| 184. | POP | PSW |  | F1 | 1 | 2 |
| 185. | PUSH | B |  | C5 | 1 | 2 |
| 186. | PUSH | D |  | D5 | 1 | 2 |
| 187. | PUSH | H |  | E5 | 1 | 2 |
| 188. | PUSH | PSW |  | F5 | 1 | 2 |
| 189. | [RAL](https://electricalvoice.com/8-bit-logical-rotational-instructions-8085-microprocessor/) |  |  | 17 | 1 | 1 |
| 190. | [RAR](https://electricalvoice.com/8-bit-logical-rotational-instructions-8085-microprocessor/) |  |  | 1F | 1 | 1 |
| 191. | RC |  |  | D8 | 1 | 1 |
| 192. | RET |  |  | C9 | 1 | 1 |
| 193. | RIM |  |  | 20 | 1 | 1 |
| 194. | [RLC](https://electricalvoice.com/8-bit-logical-rotational-instructions-8085-microprocessor/) |  |  | 07 | 1 | 1 |
| 195. | RM |  |  | F8 | 1 | 1 |
| 196. | RNC |  |  | D0 | 1 | 1 |
| 197. | RNZ |  |  | C0 | 1 | 1 |
| 198. | RP |  |  | F0 | 1 | 1 |
| 199. | RPE |  |  | E8 | 1 | 1 |
| 200. | RPO |  |  | E0 | 1 | 1 |
| 201. | [RRC](https://electricalvoice.com/8-bit-logical-rotational-instructions-8085-microprocessor/) |  |  | 0F | 1 | 1 |
| 202. | RST | 0 |  | C7 | 1 | 6 |
| 203. | RST | 1 |  | CF | 1 | 6 |
| 204. | RST | 2 |  | D7 | 1 | 6 |
| 205. | RST | 3 |  | DF | 1 | 6 |
| 206. | RST | 4 |  | E7 | 1 | 6 |
| 207. | RST | 5 |  | EF | 1 | 6 |
| 208. | RST | 6 |  | F7 | 1 | 6 |
| 209. | RST | 7 |  | FF | 1 | 6 |
| 210. | RZ |  |  | C8 | 1 | 1 |
| 211. | SBB | A |  | 9F | 1 | 2 |
| 212. | SBB | B |  | 98 | 1 | 2 |
| 213. | SBB | C |  | 99 | 1 | 2 |
| 214. | SBB | D |  | 9A | 1 | 2 |
| 215. | SBB | E |  | 9B | 1 | 2 |
| 216. | SBB | H |  | 9C | 1 | 2 |
| 217. | SBB | L |  | 9D | 1 | 2 |
| 218. | SBB | M |  | 9E | 1 | 2 |
| 219. | SBI | Data (8) |  | DE | 2 | 3 |
| 220. | SHLD | Address (16) |  | 22 | 3 | 5 |
| 221. | SIM |  |  | 30 | 1 | 1 |
| 222. | SPHL |  |  | F9 | 1 | 1 |
| 223. | STA | Address (16) |  | 32 | 3 | 5 |
| 224. | STAX | B |  | 02 | 1 | 2 |
| 225. | STAX | D |  | 12 | 1 | 2 |
| 226. | STC |  |  | 37 | 1 | 1 |
| 227. | SUB | A |  | 97 | 1 | 2 |
| 228. | SUB | B |  | 90 | 1 | 2 |
| 229. | SUB | C |  | 91 | 1 | 2 |
| 230. | SUB | D |  | 92 | 1 | 2 |
| 231. | SUB | E |  | 93 | 1 | 2 |
| 232. | SUB | H |  | 94 | 1 | 2 |
| 233. | SUB | L |  | 95 | 1 | 2 |
| 234. | SUB | M |  | 96 | 1 | 2 |
| 235. | SUI | Data (8) |  | D6 | 2 | 3 |
| 236. | XCHG |  |  | EB | 1 | 1 |
| 237. | XRA | A |  | AF | 1 | 2 |
| 238. | XRA | B |  | A8 | 1 | 2 |
| 239. | XRA | C |  | A9 | 1 | 2 |
| 240. | XRA | D |  | AA | 1 | 2 |
| 241. | XRA | E |  | AB | 1 | 2 |
| 242. | XRA | H |  | AC | 1 | 2 |
| 243. | XRA | L |  | AD | 1 | 2 |
| 244. | XRA | M |  | AE | 1 | 2 |
| 245. | XRI | Data (8) |  | EE | 2 | 3 |
| 246. | XTHL |  |  | E3 | 1 | 1 |

Microprocessor 8085-A has 246 instructions. These instructions can be further classified into 9 groups for further processing by assembler as given below.

**Instruction Type Classification**

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction Type** | **Mnemonic** | **Operand1 , Operand2** | **Description** |
| **1** | Mnemonic | -----  **[ L(I) = 1 Byte ]** | Only Mnemonics, Op1 and Op2 both are absent  **(Only CMA, CMC, DAA, DI, EI, HLT, NOP, PCHL, all Rotate, all Return, RIM, SIM, SPHL, STC, XCHG, XTHL Instructions)** |
| **2** | Mnemonic | Rg  **[ L(I) = 1 Byte ]** | Rg : A, B, C, D, E, H, L, M, SP, PSW  (Only Op1 is present), (Op2 is absent)  **(All Arithmetic & Logical instructions with 1 operand only)** |
| **3** | Mnemonic | Data (8)  **[ L(I) = 2 Bytes ]** | (Only Op1 is present), (Op2 is absent)  **(Only all Immediate Data(8) with 1 operand Instructions: ACI,ADI,ANI,CPI,ORI,SUI,XRI)** |
| **4** | Mnemonic | Address (8)  **[ L(I) = 2 Bytes ]** | (Only Op1 is present), (Op2 is absent)  **(Only IN and OUT Instructions)**  **(Op1 is 8 bit address)** |
| **5** | Mnemonic | Address (16)  **[ L(I) = 3 Bytes ]** | All CALL and JMP Instructions  (Only Op1 is present), (Op2 is absent)  **(Only all: JUMP, CALL, LDA, LHLD, SHLD, STA Instructions)** |
| **6** | Mnemonic | N  **[ L(I) = 1 Byte ]** | N : 0, 1, 2, 3, 4, 5, 6, 7  **(Only All RST Instructions)** |
| **7** | Mnemonic | Rg , Rg  **[ L(I) = 1 Byte ]** | Rg : A, B, C, D, E, H, L, M  (Both Op1 & Op2 should not be ‘M’)  **(Only all MOV Instructions)** |
| **8** | Mnemonic | Rg , Data (8)  **[ L(I) = 2 Bytes ]** | Rg : A, B, C, D, E, H, L, M  **(Only all MVI Instructions with Op1 & Op2)**  **( Op1 is Rg and Op2 is Immediate Data(8) )** |
| **9** | Mnemonic | Rp , Data (16)  **[ L(I) = 3 Bytes ]** | Rp : B, D, H, SP  **(Only all LXI Instructions with 16 bit immediate data)** |

**Instruction Format used in Assembler development:**

LABEL : Mnemonic Operand1 , Operand2 // Comment OR

LABEL : Mnemonic Operand1 , Operand2 ; Comment

// Complete Comment Line OR

; Complete Comment Line

Therefore, the delimiters ‘ : ’ , ‘ , ’ ‘ ’ , ‘ // ’ , and ‘ = ’ should not be used in defining and referencing of LABEL, Operand1, and Operand2.

**This Means you cannot use:**

ADI DATA 8 // Space between Data and 8 is not permitted

DATA 8 : EQU 4FH // Maybe represented as 4F or 4FH



ADI DATA:8 // Colon between Data and 8 is not permitted

DATA:8 : EQU 4F // Maybe represented as 4F or 4FH

**Data (8) May possibly be represented as:**

1. By using label

ADI DATA8

DATA8 : EQU 4FH // Immediate data is defined by EQU

1. By using literal constant

ADI =4FH // Hex 4F OR

ADI =10D // Decimal 10

In the case of (=10D) the decimal number 10 will be converted into its hexadecimal equivalent, i.e. ‘A’.

**Data (16) May possibly be represented as:**

1. By using label

LXI H, DATA16

DATA16 : EQU 20F0H

1. By using literal constant

LXI H, =20F0H // Hex 20F0 OR

LXI H, =1547D // Decimal 1547

In the case of (=1547D) the decimal number 1547 will be converted into its hexadecimal equivalent, i.e., ‘060B’.

ADI H, =10D // Decimal 1256

In the case of (=10D) the decimal number 10 will be converted into its hexadecimal equivalent, i.e., ‘0A’.

**Address (8) May possibly be represented as:**

1. By using label

IN ADDRESS8

ADDRESS8 : EQU 01 // May also be written as 01H

1. By using literal constant

IN =01H // Hex 4F OR

IN =10D // Decimal 10

In the case of (=10D) the decimal number 10 will be converted into its hexadecimal equivalent, i.e. ‘A’.

**Address (16) May possibly be represented as:**

1. By using label

JMP ADDRESS16

ADDRESS16 : EQU 20F0

1. By using literal constant

JMP =20F0H // Hex 4F OR

JMP =1256D // Decimal 10

In the case of (=1025D) the decimal number 1256 will be converted into its hexadecimal equivalent, i.e., ‘04E8’.

Pass 1 MOT searching is based on only Mnemonics

Pass 2 MOT searching is based on following criterion:

Type : 1, 3, 4, 5 -- No further search is required

Just return with MOTIndex value.

Type : 2, 6, 8, 9 -- Concatenate (Mnemonics||Operand1)

Type : 7 -- Concatenate (Mnemonics||Operand1||Operand2)

**Data Structures used in implementation of an Assembler**

The Source program means the assembly language program written for the microprocessor 8085. It is stored in a text file which will serve as the input to the assembler. The assembler will have 2 passes to generate the assembled program. Pass 1 will take the Source program as an input and generates intermediate output file to be used as an input by Pass2. The purpose of Pass1 is to generate Symbol Table (ST). Pass 2 will use the intermediate output and symbol table files generated by Pass1 and will generate the final assembled program as an output file. The data structures used in the implementation are described as follows.

**Source Program File:**

The source program file contains assembly language program statements for the microprocessor 8085. The format of these statements will be as follows:

LABEL : MNEMONIC Operand1 , Operand2 // Comment

Mnemonics will be machine ops or pseudo ops. For example following are few statements in the source program file.

PROG : START \*

MVI A, =64H // Load accumulator with 64 hexadecimal

LOOP : DCR A

CPI A, =00H // Compare accumulator with 00 hexadecimal

JZ LOOP // Jump to LOOP if the result is 0

END

Pass 1 will systematically read the source program line by line and separate each line into fields (Label, Mnemonic, Operand1, Operand2, and Comment) for further processing. The structure used to hold these fields is as follows

**Preprocessor Symbols Used for Defining the Source Program Instruction:**

#define LABEL\_LENGTH 11

#define MNEMONIC\_LENGTH 9

#define OPERAND1\_LENGTH 11

#define OPERAND2\_LENGTH 11

#define COMMENT\_LENGTH 50

**Structure used to represent Source Program Instruction:**

struct SPROG // PROG stands for program

{

char Label[LABEL\_LENGTH + 1];

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

char Comment[COMMENT\_LENGTH +1] ;

};

**Structural variable used to hold Source Program Instruction:**

struct SPROG SourceInst;

for example:

LOOP1: LXI H, =20F0H // This is a comment

This instruction will be stored in the source instruction variable (SourceInst) as

SourceInst.Label = LOOP1

SourceInst.Mnemonic = LXI

SourceInst.Operand1 = H

SourceInst.Opernad2 = =20F0H

SourceInst.Comment = // This is a comment

**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 |

If the operand1 is not present it is represented by ‘??’ and when operand2 is not present it will be represented by ‘!!’.

**Preprocessor Symbols used for defining the MOT structure**

#define MNEMONIC\_LENGTH 9

#define OPERAND1\_LENGTH 11

#define OPERAND2\_LENGTH 11

#define OPCODE\_LENGTH 2

**Structure used to represent MOT**

struct MOT // For MachineOp Table

{

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

char OpCode[OPCODE\_LENGTH + 1];

int Length;

int Type;

};

**Structural variable used to hold MOT:**

#define N 247

struct MOT MOTInst[N];

Zeroth location is not used in this implementation. Therefore a structural array is declared of the size of 247 to hold 246 instructions (1 , … , 246).

Here zeroth location Is not used as it may be used as a return value from the MOTGet\_Pass1 function to indicate its failure when match is not found.

**Pseudo op Table (POT):**

char POT[ ][6] = { "ZZZZ", "DB", "DS", "DW", "END", "EQU", "ORG", "START", "\0" };

Zeroth location is not used in this implementation.

Here zeroth location Is not used as it may be used as a return value from the POTGet\_Pass1 function to indicate its failure when match is not found.

**Symbol Table (ST)**

|  |  |
| --- | --- |
| **Symbol/Label** | **Value** |
| String | String |
| 11  Characters | 04  Characters |
| 12 Bytes | 05 Bytes |
| PROG | 0000 |
| COUNTER | C |
| RP | B |
| OUTPUTPORT | 02 |
| : | : |
| : | : |
| DATA | 01F0 |

**Preprocessor Symbols used for defining the POT structure**

#define LABEL\_LENGTH 11

#define LABEL\_VALUE\_LENGTH 4 // Maximum label value will be FFFF

**Structure used to represent MOT**

struct SYMBOLTABLEDEF

{

char Symbol[LABEL\_LENGTH + 1];

char Value[LABEL\_VALUE\_LENGTH + 1];

};

**Structural variable to hold symbol table (ST) :**

#define MAX\_NO\_OF\_ST\_ENTRIES 200 // Max. no. of ST entries (Can be changed)

struct SYMBOLTABLEDEF ST[MAX\_NO\_OF\_ST\_ENTRIES]; // ST for Symbol Table

**Pass1 Output**

Pass 1 generates intermediate output in a text file “Pass1Output.txt”. The exact format of this file is as shown below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Source Program Line No** | **Location Counter Value** | **Label** | **Mnemonic** | **Operand 1** | **Operand 2** | **Instruction Type** | **Comment** |
| **Unsigned**  **integer** | **Unsigned**  **integer** | **String** | **String** | **String** | **String** | **Integer** | **String** |
| **02 Bytes** | **02 Bytes** | **11 Characters** | **O9 Characters** | **11 Characters** | **11 Characters** | **02 Bytes** | **50 Characters** |
| 0001 | 0000 | PROG | START | 20F0 | ? | -99 | // Line 01 |
| 0002 | 0000 | BEGIN | MVI | A | =64H | 8 | ? |
| 0003 | 0002 | ? | NOP | ? | ? | 1 | ? |
| 0004 | 0003 | ? | DCR | A | ? | 2 | ? |
| 0005 | 0004 | ? | JNZ | BEGIN | ? | 5 | // Jump to BEGIN |
| 0006 | 0007 | ? | HLT | ? | ? | 1 | ? |
| 0007 | 0007 | ? | END | ? | ? | -99 | // END |

The fields which are absent are filed with ‘?’ for ease of further processing.

For machine-ops the instruction types are between 1 to 9 and for pseudo-ops it will be -99.

**Preprocessor Symbols used for defining Pass1 Output structure**

#define LABEL\_LENGTH 11

#define MNEMONIC\_LENGTH 9

#define OPERAND1\_LENGTH 11

#define OPERAND2\_LENGTH 11

#define COMMENT\_LENGTH 50

**Structure used to represent Pass1 intermediate Output**

struct P1OUTPUTDEF

{

unsigned int LN;

unsigned int LC;

char Label[LABEL\_LENGTH + 1];

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

int IType;

char Comment[COMMENT\_LENGTH + 1];

char LN\_Output\_Flag;

};

**Structural variable to hold Pass1 Intermediate Output :**

struct P1OUTPUTDEF P1OutputInst;

**PASS 2 :**

Pass 2 will use the files SymbolTable.txt and Pass1Output.txt generated by Pass 1 as an input. Pass 2 will read lines one by one from the Pass1Output.txt and store it in the structural variable “P1OutputInst” as shown below.

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 |

This line will be represented in the structural variable as follows :

P1OutputInst.LN = 02 // Source Program line number

P1OutputInst.LC = 20F0

P1OutputInst.Label = BEGIN

P1OutputInst.Mnemonic = MVI

P1OutputInst.Operand1 = A

P1OutputInst.Operand2 = =64H

P1OutputInst.IType = 8

P1OutputInst.Comment = Initialize A

Pass 2 will process each line from the file P1Output.txt and assemble each instruction. The assembled instructions will be stored in output file named “P2Output.txt”.

**Pass 2 Output File Format ( format of “P2Output.txt” ) :**

Pass 2 will generate the final assembled output which is to be stored in a text file “Pass2Output.txt”. The format of this final output file will be as follows:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Memory Address (LC Value)** | **Opcode** | **Label** | **Mnemonic** | **Operand 1** | **Operand 2** | **Comment** | **Source Program Line No** |
| **Hex** | **String** | **String** | **String** | **String** | **String** | **String** | **Unsigned**  **integer** |
| **02 Bytes** | **02 Characters** | **11 Characters** | **O9 Characters** | **11 Characters** | **11 Characters** | **50 Characters** | **02 Bytes** |
| 0000 |  | PROG | START | 20F0 | ? | // PName | S\_LN# 1 |
| 20F0 | 3E | BEGIN | MVI | A | =64H | // Count | S\_LN# 2 |
| 20F1 | 64 |  |  |  |  |  |  |
| 20F2 | 00 |  | NOP | ? | ? | ? | S\_LN# 3 |
| 20F3 | 3D |  | DCR | A | ? | ? | S\_LN# 4 |
| 20F4 | C2 |  | JNZ | BEGIN | ? | // Jump to BEGIN | S\_LN# 5 |
| 20F5 | F0 |  |  |  |  |  |  |
| 20F6 | 20 |  |  |  |  |  |  |
| 20F7 | 76 |  | HLT | ? | ? | ? | S\_LN# 6 |
|  |  |  | END | ? | ? | // END | S\_LN# 7 |

**Choice for printing “Source Program Line Numbers” in final assembled file:**

The developed program asks the user:

Do you need Source Program Line Numbers in assembled File (Y/N)::

The user response will be recorded in “LN\_Output\_Flag” variable and passed it to Assembler\_Pass2() function and subsequently copied in “P1OutputInst.LN\_Output\_Flag” variable. While writing in pass2 output file if user response is ‘Y’ the source program line numbers will be printed after comment in final assembled file. If the user response is other than “Y”, it will be treated as “N” and the numbers will not be printed in final assembled file. So the last column of the above table will not be present in the final assembled file. In such case the output file structure will be as shown below:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Memory Address (LC Value)** | **Opcode** | **Label** | **Mnemonic** | **Operand 1** | **Operand 2** | **Comment** |
| **Hex** | **String** | **String** | **String** | **String** | **String** | **String** |
| **02 Bytes** | **02 Characters** | **11 Characters** | **O9 Characters** | **11 Characters** | **11 Characters** | **50 Characters** |
| 0000 |  | PROG | START | 20F0 | ? | // PName |
| 20F0 | 3E | BEGIN | MVI | A | =64H | // Count |
| 20F1 | 64 |  |  |  |  |  |
| 20F2 | 00 |  | NOP | ? | ? | ? |
| 20F3 | 3D |  | DCR | A | ? | ? |
| 20F4 | C2 |  | JNZ | BEGIN | ? | // Jump to BEGIN |
| 20F5 | F0 |  |  |  |  |  |
| 20F6 | 20 |  |  |  |  |  |
| 20F7 | 76 |  | HLT | ? | ? | ? |
|  |  |  | END | ? | ? | // END |

**Program Structure:**

**main() :**

1. **Assembler\_Pass1()**
2. DeleteExcessWhiteSpaces\_Pass1()
3. GetFields\_Pass1()
4. IsPseudoOp\_Pass1()
5. **ProcessPseudoOp\_Pass1()**
6. ProcessORGorSTART\_Pass1()
7. Are\_All\_Hexadecimal\_Digits()
8. STSTO\_Pass1()
9. ProcessEQU\_Pass1()
10. Is\_Register()
11. Are\_All\_Hexadecimal\_Digits()
12. Are\_All\_Decimal\_Digits()
13. STSTO\_Pass1()
14. ProcessDS\_Pass1()
15. Are\_All\_Hexadecimal\_Digits()
16. Are\_All\_Decimal\_Digits()
17. STSTO\_Pass1()
18. ProcessDB\_Pass1()
19. STSTO\_Pass1()
20. ProcessDW\_Pass1()
21. STSTO\_Pass1()
22. ProcessEND\_Pass1()
23. STSTO\_Pass1()
24. Write\_Pass1\_OutputToFile()
25. MOTGet\_BS\_Pass1()
26. STSTO\_Pass1()
27. Write\_Pass1\_OutputToFile()
28. CheckSymbolTableForDuplicates\_Pass1( )
29. **Assembler\_Pass2()**
30. ReadSymbolTable\_Pass2( ST )
31. GetP1OutputInst\_Pass2()
32. IsPseudoOp\_Pass1()
33. **ProcessPseudoOps\_Pass2()**
34. ProcessORGorSTART\_Pass2()
35. ProcessEQU\_Pass2()
36. ProcessDS\_Pass2()
37. Are\_All\_Hexadecimal\_Digits()
38. Are\_All\_Decimal\_Digits()
39. ProcessDB\_Pass2()
40. Are\_All\_Hexadecimal\_Digits()
41. Are\_All\_Decimal\_Digits()
42. ProcessDW\_Pass2()
43. Are\_All\_Hexadecimal\_Digits()
44. Are\_All\_Decimal\_Digits()
45. ProcessEND\_Pass2()
46. **ProcessMachineOps\_Pass2()**
47. **ProcessMOType1\_Pass2()**

* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType26\_Pass2()**

* STGet\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType3\_Pass2()**

* STGet\_Pass2()
* ProcessLiteral\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType4\_Pass2()**

* STGet\_Pass2()
* ProcessLiteral\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType5\_Pass2()**

* STGet\_Pass2()
* ProcessLiteral\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType7\_Pass2()**

* STGet\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType8\_Pass2()**

* STGet\_Pass2()
* ProcessLiteral\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

1. **ProcessMOType9\_Pass2()**

* STGet\_Pass2()
* ProcessLiteral\_Pass2()
* **MOTGet\_BS\_Pass2()**
* Error\_Pass2()

**SetColor()** → **Is called by every function to display various interactive messages in different colours to assist user.**

**All Preprocessor Definitions:**

#define N 247 // No of entries in MOT are 246. 0th location is not used.

#define LABEL\_LENGTH 11

#define MNEMONIC\_LENGTH 9

#define OPERAND1\_LENGTH 11

#define OPERAND2\_LENGTH 11

#define OPCODE\_LENGTH 2

#define LABEL\_VALUE\_LENGTH 4 // Maximum label value will be FFFF

#define COMMENT\_LENGTH 50

#define MAX\_NO\_OF\_ST\_ENTRIES 200 // ST stands for Symbol Table

**All Structural Definitions:**

struct MOT // For MachineOp Table

{

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

char OpCode[OPCODE\_LENGTH + 1];

int Length;

int Type;

};

struct SPROG // SPROG stands for Source PROGram

{

char Label[LABEL\_LENGTH + 1];

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

char Comment[COMMENT\_LENGTH +1] ;

};

struct SYMBOLTABLEDEF // Symbol Table Definition

{

char Symbol[LABEL\_LENGTH + 1];

char Value[LABEL\_VALUE\_LENGTH + 1];

};

struct P1OUTPUTDEF // Pass 1 Output

{

unsigned int LN;

unsigned int LC;

char Label[LABEL\_LENGTH + 1];

char Mnemonic[MNEMONIC\_LENGTH + 1];

char Operand1[OPERAND1\_LENGTH + 1];

char Operand2[OPERAND2\_LENGTH + 1];

int IType;

char Comment[COMMENT\_LENGTH + 1];

char LN\_Output\_Flag;

};

**Function Prototype Used:**

1. int Assembler\_Pass1( struct MOT MOTInst[], char POT[][6], char SourceFileName[] );
2. char \*DeleteExcessWhiteSpaces\_Pass1( char str[] );
3. struct SPROG GetFields\_Pass1( char Instruction[] );
4. int ReadMachineOpTable( struct MOT MOTInst[] );
5. int MOTGet\_LS\_Pass1( struct MOT MOTInst[], char Mnemonic[] );
6. int MOTGet\_BS\_Pass1( struct MOT MOTInst[] , char Mnemonic[] );
7. void STSTO\_Pass1( FILE \*fp2, struct SPROG SourceInst, unsigned int LC );
8. int IsPseudoOp\_Pass1( char Mnemonic[], char POT[][6] );
9. int ProcessPseudoOps\_Pass1( int p, struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
10. int ProcessORGorSTART\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2 , unsigned int LN );
11. int ProcessEQU\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
12. int ProcessDS\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
13. int ProcessDB\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
14. int ProcessDW\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
15. int ProcessEND\_Pass1( struct SPROG SourceInst, unsigned int \*LC, FILE \*fp2, unsigned int LN );
16. int CheckSymbolTableForDuplicates\_Pass1( void );
17. int Write\_Pass1\_OutputToFile( struct SPROG SourceInst, unsigned int LC, FILE \*fp3, int Type, unsigned int LN );
18. int Are\_All\_Hexadecimal\_Digits( char str[] );
19. int Are\_All\_Decimal\_Digits( char str[] );
20. int Is\_Register( char str[] );

///\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

1. int Assembler\_Pass2( struct MOT MOTInst[], char POT[][6], char LN\_Output\_Flag );
2. struct P1OUTPUTDEF GetP1OutputInst\_Pass2( FILE \*P1OutputFP );
3. int ReadSymbolTable\_Pass2( struct SYMBOLTABLEDEF ST[] );
4. int STGet\_Pass2( char Operand1[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries );
5. int ProcessPseudoOps\_Pass2( struct P1OUTPUTDEF P1OutputInst, FILE \*P2OutputFP, unsigned int p );
6. int ProcessORGorSTART\_Pass2( struct P1OUTPUTDEF P1OutputInst, FILE \*P2OutputFP );
7. int ProcessEND\_Pass2( struct P1OUTPUTDEF P1OutputInst, FILE \*P2OutputFP );
8. int ProcessEQU\_Pass2( struct P1OUTPUTDEF P1OutputInst, FILE \*P2OutputFP );
9. int ProcessDS\_Pass2( struct P1OUTPUTDEF P1OutputInst, FILE \*P2OutputFP );
10. int ProcessDB\_Pass2( struct P1OUTPUTDEF Temp, FILE \*P2OutputFP );
11. int ProcessDW\_Pass2( struct P1OUTPUTDEF Temp, FILE \*P2OutputFP );
12. int MOTGet\_LS\_Pass2( struct MOT MOTInst[], char ConcatenatedString[], struct P1OUTPUTDEF P1OutputInst );
13. int MOTGet\_BS\_Pass2( struct MOT MOTInst[], char ConcatenatedString[], struct P1OUTPUTDEF P1OutputInst );
14. int ProcessMachineOps\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
15. int ProcessMOType1\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], FILE \*P2OutputFP );
16. int ProcessMOType26\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
17. int ProcessMOType3\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
18. int ProcessMOType4\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
19. int ProcessMOType5\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
20. int ProcessMOType7\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
21. int ProcessMOType8\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
22. int ProcessMOType9\_Pass2( struct P1OUTPUTDEF P1OutputInst, struct MOT MOTInst[], struct SYMBOLTABLEDEF ST[], int NoOfSTEntries, FILE \*P2OutputFP );
23. int Error\_Pass2( struct P1OUTPUTDEF P1OutputInst );
24. int ProcessLiteral\_Pass2( char \*Operand, int LiteralLen, unsigned int LN ) ;
25. void SetColor( int ForgC );

**Pseudo Ops:**

A pseudo-op is an assembly language instruction that specifies an operation of the assembler; it is distinguished from a machine-op which represents to the assembler a machine instruction. Pseudo-op will not be converted in an equivalent machine instruction that can be executed by the underlying microcomputer / microprocessor for which the assembler is written. Pseudo-op only provides information to the assembler required for *assembling* (convention to machine instructions) of the assembly language program.

Following are the pseudo-ops implemented with the current project.

**START OR ORG:**

START is a pseudo-op that tells the assembler where the beginning of the program is and allows the user to give a name to the program.

PROG1 : START 20F0H or PROG1 : ORG 20F0H

In this case the name of the program is PROG1 and it starts in main memory from address 20F0H. The first executable instruction of the program starts from an address 20F0H. There may be more than 1 ORG or ORG pseudo-ops in a single program. Subroutine may also start from START or ORG pseudo-op such as:

PROG1 : START 20F0H

------------- // Definition part of prog1

CALL SUBROUTINE // Call subroutine

------------- // Definition part of prog1

END // End of the program PROG1

SUBROUTINE : START 30F0H

------------- // Definition of Subroutine

RET // Return to the calling program

END // END of SUBROUTINE definition.

Only 16 bit Hexadecimal address is permitted in operand1 field. Decimal Numbers are not permitted here. Therefore, only ‘H’ or ‘h’ data type modifier is allowed, ‘I’ or ‘i’ is not allowed in operand1 field. Label is optional.

Following are few examples:

PROG1 : START 20F0H

PROG1 : START 20F0

PROG1 : START F0H // Treated as 00F0H

PROG1 : START F0 // Treated as 00F0H

: ORG 30F0 // Label is optional

PROG1 : START 20FZH // ERROR : Z is not HEX DIGIT

PROG1 : START 20FZ // ERROR : Z is not HEX DIGIT

**Define Byte (DB):**

8-bit hexadecimal or decimal constant is defined using DB pseudo op. The data type modifiers ‘H’ or ‘h’ are used for defining hexadecimal constants, and ‘I’ or ‘i’ is used to define decimal constants. As the internal processing by the 8085 CPU is in binary, the decimal constants defined by data type modifier ‘I’ or ‘i’ is converted into hexadecimal constant in final assembled program. Label is optional while defining a byte.

Following are few examples:

DATA : DB 3FH

DATA : DB 3F

DATA : DB 3 // Treated as 03H

// (as H or I is not mentioned by default, constant is treated as hex)

: DB FF // Label is optional

DATA : DB 255I // Treated as FF

DATA : DB 255 // ERROR, as I is not mentioned, treated as hex, 255 is 12 bit long, i.e. > FF

DATA : DB 256I // ERROR, 256 > FF

**Define Word (DW):**

16-bit hexadecimal or decimal constant is defined using DW pseudo op. The data type modifiers ‘H’ or ‘h’ are used for defining hexadecimal constants, and ‘I’ or ‘i’ is used to define decimal constants. As the internal processing by the 8085 CPU is in binary, the decimal constants defined by data type modifier ‘I’ or ‘i’ is converted into hexadecimal constant in final assembled program. Label is optional while defining a byte.

Following are few examples:

DATA : DW F0F0H

DATA : DW F0F0

DATA : DW F // Treated as 000FH

DATA : DW 56535I // Treated as FFFFH

DATA : DW 2I // Treated as 0002H

DATA : DW 56536I // ERROR (HEX equivalent > FFFFH)

DATA : DW 56536 // ERROR, Default case, Treated as Hex which is 56536 > FFFFH

DATA : DW F0FZ // ERROR, Default case, Treated as Hex, Z is not Hex digit

DATA : DW F0FZH // ERROR, Z is not Hex digit

The 16 bit data is converted in 2 bytes as higher byte and lower byte. Lower byte is stored in lower memory location and higher byte is stored in higher consecutive address as

30F0 F0 DATA : DW 20F0H

30F1 20

**Define Storage (DS):**

Reserves memory area in terms of given number of bytes.

The data type modifiers ‘H’ or ‘h’, and ‘I’ or ‘i’ are used for requesting number of bytes to be reserved. The hexadecimal constants used for requesting number of bytes to be reserved, are converted into integer constants and the location counter is incremented by the same amount. Label is optional while reserving the required memory locations.

Following are few examples:

STORE1 : DS AH // Reserve 10 Bytes area

STORE1 : DS A // Reserve 10 Bytes area

STORE1 : DS 3F // Reserve 3 Bytes area

STORE1 : DS 3I // Reserve 3 Bytes area

STORE1 : DS 10I // Reserve 10 Bytes area

STORE1 : DS 10 // Reserve 10 Bytes area

: DS 10 // Reserve 10 Bytes area, Label is optional

**EQU Pseudo-op:**

The EQU pseudo-op allows the programmer to define variables. It acts as a preprocessor.

For example

COUNTER : EQU C

Indicates wherever ‘COUNTER’ appears in the program it will be replaced by ‘C’. In case, if the programmer wished to use a different register as a counter, he would need only to change the EQU statement. Or

DLYCOUNTER : EQU FFFFH

Means wherever ‘DLYCOUNTER’ appears in the program it will be replaced or evaluated as ‘FFFFH’.

Remember:

1. If the value of symbol defined by EQU pseudo-op is having length less than 2 hex digits, it is assumed that the user want to define 2 hex digit (1 Byte) long symbol and hence a 0 is appended as higher nibble, for example

COUNT\_VAL : EQU F

Treated as

COUNT\_VAL : EQU 0F

And the programmer is warned about the same.

COUNT\_VAL : EQU 10I // Treated as 0A (Converted to HEX)

Treated as

COUNT\_VAL : EQU 0A

And the programmer is **NOT warned** about it.

1. Similarly, If the value of symbol defined by EQU pseudo-op is having length less than 4 and greater than 2 hex digits, it is assumed that the user want to define 4 hex digit (2 Byte) long symbol and hence 0 is appended as higher nibble, for example

COUNT\_VAL : EQU FFA // Treated as 0FFA (Programmer is warned)

COUNT\_VAL : EQU 256I // Treated as 0100H (Converted to HEX)

Here, the programmer is **NOT warned** about it.

Following are few examples:

COUNT\_VAL : EQU FF0FH

COUNT\_VAL : EQU FF0F

COUNT\_VAL : EQU FFAH // Treated as 0FFA

COUNT\_VAL : EQU FFA // Treated as 0FFA

COUNT\_VAL : EQU FFH // Treated as FF

COUNT\_VAL : EQU FF // Treated as FF

COUNT\_VAL : EQU FH // Treated as 0F

COUNT\_VAL : EQU F // Treated as 0F

COUNT\_VAL : EQU 56535I // Treated as FFFF (Converted to HEX)

COUNT\_VAL : EQU 10I // Treated as 0A (Converted to HEX)

COUNT\_VAL : EQU 10 // Treated as 10H (Default Case)

COUNT\_VAL : EQU 255I // Treated as FFH (Converted to HEX)

COUNT\_VAL : EQU 256I // Treated as 0100H (Converted to HEX)

COUNT\_VAL : EQU 56536I // Error : Treated as 10000H (Converted to HEX) > FFFFH

COUNT\_VAL : EQU 20FZH // Error : Z is not HEX digit

COUNT\_VAL : EQU 20FZ // Error : Z is not HEX digit

COUNT\_VAL : EQU 56536I // Error : Treated as 10000H (Converted to HEX) > FFFFH

COUNT\_VAL : EQU 5653PI // Error : P is not decimal Digit

REGC : EQU C

PSW1 : EQU PSW

RP : EQU B

ADDRESS : EQU \* // ADDRESS will have value as current value of LC

**END pseudo-op:**

End is the pseudo-op that tells the assembler that the last statement / instruction of the program has been reached. It does not mean that the end of the program, it just means that it is the last instruction of that definition is reached. There may be more than 1 end in a single program as explained below.

PROG1 : START 20F0H

------------- // Definition part of prog1

CALL SUBROUTINE // Call subroutine

------------- // Definition part of prog1

END // End of the program PROG1

SUBROUTINE : START 30F0H

------------- // Definition of Subroutine

RET // Return to the calling program

END // END of SUBROUTINE definition.

**Literal Processing:**

Literals are mechanisms by which the assembler creates data areas for the programmer, containing constants he requested

Literal constants may be of 1 byte or 2 byte in length. They are used to force immediate data constants in the instruction itself. Literals may be hexadecimal or decimal. Decimal literals are converted to hex constants for further processing. To identify the literal constant definition ‘=’ sign is used.

* Hexadecimal literals may be defined by using ‘H’ or ‘h’ data type modifier. If the literals are defined by ‘H’ or ‘h’ modifier, and if they are not composed of hexadecimal digits, an appropriate error message is flagged.
* Decimal literals are to be defined by using ‘I’ or ‘i’ data type modifiers. If the literals are defined by ‘I’ or ‘i’ modifier, and if they are not composed of decimal digits, an appropriate error message is flagged.
* If the data type modifiers are not used at all, the literal constant are to be assumed to be of hexadecimal type, and if they are not composed of hex digits, an appropriate error message is flagged.

Following are few examples:

MVI A, =FFH // Treated as FFH

MVI A, =FF // Treated as FFH

MVI A, =20I // converted to Hex, 20I means 14 Hex

MVI A, =20 // Treated as 20H , Default case

LXI H, =30F0H // Treated as 30F0H

LXI H, =65535I // converted to Hex, 65535I means FFFFH Hex

LXI H, =65536I // ERROR : converted to Hex, 65536I means 10000H > FFFFH

MVI A, =FZH // ERROR : Z is not hex digit

MVI A, =FZ // ERROR : Z is not hex digit, Default case

LXI H, =655Z5I // ERROR : Z is not decimal digit.

In a machine instruction if 8-bit literal constant is expected, but the constant value is only 4-bit (1 hex digit) long, and the lower nibble (4-bit) is not mentioned, a higher nibble of 4-bit (1 hex digit) is appended and the programmer is warned about the same. This is shown below

MVI B, =FH // 8-bit immediate data required as a literal constant

Will be treated as

MVI B, =0FH

And the programmer is warned about the same.

Similarly, in a machine instruction if 16-bit literal constant is expected, but the constant value is only 4, 8, or 12-bit (1,2, or 3 hex digit) long, and the lower nibbles are not mentioned, appropriate number of higher nibbles are appended and the programmer is warned about the same. This is shown below

LXI H, =FH // 16-bit immediate data required as a literal constant

Will be treated as

LXI H, =000FH

And the programmer is warned about the same.

