# Computer Organisation

# Assembler Project

**Submitted by: Tanya Sanjay Kumar - 2018109**
**Vibhu Agrawal - 2018116**

**Objective:** To make an assembler for 12-bit accumulator architecture that can handle the following OPCODES:

| Opcode | Meaning | Assembly Opcode |
|---|---|---|
| 0000 | Clear accumulator | CLA |
| 0001 | Load into accumulator from address | LAC |
| 0010 | Store accumulator contents in address | SAC |
| 0011 | Add address contents to accumulator contents | ADD |
| 0100 | Subtract address contents from accumulator contents | SUB |
| 0101 | Branch to address if accumulator contains zero | BRZ |
| 0110 | Branch to address if accumulator contains negative value | BRN |
| 0111 | Branch to address if accumulator contains positive value | BRP |
| 1000 | Read from terminal and put in address | INP |
| 1001 | Display value in address on terminal | DSP |
| 1010 | Multiply accumulator and address contents | MUL |

| | | |
|---|---|---|
| 1011 | Divide accumulator contents by address content. Quotient in R1 and remainder in R2 | DIV |
| 1100 | Stop execution | STP |

Along with this variables and macros have to be handled.

**Background:**

Assembler is a piece of code that converts assembly level language instructions to corresponding machine code for the computer hardware to understand and execute the actual instruction. To perform the conversion, the assembler scans through the file line by line. Every scan through the file is termed as a pass. Assemblers can be implemented in 1-pass, 2-pass and multi-pass formats. However in one pass, the object code created is directly bound to physical memory addresses and it does not provide the flexibility of relocation. In 2-pass assembler, relocation can occur as only virtual memory mapping is done in the object code. The binding to physical addresses occurs during the loading phase which is performed by the loader. In multi-pass, the further passes are meant for code optimisation to reduce usage of hardware resources.

**Implementation:**

**<<The code has been implemented using Java 11.>>**

The assembler created takes a text file as input and converts it to the corresponding binary and save to the provided text file or if an invalid output file is provided then save the output to default output.txt. An intermediate file is created named intermediate_<input_file_name>.txt after macro expansion.

Input Format:

- A START directive in the beginning.
- An END directive in the end.
- A label is followed by a colon.
- A comment begins with “//”.
- Macro declaration is given as <macro_name> MACRO <parameters (space separated)>

- End of macro is indicated by ENDM or MEND.
- Macro is used as <macro_name> <arguments>.
- An opcode and an operand are separated by a space (“ “) character.
- Opcode should be supplied exactly as given in the table above. (Should be in uppercase.)

Output Format:

Two files are generated as an output. One is the object file that contains the conversion to binary. Second is the LabelNSymbolTable.txt which contains the labels and address mapping, followed by the word “Variables” and then variables and their addresses mapping.

The object file contains binary conversion of each instruction. The first 8 bits of each line are the virtual addresses allocated (on the basis of value of location counter). The next 4 bits represent the Opcode. The following 12 bits are the address of the variable or address as in the input.

Code Flow:

Assumptions:

- “111111111111” (12-1s or 4095) represent a reserved address supplied to operands CLA and STP, which don’t take in any parameter.
- R1 and R2 (outputs of DIV) have been assigned reserved addresses "000010001000" and "000010001001".
- Maximum value of location counter has been assumed to be 255 and maximum value of memory address as 4095.
- Output has been separated into lines just for the sake of readability.

Code explained:

The code comprises of 5 Java classes.

1) Assembler

This class converts a given file written in assembly level language into machine language. This files takes as input location of two files, the input and output file. It reads from the input file, calls the zeroth, first and second pass for file conversion and finally write the desired result into the output file.

2) FileConversion

This is the class which converts the input file into binary by passing the file through 3 passes. Pass 0 is used for macro expansion, which creates an intermediate file by the name intermediate_<inputfile_name>.txt where all the macros have been expanded with the help of MacroDefinitionTable. Pass 1 is used for storing all the labels and variables used in the input file into LabelsNSymbolsTable.txt along with the address allocated to them in the memory by the location counter. A SymbolTable and LabelTable is created in this pass. Then in Pass 2 the intermediate file is converted to binary based on the opcodes, symbol and label table.

3) LabelTable

During assembly, the labels are allocated the current value of the program counter, when their definition is found. This class is used for storing the labels along with their allocated addresses.

4) MacroDefinitionTable:

Macros are replicated at all points of call in the assembly program by the assembler. This is called macro expansion. So in order to achieve, macro expansion, MDT stores all the macro definitions and as a call is encountered it writes down the complete macro definition again with proper parameter substitutions. Also if there are labels in the macro, they are renamed in every invocation of the macro.

5) SymbolTable

This class contains the various variables used in the input along with the corresponding addresses allocated to them in the memory.

**Errors Handled:**

In the assembler created, a few errors and warnings have been handled. Here we classify error as an input which cannot be processed further, without making changes in the provided input. For eg: Not supplying any argument to a branch instruction is an error. Warning are those which can be resolved by the assembler by making some assumptions, however the output maybe a bit arbitrary. For eg: Supplying an operand to CLA, clear accumulator opcode has been treated as warning and in the output, operand has been simply ignored.

The conversion to binary is terminated in case of an error. However, in case of warnings, the execution is not terminated, but the user is appropriately warned.

Errors:

1. The input file provided does not exist.
2. The input file provided exists but does not provide read permissions.
3. A line does not have an Opcode or an invalid opcode has been provided.
4. R1 or R2 (Memory addresses reserved for Quotient and Remainder of DIV) have been used as labels.
5. Multiple definitions are found for a single label.
6. A label cannot be resolved i.e. a label has been used but not defined.
7. No operand is supplied to opcodes other than CLA and STP.
8. No definition is found for a label used.
9. Symbol Table cannot be written to hard disk.
10. Symbol table cannot be read back from hard disk.
11. Label Table cannot be written to hard disk.
12. Label Table cannot be read back from hard disk.
13. The memory address provided as an operand is not a valid one: Either it is negative or out of bounds (above 4095 in this case).
14. A macro is used before it is defined.
15. Excess or less arguments are supplied to a macro. Macro expansion is terminated.
16. Memory limit is exhausted, and cannot assign address to a variable.

Warnings and their corresponding assumptions and resolutions:

1. Too many arguments are supplied while invoking the Assembler. The first argument is assumed to be input file, second to be output file and the rest of the arguments are ignored.
2. The output file does not exist or it does not provide write permissions. In both cases the output is redirected to output.txt.
3. START assembler directive is not found. Binary conversion is started from the very beginning of the file.
4. A blank line is encountered. The line is simply ignored.
5. ENDM or MEND is not found and end of file is reached. The whole code after starting of macro is treated as part of macro. The code before the macro declaration begins is converted to binary.

6. A new macro starts while ENDM or MEND is not found for previous macro. The last line before the next macro declaration is assumed to be the end of previous macro.
7. While supplying operands to opcodes, excess operands are ignored. On the same grounds any operand supplied to CLA or STP is ignored.
8. END assembler directive is missing. EOF is treated as END.

**Running The Project:**

Once all the Java files have been compiled to produce corresponding class files, one of the following methods can be used to invoke the assembler:

1. java Assembler <input_filename.txt> <output_filename.txt>
2. java Assembler <input_filename.txt>

   The user is prompted to enter output file name.

3. java Assembler

   The user is prompted to enter both input and output file name.