

School of Science and Technology

## PROGRAMMING ASSIGNMENT DOCUMENTATION

**PROJECT NAME :** Meta Assembler

**STUDENT NAME :** Kristijan Zrno

**MODULE CODE :** SOFT10101: Computer Science Programming

**MODULE TITLE :** Computer Science Programming

**DATE :** 22.4.2018.

# **Specification**

# Overview

The goal of this project was to create a program that gives the user ability to write simple assembly language programs and easily transform them into Cedar Logic Memory format. The program allows the user to open, edit or create a new assembly language file, and load it into the editor which is the main part of the user interface. All supported instructions are contained inside of a file and user has complete ability to edit them using the program options.

Instruction table example:

|  |  |
| --- | --- |
| Instruction | Definition |
| *MOVEI* | *C* |
| *LDSP* | *7* |
| *BZ* | *FE* |
| *RTS* | *FFFB* |

Besides instructions, Assembler is also able to handle labels and directives. Labels can be used to hold the addresses for subroutines or to hold whole values (in combination with EQU directive). The program is able to decode 4 types of directives: ORG, EQU, DC, DS in the following format:

*ORG address*

*name EQU value (constant/label)*

*name DC value (constant/label)*

*name DS value (constant/label)*

Program is also able to differ hexadecimal and decimal values using a specific prefix before the value.

Decimal format: *#value or value*

Hexadecimal format: *$value*

After the programming is done and assembly program is completely written in the editor using different labels, instructions and directives, user is able to hit the ‘Convert to CDM’ button. This button will loop through the lines, search for instructions, labels or directives and decode them. During the conversion there are many heavy error checks on the code to ensure that the program is correctly written. If there are any errors the program will prompt them via status view located at the bottom of the screen. However, if the program passes all error checks it will compile the CDM format in a string and output it. The user is then able to save this program as a \*.CDM, \*.MASP or \*.TXT file.

Output example:

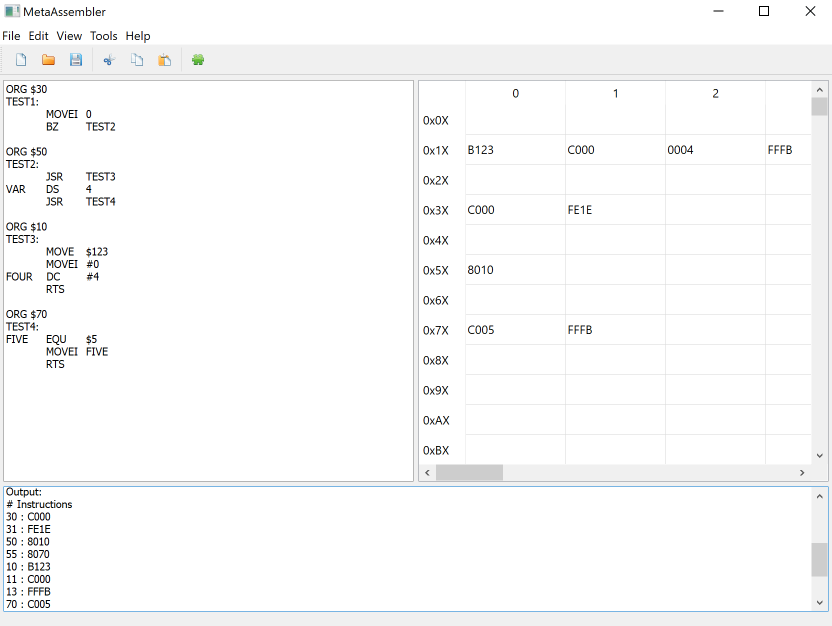
*30 : C000*

*31 : FE1E*

*50 : 8010*

Analysis of Requirements

1. User Interface



The program required an interface that would let the user to write a set of instructions in a ‘terminal’. This required a text box of some sort and this was implemented using Qt library. Actually, the whole program is designed using the gadgets provided by Qt. Next to the text editor lays a table view which represents the Cedar Logic Memory table. CDM Table consists of 256 rows with 16 columns and its purpose is to show the user CDM output in an efficient way. Each time user hits the ‘Convert to CDM’ button the table will get repopulated with output if the assembling process is successful. Under those two gadgets lays another text box. This one is used to show the user different messages related to the current process. For example, while assembling the program, if there are any present errors or warnings they will appear inside of this view.

1. File Handling

One of the most important program requirements was to give the user a safe and easy way to export the output as a CDM file. To save a program to a file user needs to hit the ‘Save’ button. This will show a dialog which asks the user where to save the project. This part of the project was done using Qt file dialogs. When saving the project there are 3 different modes (\*.cdm, \*.masp, \*.txt). If the user decides to save the project as CDM, then the project will first get assembled and the output string will be saved as a CDM file. However, if the user decides to save it as \*.txt or \*.masp file, then the assembly program from the editor will be saved in a file instead of the CDM output. File opening is done in a similar way. User can choose which file he wants to open using a Qt file dialog and if the file is read correctly, it will get loaded into the editor.

To ensure safe file handling, reading and writing procedures are separated into a ‘FileHandler’ class. This allows the program to use an instance of ‘FileHandler’ class and just pass it values each time it needs to save/open something instead of having file-handling bits of code all over the source files.

1. Data Storing

Since the program is working with loads of dynamic data, there was a requirement to find an efficient and safe way to store it somewhere. To separate instructions from labels and relatives, the program is using different classes for them. However, during the assembly process it is needed to have multiple instances of these classes to handle each instruction/label/directive. In order to fix this problem, the program used vectors of classes to safely store each instruction class and its attributes. The program also used maps to assign instruction definitions to instructions names.

1. Assembling

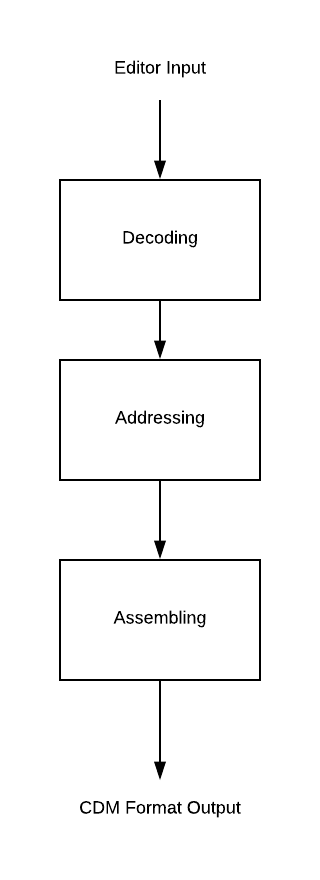
In the end, assembling is the main and essential requirement of the program. Assembler needs to be able to recognize each instruction, decode its value, differ hex from decimal values, differ constants from labels… The main idea is to go through written instructions line by line and keep each instruction for itself. While looping through instructions, it’s essential to do many error checks and stop the looping if there are any errors found.

# **Design**

|  |  |
| --- | --- |
| Class Name | Purpose and functions. |
| Assembler | Main part of the program. Takes the input from editor, loops through all the lines, decodes instructions, directives and labels, checks each line for errors, calculates addresses, outputs the CDM format… |
| ConversionUtils | This class is used as a utility which contains useful conversion functions that are needed all across the program (for example. string encoded value to integer) … |
| Directive | Used to create an instance of **directive** when it’s found in the code by assembler, stores directive information (address, value, type…). |
| FileHandler | Used all across the program, called each time when there’s a need to save/open something from somewhere. Safely handles the file reading and writing. |
| Instruction | Used to create an instance of **instruction** when it’s found in the code by assembler, stores directive information (address, line, definition…). |
| InstructionEditor | Handles the Instruction Editor window and is responsible for instruction editing. Takes the instruction list from Options window and edits it according to the user input. |
| Label | Used to create an instance of **label** when it’s found in the code by assembler, stores directive information (address, line, definition…). |
| MetaAssembler | Handles the main window and all of its actions and triggers. |
| Options | Handles the Options window, loads the instructions in a table view and enables the user to edit and rewrite them. |
| StatusOutput | Used to handle and output status messages from all over the program. Messages are identified by status code. |

Assembling Process

This is the main part of the program and it’s done in 3 cycles: decoding, addressing and assembling.



1. Decoding

Program takes the editor text on the input and sets up a while loop. This while loop will enable it to loop through each line from the input. In each line, program will first search for instruction, then directive and then a label. If nothing is found program proceeds to the next line. However, if for example, a instruction is found, the program will create a new Instruction object and set it up with its attributes loaded from the line. This is also a perfect opportunity to check for syntax errors cause the program won’t waste time and won’t proceed to the next process if there are any errors present. After creating the instruction/label object, the object is pushed into a proper vector. There are 3 vectors, one for instructions, one for labels and one for directives. However, we don’t create any directive objects here yet. Directives are essential part in addressing so in this cycle, we only process EQU directives which are directly tied with labels so the program will assign the label values to label objects. If the program passes every security check here, it will proceed to the next process (Addressing).

At the end of this stage, the program essentially has a list of labels and instructions.

1. Addressing

In this part we go back to the first line and start with the address 0. Once again, we loop through each line, check if there are any instructions/labels/directives. This time, if an instruction is found it will be identified in the instruction vector and assigned an address defined by address counter. Same goes for labels and directives. Each time an address was assigned (except in case of Labels), address counter will be iterated so the correct address can be assigned to the next instruction/label/directive. This time, if an ORG directive is found, it will directly set the address counter to the given operand. Also, if there are any DC, DS directives found, their objects will be created and set up. They will also go through space calculation process. This process will always return 1 if directive type is DC and if directive type is DS, this process will return number of words used by DS directive.

In this cycle, just as in the first one, on each line loop there will be a lot of error checking to ensure that the program is written correctly (checking operand sizes, syntax…) and if everything is all right the program will proceed to the last stage.

At the end of this stage, program will have list of labels, instructions and directives with their proper addresses assigned to them.

1. Assembling

This is the last step of the whole assembly process. This time, the program won’t loop through each line but will instead loop through Instruction and Directive vectors. The program will fetch the address and calculate the value for each instruction, do the error checking and calculate the relative address if instruction requires relative address.

Similar process goes for directives; however, the program is only considering DC directives here because they are the only ones that contain any data physically. While looping through the instruction and directives, the program will append correct CDM format of the instruction to the output string. When the looping is finished and if all error checks have passed, Assembler will output final CDM format program.

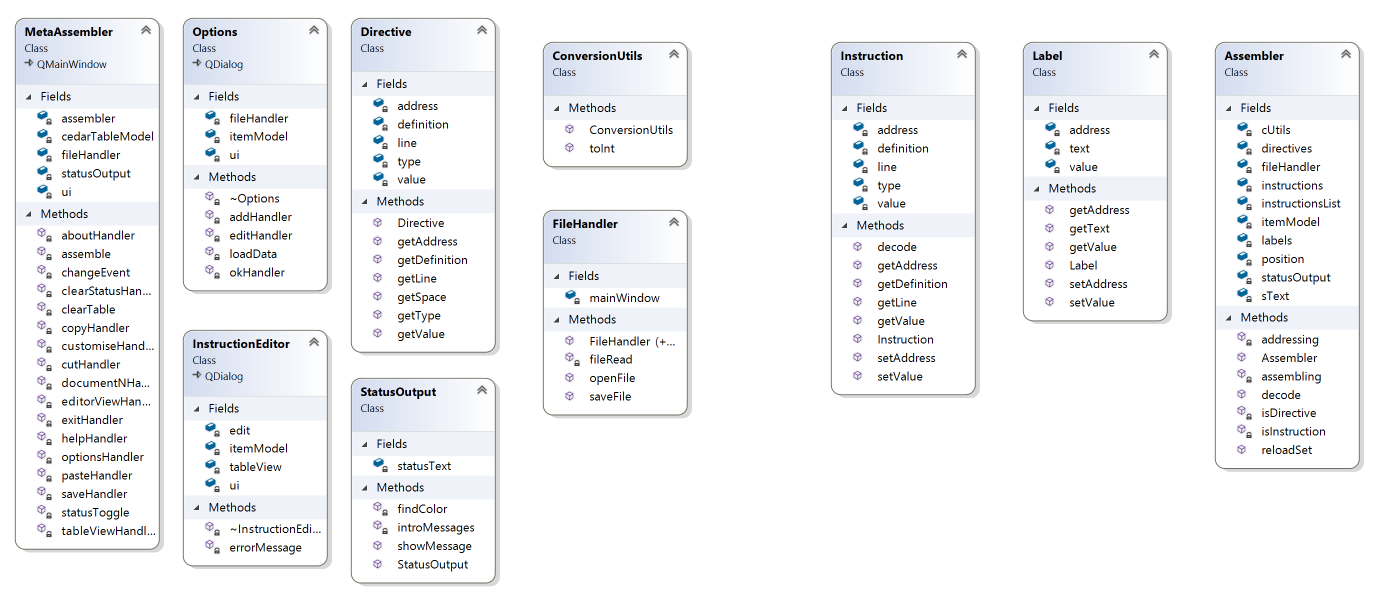


Diagram 1 - Class Diagram

# **Testing**

**Instructions to program testing**

To successfully test and build the program in Visual Studio, it’s important to follow these steps:

1. Install the latest version of Qt with 2017 Visual Studio support
2. Install Visual Studio “Qt Visual Studio Tools” extension (Tools – Extensions and Updates…)
3. Under the QT settings in VS, choose your Qt installation folder as source.
4. Rebuild

*Working .exe with all DLLs has been provided.*

|  |  |
| --- | --- |
| Problem | Outcome |
| Recognizing instructions | The program was able to recognize all instructions in 100% cases if there are no any syntax errors. |
| Syntax Errors | Program was able to detect any syntax errors and highlight the line they were on. |
| Using labels in instructions even if they are defined below the instruction in the file | In the first cycle (decoding), the program creates a table of labels(symbols) and gives the user ability to use the label at any given position in the program. |
| Entering incorrect values | The program detects the corrupt values and stops the assembling process. |
| Overflowing data | The program will cut the overflowing data and warn the user about the potential error. |
| Overflowing address | If the address goes over 4096 (max in CDM), the program will warn the user about the potential error but won’t stop the assembling process. |
| Using labels with labels | Using labels with labels (example. test2 equ test1) is allowed only if labels are properly aligned (test 1 comes before test 2 in the file) |
| Saving and loading | The program should successfully save and load the file if the path is correct and if it has the right permission. However, if there are any errors in loading and saving, the program will warn the user via status view. |
| Converting empty file | The program recognizes that the file is empty and doesn’t proceed with assembling |

Example 1

|  |  |
| --- | --- |
| Input | Output |
| ORG $30  TEST1:  MOVEI 0  BZ TEST2  ORG $50  TEST2:  JSR TEST3  VAR DS 4  JSR TEST4  ORG $10  TEST3:  MOVE $123  MOVEI #0  FOUR DC #4  RTS  ORG $70  TEST4:  FIVE EQU $5  MOVEI FIVE  RTS | Output:  # Instructions  30 : C000  31 : FE1E  50 : 8010  55 : 8070  10 : B123  11 : C000  13 : FFFB  70 : C005  71 : FFFB  # Directives  12 : 0004 |

Results

In this example, the program has successfully built a 100% correct CDM output without any problems. Assembler was tested with multiple different instructions in combination with labels and all directives. The program has also calculated the relative addresses correctly and outputted everything in a proper fixed length hexadecimal format.

Critique

At the very end, the program fulfils all of its objectives and is able to successfully translate Assembly instructions into a proper CDM format. What I would change if I had more time is simplify the error checking part. I’m aware that it looks messy but it’s absolutely essential to be in there. I’d create a separate ErrorChecker class in the future and it would take values on the input, do an error check and return a Boolean on the output. This would make the code much cleaner. I’d also like to eliminate constant conversion between QStrings and string but at this very moment it’s impossible to work with Qt using the simple std:string format. My first attempt at this project was to design it with CLI, but I had to deal with even more conversions (std::string – System::String…) and I decided to switch to Qt. However, I’m satisfied with the project functionality and overall code, use of classes and design (Totaling 3335 lines of code, including design files).