

# PSEUDO - ASSEMBLER INTERPRETER

Author: Dawid Kozykowski

## MANUALS AND DOCUMENTATION

### Table of contents:

- 1. Basic informations
- 2. Manuals
  - 2.1 Structure of language
  - 2.2 Addressing
  - 2.3 Allocating memory commands
  - 2.4 Operating commands
  - 2.5 Terminal window
  - 2.6 Getting started
- 3. Documentation
  - 3.1 Structure of the program
  - 3.2 Mechanic of simulation
  - 3.3 Mechanic of display

## 1. BASIC INFORMATIONS

This program is an interpreter for the pseudo-assembly language designed by prof. Homenda. The program reads list of commands with arguments from a file, simulates them and lets user see changes made by every single command. This project is licensed under MIT License.

## 2. MANUALS

### 2.1 Structure of language

It is assumed, that there are **15 registers** user can use, 2 different types of commands (allocating memory or operating commands) and 4 different program states:

„00” – idle state

„01” – positive state

„10” – negative state

“11” – error.

First block of commands is obligated to be allocating memory type. After first usage of operating command there should **NOT** be a single memory allocation command left.

Every command in this language looks as follows:

“<label> <type> <argument1>,<argument2>”, e.g. “FOUR DC 1,4”

**IMPORTANT:** Labels may only consist of capital letters, digits and underscore (\_)

## 2.2 Addressing

There are few different ways of addressing. These are their interpretations:

<4096> (e.g.: A 1, 4096) – 4 byte cell with address 4096

<4096(12)> (e.g.: A 1, 4096(12)) – 4 byte cell with address (4096 + address stored in 12th register)

<TAB> (e.g.: A 1, TAB) – 4 byte cell with address stored under label “TAB”

<TAB(12)> (e.g.: A 1, TAB(12)) – 4 byte cell with address stored under label “TAB” increased by address stored in 12th register)

## 2.3 Allocating memory commands

Allocating memory commands are placed at the beginning of program. All of them adds some number with or without value to program memory. Labels are necessary. These are all of them with interpretations:

<label> DC INTEGER(<arg1>) - add a single number named <label> with value <arg1>

<label> DC <arg1>\*INTEGER - add an array with <arg1> cells named <label>

<label> DC <arg1>\*INTEGER(<arg2>) – add an array with <arg1> cells named <label>, each cell has value <arg2>

<label> DS INTEGER - add a single number named <label>

<label> DS <arg1>\*INTEGER - add an array with <arg1> cells named <label>

## 2.4 Operating commands

Operating commands are placed after allocating memory ones. They are used to do calculations, comparisons etc. Labels are not necessary. These are all of them with interpretations:

<label> A <arg1>, <arg2> - adds value stored in cell with address <arg2> to register with number <arg1>

<label> AR <arg1>, <arg2> - adds value stored in register with number <arg2> to register with number <arg1>

<label> S <arg1>, <arg2> - subtracts value stored in register with number <arg1> by value stored in cell with address <arg2>

<label> SR <arg1>, <arg2> - subtracts value stored in register with number <arg1> by value stored in register number <arg2>

<label> M <arg1>, <arg2> - multiplies value stored in register with number <arg1> by value stored in cell with address <arg2>

<label> MR <arg1>, <arg2> - multiplies value stored in register with number <arg1> by value stored in register number <arg2>

<label> D <arg1>, <arg2> - divides value stored in register with number <arg1> by value stored in cell with address <arg2>

<label> DR <arg1>, <arg2> - divides value stored in register with number <arg1> by value stored in register number <arg2>

<label> C <arg1>, <arg2> - compares value stored in register with number <arg1> with value stored in cell with address <arg2>, sets state: "00" if they are equal, "01" if value in arg1 is greater than value in arg2, "10" in different case

<label> CR <arg1>, <arg2> - compares value stored in register with number <arg1> with value stored register number <arg2>, sets state: "00" if they are equal, "01" if value in arg1 is greater than value in arg2, "10" in different case

<label> J <arg1> - moves to line with label equal to arg1

<label> JP <arg1> - if program state is equal to "01" moves to line with label equal to arg1

<label> JN <arg1> - if program state is equal to "10" moves to line with label equal to arg1

<label> JZ <arg1>, <arg2> - if program state is equal to "00" moves to line with label equal to arg1

<label> L <arg1>, <arg2> - sets value stored in register with number <arg1> to value stored in cell with address <arg2>

<label> LA <arg1>, <arg2> - sets value stored in register with number <arg1> to arg2

<label> LR <arg1>, <arg2> - sets value stored in register with number arg1 to value stored in register with number arg2

<label> ST <arg1>, <arg2> - sets value stored in cell with address arg2 to value stored in register with number <arg1>

## 2.5 Terminal window

Commands:

	Line nr:	Label:	Type:	Argument1:	Argument2:
current line ->	1	ONE	DC	INTEGER(1)	<NONE>
	2	TAB	DC	100*INTEGER(30)	<NONE>
	3	CZTERY	DC	INTEGER(4)	<NONE>
	4	<NONE>	A	1	CZTERY
	5	<NONE>	S	2	TAB(1)

The green line in the commands section is a line of program that has just been executed.

Memory:

Address:	Variable:	Label:
100	1	ONE

Registers:

Index:	Value:
0:	NONE
1:	4

Red lines in memory or register section shows differences made by current line of code

## 2.6 Getting started

To get started just simply write commands in „commands.txt” and run compiled interpreter.exe

## 3. DOCUMENTATION

### 3.1 Structure of the program

The program consists 4 cooperating files: `interpreter.c`, `core.h`, `GUI.h` and `structures.h`. Each one has its own contribution in final result. Full separation of front-end and back-end is assumed.

`interpreter.c` – main file of the project, responsible for declaring memory and initiating core processes

`core.h` – back-end file, responsible for simulating operations in pseudo-assembler, translating given input into clear orders, operating on memory

`GUI.h` – front-end file, responsible for whole interaction with user, displays output on screen

`structures.h` – declaration of two structures used to store memory and operating commands

### 3.2 Mechanic of simulation

First of all, `core.h` reads input from a file, deletes unnecessary signs (like whitespace characters, commas, etc) and inserts raw commands 'input', which is an array of following structure:

```
struct singleCommand
{
    char label[30];
    char type[5];
    char argument1[30];
    char argument2[30];
};
```

`input.label` stores the given label of command

`input.type` stores type of command (like „DC” or „A”)

`input.argument1` stores first given argument unformatted

`input.argument` stores second given argument without unformatted

If the given command allocates memory, then another cell to 'memory' (which is an array of following structure) is added

```
struct variable
{
    char label[30];
    int firstIndex;
    int lastIndex;
};
```

**memory.label** - stores the given label of variable

**memory.firstIndex** - stores the index of this variable in stack of all variables

**memory.lastIndex** - stores the first index unused by this variable in the stack of variables, for example:

if user adds a **1**-cell-variable, then `memoryStack[ firstIndex ]` = this variable, and `firstIndex + 1 * 4 = lastIndex`

if user adds an array of **10** variables, then `memoryStack[ firstIndex ]` = first number in this array, `memoryStack[ firstIndex + 4 ]` = second number in this array and so on...  
`lastIndex = firstIndex + 10 * 4`

Afterwards `core.h` performs following actions on every command line by line:

- transform arguments into memory addresses
- execute command
- update all arrays
- call GUI to display updated program state on screen

### 3.3 Mechanic of display

**Only once**, at the program start, whole terminal is rendered. To optimise display updating, GUI never renders unnecessary characters - instead of rendering whole terminal afresh, it uses arrays with „**Previous**” suffix in their names. They are used to remember old value of each cell. If the old value does not equal new one, then GUI uses function `moveTo(a, b)`, to move to a specific place in terminal, function `clear()` to remove the outdated informations, function `color(x)` to change the color of font and finally overwrites new data.

Explanation of basic functions:

`moveTo(a, b)` – moves cursor in terminal to a-th line and b-th character

`color(x)` – changes color of font (x: 0-white, 1-green, 2-yellow, 3-red)

`clear()` – clears next 70 characters by overwriting then with spacebars

`rightSide()` - responsible of displaying lines of input

`leftSide()` – responsible of displaying variables stored in registers and in memory