

1. Introduction.
2. General design.
3. Textual processing.
4. Post processing.
5. Compilation.
6. Errors.
7. Function descriptions from headers.

Software design document

University project

Yair zafrany

ASSEMBLER DESCRIPTION DOCUMENT.

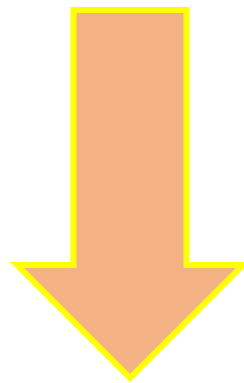
Student: Yair Zafrany.

Introduction:

University project of a basic assembler for an assembly language of a fictional machine, defined in the assignment document provided by the university. The purpose of this document is to describe the inner workings of the assembler and the methodology used in it. **As of currently this document isn't complete.**

General Design.

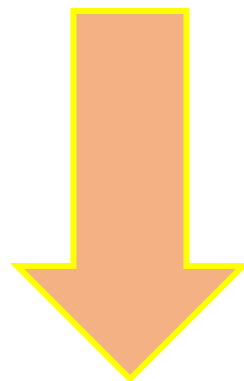
Assembly code



Textual processing and phrasing. Look for syntax errors and multiple definition errors.

Convert the code in to an array of numbers

Array of numbers representing the code.



Check the array representing the code for logical errors and illegal operations. If no errors, create instruction and data segment, entries and externals. And generate files accordingly.

Binary object file.
Externals and
Entries files.

Textual processing.

The code consists of two main things, language defined words (commands, registers, etc.) and user defined words (labels, string, number, etc.)

The first stage will gather all the user defined words, and save them in the memory in the order they are presented in the code. That includes all the labels at the beginning of lines, labels defined in the external specification, and numbers presented as arguments. However the first stage doesn't handle the strings and numbers in the data\string specifications. All the data that the first stage is gathering will be checked on-the-fly for its legality (already defined labels).

The second stage will look for language defined and the saved user defined words in their legal order accordingly to the rules of the language. And generate a two-dimensional array of numbers representing the code. If the code processor expected a command or a label and got something else or it didn't expect anything (end of line) and got something, it will report an error. Additionally, the second stage adds the strings and numbers from the string\number specifications to a number array that later will be converted to the data segment.

Each stage processes a single line at their given order, both stages are dissecting the line into tokens, the dissection is predicated by where the user intended to create separation in the line. The separation symbols are; white spaces, brackets, commas, colons. The dissection will not occur inside a string.

The first stage will look for label at the first token of a line, and numbers and external labels at later tokens. The second stage will compare the tokens to the language defined words and to the gathered user defined words. Depending on what it expects. first token can be a label or a command or a specification, and if for example a token is a command the next tokens are expected to be arguments.

Post processing.

the generated array is represented as following,

each line represented by a single array in the two-dimensional array:

{n1,n2,n3,n4,n5,n6}

n1 – index of a label.

n2 – opcode\specification.

n3 - index of label used as destination.

n4 – first argument.

n5 – second argument.

n6 – line number it was generated from for further reporting.

How indexes are differenced:

Operation codes: 0-15.

Specifications: 20-23.

Labels: 10 to infinity.

Registers: 0-7.

Numbers: -10 to negative infinity.

Nothing: -1.

For example, the following lines:

MAIN:

MOV #3,r2

lea MAIN

END: stop

STRING: .string "abc"

Will be generated into:


{10, -1, -1, -1, -1, 1}

{-1, 0, -1, -10, 2, 2}

{-1, 6, -1, -10, -1, 3}

{11, 15, -1, -1, -1, 4}

{12, 21, -10, -1, -1, 5}



This is an offset to the data array that is returned by the string specification in the second stage of text processing, begins from negative ten, lets the compiler know the "STRING" label is addressed for the data segment with a 0 offset (-15 for example is an offset of 5).

The data and string specifications are generating a one-dimensional array. e.g.

.data 2,3,5

{2,3,5}

The ".data" specification line from the code will return to the command array from which point the numbers were appended, so if it were to be a label before the specification the compiler will know that the reference to that label should be addressing the given position in the data segment.

Compilation.

The first stage before compiling is to check for any logical errors.

For example, using an immediate number as a source destination operand.

If no errors were to be found in this stage or any before it (0 errors), the program will proceed to compiling.

The compiling process generates the instruction segment and assigns the current instruction address to each label by saving it to an array that is corresponded with the order of the defined arrays. If it's a label that's pointing to the data segment an offset will be assigned beginning from negative 10. Since we don't know all the addresses of the different labels. Arguments in the code that represent labels will generate their index in the label array and that index will be added to the instruction segment. After the instruction segment is finished generating we will generate the data segment. After we have the instructions and data ready we will go over the instruction segment replacing label indexes to their corresponding addresses, if it's a negative address we know we need to assign that certain offset to the final instruction counter and we will get the data address we need, this process will also generate the external file data since we are checking for labels as arguments, of course external labels don't have an address in the instruction segment.

Entries will be generated from the label indexes given by the program number array, and their corresponding addresses in the address array generated from the compilation.

Errors.

Errors and warnings will be reported on the fly. Saving only how many times a report occurred. This will serve us to know if any errors occurred at all, and we will know how many occurred at the end of a file processing.

