Assembly Lab:

- Authors:
    - Arash Ghafoori, ghafoa4@mcmaster.ca
    - Ahmed Hamoudi, hamoudia@mcmaster.ca
- Group ID on Avenue: 45
- Gitlab URL: https://gitlab.cas.mcmaster.ca/ghafoa4/l3-assembly

F1:

What are the global variables?

After translating the high-level language into their assembly format, we realize that at the beginning of the PEP9 program, we must initialize variables that will later on be used in the program. These variables that are declared outside of the function in the high level language, and also are initially declared in the assembly language for the memory to allocate space, are considered global variables. In the high level language we can distinguish them by checking if they are declared outside of the function or have the keyword global. In the assembly language PEP9, we should check that if they were declared as variables and if they allocate a space within the memory for themselves.

Why does the translator use NOP1 instructions?

By NOP1 instruction, we are referring that the program is using an unary no operation trap. This instruction mainly does not do anything, and no registers are affected by it and it will continue the execution to the next instruction. However, the translation program has implemented this instruction possibly because it will reserve space in the memory of the code which would prevent any hazards and align the memory. We also realized that by implementing the NOP1 instruction, the program counter and the memory dump were both incremented by one space.

Translator:

Main

The translator file is mainly composed of 3 methods, the main method will interact with the two methods process_cli and process. It will use the process_cli command to receive the input file, and a boolean variable which will indicate if the program has to print the AST of the input file. It will then create a node that splits the source code into tokens based on their grammar. Depending on the print_ast boolean variables it will either print the AST structure of the input file or it will implement the process method.

Process_cli
The process_cli method mainly interacts with the command-line interface, and will return the input file and a boolean variable which will indicate if we want to print the AST of the input file. It will achieve these by first creating a parser object, and adding two arguments to the parser. The first argument corresponds to the filename, and the second argument corresponds to the boolean variable. Then by converting the parser object to a dictionary, we can return the values of the keys with -f and ast-only which will correspond to the filename the program compiles and the boolean variable for the AST respectively.

Process
The process method will initially receive the input file and the node that the AST node that was created in the main method. It will then create a GlobalVariableExtraction object that is an extension of the parent class ast.NodeVisitor. Therefore, this object can pass on the visit method in its parent class which will call a generic visit on any of the methods that was not defined in the GlobalVariableExtraction object. The program will then implement the function StaticMemoryAllocation and will pass on the results from the GlobalVariableExtraction.

Visitors:

The visitors are composed of 2 classes of GlobalVariableExtraction and TopLevelProgram that are both extensions of the class ast.NodeVisitor. Both of these classes can implement this visit method and will avoid calling the generic visit for the visit_classname that are overwritten in the classes.

GlobalVariableExtraction

The first class is the GlobalVariableExtraction which will extract all the left hand side of the global assignments. This class will extend from a parent class of ast.NodeVisitor which is a base class defined in the ast module. This class will overwrite two of the functions from the ast.NodeVisitor class. The first method is the Visit_Assign method which will add the id of each **Assign** statement in the AST node. And the second method is the visit_FunctionDef method, which will just pass because functions are not considered global by definition.

TopLevelProgram

This class will also implement the same methods as the GlobalVariableExtraction class, however with few differences. Initially after passing on the node to the class, the class will define the variables below and will implement some new functions as well:

Self._instructions_: this variable is a set that will record all the instructions, in each of the overwritten visit_classname functions, this variable is updated by the call of the _access_memory and _record_instruction.

Self._record_instructions: This function will pass on the instruction and its addressing mode into the self._instructions set.

Self._should_save: This boolean variable is implemented in the visit_Assign and visit_Call methods which will indicate whether or not the input has to be saved in the memory or not.

Self._current_variable: This variable will constantly change and could be of any type. In the visit_Assign method, it will take the name of the variable for storing it, and in the visit_Call method it is used for inputs.

Self._elem_id: This variable is mainly used for while loops to keep track of the number of iterations that have been completed.

_access_memory: This variable will call the record_instruction method depending on the addressing mode of the instructions, whether it’s instance or not.

_identify: This method is used in the visit_While method to keep track of and update the self._elem_id variable.

_finalize: This method will solely add an end to the self._instruction and will ultimately return the set of instructions.

Generators:

The Generators are composed of 2 classes as well, The StaticMemoryAllocation class and the EntryPoint class. These classes are the generators because they will print all the data that were visited by using the Visitors.

StaticMemoryAllocation
The first class is the StaticMemoryAllocation class that is fairly straightforward. This class will pass on a set of dictionaries that in this program is considered the global variables, and by implementing the generate method it will iterate through all the global variable’s id and will print all the global variables with a .BLOCK 2 operand after each variable.

EntryPoint:
The second class is very similar to the StaticMemoryAllocation class as well. This class is first initialized by passing on the set of instructions that were visited by the TopLevelProgram object. It will then iterate through each element of the set and will print each instruction. In case the instruction also has a label, the function will print the label then the instruction respectively.

Limitations:

However, there still are some limitations within this translator program. First off, all the inputs in this program are assumed to be Decimal and other types of variables will be regarded as a decimal which will pose an error in the assembly language. Secondly, all the variables that are initially defined in the static memory are all considered to be the type .BLOCK which is not the actual case if we consider the type of each variable. Although the .BLOCK type would be more generic, preferably a more accurate representation of the data type would be better. In addition, function calls have not been defined yet, which in this case would lead to an error. The program could have implemented an error message which would indicate that the program cannot be run for function calls then.

F2:

Memory allocation:

According to the explanation mentioned above, in order to change the global variables, we would have to refactor the classes GlobalVaribaleExtraction to change the Block2 format and refactor the TopLevelProgram to save the usage of LDWA and STWA in the instructions.

GlobalVariableExtraction:
After reviewing the AST formats of the code, we realize that all the value attributes of the global variables are an instance of the ast.constant subclass. Therefore, we would have to extract the node.target value of each node and set an appropriate variable type accordingly. We also implemented a dictionary data structure for the data types instead of a set, where the keys would be the name of the variables and the value would be an array with the elements .Word and the value of the variable.

StaticMemoryAllocation:
In this class, we only refactored the way the function staticMemoryAllocation prints the data, where it will first print the variable name, then the variable followed by its value.

TopLevelProgram:
However, since we are allocating the variable names in the beginning of the assembly language instructions, we would have to get rid of the additional LDWA and STWA instructions in the TopLevelProgram class as well. First we commented out the visit_Constant method, which implements the LDWA instruction. Then we would have to adjust the visit_Assign overwritten method, to avoid using the STWA instruction. Since in the GlobalVariableExtraction class we checked if the AST format is an instance of ast.constant, we would have to ensure in the TopLevelProgram we do not repeat the same process as well. In the visit_Assign method condition, the program will only use the STWA, only if the node is not an instance of the ast.constant class. However, in case if any variable is assigned to another constant integer, the function would also have to implement “LDWA integer i”, and “STWA variable, d” respectively. To implements the reassigning instructions, the function will check if the name of the variable

exists in the previous initializations, and if it does it will perform the reassigning instructions of LDWA and STWA respectively.

Constants:

In order to implement the constants, we would have to make similar refactors to the changes mentioned above.

GlobalVariableExtraction:

After implementing the above refactors, and checking if the node is an instance of the ast.constant class, the function will check if the name of the variable is all in upper class and if it includes the "_" sign at the beginning. In case all the conditions are satisfied, the function will assign the ".EQUATE" type of variable for the variable, otherwise it would assign ".WORD" type for the variable.

Symbol Table:

In order to change the variable names, we would have to first replace the name of each of the global variables that were assigned, and readjust all of the global variable names while the top level instructions are being printed.

StaticMemoryAllocation:

In this class, the generate function will first iterate through the set of keys of global variables, and will add an additional element to the array of values for the dictionary. This additional element will indicate the index of the name formate "Var_#" where # is the number. Then for printing the global variables, the function would print the number Var_# value of the dictionary instead of the key value followed by its data type and value.

EntryPoint:

In order to change the name of the global variables in the top level instructions as well, we only needed to interfere with the generation of the commands, and not the visitors which could add additional complication. In this class, we implemented an addition function called translate which will receive an instruction as its attribute and will return a modified instruction. The function will evaluate and check if the variable name matches the name of the variables in the dictionary of the global variables. If the variable exists in the keys, the function will replace the name of that variable with its Var_# format which is saved in its array value. The function will then return the modified instruction in case a variable exists in the command. In each iteration of printing the instructions, the translate function is called and will return the modified instruction in case a variable exists in the instruction.