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.

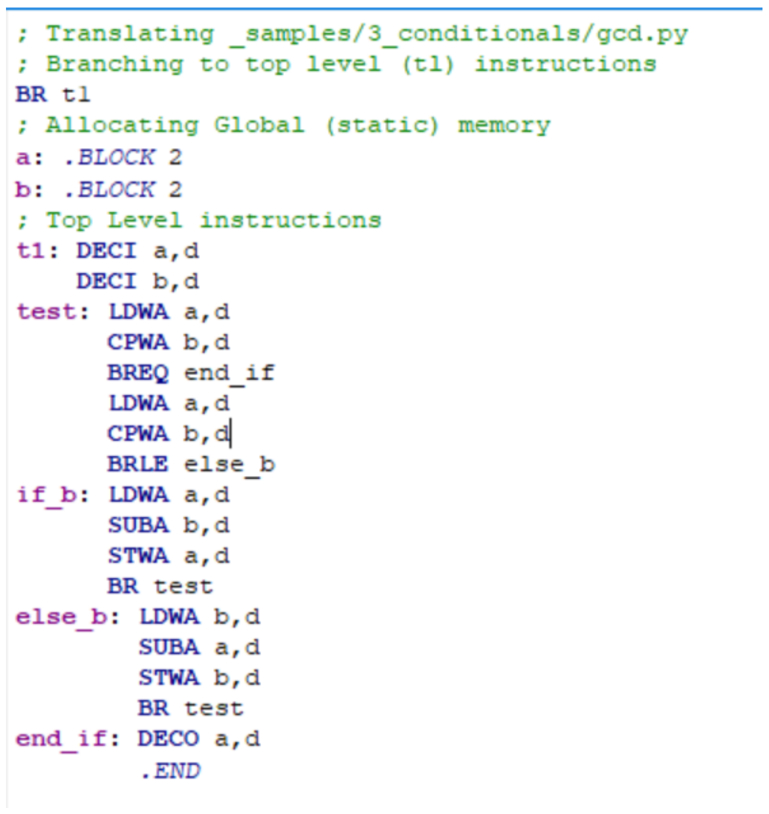
EntryPoint:
In order to change the name of the global variables in the top level instructions as well, we only needed to interfere with the generation of the commands, and not the visitors which could add additional complication. In this class, we implemented an addition function called translate which 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.

Avoiding Overflows:

Normally overflows occur, when the data that is being written overruns the boundary of the buffer and it overwrites the adjacent memory locations. In order to prevent such an anomaly, our program should set a boundary for the input value it's receiving. If the input value exceeds a certain boundary, the program should then return a valueError which would prevent the program to run any further and overwrite the adjacent memory locations.

F3:

Manual translation for _samples/3_conditionals/gcd.py:



```
; Translating _samples/3_conditionals/gcd.py
; Branching to top level (tl) instructions
BR tl
; Allocating Global (static) memory
a: .BLOCK 2
b: .BLOCK 2
; Top Level instructions
tl: DECI a,d
    DECI b,d
test: LDWA a,d
      CPWA b,d
      BREQ end_if
      LDWA a,d
      CPWA b,d
      BRLE else_b
if_b: LDWA a,d
      SUBA b,d
      STWA a,d
      BR test
else_b: LDWA b,d
        SUBA a,d
        STWA b,d
        BR test
end_if: DECO a,d
        .END
```

Automation of Conditional Translation:

Test:

Similar to a while loop, the first part of the translation is the test. So we first test the condition by loading the left hand side of the condition to the accumulator using a LDWA instruction, and then comparing it to the right hand side of the condition using a CPWA instruction. After this we branch only if the evaluated condition is false. This means we branch if the inverted condition is true.

For example for if a <= b, we would first load a into the accumulator and then compare it with b. We would then branch if the inverted condition is true, meaning we would branch if a > b, so our branch instruction would be BRGT (branch if greater than).

If Body:

The contents of the if statement and their corresponding instructions are put directly after the branch instruction. This is because we execute the branch instruction if the inverted condition on the if statement is true, meaning if we want to enter the else clause (if it exists). So if we do not branch to the else clause we perform the instructions of the if statement.

Else Body:

The else body is labeled by the tag end_if_i where i is the id. This is the label that we branch to if the inverted condition is true. This is put after the instructions of the if statement body. Therefore, if we branch we will skip over the if statement instructions and only execute the else instructions.

Exiting If Body:

At the end of the if statement instructions we add a branch instruction (BR) to the label exit_i where i is the id which will be located after all instructions. We do this so that if the if clause is entered it will not continue to execute the else instructions.

Handling Elif:

The AST module does not define a special class for elif statements. The If class has an orelse field which contains the instructions of the ‘else’ statements. In the case where it is meant to act as an else, the field will contain non-if instructions to be executed. In the case where there is an elif statement, the orelse field will contain another If class. So in the case where we have an if and then an elif, the test of the if statement will branch to another if statement (with its own test and so on) rather than a set of non-if instructions. So iteratively go through the elif’s in the case where they exist.

General Structure:

```
test_i: LDWA var_1, d
        CPWA var_2, d
        BR_condition test end_if_i
        ; if statement instructions
        BR exit_i
end_if_i:  NOP1
           ; else instructions, could be another if in the case of elif
exit_i: NOP1
```

Visitors / Generators:

To achieve this change in the code we just had to introduce a visit_If visitor in TopLevelProgram.py. We first perform a load instruction (LDWA) to load the left hand side of the condition and record that instruction. We then compare it to the right hand of the condition using CPWA, and then branch if the inverted condition is true (BR_). We record these instructions using __access_memory() method and the __record instruction() method. To get the instructions of the if statement we visit every node in the node body of the if statement, each will be taken care of with its own specific visitor. The else clause functions the same way, we visit every node in the orelse field each with its own visitor to record its instructions. If the node happens to be another If, then the visit_If function will be called recursively.

No changes were made to the generators as we simply recorded the instructions as usual for the top level program.

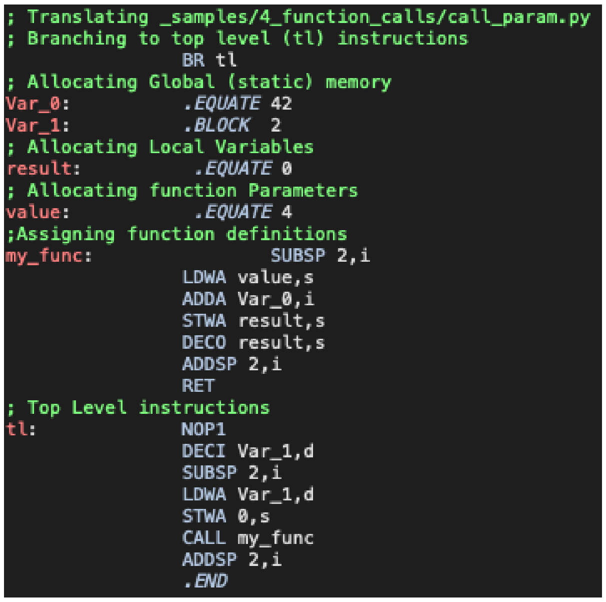
F4:

Complexity ranking:

The first 3 functions call_param, call_return and call_void were the basic functions where it just evaluated whether the parameters and return values work properly. The least complex file after the basic 3 calls was the fibonacci file. The fibonacci file consisted of other extra operations such as a while loop in the function, however the return value and parameter structure was similar to the previous 3. The more complex one after that was the fib-rec, this function recursively called fibonacci. Then the more complex one after that was the factorial, which consisted of 2 functions, where one function was called in one another. And the most complex file was the factorial-rec which consisted of both calling another function in one another, and also recursively calling another function. Therefore the final order would be:

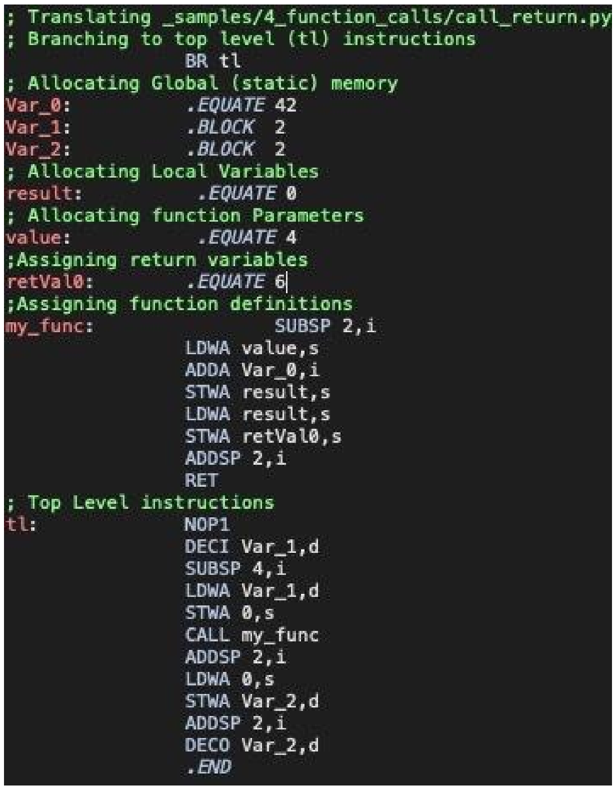
1. Factorial-rec
2. Factorial
3. fib-rec
4. Fibonnaci
5. Call_return
6. Call_param
7. Call_void

Manual Translations:



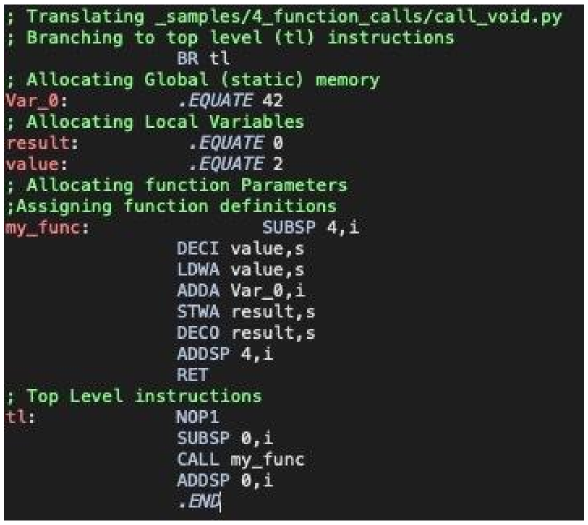
```
; Translating _samples/4_function_calls/call_param.py
; Branching to top level (tl) instructions
                BR tl
; Allocating Global (static) memory
Var_0:          .EQUATE 42
Var_1:          .BLOCK  2
; Allocating Local Variables
result:          .EQUATE 0
; Allocating function Parameters
value:           .EQUATE 4
;Assigning function definitions
my_func:                SUBSP 2,i
                LDWA value,s
                ADDA Var_0,i
                STWA result,s
                DECO result,s
                ADDSP 2,i
                RET
; Top Level instructions
tl:             NOP1
                DECI Var_1,d
                SUBSP 2,i
                LDWA Var_1,d
                STWA 0,s
                CALL my_func
                ADDSP 2,i
                .END
```

Call_param translation



```
; Translating _samples/4_function_calls/call_return.py
; Branching to top level (tl) instructions
                BR tl
; Allocating Global (static) memory
Var_0:          .EQUATE 42
Var_1:          .BLOCK  2
Var_2:          .BLOCK  2
; Allocating Local Variables
result:          .EQUATE 0
; Allocating function Parameters
value:           .EQUATE 4
;Assigning return variables
retVal0:        .EQUATE 6
;Assigning function definitions
my_func:                SUBSP 2,i
                LDWA value,s
                ADDA Var_0,i
                STWA result,s
                LDWA result,s
                STWA retVal0,s
                ADDSP 2,i
                RET
; Top Level instructions
tl:             NOP1
                DECI Var_1,d
                SUBSP 4,i
                LDWA Var_1,d
                STWA 0,s
                CALL my_func
                ADDSP 2,i
                LDWA 0,s
                STWA Var_2,d
                ADDSP 2,i
                DECO Var_2,d
                .END
```

Call_return translation



```
; Translating _samples/4_function_calls/call_void.py
; Branching to top level (tl) instructions
                BR tl
; Allocating Global (static) memory
Var_0:          .EQUATE 42
; Allocating Local Variables
result:          .EQUATE 0
value:           .EQUATE 2
; Allocating function Parameters
;Assigning function definitions
my_func:                SUBSP 4,i
                DECI value,s
                LDWA value,s
                ADDA Var_0,i
                STWA result,s
                DECO result,s
                ADDSP 4,i
                RET
; Top Level instructions
tl:             NOP1
                SUBSP 0,i
                CALL my_func
                ADDSP 0,i
                .END
```

Call_void translation

Translation Process:

The translation for the functions made us change multiple structures which affected both the top level programs and made us implement new classes and methods as well. However in general our program will follow these steps:

1. The program will iterate Through the top level global Variables and will allocate static memory
2. The program will iterate through each function, and each iteration it will:
    a. Allocate a stack memory for the local Variables of the function
    b. Allocate a stack memory for the parameters of the function
    c. Translate the function into assembly language
3. The program will then translate the top level instructions

In order to implement the following steps we introduced 4 new classes and refactored the existing classes. The new classes that we introduced are:

LocalVariablesExtraction:
This file has a very similar structure to the globalVariableExtraction. This class will visit all the assignment statements in the function, and will allocate a size for its stack. It will also iterate through all the arguments in the function, and will record all the parameters with their corresponding size as well. This function mainly overwrites the visit_Assign for the local variables and visit_arguments for the parameters.

LocalsGenerators:
This function will iterate through the local variables and arguments that it receives as an argument. It will then print all the local variables in the format of “**var: .EQUATE #”** where the number corresponds to the size of the stack. It will then start printing the parameters of the function which starts from 4 + maximum # that was achieved in the local Variable generation.

FuncTranslate:
This function has a very similar structure to the TopLevelProgramming, it will overwrite all the **visit_ast.class** methods that are overwritten in the TopLevelProgramming file as well. However, the main differences to the topLevelProgramming is the addressing mode for the data, where in the top level, the addressing mode for the variables is “d” and for the function it is “s”. In addition, the function will also take the local variables as an argument where the top level does not. The FuncTranslate class also overwrites an additional method visit_Return that is not implemented in the TopLevelProgram, this method mainly destroys the stack that is introduced at the beginning of the function and returns the functions. The rest of the overwritten **visit_ast.class** methods all have a similar structure to the topLevel formate.

FuncGenerator:
This function has a similar structure to the EntryPoint file, where it will translate all the global variables in the functions as well. And then depending on whether the function has a return value or not, it will also print the “**retVal#: .EQUATE #**” Where the numbers will correspond to the function index, and the stack size of the return value respectively. It will then iterate through each instruction, and print each of the instructions visited by FuncTranslate.

Overall, all these additional classes were made and all the previous classes were reformatted to follow the following instructions:

1. **For Functions:**
    a. Initialize a stack at the beginning of the function with the **“SUBSP #,i”** where the sizes will correspond to the **(number of the local variables) * 2**
    b. If the function has a return value, load the return value and store it in **retVal#**
    c. Destroy the stack at the end of the function and add the **RET** instruction

2. **For function calls:**
    a. Allocate a function with the “**SUBSP #,i**” instruction where the value will correspond to **(number of local variables*2) + 2(if the function has a return value)**. Load all the variables in the function by starting with the negative of the size of the stacks and incrementing it by 2 every time a variable is loaded**.**
    b. Delete the stack for the local variables with the “**ADDSP #,i**” instruction where the # will correspond to the (**number of local variables * 2**).
    c. If the function has a return value, load the return value by **LDWA 0,i** instruction and store it in the variables, then destroy the remains of the stack.

Stack Overflow:

In most cases, the cause of a stack overflow in the assembly language would be the infinite number of recursions. The excess amount of data will then corrupt both the variables and the corresponding addresses which could then cause an arbitrary value to be outputted for a higher input value.

F5:

F5:

Automation of Arrays:

Initialization:

The first part of automating the arrays was to allocate the memory properly. Array allocations have the form of i * [0] where i is some number representing the size of the array. For initial assignments to an array variable, we check if the assignment has this form. If it does we do a block allocation of 2 * i since we want to allocate enough room for i integers. The instruction would have the format:

```
        Array_name: .BLOCK 2*i
```

This change impacted the visitors in the GlobalVariables.py file, and specifically the visit_Assign() function. We added a check for array initializations by searching for an assignment of the form described above and then recorded the instruction. This had no impact on the generators as the instruction was recorded as usual.

Assigning A Value To An Array Element:

When we assign value to an array element, we essentially first find the value of the assignment. This is done using visitors defined in the previous parts of the lab. The accumulator will have the value of the assignment. However, we cannot simply do a STWA operation. We first have to load the offset value into the index register using LDWX. We then do an ASLX operation to multiply the offset by 2 since each integer takes two bytes. And then finally we do a LDWA operation to the array, except the mode addressing is x, so that we store the value at the right index which is defined in the index register.

This changed the visit_Assign() visitor since we have to add the instructions to load the index into the index register in the case where we assign to an array element. We do this by checking for cases where we assign to arrays.

Example:
code:

```
a[i] = value
```

pep-9:

```
; compute value as usual, result stored in accumulator
LDWX i,d
ASLX
STWA a,x
```

Accessing The Value of Arrays:

In order to access array elements we also have to change our instructions. In order to perform any operations with array values we first have to load the index value to the index register, using LDWX. Then we perform a left shift (ASLX) to multiply that value by 2 as each integer takes 2 bytes. After that we can perform the correct operation, such as adding the value to the accumulator in the case of an addition operation using ADDA and the x addressing mode or subtracting the value from the accumulator in the case of an subtraction operation using SUBA and the x addressing mode.
In order to avoid loading an array value to the accumulator, we first initialize the accumulator value to 0 at the start so that we can directly perform add or subtraction operations.

To allow for these changes, we only had to edit the visit_BinOp() visitor. We check if the one or both of the operands is an array value and then adjust the instructions to first load the offset into the index register. We also change any ADDA or SUBA instructions to address array variables with addressing mode x, to get the correct array element.

Ex.

Code:
```
x = a[i] + 1
```

Pep-9:
```
; initialize accumulator to 0
LDWA 0,i
; load first operand, which is an array value
LDWX i,d
ASLX
ADDA a,x
; Add the second operand since it is a addition operation
ADDA 1,i
; store the value in x
STWA x,d
```

Unbounded Arrays:

One way this can be done is to allocate all unbounded lists to a portion of the stack. Each time the size of the list changes we can copy over all elements of the stack except with extra room for the list that was expanded. For each list we have variables that keep track of their start and end on this “unbounded list stack”.

Self Reflection:

Arash Ghafoori (ghafoa4):

How much did you know about the subject before we started? (backward)
The ast.module was a completely new platform for me and I have never interacted with it before I saw this lab. However, I was familiar with how the assembly language instructions work and what each instruction means and when it needs to be applied.

What did/do you find frustrating about this assignment? (inward)
I found the F4 section of the assignment very frustrating to deal with. It was very hard to understand how the stack allocation normally works and what pattern am I supposed to implement for the instructions to work.

If you were the instructor, what comments would you make about this piece? (outward)
If I were the instructor, I would give better background information for each of the problems. Some of the problems that we faced, we had no clue about the basic instructions which made this assignment very lengthy and time consuming.

What would you change if you had a chance to do this assignment over again? (forward)
I believe time management was again an important factor in this assignment. Also the distribution of work was very challenging for this assignment, as each section was very dependable on the other sections. I believe correctly distributing the assignment would be very beneficial in terms of completing the assignment in time and not encountering additional merging problems.

Ahmed Hamoudi (hamoudia):

How much did you know about the subject before we started? (backward)
I was familiar with the assembly language instructions throughout the previous courses we had in computer architecture. I was also familiar with how methods are overwritten in python which made it easier to understand the visitors. However, I have never used the ast.module, and the stack allocations before.

What did/do you find frustrating about this assignment? (inward)
At the beginning, it was very challenging to understand how the ast.module works and how it's implemented. Also understanding the stack allocation works at the beginning was very time consuming which made me spend hours to understand the manual translations of the programs.

If you were the instructor, what comments would you make about this piece? (outward)
I believe providing resources for the students to understand the basic concepts would be very beneficial for this project. Possibly the majority of the students did not have the background

information for the assembly language instruction, and the additional resources were not sufficient for the students to understand the basic concepts.

What would you change if you had a chance to do this assignment over again? (forward)
I believe how we distributed the assignment highly affected our works. All the sections in the assignments were highly correlated to one another which made it very challenging to merge our work together and ensure that all the programs will run correctly.