Assembly Lab

- Authors:
  - Henushan Balachandran, balachah@mcmaster.ca
  - Isaac Giles, gilesi@mcmaster.ca
  - Tushar Aggarwal, aggarwt@mcmaster.ca
- Group ID on Avenue: 43
- Gitlab URL: https://gitlab.cas.mcmaster.ca/balachah/l3-assembly

F1

1.a) This is the manually translated code for the **simple.py** code

```
    BR program
x:  .BLOCK  2
program: LDWA    3,i
         ADDA    2,i
         STWA    x,d
         DECO    x,d
         LDBA    '\n',i
         STBA    charOut,d
         .END
```

This is the manually translated code for the **add_sub.py** code

```
    BR program
_UNIV:           .WORD 42
variable:        .WORD 3
tempOne:         .WORD 1
value:           .BLOCK 2
result:          .BLOCK 2
program:         DECI value,d
                 LDWA _UNIV,d
                 ADDA value,d
                 SUBA variable,d
                 SUBA tempOne,d
                 STWA result,d
                 DECO result,d
                 .END
```

1.b) A **global variable**, specifically in relation to Assembly level language, is a variable that you can load and store from the registers. Thus at any point in the program, you are able to load these variables and store into them as well. This gives it the name so called name "global variables" due to the fact that they are accessible wherever in the program and at any point, even in function calls. In the RBS (Reb-bellied Program) program, we can deduce a variable to be global if it is declared outside of functions in the program. Simply as long as it's not defined in the functions, it is seen as a global variable. On the contrary, local variables are variables that are declared in the body of functions in RBS. The variable is only 'seen' in the execution of the function hence the name local.

2.d) The translator translates functions not knowing whether the function contains any body or not. To avoid compilation errors, the translator places a **NOP1** command at the beginning of each function to ensure that every function has a body. This is conceptually similar to the pass function in python. For example, if you try to define a class with no body, this will not work because the compiler expects to see a body following the definition of a class. To resolve this issue, you can add a pass function to serve as the body of the class. The NOP1 command serves a similar purpose and has no effect on the output of the code.

3.a)

Visitors: GlobalVariables - is a vistor which extracts information for the global variables. Variables that are defined initially to be used throughout the program. It takes in a library called ast which allows the class to use methods to easily pull out variables, the global variables and the only variables(not functions) and break down the syntax (similar to a AST format).

TopLevelProgram - is a visitor which extracts the necessary Pep9 instructions for a given python line of code. The program is broken down into the basic operations and assignments in our RBS language, and has different visit methods for different type ops and assignments. There are default visit functions, but we overwrite the methods within the TopLevel to meet requirements in our RBS lang.

Generators: EntryPoint - is simply a generator that prints out the Pep9 instructions after all the logic has been done. This simply has the results/instructions from the visits and translation, and then prints it line by line.

StaticMemoryAllocation - generator reserves memory for these global variables to exist. This is done prior to identifying the global variables and beginning to assemble the PEP/9 code for reserving such memory.

3.b) The limitations of the current translation in terms of software engineering is that it is not reusable because it is conditioned towards the language we are using, the RBS language. Therefore this translation is limited to not only python, but a subset of python with specific rules that set further limitations in terms of reusability. Also the integration of AST is very niche and undefined. Some methods of AST are unknown and hard to trace. This leads to another point which is the translator is difficult to trace, especially in terms of visiting AST's.

F2

1.a,b,c) To extract the necessary ast data to achieve better memory allocation, we converted the results set from GlobalVariable.py to a dictionary. This dictionary has variable names as keys, and the values are the values of these variables. We are able to extract these values by writing node.value, which doesn't directly give the value of the variable but gives all of the information about the right hand side of the variable. In StaticMemoryAllocation.py, we look at the name of the variable, and if it begins with an underscore followed by all capital letters, we know that we have a constant, and so we use EQUATE and also the value of the variable by writing self.__global_vars[n].value, where self.__global_vars is a reference to the results dictionary returned from GlobalVariables.py and n is a key in this dictionary. Otherwise, if self.__global_vars[n] is of type ast.Constant then we know that we are dealing with an integer assignment that is not a constant, so we use WORD and extract the value of the variable in the same way as before. Otherwise we just use BLOCK. In the naive implementation, it removed the LDWA and STWA for all assignments that consisted of the integers. Which is not the right approach since only the first assignment need not to be Loaded and Stored but the others needed to be stated in Pep 9 language. This is done by keeping track of all the variables that have been visited and if they already have been visited, STWA and LDWA are needed. But if the value is a constant and its first visit, then STWA and LDWA can be ignored.

Additionally, we tried our best to cover all possible boundary cases while writing the code and handled them by implementing the additional code blocks and conditional statements. Along with checking if the name of the constant starts with "_" followed by all uper case letters, we also checked that the minimum length of the variable name is 2 characters. This will make sure we do not recognise any arbitrary variable as constant in our code (for instance, name of variable as, "_") and we do not use EQUATE for such variables.

Moreover, we know that RBS variable names are not length limited (we can have long variable names in our pytohn code). But Pep/9 symbols are limited to 8 characters (Pep/9 does not accept name of variables/constants whose length is greater than eight characters. It will throw an error and not run the assembly code). Manually translating from an RBS variable name into a Pep/9 symbol is a naive approach - This is time consuming, error prone and it is too hard to keep a track of variable names this way. To solve this problem, we used a dictionary called "symbol_table" in the GlobalVariables.py file. If the RBS variable name is less than 8 characters, we just use the RBS variable name as the Pep/9 symbol (For instance, if the RBS variable name is **"_ABC" or "xyz", we keep their name as "_ABC" or "xyz", respectively** as the Pep/9 symbol and so on). For constants, we renamed them as, "_UNIV" and append a number which is equal to the current length of the dictionary at that point of time (For instance, if the RBS variable name is **"_ABCDEFGHI", we use "_UNIV0"** as the Pep/9 symbol and so on) as the Pep/9 symbol. For variables, we renamed them as, "VAR" and append a number which is equal to the current length of the dictionary at that point of time (For instance, if the RBS variable name is **"pqrstuvwx", we use "VAR1"** as the Pep/9 symbol and so on) as the Pep/9 symbol. We store the new names as values referenced to the original names of the variables as keys in the symbol table. We then store all the original variable names who length is less than 8 characters, as well as, the the new variable names from the symbol table into our another dictionary, "results" as keys along with the node values as values in the dictionary. This way, we can keep track of all the Pep/9 symbols and we do not have to manually translate the RBS variable names into Pep/9 symbols. This is a much better approach than the naive approach. We then pass the dictionary, "results" to StaticMemoryAllocation and TopLevelProgram through the translator. This way, we can use the new Pep/9 symbols in the StaticMemoryAllocation and TopLevelProgram files.

2.) After trial and error, the **maximum N value for fibonnaci is 24** and any N after 24 results in an overflow in assembly language. Most languages do already have a flag or handling errors when overflow occurs in additon or subtraction. In general, some ways to handle these errors could be (1) Detection, which is usually in most languages by overloaded methods that error and catch errors. (2) You can simply avoid it by increase sizes of data types, In java there exists data types as long and double for a higher size/byte data type created specially for overflows. (3) The program takes more steps to add numbers that overflow using techniques like multi-precision arithmetic.

F3

1.a) This is the manual translation of GCD.py

```
    BR program
a:       .BLOCK 2
b:       .BLOCK 2

program: DECI a,d
         DECI b,d
test:    LDWA a,d
         CPWA b,d
         BREQ end_w
         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_w:   DECO a,d
         .END
```

2.) We add a visit_If() function and add a LDWA statement to load the left side of the conditional and we label this with a label of test_conditionID where conditionID is the id of the if statement. Next we add a CPWA statement with an input of comparators[0]. Then we check to see if node.orelse exists, and if it does we add a BR statement to branch to else_conditonID. Whatever condition is stored in node.test.ops[0], we branch on the opposite condition, so for example ast.LT should branch as BRGE and we use a dictionary to store all these opposite actions. If there is no orelse condition, then we simply branch to end_conditionID using the same BR command as described before. Next we label the beginning of our if condition as if_conditionID, where the first instruction that is associated to this label is a NOP1 instruction. This if_conditionID should only run if the condition yields true. Then we iterate through node.body and visit all of the contents. Then we record a BR end_conditionID command such that any else statements are skipped. After this we write the else condition if there exists one, by labelling a NOP1 instruction as else_conditionID and then iterate through the contents of node.orelse and visit all. This will handle all elif statements because all elif statements will be nested inside of this else statement. Finally we label a NOP1 instruction as end_conditionID to close off the if conditions.

F4

1.) Hardest to easiest:

1. fib_rec
2. factorial_rec.py
3. factorial.py
4. fibonnaci.py
5. call_return.py
6. call_param.py
7. call_void.py

We decided to choose the programs with recursive calls the hardest in terms of translation complexity because we feel that the translation for recursive calls would require a lot of changes in terms of visiting and calling. With the multiple recursive calls in fib_rec, stack of local variables will get very hard. We decided factorial_rec is less complexity as opposed to fib_rec because it only has one recursive call whereas fib_rec has two, which indoubtedly increases stack size. Then we chose factorial and fibonnaci after because of their complexity in terms of having a lot of nested statements and while and if conditions. This simply makes the pep9 code much longer and variables are constantly in changed in the stack. Following these, we have the simple functions of call_*.py and void is the easiest in terms of complexity because it doesn't return any value and we havent looked at returns, yet. We also believe call_return is more complex than call_param because once again, it has a return statement where as, call_param just prints the value that was returned earlier.

2.) Manual translation of call_param.py

```
    BR program
_UNIV:   .EQUATE 42
x:       .BLOCK 2


result:  .EQUATE 0
value:   .EQUATE 4

my_func: SUBSP 2,i
         LDWA value,s
         ADDA _UNIV,i
         STWA result,s
         DECO result,s
         ADDSP 2,i
         RET
program: SUBSP 2,i
         DECI x,d
         LDWA x,d
         STWA 0,s
         CAll my_func
         ADDSP 2,i
         .END
```

Manual translation of **call_return**.py

```
 BR program
_UNIV:   .EQUATE 42
x:       .BLOCK 2
result:  .BLOCK 2

result1:  .EQUATE 0
value:   .EQUATE 4
retVal:  .EQUATE 6

my_func: SUBSP 2, i
         LDWA value, s
         ADDA _UNIV, i
         STWA result1, s
         LDWA result1, s
         STWA retVal, s
         ADDSP 2,i
         RET
program: SUBSP 4,i
         DECI x, d
         LDWA x, d
         STWA 0, s
         CALL my_func
         ADDSP 2,i
         LDWA result1, s
         STWA result, d
         ADDSP 2,i
         DECO result, d
         .END
```

Manual translation of **call_void**.py

```
BR program
_UNIV:   .EQUATE 42

result:  .EQUATE 0
value:   .EQUATE 2

my_func: SUBSP 4,i
```

```
         DECI value,s
         LDWA value,s
         ADDA _UNIV,i
         STWA result,s
         DECO result,s
         ADDSP 4,i
         RET
program: CALL my_func
         .END
```

3.) We were not able to get the code working though we believe we are in the right track and got the variable initialization working. After manually translating the code, we found a pattern on how local variables are initialized in assembly. If a function does have a return variable, then we have a condition that EQUATE 2 will not be used because all the translations end up using EQUATE 2 as the return address of the function. Moreover, local variables are defined next in incrementing order of EQUATE (obviously 2 is not used if return does exist) and the EQUATES increase by a factor of 2. AFter this, then the parameter's are then initialized using EQUATE starting off where local variables finished. This pattern was common among all the manually translated assembly files. We tried to code out this pattern using an extra generator which was the LocalVariables visitor that would visit the local functions and return its variables, all renamed. Consequently, we also had a dynamicMemory generator for these variables to be printed out. This impacts the current visit and generation because their limited to only global variables and assignments and conditions. This limitation makes the fact about local variables and conditions more difficult and we would need to extend the current visit and modify the functionality to ensure software engineering principles are met. The assumption call-by-value means that the actual value is passed to the parameter instead of the reference to the variable. Any modification to the parameters does not actually affect the variable. This assumption is helpful as it helps simpify the complexity of passing in parameters and STWA and LDWA these variables from the stack.

4.) We did not capture the case of "stack overflow" in our code nor did we implement the functionality of "stack overflow" mechanishm in our code. Based on our assumptions and the lack of test case to check this condition, we could say that our program will crash and the Pep/9 code will not be able to run if a stack overflow occurs. We will not be able to handle this situation because we do not have a stack overflow mechanism. In such a case, we may have to manually fix the code to make sure that the stack does not overflow.

F5

1.) We can allocate memory for arrays using a BLOCK statement and a value of 2*(size of array). Most other parts are the same, if the variable name is too long, we save it to the symbol table and assign a new variable name to the array, and then the appropriate variable name is saved to the results dictionary, with it's corresponding value. We can access entries in the array by using indexed addressing which corresponds to the 'x' character. This makes it such that we access entries within the array rather than the entire array as if it were a normal value.

2.) For local arrays, there will be some parallels with global arrays but we will have to store these arrays on the stack. We allocate 2*(size of array) bytes of memory on the stack. On the stack, we simply store the first index of the array and then whenever we are dealing with a local variable, we use stack-indexed addressing to access and store elements of the local array. We have the benefit of only having to check whether we are dealing with a local array if we are inside of a function, otherwise we can assume that all arrays are global.

4.) An approach to achieving unbounded arrays would be implementing Linked Lists in some way. Linked lists are very useful and they are dynamic structures which are used in many languages. This can be applied in assembly by always pointing to an address continously obviously until all the addressed are used up but that is very unlikely. There are very useful and they store the addresses and values in a given order which is also another useful aspect to have which is order in an array. Initially they can be set to a size but they can be dynamically changed by taking the "last" element and pointing to a new address.

Self-Reflection Questions

Henushan's Balachandran

I did not know much about the topics regarding this lab before starting this lab. Assembly has never been my strong suit and I always found it very tedious. This could be because I learnt python first but regardless it is not something I completely enjoy. Though I did know about assembly instructions like STWA, LDWA and basic input output operations through previous courses taken here. This helped me build a fundamental understanding of this project before diving in too deep. There are a lot of things frustrating about this assignment. One main thing was that we were coding in assembly. Another aspect is there is a large knowledge gap needed to complete this assignment fully. I believe this type of assignment is a very basic part of programming and to generalize we are creating a compiler for python, rather a subset of python. I do realize its much more complicated but this basic compiler is already confusing enough. I also found having to trace the code back on multiple levels was really annoying. Specifically with generators and visitors, knowing what it does requires you to trace your code back a lot and it stacks a lot which gets very confusing. This was present in about 70% of the code which made the hard part of understanding the code functionality. If I was the instructor, some comments I would make are that this may require too in depth of an understanding for the average class. Though it is a very useful piece of work to get an understanding of low level programming because you have to dive deep into pep9. If given more time, I think this project would be a cool one, given maybe a semester time instead of a few weeks. Similar to course project, or capstone project. If there is something I would change about the assignment would be to reduce the workload on the tasks. I believe that more than half of the tasks in the assignment were VERY difficult and time consuming, It makes to sense to have perhaps one task that is challenging so the assignment isn’t done with ease but making more than half difficult is redundant I believe, especially with the time given. This ensures that we still get a good understanding of the content while still being able to retain knowledge without completely being mentally exhausted. If I had the chance to do this assignment again, I wouldn’t spend as much time understanding certain concepts that were unnecessary. I believe there were some parts where I was overthinking concepts and it just wasted my time and exhausted me. Moving forward, I will remember the experience and lessons learnt from this assignment and continue to exhibit better performance throughout my work.

Isaac Giles How much did you know about the subject before we started?

I had little to no knowledge of assembly before beginning this project. I have only learned a little bit through some of the hardware courses that I took in second year of university but we did not have to write very much actual assembly code at that time. The assembly code that I have written prior to this project was also for a different computer set architecture.

What did/do you find frustrating about this assignment?

I think that the nature of assembly is very rigid and strict. Learning assembly is also very difficult and requires some knowledge of what the computer is doing at the lowest levels. I personally prefer to deal with software on the higher end and I simply don’t have that much interest in hardware so just working with assembly as a whole was honestly quite frustrating for me.

If you were the instructor, what comments would you make about this piece?

This project is meant to teach students how high level code is being translated such that the higher level programs that the majority of developers are using in today's day and age can function. This project stresses the difficulty in enabling all of the powerful features in modern day programming languages through very complicated code translation. I would also hope that the prof would recognize that this project is a massive undertaking and a very difficult task for anyone who does not already have familiarity with assembly and ast.

What would you change if you had a chance to do this assignment over again?

I would do more planning before beginning the project. I feel like as a group, we identified new problems, and for each new problem we added to our solution, and when something didn’t work we changed things, and added new parts. We were always tackling problem after problem, without fully understanding what problem we were trying to solve, and without having a fully fledged plan in place. This all led to some portions of our code becoming very convoluted and hard to follow. Allowing for more time to read, understand, and plan, before writing so much code would have enabled our group to better streamline our workflow.

Tushar Aggarwal How much did you know about the subject before we started? (backward)

Before working on this lab, I had no clue about how a high-level programming language works in context with a low-level assembly language. Before this lab, I had never heard of the Pep/9 assembly language. The only subject I know and understand is python programming language. Also, I did not know that there are subsets of python languages, such as RBS.

What did/do you find frustrating about this assignment? (inward)

It is the nature of the assembly language. The semantics of the assembly language is so strict/rigid that there is no room for mistake. Even if the syntax of the assembly code is correct and should output as expected, the instruction must be accurate to the pin-point level. This lab required a good pre-requisite knowledge about the design patterns (Visitor pattern) and data structures (Abstract Syntax Tree) which was assumed the students would already know. To convert the high-level language into a lower level of abstraction, we had to build the program from ground to bottom, such as not even the basic operations such as “x = 2 + 3” is valid in the assembly language. Moreover, we had to do a lot of research on an individual level to first understand what is involved in this lab, such as going through the Visitor pattern, as well as understanding the structure of an AST. Also, a lot of functions in the provided code were referenced to the existing libraries of python language which we had to go back and forth to understand some of the critical functions. Overall, this is the hardest lab we have done till now and it consumed the maximum time compared to any of the previous labs.

If you were the instructor, what comments would you make about this piece? (outward)

It is a tough piece of work that every student may not be motivated to do. It can be expected that not all the students will be able to understand the whole assignment too. Although, this assignment is good to provide students with an overview of what an assembly language is and how it works with high-level programming language and machine code. It can also be expected that given the difficulty level of the assignment, not all the students will be able to finish all parts of the lab.

What would you change if you had a chance to do this assignment over again? (Forward)

I had issues with the allocation of the group in this lab. We could have started the assignment in due time. But if I consider the time our group dedicated to this lab, I am not sure it would have made any difference. Since I have experience with the assembly language now, I guess it would be better the next time If I had to do it.