Assembly Lab

- Authors:
  - Angela Zeng, zenga8@mcmaster.ca
  - Emma Wigglesworth, wigglee@mcmaster.ca
- Group ID on Avenue: 50
- Gitlab URL: https://gitlab.cas.mcmaster.ca/zenga8/l3-assembly

F1: Global Variables and First Visits

In [ ]:
```
# add_sub.py

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

In [ ]:
```
# simple.py
         BR      program
x:       .BLOCK  2
program: LDWA    3, i
         ADDA    2, i
         STWA    x, d
         DECO    x, d
         .END
```

Explain in natural language what is the definition of a “global variable” in theseprograms.

A global variables are static pieces of data. This means that in no place or time during the execution of the program will the variable change. In order to do this, they are typcically given a specific address in memory. This is different to local variables which are stored on a stack and are accessed through a pointer. Local variables are subject to change in this way. Global variables typically take up more memory than local variables because of these differences.

If you randomly pick on variable in an RBS program, under what condition can you

decide it is a global one or a local one

Because of the difference between how global and local variables are stored, you can decide whether a variable is global based its address (if it is being stored in a static address in memory). The address must remain and be treated as static.

The translator uses NOP1 instructions. Any ideas why?

The NOP instruction is an instruction that essentially does nothing. NOP1 is used in case there is no instruction after a label (in this case, t1). This will ensure the program is assembled correctly since its not guaranteed an instruction will follow after the label.

Look at the code of translator.py
It relies on two visitors and two generators. Explain the role of each element.

In translator.py, we use two visitors and two generators. The visitor traverses the AST recursively to extract information. There are two visitors. The GlobalVariableExtraction class records the variable names (to allocate later in the generator). The TopLevelProgram class gives different assembly instructions depending on what node it visits. Generators are responsible for printing the instructions in PEP/9 assembly. You can also have backend generators for other assembly languages. The EntryPoint class generates the assembly instructions produced by the TopLevelProgram visitor. The StaticMemoryAllocation class reserves the memory for the variables visited by the GlobalVariableExtraction.

Explain the limitations of the current translation code in terms of software engineering.
One limitation of the translation code is that it only works for Pep/9 so its very limited in terms of translatability. To overcome this, you can make a general translator interface so a class implementing that interface can be used to generate platform specific assembly. Another limitation is that because the output to assembly is printed instead of being written to a file, it becomes more difficult to debug the translator because if we were to print something (for debugging purposes), it would get mixed in with the printed output to assembly.

F2: Allocation, Constants, and Symbols

For each improvement, explain in natural language how you extract (visit) the necessary inoformation form the AST, and how you translate (generate) it into Pep/9

Memory Allocation / Constants

To improve memory allocation, we want to implement the canonical way of allcoating memory when the integer value is known where we would use `.WORD n` rather than using `.BLOCK 2` along with `LDWA` and `STWA` in the program.

In order to differentiate the global variables during the memory allocation process, we altered `GlobalVariablesExtractor` to pass type `dict()` instead of a `set()` to the `StaticMemoryAllocation` class. The dictionary is structured with the node ids as the keys (this allows the unique entries property of `set()` to remain) and the item values as either the node's constant value (if the integer value of the variable is known) else a `None` type. `StaticMemoryAllocation` can then allocate either `.WORD n` or `.BLOCK 2` if the current node in `__global_vars` has a value or not.
We only do this when a variable is first encountered so that if the variable is set to a different value later on, it won't change the initial value. If there is a while loop we have a check `num_loops` to see how many while loops the assignment statement is in. If an assignment is in a while loop then we must do a load and store.

Symbol Table

In order to account for the limitation of PEP/9 symbols being limited to 8 characters, we are to account for this problem in our translator. We map every variable name to a corresponding unique identifier (a unique number) in the `TopLevelProgram` class to later use the unique identifier as a label for the assembly translation in the `StaticMemoryAllocation` class. This will ensure that every variable has a unique name so that we will not run into any problems such as the variables 'variable1' and 'variable2' having the same variable name in assembly (which would cause confusion).

Compute "large" Fibonnaci or factorial numbers. Explain how overflows should be handled in a "real" programming language

The 16 bit registers we use can store a decimal value of up to 32767. Past this, the signed 2's complement requires more bits and overflow would occur. This can result in the misrepresentation of values. For example, when attempting to represent 8! (= 40320), the output is misinterpreted as -25216 because of a missing 17th significant bit.

In a "real" programming language, there would be some kind of flag implemented in the system that indicates whether or not overflow has occurred. This could be handled with a single bit (0 for no overflow, 1 for overflow) in a system status register. The system could use this flag to prompt some kind of solution to the overflow problem, or simply use it to signal an error.

F3: Conditionals

In [ ]:
```
# 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
         LDWA    a, d
         CPWA    b, d
         BRLE    else
if:      LDWA    a, d
         SUBA    b, d
         STWA    a, d
         BR      end_if
else:    LDWA    b, d
         SUBA    a, d
         STWA    b, d
         BR      test
end_w:   deco    a, d
         .END
```

To automate the translation of conditionals, we added a `visit_conditionals` function in the `TopLevelsProgram` class. If a conditional statement is recognized, a branch to the else body will be printed followed by the if body (which gets skipped if branched condition) and the else body. Both of which will end with a bramch statement to an `end_else_#` body which will conclude the conditional and return to the main body of the program. Similarly to how while loops are handled, each conditional block will be indentified using a conditional id to track which if/else/end_if blocks the program branches to.

Self-reflection questions