SFWRENG 3XB3
Lab 3 - Assembly
Authors: armanm5, kasturij
GroupID: 32
gitlabURL: https://gitlab.cas.mcmaster.ca/armanm5/l3-assembly

L3 - Assembly Lab (Group 32)

F1

Manually translated code

simple.py:

```
          BR program
 x:       .BLOCK 2
 program: LDWA 3,i
          ADDA 2,i
          STWA x,d
          DECO x,d
          .END
```

add_sub.py:

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

What is a Global Variable?

In essence, a global variable is a variable which can be accessed from at any point in the function. This is different from local variables, which can only be accessed from certain scopes of the program. The easiest way to distinguish a global variable, is to see if it was declared outside of a function.

The Purpose Of NOP1 Instructions

NOP1 instructions are a unary no-operation instruction, meaning they do not do anything and are like assembly's version of python's "pass" statement. Oftentimes, compilers such as the GCC compiler use no-operation instructions to prevent hazards, timing reasons, or to occupy a delay slot [1].

The Role Of The Visitors And Generators

There are two visitors, GlobalVariables and TopLevelProgram. GlobalVariables extracts all the LHS and creates a set of all the global variables (all variables in the top level program). The second visiter, TopLevelProgram, determines the TopLevelProgram

The Current Limitations Of The Translator

Some limitations we noticed with the translator are issues with memory allocations and variable names. The translator always allocates variables with the .BLOCK keyword, but there are certain times where other variable declarations like .EQUATE (global constant) and .WORD (variable with a known integer value) are more appropriate. For example, .EQUATE makes variables immutable in PEP9 and saves 2 operations at runtime, making it more appropriate for global constants. Additionally, while manually translating add_sub.py to PEP9 code, we noticed PEP9 is unable to accommodate variable names longer than 8 characters. Therefore the translator has to account for that when translating variables to the equivalent PEP9 code.

F2

Improvements - Memory Allocation

For this improvement, the global variable visitor was changed to check if the value of the node was a constant type. The output data type was updated from a set to a dictionary, so that values could be associated with the variable ids. So, if the node was assigned to a constant, the visitor would set the value in the dictionary to the value of the constant. Otherwise, the dictionary value would be set to null. Then, in the memory generator, each id in the dictionary is checked. If the value associated with the id isn't null, .WORD *n* is printed after the id, by accessing its associated value *n*. If not, .BLOCK 2 is printed after the id. Furthermore, the top level program was also adjusted with the same check as the visitor to pass unnecessary/redundant LDWA and STWA operations if the value of the node is a constant type.

Improvements - Constants

On top of the adjustment for memory allocation, a second check was added in the memory generator. If the value of the element is not null, and its id begins with an underscore and contains only uppercase characters, then .EQUATE *n* is printed instead. Additionally, the top level program was also adjusted in the __access_memory method, with a new check for global constants. If the passed node met the requirements of being a global variable, then the added instruction includes "[id],i" at the end of the line instead of "[id],d".

Improvements - Symbol Table

We created a symbol table class, which when fed a variable name, maps it to a pep9 friendly variable name. The function first looks to remove vowels, then splices the variable name if needed. In the generators, the symbol table is passed and called upon instead throughout the code body. Since the symbol table is a hashmap, finding the abbreviated variable name is almost instant and barely affects the translator's performance.

Handling Overflows

In "real" programming languages, overflows are often handled via "wrapping". This is when a modulo operator is applied to the result over a power of the radix. For example, two 4-bit numbers 1110 and 0010 are being added, this creates the 5 bit number 10000 but since this can't be stored in a 4-bit system, the result is 0000. This can have severe consequences for the validity of calculations and can be handled by throwing an overflowerror.

F3

Manually translated code (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_l
if:         LDWA a,d
            CPWA b,d
            BRLE else
            LDWA a,d
            SUBA b,d
            STWA a,d
            BR end_if
else:       LDWA b,d
            SUBA a,d
            STWA b,d
end_if:     BR test

end_l:      DECO a,d
            .END
```

Automation of conditional translation

1. Mark the beginning of the if statement with “if_{id}”
2. Check the conditional statement and branch if the condition is false
    a. When we branch, we branch to the next if statement to check its condition
    b. The next if statement exists if the node’s “orelse” array is an if node
    c. If the “orelse” array is non-empty but does not contain an if node, it is an else statement
3. We insert the body of the if statement by running the visit command on the contents of “node.body”
4. After the body, we insert a branch command to pass all other conditional statements
5. After the branch command, We insert the body of the elif statement by running the visit command on the contents of node.orelse
    a. If it doesn't exist, we skip this step
6. After inserting all conditional statements, we insert a reference that is branched to by the conditional statement, after its body is executed

All of this code was inserted inside a function called visit_If inside of TopLevelProgram. No new classes, generators, or visitors were needed for this. This code does not impact the other visitors or generators, as all it does is insert some branch references and branch statements before and after the contents of the node’s body. This means it does not interfere with any other aspect of the code. When testing the newly changed codebase on previous sample files, the results were unchanged while maintaining correctness, as proof of this.

Citations

1. “NOP (code),” Wikipedia, 18-Nov-2022. [Online]. Available: https://en.wikipedia.org/wiki/NOP_(code). [Accessed: 20-Nov-2022].