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. Since we are translating Python code we have to be careful about assignments that occur inside loops and conditionals when compiling the sample code. Luckily, the ast tree provides an easy way of recursively visiting all the nodes.

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]. In our case, we use the NOP1 instructions as placeholders after labels in the translated code. This will be especially helpful when we will begin translating conditionals as instead of having to write extra code to place the first line of the conditional after the label, we can just place an NOP1 instruction.

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 for the extractor 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. While these changes save us unnecessary operations for constant variables, they also enable us to scale our code easier. Changing the output to a dictionary allows us to retain what values each variable equates to and lets other generators/visitors have access to this data without doing redundant visits.

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. It is to be noted that the symbol table is very trivial and possible error prone. As the algorithm to reduce the length of the string is trivial and it is likely that using similar variable names can lead to two or more RBS variables being mapped to the same variable in pep9. For now, we will leave this as is and revisit it when local variables and functions get introduced to our problems.

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. Initially, one challenge we faced while implementing conditionals was what to place beside the labels. We attempted to not print a new line on the label so that the next line of code would be beside it, but in cases where an empty function existed it led to incorrect code translations. We decided to use the empty statement “noop1” beside each label for consistency and to avoid writing a lot of logic for something which adds little benefit to the compiled code.

F4

Complexity of the algorithms ascending order

- call_void.py
- call_param.py
- call_return.py
- fibonnaci.py
- factorial.py
- fib_rec.py
- factorial_rec.py

Call void is the least complex function among the seven. It returns no value and does not require any value to be passed into it. The only difficulty in translation is allocating the memory and calling the function.

Call param is slightly more difficult, as it requires passing input and call return is more difficult because it requires us to return the value computed and then print it.

Fibonacci and factorial are the next functions on the complexity list and factorial is more complex than fibonacci because it calls a function “mult” inside fac, requiring us to allocate more room on the stack mid function and move the stack pointer repeatedly to allocate variables further down the stack

Factorial_rec and fib_rec are very similar in terms of translating complexity. The only reason we ranked factorial_rec as a higher difficulty is because in addition to calling itself repeatedly, it also calls the function “mult”.

call_param.pep

```
         BR program
; memory allocation
x:       .BLOCK 2
result:   .BLOCK 2
_UNIV:    .EQUATE 42
; FUNC
lresult:   .EQUATE 0
value:   .EQUATE 4
my_func: SUBSP 2,i
         LDWA value, s
         ADDA _UNIV, i
         STWA lresult, s
         LDWA lresult ,s
         STWA 6,s
         ADDSP   2,i
         RET


program: DECI x,d
         LDWA x,d
         SUBSP 4,i
         STWA 0, s
         call my_func
         ADDSP 2,i
         LDWA 0,s
         STWA result,d
         ADDSP 2,i
         DECO result,d
         .end
```

call_return.pep

```
         BR program
; memory allocation
x:       .BLOCK 2
result:   .BLOCK 2
_UNIV:    .EQUATE 42
; FUNC
lresult:   .EQUATE 0
value:   .EQUATE 4
my_func: SUBSP 2,i
         LDWA value, s
         ADDA _UNIV, i
         STWA lresult, s
         LDWA lresult ,s
         STWA 6,s
         ADDSP   2,i
         RET


program: DECI x,d
         LDWA x,d
         SUBSP 4,i
         STWA 0, s
         call my_func
         ADDSP 2,i
         LDWA 0,s
         STWA result,d
         ADDSP 2,i
         DECO result,d
         .end
```

call_void.pep

```
          BR program
; memory allocation
_UNIV:    .EQUATE 42

; FUNC
lresult:  .EQUATE 2
lvalue:   .EQUATE 0

my_func: SUBSP 4,i
         DECI lvalue, s
         LDWA lvalue, s
         ADDA _UNIV, i
         STWA lresult, s
         DECO lresult, s
         ADDSP 4,i
         RET

program: call my_func
         .end
```

Overview of How To Translate functions

We knew that to translate the functions appropriately, we would have to make a new visitor and generator. As of now, we have a generator for the entry point of the program as well as a visitor for the top level program. Functions are not top level and we would like to generate the function code before the top level program so a new generator and visitor is required.

We will call on our new visitor before toplevelprogram and entrypoint inside translator.py. This visitor, we will call “functionsVisitor.py” will visit only the functions in the root node and print out the body of the functions and the label for them.
FunctionsVisitor will be similar to toplevel program in that it contains different visitcommands for each type of ast.node, but there are slight differences in loading data to and from variables since most variables will be on the stack. We will also need to create a new visit function for Return nodes, which will load the return value (if it exists) and call the “RET” command. Finally we must update visit_Call to include function calls by using “CALL {function_label}” and allocate room for parameters and the return value.

Stack Overflow in PEP9

In a “real” programming language, stack overflows would throw errors at the user and end the execution of the programs. In our current implementation, we do not respond to a stack overflow, instead the stack pointer resets back to the top without any indication to the user. This method of “ignoring” stack overflows is dangerous because programs which cause stack overflows will continue running and will overwrite unpopped values in the stack and possibly return to incorrect values in the code.

F5

Translation of Global Arrays

For translation of global arrays, the process should be quite simple/straightforward. From the AST tree, since all arrays are of fixed length that are determined at initialization, the node.value.op.right.value will indicate the size of the array. As such, the global declaration of an array will be formatted as follows: [array] .WORD [2 * size(array)], implemented in the global variable extractor and static memory allocation generator. The codebase would include some way to track the names of arrays (i.e. having a self.__array_names variable in the extractor and passing it as an argument to topLevelProgram). Then, when the top level program recognizes that array computation is needed, the systematic process that is described in the lecture slides for advanced assembly programming would be executed, with the relevant instructions for array reading/writing arrays being added to the set of instructions (i.e. LDWX, ASTX, ADDX SUBX, STWX).

Translation of Local Arrays

Similarly to global arrays, the node.value.op.right.value element is accessed whenever the translator encounters a local array. However, instead of allocating memory as a .WORD, the localVariableExtractor would instead allocate memory on the function stack for the array. Tracking names of local arrays is similarly important and would be required for this our implementation, indicating to the translator that the relevant array instructions should be added to the set.

Treating Data Structures of Unbound Length

To handle unbounded arrays, we would have to dynamically allocate array length. So when we initially create an array we would grant it some arbitrary predetermined length (let's say 50) and create a variable to keep track of how many elements are actually inside the array. As the array increases or decreases in size, we allocate more or remove memory to it to make it seem like that array is unbounded. The difficulty when implementing this is deciding when to adjust the memory of the array and by how much as to not impact performance.