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
```

Explain in natural language how you will automate the translation of conditionals, and how it impacts the visit and generation.

To automate the translation of conditionals, we altered visitors by adding 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. There are no changes to generators as there are no changes or additions to memory allocation to icnlude conditionals.

F4: Function Calls

Rank Translation Complexity
From easiest to hardest:

1. call_void (no parameters and no return)
2. call_param (parameters but no return)
3. call_return (both parameters and return)
4. fibonnaci (call_return but with more stuff)
5. factorial (2 functions and 1 function has multiple parameters)
6. fib_rec (fibonacci but recursive)
7. factorial_rec (factorial but recursive)

Provide manual translations for call_param.py, call_return.py and call_void.py

In [ ]:
```
# call_param.py

         BR      program
UNIV:    .WORD   42
value:   .BLOCK  2
result:  .BLOCK  2
func:    SUBSP   2,i
         DECI    value,d
         LDWA    value,d
         ADDA    UNIV,d
         STWA    result,s
         DECO    result,s
```

```
         ADDSP   2,i
         RET
program: CALL    func
         .END
```

In [ ]:
```
# call_return.py

         BR      program
UNIV:    .WORD   42
x:       .BLOCK  2
result:  .EQUATE 0
retval:  .EQUATE 4
func:    SUBSP   2,i
         STWA    result,s
         LDWA    result,s
         ADDA    x,d
         ADDA    UNIV,d
         STWA    result,s
         LDWA    result,s
         STWA    retval,s
         ADDSP   2,i
         RET
program: DECI    x,d
         SUBSP   2,i
         CALL    func
         LDWA    0,s
         STWA    result,d
         ADDSP   2,i
         DECO    result,d
         .END
```

In [ ]:
```
# call_void.py

  BR      program
UNIV:    .WORD   42
value:   .BLOCK  2
result:  .BLOCK  2
func:    SUBSP   2,i
         DECI    value,d
         LDWA    value,d
         ADDA    UNIV,d
         STWA    result,s
         DECO    result,s
         ADDSP   2,i
         RET
program: CALL    func
         .END
```

Explain in natural language how you will automate the translation of function calls, and how it impacts the visit and generation. Emphasize how the “call-by-value” assumption helps here

Parameters: Stored between return address and return value of the stack frame.

Returns: Return value is at the bottom of the stack frame.

Local variables: We must know how many local variables there are to set the offset of the stack pointer for the stack (e.g, if there're 3 local variables, subtract 6 bytes from the stack pointer).

Recursion: If setup correctly, it will make another stackframe on top.

When visiting, it must know the number of local variables to determine how big the stack frame should be. This applies to parameters too. When generating, the bytes determined when visiting is used to generate the subsp/addsp instructions required to allocate the variables on the stack.

Call-by-value assumption helps us so that we don't have to use pointers. We make a copy of the value on the stack so pointers aren't required (pointers are required for call-by-reference).

Code this new feature. You are free to modify the given code structure, e.g., add new generators, visitors, introduce new classes

We were unable to implement the feature into code but what we would've done is:

Returns: Before calling a function, record the instruction for subsp 2 bytes to make space on the stack for the return value. After calling the function, the value is taken and put into the global variable it is assigned.

Parameters: For each parameter, we subtract 2 bytes from the stack pointer before calling to make space for the parameters of the function. After a call, we add back the same number of bytes to the stack.

Local Variables: write a function to count the number of local variables in each function by recursively traversing the AST and keeping a dictionary of unique variable names. For each local variable we subsp when entering the function and then when exiting the function, we addsp the local variable, before returning.

We do not provide any “stack overflow” mechanism. What will happen if such a situation happen in a program
If we do not provide a stack overflow mechanism, then a program like Pep/9 will detect overflow is happening and terminate the program. Other programs like Java will throw an exception. If run in C, then it will crash due to accessing protected memory.

F5: Arrays

Explain in natural language how you will automate the translation of global arrays, and how it impacts the visit and generation.
Explain in natural language how you will automate the translation of local arrays, and how it impacts the visit and generation

We need to do two things for automating the translation of arrays.

Allocating: For global allocation, we would use the .Block directive to generate the size for the array. For local allocation, we take the number of bytes and subsp into the stack frame for our local variables. This impacts generation (and visiting) since we have to know the array size to make the block that size.

Indexing: When indexing into an array, there are 3 steps:

1. Computing the value of the subscript and load it into the X register.
2. Shift the value in the x register left by 1 (multiplying by 2 to get the number of bytes to offset by)
3. Access the memory using the indexed/stack-indexed addressing mode.

This impacts visiting since we're implementing functionality to visit the subscripts. This impacts generation because of the different addressing mode we're using to visit the array. When visiting an array locally, we use sx instead of x (which is used for global arrays).

Code this new feature. You are free to modify the given code structure, e.g., add new generators, visitors, introduce new classes

We were unable to implement the feature into code but what we would've done is:

For indexing, we would change the `__access_memory` method in `TopLevelProgram` by adding an if statement to see if the variable we're looking at is an array. If so, we use the X addressing mode (and SX addressing mode if the array is in a function). Visiting the subscript, we add the instructions LDWX (load into X) and ASLX (shift left by 1).

For allocation, we visit binopp that multiply an array with a constant. The constant becomes the size of the array. We would then add into the `LocalMemoryAllocation` class where we would, depending on the size of the array, implement .Block of the number of bytes multiplied by the size of the array.

The real python language handles lists/arrays in an non bounded way. Without doing it, can you envision how such unbounded data structure can be managed on a virtual machine like Pep/9

We allocate lists/arrays to the heap. This is so we can resize the lists/arrays without worrying about overriding other data in the program. If the lists/arrays on the heap runs into any other data on the heap, the lists/arrays memory's is then copied to another location on the heap. We access the memory on the heap by using a pointer stored on the stack. With the pointer, we also store 2 integers representing the size and capacity of the array. We access the heap using the indirect (n) and stack-relative deferred (sf) addressing modes.

Self-reflection questions

Emma

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

I knew some basic assembly knowledge from taking 2DA4 last year. We learned the use of some key words using ARM assembly code. I already had some understanding of how adding, subtracting and multiplying works in assembly. I had never used pep9 or even run programs in assembly code in general.

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

I found it frustrating to recall certain fundamental aspects of low-level programming needed to fully understand key parts of the lab. For example, it was difficult for me to understand when you should use words instead of blocks for certain memory allocations. I was most frustrated during the translations of function calls because I found the use of the stack between the different parts of the program hard to understand.

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

If I were the instructor, I would commend us on finishing the work we were able to, and for the implementations we could not complete but still reported on. I would encourage next steps to be to ask for more assistance earlier on in the lab to we can get the knowledge we need to be able to complete the lab sooner.

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

If I were to do this assignment again, I would dedicate more time to more resources such as help from the TAs or instructor or doing some more research on techniques/structures to use in order to complete the implementations for function calls and arrays.

Angela

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

In terms of assembly, I learned the basics of it from 2DA4. We learned about ARM and used programs like Quartus to run it. However, there was a lot of basic concepts I forgot about and had to refresh myself on them through Google and previous textbooks. I did not recall learning

much of the Visitor design pattern so I also had to research it. I was familiar with trees in 2C03 so that part of the lab I had some understanding of when looking at the code.

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

Relearning assembly was a big part that frustrated me as it took a lot of my lab time to understand it (e.g., the addressing modes, the formatting of assembly code, ARM vs. PEP/9, etc.). Also, while wanting to improve on this, I found it difficult to work on this lab incrementally between the 4-week time given to us because of other commitments from other classes.

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

I would appreciate the organization of the report as well as the code. Though, for the code, I would point out that it could always continued to be better organized and also that some parts of implementation (implementation code from F4 and F5 is incomplete) and would look at the report for why this is the case, encouraging to try implementing even if it is incorrect.

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

If I had a chance to redo this assignnent, I would like to take more time to relearn assembly and ask the TA for clarification on any tasks I was unsure about. I would also try to work on the lab incrementally, as purposed by the lab since we had 4 weeks to work on it. Also, I would, even if incorrect, try to implements F4 and F5 and then try to get help about why my code is not working and receive feedback on it from the TA/professor/online.