

# 3XB3 L3 - Assembly Lab

Luigi Quattrocioni (quattro1@mcmaster.ca)

Dennis Fong (fongd1@mcmaster.ca)

Avenue Group: 42

December 8, 2022

## F<sub>1</sub>: Global Variables and First Visits

### Manual Translation

The following is a translation of `simple.py` into Pep/9.

```
1      BR      main
2  x:    .BLOCK 2
3  main: LDWA 3, i
4        ADDA 2, i
5        STWA x, d
6        DECO x, d
7        .END
```

The following is a translation of `add_sub.py` into Pep/9. Note that there are multiple instances of `STWA` immediately followed by `LDWA`. These pairs are redundant and can be removed.

```
1      BR      main
2  _UNIV: .WORD 42
3  variable: .WORD 3
4  value:   .BLOCK 2
5  result:  .BLOCK 2
6  main: DECI value, d
7        LDWA value, d
8        ADDA _UNIV, d
9        STWA result, d
10       LDWA result, d
11       SUBA variable, d
12       STWA result, d
13       LDWA result, d
14       SUBA 1, i
15       STWA result, d
16       DECO result, d
17       .END
```

### Global vs Local variables

In python, global variables are any variables that are declared at the top level of the program (not within any function). These variables are typically referred to as "static" variables, and they are stored within the data or bss sections of the binary. Local variables in python exist at the function scope level. These variables are stored on the call stack, and they exist until the function containing them returns and the scope is destroyed.

## NOP1 Instructions

The translator emits `NOP1` instructions after the entry point label for seemingly no reason. However, this instruction is necessary to avoid syntax errors due to how Pep/9 works. Every label must have an instruction on the same line, otherwise the assembler gives the error `ERROR: Must have mnemonic or dot command after symbol definition..` Since we can't guarantee that an instruction will follow the label (the source file could be empty), we emit a dummy instruction regardless.

## Explanation of Visitors and Generators

Visitors and Generators are two concepts used in compilers after the AST is parsed. Visitors (from the Visitor design pattern) are commonly used in traversals of recursive structures (such as trees) in which the behaviour of the visitation is polymorphic and independent of the structure itself. Most compilers (notably clang) use the visitor pattern to traverse their AST. Each node in the tree is a visitor which is responsible for walking the tree and visiting its children. After the visitors are used to traverse the syntax tree and accumulate the instructions that should be emitted, the generators are used to emit said instructions. Generators are useful because if implemented correctly, cross platform support can easily be achieved. In our case, the bytecode created by our visitors are already basically in Pep/9 format, so the generators emit them verbatim.

## Limitations of Current Translation

Currently the translator is very simple, leading to a couple common software engineering pitfalls. Firstly, the code is not very scalable. Everything is dumped to stdout rather than a specific file descriptor, the visitors violate the single responsibility principle by directly interpolating the string instructions to be emitted, and there is no room for adding optimization passes without violating the open-closed principle. Secondly (and probably less importantly), the code is emitted in a very naive way. The `NOP1` instruction after the entry label, constant still stored in memory (and are not actually constant), and the translation is very literal and error prone (for instance, the length of labels is not constrained).

## F<sub>2</sub>: Allocation, Constants, and Symbols

### Improvements

The first improvement of initializing global static variables with their initial value (if the expression can be evaluated at compile time) has us change allocating with `.BLOCK` to `.WORD` in order to remove redundant load and store instructions. We check if we can perform this upon every visitation of any assignment node. To determine if a variable can be initialized to a constant value, the expression must only contain constant values or other variables that can be evaluated. This is called constant propagation, and we try to do this for every assignment node that is visited at the top level of the program. If successful, we can statically initialize the memory pointer to by the variable to be the resulting value. Specifically, we

emit `.WORD x` where `x` is the result of the expression, rather than `.BLOCK 2` wherever this optimization succeeds.

The next improvement changes constant variables to be textually substituted (basically macros), rather than store them in memory. It makes sense to not allocate space for these variables only if we disallow modifying them. Similar to the previous improvement, this also relies on constant propagation. If we ever visit an assignment node in which the identifier node on the left of the equal sign starts with an underscore and is capitalized, then we can simplify the generated code by hardcoding the evaluated expression as the equated value. In the assembly we emit a `.EQUATE` instead of a `.WORD` or `.BLOCK` directive wherever this applies. Additionally, the addressing mode must become `'i'` rather than `'d'`, since the label now refers to an immediate value and not an address.

The last improvement fixes the susceptibility of labels to clobbering each-other if any share prefixes that are at least 8 characters long. Instead of naively naming each label based on their corresponding variable or other naming scheme, we can map every variable name to a unique alphanumeric permutation (using a symbol table). In this way, we have at least  $26^8$  possible label names. Whenever code that uses a variable is generated, the symbol table maps from the variable's identifier to the label used in the generated code.

## Integer Overflow

When attempting to store large integers in most programming languages, we must be mindful of the width of the integer type used. In Pep/9 For example, the word size is only 2 bytes (16 bits), which means that we can only store integers between -32768 ( $-2^{15}$ ) and 32767 ( $2^{15} - 1$ ) inclusively (we represent signed integers as two's complement). Integer overflow (or underflow) occurs when a computation would result in a number too large or small to fit within the integer width. For instance our Pep/9 program cannot compute factorials larger than 7 (5040), since the result of 8 factorial (40320) would overflow a 16-bit integer. In other "real" programming languages, integer overflow is handled in various different ways. The most natural way is to take the result modulo  $2^n$  (equivalently, the result bitwise ANDed with  $2^n - 1$ ) where  $n$  is the number of bits in the integer. This is because at a hardware level integers are (usually) represented as two's complement, and overflowing the width simply means discarding the overflowed bits. Other languages either promote the data type to one with a larger width, or throw exceptions.

## F<sub>3</sub>: Conditionals

### Manual Translation

The following is a manual translation of `gcd.py`. Due to the nature of manual translation, some optimizations have been implemented such as removing redundant labels and jumps, not loading values if they are already in the register, and reusing the result of a single comparison instruction for multiple conditions.

```
1      BR      main
2  a:    .BLOCK 2
3  b:    .BLOCK 2
4  main: DECI a,d
5        DECI b,d
6  while:LDWA a,d
7        CPWA b,d
8        BREQ done
9        BRLE else
10       SUBA b,d
11       STWA a,d
12       BR while
13  else: LDWA b,d
14       SUBA a,d
15       STWA b,d
16       BR while
17  done: DECO a,d
18       .END
```

## Automation of Conditional Translation

Implementing conditionals is very similar to implementing loops, which conveniently is already done for us. To implement all of these conditionals, all we need to do is implement a visitor for `if` statements (`else` statements don't have their own visitor since they are contained in the `if`). In terms of code generation, each `if/else` block adds two labels and two branches so that the control flow branches into two paths, then joins back into one. The translation will always look like the following:

```
1      ; Evaluate condition (usually CPWA) and branch if false
2      ; if a == b:
3      LDWA a,d
4      CPWA b,d
5      BRNE else
6      ; true branch
7      BR fi
8  else: NOP1
9      ; false branch
10  fi:  NOP1 ; get here by jumping (if) or falling through (else)
```

Of course, the `NOP1` instructions are included for simplicity, but they could be omitted if optimized. Luckily, an `elif` statements is equivalent to an `else` statement with the `if` statement inside of it. For this reason, the python `ast` module doesn't even recognize `elif`s, which means we don't even have to consider this case.