

Final Work - Pilot

Computer Architecture

Gonzalo Turconi - T-2820/7
Ignacio Cattoni - C-6415/7
Maximiliano Redigonda - R-4079/7

January 8th, 2018

Introduction

The aim of this draft is to develop a compiler from the Pilot language to the ARM assembler language. **Pilot** is a simple language; here is provided an accurate description of this language, based on the given model.

Every sentence on Pilot is written on a single line. Each line begins with a letter or a symbol that designates the operation to be executed, or with the name of a variable to assign its value.

Variables are alphanumeric strings that start with a lower-case letter, and have to be followed by a label number¹. Thus:

- a0, z24, x231, p2147483647, are valid names, whereas:
- a-1, b, romeosantos, 23z, w2147483648, r00, are not valid names.

The supported operations are the following:

- Assign: it begins with a variable's name and is followed by an expression. Its effect is to assign the expression's value to the variable.
- Input: it starts with an R, followed by the name of a variable. Its effect is to read an integer from the standard input, and assign it to the variable.
- Output: it begins with an O, followed by an expression. Its outcome is the expression's value printed on the screen.
- End: it consists of an E, which means the end of the program.
- Jump: it is first represented by a G, followed by a label number. This sentence modifies the control flow, and resumes the execution on the label given by the number.

¹A label number is a non-negative integer number, represented by a signed 32-bits integer (i.e. any integer between 0 and $2^{31} - 1$, without leading zeros)

- Label: it consists of an L and a label number. This defines the label characterized by the specified integer.
- Conditional jump: it is represented by a letter I, then it is followed by an expression, and ends with a label number. Its effect is to jump to the corresponding label number if the value of the expression is not zero.
- Comment: it is represented by a symbol #, and it has the effect to tell the compiler to ignore everything until the end of line.
- Function definition: it is represented by a F and it is followed by the name of the function.
- Call a function: it is represented by a C, and it is followed by the name of a function, it has the effect to execute the function.
- Function return: it is represented by a word RET, followed by the name of the function from which to return.

A component is a constant or a variable. An expression consists of a single component, or an operator followed by one or two components (depending on the operation). Blank lines are not allowed: the compiler interprets them as the end of the program.

Process

To complete this project:

1. A precise definition of the Pilot language was derived.
2. A research on how to structure and split the program in various files was carried out.
3. A research on tools and methods to document the files was carried out.
4. A reader and classifier of instructions was developed (`instruction-decoder.c`).
5. A data structure to map variable names into memory addresses was designed (`storage.c`).
6. A system to read and evaluate expressions was created (`expressions.c`).
7. A system to print useful data during the development process was created (`debug.c`).
8. A system in charge of writing the resulting ARM program was developed (`writer.c`).
9. More than 20 Pilot programs were created to test the compiler (`tests/*`).

Note

From the beginning, the objective of this project has been to develop a compiler that, given a valid Pilot code, it generates a valid ARM assembly code that represents it. Due to this fact, if the program receives an invalid Pilot code, the compiler's behavior is undefined.

Problems

During the development process several challenges arose. Some of them are showed next.

How to store the variables' values?

It was researched the possibility of storing the variables' values into registers. This would be efficient unless the number of variables was big enough to turn this operation tedious and chaotic. In this case, it would be needed some extra storage.

It was also considered the usage of the stack, which meant assigning an offset from the top of the stack to every variable. This option also has limitations when the number of variables is big. Besides, it is likely that the variables were not the only values stored in the stack.

Finally, an array with the minimum amount of memory for storing the value of every variable was determined as the best option. This method is easy and handy, a position in the array is assigned for each variable. Then, the array is only composed of variables, and the memory is used optimally.

Once an expression is stored, how to obtain its value and how to store it?

Since an expression has at most 2 components, it was developed an expression-evaluator that uses the registers `r2` and `r3` (this can be configured) to store auxiliary values, and saves the result in the register `r1`.

It was decided to store the result in the register `r1` since it is convenient for future calls to the `printf` function, avoiding an unnecessary `mov` instruction.

¿How to support the integer division operator? (/)

One of the drawbacks arose when adding the integer division operator. The ARM set of instructions contains `sdiv` and `udiv`: signed and unsigned division, respectively.

When executing these instructions, the following error message was displayed: «**Error: selected processor does not support 'sdiv r1,r2,r3' in ARM mode**».

Said instructions were the only ones with which the compiler showed such error. After a careful research it was observed that integer division operations have been added since ARMv8, and that all emulators run ARMv6 by default.

This issue was resolved, then, by adding `-march=armv8-a` as an option in the compilation process, which tells the compiler to run ARMv8 making division operators available.

Next Steps

The compiler supports all additional features suggested. It also supports comments, and it has a documentation made with `Doxygen`. Besides from these extensions, the project can continue by:

- Adding more tests

A compiler is a particularly difficult program to test. Adding more tests is one of the best options to discover flaws and make the compiler more robust. Future changes will then be less likely to produce problems that go unnoticed.

- Adding a `for` to Pilot

The current code allows the integration of new instructions with relative ease. Because of this, implementing a `for` is a good idea to continue this project.

A possible syntax can be `FOR var from to`, where `var` is the iterator variable, and `from` and `to` are components. To end the `for`, `END FOR` could be used.

Given that they could be nested, and that functions are already implemented in Pilot, it might be necessary to change the way the compiler stores variables in the generated ARM program to allow this feature.

- Determine if the entered program is a valid Pilot program

It was agreed that this feature is not critical, since it doesn't affect the correctness of the compiler. However, it would make developer's life easier.

To do this, it is require to at least:

1. Save the labels and functions defined so far, to verify that there were no two definitions of the same label or function.
2. Verify the number of words of each instruction, depending on the type of instruction.
3. Avoid the declaration of a function inside another.

- Optimize the translation

When inspecting an ARM code generated by the compiler, it is clear that the efficiency overhead introduced by the compiler is high.

By maintaining the values stored in the registers, and using more of them, it is possible to prevent the compiler from using redundant operations, making the generated program more efficient.

However, conditional jumps, labels and functions can make the exact values of the registers difficult to track.

Results

More than 20 programs in Pilot have been created to test the correctness of the compiler. Among the most prominent Pilot programs created are:

1. `test7.txt`: Calculates the n-th Fibonacci number.
2. `test8.txt`: Reverses the digits of a number.
3. `test16.txt`: Prints all prime numbers in a given range.

4. `test17.txt`: Multiplies two number using the russian peasant algorithm.
5. `test19.txt`: Implements a 'guess my number' game.