

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 to 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 to test the compiler were created (`tests/*`).

Nota

Desde un principio, el objetivo del proyecto ha sido desarrollar un compilador que dado un código válido en Pilot, genere un código válido en lenguaje ensamblador ARM que lo represente. Como consecuencia de esto, si se recibe un código no válido en Pilot, el comportamiento del compilador es indefinido.

Problemas

Durante el proceso de desarrollo, han surgido numerosos desafíos. Algunos de ellos se listan a continuación.

¿Cómo almacenar los valores de las variables?

Se investigó la posibilidad de guardar los valores de las variables en los registros, esto resultaría eficiente, pero si la cantidad de variables a manipular crece lo suficiente, esta opción se torna tediosa y caótica, ya que se requeriría de almacenamiento extra.

También se evaluó utilizar la pila, y asignarle a cada variable un offset desde el tope de la pila. Esta opción también tiene limitaciones cuando el número de variables es elevado, además de que es probable que no sean sólo variables los valores que se guarden en la pila.

Finalmente, se determinó que declarar un arreglo con la cantidad mínima requerida de memoria para guardar el valor de todas las variables era la mejor opción. Este método resulta fácil y cómodo de manejar, sólo asignamos a cada variable una posición en el arreglo, en el cual sabemos que sólo va a haber variables guardadas, y además resulta óptimo en el uso de la memoria.

Una vez que se tiene guardada una expresión, ¿cómo obtener su valor? ¿dónde guardarlo?

Debido a que una expresión posee a lo más 2 componentes, se realizó un evaluador de expresiones, que utiliza los registros `r2` y `r3` (esto es configurable) para almacenar valores auxiliares y guarda el resultado en el registro `r1`.

Se determinó guardar el valor en el registro `r1` ya que resulta conveniente para las llamadas posteriores a `printf`, ahorrando una instrucción `mov`.

¿Cómo soportar la operación de división entera? (/)

Uno de los inconvenientes se presentó al agregar el cociente a la lista de operadores. El conjunto de instrucciones de ARM contiene a `sdiv` y `udiv`, división con signo y sin signo, respectivamente.

Al ejecutar estas instrucciones se ha obtenido, en ambos casos, el siguiente error: «**Error**: selected processor does not support ‘sdiv r1,r2,r3’ in ARM mode» con la instrucción correspondiente.

Dichas instrucciones fueron las únicas con las que el compilador arrojó tal error. Luego de consultar foros informáticos y manuales de ARM en busca de alternativas, se pudo observar

que las operaciones de cociente han sido agregadas en ARMv8, y todos los emuladores de dicha arquitectura ejecutan, por defecto, ARMv6.

Esto se pudo solucionar agregando `-march=armv8-a` como opción en la compilación, dando por finalizado el problema.

¿Cómo seguir?

El compilador soporta todas las características adicionales sugeridas en las consignas del trabajo. También soporta comentarios, y posee una documentación realizada con *Doxygen*. Además de estas extensiones, el proyecto puede continuar por medio de:

- Agregar más tests

Un compilador es un programa particularmente difícil de probar. Agregar más tests es una de las mejores opciones para dejar en evidencia fallas, y volver el compilador más robusto a cambios que puedan provocar problemas en el futuro.

- Agregarle un `for` a Pilot

El código actualmente permite la integración de nuevas instrucciones con relativa facilidad. Debido a esto, implementar un `for` es una buena idea para continuar el proyecto.

Una posible sintaxis puede ser `FOR var comp1 comp2` donde `var` es la variable que itera, mientras que `comp1` y `comp2` son componentes. Para terminarlo, se podría usar `END FOR`.

Debido a que deberían poder anidarse, es necesario implementar una pila que recuerde la información de cada `for`, de manera que un `END FOR` cierre el último `for` que se abrió.

- Determinar si el programa ingresado en Pilot es válido.

Se determinó que esta característica no es crucial para el compilador, debido a que no afecta a la correctitud de éste, sino a su facilidad de uso.

Para realizarlo, se requiere de al menos:

1. Guardar las etiquetas definidas, para verificar que no haya dos definiciones de la misma etiqueta.
2. Hacer verificaciones de la cantidad de palabras de cada instrucción, dependiendo del tipo de instrucción.
3. Evitar la declaración de una función dentro de otra, y verificar que el `RET` al final de una función corresponde a esa función.

- Optimizar la traducción

Al inspeccionar un código `ARM` generado por el compilador, resulta evidente que el impacto de eficiencia debido al proceso de compilación es elevado.

Por medio de mantener el valor de los registros guardados en el compilador, se puede evitar que el compilador ingrese ciertas operaciones redundantes, haciendo que el programa generado sea más eficiente.

De todos modos, se debe ser cuidadoso, debido a los saltos condicionales, etiquetas y funciones que pueden hacer difícil mantener el valor exacto de los registros.

Resultados

Se han creado 20 programas en Pilot para probar la correctitud del compilador, con los que se han obtenido resultados satisfactorios.

Entre los programas en Pilot más destacados creados se encuentran:

1. `test7.txt`: Calcula el n -ésimo número en la sucesión de Fibonacci.
2. `test8.txt`: Invierte los dígitos de un número.
3. `test16.txt`: Recibe un rango $[a, b]$ e imprime todos los números primos en ese rango.
4. `test17.txt`: Multiplica dos números con el famoso algoritmo del campesino ruso.
5. `test19.txt`: Implementa un juego de frío-caliente.