The Value of Decaf470: An Extensible Compiler for EECS 470

By Douglas Li

Abstract

In an effort to bridge the gap between the computer architecture course (EECS 470) and compiler construction course (EECS 483) at the University of Michigan, I have refined the design of a high-level language used in EECS 483 (and similar classes at several other schools), Decaf, and implemented a fully-featured compiler supporting this language: Decaf470. The target platform is the subset of DEC Alpha 64-bit instruction set, which has long been the concern of 470 student projects. A goal of the project was to produce a set of tools useful in assisting these students on revealing errors in their pipeline designs – perhaps raising the bar on correctness. Moreover, it serves to tangibly demonstrate the broader view of how a program comes to life.

Introduction

Decaf470 is a compiler for a variant of the Decaf language into the subset of DEC Alpha 64-bit instruction set architecture (ISA), which is used in EECS 470 – Computer Architecture, a major design experience course (MDE), at the University of Michigan. For an overview of the specific ISA, go to *http://www.eecs.umich.edu/courses/eecs470/Alpha64_new.pdf*.

The standard version of the Decaf language is a simple high-level language model, which is used by compiler implementation courses at various universities. A handful of professors around the country have helped created it, modify it, or add to it, and there are more specific variations, all within the same level of complexity. At its core, it is based on a subset of Java combined with some C/C++ flavor. Notably, it was featured in EECS 483, in the winter term of 2009 under Prof. Satish Narayanasamy.

Decaf is a strongly-typed, simple, object-oriented language with support for inheritance and encapsulation. The design of Decaf has many similarities with other programming languages that are familiar to us, such as C, C++, and Java. However, Decaf is not an exact match to any of those languages and has its own properties. The feature-set has been trimmed down considerably from what is usually part of a full-fledge programming language. Despite these limitations, a Decaf compiler will be able to produce interesting and non-trivial programs. The specification can be found here: *http://www.eecs.umich.edu/courses/eecs483/decafOverview.pdf*.

The main distinctions of Decaf are that there are no pointers or addresses (every object is a reference just like Java), arrays are like those in Java, no multiple inheritance (except for *implementing* multiple *interfaces*), functions maybe declared outside classes like in C, and no `#include`. Decaf does not allow separate modules or declaration separate from definition.

Definition of the Problem

As it stands, Decaf lacks many keywords and features which would make it comparable to actual C++ or Java. Coupled with the academic model of Decaf are some skeleton files which act as the framework to completing the assignments in EECS 483. The framework targets a 32-bit MIPS platform, and although initially it appears to support changing the backend, upon further investigation it was determined that it lacks the flexibility to be fully cross-targeting.

From observing many semesters of EECS 470 at the University of Michigan, it was noticed that many students desired both more stringent assembly code test cases for their final project, and for a painless means to generate them. One rather natural method to generate assembly code is by feeding a high-level program written in a language such as C++ or Java, into a compiler. However, the specific ISA is very specific to the course and currently there exists no such compiler. Past attempts to employ a mainstream compiler such as GCC for its cross-compiler capabilities into the entire Alpha ISA, have yielded code that requires *significant* post-processing in order to have a remote possibility of being 470-legal.

Discussion of the Potential Solution

Immediately after taking EECS 483 and with past experience in EECS 470, I possessed both the ability and motivation to undertake such a project. The natural solution is to design a full-fledged computer language that suits the nature of 470 target hardware: no floating-point, no I/O, and target a 64-bit architecture from making modifications to the Decaf standard. An objective was to go from a 'toy' language to a powerful and expressive real-world language. Ideally, any C, C++, or Java program – which *can* (in theory) be executed on a working 470 project pipeline – may be slightly altered to become valid Decaf470.

The bulk of the project was to actually implement a compiler based off of the 483 framework – after making some needed improvements in increased extensibility. Additions to standard Decaf, which was featured include **`++`**/**`--`** (pre- and post-, increment and decrement), **`switch`** statements, **`do-while`** loops, built-in **`exit()`**, type-casting functions, adding *initializers*, better built-in ASCII string support, and an independent preprocessor which performs any **`#define`**/**`#ifdef`**/**`#ifndef`**/**`#endif`** and comment removal. Also, in light of the opcodes in this Alpha subset, the usual bit-manipulation operators will be supported, as are signed versus unsigned values. The high-level code would be mapped one-to-one to assembly, with little to no compiler optimizations. The end result is both a "*.s file" containing assembly code and also a file consisting of the machine code, ready to be loaded into a 470 pipeline.

The C++ ANSI standards were consulted whenever an implementation detail needed to be resolved that wasn't already evident in standard Decaf, such as the operator precedence (i.e. order of operations) or what order to perform the various preprocessor passes. Hence, Decaf470 is also very coherent: coding idioms which perform a certain task in C/C++ will do the same in Decaf470.

As an added bonus, the custom made *malloc()* and *free()* low-level functions will serve to expose a multitude of faulty pipelines. Furthermore, the set of tools developed alongside Decaf470, to expedite debugging the compiler while under development, would be published open-source and potentially help 470 students. These tools include: a *disassembler*, a *hardware emulator*, an *interactive debugger* for the machine code, a layout of the decode for the full Alpha instruction set, detection of non-subset instructions or other invalid instructions, and coinciding documentation – treatises which would otherwise still be useful for EECS 470 and for EECS 483 regardless of the project itself.

The assembly code generated by Decaf470 contains ample comments which help the end-user see how each line of their high-level code corresponds to the low-level code.

The compilation errors Decaf470 emits provides the line number, column number, and shows the actual code in question underlined. After encountering one error, the compiler continues on and outputs other independent errors in the code – allowing the end-user to make multiple edits at a time before initiating another compilation.

Example of Decaf470 Code

simple.decaf:

```
    ├----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+---
 1  void main() {¶
 2  ····string str; // strings are built-in types, no need for #include <string>¶
 3  ····str = "Hel\¶
 4  lo "¶
 5  ···········"World!";¶
 6  ····/* comprehensive preprocessing of escape characters,¶
 7  ·····and implicit string literal concatenation */¶
 8  ····// btw, you can't nest block-comments¶
 9  ¶
10  ····for(i = 0; i < 10; i++) // where is 'i' declared?¶
11  ·····func(i); // where is func declared?¶
12  }¶
13  ¶
14  void func(int x) {¶
15  ····return;¶
16  }¶
17  ¶
18  int i;¶
19  ¤
```

Decaf has some strange scoping rules, which I decided to carry over to Decaf470. Invoke the compiler using "./dcc simple.decaf > simple.s" to produce the commented assembly, or go directly to machine code using "./dcc simple.decaf | ./vs-asm > simple.mem".

Also try "./dcc -d preprocessor" to see a preprocessing pass only. It has very powerful preprocessing capabilities, and supports #define, #undef, #ifdef, #ifndef, #else, and #endif. However, observe the following behavior:

```
#define PROCEDURE void
#define BEGIN {
#define END }
#define BINKY 2 + 2 * 10

#ifdef PROCEDURE
#PROCEDURE not_main()
#BEGIN
    int x;;
    x = #BINKY;
    return;
#END
#endif

#ifndef PROCEDURE
#PROCEDURE not_main_either()
#BEGIN
    int y;
    y = #BINKY;
    return;
#END
#endif
```

"./dcc -d tokenizer" is also somewhat useful to see how it performs initial tokenization (this option may be combined with the preprocessor option in any order).

Decaf470 is currently as strongly type-safe as a language could possibly be. Thus recall that "123" is a signed value, while "123U" indicates unsigned in C, and that strictly speaking you should be careful mixing the two types. In Decaf470, the compiler explicitly enforces such rules.

```
long[] a;

void main()
{
    unsigned int i, j;
    signed long temp;
    long[] input; // needed to split these lines, since we don't have Initailizers in Decaf (yet)
    input = {3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9, 3}; // 16 elements
    // try this with negative numbers, it should work too

    // populate it into heap, since "input" is const/literal array
    a = NewArray(input.length(), long);
    for(i = 0U; i < input.length(); i += 1U)
        a[i] = input[i];

    for(i = 0U; i < (a.length() - 1U); ++i) // btw, pre-inc gives more efficient asm code
        for(j = 0U; j < (a.length()-i-1U); ++j)
            if(a[j] < a[j+1U]) {
                temp = a[j+1U];
                a[j+1U] = a[j];
                a[j] = temp;
            }
}
```

This program "bubble_sort_test.decaf", shows just how nitpicky the compiler can be. Note that this corresponds with "sort.s" in the suite of 470 test cases. Also, the lack of "DeleteArray()", does leave a so-called "memory leak". Other heap-management functions include "New()", "Delete()", "NewString()", and "DeleteString()". The underlying "malloc()" and "free()" code it injects from the assembly library, is incredibly useful for testing 470 pipelines, as it is currently the densest code output from this compiler (in terms of back-to-back dependencies).

This barely scratches the surface of what type of code that Decaf470 is capable of handling. It supports “abstract classes” and “pure virtual functions” through the use of Java-style “interfaces”. It fully supports many varieties of polymorphism, and can handle single-inheritance of arbitrary depth (by default all functions are virtual). It can accept typical “classes” used in C++ (except for templates), but does not allow splitting the declarations and definitions (instead of *.h files and *.cc files, class declarations follow closely to Java). Typecasting can be done using the syntax “StaticCast(expr, to_type)”. The use of “void *” can be imitated using the Java notion of “Object”, an empty class that all other classes implicitly derive from. But to be absolutely clear, there are no pointers or addresses to worry about in Decaf (or Decaf470).

Keywords that are supported are listed below, and those which are not do generate an error or warning by the compiler. These are from standard Decaf: “void”, “class”, “extends”, “this”, “interface”, “implements”, “while”, “for”, “if”, “else”, “return”, and “break”. Additionally in the 470 variant, we have: “switch”, “case”, “default”, “do”, “continue”, “null”, “true”, “false”, “signed”, and “unsigned”. Due to the lack of floating-point instructions, “double” was removed (although “float” aliases to double) – these types are not flagged as errors until much later in front-end stages of compiling.

Types allowed are “int/uint”, “char”, and “string”, but “short/ushort” and “long/ulong”, alias to “int/uint”. Also, you create your own types using “class”, “interface”, and any number of “[]” for arrays. Built-in functions include “New()”, “NewArray()”, “Print()”, “ReadInteger()”, “ReadLine()”, “Exit()”, “Delete()”, “DeleteArray()”, “NewString()”, “DeleteString()”, and “StaticCast()”. Those relating to I/O are obviously going to throw errors (unless passed a command-line flag to compile into MIPS).

Operators include: “<=”, “>=”, “==”, “!=”, “&&”, “||”, “++”, “--”, “<<”, “>>”, “[]”, “+=”, “-=”, “*=”, “/=”, “%=”, “<<=”, “>>=”, “&=”, “|=”, “^=”, and single character operators “+\-*/%<>=!;,.[\](){}?:~&|^” (although anything involving division or modulus will throw errors due to the lack of such operators in Alpha64, not just the 470 subset). Pre- and Post- increment/decrement work as expected. In case you are wondering, all the operators (especially the left and right shift) do consider whether the operands are signed or unsigned and emit the correct assembly code (say logical- versus arithmetic- shift).

The operator precedence works according to ANSI C standards, with the exception of disallowing commas to separate multiple statements in between ‘?’ and ‘:’ of the ternary operator. Within a switch statement, the “default” label is optional, but if given, must be last. Every escape sequence (in ANSI C) is supported, and this includes “\n”, “\t”, “\\”, the many octal codes, and the many hex codes. There is a built-in “.length()” for any array, and also for strings it functions like a “strlen()” call in C (e.g. it might segfault if not null-terminated). Array indexing can be applied to a string to return a char, and similarly the Java style of “.charAt()” gives the expected result. Another neat behavior is that it very carefully *sanitizes* and reconstructs string literals (appropriately respecting escape sequences).

There is runtime array index out-of-bounds checking like Java, and if the number passed to a "New()" is negative, that will also trigger an exception. Exceptions are encoded as "call_pal 0xDECAF" and cause a 470 pipeline to "halt on illegal instruction". This exception could be encountered at runtime for various other things as well.

Some amount of compiler optimizations are performed, such as converting a post-increment into a pre-increment wherever allowed, as they occupy one less assembly instruction. Also, multiplications by powers of 2, are replaced by equivalent bit-shifts – as are small numbers close to a power or 2. This behavior can be disabled if desired.

Any errors in the source Decaf code, are output to the terminal with line number, and the underlined code. It can also report multiple further errors past the first error. Try it!

Alternative Solutions Deemed Inappropriate

When employing GCC for its cross-compiler capabilities, we would yield code that requires significant post-processing in order to have a remote possibility of being 470-legal. Discussing the outcome of this approach with the various people who have attempted this, it was clear that a more invasive solution was needed.

An obvious suggestion is to instead modify the GCC source code to target the 470 ISA; after all, GNU's GCC is open-source. C/C++ has a large number of mechanisms which would have been impossible to map to our Alpha-subset: most noticeable being the lack of support for floating-point. Recall that C/C++ has exotic keywords, low-level libraries, and unmanageable compiler optimizations (many of which cannot be turned off). Another problem is that adding such a backend to GCC or modifying the current Alpha-backend, was found to be very difficult to accomplish. However, the exercise of dissecting GCC's source code did prove helpful in solving issues encountered on Decaf470.

Attractive Maintainability

With the proposed solution, the EECS 470 staff can maintain and improve Decaf470 very easily. Moreover, future enhancements might come from 483 project teams. Every single aspect is not only possible to control, but typically requires editing merely a few lines in the source code. For instance, it would require only commenting-out a single line in the *lexicon tokenizer* in order to remove **`char`** or **`unsigned`** from being a keyword. Commenting out one small block of code could allow the maintainer to prevent replacing **`for(i=0; i<N; i++)`** with **`for(i=0; i<N; ++i)`**. Say you want to require every if-statement to be followed by an **`{…}`** (instead of also allowing a single line body) – alter a few lines in the *parser* to adjust the actual grammar. Adding an entirely new keyword or changing the behavior of a current keyword is very much achievable for a typical undergraduate student who has taken a course such as EECS 281.

The possibilities of supplementing the compiler with features and static optimizations are quite limitless – both maintainer and end-user may do so to their heart's content. It was written in such a way as to require minimal coding experience to perform a large majority of the changes one might desire.

On the other end, the assembly code output of Decaf470 could also be adjusted. It is possible to change how nodes of the *abstract syntax tree* (AST) correspond to actual assembly code.

Limitations

Decaf470 does have its limitations. Most glaring, the assembly code being produced is terribly inefficient – register values spilling to and filling from after every use. Fixing this behavior requires designing some register allocation scheme. Currently, there is no scheduled plan to do so.

Another possible issue is that the heap management system is very simple, it grabs the first contiguous block of memory it can. This can lead to fragmentation for certain kinds of input cases. For programs with large heap utilization, the heap allocation subroutine occupies a significant number of clock cycles.

In the ISA, there was no way to tell the pipeline if a runtime exception had occurred. The convention for checking array bounds at runtime that Java does, was adopted by Decaf (and thus Decaf470). Many other scenarios where a program must throw an exception exist. What happens now is that a **`call_pall 0xDECAF`** instruction is executed. This should help for various debug purposes, but without prior knowledge of this convention, a 470 pipeline would treat it as an invalid instruction and halt on it. This may or may not be clear to the student what the nature of the halt was.

A rather unexpected side-effect was discovered: some Python syntax does work in Decaf470. Code such as **`"some string".length()`**, **`"some string"[i]`**, and **`{1, 2, 3}[i]`** are syntactically acceptable due to the loose type-checking performed.

The typical C/C++ operators of '`/`' and '`%`' are not supported, due to not having a direct opcode equivalent available in the Alpha ISA (not just the 470-subset).

Known Bugs

Fortunately, all of the defects revealed during its extensive validation were solved. Thus, there are no known bugs currently, in terms of pure correctness. Time will tell if issues arise during the first semester of beta testing among fall 2009 semester of 470 students.

Conclusions

Decaf470 is a usable and validated compiler for EECS 470 students. It might help them expose problems in their final project designs, and enlighten them as to how their course work or chosen specialty fits into a larger whole. Personally, I know that when I was taking the course, I would have found such a tool useful. It completes the picture of how ideas of a computer programmer ends up being executed: the ideas become high-level code, which gets compiled into assembly, which gets translated into machine code, which in turn is executed on the pipeline of a processor. Analogous to how their pipeline design has several stages that act as an 'assembly line' completing the work, a compiler also has several stages which partition its work. It is only with the unification of EECS 483 material and EECS 470 material is the complete image of this process clear.

By clearly organizing the compiler's code into stages, and by anticipating future extensibility, any maintenance task is quite pleasant. The stages are *preprocessing*, *lexical tokenization*, *syntax parsing*, *semantic analysis*, *control-flow analysis*, *optimizations* (optional), and finally two layers of *code generation*. It turns out that Decaf470 compilation itself is quite fast, and comparable to GCC due to the amount of data caching it does internally (passing additional information from one stage to the next instead of regenerating it later), various coding tricks, and taking some broad ideas from GCC's source code.

This project was suggested to Prof. Mark Brehob, who typically teaches the 470 course, and the work done was supervised by him. Moreover, it may be the case that this project can be further leveraged for EECS 483 (Compiler Construction) or even to provide synergy between the two classes. Optimistically, spreading awareness of Decaf470 among 483 course staff and offering this tool to 470 students will assist in pushing the EECS department into turning 483 back into an MDE course, and also to institute a symbiotic relationship between the two classes.

The 470 students, who do not elect 483, would not otherwise appreciate the process. The 470 students, who do elect 483, would have a framework that allows them to develop compiler optimizations, and observe how their pipeline design would have handled it. Conversely, they can make changes to their pipeline and see how it copes with code that is more accurately depicts real-world code. Decaf470 is the start to building a bridge between the two course materials, and will demonstrate the tremendous untapped synergy that exists there.

References

"Compiler Construction EECS 483." *Electrical Engineering and Computer Science at the University of Michigan*. Ed. Prof. Satish Narayanasamy. Web. 22 Oct. 2009. <http://www.eecs.umich.edu/courses/eecs483/project.html>.

"EECS 470 Term Project Fall '09." *Electrical Engineering and Computer Science at the University of Michigan*. Ed. Prof. Mark Brehob. Web. 22 Oct. 2009. <http://www.eecs.umich.edu/courses/eecs470/470F09Project.pdf> .

"CMPT 379 - Fall 2007: Principles of Compiler Design." *School of Computer Science at Simon Fraser University*. Ed. Dr. Anoop Sarkar. Web. 22 Oct. 2009. <http://www.cs.sfu.ca/~anoop/courses/CMPT-379-Fall-2007/index.html#assignments>.

"EECS 470 Subset of the Alpha 64-bit ISA for Project 3 and Final Project." *Electrical Engineering and Computer Science at the University of Michigan*. Ed. Douglas Li. Web. 22 Oct. 2009. <http://www.eecs.umich.edu/courses/eecs470/Alpha64_new.pdf>.

"Decaf Language Definition." *School of Computer Science at Simon Fraser University*. Ed. Dr. Anoop Sarkar. Web. 22 Oct. 2009. <http://www.cs.sfu.ca/~anoop/courses/CMPT-379-Fall-2007/decafspec.pdf> .