Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.035, Fall 2018 Handout — Code Generation Project Thursday, Oct 4

DUE: Friday, Oct 26, 11:59 pm

Code Generation involves producing correct x86-64 assembler code for all Decaf programs. The two following projects will involve code optimizations. For now, we are not interested in whether your generated code is efficient. In fact, make it very inefficient.

By the end of code generation, you should have a fully working Decaf compiler. You'll be able to write, compile, and execute real programs on a real machine!

Project Assignment

For Code Generation, your compiler will translate your high-level IR into a low-level IR. Your low-level IR will include structures that more closely match the machine instructions of a modern architecture. Your compiler will then translate your low-level IR into x86-64 assembly code to be run on Athena. You should target the subset of the x86-64 ISA defined in the *x86-64 Architecture Guide*, which could be found on the course website, in either the Reference Materials section or on the course calendar.

Your generated code must include all runtime checks listed in the Decaf language specification. Additional checks such as integer overflow are not required.

The two final assignments, *Dataflow Analysis* and *Optimization*, will focus on improving the efficiency of the target code generated by your compiler. For this assignment, you are not expected to produce great code. In fact, even horrendous code is acceptable. When considering tradeoffs, always choose simplicity of implementation over performance.

You are not constrained as to how you go about generating your final assembly code listing. However, we suggest that you follow the general approach presented in lecture.

You will have a number of opportunities to do some creative design work for the code optimization projects. For this assignment, you should focus your creative energies on designing your Control Flow Graph, familiarizing yourself with our target ISA, your machine-code representations of the run-time structures, and generating correct assembly code. Do not try to produce an improved register allocation scheme; you will be addressing these issues later.

Compiling and Libraries

Your compiler will create a `.s` file, which can be compiled to create an executable file. You can use `gcc` to compile your assembly code with the following command on athena:

```
gcc foo.s -o foo
```

Then you could run the executable below:

`./foo`

Decaf does not have any input/output functions. Part of the assignment is to implement the standard x86-64 calling convention for `import` statements, so that you can interface with the outside world. Any function that is called using `import` needs to be linked in separately. `gcc` will link against any standard libraries, such as `printf` (you may need to use the `-l` argument for `gcc` to link some libraries). The testing files provided to you link against the standard C library. If you want to use functions that are not easy to use in Decaf (handle pointers, etc), you are welcome to write your own library calls in `C`, compile them to object files (using `gcc -c`) and then link them in by hand when compiling your assembly.

What to Hand In

Follow the directions given in project overview handout when writing up your project (repeated at the bottom of this handout). Reminder, the documentation will count towards 20% of your grade. Submitted repositories should have the following structure:

```
GROUPNAME
|
|-- src
|   |
|   ...                (full source code, can build by running `./build.sh`)
|
|-- doc
|   |
|   ...                (write-up, described in project overview handout)
|
|-- build.sh
|-- run.sh
|-- (etc)
```

You should be able to run your compiler from the command line with:

```
./run.sh --target=assembly <filename>
```

Your compiler should then write a x86-64 assembly listing to standard out or the output file specified by `-o`.

Nothing should be written to standard error for a syntactically and semantically correct program (or to standard out, if an output file is specified) unless the `--debug` flag is present. If the `--debug` flag is present, your compiler should still run and produce the same resulting assembly listing. Any debugging output is left to your own discretion. All errors should be written to standard error, with the exit code of array bounds errors being -1 and no return errors being -2.

You should submit the project using the course's submission web site by the due date. We will run your compilers on the test cases in the tests repository and on a set of hidden tests. We strongly recommend you write additional tests of your own, because the provided tests are nowhere near comprehensive.

Documentation

Documentation should be included in your source archive in the `doc/` folder, and you will have one design document/report written for each of the five projects. It should be clear, concise and

readable. Fancy formatting is not necessary, just give us a clear idea what you did for each project. Acceptable file formats are pdf.

Your documentation must include the following parts, which could be described as Design, Extras, Difficulties, and Contribution. Not every question or point of each part need to be addressed, just enough information to describe each portion effectively:

1. **Design** - An overview of your design, an analysis of design alternatives you considered, and key design decisions. Be sure to document and justify all design decisions you make. Any decision accompanied by a convincing argument will be accepted. If you realize there are flaws or deficiencies in your design late in the implementation process, discuss those flaws and how you would have done things differently. Also include any changes you made to previous parts and why they were necessary. This section should aid the TA in being able to read and give feedback on the code written.

2. **Extras** - A list of any clarifications, assumptions, or additions to the problem assigned. This include any interesting debugging techniques/logging, additional build scripts, or approved libraries used in designing the compiler. The project specifications are fairly broad and leave many of the decisions to you. This is an aspect of real software engineering. If you think major clarifications are necessary, consult the TA.

3. **Difficulties** - A list of known problems with your project, and as much as you know about the cause. If your project fails a provided test case, but you are unable to fix the problem, describe your understanding of the problem. If you discover problems in your project in your own testing that you are unable to fix, but are not exposed by the provided test cases, describe the problem as specifically as possible and as much as you can about its cause. If this causes your project to fail hidden test cases, you may still be able to receive some credit for considering the problem. If this problem is not revealed by the hidden test cases, then you will not be penalized for it. It is to your advantage to describe any known problems with your project; of course, it is even better to fix them. Also describe any section of your project that you would like to highlight for more feedback on/had questions on.

4. **Contribution** -A brief description of how your group divided the work. This will not affect your grade; it will be used to alert the TAs to any possible problems. (Projects 2 through 5 only.)