**Massachusetts Institute of Technology**
**Department of Electrical Engineering and Computer Science**

6.035, Spring 2010 Handout — Code Generation Project Monday, Mar 1

---

**Checkpoint and Design Document DUE: Wednesday, Mar 10**

**DUE: Monday, Tuesday, Mar 16**

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.

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 an AMD Opteron machine. You should target the subset of the x86-64 ISA defined in the *x86-64 Architecture Guide*, which is an appendix to this document. For a given input file containing a Decaf program, your compiler must generate an assembly language listing ($\langle filename \rangle$`.s`).

Your generated code must include all runtime checks listed in 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. (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 low-level IR, 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.

## System Usage

We have setup two machines for this course for you to use. These are:

- `tyner.csail.mit.edu`

- `silver.csail.mit.edu`

These machines are setup to be part of the CAG (Compilers and Architecture) group at CSAIL. The accounts and passwords are different from your regular Athena accounts. Each group will use a single account in the form `le0X`. Your passwords have been put in your Athena lockers in files `le0X-pass`. Please change your passwords immediately. You are free to use these machines for your 6.035 related needs. You may, however, find it more convenient to do most of the development on a laptop or desktop, and use these servers for testing. This would be particularly true if you use an IDE like Eclipse for your development.

## Compiling and Libraries

Your compiler will create a `.s` file, which can be compiled to create an executable file. You can use `gcc4` (`gcc` 4.1.1) or `cc` (which is soft-linked to `gcc` 3.4.6) to compile your assembly code.

Decaf does not have any input/output functions. Part of the assignment is to implement the standard x86-64 calling convention for `callout` statements, so that you can interface with the outside world. Any function that is called using `callout` needs to be linked in separately. `gcc` will link against any standard libraries (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. Your design documentation should include a description of how your IR and code generation are organized, as well as a discussion of your design for generating code.

Each group must place their completed submission at:

/mit/6.035/group/**GROUP**/submit/**GROUP**-codegen.tar.gz

Submitted tarballs should have the following structure:

```
GROUPNAME-codegen.tar.gz
|
‘-- GROUPNAME-codegen
    |
    |-- AUTHORS            (list of students in your group, one per line)
    |
    |-- code
    |   |
    |   ...                (full source code, can build by running ‘ant‘)
    |
    |-- doc
    |   |
    |   ...                (write-up, described in project overview handout)
    |
    ‘-- dist
        |
        ‘-- Compiler.jar   (compiled output, for automated testing)
```



* Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

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

```
java -jar dist/Compiler.jar -target codegen <filename>
```

Your compiler should then write a x86-64 assembly listing to: `<filename>.s`

Nothing should be written to standard out or standard error for a syntactically and semantically correct program 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.

## Grading Script

As with the previous project, we are providing you with the grading script we will use to test your code (except for the write-up). This script only shows you results for the public test cases.

The script can be found on athena in the course locker:

/mit/6.035/provided/gradingscripts/p3grader.py

The script takes your tarball as the only arg:

/mit/6.035/provided/gradingscripts/p3grader.py GROUPNAME-codegen.tar.gz

Or it works on the extracted directory:

/mit/6.035/provided/gradingscripts/p3grader.py GROUPNAME-codegen/

Please test your submission against this script. It may help you debug your code and it will help make the grading process go more smoothly.

## Checkpoint and Design Document

For this stage of the project, you are required to provide a checkpoint of your implementation and to submit a design document one week prior to the due date of the project. Follow the submission instructions above for the checkpoint. The TA will grab the checkpoint files from your group directory at 11:59 on the checkpoint due date. Your compiler does not have to produce correct code at the time of the checkpoint. The checkpoint exists to encourage you to start working early on the project. If you get your project working at the end, the checkpoint will have little effect. However, if your group is unable to complete the project, the checkpoint submission will play a critical role in your grade. If we determine that your group did not do a substantial amount of work before the checkpoint, you will be severely penalized.

The design document should describe your design and implementation plans for the code generation stage. This description will include the design of your low-level IR and a discussion of your code generation scheme and implementation. This document will be reviewed by the TA and feedback will be provided. This document also counts towards the project grade.



* Athena is MIT's UNIX-based computing environment. OCW does not provide access to it.

## Test Cases

We will run your compilers on the test cases in:

```
/mit/6.035/provided/codegen/
```

and on a set of hidden tests.

## Related Handouts

The *X86-64 Architecture Guide*, provided as a supplement to this handout, presents a not-so-gentle introduction to the x86-64 ISA. It presents a subset of the x86-64 architecture that should be sufficient for the purposes of this project. You should read this handout before you start to write the code that traverses your IR and generates x86-64 instructions.

6.035 Computer Language Engineering
Spring 2010