

Massachusetts Institute of Technology  
Department of Electrical Engineering and Computer Science

6.035, Fall 2006

Handout 8 — Code Generation Project

Thursday, October 5

---

**DUE: Thursday, October 26**

Code Generation involves producing correct x86-64 assembler code for all Decaf programs. The next two project assignments will involve code optimizations. For now, we're 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 architecture!

## Project Assignment

For CG (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 AMD64 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 (*(filename).s*).

Your code must include instructions to perform the runtime checks listed in Handout 5. Additional checks such as integer overflow are not required.

The two final assignments, *Dataflow Analysis* and *Optimizer*, 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

For now, you will develop your compiler on Athena or your own computer, perhaps using your group locker as a shared repository. To test your generated assembly, your group has an account on two AMD64 test machines:

- `chocura.csail.mit.edu`

- `urth.csail.mit.edu`

Your initial password is given in the file `1e0X-pass` located in your group locker. Please change the password immediately.

Unfortunately, these machines are used by members of the Computer Architecture Group for research. Due to this, we cannot allow you to develop on these machines. Thankfully, this situation is only temporary. Two additional AMD64 machines are on order (each with 2 dual core processors and 8GB of memory). Once these machines are available, you can use them for development as well as testing. The two new machines should be live within 1.5 weeks.

Until we get the new machines live, you will be forced to develop and test on separate machines. Once you generate an assembly output file, say `output.s`, you will need to copy it to one of the AMD64 test machines, somewhere under your group's home directory. Once copied, you can assemble it and link it using `cc` or `gcc4`. Then you can run it. The sequence will look similar to this:

```
athena% scp output.s le99@chocura.csail.mit.edu:
athena% ssh le99@chocura.csail.mit.edu
le99@chocura.csail.mit.edu's password:
-bash-3.00$ ls
output.s
-bash-3.00$ gcc4 -L. -l6035 output.s -o output
-bash-3.00$ ./output
Hello, world!
-bash-3.00$ logout
Connection to chocura.csail.mit.edu closed.
athena%
```

We have provided you with a static library, `lib6035.a`, that includes the `get_int_035()` function. The function will read a single integer value from the stdin (terminated by a newline) and return its value. You can find the file in:

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

Copy (`scp`) the file to your testing directory on the testing machines.

## What to Hand In

As always, follow the directions given in Handout 3 when writing the hardcopy documentation for your project. The electronic portion of the hand-in procedure should also be familiar: Provide a gzipped tar file named `leNN-codegen.tar.gz` in your group locker, where `NN` is your group number. This file should contain all relevant source code and a `build.xml`. Also provide a Java archive named `leNN-codegen.jar`.

Unpacking the tar file and running `ant` should produce the same Java archive. One should be able to run your compiler from the command line with one of the following:

```
java -jar leNN-semantic.jar <filename>
java -jar leNN-semantic.jar -target assembly <filename>
```

Your compiler should write a x86-64 assembly listing to a file of the same name with a `.s` extension. 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.

## Test Cases

We will run your compilers on the test cases in:

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

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.