

1 Introduction

In this lab assignment, we get to the final part of the compiler: the back-end. The task in the back-end is to generate machine code from the FIR we produced in the middle-end. The back-end has three parts:

1. The FIR code is converted to real assembly code. In our case, we are producing code for the Intel x86 architecture. The x86 code generator is defined in the file `arch/x86/x86_codegen.ml`.
2. Dataflow analysis is used to calculate the *interference graph* that defines which variables can't be stored in the same machine registers. The dataflow analysis is defined in `arch/regalloc/ra_live.ml`.
3. The *register allocator* brings it all together. The register allocator is passed the assembly, and it performs a loop:
 - (a) perform dataflow analysis to get an interference graph,
 - (b) try to assign the variables to registers. If some variables are spilled, rewrite the assembly code and start over.

2 What you need to do

For this lab, you will implement three parts:

1. x86 code generation (in the `arch/x86/x86_codegen.ml` file),
2. dataflow analysis (in the `arch/regalloc/ra_live.ml` file),
3. register allocation (in the `arch/regalloc/ra_main.ml` file).

The register allocator will be the most difficult part, you will benefit greatly by reading the book (especially Chapter 11).

3 The x86 instruction set

For documentation on the Intel instruction set, you should refer to the Intel documentation (on the CS134b home page). The instructions follow the Intel notation: if there are two operands, the first is the destination. This differs from `gas` assembler syntax, where the operands are in reverse order (why, I don't know).

4 The runtime

The runtime includes a garbage collector in `arch/x86/runtime/x86_runtime.c` and some glue code in `arch/x86/runtime/x86_glue.s`. If you run `cons -t` in the runtime directory, it will produce a library `x86runtime.a`.

You can link against the runtime with `gcc`. For example, here is how you would compile `Test5.java` and link `Test5.java`.

```
// In the main directory
% ./fjc -o Test5.s Test5.java
% gcc -o Test5 Test5.s ../arch/x86/runtime/x86runtime.a
% ./Test5 0 0 2000 > x.pgm
```

5 What to turn in

You should turn in your entire compiler in your submission directory on mojave (you should probably keep a copy in your CS account too). If you are working in a group, only one of you should do the submission, and you should send mail to `cs134-admin` to tell us of your submission.

In addition, you should include the following.

- A README file explaining what you did, how it works, and whether you had any problems.
- A DIFF file generated using the command `cvs diff` in the `arch/i386` and `cg` directories. If you like, you can insert this into the README file, with brief explanations of what you changed.
- The files `Test1.s`, `Test2.s`, `Test3.s`, `Test4.s`, and `Test5.s` generated using your compiler with the `-o` option from the main directory.

```
% ./fjc -o Test1.s Test1.java
% ./fjc -o Test2.s Test2.java
% ./fjc -o Test3.s Test3.java
% ./fjc -o Test4.s Test4.java
% ./fjc -o Test5.s Test5.java
```

- Output and timing information for each of the programs.

Finally, all extra credit optimizations should be submitted with lab5. Indicate which optimizations you implemented, so we can be sure to grade them.