CS 322 Winter 2014 Homework 5: Generating X86-64 Assembly Code
Due (via DTL) on Friday, March 14, 2014 at noon

This assignment asks you to make extensions and improvements to a code generator that converts 3-address IR (similar to that used in Homework 3) into X86-64 assembly code (as explored in Homework 1). The assignment assumes familiarity with the contents of Lab 8, where the code generator was introduced and modified.

To complete the assignment, you will need a Java 1.6 compiler (as usual) and access to an X86-64 compatible processor (as in Homework 1). With appropriate minor tweaks, the generated code should work on either Linux (e.g. the `linuxlab.cs.pdx.edu` machines) or MacOS. If you want to use Windows, you are on your own.

Your homework solution should be submitting using the D2L dropbox in the form of a single file called `sol5.zip`, which contains your revised implementations of `IR.java` and `Assignment.java`, together with any other auxiliary source files you created, and a `README` file. This zip file can be created using a command like:

```
zip sol5.zip IR.java Assignment.java README
```

More details on what should be in these files are given below.

This assignment will be scored out of 25 points, following the distribution indicated below. Your score on this assignment contributes 15% towards your final grade. The minimum passing score (i.e. to avoid an F for this assignment, and for the class) is 5 points.

Provided Materials

You should download the "Homework 5 Materials" (`hw5.zip`) from the D2L website. Upon unzipping, you should see a `hw5` directory with the following iterms:

`X86Gen.java`	main program for generator
`IR.java`	IR representation extended with methods for X86-64 code generation (corresponding to final version from Lab 8)
`Assignment.java`	simplistic register assignment (final version from Lab 8)
`Liveness.java`	computation of live ranges and intervals
`X86.java`	utility support library for generating X86-64 code
`irParser.jj`	parsing for IR
`lib.c`	runtime library for generated X86-64 code
`IRInterp.jar`	executable version of IR interpreter
`Makefile`	for the generator
`tst/`	directory with sample test programs
`run`	sample script for running tests

The IR used here is similar to that of Homework 3, but with a few small changes: notably, type information is associated only with `LOAD` and `STORE` instructions, not with variables.

An X86 code generator corresponding to the final version seen in Lab 8 can be built using the `Makefile`. On MacOS, first edit function `globalize` at the top of file `X86.java` in the obvious way.

The code generator reads an `.ir` file specified on the command line, uses the parser specified in `irParser.jj` to convert it to internal form, and then generates assembly code on standard output, which is normally captured into a `.s` file. The generator should be run with the `-ea` flag to enable Java assertion checking, e.g. as follows:

```
java -ea X86Gen f.ir 1> f.s
```

The generated `.s` file can then be assembled and linked via `gcc` with the "standard library" `lib.c` to make an X86-64 executable, as follows on Linux:

```
gcc -o foo foo.s lib.c
```

or on MacOS:

```
gcc -Wl,-no_pie -o foo foo.s maclib.c
```

When executed, programs should behave as they do when evaluated using the IR interpreter (and, for those derived from `miniJava` sources, as they do when compiled using `javac`).

The `tst` directory contains examples of IR generated from `miniJava` source files, which are also included for your reference. The output to be expected by running each of these tests is in the corresponding `.out` file. You will probably wish to develop additional IR files to use as tests, which is most easily done by modifying one of the existing files. You can test the behavior of your IR code by running the interpreter; for example, to run file `foo.ir`, type

```
java -jar IRInterp.jar foo.ir
```

The `run` script can be used to automate some simple regression testing.

Your Tasks

Make the following three improvements to the existing generator implementation in `IR.java` and `Assignment.java`. It is recommended that you attempt these improvements in the order listed here. Combine all your changes into single new versions of the files `IR.java` and `Assignment.java`. You may introduce additional java source files if you find it useful. Also, write a brief `README` file indicating which improvements you attempted, and giving a summary of what you did for improvement 3 (below).

1. [5 pts.] Binary operations involving the relational operators are not supported in `IR.java`, making it impossible to generate assembly code for `test03` or `test19`. Add appropriate support for this feature. Hint: Use the X86 `set` instructions, but be careful about sizes.

2. [10 pts.] The code generator assigns a unique X86 register to each `IR.Reg`. It does this once and for all for the entire function; this approach doesn't always produce great code, but it is simple and adequate. However, the register allocator in `IR.java` is much *too* simplistic: once an X86 register is used for one IR register, it is never re-used for another one, even if the two IR registers have disjoint live ranges. Thus, the generator quickly runs out of registers when given functions of even modest size (e.g. `test18`, `test19`, `test35`, and `test38`). Improve this situation by implementing a linear scan register allocator instead, using the algorithm described in lecture. As the basis for your implementation, use the `liveIntervals` data structure, which maps `IR.Reg`s to `Liveness.Interval`s. The latter objects have two integer fields `start` and `end`, representing the first and last indices in the function's `code` array at which the register is "live out." (The existing `IR.java` code already computes `liveIntervals` wiithin the `assignRegisters` routine.)

3. [10 pts.] The current generator produces distinctly sub-optimal code for IR code derived from miniJava array operations. For example, consider the code generated for the first evaluation of `A[j]` in the `selectionSort` method in `test38` (once you have completed part 2, above). It requires six X86 instructions, including an expensive (multicycle) `imulq`, something like this (your register names may differ):

```
# 9.   t2 = j * 4
    movq %rcx,%rdi
    imulq $4,%rdi
# 10.  t3 = A + t2
    movq %rdi,%r10
    movq %rsi,%rdi
    addq %r10,%rdi
# 11.  t4 = [t3]:I
    movslq (%rdi),%rdi
```

This operation can certainly be done with fewer instructions; your task is to alter the generator so that it produces shorter and/or cheaper code sequences for array dereferences like this one. In fact, a one instruction sequence suffices:

```
movslq (%rsi,%rcx,4),%rdi
```

but you are *not* required to achieve this. *Any* improvement in the sequence will do to obtain some credit; the more you improve it, the better your score. Note that your generator needs to produce improved code for *arbitrary* array dereferences, not just this particular one!

Obviously, this task is rather open-ended, and is meant to encourage your creativity. Here are some hints and ideas to get you started:

- There are several cheaper ways to encode multiplication by powers of 2.
- The existing generator doesn't notice that most binary operations are commutative, and thus misses an easy way to shorten the sequence generated at line 10.
- One obvious problem is that the IR code generator itself produced three instructions (and two temporaries) from the original miniJava source, but the current X86 generator examines only one IR instruction at a time. You can alter the generator to examine multiple IR instructions at once to look for particular patterns. (This is awkward to do in an object-oriented style: be prepared to make heavy use of `instanceof`.)
- Alternatively, although you cannot change the form of the IR fed into your generator, you can alter the IR internally by *pre-processing* it into a variant form, perhaps having a large instruction set, before doing X86 instruction generation.
- In principle, another approach would be to *post-process* the generated X86 code, e.g. using peep-hole optimization to improve the instruction sequences. But since the generator currently just prints out X86 assembler directly rather than generating an internal form of it, this will be a lot of work, and is not recommended.
- For general inspiration, try writing simple C programs and compiling them using `gcc -S` to see what the generated `.s` assembly file looks like.

Whatever you do, your generator must continue to produce valid output on *all* IR inputs! Be careful: some changes that are innocent on these particular test IR files will cause other IR files to break. In particular, be sure to make use of the `liveRanges` information to make sure that you do not remove instructions that set registers whose values may be needed in the future.

To avoid getting bogged down in large implementation efforts that might not get finished (or yield improvements), it is strongly recommended that you start by making simple changes, and proceed to fancier ideas only as your time allows. Don't forget to record what you did in the `README` file.

Other Information

You may wish to refer back to the Week 1 handouts on the X86-64 instruction architecture and using `gdb` for debugging assembly code. Remember to add a `.file` directive to the top of any `.s` file that you want to debug under `gdb`.

The existing code generator uses a convention whereby all values in registers are valid 64-bit quantities. (This is unlike the approach used in the Week 1 lab and Homework 1, where we tracked how much of each register was valid at each point, and selected the correct instruction forms accordingly.) This has the advantage that we can just use quad (64-bit) instruction forms throughout—except when loading or storing from memory, or for certain instructions that don't exist in quad form.

The existing code generator completely avoids the possibility of storing arguments or locals on the stack:

- Generation fails on functions with more than six arguments.
- Generation fails if there are insufficient registers for the register assignment algorithm we're using (i.e., we don't attempt to spill).

You should not worry about fixing these limitations (although of course they would be unacceptable in a real compiler). But of course one of the purposes of task 2 is to reduce the demand for registers.

The generated code obeys the X86-64 calling and register usage conventions. It does not use a frame pointer, since there is (essentially) no frame! Any callee-save registers used by a function need to be pushed onto the stack at the start of the function and popped back off at the end. For simplicity, we do not use caller-save registers across calls at all; that way, there is no need to generate code to save and restore them around calls.

Two registers (`%r10` and `%r11`) are reserved as local temporaries for use *inside* the translation of a single IR instruction.

Each IR file should contain a function called `_main`, representing the "main program" where the generated program should start running. The file `lib.c` contains an ordinary C `main` function that simply invokes `_main`. It also contains implementations of all the "built-in" functions used by `miniJava` programs.

MacOS configures `gcc` to associate the assembly-level name `_foo` with the C-level name `foo`. The function `globalize` at the top of `X86.java` compensates for this.