CS 322 Homework 4 – due 1:30pm, June 7, 2012

Generating X86-64 Assembly Code for fab

Working individually or in teams of two, write an X86-64 code generator for the **fab** IR. The IR is the same as the one used in assignment 3. In principle, your generator should work on all IR programs; however, in practice it suffices for it to work on IR programs generated from **fab** programs using the reference IR code generator from assignment 3.

Don't worry about the runtime behavior of programs that attempt to allocate more heap space than is available at runtime, or that pass more than six arguments to a function.

The code generator should read an `.ir` file specified on the command line, use the existing parsing code in `ir.jar` to convert it to internal form, and then generate assembly code on standard output, which is normally captured into an `.s` file. This file can then be assembled and linked via `gcc` with a "standard library" `fablib.o` to make an X86-64 executable. When executed, programs should behave as they did under the various interpreters in previous assignments.

Runtime errors such as array bounds violation should be handled by generating code to call the appropriate built-in runtime library function, e.g. `_array_bounds`, as detailed below.

As usual, a "correct" generator is available in the jar file `X86gen.jar`; it can be run on file `foo.ir` by typing

```
java -classpath X86gen.jar:ir.jar X86GenDriver foo.ir > foo.s
```

It is not necessary that the code you generate be identical to what the reference `X86gen` generates, but it must have the same observable behavior when assembled and executed.

Linkage and Library

Each function of a **fab** program should generate a corresponding assembly language function of the same name, but with two leading underscores added. In particular, the **fab** main function should generate an assembler function `___MAIN`. The `fablib.c` file contains an ordinary C `main` function that simply invokes `___MAIN`, and returns its integer return value as the completion status code (always 0, for a **fab** program). Thus a complete executable can be obtained by assembling your `.s` file, compiling `fablib.c`, and linking them together, e.g.:

```
gcc -m64 -g -o foo foo.s fablib.c
```

IO and heap memory allocation are performed by issuing C-style procedure calls to library functions also provided in `fablib.c`. These functions are: `_read_int` (which returns an integer result value), `_write_int` (which takes an integer argument), `_write_string` (which takes the address of a string as argument), `_write_bool` (which takes a boolean byte value as an argument), `_write_newline` (which takes no arguments), `_bounds_error` (which takes no arguments, issues the message `Array bounds violation` to `stderr`, and exits), `_nil_pointer` (which takes no arguments, issues the message `NIL pointer dereference` to `stderr`, and exits), and `_alloc` (which takes an integer size argument in bytes and returns the address of the allocated storage).

MacOS configures `gcc` to associate the assembly-level name `_foo` with the C-level name `foo`. For convenience of those developing code on MacOS, there is a separate version of `fablib.c` that is identical except that it drops the leading underscores from function names (since `gcc` will put them back on again).

Implementation

Source file `IR.java` documents the IR representation. It is the same as before except that each IR instruction line now has an initial line number (as it did in the original HW3 version, but not the revised version). File `ir.jar` contains classes for representing, parsing, and generating IR. It is slightly modified from the HW3 version to include all the files needed for IR parsing (so frontend.jar should no longer be needed at all), and the parser had been modified to expect (and ignore) the instruction line numbers. Source file `X86.java` contains support classes for emitting X86-64 assembly code. File `Liveness.java` contains support classes for calculating live ranges of IR operands. File `X86GenDriver.java` contains a top-level driver.

Your code generator must be defined by a class `X86Gen` that compiles and links with `ir.jar`, and the `X86`, `Liveness` and `X86GenDriver` classes provided; you must not change these classes. Thus, your `X86Gen` class must implement the method

```
  static void genProgram(IR.Program p)
```

There is no provision for your generator to throw catchable exceptions, but it can throw `Error` exceptions or use `assert`s to report uncatchable errors. Remember to use the `-ea` flag to `java` to enable assertion checking at runtime.

Your generator class should be placed in a separate file `X86Gen.java`, which can be compiled using

```
  javac -classpath .:ir.jar X86Gen.java
```

assuming that `X86.java` and `Liveness.java` (or their class files) are already in the current directory. A skeleton for a working generator is available in `X86Gen0.java`; you are encouraged to use this as the basis for your generator. Places that need attention are marked with "..." or "!!!" in comments.

An important task of the generator is to map `IR.NamedReg` and `IR.TempReg` operands to X86 registers or memory locations. It is simplest to build this mapping once and for all for the entire function; this approach doesn't always produce great code, but it is adequate for our purposes. However, the register allocator in the skeleton is *too* simplistic: once a register is used for one IR operand, it is never re-used for another one, even if the two operands have disjoint live ranges. You should implement a linear scan or graph-coloring register allocator instead; the linear scan approach is easiest. You can use the methods in class `Liveness.java` to calculate which temporaries are live after each instruction, and to calculate the corresponding live intervals. In particular, you can use the method `calculateLiveIntervals` which calculates the live interval associated with each register used in a function, and returns a `Map` from `IR.Reg`s to `Liveness.Interval`s. The latter objects have two integer fields `start` and `end`, representing the interval by its first and last indices in the function's `code` array. The skeleton generator uses this map to enumerate all the operands that appear in the function, but it doesn't make much use of the range information; you can do better.

The generated code must obey the X86-64 calling and register usage conventions, which are detailed in the reference document on the course web page. There is no need to use a frame pointer; variables in the frame can be accessed relative to the stack pointer, which seldom needs to change in the middle of a function (and always in a very controlled way). 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. It is simplest not to use caller-save registers across calls at all; that way, there is no need to generate code to save and restore them around calls.

You will also want to reserve at least two registers (`%r10` and `%r11` are most suitable) as local temporaries for use *inside* the translation of a single IR instruction.

Assembly Code Features and Tricks

Even local labels must be unique across an entire `.s` file, so the IR's label numbers (which are per-function) need to be disambiguated. A simple way to do this is just to give each function in the program a unique number.

String literals can be defined using `.asciz` assembler directives. It is easiest just to store these up and emit them all at the end of the function.

Most processors have certain idiosyncratic instructions, and the X86 is certainly no exception. The most troublesome is `idivl`; see the X86-64 handout for details, and look at the output of the reference generator to see one (quite inefficient) way of coping with the associated problems.

The skeleton code generator implements a fairly simple algorithm for converting three-address IR `add` instructions to two-operand machine code. You can use this as the basis for generating `sub` and `imul` instructions as well. But note that the two-operand form of `imull` requires a *register* as its destination operand. There are also a number of ways in which the skeleton's approach could be improved.

Extra Credit

Almost every aspect of the reference generator's output code could be improved. For extra credit, pick one or more aspects that you would like to improve, identify and document a strategy for doing so, and implement the strategy in your submission. Consulting with the instructor first might be wise.

Submitting the Program

Place your `X86Gen` class in the single file `X86Gen.java`, and mail it to `cs322-01@cs.pdx.edu`. You must mail this file as a plain text *attachment*; the contents of the message itself don't matter. If you are working in a team of two, only one team member should submit a solution, which must have the names of both team members in a comment at the top; the other team member should send mail identifying him- or her-self as a team member.