```
1  // Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas
2
3  package jminusminus;
4
5  import static jminusminus.CLConstants.*;
6  import java.io.BufferedReader;
7  import java.io.FileReader;
8  import java.io.File;
9  import java.io.FileNotFoundException;
10 import java.io.IOException;
11 import java.io.PrintWriter;
12 import java.util.ArrayList;
13 import java.util.Calendar;
14 import java.util.HashMap;
15
16 /**
17  * A class for generating native SPIM code.
18  */
19
20 public class NEmitter {
21
22     /** Source program file name. */
23     private String sourceFile;
24
25     /**
26      * Map of maps, one per class in the compilation unit. Each one of them maps
27      * methods of a class to their control flow graph.
28      */
29     private HashMap<CLFile, HashMap<CLMethodInfo, NControlFlowGraph>> classes;
30
31     /** Destination directory for the native SPIM code. */
32     private String destDir;
33
34     /**
35      * Whether an error occurred while creating/writing SPIM code.
36      */
37     private boolean errorHasOccurred;
38
39     /**
40      * Report any error that occurs while creating/writing the spim file, to
41      * STDERR.
42      *
43      * @param message
44      *            message identifying the error.
45      * @param args
46      *            related values.
47      */
48
49     private void reportEmitterError(String message, Object... args) {
50         System.err.printf(message, args);
51         System.err.println();
52         errorHasOccurred = true;
53     }
54
55     /**
56      * Emits SPIM code to setup a stack frame for the procedure denoted by cfg.
57      * This involves saving the return address (ra), saving the frame pointer
58      * (fp), saving any physical registers (t0, ..., t9, s0, ..., s7) used by
59      * the procedure, and setting up the new value for fp (i.e. pushing a stack
60      * frame).
61      *
62      * @param cfg
63      *            the control flow graph instance.
64      * @param out
65      *            output stream for SPIM code.
66      */
```

```
67
68      private void pushStackFrame(NControlFlowGraph cfg, PrintWriter out) {
69          int frameSize = cfg.pRegisters.size() * 4 + cfg.offset * 4 + 8;
70          out.printf(
71                  "    subu    $sp,$sp,%d \t # Stack frame is %d bytes long\n",
72                  frameSize, frameSize);
73          out.printf("    sw      $ra,%d($sp) \t # Save return address\n",
74                  frameSize - 4);
75          out.printf("    sw      $fp,%d($sp) \t # Save frame pointer\n",
76                  frameSize - 8);
77          int i = 12;
78          for (NPhysicalRegister pRegister : cfg.pRegisters) {
79              out.printf("    sw      %s,%d($sp) \t # Save register %s\n",
80                      pRegister, frameSize - i, pRegister);
81              i += 4;
82          }
83          out.printf("    addiu   $fp,$sp,%d \t # Save frame pointer\n",
84                  frameSize - 4);
85          out.println();
86      }
87
88      /**
89       * Emits SPIM code to pop the stack frame that was setup for the procedure
90       * denoted by cfg. This involves restoring the return address (ra), the
91       * frame pointer (fp), any physical registers (t0, ..., t9, s0, ..., s7)
92       * used by the procedure, setting fp to the restored value (i.e. popping the
93       * stack frame), and finally jumping to ra (the caller).
94       *
95       * @param cfg
96       *            the control flow graph instance
97       * @param out
98       *            output stream for SPIM code.
99       */
100
101     private void popStackFrame(NControlFlowGraph cfg, PrintWriter out) {
102         int frameSize = cfg.pRegisters.size() * 4 + cfg.offset * 4 + 8;
103         out.printf("%s.restore:\n", cfg.labelPrefix);
104         out.printf("    lw      $ra,%d($sp) \t # Restore return address\n",
105                 frameSize - 4);
106         out.printf("    lw      $fp,%d($sp) \t # Restore frame pointer\n",
107                 frameSize - 8);
108         int i = 12;
109         for (NPhysicalRegister pRegister : cfg.pRegisters) {
110             out.printf("    lw      %s,%d($sp) \t # Restore register %s\n",
111                     pRegister, frameSize - i, pRegister);
112             i += 4;
113         }
114         out.printf("    addiu   $sp,$sp,%d \t # Pop stack\n", frameSize);
115         out.printf("    jr      $ra \t # Return to caller\n", frameSize);
116         out.println();
117     }
118
119     /**
120      * Construct an NEmitter instance.
121      *
122      * @param sourceFile
123      *            the source j-- program file name.
124      * @param clFiles
125      *            list of CLFile objects.
126      * @param ra
127      *            register allocation scheme (naive, linear, or graph).
128      */
129
130     public NEmitter(String sourceFile, ArrayList<CLFile> clFiles, String ra) {
131         this.sourceFile = sourceFile.substring(sourceFile
132                 .lastIndexOf(File.separator) + 1);
133         classes = new HashMap<CLFile, HashMap<CLMethodInfo,
NControlFlowGraph>>();
134         for (CLFile clFile : clFiles) {
```

```
            CLConstantPool cp = clFile.constantPool;
            HashMap<CLMethodInfo, NControlFlowGraph> methods = new
HashMap<CLMethodInfo, NControlFlowGraph>();
            for (int i = 0; i < clFile.methodsCount; i++) {
                CLMethodInfo m = clFile.methods.get(i);

                // Build a control flow graph (cfg) for this method.
                // Each block in the cfg, at the end of this step,
                // has the JVM bytecode translated into tuple
                // representation.
                NControlFlowGraph cfg = new NControlFlowGraph(cp, m);

                // Write the tuples in cfg to STDOUT.
                PrettyPrinter p = new PrettyPrinter();
                p.printf("%s %s\n", cfg.name, cfg.desc);
                cfg.writeTuplesToStdOut(p);

                // Identify blocks in cfg that are loop heads and
                // loop tails. Also, compute number of backward
                // branches to blocks.
                cfg.detectLoops(cfg.basicBlocks.get(0), null);

                // Remove unreachable blocks from cfg.
                cfg.removeUnreachableBlocks();

                // Compute the dominator of each block in the cfg.
                cfg.computeDominators(cfg.basicBlocks.get(0), null);

                // Convert the tuples in each block in the cfg to
                // high-level (HIR) instructions.
                cfg.tuplesToHir();

                // Eliminate redundant phi functions, i.e., replace
                // phi functions of the form x = (y, x, x, ..., x)
                // with y.
                cfg.eliminateRedundantPhiFunctions();

                // Perform optimizations on the high-level
                // instructions.
                cfg.optimize();

                // Write the HIR instructions in cfg to STDOUT.
                cfg.writeHirToStdOut(p);

                // Convert the HIR instructions in each block in the
                // cfg to low-level (LIR) instructions.
                cfg.hirToLir();

                // Resolve phi functions;
                cfg.resolvePhiFunctions();

                // Compute block order.
                cfg.orderBlocks();

                // Assign new ids to LIR instructions.
                cfg.renumberLirInstructions();

                // Write the LIR instructions in cfg to STDOUT.
                cfg.writeLirToStdOut(p);

                // Save the cfg for the method in a map keyed in by
                // the CLMethodInfo object for the method.
                methods.put(m, cfg);

                // Perform register allocation.
                NRegisterAllocator regAllocator;
                if (ra.equals("naive")) {
                    regAllocator = new NNaiveRegisterAllocator(cfg);
                } else if (ra.equals("linear")) {
```

```
                    regAllocator = new NLinearRegisterAllocator(cfg);
                } else {
                    regAllocator = new NGraphRegisterAllocator(cfg);
                }
                regAllocator.allocation();

                // Write the intervals in cfg to STDOUT.
                cfg.writeIntervalsToStdOut(p);

                // Replace references to virtual registers in LIR
                // instructions with references to physical registers.
                cfg.allocatePhysicalRegisters();

                // Write the LIR instructions in cfg to STDOUT.
                cfg.writeLirToStdOut(p);
            }

            // Store the cfgs for the methods in this class in a map.
            classes.put(clFile, methods);
        }
    }

    /**
     * Set the destination directory for the SPIM files to the specified value.
     *
     * @param destDir
     *            destination directory.
     */

    public void destinationDir(String destDir) {
        this.destDir = destDir;
    }

    /**
     * Has an emitter error occurred up to now?
     *
     * @return true or false.
     */

    public boolean errorHasOccurred() {
        return errorHasOccurred;
    }

    /**
     * Write out SPIM file(s) to the file system. The destination directory for
     * the files can be set using the destinationDir(String dir) method.
     */

    public void write() {
        String file = "";
        try {
            file = destDir + File.separator + sourceFile.replace(".java", ".s");
            PrintWriter out = new PrintWriter(file);

            // Header.
            out.printf("# %s\n", file);
            out.printf("# Source file: %s\n", sourceFile);
            out.printf("# Compiled: %s\n\n", Calendar.getInstance().getTime()
                    .toString());

            // Translate classes and their methods to SPIM.
            for (CLFile clFile : classes.keySet()) {
                HashMap<CLMethodInfo, NControlFlowGraph> aClass = classes
                        .get(clFile);
                CLConstantPool cp = clFile.constantPool;
                int nameIndex = ((CLConstantClassInfo) cp
                        .cpItem(clFile.thisClass)).nameIndex;
                String className = new String(((CLConstantUtf8Info) cp
                        .cpItem(nameIndex)).b);
```

```
                for (CLMethodInfo m : aClass.keySet()) {
                    NControlFlowGraph cfg = aClass.get(m);
                    String methodName = cfg.name;
                    String methodDesc = cfg.desc;
                    if (methodName.equals("<init>")) {
                        continue;
                    }
                    out.printf(".text\n\n");
                    if (methodName.equals("main")
                            && methodDesc.equals("([Ljava/lang/String;)V")) {
                        out.printf("%s:\n", methodName);
                        cfg.labelPrefix = methodName;
                    } else {
                        out.printf("%s.%s:\n", className, methodName);
                        cfg.labelPrefix = className + "." + methodName;
                    }

                    // Setup stack frame for this method
                    pushStackFrame(cfg, out);

                    for (NBasicBlock block : cfg.basicBlocks) {
                        out.printf("%s.%d:\n", cfg.labelPrefix, block.id);
                        for (NLIRInstruction lir : block.lir) {
                            lir.toSpim(out);
                        }
                        out.printf("\n");
                    }

                    // Pop the stack frame for this method
                    popStackFrame(cfg, out);

                    // Data segment for this cfg storing string
                    // literals.
                    if (cfg.data.size() > 0) {
                        out.printf(".data\n\n");
                        for (String line : cfg.data) {
                            out.printf(line);
                        }
                    }

                    out.printf("\n\n");
                }
            }

            // Emit SPIM runtime code; just SPIM.s for now.
            String[] libs = { "SPIM.s" };
            out.printf("# SPIM Runtime\n\n");
            for (String lib : libs) {
                file = System.getenv("j") + File.separator + "src"
                        + File.separator + "spim" + File.separator + lib;
                BufferedReader in = new BufferedReader(new FileReader(file));
                String line;
                while ((line = in.readLine()) != null) {
                    out.printf("%s\n", line);
                }
                in.close();
            }

            out.close();
        } catch (FileNotFoundException e) {
            reportEmitterError("File %s not found", file);
        } catch (IOException e) {
            reportEmitterError("Cannot write to file %s", file);
        }
    }

}
```