```
// Copyright 2013 Bill Campbell, Swami Iyer and Bahar Akbal-Delibas

package jminusminus;

import static jminusminus.CLConstants.*;
import static jminusminus.NPhysicalRegister.*;
import java.io.PrintWriter;
import java.util.ArrayList;

/**
 * Low-level intermediate representation (LIR) of a JVM instruction.
 */

abstract class NLIRInstruction {

    /**
     * Maps JVM opcode to a string mnemonic for LIR instructions. For example,
     * imul is mapped to the "MUL".
     */
    protected static String[] lirMnemonic;
    static {
        lirMnemonic = new String[256];
        lirMnemonic[IADD] = "ADD";
        lirMnemonic[IMUL] = "MUL";
        lirMnemonic[ISUB] = "SUB";
        lirMnemonic[MULTIANEWARRAY] = "MULTIANEWARRAY";
        lirMnemonic[AALOAD] = "AALOAD";
        lirMnemonic[IALOAD] = "IALOAD";
        lirMnemonic[IASTORE] = "IASTORE";
        lirMnemonic[IF_ICMPNE] = "NE";
        lirMnemonic[IF_ICMPGT] = "GT";
        lirMnemonic[IF_ICMPLE] = "LE";
        lirMnemonic[GETSTATIC] = "GETSTATIC";
        lirMnemonic[PUTSTATIC] = "PUTSTATIC";
        lirMnemonic[INVOKESPECIAL] = "INVOKESPECIAL";
        lirMnemonic[INVOKESTATIC] = "INVOKESTATIC";
    }

    /** The block containing this instruction. */
    public NBasicBlock block;

    /** Unique identifier of this instruction. */
    public int id;

    /** Registers that store the inputs (if any) of this instruction. */
    public ArrayList<NRegister> reads;

    /**
     * Register that stores the result (if any) of this instruction.
     */
    public NRegister write;

    /**
     * Construct an NLIRInstruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     */

    protected NLIRInstruction(NBasicBlock block, int id) {
        this.block = block;
        this.id = id;
        reads = new ArrayList<NRegister>();
    }
```

```
67
68      /**
69       * Replace references to virtual registers in this LIR instruction with
70       * references to physical registers.
71       */
72
73      public void allocatePhysicalRegisters() {
74          // nothing here.
75      }
76
77      /**
78       * Translate this LIR instruction into SPIM and write it out to the
79       * specified output stream.
80       *
81       * @param out
82       *             output stream for SPIM code.
83       */
84
85      public void toSpim(PrintWriter out) {
86          // nothing here.
87      }
88
89      /**
90       * Return a string representation of this instruction.
91       *
92       * @return string representation of this instruction.
93       */
94
95      public String toString() {
96          return "" + id;
97      }
98
99  }
100
101 /**
102  * LIR instruction corresponding to the JVM arithmetic instructions.
103  */
104
105 class NLIRArithmetic extends NLIRInstruction {
106
107     /** Opcode for the arithmetic operator. */
108     private int opcode;
109
110     /**
111      * Construct an NLIRArithmetic instruction.
112      *
113      * @param block
114      *             enclosing block.
115      * @param id
116      *             identifier of the instruction.
117      * @param opcode
118      *             opcode for the arithmetic operator.
119      * @param lhs
120      *             LIR for lhs.
121      * @param rhs
122      *             LIR for rhs.
123      */
124
125     public NLIRArithmetic(NBasicBlock block, int id, int opcode,
126             NLIRInstruction lhs, NLIRInstruction rhs) {
127         super(block, id);
128         this.opcode = opcode;
129         reads.add(lhs.write);
130         reads.add(rhs.write);
131         write = new NVirtualRegister(NControlFlowGraph.regId++, "I", "I");
132         block.cfg.registers.add((NVirtualRegister) write);
133     }
134
135     /**
```

```
136      * @inheritDoc
137      */
138
139     public void allocatePhysicalRegisters() {
140         NInterval input1 = block.cfg.intervals.get(reads.get(0).number())
141                 .childAt(id);
142         NInterval input2 = block.cfg.intervals.get(reads.get(1).number())
143                 .childAt(id);
144         NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
145         reads.set(0, input1.pRegister);
146         reads.set(1, input2.pRegister);
147         write = output.pRegister;
148     }
149
150     /**
151      * @inheritDoc
152      */
153
154     public void toSpim(PrintWriter out) {
155         switch (opcode) {
156         case IADD:
157             out.printf("    add %s,%s,%s\n", write, reads.get(0), reads.get(1));
158             break;
159         case ISUB:
160             out.printf("    sub %s,%s,%s\n", write, reads.get(0), reads.get(1));
161             break;
162         case IMUL:
163             out.printf("    mul %s,%s,%s\n", write, reads.get(0), reads.get(1));
164             break;
165         }
166     }
167
168     /**
169      * @inheritDoc
170      */
171
172     public String toString() {
173         return id + ": " + lirMnemonic[opcode] + " " + reads.get(0) + " "
174                 + reads.get(1) + " " + write;
175     }
176
177 }
178
179 /**
180  * LIR instruction corresponding to the JVM instructions representing integer
181  * constants.
182  */
183
184 class NLIRIntConstant extends NLIRInstruction {
185
186     /** The constant int value. */
187     public int value;
188
189     /**
190      * Construct an NLIRIntConstant instruction.
191      *
192      * @param block
193      *             enclosing block.
194      * @param id
195      *             identifier of the instruction.
196      * @param value
197      *             the constant int value.
198      */
199
200     public NLIRIntConstant(NBasicBlock block, int id, int value) {
201         super(block, id);
202         this.value = value;
203         write = new NVirtualRegister(NControlFlowGraph.regId++, "I", "I");
204         block.cfg.registers.add((NVirtualRegister) write);
```

```
205     }
206
207     /**
208      * @inheritDoc
209      */
210
211     public void allocatePhysicalRegisters() {
212         NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
213         write = output.pRegister;
214     }
215
216     /**
217      * @inheritDoc
218      */
219
220     public void toSpim(PrintWriter out) {
221         out.printf("    li %s,%d\n", write, value);
222     }
223
224     /**
225      * @inheritDoc
226      */
227
228     public String toString() {
229         return id + ": LDC [" + value + "] " + write;
230     }
231
232 }
233
234 /**
235  * LIR instruction corresponding to the JVM instructions representing string
236  * constants.
237  */
238
239 class NLIRStringConstant extends NLIRInstruction {
240
241     /** The constant string value. */
242     public String value;
243
244     /** */
245     private static int labelSuffix;
246
247     /**
248      * Construct an NHIRStringConstant instruction.
249      *
250      * @param block
251      *             enclosing block.
252      * @param id
253      *             identifier for the instruction.
254      * @param value
255      *             the constant string value.
256      */
257
258     public NLIRStringConstant(NBasicBlock block, int id, String value) {
259         super(block, id);
260         this.value = value;
261         write = new NVirtualRegister(NControlFlowGraph.regId++, "L",
262                 "Ljava/lang/String;");
263         block.cfg.registers.add((NVirtualRegister) write);
264         labelSuffix = 0;
265     }
266
267     /**
268      * Create a label for LIR code.
269      *
270      * @return the Label.
271      */
272
273     private String createLabel() {
```

```
274         return "Constant..String" + labelSuffix++;
275     }
276
277     /**
278      * @inheritDoc
279      */
280
281     public void allocatePhysicalRegisters() {
282         NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
283         write = output.pRegister;
284     }
285
286     /**
287      * @inheritDoc
288      */
289
290     public void toSpim(PrintWriter out) {
291         String label = createLabel();
292         String s = label + ":\n";
293         int size = 12 + value.length() + 1;
294         int align = (size % 4 == 0) ? 0 : (size + 4) / 4 * 4 - size;
295         s += "    .word 2 # Tag 2 indicates a string\n";
296         s += "    .word " + (size + align) + " # Size of object in bytes\n";
297         s += "    .word " + value.length()
298                 + " # String length (not including null terminator)\n";
299         s += "    .asciiz \"" + value
300                 + "\" # String terminated by null character 0\n";
301         s += "    .align " + align + " # Next object is on a word boundary\n";
302         block.cfg.data.add(s);
303         out.println("    la %s,%s" + write, label);
304     }
305
306     /**
307      * @inheritDoc
308      */
309
310     public String toString() {
311         return id + ": LDC [" + value + "] " + write;
312     }
313
314 }
315
316 /**
317  * LIR instruction representing an conditional jump instructions in JVM.
318  */
319
320 class NLIRConditionalJump extends NLIRInstruction {
321
322     /** Test expression opcode. */
323     public int opcode;
324
325     /** Block to jump to on true. */
326     public NBasicBlock onTrueDestination;
327
328     /** Block to jump to on false. */
329     public NBasicBlock onFalseDestination;
330
331     /**
332      * Construct an NLIRConditionalJump instruction.
333      *
334      * @param block
335      *            enclosing block.
336      * @param id
337      *            identifier of the instruction.
338      * @param lhs
339      *            lhs LIR.
340      * @param rhs
341      *            rhs LIR.
342      * @param opcode
```

```
     *            opcode in the test.
     * @param onTrueDestination
     *            block to jump to on true.
     * @param onFalseDestination
     *            block to jump to on false.
     */

    public NLIRConditionalJump(NBasicBlock block, int id, NLIRInstruction lhs,
            NLIRInstruction rhs, int opcode, NBasicBlock onTrueDestination,
            NBasicBlock onFalseDestination) {
        super(block, id);
        this.opcode = opcode;
        reads.add(lhs.write);
        reads.add(rhs.write);
        this.onTrueDestination = onTrueDestination;
        this.onFalseDestination = onFalseDestination;
    }

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        NInterval input1 = block.cfg.intervals.get(reads.get(0).number())
                .childAt(id);
        NInterval input2 = block.cfg.intervals.get(reads.get(1).number())
                .childAt(id);
        reads.set(0, input1.pRegister);
        reads.set(1, input2.pRegister);
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        switch (opcode) {
        case IF_ICMPNE:
            out.printf("    bne %s,%s,%s\n", reads.get(0), reads.get(1),
                    block.cfg.labelPrefix + "." + onTrueDestination.id);
            break;
        case IF_ICMPGT:
            out.printf("    bgt %s,%s,%s\n", reads.get(0), reads.get(1),
                    block.cfg.labelPrefix + "." + onTrueDestination.id);
            break;
        case IF_ICMPLE:
            out.printf("    ble %s,%s,%s\n", reads.get(0), reads.get(1),
                    block.cfg.labelPrefix + "." + onTrueDestination.id);
            break;
        }
        out.printf("    j %s\n", block.cfg.labelPrefix + "."
                + onFalseDestination.id);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": BRANCH [" + lirMnemonic[opcode] + "] " + reads.get(0)
                + " " + reads.get(1) + " " + onTrueDestination.id();
    }

}

/**
 * LIR instruction representing an unconditional jump instruction in JVM.
 */
```

```
412 class NLIRGoto extends NLIRInstruction {
413
414     /** The destination block to unconditionally jump to. */
415     private NBasicBlock destination;
416
417     /**
418      * Construct an NLIRGoto instruction.
419      *
420      * @param block
421      *             enclosing block.
422      * @param id
423      *             identifier of the instruction.
424      * @param destination
425      *             the block to jump to.
426      */
427
428     public NLIRGoto(NBasicBlock block, int id, NBasicBlock destination) {
429         super(block, id);
430         this.destination = destination;
431     }
432
433     /**
434      * @inheritDoc
435      */
436
437     public void toSpim(PrintWriter out) {
438         String label = block.cfg.labelPrefix + "." + destination.id;
439         out.printf("    j %s\n", label);
440     }
441
442     /**
443      * @inheritDoc
444      */
445
446     public String toString() {
447         return id + ": BRANCH " + destination.id();
448     }
449
450 }
451
452 /**
453  * LIR instruction representing method invocation instructions in JVM.
454  */
455
456 class NLIRInvoke extends NLIRInstruction {
457
458     /** Opcode of the JVM instruction. */
459     public int opcode;
460
461     /** Target for the method. */
462     public String target;
463
464     /** Name of the method being invoked. */
465     public String name;
466
467     /**
468      * Construct an NHIRInvoke instruction.
469      *
470      * @param block
471      *             enclosing block.
472      * @param id
473      *             identifier of the instruction.
474      * @param opcode
475      *             opcode of the JVM instruction.
476      * @param target
477      *             target of the method.
478      * @param name
479      *             name of the method.
480      * @param arguments
```

```
481      *            list of register storing the of arguments for the method.
482      * @param sType
483      *            return type (short name) of the method.
484      * @param lType
485      *            return type (long name) of the method.
486      */
487
488     public NLIRInvoke(NBasicBlock block, int id, int opcode, String target,
489             String name, ArrayList<NRegister> arguments, String sType,
490             String lType) {
491         super(block, id);
492         this.opcode = opcode;
493         this.target = target;
494         this.name = name;
495         for (NRegister arg : arguments) {
496             reads.add(arg);
497         }
498         if (!sType.equals("V")) {
499             write = NPhysicalRegister.regInfo[V0];
500             block.cfg.registers.set(V0, write);
501         }
502     }
503
504     /**
505      * @inheritDoc
506      */
507
508     public void allocatePhysicalRegisters() {
509         for (int i = 0; i < reads.size(); i++) {
510             NRegisterAll input = block.cfg.intervals.get(reads.get(i).number())
511                     .childAt(id);
512             reads.set(i, input.pRegister);
513         }
514     }
515
516     /**
517      * @inheritDoc
518      */
519
520     public void toSpim(PrintWriter out) {
521         out.printf("    jal %s.%s\n", target.replace("/", "."), name
522                 .equals("<init>") ? "__init__" : name);
523     }
524
525     /**
526      * @inheritDoc
527      */
528
529     public String toString() {
530         String s = id + ": " + lirMnemonic[opcode] + " "
531                 + (write != null ? write + " = " : "") + target + "." + name
532                 + "( ";
533         for (NRegister input : reads) {
534             s += input + " ";
535         }
536         s += ")";
537         return s;
538     }
539
540 }
541
542 /**
543  * HIR instruction representing a JVM return instruction.
544  */
545
546 class NLIRReturn extends NLIRInstruction {
547
548     /** JVM opcode for the return instruction. */
549     public int opcode;
```

```
550
551     /**
552      * Construct an NLIRReturn instruction.
553      *
554      * @param block
555      *             enclosing block.
556      * @param id
557      *             identifier of the instruction.
558      * @param opcode
559      *             JVM opcode for the return instruction.
560      * @param result
561      *             physical register storing return value, or null.
562      */
563
564     public NLIRReturn(NBasicBlock block, int id, int opcode,
565             NPhysicalRegister result) {
566         super(block, id);
567         this.opcode = opcode;
568         if (result != null) {
569             reads.add(result);
570         }
571     }
572
573     /**
574      * @inheritDoc
575      */
576
577     public void toSpim(PrintWriter out) {
578         out.printf("    j %s\n", block.cfg.labelPrefix + "_restore");
579     }
580
581     /**
582      * @inheritDoc
583      */
584
585     public String toString() {
586         if (reads.size() == 0) {
587             return id + ": RETURN";
588         }
589         return id + ": RETURN " + reads.get(0);
590     }
591
592 }
593
594 /**
595  * LIR instruction representing JVM (put) field instructions.
596  */
597
598 class NLIRPutField extends NLIRInstruction {
599
600     /** Opcode of the JVM instruction. */
601     public int opcode;
602
603     /** Target for the field. */
604     public String target;
605
606     /** Name of the field being accessed. */
607     public String name;
608
609     /**
610      * Construct an NLIRPutField instruction.
611      *
612      * @param block
613      *             enclosing block.
614      * @param id
615      *             identifier of the instruction.
616      * @param opcode
617      *             JVM opcode for the return instruction.
618      * @param target
```

```
     *            target for the field.
     * @param name
     *            name of the field.
     * @param sType
     *            type (short name) of the field.
     * @param lType
     *            type (long name) of the field.
     * @param value
     *            LIR of the value of the field.
     */

    public NLIRPutField(NBasicBlock block, int id, int opcode, String target,
            String name, String sType, String lType, NLIRInstruction value) {
        super(block, id);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
        reads.add(value.write);
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    NLIRPutField.toSpim() not yet implemented!\n");
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": " + lirMnemonic[opcode] + " " + target + "." + name
                + " = " + reads.get(0);
    }

}

/**
 * LIR instruction representing JVM (get) field instructions.
 */

class NLIRGetField extends NLIRInstruction {

    /** Opcode of the JVM instruction. */
    public int opcode;

    /** Target for the field. */
    public String target;

    /** Name of the field being accessed. */
    public String name;

    /**
     * Construct an NLIRGetField instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the return instruction.
     * @param target
     *            target for the field.
     * @param name
     *            name of the field.
     * @param sType
     *            type (short name) of the field.
```

```
     * @param lType
     *            type (long name) of the field.
     */

    public NLIRGetField(NBasicBlock block, int id, int opcode, String target,
            String name, String sType, String lType) {
        super(block, id);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
        write = new NVirtualRegister(NControlFlowGraph.regId++, sType, lType);
        block.cfg.registers.add((NVirtualRegister) write);
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    NLIRGetField.toSpim() not yet implemented!\n");
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": " + lirMnemonic[opcode] + " " + write + " = " + target
                + "." + name;
    }

}

/**
 * LIR instruction representing JVM array creation instructions.
 */

class NLIRNewArray extends NLIRInstruction {

    /** Opcode of the JVM instruction. */
    public int opcode;

    /** Dimension of the array. */
    public int dim;

    /**
     * Construct an NLIRNewArray instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param dim
     *            dimension of the array.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

    public NLIRNewArray(NBasicBlock block, int id, int opcode, int dim,
            String sType, String lType) {
        super(block, id);
        this.opcode = opcode;
        this.dim = dim;
        write = new NVirtualRegister(NControlFlowGraph.regId++, sType, lType);
        block.cfg.registers.add((NVirtualRegister) write);
```

```
757     }
758
759     /**
760      * @inheritDoc
761      */
762
763     public void toSpim(PrintWriter out) {
764         out.printf("    NLIRNewArray.toSpim() not yet implemented!\n");
765     }
766
767     /**
768      * @inheritDoc
769      */
770
771     public String toString() {
772         return id + ": " + lirMnemonic[opcode] + " [" + dim + "]" + " " + write;
773     }
774
775 }
776
777 /**
778  * LIR instruction representing JVM array load instructions.
779  */
780
781 class NLIRALoad extends NLIRInstruction {
782
783     /** Opcode of the JVM instruction. */
784     public int opcode;
785
786     /**
787      * Construct an NLIRALoad instruction.
788      *
789      * @param block
790      *            enclosing block.
791      * @param id
792      *            identifier of the instruction.
793      * @param opcode
794      *            JVM opcode for the instruction.
795      * @param arrayRef
796      *            LIR of the array reference.
797      * @param index
798      *            LIR of the array index.
799      * @param sType
800      *            type (short name) of the array.
801      * @param lType
802      *            type (long name) of the array.
803      */
804
805     public NLIRALoad(NBasicBlock block, int id, int opcode,
806             NLIRInstruction arrayRef, NLIRInstruction index, String sType,
807             String lType) {
808         super(block, id);
809         this.opcode = opcode;
810         reads.add(arrayRef.write);
811         reads.add(index.write);
812         write = new NVirtualRegister(NControlFlowGraph.regId++, sType, lType);
813         block.cfg.registers.add((NVirtualRegister) write);
814     }
815
816     /**
817      * @inheritDoc
818      */
819
820     public void toSpim(PrintWriter out) {
821         out.printf("    NLIRALoad.toSpim() not yet implemented!\n");
822     }
823
824     /**
825      * @inheritDoc
```

```
826      */
827
828     public String toString() {
829         return id + ": " + lirMnemonic[opcode] + " " + write + "= "
830                 + reads.get(0) + "[" + reads.get(1) + "]";
831     }
832
833 }
834
835 /**
836  * LIR instruction representing JVM array store instructions.
837  */
838
839 class NLIRAStore extends NLIRInstruction {
840
841     /** Opcode of the JVM instruction. */
842     public int opcode;
843
844     /**
845      * Construct an NLIRAStore instruction.
846      *
847      * @param block
848      *             enclosing block.
849      * @param id
850      *             identifier of the instruction.
851      * @param opcode
852      *             JVM opcode for the instruction.
853      * @param arrayRef
854      *             LIR of the array reference.
855      * @param index
856      *             LIR of the array index.
857      * @param value
858      *             LIR of the value to store.
859      * @param sType
860      *             type (short name) of the array.
861      * @param lType
862      *             type (long name) of the array.
863      */
864
865     public NLIRAStore(NBasicBlock block, int id, int opcode,
866             NLIRInstruction arrayRef, NLIRInstruction index,
867             NLIRInstruction value, String sType, String lType) {
868         super(block, id);
869         this.opcode = opcode;
870         reads.add(arrayRef.write);
871         reads.add(index.write);
872         reads.add(value.write);
873     }
874
875     /**
876      * @inheritDoc
877      */
878
879     public void toSpim(PrintWriter out) {
880         out.printf("    NLIRAStore.toSpim() not yet implemented!\n");
881     }
882
883     /**
884      * @inheritDoc
885      */
886
887     public String toString() {
888         return id + ": " + lirMnemonic[opcode] + " " + reads.get(0) + "["
889                 + reads.get(1) + "] = " + reads.get(2);
890     }
891
892 }
893
894 /**
```

```
 * LIR instruction representing phi functions.
 */

class NLIRPhiFunction extends NLIRInstruction {

    /**
     * Construct an NLIRPhiFunction instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param sType
     *            type (short name) of the phi function.
     * @param lType
     *            type (long name) of the phi function.
     */

    public NLIRPhiFunction(NBasicBlock block, int id, String sType, String lType)
{
        super(block, id);
        write = new NVirtualRegister(NControlFlowGraph.regId++, sType, lType);
        block.cfg.registers.add((NVirtualRegister) write);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": phi " + write;
    }

}

/**
 * LIR move instruction.
 */

class NLIRMove extends NLIRInstruction {

    /**
     * Construct an NLIRMove instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param from
     *            LIR to move from.
     * @param to
     *            LIR to move to.
     */

    public NLIRMove(NBasicBlock block, int id, NLIRInstruction from,
            NLIRInstruction to) {
        super(block, id);
        reads.add(from.write);
        write = to.write;
    }

    /**
     * Construct an NLIRMove instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param from
```

```
963      *            register (virtual or physical) to move from.
964      * @param to
965      *            register (virtual or physical) to move to.
966      */
967
968     public NLIRMove(NBasicBlock block, int id, NRegister from, NRegister to) {
969         super(block, id);
970         reads.add(from);
971         write = to;
972     }
973
974     /**
975      * @inheritDoc
976      */
977
978     public void allocatePhysicalRegisters() {
979         NInterval input = block.cfg.intervals.get(reads.get(0).number())
980                 .childAt(id);
981         ;
982         NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
983         reads.set(0, input.pRegister);
984         write = output.pRegister;
985     }
986
987     /**
988      * @inheritDoc
989      */
990
991     public void toSpim(PrintWriter out) {
992         out.println("    mov %s,%s", write, reads.get(0));
993     }
994
995     /**
996      * @inheritDoc
997      */
998
999     public String toString() {
1000         return id + ": MOVE " + reads.get(0) + " " + write;
1001     }
1002
1003}
1004
1005/**
1006 * LIR instruction representing a formal parameter.
1007 */
1008
1009class NLIRLoadLocal extends NLIRInstruction {
1010
1011     /** Local variable index. */
1012     public int local;
1013
1014     /**
1015      * Construct an NLIRLoadLocal instruction.
1016      *
1017      * @param block
1018      *            enclosing block.
1019      * @param id
1020      *            identifier of the instruction.
1021      * @param local
1022      *            local variable index.
1023      * @param sType
1024      *            short type name of the instruction.
1025      * @param lType
1026      *            long type name of the instruction.
1027      */
1028
1029     public NLIRLoadLocal(NBasicBlock block, int id, int local, String sType,
1030             String lType) {
1031         super(block, id);
```

```
1032        this.local = local;
1033        if (local < 4) {
1034            write = NPhysicalRegister.regInfo[A0 + local];
1035            block.cfg.registers.set(A0 + local, NPhysicalRegister.regInfo[A0
1036                    + local]);
1037        } else {
1038            write = new NVirtualRegister(NControlFlowGraph.regId++, sType,
1039                    lType);
1040            block.cfg.registers.add((NVirtualRegister) write);
1041        }
1042    }
1043
1044    /**
1045     * @inheritDoc
1046     */
1047
1048    public String toString() {
1049        return id + ": LDLOC " + local + " " + write;
1050    }
1051
1052}
1053
1054/**
1055 * LIR instruction representing a load from memory to register.
1056 */
1057
1058class NLIRLoad extends NLIRInstruction {
1059
1060    /** Stack offset to load from. */
1061    private int offset;
1062
1063    /**
1064     * Whether offset is relative to stack pointer (sp) or frame pointer (fp).
1065     */
1066    private OffsetFrom offsetFrom;
1067
1068    /** Register to load to. */
1069    private NRegister register;
1070
1071    /**
1072     * Construct an NLIRLoad instruction.
1073     *
1074     * @param block
1075     *            enclosing block.
1076     * @param id
1077     *            identifier of the instruction.
1078     * @param offset
1079     *            stack offset to load from.
1080     * @param offsetFrom
1081     *            whether offset relative to stack pointer (sp) or frame pointer
1082     *            (fp).
1083     * @param register
1084     *            register to load to.
1085     */
1086
1087    public NLIRLoad(NBasicBlock block, int id, int offset,
1088            OffsetFrom offsetFrom, NRegister register) {
1089        super(block, id);
1090        this.offset = offset;
1091        this.offsetFrom = offsetFrom;
1092        this.register = register;
1093    }
1094
1095    /**
1096     * @inheritDoc
1097     */
1098
1099    public void toSpim(PrintWriter out) {
1100        if (offsetFrom == OffsetFrom.FP) {
```

```
            out.printf("    lw %s,%d($fp)\n", register, offset * 4);
        } else {
            out.printf("    lw %s,%d($sp)\n", register, offset * 4);
        }
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": LOAD "
                + (offsetFrom == OffsetFrom.FP ? "[frame:" : "[stack:")
                + offset + "] " + register;
    }

}

/**
 * LIR instruction representing a store from a register to memory.
 */

class NLIRStore extends NLIRInstruction {

    /** Stack offset to store to. */
    private int offset;

    /**
     * Whether offset is relative to stack pointer (sp) or frame pointer (fp).
     */
    private OffsetFrom offsetFrom;

    /** Register to store from. */
    private NRegister register;

    /**
     * Construct an NLIRStore instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param offset
     *            stack offset to store to.
     * @param offsetFrom
     *            whether offset relative to stack pointer (sp) or frame pointer
     *            (fp).
     * @param register
     *            register to store from.
     */

    public NLIRStore(NBasicBlock block, int id, int offset,
            OffsetFrom offsetFrom, NRegister register) {
        super(block, id);
        this.offset = offset;
        this.offsetFrom = offsetFrom;
        this.register = register;
        reads.add(register);
    }

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        NInterval input = block.cfg.intervals.get(reads.get(0).number())
                .childAt(id);
        if (input.vRegId >= 32) {
            reads.set(0, input.pRegister);
```

```
1170        }
1171    }
1172
1173    /**
1174     * @inheritDoc
1175     */
1176
1177    public void toSpim(PrintWriter out) {
1178        if (offsetFrom == OffsetFrom.FP) {
1179            out.printf("    sw %s,%d($fp)\n", reads.get(0), offset * 4);
1180        } else {
1181            out.printf("    sw %s,%d($sp)\n", reads.get(0), offset * 4);
1182        }
1183    }
1184
1185    /**
1186     * @inheritDoc
1187     */
1188
1189    public String toString() {
1190        return id + ": STORE " + reads.get(0) + " "
1191                + (offsetFrom == OffsetFrom.FP ? "[frame:" : "[stack:")
1192                + offset + "]";
1193    }
1194
1195}
1196
```