NLIRInstruction.java

```java
// 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>();
    }

    /**
     * Replace references to virtual registers in this LIR instruction with
     * references to physical registers.
     */

    public void allocatePhysicalRegisters() {
        // nothing here.
    }

    /**
     * Translate this LIR instruction into SPIM and write it out to the
     * specified output stream.
     *
     * @param out
     *            output stream for SPIM code.
     */

    public void toSpim(PrintWriter out) {
        // nothing here.
    }

    /**
     * Return a string representation of this instruction.
     *
     * @return string representation of this instruction.
     */

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

}

/**
 * LIR instruction corresponding to the JVM arithmetic instructions.
 */

class NLIRArithmetic extends NLIRInstruction {

    /** Opcode for the arithmetic operator. */
    private int opcode;

    /**
     * Construct an NLIRArithmetic instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            opcode for the arithmetic operator.
     * @param lhs
     *            LIR for lhs.
     * @param rhs
     *            LIR for rhs.
     */

    public NLIRArithmetic(NBasicBlock block, int id, int opcode,
            NLIRInstruction lhs, NLIRInstruction rhs) {
        super(block, id);
        this.opcode = opcode;
        reads.add(lhs.write);
        reads.add(rhs.write);
        write = new NVirtualRegister(NControlFlowGraph.regId++, "I", "I");
        block.cfg.registers.add((NVirtualRegister) write);
    }

    /**
```

```java
     * @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);
        NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
        reads.set(0, input1.pRegister);
        reads.set(1, input2.pRegister);
        write = output.pRegister;
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        switch (opcode) {
        case IADD:
            out.printf("    add %s,%s,%s\n", write, reads.get(0), reads.get(1));
            break;
        case ISUB:
            out.printf("    sub %s,%s,%s\n", write, reads.get(0), reads.get(1));
            break;
        case IMUL:
            out.printf("    mul %s,%s,%s\n", write, reads.get(0), reads.get(1));
            break;
        }
    }

    /**
     * @inheritDoc
     */

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

}

/**
 * LIR instruction corresponding to the JVM instructions representing integer
 * constants.
 */

class NLIRIntConstant extends NLIRInstruction {

    /** The constant int value. */
    public int value;

    /**
     * Construct an NLIRIntConstant instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param value
     *            the constant int value.
     */

    public NLIRIntConstant(NBasicBlock block, int id, int value) {
        super(block, id);
        this.value = value;
        write = new NVirtualRegister(NControlFlowGraph.regId++, "I", "I");
        block.cfg.registers.add((NVirtualRegister) write);
    }

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
        write = output.pRegister;
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    li %s,%d\n", write, value);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": LDC [" + value + "] " + write;
    }

}

/**
 * LIR instruction corresponding to the JVM instructions representing string
 * constants.
 */

class NLIRStringConstant extends NLIRInstruction {

    /** The constant string value. */
    public String value;

    /** */
    private static int labelSuffix;

    /**
     * Construct an NHIRStringConstant instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier for the instruction.
     * @param value
     *            the constant string value.
     */

    public NLIRStringConstant(NBasicBlock block, int id, String value) {
        super(block, id);
        this.value = value;
        write = new NVirtualRegister(NControlFlowGraph.regId++, "L",
                "Ljava/lang/String;");
        block.cfg.registers.add((NVirtualRegister) write);
        labelSuffix = 0;
    }

    /**
     * Create a label for LIR code.
     *
     * @return the Label.
     */

    private String createLabel() {
            return "Constant..String" + labelSuffix++;
    }

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
        write = output.pRegister;
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        String label = createLabel();
        String s = label + ":\n";
        int size = 12 + value.length() + 1;
        int align = (size % 4 == 0) ? 0 : (size + 4) / 4 * 4 - size;
        s += "    .word 2 # Tag 2 indicates a string\n";
        s += "    .word " + (size + align) + " # Size of object in bytes\n";
        s += "    .word " + value.length()
                + " # String length (not including null terminator)\n";
        s += "    .asciiz \"" + value
                + "\" # String terminated by null character 0\n";
        s += "    .align " + align + " # Next object is on a word boundary\n";
        block.cfg.data.add(s);
        out.printf("    la %s,%s+12\n", write, label);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": LDC [" + value + "] " + write;
    }

}

/**
 * LIR instruction representing an conditional jump instructions in JVM.
 */

class NLIRConditionalJump extends NLIRInstruction {

    /** Test expression opcode. */
    public int opcode;

    /** Block to jump to on true. */
    public NBasicBlock onTrueDestination;

    /** Block to jump to on false. */
    public NBasicBlock onFalseDestination;

    /**
     * Construct an NLIRConditionalJump instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param lhs
     *            lhs LIR.
     * @param rhs
     *            rhs LIR.
     * @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.
 */

```

```java
class NLIRGoto extends NLIRInstruction {

    /** The destination block to unconditionally jump to. */
    private NBasicBlock destination;

    /**
     * Construct an NLIRGoto instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param destination
     *            the block to jump to.
     */

    public NLIRGoto(NBasicBlock block, int id, NBasicBlock destination) {
        super(block, id);
        this.destination = destination;
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        String label = block.cfg.labelPrefix + "." + destination.id;
        out.printf("    j %s\n", label);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": BRANCH " + destination.id();
    }

}

/**
 * LIR instruction representing method invocation instructions in JVM.
 */

class NLIRInvoke extends NLIRInstruction {

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

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

    /** Name of the method being invoked. */
    public String name;

    /**
     * Construct an NHIRInvoke instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            opcode of the JVM instruction.
     * @param target
     *            target of the method.
     * @param name
     *            name of the method.
     * @param arguments
     *            list of register storing the of arguments for the method.
     * @param sType
     *            return type (short name) of the method.
     * @param lType
     *            return type (long name) of the method.
     */

    public NLIRInvoke(NBasicBlock block, int id, int opcode, String target,
            String name, ArrayList<NRegister> arguments, String sType,
            String lType) {
        super(block, id);
        this.opcode = opcode;
        this.target = target;
        this.name = name;
        for (NRegister arg : arguments) {
            reads.add(arg);
        }
        if (!sType.equals("V")) {
            write = NPhysicalRegister.regInfo[V0];
            block.cfg.registers.set(V0, write);
        }
    }

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        for (int i = 0; i < reads.size(); i++) {
            NInterval input = block.cfg.intervals.get(reads.get(i).number())
                    .childAt(id);
            reads.set(i, input.pRegister);
        }
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    jal %s.%s\n", target.replace("/", "."), name
                .equals("<init>") ? "__init__" : name);
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        String s = id + ": " + lirMnemonic[opcode] + " "
                + (write != null ? write + " = " : "") + target + "." + name
                + "( ";
        for (NRegister input : reads) {
            s += input + " ";
        }
        s += ")";
        return s;
    }

}

/**
 * HIR instruction representing a JVM return instruction.
 */

class NLIRReturn extends NLIRInstruction {

    /** JVM opcode for the return instruction. */
    public int opcode;

    /**
     * Construct an NLIRReturn instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the return instruction.
     * @param result
     *            physical register storing return value, or null.
     */

    public NLIRReturn(NBasicBlock block, int id, int opcode,
            NPhysicalRegister result) {
        super(block, id);
        this.opcode = opcode;
        if (result != null) {
            reads.add(result);
        }
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    j %s\n", block.cfg.labelPrefix + "_restore");
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        if (reads.size() == 0) {
            return id + ": RETURN";
        }
        return id + ": RETURN " + reads.get(0);
    }

}

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

class NLIRPutField 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 NLIRPutField 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.
     * @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);
    }

    /**
     * @inheritDoc
     */

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

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": " + lirMnemonic[opcode] + " [" + dim + "]" + " " + write;
    }

}

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

class NLIRALoad extends NLIRInstruction {

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

    /**
     * Construct an NLIRALoad instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param arrayRef
     *            LIR of the array reference.
     * @param index
     *            LIR of the array index.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

    public NLIRALoad(NBasicBlock block, int id, int opcode,
            NLIRInstruction arrayRef, NLIRInstruction index, String sType,
            String lType) {
        super(block, id);
        this.opcode = opcode;
        reads.add(arrayRef.write);
        reads.add(index.write);
        write = new NVirtualRegister(NControlFlowGraph.regId++, sType, lType);
        block.cfg.registers.add((NVirtualRegister) write);
    }

    /**
     * @inheritDoc
     */

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

    /**
     * @inheritDoc
     */

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

}

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

class NLIRAStore extends NLIRInstruction {

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

    /**
     * Construct an NLIRAStore instruction.
     *
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param opcode
     *            JVM opcode for the instruction.
     * @param arrayRef
     *            LIR of the array reference.
     * @param index
     *            LIR of the array index.
     * @param value
     *            LIR of the value to store.
     * @param sType
     *            type (short name) of the array.
     * @param lType
     *            type (long name) of the array.
     */

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

    /**
     * @inheritDoc
     */

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

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": " + lirMnemonic[opcode] + " " + reads.get(0) + "["
                + reads.get(1) + "] = " + reads.get(2);
    }

}

/**
```

```java
 * 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
     *            register (virtual or physical) to move from.
     * @param to
     *            register (virtual or physical) to move to.
     */

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

    /**
     * @inheritDoc
     */

    public void allocatePhysicalRegisters() {
        NInterval input = block.cfg.intervals.get(reads.get(0).number())
                .childAt(id);
        ;
        NInterval output = block.cfg.intervals.get(write.number()).childAt(id);
        reads.set(0, input.pRegister);
        write = output.pRegister;
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        out.printf("    move %s,%s\n", write, reads.get(0));
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": MOVE " + reads.get(0) + " " + write;
    }

}

/**
 * LIR instruction representing a formal parameter.
 */

class NLIRLoadLocal extends NLIRInstruction {

    /** Local variable index. */
    public int local;

    /**
     * Construct an NLIRLoadLocal instruction.
     * 
     * @param block
     *            enclosing block.
     * @param id
     *            identifier of the instruction.
     * @param local
     *            local variable index.
     * @param sType
     *            short type name of the instruction.
     * @param lType
     *            long type name of the instruction.
     */

    public NLIRLoadLocal(NBasicBlock block, int id, int local, String sType,
            String lType) {
        super(block, id);
        this.local = local;
        if (local < 4) {
            write = NPhysicalRegister.regInfo[A0 + local];
            block.cfg.registers.set(A0 + local, NPhysicalRegister.regInfo[A0
                    + local]);
        } else {
            write = new NVirtualRegister(NControlFlowGraph.regId++, sType,
                    lType);
            block.cfg.registers.add((NVirtualRegister) write);
        }
    }

    /**
     * @inheritDoc
     */

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

}

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

class NLIRLoad extends NLIRInstruction {

    /** Stack offset to load from. */
    private int offset;

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

    /** Register to load to. */
    private NRegister register;

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

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

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        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);
```

```java
        }
    }

    /**
     * @inheritDoc
     */

    public void toSpim(PrintWriter out) {
        if (offsetFrom == OffsetFrom.FP) {
            out.printf("    sw %s,%d($fp)\n", reads.get(0), offset * 4);
        } else {
            out.printf("    sw %s,%d($sp)\n", reads.get(0), offset * 4);
        }
    }

    /**
     * @inheritDoc
     */

    public String toString() {
        return id + ": STORE " + reads.get(0) + " "
                + (offsetFrom == OffsetFrom.FP ? "[frame:" : "[stack:")
                + offset + "]";
    }

}
```