```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;


public abstract class BranchInstruction extends Instruction {

	protected String _formatCode;
	protected String _cc;
	protected String _flag;
	protected String _offset;

	/**
	 * Construct a branch coprocessor instruction (BC-format)
	 * @param binaryString
	 */
	public BranchInstruction(String binaryString) {
		super(binaryString);
		_format = "BC";
		_formatCode = binaryString.substring(6, 11);
		_cc = binaryString.substring(11, 14);
		_flag = binaryString.substring(14,16);
		_offset = binaryString.substring(16, 32);
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int formatCode = Integer.valueOf(_formatCode, 2);
		int cc = Integer.valueOf(_cc, 2);
		int flag = Integer.valueOf(_flag, 2);
		int offset = Integer.valueOf(_offset, 2);
		output.write(opCode + " " + formatCode + " " + cc + " " + flag +
" " + offset);

	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String formatCode = "0x" + Integer.toHexString(Integer.valueOf(_
formatCode, 2));
		String cc = "0x" + Integer.toHexString(Integer.valueOf(_cc, 2));
		String flag = "0x" + Integer.toHexString(Integer.valueOf(_flag,
2));
		String offset = "0x" + Integer.toHexString(Integer.valueOf(_offs
et, 2));
		output.write(opCode + " " + formatCode + " " + cc + " " + flag +
" " + offset);
	}

}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;

public abstract class CoprocessorInstruction extends Instruction {

        public static String[] FREGISTER_NAME = {       "$f0",
"$f1",
"$f2",
"$f3",
"$f4",
"$f5",
"$f6",
"$f7",
"$f8",
"$f9",
"$f10",
"$f11",
"$f12",
"$f13",
"$f14",
"$f15",
"$f16",
"$f17",
"$f18",
"$f19",
"$f20",
"$f21",
"$f22",
"$f23",
"$f24",
"$f25",
"$f26",
```



```
"$f27",
"$f28",
"$f29",
"$f30",
"$f31"
        };

        protected String _formatCode;
        protected String _rt;
        protected String _rdfs;
        protected String _end;

        /**
         * Construct a coprocessor instruction (C-format)
         * @param binaryString
         */
        public CoprocessorInstruction(String binaryString){
                super(binaryString);
                _format = "C";
                _formatCode = binaryString.substring(6, 11);
        }

        @Override
        public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
                int opCode = Integer.valueOf(getOpCode(), 2);
                int functionCode = Integer.valueOf(_formatCode, 2);
                int rt = Integer.valueOf(getRt(), 2);
                int rdfs = Integer.valueOf(getRd(), 2);
                int end = Integer.valueOf(_instructionString.substring(21, 32),
2);
                output.write(opCode + " " + functionCode + " " + rt + " " + rdfs
+ " " + end);

        }

        @Override
        public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
                String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
                String functionCode = "0x" + Integer.toHexString(Integer.valueOf
(_formatCode, 2));
                String rt = "0x" + Integer.toHexString(Integer.valueOf(getRt(),
2));
                String rdfs = "0x" + Integer.toHexString(Integer.valueOf(getRd()
, 2));
                String end = "0x" + Integer.toHexString(Integer.valueOf(_instruc
tionString.substring(21, 32), 2));
                output.write(opCode + " " + functionCode + " " + rt + " " + rdfs
+ " " + end);
        }

        protected String binaryToFReg(String binaryString) {
                int regNumber = Integer.valueOf(binaryString, 2);
                assert(regNumber >= 0 && regNumber <= 31);
                return FREGISTER_NAME[regNumber];
```

```
        }
}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;

public abstract class EretInstruction extends Instruction {

	protected String _functionCode;
	protected String _flag;
	protected String _instruction;

	/**
	 * Construct an eret instruction (E-format)
	 * @param binaryString
	 */
	public EretInstruction(String binaryString) {
		super(binaryString);
		_format = "E";
		_functionCode = getFuncCode();
		_flag = binaryString.substring(6, 7);
		_instruction = binaryString.substring(7, 26);
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int functionCode = Integer.valueOf(_functionCode, 2);
		int flag = Integer.valueOf(_flag, 2);
		int instruction = Integer.valueOf(_instruction, 2);
		output.write(opCode + " " + flag + " " + instruction + " " + func
tionCode);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String functionCode = "0x" +  Integer.toHexString(Integer.valueO
f(_functionCode, 2));
		String flag = "0x" + Integer.toHexString(Integer.valueOf(_flag,
2));
		String instruction = "0x" +  Integer.toHexString(Integer.valueOf
(_instruction, 2));
		output.write(opCode + " " + flag + " " + instruction + " " + func
tionCode);
	}

}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_cclbl extends BranchInstruction {

        public static int[] FORMAT_CODE = {8};
        public static int[] FLAG_CODE = {0 , 1 };
        public static String[] FUNCTION_NAME = {"bc1f", "bc1t"};

        public Ins_cclbl(String binaryString){
                super(binaryString);
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, FLAG_CODE, Intege
r.valueOf(_flag, 2));
                output.write(_functionName + " " + Integer.valueOf(_cc, 2) + " "
+ Integer.valueOf(_offset, 2));
        }

}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_eret extends EretInstruction {

	public static int[] FUNCTION_CODE = {24};
	public static String[] FUNCTION_NAME = {"eret"};

	public Ins_eret(String binaryString) {
		super(binaryString);
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode, 2));
		output.write(_functionName);
	}

}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rtfs extends CoprocessorInstruction {

        public static final int[] FUNCTION_FORMATCODE = { 0, 4 };
        public static final String[] FUNCTION_NAME = { "mfc1", "mtc1"};

        public Ins_rtfs(String binaryString){
                super(binaryString);
                _rt = binaryToReg(getRt());
                _rdfs = binaryToFReg(getRd());
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_FORMATCO
DE, Integer.valueOf(_formatCode, 2));
                output.write(_functionName + " " + _rt + " " + _rdfs);
        }

}
```

```
package Coprocessor;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rtrd_rdrt extends CoprocessorInstruction {

	public static final int[] FUNCTION_FORMATCODE = { 0, 4 };
	public static final String[] FUNCTION_NAME= { "mfc0", "mtc0"};

	public Ins_rtrd_rdrt(String binaryString){
		super(binaryString);
		_rt = binaryToReg(getRt());
		_rdfs = binaryToReg(getRd());
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		String firstRegister, secondRegister;
		_functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_FORMATCO
DE, Integer.valueOf(_formatCode, 2));
		if(Integer.valueOf(_formatCode, 2).equals(0)){
			firstRegister = _rt;
			secondRegister = _rdfs;
		}else{
			firstRegister = _rdfs;
			secondRegister = _rt;
		}
		output.write(_functionName + " " + firstRegister + " " + secondRe
gister);
	}

}
```

```
package Immediate;
import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;


public abstract class ImmediateInstruction extends Instruction {

	protected String _rs;
	protected String _rt;
	protected String _imm;


	/**
	 * Construct an Immediate instruction (I-format)
	 * @param binaryString
	 */
	public ImmediateInstruction(String binaryString) {
		super(binaryString);
		_format = "I";
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int rs = Integer.valueOf(getRs(), 2);
		int rt = Integer.valueOf(getRt(), 2);
		int imm = Integer.valueOf(getImm(), 2);
		output.write(opCode + " " + rs + " " + rt + " " + imm);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String rs = "0x" + Integer.toHexString(Integer.valueOf(getRs(),
2));
		String rt = "0x" + Integer.toHexString(Integer.valueOf(getRt(),
2));
		String imm = "0x" + Integer.toHexString(Integer.valueOf(getImm()
, 2));
		output.write(opCode + " " + rs + " " + rt + " " + imm);
	}

	protected String getImm() {
		return _instructionString.substring(16, 32);
	}

}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rsimm extends ImmediateInstruction {

	public static final int[] RT_CODE = { 8, 9, 10, 11, 12, 14 };
	public static final String[] FUNCTION_NAME = { "tgei", "tgeiu", "tlti", "tltiu"
, "teqi", "tnei" };

	public Ins_rsimm(String binaryString) {
		super(binaryString);
		_rs = binaryToReg(getRs());
		_imm = Integer.valueOf(getImm(), 2).toString();
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, RT_CODE, Integer.
valueOf(getRt(), 2));
		output.write(_functionName + " " + _rs + " " + _imm);
	}


}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rslbl extends ImmediateInstruction {

	public static final int[] RT_CODE = { 0, 1, 16, 17 };
	public static final String[] RT_FUNCTION_NAME = { "bltz", "bgez", "bltzal",
"bgezal" };
	public static final int[] OP_CODE = { 6, 7 };
	public static final String[] OP_FUNCTION_NAME = { "blez", "bgtz" };

	public Ins_rslbl(String binaryString) {
		super(binaryString);
		_rs = binaryToReg(getRs());
		_imm = Integer.toString(Integer.valueOf(getImm(), 2), 16);
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		if (Integer.valueOf(_opCode, 2) == 1) {
			_functionName = getNameFromCode(RT_FUNCTION_NAME, RT_COD
E, Integer.valueOf(getRt(), 2));
		}
		else {
			_functionName = getNameFromCode(OP_FUNCTION_NAME, OP_COD
E, Integer.valueOf(_opCode, 2));
		}
		output.write(_functionName + " " + _rs + " 0x" + _imm);
	}

}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rsrtimm extends ImmediateInstruction {

	public static final int[] OP_CODE = { 9 };
	public static final String[] FUNCTION_NAME = { "addiu" };

	public Ins_rsrtimm(String binaryString) {
		super(binaryString);
		_rs = binaryToReg(getRs());
		_rt = binaryToReg(getRt());
		_imm = Integer.valueOf(getImm(), 2).toString();
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
		output.write(_functionName + " " + _rs + " " + _rt + " " + _imm);
	}

}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rsrtlbl extends ImmediateInstruction {

	public static final int[] OP_CODE = { 4, 5 };
	public static final String[] FUNCTION_NAME = { "beq", "bne" };

	public Ins_rsrtlbl(String binaryString) {
		super(binaryString);
		_rs = binaryToReg(getRs());
		_rt = binaryToReg(getRt());
		_imm = "0x" + Integer.toHexString(Integer.valueOf(getImm(), 2));
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
		output.write(_functionName + " " + _rs + " " + _rt + " " + _imm);
	}

}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rtaddr extends ImmediateInstruction {

        public static final int[] OP_CODE = { 32, 33, 34, 35, 36, 37, 38, 40, 41
, 42, 43, 46, 48, 56 };
        public static final String[] FUNCTION_NAME = { "lb", "lh", "lwl", "lw", "lb
u", "lhu", "lwr", "sb", "sh", "swl", "sw", "swr", "ll", "sc" };

        public Ins_rtaddr(String binaryString) {
                super(binaryString);
                _rt = binaryToReg(getRt());
                _rs = binaryToReg(getRs());
                _imm = Integer.valueOf(getImm(), 2).toString();
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
                output.write(_functionName + " " + _rt + " " + _imm + "(" + _rs +
 ")");

        }

}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rtimm extends ImmediateInstruction {


        public static final int[] OP_CODE = { 15 };
        public static final String[] FUNCTION_NAME = { "lui" };

        public Ins_rtimm(String binaryString) {
                super(binaryString);
                _rt = binaryToReg(getRt());
                _imm = Integer.valueOf(getImm(), 2).toString();
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
                output.write(_functionName + " " + _rt + " " + _imm);
        }


}
```

```
package Immediate;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rtrsimm extends ImmediateInstruction {

	public static final int[] OP_CODE = { 8, 10, 11, 12, 13, 14 };
	public static final String[] FUNCTION_NAME = { "addi", "slti", "sltiu", "andi"
, "ori", "xori" };

	public Ins_rtrsimm(String binaryString) {
		super(binaryString);
		_rs = binaryToReg(getRs());
		_rt = binaryToReg(getRt());
		_imm = Integer.valueOf(getImm(), 2).toString();
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
		output.write(_functionName + " " + _rt + " " + _rs + " " + _imm);
	}

}
```

```
package Interruption;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_irq extends InterruptionInstruction {

	public static int[] FUNCTION_CODE = {12, 13};
	public static String[] FUNCTION_NAME = {"syscall", "break"};

	public Ins_irq(String binaryString) {
		super(binaryString);
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		String code = "";
		_functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode,2));
		if(Integer.valueOf(_functionCode, 2).equals(13)){
			code = " " + "0x" + Integer.toHexString(Integer.valueOf(
_code, 2));
		}
		output.write(_functionName + code);
	}

}
```

```
package Interruption;

import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;

public abstract class InterruptionInstruction extends Instruction {

	protected String _functionCode;
	protected String _code;

	/**
	 * Construct an interruption instruction (IRQ-format)
	 * @param binaryString
	 */
	public InterruptionInstruction(String binaryString) {
		super(binaryString);
		_format = "IRQ";
		_functionCode = getFuncCode();
		_code = binaryString.substring(6, 26);
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int functionCode = Integer.valueOf(_functionCode, 2);
		int code = Integer.valueOf(_code, 2);
		output.write(opCode + " " + code + " " + functionCode);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String functionCode = "0x" + Integer.toHexString(Integer.valueOf
(_functionCode, 2));
		String code = "0x" + Integer.toHexString(Integer.valueOf(_code,
2));
		output.write(opCode + " " + code + " " + functionCode);

	}

}
```

```
package Jump;

import java.io.BufferedWriter;
import java.io.IOException;
import java.math.BigInteger;

public class Ins_jump extends JumpInstruction{

        public static final int[] OP_CODE = { 2, 3 };
        public static final String[] FUNCTION_NAME = { "j", "jal" };

        public Ins_jump(String binaryString) {
                super(binaryString);
                _target = BigInteger.valueOf(Long.valueOf(getTarget(), 2)).toStr
ing(16);
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, OP_CODE, Integer.
valueOf(_opCode, 2));
                output.write(_functionName + " 0x" + _target);
        }



}
```

```
package Jump;
import java.io.BufferedWriter;
import java.io.IOException;
import java.math.BigInteger;

import Main.Instruction;


public abstract class JumpInstruction extends Instruction {

	protected String _target;

	/**
	 * Construct a Jump instruction (J-format)
	 * @param binaryString
	 */
	public JumpInstruction(String binaryString) {
		super(binaryString);
		_format = "J";
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(_instructionString.substring(0, 7),
 2);
		BigInteger target = BigInteger.valueOf(Long.valueOf(_instruction
String.substring(7, 32), 2));
		output.write(opCode + " " + target);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(_inst
ructionString.substring(0, 7), 2));
		String target = "0x" + BigInteger.valueOf(Long.valueOf(_instruct
ionString.substring(7, 32), 2)).toString(16);
		output.write(opCode + " " + target);
	}

	protected String getTarget() {
		return _instructionString.substring(6, 32);
	}
}
```

```
package Main;

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.AbstractMap.SimpleEntry;


public class Dissasembler {

	private static final String BEAUTIFUL_CSS = "<style type=\"text/css\">body{ padding: 0; margin:0; width: 100%;}" +
			"table{ width: 100%; margin:0; padding:0; border:none; border−spacing: 0;}" +
			"tr:nth−child(even) {background: #CCCCCC; border:none;}"+
			"tr:nth−child(odd) {background: #FFFFFF; border:none;}"+
			"td{ margin:0; padding:0; text−align:center; border:none; padding: 8pt;}" +
			"th{ border−bottom: 4px solid black; padding: 8pt;}"+
			"</style>";
	/**
	 * @param args
	 */
	public static void main(String[] args) {
		if (args.length < 2) {
			System.out.println("Usage : Dissasembler <input file name> <output file name>");
			System.out.println("Output is an html file");
			System.exit(0);
		}
		else {
			String inputFileName = args[0];
			String outputFileName = args[1];
			try {
				FileWriter fw = new FileWriter(outputFileName);
				BufferedWriter output = new BufferedWriter(fw);
				ArrayList<SimpleEntry<String, String>> entryList = InstructionReader.readFile(inputFileName);
				printHtmlHeader(output);
				for (SimpleEntry<String,String> entry : entryList) {
					Instruction ins = InstructionFactory.createInstruction((String)entry.getKey());
					output.write("<tr>\n<td>");
					output.write((String)entry.getValue());
					output.write("</td><td>");
					ins.printFormat(output);
					output.write("</td><td>");
					ins.printMnemonic(output);
					output.write("</td><td>");
					ins.printDecomposedDecimal(output);
					output.write("</td><td>");
					ins.printDecomposedHexa(output);
					output.write("</td>\n</tr>\n");
				}
				printHtmlFooter(output);
				output.close();
			}
			catch (IOException e) {
				// TODO Auto−generated catch block
				e.printStackTrace();
			}
		}
```



```
	}

	private static void printHtmlHeader(BufferedWriter output) throws IOException {
		output.write("<html>\n"+BEAUTIFUL_CSS+"<body>\n<table>\n<tr>\n<th>Value</th><th>Format</th><th>Mnemonic</th><th>Decimal decomposition</th><th>Hexadecimal decomposition</th>\n</tr>\n");
	}

	private static void printHtmlFooter(BufferedWriter output) throws IOException {
		output.write("</table>\n</body>\n</html>");
	}

}
```

```
package Main;

import java.io.BufferedWriter;
import java.io.IOException;

public class IncorrectInstruction extends Instruction {

	private String _message;

	public IncorrectInstruction(String binaryString, String message) {
		super(binaryString);
		_message = message;
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		output.write("Incorrect instruction : " + _message);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		output.write("Incorrect instruction : " + _message);
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		output.write("Incorrect instruction : " + _message);
	}



}
```

```
package Main;

import Coprocessor.Ins_cclbl;
import Coprocessor.Ins_eret;
import Coprocessor.Ins_rtfs;
import Coprocessor.Ins_rtrd_rdrt;
import Immediate.Ins_rsimm;
import Immediate.Ins_rslbl;
import Immediate.Ins_rsrtimm;
import Immediate.Ins_rsrtlbl;
import Immediate.Ins_rtaddr;
import Immediate.Ins_rtimm;
import Immediate.Ins_rtrsimm;
import Interruption.Ins_irq;
import Jump.Ins_jump;
import Register.Ins_rd;
import Register.Ins_rdrs;
import Register.Ins_rdrsrt;
import Register.Ins_rdrtimm;
import Register.Ins_rdrtrs;
import Register.Ins_rs;
import Register.Ins_rsrd;
import Register.Ins_rsrt;


public class InstructionFactory {

	private final static int OP_FUNC0 = 0;
	private final static int OP_FUNC1 = 1;
	private final static int OP_FUNC16 = 16;
	private final static int OP_FUNC17 = 17;
	private final static int OP_FUNC28 = 28;

	private static int _opCode;
	private static int _funcCode;
	private static int _rtCode;

	/**
	 * Create an instruction from a binary string
	 * @param binaryString representation of the instruction in binary strin
g
	 * @return
	 */
	public static final Instruction createInstruction(String binaryString) {
		Instruction ins = null;
		_opCode = Integer.valueOf(binaryString.substring(0, 6), 2);
		_funcCode = Integer.valueOf(binaryString.substring(26, 32), 2);
		_rtCode = Integer.valueOf(binaryString.substring(11, 16), 2);
		/*
		 * Check for special functions
		 */
		if (isSpecialFunction(binaryString)) {
			ins = new SpecialInstruction(binaryString);
		}
		/*
		 * Should we look at the function field ?
		 */
		else if (_opCode == OP_FUNC0) {
			if (containsFuncCode(Ins_rd.FUNCTION_CODE)) {
				ins = new Ins_rd(binaryString);
			}
			else if(containsFuncCode(Ins_rdrtimm.FUNCTION_CODE)){
```



```
				ins = new Ins_rdrtimm(binaryString);
			}
			else if(containsFuncCode(Ins_rs.FUNCTION_CODE)){
				ins = new Ins_rs(binaryString);
			}
			else if(containsFuncCode(Ins_rsrt.FUNCTION_CODE_OPCODE0)
){
				ins = new Ins_rsrt(binaryString);
			}
			else if (containsFuncCode(Ins_rsrd.FUNCTION_CODE)) {
				ins = new Ins_rsrd(binaryString);
			}
			else if (containsFuncCode(Ins_rdrtrs.FUNCTION_CODE)) {
				ins = new Ins_rdrtrs(binaryString);
			}
			else if (containsFuncCode(Ins_rd.FUNCTION_CODE)) {
				ins = new Ins_rd(binaryString);
			}
			else if(containsFuncCode(Ins_rdrsrt.FUNCTION_CODE_OPCODE
0)){
				ins = new Ins_rdrsrt(binaryString);
			}else if(containsFuncCode(Ins_irq.FUNCTION_CODE)){
				ins = new Ins_irq(binaryString);
			}
			else {
				ins = new IncorrectInstruction(binaryString, _fu
ncCode + " is not a valid function code for opcode 0");
			}

		}
		else if (_opCode == OP_FUNC1) {
			if (containsRtCode(Ins_rslbl.RT_CODE)) {
				ins = new Ins_rslbl(binaryString);
			}
			else if (containsRtCode(Ins_rsimm.RT_CODE)) {
				ins = new Ins_rsimm(binaryString);
			}
			else {
				ins = new IncorrectInstruction(binaryString, _rt
Code + " is not a valid value for the rt field for opcode 1");
			}
		}
		else if(_opCode == OP_FUNC16){
			if(containsFuncCode(Ins_eret.FUNCTION_CODE)){
				ins = new Ins_eret(binaryString);
			}else{
				int formatCode = Integer.valueOf(binaryString.su
bstring(6, 11), 2);
				if(containsFormatCode(Ins_rtrd_rdrt.FUNCTION_FOR
MATCODE, formatCode)){
					ins = new Ins_rtrd_rdrt(binaryString);
				}
				else {
					ins = new IncorrectInstruction(binaryStr
ing, "Unrecognized instruction");
				}
			}
		}
		else if(_opCode == OP_FUNC17){
			int formatCode = Integer.valueOf(binaryString.substring(
6, 11), 2);
			if(containsFormatCode(Ins_rtfs.FUNCTION_FORMATCODE, form
```

```
atCode)){
                                ins = new Ins_rtfs(binaryString);
                        }else if(containsFormatCode(Ins_cclbl.FORMAT_CODE, forma
tCode)){
                                ins = new Ins_cclbl(binaryString);
                        }
                        else {
                                ins = new IncorrectInstruction(binaryString, "U
nrecognized instruction");
                        }
                }
                else if (_opCode == OP_FUNC28) {
                        if (containsFuncCode(Ins_rsrt.FUNCTION_CODE_OPCODE28)) {
                                ins = new Ins_rsrt(binaryString);
                        }
                        else if (containsFuncCode(Ins_rdrs.FUNCTION_CODE)) {
                                ins = new Ins_rdrs(binaryString);
                        }
                        else if(containsFuncCode(Ins_rdrsrt.FUNCTION_CODE_OPCODE
28)){
                                ins = new Ins_rdrsrt(binaryString);
                        }
                        else {
                                ins = new IncorrectInstruction(binaryString, _fu
ncCode + " is not a valid function code for opcode 28");
                        }
                }
                /*
                 * op code is enough to tell which function we want
                 */
                else if (containsOpCode(Ins_rsrtlbl.OP_CODE)) {
                        ins = new Ins_rsrtlbl(binaryString);
                }
                else if (containsOpCode(Ins_jump.OP_CODE)) {
                        ins = new Ins_jump(binaryString);
                }
                else if (containsOpCode(Ins_rtrsimm.OP_CODE)) {
                        ins = new Ins_rtrsimm(binaryString);
                }
                else if (containsOpCode(Ins_rtimm.OP_CODE)) {
                        ins = new Ins_rtimm(binaryString);
                }
                else if (containsOpCode(Ins_rtaddr.OP_CODE)) {
                        ins = new Ins_rtaddr(binaryString);
                }
                else if (containsOpCode(Ins_rsrtimm.OP_CODE)) {
                        ins = new Ins_rsrtimm(binaryString);
                }
                else if (containsOpCode(Ins_rslbl.OP_CODE)) {
                        ins = new Ins_rslbl(binaryString);
                }
                else {
                        ins = new IncorrectInstruction(binaryString, "Unrecognized
instruction ( opcode = " + _opCode + ")");
                }
                assert(ins != null);
                return ins;
        }

        private static boolean containsOpCode(int[] opCodeClass) {
                assert(_opCode != 1 && _opCode != 0 && _opCode != 28);
                for (int i = 0; i < opCodeClass.length; i++) {
```



```
                        if (_opCode == opCodeClass[i]) {
                                return true;
                        }
                }
                return false;
        }

        private static boolean containsFuncCode(int[] funcCodeClass) {
                assert(_opCode == 0 || _opCode == 28 || _opCode == 16 || _opCode
 == 17);
                for (int i = 0; i < funcCodeClass.length; i++) {
                        if (_funcCode == funcCodeClass[i]) {
                                return true;
                        }
                }
                return false;
        }

        private static boolean containsFormatCode(int[] formatCodeClass, int for
matCode){
                assert(_opCode == 16 || _opCode == 17);
                for (int i = 0; i < formatCodeClass.length; i++) {
                        if (formatCode == formatCodeClass[i]) {
                                return true;
                        }
                }
                return false;
        }

        private static boolean containsRtCode(int[] rtCodeClass) {
                assert(_opCode == 1);
                for (int i = 0; i < rtCodeClass.length; i++) {
                        if (_rtCode == rtCodeClass[i]) {
                                return true;
                        }
                }
                return false;
        }

        private static boolean isSpecialFunction(String binaryString) {
                if (binaryString.equals(SpecialInstruction.OP_NOP)) {
                        return true;
                }
                return false;
        }
}
```

```
package Main;

import java.io.BufferedWriter;
import java.io.IOException;
import java.math.BigInteger;

public abstract class Instruction {


	protected static final String[] REGISTER_NAME= {	"$zero",
			"$at",
			"$v0",
			"$v1",
			"$a0",
			"$a1",
			"$a2",
			"$a3",
			"$t0",
			"$t1",
			"$t2",
			"$t3",
			"$t4",
			"$t5",
			"$t6",
			"$t7",
			"$s0",
			"$s1",
			"$s2",
			"$s3",
			"$s4",
			"$s5",
			"$s6",
			"$s7",
			"$t8",
			"$t9",
			"$k0",
```



```
			"$k1",
			"$gp",
			"$sp",
			"$fp",
			"$ra" };
	protected BigInteger _originalNumber;
	protected String _instructionString;
	protected static String _format;
	protected String _functionName;
	protected String _opCode;

	/**
	 * Construct an instruction from a binary string
	 * @param binaryString
	 */
	public Instruction(String binaryString) {
		_instructionString = binaryString;
		_opCode = getOpCode();
		_originalNumber = BigInteger.valueOf(Long.parseLong(_instruction
String, 2));
	}

	/**
	 * Display the decimal decomposition of the instruction
	 * @param output the buffer of output file
	 * @throws IOException
	 */
	public abstract void printDecomposedDecimal(BufferedWriter output) throw
s IOException;

	/**
	 * Display the hexadecimal decomposition of the instruction
	 * @param output the buffer of output file
	 * @throws IOException
	 */
	public abstract void printDecomposedHexa(BufferedWriter output) throws I
OException;

	/**
	 * Display the mnemonic decomposition of the instruction
	 * @param output the buffer of output file
	 * @throws IOException
	 */
	public abstract void printMnemonic(BufferedWriter output) throws IOExcep
tion;

	/**
	 * Display the value of the instruction in the input file
	 * @param output the buffer of output file
	 * @throws IOException
	 */
	public void printValue(BufferedWriter output) throws IOException {
		output.write(_originalNumber.toString());
	}

	/**
	 * Display the format of the instruction
```

```
         * @param output the buffer of output file
         * @throws IOException
         */
        public void printFormat(BufferedWriter output) throws IOException {
                output.write(_format);
        }

        /**
         * Get the mnemonic representation of a register from a binary string
         * @param binaryString
         * @return
         */
        protected String binaryToReg(String binaryString) {
                int regNumber = Integer.valueOf(binaryString, 2);
                assert(regNumber >= 0 && regNumber <= 31);
                return REGISTER_NAME[regNumber];
        }

        /**
         * Get the integer representation of a binary string
         * @param binaryString
         * @return
         */
        protected String binaryToInt(String binaryString) {
                return Integer.valueOf(binaryString, 2).toString();
        }

        /**
         * Get function name from an array
         * @param nameArray array of function name
         * @param codeArray array of function code
         * @param code code of the instruction
         * @return
         */
        protected String getNameFromCode(String[] nameArray, int[] codeArray, in
t code) {
                assert(nameArray.length == codeArray.length);
                int namePosition = -1;
                for (int i = 0; i < codeArray.length; i++) {
                        if (codeArray[i] == code) {
                                namePosition = i;
                        }
                }
                assert(namePosition != -1);
                return nameArray[namePosition];
        }

        /**
         * Retrieve the opcode of an instruction
         * @return
         */
        protected String getOpCode() {
                return _instructionString.substring(0, 6);
        }

        /**
         * Retrieve the function code of an instruction
         * @return
         */
        protected String getFuncCode() {
                return _instructionString.substring(26, 32);
        }
```



```
        /**
         * Retrieve the rs value of an instruction
         * @return
         */
        protected String getRs() {
                return _instructionString.substring(6, 11);
        }

        /**
         * Retrieve the rt value of an instruction
         * @return
         */
        protected String getRt() {
                return _instructionString.substring(11, 16);
        }

        /**
         * Retrieve the rd value of an instruction
         * @return
         */
        protected String getRd() {
                return _instructionString.substring(16, 21);
        }
}
```

```
package Main;
import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.util.AbstractMap.SimpleEntry;
import java.util.ArrayList;


public final class InstructionReader {


	/**
	 * Parse the input file
	 * @param filename path to the input file
	 * @return An tuple array containing strings in file and their binary re
presentation
	 */
	public static final ArrayList<SimpleEntry<String, String>> readFile(Stri
ng filename) {
		ArrayList<SimpleEntry<String, String>> stringList = new ArrayLis
t<SimpleEntry<String, String>>();
		try {
			FileInputStream input = new FileInputStream(filename);
			BufferedReader reader = new BufferedReader(new InputStre
amReader(new DataInputStream(input)));
			String line;
			while ((line = reader.readLine()) != null) {
				String binaryString = "";
				//System.out.println("Line read = " + line);
				if (line.contains("0x")) {
					BigInteger value = new BigInteger(line.r
eplace("0x", ""), 16);
					binaryString = value.toString(2);
				}
				else {
					BigInteger value = new BigInteger(line);
					binaryString = value.toString(2);
				}
				// Add 0 to the begining if the string is less t
han 32 bit long
				if (binaryString.length() < 32) {
					int nbMissingZero = 32 - binaryString.le
ngth();
					for (int i = 0; i < nbMissingZero; i++)
{
						binaryString = "0" + binaryStrin
g;
					}
				}
				assert(binaryString.length() == 32);
				SimpleEntry<String, String> entry = new SimpleEn
try<String, String>(binaryString, line);
				stringList.add(entry);
			}
			reader.close();
		} catch (FileNotFoundException e) {
			System.out.println("Unable to find or open file : " + filename + "\
nclosing");
```



```
			e.printStackTrace();
			System.exit(0);
		} catch (IOException e) {
			e.printStackTrace();
		}
		return stringList;
	}

}
```

```
package Main;

import java.io.BufferedWriter;
import java.io.IOException;
import java.math.BigInteger;

public class SpecialInstruction extends Instruction {

	public static final String OP_NOP = "00000000000000000000000000000000";

	protected String _mnemonic;

	/**
	 * Construct a special instruction
	 * @param binaryString
	 */
	public SpecialInstruction(String binaryString) {
		super(binaryString);
		if (binaryString.equals(OP_NOP)) {
			_mnemonic = "nop";
		}
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		output.write(BigInteger.valueOf(Long.valueOf(_instructionString,
 2)).toString());
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		output.write("0x" + BigInteger.valueOf(Long.valueOf(_instruction
String, 2)).toString(16));
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		output.write(_mnemonic);
	}

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rd extends RegisterInstruction{

        public static final int[] FUNCTION_CODE = { 16, 18 };
        public static final String[] FUNCTION_NAME = { "mfhi", "mflo" };

        public Ins_rd(String binaryString) {
                super(binaryString);
                _rd = binaryToReg(getRd());
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode, 2));
                output.write(_functionName + " " + _rd);
        }

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rdrs extends RegisterInstruction {

        public static final int[] FUNCTION_CODE = { 32, 33 };
        public static final String[] FUNCTION_NAME = { "clz", "clo" };

        public Ins_rdrs(String binaryString) {
                super(binaryString);
                _rd = binaryToReg(getRd());
                _rs = binaryToReg(getRs());
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName =  getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, I
nteger.valueOf(_functionCode, 2));
                output.write(_functionName + " " + _rd + " " + _rs);
        }

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rdrsrt extends RegisterInstruction {

        public static final int[] FUNCTION_CODE_OPCODE0 = { 10, 11, 32, 33, 34,
35, 36, 37, 38, 39, 42, 43, };
        public static final String[] FUNCTION_NAME_OPCODE0 = { "movz", "movn", "
add", "addu", "sub", "subu", "and", "or", "xor", "nor", "slt", "sltu" };
        public static final int[] FUNCTION_CODE_OPCODE28 = { 2 };
        public static final String[] FUNCTION_NAME_OPCODE28 = { "mul" };

        public Ins_rdrsrt(String binaryString) {
                super(binaryString);
                _rd = binaryToReg(getRd());
                _rs = binaryToReg(getRs());
                _rt = binaryToReg(getRt());

        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                if(Integer.valueOf(_opCode, 2) == 0){
                        _functionName = getNameFromCode(FUNCTION_NAME_OPCODE0, F
UNCTION_CODE_OPCODE0, Integer.valueOf(_functionCode, 2));
                }else{
                        _functionName = getNameFromCode(FUNCTION_NAME_OPCODE28,
FUNCTION_CODE_OPCODE28, Integer.valueOf(_functionCode, 2));
                }
                output.write(_functionName + " " + _rd + " " + _rs + " " + _rt);
        }

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rdrtimm extends RegisterInstruction {

	public static final int[] FUNCTION_CODE = { 0, 2, 3};
	public static final String[] FUNCTION_NAME = { "sll", "srl", "sra" };

	private String _shamt;

	public Ins_rdrtimm(String binaryString) {
		super(binaryString);
		_rd = binaryToReg(getRd());
		_rt = binaryToReg(getRt());
		_shamt = getShamt();
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode, 2));
		output.write(_functionName + " " + _rd + " " + _rt + " " + Intege
r.valueOf(_shamt, 2));
	}

	public String getShamt(){
		return _instructionString.substring(21, 26);
	}

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rdrtrs extends RegisterInstruction {

	public static final int[] FUNCTION_CODE = { 4, 6, 7 };
	public static final String[] FUNCTION_NAME = { "sllv", "srlv", "srav" };

	public Ins_rdrtrs(String binaryString) {
		super(binaryString);
		_rd = binaryToReg(getRd());
		_rs = binaryToReg(getRs());
		_rt = binaryToReg(getRt());
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		_functionName =  getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, I
nteger.valueOf(_functionCode, 2));
		output.write(_functionName + " " + _rd + " " + _rt + " " + _rs);
	}

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rs extends RegisterInstruction {

        public static final int[] FUNCTION_CODE = { 8, 17, 19};
        public static final String[] FUNCTION_NAME = { "jr", "mthi", "mtlo"};

        public Ins_rs(String binaryString) {
                super(binaryString);
                _rs = binaryToReg(getRs());
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode, 2));
                output.write(_functionName + " " + _rs);
        }

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rsrd extends RegisterInstruction {

        public static final int[] FUNCTION_CODE = { 9 };
        public static final String[] FUNCTION_NAME = { "jalr" };

        public Ins_rsrd(String binaryString) {
                super(binaryString);
                _rs = binaryToReg(getRs());
                _rd = binaryToReg(getRd());
        }

        @Override
        public void printMnemonic(BufferedWriter output) throws IOException {
                _functionName = getNameFromCode(FUNCTION_NAME, FUNCTION_CODE, In
teger.valueOf(_functionCode, 2));
                output.write
                (_functionName + " " + _rs + " " + _rd);
        }

}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

public class Ins_rsrt extends RegisterInstruction {

	public static final int[] FUNCTION_CODE_OPCODE0 = {24, 25, 26, 27, 48, 4
9, 50, 51, 52, 54};
	public static final String[] FUNCTION_NAME_OPCODE0 = {"mult", "multu", "di
v", "divu", "tge", "tgeu", "tlt", "tltu", "teq", "tne"};
	public static final int[] FUNCTION_CODE_OPCODE28 = {0, 1, 4, 5};
	public static final String[] FUNCTION_NAME_OPCODE28 = {  "madd", "maddu"
, "msub", "msubu"};

	public Ins_rsrt(String binaryString) {
		super(binaryString);
		// TODO Auto-generated constructor stub
		_rs = binaryToReg(getRs());
		_rt = binaryToReg(getRt());
	}

	@Override
	public void printMnemonic(BufferedWriter output) throws IOException {
		if(Integer.parseInt(_opCode, 2) == 0){
			_functionName = getNameFromCode(FUNCTION_NAME_OPCODE0, F
UNCTION_CODE_OPCODE0, Integer.valueOf(_functionCode, 2));
		}else{
			_functionName = getNameFromCode(FUNCTION_NAME_OPCODE28,
FUNCTION_CODE_OPCODE28, Integer.valueOf(_functionCode, 2));
		}
		output.write(_functionName + " " + _rs  + " " + _rt);
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int rs = Integer.valueOf(getRs(), 2);
		int rt = Integer.valueOf(getRt(), 2);
		int part5 = Integer.valueOf(_instructionString.substring(16, 26)
, 2);
		int funcCode = Integer.valueOf(getFuncCode(), 2);
		output.write(opCode + " " + rs + " " + rt + " " + part5 + " " + fu
ncCode);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String rs = "0x" + Integer.toHexString(Integer.valueOf(getRs(),
2));
		String rt = "0x" + Integer.toHexString(Integer.valueOf(getRt(),
2));
		String part5 = "0x" + Integer.toHexString(Integer.valueOf(_instr
uctionString.substring(16, 26), 2));
		String funcCode = "0x" + Integer.toHexString(Integer.valueOf(get
FuncCode(), 2));
		output.write(opCode + " " + rs + " " + rt + " " + part5 + " " + fu
ncCode);
```



```
	}
}
```

```
package Register;

import java.io.BufferedWriter;
import java.io.IOException;

import Main.Instruction;


public abstract class RegisterInstruction extends Instruction {

	protected String _rs;
	protected String _rd;
	protected String _rt;
	protected String _functionCode;

	/**
	 * Construct a register instruction (R format)
	 * @param binaryString
	 */
	public RegisterInstruction(String binaryString) {
		super(binaryString);
		_format = "R";
		_functionCode = binaryString.substring(26, 32);
	}

	@Override
	public void printDecomposedDecimal(BufferedWriter output) throws IOExcep
tion {
		int opCode = Integer.valueOf(getOpCode(), 2);
		int rs = Integer.valueOf(getRs(), 2);
		int rt = Integer.valueOf(getRt(), 2);
		int rd = Integer.valueOf(getRd(), 2);
		int part5 = Integer.valueOf(_instructionString.substring(21, 26)
, 2);
		int funcCode = Integer.valueOf(getFuncCode(), 2);
		output.write(opCode + " " + rs + " " + rt + " " + rd + " " + part5
 + " " + funcCode);
	}

	@Override
	public void printDecomposedHexa(BufferedWriter output) throws IOExceptio
n {
		String opCode = "0x" + Integer.toHexString(Integer.valueOf(getOp
Code(), 2));
		String rs = "0x" + Integer.toHexString(Integer.valueOf(getRs(),
2));
		String rt = "0x" + Integer.toHexString(Integer.valueOf(getRt(),
2));
		String rd = "0x" + Integer.toHexString(Integer.valueOf(getRd(),
2));
		String part5 = "0x" + Integer.toHexString(Integer.valueOf(_instr
uctionString.substring(21, 26), 2));
		String funcCode = "0x" + Integer.toHexString(Integer.valueOf(get
FuncCode(), 2));
		output.write(opCode + " " + rs + " " + rt + " " + rd + " " + part5
 + " " + funcCode);
	}


}
```