

# The RedCode Assembler

M. B. Zanussi

May 12, 2004

## 1 Designing the RedCode Assembler

### 1.1 About the RedCode Assembler

The RedCode assembler converts a human-readable text file, made up of RedCode assembly language, into a binary program image file. The RedCode assembly language used in the DCoreWars program is derived from the MIPS instruction set, a register-to-register RISC language. Every instruction is specified by a single 32-bit word, broken up into fields indicating its opcode and arguments. RedCode uses a subset of the MIPS instruction set and also extends it with a custom set of operations defined specifically for DCoreWars. These additional instructions replace existing MIPS instructions not used in this project.

The following sections describe a number of assembler features in more detail.

### 1.2 The Lexer

Figure 1 shows the class hierarchy for the lexer. Given a RedCode assembly file (the warrior), the lexer breaks up the file into individual tokens, which are passed along to the assembler for encoding into a 32-bit word (integer) representing a complete RedCode instruction. The lexer interface defines three methods: `hasMoreTokens()`, `nextToken()`, and `pushBack()`.

The state table for the RedCodeLexer FSM is also given below. For a given current state and an input symbol, the table gives the next state and action to be performed. In RedCodeLexer this state table is defined by a *switch* statement to handle the current state, inside a series of *if-else* statements representing the input character.

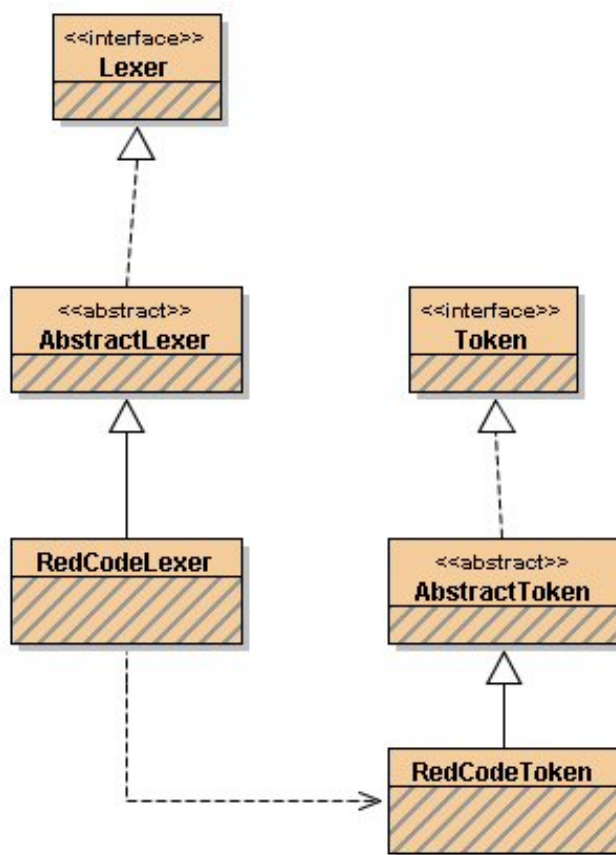


Figure 1: The RedCode lexer class hierarchy.

Current	[A-Za-z]	[0-9]	[\$]
NONE	OPCODE	ARG	ARG
OPCODE	OPCODE	ARG <sup>r</sup>	ARG <sup>r</sup>
ARG	ARG	ARG	ARG
SYMBOL	OPCODE <sup>r</sup>	ARG <sup>r</sup>	ARG <sup>r</sup>

Current	[ ( ]	) ]	[ , ]	Whitespace
NONE	SYMBOL	SYMBOL	NONE <sup>i</sup>	NONE <sup>i</sup>
OPCODE	SYMBOL <sup>r</sup>	SYMBOL <sup>r</sup>	NONE <sup>x</sup>	NONE <sup>x</sup>
ARG	SYMBOL <sup>r</sup>	SYMBOL <sup>r</sup>	NONE <sup>x</sup>	NONE <sup>x</sup>
SYMBOL	SYMBOL <sup>r</sup>	SYMBOL <sup>r</sup>	NONE <sup>x</sup>	NONE <sup>x</sup>

State actions (superscripted modifiers):

*i*: Ignore current character, then continue parsing.

*r*: Save current character to next token, then return token.

*x*: Ignore current character, then return token.

*no modifier*: Add current character to token, then continue parsing.

### 1.3 The Assembler

The assembler is an integrated component of the DCoreWars program. In addition to assembling into object code, it also disassembles a RedCode program binary image into human-readable text. The assembler for this project does not support the more sophisticated features of MIPS assemblers such as conditional assembly, pseudo instructions, etc.

The assembler interface `RedCodeInstruction` defines the following three methods: `encode()`, `decode()` and `exec()`. Given a lexer object to the input file, `encode()` parses the current line of assembly code and returns the instruction as a 32-bit word. The `decode()` method reverses this procedure, so provided the object code as a 32-bit word, it returns the human-readable assembly instruction. Finally, given a reference to the RVM's register set and memory, `exec()` executes the current instruction.

The original design called for the entire instruction set, both the standard MIPS instructions and the extended instructions, to be fully described by a mnemonic table. The mnemonic table contained instruction-specific data for each RedCode assembly language instruction, including instruction-format, opcode, etc., which was initially stored to disk and then read into memory (a hash table) by the loader. The idea was that it would have been easy to extend the RedCode language by adding to the file rather than having to modify the code directly, and more importantly, no instruction-specific code would have to be hardcoded into the application.

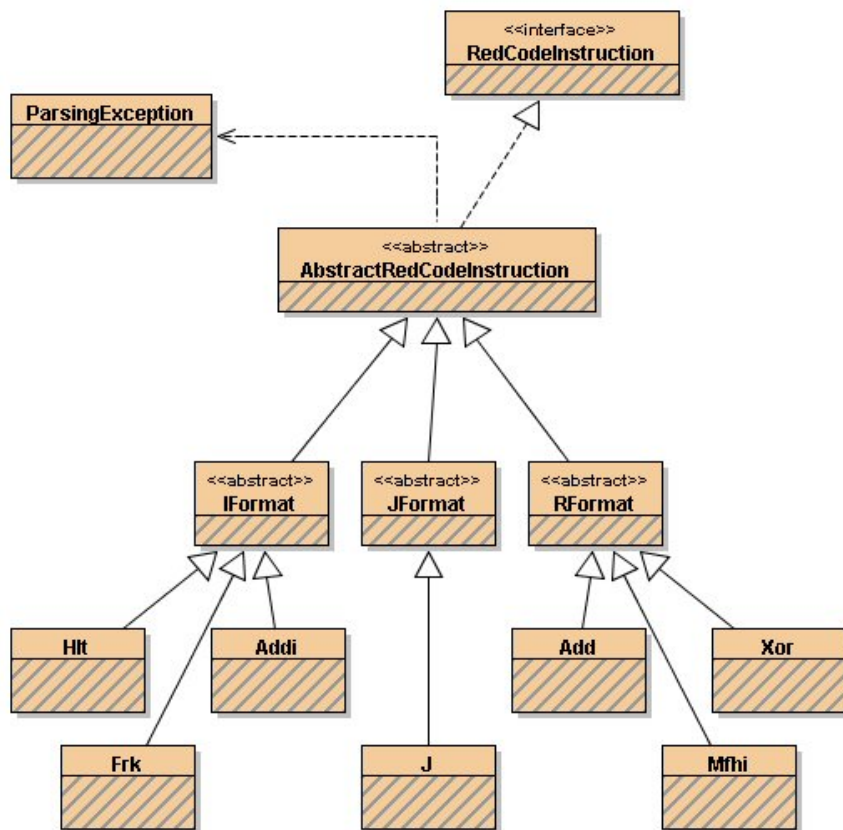


Figure 2: The RedCode assembler class hierarchy.

***Milestone 1 Update:*** As work progressed on the assembler, however, it became impossible not to avoid having to hardcode certain information. For example, while the instruction formats (I, J and R) provide a certain amount of uniformity, the number and order of the arguments often vary. Certainly this could be described within the mnemonic table file, but doing so would cause the system complexity to grow considerably, not to mention require additional time to implement, which really isn't a luxury with this project. Therefore, the choice was made to combine hardcoding with a mnemonic table. It seemed a good compromise at the time...

Unfortunately, though, as coding progressed further and more thought was put into the loader and instruction execution, I came to realize that the original design was beginning to complicate matters. It was no longer extensible, it wasn't nearly as object oriented in design as it should be, the code was becoming more jumbled and complex (more hardcoding was going to be needed), and I could see future problems down the road when it came to instruction execution, in particular a lot of work on the loader (or whatever component) to perform preliminary decoding before passing it on to the assembler to finish decoding. Realizing that the design really wouldn't work as I wanted, I decided to switch mid-stream and throw out the whole mnemonic table design and instead implement a class-per-instruction design. Figure 2 above shows the class hierarchy for the assembler using this new approach.

The abstract class `AbstractRedCodeInstruction` is the base class for all derived instruction classes. Its primary purpose is to provide parsing capabilities such as parsing address values and immediate values. Below this abstract class are the three instruction formats (I, J and R). These three abstract classes provide the field data variables, a `toString()` method, and the method which actually creates the 32-bit integer instruction from each of the data fields and also breaks a 32-bit word into its various components. Finally, derived from the various instruction formats, are the individual instruction classes themselves. In these classes is the interface implemented.

Obviously, my first concern was the amount of time required to implement this new design; I thought it would take much longer to do. In fact, since a good portion of the code I needed was already under the hood, the actual conversion only took about four hours since it was mostly a matter of creating new classes and shifting code around. Another concern was the considerable growth in the number of classes for the assembler, which added about 44 classes. That, of course, could not be avoided, but in the overall scheme of

things, it's really not too difficult juggling that many classes.

On the plus side, with a class-per-instruction implementation, it's now fully object oriented, which makes for a much more elegant and efficient solution. In addition, it's also fully extensible – all hardcoding is gone. This is possible because each and every instruction object is now dynamically instantiated, and each object contains the data it requires to do its job and doesn't rely on various hardcoded data to identify the instruction type or instruction itself. If a future version wishes to extend the RedCode assembly language, it's a simple matter of adding a new instruction class. No other coding is necessary. Also, instruction execution is now a matter of calling the `exec()` method. The RVM will have no need to know what type of instruction or anything about it, to execute the instruction.

***Milestone 2 Update:*** In order to handle new instructions created on the fly by the instructions themselves (i.e., self-modifying instructions), the hash table model as implemented isn't adequate, since new instructions wouldn't be in the table. So, an implementation was devised that combined the current design with a mnemonic table. The new mnemonic table includes the instruction name with the opcode/funct value, so that each instruction is associated with a unique value, which during the course of execution is matched with the complete instruction set in the hash table. This way, newly created instructions are handled seamlessly. Also, the system remains fully extensible without the need to hardcode new instructions. As before, an instruction class is created for the new instruction, but now, a new entry is also placed in the mnemonic table.

***Milestone 3 Update:*** No significant changes to design.

## 1.4 The Loader

Figure 3 shows the simple class hierarchy for the loader.

The job of the loader is to process a RedCode assembly file into object code. First, the warrior is opened and fed into a lexer. The instruction name is parsed so that a new instruction object can be instantiated:

```
Class.forName(instruction).newInstance().
```

Once we have an object, its `encode()` method is called and the assembler spits back the 32-bit word representing the instruction, placing it in a vector which will then be used by the RVM to populate the its memory by locating a

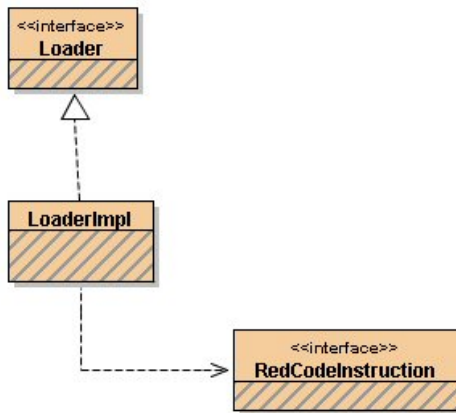


Figure 3: The Loader class hierarchy.

random location in its memory and inserting the instructions sequentially in contiguous memory (the RVM will call the `getInstructionSet()` method).

During the fetch phase, the fetched instruction in 32-bit word format will be used to recreate the assembly language instruction and ultimately create an instruction object via the `getInstructionObject()` method, and from there `exec()` can be called from the instruction object without any fuss. That is, the component responsible for executing the current instruction doesn't need to know anything at all about the instruction it's going to execute.

*Milestone 2 Update:* To facilitate easier integration into the RVM, added the `Loader` interface and moved all implementation to `LoaderImpl`.

*Milestone 3 Update:* No significant changes to design.

## 2 Testing the RedCode Assembler

### 2.1 Methodology

Testing was performed using a custom-designed GUI simulator, `TestLoader`. The simulator has the ability to load a warrior from disk and encode/decode the individual instructions in the RedCode assembly file. In addition, each instruction can be viewed in its binary format as contained in the `Vector` (which the RVM ultimately loads) using a binary dump feature. This allows one to verify that the 32-bit word is constructed correctly.

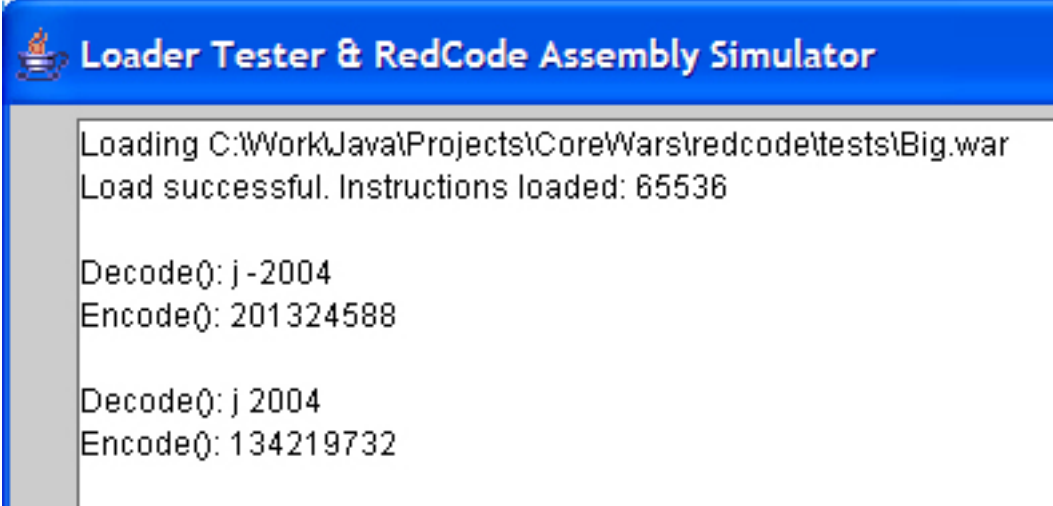
**Milestone 2 Update:** Originally this was the extent of the tester, but when it was decided to include the execute method within each instruction object, the tester was extended to provide instruction execution and it basically morphed into a simulator along the lines of SPIM, but no where near as sophisticated. In addition to being able to singly execute an encoded instruction from a loaded warrior file, the user can execute any instruction they wish to through a simple text box interface. To verify that instructions have executed correctly, a register dump feature is included.

The scope of the loader/assembler/disassembler testing was as follows: individual instruction encoding, decoding and execution; warrior loading and parsing; instruction set creation. Note: Testing of individual instructions which directly manipulate RVM memory (such as `lw`, `rsw`, etc.) will be covered by the RVM testing.

**Milestone 3 Update:** No significant changes to design.

## 2.2 Loader Results

Since the max RVM size is  $2^{16}$ , the system was stress tested using `big.war`, which contains 65536 instructions. The first and last instructions, `j -2004` and `j 2004`, respectively, are used in determining if the file was only partially loaded. Figure 4 shows the results.



```
Loader Tester & RedCode Assembly Simulator
Loading C:\Work\Java\Projects\CoreWars\redcode\tests\Big.war
Load successful. Instructions loaded: 65536

Decode(): j -2004
Encode(): 201324588

Decode(): j 2004
Encode(): 134219732
```

Figure 4: Loading a very large warrior.

The loader was also tested to see if it would recognize an empty warrior. For this test, `empty.war` was used, and the results are shown in Figure 5.

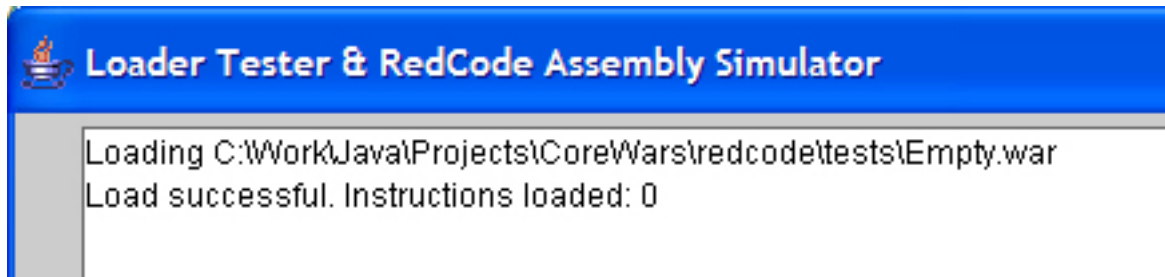


Figure 5: Loading an empty warrior.

After loading, the system was tested to see if the warrior could be successfully parsed. Since exceptions are passed back to the application that instantiated the loader, error messages were only sent to standard output for testing purposes and not reflected in this document. Figure 6 shows an example of RedCode assembly which would cause such exceptions and stop the parsing of a warrior, while Figure 7 is an example of good assembly code which successfully passed parsing tests.

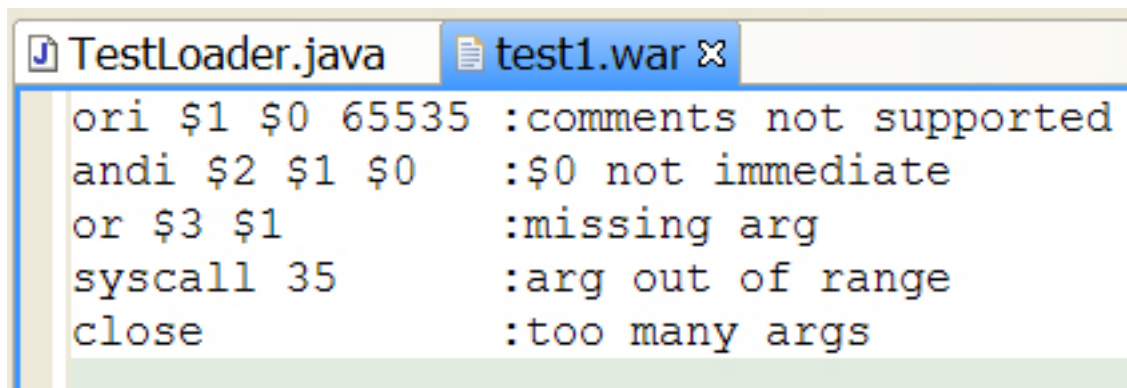
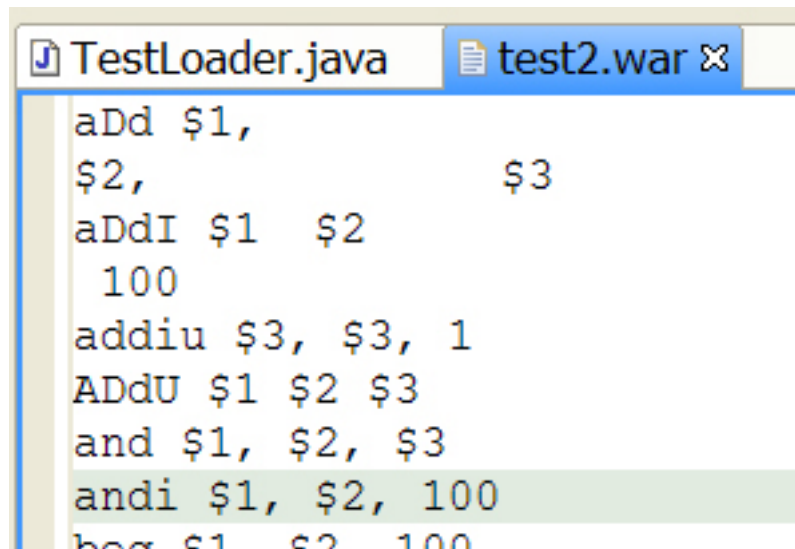


Figure 6: Bad RedCode assembly instructions.

After the successful parsing of a warrior, an instruction set is generated. This instruction set consists of a Vector of `ints`, which are the results of encoding the plain-text into object code. To test this and verify the accuracy of the `encode()` and `decode()` methods for all 45 instructions, a special warrior



```
TestLoader.java test2.war
aDd $1,
$2,          $3
aDdI $1 $2
  100
addiu $3, $3, 1
ADdU $1 $2 $3
and $1, $2, $3
andi $1, $2, 100
hex $1 $2 100
```

Figure 7: Good RedCode assembly instructions.

file consisting of all available instruction types was read in (see `test2.war`), encoded into a single word, and then decoded back into human-readable assembly. To verify the results, the decoded assembly was re-encoded and that in turn was itself decoded. An example of this testing can be seen in Figure 8, where the instructions `add` and `lw` are put through their paces. Each instruction, then, was compared directly with the original warrior file to verify that the methods functioned properly.

It was also useful being able to debug the actual instruction in its binary format, as Figure 9 shows. At times where decoding or encoding didn't work, it was helpful seeing the binary representation of the instruction.

### 2.3 Instruction Execution Notes

For the testing of execution instructions, two methods are provided by the simulator. In the first method, a loaded warrior can be executed instruction-by-instruction by selecting the instruction from the drop-down list (the list only shows the instruction as an `int`) and then clicking the `Execute` button. The human-readable instruction will be displayed in the top text area while a register dump will appear in the bottom text area.

The other method is to manually input an instruction into the text box and then press the `Execute` button. The output will mirror what the first

```
Loader Tester & RedCode Assembly Simulator
Loading C:\Work\Java\Projects\CoreWars\redcode\tests\test2.war
Load successful. Instructions loaded: 45

The instruction: 4392992
1. Decode() : add $1, $2, $3
2. Encode() : 4392992
3. Decode() : add $1, $2, $3

The instruction: -1941897116
1. Decode() : lw $1, 100($2)
2. Encode() : -1941897116
3. Decode() : lw $1, 100($2)
```

Figure 8: Testing the encoding and decoding functionality.

method does. See Figure 10 for a snippet of the bottom text area.

It is important to note that only instructions that manipulate the registers have been tested within the assembler. Those instructions that touch RVM memory will be tested and results documented under the RVM documentation.

## 2.4 Branch Instruction Execution Results

Branch instructions use signed 16-bit offsets, so ranges were tested in addition to verifying that relative addressing functioned properly. For the branch instruction `j`, range checking was performed on a 26-bit value. It should be noted that the assembler does not check if the resultant PC is valid or not, as only the RVM knows the state of its own memory. This includes negative PC values, since the RVM is in charge of handling illegal operations like these. For brevity, only the results for the `beq`, `j` and `jr` instructions are listed. For testing purposes, the pre-loading of the registers is as follows:  $PC = 0$ ,  $\$2 = -100$ , and  $\$3 = 100$ . All other registers are initialized to 0. In addition to the branch instructions, the movement instruction `mfpc` was tested also.

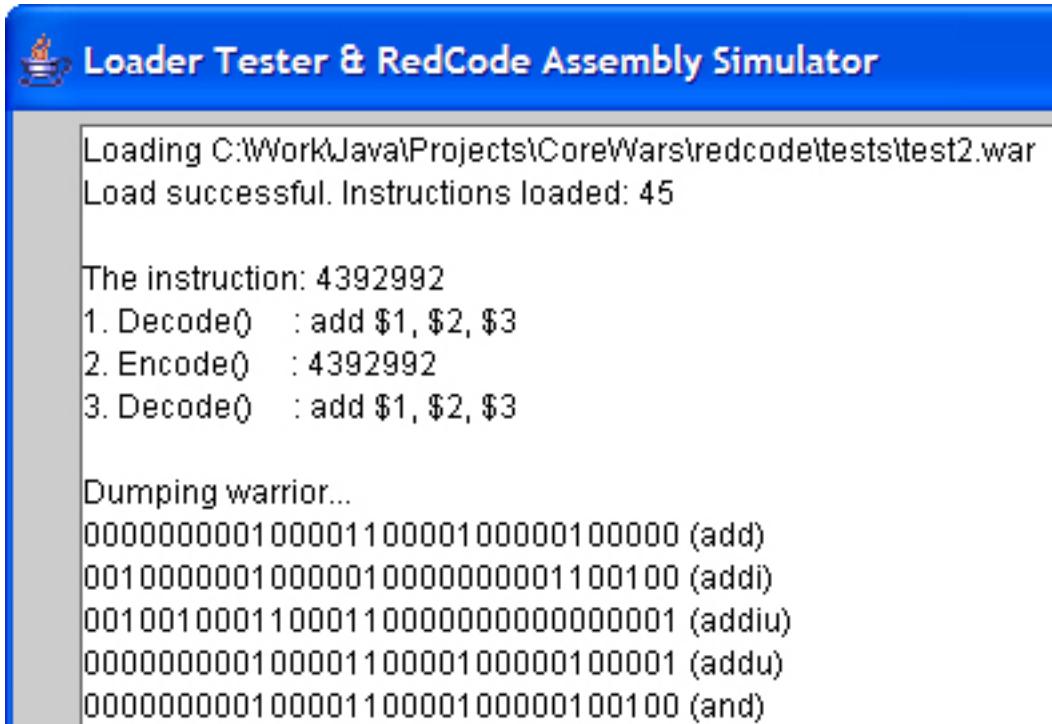


Figure 9: A sample of a warrior binary dump.

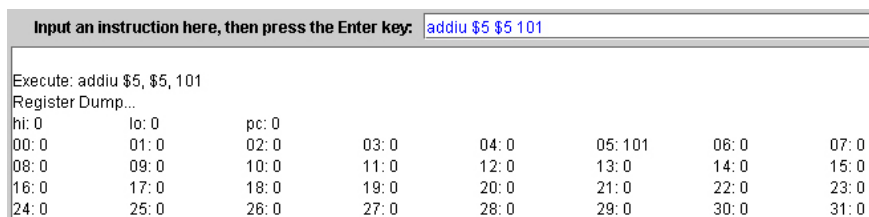


Figure 10: Executing an instruction manually.

The results are not displayed here.

Instruction	PC (before)	PC (after)	Notes
beq \$0 \$0 32767	0	32767	<b>Pass</b>
beq \$2 \$2 32768	0	0	Illegal - out of range
beq \$3 \$3 -32768	0	-32768	<b>Pass</b> <sup>1</sup>
beq \$3 \$3 -32769	0	-32768	Illegal - out of range
beq \$2 \$3 -32768	0	0	<b>Pass</b>
j 33554431	0	32767	<b>Pass</b> <sup>1</sup>
j 33554432	0	0	Illegal - out of range
j -33554432	0	32767	<b>Pass</b> <sup>1</sup>
j -33554433	0	0	Illegal - out of range
jr \$3	0	100	<b>Pass</b>
jr \$2	0	-100	<b>Pass</b> <sup>1</sup>

Caveats:

1: This is a legal MIPS instruction that the RVM will flag as illegal.

## 2.5 Arithmetic Instruction Execution Results

Where the arithmetic instructions require an immediate value, range checking was performed. Depending on the instruction, this immediate could be either signed or unsigned. Since the specifications state all operations as ignoring overflow, instructions such as `add` and `addu` were treated as equivalent. For testing purposes, the pre-loading of the registers is as follows: `$1 = 50`, `$2 = -100`, `$3 = 100`, and `$13 = 13`. In addition, registers `$9` and `$10` each have `65536` loaded into them. All other registers are initialized to 0. In addition to the arithmetic instructions, the two movement instructions `mfhi` and `mflo` were tested immediately after `mult` and `divu`. Those results are not displayed here.

Instruction	Result	HI	LO	Notes
add \$4 \$1 \$2	-50	0	0	Pass
add \$4 \$1 \$3	150	0	0	Pass
addiu \$4 \$3 32767	32867	0	0	Pass
addiu \$4 \$3 32768	0	0	0	Illegal - out of range
addiu \$4 \$2 -32768	-32868	0	0	Pass
addiu \$4 \$2 -32769	0	0	0	Illegal - out of range
sub \$4 \$1 \$2	150	0	0	Pass
sub \$4 \$1 \$3	-50	0	0	Pass
mult \$1 \$3	0	0	5000	Pass
mult \$9 \$10	0	1	0	Pass
divu \$1 \$13	0	11	3	Pass
divu \$13 \$1	0	13	0	Pass
divu \$2 \$13	0	-9	-7	Pass

## 2.6 Logical Instruction Execution Results

Where the logical instructions require an immediate value, range checking was performed. This immediate value is always zero-extended and thus unsigned. Results were simply verified against a scientific calculator. For testing purposes, the pre-loading of the registers is as follows: \$1 = 12828 and \$2 = 7800.

Instruction	Result	Notes
and \$3 \$1 \$2	4632	Pass
andi \$3 \$1 65535	12828	Pass
andi \$3 \$1 65536	0	Illegal - out of range
andi \$3 \$1 -1	0	Illegal - out of range
or \$3 \$1 \$2	15996	Pass
ori \$3 \$1 65535	65535	Pass
xor \$3 \$1 \$2	11364	Pass
nor \$3 \$1 \$2	-15997	Pass

## 2.7 Shift Instruction Execution Results

The shift instructions are easy to verify. For `sll`, shifting a value left by  $n$  places is equivalent to multiplying that number by  $2^n$ . For `sra` and `srl`, shifting a value right by  $n$  places is equivalent to dividing that number by  $2^n$ . The difference, however, is that `sra` is sign-extended (if original high-order bit was 0, 0s are shifted in while if the original high-order bit was 1, 1s are shifted in) while in `srl` it's zero-extended (0s are always shifted into high-order bits). The valid shift amounts are 0 through 31, so range checking

was also tested. For testing purposes, the pre-loading of the registers is as follows: \$1 = 2, \$2 = 256, \$3 = -2147483648, \$5 = -1, \$6 = 1 and \$7 = 32.

<b>Instruction</b>	<b>Result</b>	<b>Notes</b>
sll \$4 \$1 1	4	<b>Pass</b>
sll \$4 \$1 2	8	<b>Pass</b>
sll \$4 \$1 30	-2147483648	<b>Pass</b>
sll \$4 \$1 31	0	<b>Pass</b>
sll \$4 \$1 32	0	Illegal - out of range
sll \$4 \$1 0	2	<b>Pass</b>
sll \$4 \$1 -1	0	Illegal - out of range
sllv \$4 \$1 \$6	4	<b>Pass</b>
sllv \$4 \$1 \$5	0	Illegal - out of range
sllv \$4 \$1 \$7	0	Illegal - out of range
sra \$4 \$3 30	-2	<b>Pass</b>
srl \$4 \$3 30	2	<b>Pass</b>
srlv \$4 \$2 \$6	128	<b>Pass</b>

## 2.8 Comparison Instruction Execution Results

Some comparison instructions use 16-bit signed immediates, so ranges were tested in addition to verifying that the set-less-than worked. For testing purposes, the pre-loading of the registers is as follows: \$0 = 0 (always), \$2 = -100, and \$3 = 100. All other registers are initialized to 0.

<b>Instruction</b>	<b>Result</b>	<b>Notes</b>
slt \$1 \$0 \$0	0	<b>Pass</b>
slt \$1 \$0 \$2	0	<b>Pass</b>
sltu \$1 \$0 \$3	1	<b>Pass</b>
slti \$1 \$2 -32769	0	Illegal - out of range
slti \$1 \$2 -32769	0	<b>Pass</b>
sltiu \$1 \$2 32768	0	Illegal - out of range
sltiu \$1 \$2 32767	1	<b>Pass</b>