

CS351 Spring 2004, Project 3

DCoreWars

MondoSoft, Inc.

March 29, 2004

STATUS Specification and requirements document

VERSION 1.0

DATE Mar 29, 2004

0 Changelog

Version 1.0 Initial release. Mar 29, 2004.

1 Summary

It's a hard network out there. A program's gotta fight just to keep its clock cycles, not to mention holding down its own bit of memory real estate from interloping programs that would like nothing better than to snatch that prime bit of RAM away and consign the program to the great bit bucket in the sky...

In this project, the design team will develop a game engine for the “distributed CoreWars” game. CoreWars is a game of conquest in cyberspace. Each player writes an assembly language program in the special-purpose assembly language RedCode. All players’ programs are loaded into the memory of a RedCode Virtual Machine (RVM) engine and attempt to wipe out other programs by overwriting the memory in which they live. The winner is the last program left running, or the program “owning” the most memory when the time limit for the game expires.

This project includes two variants of this game: a “uniprocessor” version (similar to the standard CoreWars game) in which all processes live in a single virtual machine (the RedCode Virtual Machine, or RVM) and a “distributed” version in which there are multiple virtual machines along with a virtual network for communicating among them. In this version of the game, processes can copy themselves across the network onto other RVMs, and the goal is to take over the entire network.

Because this game is quite complex, a critical component of this project is building a good graphical user interface (GUI) to visualize and control the game. The project will be assessed, in part, on the quality and intuitiveness of this UI.

2 Definitions

ASSEMBLER Program that converts a human-readable, text RedCode assembly language file into a binary PROGRAM IMAGE file. The ASSEMBLER MAY be part of the main DCoreWars program, or it MAY be a separate program. The ASSEMBLER MUST also be capable of disassembling PROGRAM IMAGES — that is, of turning a binary image of a PROGRAM into a human-readable, text RedCode assembly language file.

CLIENTS Prof. Lane and TA Barrick.

HALT Termination of a PROCESS. When a PROCESS HALTs it is removed from the list of RUNNABLE PROCESSES for its PROCESS GROUP. A HALTed PROCESS can no longer execute instructions, read or write to memory, write to the network, or otherwise influence the state of the game.

ILLEGAL OPERATION An exceptional condition that causes the PROCESS that generated it to HALT. Examples of ILLEGAL OPERATIONs include, but are not limited to, attempting to execute a bit pattern that does not correspond to a legal instruction; attempting to read, write, or execute a word from outside the bounds of memory for the RVM on which the PROCESS is executing; or attempting to write to the network when no network connection has been established.

MAY A requirement that the product can choose to implement if desired. Can also indicate a choice among acceptable alternatives (e.g., “The program MAY do x, y, or z.” indicates that the choice of behavior x, y, or z is up to the designer.)

MUST A requirement that the product must implement for full credit.

MUST NOT A behavior or assumption that must not be violated. Violating a MUST NOT restriction will result in a penalty on the assignment.

PLAYER A human player who wrote one of the PROGRAMs that are battling in the DCoreWars game. The DCoreWars game engine MUST have some way to track the names of players and associate them with the PROCESS GROUPs that their PROGRAMs run under.

PROCESS A single thread of execution in a RVM.

PROCESS GROUP All the PROCESSES belonging to a single PLAYER on a single RVM. Each PROCESS GROUP has a unique ID corresponding to that PLAYER (e.g., a unique number for that PLAYER or that PLAYER’s name). Different RVMs in a distributed game may have different PROCESS GROUPs running on them, but all PROCESS GROUPs belonging to the same PLAYER must have the same ID on every RVM.

PROGRAM The assembly code representing an executable program — i.e., the code for one of the warriors.

PROGRAM IMAGE The assembled, binary image of a PROGRAM, resulting from converting each of the text assembly instructions in the PROGRAM file into the corresponding bit pattern.

RECOVERABLE ERROR An error condition that the software can ignore, correct, or otherwise recover from. The program **MUST** produce a warning message and then cleanly continue with no corruption or loss of valid data.

RedCode Virtual Machine The virtual CPU architecture, including instruction set emulation, memory representation, **PROCESS** and **PROCESS GROUP** handling, etc. that interprets the RedCode language.

RUNNABLE A **PROCESS** is **RUNNABLE** if it has not been **HALTED** (i.e., has not performed an **ILLEGAL OPERATION** or the `halt` instruction). A **PROCESS GROUP** is **RUNNABLE** if it contains at least one **RUNNABLE PROCESS** and owns at least one cell of memory.

RVM See RedCode Virtual Machine.

SOCKET Special data structure maintained by each **PROCESS**, used to record a network connection between that **PROCESS** and another **RVM**. Each **PROCESS** can have only a single open **SOCKET** at a time.

UNRECOVERABLE ERROR An error condition from which recovery is impossible. The program **MUST** produce an error message describing the condition and then cleanly halt.

3 Rules

There are two levels of rules (and simulator complexity): the basic, uniprocessor CoreWars; and the “distributed”, multiprocessor, DCoreWars version. For full credit, the final project submission **MUST** support the full, DCoreWars version.

3.1 Uniprocessor CoreWars (UCoreWars)

In the uniprocessor version, there is only a single **RVM** and all programs compete for ownership of memory in its **RAM**. Each player writes a single RedCode assembly language **PROGRAM** and uses the **ASSEMBLER** to assemble it into a binary **PROGRAM IMAGE**. The **RVM** loads the **PROGRAM IMAGES** into its **RAM**, creates an initial **PROCESS** for each **PROGRAM**, and assigns each to a **PROCESS GROUP** for the corresponding player. The engine then waits for the user to start the game. When the game is started, the **RVM** executes **PROCESSES** from each **PROCESS GROUP** in round-robin fashion (as specified below) until the game ends.

Each memory cell in **RAM** is either unowned or owned by a single **PROCESS GROUP**. Whenever a **PROCESS** writes to a memory cell, that cell becomes owned by the corresponding **PROCESS GROUP**. If a **PROCESS** attempts to execute an instruction from a cell owned by a different **PROCESS GROUP**, it is considered to be an **ILLEGAL OPERATION** and the executing **PROCESS** is **HALTED**.

The game runs until either the user stops the game, the time expires, or only a single **PROCESS GROUP** contains any remaining running **PROCESSES**. When the game ends, the last **PROCESS GROUP** left alive (and the corresponding **PLAYER**) wins. If more than one **PROCESS GROUP** is alive when the game ends, the **PROCESS GROUP** owning the greatest number of memory cells wins.

RVM Initialization At the start of a game, before loading any PROGRAM IMAGES, every cell in the RVM's RAM is initialized to UNOWNED and containing the HLT instruction. One PROCESS GROUP is created for each PLAYER, but no PROCESSES are created for any PLAYER until a PROGRAM IMAGE is loaded for that PLAYER.

PROGRAM IMAGE Loading Each binary program is initially loaded into RAM beginning at a random location. The initial locations are picked such that:

1. A program MUST be loaded contiguously. That is, if a program is of length l words and is loaded starting at cell i , then after loading the program will occupy cells $i, i + 1, i + 2, \dots, i + l - 1$.
2. No program may be loaded overlapping another program.
3. No program may run off the end of memory. I.e., if the RAM of the current RVM is k words and a program is of length l , then the highest legal starting location for that program is $k - l$.

If a set of programs is too large to all fit into memory simultaneously, the RVM engine MUST treat this as a RECOVERABLE ERROR by reporting this condition to the user and allowing the user to specify a new set of programs to load.

Process Initialization When a PROCESS is created, its instruction pointer is initialized to the beginning of the program code (if the PROCESS was created at the start of the game) or with the location specified by the `fork` instruction (if the PROCESS was created by a fork from an existing process). All other registers for the new PROCESS are initialized to 0.

Round Robin Scheduling The RVM executes one instruction from each RUNNABLE PROCESS GROUP in turn, returning to the first PROCESS GROUP after the last has executed. Within each PROCESS GROUP, each PROCESS is scheduled in turn until all PROCESSES have received one cycle, and then execution returns to the first process.

Example: Suppose that there are currently two PROCESS GROUPS, G_1 and G_2 . G_1 contains three PROCESSES, P_{11} , P_{12} , and P_{13} , while G_2 contains two PROCESSES, P_{21} and P_{22} . Then the scheduling order will be $P_{11}, P_{21}, P_{12}, P_{22}, P_{13}, P_{21}, P_{11}, P_{22}, P_{12}, P_{21}, P_{13}, \dots$

Scoring Each PROCESS GROUP has a score equal to the number of memory cells that it owns. When the score of a PROCESS GROUP reaches 0, that PROCESS GROUP becomes non-RUNNABLE.

End of Game The game ends when either:

1. The user ends the game from the user interface.
2. The specified duration (total number of clock cycles) is exceeded.
3. Only a single RUNNABLE PROCESS GROUP remains.

Victory Conditions When the game ends, the winning PROCESS GROUP (i.e., PLAYER) is:

1. If just a single PROCESS GROUP remains RUNNABLE at the end of the game, that PROCESS GROUP (and its PLAYER) wins.

2. If more than one PROCESS GROUP remains RUNNABLE at the end of the game, the PROCESS GROUP that owns the most memory wins.

Instruction Set In the uniprocessor version, the RVM executes all instructions *except* the `open`, `rsw`, `rfrk`, and `close` instructions. Attempt to execute those instructions in a uniprocessor game is an ILLEGAL OPERATION by the executing PROCESS. A PROCESS MAY, however, execute a distributed game system call (see Appendix A, Table 2) in a uniprocessor game. In this case, the system call simply has no effect.

3.2 Distributed CoreWars

In the “distributed” version of the rules, there are multiple virtual machines, connected by a virtual network. Programs are able to copy themselves across the network to other RVMs and attempt to take over those machines as well. The object of the game is still the same, however — be the only PROCESS GROUP remaining on any RVM or, failing that, control the most RAM across all RVMs.

The mechanics of a single RVM are the same as in a uniprocessor game, except that the RVM allows four additional instructions:

open The `open` instruction allows a PROCESS to create a connection to another RVM. The precise semantics of this instruction are specified in Appendix A, Table 3, but the gist is that this instruction creates a connection from the process that executed it to a remote RVM. Once a PROCESS has successfully opened a network connection, it can write to the remote machine with the `rsw` instruction.

rsw Remote store word. This instruction allows a PROCESS to store a word to the memory of the remote machine that it contacted with `open`. Syntactically, it’s the same as the `sw` instruction, but rather than specifying an address on the local machine, it writes to an address on the remote machine.

rfrk Remote fork. This instruction allows a PROCESS to create a PROCESS on the remote machine that it contacted with `OPEN`. Syntactically, this instruction is the same as the `frk` instruction, but the new PROCESS is created on the remote RVM rather than the local one.

close Close the connection to the remote machine.

Additional details on each of these instructions is found in Appendix A. In addition, there are a number of system calls that are supported in the distributed game — see Appendix A, Table 2 for details.

The DCoreWars version also requires a few additional rules:

Program Image Loading In DCoreWars, there are three possible initialization modes (to be specified by the user before the game begins):

1. All programs loaded into a single RVM. In this initialization mode, all programs are loaded into the first RVM under the rules for uniprocessor loading, and all other RVMs are initialized to empty (i.e., no PROCESS GROUPS, all memory unowned, and all memory cells initialized to HLT).

2. Each program loaded into a different RVM. In this mode, each PROGRAM IMAGE is loaded onto a different, randomly selected RVM. All other RVMs are initialized to empty, as in the previous rule.
3. One copy of each program loaded onto each RVM. In this mode, every RVM is initialized with all of the PROGRAM IMAGES, according to the rules for uniprocessor loading.

PROCESS Initialization PROCESSES are initialized as in UCoreWars, but DCoreWars processes additionally have a SOCKET to support network communications. All SOCKETs are initialized to closed (i.e., no network connection when the PROCESS is created).

Round Robin Scheduling On each RVM, scheduling happens in the same way as on a single, uniprocessor RVM (see above). If only a single PROCESS GROUP exists on a single RVM, it clearly gets all of the cycles for that RVM.

The individual RVMs are *not*, however, synchronous. Each one can have its own CPU speed, different from all other RVMs and each RVM can (and most likely will) execute its instructions at different literal times.

In addition to the normal, uniprocessor execution cycle, each cycle in the distributed CoreWars game also includes a network data loading phase. See below, under **Network Communications** for details.

Network Initialization Initially, no connections are open between any machines.

Multithreading The asynchronous scheduling of the multiple RVMs in a DCoreWars game is supported by a multithreaded architecture. Each RVM is associated with a thread that handles its execution and decides when its cycle occurs, based on that RVM's CPU speed.

Network Communications When an RVM sends a word of memory to another RVM (via the `rsw` instruction), that word is queued in an incoming data buffer for the receiving RVM. At the beginning of each of its cycles, each RVM MUST check its incoming data buffer. If there is data queued and waiting, then the RVM removes the first word from the buffer and stores it at the requested memory location. The ownership of that memory location is changed to the PGID that sent the data. This operation takes place before the execution of the instruction for the current PROCESS. Only one word can be stored in this way per cycle.

The incoming data buffer for each RVM is a limited-length buffer (specified by the Data Buffer Size parameter for that RVM). When that capacity is reached (i.e., more data has been sent to the buffer by other RVMs than removed from it by this RVM), additional data messages are dropped.

4 RedCode and the RVM

4.1 Instruction Set and Assembly Language

There are a number of variants of the RedCode assembly language and instruction set available on the web. The one that will be used in this project is derived from the MIPS instruction set, a

register-to-register RISC language. Every instruction in this RedCode is specified by a single 32-bit word, broken up into fields indicating the opcode and its arguments. For details on the binary representation of instructions and their syntax and semantics, refer to the MIPS assembly language instruction guide (separate handout and web resources).

The RVM for the DCoreWars game does *not* need to implement the entire MIPS instruction set. In particular, it need not handle the floating point instructions or any of the other special coprocessor instructions. The RVM **MUST** support at least the instructions given in Table 1 in Appendix A and the system calls given in Table 2 in Appendix A.

In addition, the RVM **MUST** support a set “extended” instructions that are specific to this project and are not part of the standard MIPS instruction set. Each of these instructions displaces one of the existing MIPS instructions that will not be implemented in the RVM, employing the same bit representation as the standard MIPS instruction. The extended instructions are specified in Table 3 in Appendix A.

The assembly language is *not* required to support the more sophisticated features of common MIPS assemblers such as conditional assembly, pseudoinstructions, directives, pragmas, etc., but the design team may choose to implement some such features if they desire. All such features **MUST** be documented in the design and user documentation.

4.2 The RedCode Virtual Machine

RedCode executes on a special virtual machine, the RedCode VM (RVM), a multiprocessing, 32-bit, little-endian, register-based machine with a single, linear memory space and no virtual memory mechanism or memory protection whatsoever. The memory of a RVM is made up of 32-bit word cells. A read or write to a single cell is an atomic operation.

Each RVM may have the following unique characteristics:

CPU Speed Each RVM may run at a different speed from all of the others. The CPU speed of a RVM is given by a single, strictly positive integer, where 1 is the fastest speed. A RVM with a speed of 2 takes, on average, twice as long for each clock cycle as does one with a speed of 1; a speed of 3 takes three times as long per clock cycle, etc.

Memory Size Each RVM may have a different memory size from every other. The memory size is given by a single, strictly positive integer in the range $[2^0, 2^{16}]$ specifying the number of 32-bit words in the memory. An attempt to access (read, write, or execute) a cell outside the memory range causes an ILLEGAL OPERATION fault in the executing process.

Data Buffer Size Each RVM may have a different capacity for the size of its incoming data buffer. As data is transmitted to an RVM via the `rsW` instruction, it is queued on an incoming data buffer. When the capacity is reached, additional data messages are dropped.

Every PROCESS has 32 32-bit general purpose registers, denoted \$0-\$31, and three special purpose registers, PC, HI, and LO. PC is the program counter and contains the 32-bit address of the instruction currently being executed, while HI and LO are 32-bit registers used for the results of `mult` and `div` instructions. HI and LO cannot be manipulated directly, but their values can be copied to standard registers via `mFhi` and `mFlo`. The register \$0 always contains the value 0, regardless of what data has been written to it. The registers \$1 and \$2 are not reserved, but they do

have a special meaning to the system call instruction (see Appendix A). Unlike the standard MIPS assembly language, no other registers are reserved or have special names or meanings.

Finally, every process is associated with a special data structure, the `SOCKET`, not present in the standard MIPS architecture, that is used to track connections to remote RVMs. `SOCKET` can be modified *only* through the special extended instructions `open` and `close` and it is used only by the `rsw` and `rfrk` instructions. The implementation of `SOCKET` is not specified by this document — it is the design team’s choice how to provide this service. The important criterion is that every `PROCESS` can have its own `SOCKET`, but each `PROCESS` can have only a single `SOCKET` open to a remote RVM at a time.

5 Requirements

5.1 Assembly/Disassembly

The DCoreWars suite **MUST** provide a mechanism for assembling human-readable text assembly programs into binary `PROGRAM IMAGE` files. The game engine itself **MUST** be capable of loading the assembled `PROGRAM IMAGE` and executing it. The assembly capability **MAY** be provided as part of the main client program (via command-line switches or the UI) or it **MAY** be provided as a standalone assembler application.

The DCoreWars suite **MUST** also provide a mechanism for *disassembling* a binary `PROGRAM IMAGE` file and producing a corresponding human-readable text assembly language file.

The assembly language supported by the assembler **MUST** include the entire instruction set documented in Appendix A. It **MAY** also include standard MIPS pseudoinstructions and/or conditional assembly or other advanced functionality, at the design team’s choice.

5.2 User Interface

The DCoreWars program **MUST** provide at least one client program that provides a visualization of the game state at all time points. The program team **MAY** also provide additional support programs, at their discretion. For example, the assembly/disassembly functionality **MAY** be provided by the DCoreWars program itself, or it **MAY** be provided by an auxiliary support program.

The main DCoreWars program **MUST** provide at least the following user interface elements. It **MAY** also provide additional UI functionality. The specific ways in which the UI is implemented is up to the design group, but part of the evaluation of the program will be on the quality and intuitiveness of the user interface.

1. Ability to specify the number of players and a name for each.
2. Ability to specify uniprocessor or distributed versions of the game. This **MAY** be implemented via a GUI element, a command-line option, or a by providing separate client programs for UCoreWars and DCoreWars.
3. For the full, distributed DCoreWars game, the ability to specify the number of RVMs and the CPU speed and memory size for each.
4. Ability to load assembled RedCode files from an arbitrary file for any user.

5. Ability to single-step a single RVM, all RVMs, or to run all RVMs in free-running mode. In free-running mode, the run-speed of the game (i.e., the number of milliseconds per CPU cycle for the fastest RVM) should be a dynamically selectable, user-controlled parameter.
6. Display of the full memory state of any single RVM at any timestep. This display MAY be raw hex or MAY be disassembled RedCode contents or the UI MAY allow the user to toggle between them. This display MUST have some way to indicate which user owns each memory location and which memory locations are unowned.
7. Ability to switch the above display to any RVM.
8. Display of the current score for each player on the currently displayed RVM and, optionally, on all RVMs.
9. Display of the number of PROCESSES currently alive for each player on each RVM.
10. Display of current network activity between any pair of RVMs.
11. Ability to cleanly quit the program, stop a current game without quitting, pause/resume a game without stopping, and start a new game.

The program MAY support a configuration file format to load items (1)-(3), above, but MUST also support some way to enter that data through the user interface.

The ability to assemble and disassemble RedCode programs MUST be provided but it MAY be provided by a separate program with its own UI/command-line behavior.

The program MAY provide command line switches to modify its behavior as well. All such switches MUST be documented in the user documentation.

5.3 Multithreading

The DCoreWars program MUST provide a multithreaded implementation of the distributed game. That is, each RVM in a distributed game MUST employ its own thread to run its emulation during free-running mode. This implementation MUST be correctly synchronized, robust, and deadlock-free. The design of the multithreaded component MUST be described in design and final performance documentation. The implementation MUST NOT have more than one living thread per RVM at a time (e.g., it MUST NOT spawn more threads than it needs, it MUST NOT fail to clean up threads that are done, etc.)

6 Deliverables

This section describes the content to be delivered during the project. Unlike previous projects, the actual milestones in this project are to be specified by the design group. The first deliverable is the development schedule along with milestones. *The team will be held to the schedule they provide in Milestone 1.*

6.1 Milestone 1: Development Plan

The first part of this project is a planning phase. The development group **MUST** meet among themselves soon after the project is released and work out a full development plan including modular decomposition, testing plan, group member responsibilities, timeline (including, but not limited to, intermediate milestones), and intermediate deliverables. The design group is to schedule time with the **CLIENTS** on April 1 to deliver and discuss the development plan and schedule. The initial design documents are to be provided hardcopy, in duplicate, and signed off by the design group and **CLIENTS** during the meeting.

The timeline **MUST** include at least 3 intermediate milestones. Each milestone **MUST** specify a delivery date and time as well as the content due. Specific content for each milestone is up to the design group, but **MAY** include items such as design documentation, class hierarchies, interfaces, code, test suites (unit, regression, and/or integration), working prototypes, API documentation, user documentation, demonstrations, etc.

The design group will be held to and graded on their adherence to their development schedule. Modifications to the development schedule are permitted, *so long as the modification is requested and approved at least three days in advance of the milestone delivery date, a new date is proposed and approved, and a new set of deliverables are proposed for the revised milestone.*

In addition to intermediate milestones, each group is to schedule a weekly, half-hour progress meeting with the **CLIENTS**. (This meeting may be included with the milestone meeting during the weeks of milestone deliveries.) Other than milestones, there need be no specific deliverables for such progress meetings, but the group members **MUST** be able to demonstrate progress during the week and account for time spent.

Dates and times of meetings and intermediate milestone delivery **MUST** be coordinated with the **CLIENTS** (specifically, with Prof. Lane). Feel free to see Prof. Lane during office hours before April 1 to work out acceptable meeting times. Meeting times may have to be rescheduled dynamically as Prof. Lane's other commitments impinge, but he will make a strong effort to stick to the original schedule.

6.2 Rollout

The final delivery of the full project is to be scheduled in the Development Plan, but delivery **MUST** be between 9:00 AM on Mon May 10, and noon on Fri, May 14. **NOTE:** The drop-dead date for this project is noon, Fri, May 14. *NO Project deliveries will be accepted after this time.*

In addition to deliverable content (listed below), rollout will include a final delivery meeting with the **CLIENTS**. At this time, the group must demonstrate the full functionality of its project, provide a walk-through of usage, demonstrate example test code, etc.

Deliverable content for this project includes:

DCoreWars.jar A `jar` archive file containing all class files necessary to run the DCoreWars program and provide all specified functionality.

Java source files A subdirectory named `src/` **MUST** be provided that includes the entire Java source code tree for all `.class` files in the `DCoreWars.jar` file, as well as any other support code (e.g., programs used in testing the code that are not part of the DCoreWars program itself). *Note:* if these programs depend on external library code other than the Java JDK

or the `gnu.getopt` suite, the submission tarball **MUST** either include the library whole or provide easy and explicit instructions on how and where to access such libraries. This documentation **MUST** be provided in the `README.TXT` file. The designer is responsible for ensuring that all copyright and distribution conditions are adhered to.

README.TXT This file **MUST** describe how to compile, configure, and install the `DCoreWars` program, including how to construct the `.jar` file from the raw source code. It **MUST** also list any dependencies on additional software support libraries. Furthermore, if the software suite provides more than one executable (e.g., if the assembler is separate from the main game client, or there are auxiliary testing programs provided), the `README.TXT` file should indicate what programs are included in the suite and how to run them. Complete documentation for such auxiliary programs **MUST** be included in the user documentation.

Internal documentation The handin **MUST** also include the full, compiled JavaDoc documentation for all Java source files in the submission tarball. This documentation **MUST** include full descriptions of every public or protected method, field, sub-class, enclosed class, or constructor employed by the code. This documentation hierarchy **MUST** be included in a sub-directory named `documentation/` within the submission tarball package.

User documentation The handin submission **MUST** include complete user-level documentation for the `DCoreWars` program suite. This documentation **MUST** include descriptions of how to use the main game client program as well as any auxiliary programs (e.g., a separate assembler). It must describe the User Interface of the program, how to interpret output/buttons/dialogs/screens/etc., and at least one step-by-step example of running the program, playing a game, interpreting the results, etc. This document **MUST** be named `USERDOC.extension`, but it **MAY** be a plain text, HTML, PDF, or PostScript document (with the appropriate extension). It **MUST NOT** be a Microsoft Word or other nonportable format document.

Design documentation The final submission must also include one or more documents that describe both the initial design of the system (as proposed in Milestone 1), the final design and features of the system (as delivered in the Rollout), and a rationale justifying the modifications (if any). **Note:** Modifications to the original design are not bad. In fact, they are expected and encouraged. But the design documentation **MUST** describe *why* any such changes are made. E.g., refined understanding of the specification, design discussions with the CLIENTS, improved efficiency or modularity, improved use of Java language features, scaling back design to meet deadlines, untenable original design, improved support for requirements, etc. are typical reasons for modifying the design.

Test cases The submission tarball **MUST** include a subdirectory named `tests/` that includes all of the data or test cases used in unit, regression, or integration testing of the `DCoreWars` system.

CVS log file(s) For each Java source file, the submission tarball **MUST** include a corresponding `.log` file including the CVS log for that sourcecode.

Team evaluations Each team member must independently provide a written assessment of their own performance within the design team as well as the performance of the other team members. This is to be delivered by email directly to the CLIENTS and is not to be included in the group submission package. This information will be kept strictly confidential by the CLIENTS.

7 Grading

The grade for this project is a group grade — each individual will receive credit for the performance of their team’s entire project. It behooves the team members to work hard and support each other to provide the best possible project.

8 Timeline

Mar 29, 3:00 PM Project description handed out

Apr 1, 9:00AM-3:00PM Milestone 1 due

May 10-14 Rollout and delivery meeting (to be scheduled)

May 12, 3:00PM-5:00PM Final exam

May 14, noon Drop dead date for P3 rollout

May 14, noon-2:00 Class demos and DCoreWars playoff¹

All times are given in the Mountain time zone.

¹If all groups complete their projects and have runnable DCoreWars engines, we will have a class play-off and Prof. Lane will spring for lunch during the demos.

Appendix A: RedCode Instruction Set

8.1 MIPS Standard Instructions

The following MIPS standard instructions MUST be supported in the DCoreWars game engine (both uniprocessor and distributed versions). For details of the binary encoding of these instructions, refer to the attached “MIPS Assembly Language” handout and/or the online reference documentation.

Instruction	Short description
Branch/Jump instructions	
beq	Branch on equal
bne	Branch on not equal
bgtz	Branch on greater than zero
bgez	Branch on greater than equal zero
bltz	Branch on less than zero
blez	Branch on less than equal zero
j	Unconditionally jump
jr	Unconditionally jump to register
Arithmetic instructions	
add	Add signed quantities
addi	Add signed immediate
addu	Add unsigned quantities
addiu	Add unsigned immediate
sub	Subtract
subu	Subtract unsigned quantities
mult	Multiply
divu	Divide
and	Logical AND
andi	Logical AND, immediate
or	Logical OR
ori	Logical or, immediate
nor	Logical NOR
xor	Logical XOR
xori	Logical XOR, immediate
sll	Shift left logical
sllv	Shift left logical variable
srl	Shift right logical
srlv	Shift right logical variable
sra	Shift right arithmetic
srav	Shift right arithmetic variable
Cont'd	

Instruction	Short description
Conditional instructions	
<code>slt</code>	Set less than
<code>slti</code>	Set less than immediate
<code>sltu</code>	Set less than, unsigned quantities
<code>sltiu</code>	Set less than immediate, unsigned
Data movement instructions	
<code>lw</code>	Load word
<code>sw</code>	Store word
<code>mfhi</code>	Move from HI
<code>mflo</code>	Move from LO
Miscellaneous instructions	
<code>syscall</code>	System call (see below)

Table 1: Standard MIPS instructions that MUST be supported by the RVM of the DCoreWars game.

Technically, many of these instructions detect overflow. For the purposes of this project, however, design groups MAY ignore that and treat all operations as ignoring overflow.

8.2 System call

The system call instruction supports special, “OS-like” calls that provide useful services to assembly language programs. These calls work by putting a “system call ID” value into register \$1 and the argument to the call in register \$2 and then executing the `syscall` instruction. The result of the instruction (if any) is placed into register \$1. While it may take multiple instructions to fill the registers for the system call, actually executing the system call is an atomic operation.

The DCoreWars engine MUST support the following system calls:

Syscall ID (\$1)	Argument (\$2)	Description
0	none	HALT — equivalent to the <code>hlt</code> instruction
1	none	Random — return a random, unsigned integer in the range $[0 \dots 2^{32} - 1]$.
2	none	Memsize — return the memory size of the RVM on which the instruction is executed
3	none	Netsize — return the number of hosts on the current network
4	none	GetPID — return a unique identifier for the currently executing PROCESS
5	none	GetPGID — return a unique identifier for the currently executing PROCESS GROUP
6	none	GetNPG — return the number of RUNNABLE PROCESS GROUPS on the current RVM
7	none	GetNPlayers — return the number of players in the current game
Cont'd		

Syscall ID (\$1)	Argument (\$2)	Description
8	PGID	Score(PGID) — return the current score (number of memory cells owned by) the specified PROCESS GROUP
9	RPGID	RemoteProc(RPGID) — return 1 if the specified PROCESS GROUP is RUNNABLE on the remote host currently connected to the SOCKET of the executing PROCESS, or 0 otherwise. Invoking this system call when the current SOCKET is empty (not yet connected or connection terminated) returns 0.
10	RPGID	RemoteScore(RPGID) — return the score (number of memory cells owned by) the specified PROCESS GROUP on the remote host currently connected to the SOCKET of the executing PROCESS. Invoking this system call when the current SOCKET is empty (not yet connected or connection terminated) returns 0.
11	PGID	GetNProcs(PGID) — return the number of RUNNABLE PROCESSES in the specified PROCESS GROUP.

Table 2: System calls available in the DCoreWars game engine. All system calls are available on both uniprocessor and distributed versions of the game, but a distributed system call (e.g., RemoteProc(RPGID)) called in a uniprocessor game has no effect.

The implementors are free to provide additional system calls, at their option, but all such system calls MUST be fully documented. Invoking an undocumented system call is an ILLEGAL OPERATION for the invoking process.

8.3 DCoreWars Extended Instructions

The following special instructions are not part of the standard MIPS instruction set, but are required by the DCoreWars game. Each instruction is given with a “standard” MIPS instruction that it replaces (obviously, those instructions have been omitted from the DCoreWars instruction set). Each of the following instructions should use the exact same binary coding pattern as the corresponding MIPS instruction that it replaces.

Instruction	Replaces	Description
hlt	break	Halt — terminates the current PROCESS and removes it from the current PROCESS GROUP.
frk \$s	lb	Fork — create a new PROCESS within the same PROCESS GROUP as the executing PROCESS. The new PROCESS is initialized as specified by the rules (Section 3.1), and its PC is set from \$s.
Cont'd		

Instruction	Replaces	Description
open \$s	lbu	Open connection to the remote RVM whose address is given by register \$s. If the connection is successful, 1 is stored in \$s, otherwise 0 is stored in \$s. When a successful connection is made, the SOCKET of the executing PROCESS is initialized to a record of the remote RVM. Attempting to open a new connection when the process already has an open SOCKET is an ILLEGAL OPERATION for the executing process. Attempting to open a connection to a non-existent RVM is not illegal, but does cause the open operation to fail.
close	lh	Close the current remote connection (if any) and reset the SOCKET of the currently executing process to empty. Executing this instruction when there is no open SOCKET (e.g., before executing open or after an unsuccessful open) has no effect.
rsw \$s, \$t	lhu	Remote Store Word — write the contents of register \$s to the address \$t on the remote RVM specified by the executing PROCESS's SOCKET. If the operation succeeds, a 1 is stored in register \$s, otherwise a 0 is. Attempting to execute rsw when there is no open SOCKET available (e.g., before executing open or after an unsuccessful open) is an ILLEGAL OPERATION for the executing PROCESS. Executing this instruction actually causes the specified word to be buffered in an incoming data buffer for the remote machine. All such buffers are of finite, bounded length. If a buffer is full when this instruction is executed, the word is dropped (not inserted into the incoming data buffer) and this instruction fails (causing 0 to be stored in \$s).
rfrk \$s	lwl	Remote Fork — create a new PROCESS on the remote RVM specified by the executing PROCESS's SOCKET. If the remote RVM already has a PROCESS GROUP with the same ID as the executing process, the new, remote PROCESS is created in that PROCESS GROUP. If the remote RVM does not have such a PROCESS GROUP, a new PROCESS GROUP with the same ID as the currently executing PROCESS's GROUP is created on the remote RVM and the new, remote PROCESS is created in that group. Attempting to execute rfrk when there is no valid SOCKET available (e.g., before executing open or after an unsuccessful open) is an ILLEGAL OPERATION error for the executing PROCESS.
Cont'd		

Instruction	Replaces	Description
-------------	----------	-------------

Table 3: Extended instructions available only in DCoreWars. The instructions `hlt` and `frk` are available in both uniprocessor and distributed versions of the game; the instructions `open`, `close`, `rsw`, and `rfrk` are available only in the distributed version.